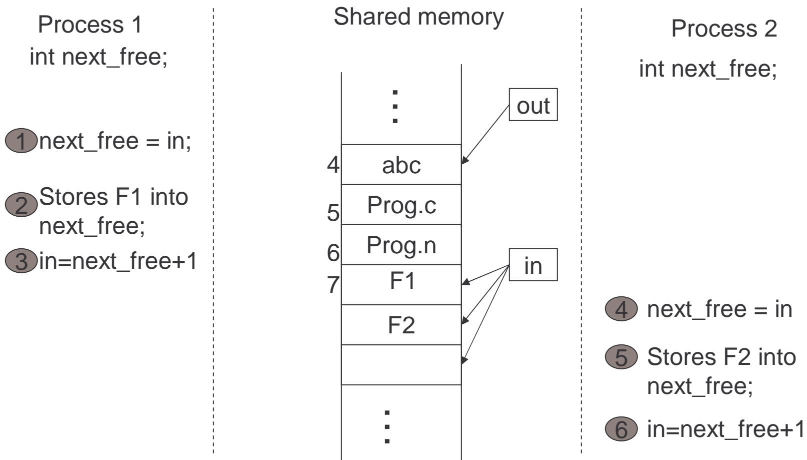
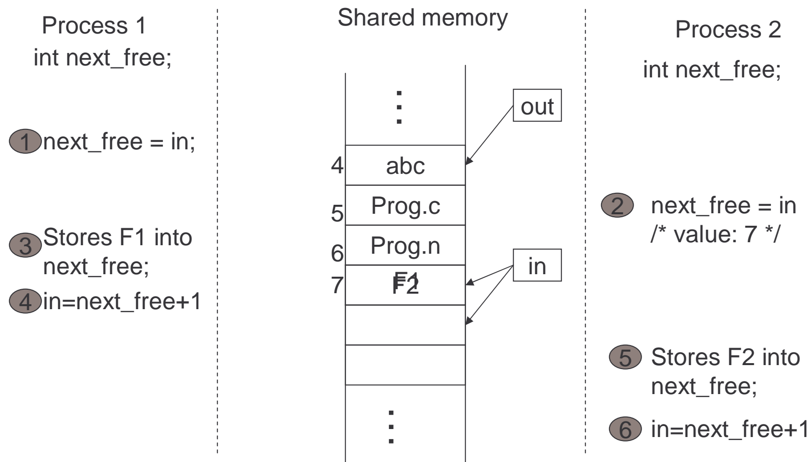
Content

- Critical region and mutual exclusion
- Mutual exclusion using busy waiting
  - Disabling Interrupts
  - Lock Variables
  - Strict Alternation
  - Peterson's solution
  - TSL
  - Sleep and Wakeup
- Summary

Spooling Example: No Races



Spooling Example: Races



Critical Section (Thread/Process)

- N threads/processes all competing to use the same shared data
- Race condition is a situation where two or more threads/processes are reading or writing same shared data and the final result depends on who runs precisely when
- Each thread/process has a code segment, called a critical section, in which shared data is accessed
- We need to ensure that when one thread/process is executing in its critical section, no other thread/process is allowed to execute in its critical section

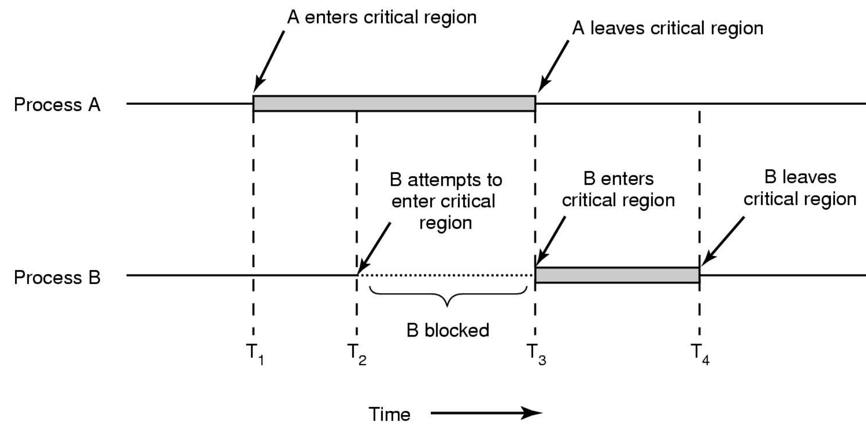
Critical Region (Critical Section)

```
Process {
   while (true) {
        ENTER CRITICAL SECTION
         Access shared variables; // Critical Section;
        LEAVE CRITICAL SECTION
        Do other work
   }
}
```


Requirements for Critical Section

- Mutual Exclusion
  - No other process must execute within its own critical section while a process is in it.
- Progress
  - If no process is waiting in its critical section and several processes are trying to get into their critical sections, then entry to the critical section cannot be postponed indefinitely.
    - No process running outside its critical section may block other processes
- Bounded Wait
  - A process requesting entry to a critical section should only have to wait for a bounded number of other processes to enter and leave the critical section.
    - No process should have to wait forever to enter its critical section
- Speed and Number of CPUs
  - No assumption may be made about speeds or number of CPUs

Critical Regions (2)



Mutual exclusively using critical regions

Synchronization With Busy Waiting

- Possible Solutions
  - Disabling Interrupts
  - Lock Variables
  - Strict Alternation
  - Peterson's solution
  - TSL
  - Sleep and Wakeup

Disabling Interrupts

- How does it work?
  - Disable all interrupts just before entering a critical section and re-enable them just after leaving it.
- Why does it work?
  - With interrupts disabled, no clock interrupts can occur. (The CPU is only switched from one process to another as a result of clock or other interrupts, and with interrupts disabled, no switching can occur.)
- Problems:
  - What if the process forgets to enable the interrupts?
  - Multiprocessor? (disabling interrupts only affects one CPU)
  - Only used inside OS

Lock Variables

```
int lock;
lock=0

while (lock);
lock = 1;
   Access shared variable; //Critical Section
lock = 0;
```

Does the above code work?