Shared resources

Date @October 31, 2024

A resource is any software structure used by processes to advance its execution. A resource can be private, if dedicated to a particular process, or shared, if it can be used by more tasks. A resource can be also exclusive, meaning that it is a shared resource protected against concurrent accesses, so that only one process at a time can use the resource.

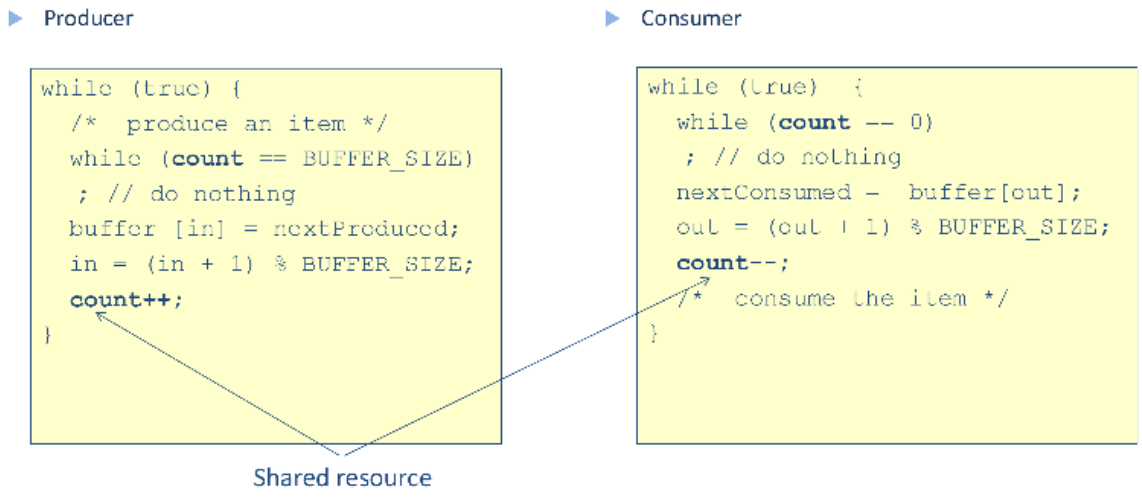
Critical section

A critical section is a piece of software that accesses a shared exclusive resource. When a process runs its critical section, other processes willing to use the shared exclusive resource have to wait.

An application can be structured through multiple concurrent tasks, that can be independent, if cannot affect or be affected by the execution of another task, and cooperating, if can affect or be affected by the execution of another task.

Cooperation has many advantages, that are information sharing, computation speed-up and modularity, but a problem called race problem may occur and refers to the contention of exclusive shared resources.

Race problem



Race condition is originated by tasks whose critical sections are executed concurrently.

There are different ways to solve the race condition:

- stop preemption mechanism, by disabling the interrupt; it is simple but interrupt latency is not predictable and disable preemption may prevent scheduling of higher priority independent tasks

```
do {
    disable_interrupt();
    //critical section
    enable_interrupt();
    //remainder section
} while (TRUE);
```

- use atomic instructions provided by the CPU such as test and set or swap

```
boolean TestAndSet(boolean *target)
{
  boolean rv = *target;
  *target = TRUE;
  return rv;
}
```

```
void Swap(boolean *a, boolean *b)
{
  boolean temp = *a;
  *a = *b;
  *b = temp;
}
```

Spinlock - busy waiting

This solution provides a variable `lock` is created and initialized to `FALSE` .

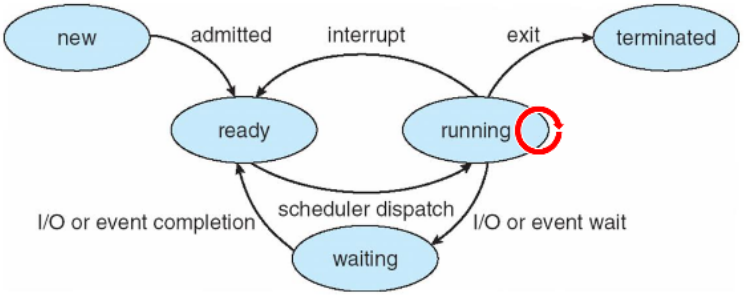
```
do {
  while( TestAndSet( &lock ) == TRUE)
    ;     // do nothing, just wait

  //critical section

  lock = FALSE;

  //remaindor section
} while (TRUE);
```

As long as `lock == TRUE` the process cannot proceed. When running the process remains in that state for its time slice, but it does nothing, so CPU may be wasted.



Spinlock allows a thread to spin (actively wait) until it acquires a lock. It is usually used in multiprocessor systems where waiting is expected to be brief. It offers some advantages as it simple and fast for short waits and no context switching overhead. But it wastes CPU cycles while waiting and it is not ideal for single-processor systems.

Semaphores

A semaphore S contains an integer value and a waiting queue, which is a list of tasks in waiting state, waiting for the semaphore to become available. Two atomic operations are used to modify the value of S:

- wait, in order to decrement the semaphore value and block the task when the semaphore is 0
- signal, to increase the semaphore value and set to ready a previously blocked task

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    add this task to S->list;
    set this process WAITING;
    schedule();
  }
}
```

```
signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
    remove a task T from S->list;
    set T READY;
    schedule();
  }
}
```

So when a semaphore is initialized to 1

```
do {
  wait( S );

  //critical section

  signal( S );

  //remaindor section
} while (TRUE);
```

If `S == 1` the task keeps running, if `S == 0` the task becomes waiting, and when S is incremented, one waiting task moves to ready.

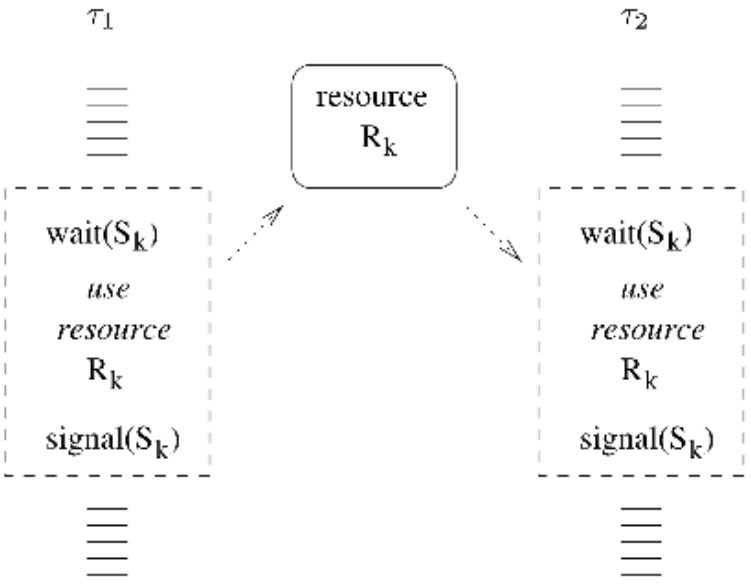
Busy waiting vs semaphore

For small critical sections (few CPU cycle) busy waiting is better.

For big critical sections (many CPU cycle) semaphore is better.

Priority inversion problem

Let's assume that $priority(\tau_1) > priority(\tau_2)$



The blocking time of a task on a busy resource cannot be bounded by the duration of the critical section executed by the lower-priority task.

There are three solutions to this problem:

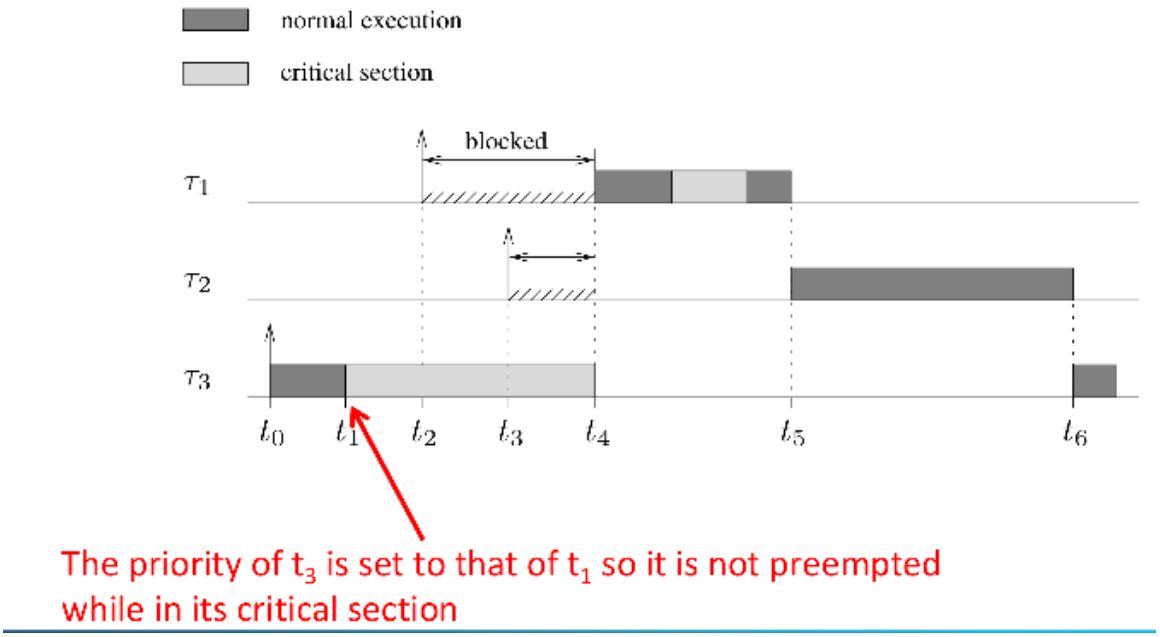
- non-preemptive protocol NPP
- immediate priority ceiling IPC

- priority inheritance protocol PIP

but the assumption of using a rate monotonic scheduling has to be made.

Non-preemptive protocol

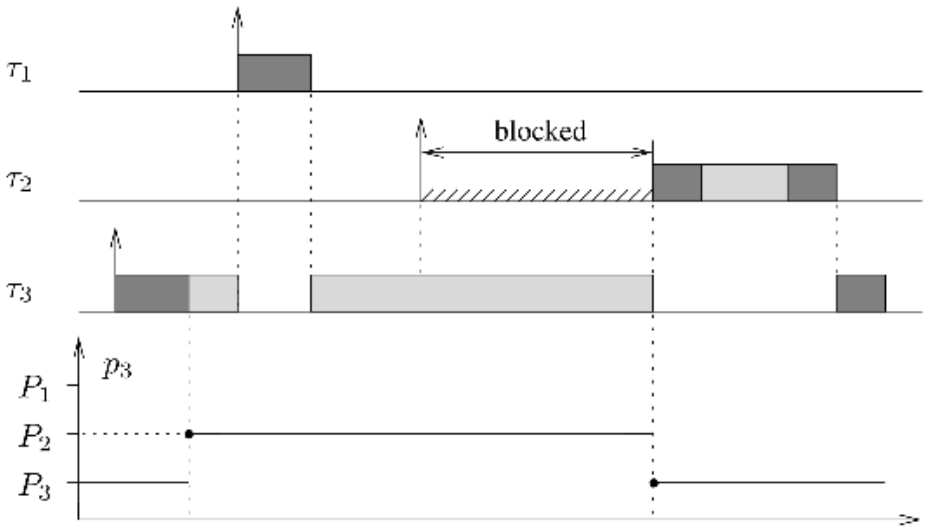
In this solution preemption is not allowed during the execution of any critical section.



The problem is that a high priority task may be blocked for long time even if it does not need the exclusive resource.

Immediate priority ceiling

In this solution the priority of a task that enters the critical section for an exclusive resource R is set to the highest priority among the tasks sharing that resource. While the priority is lowered to the initial value when the task exits the critical section.



Here τ_1 is not blocked as it does not need the exclusive resource. τ_3 get the priority of τ_2 when in critical section.

Priority inheritance protocol

In this solution then a task τ_1 blocks one or more higher-priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. This prevents medium-priority tasks from preempting τ_i and prolonging the blocking duration experienced by the higher-priority tasks.