Principles of Operating Systems

David Ng

July 4, 2017

Contents

§1 May 16, 2017

§1.1 Course Contents

In this class, we will cover Processes, Threads, Concurrency, Scheduling, Deadlock, Memory, File systems, Input/output, and Multiple Processor Systems.

§1.2 Introduction to Operating Systems

An **Operating System (OS)** is essentially just another piece of software that sits between applications and hardware. Its purpose is to make it easier to write applications. The OS deals with many familiar issues:

- **Performance**: CPU, Memory, etc.
- **Resource Utilization**: Scheduling, Management, etc.
- **Security**: Protection, Operation Mode, etc.

Understanding OS is the key to system programming. Useful techniques regarding data structures, conflict resolution, concurrency, resource management, and communication will be discussed. We use different operating systems every day. For instance,

- **Mainframe OS** allow for batch, transaction processing, and time sharing services.
- **Server OS** allow for multi-users, and the sharing of hardware/software. For example, Solaris, FreeBSD, Linux, Mac OS Server and Windows Server.
- **Multiprocessor OS** support parallel computing. For instance, Windows, Linux, and Mac OS.
- **Personal Computer (PC) OS** support a single user on various tasks. For instance, Windows, Linux, and Mac OS.
- **Mobile OS** begin to resemble PC OS. For example, iPhone, Samsung Galaxy, etc.
- **Embedded OS** contain no untrusted software support. They are used in TVs, cars, cell phones, and MP3 players.
- **Sensor Node OS** are event driven, special-purpose. They may be used to detect forest fires, measure temperature, etc.
- **Real-Time OS** are deadline driven. They are used in assembly line and multimedia systems.
- **Smart Card OS** are very primitive. They use proprietary systems. An example would be those used on credit cards.

The OS provides common functions for controlling and allocating resources for application programs. The application programs use hardware, but hardware is notoriously difficult to use at low level. The OS thus provides controlled allocation for efficient and fair resource use. It hides the complexity of the underlying hardware and give the user/applications a better view of the computer.

While there is no precise definition of an Operating System, it is essentially a layer of software that provides application programs with a better, simpler, cleaner, model of the computer (hardware). It manages all of the resources, and is the software that runs all the time (in kernel mode). There are many ways to define an OS, including as an extended machine, and as a resource manager:

1. As an extended machine, abstraction/generalization is the key to managing complexity. The first step is defining and implementing the abstraction. For example, files such as pictures, emails, and webpage are easier to deal with than raw disk space. The second step is using the abstractions to solve problems, such as editing files. The OS masks the ugly hardware and provides a beautiful interface. Some notable operating systems include Windows, Mac OS, Linux Gnome and KDE. Many OS concepts are abstractions.

2. As a resource allocator, it can multiplex resources. Multiplex in time concerns multiple programs trying to use the same resource (spooling), and multiple programs trying to run at the same time (scheduling). Multiplex in space concerns the programs allocating memory. It also manages conflicts among multiple programs or users. It can be thought of as a control program that controls the execution of programs (such as interrupts), and prevents errors and improper use (such as traps).

§1.3 History of Operating Systems

The first OS was invented in the 1950s. Unix and Apple OS first appeared in the 1970s.

1. **First Generation (1945-1955) Vacuum Tubes**: Programs were hard-wired or on punch cards. Programs were written in machine language, and involved complicated wiring. There was no OS (since there was no need), and programs only included basic numerical calculations.

2. **Second Generation (1955-1965) Transistors and Batch Systems**: This included mainframe computers. FORTRAN programs were written on punch cards. Some operating systems that developed include FMS (Fortran Monitor System) and IBSYS (IBM's OS). An important concept introduced was batch systems. A **batch system** consists of a card reader, a tape drive, an input tape, a system tape, an output tape, and a printer. The IBM 1401 consisted of the card reader and tape drive that converted the instruction to magnetic tape. The IBM 7094 would then operate on the input tape and produce an output tape. Another IBM 1401 would then take this output tape and print the results. A typical FMS job consists of a job description, followed by the type of language, followed by the program, a load instruction, a run instruction, data for the program, and an indication that this is the end of the job.

3. **Third Generation (1965-1980) Integrated Circuits and Multiprogramming**: These systems use ICs (Integrated Circuits). OS developed in this time include IBM OS/360, CTSS (by MIT), MULTICS (today's client-server model), and UNIX (one-user version of MULTICS), and Linux. Some important concepts are:

    - **Multiprogramming**: A different job is executed in each memory partition. The CPU executes other jobs while waiting for the I/O of some jobs. As opposed to running a single program at a time where the CPU is idle while parts of a program are being run, multiprogramming allows the CPU to be used more efficiently by running parts of other programs while the CPU would otherwise be sitting idle.
    - **Spooling (Simultaneous Peripheral Operation On Line)**: Jobs are read from cards to disk, and jobs are loaded from the disk automatically, so

no more tapes are required. This is a way of dealing with slow devices and peripherals, as fast devices enter instructions to the spooler, which then feeds this information to a slower device. The computer can therefore send the information to the printer at its maximum speed, without needing to wait for the printer to finish.

- **Timesharing**: Interactive service to multiple users and work on big batch jobs in the background.

4. **Fourth Generation (1980-Present) Personal Computers**: Cheap mass-produced computers leads to friendly shells on top of the OS. This includes Windows, Mac OS, GNOME, and KDE.

5. **Fifth Generation (1990-Present) Mobile Computers**

§2 May 18, 2017

§2.1 Hardware

Common components of a desktop computer include the CPU, memory, video controller (monitor), USB controller (keyboard and mouse), and the HD controller (hard disk).

- **CPU** is the brain of the computer. **CPU registers** are on-board registers that are used for faster computation. Accessing information in registers is orders of magnitude faster than from memory. There are also some special purpose registers:
  - **Program Counter** contains the memory address of the next instruction to be fetched.
  - **Stack Pointer** points to the top of the current stack in memory.
  - **Status Register** indicates the interrupt flag, privilege mode, zero flag, carry flag, etc.

  A CPU cycle consists of fetching an instruction from memory, decoding it to determine its type and operands, and executing the instructions. These steps are repeated for the next instruction until the program finishes. A performance problem arises since fetching from memory takes longer than executing an instruction. The solution is to pipeline the operations. That is, while executing instruction n, the CPU could be decoding instruction $n+1$ and fetching instruction $n+2$. To implement this, we make use of independent fetch, decode, and execute units. The three stage pipeline goes from the Fetch unit to the Decode unit, and finally to the Execute unit. The advantage of this is that the CPU can now execute more than one instruction at a time, and essentially hide the memory access time. However, this comes at the cost of more complexity. A superscalar CPU has multiple FDE instructions, with the Fetch units sending to their respective Decode units, which all send to the Buffer before being sent from the Buffer to the respective Execute units.

- **Memory** should be ideally fast, large, and cheap. In practice, we can generally obtain two of the three conditions simultaneously. Main memory is **random access memory (RAM)**. Memory consists of an array of words, where each word has its own memory address. The operations that can be performed in memory are load, where a word is moved from memory to the CPU register, and store, where the contents of a register are moved to memory. Both load and store are slow operations compared to the speed of the CPU.

Typical access time (speed)	Component	Typical capacity (cost per byte)
1 nsec	Registers	< 1 KB
2 nsec	Cache	4 MB
10 nsec	Main memory	$1 - 8$ GB
10 msec	Magnetic Disk	$1 - 4$ TB

Most heavily used data from memory is kept in a high-speed cache located inside or very close to the CPU. When the CPU needs to obtain data from memory, it first checks the cache. This process is known as **CPU caching**. A cache hit occurs when the data needed by the CPU is in the cache, while a cache miss occurs when the CPU needs to fetch the data from main memory. The following are two types of CPU caches:

1. L1 cache (16KB) are always inside the CPU and usually feeds decoded instructions into the CPU execution engine.
2. L2 cache (xMB) are used for recently used memory words. It is slower than L1.

Example 2.1

A quad-core chip with a shared L2 cache would have 4 cores, each with their own L1 cache. They would all be connected to a shared L2 cache. A quad-core chip with separate L2 caches on the other hand, would have 4 cores, each with their own L1 and L2 caches.

Caching is a very useful concept in general. The goal of caching is to increase performance. This is usually done by copying information from slow storage to faster storage (cache). There are many uses for caching, including for disk cache, DNS, database, and memoization. The cache is fast but expensive, so it is usually much smaller than slow storage. Some general caching issues are:

- When to put a new item into the cache.
- Which cache line to put the new item in.
- Which item to remove from the cache when the cache is full.
- Where to put a newly evicted item in the larger memory.
- Multiple cache synchronization.
- How long the data in the cache is valid (expiration).

Different applications require different solutions to these problems.

Memoization is similar to caching, and is an optimization technique used to speed up programs, but at the cost of storing results of expensive computations.

- **Input and Output Devices** usually consist of two parts, the device controller and the device. The **device controller** is a chip or a set of chips that physically control the device. Controlling the device is complicated, and the CPU could be doing other things, so the controller presents a simpler interface to the OS. There are many different types of controllers, such as those for video, USB, and hard disks. The **device** connects to the computer through the controller. It follows some communication standard. The **device driver** is the software that talks to a controller, issues commands, and accepts responses. It is usually written by the controller manufacturer and follows some abstraction. It is needed so that the OS knows how to communicate with a controller (often in the form of modules).

- **Buses** are a communication system for transferring data between computer components. Modern computer systems have multiple buses, such as cache, memory, PCI, and ISA. Each type of bus has a different transfer rate and function. The OS must be aware of all of them for configuration and management. For example, it collects information about the I/O devices, and assigns interrupt levels and I/O addresses. Much of this is done during the boot process.

§2.2 Booting

When a computer is booted, the BIOS is started. **BIOS (Basic Input Output System)** is a program on the motherboard. It does the following:

1. Check RAM, keyboard, other devices by scanning the ISA and PCI buses.
2. Record the interrupt levels and I/O addresses of the devices, or configure new ones.
3. Determine the boot device by trying against the list of devices stored in CMOS memory.
4. Read the first sector from the boot device into the memory.
5. Read the secondary boot loader into the memory.
6. This loader reads in the OS from the active partition and starts it.
7. The OS queries the BIOS to get the configuration information and initialize all device drivers in the kernel.
8. The OS creates the device table and necessary background processes, then wait for I/O events.

The **kernel** is a basic unit of the operating system. It is the program that is running at all times on the computer. System programs are part of the OS, while application programs are not. The kernel is running at all times in the sense that it is listening and responding to events from hardware. The bootstrap program locates the kernel, loads it into memory, and starts it.

This kernel runs in a **kernel mode** (unrestricted mode) where all instructions are allowed, all input/output operations are allowed, and all memory can be accessed. Most modern CPUs support at least two privilege levels. This is usually controlled by modifying the status register. On CPUs without this support, there is only kernel mode. Kernel mode concerns the OS (Linux, Windows, and Mac OS), and hardware (CPU, memory, and I/O devices).

When the kernel runs a regular application, it runs it in a **user mode**. In user mode, only a subset of operations are allowed. Accessing the status register is not allowed (as this would allow one to run kernel mode). User mode concerns the compilers, assemblers, text editors, databases, and application programs such as web browsers and media players. These are the things that the users (human, computer, devices) interact with. Since applications run in user mode, this means that applications cannot talk to hardware. The applications must therefore ask the kernel for input and output. This cannot be performed by a simple function call. Instead, this is done by invoking a **trap**, otherwise known as making a **system call**.

§2.3 Traps

A trap is usually a special instruction that switches from user mode to kernel mode and invokes a predefined function such as SWI n, and INT n. It pauses the application and executes a kernel routine configured by the OS. When the kernel routine is finished, user mode is restored and the application resumes. This is very similar to an interrupt (often called a software interrupt). There are several ways for the kernel to perform I/O:

- **busy waiting / spinning / busy looping**: The CPU instructs the disk to read a file. The CPU then loops asking whether the disk is finished. If it is, we break out of the loop. The CPU then instructs the disk to return the result. The problem with this method is that the CPU is tied up while the slow I/O completes the operation. We are therefore wasting power.
- **busy wait with sleep**: The CPU instructs the disk to read a file. The CPU then loops a sleeping step, before asking whether the disk is finished. If it is, we break out of the loop. The CPU then instructs the disk to return the result. In this method, sleep could be detected by the OS, and the CPU could then run another program. The problem with this method is that it is hard to estimate the right amount of sleep, and the program might end up running longer than necessary.
- **interrupts**: The CPU instructs the disk to read a file. Then, it instructs the disk to wake it up when it is finished. The CPU then instructs the disk to return the result after a sleep step. When the I/O device finishes the operation, it generates an interrupt to let the application know it is done, or if there was an error. Sleep could be detected by OS, and the CPU could then run another program. The problem with this method is that the I/O device must support interrupts. Most devices support interrupts, and even those that do not can be connected through controllers that do.

§2.4 Interrupts

Interrupts generally follow this procedure:

- The kernel talks to the device driver, requesting an operation.
- The device driver tells the controller what to do by writing onto its device registers.
- The controller starts the device and monitors its progress.
- When the controller has finished the job, it signals the interrupt controller.
- The interrupt controller informs the CPU and puts the device information on the bus.
- The CPU suspends whatever it is doing, and handles the interrupt by executing the appropriate interrupt handler (in kernel mode).
- The CPU then resumes its original operations.

The CPU can process other programs while waiting for I/O, but the CPU could be interrupted for every single byte of I/O. Many devices/controllers have limited memory and interrupts take many CPU cycles to save/resume the original operation. A better solution would be to combine interrupts with **direct memory access (DMA)**, resulting in fewer interrupts and bulk data transmission between the controller and memory.

DMA is facilitated through a special piece of hardware on most modern systems. It is used for bulk data movement such as disk I/O. It is usually used with slow devices, so the CPU can do other useful things. It is also used with some fast devices that could overwhelm the CPU. The device controller transfers an entire block of data directly to main memory without CPU intervention. Only one interrupt is generated per-block, and is used to tell the device driver that the operation has completed.

§2.5 Summary of Traps and Interrupts

To summarize, interrupts are external events delivered to the CPU. They have origins in I/O, timer, and user input. They are asynchronous with the current activity of the CPU, and the time of the event is not known and unpredictable. On the other hand, traps are internal events caused by system calls or error conditions (such as division by 0). They are synchronous with the current activity of the CPU, and occurs during the execution of a machine instruction.

Traps and interrupts also share many similarities. For instance, both put the CPU in kernel mode, and both save the current state of the CPU. Traps and interrupts both invoke a kernel procedure defined by the OS, and resume the original operations when finished.

§2.6 OS Structure

In designing a kernel, the main problem is that code in the kernel runs faster, but big kernels have more bugs and can lead to system instability. There are different kernel-wide design approaches:

1. **Monolithic kernels**: The entire OS runs as a single program in kernel mode. Monolithic kernels are faster, but contain more bugs, are harder to port, and are potentially less stable. Examples include MS-DOS and Linux.

2. **Microkernels** : Only essential components run in kernel mode. The remaining components are implemented as system and user-level programs that run in user mode. Microkernels contain less bugs, are easier to port, easier to extend, and more stable. They may be slower in comparison however. Examples include Mach and QNX.

3. **Modular (Hybrid) kernels**: Consists of a small kernel with mostly essential components and dynamically loadable modules. Recent versions of Windows and MacOS fall under this category. Some parts of Linux are implemented as modules. It follows a **layered approach**, where components are organized into a hierarchy of layers, each constructed upon the one below it. This sounds good in theory, but it is hard to define layers and requires careful planning. It is less efficient since each layer adds overhead, and not all problems can be easily adapted into layers. Many modern OS are hybrid systems.

> **Example 2.2**
>
> In Mac OS X, the top layer consists of the application environment and a set of services providing GUI. The bottom layer is the kernel environment. This is comprised of the Mach microkernel, which is responsible for memory management, remote procedure calls (RPC), inter-process communication (IPC), and thread scheduling, and the BSD kernel, which is responsible for the BSD command line interface, networking, file systems, and POSIX APIs.

§2.7 Virtual Machines

The fundamental idea of **virtual machines** is to abstract the hardware of a single computer into several different execution environments. The host OS creates the illusion that each process (guest OS) has its own processor with its own memory and devices. Examples of virtual machines include VMWare, Java VM, UML (user mode linux), and Docker. The **hypervisor** is software that runs the virtual machines:

- **bare-metal** hypervisors run directly on hardware. These are usually on big servers, and are faster.
- **hosted** hypervisors run on top of another OS. These are usually on desktops, and are slower.
- **hybrid** hypervisors combine aspects of bare-metal hypervisors with hosted hypervisors. The Linux kernel can function as a hypervisor through the virtualization infrastructure of kernel based virtual machine (KVM).

There are many advantages associated with VM. Firstly, the host system is protected from the VM, so it can run unsafe programs in the VM. Additionally, normal system operation seldom needs to be disrupted from system development. Multiple different OS can be run on the same computer concurrently. This is the perfect vehicle for OS research and development. System consolidation also means that it could potentially save a lot of money. For instance, one big server could be used instead of many smaller ones.

§3 May 23, 2017

§3.1 System Calls

We recall that the operating system provides access to hardware through abstractions, and allows for resource management. It is accessible through system calls that are often implemented through traps. Modern OS use other implementations that do not rely on traps to perform system calls, but we will assume for this course that system calls are performed through traps.

When an application wants to access a service or resource of the system:

1. The application issues an appropriate system call, which is a routine provided by the OS.
2. Inside the system call, the OS switches from user mode to kernel mode (usually via a trap).
3. The OS saves the application state.
4. The OS does the requested operation.
5. The OS switches back to user mode and restores the application state
6. The application resumes.

A **system call** provides an interface to the services made available by the OS. System calls are accessed by the kernel. It can be thought of as an API provided by the OS. The interface for system calls varies between different operating systems, but the underlying concepts remain the same. The OS often executes thousands of system calls per second.

Example 3.1 (Copying a File)

First, the input file name is acquired from printing a prompt to the screen and accepting input. The output file name is acquired similarly. The system then attempts to open the input file. If the file does not exist, then the operation is aborted. The output file is then created. If the file already exists, then the operation aborts. A loop is then run to read from the input file and subsequently write to the output file. When read or write fails, the input and output files are closed. Finally, a completion message is sent to the screen, and the operation is terminated normally. We note that there are many system calls, even for simple programs.

System calls are minimalistic, and not very easy to use. Many system calls are implemented in assembly, and optimized for performance. The system call number is used to perform an operation, and parameters are usually passed in registers. This can be cumbersome to use from higher level languages. The OS often provides libraries for applications to use to access system calls. On Unix-like systems, this is usually **libc**, which is a C library. For C++, this is through **libstdc++** or **libc++**.

§3.2 Libraries

A **library** provides a set of functions that are available to an application programmer, including the parameters and the return values (APIs). An application can compile and run on any system that supports the same API. APIs hide the implementation details, such as the implementation of system calls. This makes system calls easier and more convenient. Some common API include the Win32 API for Windows, the POSIX API for POSIX-based system (Unix, Linux, and Mac OS X), and the Java API for the Java virtual machine. There is usually a strong correlation between a function in the API and its associated system call within the kernel, but the API is not the same as the system call.

Example 3.2 (printf)

The standard C library provides a portion of the system-call interface. For the `printf()` function, the C library intercepts the call and invokes the necessary system call `write`. The C library then takes the value returned by `write` and passes it back to the user program.

Higher level API often hide the implementation of system call. They are usually implemented through a **system call table**, which is a table of all system calls indexed by a unique number associated with each system call. Thus, when the user application uses a function, the library functions are then called, which then refers to the system call interface (trap, syscall, etc) that implements the system call. This last component of the system call interface acts as a **black box**, as there is no need to know how the system call works, and only a need to obey the API and understand the functionality of the calls.

Example 3.3 (read())

The function `read()` is implemented in common libraries. It takes as parameters the file descriptor, buffer, and number of bytes. In the user space, the user program calls read by first pushing the number of bytes, then the buffer, followed by the file descriptor. It then called read. The library read procedure then puts the code for read in the register, and traps to the kernel. In the kernel space (operating system), it dispatches, performs some operation, then goes to the sys call handler. This returns to the library procedure read in the user space, which then returns to the caller. Back in the user program calling the read, the SP is incremented.

§3.3 Tracing System Calls

To trace system calls, we use `strace` for Linux, `truss` for Solaris, and `dtruss` for Mac OS. On Windows, system calls can be found through the Windows Performance Analysis Tools. Different OS handle programs and commands differently. Thus, the same program or command may invoke different sets of system calls. For more information on these commands, we can refer to the `man` page. The `-c` option can be included to obtain a summary of all the system calls used.

§3.4 Processes

Processes are a key concept in all operating systems. They can be thought of as a program in execution. Processes are associated with an address space, a set of resources, a program counter, a stack pointer, and a unique identifier (process ID). Processes can be considered a container that holds all information needed by an OS to run a program.

A process tree provides a visualization of processes. Processes are allowed to create new processes. **Child processes** are created by a **parent process**. **Ancestor processes** are generated before subsequent processes. Process trees change very frequently, as processes are created and destroyed. File system trees on the other hand, are used in a very different way, as they may exist for a prolonged period of time.

File systems are also implemented through a tree structure, as they include subdirectories and files. On most Unix filesystems, other filesystems are mounted onto an existing filesystems at an arbitrary location (subdirectory). Before mounting, we have the present tree structure. After mounting, the OS mounts the guest filesystem in the original filesystem as a subdirectory. The location where this is mounted can be changed, but this is usually on `mnt`.

On Unix systems, two processes can communicate with each other via a **pipe**. Pipes are accessed using file I/O API. UNIX-like OS make use of files and associated API for different operations and services:

- Pipes allow for interprocess communication.
- Sockets allow for networking.
- Devices (`/dev`) allows for applications to obtain information quickly from devices. This includes block devices (disks) and character devices (terminals).
- Random number generator (`/dev/random`) acts as a source for random numbers.
- Exporting kernel parameters (`/proc` and `/sys`) can be achieved through pseudo-filesystems containing virtual files. For example, information about processes, memory usage, and information on hardware devices.

Example 3.4

For file management, there are system calls for opening and writing to a file, closing a file, reading data from a file into a buffer, writing data from a buffer into a file, moving the file pointer, and obtaining status information on a file. For directory and file system management, there are system calls to create directories, remove empty directories, create new entries, remove directory entries, mount a file system, and unmount a file system. Other signals allow one to change the working directory, change the protection bits of a file, send a signal to a process, and obtain the time. System calls may differ between operating systems, and require different keywords to access.

§3.5 Advantages of Processes

Early systems ran a single program at a time with full control. Over time, CPU speed increased and memory increased. The CPU was often left idle (such as during I/O). It became more advantageous to load and execute multiple programs at the same time. To do this, we needed firmer control and more compartmentalization of the various programs, along with a bit of extra hardware for memory protection. The result was the notion of a process, which is a program in execution. Processes allow for **multitasking**. Instead of idly waiting while performing input and output, multitasking became possible when memory was big enough to hold multiple programs. Processes also provide for the illusion of **parallelism**. Even with a single CPU, program i could run for a few thousandths of a second, then switch to program $i + 1$. This alternating program execution would then be repeated.

A program is a **passive entity**, as it is an executable file containing a list of instructions, and is stored on the disk. A process is an **active entity** with a program counter and other resources. A program becomes a process when it is loaded into memory for execution. A program can have more than one process.

§3.6 Implementation of Processes

Each process has its own **address space**. The OS makes part of physical memory available to a process. Its virtual address space goes from 0 to max, and is isolated from other processes. Going from 0 to max, we encounter the **text section** containing the program code, the **data section** containing the global variables and constant variables, the **heap** containing memory for dynamic allocation during runtime, followed by the **stack** containing temporary data such as parameters, the return address, and local variables. The program counter (PC) points to the current activity (next instruction), while the contents of CPU registers are accessible to processes.

Each process is represented in the OS by a **process control block (PCB)**. The process table is a collection of all PCB. A typical PCB includes:

- Process state.
- Program counter.
- CPU registers.
- CPU-scheduling information priority, pointers to the queue, and other parameters.
- Memory management information such as page tables, segment tables, etc.

- Accounting information such as CPU time, timeout values, process numbers, etc.
- I/O status information such as open files, I/O devices, etc.

Some fields of a PCB are those for process management, memory management, and file management:

Process Management	Memory Management	File Management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack Pointer		User ID
Process State		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

§3.7 Operations on Processes

Processes are created and deleted dynamically, so the OS must provide mechanisms for this.

1. **Process Creation** (`fork()`): The parent process is the creating process, while the child process is the newly created process. The processes in the system form a tree. Each process gets `pid`, which is a unique process identifier for each process.
2. **Process Execution** (`fork()`)
3. **Process Termination** (`exit()`): This allows the OS to delete the process. Termination can only be issued by the process or its parent.
4. Other operations: this includes operations such as synchronization and communication.

In Unix, the parent process and child process continue to be associated, forming a process hierarchy. In Windows, all processes are equal, since the parent process can give the control of its children to any other process.

Example 3.5 (init)

`init` is the first process started during the booting of a computer. Older UNIX systems used System V or BSD init systems. Many (most) Linux systems have now switched to `systemd`. `init` is the ancestor of all user processes, and has `pid = 1`. System processes (such as swapper and pagedaemon) are created during bootstrap, but are not descendants of init, since `kthreadd` is usually the parent of system processes. Orphaned processes are adopted by `init`. To print a process tree, we use the command `pstree`, `ps axjf`.

§4 May 25, 2017

§4.1 Multiprogramming

Multiprogramming can be thought of as each program having a separate program counter and executing concurrently. This is not the case (they are virtual program counters). In reality, we switch execution of each process so that it appears that they are being run at the same time. However, only one process is being executed at a time. CPU utilization is given by

$$1 - p^n,$$

where p is the fraction of the process' time spent on I/O and n is the degree of multiprogramming.

Example 4.1

Suppose that a computer has 8 GB of RAM. 2 GB are taken up by the OS, so there remains 6 GB for user programs. The user wants to run a process that requires 2 GB of RAM, with an average of 80% I/O. With 6 GB remaining, the user could potentially run three copies of the same program. Thus, CPU utilization is $1 - 0.8^3 \approx 49\%$. Consider that the user obtains 8 GB more of RAM. With 14 GB remaining, the user could run 7 copies of the same program. CPU utilization s therefore $1 - 0.8^7 \approx 79\%$. Throughput has increased by around 30%. With an additional 8 GB, CPU utilization becomes around 91%, where throughput has increased by only around 12% this time. We notice diminishing returns.

§4.2 Process Creation

In a low level view, the `init` process is created at boot time. Only existing processes can create new processes (`fork`). All processes are descendants of `init`. In Unix, `fork()` is followed by `exec*()` to spawn a different program. In Windows, this is accomplished by `CreateProcess()`.

In the higher level view, a process can create new processes for a variety of reasons:

- System initialization (boot). The init process will spawn many background processes called daemons (on Unix), and services (on Windows).
- The application decides to spawn additional processes. For example, to execute external programs or to do parallel work.
- When a user requests to create a new process. For example, launching an application from the desktop.
- Starting a batch job, such as in mainframes.

In a low level view, all of the higher level processes use fork to create new processes. Each process has its own address space, which is created during process creation. When fork is called, the address space is duplicated (almost identical). The next instruction is the same, but code flow may differ. By using fork, we create a child process from the parent process. There are several options for allocating resources for a new process:

- The child obtains resources directly from the OS. This is the most common scheme, as it is easy to implement.

- The child obtains a portion of the resources allocated to the parent. The parent gives the child a subset of its resources.
- The child shares some or all resources with the parent.
- Hybrid models that make use of combinations of the above schemes.

§4.3 Process Execution

We need to consider whether a process should be allowed to exhaust the resources of the entire OS. When the child process is created, the parent process usually does one of three things. The parent can wait until the child process is finished (often used when the child does `exec()`, or simply `system()`), the parent can continue to execute concurrently and independently of the child process, or the parent can continue to execute concurrently, but occasionally synchronizes with the child. This last option can be quite complicated.

§4.4 Process Termination

A process may terminate when it is finished, through exceptions or errors, when it is terminated by the parent, when it is killed by users, or when the computer is shut down. When terminating a process, memory is freed, child processes are assigned a new parent, and the PCB is deleted. All allocated resources are freed, and the processes are removed from the process table. Typical conditions that terminate a process are:

- **Normal exit**: This action is voluntary. For example, an application decides it is done, or the user decides to exit. The application calls `exit()` or `ExitProcess()`.
- **Error exit**: This action is voluntary. For example, an application detects an error and optionally notifies the user. The application calls `exit()` or `ExitProcess()`.
- **Fatal error**: This action is involuntary, and is usually due to a program bug. For example, accessing invalid memory or division by zero.
- **Killed by another process**: This action is involuntary. The parent or another process calls `kill()` or `TerminateProcess()`. For example, during shutdown or pressing `ctrl-c` in terminal.

The parent may terminate its children for different reasons. For example, the child has exceeded its usage of some of the resources, the task assigned to the child is no longer required, or the parent needs/wants to exit and wants to clean up. In Unix, to maintain the process hierarchy, a process may be terminated or assigned to the grandparent process or the init process as a child if its parent process is terminated. The default behavior on Linux is to re-parent the child process to the init process. This can be changed to kill the children or to re-parent to another process. See `prctl()`.

§4.5 Process Scheduling

The objective is maximize CPU utilization by having a process executing on CPU at all times. The **process scheduler** is a kernel routine/algorithm that selects an available process to execute on the CPU. Scheduling is accomplished by maintaining **scheduling queues**:

- **job queue**: This contains all processes in the system. This can be an array or process table.

- **ready queue**: This contains the processes in memory that are ready to be executed. This can be a linked list, implemented via a pointer in PCB.
- **device queues**: This contains processes waiting for a particular I/O device. Each device has its own queue.

Example 4.2 (Scheduling)
A process goes into the ready queue, enters the CPU, then ultimately leaves the CPU. The CPU may also send the process back to the ready queue through an I/O request (where the process enters an I/O queue before being handled by I/O), when a certain time splice has expired, when forking a child (so the child can execute), or when waiting for an interrupt (so the interrupt occurs).

We can also visualize process scheduling through states. There are three process states:

1. **Running** means that the process is actually running on the CPU.
2. **Blocked** means that the process is waiting for some event to occur, such as I/O.
3. **Ready** means that the process is ready to be executed by the CPU.

Only four transitions between states are allows. They are from ready to running (scheduler dispatch), running to ready (timeout or yield), running to blocked (blocking request), and blocked to ready (unblocking).

§4.6 Context Switching

Context switching is an essential feature of any multitasking OS, since it allows for switching of the CPU from one process to another. The context or state includes the CPU state (the contents of the CPU's registers and the program counter) and the pointer to the running process' PCB. Context switching occurs in kernel mode, and may be initiated when the current process voluntarily relinquishes CPU (known as **yielding**), due to a timer interrupt used to signal the OS that the current process has exceeded its allocated **time slice** (needed for the illusion of concurrency), or due to other hardware interrupts (from the keyboard, mouse, or network).

When the OS switches between processes, the OS saves the state of the old process in PCB and loads the saved state for the next process from PCB. Context switch introduces a time overhead, since the CPU spends cycles on no useful work. Context switching is often one of the most optimized parts of kernels. It can be improved by hardware support, such as by saving/restoring registers in a single instruction or having multiple sets of registers. Software based context switching is slower, but more customizable and more efficient.

§4.7 Threads and Processes

A **thread** is a process within a process. Multiple threads in a process share the resources of the process. Threads are used to improve performance. Processes are expensive to create, terminate, and switch, while threads are cheap. Threads are not strictly needed, but other alternatives are complicated. Processes are typically independent, while threads exist as subsets of a process. Threads belonging to the same process share many resources with each other, such as the address space and open files. On the other hand, processes

interact only through OS mechanisms such as **interprocess communication (IPC)**. Threads have more options available for communication. While processes have more capabilities, they are usually less efficient than threads.

We can think of a process as a way to group related resources together. It has an address space containing program text and data, as well as other resources, including open files, child processes, pending alarms, signal handlers, accounting information, etc. A process has a thread of execution, where a thread has a program counter, registers, and a stack. By default, a process has a single thread. Processes are used when the tasks are unrelated.

Threads in a process allow multiple executions to take place in the same process environment. A thread is a unit of execution of a process (mini processes within a process). Threads are scheduled independently and can make system calls simultaneously. While a thread requires an address space and other resources, it can share many of those resources with other threads. Threads are used when tasks are actually part of the same job and are actively and closely cooperating with each other.

Example 4.3

In a single-threaded process, a single thread has access to the code, data, and files. It is associated with registers and a stack. In a multithreaded process, each thread retains access to the code, data, and files. However, each are associated with their own registers and stack.

Threads allow for multiple activities within an application, and are useful for parallel computing on multiple CPU. For example, a browser downloading Ubuntu in one tab, while playing a movie in another. Threads essentially allow multitasking within an application. They are especially useful if the application needs to do substantial I/O while also performing CPU operations. It allows us to create interactive GUI applications through a dedicated UI thread. Threads are lighter weight than processes, easy to create and destroy, and consume less memory to copy and manipulate. Thread creation is usually tens to hundreds of times faster than process creation.

Processes
Group resources.
Each process has its own address space and PCB.
Address spaces are protected from each other.
Switching between processes is done at the kernel level.

Threads
Entities scheduled for execution on a CPU
Threads belonging to the same process share the process' address space, code data, and files.
Registers and stacks are not shared.
There are no address space protections.
Switching between threads can be done at either the user or kernel level.

Per process items include the address space, global variables, open files, child processes, pending alarms, signals and signal handlers, and accounting information. Per thread items include the program counter, registers, stack, and state. For instance, if one thread opens a file, that file is visible to the other threads in the process. They can all read and write it to it. If one thread changes a global variable, it will be changed in all other threads. If one thread calls `exit()`, all threads will be killed.

Example 4.4 (Word Processor)

You are editing a document with 1000 pages. On page 1, you delete a paragraph, then decide to jump to page 900. The application will be busy reformatting the entire document from the first page so that the content on page 900 can be displayed correctly. Different threads are used in word processors for interacting with the user, formatting, spell checking, and auto-saving.

Example 4.5 (Web Server)

Requests for pages come in and the requested page is sent back to the client. The tasks, such as receiving requests from the network interface, fetching the requested page from the disk, and sending the page to the network interface, are all I/O bound. We need to serve as many requests per second as possible. Different threads are used for receiving the requests, sending the pages, fetching the page from the disk, etc.

Some common thread scenarios are described below:

1. Manager/Worker: One manager thread assigns work to worker threads. The manager thread handles I/O. Worker threads can be static or dynamic. This is sometimes called a master/slave scenario.
2. Pipeline: A task is broken into a series of operations, where each operation handled by a different thread.
3. Peer: All threads work on the same or different tasks in parallel.

There are many benefits to threading. Some of these benefits are described below:

- **Responsiveness**: Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation.
- **Resource sharing**: Threads share the memory and the resources of a process to which they belong to by default. This implies that multiple threads reside in the same address space.
- **Economy**: It is more economical to create and to context switch between threads.
- **Scalability**: More tasks can be scheduled on a system.
- **Ease of Use**: Aternatives are often much more complicated. For instance, complicated finite state machines and non-blocking I/O and events.

Thread libraries provide the programmer with an API for creating and managing threads. These are higher level wrappers around low level system calls. Some examples include POSIX threads, Win32 threads, and Java threads. To use POSIX threads (pthreads), we use `#include <pthread.h>` and compile with `-lpthread`:

- `pthread_create(thread, attr, start_routine, arg)` starts a thread, and is similar to `fork()`.
- `pthread_exit(status)` terminates the current thread, and is similar to `exit()`.
- `pthread_join(thread, status)` blocks the calling thread until the specified thread terminates, and is similar to `wait()`.

- `pthread_attr_init(attr)`.
- `pthread_attr_destroy(attr)` initializes/destroys thread attributes.

§4.8 Thread Implementation

1. **User-Level Threads** are entirely implemented in user space, usually as a library. The kernel knows nothing about the threads. Thread implementation is entirely in user space, and requires no support from the OS. Thus, it can be used on OS that do not support threads. Each process has its own thread table and scheduler. Threads switch on system calls for I/O. There is no need to trap into the kernel when switching threads, so they are very efficient. They make use of simple/custom management and scheduling. It requires OS to support non-blocking I/O. However, for processes with many threads, each additional thread makes other threads run slower. There may also be paging issues.

   The process table resides in kernel space, while the thread table, run-time system, process, and thread reside in user space. User level threads require no OS support, have fast context switching, do not require traps, and use customized scheduling. However, it needs non-blocking system calls, a thread may run forever, page faults exist, it is inefficient for threads with many blocking procedure/system calls, and all threads get one time slice. User-level threads cannot easily be run on multiple cores.

2. **Kernel-Level Threads** are managed by the kernel/OS. There is one master thread table at the kernel level. Thread creation and deletion are done in the kernel space. This is better for processes doing lot of blocking I/O. Processes with multiple threads run faster, since each thread obtains the same amount of CPU time. However, it is less efficient, since thread operations need to trap into the kernel. There is increased kernel complexity.

   The process table and thread table reside in kernel space, while the process and thread reside in user space. Kernel level threads do not experience a problem with blocking calls, and has a global view of all threads and processes, leading to efficient global scheduling. However, they have issues with `fork()`, and require sending signals to threads.

3. **Hybrid** threads combine the advantages of user-level threads with kernel-level threads. The idea is to multiplex user-level threads into some or all of the kernel threads. The kernel is aware of only the kernel-level threads and schedules those, while the user-level threads are managed in the user space. It is up to the application to decide how many kernel-level and user-level threads to create. The result is more flexibility

   Multiple user threads exist on a kernel thread. The kernel threads link from the kernel space to the user space. This is often considered a many-to-many model. Each kernel-level thread has some set of user-level threads that take turns using it.

Scheduler activations is a threading mechanism to allow closer integration between user threads and the kernel. For instance, the kernel does processor allocation, while the thread library does scheduling. This allows for hybrid kernel-level and user-level threads. Scheduler activations is supported by some kernels. The kernel will notify the application of events. For example, when a thread has been blocked, it could deal with page faults. This notification is called an **upcall**. The application reacts by rescheduling its threads.

§4.9 Thread Models

- **Many-to-One (N:1)** or user-level threads map many user-level threads to one kernel thread. Thread management is done by the thread library in the user space. Examples include Solaris Green Threads, and GNU Portable Threads.
- **One-to-One (1:1)** or kernel-level threads map each user thread to a kernel thread. Examples include Windows NT/XP/2000, Linux, and Solaris 9.
- **Many-to-Many (M:N)** or hybrid user/kernel level threads multiplex many user-level threads to a smaller or equal number of kernel threads. For example, Marcel, the multithreading library for HPC.

§5 May 30, 2017

§5.1 Threading Issues

We should not call `fork()` in a program with multiple threads. We should only do this if `fork()` is followed by `execve)` (or just call `system()`). In reality, only the calling thread survives, and the other threads are not copied over. The application can be left in an invalid state. This creates a problem if synchronization mechanisms were used. It is possible to register a callback in case `fork()` is called by using `pthread_atfork()`.

Thread cancellation may occur when we wish to cancel threads:

1. **Asynchronous cancellation** can be issued with `pthread_kill(threadid, SIGUSR1)`. One thread immediately terminates the target thread. The problem with this approach is that the data currently updated by the thread is terminated. The killed thread has no chance to clean up, and may leave data in an undefined state.
2. **Deferred (synchronous) cancellation** asks the target thread to periodically check whether it should terminate. This is set via a boolean flag. It only checks at **cancellation points** at which it can be safely cancelled. The problem with this approach is that we hinder performance, and the thread may run for a while after a cancellation request is sent (and maybe report results). It is more flexible, but also more complex to implement.

Example 5.1

Suppose we are searching through a database with multiple threads searching different parts of the database. When one thread finds the result, we need a way to tell the other threads to stop searching.

§5.2 UNIX Signals

Signals are a form of inter-process communication. They provide limited IPC since only a small set of predefined integers can be used. Signals are asynchronous, similar to interrupts. A signal is used to notify a process/thread that a particular event has occurred. One process/thread sends a signal, while another process/thread receives it. Note that it is possible for a process/thread to signal itself. The signal lifetime starts when a signal is **generated/sent**, usually as a consequence of some event. The signal is then **delivered** to a process/thread. This delivered signal must then be **handled** by the process/thread via a **signal handler**.

Example 5.2

Pressing `<ctrl-c>` in a terminal will deliver the `SIGKILL` signal to the running process.

§5.3 Signal Generation

We can generate a signal manually from a program using `kill(pid, signal)` where `pid` can be the current process. This can also be achieved periodically using a timer `alarm()` or `setitimer()`. A signal can also be generated automatically to handle exceptions. An example is a segmentation fault that automatically generates `SIGSEGV`.

Example 5.3

From the command line, `kill 1234` tries to kill a specific process with the signal `SIGTERM`, which can be intercepted. `kill -9 1234` kills a specific process with the signal 9, which is `SIGKILL`. `kill -9 -1` kills all processes except `init`. Running this as root will kill all processes and shut down the computer.

§5.4 Signal Handling

Signal handling is performed by the **signal handler**, which is a function that will be invoked when a signal is delivered.

- **Default signal handler**: All programs have default handlers installed.
- **User-defined signal handler**: Programs can override the default handlers.

Some signals such as `SIGKILL` cannot be caught, as it will always terminate the process. Signal handling is more complicated with threads, since we need to consider which threads need to handle the signal, and what to do about user-level threads. In Linux, signal delivery depends on the type of the signal:

- They may be delivered to the thread that caused the signal, such as when there is invalid memory access (`SIGSEGV`).
- They may be delivered to every thread in the process, such as `<ctrl-c>` (`SIGKILL`).
- They may be delivered to certain threads in the process, such as with `pthread_kill(thread_id, signal)`.
- They may be assigned to a specific thread (usually the manager thread in the master-slave model) to receive all signals for the process.

There are many possible issues with signals. Signals can be delivered anytime, even when one is in the middle of a function, or in the middle of applying an operator. The state of the data might be in an inconsistent state. Additionally, the signal handler could itself be interrupted by another signal. To write a handler, one should keep it simple, only modify global variables (such as a flag to signal that an interrupt has occurred, and let the program handle the interrupt). One should thus declare global variables with a `volatile` keyword, and only call **reentrant** functions in the handler. In general, we should avoid signals as an IPC mechanism, especially in multi-threaded programs. Only use signals if necessary, such as in background processes.

Reentrant functions are functions that can be interrupted in the middle of an operation, and then called again (re-entered) and finally the original function call can finish executing. This is used in interrupt handlers, signal handlers, and multi-threaded applications. To write reentrant functions, we should avoid the use of global variables, unless using atomic operations, and not call other reentrant functions, unless we can temporarily disable interrupts.

Thread pools are a software design pattern. A program creates and maintains a set/pool of worker threads, where the pool size can be tuned to the available computing resources. When the program needs a thread, it takes one out of the pool, and when the thread is done, the program returns the thread back to the pool. If the pool contains no available thread, the program waits until one becomes free. This is known as **thread recycling**. With thread pools, thread creation/destruction costs are reduced, and the number of possible concurrent threads is limited. However, a problem occurs when a program needs more threads than the size of the pool.

Example 5.4

Whenever a server receives a request, it creates a separate thread to service the request. While creating a separate thread is superior to creating a separate process, a multithreaded server like this still has potential problems. We may be concerned with frequent thread creation and termination leading to performance problems. This may potentially create a large number of concurrent threads and lead to a resource problem. Thus, we use thread pools. Servicing a request with an existing thread is usually faster than waiting to create a thread. Additionally, a thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

Thread pools are usually combined with a **task queue**. Instead of asking for a thread, a task is inserted into a task queue. Available threads in the thread pool take tasks from the task queue, and finish them. The task queue can be augmented to support multiple priorities, but one should beware of possible dependencies.

§5.5 Race Conditions

A **race condition** is a behavior where the output is dependent on the sequence or timing of other uncontrollable events, such as those from context switching or scheduling on multiple CPU. It is often a result of multiple processes/threads operating on a shared state/resource. For instance, they may be modifying shared memory, reading/writing to files, modifying filesystems, or reading/writing to databases. Debugging race conditions is difficult, as many test runs may produce the same output that is often correct, but on occasion, the output will be different. We want to avoid race conditions.

To avoid race conditions, we need to find a way to prevent more than one process/thread from accessing the shared resource at any given time. That is, we need a process to be finished with a shared resource before another can access it. We need to identify critical sections in the code, and enforce mutual exclusion. A **critical section/critical region** is a part of the program that accesses the shared resource in a way that could lead to races. If we can arrange tasks such that no two processes or threads will ever be in their critical sections at the same time (by blocking access if another process attempts to enter its critical region), we could avoid the race condition through **mutual exclusion**. Some requirements to avoid race conditions are:

- No two processes may be simultaneously inside their critical regions.

- No assumptions may be made about speeds or the number of CPU.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.

Example 5.5 (Dining Philosophers Problem)

Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve, so that each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

Supposing that the philosophers are all synchronized. If they all grab the fork on the left at the same time, they reach a **deadlock**. This occurs since each will have one fork and need to wait for the others. Nobody gets to eat at all. A **livelock** occurs when the philosophers indefinitely switch between thinking and attempting to eat. Nobody will eat, so they still starve. When only one philosopher is eating, we obtain a non-optimal use of resources, resulting in **reduced parallelism**. This is called an **arbiter solution**. Alternatively, the philosophers can sleep for a random amount of time before attempting to eat. This is a **random timeout** mechanism for preventing deadlocks that is likely to work. However, there is still a small chance that no one eats, or some philosophers eat less often than others. This introduces the **fairness problem**. We can implement a **resource hierarchy** solution by establishing a partial order on resources. This works well, but starvation is still possible, and it is not practical for large resources.

§5.6 Mutex

Mutex is a synchronization mechanism for ensuring exclusive access to a resource in concurrent programs. It has the states locked and unlocked, with the two operations `lock()` and `unlock()`. Only the process/thread that locks the mutex can unlock it. Thus, a waiting queue is used to hold entities waiting on the mutex. This can be implemented in software via **busy waiting**, and is usually supported by the hardware and OS. Libraries often try to use hardware mutex, but can fall back to software. The following is a description of mutexes in pthreads:

- `pthread_mutex_init` creates a mutex.
- `pthread_mutex_destroy` destroys an existing mutex.
- `pthread_mutex_lock` acquires a lock or block.

- `pthread_mutex_trylock` acquires a lock or fails.
- `pthread_mutex_unlock` releases a lock.

§5.7 Summary

- **Critical Section**: The part of the program where a shared resource is accessed.
- **Mutual Exclusion**: Only one process can enter the critical section at any given time.
- **Mutex/Lock**: A mechanism to achieve mutual exclusion, two states, queue.
- **Deadlock**: A state where each process/thread is waiting on another process/group to release a lock. No progress is made.
- **Livelock**: States of the processes change, but none are progressing.
- **Starvation**: One process does not get to run at all.
- **Fairness**: A stronger requirement than starvation. All processes get equal opportunity to progress.

§6 June 1, 2017

§6.1 Producer-Consumer Problem

Two processes may share a fix-sized buffer that is used as a queue. The **producer** puts data into the buffer, while the **consumer** takes data out of the buffer. The consumer must wait if the buffer is empty, and the producer must wait if the buffer is full. A circular buffer is a common way to implement a queue.

Example 6.1

In a simple implementation of the circular buffer, we may encounter a race condition if we run the consumer and producer in different threads. Using a mutex, there is no race condition, but it may only work for a single producer and consumer. In the case that we wanted multiples of each, we encounter the problem of busy wait. Deadlocks may occur.

§6.2 Condition Variables

A **condition variable** is another synchronization primitive that is useful for implementing critical sections containing loops that wait for a condition. It can block one or more threads, while the other thread does something to satisfy the condition. It is used in conjunction with mutexes. This allows us to avoid deadlocks and busy waiting.

Condition variables are often used in a general pattern. For one thread, it locks the mutex and waits on a condition variable if it is unable to get what it needs. This unlocks the mutex, but puts the thread to sleep. Eventually, some other thread locks the mutex (optional) and then changes to a state that will satisfy the condition. This notifies the waiting thread, which then releases the mutex (optional). The waiting thread then wakes up, and gets the mutex back. We use the following commands:

- `pthread_mutex_t mutex` is the mutex, while `pthread_cond_t cond` is the condition variable.
- `pthread_cond_wait(&cond, &mutex)` automatically releases the mutex and causes the calling thread to block until some other thread calls `pthread_cond_signal(&cond)`. After returning, the condition must be rechecked (spurious wakeups). The mutex is then automatically re-acquired.
- `pthread_cond_signal(&cond)` wakes up one thread waiting on the condition. If no threads are waiting on the condition, the signal is lost. This must be followed by `pthread_mutex_unlock()` if the blocked thread uses the same mutex.
- `pthread_cond_init(&cond, &attr)` creates a condition variable.
- `pthread_cond_destroy(&cond)` destroys a condition variable.
- `pthread_cond_broadcast(&cond)` wakes up all threads waiting on the condition.

Example 6.2

In the previous producer and consumer example, we encountered a deadlock because one thread is stuck in an infinite loop while in its critical section. The other thread has no chance to run its critical section to allow the other thread to exit the loop. With condition variables, this can be resolved.

§6.3 Semaphore

A **semaphore** is another synchronization primitive. It is a special integer variable used for signaling among processes. The value indicates the number of available units of some resource. It supports three operations:

1. Initialization: It can be initialized with any value between 0 and max.
2. Increment: It can be incremented, and possibly unblock a process. One would use `up(S)`, `signal(S)` or `sem_post(S)`.
3. Decrement : It can be decremented, and possibly block a process. One would use `down(s)`, `wait(S)` or `sem_wait(S)`.

A **binary semaphore** is a special type of semaphore with the value being either 0 of 1. We note that the bodies must be executed atomically. When the binary semaphore is locked by a thread, it can be unlocked by any thread. As opposed to mutex, where a locking/unlocking must be done by the same thread, each semaphore maintains a queue of processes blocked on the semaphore.

Example 6.3

The critical section would fall between `wait(s)` and `signal(s)`, where `s` is the semaphore. Since the semaphore can only be incremented or decremented at the beginning or end, the critical section can evaluate properly.

Comparing semaphores with condition variables, semaphores use `up()` (this always increments the semaphore, and possibly wakes up the thread) instead of `cv_signal()`

(this is lost and has no effect if there is no thread waiting). Semaphores also use `down()` (this checks the value of the semaphore, and may or may not block) instead of `cv_wait()` (this always blocks, and does not check the condition).

Counting semaphores, also known as **general semaphores** or just **semaphores** represent an integer value s.

1. When $s > 0$, the value of s is the number of processes/threads that can issue a wait and immediately continue to execute.
2. When $s = 0$, all resources are busy, so the calling process/thread must wait.
3. When $s < 0$, s represents the number of processes that are waiting to be unblocked. Some implementations might do this.

We use the following commands to use semaphores:

- `int sem_init (sem_t *sem, int pshared, unsigned int value)` initializes a semaphore to the given value.
- `int sem_destroy (sem_t *sem)` destroys the semaphore, and fails if some threads are waiting on it.
- `int sem_wait (sem_t *sem)` suspends the calling thread until the semaphore is non-zero. It then atomically decreases the semaphore count.
- `int sem_post (sem_t *sem)` atomically increases the semaphore, never blocks, may unblock blocked threads, and is safe to use in signal handlers in Linux on 486+ hardware.
- `int sem_getvalue (sem_t *sem, int *sval)` returns the value of the semaphore via `sval`.
- `int sem_trywait (sem_t *sem)` is a non blocking version of `sem_wait()`.

Example 6.4

When we initialize the semaphore with a value of 2, two threads will enter their critical sections simultaneously. The other threads will be blocked until one of the two threads leave their critical sections.

Semaphore can be implemented in a data structure that holds a value and a list of PCB. A process can `block()` itself, and unblock by `wakeup()`. The list of PCB should be first in, first out.

Example 6.5

When managing a pool of $N = 10$ resources, semaphore s is used to keep track of the number of available resources. Initialization is called using `init(s,N)`. Each process may request $K \leq N$ resources at a time. This could potentially lead to a deadlock. For instance, when the first thread needs 7 resources and the second thread needs 6 resources. Depending on scheduling, we may get a deadlock. This could be related to the order of operations. Suppose that the first thread requests 6 resources. The scheduler then switches to the second thread, which requests 4 resources, exhausting all available resources. Both threads are stuck, resulting in a deadlock.

Suppose we have a pool of N resources, say fixed size buffers. We may want to use a counting semaphore initialized to N to keep track of the number of the buffers available. When a process wants to allocate a buffer, it calls P on the semaphore and gets a buffer. If there are no buffers available, a process waits until some other process releases a buffer and invokes V on the semaphore. Consider that there are two processes that respectively want to acquire $K < N$ and $L < N$ buffers, such that $K + L > N$. The naive implementation would have the first process call the simple decrementing variant P on the semaphore K times, and it would have the second process call the simple decrementing variant P on the semaphore L times. In this case, it is preferable to use an extended semaphore that can handle this situation.

Implementation of concurrent programs can be tricky, as one subtle error could result in everything coming to a grinding halt. Concurrent programming can be even worse than programming in assembly language. Any error with semaphores will potentially result in race conditions, deadlocks, and other forms of unpredictable and irreproducible behaviour. We could potentially violate mutual exclusivity if we swap the `signal` and `wait` that surround the critical section when using a semaphore, or cause a deadlock of we encapsulate the critical section with `lock` when using a mutex.

§6.4 Monitors

A **monitor** is a programming language construct that controls access to shared data. It is synchronization code added by the compiler and enforced at runtime. It can be implemented in Concurrent Pascal, C#, D, Modula-3, Java, Ruby, Python, etc. It can be somewhat emulated in C++ with classes, and even to some extent in C using struct and functions or function pointers. A monitor is a **module** that encapsulates shared data structures, procedures that operate on the shared data structures, and synchronization between concurrent procedure invocations. The data in monitors can only be accessed via the published procedures. A properly implemented monitor is virtually impossible to use in a wrong way, as a monitor is a higher-level construct compared to mutexes and semaphores.

Monitors ensure mutual exclusion, as only one thread can execute any monitor procedure at any time. All other threads would be blocked. Monitors have their own unique queues.

Example 6.6

Suppose a monitor construct contains the `incr()` and `decr()` functions. Calling `incr()` or `decr()` from multiple threads would allow one thread in, as the rest would be blocked. We can think of all bodies of all procedures being critical sections, protected by one mutex. In C++, one can emulate this by making a private mutex and locking it at the beginning of every method.

Monitors can have their own condition variables that are declared as a part of the module. They are only accessible from within the module, and are quite similar to `pthread_cond_t`. Once a monitor is correctly programmed, access to the protected resource is correct, for accessing from all processes. With semaphores or mutexes, resource access is correct only if all of the processes that access the resource are programmed correctly. When programming with monitors, one simply needs to test and debug the monitor, while programming with mutexes/semaphores, one needs to debug the code using them.

§6.5 Spinlocks

Spinlocks are another synchronization mechanism, with locks implemented using busy waiting loops. They are essentially a lightweight alternative to mutex. They are often implemented in assembly, using atomic operations, thus making them very efficient if one knows the wait time will be very short. No re-scheduling required. Spinlocks are used inside kernels, often in the same way as mutexes.

Definition 6.7. A **atomic operation** is an operation that executes instantaneously. That is, it cannot be interrupted by anything else.

Example 6.8

The structure of a spinlock is very similar to that of a mutex. The difference is that a spinlock has a very short critical section. Compare-and-swap (CAS) is an atomic operation used for synchronization, and is supported by most CPUs today. The general algorithm consists of first comparing the contents of memory to `val1`. If they are the same, it changes the memory to `val2`. These two actions are atomic. It then returns the old contents of memory.

`gcc` provides a number of atomic operations, including CAS. In the following, `type` can be an 8, 16, 32 or 64 bit integer or pointer:

1. `type __sync_val_compare_and_swap (type *ptr, type oldval type newval)` is an atomic compare and swap. If the current value of `*ptr` is `oldval`, then write `newval` into `*ptr`. Return the original `*ptr`.
2. `bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval)` is the same as above, but returns `true` if `newval` was written.

Spinlocks can be implemented using compare-and-swap, or through pthreads:

- `int pthread_spin_init(pthread_spinlock_t *lock, int pshared)`
- `int pthread_spin_destroy(pthread_spinlock_t *lock)`
- `int pthread_spin_lock(pthread_spinlock_t *lock)`
- `int pthread_spin_trylock(pthread_spinlock_t *lock)`
- `int pthread_spin_unlock(pthread_spinlock_t * lock)`

§6.6 Additional Synchronization Mechanisms

Event flags are a memory word. Different event may be associated with each bit in a flag. Some operations are set/clear flag, wait for 1 flag, wait for any flag, and wait for all flags. **Message passing** occurs when processes send each other **messages**. Messages can contain arbitrary data. Delivered messages can be queued in **mailboxes**. Processes can check contents of mailboxes, take messages out, or wait for messages. A common implementation is MPI (message passing interface), which is used in high performance computing.

Suppose we have three processes, L, M, and H whose priorities from highest to lowest are in alphabetic order. Assume that process H requires resource R, which is currently being accessed by process L. While H is waiting for L to finish using resource R, M

becomes runnable, thereby following process L. Now, H has to wait for both M and L to finish. Effectively, H has the lowest priority in this execution. This problem is known as **priority inversion**. To avoid this, we use **priority inheritance**. As soon as H requests resource R, the process P holding resource R automatically inherits the priority of H if P has lower priority. Once P releases the resource, its original priority is restored. As a result, we would have the execution order of L followed by H, then M.

Example 6.9

Race conditions can occur with separate programs. A print server monitors a directory for jobs to print. When it finds a PDF file, it prints it. If it does not find a PDF file, it sleeps for one second. Other programs can submit jobs by writing files to the directory. The problem is that the print server may print an incomplete file. Some filesystems support file/directory locking mechanisms, but we will not rely on those. We will assume the only atomic operations are file creation and file rename.

We modify the programs that print, so the print server needs no modification. When a program needs to print, it creates a temporary file in the directory, called randomly. For example, `job-xxxxxx.tmp`, where `xxxxxx` is a unique number. It then writes the output to this temporary file, then closes the file. Now, it renames `job-xxxxxx.tmp` to `job-yyyyyy.pdf` where `yyyyyy` is a unique number. We can then create a random file name (there is a UNIX utility for this, called `mkstemp()`). We can make our own program to rename to a random filename by looping the generation of a random string, following by attempting to call rename and breaking if successful.

Example 6.10 (Readers/Writers Problem)

A single resource is shared among several threads. Some threads only read the resource, while others only write to it. The resource supports multiple concurrent readers, but only a single writer. We can use semaphores to implement this efficiently, using three variables. `int readCount` represents number of readers reading, `semaphore cs` is used as a mutex to control access to the critical section, and `semaphore w_only` is used as a flag representing whether write is available.

§7 June 6, 2017

§7.1 Alternative Synchronization Mechanisms

We recall that a race-free solution satisfies the following four requirements:

1. **Mutual Exclusion**: No two processes/threads may be simultaneously inside their critical sections (CS).
2. **Progress**: No process/threads running outside its CS may block other processes/threads.
3. **Bounded Waiting**: No process/thread should have to wait forever to enter its CS.
4. **Speed**: No assumptions may be made about the speed or the number of CPU.

We have already seen that mutex, semaphore, spinlock, monitor, and condition variables permit mutual exclusion. We now consider several alternatives:

- **Disabling interrupts**: Each process disables all interrupts just before entering its CS and re-enables them just before leaving the CS. Once a process has disabled interrupts, it can examine and update the shared memory without interventions from other processes. Problems occur when a process never re-enables the interrupts. On multi-CPU systems, disabling interrupts affects only one CPU. Disabling interrupts is mostly used inside kernels, but even that is becoming problematic.
- **Lock variables**: A single shared (lock) variable is initialized to 0. When `lock == 0`, there is no process in its CS. When `lock == 1`, a process is in its CS. A process can only enter its CS if `lock == 0`. Otherwise, it must wait. However, there is no mutual exclusion since a first thread can pass the while loop, context switch to the second thread that also passes the while loop (checking that `lock == 1`, and then both set the lock. Both threads are in their CS. To overcome this, we need to make entering the CS an atomic operation (such as using compare and swap).
- **Strict alternation**: Two processes alternate entering their critical sections. This can be achieved using a global variable `turn = 0`. Thread 1 only enters when `turn != 0` and sets `turn = 1` when it leaves its critical section, while thread 2 does the reverse. This achieves mutual exclusion. However, we have busy waiting due to the while loop, this only works for two processes, and this does not satisfy the progress property, since a process is blocked by another (slower) process not in its CS.
- **Peterson's algorithm**: This is a software only solution for two processes. It may fail on some CPU that use out-of-order execution or memory reordering. It makes use of a shared integer `turn` that indicates whose turn it is, and a shared array `flag[2]` that indicates who is interested in entering the CS. This is initialized to 0. The while loop therefore checks that both the flag and turn are set appropriately before the while loop is exited. Peterson's algorithm achieves mutual exclusion, progress, and bounded waiting.
- **Synchronization hardware**: Race conditions are prevented by ensuring that critical sections are protected by locks. A process must acquire a lock before entering its CS, and releases the lock when it exits the CS. Many modern computer systems provide special hardware instructions that implement useful atomic operations that can be used to create atomic locking and unlocking mechanisms:
  - **compare-and-swap** is an atomic operation used for synchronization. It is supported by most CPU today, such as CMPXCHG on Intel. It compares the contents of memory to `val1`. If they are the same, it change the memory to `val2`, then returns the old contents of memory.
  - **test-and-set** is a specialized version of compare-and-swap. Old hardware used test-and-set, while newer hardware uses the more generalized compare-and-swap. First, it remember the contents of memory. Then, it sets memory to true and returns the old contents of memory.
  - **swap** is another atomic operation that can be used for synchronization. It atomically swaps the contents of two memory locations. Thus, we combine it with a boolean value initially set to true. While the condition is true, we repeatedly swap with a lock variable that is set to exit the loop only when a critical section is left.

When used correctly, the atomic operations such as compare-and-swap, test-and-set, and swap can be used to achieve mutual exclusion, progress and speed. However, they are too low level to achieve bounded waiting, especially for $N > 2$ processes. Bounded waiting can be added via two shared variables.

Synchronization hardware can be used to implement mutual exclusion, progress, speed and even bounded waiting. It avoids system calls, and can be more efficient if the expected wait time is short. However, we encounter busy-waiting (spinlocks) and extra coding (especially for bounded waiting). It also only makes sense on multicore systems.

§7.2 CPU Scheduling

Recall that in multiprogramming, our main objective is to maximize CPU utilization by having a process running at all times. Several processes are kept in memory at one time, and a process runs until it must wait. Instead of having the CPU sit idle, the OS takes the CPU away from the waiting process and gives it to another process that is ready to run. The software that decides which process runs next is called a **scheduler**, and is usually a part of the kernel.

Most processes alternate bursts of CPU activity with bursts of I/O activity:

1. **Compute-bound**, or **CPU-bound** processes contain long CPU bursts and infrequent I/O waits.

2. **I/O-bound** processes contain short CPU burst and frequent I/O waits.

As CPU get faster, processes tend to get more I/O-bound. It takes quite a few I/O-bound processes to keep the CPU fully occupied. Scheduling may be needed for process creation (parent and child), process termination, blocking system calls (such as I/O or mutex), I/O interrupts that may unblock a process (deciding whether the process is run immediately or put into the ready queue), and periodic clock interrupts (used to implement a time-slice).

Scheduling algorithms may be **non-preemptive**, where a process runs until it does I/O, exits, or voluntarily yields CPU. Context switching occurs voluntarily. Multitasking is possible, but only through cooperation. Scheduling algorithms may also be **preemptive**, where processes can be context switched without cooperation. This may be due to an interrupt, but not necessarily a clock. for instance, a new job is added, so an existing process is unblocked. Strictly speaking, there is no concept of time-slice. However, preemptive is often misused to mean preemptive time-sharing. **Preemptive time-sharing** is a special case of preemptive. Processes are context switched periodically, usually to enforce a time-slice policy. It is implemented through clock interrupts. On systems without a clock for clock interrupts, only cooperative multitasking (non-preemptive) scheduling is possible.

Scheduling algorithms generally fall under the following three categories:

1. **Batch**: Usually on mainframes, processing payroll, bank interests, and insurance claims. HPC systems also use batch algorithms, since no interactivity is needed, so no preemption is needed.

2. **Interactive**: General systems running many tasks, many of them must remain interactive.

3. **Real Time**: Applications are guaranteed CPU cycles per second. This is often tied closely to some hardware, such as for robots, planes, cars, video/audio capture, etc.

§7.3 Scheduling Metrics

We make use of the following statistics to compare the efficiency and usefulness of different scheduling algorithms:

- **Arrival time** is the time a process arrives (such as when you double-click the Firefox icon).
- **Start time** is the time the process first gets to run on the CPU. It is different from arrival for batch systems, and nearly identical to arrival on interactive systems.
- **Finish time** is when the process is done (time of the last instruction).
- **Response time** is how long before we get first feedback, and is often equal to start time minus arrival time.
- **Turnaround time** is the time from arrival to finish, and is calculated as the finish time minus the arrival time.
- **CPU time** is how much time the process spent on the CPU.
- **Waiting time** is the total time spent in the waiting queue, and is often calculated as the turnaround time minus the CPU time and minus the I/O time.
- **Average turnaround time** is the average turnaround time for multiple processes.
- **Average wait time** is the average wait time for multiple processes.
- **Throughput** is the number of jobs finished per unit of time

§7.4 Scheduling on Batch Systems

Below are the main scheduling algorithms for batch systems. We can use Gantt charts to visualize the scheduling of multiple processes.

1. **First Come First Serve Scheduling (FCFS)**: This is one of the simplest scheduling algorithms. It is non-preemptive, and the CPU is assigned in the order the processes request it, using a FIFO ready queue. New jobs are appended to the ready queue. A running job keeps the CPU until it is either finished, or it blocks. When a running process blocks, the next process from the ready queue starts to execute. When a process is unblocked, it is appended at the end of the ready queue. In this algorithm, we use the minimum number of context switches, since there are only N switches for N processes.

   A big disadvantage is the **convoy effect**. Consider the scenario with one CPU-bound process and many I/O-bound processes. The CPU-bound process will tie up the CPU, making the I/O-bound processes wait a long time.

   Example 7.1

   CPU-bound process A has 1 second CPU burst cycles, while I/O bound processes B_i need 1000 I/O operations, each 1/100 seconds long. With FCFS, each process B_i will spend 1000 seconds executing in the presence of A. If A did not exist, or if A could be preempted, each process B_i would finish in around 10 seconds.

2. **Shortest Job First Scheduling (SJF)**: This is another non-preemptive scheduling algorithm. It is applicable to batch systems, where job length (execution time) is known in advance. When the CPU is available, it is assigned to the shortest job. Ties are resolved using FCFS. SJF is similar to FCFS, but the ready queue is sorted. Sorting may be basic, where execution time can be static, based on a submitted estimate. Alternatively, sorting may be advanced, as the execution time can be dynamically computed based on the history of CPU bursts.

   SJF uses the minimum number of context switches (just like FCFS), and provides optimal turnaround time if all jobs arrive simultaneously (by minimizing average waiting time). However, this algorithm requires advance knowledge of how long a job will execute, and it thus often limited to batch job systems. It also has a potential for job starvation, as long programs will never get to run if short programs are continuously added. This can be solved by **aging** by increasing a job priority based on how long it has waited. Priority could be calculated as the total wait time divided by the estimated run time. We can then sort the ready queue based on priority.

3. **Shortest Remaining Time Next Scheduling (SRTN)**: This is a preemptive version of SJF. The next job is picked based on the remaining time, where remaining time is equal to the total time minus the time already spent on the CPU. SRTN is similar to RR. A ready queue is sorted by remaining time, where remaining time can be static, or dynamically calculated. Context switches can happen as a result of adding a new job.

   SRTN has similar advantages to SJF, including optimal turnaround time even if jobs do not arrive at the same time. However, it has slightly more context switches. Like SJF, it requires advance knowledge of how long a job will execute, and has a potential for job starvation. The cost of context switching must also be considered.

§8 June 8, 2017

§8.1 Operation Environments

We recall the following scheduling operation environments:

1. **Batch Systems**: No impatient users. Both nonpreemptive and preemptive algorithms with long time periods for each process are often acceptable (for example, corporate mainframe computing, payroll, inventory, accounting, and banking). Scheduling algorithms for batch systems should consider **throughput** to maximize jobs per hour (or per minute), **turnaround time** to minimize the time between submission and termination, **CPU utilization** to keep the CPU busy all the times, and **waiting time** to minimize turnaround time minus execution time.

2. **Interactive Systems**: Interact with users. Preemption is essential to keep one process from hogging the CPU and denying service to the others (for example, chat programs and servers). Scheduling algorithms for interactive systems should consider **response time** to minimize the time between submission and the responses, and **proportionality** to meet users? expectations.

3. **Real Time Systems**: Preemption is sometimes not needed because the processes know that they may not run for a long period of time. They usually do their work and block quickly (for example, gaming, video conferencing, and VoIP). Scheduling

algorithms for real time systems should consider **meeting deadlines** to avoid losing data, and **predictability** to avoid quality degradation in multimedia systems.

In all of these systems, a scheduling algorithm should additionally consider **fairness** by giving each process a fair share of the CPU, **policy/priority enforcement** by ensuring that the stated policy or priority is carried out, and **balance** by keeping all parts of the system busy.

§8.2 Scheduling on Interactive Systems

Below are the main scheduling algorithms for interactive systems.

- **Round Robin Scheduling (RR)**: This is a preemptive version of the FCFS algorithm. Each process is assigned a time interval, called a **time slice** or **quantum**, during which it is allowed to run (for instance, 10 msec). If the process exceeds the quantum, the process is preempted (context switched), and the CPU is given to the next process in the ready queue. The preempted process goes to the back of the ready queue.

  The performance of RR depends significantly on the size of the time quantum (Q) and the time required for a context switch (S). A very small Q implies heavy overhead, but a highly responsive system, while a very large Q implies minimum overhead, but a non-responsive system. Although Q should be large compared to S, it should not be too large. A rule of thumb is that 80% of the CPU bursts should be shorter than the time quantum. A quantum of around 20 to 50 msec is often a reasonable compromise.

  Example 8.1
  If $S = 1ms$ and $Q = 4ms$, then the CPU will spend $1/(4+1) = 20\%$ of its time on useless tasks.

- **Shortest Process Next Scheduling (SPN)**: This is very similar to SJF scheduling. The ready queue is sorted by a predicted next CPU burst. It uses time-sharing preemption. Prediction can be done using exponential averaging.

  Example 8.2 (Estimating Burst via Exponential Averaging)
  After every burst B, update the prediction P:

  $$P = a \cdot B + (1 - a) \cdot P',$$

  where P is the new prediction that is initialized to some value (even 0 would work), P' is the previous prediction, and a is a smoothing factor, commonly set to a $1/2$.

- **Fair Share Scheduling (FS)**: This is a scheduling algorithm that takes into account the owners of the processes. It is used to ensure all users of a system get a fair share of the CPU by allocating CPU among users/groups instead of processes. This ensures that a user does not run much more processes than another user, and use much more CPU than another user. Each user is allocated some fraction of the CPU. In **equal share**, N users each get $100/N\%$. In **unequal share**, important users get a big chunk of CPU time, but even more important users get an even

bigger chunk of CPU time. All processes belonging to an owner have to share the owner's CPU share.

Example 8.3 (Equal Share)

User 1 has 50% CPU share, and the five processes A, B, C, D, and E. User 2 has 50% CPU share, and one process F. A possible scheduling sequence could be A F B F C F D F E F A F B F C F D F E F A F B F C F D F E F ...

Example 8.4 (Unequal Share)

User 1 has 75% CPU share, and the five processes A, B, C, D, and E. User 2 has 25% CPU share, and one process F. A possible scheduling sequence could be A B C F D E A F B C D F E A B F C D E F A B C F ...

- **Multilevel Queue Scheduling (MQ)**: This is a preemptive time-sharing scheduling algorithm. The ready queue is partitioned into separate queues. For instance, there is a **foreground queue** for interactive processes such as browsers and games, and a **background queue** for non-interactive process such as weather widgets and web servers. A process is permanently assigned to one of the queues. Each queue can have a different scheduling algorithm. For instance, the foreground queue uses RR with a time slice of $10ms$ while the background queue uses RR with time slice of $100ms$. Scheduling is done based on queues:

  1. **Static priority scheduling** occurs when each queue has a different, but fixed priority. The scheduler processes jobs from the highest priority queue, until it is empty. Once empty, it switches to the next highest priority queue. Starvation is a problem.

     Example 8.5

     System processes are the highest priority. Priority then decreases as we reach interactive processes, interactive editing processes, batch processes, and eventually student processes.

  2. **Fixed CPU share for each queue** occurs when background and foreground processes obtain a fixed percentage of CPU. This is not a very dynamic solution

     Example 8.6

     The background gets 20% of the CPU, while the foreground gets the remaining 80%. Thus, processes in the background queue would share 20% of the CPU, while processes in the foreground queue share 80% of the CPU.

- **Multilevel Feedback Queue Scheduling (MFQ)**: This is a scheduling algorithm similar to multilevel queues, since there are multiple queues, each representing a different priority. Scheduling is also done based on queues, and each process belongs to a queue. Now, a process can move between queues (this is a form of aging). This now solves the starvation problem, as we can dynamically react to a job changing from CPU-bound to IO-bound. CPU-bound processes move to

low priority queues, while IO-bound move to high priority queues. The scheduler is defined by the number of queues, the scheduling algorithms for each queue (including different time slices), when to move a process between queues, and to which queue a process is assigned when the process needs service.

Example 8.7

There are three queues, Q_1, Q_2 and Q_3. All three queues are using RR scheduling with time-slices of $8ms$, $16ms$ and $32ms$ respectively. All jobs in Q_1 are processed first, then Q_2, then Q_3. New jobs enters Q_1. If a job in Q_1 does not finish in $8ms$, it is demoted to Q_2. If a job in Q_2 does not finish in $16ms$, it is demoted to Q_3. If a job in Q_2 or Q_3 waits for too long, it is promoted up to Q_1 (or Q_2).

- **Lottery Scheduling (L)**: This is a preemptive time-sharing algorithm where each process gets some number of "lottery tickets". The number of tickets is based on process priority. Higher priority results in more tickets. The scheduler picks a random number and the process with that ticket obtains a time-slice of CPU. Higher priority processes have a higher chance of running. Over a long time, the job's priority will determine the job's total CPU share. Cooperating processes may exchange tickets, allowing dynamic fine-tuning of the priorities based on needs. For instance, in a producer/consumer setting, the producers could swap tickets with consumers. Starvation is not a problem, but there are some technical issues with implementing efficient algorithm for large numbers of tickets and jobs.

In some of the above algorithms, we make use of the concept of priority, where some processes have a higher priority than others. **Priority Scheduling** concerns a set of scheduling algorithm that can schedule processes with different priorities. Important processes should get more CPU time than less important processes. The higher the priority of a process, the higher the CPU time. Priority could be determined by the status of the owner. A background process (such as bittorrent) could have lower priority than an interactive one (such as a game). Priorities can be **static** or **dynamic**.

Example 8.8

Static priority could be setting the bittorrent priority to 10, while the game priority is set to 50. Dynamic priority could be giving I/O-bound processes higher priority than CPU-bound processes.

Example 8.9 (Non-Preemptive Priority)

To implement priority scheduling with non-preemptive scheduling, a simple approach to handle priorities could work. Consider SJF-like scheduling, where the ready queue is sorted by priority. All jobs eventually finish, although a newly added job with very high priority might have to wait.

Example 8.10 (Preemptive Priority)

Accounting for priority with preemptive scheduling is more complicated. Consider RR-like scheduling, where the ready queue is sorted by priority. The problem is starvation, since a high priority CPU-bound process would ensure no other processes get to run at all.

§8.3 Scheduling on Real Time Systems

Programs for real time systems are generally expressed as a set of event handlers or **tasks** responding to events. For example, the handling of interrupts from a device or timer. A task in a real time OS CPU (RTOS) must respond within a fixed amount of time from the time of the event (**deadline**). Task types can be considered **periodic** if they occur at regular intervals/periods, or **aperiodic** if they occur unpredictably with deadlines for start and/or finish. Correctness depends both on the logical result, as well as the response time. Tasks have deadlines:

- **Hard deadline**: It must meet its deadline. A miss will cause system failure, so it needs **Hard RTOS**. An example would be missing an interrupt in a pacemaker device.
- **Soft deadline**: The occasional miss is acceptable. It needs **Soft RTOS**. An example would be a game lag, or diminishing level of detail.
- **Firm deadline**: This is between hard and soft. An infrequent miss is tolerable, but the value of task completion is 0 after the deadline. For example, in automated manufacturing, a few bad products are okay.

Below are the main scheduling algorithms for real time systems.

- **Rate Monotonic Scheduling (RM)**: This is a simple algorithm that is well suited for periodic tasks. A task has a static priority based on the inverse of the period of the task. Thus, a shorter period results in a higher priority, and a longer period results in a lower priority. It is usually preemptive, as an event associated with a higher priority task will preempt lower priority tasks. This simple formula can be used to determine whether a set of tasks is **schedulable**. This means that a schedule exists where no deadlines are missed. A new task could be rejected if the system would become unschedulable. This is a very common RTOS scheduler.
- **Earliest Deadline First Scheduling (EDF)**: This is a scheduling algorithm similar to RM scheduling. A task with a shorter deadline has a higher priority (dynamic). For instance, the scheduler picks a task with the earliest deadline. It is usually a preemptive scheduler, as an event for higher priority tasks will preempt lower priority tasks. It features optimal dynamic priority scheduling. If a schedule exists, EDF will find it. It works better for aperiodic tasks than RM, and can achieve 100% CPU utilization. This simple formula can determine whether tasks can be scheduled without missing deadlines.

§8.4 Thread Scheduling

User level and kernel level threads can similarly be scheduled.

- User level threads can be scheduled when the kernel assigns a quantum to a process. Threads within the process share the quantum. Each process has its own thread scheduler. There is no clock to interrupt a thread that runs too long. There is very fast context switching.

> **Example 8.11**
> The kernel picks a process to execute. This could be at a $50ms$ time slice, so there are $5ms$ CPU bursts per thread. The runtime system then picks a thread to run. Thus, for two processes A and B, each with three threads, a possible order of execution could be A_1, A_2, A_3, A_1, A_2, A_3. It is not possible for A_1, B_1, A_2, B_2, A_3, B_3.

- Kernel level threads can be scheduled when the kernel assigns quantum to threads. A full context switch is required. A thread blocking on I/O will not suspend the entire process. Modern schedulers will often pick the next thread from the same process to reduce context switching costs.

> **Example 8.12**
> The kernel picks a thread to execute. This could be at a $50ms$ time slice, so there are $5ms$ CPU bursts per thread. Thus, for two processes A and B, each with three threads, a possible order of execution could be A_1, A_2, A_3, A_1, A_2, A_3. It is also possible for A_1, B_1, A_2, B_2, A_3, B_3.

§8.5 Scheduling Algorithms

Many modern day operating systems make use of different algorithms, with different levels of preemption. Amiga OS has preemption and uses prioritized round robin scheduling. Solaris and macOS have preemption, and use multilevel feedback queues. The latest Linux OS has preemption and uses a completely fair scheduler. Windows 95 had half preemption, and used a preemptive scheduler for 32 bit processes, and a cooperative scheduler for 16 bit processes.

We also have other schedulers:

- **Long-term scheduling** decides which programs to run and which ones to delay. Typical goals are to achieve a good mix of CPU-bound and I/O-bound processes. This is important on batch systems.
- **Medium-term scheduling** decides which programs to swap in and out when running low on resources, or when a process is idle for too long.
- **Short-term scheduling** is also known as **CPU scheduling**, and is responsible for allocating CPU to processes in memory. For instance, it is responsible for managing ready/blocking/waiting states.
- **I/O scheduling** is responsible for ordering in the I/O queue to increase throughput.

§9 June 13, 2017

§9.1 Deadlocks

A set of processes is **deadlocked** if each process in the set is waiting for an event, and that event can be caused only by another process in the set. An event could be a resource

becoming available, mutex/semaphore/spinlock being unlocked, a message arriving, etc.

A **system** consists of n processes $P_1, P_2, ..., P_n$, and m resource types $R_1, R_2, ..., R_m$. The resource types could be CPU, memory space, I/O devices, etc. Each resource type R_i has W_i instances. For instance, we have 1 CPU, 5 disks, and 3 printers. Each process utilizes a resource in the same manner. First, it **requests** a resource (may block), then it **uses** a resource for a finite amount of time, then **releases** a resource (potentially unblock related processes).

A deadlock can arise only if four conditions hold simultaneously:

1. **Mutual Exclusion**: The involved resources must not be shareable. There is a maximum of one process per resource.
2. **Hold and Wait**: A process holding at least one resource is waiting to acquire additional resources.
3. **No Preemption** A resource can be released only by the process holding it voluntary.
4. **Circular Wait** There is an ordering of processes $\{P_0, P_1, ..., P_n\}$ such that P_1 waits for P_2, P_2 waits for P_3, ..., and P_n waits for P_0. Thus, there is a cycle.

Example 9.1 (Trivial Deadlock Example)

Deadlocks can occur via system calls, locking, etc. A simple deadlock example consists of two mutexes, where both need to be locked in order to access the critical section. Two threads that lock in the reverse order can set up a deadlock when both lock one mutex, but not the other. All 4 necessary conditions are present.

§9.2 Resource Allocation Graph

A **resource allocation graph** consists of a set of vertices V and a set of edges E. V is partitioned into two types, with $P = \{P_1, P_2, ..., P_n\}$ being the set consisting of all the processes in the system, and $R = \{R_1, R_2, ..., R_m\}$ being the set consisting of all resource types in the system. The request edge is a directed edge from $P_i \to R_j$, while the assignment edge is a directed edge from $R_j \to P_i$.

Example 9.2

A process P_i could be represented by a circle with P_i written inside. Resource R_j with three instances could be represented by a square with three dots inside. An arrow from P_i to R_j would indicate that P_i is requesting an instance of R_j. An arrow from one of the dots in R_j to P_i would indicate that P_i is holding an instance of R_j.

When there is no cycle in a resource allocation graph, this means that there is no deadlock. However, we should be careful in our interpretation of these graphs, as a cycle does not necessarily imply a deadlock. A cycle is one component that is necessary for a deadlock to occur. If the graph has a cycle and there is only one instance per resource type, then this guarantees a deadlock. If the graph has a cycle and there are multiple instances per resource type, then there may be a deadlock. Operation order is important, as the same processes with the same resource requests may lead to a deadlock in one order, and not lead to a deadlock in another.

To deal with deadlocks, we may choose to ignore the problem by pretending that the deadlocks have never occurred in the system. This is the approach of many operating systems, including UNIX. It is up to the applications to address their deadlock issues. We may alternatively choose to ensure that the system will never enter a deadlock state through **deadlock prevention** or **deadlock avoidance**. We may also choose to allow the system to enter a deadlock state and then recover through **deadlock detection** or **recovery** from deadlock.

§9.3 Deadlock Prevention

Prevention is achieved by attacking one of the four necessary conditions.

- Avoiding **mutual exclusion** condition: Mutual exclusion is not required for sharable resources such as read only files. However, it is necessary for non-sharable resources. Thus, it is not practical in most cases. **Spooling** can help for some resources such as printers.
- Avoiding **hold and wait** condition: We must guarantee that whenever a process requests a resource, it does not hold any other resources. The first option is that a process must request all needed resources at the beginning. The second option is that a process can request resources only when is has no resources. However, this may lead to low resource utilization, and starvation is possible.

  Example 9.3

  To avoid the trivial deadlock example with two locks that must be locked, we can avoid the hold and wait condition by locking both mutexes at the same time. By acquiring all resources at the beginning, we have avoided the hold and wait condition. Instead of using `lock()`, we can use `lockn()`, which atomically locks multiple mutexes at once. Alternatively, we may choose to lock the first mutex, then use `unlockAndLock()`, which unlocks all locked mutexes first, then locks them all. This follows the principle of releasing resources before acquiring more. The related general concept is called **two-phase locking**.

- Avoiding **no preemption** condition: If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released, and the process is suspended. Preempted resources are added to the list of resources for which the process is waiting. The process will be resumed when it can regain its old and new resources. This only works with resources for which we can save and restore the state (such as CPU registers). It is a complicated mechanism that may lead to possible starvation. There is non-optimal use of resources.
- Avoiding **circular wait** condition: This is the most practical condition to avoid. For instance, we could impose a total ordering of all resource types, and require that each process requests resources in an increasing order of **enumeration**.

  Example 9.4

  In the trivial deadlock example, we could change it so that both the first and second thread lock `mutex1` before locking `mutex2`. By locking mutexes in the same order in all threads, we can avoid a deadlock.

Example 9.5 (Deadlock with Lock Ordering)

Two transactions execute concurrently. Transaction 1 transfers twenty five dollars from account A to account B, while Transaction 2 transfers fifty dollars from account B to account A. This could lead to a potential deadlock. We avoid the circular wait condition by ordering resources.

§9.4 Deadlock Avoidance

Deadlock prevention schemes can lead to low resource utilization. On the other hand, **deadlock avoidance** can increase resource utilization if some **a priori information** is available. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition. **Resource-allocation state** is defined by the number of available and allocated resources, and the maximum demands of the processes.

When a process requests an available resource, the system must decide if immediate allocation leaves the system in a **safe state**. If the new state is safe, the request is granted. Otherwise, request is denied and the process waits. The system is in a safe state if there exists a sequence $< P_1, P_2, ..., P_n >$ of all running processes in the system where they can all finish. When the system is in a safe state, no deadlocks are possible. If the system is in an unsafe state, then there is a possibility of a deadlock. Avoidance ensures that a system will never enter an unsafe state, regardless of whether a deadlock is possible.

§9.5 Deadlock Avoidance Algorithm

When there is a single instance per resource type, we use a resource-allocation graph algorithm. The **claim edge** from $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j. This is represented by a dashed line. The claim edge converts to a request edge when a process requests a resource. The request edge converts to an assignment edge when the resource is allocated to the process. When a resource is released by a process, the assignment edge reconverts to a claim edge. Resources must be claimed a priori in the system. A cycle occurs when there is a directed route that leads to the same starting position.

Example 9.6 (Resource Allocation Graph Algorithm)

Suppose that we have two processes P_1 and P_2, and two resources R_1 and R_2. P_1 holds R_1, and P_2 requests R_1. P_1 may request R_2, and P_2 may request R_2. Suppose that a process requests a resource. The request can be granted only if allowing the request does not violate safe state. That is, converting the request edge to an assignment edge does not result in the formation of a cycle. This cycle can be comprised of any type of arrow, so long as there is a directed path back to the starting position.

When there are multiple instances per resource type, we use the banker's algorithm instead. This is a more general avoidance algorithm than the resource-allocation graph algorithm. It works with multiple instances per resource type. For the algorithm to work, each process must a priori claim its maximum use of resources. When a process requests

a resource it may have to wait (even if the resource is available). When a process gets all of its resources, it must return them in a finite amount of time.

§9.6 Banker's Algorithm

Let n be the number of processes, and m be the number of resources types. $Available$ is a vector of length m, where $Available[j] = k$ means there are k instances of resource type R_j available. Max is an $n \cdot m$ matrix, where $Max[i, j] = k$ means that process P_i may request at most k instances of resource type R_j. $Max[i]$ is the ith row of Max. $Allocation$ is an $n \cdot m$ matrix, where $Allocation[i, j] = k$ means that process P_i is currently allocated k instances of R_j. $Need$ is an $n \cdot m$ matrix where $Need[i, j] = k$ means that P_i may need k more instances of R_j to complete its task. Note that $Need[i, j] = Max[i, j] - Allocation[i, j]$.

Now, we need to find a sequence of process executions that would finish all running processes. We list the processes on the vertical axis, and the resources on the horizontal axis. The resources are repeated for each of the headings of $Allocation$, Max, and $Need$. $Available$ is the number of instances of each resource remaining after the initial allocation. We determine that the state is safe when there is some possible sequence of adding the available resources to a particular process so that it obtains the resources that it needs. When it is finished, all of the resources given to the process are returned and now available. This is repeated until all processes have finished.

Example 9.7 (Banker's Algorithm)

Given the state of the system, we can additionally consider whether it is possible to grant an additional request of resources for a particular process. To do so, we change the available resources and add them to the particular process allocation. We then calculate the needed resources again for each process. Then, we try to deduce whether this new state is safe or not. We can grant the request when there is a possible execution sequence.

§9.7 Deadlock Detection

Deadlock detection allows a system to enter a deadlock state. Later, we detect the deadlock and recover. The detection algorithm may be for a single instance per resource type, or multiple instances per resource type.

Deadlock detection with a single instance per resource type can be achieved by maintaining a **wait-for** graph. The nodes are processes, with $P_i \to P_j$ if P_i is waiting for P_j. This can be obtained by collapsing the resource-allocation graph. Detection involves periodically invoking an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.

Example 9.8

We can convert a resource-allocation graph to its corresponding wait-for graph by removing the resources. The arrows and their directions are maintained. This indicates the processes that are waiting on each other.

Deadlock detection with multiple instances per resource type is achieved differently. We let n be the number of processes, m be the number of resource types, $Available[]$ be the vector of length m that indicates the number of available resources of each type,

Allocation[][] be the $n \cdot m$ matrix that indicates the current allocation of resources per process, and *Request*[][] be the $n \cdot m$ matrix that indicates the current requests for resources per process. We use a strategy analogous to the banker's algorithm, as we try to determine if there is a sequence in which running processes can finish executing.

Remark 9.9. The algorithm requires $O(m \cdot n^2)$ operations to detect whether the system is in a deadlock state.

We list the processes on the vertical axis. The resource types are listed on the horizontal axis, with each repeated under the headings of *Allocation* and *Request*. *Available* is the number of instances of each resource remaining after the initial allocation. The goal is to find a sequence that will result in the processes finishing.

Example 9.10 (Detection Algorithm)

We can additionally consider whether an additional resource request by a particular process is possible. Consider the processes that can run and finish, with the system reclaiming resources when these processes finish. If there remains a process with insufficient resources to meet its request, then a deadlock exists. The deadlock consists of those processes with insufficient resources to run.

Detection algorithms are expensive and cannot check on every resource request. Other ideas for invoking detection involve checking every few minutes, or checking when the CPU goes idle. The question of when and how often detection is used depends on how often a deadlock is likely to occur, and how many processes are affected. We can use one for each disjoint cycle. If we check too often, we spend too many CPU cycles on useless work. But if we do not check often enough, there may be many cycles in the resource graph and we would not be able to tell which of the many deadlocked processes caused the deadlock.

§9.8 Deadlock Recovery

There are three main ways to recover from a deadlock:

1. **Process Termination**: We could abort all deadlocked processes. This is simple, but often a bad choice. We could alternatively abort one process at a time until the deadlock cycle is eliminated. This is a better choice, but we need to decide the order in which we abort. This order could depend on the priority of the process, the age of the process, how much longer till completion, the resources the process has used, the resources the process needs to complete, how many processes will need to be terminated, or whether the process is interactive or batch.

2. **Process Rollback**: This uses an idea similar to process termination, but programs can cooperate. Programs can be written to periodically save the current state (**checkpoint**). When restarted, the program detects a checkpoint and resumes computation from the last checkpoint (**rollback**). Programs can checkpoint themselves just before requesting resources. Thus, when a deadlock is detected, a program can be terminated and re-scheduled to run later. For instance, it could be run after the other affected deadlocked processes are finished. This scheme does not work well with all resource types (such as printers). However, it is useful for long computations and simulations.

3. **Resource Preemption**: When a deadlock occurs, we first pick a victim process. This victim process is suspended. We then save the state of the victim's resources, and give the victim's resources to other deadlocked processes. When the other processes release the resources, we restore their state. Then, the resources are returned to the victim. The victim is then unsuspended. This uses a similar idea to rollback, but instead of checkpointing the program, we checkpoint the resources of the program. However, this only works with some resource types.

§10 June 15, 2017

§10.1 Address Space

Operating systems need to run multiple processes simultaneously. Each process needs some amount of memory. There are several concerns related to memory management. Firstly, the OS must give each process some portion of the available memory (**address space**). However, it also needs to determine which part of memory and how much memory to give each process. The OS must also protect the memory given to one process from other processes, but there needs to be a method to accomplish this. If programmers do not know where the program will be loaded, how do they write code? Working with **physical (direct) addresses** is not a good solution.

Example 10.1

When working with physical addresses, we may need to load two programs into physical memory in RAM. Doing so changes the memory address of each instruction. However, program instructions such as `JMP 28` tell the program to jump to a specific location in memory. With two programs loaded, this may result in the program jumping to another program. Protection is another problem that arises from working with physical addresses.

A pair of **base** and **limit registers** define the allowed range of addresses available to the CPU. The base register indicates the starting memory, and the limit register indicates the size of memory. The base and limit registers can only be modified in kernel mode. The CPU checks every memory access generated by a process. When the process tries to access an invalid address, it traps to the OS. Base and limit registers are stored in the PCB.

Example 10.2

To visualize this, we can consider the OS as starting at 0 in memory, up to some bound. This bound is the base for a new process, until it reaches the limit. This is then the base of a new process, which has another limit.

Address protection in hardware is accomplished by a series of comparisons when the CPU tries to access an address. If the address is greater than or equal to the base, and less than the base plus the limit, then it can access memory. Otherwise, it traps to the operating system monitor, as there is an addressing error.

§10.2 Address Binding

When programs are written, the physical address space of the process is not known. A possible solution would be to write programs in a way that allows them to be **relocated**.

When needed, we can bind the addresses to the actual memory location. The addresses in a program are represented in different ways at different stages of a program's life. To accomplish this, source code addresses are usually symbolic (`int main()`), addresses in compiled code can bind to relocatable addresses (`main = "14 bytes from beginning of this module"`), and before execution, the loader will bind the relocatable addresses to physical addresses (`main = "14 bytes from beginning of this module" = 1014`). Each binding maps one address space to another.

Address binding of instruction and data to memory addresses can happen at three different stages:

1. **Compile Time**: If the memory location is known a priori, **absolute code** can be generated and stored. We must recompile the code if the starting location changes.
2. **Load Time**: The compiled code must be stored as **relocatable code**. Binding is done before the program starts executing.
3. **Execution Time**: If the process can be moved during its execution, binding is done at run-time dynamically. This needs hardware support for address maps, such as from the memory management unit.

Example 10.3

The source program is sent to the compiler or assembler. The time spent here is compile time. It is then sent to the object module. Along with other object modules, this enters load time as it is sent to the linkage editor. This leads to the load module. Along with a system library, this is fed to the loader. The result is combined with a dynamically loaded system library as it leaves load time and enters execution time (runtime). Runtime consists of in-memory binary memory images, where the dynamically loaded system library was dynamically linked.

Execution time address binding and memory protection can be achieved by virtualizing memory. Each process is given a **logical address space** (**virtual address space**). It is a contiguous space, ranging from 0 to MAX. Addresses that are generated by the CPU as a process executes are known as **logical addresses** in this space. If the logical address does not fall into the logical address space range, this is a violation that leads to a trap. **Physical address** is a real memory address. Logical addresses are mapped to physical addresses before reaching memory. The **physical address space** of a process is the subset of RAM allocated to a process, and is the set of all mappings from logical addresses.

§10.3 Memory Management Unit

The **Memory Management Unit (MMU)** is a hardware device that maps virtual addresses to physical addresses, and is often a part of the CPU. There are many possible implementations. A CPU executing a process uses logical addresses, as it never sees the real physical addresses. Execution-time binding occurs automatically whenever a memory reference is made. The CPU sends logical addresses to the MMU, which then sends physical addresses to memory.

Example 10.4

A simple MMU implementation consists of a CPU with a **relocation register**. The value in the relocation register (14000) is added to every address generated by a process at the time it is sent to memory. Thus, when the CPU sends a logical address (346) to the relocation register, it adds the value and then accesses memory at that physical address (14346).

It is possible to combine the relocation register with the limit register. The relocation (base) register is the smallest allowed physical memory address, while the limit register is the size of the chunk of physical memory a process is allowed to use. This achieves execution-time binding and memory protection.

Example 10.5

The CPU sends a logical address to the limit register. If it is less than the limit register, it is permitted to proceed. Otherwise, there is a trap since there is an addressing error. After passing the limit register, it reaches the relocation register where a value is added. It then accesses the physical address in memory. For instance, one program uses memory from 0 to 16380. The second program has a base (relocation) register that starts at 16384. The limit register is set to 16384.

§10.4 Swapping

A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution. The **backing store** is a fast disk large enough to accommodate copies of all memory images for all processes. Swapping allows the OS to run more processes than the available physical memory.

Example 10.6

The operating system and user space operate with main memory. A process P_1 may be swapped out from the user space into the backing store. Another process P_2 can then be swapped in from the backing store into user space.

However, we need to consider whether the swapped out process needs to swap back into the same physical addresses. This usually depends on the address binding method. This is not a problem if MMU is used. We also must be careful with pending I/O, especially when using memory-mapped device registers. I/O results could be sent to the kernel, then to the process (double-buffering). Additionally, context switch times can be extremely high.

Standard swapping is not used in modern operating systems, but modified versions of swapping are used on many systems, such as Linux and Windows. On these systems, swapping is disabled initially, and started if more than the threshold amount of memory is allocated. It is then disabled once the memory demand reduces below the threshold.

Example 10.7 (Swapping and Memory)

Memory allocation changes as processes are swapped in and out. For instance, we consider the usage of memory over time. The operating system is always at the bottom, taking up some amount of memory. Process A comes in and fills the next available memory location. The same is true of processes B and C that arrive later. Process A is now swapped out, while a process D with less memory requirement fills in some of the space left by A. Process B is then swapped out, while Process A swaps back in, taking some of its original space, and some left by process B.

§10.5 Memory Allocation

Memory allocation is concerned with the memory that the OS allocates to each process. It is the case that most programs increase their memory usage during execution. A possible solution is to swap processes out, find a bigger free memory chunk, then swap it back in. A better solution is for the OS to allocate extra memory for each process.

Example 10.8

There is some memory that is actually in use, and some that is allocated room for growth for each process. The actual program, data, and stack are in use, while the remaining memory is room for growth.

At some point, the OS needs to find a free chunk of memory, mark it as used, and later free it up. Simple approaches can lead to **fragmentation** with lots of tiny free chunks of memory, none of which are big enough to satisfy any requests. The OS needs to manage the memory in an efficient way that is both fast, and minimizes fragmentation. The two general approaches are fixed partitioning and dynamic partitioning.

- **Fixed Partitioning**: Memory is divided into equal sized partitions. The problem with this approach is that this can lead to **internal fragmentation** since memory internal to a partition becomes fragmented. This leads to low memory utilization if partitions are large.

  Example 10.9

  For instance, the total memory is $64MB$, minus $8MB$ taken by the OS. The partition size is $8MB$. Thus, to load 3 processes $P_1 = 4MB, P_2 = 8MB$, and $P_3 = 10MB$, the first process takes up a part of the first partition, the second process takes up the entire second partition, and the third process takes up the third and part of the fourth partition.

  In the above example, the actual free memory is $34MB$, but the usable free memory is $24MB$ since we cannot access the memory available in partition 1 and partition 4.

- **Dynamic Partitioning**: Partitions are created to fit a request perfectly. There is no more internal fragmentation, but there is now **external fragmentation** since the memory that is external to all partitions becomes increasingly fragmented. This leads to low memory utilization.

Example 10.10

With the same situation, we note that the OS, P_1, P_2, and P_3 now take up one partition that is exactly $8MB + 4MB + 8MB + 10MB$ in size. Thus, the actual free memory of $34MB$ is exactly the same as the usable free memory of $34MB$.

In the above example, we consider the case that P_2 finishes when $P_4 = 18MB$ arrives. We now have $24MB$ of free memory, where the actual free memory is equal to the usable free memory. However, a fifth process $P_5 = 17MB$ could not start since there is no contiguous block of memory that is $17MB$.

Memory compaction occurs when the OS periodically rearranges the used blocks of memory so that they are contiguous. Free blocks are merged into a single large block. This is a CPU intensive operation. To accomplish this, we need a way to keep track of free memory and allocated memory. Ideally, we would like to use a data structure that is efficient at searching and reclaiming free space, and can deal with fragmentation:

- **Bitmaps and Fixed Partitions**: Memory is divided into equal partitions as small as a few words and as large as several KB. The OS maintains a bitmap, with one bit per partition, where 0 means the partition is free, while 1 means the partition is occupied. Searching is an $O(n)$ operation, where n is the size of the bitmap. Smaller partitions lead to less fragmentation, but a larger bitmap. The reverse is true of larger partitions.

  Example 10.11

  Process A takes up 5 partitions. This is followed by 3 empty partitions, then process B which takes up 5 partitions. The bitmap would then be 1111100011111.

- **Linked Lists**: Memory is divided into segments of dynamic size. The OS maintains a list of allocated and free memory segments (**holes**), sorted by address. Searching is an $O(n)$ operation, where n is the number of segments in the linked list. Reclaiming free space can be an $O(1)$ operation if a doubly-linked list is used and linked list data is stored within segments. With linked lists, when a process X terminates, its location in memory is cleared. There is no shifting of other processes once X has terminated.

  Example 10.12

  The list segment would store P representing process or H representing hole. The segment would also store where this starts, and its length, followed by a pointer to the next segment. Instead of a pointer to the next segment when we arrive at the last segment, we represent this with an X instead. Thus, with the previous example, we would have $(P, 0, 5)$ indicating the first process pointing to $(H, 5, 3)$ indicating the hole. This would then point to $(P, 8, 5, X)$.

There are different algorithms for finding free space (holes) in a linked list:

- **First Fit**: Find the first hole that is big enough. Leftover space becomes a new hole.

- **Best Fit**: Find the smallest hole that is big enough,. Leftover space becomes a new hole. This new hole is very likely useless because of its small size.
- **Next Fit**: This is the same as first fit, but we start searching at the location of last placement.
- **Worst Fit**: Find the largest hole. Leftover space is likely to be usable.
- **Quick Fit**: Maintain separate lists for common request sizes. This leads to faster search, but results in more complicated management.

§10.6 Virtual Memory

Virtual memory is a memory management technique that allows the OS to present a process with a logical address space that appears contiguous. Physical address space can be discontiguous, with some parts of logical address space mapped to a backing store. This improves memory management and allows parts of programs to be "swapped" in and out.

> **Example 10.13**
> A process' virtual memory may have discontiguous parts of it mapped to a disk, while the remaining parts are mapped to discontiguous parts of physical memory. The remaining parts of physical memory are mapped to from another process.

Paging is the most common virtual memory implementation. The virtual address space of the process' virtual memory is divided into **pages** of fixed size blocks, usually powers of 2 ranging from $512B$ to $16MB$. Physical memory is divided into **(page) frames** that are the same size as pages. The pages map to frames via a lookup table called a **page table** that has logical to physical address mappings. This avoids external fragmentation, since there are no holes. Furthermore, each process has its own page table (ptr in PCB).

If a program tries to address a page that does not map to physical memory, the CPU issues a trap called a **page fault**. The OS suspends the process, loads the page from the disk, updates the page table, then resumes the process. If the OS only loads pages as a result of a page fault, we call that **demand paging**.

> **Example 10.14**
> The CPU package consists of the CPU and the memory management unit. The CPU sends the virtual addresses to the MMU within the CPU package. The MMU then sends the physical addresses to the memory. This occurs as a bus takes this data out of the CPU package and into memory or the disk controller.

In paging, the OS keeps track of all free frames. This means that we still need memory management techniques. To run a program of size N pages, the OS needs to find N free frames before loading the program. The page table translates the logical addresses to physical addresses. A similar mechanism is needed for the backing store, since it is also split into pages. Unfortunately, we still have internal fragmentation with paging.

Example 10.15

Given that the virtual address space is $64B$, the physical address space is $32KB$, and the page size is $4KB$, we know that the frame size must be the same as the page size, so it is also $4KB$. The number of pages is the virtual address space divided by the page size, so there are 16 page entries. The number of frames is equal to the physical address space divided by the page size, so there are 8 frames.

Example 10.16

The page size is $2KB$ and a process needs $71KB$ to load. We note that we need $35 + 1 = 36$ pages. Thus, the OS needs to find 36 free frames. However, these frames do not need to be contiguous, since the OS can allocate 36 frames anywhere. We observe that one frame will have $1KB$ of unused space (internal fragmentation). There is no external fragmentation since there are no holes between frames.

§10.7 Paging Performance

Demand paging performance is commonly evaluated by the **effective access time** for memory access. Let p be the probability of a page fault (**page fault rate**) where $0 \leq p \leq 1$. When $p = 0$, this means that all pages are in memory, so there is no page fault. When $p = 1$, this means that all pages are on disk, so all memory accesses are page faults. Let ma be the memory access time, and $pfst$ be the page fault service time, which is the time it takes to service a page fault. Then, effective access time EAT is given by

$$EAT = (1 - p) * ma + p * pfst.$$

Example 10.17

A non-realistic example would be to calculate EAT when the page fault probability is 50%, $ma = 1ms$ and $pfst = 10ms$. We would find that $EAT = (1 - 0.5) * 1 + 0.5 * 10 = 0.5 + 5 = 5.5ms$. A realistic example would be to calculate EAT when $p = 1/1000$, $ma = 100ns$ and $pfst = 10ms$. We would find that $EAT = (1 - 0.001) * 100ns + 0.001 * 10000000ns = 99.9ns + 10000ns \approx 10099.9ns$. We note in the realistic example that the effective access time is almost 101 times slower than the memory access time.

§10.8 Paging Hardware and Models

The CPU uses an **address translation scheme**. The address generated by the CPU is divided into the **page number** p, which is used as an index into a page table that contains the base address of the corresponding frame in physical memory, and the **page offset** d, which is combined with the base address to define the physical memory address that is sent to the memory unit. For a given logical address space of size 2^m and page size 2^n, the page number uses $m - n$ amount of space, while the page offset uses n amount of space.

Example 10.18

A 16 bit logical address $1010101001100101b$, and page size of 2^{10} would be split so that the page number is $101010b$ and the page offset is $1001100101b$.

Example 10.19 (Paging Hardware)

The CPU generates p and d. p is sent to the paging table, where this is used as an index to find the base address of the corresponding frame in physical memory. This frame f is combined with d to obtain the physical address. This is then used to access physical memory, since f provides the base address of the frame in physical memory, and d then permits access to the physical memory address. This is between $f0000...0000$ and $f1111...111$.

Example 10.20 (Paging Model of Logical and Physical Memory)

Logical memory is entirely split up into Pages 0 to 3. We have a page table that maps 0 to 1, 1 to 4, 2 to 3, and 3 to 7. Thus, in physical memory, Page 0 is in Frame 1, Page 1 is in Frame 4, Page 2 is in Frame 3, and Page 3 is in Frame 7. Frames 0, 2, 5, and 6 are not used.

Suppose instead we have four 4 byte pages of logical memory that needs to be mapped with 32 byte physical memory. The page table maps 0 to 5, 1 to 6, 2 to 1, and 3 to 2. Thus, Page 0 is mapped to $5 * 4 = 20$, Page 1 is mapped to $6 * 4 = 24$, Page 2 is mapped to $1 * 4 = 4$, and Page 3 is mapped to $2 * 4 = 8$. Note that each page is 4 bytes, so each frame needs to be 4 bytes.

Example 10.21 (Free Frames)

Before allocation, we have a free-frame list that consists of 14, 13, 18, 20, and 15. A new process has Page 0 to 3. We now consult a new-process page table that maps 0 to 14, 1 to 13, 2 to 18, and 3 to 20. After allocation, the free-frame list contains only 15, as the pages are mapped accordingly.

§10.9 Page Tables

The page table is kept in main memory. The **page-table base register (PTBR)** points to the page table, while the **page-table length register (PTLR)** indicates size of the page table. In this scheme, every data or instruction access requires at least two memory accesses, consisting of an instruction fetch and page table lookup. The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation lookaside buffers (TLB)**. TLB are extremely fast and extremely small (64 to 1000 entries). On a TLB miss, the value is loaded into TLB for faster access next time.

Associative memory permits a parallel search on content. The TLB stores a subset of the page table, and searches based on the page number, returning the corresponding frame number. This search is done in parallel. If the TLB does not contain a certain page number, then it must be obtained from the page table in memory.

Example 10.22 (Paging Hardware with TLB)

The process is generally the same as outlined in the example with Paging Hardware. Now, p is checked against all entries in the TLB. If there is a hit, then we have found f. Otherwise on a TLB miss, we consult the page table. The rest of the process is the same.

Memory protection is implemented by associating a **protection bit** with each frame to indicate whether read-only or read-write access is allowed. We can also add more bits to indicate page execute-only, and so on. There is also a **valid-invalid** bit attached to each entry in the page table. Valid indicates that the associated page is in the process' logical address space, and is thus a legal page. Invalid on the other hand, indicates that the page is not in the process' logical address space. We could alternatively use the page-table length register (PTLR). Any violations will result in a trap to the kernel.

Example 10.23 (Structure of Page Table Entry)

A page table entry usually consists of a caching disabled bit, a referenced bit (this is set by hardware automatically on any access), a modified bit (also known as a dirty bit, this is set by hardware automatically on write access), a protection bit (this includes various bits such as read, write, and execute), and a present/absent bit (also known as a valid/invalid bit, this indicates a page fault when invalid). This is then followed by the page frame number.

§11 June 20, 2017

§11.1 Page Table Implementations

Sometimes, it can be useful for processes to share some memory with other processes. This is implemented using **shared pages**. One could be running multiple instances of the same program, or programs using a shared library. Only one copy of the executable code needs to be in physical memory. This is implemented using shared **read-only** pages (**read-only bit** in page table entry). Shared pages are also useful for interprocess communication, which may be implemented using shared read-write pages.

Example 11.1

Consider three processes P_1, P_2, and P_3. Each of them contain ed_1, ed_2, and ed_3 in their first four pages. The last page of each of these programs contains data for their specific process. Their page tables would show similar entries for the first three pages. The last page would be mapped to different locations. Thus, the frame could store $data_1$ from P_1 in 1, $data_3$ in 2, ed_1 (used in all programs) in 3, ed_2 in 4, ed_3 in 6, and $data_2$ in 7.

Page tables can get very large using straight-forward methods. Consider a 32 bit logical address space with a page size of $4KB = 2^{12}$. The page table would have over one million entries since $2^{32}/2^{12} = 2^{20}$. If each entry is 4 bytes, then the page table would take $4MB$ of memory. For 64 bit systems, page tables can be in the petabyte range. There are some solutions to solve these size issues:

- **Hierarchical Paging**: This relies on the observation that most programs do not use all virtual address spaces at the same time, as only small parts of page tables are used at any given time. The goal is to increase page table (PT) utilization and reduce PT size. To do this, we break up the logical address space into multiple page tables. A simple technique to accomplish this is a two-level page table, where we page the page table.

  Example 11.2 (Two-Level Page Table)

  A logical address (on a 32-bit machine with 4000 page size) is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Since the page table is paged, the page number is further divided into a 10-bit page number and a 100 bit page offset. Thus, a logical address is

  $$(p_1, p_2, d),$$

  where p_1 is an index into the outer page table, and $p2$ is the displacement within the page of the inner page table. p_1 and p_2 comprise the page number, while d makes up the page offset. This is known as a **forward-mapped page table**. The outer page table has entries that are mapped to entries in another page table. This page table has pages that are page tables. The various values inside this inner page table map from the first outer page table, and maps to different locations in memory.

  The address translation scheme is achieved using three memory accesses per request. p_1 specifies the outer page table entry, which then directs us to an inner page table. p_2 then specifies the offset to retrieve the entry, which leads to the frame. d then specifies the entry in the frame we are to access. In this case, the outer page table would contain 2^{42} entries. We could add a second outer page, thus requiring 2^{32} entries, with 4 memory accesses per request. The size of an outer or inner page table is equal to 2^n where n is the number of bits specifying that outer or inner page table.

  Example 11.3 (Multi-Level Page Table)

  Consider a 64-bit system, with a $4KB$ page size (2^{12} bits), with 4 bytes per entry. A single page table would contain 2^{52} entries. Multiplied by $4B$ per entry, this results in 2^{54} bytes, or around 18 petabytes.

  A page of $4KB$ can fit $4KB/4B = 1024 = 2^{10}$ entries. We have 12 bit offsets, so we need the ceiling of 52 bits divided by 10 bits per level. This results in six levels. With 6-level hierarchical PT, each memory request would require 7 memory accesses, since 6 accesses are required for translating the logical address to the physical address, and 1 access is required for the actual memory location.

- **Inverted Page Tables**: Rather than each process having a page table keeping track of all possible logical pages, we can instead track all physical pages in one shared inverted page table (IPT). IPT has one entry for each real page of memory, containing a virtual address and an owning process ID. IPT decreases the memory needed to store each page table, since the IPT size is proportional to the amount of physical memory available. However, IPT increases the time needed to search

the table when a page reference occurs. TLB can help accelerate this lookup. For this reason, shared pages are problematic.

> **Example 11.4**
> With $16GB$ memory and a $4KB$ page size with $4B$ per entry, this would result in a $16MB$ page table. This page table has around 4 million entries. On average, a translation would require around 2 million memory accesses.

In the inverted page table architecture, the CPU sends the *pid*, p, and d comprising the logical address. A search in a page table is performed to find *pid* and p together. The index of this is i. i is the combined with d to obtain the physical address to physical memory.

- **Hashed Inverted Page Tables**: This is common in 64-bit systems. The virtual page number and process ID are hashed into a page table. This page table contains a chain of elements hashing to the same location, where each element contains a *pid*, virtual page number, and a pointer to the next element. The virtual page numbers are compared in the chain until a match is found. If a match is found, the corresponding physical frame (the index in the page table) is extracted. This index is combined with the offset to access physical memory. Otherwise, there is a page fault. A good hash function can permit an average access time of $O(1)$.

> **Example 11.5** (Page Table with Some Pages not in Main Memory)
> The page table contains gaps, where some pages are not mapped to a corresponding frame since the valid-invalid bit is set to invalid. The pages with the valid bit set are associated with a corresponding frame in physical memory. When they are not in main memory, they can be in the backing store.

§11.2 Page Fault Handling

A **page fault** is an exception raised when a process accesses a memory page that is not currently mapped by MMU. For instance, an entry in the page table marked invalid. Note that with demand paging, the first reference to a page always results in a page fault. The general page fault handling procedure is as follows:

1. The operating system looks at another table to decide if it is an invalid reference (resulting in an abort), or the reference is valid but the page not in memory (since it is in the backing store).
2. Find a free frame.
3. Load the page from the backing store into the frame via scheduled disk operation.
4. Reset the page tables to indicate the page now in memory.
5. Set validation bit to valid.
6. Restart the instruction that caused the page fault

Example 11.6

When there is an instruction such as `load M`, we consult the page table. In the event that the invalid bit is set, we trap to the operating system. The page is in the backing store, so we bring in the missing page into a free frame in physical memory. This then resets the page table, then restarts the instruction.

We need to be able to deal with **over-allocation of memory** when there are no free frames. We could find some page in memory that is not really in use and page it out by saving it to the backing store. However, we need an algorithm to find a victim page. Since we are concerned with performance, we want an algorithm that will result in the minimum number of page faults. We need a **page replacement** algorithm that can use the **modify (dirty) bit** in a page table entry to reduce the overhead of page transfers, so that only modified pages are saved to the backing store. The dirty bit is automatically set by hardware on write access. Basic page replacement is as follows:

1. Find the location of the desired page on disk.
2. Find a free frame:
   - If there is a free frame, go to the third step.
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**.
   - If the victim frame is dirty, write it to the backing store.
   - Set the invalid bit in the page table pointing to the victim frame.
3. Load the desired page into the new free frame and update the page and frame tables. The frame table is a simple data structure that keeps track of the free frames.
4. Continue the process by restarting the instruction that caused the trap.

Remark 11.7. There are now potentially two page transfers for a page fault, further increasing the effective access time (EAT).

Example 11.8 (Basic Page Replacement)

We first swap out the victim page associated with an index f in physical memory into the backing store. The page table entries indicate the frame and whether it is valid or invalid (through the valid-invalid bit). We change it to invalid since we have swapped the victim page out. We now swap the desired page in, and reset the page table for the new page.

§11.3 Frame Allocation Algorithms

A **frame allocation** algorithm determines the amount of frames to give to each process. For a single-process system, the OS claims some frames and leaves the rest to the running process. For a multiprogramming system, each process needs a **minimum** number of frames (OS/architecture dependent). The **maximum** depends on the available physical memory. There are two major allocation schemes:

1. **Fixed Allocation**: This includes equal allocation and proportional allocation. **Equal allocation** means that after the allocation of frames to the OS, each remaining process obtains an equal number of frames. **Proportional allocation** allocates according to the size of the process. This is dynamic, as the degree of multiprogramming and process sizes change.

   Example 11.9 (Proportional Allocation)
   Let s_i be the size of process p_i, $S = \sum s_i$, m be the total number of frames, and a_i be the allocation for p_i, where
   $$a_i = \frac{s_i}{S} \cdot m.$$
   In a specific example, we have $m = 62$ frames, $s_1 = 10$, and $s_2 = 127$. The allocation for these processes would therefore be $a_1 = \frac{10}{137} \cdot 62 \approx 4$ and $a_2 = \frac{127}{137} \cdot 62 \approx 57$.

2. **Priority Allocation**: This uses a proportional allocation scheme using priorities rather than size. If a process generates a page fault, it selects for replacement one of its own frames, or selects for replacement a frame from a process with lower priority.

Allocation may also be distinguished as global or local. **Global replacement** occurs when a process selects a replacement frame from the set of all frames. One process can take a frame from another. Process execution time can vary greatly, but there is greater throughput, so this is more common. **Local replacement** occurs when each process selects only from its own set of allocated frames. This results in more consistent per-process performance, but possibly underutilizes memory.

§11.4 Page Replacement Algorithms

A **page replacement** algorithm determines the victim frame, and wants to maintain the lowest possible page-fault rate on both first access and re-access. We evaluate the algorithm by running it on a particular string of memory references (**reference string**) and computing the number of page faults on that string. A string is just a list of page numbers, not full addresses. Repeated access to the same page does not cause a page fault. The results depend on the number of frames available. Generally, there is an inverse relationship, as the number of page faults decrease as the number of frames increase. In the following examples, our reference string will be $7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$

- **First In First Out (FIFO) Algorithm**: Replace the page that has been in memory for the longest time. This could be implemented using a FIFO queue.

Example 11.10

The reference string is $7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$. With three frames (3 pages can be in memory at a time per process), we obtain 15 page faults. This is because we first fill with 7, then 0, then 1. Then, we replace the 7 with 2. When we reach 0, this matches with one of the three frames already present, so we do not have a page fault. On 3 however, we replace the original 0 with 3, so there is a page fault. The number of page faults varies, and depends on both the reference string and the number of frames available. For instance, using the same reference string but with four frames, we obtain 10 page faults.

Bélády's Anomaly occurs when increasing the number of page frames results in an increase in the number of page faults for certain memory access patterns. For instance, the reference string $0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4$ using FIFO with 3 frames results in 9 page faults, while using FIFO with 4 frames results in 10 page faults.

- **Optimal (OPT) Algorithm**: Replace the page that will not be used for the longest period of time. This is not practical as it requires knowing the future. However, it is useful for measuring how well other non-optimal algorithms perform.

Example 11.11

With the reference previous reference frame and three frames, we first have 7, then 0, and 1 appear. Since 7 will not be used soon, it is replace with 2. When 3 appears, it replace 1. When 4 appears, it replaces 0. When 0 appears, it replaces 4 since there are no more occurrences of 4. When 1 appears, it replaces 3 since there are no more occurrences of 3. When 7 appears, it replace 2 since there are no more occurrences of 2. This results in 9 page faults.

- **Least Recently Used (LRU) Algorithm**: Replace the page that has not been used in the most amount of time. This uses past knowledge instead of future knowledge, as it associates the time of last use with each page. This is generally a good algorithm, and is thus frequently used. LRU and OPT are cases of stack algorithms that do not exhibit Bélády's Anomaly.

Example 11.12

Using the same reference string with three frames, 7 appears, followed by 0 and 1. 7 is replaced by 2, then 1 is replaced by 3 (0 is not replaced since it was recently used). When 4 arrives, this replaces 2 since 0 was just used again. 2 then replaces 3, 3 replaces 0, and 0 replaces 4. 1 then replaces 0 since 3 and 2 were recently used. 0 then replaces 3 and 7 replaces 2 since 1 was recently used. This results in 12 page faults. This was better than FIFO but not better than OPT.

LRU can be implemented using a counter or a stack:

1. With the counter implementation, every page entry has a counter. Every time a page is referenced through this entry, copy the current clock into the counter. When a page needs to be changed, look at the counters to find smallest value. A search through the table is needed.

2. With the stack implementation, we keep a stack of page numbers (for instance, in a doubly linked list). When a page is referenced, we move it to the bottom. The stack top contains the least recently used page. Each update is more expensive than with a counter. However, no search is needed for replacement.

- **Clock Replacement Algorithm**: Pure LRU needs special hardware and is still slow. Clock replacement is an approximation of LRU. Frames are organized as a circular buffer. One pointer (clock hand) is maintained, and points to the page to be replaced next. If the page has a reference bit equal to 0, we replace it. Otherwise, we set it to 0 and advance the pointer to the next page. This gives a page a second chance. This algorithm is very simple and provides good performance. There are many more page replacement algorithms.

§11.5 Thrashing

If a process does not have enough pages, the page-fault rate is very high. There is a page fault to get the page, where we replace an existing frame. But, we quickly need the replaced frame back. A **thrashing process** results when a process is progressing slowly due to frequent page swaps. This can lead to an entire system thrashing. Many processes thrashing results in low CPU utilization. The OS believes that it needs to increase the degree of multiprogramming, so it adds another process to the system, thus making things even worse.

Example 11.13

Visually, this can be seen by plotting CPU utilization on the y axis with degree of multiprogramming on the x axis. CPU utilization increases as the degree of multiprogramming increases, until it reaches a certain threshold. With increased multiprogramming, CPU utilization actually decreases. This region is where thrashing occurs.

There are different ways to deal with thrashing:

- **Local Page Replacement**: When a process is thrashing, the OS prevents it from stealing frames from other processes. At least the thrashing process cannot cause the entire system to thrash.
- **Working Set Model**: The OS keeps track of pages that are actively used by a process (**working set**). The working set of processes changes over time. Therefore, the OS periodically updates the working set for each process, using a moving time window. Before resuming a process, the OS loads the working set of the process.
- **Page Fault Frequency**: We establish acceptable bounds on the page fault rate. If the actual page fault rate of a process too high, the process gains a frame. If the actual page fault rate of a process is too low, the process loses a frame.

Example 11.14

Recall that there is an inverse relationship, as the number of page faults decrease as the number of frames increase. We now set a lower and upper bound, where we increase or decrease the number of frames respectively should a process lie outside of that range.

§11.6 Copy-on-Write

Copy-on-Write (COW) allows parent and child processes to initially share pages in memory. If either process modifies a shared page, only then is the page copied. The page table entries need a **copy-on-write bit**. COW allows more efficient process creation as only modified pages are copied.

Example 11.15

There are two processes, P_1 and P_2. Before P_1 tries to modify page C, P_1 and P_2 share page A, B, and C in physical memory. After P_1 tries to modify C, the only change is that now P_1 points to a copy of page C located elsewhere in physical memory instead of C.

§11.7 Disk Structure

Each disk **platter** has a flat circular shape, with a diameter between 1.8" and 5.25". These platters rotate on a spindle. A **cylinder** is any set of all of tracks of equal diameter in a hard disk drive. The surface of a platter is logically divided into circular **tracks**. A track is a ring of a certain diameter on a platter, while a **sector** is a segment of that track. A couple of adjacent sectors form a **cluster**. The **read-write head** flies just above the surface of each platter. A head crash occurs when the head makes contact with the disk surface, causing permanent damage to the disk. Each head is attached to a disk arm that moves all heads at the same time. These arms are attached to an arm assembly.

§11.8 Disk Management

A **logical block** is the smallest unit of transfer between the disk and the memory (for instance, 512 bytes). The sectors on the disk are mapped to large one-dimensional arrays of logical blocks, numbered consecutively. **Mapping** is the process of converting a logical block number into a physical disk address that consists of a cylinder number, a head number, and a sector number. On modern disks, this is done by an embedded controller because geometry is more complicated.

Low-level format or **physical format** writes low level information to the disk, dividing it into a series of tracks, each containing some number of sectors, with short gaps between the sectors. The formatted capacity is about 20% lower than the unformatted capacity. There are numerous components in a disk sector:

- **Preamble**: The sector starts with a special bit sequence, cylinder number, sector number, etc.
- **Data**: This depends on the format (for instance, 512 bytes).
- **Error Correction Code (ECC)**: This contains redundant information to be used for correcting read errors.

To use a disk to hold files, the operating system still needs to record its own data structures onto the disk. It **partitions** the disk into one or more groups of cylinders, each treated as a logical disk. **Logical formatting** then follows, where a filesystem is made. This involves abstracting blocks into files and directories. The OS can allow raw disk access for applications that want to do their own block management. Databases for instance, want to keep the OS out of the way. The **boot block** is used to initialize the system. The bootstrap is stored in ROM, and the bootstrap loader program is stored in

boot blocks of the boot partition. Methods such as **sector sparing** are used to handle bad blocks, where a spare sector per track is used if another sector becomes defective.

§11.9 Disk Scheduling

The time required for reading or writing a disk block is determined by three factors:

1. **Seek time**: The time to move the arm to the correct cylinder.
2. **Rotational delay**: The time for the correct sector to rotate under the head.
3. **Disk bandwidth** : The actual data transfer rate. This is calculated for a set of requests, and is given by the total number of bytes transferred divided by the total time taken to service all requests.

For a multiprogramming system with many processes, the requests for disk I/O are appended to the disk queue. The OS maintains separate queues of requests for each disk. We can improve the overall I/O performance by managing the order in which disk I/O requests are served by scheduling the requests in the queue based on time required to move heads:

- **First Come First Serve (FCFS) Scheduling** is intrinsically fair, but it generally does not provide fastest service.

  Example 11.16

  The queue contains 98, 183, 37, 122, 14, 124, 65, and 67. The head starts at 53, and has to move to 98, then 183, followed by the rest of the entries in the queue. This results in 640 cylinder moves.

- **Shortest Seek Time First (SSTF) Scheduling** selects the request with the least seek time from the current head position. Seek time is equal to the time to move the heads.

  Example 11.17

  Using the same queue as before with the head starting at 53, we now traverse 65 first, then 67, 37, 14, 98, 122, 124, and 183. This is similar to the Shortest Job First algorithm in that this may cause starvation of some requests. In this example, there are 236 cylinder moves.

- **SCAN (Elevator) Scheduling** occurs when the head continuously scans back and forth across the disk and serves the requests as it reaches each cylinder.

  Example 11.18

  using the previous example and starting at 53, we now traverse 37 first, then 14, before reaching 0. Now, we proceed in the opposite direction to reach 65, 67, 98, 122, 124, and finally 183. This results in 208 cylinder moves. Requests at the other end may wait the longest.

- **C-SCAN Scheduling** is the same as SCAN in one direction, but when it reaches the last cylinder, the head returns to the first cylinder.

Example 11.19

The head starts at 53, reaches 65, then 67, 98, 122, 124, 183, and then 199, which is the end. It then moves back to 0 before reaching 14, then 37. This provides a more uniform wait time compared to SCAN.

- **C-LOOK Scheduling** is a small optimization of C-SCAN, as the head only goes as far as needed by the next request. Using a similar optimization for SCAN, we arrive at **LOOK Scheduling**.

Example 11.20

Starting at 53, it reaches 65, then 67, 98, 122, 124, 183, 14, and then 37.

The performance of a scheduling algorithm depends on the number and types of requests, the file allocation method, and the location of directories and index blocks. Either SSTF or LOOK are reasonable choices for the default algorithm. C-LOOK can be used if we need more consistent wait times. Other scheduling algorithms also consider rotational latency, and the priority of the task. For instance, demand paging should receive higher priority.

§11.10 Redundant Array of Inexpensive Disks Structure

Redundant Array of Inexpensive Disks (RAID) employs multiple disk drives to provide reliability via redundancy, increasing the mean time to failure. This can also improve performance through parallelization of requests. It is accessed as one big disk. There are different types of RAID organizations:

- **Striping (RAID 0)** uses a group of disks as one unit for performance. There is no redundancy.
- **Mirroring (RAID 1)** keeps duplicates of each disk.
- **Striped Mirrors (RAID 1+0)** provides high performance and high reliability.
- **Block Interleaved Parity (RAID 4, 5, 6)** is similar to RAID 1+0 but with less redundancy.

A small number of **hot-spare** disks can be left unallocated. These automatically replace a failed disk and have data rebuilt onto them.

Example 11.21 (RAID Levels)

RAID 0 is non-redundant striping, RAID 1 is mirrored disks, RAID 2 is memory-style error-correcting codes, RAID 3 is bit-interleaved parity, RAID 4 is block-interleaved parity, RAID 5 is block-interleaved distributed parity, and RAID 6 is P + Q redundancy.

§11.11 I/O Hardware

I/O devices fall under the following primary categories:

- **Block Devices**: These devices store information in fixed-size blocks such as 512 bytes to $32KB$. Each block has its own address. Data is transferred in units of one or more entire blocks. Read or write can be done independently of each other. Examples include hard disks, CD-ROM, and USB.
- **Character Devices**: These devices deliver or accept a stream of characters, without regard for any block structure. They are not addressable, and provide no seek operations. Examples include printers, network interfaces, keyboards, and mouses.
- **Other Devices**: This includes clocks or timers.

§12 June 22, 2017

§12.1 Filesystems

To achieve long term storage, it is essential that it must be possible to store a very large amount of information, the information must survive the termination of the process using it, and multiple processes must be able to access this information concurrently. A disk can be thought of as a linear sequence of fix-sized blocks that support the two operations of reading block j, and writing to block j. This is very similar to memory, but it is **persistent**, and **much slower**.

§12.2 File Structure

A **file** is an abstraction of long term storage, implemented by the OS. Anything can be stored in a file, as long as it can be organized into a sequence of bytes. For instance, this may be source code, executables, images, movies, text, etc. From the perspective of the OS, the process sees file through contiguous logical and virtual address spaces. A file contains a sequence of bytes, which can be individually addressed. The OS simply maps these files onto physical devices, as it generally does not care about the contents of the files. The **file creator** decides on the contents of the file, including the **file format** and the **internal structure**. The file creator also decides the meaning of the file's contents. This can create an even higher level abstraction. For instance, we can treat a file as a sequence of bits, numbers, records, etc.

A byte sequence is the most common file structure, but other structures are also possible. However, these other structures can be emulated by byte sequences.

Example 12.1

A byte sequence contains a sequence of bytes. A record sequence contains a sequence of records, where a record contains more information than a single byte. We can also have tree structures.

§12.3 File Naming

Files have textual names that are given to the files at creation time, but these can be changed later. There are different file-naming rules on different systems. For instance, there may be constraints on maximum filename lengths (at least 8), allowed characters, capitalization, or file extensions. File extensions may be enforced, or follow conventions. They may also be considered separate from the filename, or considered a part of the filename.

§12.4 File Formats

Many systems support **special file types** on top of regular files and directories:

- **Regular files** are text or binary.
- **Directories** are special files for maintaining FS structure.
- **Character special files** are for I/O on character devices. For instance, `/dev/random`.
- **Block special files** are for I/O on block devices. For instance, `/dev/sdb0`.

Regular files can have custom types as well (**file format** or **file type**). These are determined by the file creator. If an OS recognizes the file format, it can operate on the file in reasonable ways. For example, it could use an appropriate program to open the file.

Example 12.2

Windows uses file extension to determine the file format. Running `file` in a bash script followed by a list of files results in a description of each file. On the other hand, UNIX uses **magic number** techniques to determine the file format. Thus, the extension is only a convention. In particular, the format is inferred by inspecting the first few bytes of a file. For example, `#!/bin/bash`.

Example 12.3

An executable file contains in its header the magic number, text size, data size, BSS size, symbol table size, entry point. Flags reside at the bottom of the header, after some other information is stored after the entry point. This is then followed by text, data, relocation bits, and the symbol table. An archive contains a header, followed by an object module. This alternates as we see a repeating pattern of headers and object modules in an archive. The header contains the module name, data, owner, protection, and size.

§12.5 File Attributes

File attributes: These vary from one OS to another, but typically consist of the following:

- **Name**: The symbolic file name is the only information kept in human readable form.
- **Identifier**: A unique tag that identifies the file within the FS.
- **Type**: This is needed for systems that support different file types, such as block devices.
- **Location**: A pointer to the location of the file on a device.
- **Size**: The current size of the file.
- **Tme and Date**: The time of creation, last modification, or last access. This is used for usage monitoring.

- **Protection**: Access control information such as read, write, and execute.
- **User ID**: The owner or owners.

There are many variations of file attributes, including extended file attributes such as file checksum. This information is often kept in the directory structure. File attributes are not stored in the file, since they are stored elsewhere. Thus, even an empty file takes up space, as we need to store the file attributes.

§12.6 File Operations

There are different operations that can be performed on files. Most systems allow the following operations on files:

- **Create**: A file is created with no data.
- **Delete***: When the file is no longer needed, it has to be deleted to free up disk space,
- **Open**: Before using a file, a process must open it. The OS can fetch attributes and a list of disk addresses into main memory for rapid access on later calls.
- **Close**: Free up space in memory by flushing unwritten data from memory to disk.
- **Read**: Read data from the current position.
- **Write**: Write data at the current position.
- **Append**: This is a special type of write on some systems.
- **Seek**: Change the current position.
- **Get attributes***: For instance, obtaining the size of a file.
- **Set attributes***: For instance, setting permissions.
- **Rename***: Change the filename.

Remark 12.4. The starred operations do not operate on the files themselves. Instead, it operates on that file's metadata.

§12.7 Open Files

The OS needs to manage open files. To do this, it keeps several data structures in memory. An **open-file table** tracks open files, per-process and system-wide. The open-file table contains he **file pointer** is a pointer to the last read and write location, and is per process that has the file open. The open-file table also contains a **file-open count** is a counter of the number of times a file has been opened. This is used to allow removal of data from an open-file table when the last processes closes it. This is system wide. The open-file table also contains the permissions, with a pointer to file contents. This is also system wide.

Example 12.5

In the user space, we call `read(index)`. This index is sent from the user space into kernel memory, where it first passes the per-process open-file table before reaching the system-wide open-file table. It is then sent out of kernel memory into the data blocks in secondary storage. Alternatively, a file-control block from secondary storage may be sent into the system-wide open-file table in kernel memory.

§12.8 File Access

There are generally two types of file access. This applies to reading and writing:

1. **Sequential Access**: This is the most common. A file is accessed byte-by-byte from beginning to end. There is no skipping, and no out-of-order access. Sequential access can be rewound. For example, `open, read, read, read, rewind, read, ...`.
2. **Random Access**: This can access any byte in any order, and is usually implemented using a `seek(position)` call. For instance, `open, read, read, read, seek, read, read, seek, read, ...`

§12.9 File Locking

File locking is provided by some operating systems and file systems. These are similar to reader-writer locks. A **shared lock** is similar to a reader lock, as several processes can acquire concurrently. An **exclusive lock** is similar to a writer lock. File locking mediates access to a file during `open()`. **Mandatory file locking** occurs when access is denied depending on the locks held and requested. **Advisory file locking** occurs when processes can find the status of locks and decide what to do.

§12.10 Directories

A a filesystem is a **collection of files**, where a file is the basic unit in a filesystem. Files are organized in a **directory structure**, which is usually a tree with one or more levels. This is implemented through **directories** which organizes files and store **metadata** about those files (such as file names). A filesystem must be **mounted** before it can be accessed.

A directory is implemented as a special file. The directory file contains entries, where each entry can be a file or directory (**subdirectory**). If subdirectories are not allowed, then we have a **single-level directory** system. This has limited uses, such as for cameras. If subdirectories are allowed , then we have a **hierarchical directory system**. These see widespread use.

Example 12.6

A single-level directory consists of a root directory that branches into various files. A hierarchical directory system contains a root directory, that branches into multiple user directories. These user directories may branch further into user subdirectories, or into user files.

§12.11 Path Names

A path name generally includes a path separator `/` (forward slash). File paths are of the form

$$dir_1/dir_2/.../dir_n/filename.$$

The root directory is denoted with `/`. An **absolute path name** begins at the root. Every process has a **working (current) directory**. A **relative path name** defines a path from the current directory.

Example 12.7

An absolute path name is `/usr/jim`, while a relate path name is `./banker` or `../..bin/cat`.

Every directory starts with one of two entries. This may be a pointer to the current directory with `.` (**dot**), or a pointer to the parent directory with `..` (**dotdot**). The pointer to the current or parent directory with dot and dotdot cannot be deleted. They are present even in empty directories, as they are just pointers.

Example 12.8

It is possible for different path names to refer to the same path. For instance, `/usr/jim` and `/./etc/../lib/./../usr/pavol/../jim ../../../usr/jim` may refer to the same file.

§12.12 Directory Operations in UNIX

There are different directory operations that can be performed:

- **Create**: An empty directory is created with `.` and `..` entries.
- **Delete**: Only an empty directory can be deleted. It is the case that `.` and `..` are a part of the empty directory.
- **Opendir**: This is analogous to open for files.
- **Closedir**: This is analogous to close for files.
- **Readdir**: Returns the next entry in an open directory
- **Rename**: This is just like file rename.
- **Link**: This is a technique that allows a file to appear in more than one directory
- **Unlink**: A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. In UNIX, the system call for deleting files (discussed earlier) is, in fact, `unlink`.

§12.13 Implementation of Filesystems

We make use of several in-memory structures. These structures are used for both FS management and performance improvement:

- **System-wide open-file table**: There is an entry for each open file. For example, the starting block number.
- **Per-process open-file table**: This provides pointers to system-wide open-file table + file pointer,...
- **Mount table**: This includes information about each mounted volume.
- **Directory-structure cache**: This contains the directory information of recently accessed directories.
- **Buffers**: These are file-system blocks when they are being read from or written to a disk.

Example 12.9 (Typical Filesystem Layout)

The following example occurs over the entirety of the disk. The first part is the **master boot record** (MBR). This is followed by the **partition table** that indicates the start and end of each partition. The remainder of the disk is several disk partitions. A disk partition consists of a boot block (bootstrap program), followed by a superblock (FS parameters such as type), free space management (free blocks), I-nodes (per file or directory information detailing the blocks that belong to each file), the root directory (entries in `/`), and finally the files and directories (where the contents of the directory are files and subdirectories).

Nearly all filesystems split files up into fix-sized blocks. The file size is rounded up to the nearest multiple. Most filesystems suffer from internal fragmentation. The **filesystem block** size is usually a multiple ($2n$) of the underlying **disk block** size (**clusters**). The FS blocks of one file are not necessarily adjacent, as it may be a **fragmented file**. In this case, there could be seek time performance issues. Performance and space utilization are inherently in conflict. That is, utilization decreases as performance increases.

Partitions can contain a contain a filesystem. Alternatively, a partition can be raw when it is just a sequence of blocks with no file system. The boot block can point to a boot volume or a boot loader, which is a set of blocks that contain enough code to know how to load the kernel from the file system The boot block can also point to a boot management program for multi-OS booting. The **root partition** with a filesystem contains the OS, which is mounted at boot time as the root directory '/'. The other partitions can hold other operating systems, other file systems, or be raw. These other partitions can be mounted automatically during boot, or manually after the boot is done. At mount time, file system consistency is checked. If is it not consistent, it is fixed and mounting is attempted again. If all metadata is correct, then we add it to the mount table and allow access.

§12.14 Virtual File Systems

Virtual File Systems (VFS) on Unix provide an object-oriented way of implementing file systems. VFS allows the same system call interface (the API) to be used for different types of file systems. It separates file-system generic operations from implementation details. Implementation can be one of many file systems types, or even network file systems. VFS dispatches operation to appropriate file system implementation routines.

Example 12.10

The API is to the VFS interface, rather than any specific type of filesystem. Thus, we can represent this graphically in a tree structure. Te file-system interface points to the VFS interface. This points to local file system type 1, local file system type 2, and remote file system type 1. These point to disk, disk, and network respectively.

In Linux, there are four object types. These are the inode, file, superblock, and dentry. VFS defines the set of operations on the objects that must be implemented. Every object has a pointer to a function table. The function table has addresses of routines to implement that function on that object. For instance `int open(...)` opens a file, `int close(...)` closes an opened file, `ssize_t read(...)` reads from a file, `ssize t write(...)` writes to a file, and `int mmap(...)` memory maps a file. A developer of a new FS needs to implement VFS API. Only then can the FS be mounted by Linux.

Directories can be implemented in different ways:

- **Linear Lists**: It could be implemented as a linked list of filenames with pointers to the file blocks. This is simple to program, but time consuming to execute. This offers linear search time. It could maintain order alphabetically via linked lists or B+ trees.
- **Hash Tables** : It could also be implemented as a linear list with a hash data structure. This decreases directory search time. However, there may be collisions where two filenames hash to the same location. This method is only good if entries are fix sized, or a chained overflow method is used.

Linux (ext2) uses linked list and hash table with balanced trees.

§12.15 Allocation Methods

An allocation method refers to how disk blocks are allocated for files:

- **Contiguous Allocation**: Each file occupies a set of contiguous blocks. This offers best performance in most cases. It is also simple, as only the starting location (block number) and length (number of blocks) are required. However, we may encounter problems such as finding space for files, knowing the file size at creation time, external fragmentation after file deletion, and the need for **compaction** offline (downtime) or online (reduced performance). This is also known as **defragmentation**. Contiguous allocation is useful for tapes and read only devices such as CD-ROMs.

  Example 12.11

  To map from logical to physical address, the logical address is split into q upper bits and r lower bits. The physical address *start* is computed as q added to the starting address. The displacement into the block is equal to r. Different files start at different locations, and have a fixed length associated with them. They occupy contiguous locations in memory.

- **Linked Allocation**: Each file is stored as a linked list of blocks. Each block contains a pointer to the next block, where files end at NULL pointers. There is no external fragmentation, so no compaction is needed. However, separate free space management needed to maintain a linked list of free blocks. Reliability can be a

problem, since we may lose a block due to disk failure. A major problem is that locating a block can take many I/O and disk seeks. Logical address to physical address mapping requires traversing the list. We could cache the next pointers, but we would need to read the entire file first. We could improve efficiency by clustering blocks into groups, but that increases internal fragmentation.

Example 12.12

In linked allocation, a start and end are indicated for a file. The block consists of a pointer, and data. Files may contain instructions that exist at different locations in memory. The final location is indicated with -1.

- **File Allocation Tables (FAT)**: This is a variation of linked allocation. The beginning of an FS volume has a table, indexed by block number. The table contains all pointers, one for each block. There is one FAT for the entire disk, and directory entries contain indices into the FAT. This is much like a linked list, but faster on disk and is cacheable. Additionally, new block allocation is simple. FAT provides easier random access, but the entire table must be in memory at all times to perform random access.

Example 12.13

The directory entry consists of a file name, some information, and a start block. This start block points to an entry in the FAT. This entry then points to another entry, and so on.

- **Indexed Allocation (I-Nodes)**: Each file has its own index blocks of pointers to its data blocks. This block is called an i-node. It is similar to FAT, but per file rather than per entire FS. An i-node block contains pointers to blocks belonging to the file, and various file attributes such as the file size in bytes, device ID, owner, permissions, timestamps, and link count. Notice that it does not store the filename, since the directory entry is used to associate the filename with the i-node. A **dentry** consists of the filename along with a pointer to the i-node. Indexed allocation easily links files, as different filenames can point to the same file (**hard link**). I-nodes are advantageous since random access is reasonable (we only need to keep the i-nodes for opened files in memory), file size is generally not limited, and files can have holes. However, at least one additional block is required for each file to point to everything else.

Example 12.14 (Indexed Allocation)

A directory entry consists of a filename along with an index block. For instance, in block 19, there is an i-node that points to 9, 16, 1, 10, and 25. These are the locations of blocks belonging to the file.

Example 12.15 (I-Nodes in Linux (ext2))

Consider a block size of $1KB$, with block addresses of $4B$. A single i-node with 12 direct entries has a max file size of $12KB$. Adding a single indirect $1KB$ block can have $1KB/4B = 256$ entries. The max file size is $256KB+12KB = 268KB$. Adding double indirect or 256 blocks each with 256 addresses results in a max file size of around 216 blocks, which is approximately $64MB$. Adding triple indirect results in a max file size of around 224 blocks, which is approximately $16GB$. The ext3 max file size is around 2 terabytes, while the ext4 max file size is around 16 terabytes (using 48 bit addresses).

Example 12.16 (Hard and Soft Links)

To hard link `file1.txt` to `file.txt`, we use `ln file.txt file1.txt`. The hard link points to the same inode as `file.txt`. If we delete the original file, then `file1.txt` will still work. We can create a soft link with `ln -s file.txt file2.txt`. This soft link points to the same filename of `file.txt`. This, if we delete the original, `file2.txt` will be broken. Thus, a hard link points to the i-node, while a soft link points to the filename. Hard links can be created only on regular files. Thus, we cannot hard link directories, as they could lead to cycles in the FS. Symbolic links can link anything, including directories, special files and other symbolic links.

§12.16 Free Space Management

File systems maintain free space lists to track available blocks:

- **Bit Vector (Bitmap)**: A bit is set to 1 if it is free, and is set to 0 if it is occupied. The block number is calculated as the number of bits per word multiplied by the number of 0 valued words, plus the offset of the first 1 bit. The CPU has instructions to return the offset within word of the first 1 bit. Bitmaps require extra space. It is easy to get contiguous files.

  Example 12.17

  A block size of $4KB = 2^{12}$ bytes and a disk size of 2^{40} bytes (1 terabyte) has a disk size of $2^{40}/2^{12} = 2^{28}$ blocks. Since 1 bit is needed per block, this means that we need 2^{28} bits of 2^{20} bytes to store the bitmap. This is around $1MB$. If we are using clusters of 4 blocks each, this results in a $256KB$ table.

- **Linked Free Space List (Free List)**: There is no waste of space. However, we cannot get contiguous space easily. There is no need to traverse the entire list so long as the number of free blocks is recorded. The free space list head simply points to the location of the next free space.

There are different free space management techniques:

- **Grouping**: Modify linked lists to store the addresses of the next $n-1$ free blocks in the first free block, plus a pointer to the next block that contains free block pointers (like this one).

- **Counting**: Because space is frequently contiguously used and freed, we keep the address of the first free block and the count of the following free blocks. A free space list then has entries containing addresses and counts.
- **Space Maps**: Divide the device space into metaslab units, each representing a chunk of manageable size. Within each metaslab, a counting algorithm is used to keep track of free space.

§12.17 Performance

Newer CPU (2016) can do about 300000 million instructions per second. Typical disk drives at 7200RPM can do around 100 input/output operations per second. $300000 MIPS/100 =$ 3 billion instructions during one disk I/O. Fast SSD drives provide about $100000 IOPS$, so this results in $300000 MIPS/100000 = 3$ millions instructions during one disk I/O. Expensive SSD arrays can deliver 10000000 IOPS. This is still around 30000 instructions during one disk I/O. It is important to try to minimize the number of I/O operations. This can be done by trying to group read/write. Bytes per second is calculated as the IOPS multiplied by the transfer size.