

Software Solution : Algorithm 5

Using flags with turns, aka Peterson's solution

A process sets its flag to indicate intent.

A process then sets/clears the turn flag

Process 0

beginning section

```
lock1 = 1
turn = 1
WHILE ((lock2 == 1) and (turn == 1))
  DO nothing
END WHILE
critical section
```

```
lock1 = 0
```

remainder section

Process 1

beginning section

```
lock2 = 1
turn = 0
WHILE ((lock1 == 1) and (turn == 0))
  DO nothing
END WHILE
critical section
```

```
lock2 = 0
```

remainder section

Software Solution : Algorithm 5 cont.

Processes fulfil mutual exclusion

- process 0 blocks when
 - $(lock2 == 1) \text{ and } (turn == 1)$

- process 1 blocks when
 - $(lock1 == 1) \text{ and } (turn == 0)$

No deadlock as one part of the condition must be false for one process.

No strict turn taking, except under contention.

Works at the machine code level.

Generalizes to any number of processes, but fairness is difficult.

What if processes cache a local copy of `turn` ?

Low Level Inter-Process Communication (IPC)

Is needed because there are problems with the software algorithms.

They rely on 'busy waiting', wasting cpu time.

All details have to be implemented by programmer.

No way to enforce use of the protocol.

It's all too complicated.

So get the O/S to provide the functionality needed.

Hence low-level IPC.

IPC & Synchronization : 1

UNIX provides :

□ pipes.

○ A circular buffer allowing 2 processes to communicate, A FIFO structure on the producer-consumer model.

○ O/S enforces mutual exclusion, only one process can access the pipe at a time.

○ unamed pipes

▷ shared between related processes only, eg parent-child.

○ named pipes

▷ shared between related & unrelated processes.

▷ `mkfifo a_pipe`

```
prw-r--r--      1 ngunton  users    0 Nov  9 11:47 mypipe
```

IPC & Synchronization 2 :

□ Messages :

○ A block of bytes with an accompanying type. Process has an associated message queue, functionally like a mailbox.

▷ `msgsnd()` , `msgrcv()`

○ Sender

▷ specifies message type when sending.

▷ blocks if destination Q is full.

○ Receiver

▷ retrieves messages either in FIFO order or by type

▷ blocks on empty Q,

▷ non-blocking if retrieving *type* and no message of that type is available

O/S provides the control.

IPC : Shared Memory

□ Shared Memory

- Common block of virtual memory shared by processes.
- No O/S support for mutual exclusion.
 - ▷ could use semaphores, see below.
- Need to communicate the VM space id to the other processes
- For unrelated processes
 - ▷ `shmget()`, `shmadd()`, `shmctl()`
- For related processes you could use the above, or
 - ▷ `mmap()`

Very fast, but needs careful programming to ensure the memory is freed up correctly &c.

IPC : Semaphores, some background.

Originally proposed by Dijkstra in 1965.

- Based on a simple, named, non-negative integer.
- Can only be modified through
 - atomic
 - un-interruptible
- calls, P (or sem_wait)
 - which decrements the semaphore
- and V (or sem_signal)
 - which increments the semaphore
- Processes only need to know the name of the semaphore.

IPC Semaphores 2

Create a binary semaphore 's' and initialize to '1'

Processes cooperate by modifying 's' using `sem_wait(s)` & `sem_signal(s)`

A process wishing to enter a critical region would perform a `semwait(s)`. This checks the value of 's'.

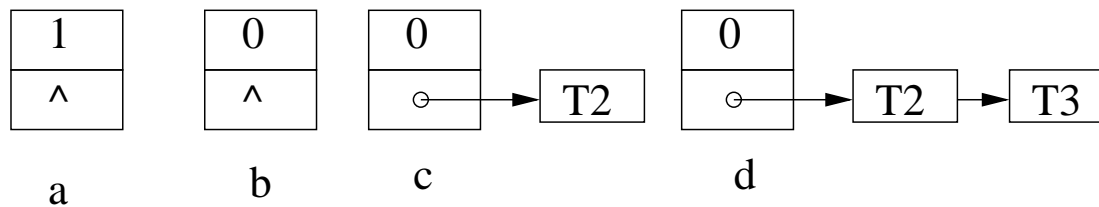
- If 0 then the process blocks.
- If 1 then 's' is decremented and the process enters the critical region

IPC Semaphores 3

A process exiting the critical region performs a `sem_signal(s)`

This checks the queue waiting on this semaphore.

- If the Q is empty then increment 's'
- If Q non-empty then next waiting process is unblocked



Semaphores and Signals in UNIX

The UNIX semaphore holds the following information

- Current semaphore value
- PID of the last process to operate on the semaphore
- length of semaphore queue

SIGNALS

- are sent to processes by the kernel or by other processes.
- similar to an interrupt.
- a process can install its own signal handlers
- a process can 'wait' on a signal
- signals of the same type cannot be queued.

IPC in GNU/Linux

Linux provides all the above plus

- Atomic Operations:

- `atomic_read()`, `atomic_set()` etc.

These calls, 16 in total, provide guaranteed atomic operations on the data type `atomic_t`, effectively a 24 bit integer, and on a bitmap.

More complex structures can be built with these primitives.

- Spinlocks:

- allow implementation of 'busy wait'.
 - only one thread can acquire spinlock
 - only use for short waits.

Compiler needs to know about your kernel to generate appropriate code.

IPC in Windows

XP & W2k3 use synchronization objects and critical section objects for thread IPC

Synchronization Objects:

□ WaitForSingleObject()

- returns when specified object is in the signal state
- time-out (non blocking) occurs,
 - ▷ time-out can be INFINITE

□ Objects include, amongst others,

- Mutex
- Semaphore
- Signal
- File change
- Console input

Windows 2

□ Critical Section Objects

- only apply to threads of a single process
- faster than the equivalent Synchronization Objects
- EnterCriticalSection()
 - ▷ thread blocks indefinitely waiting for entry
- TryEnterCriticalSection()
 - ▷ non-blocking attempt to enter.