Interprocess Communication

Interprocess Communication

- Consider shell pipeline
  - *`cat`* `chapter1 chapter2 chapter3 |` *`grep`* `tree`
  - 2 processes
  - Information sharing
  - Order of execution

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes require a mechanism to exchange data and information

IPC issues

1. How one process passes information to another ?
2. How to make sure that two or more processes do not get into each other’s way when engaging in critical activities?
3. How to do proper sequencing when dependencies are present?

- Ans 1: easy for threads, for processes different approaches (e.g., message passing, shared memory)
- Ans 2 and Ans 3: same problems and same solutions apply for threads and processes
  - Mutual exclusion & Synchronization

Spooling Example: Correct

Process 1

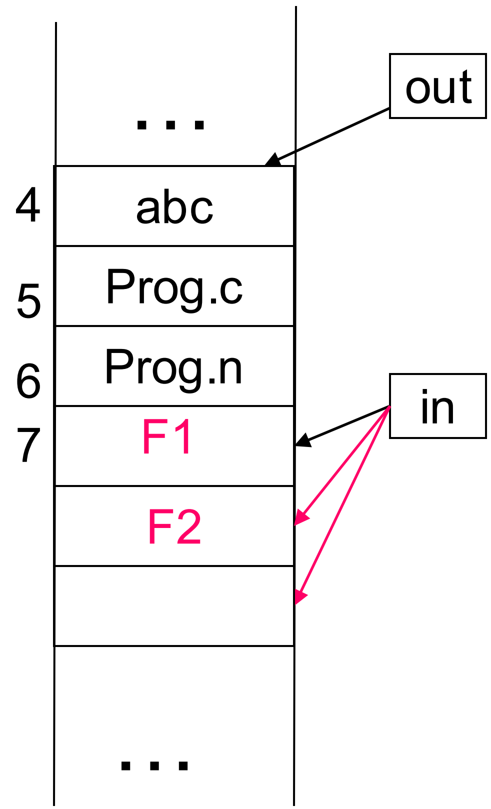
int next_free;

1 next_free = in;

2 Stores F1 into next_free;

3 in=next_free+1

Shared memory



Process 2

int next_free;

4 next_free = in

5 Stores F2 into next_free;

6 in=next_free+1

Spooling Example: Races

Process 1

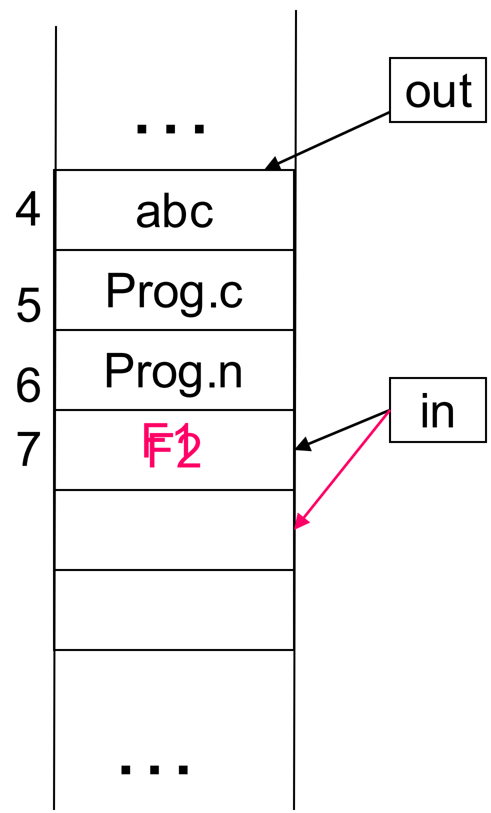
int next_free;

1 next_free = in;

3 Stores F1 into next_free;

4 in=next_free+1

Shared memory



Process 2

int next_free;

2 next_free = in /* value: 7 */

5 Stores F2 into next_free;

6 in=next_free+1

Better Coding?

- In previous code

```
for(;;){
   int next_free = in;
   slot[next_free] = file;
   in = next_free+1;
}
```

- What if we use one line of code?

```
for(;;){
   slot[in++] = file
}
```

When Can process Be switched?

- After each machine instruction!
- `in++` is a C/C++ statement, translated into three machine instructions:
  - load mem, R
  - inc R
  - store R, mem
- Interrupt (and hence process swichting) can happen in between.

Race condition

- Two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
- Very hard to Debug

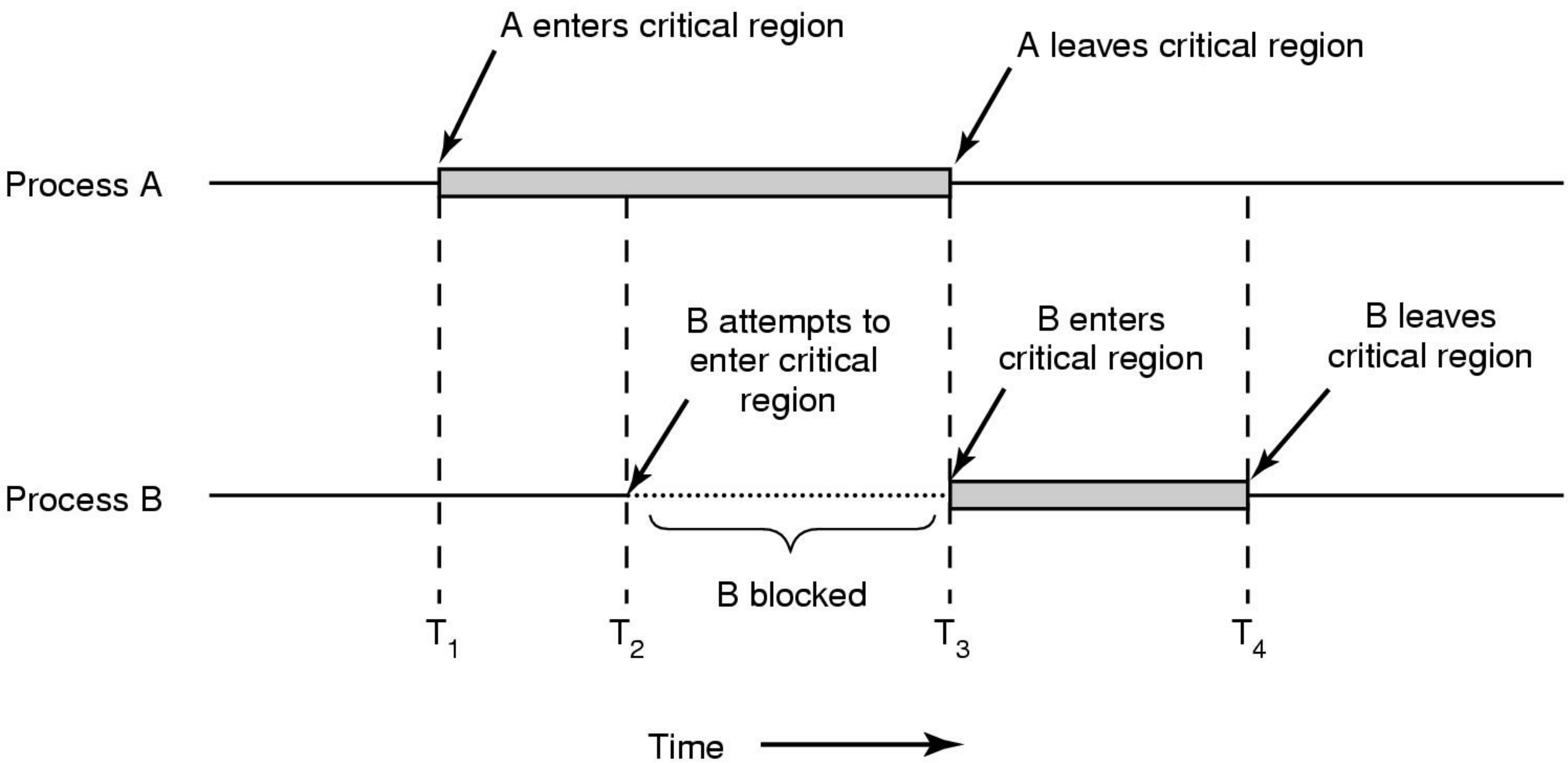
Critical Region

- That part of the program that do critical things such as accessing shared memory
- Can lead to race condition

Solution Requirement

1) No two processes simultaneously in critical region
2) No assumptions made about speeds or numbers of CPUs
3) No process running outside its critical region may block another process
4) No process must wait forever to enter its critical region

Solution Requirement



Sleep & wakeup

- When a process has to wait, change its state to **BLOCKED**
- Switched to **READY** state, when it is OK to retry entering the critical section
- Sleep is a system call that causes the caller to block
  - be suspended until another process wakes it up
- Wakeup system call has one parameter, the process to be awakened.
- Let's illustrate the use of sleep & wakeup with an example: The producer consumer problem

Producer Consumer Problem

- Also called bounded-buffer problem
- Two (or *m*+*n*) processes share a common buffer
- One (or *m*) of them is (are) **producer**(s): put(s) information in the buffer
- One (or *n*) of them is (are) **consumer**(s): take(s) information out of the buffer
- Trouble and solution
  - Producer wants to put but buffer full- Go to **sleep** and **wake up** when consumer takes one or more
  - Consumer wants to take but buffer empty- go to sleep and wake up when producer puts one or more

Sleep and Wakeup

```
#define N 100                    /* number of slots in the buffer */
int count = 0;                   /* number of items in the buffer */
```

```
void producer(void)
{
      int item;

      while (TRUE) {
            item = produce_item( );
            if (count == N) sleep( );
            insert_item(item);
            count = count + 1;
            if (count == 1) wakeup(consumer);
      }
}
```

```
void consumer(void)
{
      int item;

      while (TRUE) {                                          /
            if (count == 0) sleep( );                         /
            item = remove_item( );                            /
            count = count − 1;                                /
            if (count == N − 1) wakeup(producer);
            consume_item(item);                               /
      }
}
```

Producer-consumer problem

Sleep and Wakeup: Race condition

- **Busy waiting problem is resolved but the following race condition exists**
- Unconstrained access to *count*
  - CPU is given to P just after C has read count to be 0 but not yet gone to sleep.
  - P calls wakeup
  - Result is lost wake-up signal
  - Both will sleep forever

Semaphores

- A new variable type
- A kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are *up* and *down* – can't read or write value, except to set it initially

Semaphores: Types

- **Counting semaphore**.
  - The value can range over an unrestricted domain
- **Binary semaphore**
  - The value can range only between 0 and 1.
  - On some systems, binary semaphores are known as mutex locks as they provide mutual exclusion

Semaphores: Operation

- Operation "down":
  - if value > 0; value-- and then continue.
  - if value = 0; process is put to sleep without completing the down for the moment
    - Checking the value, changing it, and possibly going to sleep, is all done as an ***atomic*** *action*.
- Operation "up":
  - increments the value of the semaphore addressed.
  - If one or more process were sleeping on that semaphore, one of them is chosen by the system (e.g. at random) and is allowed to complete its *down*
    - The operation of incrementing the semaphore and waking up one process is also indivisible
  - No process ever blocks doing an *up.*

Semaphores: Atomicity

- Operations must be atomic
  - Two *down*'s together can't decrement value below zero
  - Similarly, process going to sleep in *down* won't miss wakeup from *up* – even if they both happen at same time

Producer & consumer

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */
```

```
void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item( );
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item( );
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

Semaphores in Producer Consumer Problem: Analysis

- 3 semaphores are used
  - *full* (initially 0) for counting occupied slots
  - *Empty* (initially *N*) for counting empty slots
  - *mutex* (initially 1) to make sure that Producer and Consumer do not access the buffer at the same time.
- Here 2 uses of semaphores
  - Mutual exclusion (mutex)
  - Synchronization (full and empty)
    - To guarantee that certain event sequences do or do not occur

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed

Semaphores: Usage

1. Mutual exclusion
2. Controlling access to limited resource
3. Synchronization

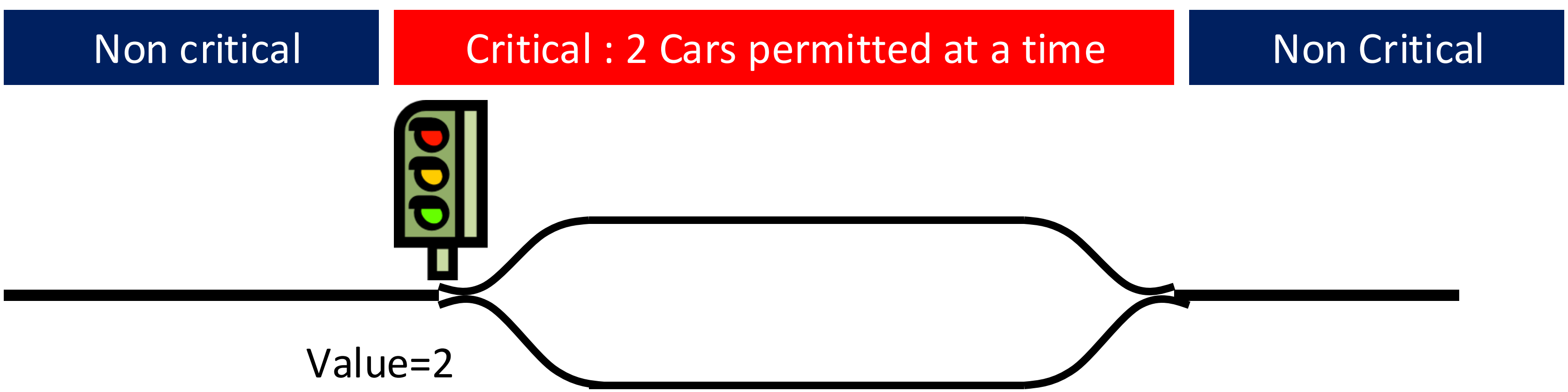
Mutual exclusion

- How to ensure that only one process can enter its C.R.?
- Binary semaphore initialized to 1
- Shared by all collaborating processes
- If each process does a *down* just before entering CR and an *up* just after leaving then mutual exclusion is guaranteed

```
do {
      down(mutex);
      // critical section
      up(mutex);
      // remainder section
} while (TRUE);
```

Controlling access to a resource

- What if we want maximum **m** process/thread can use a resource simultaneously ?
- Counting semaphore initialized to the number of available resources
- Semaphore from railway analogy
  - Here is a semaphore **initialized to 2** for resource control:



Synchronization

- How to resolve dependency among processes
- Binary semaphore initialized to 0
- consider 2 concurrently running processes:
  - P1 with a statement S1 and
  - P2 with a statement S2.
  - Suppose we require that S2 be executed only after S1 has completed.

P1

```
S1;
up(synch);
```

P2

```
down(synch);
S2;
```

Semaphores: "Be Careful"

- Suppose the following is done in Producer's code

```
...
down(&empty)
down(&mutex)
...
```

```
...
down(&mutex)
down(&empty)
...
```

Just the order is reversed

- If buffer full P would block due to down(&empty) with mutex = 0.
- So now if C tries to access the buffer, it would block too due to its down(&mutex).
- Both processes would stay blocked forever: DEADLOCK