OPERATING SYSTEMS

by
Marwa Yusuf

Lecture 6
Thursday 12-11-2020

Chapter 2 (2.3 to 2.3.3)

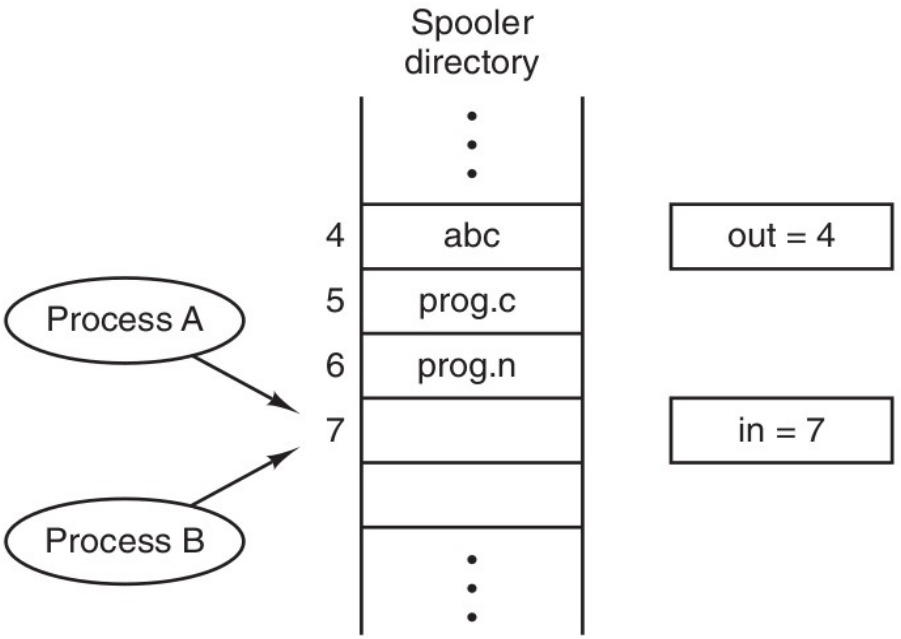
Processes and Threads

Interprocess Communication (IPC)

- Ex: shell pipeline (ls | grep Music)
- 3 issues:
  1) How to pass info to another process?
  2) How two processes do not collide with each other (ex: seat reservation)?
  3) Proper sequencing in case of dependencies.
- The last 2 issues apply equally to threads. (why not the first one?)

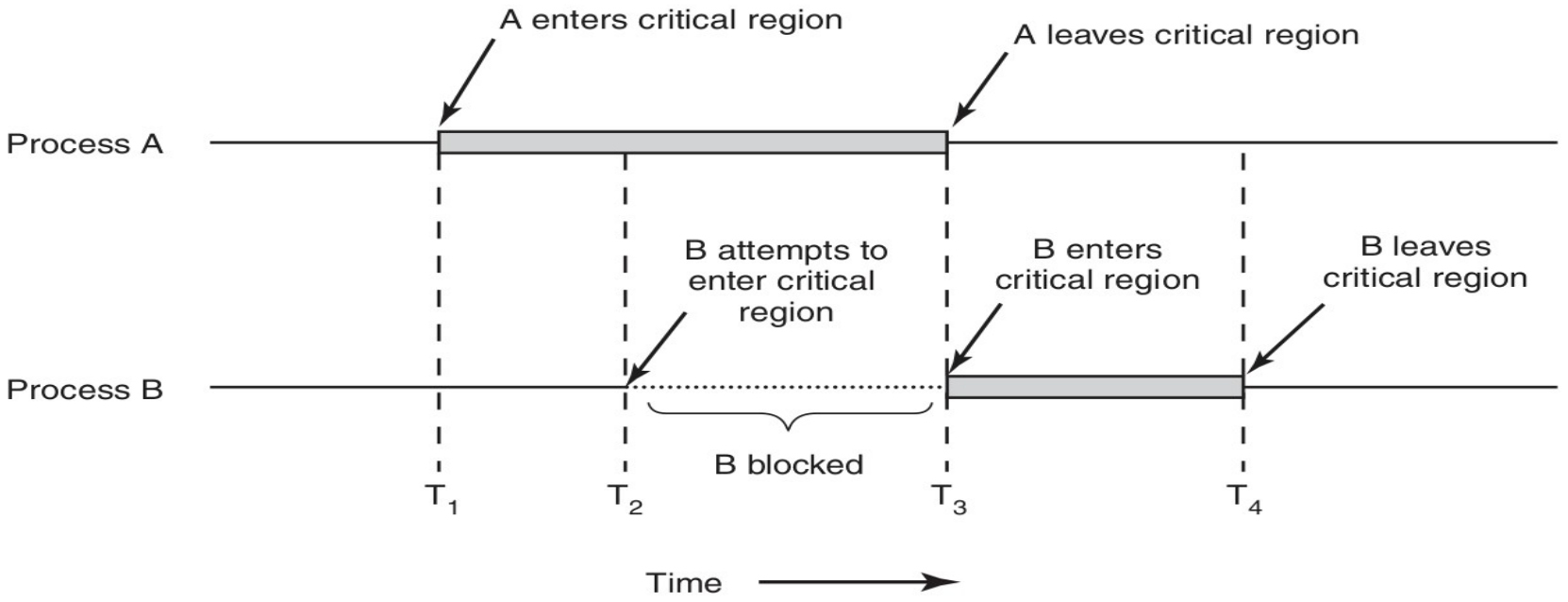
Race Condition

- In some OSs, processes may share some memory (in RAM, kernel data structure, file).
- Ex: Spooler directory.
- X = 7
- Write in 7 “text”
- Y = 7
- Write in 7 “data”
- In = 8



Critical Regions (Sections)

- The solution to race condition is to achieve **mutual exclusion** protecting **critical sections**.



Critical Regions (cont.)

- Just mutual execution is not enough by itself.
- Conditions for correct and efficient cooperation:
  1) Only one process at a time in the critical region.
  2) No assumptions about speed or CPU number.
  3) No process outside critical region can block another process.
  4) No process should wait forever.

Mutual Execution with Busy Waiting

- Some approaches for achieving mutual execution.

Disabling Interrupts

- Once a process enters critical region, it disables interrupts.
- How this guarantees mutual execution?
- This works only on single processor. Why?
- Is it wise to give such a power to some process?
- What about the kernel itself?
- This technique is becoming more unsuitable nowadays. Why?

Lock Variables

```
//P1
If (lock ==0)
    lock = 1;
    //critical section;
    Lock = 0;
Else wait;
```

```
//P2
If (lock ==0)
    lock = 1;
    //critical section;
    Lock = 0;
Else wait;
```

- Does it work?

Strict Alteration

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

- **Busy waiting → spin lock**.

  1)Remember rule 3?: No process outside critical region can block another process. Does it hold here?

Peterson's Solution

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
        int other;                      /* number of the other process */

        other = 1 – process;            /* the opposite of process */
        interested[process] = TRUE;     /* show that you are interested */
        turn = process;                 /* set flag */
        while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
        interested[process] = FALSE;    /* indicate departure from critical region */
}
```

The TSL instruction

- In some computers (specially multi CPU)

```
TSL RX,LOCK   //test and set lock
```

- Executed entirely at one step **(atomically),** locking the but to the memory bus. (different from disable interrupt. How?)
- One possible sol. In the next slide.
- Busy waiting.
- Processes must be cooperating for it to work. How a process can abuse?

The TSL instruction (cont.)

```
enter_region:
        TSL REGISTER,LOCK       | copy lock to register and set lock to 1
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was not zero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

XCHG instruction

- Similar to TSL, exchanges 2 locations.
- All x68 CPU use it.

```
enter_region:
        MOVE REGISTER,#1            | put a 1 in the register
        XCHG REGISTER,LOCK          | swap the contents of the register and lock variable
        CMP REGISTER,#0             | was lock zero?
        JNE enter_region            | if it was non zero, lock was set, so loop
        RET                         | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                | store a 0 in lock
        RET                         | return to caller
```