Operating Systems

Jacob Denson

December 18, 2015

Table Of Contents

Chapter 1

Introduction

Managing the hardware of a computer is a fragile task, not for the faint of heart. In computer science, we desire abstraction and modulization, to simplify tasks and reduce the load on our brain, so we are able to attack more complicated and interesting issues. An **operating system** is a program which enables this abstraction. It manages a computer's hardware, acting as an intermediary between the core resources of the computer and higher level programs, enabling a simpler interafce for the computer. Specifically resources are managed in the **time** a process takes to use a resource, and the **space** of a resource, management of units of the same resource. It is up to the operating system to enable **CPU Scheduling**, **Memory Management**, and **Communication**.

Software can be grouped into two categories. **Systems Software** is the software operating on computer hardware, explicitly managing the physical resources of the computer. If your software requires detailed knowledge of the specific system you are using (memory endian, etc), it is probably systems software. Of course, all operating system software is systems software, but UIs and Utility software (like a virus scanner) control computer hardware but are not often identified as part of the operating system. Systems software provides a platform for running user-opearted software, known as **Applications Software**. This software is used concientiously by the user, unlike systems software, which definitely aids the user in using the computer, but runs in the background. To further divide terms, the operating system can be broken up into the **kernel**, which are the core, unremovable components of a computer, and the **transient** components which are still important but come and go as the computer is running.

At the CPU level, the computer just follows instructions one by one, as they appear in memory. How then, does an OS control the computer, if it must give CPU control to another set of instructions to run a program? The secret is a precise use of interrupts – events which force the switch from **user mode** to **kernel mode** – where the operating system can do the dirty work of the computer. When the CPU acknowledges the interrupt, it reports to an interrupt vector, normally at the bottom of memory, which tells the CPU where to go for the next instruction. The CPU also saves all the registers to the stack, and then jumps to the kernel code. From then on out, its up to the kernel to do what it wants. The OS begins by saving register entries, then uses the temporary stack to execute its code. If the interrupt is synchronous, it is known as a **trap** (a system call, a page fault, or an error).

Integral to the study of operating systems is an abstraction of computer part manipulation to general principles. **Mechanisms**, those parts of the system which control operation authorization and allocation of resources, should not dictate **policy**, deciding what the OS is allowed to do. A **mechanism** decides how it is done. One can divide the study of operating systems between deciding upon what policies an OS should be enforced by, and the mechanisms by which these policies are executed.

Chapter 2

Process Management

The atomic components of a computer in motion are **processes**, which can be understood as a program in execution. A process is not a static object, in the sense that one can 'store', 'retrieve', or 'edit' a process in memory. A program is a description of an algorithm, whereas while a process does contain such a description (part of its **text section**), it also contains all other junk that is gathered when running a program: for instance, the status of the process as it is executing (the program counter, process number etc.), collectively called the **process control block**, the **data section**, and **address space**.

It is up to the processer to manage how the processes operate on the computer. In the past, systems used a **Uniprogramming** scheme, where only one program was ever on the go at any point in time. **Multiprogramming** is the modern approach, where multiple processes can be ran at any point in time – this is what enables you to read the PDF copy of this book, while playing music, while an update to your computer is being downloaded, all at the same time, on the same device. Furthermore, it means processes can run much more efficiently; if a process needs to wait for I/O, or some other event, it doesn't block other processes use the CPU. Back in the day, this approach was unviable due to memory constraint, but the obvious advantages of the multiprogramming approach are now easily obtainable.

Our first concern is how an OS manages processes. Firstly, the OS must know all useful info about a process, so it may intelligently manage them. This information is often kept in a **process table**, an array containing structures for each process. The process control block collects all of the status of a process into one place. That is, it contains that:

1. **State**: If the process is new, ready, running, waiting, halted, etc.

2. **Program Counter**: Address of the next instruction for a process.

3. **CPU registers**: Basic memory slots used to manipulate a program.

4. **Memory-management information**: Additional registers used by the OS, as well as additional memory allocated in RAM.

5. **Accounting Information**: Amount of CPU/real time used, job/process numbers, etc.

6. **I/O status information**: I/O processes allocated to the process, plus access priviledges to other files on the system.

In truth, 'multiprogramming' does not need programs to execute all at one time (though with the rise of multiple core CPUs, this is not impossible) – it needs only keep up this illusion to the user by rapidly cycling between running different processes. The **dispatcher** is that component of the operating system which decides which order processes should execute in. Most of the time, the status of a process follow a simple state transition diagram, cycling between 6 states:

1. New: processes which haven't been through **resource acquisition**.

2. Ready: Processes ready to run, but which are not executing.

3. Active: Processes that are executing at a given moment.

4. Blocked: Processes that cannot execute until a given event occurs.

5. Stopped: A special case of blocked, where a user suspends a process.

6. Exiting: A process about to go through **resource release**.

An operating system implements an array known as a process table to make decisions with multiple processes. Each entry is a process control block. When it needs to, the dispatcher uses one of a selection of strategies to pick a 'Ready' process, and give it control of a CPU.

When a process is running, the dispatcher process has no control over the CPU. The common manner of doing this is to set a timer interrupt (the

alarm clock approach), which will automatically exit out of a process to an **interrupt service routine**, known as the **sleeping beauty** approach. It is up to this routine to save program state, and restore the state to the next process used. Either we can save this to the disk (slow), isolate the memory from the other processes (memory-inefficient), or just trust memory won't be overwritten by other programs (volatile).

In Unix, we create new processes using the *fork()* command, which clones the PCB of an existing process. The process created is the same in all respects (with separate memory for copied variables) except that the return value of the fork command is 0, not 1. The OS uses fork to start running any other process (by using the exec() command, one replaces the program a process is using by another program). In fact, to begin running a computer, a technique known as **bootstrapping** is used, where we initialize a single process, which begins to fork into other process, eventually causing a tidalwave of processes to emerge, 'pulling the computer up by its own bootstraps'. After a parent has forked, it should eventually use the *wait()* command to ensure all its processes have been terminated. Otherwise, a child process may become an **orphan**. Fortunately, Linux has an 'init' process, which adopts all orphans, and then kills them by periodically waiting, ensuring that no processes are left-behind. Nonetheless, their (now defunct) PIDs may still occupy the resource table, a resource leak. If this happens, the process is known as a **zombie** process.

2.1 Threads

A **thread** is like a process, but more lightweight. When you create a thread, only the minimum functional information is copied for both threads to function (i.e. the stack and program counter), but RAM is not copied. A process is an address space with one thread. All stacks for each thread must be in the same address space, since the address is shared. Threads give us the advantage of a lightweight method of enabling concurrency, but the come at the price of having to look after data much more carefully. Real parallelism can only be achieved by having multiple CPU cores. **Psuedo parallelism** use rapid switching of CPUs to simulate the effect of real parallelism. **Hardware parallelism** is where the CPU has multiple I/O devices running in tandem with the CPU, enabling the computer to do a (albeit more restricted) set of instructions at the same time.

We can either implement threads in user space or in kernel space. In the first case, the implementation is OS independant, but requires much more busywork. Processes need to implement their own thread table to keep track of threads specific to the process, and we can't use interrupts to switch between threads. Nonetheless, this enables processes to schedule their own threads efficiently, and multiple threads can be run in a single time **quanta**, the amount of time a process has before it is interrupted by another process. When thread management is implemented in kernel level, you get all the process management for free, but you have to switch contexts every time you want to switch threads.

2.2 Resource Sharing

Processes **cooperate** if they share resources, and **compete** if they fight for individual resources and CPU time. Competing processes cannot influence side effects with one another. Processes each have individual memory locations. They compete, which means that no data between the processes is shared. How, then, are processes able to communicate with one another? When they do, how do they avoid buggering each other up? These issues are encompassed under the topic of **Interprocess Communication**.

Even assuming that data is able to be easily shared between processes, we still have the problem of data confliction - how do we ensure that we don't void the other processes's correctness by changing the data when it is running. **Race conditions** occur when the output of a process depends explicitly on when it is run relative to other processes – a big no no, since processes don't get to pick, and cannot ensure which order they will run in! What we desire is **mutual exclusion** – only one process can access or modify shared data at any one time, regardless of the order processes are run. The code that accesses shared memory is a **critical section**.

One way to ensure mutual exclusion is to disable interrupts. Of course, no other program can edit data, if a process can never be switched out until it is done with the data. This is a simple solution, but for security reasons it is unwise to give user access to interrupt control, and with multiple CPUs, it is impossible to disable interrupts, for we must have a way to manage processes on the other cores.

Another solution is to have shared lock variables that indicate when a process is being used. The problem with this solution is that the act of reading the variable, changing it, and checking if it set, is not atomic. Both processes might read the variable as empty before changing it, causing a race condition.

An alternative method is **strict alternation**, where a certain integer valued 'turn variable' *x* is given. Each process is assigned a unique integer, and checks whether *x* equals its turn, and runs the criticial code if it is. When it is done, it increments *x*, giving access to the criticial section to the next process. The problem of this solution is that we have to wait our turn for an unbounded amount of time – other processes might never run their turn, and thus never increment *x*. If you continually test a variable until it is your turn, this is **busy waiting**. A lock which uses busy waiting is called a **spinlock**.

A good solution to the mutual exclusion problem is to have the qualities of

1. **Mutual Exclusion**: No processes run the code at the same time.

2. **Fair Scheduling**: Every process will eventually be able to run the section. That is, **livelock** will not occur, where some processes are eternally blocked (**starved**) while other processes continue to run.

3. **Fault Tolerence**: Failing programs should not cause other processes to fail.

4. **Lack of Assumption**: Makes no assumptions about program interleaving order and CPU speeds.

5. **Efficiency**: Little blocking and busy waiting should occur.

A **preemptive resource** is a resource that can be taken away from a process and given to another process at any time without error. A **non-preemptive** resource is one we are worrying about now. Once a program begins using the resource, it enters a critical section, and the resource cannot be given away until it is finished.

Dekker's algorithm was the first synchronization algorithm to work property, and works for two processes,

```
while (TRUE) {
    pAtrying = TRUE;
    while (pBtrying) {
```

```
if (turn == B) {
    pAtrying = FALSE;
    while (turn == B);
```

```
            pAtrying = TRUE;
        }

        <CRITICAL SECTION>
    }

    turn = B;
    PAtrying = FALSE;
}

while (TRUE) {
    pBtrying = TRUE;
    while (pAtrying) {
        if (turn == A) {
            pBtrying = FALSE;
            while (turn == A);
            pBtrying = TRUE;
        }

        <CRITICIAL SECTION>
    }

    turn = A;
    pBtrying = FALSE;
}
```

The bakery algorithm solves the problem in full, invented by Leslie Lamport.

```
boolean choosing[n];
int number[n];

while (TRUE) {
    choosing[i] = TRUE;
    number[i] = max(number[0],...,number[n-1]) + 1;
    choosing[i] = FALSE;

    for (j = 0; j < n; j++) {
        while (choosing[j]);
        while (number[j] != 0
               && (number[j], j) < (number[i], i));
    }

    <CRITICAL>

    number[i] = 0;
}
```

These solutions work, but are complicated to implement. Luckily, hardware support has enabled these problems quite trivial to solve. Most modern day processers include an atomic **test and set** instruction, which reads the content of some memory word, called the lock, sends the lock value to a register, and sets the lock to one. Since CPU instructions are indivisible, the value of the lock guarentees whether the value is locked or not.

Using the test and set instruction, we can implement a **semaphore**, a variable that is used as a condition to control access to a shared resource. A **counting semaphore** is integer valued: a positive value indicates the number of shared resources that can be accessed as a certain time, a zero indicates a block, and a negative value indicates the number of processes in the queue to access a variable. This is implemented customarily using two operations. The increment, or V command frees a process, and the P, or decrement operation lcoks the semaphore, or busy waits until its available. A **binary semaphore** takes only two values.

A **monitor** is a high-level abstraction which combines and hides away shared ata, operations on data, and synchronization, so that a programmer does not have to ensure mutual exclusion himself.

The **bounded buffer** problem, or **producer-consumer** problem, is as follows. A producer has to send information down a fixed-size buffer to a consumer of the data. We can solve this using semaphores, as follows:

```
wait(empty);                    wait(filled);
wait(accessOK);                 wait(accessOK);
    get empty buffer                get full buffer
    from pool of empties            from pool of buffers
free(accessOK);                 free(accessOK);
    produce data                    consume data
wait(accessOK);                 wait(accessOK);
    add full buffer to              add empty buffer to
    pool of fulls                   pool of empties
free(accessOK);                 free(accessOK);
free(filled);                   free(empty);
```

A set of processes is in **deadlock** if each process is waiting for an event that only another process can execute. Because none of the processes can run, they are stuck enternally. Coffman showed that a deadlock only occurs when:

1. **Mutual exclusion**: Some resource can only be held by one process.

2. **Hold and Wait**: Processes holding resources can request new resources.

3. **No Preemption**: Resources granted cannot be forcibly taken away from a process.

4. **Circular Wait**: There must be a cycle of processes, each waiting for a resource held by the next member of the chain.

It is helpful to draw a resource allocation graph (a DAG to be precise) to detect circular wait conditions. The vertices are processes and resources. There is an edge from a process to a resource if that process is currently requesting the resource, and an edge from resource to process if that process is using the resource. The cycle condition is equivalent to this graph possessing a knot – a cycle such that every outgoing path is a cycle.

The best way to prevent deadlocks is to prevent them from ever happening in the first place. Enforcing deadlocks often imposes undue restraints on a computer – a high price to pay for a potential deadlock that is probably easier to manage from recognizing them in the first place. Direct methods prevent a circular wait condition. Indirect methods prevent the other conditions from occuring.

The Banker's algorithm prevents deadlock from occuring. Consider n processes $P_1, \ldots, P_n$, and m resources $R_1, \ldots, R_m$, together with an 'availability vector' A_j, giving us the number of units of a resource available. We also need a matrix M_{ij}, telling us the maximum amount of resource R_j process P_i will allocate. Initialize the $H = 0$ matrix, where H_{ij} is the number of resources R_j allocated to P_i. $N = M - H$ is the number of resources a process could possibly aquire. To determine whether it is 'safe' to allocate some resource R_j to a process P_i in the following manner. We see if we could allocate all needed resources to all processes. First, if the process tries to get more resources then it promised it should have, do not give it the resource. If it is not possible to get the resources, the process must wait – by an inductive hypothesis, this should not causes deadlock as long as we have been following the baker's algorithm from the beginning of allocation. Otherwise, simulate allocating the resource to the process – provided all processes are able to finish with the resources at the time, then a deadlock cannot occur:

We initialize $W_i = A_i$, and $F_i = FALSE$. Next, find a process for which $F_i = False$ (not finishd), and $N_{ij} \leq W_j$ (the process can feasibly complete its task with the current resources). Then, we pretend the process is done, we let $W_j = W_j + H_{ji}$ (the process has finished with the resources it was holding). Then set $F_i = TRUE$. Then rinse and repeat. If we 'finish' all processes ($F_i = (TRUE, TRUE, \ldots)$), then no deadlock can possibly occur. Otherwise, the system could potentially lead to a deadlock, and we cannot feasibly allocate the process with memory.

A state is very safe if we can run processes in any order to completion. A state is safe if there is at least one correct way to execute states. A state is unsafe if the algorithm fails, and deadlock has already occured if we are never able to allocate resources to any processes. One can recover from deadlock in three main ways: Preemption takes away resources from a process to allocate it to other processes. Rollback restores a previous state by periodically saving data. Termination recovers data by KILLING PROCESSES – this will probably cause errors.

Chapter 3

Interprocess Communication

Interprocess communication is effectively sharing information between processes. For instance shared memory explicitly shares information. **Message passing** uses a messaging system to explicitly send information without sharing memory locations. For instance, UNIX pipes implement a form of message passing.

Just like memory sharing, we need to have ideal properties our methods should have. We need to decide on

1. **Form of Communication**: Do we send messages **directly** or **indirectly**.

2. **Buffering**: How and where message are stored.

3. **Error Handling**: Dealing with exception conditions.

We need a send() command and a recieve() command to send messages. **Synchronous** forms of communication ensure that send and recieve block. A process only returns from a send() command when it is guarenteed that some other process has recieved the message, and vice versa. This is known as a **rendezvous**. An **Asynchronous** method does not block on a send command, but may block on a recieve (possibly many variants of the recieve command exist, to be used at the programmer's whim).

Processes **directly communicate** if one process is sending a message to one other process. The reciever knows exactly where the message is coming from. Indirect messaging is less secure, but requires less synchronization between programs. Indirect communication is often implemented as a **mailbox**, which are special repositories where information can be stored and accessed. In the case one reciever uses a mailbox, the mailbox is called a **port**. A port usually needs a buffer to store info. When the port is **bounded**, the buffer has a limited capacity, and a sender must wait until an empty spot is available before sending the message. When a buffer has **indefinite capacity**, the sender never waits, whereas if the buffer has **zero capacity**, message sending must be synchronized, the sender must send the message directly when the recieved gets it. In the asynchronous case, the sender must have additional mechanisms to guarantee the message is sent correctly.

We need to handle errors in message communication. The most common errors are:

1. The process could terminate before a message is processed.

2. A message could be lost in the communication network.

3. A message could be corrupted in transmission.

We won't discuss how to avoid these errors, but keep them in mind.

One cannot use semaphores or monitors to pass a message, since semaphores require globally shared memory, whereas monitors require centralized control. Nonetheless, message passing, in tandem with blocking, may be used for synchronization. In Unix, one may use signal between processes. A signal is **generated** when the event first occurs, and **delivered** when the process takes action on that signal. A signal is **pending** when generated but not delivered. Signals, also called *software interrupts*, generally occur asynchronously. A process sends a signal via the *kill()*, *raise()* (to send the signal to itself). To choose what code is executed when a signal is delivered, one uses the *signal()* command.

Chapter 4

Scheduling

When we learnt about processes and psuedo-parallelism, we acknowledged that a dispatcher needs to be able to decide which processes should run, given that there are multiple 'ready' processes at a given time. This is the task of (job) scheduling, controlling the order of work to be performed by a computer system. A process alternates between a **CPU burst** and an **IO burst**, using the processor, and aquiring more information from external devices. The CPU burst is of primary importance, since all processors must use it. One must maximize CPU burst efficiency for scheduling.

Long term scheduling determines when processes are allowed to aquire resources. When resource utilization is low, more jobs should be admitted, whereas the opposite should be true if resource utilization is too high. Switching between processes requires some small overhead, but this overhead is multiplied when switching between more and more processes. **Thrashing** occurs when the CPU spends more cycles switching between processes than actually running programs. One controls the **degree of multiprogramming**, or **multiprogramming level**, the number of processes in memory, to prevent thrashing and livelock. The only way to avoid programming is to reduce the multiprogramming level, by suspending a victim process, which are either lowest priority, newest, faulting most, has the smallest resident set, or the largest (which will give the biggest boost in memory).

The job of deciding when to suspend or resume a process is **Medium term** scheduling. When the main memory becomes over-commited. It is up to this component of the scheduling algorithm to swap out (suspend/block) processes in memory, as well as swapping back in processes.

Short term scheduling decides directly which ready processes should have CPU next when the CPU is free. This scheduler must run fast, and select processes fairly. The goal of such a process is to optimize throughput, while also providing responsive service (latency). Note that optimizing one performance criteria may decrease the performance via another, so that one must make trade-offs to have a well designed system.

Scheduling policies are either **preemptive** or **non-premptive**. Non-premptive systems let a process run until it blocks or terminates. Preemptive systems force the currently active process to release the CPU, via a clock interrupt, I/O interrupt, or system call. Non-premptive systems are normally better for managing 'batch' jobs, which run in the background for a long time, and normally need little to no I/O. On the other time, Non-preemptive systems are better for jobs which run in the foreground.

Non-preemptive scheduling policies include **First-In-First-Out** (FIFO) and **Shortest job first**. FIFO does what the name says – we put ready processes in a queue, and just run through them in the order of the queue. The algorithm performs better on long jobs, but does not account for the importance, or the estimated time to run a job. Short jobs may be bottlenecked by a long process that's in front of them in the queue, leading to the 'convoy effect'. Shortest job first runs processes in order of how fast they predict they will run for – of course, this is good for short jobs, but may starve longer jobs. One can't just lie about how long their process will take to run, because processes which exceed their run time will be killed.

On the preemptive side, one may apply round robin or priority based scheduling policies. Round robin cycles through jobs in FIFO by cycling processes out after a specific **quantum** of time has elapsed. Care must be taken to choose the time quantum correctly – too big, and interactive processes will suffer, too small and the CPU will thrash. Priority based policies schedule policies in order of their priority. One variation of this approach is to increase the priority of a process as it remains in the queue to ensure that it eventually is ran.

The best solution for general use combines multiple solutions to match the general variety of tasks available in an operating system. **Multi level queues** maintain multiple queues for different types of processes (system, interactive, and batch processes, for example), and apply different selection policies to the different queues. In a **Multi level feedback queue**, if a process exceeds its time quantum, it is pushed to a lower priority queue with a larger time quantum.

Chapter 5

Memory Management

Memory management decides how to use the limited supply of memory on a computer. A CPU can only directly access RAM and register memory values, so there must be a way to translate secondary storage (hard drive) memory onto the CPU. The OS must also protect memory to keep up the illusion that processes are not sharing physical memory, and to provide security measures. The policies must be **safe**, in the sense that a process will never corrupt the memory of another process. This must have a **low overhead**, and programs should not be **location dependant** – they shouldn't be required to excecute in portions of the memory.

A basic method to managing memory is to associate with each process a **base** and **limit** register value, which tells us where in physical memory a process' memory begins, and the total size of the memory. CPU hardware compares every address taken from memory with the registers, and gives a segmentation fault if a process is trying to access memory outside the specified range.

5.1 Creating a Program

The information required to run a process normally resides on secondary storage before it is ran – furthermore, this data may be moved back and forth from memory to storage as the program is ran. The processes on a disk waiting to be brought into memory for execution form the **input queue**. To create executable data from code, we proceed through multiple steps:

1. **Compiling** – generating object code from a program file. Normally, code is written in a symbolic form, which is **binded** by the compiler into more physical addresses. if the compiler knows where code will be ran, one may generate these addresses **statically**, the actual locations in memory where code will be ran. This is rarely used since it introduces location dependancy.

2. **Linking** – combining object code into an executable. Typically, a program consists of different files. The linker combines these files into a single executable file. The **static linker** collects multiple modules of code into a single executable. **Dynamically linked libraries** instead are linked when the programs run. This increases memory efficiency since multiple programs may use the same library, but decreases the easyness of distribution, since it is very easy to transport a single binary file.

3. **Loading** – copies executable code into memory. If it is not known at compile time where a program will be ran, a compiler must generate **reloadable code**. Final binding (conversion from relative to physical addresses) does not occur until a process is loaded into memory. In this method, one often uses **overlay trees** to manage the programs in code – the user organizes code into separate modules (**overlays**), which are distributed throughout memory. When the module is needed, it overwrites existing code in memory. There is always a root overlay in main memory, which manages the rest of the code. It is a simple way to allow programs to be larger than main memory, but nowadays is only really used in embedded systems.

4. **Execution** – dynamic memory allocation. If a process is moved during execution time, then binding must be delayed until the process is running. Special hardware must be used for this dynamic method, known as a **memory management unit**. In this case, the base register is known as the **relocation register**, and we add it to ever address generated by a user process. This is the most common method in operating systems.

5.2 Memory Abstraction

One of the main ideas of memory management is separation of a **logical address** from a **physical address**. A logical or relative address to a description of where memory should be located relative to some beginning (chosen by the base register, for instance), and is a re-enumeration of physical memory. Physical addresses are an actual location in memory. This means processes can effectively work as if they were the only process running on the computer, and removes location dependance. A user should never ever sees the real addresses – in fact, the registers required to find the address are restricted only for kernel access, so that the system is safe, no user can manipulate physical data.

The next trick to memory management is **swapping**, which treats main memory as a preemptable resource. It can be removed or given to processes as needed – processes are swapped in or out of main memory into a **backing store**, which is normally a fast disk (for instance, a flash drive, like the android operating system uses). The swapper selects processes which are suspended, blocked, low priority, or have been running for a long time, and places them into the store. Conversely, after some time in the store, a swapper will choose to swap the process back in. This is part of the previous discussed medium term scheduling.

One must divide memory up so that multiple programs can run on a system in a safe manner. A simple method to accommodating programs in memory is **contiguous memory allocation**. Each process has a contiguous **partition** of physical memory in which to run. Each partition contains exactly one process. When a process finishes, that partition can be allocated to a different process. In a **variable-partition** scheme, the OS keeps a **partition table**,indicating which parts of memory are available and which are occupied. Initially, all memory can be allocated to any process (and a process can have as much memory as it wants). Programs begin running in the smallest size partitions available at a certain time which can handle the information. This is slow, but is required by some applications like I/O Device readers. A **buddy system** gives blocks of memory in certain incremented sizes (powers of two, fibonacci), as more memory is required by a process. The reason for this is that two 'buddies' (memory blocks) can be **coalesced** into a single memory block for a contiguous region of memory

In the above method, some memory may be wasted, since partitions cannot be dynamically divided into exact contiguous segments for each partition. This is **fragmentation**. **internal fragmentation** is a waste of memory within a partition. **external fragmentation** is a waste of memory between partitions, caused by scattered free space. **compaction** is a technique to overcome external fragmentation, by shuffling memory contents into one large block. One cannot perform compaction in static allocation schemes, and requires overhead. **direct placement** occurs when code is loaded into memory in a specific location. It turns out that approximately half of the memory allocated is lost to fragmentation, known as the **50% rule**.

5.3 Segmentation and Paging

Segmentation provides a way to fix the fragmentation causes by contiguous partitions. Rather than having a single base and limit register for each process, one has a table for multiple segments of memory for each process, each of differing lengths and types (you may have stack segments, program binary segments, etc.). The MMU is able to decipher memory access into a physical location. Each entry in the table has a segment base and segment limit. One accesses memory in the segment table, first by declaring which segment posseses the data (the **segment number**), and the offset of bytes in the specific segment where the memory is located. The segment table is created in the PCB during initialization. Entries of the table fill up as new segments are allocated to the process, each giving memory which the process is allowed to use. The advantage of a process having multiple segments is that one need not swap the whole process out of memory at a single time.

Paging is a more robust technique in the same light as segmentation. External fragmentation is removed almost entirely by pre-partitioning physical memory into separate, fixed size blocks known as **frames**. Likewise, logical memory is split up into **pages**. A **page table** maps pages for frames for each process, and is very similar to a segment table (with page numbers and page offsets to index into the table). Page tables may be implemented by dedicated registers in memory for efficient. If the page tables must be larger, one puts the tables in memory with a base register indexing into the table. This requires a large overhead, since we first index into the page table to get a physical location, then use the physical location to get more from main memory. To compensate, a **Translation Look Aside Buffer** is used, a piece of associative memory (so all values can be looked at the same time), which works like a cache to look up some pages faster using the principle of **locality**: processes only use a small amount of the memory they possess at any one time. If we require even more pages, we use a multi-level page table (where a tier one page links to a tier two page, etc). The advantage of this is we need not store the whole page-table in main memory at once, even if it is slower than a look aside buffer (we again take advantage of locality). Alternatively, one can use **inverted tables**, which contain an entry for each frame of entry. Logical addresses are hashed into memory.

One can also use buffers in main memory, called **cache memory**, where frequent data usage is very fast (when data is accessed (the cache is **hit** rather than missed), it is moved to the cache). The first reference will always miss (it is **compulsory**). The size, mapping function used, and replacement algorithm are important. Also, if information is edited in memory, how will it be edited in the cache (**write through**, modify original memory as well as cache, and **write back**, update main memory only after location evicted). Mapping is either direct (modulo), associative (any line to any line), or via a set associative mapping.

5.4 Dynamic Memory

Dynamic memory is more tricky, for we cannot know how much memory we need. We use two functions, $malloc()$ and $free()$, to allocate and free dynamic memory. Dynamic allocation can be handled on the stack (restrictive and heirarchial, but efficient), or the heap (more general, but less efficient, and 50% fragmentation for each allocation). To organize heaps (keep track of free bits), one can use **bit maps**, which divide memory into fixed size blocks and keep an array of bits, one bit for each block. Linked lists can also be used to keep track of free memory blocks.

Memory is reclaimed when no process is using it anymore. How do we know when to free memory? If memory is freed from one process while another is using it, a **dangling pointer** occurs. **Memory leaks** occur when we forget to free allocated memory, so the OS cannot free it very intelligently. Memory reclamation can be done using reference counters, one for each memory block. Or, a program should keep track of pointers themselves, and free them when they're not being used (**Garbage Collection**).

5.5 Virtual Memory

As with pages, we normally only require a small amount of a processes memory at once. Virtual memory enables one to run a process while only a segment of all data are in the main memory at once. Rather than a logical address (now referred to as a virtual address) translating directly to physical main memory, the address may translate also to data in a paging device. The operating system can run programs which are only partially loaded into memory, taking advantage of locality, that programs only use very small amounts of their memory. When a program attempts to access memory on the paging device, the MMU checks whether the **present** bit (one for each device in the page table) is checked for the page. If the bit is not set, a **page fault** occurs – the process is blocked, the system attempts to find an empty frame in main memory, finds the memory on the paging device, fetches the data using an I/O operation, and triggers a 'page fetched' event, to unblock the process. The OS stills needs to make choices the hardware cannot. For instance, how much main memory to allocate to each process, when to fetch pages into main memory, which frames a page should be loaded into, and when to remove pages from memory.

The allocation policy must deal with conflicting requirements. The fewer the frames for a program, the higher the page fault rate, but the more programs can be allocated into memory. A program doesn't need too many frames due to locality, so an estimate must be chosen. The number of frames allotted to a process (the **resident set size**) can be fixed or variable.

Fetch policies also have various choices. **Demand paging** starts a program with no pages loaded; the program waits until a page is referenced before loading it into main memory. **Request paging** lets the user identify key pages for preloading (not practical, since it removes abstraction, and can lead to overestimation). **Prepaging** is an inbetween, choosing a few key pages for preloading, while using demand paging afterward. The **cleaning** policy decides when a page should be written back to the paging device.

Placement policies follows rules similar to paging and segmentation discussed earlier. Given that a page is the same size as a state, placement in tandem with paging is straightforward. Segmentation requires more effort when combined with paging. **non uniform memory access** (NUMA) (such as a system of servers), placement is a major concern.

The **Replacement** policy is the most studied area of memory management, also known as **victim selection**. In FIFO, the longest resident is removed. In LRU (least recently used), the frame which has not been used for the longest time is removed. Random policies also exist. One compares the success of these algorithms against the optimal solution, a theoretical policy which removes pages which will not be used for the longest time. **Global** policies select victims from all policies, whereas **local** policies select victims from only the faulted process. If a frame is vitally important, it may be **frame locked / page pinned** so it cannot be removed (belong to the kernel, or are critical for performance). Victim frames are grouped into **clean** (unedited frames), and **dirty** (edited frames).

The **clock algorithm** selects the least recently used process. Each frame in memory is allocated a 'recently used' bit. All the frames are kept in a circular queue. A pointer indicates which frame wasjust replaced. When a frame is required, the pointer advances to the first frame with a zero used bit, changing all one bits to zero bits as it moves. Once a frame is chosen, its used bit is set to one. The hardware sets the used bit whenever the page is referenced. The **second chance algorithm** is the same, but uses a circular queue upon which bits with a one bit are set to the back. Some systems also include a 'dirty bit' to give preference to dirty pages, since it is more expensive to victimize a dirty page. If the clock hand is moving fast, then we don't have enough memory. If the clock is slow, then there are not many page faults (system is loaded too lightly).

The **working set** of a program is the set of pages used by the last Δ references, denoted $W(t,\Delta)$. As an example, for $\Delta = 10$

$$\underbrace{2615777751}_{t=0}\,623412344443434441323444\,\underbrace{3444233312}_{t=33}\,334$$

Denning's **Working set Principle** states that a program should only run if its working set is in memory, and a page should not be victimized if it is a member of the current working set of a runnable (not blocked) program. The problem with this approach is that the working set principle requires a large overhead. A solution maintains idle time for each process (amount of CPU time allocated to a process since it last accessed the page). Every once in a while, scan all pages. If a used bit is on, clear the idle time, or add time since last check, and then turned off all used bits. The collection of pages with the lowest idle time is the working set.

An alternative to the working set principle is the **page fault frequency** algorithm, defined as the inverse of the **critical inter-page fault time** (the average time between page faults). When a process goes below a lower bound, a frame is taken away from it (it's resident set size is reduced), whereas when it goes above an upper bound, a frame is allocated to it.

5.6 Secondary Storage

Secondary storage is non-volatile storage for data and programs. The **file system** manages this information, and is considered an integral part of the operating system. One uses secondary storage to store various forms of programs, and swap out virtual memory units. Information in secondary storage is any raw data.

A **file** is a *named* collection of bytes. The **logical view** is how the user sees the bytes, after interpretation. The **physical view** is purely how the bytes reside in memory. Files are different then data structure in that they are intended to be non-volatile, and be moved around and accessed by different processes. One can manipulate files by six basic instructions

create write read delete seek truncate

Each file has a collection of attributes associated with itself, such as the filename, owner, type, etc. A common implementation technique is to include the type as part of the file name. Files are accessed either sequentially, randomly, or keyed in a hask. File sharing between users brings security into question. The best way is to allocate permission attributes to each user. A user may be allocated into a group, with a general class of permissions. In UNIX, attributes are kept in an Inode in the table of a directory.

A **directory** is a symbol table, which can be used to search for file information. A directory entry contains information about a file. Common structures are **flat structures** (shared by all users), **two level** (used by two), and **trees** (arbitrary subtrees for each user). UNIX uses a DAG structure.

A **file system** is map between logical and physical memory. The file system hides device-specific aspects of file manipulation from users. Basic services include keeping track of files, I/O support, secondary storage support, and providing protection mechanisms. There are three mapping levels for a file system. **File relative** is high level, mapping in the form $<$ filename, offset $>$ to data. **Volume relative** is still device independent, and uses $<$ sector, offset $>$. At the lowest level, one uses $<$ cylinder, head, sector $>$.

File organization is the logical structure of a system, such as how the files are organized in memory at a high level. **File allocation** allocates disk space when a file is created. Common techniques are continguous, chained, or indexed allocation. All techniques allocate on a per block basis.

As with the RAM, we need a policy for allocating space on a hard drive. The **contiguous memory** policy allocates space like paged, segmented memory. Contiguous allocation is like paged memory. We just keep a free list of unused disk space. This results in easy access and is simple, but causes external fragmentation as we rewrite memory, and we may not know how big a file is in advance. **Chained allocation** marks allocated blocks in use, and uses a linked list method to chain blocks together. It has no external allocation, and files can grow easily, but there are lots of seeking, and random access is difficult. FAT uses this. **Indexed allocation** allocates pointers during file creation to a specific index block of data, and fills the block as new disk blocks are allocated for the file. This results in small internal fragmentation, and has easy sequential access, but requires lots of seeks, and the maximal file size is limited. UNIX uses this.

In Multiprogramming systems, there may be several disk I/O requests at the same time, and the OS must decide which request to process first. The most costly component of a disk I/O operation is the **seek time**, the time to find a file on the disk. FIFO is a commonly used policy, just processes requests as the come. One can also follow the shortest service time first algorith, by processing memory that is closest in physical space. This may cause a livelock, however, so the SCAN policy may be used instead, where the disk completes a full cycle of the data, processing the data in clockwise order, for instance. LOOK scheduling is a modification of SCAN, but only goes as wide on the disk as it needs to to processes current requests.

The amount of disk space is limited, it is necessary to reuse disk space released by deleted files. In general, systems keep a list of free data frames. Bit maps can be used to indicate if a block is full – we have a bit for each block, and each simply tells us if a block is being used. Another method is to link all free blocks together, which we can append to in constant time.

Other issues include disk blocking, disk quotas, reliability, and performance.