COMP310/ECSE427 Study guide

Francis Piché

November 6, 2018

Contents

Part I
Preliminaries

1 Disclaimer

These notes are curated from Professor Muthucumaru Maheswaran COMP310/ECSE427 lectures at McGill University, and *A. Tenenbaum and H. Bos, Modern Operating Systems, 4th Edition, Pearson, 2015.* They are for study purposes only. They are not to be used for monetary gain.

2 About This Guide

I make my notes freely available for other students as a way to stay accountable for taking good notes. If you spot anything incorrect or unclear, don't hesitate to contact me via Facebook or e-mail at `http://francispiche.ca/contact/`.

Part II
Overview of OS Concepts

3 Introduction

3.1 What is an OS?

An operating system is a trusted software which interfaces between the hardware and user applications to provide:

- Security
- Usability
- Efficiency
- Abstractions
- Resource management

They attempt to solve the problem of maximizing utilization, and minimizing idle time, to maximize throughput.

3.2 Design Concerns For Different OS's

For a personal/embedded system: response time should be minimal

For a time-sharing system: there should be *fair* time sharing

In batch-processing systems: goal is to maximize throughput.

3.3 Booting

First, the hardware is powered and the CPU is in "real-mode" which is essentially "trust everything mode". From here, the BIOS are loaded from ROM, and the CPU switches to "protected" or "user" mode. Finally the Kernel finishes initialization and the kernel services (OS) are started.

3.4 Processes

A process is a running program. Each process has an "address space" or "**core image**" in memory which it is allowed to use.

This address space contains the program's:

- executable code
- data (variables etc)
- call stack

The **process table** is an array of structures containing each process in existence. This includes all of the state information about each program, even if it is in a suspended or background state. (Some programs may run periodically or in the background).

A **child process** is a process that was created by another process.

Each person using a system is assigned a **UID**. Each process has the UID of the person who started it. Child processes have the UID of their parent.

Multiple processes allows for better utilization since while one process is idle (for example waiting for I/O) another process can work.

3.5 Memory Management

Processes need to be kept separate from other processes, and memory must also allow more than one process to exist in RAM at the same time.

We might also need to have these processes communicate.

Thanks to virtual memory, the address space can be larger than the actual amount of physical memory addresses. For more on virtual memory, see my COMP273 guide (or this guide in the later sections.) This also allows for better "chunking" of memory to keep processes separate. We can also provide shared memory spaces for inter-process communcation. Virtual memory also allows for processes to not care where in memory they actually are, which is useful since they may be moved around when they get "kicked out" by another process.

3.6 Storage

Need persistent storage, but it's slow. A hard disk is broken up into blocks which contain binary data. So when you use files, for example, they are saved as a series of blocks of binary data on the secondary storage.

3.7 OS and Interrupts

There are two kinds of Interrupts, hardware and software. Hardware is when a device sends a signal to the CPU, for example, a mouse is moved and needs to be processed. Software is when a program (OS or otherwise) throws an exception (trap). This alters the regular flow of the CPU and is therefore an interrupt.

3.8 Dual-Mode OS

Dual mode is the idea of keeping Users (unprivileged) and the kernel (privileged) separate. This provides greater security since this ensures on the trusted OS can make potentially dangerous operations on the core of the computer.

When a program running in user mode sends a system call, the OS then switches to kernel mode to complete the operation, and back when complete.

3.9 Monolith vs Micro-Kernel

There are two main architectures for an OS, the Monolith (one big program that handles everything) and the Micro-kernel. The latter's kernel is just the bare minimum inter-process communication, while the rest is a series of micro-services each capable of executing one OS task.

Part III
Scheduling

4 Processes

A process is an abstraction of a running program. (As stated previously). It is necessary because we need to be able to handle multi-programming. For example, a webserver handling many user requests at the same time.

This allows for better utilizaton (even if only 1 core CPU) since while one program needs to wait (IO or something) another can run. While there is overhead for context-switching, it is still generally faster to do this.

4.1 Process Representation

The process needs to contain:

- Program counter
- Stack
- Register state
- Memory state
- Stack etc.

A process is represented by a process control block. This is a table whose entries (indexed by process id's (PID)), are a sub-table containing (at least):

- CPU State
- Processor ID
- Memory
- Open files
- Status
- Parent and Child process'
- Priority

So to switch context, we would need to save all the things we need, load the new process, run it for a while, and repeat.

4.2 Lifecycle Management

We need to manage how process are handled through their whole lifecycle. This includes their creation, state changes and termination.

Processes are created using a system-call (can't just make processes willy-nilly, need to call the kernel).

There are two approaches to this, either we build the table from scratch, or we clone an existing process. The latter is the UNIX approach and the one we'll focus on. In this, we stop the current process and save it's state. We then make a copy of all the code, data, heap and PCB and give this new process a new PID. Then, we pass it to the dispatcher which will schedule the process.

4.2.1 fork()

This cloning method is done using the `fork()` system call, after which the parent and child run concurrently. Note that the `fork()` call returns the PID of the child process to the parent, and 0 to the child if successful, and -1 otherwise.

For example:

```
main() {
   int i;
   i = 10;
   if (fork() == 0) i+= 20;
   printf(" %d ", i);
}
ouput> 10, 30
```

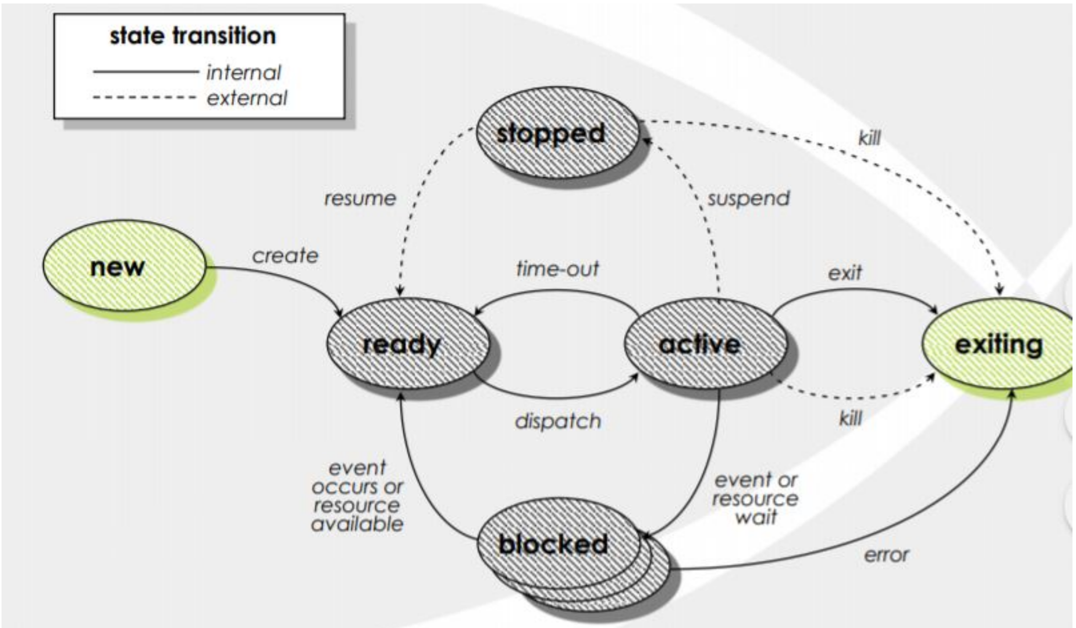
The parent outputs 10 (unchanged) and the child outputs 30 (was affected by the if statement).

4.2.2 State

A process can change state while it is executing. For example if theres no memory or processor available, waiting for some outside event, we finished the task and exit, etc. The possible states are:

- New (Process creation begins)
- Ready (Process has been created and is ready to be loaded and run.)
- Blocked (Waiting for some reason)
- Active (Running)

- Stopped (Suspended by the scheduler)
- Exiting (has been terminated (either normally or by killing))



A process enters the exiting state when either it finishes its task (uses an `exit()` call), caught an exception, or some user decided to kill it.

4.2.3 Tiny Shell

This is Prof Maheswaran's example of a shell.

```
while(1){
   printf("Prompt>")
   getline(line)
   if(strlen(line) > 1){
      if(fork() == 0){
         exec(line)
      }
      wait(child)
   }
}
```

In this, the parent creates a new child process for each command, and waits for the child to complete.

4.3 Dispatcher

There are two ways for the dispatcher to get control of the CPU (since it can only be used by one process at a time, and the dispatcher itself is a process), waiting (trust the other process) or by interrupts. The latter is preferred, since the other process may have an infinite loop, otherwise be bad.

When an interrupt is sent, the OS saves the state of the active process, and runs the interrupt routine. (This is the process of saving the PC, status, registers, file pointers, memory etc.) Note that while this process occurs, no other interrupts are allowed.

Also note that memory is not always saved to disk. Since we use isolated address spaces for each process, the memory of one process (even if idle) need not be overwritten by another.

5 Process I/O

How do processes deal with I/O?

They have an array of "handles" which are each hooked up to an external device. (Mouse, keyboard, file, etc) These handles allow for easy communication between the process and any external I/O devices.

This table exists in the Kernel memory. (Restricted block of RAM). So the only way for a process to access this table is through a system call.

5.1 File Descriptors

We call the slots of the array `file descriptors`. In code, this looks like:

```
main() {
   char buf[BUFFERSIZE];
   const char* note = "Write failed\n";

   while ((n = read(0, buf, sizeof(buf)) > 0))
      if(write(1, buf, n) != n){
         (void)) write(2, note, strlen(note));
         exit(EXIT_FAILURE);
      }
   return(EXIT_SUCCESS)
}
```

Where we are reading from 0, writing to 1 and 2. We call 0 `standard input`, 1 `standard output`, and 2 `standard error`.

We can then use the `read()` and `write()` methods to specify which file descriptor to use.

If we use `open()` to open a file. This returns a file descriptor of $n > 2$ (assuming close(0) or close(1) was not called before). We can then pass this number to a system call such as `read()` or `write()` to access the file.

It's worth noting that the operations are not being sent/recieved directly to/from disk, since that would be slow. We actually interface with a cached version of the file from kernel memory.

We can do: `close(1)` and `open("file")` to overwrite the 1 to point to the file. (Since open goes to the first available array slot)

Similarly we can do `close(0)` to do input redirection.

We can also implement piping. This is when we have one process' output be the input for another process. We need to pass through the kernel space. We use the `pipe()` system call. The pipe creates two file descriptors, one for each side of the pipe. This works since the pipe (a space in the kernel) is created by a parent process, and since the `fork()` system call clones everything from the parent to the child, the child will have access to the pipe.

Then, we rewire the file descriptors such that we set one end of the pipe to 1 (in the parent process), and the output end to 0. (in the child process) This is done using a close, and a `dup()`, which duplicates a file descriptor and puts it in the first open spot.

6 Threads

6.1 Concurrency vs Parallelism

Concurrency is running multiple, discrete tasks on a single-core system. Parallelism is running on completely separate cores (actually running at the same time). Concurrency is sort of "broader" and more conceptual than parallelism. It is not necessarily caring about performance, while parallelism is entirely concerned about performance.

6.2 Parallelism

We need parallelism to do things faster (duh).

Most applications are not completely parallel. They normally have a serial portion (non-parallel) and a parallel portion. But how much of a parallel portion can we make to make things go faster?

6.3 Amdhals Law

Performance improvement obtained by applying an enhancement on an application execution is limited by the fraction of time the enhancement can be applied.

Basically if your improvement only speeds up half of the application, the best you can do is speed up your application by one half, since the other half is unchanged.

So a speedup is given by:

$$speedup \leq \frac{1}{S + \frac{1-S}{N}}$$

where S is the serial portion, and N is the number of cores. So then $1 - S$ is the parallel portion.

6.3.1 Example of Ambhals Law

Suppose you have a problem that takes $500s$ to complete on a single core machine. How long would it take an 8 core machine if 60% of the problem can be run in parallel?

First, 40% that cannot be parallelized is $200s$. Next, we can divide the remaining $300s$ could be separated amongst 8 cores, so $300/8 = 37.5$. In total we have $200 + 37.5 = 237.5$

Now suppose we have 64 cores. Again we have $200s$ for the non-parallel part, then $300/64 = 4.6$ So in total $204.6s$. Not a big increase in speed for a massive increase in complexity (for managing 64 cores).

Now what if our process takes $5000s$? How does it run on 8 cores? 64 cores?

$$2000 + 365 = 2375s$$

$$2000 + 46 = 2046s$$

So again it doesn't scale all that great.

Now suppose further that we have 10 instances of the $500s$ application instead. On 8 cores, we can send 8 instances to individual cores to complete in $500s$, then divide the remaining two instances amongst the 8 cores (4 cores for each instance). So then we have

$$500 + 200 + 300/4 = 700.25s$$

Now what if we had 64 cores? We can break up the 64 cores into 9 groups of 6, and one group of 10. So we have 10 groups total. We can split each instance amongst a group. So the groups of 6 would take time:

$$200 + 300/6 = 250s$$

and the group of 10 would take time:

$$200 + 300/10 = 230s$$

All groups run at the same time, so the total time is the max of the group of 6, and the group of 10. So in total:

$$max(250, 230) = 250s$$

Obviously, all of these calculations don't include overhead.

Since the max is take, we could have just divided 10 groups of 6, and left 4 cores idle. This would have less overhead and be more beneficial.

6.4 Threads

Threads can make things happen at the same time, and are much more lightweight than a process. For example, you may want a web server that can handle multiple threads at the same time, one for each request. If you were to use a new process, the application would be much heavier. This is why Chrome is so RAM heavy, since they use processes and not threads.

Threads can also share memory, which is easier for programming.

Threads however are much less fault tolerant, and can introduce a lot of synchronization issues if not implemented correctly.

Processes are heavy because we have the whole memory, resource allocation, table etc to worry about, while threads are light since they live within processes, and share resources with other threads in the process (minus the call-stack).

For this reason, you should be careful with the scope of your variables in threads, or else a change in one thread could affect another thread. Registers and the stack are the only things that are separate in threads.

6.5 User vs. Kernel Threads

User threads are when the management is done by the user-level libraries. This cannot use the scheduler, since it does not have access to the kernel. There is one kernel-level thread that all the user-level threads map into. If one of the threads block, all of the threads block. the advantage to this is that thread-switching would be faster since we wouldn't need to make system calls.

Meanwhile kernel threads are managed by the kernel scheduler and use kernel-level libraries.

Only kernel level threads are able to manage the multiple threads to run at the same time (or at least concurrently).

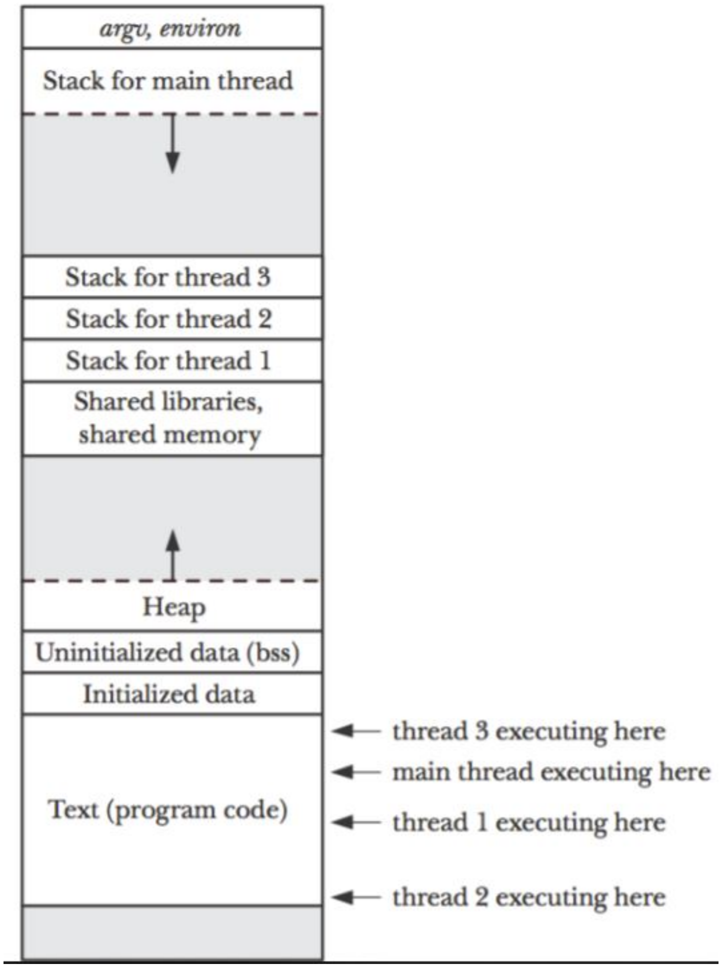
In this way, each user-level thread maps to one kernel-level thread.

6.5.1 Hybrid Threads

This is when there are multiple kernel threads, multiple kernel threads and they are mapped together (not necessarily 1 to 1). So this allows many user level threads to be mapped to a fewer number of kernel level threads. This is not often used.

6.6 Threads in Linux

Recall that threads share memory. However, they also have their own specific stacks. So in memory, it would look like:



We are therefore limited in the number of threads we can have.

6.7 Pthreads

Pthreads allow for portable threads. It is an interface (not implemented library) according to the POSIX standard so that we can develop for different operating systems.

To create one:

```
#include <pthread.h>
```

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void
    *(*start)(void*), void *arg);
```

Note that this signature is similar to clone(). This thread will be kernel level.

To then exit the thread, we can:

- The function we specified to `pthread_create` returns.
- Thread calls `pthread_exit()`
- Thread is canceled by `pthread_cancel()` (preferred).

Each thread is identified by an ID. We can get the thread ID with: `pthread_self()` or we can check if two threads are the same with `pthread_equal(pthread_t t1, pthread_t t2)`

One thread can wait for another thread using the `pthread_join(pthread_t thread, void **retval)` function.

7 Inter-Process Communication

Suppose we want to have synchronization between two processes. This is accomplished via either shared memory or message passing.

In shared memory, the kernel sets up a space in memory that multiple programs can access.

But how is this actually implemented? We can send messages directly, or indirectly (via a buffer obejct). How do we use buffering, and how do we deal with errors?

7.1 Direct Communication

This can be achieved either synchronously or asynchronously.

In the synchronous case, all send and receives are blocking operations.

In asynchronous case, send operations is usually non-blocking, but the receive may be blocking, or non-blocking.

We can also do symmetric or asymmetric addressing. This is when we either specify the name of the process we are talking with (symmetric) or we simply listen for anyone (asymmetric).

7.2 Indirect Communication

Here we use mailboxes. We simply write to the mailbox, or read from the mailbox. We don't care who the sender or receiver was.

A special case of this mailbox is ports. With ports many process can write, but only one can read. We need ports because if we just sent messages to the whole system, there would be no way of knowing which process the message is for.

7.3 Buffering

We can use `zero capacity` buffer for synchronous communication. This means that both processes must be ready to read/write to make the successful message.

`Bounded capacity` is when the buffer is full, the sender waits. `Infinite capacity` is when the sender never waits.

7.4 Error Handling

Messages can get messed up. They can get duplicated, delayed, or delivered out of order.

8 Synchronization

Concurrent processes can be of two main types: `competing` and `cooperating`. The OS manages competing processes through context switching and resource management. For this reason, competing processes are independent of eachother, and thus are deterministic and reproducible. So its the cooperating processes we need to worry about. These, by contrast, can influence each other, and in many cases are not deterministic. They are aware of each other, and pass messages (either directly or indirectly).

8.1 Race Conditions & Critical Sections

A race condition is when the order in which the steps of a procedure occur affects the outcome of the procedure.

We can avoid race conditions using mutual exclusion. This is when we declare sections of code in which a race condition occurs to be a critical section, and enforce the condition that only one process may be in the critical section at one time.

More formally, a critical section must meet these requirements:

- No two processes may be simultaneously in their critical sections
- No assumptions can be made about the speed or number of CPU's
- No process running outside its critical section may block other processes
- No process should have to wait forever to enter its critical section

Critical sections should always run to completion, and must never be interrupted in the middle.

For efficiency, we should try to minimize the lengths of the critical sections.

We could try to handle them by disabling all interrupts. This would work in a single processor case, since interrupts are generally the cause of out-of-order execution. However, this is not practical, since the OS would be frozen during the critical sections execution, and it doesn't work with multiple processors.

8.2 Locks

We only want one thread running through the critical section at a time, so how can we do this?

8.2.1 Attempts at Locks

: We could simply try to "take turns" by having a shared variable that is set after the critical section using a busy-wait. This is bad, because we violate the critical section conditions, namely, its possible that one process wait forever to enter.

We could also try to replace the turn variable with two flags. Process 0, waits for `flag[1]`, and sets `flag[0]` to true when entering, and then to false when exiting. Process 1 would do a symmetrical thing on it's side.

This doesn't work, since we would need a lock, for the lock! Imagine a scenario where the busy wait loop of process 0 completes, and it enters the block. Then, we have a context switch just before the flag can be set to true. Then, process 1 would be able to proceed into the critical section, ignoring the lock, and both processes would be executing the critical section.

But what if we set both flags to true *before* the busy wait. Well, in this case, both would be locked out of the critical section. This happens similarly to the last attempt, where if a context switch happens just after the flag is set, and before the busy wait, both flags would be true, and neither could enter. This is known as **deadlock**.

In yet another attempt, we can expand the while loop busy wait to include an unset of the flag, a delay, and then a reset of the flag. This allows time for the other process to "unlock" itself. This works if the delays are truly random, and can never get synchronized in both processes. This is known as **livelock**

8.2.2 Petersons Algorithm

This algorithm uses a combination of the shared turn variable, and setting flags before a busy loop. The only difference is that the condition in the wait includes both the flag and the turn. This only works for 2 processes, but it's good that it doesn't use a system call or any OS involvement. It is a "software only" solution.

8.3 Hardware Solutions

How can hardware help with mutual exclusion?

TSL, RX, LOCK are all CPU instructions that are often found on modern CPU's. This works by setting a lock variable in memory, then, when a program calls `TSL R1 lockaddress`, it will pull the lock from the address, and place a 1 in the lock address. This is the same as the shared variable we were talking about, but not its in hardware so it works. It works, since hardware is completely atomic. A zero is only received from TSL when no other process is in the critical section. Whenever a 1 is received, a busy loop happens.

This has many advantages:

- works for n processes
- can be used by multiple processors (as long as they share memory)
- Simple
- works for n critical sections

But, it still uses busy waiting, and starvation is possible. The order in which processes enter is arbitrary.

8.4 Priority Inversion

This problem arises from mutual exclusion and busy waiting.

If there are two processes, one high priority and one low priority, where the scheduler always runs the high priority if it is ready, then say at some point the low priority is in it's critical section. It's possible that the low priority never leaves it's critical section, since the high priority is constantly running, and is stuck in a busy wait.

8.5 Sleep/Wakeup

The alternative to busy waiting is to have a sleep wakeup using semaphores. This is better since it does not take up so much CPU time and is not as prone to priority inversion and deadlock.

However, this has some overhead, and so if there is a very few number of processes/context switches, busywaiting can still be better, since it has less overhead.

8.6 Assembly Level Instructions

As shown in the demo in class, even if something is a single instruction (in C) it does not necessarily mean it cannot be interrupted. One command, even as simple as a variable assignment, can be interrupted since it is broken down into multiple assembly instructions. This depends on the level of complexity of the instructions, and whether a 32-bit machine or 64-bit machine is being used. (If you use a 32-bit machine, more instructions are require to split up and manipulate numbers).

A lock instruction in Assembly will actually prevent any interrupts from entering the bus while it is executed so the lock instruction itself cannot be stopped as another instruction could.

8.7 Semaphores

Basically, processes can send messages to each other, and their execution can be altered (stopped, started etc.) by these messages. We can use this for more complex coordination of processes. A semaphore is a special variable that is used to accomplish this message passing. wait() and signal() are examples of sending a message through a semaphore.

Example:
`wait(sem)` will wait on the semaphore 'sem', by decrementing the value of `sem` if it is non-negative, it will proceed. Otherwise, the running process will go to sleep. So say the 'sem' is 1 initially. Some process can come through, change it to zero, and keep going. Then, some other process would come through, change it to -1, and go to sleep.

Semaphores are implemented inside the kernel, and so are only accessible through a system call.

8.7.1 Semaphore Structure

The semaphore is a struct made up of an integer to keep track of whether the lock is released or not (negative means no, 1 means yes), and a queue containing them. The wait() function would just be decrementing the count, and if the count is negative, add the process to the queue and block the process. The signal() function is the opposite, it increases the count, and if the count is not 1, remove a process from the queue and mark it as "ready".

8.7.2 Producer Consumer with Semaphores

The producer adds things to a buffer, the consumer removes them. Implemented with semaphores, there are 3 semaphores. The "mutex", which is the simple lock to protect the critical section, the "empty" which is used to count how many slots in the buffer are empty (initialized to the size of the buffer, N), and the "full" which is used to count how many slots are full (initialized to 0). So the producer needs to:

1.) produce an item
2.) wait() for the "empty" semaphore to not add to the buffer if there is no empty slot available.
3.) wait() for the critical section
4.) Insert the item to the buffer
5.) signal() that it is done with it's critical section
6.) signal() to the "full" semaphore that it has added something to the buffer (increment it up)

The consumer then needs to

1.) wait() on the "full" semaphore until there is something in the buffer
2.) wait() for the critical section
3.) remove an item from the buffer
4.) signal that it is done in the critical section
5.) siganl to the "empty" semaphore that it has removed an item (decrement it)

The problem with the above implementation is that there are 4 system calls for each process (very heavy). It would be better if we could find a more lightweight version of this.

8.7.3 Types of Semaphores

A strong semaphore respects the FIFO queue, a weak semaphore does not.

Binary semaphores are simply an on/off switch, and can only take on two values, 0 and 1. Counting semaphores on the other hand can be used for any number, negative or positive. We saw this with the producer/consumer example.

8.7.4 Barrier() Synchronization Using Semaphores

A barrier is essentially a synchronization primitive.

Suppose we have two processes, P1 and P2, where P1 reaches a function $f()$ faster. How do make sure that P2 and P1 enter $f()$ at the same time? (not a critical section, just want them to go in the function at the same time).

Suppose we have a barrier, $b = 4$. (4 processes need to wait for eachother). Each process can call $b.bwait()$.

We use 1 binary semaphore $mutex$, and one counting semaphore $barrier$.

```
bwait(){
   wait(mutex)
   bcount--;
   if(bcount==0){
      signal(barrier) //trigger wakeup cascade
      signal(mutex)//release the mutex
   }else{
      signal(mutex) //release the mutex
      wait(barrier) //go to sleep
      signal(barrier) //wake up, this will cause a cascading effect when the
          bcount==0
   }
}
```

In the above, you can see that once the last process decrements count to 0, it causes a signal() which will wakeup a semaphore which will immediately wakeup and signal() again, which will wake up and signal() etc..

8.7.5 Light Mutex

How can we reduce the number of system calls?

First we need to assume an assembly instruction (atomic) is available to `fetch_and_add`. This will fetch whatever is add an address, and add the parameter given to it, and put it back. (Similar to TSL).

```
   lock(m){
      count = 0
      if (fetchandadd(count + 1)>0){ //increment count by one (atomically)
         wait(m) //if someone is in the lock, count is > 0
      }
   }
```

```
unlock(m){
   count = 0
   if (fetchandadd(count - 1) > 1){
      signal(m)
   }
}
```

By doing this, we can check if something is actually in the lock without needing to go to the kernel. Only when there is actually more than one process in the critical section do we need to invoke the kernel.

8.7.6 Light Semaphore

If we have a very short wait/signal we want to use busy waiting since there is less overhead. (Try as exercise).

8.8 Monitors

So far we have always looked at synchronization primitives of the form: enter -> do suff -> exit. But we have other options than semaphores (higher level). Semaphores are kind of like goto statements. They can be super fast in the right hands, but also can get very hairy.

Monitors are a high-level abstraction to allow for easy synchronization. It contains the sensitive data, and an API for safely accessing operations on the data. Monitors themselves are critical sections, but they already handle all mutual exclusion.

Monitors have condition variables to allow for flexibility in different types of synchronization. Each condition variable has only wait() and signal() operations. (These are different from the semaphore ones!)

How are monitors used for say adding to a global sum?

```
Monitor{
   int gsum
   add(s){
      gsum + s
   }
}m
```

The locking is handled by the monitor behind the scenes, so outside we can just all m.add(s) and not worry about locking.

Now how can we implement a semaphore?

```
Monitor {
   int count;
   condition waiting;

   init_sem(int i){
      count = i
   }

   sem_wait(){
      count--
      if (count <0){
         waiting.wait()
      }
   }

   sem_signal(){
      count++
      if (count <0){
         waiting.signal()
      }
   }
}m
```

Again we assume Monitor is a critical section for free :D

8.9 Lock-free Datastructures (NOT ON EXAM)

How can we implement something like a stack (which can be modified by many threads) without using locks? Locks are seemingly necessary since there is a global head pointer, and pointers to the next which can get messed up. But, killing the threads while one thread holds the lock, will cause the other to be locked out. (This is why you cannot kill threads, they must die on their own). But what if we do it without locks?

For this we will use a new atomic instruction: `compare_and_swap(new_value)` this takes a pointer, and checks if the value at that pointer is the same as the new value. It returns true or false. If true, it will swap in a new value. So, in the context of a stack, if one compare and swap passes, it will write, meanwhile the other process compare and swap will always fail, since the value it "thinks" is inside the stack, is not the "true" value.

8.10 Readers and Writers problem

Suppose you have a database, with several readers and writers. We want concurrent access to the database. How can we make sure that all readers can access at any time, but no two writers can access at the same time?

If we only allow writing when there are no readers, this can cause a problem, since there could be non-stop reads, which would starve the writer. Similarly, if we don't allow readers while writing, the readers could starve too.

This is the case that arises in the assignment. The solution is to have the writers have a simple lock, while the readers have something more involved.

```
READER:

wait(mutex)
readers_count ++;
if(readers_count == 1) wait(write_lock);
signal(mutex);

//DO READS

wait(mutex)
readers_count --;
if(readers_count == 0 ) signal(write_lock);
signal(mutex);
```

This has the problem of starvation, and gives readers priority. Starvation could be solved by having a maximum number of readers.

9 Deadlock

Deadlock is defined as the permanent blocking of a set of processes which compete for resources. In general, there is no efficient solution to this problem. They must involve conflicting resource needs, otherwise there is no possibility of a problem.

There are 4 conditions for a problem to be deadlock:

- Mutual exclusion: No two processes can use the resource at the same time
- Hold and wait: Process may wait for one resource while holding another resource.
- No preemption: Process cannot (or will not) give up the resource until it is done.
- Circular wait: each process in the chain holds a resource which another process is requesting.

If any of these conditions are not satisfied, no deadlock can occur. THis is useful when thinking of solutions. If we only allow shared resources (rather than competing), we can break mutual exclusion. If we abort the process when trying to request a resource in use, we break *hold and wait*. If we avoid/prevent deadlocks to begin with, we skip the cycle rule. If we allow to backout to a checkpoint, we break the irreversable process (no-preemption).

9.1 Resources

Resources can be grouped by:

Reusable: not depleted by a process using it. (cpu, memory etc)

Consumable: can be created and destroyed. (messages, signals etc)

Preemtable: Can be taken away by the process using it, with no consequences. (CPU, memory via context switches)

Non-preemptable: Cannot safely be removed from a running process (CD, printer etc)

Deadlocks mostly occur with reusable and non-preemtable resources.

9.2 Resource allocation Graphs

Processes are shown by circles, resources by squares. Arrow are interactions between processes and resources. If we have cycles in this graph, we may have deadlock but not always if there are multiple units per resource! A cycle is necessary for deadlock, but not sufficient. However, a *knot* is! A knot is when there is a cycle with no non-cycle path leaving the cycle.

9.3 Strategies for Deadlocks

- Ignore it: most of the time deadlocks are rare, so hope it doesn't happen.
- Prevent it: design the system in a way that deadlock can never happen. (Often inefficient)
- Avoid it: Add checks that will see if the next action will cause a deadlock
- Detection: Let the deadlock occur, and take action to recover.

9.4 Deadlock Prevention

Two ways:

Indirect methods: prevent something other than the circular condition

Direct: break the cycle condition

We might not want to prevent deadlock, since it is almost always more inefficient. But if we did, we can do it by attacking any of the conditioned previously mentioned. However, mutual exclusion is often not-breakable.

9.5 Deadlock Avoidance

In this approach, we monitor the system, and ensure that no action will cause a deadlock. We will have a "Resource Manager" which will do all checks. So when processes ask for resources, they must ask the Resource Manager to do it for them.

For this we need to know in advance a maximum number of resources that a process needs. This can be declared by the process to the Resource Manager. Then the algorithm will need to dynamically figure out if the request will cause a cycle.

We break up the states into three categories: safe, unsafe and deadlock. Unsafe is when deadlock is possible, but has not yet occurred. Safe is when deadlock can never occur. Ie: there exists a sequence $P_1, P_2, ..., P_n$ of all processes in the system, such that for all P_i, the resources that P_i can request can be satisfied by the currently available resources and the resources held by P_j, for $j < i$. If there is not enough available resources, the P_i waits for enough P_j's to complete so that P_i can claim those resources. So, our Resource Manager will need to ensure that no process moves from the safe state to an unsafe state.

Using the graph we can easily check if allocating a resource will cause a cycle in the graph, and it it would, then don't allocate it.

9.5.1 Bankers Algorithm

Same idea as above, but the resources can have multiple units. We keep a *max* matrix of the processes as rows and resources as columns. The entries are how many units of a resource a process wants to use. We keep another matrix, *have*, similar to the other, which contains how many units the processes actually contain at this time. We keep a third matrix, *need* which has $max_{ij} - have_{ij}$. So when about to allocate a resource, we simply compare the resource vector to the *need* matrix. If there is a process that could run to completion given those resources, do it. Afterwards, update the *have* and *need* matricies accordingly.

For this to work, each resource must have multiple units, each process must pre-state its maximum usage of a resource, all processes must be able to wait for resources, and all processes must return its resources in a finite amount of time.

9.6 Deadlock avoidance example

Suppose you have a printer, and a disk. Two processes want to be able to send files from the disk to the printer.

```
program(){
   get_printer()
   get_disk()
      transfer_document()
   release_disk()
   release_printer()
}
```

suppose we have another version in which some orders are flipped, and suppose each process is using a different version.

```
program(){
   get_disk()
   get_printer()
      transfer_document()\end{document}
   release_printer()
   release_disk()
}
```

Assume only one process can access a resource at one time. If they were both using the same version everything would be fine. However, if they are using different versions, problems can arise. Suppose P_2 gets the disk, and P_1 gets the printer before the other process gets it. This results in a deadlock since both need both resources to

9.7 Deadlock Detection

Here we don't try to prevent deadlock, we instead try to detect it once it has occurred. Detect by running through the graph for cycles.

Can do on each resource request, or periodically. We can recover by either rolling back processes (keeping checkpoints), terminating processes, or trying to take the resource and give it to another process.

Part IV
Virtualization

In general, the goal of virtualization is to create a "virtual cpu" as a thread to give the impression that it is its own CPU, independent of others. If implemented as processes, each process has its own memory view. However, the filesystem and kernel are still shared. Virtual machines is the idea of giving a process its own version of all of these things.

At the base level, a virtual machine monitor is placed on top of the hardware layer, which allows multiple virtual machines to run simultaneously. It does NOT run all of the instructions itself, it just takes care of housekeeping tasks and managing the VM's. The real hardware still takes care of instructions.

The **Host** is the underlying hardware system. The Virtual Memory Manager (VMM) creates an identical interface to the host. The Guest is a processes provided with a virtual copy of the host.

10 Types of VMMs

We also need to create logical partitions between multiple running VM's. This can be done by implementing the VMM in the actual hardware itself, some OS-like software, or run as an application on top of the OS.

- Paravirtualization: the guest OS is recompiled to be compatible with the VMM
- Programming-environment virtualization: VMMs that are entirely software-based.
- Emulators: Allow any machine to run things meant for a very specific hardware
- Application containment: Not actually virtualization, but segregates applications entirely from the OS.

11 Privilege Level In VMs

In a typical non-virtualized environment, we have the kernel between the user and hardware. In a virtualized environment, we have the VMM layer between the hardware and the guest kernel.

Suppose you have an instruction which can run in either user-mode or kernel mode, which sets a flag bit in the status register which disables interrupts. Obviously, we don't want this to be possible for user-level. So, if the CPU is in user-mode, the bit is ignored, and if in kernel mode, it is not.

In a virtualized environment, the guest kernel is running in user mode, so how can we disable interrupts only for the guest kernel?

First, we need to break sensitive instructions into:

- Control-sensitive instructions: Attempt to change configuration of resources
- Behavior-sensitive: Instructions who's behavior is determined by configuration of resources.

If you try to run a privileged instruction in user mode, this causes a fault, whereas if you're in privileged mode it should run fine. So with that in mind, consider this theorem:

For any third-generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

The solution is to have sensitive instructions which cause a trap to go to the VMM. The VMM will then figure out which VM caused the trap, verifies and emulates the instruction in the guest OS.