Operating Systems

Semaphores and Monitors

Busy Waiting Solutions

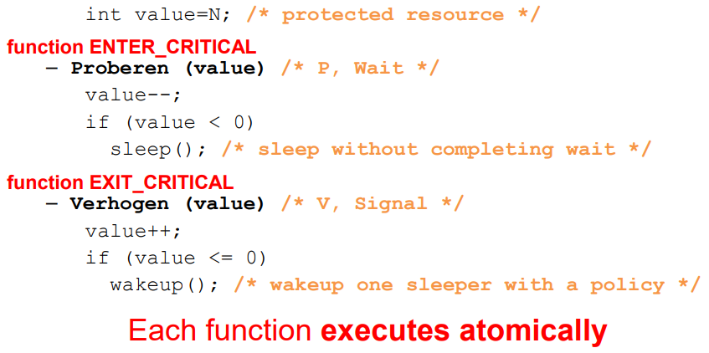
Busy waiting solutions have a series of disadvantages such as wasting CPU time (while waiting) High priority jobs may hinder progress of low priority jobs (in scheduling) they only protect a single resource.

Sleep() and Wakeup()

Instead of busy waiting we can let the process sleep. We introduce two new operating system functions sleep() (caller gives up the CPU for some time/ until a thread/process wake it up) and wakeup() (caller wakes up some sleeping thread process).

In practice a thread/process may call the yield() syscall as sleep() and may be woken with a signal and additionally each also has a kernel-level implementation.

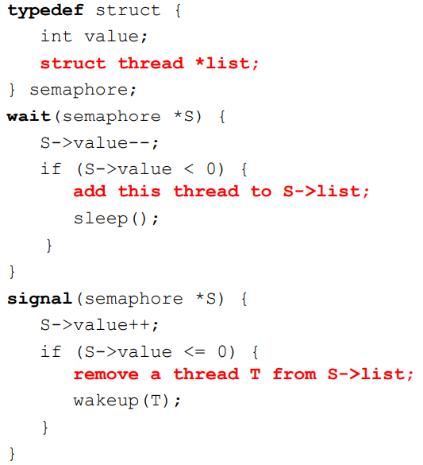
Semaphore

A semaphore is an example of a mechanism that uses this sleep and wakeup system.

Here the semaphore is the variable value initialized to N (the number of resources available). The enter\_critical function is traditionally called Proberen but is known as wait. The exit\_critical function is traditionally called Verhogen but is known as signal.

In wait the value is decremented to indicate one of the resources is no longer available (or if none are available, to indicate that it wants to use a resource) then if the value is less than zero (there wasn’t a resource available) it calls sleep().

In signal, the value is incremented indicating one of the resources is now available (or if the value is negative that the resource has been passed to a waiting thread/process) then if the value is less than or equal to 0 (there was at least one waiting thread/process) it wakes up a thread/process to use said resource.

Semaphore Implementation

The actual implantation is as described above but in wait, before sleeping the thread/process is added to a list of waiting threads/process and in signal a thread is removed from this list. (if the value is negative it means this list is not empty).

This list is a list of user or kernel level TCBs kept in any order (e.g. FIFO). Additionally both wait and signal should be atomic.

How to Provide Atomicity

In order to provide atomicity for wait and signal we need to use a spinlock.

Wait will acquire a mutual exclusion lock (mutex lock) and decrement the semaphore. If the semaphore is available (>0) we decrement the semaphore, release the mutex lock and let the thread continue. If not we place the thread on the associated queue, release the mutex lock and run some other thread.

Signal will acquire a mutex lock and increment the semaphore. If threads are waiting on the associated queue, unblock one and place it on the ready queue, if no threads are on the queue, do nothing. Then release the mutex lock and the signaling thread continues.

Semaphores vs Spinlocks

With Semaphores threads/processes are blocked at the level of program logic by the semaphore wait operation being placed on queues rather than busy-waiting. The scheduler is aware of the thread/process waiting and other tasks can execute.

Busy-waiting may be used for the mutex lock to implement wait and signal as these are very short critical sections (independent of the program logic) and they’re no implemented by the application programmer.

Semaphore Usage

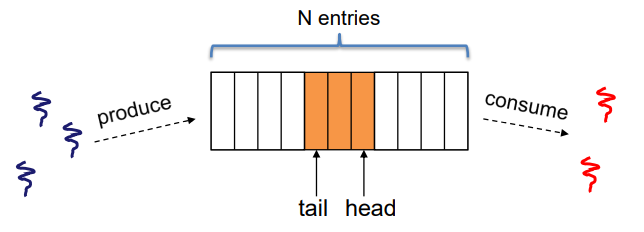
There are three main uses of semaphores: binary semaphores, counting semaphores and synchronization semaphores.

Binary semaphores are where the integer value is initialized and capped to 1. A value of 0 means the lock is held. This behaves exactly like a lock but has no busy waiting.

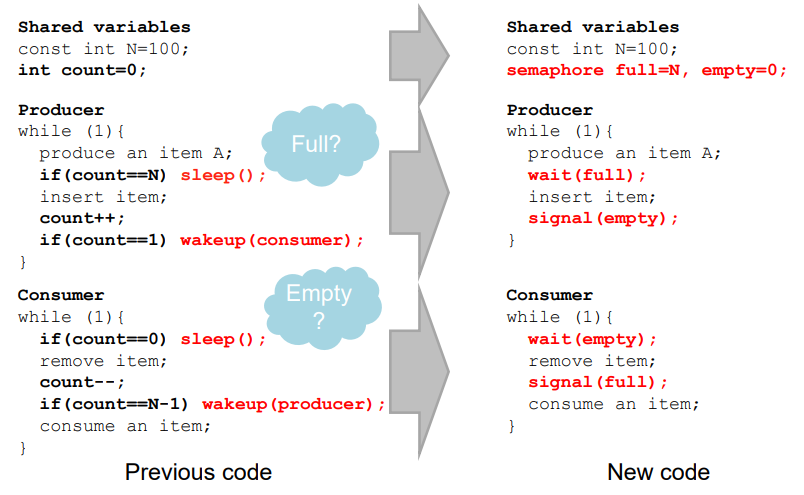
Counting semaphores are where the integer value is initialized and capped to the number of protected resources. Positive value means still resources available, negative or zero values represent how many threads/processes are waiting for resources (and that there are no resources available).

Synchronisation semaphores are where the integer value is initialized to 0. A value of 0 means there are no events pending, a positive value means there are events pending.

Producer-Consumer Problem

Also known as the bounded-buffer, there is a circular buffer in memory with N entries (slots). The producer threads insert entries into it (one at a time) and the consumer threads remove entries from it (one at a time). These threads/processes are concurrent/parallel and so must use synchronisation constructs to control access to the shared variables describing the buffer’s state.

Producer-Consumer Problem Counting Semaphores

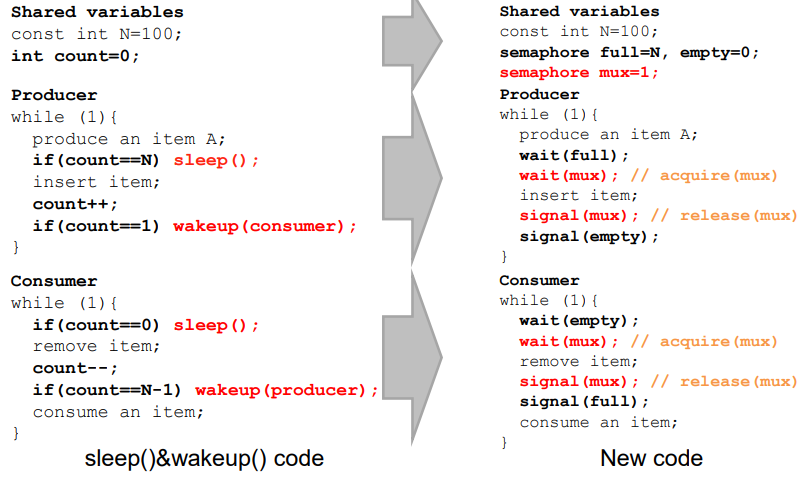
In the producer-consumer problem, count controls two conditions: buffer is empty (if count ==1 or count == 0) and buffer is full (count == N or count = N-1). A semaphore controls a single condition on value, so two semaphores are needed.

Here the checks on count are replaced with these semaphores being used.

In the producer instead of checking if the count == N (and then sleeping) we can simple wait on the full semaphore, if it’s 0 (indicating all the buffer is 0) the producer will decrement it and sleep exactly as we want. Instead of checking if the counter is 1 and then waking up a consumer, we simply signal the empty semaphore which the consumer will be waiting on. Similar things are done in the Consumer.

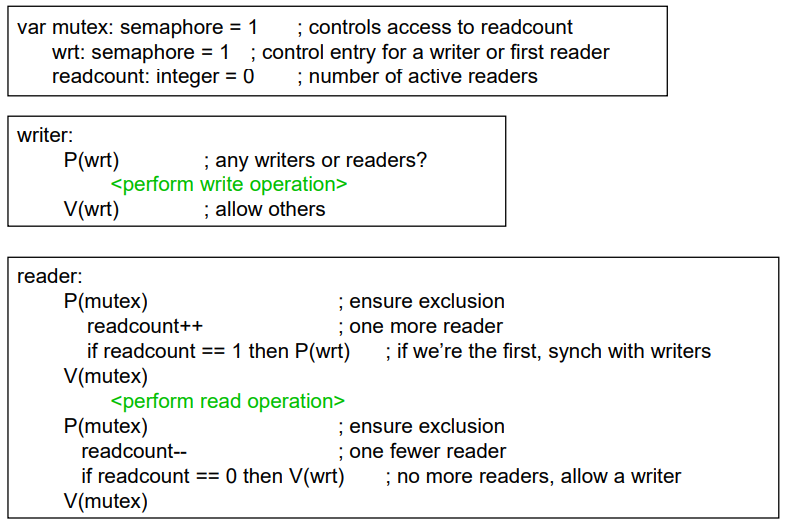
Note that the signal(empty) will keep incrementing the empty semaphore so it represents the number of filled slots in the buffer meaning the consumers will run until it returns to zero (not necessarily only once). The same thing happens with the signal full but in reverse, the full semaphore represents the number of empty slots, the produces decrement this value as they run and the consumers increment it.

Producer-Consumer Problem Counting and Binary Semaphores

We may have multiple threads/processes trying to add/remove different items on the buffer, these are critical operations so to protect the buffer we can use a mutex (implemented as a binary semaphore to prevent busy-waiting) and avoid races.

Here you can see we have added a binary semaphore mux to ensure only one thread/process can operate on the buffer at any time.

Readers/Writers

Sometimes when a single object is shared among several threads/processes sometimes a thread will just read the object, sometimes a thread will update the object. In these cases we can allow multiple readers at a time as we don’t change its state and so no race conditions occur. BU we can only allow one writer at a time as they do change the state (race conditions).

To the right is an implementation of these readers and writers using semaphores (remember P is wait and V is signal).

We have two binary semaphores mutex and wrt.

The writers are only concerned with wrt this is the semaphore indicating the resource is being used by a writer.

The readers on the other hand are concerned with both. The mutex is used to protect the readcount variable that keeps track of the number of active readers. The reader first gets the mutex to allow it to increment the readcount and then check if it’s the only reader, if it is then it will ensure the resource isn’t being written to by waiting on wrt too, once it’s done waiting it will release the mutex allowing other readers to start the read (notice wrt is still held). Once the read is complete it will again wait for the mutex to decrement and check if it’s the last reader available, if it is then it will signal wrt making the resource available to the writers again and then signal the mutex releasing readcount.

Notice that only the first reader will block on wait(wrt) if there is a writer, the rest will then block on P(mutex), this means they aren’t all added to the writer queue. Also notice that if the resource is being read and a writer waits for it, then the next reader cannot get in while a writer is waiting. And when a writer exists and both a reader and both a reader and writer are waiting the reader will go next.

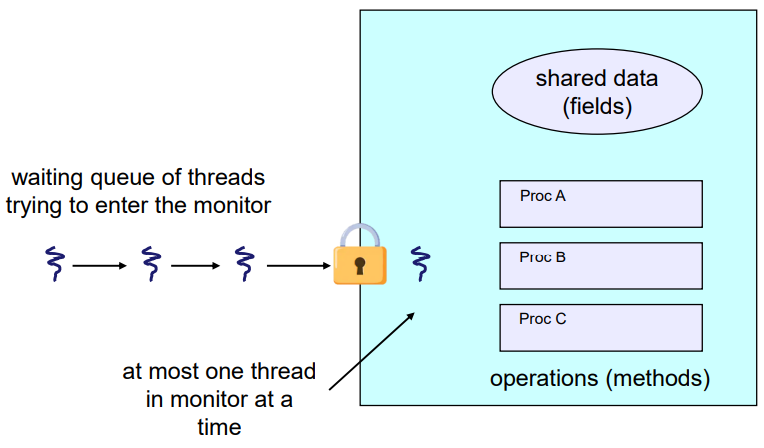
Problems With Semaphores and Locks

We can solve any of the traditional synchronisation problems using semaphores and locks but it’s easy to make mistakes like:

* Shared global variable that can be accessed from anywhere (bad software engineering)
* No connection between the synchronisation variable and the data being controlled
* No control over their use, no guarantee of proper usage for semaphores will there ever be a signal() and for locks did you lock when necessary and unlock at the right time or at all?
* Prone to bugs, we can reduce the change of bugs by “stylizing” the use of synchronisation and the language is helpful for this.

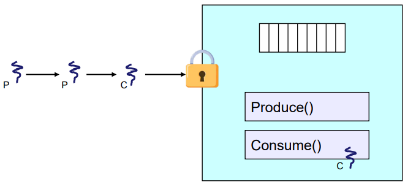
Monitors

Monitors address the key usability issues with semaphores. Monitors are a programming language construct used to support controlled shared data access with synchronisation code being added by the compiler/language VM.

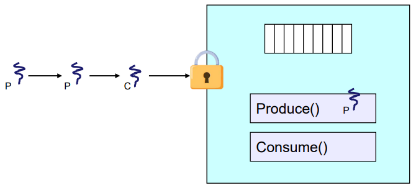
A monitor is an abstract data type/class in which every method automatically acquires a lock on entry and releases the lock on exit. It includes: shared data structures (object fields); procedures that operate on the shared data (object methods); and synchronisation between concurrent execution flows that invoke those procedures.

Data can only be accessed from within protecting the data from unstructured access and preventing ambiguity about synchronisation.

Monitor Facilities

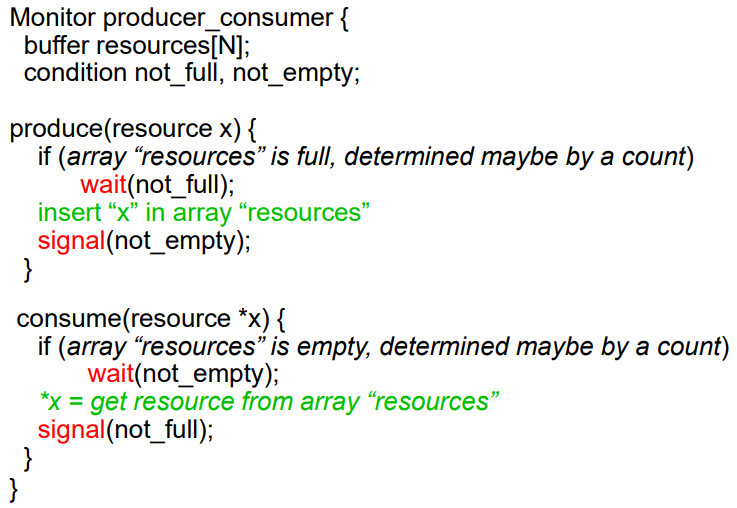
Monitors provide automatic mutual exclusion allowing only one thread to be executing inside it at any time and synchronisation is implicitly associated with the monitor. If a second thread tries to execute a monitor procedure it blocks until the first has left the monitor, this is more restrictive than semaphores but is also easier to use (most of the time)

Problem: Producer-Consumer Scenario

Consider the producer-consumer problem implemented with a monitor. Now consider the scenario where the consumer/producer is using the monitor but the buffer is empty/full (respectively) what do we do?

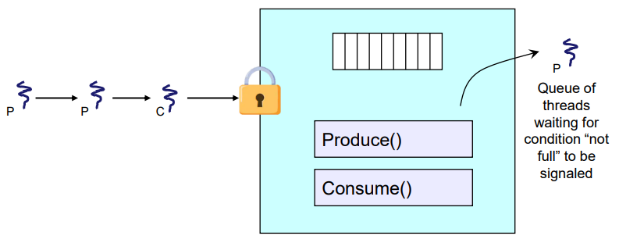
To solve this we can use condition variables. The operations we can perform on these variables are:

* wait(c) which will release the monitor lock so somebody else can get in, then wait for somebody else to signal the condition. Note that condition variables have associated wait queues.
* signal(c) which will wake up at most on waiting thread, a “Hoare” monitor will wakeup the waiting thread immediately and the signaler will step outside the monitor. If there are no waiting thread then the signal is ‘lost’ (has no effect), this is different than semaphores as it doesn’t store the history.
* broadcast(c) which will wake up all threads waiting on c

Producer-Consumer Using Hoare Monitors

Here is an implementation of the producer-consumer problem using Hoare monitors. Much like the semaphore implementation we have two conditions though these indicate the absence of being full or empty, whether the buffer is full or not will still need to be checked, but could use a count or something else.

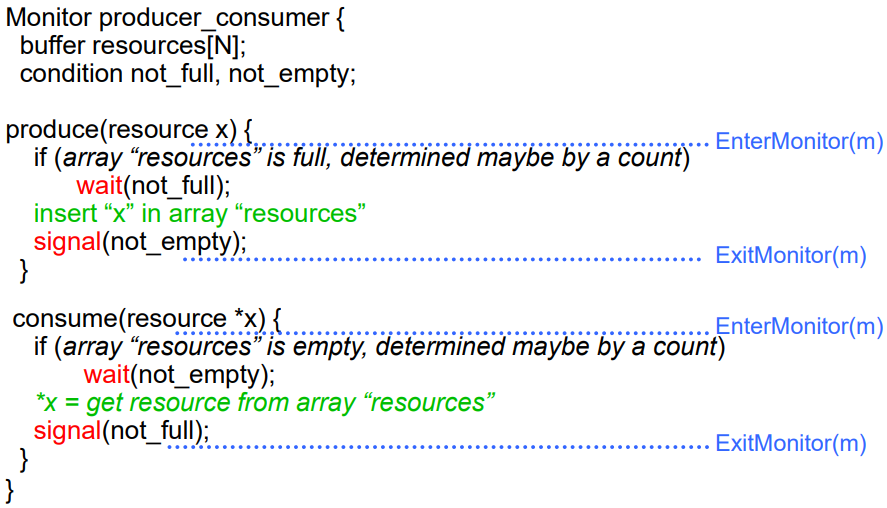
If the buffer is full, the produces will wait until the not\_full condition is true, then will insert whatever it needs to and signal not empty letting the consumers know there is an item to read.

If the buffer is empty, the consumers will wait until the not\_empty condition is true, then get the resource available and signal not\_full letting the producers know they can input another item.

Now when our previous problem arises the produces our producers/consumers will give-up the monitor allowing the next producer/consumer to use it.

Runtime Functions for (Hoare) Monitors

EnterMonitor(m) – guarantees mutual exclusion

ExitMonitor(m) – releases the monitor letting someone else use it

Wait(c) – step out of the monitor until the condition is satisfied (blocking until then)

Signal(c) – if someone is waiting for c and we just made c true and don’t need the monitor anymore, step out and let them run.

EnterMonitor and ExitMonitor are inserted automatically by the compiler guaranteeing mutual exclusion for code inside of the monitor.