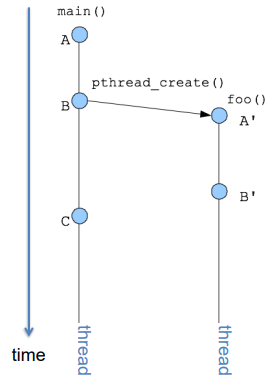
Operating Systems

Synchronisation

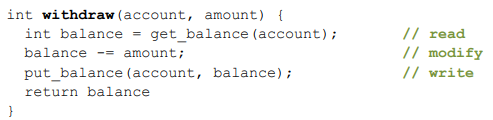
Why Synchronisation

Cooperating tasks need to access the same data and code, when they share data, we can run into issues. Concurrent processes/threads virtually run at the same time on a core (switching between them at any time based on the scheduling). Parallel processes/threads run at the same time of different cores. This can result in data inconsistency (compromising the integrity of the data), to prevent this we need to ensure ordered execution of cooperating tasks when operating on shared data, this is known as synchronisation.

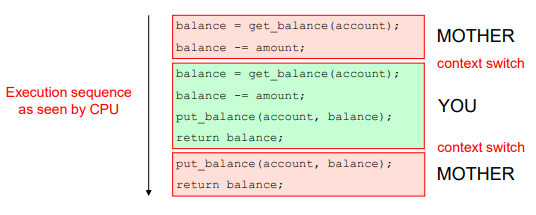
Ordered Execution: Temporal Relations

Instructions executed by a single thread/process are totally ordered meaning there is an order to the instructions within a thread. Instructions can be ordered A < B < C meaning A occurs before B which occurs before C.

In the absence of synchronisation instructions executed by distinct threads/processes must be considered unordered/simultaneous. There is no way to order the instructions between processes B’ < C is wrong as is C < B’. Creation relations always hold though, so A < B < A’ < B’.

Example: Shared Bank Account

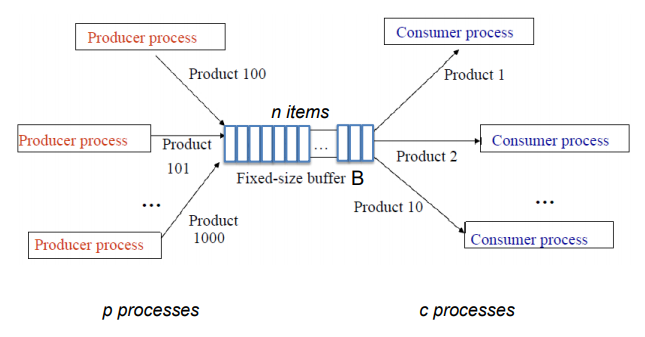
To the right is a function to withdraw money from a bank account. You and your mother share a bank accoutn with a balance of £1500. What happens if you both go to separate ATM machines and simultaneously withdraw £50.

The Bank’s application is multi-threaded with a each ATM performing its withdrawal on a separate thread (on the same server). Each thread can context switch after each instruction.

Think about the interleaved execution of the two threads:

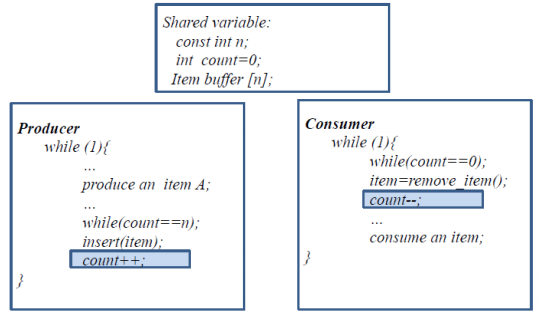
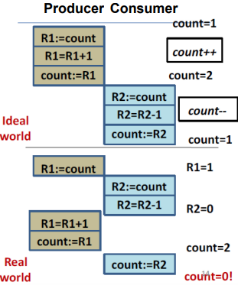
Here you can see, your mother gets the balance and subtracts the amount. Then you get the balance and subtract the amount, then put the balance back and return. Then your mother puts her calculated balance back and returns.

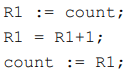
In correct execution the balance would have decreased by £100, but in this case it only decreases by £50. This is because a local copy of the data is made, then edited then sent back to the server, and the way in which it was interleaved, both you and your mother get the same initial unedited amount.

Producer-Consumer Problem

Also known as the bounded-buffer problem:

We have a fixed-size buffer B with n elements, p producer processes and c consumer processes. The producer and consumer processes share the buffer B. The producer processes put info into the buffer and the consumer processes take data out of the buffer.

Single-Producer Single-Consumer

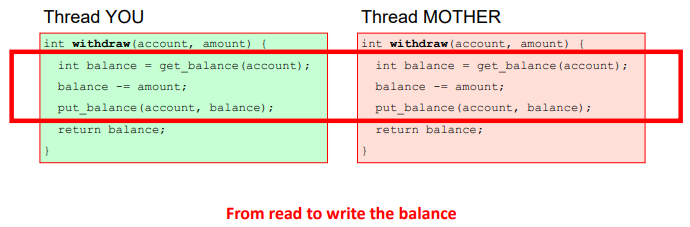
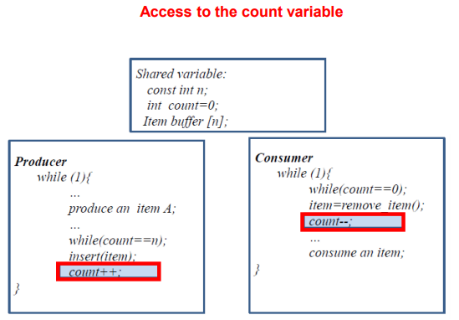
From the code on the right there is no guarantee that the count++; and count—; instructions are executed as single machine instructions (in fact they likely won’t). Count++ (and --) is likely to have an assembly implementation of:

In an ideal world the execution of this code would not be interleaved with the execution of the count— code, but there is no such guarantee, leading to the same issue as the bank example.

Race Conditions

These are examples of race conditions, formally: when two or more processes read or write shared data at the same time, the final result depending on which runs precisely when. This is the result of concurrent or parallel access being non-deterministic and depends on timing, when context switches occurred and what processes/threads ran at the context switch.

Modelling programs to Solve Race Conditions

The part of the program where the shared data is accessed is known as the critical region or section. Uncoordinated read/writes of the data in the critical section may lead to races. The common patter to identify is when read-modify-write of a shared data (variable, global and heap-allocated variables) occur and in code that can be executed by concurrent or parallel threads.

We have drawn a box around the critical section of the bank’s withdraw function and around the critical section of the producer-consumer problem illustrated.

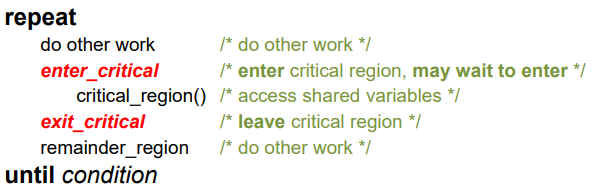
Avoid Race Conditions

We need to find some way to prohibit more than one process from being in its critical region(s) at the same time, known as mutual exclusivity. If one process is using a shared variable, the other process will be excluded from doing the same, only on process/thread is in a critical section at any time.

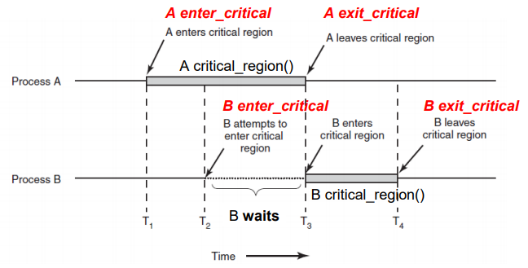
Requirements to Avoid Race Conditions

Critical regions are not enough on their own to prevent race conditions we have a series of requirements:

* Mutual Exclusion – there can be at most one thread/process in the critical section at a time
* Progress – no process running outside its critical region may block other processes
* Bounded waiting – no process should have to wait forever to enter its critical region
* Performance – the overhead of entering and exiting a critical section is small with regards to the work being done within it (with no assumption being made about the processing speed)

How the Code of a Critical Region Looks

To the right is a pseudocode example:

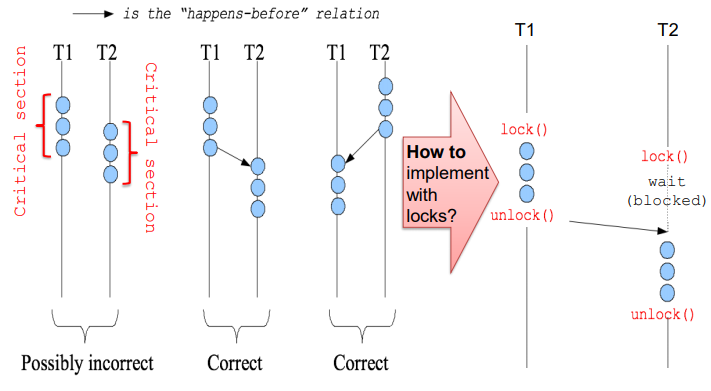
How do we implement enter\_critical and exit\_critical to guarantee mutual exclusion?

Say we have two processes, A and B and A enters its critical region. We need some way to ensure that enter\_critical will check to see if A is in it’s critical region and then wait until it’s finished.

Mechanisms

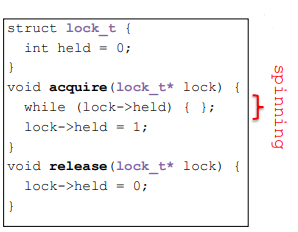
There are many mechanisms to achieve this including Disabling interrupts (needs operating system support and has high overhead, especially on multi-CPU systems), Locks/Spinning locks (Spinlocks, primitive, minimal semantics, used to build others), Semaphores (and non-spinning locks, basic, easy to get the hang of, somewhat hard to program with) and Monitors (higher level, require language support, easier to program with).

Locks

A lock is an object with methods acquire() which obtains the right to enter the critical section (and prevents progress until the lock is acquired) and release() which gives up the right to be in the critical section immediately. Theses are also known as lock() and unlock().

Here you can see two processes/threads T1 and T2 and the possible ways they could occur. We then also see the incorrect situation remedied using locks.

Programmers need to pair up calls to acquire() and release(). From acquire() to release() a thread holds the lock. Acquire() does not return until the caller “owns” (holds) the lock blocking the thread, preventing it from entering its critical section. Finally at most one thread can hold a lock at any time.

How To Implement Locks

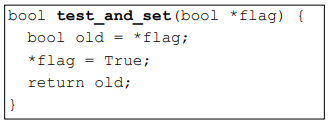
We can implement our acquire and release by spinning on a lock variable:

Here you can see if the lock is held (==1) then acquire will spin until it’s released, at which point it will set the lock to be held and move into the critical section. Here release simply sets the lock’s held flag to false indicating it’s no longer being held and can be acquired.

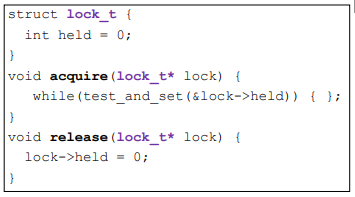
Howeber we can have race conditions on the lock itself! Say T1 sees lock = 0 and moves on to set the lock to held, but before it can do so a context switch occurs and T2 does the same, this will result in both T1 and T2 entering their critical sections.

The implementation of spinning on a lock variable has critical sections, acquire/release must atomic (executes all or nothing). Can we solve this with software? We could use strict alternation, Peterson’s solution (but this doesn’t easily scale to multiple threads) or locking in the OS (only works for user code and is VERY expensive).

In the end we need hardware, atomic instructions (test-and-set, compare-and-swap,…)

Hardware Test-and-Set

The CPU provides the following as a single atomic instruction:

Different CPUs implemente this differently but always in one assembly instruction (TAS register, flag\_address).

Spinning on a lock variable using TAS looks like:

A thread block on an acquire will yield the CPU in two ways, by voluntarily calling yield() (spin-then-block) or involuntarily on a context switch (e.g. timer interrupt).

Spinning on Lock Variables

Spinlocks waste CPU resources, when a thread is spinning on a lock it cannot make any progress and simply burns CPU cycles. To solve this we should use spinlocks as primitives to build higher-level synchronization constructs that ensure we acquisition only happens for a short time.