CPSC 3220 Final Exam Study Guide

Brendan McGuire (bmmcgui@clemson.edu) • Patrick Smathers (psmathe@clemson.edu)

Disclaimer: While I attempt to be as accurate as possible, I am a student like you. If you notice anything incorrect or misleading in this document, please let me know, and I can correct it!

Good Luck on the Final! ❤️

Operating Systems

An operating system is one of the lowest levels of code that runs on a machine. It directly interfaces with hardware, manages the system's resources, and protects the system from the code that runs above it.

The operating system provides a layer of abstraction around the hardware. We don't want to have to write our program to be able to account for every possible combination of hardware our users run it on. We use the operating system to provide a common interface to interact with that hardware.

Kernel/User Space & Microkernels

The operating system's core is the kernel, which manages the system's resources. Any code that runs in the kernel has ultimate access to the hardware. The kernel creates its own abstractions to interact with the hardware in a more protected way.

Kernel Space	User Space
- Utilized by kernel only - Access to entire memory - Unrestricted access to hardware	- Runs user programs - Separate virtualized memory - Access to hardware mediated through abstractions managed in kernel space

Microkernel – Moves as much to user space as possible

- ➔ Kernel usually provides just memory management/virtualization, interprocess communication, and thread management
- ➔ To interact with other operating system functions (device drivers, file system, etc) user programs use IPC to other user space programs
- ➔ Source code of kernel often smaller
- ➔ Performance often worse (due to excess messaging passing)

Monolithic Kernel – The entire operating system happens in kernel space

- ➔ Kernel provides primitives (system calls) to interact with all OS abstractions (process management, memory management, file system, I/O)
- ➔ Device drivers are kernel modules
- ➔ Source code significantly bigger than microkernel
- ➔ Performance often better due to tighter integration

Worked Example (Quiz 2)
Which of the following statements about microkernels are true?

A microkernel probably has less code than a Linux kernel.
This is correct. Because more of the operating system's features are handled in user space, less code needs to be in the microkernel

Microkernels are usually more stable (harder to crash) than monolithic kernels.
This is correct. Not only is there less to go wrong, but because microkernels are more split up than monolithic systems, they can be more fault resistant. For example, in a monolithic kernel, a single kernel panic usually results in your entire system crashing. However, if a user space process running a component crashes, it is significantly less likely the entire system will crash

The Linux kernel is a microkernel.

Most desktop/laptop/server operating systems use a microkernel design.
This is incorrect. Most operating systems are based loosely on the linux kernel, which is a monolith kernel, the more traditional implementation.

Interprocess communication is more important for microkernels than monolithic kernels.
This is correct. Because more functionality of the operating system occurs in user space, one of the primary functions of the microkernel is to facilitate interprocess communication.

Microkernels are usually faster than monolithic kernels.
This is incorrect. Because of the overhead of interprocess communication, they tend to be slower.

A microkernel is easier to secure than a monolithic kernel.
This is correct. Because there is more isolation between the components of the kernel, compromising one component does not necessarily compromise the entire system

Processes & Threads

To run multiple instances of programs "at once", the operating system creates an abstraction, processes. These represent a running instance of a program, and each process is convinced that it has unfettered access to the system. The isolation of a process gives it increased stability

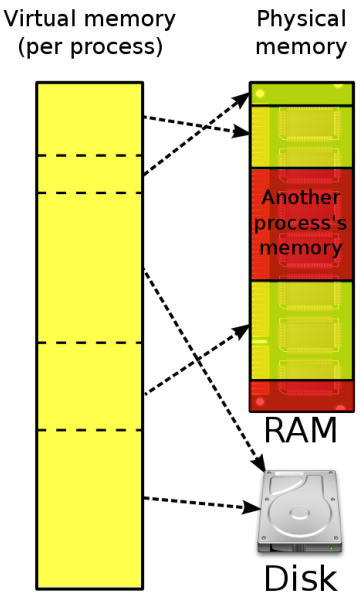
A primary job of an operating system is managing these different processes. Process and threads enable *parallelism* and *concurrency,* two distinct but related terms.

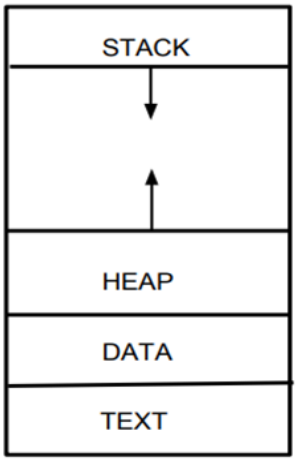
Parallelism – In a multi-core system, simultaneously executing several instructions at once by dividing the work between your cores.

Concurrency – Time-sharing (scheduling) so that task execution is interleaved, quickly switching between tasks. Note that we are never performing multiple tasks at the same time, instead, we are switching very quickly between tasks

Virtual Memory

Each process is assigned **virtual memory space** by the operating system. From the process' perspective, they have unfettered access to the system's memory. The operating system manages this abstraction and will move memory to swap (files on disk) when memory usage exceeds system memory. Memory is organized into different sections, as shown on the left. Note that there are often gaps between each section to prevent overlap, especially with larger memory spaces in more modern systems.





Stack – Grows towards the heap, contains all the information about the caller when a function is called so that execution can correctly resume

Heap – The segment where dynamically allocated memory is placed. Grows towards larger addresses

Data – Stores global and static variables, and string and numeric literals. Usually divided into read-only and read-write sections

Text – Stored near address zero, this contains all of the executable instructions in the program

Interprocess Communication (IPC)

There are several different ways that processes can communicate, each with their own considerations.

Messaging Passing

One of the simpler ways of IPC is message passing, which allows you to transmit data between two active processes.

```
//Create a mailbox for this process
int msqidS = msgget(MAILBOX_NUMBER, 0600 | IPC_CREAT);
// get length bytes from the other end of the mailbox and write to result
result = msgrcv(msqidS, &cmbox, length, 1, 0);
//send the response.
```

```
msgsnd(msqidC, &msgp, length, 0);
```

Named Pipes

A named pipe allows you to take advantage of standard file I/O functions like fgets when passing data between processes. This allows you to use standardized functions to communicate.

```
// receive data
umask(0);
mkfifo(FIFO_FILE, 0666);
while (1) {
    fp = fopen(FIFO_FILE, "r");
    fgets(readbuf, 80, fp);
    printf("Received string: %s\n", readbuf);
    fclose(fp);
}

// write
fp = fopen(FIFO_FILE, "w")
fputs(argv[1], fp);
fclose(fp);
```

Networks

A similar but more elaborate means of communication happens through the use of network sockets. Just like message passing, network sockets in C use the same FILE pointer in all standard functions, allowing you to make use of existing file I/O code.

```
int sockfd, connfd, port, clilen;
struct sockaddr_in saddr, caddr;

char buffer[BUFLEN];

sockfd = socket(AF_INET, SOCK_STREAM, 0);
memset(&saddr, 0, sizeof(struct sockaddr_in));
port = PORTNUM;
saddr.sin_family = AF_INET;
saddr.sin_addr.s_addr = INADDR_ANY;
saddr.sin_port = htons(port);

if (bind(sockfd,
      (struct sockaddr*)&saddr,
      sizeof(saddr)) < 0) {
      perror("bind failed!");
      exit(1);
}

listen(sockfd, QUEUE_LENGTH);
```

For more information about using network sockets to communicate, see the example provided by Dr. Sorber, or recall your notes from CPSC 3600 😤

Shared Memory

For applications where message passing can be cumbersome, sharing a block of memory between 2 processes may be appropriate. However, a wise programmer should be cognizant of the additional performance costs that can be associated with this, and where the implementation would be better served by two threads inside of the same process, where memory is shared automatically for free.

```
char *sharedblock = (char*)mmap(NULL, BLOCKSIZE, PROT_READ|PROT_WRITE, MAP_ANON |
MAP_SHARED, -1, 0);
strcpy(sharedblock, "Hello World");

// Depending on whether the child or parent runs first, we should see the child print
// out "parent was here" or the parent print out "child was here"
if(fork() != 0) {
   // In parent
   printf("%s\n", sharedblock);
   strcpy(sharedblock, "parent was here");
} else {
   // In child
   printf("%s\n", sharedblock);
   strcpy(sharedblock, "child was here");
}
```

Signals

A means of communication between processes. This is a system call a user process can make to interact with another currently running process. In POSIX, kill is the syscall used for signaling. The operating system can also send signals to your program to control it.

Common signals defined in POSIX		
Signal	**Number**	**Effect**
SIGABRT	6	The SIGABRT signal is sent to a process to tell it to abort, usually self initiated.
SIGTERM	14	The SIGTERM signal is sent to a process to request its termination. Unlike the SIGKILL signal, it can be caught and interpreted or ignored by the process. This allows the process to perform nice termination releasing resources and saving state if appropriate
SIGKILL	9	The SIGKILL signal is sent to a process to cause it to terminate immediately (kill). In contrast to SIGTERM and SIGINT, this signal cannot be caught or ignored, and the receiving process cannot perform any clean-up upon receiving this signal.
SIGFPE	8	Sent by the kernel when the process mode has a Floating Point Exception (i.e. dividing by zero). Floating point here is a misnomer since this covers all arithmetic errors, not just floating point

Read more about POSIX symbols: https://www.man7.org/linux/man-pages/man7/signal.7.html

Creating New Processes

Fork

The fork operation copies the current process into an exact identical copy, and then continues execution in both contexts. In the C standard library, the fork() function will return 0 in the child process and the child's process ID in the main process. This allows you to differentiate between the two processes.

Worked Example (Quiz 2)

Example: what would the following code output in Linux?

```
int main() {
    int x = 2;
    if (fork() != 0) {
        x--;
        wait(NULL);
        printf("%d \n",x);
    } else {
        x -= 3;
        if (fork() != 0) { wait(NULL); x++; }
        printf("%d ",x);
    }
}
```

This would output **-1 0 1.** Consider the following timeline:

<table>
<tr><th colspan="3">Original Process</th><th>X</th><th>Forked Process</th><th>X</th></tr>
<tr><td colspan="3"><code>int x = 2;</code></td><td>2</td><td></td><td>2</td></tr>
<tr><td colspan="3"><code>if (fork() != 0) {</code></td><td>2</td><td><code>if (fork() != 0) {</code></td><td>2</td></tr>
<tr><td colspan="3"><code>x -= 3;</code></td><td>-1</td><td><code>x--;</code></td><td>1</td></tr>
<tr><td colspan="3"><code>if (fork() != 0) { wait(NULL); x++; }</code></td><td>-1</td><td><code>wait(NULL);</code></td><td>1</td></tr>
<tr><td>Forked Process</td><td>X</td><td>Original Process</td><td>-1</td><td><code>wait(NULL);</code></td><td>1</td></tr>
<tr><td><code>printf("%d ",x);</code></td><td>-1</td><td><code>wait(NULL);</code></td><td>-1</td><td><code>wait(NULL);</code></td><td>1</td></tr>
<tr><td>Process Ends</td><td></td><td><code>x++;</code></td><td>0</td><td><code>wait(NULL);</code></td><td>1</td></tr>
<tr><td></td><td></td><td><code>printf("%d ",x);</code></td><td>0</td><td><code>wait(NULL);</code></td><td>1</td></tr>
<tr><td></td><td></td><td>Process Ends</td><td></td><td><code>printf("%d \n",x);</code></td><td>1</td></tr>
</table>

Exec

Vector Arguments (execv, execvp, execve, execvpe)

The execv variants allow you to pass a vector (NULL terminated array) as the command line arguments. The execv(), execvp(), and execvP() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers must be terminated by a NULL pointer. See below for an example.

```
// Copy over the arguments, and then null terminate
char **argv = (char **)malloc(sizeof(char *) * (args_len + 1));

// The list of arguments must be null-terminated.
argv[args_len] = NULL;
execv("./bin/executable.out", argv);
```

Variadic Arguments (execl, execlp, execle)

These variants allow you to pass command line arguments as arguments to the function. The const char *arg0 and subsequent ellipses in the execl(), execlp(), and execle() functions can be thought of as arg0, arg1, ..., argn. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments must be terminated by a NULL pointer. See below for an example.

```
execl("./bin/executable.out", "hello", "world");
```

PATH Searching (execvp, execlp, execvpe)

The functions execlp() and execvp() will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash "/" character. For execlp() and execvp(), search path is the path specified in the environment by "PATH" variable. See below for an example.

```
// Copy over the arguments, and then null terminate
char **argv = (char **)malloc(sizeof(char *) * (args_len + 1));
// fill in arguments here
// The list of arguments must be null-terminated.
argv[args_len] = NULL;
execvp("ls", argv);
```

Environment Variables (execle, execvpe)

Variants that include e allow you as the user to specify additional *Environment Variables* to the process as you run it. See below for an example.

```
// Create a new environment variable array to specify environment variables .
char *envp[] = {"ENV=PRODUCTION", NULL};

// Copy over the arguments, and then null terminate
char **argv = (char **)malloc(sizeof(char *) * (args_len + 1));

// Fill in arguments here

// The list of arguments must be null-terminated.
```

```
argv[args_len] = NULL;

execvpe(program, argv, envp);
```

Note: execvpe is not defined as part of the standard, so you must #define _GNU_SOURCE to include the GNU extensions.

Threads

Inside of a process, smaller sequences of execution that can be paused and resumed are known as threads. These are more lightweight than processes, as they don't have their own memory. Instead, threads share all dynamically allocated and non-thread-local global variables. For many situations, this can be advantageous.

Kernel Threads

These are components of the process which are managed by the kernel. The process will register threads (by using the pthreads API), and the operating system will manage them.

- ➔ Threads managed by kernel
- ➔ Threads can be placed across multiple cores on the CPU, so parallelism is possible.
- ➔ Do not have their own resources except for stack, copy of registers, and thread-local storage
- ➔ Switching between threads much faster than switching between processes

User Threads

Threads can also be implemented in userspace libraries, where the kernel is not aware of them. Instead, the user library will write their own scheduler to manage parallel execution

- ➔ Threads managed by user library
- ➔ Cannot usually take advantage of multiprocessor/multicore machines, because kernel not aware of them
- ➔ Extremely fast to switch, often times a context switch to kernel space not needed
- ➔ More control over scheduling than kernel threads

> *Example: Are kernel threads or user threads more likely to speed up a CPU-bound workload?*
>
> In a multi-core system, kernel threads will be more likely to speed up a CPU-bound workload. This is because a key differentiating kernel thread feature is the ability to assign different threads to different processes.
>
> For tasks that can be easily divided without much sharing between threads, this is especially true. Speedups from parallel execution significantly diminish when the task requires data to be transferred between CPUs so that threads can coordinate and synchronize.

Thread Safety and Coordination

The lack of total isolation between threads inside of a process is a bit of a sword without a hilt. The freedom to properly share memory allows you to massively increase the speed of communication between threads, but must be carefully handled, as the order of execution is not guaranteed.

Race Conditions

For example, consider the following simple example. *A web server is handling requests from clients by routing them to one of many different threads. It keeps track of the total number of requests in a global variable. When a client connects to the server, a thread will read the global request count variable, increment it, and then store it into the server.*

However, when a thread is interrupted in the middle of updating the count, the next thread will read the incorrect value, increment it, and then update it. This will lead to us massively undercounting the number of requests made, because we are incrementing on the same base value.

The way to fix this is to make the read, increment, and write operation **atomic** (unable to be interrupted). We can achieve this one of several ways, depending on how we need to coordinate.

Synchronization Primitives
Here each primitive is described conceptually. For more precise examples on implementations in the pthreads library, see the pthread

Mutex
A mutex will protect a shared resource and ensure that only one thread has access to it at once. This is accomplished by 2 functions, lock and unlock.

Lock will block execution in the calling thread until it can get access to the resource. In most threading libraries, this is implemented intelligently enough that the thread is not scheduled until they can acquire the lock.

Unlock will release control of a resource that you have access to, and allow others to lock it. You can only unlock if you have the lock in your thread.

RW Lock
In situations where reading is much more common than writing, a mutex can lead to unnecessary locking. For example, consider sharing a cache value between all of your threads. Each thread may need the cache value consistently, but since it is updated only occasionally, it is wasteful to ensure that only 1 thread can read at a time.

A Readers-Writer lock solves this problem by allowing many simultaneous threads to hold *reader locks* OR a single thread to hold the *writer lock*. In this way, many readers can access the data concurrently, and writes are deferred until all reads have been completed. Similarly, readers can guarantee that while they have the reader lock, the data will not change.

Binary Semaphore
A binary semaphore is extremely similar to a mutex, except that it does not necessarily indicate exclusive ownership of a resource. A simple integer value is associated with a semaphore, which represents the state of the synchronization. In a binary semaphore, this value is restricted to either 0 or 1

The wait operation will block thread execution until the semaphore reads a specific value (usually 1). The post operation will transition the state to that value.

Counting Semaphore
A counting semaphore extends the idea of a binary semaphore by allowing the internal value to accumulate many values beyond just 0 and 1.

Condition Variable
By using a condition variable, we can put threads to sleep until *specific conditions* are met. When this occurs, there will be a queue of threads that are dependent on that condition that can now execute. So a condition variable is composed primarily of an execution queue. Calling wait() on the condition variable indicates that the system

Reentrance
One of the important concepts of concurrent execution is the idea of preemption. In a preempting scheduler, the processor could stop execution of your program at any time, and your code needs to handle this.

An important consideration for functions in a multithreading context is the idea of *reetrance*. A function is reentrant if it can be interrupted in the middle of execution, and have a new invocation begin before the previous invocation completes. An important consideration for reentrance is the *state that is shared between all function invocations*.

Worked Example (Quiz 5)
I am writing a program with multiple threads that all access a shared database. If 99% of my threads' interactions with the database involve reading (1% of the time we're updating information), **which of the following statements are true?**

Since we're mostly reading, we don't need synchronization.
Because there are occasional writes, we need some level synchronization to ensure that multiple writes do not happen in parallel in such a way to destroy data.

Using a reader/writer lock to protect the database would be less efficient than a mutex lock.
This is incorrect.

Using a mutex lock to protect the database will not be sufficient to ensure correct operation.
Using a mutex would be perfectly fine, it would just be incredibly slow

Using a reader/writer lock would be more efficient than using a mutex.
Because the vast majority of operations are reads, it does not make sense to allow only one reader at a time (like the mutex). Instead using a RW lock would allow multiple concurrent readers while they are no writers. When a writer requests the writer lock, it would block until all readers were finished, grant the writer lock, and then prevent readers from reading until the writer completes its work.

Using a mutex lock to protect the database, will result in functionally correct operation.
Allowing only one thread to access the database at a time would result in correct operation, it would just be needlessly slow.

Using a reader/writer lock will have roughly the same efficiency as a mutex lock.
This is incorrect, see above.

A reader/writer lock cannot provide adequate protection for this database.
This is incorrect. Because the RW lock allows only one writer at a time and ensures that there are no readers while there is a writer and that writers write sequentially.

Consider the following pseudocode. What kind of semaphore is this?

```
semaphore_init(s1, 1); //initialize semaphore s1 to a value of 1 at startup.
Thread1 () {
   semaphore_wait(s1);
   count++;
   semaphore_post(s1);
}

Thread2 () {
   semaphore_wait(s1);
   count--;
   semaphore_post(s1);
}
```

This is a binary semaphore, because the only values the semaphore can contain is 0 or 1.

Consider the following pseudocode. Which of the following are possible outcomes of this program?

```
int count = 0;
Thread1 () {
   semaphore_wait(s1);
   semaphore_wait(s1);
   count=count+2;
   semaphore_post(s1);
}

Thread2 () {
   semaphore_wait(s1);
   count=count-1;
   semaphore_post(s1);
   semaphore_post(s1);
}

main() {
   semaphore_init(s1, 0); //initialize semaphore s1 to a value of 0 at startup.
   start(Thread1);
   start(Thread2);
   semaphore_post(s1);
   join(Thread1);
   join(Thread2);
   print(count);
}
```

The behavior of this program is dependant on which thread the scheduler designates to run first. Both threads will immediately start, and then suspend due to the semaphore containing a value of 0. The main thread will then run, posting a value of 1. This will cause either thread 1 or thread 2 to run

If thread 1 runs first
Thread 1 will wait (via the second semaphore_wait call), causing all threads to be waiting for the semaphore, which will never increase in value.

If thread 2 runs first
The count will be reduced to -1, and then the thread will post the semaphore, causing thread 1 to run (and ending thread 1). This will increase count back to 1, and then post, ending thread 1. Now that both threads have finished executing, the print statement in main will print out the value of count, 1

Amdahl's Law

A quick way to determine how much your computation can benefit from parallel execution. First, create an estimate *p* of the portion of your program that can be computed in parallel. Then, an upper bound for the speedup you can attain by running the process on *N* parallel nodes is shown below.

$$speedup = \frac{1}{(1-p) + p/N)}$$

It's important to remember that this is an **upper bound**, the maximum speedup factor you can expect from running your computation in parallel. Synchronization costs (like waiting for shared resources) mean you will rarely get this level of speedup from your program.

It is also important to note that increasing the number of parallel executions has diminishing returns. Since it is impossible to parallelize some sections of your process, you cannot provide a speedup that exceeds that minimum time

Example: A computation takes 60 hours to complete on a single core, where 20% of the task must be performed in sequence.

a) What is the minimum time this process would take if it were run on 10 cores?

$$p = 0.80$$

$$Speedup = \frac{1}{(1-0.8) + 0.8/10}$$

$$Speedup = \frac{1}{0.2 + 0.08} \approx 3.5714$$

$$Time = 60/3.5714 = 16.8001344$$

So, if run on 10 cores, the computation will take 16.8 hours

b) How many cores are needed for a 2.5x speedup?

$$2.5 = \frac{1}{(1-0.8) + 0.8/N}$$

$$\frac{1}{2.5} = (1-0.8) + 0.8/N$$

$$0.4 = 0.2 + 0.8/N$$

$$0.2 = 0.8/N$$

$$N = 0.8/0.2 = 4$$

So, running this process on 4 cores will give you at most a 2.5x speedup.

Time for *n* cores based on single core time:

$$T_n = T_1(S + \frac{P}{n})$$

$$T_n = \frac{T_1}{speedup}$$

Example: There is a task that takes 10 hours on a single core and 80% of the task can be done in parallel.

a) How long (best case) would it take if 10 cores are used?

$T_{10} = 10(0.2 + \frac{0.8}{10})$

$T_{10} = 2 + 0.8$

$T_{10} = 2.8\ hours.$

b) How many cores would be needed to get the runtime to 2 hours?

$2 = 10(0.2 + \frac{0.8}{n})$

$2 = 2 + \frac{8}{n}$

$0 = \frac{8}{n}$

$0n = 8$

$n = \frac{8}{0}$

Not possible. The sequential work will always need 2 hours, so the execution time with parallelism asymptotically approaches 2 hours

Scheduling

When we have several tasks to execute concurrently, we need to decide what tasks to run when. The operating system scheduler manages this for processes, and kernel threads. Userspace thread libraries will often contain a scheduler to manage concurrent execution.

Cooperative Scheduling

In cooperative scheduling, the scheduler picks a task and runs it until it yields control to another task. At this point, the scheduler picks a different task and resumes its execution. How exactly the next task is chosen depends on the scheduling discipline of the implementation.

- ➔ Easier to implement
- ➔ Single tasks can hog execution time by never yielding control back to the scheduler
- ➔ Often used on microcontrollers, or other situations where you can implicitly trust every task to behave

Preemptive Scheduling

In a preemptive scheduler, tasks can be interrupted in the middle of execution without yielding. This allows the scheduler to have control over how execution proceeds. Typically preemptive scheduling divides time into a number of **quanta** (a small unit of time, say 10 microseconds). The scheduler will run a task for one quantum, and then decide what to do next. How exactly it decides how to switch is dependent on the scheduling discipline of the implementation

- ➔ Preemption usually requires hardware (timer interrupt)
- ➔ Stops one task from starving out execution by never yielding

Scheduling Discipline

How exactly the scheduler decides what to execute next can vary from implementation, and is generally referred to as the discipline of the scheduler. Different disciplines make different trade offs in task lead time and execution time.

Wait Time – The time from work becoming ready and the first point it begins execution
Response Time/Turnaround Time – The time from work becoming ready until it is completed

First Come, First Served (FIFO) – Cooperative

There is a queue of tasks to be performed. The scheduler will switch to the next task in the order when a previous task yields.

Priority Queue – Cooperative

There is a priority queue of tasks to be performed, with each task being assigned a priority. Processes with lower priority can have extremely long lead-times if higher priority tasks consistently preempt them.

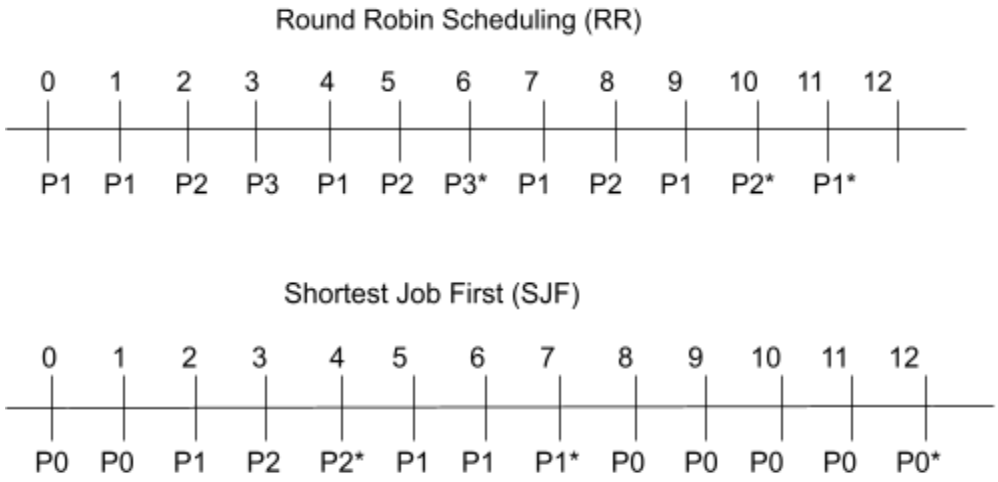
Round Robin (RR) – Preemptive

The scheduler will switch processes every quantum and move to the next process in the queue.

Shortest Job First (SJF) – Preemptive

The scheduler will estimate how long the task will take, and run the shortest one. This can lead to starvation if short jobs consistently come in and preempt longer running tasks.

P#	Arrival	Burst Length
P1	0	7
P2	2	4
P3	3	2



Proportional Fair Scheduling

Each task is assigned a priority (either given by the user or estimated like SJF). The higher priority tasks are *more likely*, but not guaranteed to be executed first.

Worked Example (Quiz 6)
Consider the following set of CPU bursts.

```
Process     Quantum Arrived/Issued  Burst Length (# of quanta)
P1          0                       3
P2          1                       4
P3          2                       2
P4          3                       1
```

For all of the following questions, if you ever run into a tie (where two bursts could be selected) use the lower-numbered process. So, in a tie between P2 and P4, you would select P2.

1) **If I schedule these bursts with preemptive SJF, in what order will they complete?**

   Consider the following timeline. When a process completes, that column is bolded. Remember, in a preemptive context, we ca

Time	0	1	2	3	4	5	6	7	8	9
Process	P1	P1	**P1**	**P4**	P3	**P3**	P2	P2	P2	**P2**

   So, the processes will complete the following order: **P1, P4, P3, P2**

2) **Again, with preemptive SJF, what is the wait time (in quanta) of P3?**

   Wait time is the time from when the work arrives until it begins execution. P3 arrives at t=2, and begins work at t=4, so the wait time is **2 quanta**

3) **If I switch to cooperative SJF what is the wait time (in quanta) of P3?**

   Consider the following timeline. When a process completes, that column is bolded. Interesting the cooperative SJF execution timeline is identical to the preemptive timeline shown above

Time	0	1	2	3	4	5	6	7	8	9
Process	P1	P1	**P1**	**P4**	P3	**P3**	P2	P2	P2	**P2**

   Just as above, P3 arrives at t=2, and begins work at t=4, so the wait time is **2 quanta**

Predicting Task Length with Exponential Weighted Moving Average

Previous scheduling disciplines have relied on an estimation of the length of the bursts as a scheduling consideration. But those familiar with the halting problem will know that accurately knowing this number is mathematically impossible, so we use heuristics to estimate.

Instead, operating systems will make estimations about how long a task will burst for based on previous execution time. This is often done using an **Exponential Weighted Moving Average**, which takes into account the most recent execution time to converge on an estimate for the execution time. *Note: this method is bad for tasks which have inconsistent oscillations in burst time, as EWMA will constantly chase the old value up and down*.

$$p_n = \alpha p_{n-1} + (1 - \alpha)m$$

Where

P_n refers to the prediction at time n

P_{n-1} refers to the prediction average at the previous quantum (n-1)

α is a tuning value, usually on the order of (0, 1)

m is the previously measured actual execution time

Example: Consider an operating system scheduler that uses Shortest Job First and estimates task execution time by Exponential Weighted Moving Average, with α = 0.3. Shown below are a list of tasks, the current estimation for duration, and their previous execution times. Assuming that all tasks are ready to be executed, which task will the scheduler pick to execute first?

Task	*Current Estimation*	*Last Estimation Time*	*New Estimation* $p_n = \alpha p_{n-1} + (1 - \alpha)m$
0	*2*	*12*	$p_n = (0.3)(2) + (0.7)(12) = 9$
1	*14*	*7*	$p_n = (0.3)(14) + (0.7)(7) = 9.1$
2	*4*	*4*	$p_n = (0.3)(4) + (0.7)(4) = 4$

Long Term, Medium Term, and Short Term Scheduling

In a kernel scheduler, each of these terms refer to the different points at time in which a processor must make a decision.

Long Term Scheduler – Determines which tasks are to be admitted to the ready queue of tasks, also known as Admission Control. The long term scheduler must make decisions to admit tasks in order to balance *I/O Bound* and *CPU Bound* tasks. Schedulers in modern OSes (like Windows, Mac, Linux) don't really say "No" to tasks often

CPU Bound Tasks: Execution is mostly dependant on the CPU, task is largely doing crunchy calculations
I/O Bound Tasks: Execution is mostly dependent on responses from I/O devices.

Medium Term Scheduler – Determines which tasks have their state loaded into main memory, and which ones are swapped to disk.

Short Term Scheduler – Determines which tasks get run from quantum to quantum.

CPU Affinity

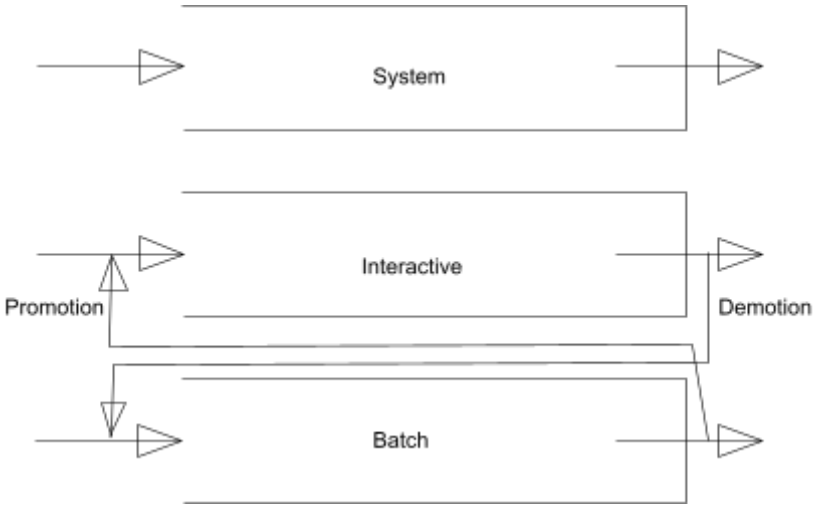
This is the means by which you can instruct the compiler to schedule a specific thread on only a specific set of cores. In the Linux API, the function `sched_getaffinity` allows you to set a bitmask of the cores which you want to allow this thread to run on.

Note: this is generally considered a bad idea. If you cannot guarantee that you know better than the kernel where to schedule this thread, you should not use this thread.

Process vs. Global Scope

The Operating System scheduler can choose to either schedule each process in the system separately from the process' threads (*process scope)*, or it can choose to schedule all threads in the system equally *(global scope)*. Usually, global scope is considered simpler to implement, but can incur higher switching costs due to needing to change out the virtual memory of the process of the thread.

Multilevel Feedback Queuing



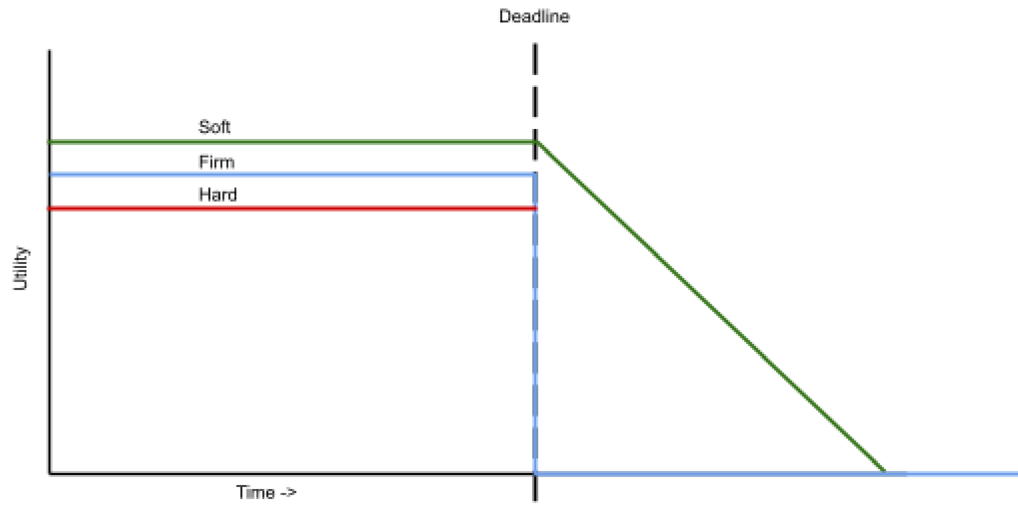
There can be different queues based on different priorities. System tasks may need to be close to real time and have their own queue and scheduler, interactive processes do a lot of waiting and little computation so they may have their own queue, and batch processes that have a lot of information to process can use their own queue so as not to slow down the other processes categories.

Something to look at here is that a process could try and say they are one process to get a shorter queue time or higher priority, so a scheduler may need the ability to "demote" a process to a lower queue. Similarly, for some processes that could do well in another queue, the scheduler might want to "promote" future jobs of that process to a higher queue, like a batch job that behaves more like an interactive process.

Real Time Operating Systems

Soft

In a soft system, there is decreasing utility to miss deadlines. It becomes increasingly important to



Firm

In a firm system, we are trying to minimize misses, and there is no utility for missed tasks. Essentially there is no point to completing a task past the deadline

Hard

In a Hard Real Time Operating System, a missed deadline is a total system failure. Think brakes in your car which fail to fire at a specific time, or a failure of your battery management system in your laptop.

RTOS Scheduling Disciplines

Just like non real-time scheduling, how exactly a real-time scheduler picks which tasks to execute in which quanta can be an important aspect of ensuring it is doing everything correctly.

Earliest Deadline First

In each quanta, we pick the task which is due soonest. It has been mathematically proven that if there is a possible way to schedule all the tasks and have them complete, that this will find it.

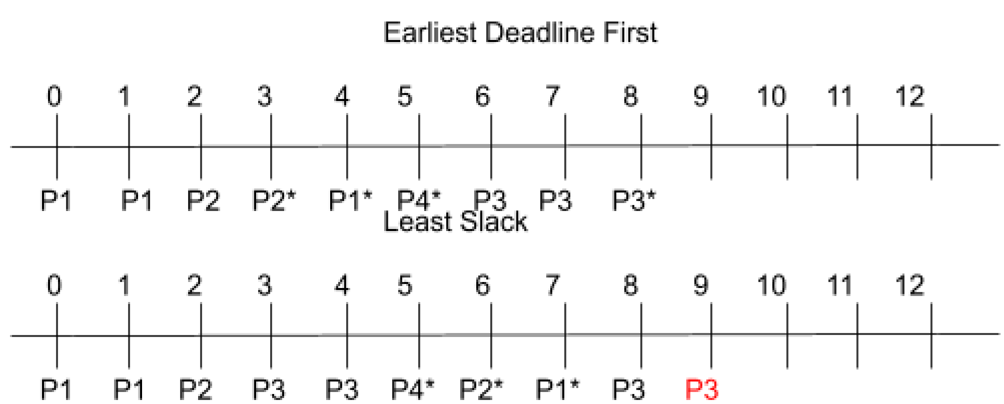
Least Slack

Similar to Earliest Deadline First, this discipline will pick the task with the least slack (that is how many quanta can be spent *not running* this task before it becomes impossible to complete). Slack is calculated by subtracting the due date from the duration.

Rate Monotonic

In situations where you have tasks that have repeated deadlines (i.e. you can complete this task once every 1ms, and this one needs to be completed every 3ms). The priority of each task in a particular quanta is inversely proportional to the duration of the cycle for that process. For example, a task which takes 1ms is higher priority than a task which takes 3ms to complete.

P#	Arrival	Length	Deadline
1	0	3	8
2	2	2	7
3	3	5	9
4	5	1	6



Context Switching

An operating system stores all of the state of your process/thread so that it can be paused and resumed at will. All of this information is stored in the **Process Control Block** or **Thread Control Block**, and enables the operating system to execute multiple contexts concurrently.

Process Control Block – Contains all of the information needed to resume execution of a process at a later point. This information includes:

- ➔ Stack Pointer
- ➔ Program Counter
- ➔ Register Values
- ➔ The status of the memory itself
- ➔ Anything the operating system needs to continue executing

Context Switches

These are all different ways your program can pause execution, so the CPU can handle something else, and each of them represent something different.

System Calls

When a user program wants to interact with the kernel, it uses the standardized System Calls, usually a few hundred different instructions to the kernel. Calling a system call will set the syscall number for the system call on a specific register, and then release control to the kernel to handle it. Doing this usually requires a *context switch*.

Library Call

This is a context switch from the user program to a dynamically linked library, usually libc/libc++. These programs are attached to the runtime, and provide a function interface. An example of a library call is malloc(). This involves a context switch from your program to the library.

Kernel Initiated Traps

When a user program does an illegal operation (invalid memory access, division by zero, etc) the CPU's error handling functionality will automatically jump execution to the kernel to handle the error gracefully (usually by quitting your program).

Interrupt

This is the process by which hardware devices update their state to the software. Possible interrupts can come from network access, input devices, or disk/hard drives. When these devices send interrupts, execution is automatically jumped to the kernel-defined **Interrupt Service Register** to handle the interrupt. Usually, you can mask (disable) some interrupts for better performance in a critical hotpath of code. This obviously comes at the expense of the responsiveness of the system.

Worked Example (Quiz 1)

Which of the following statements are true about interrupt-driven and polling operating systems? Select all that apply.

Interrupt-driven OSes use ISRs to respond to hardware events

This is correct. An interrupt-driven system registers special functions (called Interrupt Service Routines or ISRs) to handle a hardware interrupt.

Polling OSes usually consume more energy
This is correct. Because a polling OS has to continuously check to see if it needs to respond to a hardware event, it will waste more CPU cycles and consume more energy

Polling OSes use ISRs to respond to hardware events
This is incorrect.

They are two terms for the same thing.
This is incorrect.

Interrupts-driven systems are more responsive
This is correct. Because a polling OS has to continuously check to see if it needs to respond to a hardware event, it can only respond to them as fast as it is checking.

Interrupt-driven OSes usually consume more energy
This is incorrect.

Polling is simpler to implement
This is correct. Instead of implementing ISRs, the OS can continuously check to see if a hardware event has occurred, and respond appropriately.

Which statements accurately describe the difference and similarities between a trap and a system call? Select all that apply.functionality

Traps are a switch from user mode to kernel mode
This is correct. A trap can occur for many different reasons, but they involve a switch between execution in kernel space and user space.

When a process requests a privileged action from the kernel, we call that a system call.
This is correct. The POSIX standard defines a series of system calls that a process in user space can make to the kernel. These system calls are requests to the kernel.

A system call is a type of trap.
This is correct. A system call is a type of trap, a switch from user to kernel mode

A trap is when control passes from the user process back to the kernel
*This is correct. A trap refers to both the switch **from** user mode and the switch back to user mode*

System calls are real, traps are imaginary.
This is incorrect. A system call is a type of trap

A trap could occur in response to a system call
This is correct. While this is not the only way a trap can occur, a system call will always result in a trap to handle the request in the kernel.

A trap could occur in response to a hardware event.
This is correct. An example of this would be an illegal arithmetic operation, which would trigger a hardware event leading to a trap.

All traps are system calls.
This is incorrect. Other types of traps include interrupts or library calls.

Memory

The system's memory is a crucial resource that needs to be allocated between many different programs. Initially, in many operating systems, where multiple programs could not run at once, the program was given complete access to the memory (except for the parts reserved by the operating system). As operating systems developed capabilities of running multiple programs at once, the memory needed to be divided to prevent programs from overwriting each other and to isolate processes from each other.

The modern solution to this problem is Virtual Memory, described in detail below. But first, consider some of the more primitive approaches to assigning memory to programs and their potential drawbacks.

The Process of Assigning Memory

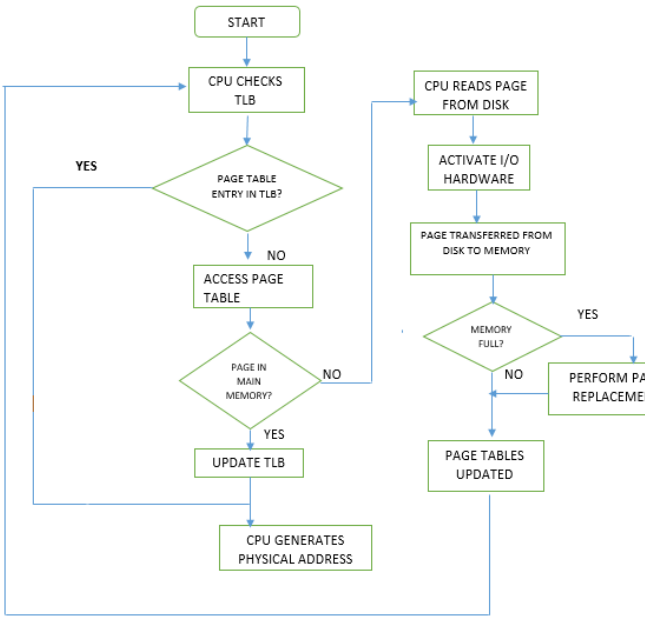
The initial approach one might consider when developing a system to share memory between different programs running concurrently is to divide the memory into sections. For example, the Operating System would obtain a block of memory starting at 0 of a specific size, program #0 would obtain an entirely disjoint block of memory, etc…

Virtual Memory

In Virtual Memory, the Operating System lies to each program to convince them that they have the entirety of the memory space to themselves. So each process has a virtual memory space laid out in the familiar structure. However, when the program requests a specific value in memory, the *virtual address* is translated into the actual frame that is used. Virtual Memory solves two problems: isolation, and abstraction. Each program can have an identical memory layout, without depending on the operating system to grant them a block of addresses. Moreover, the amount of memory granted can expand dynamically at runtime using techniques described later, without needing to reassign entire memory sections of programs. Second, each program's memory is completely isolated from each other, prevent a malicious process from interfering with one running at the same time.

Memory Management Unit

The memory management unit is hardware on the machine responsible for translating a specific virtual memory address (known as a logical address), into an actual hardware address of memory. Part of memory management unit is the **Translation Lookaside Buffer**, a cache which saves commonly accessed translations from virtual to physical memory. If a specific address is not in the TLB, then the CPU must obtain the appropriate address from the page table, as appropriate.



TLB Shootdown

Each core in a CPU maintains its own TLB. When some action invalidates that TLB (most often when a core changes processes), the TLB needs to be regenerated from the page table. **This performance degradation is the main reason why threads typically are more performant than processes, when shared memory is an acceptable solution**. Switching processes requires the CPU to discard the TLB and obtain addresses from the page table.

Segmentation

In segmentation, memory is divided between processes in multiple segments, which can be variable size. This is the largest distinction between segmentation and paging. Every virtual memory address can be divided into the segment address and the offset into that segment. The Memory Management Unit is then responsible for translating the virtual address into an actual hardware address.

Each segment is composed of a different length, so specific address which offset *outside* the maximum allowed size of the segment they refer to could lead to a segmentation fault.

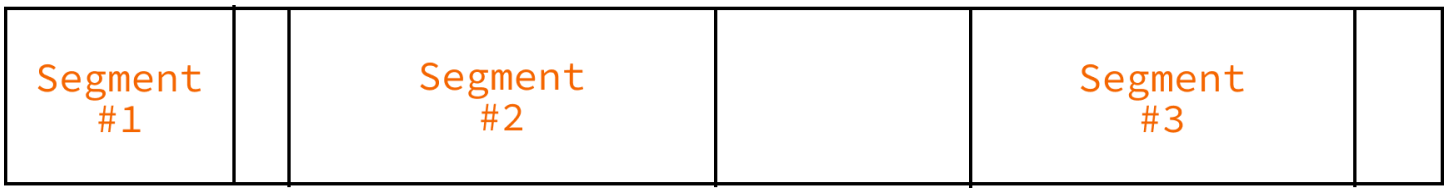


Figure: In segmentation, memory is divided into a series of heterogeneous section, with each segment having specific start and size. New memory is placed between existing segments.

Worked Example (Quiz 7)

I have a system that uses segmentation, uses 16-bit logical addresses and 16-bit physical addresses, and allows each process can have up to 8 memory segments.

```
Index   Start       Segment Size
0       0x0020      0x030
1       --          ---  //unused
2       0x0005      0x012
3       0x0100      0x100
4       --          ---  //unused
5       0x02a0      0x01f
6       0x0300      0x100
7       0xff00      0x0ff
```

1) **Which segment does the logical address 0x4010 map to?**

   Because each process can have up to 8 memory segments, we need 3 bits to address them (because $2^3 = 8$). So the first 3 bits of the address refer to the segment, and the remaining 13 bits refer to the offset inside that segment.

   So, let's convert the logical address to binary to easily divide it.

   `0x4010 = 0100 0000 0001 0000`

   The bits in red refer to the segment, so this logical address maps to 010 = 2.

2) **Which segment does the logical address 0xA009 map to?**

   Similar as above, convert 0xA009 to binary, with red refering to the segment.

   `0xA009 = 1010 0000 0000 1001`

This logical address refers to the segment 101 = 5

3) **What physical addresses do the following logical addresses write do, and do they succeed?**

For this type of problem, identify the segment that the logical address refers to, and offset into the physical address by the logical amount. If the offset exceeds the size of the segment, then this write could cause a segmentation fault.

a) 0x010A

Convert to binary, as above:

`0x010A = 0000 0001 0000 1010` (Segment 0, Offset 266)

So this refers to Segment 0 (000 = 0). By referring to the provided segment table, we can determine that physical addresses in segment 0 start at `0x0020` and the segment has a size of `0x030` (48). Any offsets into this segment larger than the size will trigger a segmentation fault.

`0x0020 + 0x010A = 0x012A`

The offset is `0x10A` (266), which is greater than the segment size, so this write would be a **segmentation fault**.

b) 0x400A

`0x400A = 0100 0000 0000 1010` (Segment 2, Offset 10)

The segment size is `0x012` so we can index into it without a segmentation fault. Therefore, the address is:

`0x0005 + 0x000A = 0x000F`

c) 0xA009

`0xA009 = 1010 0000 0000 1001` (Segment 5, Offset 9)

The segment size is `0x012` so we can index into it without a segmentation fault. Therefore, the address is:

`0x02A0 + 0x000A = 0x02AA`

Segment Allocation Strategy

When you have a number of heterogenous blocks, as in segmentation, you need to determine how to place new blocks within the existing ones. There are several approaches to this, each with their own considerations.

First – First fit scans the memory address spaces starting from the lowest address for the first starting address which has enough space to contain a block of the requested size. This is a simple and quick way to allocate a segment.

Last – Last fit scans the memory address spaces from the highest address for the first starting address which has enough space to contain a block of the requested size.

Best – Best fit searchers the entire segment space for the smallest gap that can properly accommodate the space. This is slower than first fit, but can reduce segmentation. Best fit typically leaves lots of small gaps, with significant external fragmentation.

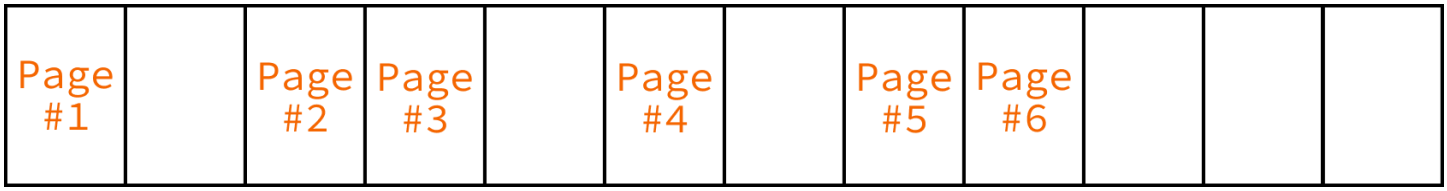
Worst – Worst fit will place the segment into the largest gap that will accommodate it, with the idea that the two divided sections will still be big enough to use in the future.

Random Fit – This strategy will find any random address that accommodates the segment and places it there. This can lead to reduced segmentation.

Paging

To avoid external fragmentation, we can utilizing paging, where we assign processes fixed blocks of memory to processes. Usually, these blocks are around 4096 bytes each, but modern operating systems may choose to assign more. When the process requests memory from the operating system, the OS will allocate a specific frame to the process' page table.

Important: Generally speaking, we use *page* to refer to the block of memory from the perspective of the requesting process (the virtual memory), and *frame* to refer to physical block of memory located on RAM. This is mostly a semantic difference, but can be an important distinction when working in the translation from virtual memory addresses to physical addresses.



Page Table Flags

In addition to the specific frame for a given logical address, the page table also stores a bit field of flags containing information about that specific page. *Typically, a hardware interrupt will reset the dirty and reference flags on a regular interval.*

Valid

Addresses which are not used are marked with a valid bit 0. For example, in programs which do not use the entire address space (read: almost all of them), addresses which have not yet been allocated would have the valid bit set to 0.

Resident

If the resident bit is set to 0, then that specific page does not have a frame in physical memory, usually because that paged was swapped out to disk. Accessing a page which is not resident is a page fault.

Dirty

The dirty bit has been set by the hardware when the page has been written to in the last cycle. The dirty bit indicates that a block of memory has been modified, but has not been saved to secondary storage yet.

Reference

The reference bit is set to 1 by the hardware when the page has been accessed in the last cycle. It is often used for determining which pages to swap to disk.

Multi-Level Page Tables

Often times, storing the entire page table in memory can be incredibly difficult, especially as the number of pages increases. To solve this problem, computer scientists introduced another level of indirection: the multiple-level page table.

In a system with multiple levels of paging, only the main 1st level page is stored in memory initially. Instead of maps to specific frames, the 1st level page table maps to a 2nd level page table, which maps to frames.

For example, consider a system with 32-bit physical and logical addresses, and 4096-byte pages. We need 12 bits to represent the offset inside of the page, so the remaining 20 bits are reserved for the page table.

In a single-level page table system, a page table with a 20 bit index would require up to 2^{20} = 1,048,576 entries. If each entry is 3 bytes (20 bit physical address, and 4 bits to indicate page flags), a single page table would be 3 MB, just for a single process.

`0000 0000 0000 0000 0000 0000 0000 0000` (first 20 bits represent page number, last 12 bits represent offset)

In a two-level page system, the 20 bits are divided into 2 sections for each page, maybe 8 bits for the first level page table (which is always stored in memory), and 12 bits for the second level page table. The second level page table is the table that actually contains the physical frame and page bits. See below for more information about accessing pages in a multi-level system. Linux typically uses a three-level paging system.

Obviously, multi-level page tables require the processor to load the secondary page tables into memory before it can determine the physical frame, which can lead to performance implications when loading addresses not in the TLB.

Worked Example (Quiz 8)

I have a system that uses two-level paging, with 12-bit logical addresses, 16-bit physical addresses, 256-byte pages, and 4 entries in each first-level page table.

If this is my first-level page table...

```
Index   Address
0       0x0050
1       --  //unmapped
2       0x0100
3       0x0200
```

...and the second-level table at address 0x0100 starts with...

```
Index    Frame
0        0x10
1        0x30
2        0xAA
...
```

...and the second-level table at address 0x0200 starts with...

```
Index    Frame
0        0xB0
```

```
1         0xCC
2         0x0D
...
```

1) **How many entries does each second-level page table have?**

   Divide the bits of a logical address into their constituent parts. We need 8 bits to store the page offset (because pages are 256 bytes, and $2^8 = 256$), and 2 bits to store the first level page table (4 entries = 2^2), so the remaining 2 bits can be used for the second level page table.

   `0000 0000 0000`

   So, because 2 bits are used for the second level page table, there are **4 entries in the 2nd level page table**

2) **What physical address does 0x60A map to?**

   `0x60A = 0110 0000 1010` (1st Level Page: 1, 2nd Level Page: 2, Offset: 10)

   Because the 1st entry of the 1st level page is unused, this is a **segmentation fault**

3) **What physical address does 0x817 map to?**

   `0x817 = 1000 0001 0111` (1st Level Page: 2, 2nd Level Page: 0, Offset: 7)

   The first level page table maps to the page table at `0x0100`, which we have a page table for. The second level address has the frame 0x10 at entry 0, so offset that by the page offset to get

   `0x1000 + 0x17 = 0x1017`

4) **What physical address does 0xEAA map to?**

   `0xEAA = 1110 1010 1010` (1st Level Page: 3, 2nd Level Page: 2, Offset: 170)

   The first level page table maps to the page table at address **0x0200**. The entry at index 2 for that page table is the frame `0x0D` so:

   `0x0D00 + 0xAA = 0x0DAA`

5) **What physical address does 0x3D5 map to?**

   `0x3D5 = 0011 1101 0101` (1st Level Page: 0, 2nd Level Page: 3, Offset: 213)

   The first level page table maps to the page table at address **0x0050**. We don't know have the page table at that address, so this **cannot be determined**

Worked Example (Quiz 9)
Two-level paging with 16-bit logical addresses, 16-bit physical addresses, 256-byte pages, and 8 entries in each first-level page table

If this is my first-level page table (not all entries shown)

```
Index    Address
0        0x0050
1        0x0400
2        --  //unmapped
3        0x0600
```

...and the second-level table at address 0x0400 starts with...

```
Index     Frame
0         0x10
1         0x30
2         0xAA
```

...and the second-level table at address 0x0600 starts with...

```
Index     Frame
0         0xB0
1         0xCC
2         0x0D
```

1) **How many entries does each 2nd-level page table have?**

   With 8 entries in the 1st level page table, we need 3 bits. We need 8 bits to store the page offset, so the remaining 5 bits are allocated to the 2nd level page table, giving us **32 entries** ($2^5 = 32$)

   `0000 0000 0000 0000`

2) **What physical address does 0x020A map to?**

   `0x020A = 0000 0010 0000 1010` (1st Level Page: 0, 2nd Level Page: 2, Offset: 10)

   The first level page table maps to the page table at address **0x0050**. We don't know have the page table at that address, so this **cannot be determined.**

3) **What physical address does 0x4217 map to?**

   `0x4217 = 0100 0010 0001 0111` (1st Level Page: 2, 2nd Level Page: 2, Offset: 23)

   The first level page table maps to the an unused 2nd level page table, so access this address would be a **segmentation fault.**

4) **What physical address does 0x10AA map to?**

   `0x10AA = 0001 0000 1010 1010` (1st Level Page: 0, 2nd Level Page: 16, Offset: 170)

   The first level page table maps to the page table at address **0x0050**. We don't know have the page table at that address, so this **cannot be determined.**

5) **What physical address does 0x61D5 map to?**

   `0x61D5 = 0110 0001 1101 0101` (1st Level Page: 3, 2nd Level Page: 1, Offset: 213)

The first level page table maps to the page table at address **0x0600**. Indexing into that page table, we get a frame of **0xCC**, so:

`0xCC00 + 0xD5 = **0xCCD5**`

Worked Example (Quiz 10)

I have a system that uses three-level paging, with 16-bit logical addresses, 256-byte pages, and 4 entries in each first-level page table, and 8 entries in each 2nd-level page table.

1) **How many entries are in each 3rd-level page table?**

   We need 8 bits to represent the offset into 256 byte page. The first level page table has 4 entries, so needs 2 bits. The second level page table has 8 entries, so needs 3 bits. So the final 3 bits are dedicated to the third level page table, meaning there can be **8 entries**.

   `0000 0000 0000 0000`

2) **In the same scenario as the previous question, if what index would you use in the 3rd-level page table for logical address, 0x46F2?**

   `0x46F2 = 0100 0110 111 0010`

   In this address, the 3rd level page table has an **index of 6**

3) **What will happen when I access a page with the following bits set.**

   valid = 1
   resident = 0
   dirty = 0
   reference = 0

   Because the memory is not resident in the system's memory, this would be a **page fault**. The operating system would need to obtain the block of memory from swap. This would be a context switch to kernel mode.

Page Replacement

One of the advantages of virtual memory is the ability to create abstractions that use main memory as their representation. One of these is Memory Mapped I/O, which is discussed later, but the primary focus of this section will be enabling swap. Page replacement occurs when the system encounters a page fault (a process requests memory that is not in main memory), so we need to *swap* one of the existing pages in main memory with the page currently in swap. This is obviously very slow, but allows us time to fix the issue (by stopping processes and freeing memory). When it is time to determine which pages to swap to secondary storage, we can use the reference and dirty bits to see what pages have been accessed recently, and which ones can be swapped to storage.

Page Replacement Algorithms

To determine what pages are swapped out to secondary storage, and which are kept in main memory. Some terms that can help us understand the language:

Resident Page – A page is *resident* if it is stored in main memory, not swap. This is usually indicated by the resident bit in the page flags.

Page Fault – When a process requests a page of memory that is not in main memory.

Victim – The page that gets swapped out into secondary storage to accommodate another page. The process of removing this page is known as *eviction.*

Demand Paging – A lazy-loading method of bringing pages into memory. Pages are only swapped into memory from secondary storage if there is a page fault.

Thrashing – Repeatedly swapping memory to and from disk. A machine is thrashing when it is encountering repeated page faults, needing to swap memory frequently. This can absolutely cripple performance, and prevent anything from running efficiently at all. We should design our page replacement algorithms to minimize thrashing

Local vs. Global

In Local Page Replacement, when a new page needs to be brought into main memory, the page replacement algorithm will only select pages that are owned by the particular process. In Global Page Replacement, the algorithm may select any page currently in main memory. Generally speaking, global page replacement is more efficient on a system-wide basis, but local page replacement leads to more consistent performance for processes. One program that consumes a lot of memory will have more of an impact on other processes in a system with global page replacement, as other processes' pages could get swapped to secondary storage to accommodate the more hungry process.

First In, First Out

In the FIFO algorithm, the operating system keeps a queue of pages as they are created. When a swap needs to be performed, the OS will chose the oldest one to swap. *Note:* in local page replacement, this will be the oldest page that is owned by the particular process.

Not Recently Used

This algorithm takes advantage of the reference and dirty bits. When a page needs to be replaced, it will group all relevant pages into 4 categories:

4. referenced, modified
3. referenced, not modified
2. not referenced, modified
1. not referenced, not modified

The OS will first try to evict a page which has not been referenced or modified since the last interrupt cycle, continuing up the priority list until it must evict a page which has been referenced and modified as a final contingency. Not Recently Used improves on FIFO by attempting to keep commonly used pages in memory longer.

Second Chance

In second chance, the system will maintain a queue of pages, as in FIFO. The system will pull the first page from the queue, and check if the reference bit is set (indicating if it has been read since the last clock cycle). If the page has been

referenced, then it will be added to the back of the queue, otherwise it will be paged out to disk. Second chance typically performs better than First In, First Out to prevent thrashing, as it is less likely that key pages will be swapped to disk. This can be similarly implemented using *Clock Page Replacement*, which operates very similarly.

Least Recently Used

In LRU page replacement, the system maintains a LRU cache, a data structure that keeps track of the pages which have been used the least in recent past. The OS selects pages from the LRU when it needs to evict a page. Performance for the LRU is very good, but can incur significant costs to store and maintain the data structure.

Copy-on-Write

When you fork a process, you create a duplicate of the original process's memory, but a lot of that memory is never changed (for example, loaded libraries, stack variables, globals). Memory that is never modified doesn't *strictly* need to be fully duplicated in RAM. So instead, when forking, an operating system will maintain the same page table (allowing the process to read the same physical frames). When either process writes to a page, then that page gets marked for duplication by setting the dirty bit.

Worked Example (Quiz 12)

1) A Linux process has 8 valid pages in its page table (all are resident). If it calls fork, and the child process reads from 4 of those pages and writes to 3 of them, **how many total physical frames will the two processes need combined (assume everything is still resident and no other memory accesses occurred)?**

   Only 11 physical frames would be required. Forked processes will retain the same physical frames in memory until they write to them, as a memory saving optimization.

Fragmentation

When you have these heterogeneous sized segments dividing your memory, you will have a number of small gaps that are ultimately unusable.

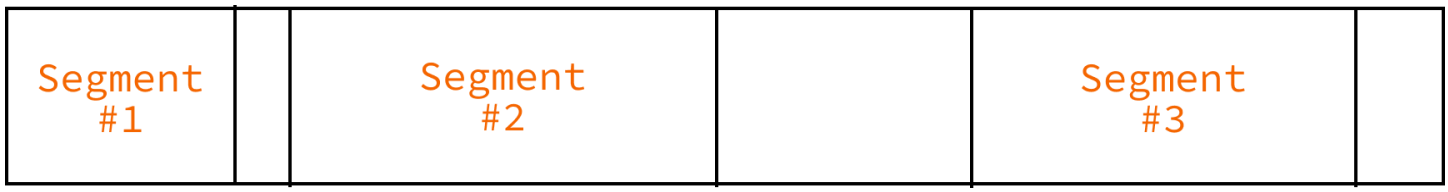


Figure: the small gaps between segments may be too small to be usable.

Fragmentation in the context of memory segmentation is usually considered to be external fragmentation. Paged memory cannot be fragmented externally, but the large size of blocks that need to be allocated in the paged memory systems can lead to internal fragmentation.

Note that internal and external fragmentation are kind of nebulous, and depend on your perspective. For example, consider a system with paged memory with blocks of 4096. Now further consider writing an allocator for this system that works in a segment style (explicit free list). The gaps in the this allocator are *external* to the individual blocks of memory returned by the allocator, but internal to the Operating System, as there is no gap for the pages to fill.

Allocators

The operating system has means of us being able to obtain virtual memory (see and example of mmap below), but usually it only works with larger blocks of memory (of say 4096 bytes). Most allocations do not need this much memory, so requesting a page from memory to store a small array can lead to significant internal fragmentation. To solve this problem we create an *allocator*, a library that can request this memory for us, and then divide it into multiple allocations to reuse.

```
int main() {
  void *page = (void *)mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE,
                            MAP_PRIVATE, ZERO_FD, 0);
  printf("%p", page);
}
```

Figure: request a page of memory from the operating system, using mmap.

The allocator will use the mmap, brk, and sbrk system calls to obtain and release memory to the operating system. brk() allows you to move the program break (the line between the process code and the heap, to allow you to write to those pages).

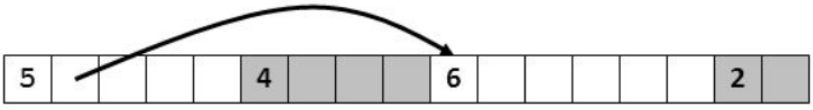
There are several different ways of programming allocators, described below. Much of the logic here follows from the discussion on segmentation vs. paging, as we are essentially performing the same function here, albeit on a smaller scale.

Implicit Free List

In an implicit list, the allocation affixes a prefix to the specific allocation with some metadata, including a size of the allocation, and whether the block has been freed or not. Note that blocks of different sizes are held in the same contiguous space. The process of looking for a free block involves iterating through the memory to find a block of memory that is large enough to accommodate the requested space and has been marked as freed.

Explicit Free List

In an explicit free list, each allocation has a prefix that points to the next free block. Then, every page begins with a small header containing a pointer to the first free block. When a block is freed, it can be added to the free list, and then the process of finding a block is iterating through the free list to find a block large enough for our needs.



Segregated Free List

A segregated free list works very similarly to an explicit free list, except that each page has only a single size of allocation (usually a power of 2). This was the method we used for Project 3.

Memory Mapped I/O

In memory mapped I/O, the operating system defines specific addresses in memory as ways to access certain I/O devices. For example, you can use memory mapped I/O to map a file (a portion of the disk) into "system memory" and be able to access that file like you would any other virtual memory. This can allow us to do file manipulations faster than we would by doing read() and write(), because we do not need to switch into kernel mode to execute those operations.

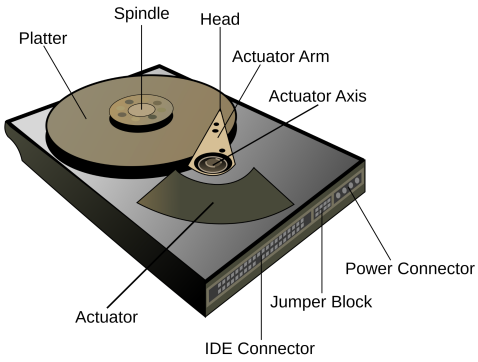
For example, take a look at this example of mapping a file into memory using mmap:

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
  FILE *handle = fopen("file.txt", "rw");
  struct stat filestat;

  if (fstat(handle, &filestat) != 0) {
    perror("stat failed");
    exit(1);
  }

  char *data;

  data = (char *)mmap(NULL, filestat.st_size, PROT_READ, MAP_SHARED, handle, 0);
  if (data == MAP_FAILED) {
    perror("mmap failed");
    exit(2);
  }
}
```

Disks

Anatomy of a Disk

A hard disk is composed of a stack of platters (divided into a series of *cylinders*), which is read by a read head. Because you physically need to move a read head and rotate the platter to read data, sequential accesses are much more efficient than random access.



Boot Sectors

The boot sector is the first sector in a File System, and usually contains information about the disk as a whole, as well as basic code that can be used to start the OS. The exact nature and details of the boot sector is beyond the scope of this course. The exact size of the boot sector is dependent on the file system the disk is formatted.

Scheduling Methods

FCFS - First Come First Serve

In this scheduling method, disk read/write requests are fulfilled in the order they come in, no matter the track. This method prevents any request from going unfulfilled and starved.

SSTF - Shortest Seek Time First

In this scheduling method, disk requests are prioritized based on the requests' seek time. Because of this, long requests could get delayed continuously, causing a process to continuously wait.

SCAN

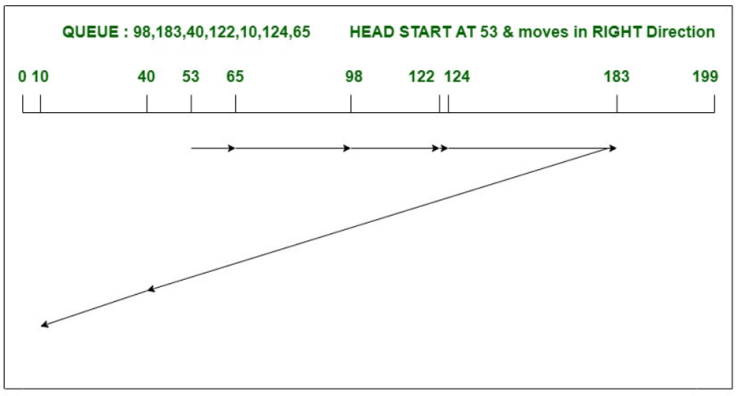
In this scheduling method, requests are fulfilled as the read head iterates sequentially from one end of the disk to the other. When it hits the end, the read head changes direction and moves towards the other end, continuing to scan and fulfill requests when it reaches a requested track.

C-SCAN (Circular Scan)

This method is very similar to SCAN, with the only exception being that the head only fulfills requests scanning one way. When it hits the end of the disk, the read head will jump back to the other side and scan the length of the disk again. This method has better and more uniform response times across the disk

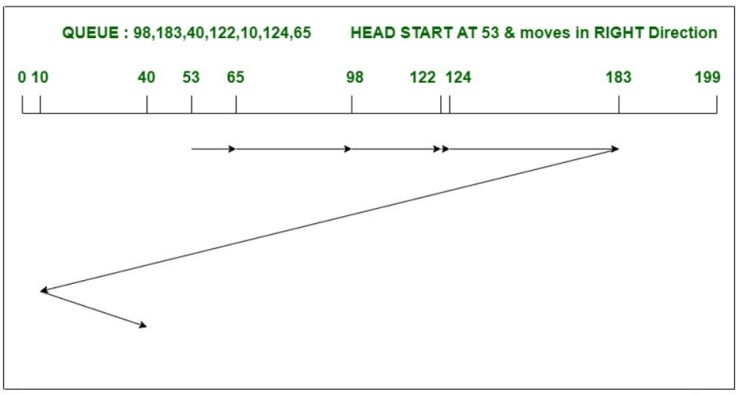
LOOK

In this scheduling method, requests are fulfilled by numerical order as the head moves along the track to the request with the largest track, then switching directs and doing the same as it moves to the last request and fulfills all requests for tracks in between.



C-LOOK (Circular Look)

This method is very similar to LOOK, with the difference being that the head only fulfills requests when it moves in a specific direction. Once it hits the last request in the direction it's going, it will jump to the first request on the opposite end and continue going the same direction as before, fulfilling all requests in between.



Worked Example (Quiz 12)

1) A magnetic hard disk drive has 100 total cylinders (numbered 0 to 99) and 5 current requests for the following cylinders in the following order (22, 7, 1, 15, 6). **If the disk head starts at cylinder 17, how far (in cylinders) will my disk have to seek to handle these requests, if I'm using LOOK?** (Compute total seek distance. If the algorithm needs an initial direction, start moving toward cylinder 99.)

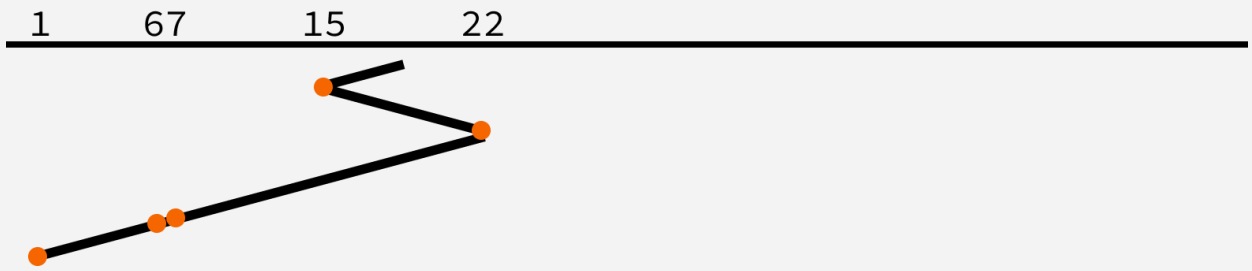
   Hint: if I start at cylinder 4 and seek to cylinder 7, the head has moved 3 cylinders.

   In LOOK, the disk head moves from one direction to the end, and then retreats back to the beginning, reading cylinders in both directions. So, in this situation we would visit the cylinders in the following order:

   17, 22, 15, 7, 6, 1

   Therefore, the head will travel (22 - 17) + (22 - 15) + (15 - 7) + (7 - 6) + (6 - 1) = **26 cylinders**

2) **What would the answer to the previous question be if I used SSTF, instead?**



   In SSTF, the head moves to the cylinder in the queue that it would currently be the least travel time to get to, so the head visits the cylinders in the following order:

   17, 15, 22, 7, 6, 1

   Therefore, the head will travel (17 - 15) + (22 - 15) + (22 - 7) + (7 - 6) + (6 - 1) = **30 cylinders**

File Systems

Historical Context

To understand how and why most file systems were designed, we need to consider the technological environment in which many of these file systems developed. Most devices today use *Solid State Drives*, which have mostly equivalent sequential reads and random reads. However, for most of computing history, hard drives with significantly different performance characteristics existed. To understand this further, consider the composition of a hard drive.

FAT File Systems

One of the earliest file systems to develop was FAT, or the File Allocation Table. In FAT, the disk is divided into a number of *sectors*, same sized blocks that the file system is divided into. For example, FAT12 typically uses 512 byte sectors. The structure of a FAT12 file system is shown below:

Boot Sector

The boot sector is the first sector in a File System, and usually contains information about the disk as a whole, as well as basic code that can be used to start the BIOS. The exact nature and details of the boot sector is beyond the scope of this course.

File Allocation Table

The File Allocation Table provides a mapping of how sectors are assigned in the data section of the disk. The number in the FATXX is the number of bits of each entry in the FAT table. For example, FAT12 is composed of 12 bit entries, FAT16 is composed of 16 bit entries, etc.

Decoding FAT12 File Allocation Tables

FAT12 is composed of 12 bits, which does not fit neatly into an integer number of bits, so instead three bytes come together to describe 2 successive entries. The structure of the two entries in the 3 bits is shown below

`F0 FF FF → FF0 FFF`

For more information on decoding the FAT, see the Project 4 description.

Each entry in the FAT table corresponds to a logical cluster in the data section. The first 2 entries begin with the end of file marker and the media descriptor, so entries beginning with 2 refer.

Root Directory

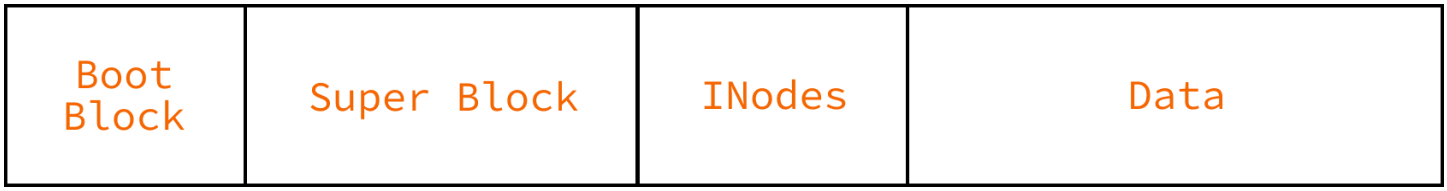
In FAT12 and FAT16, the root directory is a specific section of the disk. In these systems, a directory listing for the root directory is located here. Entries are 32 bytes, and have the following format (shown below is the format for FAT12 entries)

Data Sectors

The remaining sectors in the disk are dedicated to data sectors, i.e. the contents of the files. What file each sector is allocated to is described in the FAT.

Unix FS

The Unix FS is the basis for most modern file systems.



Boot Block

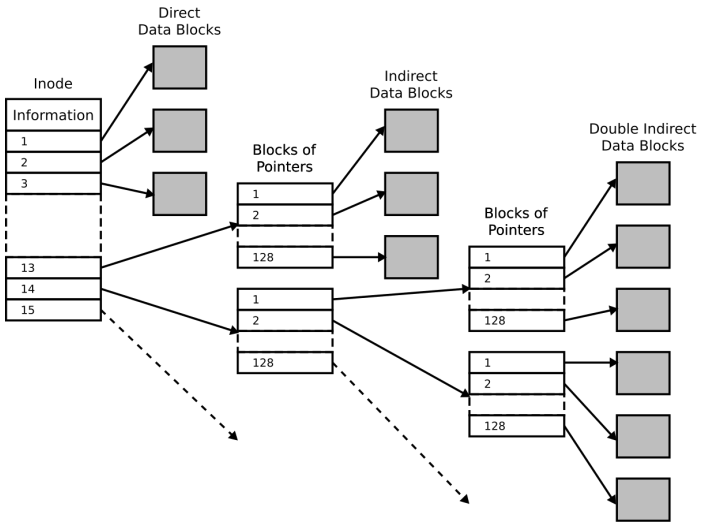
The first sector of the file system, similar to FAT. The Boot Block contains information needed to start running the operating system.

Superblock

Contains metadata about the file system as a whole, including the state, the block size (originally 512 byte blocks, now typically 4096 bytes), a pointer to a list of free blocks, and the INode of the root directory.

INodes

A large array of INode entries, each of which has an index, and represents a specific file or directory in the file system. The INode contains the metadata about the file including: the type, whether its a named pipe, whether it if it is a symlink, the file size, permissions, flags, access and modification times, and the number of links that have been created (used for hard links).



Importantly, the INode also contains pointers to data blocks, which contain the contents of the file. Typically, there are 12 direct pointers to blocks in the data section.

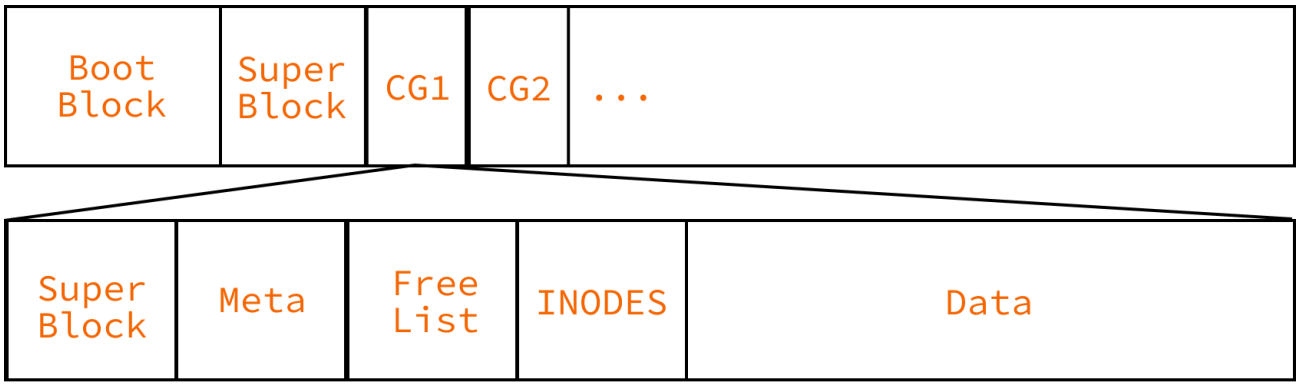
These direct pointers allow us to store upto 48Kb.

Indirect Blocks

If a file needs more than 12 blocks, there are additional indirect pointers attached to the inode. These indirect pointers can point to blocks of memory which are entirely dedicated to storing more pointers to blocks. For a single level of indirect blocks, this gives us an additional 1024 pointers, allowing us to store files up to 4 MB. Double indirect blocks would allow us to store files up to 4 GB, and triple indirect (typically the maximum) allw us to store files of up to 4 TB, which is a practical maximum for most situations.

Berkeley Fast File System

The Berkley Fast File System builds upon the Unix FS with some additional performance optimizations.



Cylinder Groups

The Berkeley Fast File System partitions the disk into a number of cylinder groups, each of which has a copy of the superblock, some metadata about the cylinder group, a number of INodes, and a data section. The intention is to keep the Inodes and data of a particular part of the file system to reduce the seeks needed to read a file. Part of the optimizations around cylinder groups was putting files into groups to keep them together, and to spread the number of writes more evenly across the disk.

Fragments

BFFS also introduced a concept called fragments, where you could intentionally divide blocks into smaller sections on the disk, to save space as the expense of speed. This never really took off within the community.

Bigger Block Sizes

BFFS also recommended moving to larger block sizes, closer to 8Kb. This has happened in some modern variants, but many operating systems find 4 Kb to be a happy medium.

Log Structured File Systems

A log structured file system is predicated on the idea that writes should be optimized, as slower reads can be compensated by keeping large portions of the disk in memory. As such, a log structured file system will try to make writes as sequential as possible, rewriting its internal datastructure to allow this. All files and metadata are written to a circular buffer. The boot block and superblock behave very similarly to BFFS.

Log

The log contains all file data, INodes, and INode Map Sections (portions of a map that maps INodes to their location in the log). Whenever a file is updated, a new entry is written to the back of the log, the INode is updated, and the INode Map section is updated to reflect the new INode location.

Checkpoint Region

The checkpoint region contains a series of pointers that allow us to access the various sections of the INode Map quickly.

Worked Example (Quiz 13)

1) **Which of the following would be valid (sensible) first entries for a FAT12 disk's FAT? Note, these are entries, not individual bytes. (They're already decoded.)**

   `0xff0 0xfff 0x003 0x004 0x005 0xfff 0x001 …`
   This is incorrect, the value 0x001 should never occur in a FAT table.

   `0xff0 0xfff 0xfff 0xfff 0xfff 0x000 0xfff ...`
   This is a sensible beginning to a FAT12 FAT table.

   `0xff0 0xfff 0x004 0x002 0x003 0x000 0x000 ...`
   This is not a sensible FAT table, as the first 3 entries create an infinite loop. The entry at index 2 points to index 4, the entry at index 4 points to index 3, and the entry at index 3 points to index 2.

   `0xff0 0xfff 0x004 0x004 0x000 0x000 0x000 ...`
   This is not a sensible fat table, as multiple entries point to 0x004.

   `0xff0 0xfff 0x003 0x004 0x005 0xfff 0xfff ...`
   This is a sensible beginning to a FAT12 FAT table.

   `0xff0 0xfff 0xfff 0x005 0x003 0xfff 0x000 ...`
   This is a sensible beginning to a FAT12 FAT table.

2) **Which of the following are innovations introduced by the Berkeley Fast File System to make the file system faster? Select all that apply.**

   INodes
   This was introduced as a concept in the original Linux file system.

   Triple Indirect Blocks
   This was introduced in the Berkley Fast File System, but as a way to increase the maximum size of files that can be stored, not as a performance optimization.

   Directory Entries
   Directory entries were introduced in earlier file systems.

   Double Indirect Blocks
   These were a feature of the earlier UNIX FS system.

   Direct blocks
   These were a feature of the earlier UNIX FS system.

   Larger block sizes
   This is correct. The Berkley Fast File system raised the block size from 512 bytes to 4096 bytes. This was possible due to increased RAM available at the time, and enabled fewer seeks at faster accesses at the expense of more memory/space waste. BFFS also introduce the concept of block fragments to try and address the space waste issue, but they remained largely unused.

   Data block fragments
   BFFS also introduce the concept of block fragments to try and address the space waste issue, but they remained largely unused.

Indirect blocks
These were a feature of the earlier UNIX FS system.

File/Directory Placement Heuristics
This is correct. The BFFS introduced heuristics to place directories and their contents close on the disk, so as to reduce the seek time when traversing the file system.

Cylinder Groups
This is correct. The BFFS introduced Cylinder Groups, which divided the disk into a series of smaller disks. The intention of cylinder groups is to keep commonly accessed files together, so as to reduce seek time, another innovation that provided initial speedups but became less relevant as disks got larger and solid state drives were introduced.

Super Blocks
This is incorrect, the superblock was introduced in earlier UNIX FS system.

Metadata
This is incorrect, metadata (stored in the INodes), was introduced in earlier systems.

3) Say I have a Unix File System, with 4kB-sized blocks, and only direct and single indirect blocks (no double or triple indirect blocks). **Assuming that each inode has 12 direct pointers and two indirect pointers and that pointers are 4 bytes, what is the largest size a file can be on this file system?**

The 12 direct pointers can each point to a block of 4096 bytes, so **49 152 bytes**

A single indirect pointer can point to an indirect block with 1024 pointers (4096 / 4), each of which can point to a block of 4096 bytes, so a single indirect block can add an additional **4 194 304 bytes**.

Because there are two indirect pointers, the maximum size of a file is **49,152 + (2 * 4,194,304) = 8,437,760 bytes** (or about **8.44 MB**)

Disk Fragmentation

As we have seen from each of the file systems, as time goes on, the placement of the contents of a file can vary widely across the disk. In moving hard drives, these nonsequential reads can kill performance. When the contents of files/directories are strewn wildly across the disk, the disk has *fragmentation*. An overly fragmented disk can lead to very slow read and write times due to needing to seek the head. To solve this program, you can run a *defrag program*, which rearranges the contents of your disk to be more continuous, leading to more sequential reads, and lower seek time.

The Berkely Fast File System's concept of cylinder groups would help to partition the disk, keeping files together more, which can reduce disk fragmentation. Modern file systems have other means of partitioning the disk to minimize fragmentation, but it is less important now due to the rise of solid state drives.

RAID

Being able to recover from data loss is very important to both consumers and businesses. Using a Redundant Array of Independent Disks (RAID; the I used to stand for Inexpensive) allows an unobtrusive way to prevent data loss, reduce recovery time, and even increase read and write speeds.

Striping

Striping is the process of sequentially spreading information being written across multiple disks. This can be done either per bit, byte, or block. Striping may increase both read and write speeds because the disks are able to write and read data independently of each other.

Mirroring

Mirroring is the process of writing multiple copies of the data being sent to the disk across multiple disks. The disks essentially become a carbon copy of each other. Mirroring allows

Parity

Parity is extra data that is calculated and stored alongside the user's data. This data can be used to recover information if a drive fails, as well as to verify the integrity of the stored data.

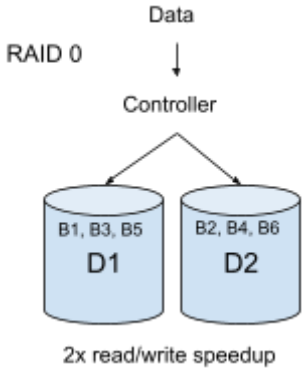
Parity can either be even or odd and is calculated based on if there needs to be a 1 added to make the number of 1 bits even or odd.

				Parity	
D1	D2	D3	D4	P	
1	0	1	1	0	Odd Parity
				1	Even Parity

RAID Levels

RAID 0

RAID 0 is made up of n disks that are striped, but not mirrored or stored with a parity bit. This allows up to a theoretical n times read and write speed increase with n disks, but no redundancy. For this reason, RAID 0 is not *quite* RAID.



RAID 1

RAID 1 is made up of n disks that are mirrored, but not striped or stored with a parity bit. This allows theoretically up to n times read speed increase since the disks can read back information independently of each other, but no increase in write speed. Even though there is no parity bit, mirroring allows for a drive to fail, as long as there is at least one other drive in the system since each drive contains the same data.

RAID 2

RAID 2 uses bit-level striping to spread the data across multiple drives and mirroring data using multiple drives for backup. *This RAID level is not used in practice.*

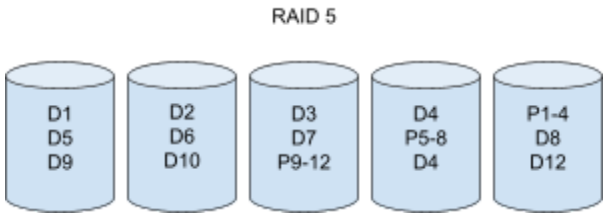
RAID 3

This RAID level uses byte-level striping and odd or even parity to recover data incase of drive failure with a dedicated parity drive. The spindles are synchronized since blocks are spread across multiple disks and can bring a speed increase for sequential reads and writes.

RAID 4

This RAID level uses block-level striping and a dedicated parity disk. This level has good performance for random reads.

RAID 5



This RAID level is similar to RAID 4, but instead of a dedicated parity disk, it uses distributed parity. *This RAID level is one of the most commonly used levels.* Distributing the parity among all the disks helps distribute the wear, since usually a dedicated parity disk would be the first to fail due to the constant writing and reading from it.

RAID 6

This RAID level is like RAID 5 but with double distributed parity, and it uses Reed Soloman codes that allow 2 disks to be lost. *This RAID level is one of the most commonly used levels.*

Note: This does not protect against correlated failures of multiple drives instantaneously