CPSC 3220 Midterm Exam Study Guide

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

Disclaimer: while I make every 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 today! ❤️

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 core of the operating system is the kernel, which manages the system's resources. 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

Which of the following statements about microkernels are true?

A microkernel probably has less code than the 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

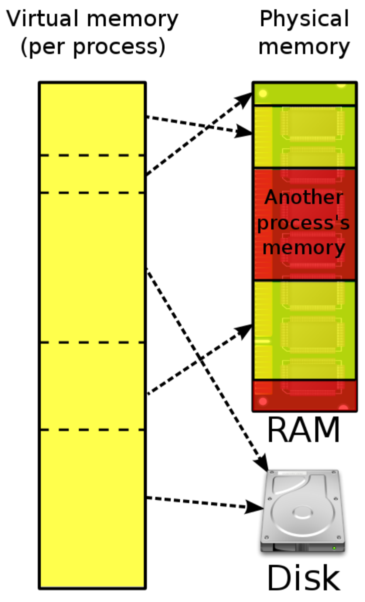
A primary job of an operating system is managing these different processes. Process and threads enable *parallelism* and *concurrency,* two distinct but related terms.

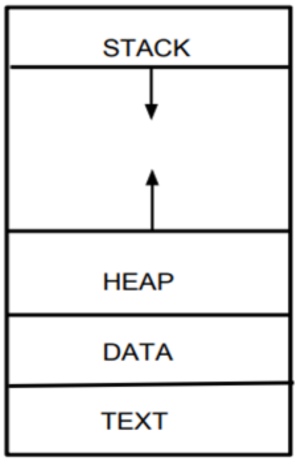
Parallelism – In a multi-core system, simultaneously executing several instructions at once by dividing the work between your cores.

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 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.

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);
    }
}
```

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.

```c
// 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.

```c
 // 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*.

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.

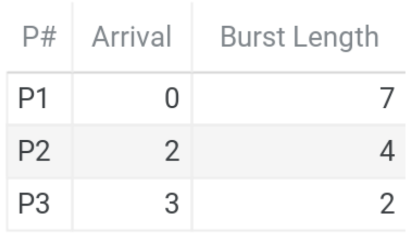
Wait Time – The time from work becoming ready and the first point it begins execution
Response Time/Turnaround Time – The time from work becoming ready until it is completed

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.

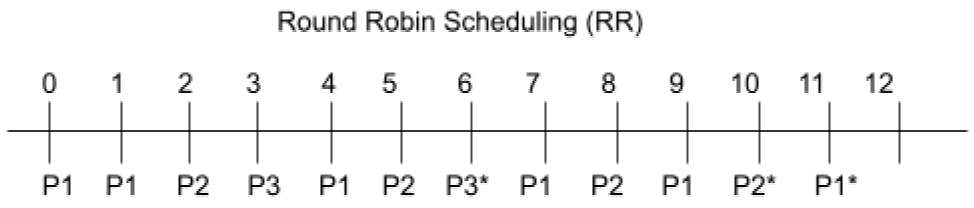
Priority Queue – Cooperative
There is a priority queue of tasks to be performed, with each task being assigned a priority. Processes with lower priority can have extremely long lead-times if higher priority tasks consistently preempt them.

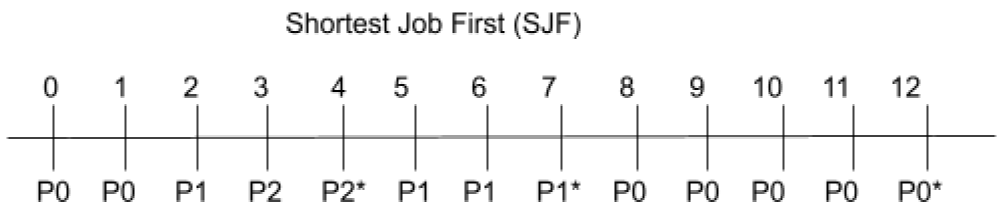
Round Robin (RR) – Preemptive
The scheduler will switch processes every quantum and move to the next process in the queue.

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


Round Robin Scheduling (RR)
0 1 2 3 4 5 6 7 8 9 10 11 12
P1 P1 P2 P3 P1 P2 P3* P1 P2 P1 P2* P1*



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.

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*	$p_n = (0.3)(4) + (0.7)(4) = 4$

So, shortest job first will select

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.

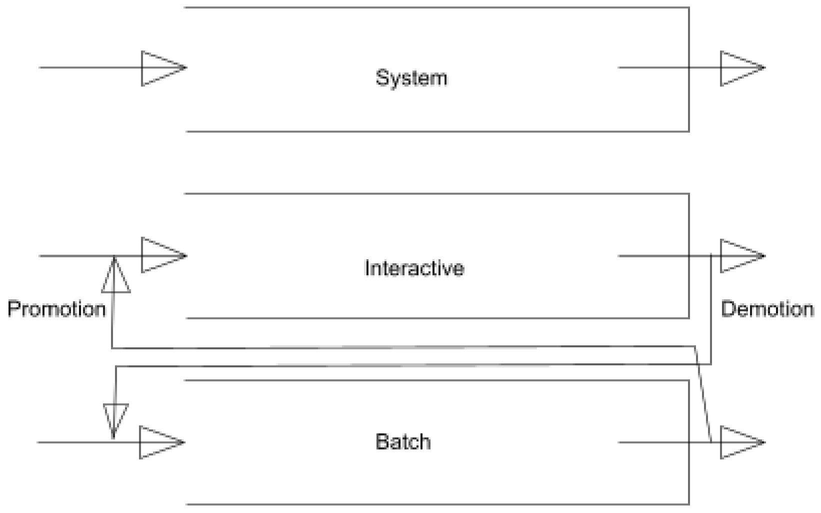
Short Term Scheduler – Determines which tasks get run from quantum to quantum.

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



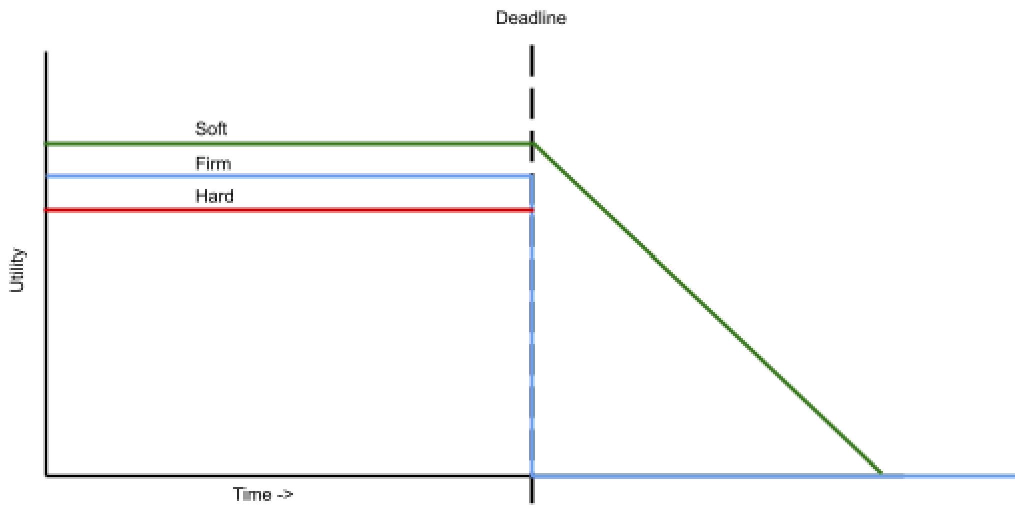
There can be different queues based on different priorities. System tasks may need to be close to real time and have their own queue and scheduler, interactive processes do a lot of waiting and little computation so they may have their own queue, and batch processes that have a lot of information to 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.

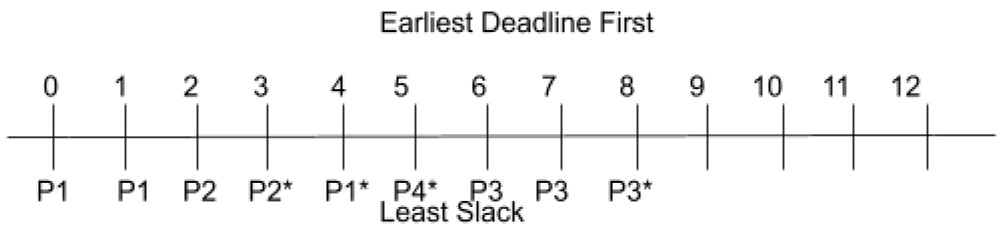
RTOS Scheduling Disciplines
Just like non real-time scheduling, how exactly a real-time scheduler picks which tasks to execute in which quanta can be an important aspect of ensuring it is doing everything correctly.

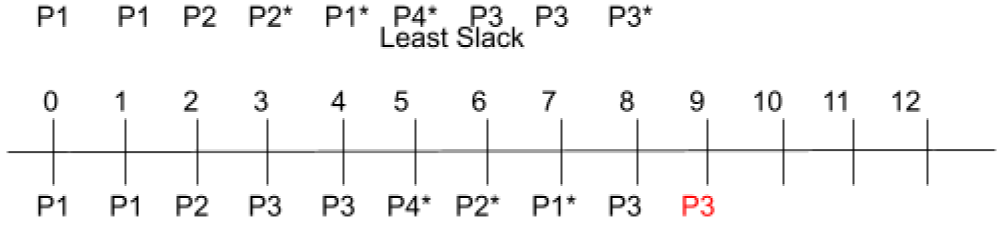
Earliest Deadline First
In each quanta, we pick the task which is due soonest. It has been mathematically proven that if there is a possible way to schedule all the tasks and have them complete, that this will find it.

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

The procedure by which the operating system handles executing multiple processes concurrently is known as scheduling, which will be covered later.

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

Example: 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.

Related Topics

Shims

A shim is a program that allows you to add additional functionality to function calls without the ability to modify the source code of the function calls. You call the function just as you would the normal function and in the shim, you call the original function and add additional functionality. The shim should define the function in the same manner as the original function, including parameters and return type.

In order to use the shim's functions in place of others, you need to use LD_PRELOAD={shim location} in the execution command to run your program. This will load your shim library *first* and your defined function will be used in function calls. In order to use the original function, you can use the dlsym function to get a function pointer to next occurrence of that function name using RTLD_NEXT

Example shim

```c
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdlib.h>

int rand(void) {
   int (*original_rand)(void) = NULL;
   original_rand = dlsym(RTLD_NEXT, "rand");
   return original_rand() % 100;
}
```