作業系統
Operating Systems

靜宜大學資訊傳播工程學系
劉國有 助理教授

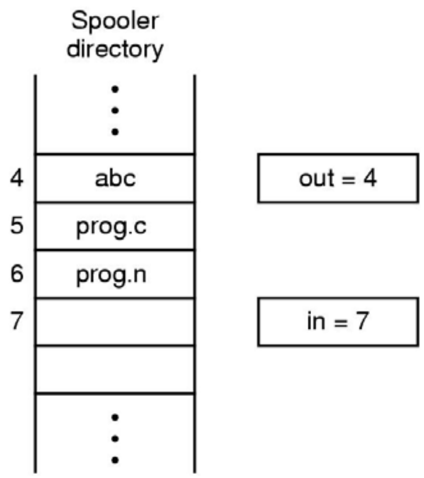
■ 行程間的通訊 (Inter-Process Communication) ■ 競賽情況 (Race Condition)

Inter-Process Communication

- Processes frequently need to communicate with other processes. (ex. cat ch1 ch2 | sort)
- Three issues :
  - How one process can pass information to another?
  - To make sure two or more processes do not get into each other's way when engaging in critical activities. (ex. to grab the last seat on a plane)
  - Proper sequencing when dependencies are present. (ex. Process A produces data and process B prints it)
- For threads, the same problems exist and the same solutions apply.

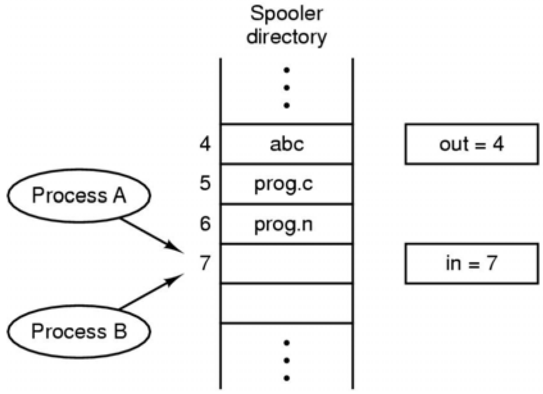
Race Conditions

- In some operating systems, processes that are working together may share some common storage that each one can read and write. (ex. shared memory or a shared file)
- Example: a print spooler (out: next file to be printed, in: next free slot in the directory)



Race Conditions

- Murphy's law: anything that can go wrong will go wrong.
  - 1. Process A reads in and stores the value, 7, in a local variable call next_free_slot.
  - 2. CPU decides that process A has run long enough, so it switches to process B.
  - 3. Process B also reads in, and also gets a 7, so it stores the name of its file in slot 7 and updates in to be an 8.
  - 4. Process A runs again, starting from the place it left off last time.
  - 5. It looks at next_free_slot, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there.
  - 6. It computes next_free_slot+1, which is 8, and sets in to 8.



Race Conditions

- Two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.

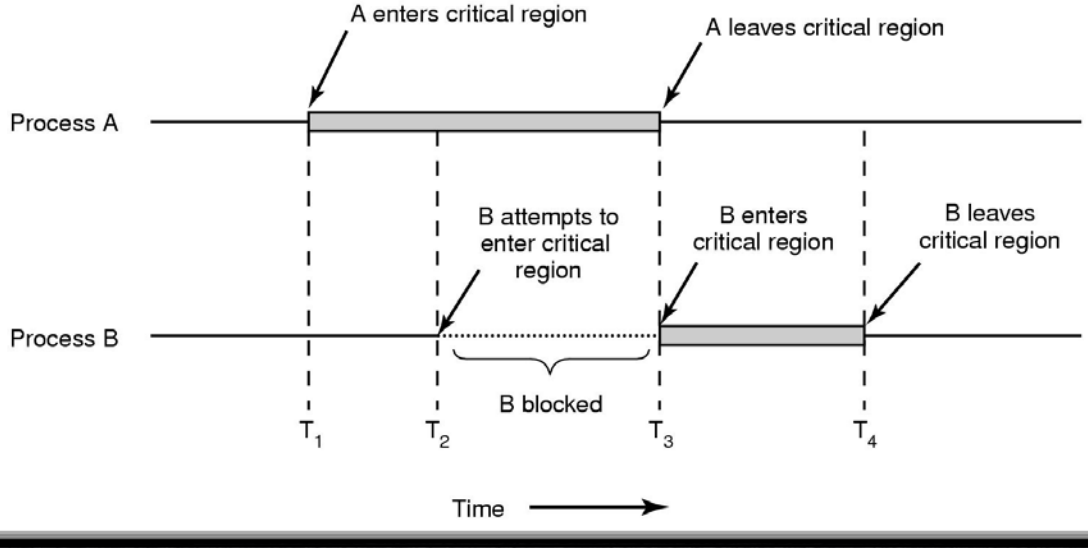
Critical Regions

- How do we avoid race conditions?
  - To prohibit more than one process from reading and writing the shared data at the same time
- What we need is *mutual exclusion*: if one process is using a shared variable or file, the other processes will be excluded from doing the same thing
- *Critical region (critical section)*: the part of the program where the shared memory is accessed
- Race condition avoidance: no two processes were ever in their critical regions at the same time

Mutual Exclusion

- Four conditions to provide a good solution for mutual exclusion:
  - 1. No two processes may be simultaneously inside their critical regions (*Mutual exclusion*).
  - 2. No assumptions may be made about speeds or the number of CPUs
  - 3. No process running outside its critical region may block other processes (*Progress*).
  - 4. No process should have to wait forever to enter its critical region (*Bounded waiting*).

Mutual exclusion using critical regions



Solution 1: Disabling interrupts

- Each process disables all interrupts just after entering its critical region and re-enable them just before leaving it.
- Disadvantages
  - Unwise to give user processes the power to turn off interrupts (i.e. privilege instruction).
  - Do not work in a multiprocessor system: disabling interrupts affects only the CPU that executed the disable instruction.
- Disabling interrupts is a useful technique within the OS but is not appropriate as a general mutual exclusion mechanism for user processes.

Solution 2: Lock Variables

- consider having a single, shared, (lock) variable, initially 0

```
enter_region:
        MOVE REGISTER, lock        | copy lock to register
        MOVE lock, #1              | store a 1 in lock
        CMP REGISTER, #0           | was lock zero?
        JNE enter_region           | waits until lock becomes 0
        RET                        | return to caller; critical region entered
leave_region:
        MOVE lock, #0              | store a 0 in lock
        RET                        | return to caller
```

- Problem: two or more processes will be in their critical region at the same time (violates condition 1)

Solution 3: TSL instruction problem

- (Test & Set Lock)- to solve the "Lock Variable"

```
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 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
```

- The processes must call enter_region and leave_region at the correct times for the method to work

Solution 4: Strict alternation

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

(a) Process 0

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

(b) Process 1

- The integer variable *turn*, initially 0, keeps track of whose turn it is to enter the critical region.
- Violates "progress" condition: a process may be blocked by another process not in its critical region (why?)

Solution 5: 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 */
}
```

Examination of Peterson's solution

- Mutual exclusion
  - Assume that both P0 and P1 want to enter critical region
- Progress
  - Assume that P0 doesn't in critical region, at that time, P1 wants to enter critical region
- Bounded waiting
  - Both P0 and P1 want to enter critical region, if P0 enters the region first and wants to enter again after leaving the critical region, what happen to P1


```
#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 */
}
```

Busy Waiting

- *Busy waiting*: Continuously testing a variable until some value appears is called busy waiting. (It should be avoided, since it wastes CPU time)
- *Spin lock*: a lock that uses busy waiting is called a spin lock.