1 Semaphores

Semaphores are an approach for *mutual exclusion* (and process synchronisation) provided by the operating system

- They contain an integer variable
- We distinguish between binary (0-1) and counting semaphores (0-N)

Two *atomic* functions are used to manipulate semaphores: `wait()` is called when a resource is **acquired**, the counter is **decremented** `signal()/post()` is called when a resource is **released**, the counter is **incremented**

1.1 Approach

Calling `wait()` will block process when the internal counter is negative (*no busy waiting*)

- The process joins the blocked queue
- The process state is changed from running to blocked
- Control is transferred to the process scheduler

Calling `post()` removes a process from the blocked queue if the counter is less or equal to 0

- The process state is changed from blocked to ready
- Different queueing strategies can be employed to remove processes (e.g. FIFO, etc.)

1.2 Specification

The negative value of a semaphore is the **number of processes** waiting for the resource.
`block()` and `wakeup()` are system calls provided by the operating system.
`post()` and `wait()` must be *atomic*.

- Can be achieved through the use of *mutexes* (or disabling interrupts in single CPU systems, hardware instructions)
- *Busy waiting* is moved form the *critical section* to `wait()` and `post()` (which are short anyway critical sections themselves are usually much longer)

1.3 Efficiency

Synchronising code does result in a **performance penalty**.
Synchronise only when necessary and as **few instructions** as possible (synchronising unnecessary instructions will delay others from entering their critical section)

2 Potential difficulties

- **Starvation**: poorly designed queueing approaches (e.g. LIFO) may result in fairness violations
- **Deadlocks**: two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- **Priority inversion** when a high priority process (H) has to wait for a resource currently held by a low priority process (L) and has to wait for the lower priority process to finish.

  Priority inversion can happen in chains, e.g., a H waits for L to release a resource, and L is interrupted by a medium high priority process (M) H waits for L which is interrupted by M *Priority inversion* can be prevented by implementing priority inheritance to boost Ls to the Hs priority.

Reference section

busy waiting

In software engineering, busy-waiting, busy-looping or spinning is a technique in which a process repeatedly checks to see if a condition is true, such as whether keyboard input or a lock is available.