CS492: Operating Systems Notes

Steven DeFalco

Spring 2023

Contents

1 Introduction

1.1 Operating System Concepts and Structure

The **operating system** offers functionality through **system calls**. When a system call is made, the kernel runs appropriate code in the **priveleged mode**.

A group of system calls implements **services** such as file system serives and process management services.

1.1.1 Process vs. Program

A **process** is a user-leve abstractin to execute a program on behalf of a user. Process is a kernel abstraction in which the program is going to run. Each process has its own **address space**. A **program** is a series of binary CPU instructions stored in a file. A program requires resources to be executed (such as CPU or memory). The operating system needs to represent an instance of a program in execution and that is a process.

1.1.2 Address Space

Address space is the range of valid memory address for a given process. Address space can be read and/or written and/or executed. Each process has its own separate address space which consists of...

Text	- binary program code
Stack	- function calls data
Heap	- dunamically allocated data

1.1.3 Files

A **file** is an abstraction of a (possible) real storage device such as a hard disk. You can read/write data from/to a file by providing postion and an amount of data to transfer. In UNIX, everything is considered a file. Files are maintained in **directories** which keep a name for each file they contain. Directories and files form a hierarchy.

Block Special Files can be written and read from block by block. (e.g. disk)

Character Special Files can be written and read from byte by byte. (e.g. serial port)

Pipes are pseudo files allowing for multiply process to communicate over a FIFO channel.

Device drivers are software inside the kernle that knows how to initialize and communicate with specific hardware devices.

1.1.4 System Calls

System calls are the interface the kernel offers to applicatins to issue service requests. They are highly specific to the operating system and hardware; therefore, you should use system calls included in the C standard library. If you use C system calls then your code will be portable because C will use the correct internal system call depending on the operating system without requiring the programmer to use the oeprating system specific call.

1.1.5 Kernels

Monolithic kernel is a very large collection of C functions linked together into a single very large executable binary program that executes in kernel mode.

- every function can call every other function
- a single bug will crash everything
- historically how most kernels were implemented

Microkernel splits the kernl into small well-defined modules.

- Only one kernel runs in kernel mode: the microkernel
- the rest of the modules run as relatively powerless ordinary user processes
- a single bug cannot crash the entire system
- there is more overhead communication between the modules

Hybrid kernel is a compromise between a microkernel and a monolithic kernel

1.2 Computer Hardware

Central Processing Unit (CPU)

- fetches instructions from memory and executes them
- each CPU has a specific set of instructions and thus different instruction sets on different CPUs will not be interchangeable
- has different **registers**
  - General register for genera data
  - Program counter to store the memory address of the next CPU instruction to execute
  - Stack pointer to store the address of the top of the stack
  - Program status word: readable/writable bits that store the state of the CPU and different conditions that resulted from the previous instruction

Memory
In order of decreasing speed and increasing capacity: registers, cahce, main memory, magnetic disk, tape

- the cache temporarily holds a piece of data from a slower memory into a faster memory

Hardware multithreading is when you run multiple processes on a single core. Each core has two (or more) sets of registers. Switches between two different sets of registers on each instruction (where needed) to eliminate waiting for data.

Flash memory stores data even when the electricity is removed (SSD).

An **interrupt** is an electric signal on the bus. Interrupt processing involves taking the interrupt, running the kernel interrupt handler, and returning to the user process. The CPU checks for interrupts in-between instructions. When there is an interrupt,t eh CPU will jump to a unique function in the kernl (the interrupt handler), the kernel handles the interrupt, then the CPU resumes the original process.

1.3 Introduction to Linux

The C standard library contains the implementation of all the functions required by the definition of the C programming language. The kernel provides an API which is also made of C functions, called system cals. Calling a system call cannot be done directly in a user program written in C because it requires use of a special assembly instruction that switches the CPU to kernel mode. Libc includes C functions (wrappers) which have the same names as the system cals which internally do the real system call for yoy. If such a wrapper is not available for a given system call, libc also provides the *syscall* function that allows you to call a system call using its system call number.

POSIX (Portable Operating System Interface) defines a portable interface including multiple aspects of the OS interface to maintain compatibility between different OSes.

1.3.1 Create your own kernel

1. Configure the kernel by hand (`make menuconfig`)
2. Compile the kernel (`make -h $(nproc) all`)
3. Install the modules (`sudo make INSTALL_MOD_STRIP=1 modules_install`)
4. Install the kernel (`sudo make install`)

Modules are a piece of compiled C code added tot he kernel while the kernel is already running, in kernel mode. Modules shorten the development cycle because modules can be loaded and unloaded without having to reboot the system. To load a module that you've made use `sudo insmod ./<module-name>.ko` and to remove the module use `sudo rmmod <module-name>`. Modules have the visibility of all the kernel, so you can use all global variables and exported functions.

1.4 System Calls

System calls are the only way a process can enter the kernel. USed to request OS services and priveleged operations such as accesing the hardware, creating other processes, and changing security permissions. System calls switch the execution context of the GPU from user mode to kernel mode.

Syscall trap / Interrupt / Exception

- hardware stacks program counter
- hardware loads new program counter from interrupt vector
- assembly language procedure saves registers
- assembly language procedure sets up new stack
- C interrupt service runs
- scheduler decides which process is to run next
- C procedure returns to the assembly code
- assembly language procedure starts up new current process

1.4.1 To add a syscall in Linux

1. Write your syscall function in either an existing file or a new file. Modify the makefile (`obj-y += my_syscall.o`)
2. Add your syscall in the architecture specific syscall table (`arch/x86/entry/syscall/syscall_64.tbl`)
3. Add your ssycall prototype (`include/linux/syscalls.h`)
4. Recompile, reinstall, reboot the kernel
5. Test using the syscall function: invoking the system call from a user-mode program to trigger your code

2 Processes and Threads

2.1 Introduction to Processes

Multiple processes can run the same program. A single CPU can run only one process at a time.

Multiple processes (multiprogramming) increases CPU utilization. This consists of overlapping one process's computation with another's wait and also reduces latency (more interactive). To distinguish multiple processes of the same program, each process has a different **process identifier (PID)**.

A **context switch** pauses execution of one process and continues with the execution of another process. The order in which processes execute is not fixed and may not be reproducible. Thus, programmers cannot make time assumptions and must write their code as if it is the only thing that will be running on the CPU.

Each process has the following key components:

- **stack** stores function arguments and local variables
- **heap** stores dynamically allocated memory
- **executable program** is made up of the initialized data segment and the text degment (CPU instructions from program file)
- **execution context** contains program counter, stack pointer, CPU registers, etc...

Memory layout randomization mixes the location of each part in memory so that the OS is more secure. It will be harder to locate specific processes in memory if it is randomized. There will also be unusable memory in the process that is there to kill the process if anything tries to access the unusable memory.

Stack is built with stack frames and includes local (function) variables). Function parameters are passed on the stack. (LIFO)

Heap is used if it is not known how much data you will need at runtime. Dynamically allocated with random access.

Processes are created during system initialization, from already running processes, due to user interaction, or execution of a scheduled batch job. UNIX process philosophy is to create a new process which is an identical copy of the current process (fork) and then in the new process load the other program.

`fork()` is a syscall used to create a new process

- the proces that calls the `fork()` syscall is called the **parent process**
- the newly created process is called the **child process**
- the child is an identical copy of the parent (runs the exact same code)
  - child process returns a zero value
  - parent process reutrns the PID of the new child
- If C code contains a sequence of n consecutive `fork()` calls, 2^n processes will be created
- most UNIX systems have a limit on the number of processes a given user can have at the same time (`ulimit -a`)

A processes **execution state** is what the process is currently doing...

- **Running** - executing instructions on the CPU, it is the process that currently has control of the CPU, cannot have more processes running htan number of cores/CPUs
- **Ready** - waitng to be assigned to CPU: ready to execute, but another process is executing on the CPU
- **Waiting/blocked** - waiting for an event: cannot make progress until event happens (most processes are in this state)

Types of **process termination**

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary) is used when there is a fatal program bug

Orphan processes are children processes whose parents have been terminated. Can be adopted by another process, or can be killed automatically when parent is killed (for security reasons).

Kernel maintains a table called the **process table**. Is stored in the kernel memory and is invisble to processes. Has one entry per process (process's PID is an index into this table). The entry is called the **process control block (PCB)**: contains all informatin about a process.
When a **context switch** occurs

- hardware stacks program counter
- hardware loads new program counter from interrupt vector
- assembly language procedure saves registers

- assembly language procedure sets up new stack
- C interrupt service runs
- scheduler decides which process is to run next
- C procdure returns to the assembly code
- assembly language procedure starts up new current process

2.2 Processes in Linux

Every process has a user and a kernel part in its virtual address space. User space is running the program; kernel space is running kernel code on behalf of the program (kernel space is invisble to the program). There will be both a user-space and kernel-space **execution context** and thus there will be 2 PCs, 2 stack pointer, 2 stacks, etc...

In Linux, a process is a **task** represented by a `task_struct` inside the kernel's memory. This is the **proces control block** for a linux task. A task struct includes:

- scheduling parameters
- memory image
- signals
- CPU registers
- system call state
- file descriptor table
- accounting
- kernel stack

`fork()` is limited to creating processes only. On Linux, task creatin in general can be done by the `clone()` syscall. When `clone()` is called.

- new child starts executing function
- may use a new stack
- has `sharing_flags` to describe the amount of sharing between the caller and the callee
- blurs the disctinction between process and thread creation
  - if you share nothing then it's the same as fork
  - if you share everything then it's the same as thread creation

Internally, both `fork()` and `clone()` call the kernel function `kernel_clone()`.

2.3 Threads

A **thread** cannot exist without a process. A process is a ”container” for threads. A process may contain one (the default) or more threads. Thread tells you what part of the program is currently being executed. **Thread** is an independent sequential execution stream within a process; programs use one or more threads per process. Items that are **shared per process** include the following:

- address space
- global variables
- open files
- child processes
- pending alarms
- signals and signal handlers
- acccounting information

Items that are **shared per thread** (private to each thread) include the following:

- program counter
- registers
- stack
- state

Some of the advantages of threads:
Performance - shared address space and therefore no communication overhead between threads. Thread creation can be 10-100 times faster than process creation.
Efficiency - allows a program to overlap I/O and communication. Allows one process to use multiple cores.

Amdahl's Law identifies performance gains from adding additional cores to an application that has both serial and parallel components. **Serial** components can only be executed by a single thread. **Parallel** components can be executed by multiple threads.

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

where S is the serial portion and N is the number of processing cores. Note that the serial portion of an application has disproportionate effect on performance gained by adding additional cores / CPUs.

All threads in a single process share the same address space. Global variables are shared between threads. Multiple threads may access the same global variable concurrently and programers are responsible to coordinate access (synchronization algorithms). Each thread gets its own local variables (in stack). Each process starts with a single "main" thread; the first thread can create new threads. Any thread can end at any point; if the main thread returns from the `main()` function, the `exit()` system call is called and the whole process terminates.

2.3.1 PCB

When making threads, the kernel will break the PCB into two pieces.

1. Information about program execution is stored in **Thread Control Block (TCB)**. This includes the program counter, CPU registers, scheduling information, and pending I/O information.
2. Other information is stored in the **Process Control Block (PCB)**. This includes memory management information and accounting informatin.

2.3.2 Threads Implemented in User Space

Pros:

- **Portable**: can be used on any OS
- a user-level threads library can be implemented on an OS that does not support threads
- thread switching is at least an order of magnitude faster than trapping to the kernel
- thread scheduling is very fast: no proces context switching, no kernel trap, no flushing of memory cache
- each process can have its own thread scheduling algorithm

Cons:

- **Doesn't support real parallelism**: kernel will schedule process on single core regardless of number of threads and therefore cannot take advantage of multicore/multiprocessor systems
- if one thread makes a blocking system call then the kernel suspends the whole process
- threads need to voluntarily give up the CPU to each other for multiprogramming so user code is more complex

2.3.3 Threads Implemented in Kernel Space

Pros:

- **supports real parallelism** (multicore/multiprocessor systems)
- no run-time thread system needed in each process
- no thread table in each process
- block system calls are not a problem (the kernel can schedule another thread)

Cons:

- **slower than user-space threads**: if thread operations are common, much more kernel overhead will be incurred because of the many system calls required
- if you fork a multithreaded process, do you copy all threads or just the on that called `fork()`?
- if signals are sent to processes, should the kernel assign it to a specific thread to handle

In practice, user threads are mapped one-to-one with kernel threads: **hybrid implementation**.

2.3.4 Context Switching with Threads

When context switching with threads...

- thread is now the unit of a context switch
- context switch causes CPU state to be copied to/from the TCB
- when context switching two threads of the same process, no need to change address space
- when context switching two threads in different processes, must change address space

2.3.5 Thread Local Storage

Thread local storage (TLS) works like a global variable but each thread has its own copy of it. Use this instead of global variables when making single-threaded code multi-threaded. Visible everywhere in the code, but each thread has its own copy.

2.4 Threads in Linux

Linux, implements threads in the kernel, not in user space. Linux uniformally handles processes and threads: processes and thread tables are unified into a single data structure. Processes and threads are considered **tasks** and are thus represented by a `task_struct`: there is one `task_struct` per thread. Linux assigns PID and TID for each thread; ecah thread has a unique TID. TO check the TID use the `gettid()` syscall.

3 Scheduling

3.1 Introduction to Scheduling

Multiple processes and threads are ready to run; the kernel's scheduler decides which runs next. Scheduling decisions may take place when a process/thread is created, terminates, blocks on an event. Scheduling decisions may also take place when an interrupt occurs; this includes clock interrupts (*running* to *ready*) or I/O interrupts (*blocked* to *ready*).

3.1.1 Clock Interrupts

Clock interrupts are a way for a system to keep track of time. Interrupts occur at periodic intervlas called a clock tick. Clock interrupts are implemented using a hardware clock interrupt and have high priority in the system.

3.1.2 Non-preemptive Scheduling

In **Non-preemptive Scheduling** proceses execute until completion or until they make a **voluntary process swtich** or **process switch on blocking calls**. The scheduler gets involved only at exit or on request (i.e. for every clock interrupt, running processes keep going).

3.1.3 Preemptive Scheduling

In **preemtive scheduling** while a process executes, its execution may be paused and another process resumes its execution. This can take the form of an **involuntary process switch**. This scheduling is used in all OSes today.

3.1.4 Goals of All Scheduling Algorithms

- **fairness** - give each process a fair share of the CPU
- **policy enforcement** - seeing that stated policy is carried out
- **balance** - keeping all parts of the system busy

3.2 Scheduling in Batch Systems

3.2.1 Goals for Scheduling in Batch Systems

- **throughput** - maximize jobs per hour
- **turnaround time** - maximize time between submission and termination
- **CPU utilization** - keep the CPU busy all the time

3.2.2 First-come First-served (FCFS)

Processes are assigned to the CPU in the order they request it (or they arrive). The non-preemptive version will let each job keep running until completion. **Convoy effect** is when a long process delays short processes. **Turnaround time** is the time taken by a job to complete after submission (including the wait time).

$$\text{turnaround time} = \text{time}_{\text{end}} - \text{time}_{\text{submitted}}$$

3.2.3 Shortest Job First (SJF)

Associate the length its CPU time wiht each process. Use the CPU time length to schedule the process with the shortest CPU time first. In the **non-preemptive version**, once the CPU is given tot he process, it cannot be taken away and then after completion of a process, will consider the next shortest job and begin executing that one until completion. In the **preemptive version**, if a new process arrives with less CPU time than the remaining time of the current ecxecuting process: then preempt.

The **preemptive version** is called **shortest remaining time next (SRTN)** and is optimizal in terms of average turnaround time (i.e. always gives minimum average turnaround time). This method also prevents the convoy effect. However, there are more context switches and a process may be starved of the processeor if short processes keep arriving.

3.3 Scheduling in Interactive Systems

3.3.1 Goals for Scheduling in Interactive Systems

- **response time** - respond to requests quickly
- **proportionality** - match duration expectation of users

3.3.2 Round-Robin (RR)

Each process is allowed to run for a specified time interval called the **time quantum**; in practice, the time quantim is likely not fixed. After this time (the time quantum) has elapsed, the process is preempted and added to the end of the ready queue and the next process is scheduled. If the process terminates or blocks before the time quantum is entirely used up, then the process loses the resty of its time quantum and the next process is scheduled with a new time quantum.

Advantages of RR

- solution to fairness and starvation
- fair allocation of CPU across jobs
- low average waiting time when job lengths vary
- good for responsiveness if small number of jobs

Disadvantages of RR

- context-switching time may add up for long jobs

Context switching may impact the choice of the time quantum...

- when there are many processes, a long quantum causes a poor response time
- a short time quantum makes things more responsive but gives a higher context switch overhead
- time quantum might be variable depending on CPU load

Cache state must be shared between all jobs which may slow down execution in RR scheduling. There is no cache sharing in FCFS.

3.3.3 Priority (PRIO) Scheduling

PRIO has four priority classes and always executes the highest-priority runnable jobs to completion. Each queue is processed in a round-robin fashion with a time quantum. Note that lower numbers represent higher priority. There are, however, some **problems** with PRIO scheduling...

- **starvation** when lower priority jobs don't get run because higher priority tasks always running
- **priority inversion** happpens when a low priority task has resource needed by high priority task, which then must wait

Priorities can be **assigned statically** based on process type, user, how much the user paid (if a cloud server); or **dynamically** based on how much a process runs vs doing I/O. Priority $= \frac{1}{f}$ where f is the size of the quantum las t used. Thus, the longer a process ran, the lower its priority and the process that runs the shortest gets the highest priority to run next.

3.3.4 Multiple Queues (MQ) Scheduling

Same as priority scheduling, but each queue has a different time quantum (shortest quantum for high-priority and longer for low-priority). Processes start at the highest priority. When a process **exceeds** its time qunatum, it's moved to the next lowest priority queue. When a process **becomes interactive** it is moved to the higher priority. If the user discovers how to make their tasks more ineractive, they can get all of their processes to be very high priority, which is a problem.

3.3.5 Other Schedulers for Interactive Systems

- **Shortest processes next**: SJF but educated guess for how much CPU a process will need next
- **Guaranteed scheduling**
- **Lottery scheduling** (probabilistic scheduling)
- **Fair-share scheduling**: round robin scheduling between users.

3.4 Scheduling in Real-time Systems

3.4.1 Goals for Scheduling in Real-Time Systems

- **meeting deadlines** - avoid losing data
- **predictability** - avoid quality degradation in multimedia systems

Time plays an essential role in real-time system scheduling. One or more physical devies external to the computer generate events and the computer must react appropriately to them within a fixed amount of time. If the computer reacts too late, it is **as bad as not reacting**.

Real-time scheduling can be split into two categories. **Hard real-time** is when there are absolute deadlines that must be met. **Soft real-time** is when missing an occassional deadline is undesirable but tolerable.

Let's assume that process behavior is predictable and known in advance (the software should be writtein in a way to make this true). Task timing is known and represented through the following values.

Release time (R_i) is the earliest time when a task can start execution
Execution time (C_i) is the expected execution time for a task
Deadline (D_i) is the time by which the proecss must be completed.

We can also assume that the **periodicity** is known. It can be one of the following options:

- periodic process

- sporadic process (aperiodic, hard deadlines)
- aperiodic proces (aperiodic, soft deadline)

3.4.2 Number of Schedulable Processes

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

where ther are m periodic events. Event i occurs with period P_i, requires C_i time on the CPU. The percentage of CPU usage for event i is $\frac{C_i}{P_i}$.

3.4.3 Rate Monotonic Scheduling

Preemptive algorithm, where the shorter the period, the higher the priority.

3.4.4 Earliest Deadline First Scheduling

Schedules processes according to the shortest remaining time until the next deadline. The shorter the remaining time, the higher the priority.

4 Concurrency

4.1 Inter-process Communication

Processes need to share information or coordinate. Threads are nice for this but do not provide isolation from each other. Processes provide isolatin but communication requires system calls. Some **problems with inter-process communication** include...

- how can one process pass information to another in such a way that porcesses do not get in each other's way
- how to maintain proper sequencing when dependencies are present between processes.

THe same problems apply to inter-thread communication because by design, threads communicate via shared memory. For one process to **pass information** to another, it can...

- pass messages through the kernel
- share memory (which can be setup with system calls)
- share a file
- use asynchronous signals or alerts
- use named pipes (which is just a message queue in side the kernel essentially)

4.2 Race Conditions

A **race condition** occurs when two or more processes/threads are reading or writing shared data and the final result depends on which runs precisely when. The **critical section** is the part of the program where the shared data is accessed. Uncoordinated read/write of the data in critical sections may lead to race conditions. Must find some way to prohibit more than one process to execute in its critical section at the same time. A solution is to have **mutual exclusive** access to critical sections: only one processor or thread in a critical section at any time.

4.2.1 Requirements to Avoid Race Conditions

1. No two processes may be simultaneously inside their critical section: **mutual exclusion**
2. No processes running outside its critical sectin may block other processes from entering a critical section: **progress**
3. No process should have to wait forever to enter its critical section (be starved): **bounded waiting**

4.2.2 Disabling Interrupts to Avoid Race Conditions

Right before entering the critical section, the process disables all hardware interrupts and when leaving the critical section, the process reenables all interrupts. When interrupts are disabled, the CPU cannot be switched to another process, so the current process can keep the CPU just for itself while it is in the critical section (unles the process makes a system call). The **advantage** to disabling interrupts is that it is very easy to implement in softare. The **disadvantages** of disabling interrupts is that it is unwise to give user processes the ability to turn off interrupts and this is not suitable for multicore systems. In practice, disabling interrupts is only used on single core machines for mutual exclusion in kernel space.

4.2.3 Using Lock Variables to Avoid Race Conditions

Use a single shared variable (**lock**) between processes. A value of 0 means that no process is in its critical section. A vaue of 1 means that some process is in its critical section. This solution creates a critical section with the lock if the lock is read and written to in two separate lines of code (i.e. if the process context switches after reading the lock but before writing/updating the lock, then both processes may enter critica sectin simultaneously). Continuously testnig a variable until some value appears (**busy waiting**) wastes a lot of CPU time. Note that a lock variable that uses busy waiting is called a **spinlock**.

4.2.4 Strict Alternation to Avoid Race Conditinos

There is a single turn variable shared between processes. A value of 0 means that it is the turn of process 0. A value of 1 means it is the turn of process 1. the turn variable is initially set with 0 or 1 based on how should start first and the processes strictly alternate. This can easily be generalized to n different processes. An **advantage** of this solution is that there are no race conditions. A **disadvantage** of this solution is that it does not achieve progres because a process can be blocked by a process not in its critical section.

4.2.5 Peterson's Solution

Use two shared variables `(turn, flags[])` between processes. `int turn` imposes an access order: which process can enter its critical section next. `bool flags[]` (array) specifies for each process whether it is currently interested in entering its critical section. This array is initialized to be false. Some **advantages** of this solution are that there are no race conditions and it satisfies all three conditions for critical regions. **Disadvantages** are that busy waiting uses a lot of CPU and generalization to more than two processes is complex.

4.2.6 Test-and-Set Lock (TSL) Instruction

Reads and modifies the content of a memory word atomically. Takes arguments `TSL register, memory_address`. Returns in a register the current value of the memory word at `memory_address` and then sets the value of the memory word at `memory_address` to true (usually abstracted as a non-zero value). Operations cannot be interrupted during execution because this locks the memory bus. Essentially, there can only be one read and one read in one atomic step.

4.2.7 TSL Solution

There is a single lock variable in memory at `LOCK_ADDR`, shared between processes. Uses TSL instead of while loops and assignment. **Advantages** of this solution are that there are no race conditions, mutual exclusion is achieved, and progress can be made. **Disadvantages** of this solution are that you must have a TSL instruction implemented in the CPU, there is no bounded waiting, and it has busy-waiting.

4.2.8 Producer-Consumer Problem with `sleep()` and `wakeup()`

Instead of busy-waiting, let the process sleep (requires kernel syscalls). The `sleep()` syscall causes the caller to give up the CPU for some duration of time or until some other process wakes it up. The `wakeup()` syscall causes the caller to wake up some sleeping process. Note that this solution doesn't sovle the priority inversion problem because process B might still end up having to wait a long time for process A to exit the critical section, even if process B is now sleeping

```
Producer
while(1){
    produce an item A;
    if (count == N) //full buffer
        sleep();
    insert item;
    count++;
    if (count == 1) //was buffer empty
        wakeup(consumer);
}

Consumer
while(1){
    if (count == 0) //empty buffer
        sleep();
    remove item;
    count--;
    if (count == N-1) //was buffer full?
        wakeup(producer);
    consume an item;
}
```

In this soution both the producer and consumer may sleep forever. An **advantage** of this solution is that there is no busy-waiting. **Disadvantages** of this solution are that there is a race condition. A wakeup sent to a process that is not (yet) sleeping is lost. Decision to go to sleep and calling sleep() must be paired. So that the test of `count` must be done at the same time as the sleep/wakeup in some way. To achieve this we use **semaphores**...

4.2.9 Sempahores

The purpose of **sempahores** is to solve the lost wakeup problem (explained just above) and avoid race conditions. Semaphores will try to store the number of wakeups. To make a semaphore, define a count variable (the semaphore): `Down(<semaphore>)` is equivalent to consume (decrease) or sleep(), `Up(<semaphore>)` is equivalent to produce (increase) and (potentially) wakeup. `down()` and `up()` are **atomic** (i.e. once the operation has started, no other process can access the semaphore until completed or blocked).

The value of the semaphore... if **positive**, represents the number of resources available and that there are no pending wakeups. If **zero** represents that there are no resources available and no pending wakeups. If **negative** represents that there are no resources available and signifes the number of pending wakeups (number of processes waiting).

4.2.10 Types of Semaphores

Counting semaphore is a sempahore based on the number of controlled resources. The value is an integer.

Binary Semaphore is a sempahore with a value of 0 or 1 which implementes mutual exclusion; this is called a **mutex**. The value is either a 0 or 1. A value of 0 represents **locked** and that the cirtical region is not available. A value of 1 represents **unlocked** and that the critical region is available.

4.2.11 Producer-Consumer Problem using Semaphores

```
Shared variables
const int N=100;
semaphore empty=N, full=0;

Producer
while(1) {
    produce an item A;
    down(emtpy);
    insert item;
    up(full);
}

Consumer
while(1) {
    down(full);
    remove item;
    up(emtpy);
    consume an item;
}
```

The sum of the semaphores muyst be equal to N. There could be a race conditino with the insert and remove actions: if the buffer is not empty or full, then they may both try to nsert/remnove at the same time.

4.2.12 Producer-Consumer Problem using Semaphores and Mutexes

```
Shared variables
const int N=100;
semaphore empty=N, full=0;
mutex mux=1;

Producer
while(1) {
    produce an item A;
    down(emtpy);
```

```
        lock(mux); // down(mux)
        insert item;
        unlock(mux); // up(mux)
        up(full);
    }

    Consumer
    while(1) {
        down(full);
        lock(mux); // down(mux)
        remove item;
        unlock(mux); // up(mux)
        up(emtpy);
        consume an item;
    }
```

The order of the `up()` and `down()` calls is important and can't be changed. **Producer** = `down(empty), down(mux), up(mux), up(full)` **Consumer** = `down(full), down(mux), up(mux), up(empty)`

4.2.13 Monitors

Monitors are a higher-level primitive. Monitors are a programming language construct: a package or module or class. Some **rules for monitors** include:

- only one process/thread can be active in a monitor at any time
- processes cannot access internal data of monitor (fields are private)
- put all critical sections inside monitor methods

A **mutex** is used to achieve mutual exclusion: locked/unlocked automnatically at function/mathod invocation and return. They also have a **condition variable** whcih is used for processes to wait/block when attempting to enter a monitor which is already currently in use and to wake up the process.

`wait(condition)`

- adds process to wait queue for condition variable
- causes the calling process to block
- releases the mutex to allow other processes to enter monitor
- re-acquire mutex once blocked process is woken up

`signal(control)`

- wakes up one blocked process on the condition, if any
- exits the monitor, leaving critical section

4.3 Readers-Writers Problem

Problem is that multiple processes / threads need to acces a shared resources concurrently (but some only need to read it). Mutual exclusion allows one process to access the resource at a time, but can we allow more than one reader at a time? We want multiple readers to access the resource concurrently, but writers must access it with mutual exclusion.

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void) {
    while(TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db)
        up(&mutex)
        read\_data\_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use\_data\_read();
    }
}

void writer(void) {
    while(TRUE) {
        think\_up\_data();
        down(&db);
        write\_date\_base();
        up(&db);
    }
}
```