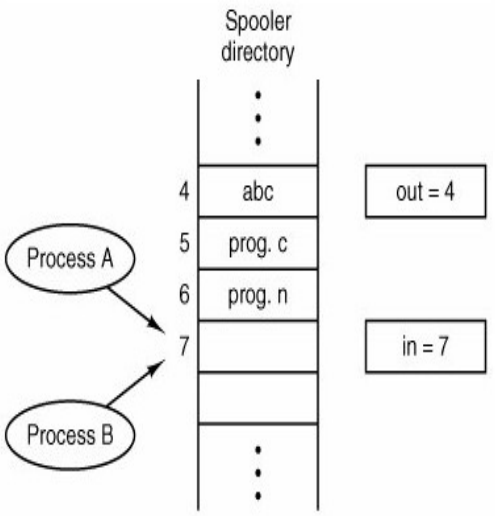
RACE CONDITION AND AVOIDING RACE CONDITIONS

Race Condition:



The situation where two or more processes are reading or writing some shared data & the final results depends on who runs precisely when are called **race conditions.**
To see how interprocess communication works in practice, let us consider a simple but common example, a print spooler. When a process wants to print a file, it enters the file name in a special **spooler directory**. Another process, the **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and removes their names from the directory.

Imagine that our spooler directory has a large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables,
out: which points to the next file to be printed
in: which points to the next free slot in the directory.
At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files to be printed). More or less simultaneously, processes A and B decide they want to queue a file for printing as shown in the fig.

Process A reads in and stores the value, 7, in a local variable called **next_free_slot**. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B.

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. Then it goes off and does other things.

Eventually, process A runs again, starting from the place it left off last time. 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. Then it computes **next_free_slot + 1**, which is 8, and sets **in** to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output.

Spooling: Simultaneous peripheral operations online

Another Example for Race Condition:
When two or more concurrently running threads/processes access a shared data item or resources and final results depends on the order of execution, we have race condition. Suppose we have two threads A and B as shown below. Thread A increase the share variable count by 1 and Thread B decrease the count by 1.

```
Thread A          Thread B
.....             .....
Count++;          Count--;
.....             .....
```

If the current value of Count is 10, both execution orders yield 10 because Count is increased by 1 followed by decreased by 1 for the former case, and Count is decreased by 1 followed by increased by 1 for the latter. However, if Count is not protected by mutual exclusion, we might get difference results. Statements Count++ and Count-- may be translated to machine instructions as shown below:

```
Thread A          Thread B
.....             .....
LOAD  Count       LOAD  Count
ADD   #1          SUB   #1
STORE Count       STORE Count
.....             .....
```

If both statements are not protected, the execution of the instructions may be interleaved due to context switching. More precisely, while thread **A** is in the middle of executing Count++, the system may switch **A** out and let thread **B** run. Similarly, while thread **B** is in the middle of executing Count--, the system may switch **B** out and let thread **A** run. Should this happen, we have a problem. The following table shows the execution details. For each thread, this table shows the instruction executed and the content of its register. Note that registers are part of a thread's environment and different threads have different environments. Consequently, the modification of a register by a thread only affects the thread itself and will not affect the registers of other threads. The last column shows the value of Count in memory. Suppose thread **A** is running. After thread **A** executes its LOAD instruction, a context switch switches thread **A** out and switches thread **B** in. Thus, thread **B** executes its three instructions, changing the value of Count to 9. Then, the execution is switched back to thread **A**, which continues with the remaining two instructions. Consequently, the value of Count is changed to 11!

Thread A		Thread B		
Instruction	Register	Instruction	Register	Count
LOAD Count	10			10
		LOAD Count	10	10
		SUB #1	9	10
		STORE Count	9	9
ADD #1	11			10
STORE Count	11			11

The following shows the execution flow of executing thread ***B*** followed by thread ***A***, and the result is 9!

Thread A		Thread B		
Instruction	Register	Instruction	Register	Count
		LOAD Count	10	10
LOAD Count	10			10
ADD #1	11			10
STORE Count	11			11
		SUB #1	9	11
		STORE Count	9	9

This example shows that without mutual exclusion, the access to a shared data item may generate different results if the execution order is altered. This is, of course, a ***race condition***. This race condition is due to no mutual exclusion.

Avoiding Race Conditions:

Critical Section:
To avoid race condition we need **Mutual Exclusion. Mutual Exclusion** is someway of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same things.

The difficulty above in the printer spooler occurs because process B started using one of the shared variables before process A was finished with it.

That part of the program where the shared memory is accessed is called the **critical region or critical section**. If we could arrange matters such that no two processes were ever in their critical regions at the

same time, we could avoid race conditions. Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data.

(Rules for avoiding Race Condition) <u>Solution to Critical section problem:</u>

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.**
4. **No process should have to wait forever to enter its critical region.**

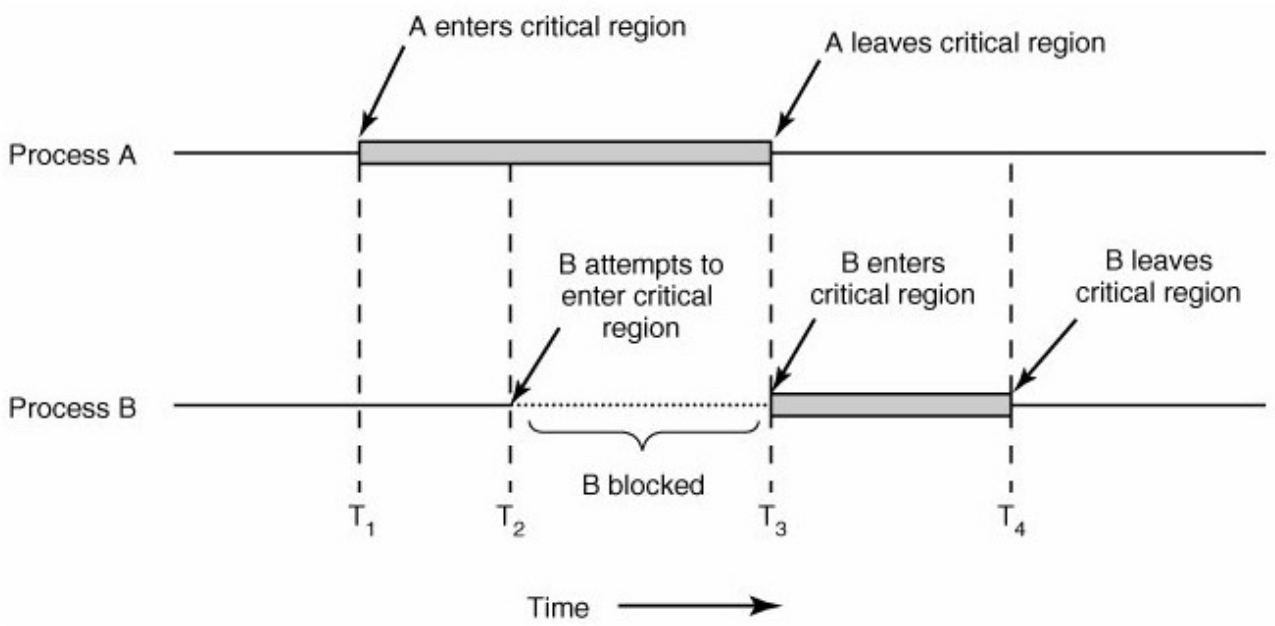


Fig:Mutual Exclusion using Critical Region

Source : http://dayaramb.files.wordpress.com/2012/02/operating-system-pu.pdf