Unit 3

Process Concurrency

(Inter-process Communication)

Mutual Exclusion

and

Synchronization

Outline

- Principles of Concurrency
- Mutual Exclusion : Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

1. Multiple Processes

- Central to the design of modern Operating Systems is managing multiple processes
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
- Big Issue is Concurrency
  - Managing the interaction of all of these processes

2. Concurrency

Concurrency arises in:

- Multiple applications
  - Sharing time
- Structured applications
  - Extension of modular design
- Operating system structure
  - OS themselves implemented as a set of processes or threads

Interleaving and Overlapping Processes

- Earlier we saw that processes may be interleaved on uniprocessors

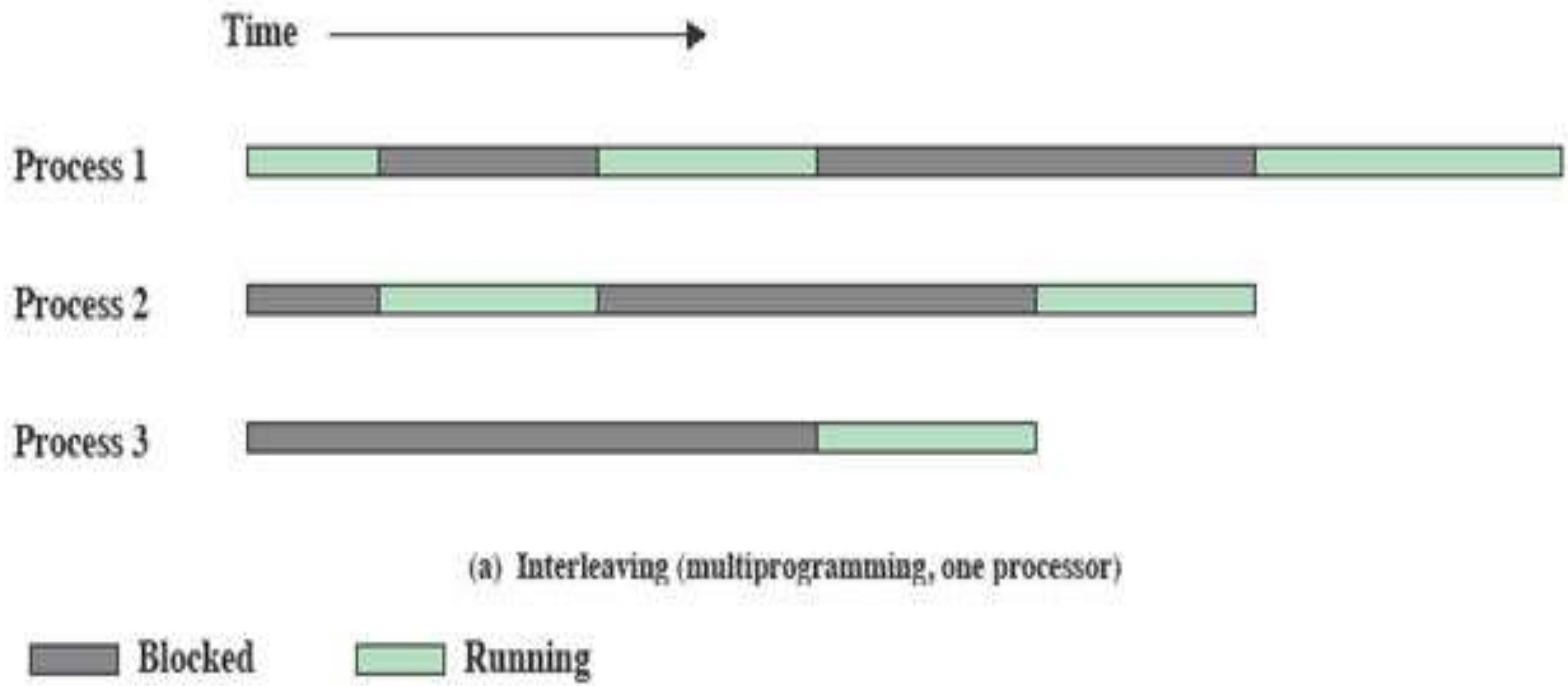


Figure 2.12 Multiprogramming and Multiprocessing

Interleaving and Overlapping Processes

- And not only interleaved but overlapped on multi-processors

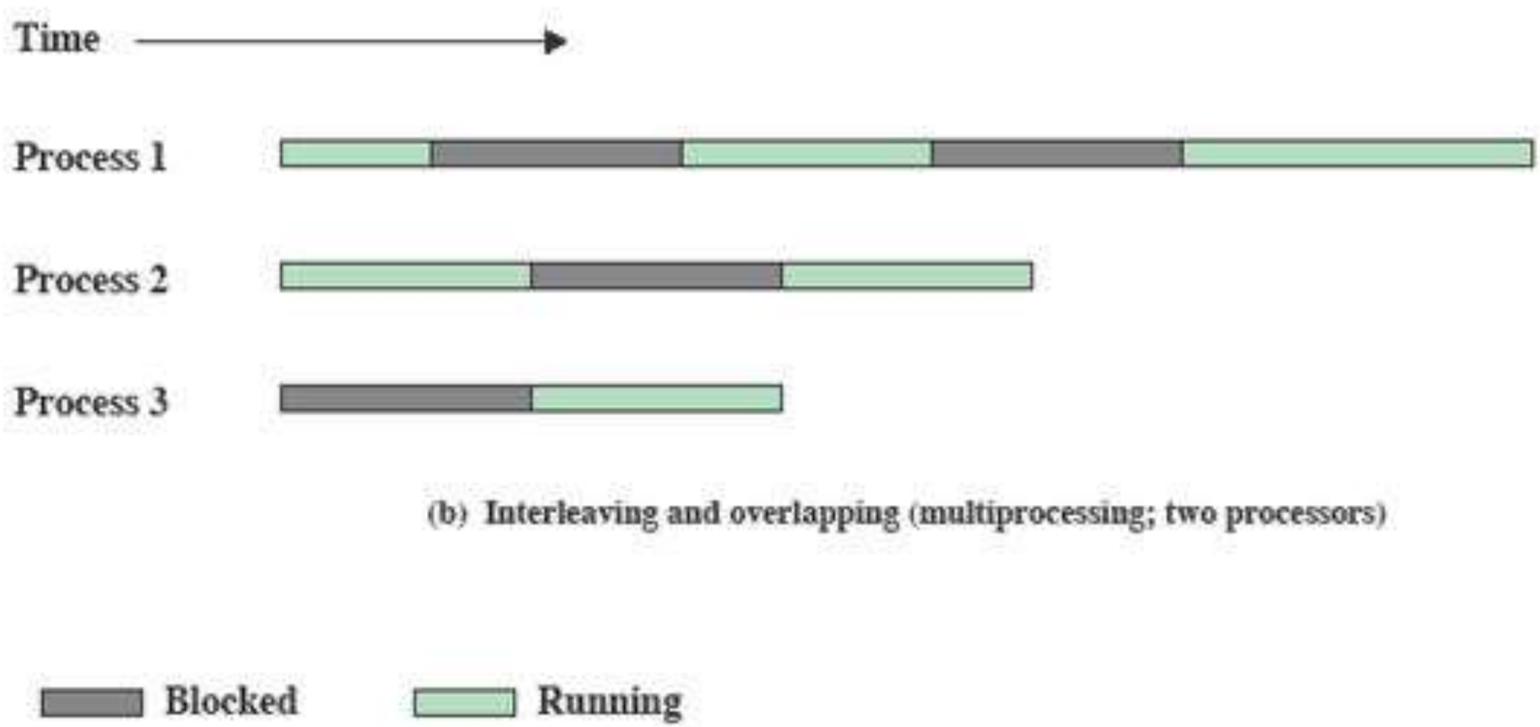


Figure 2.12 Multiprogramming and Multiprocessing

3. Difficulties of Concurrency

- Sharing of global resources (maintain the consistency)
- Optimally managing the allocation of resources (resource blocked)
- Difficult to locate programming errors (running infinite loop)

A Simple Example : Concurrency

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

An example : Call of a function by two Processes

Process P1

```
{
.
.
echo(); // critical section
.
.
}
```

Process P2

```
{
.
.
echo();  // critical section
.
.
}
```

An example : On a Multiprocessor system

Process P1

```
    .
chin = getchar();
    .
chout = chin;
putchar(chout);
    .
    .
```

Process P2

```
    .
    .
chin = getchar();
chout = chin;
    .
putchar(chout);
    .
```

Solution: Enforce Single Access

- If we enforce a rule that only one process may enter the function at a time then :

Scenario

- P1 & P2 run on separate processors
- P1 enters echo first,
  - P2 tries to enter but is blocked – P2 suspends
- P1 completes execution
  - P2 resumes and executes echo

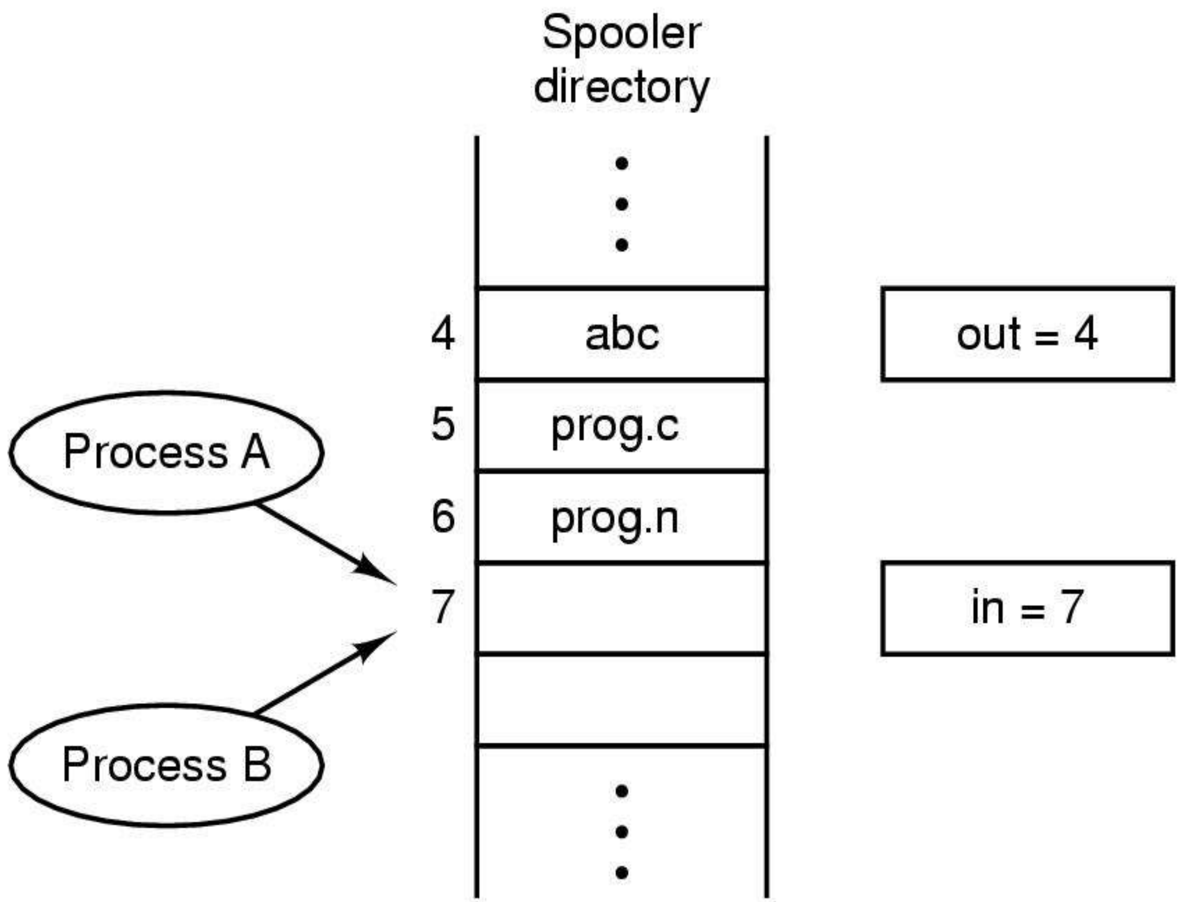
Race Condition

- A race condition occurs when
  - Multiple processes or threads read and write data items (Global resources)
  - Final result depends on the order of execution of the processes.
- The output depends on who finishes the race last.

IPC: Race Condition

Race Condition

The situation where 2 or more processes are reading or writing some shared data is called race condition



Two processes want to access shared memory at same time

4. Operating System Concerns

- What design and management issues are raised by the existence of concurrency?
- The OS must
  - Keep track of various processes
  - Allocate and de-allocate resources
  - Protect the data and resources against interference by other processes.
  - Ensure that the processes and outputs are independent of the processing speed

Process Interaction

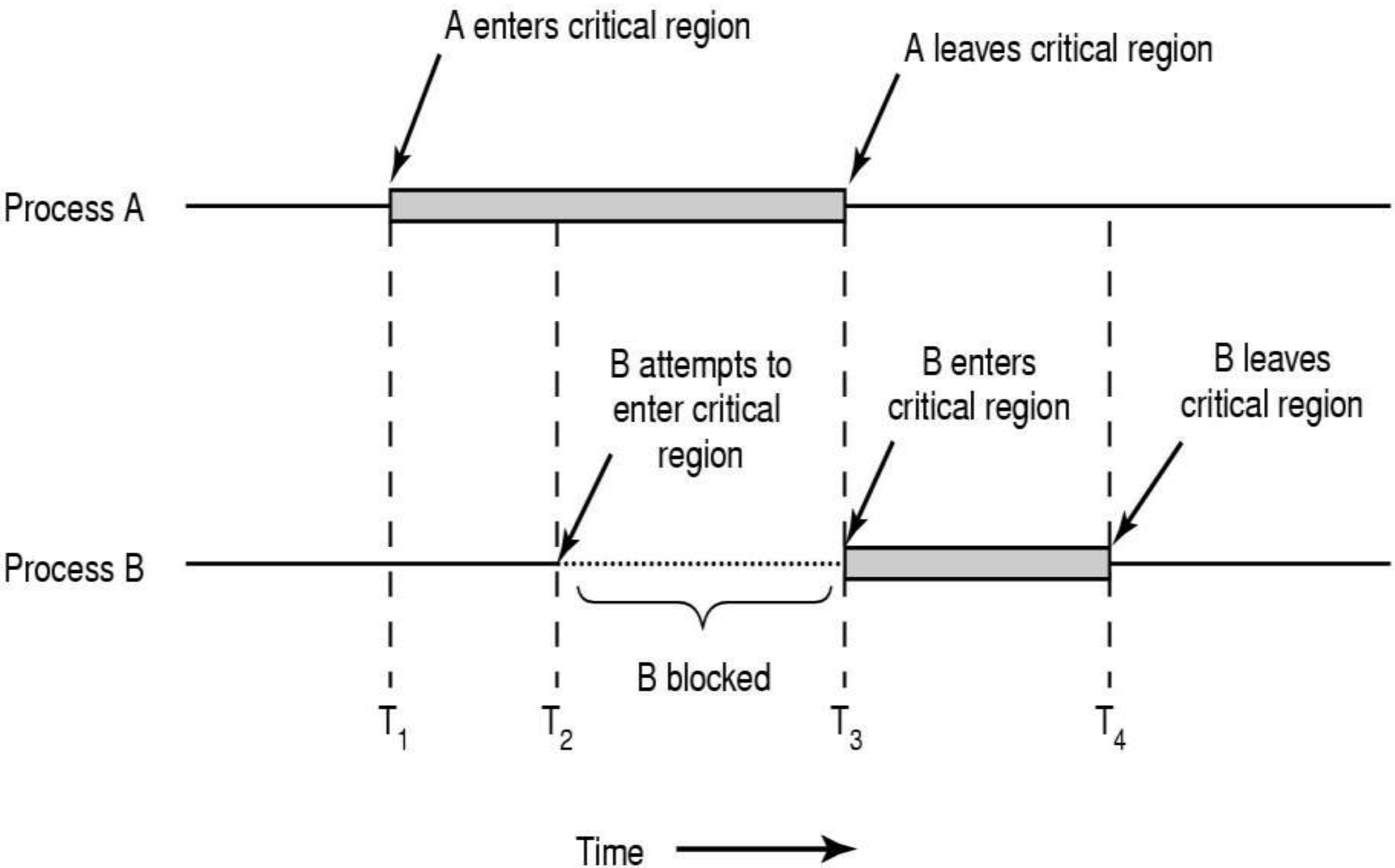
Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence That One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

5. Mutual Exclusion : Requirements

- Only one process at a time is allowed in the critical section for a resource
- A process that executes in its noncritical section must not interfere with other processes
- No deadlock or starvation
- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Mutual exclusion using Critical Regions



Outline

- Principles of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

1. Disabling Interrupts

- Uniprocessors only allow interleaving
- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Will not work in multiprocessor architecture

Pseudo-Code

```
while (true)
{
  /* disable interrupts */;
  /* critical section */;
  /* enable interrupts */;
  /* remainder */;
}
```

Synchronization Hardware : Problems

- Many systems provide hardware support for critical section code
- Uniprocessor – could disable interrupts
  - Currently running code would execute without preemption
  - Not supporting in multiprogramming environment
- Multiprocessors -
  - Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable

Machine Instructions

- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - **Either test memory word and set value**
  - **Or Swap contents of two memory words**

Mutual Exclusion: Hardware Support

- Test and Set Instruction

```
boolean TestAndSet (int lock)
{
    if (lock == 0)
    {
        lock = 1;
        return true;
    }
    else
    {
        return false;
    }
}
```

Mutual Exclusion: Hardware Support

- Exchange Instruction

```
void Swap(int register,
          int memory)
{
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
boolean TestAndSet (int lock) {
if (lock == 0)
{
        lock = 1;
        return true;
}
else
{
return false;
}
}
```

Process - 1

```
do {
        while ( TestAndSet (&lock ))
                ;   // do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

    } while (TRUE);
```

Process - 2

```
do {
        while ( TestAndSet (&lock ))
                ;   // do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

    } while (TRUE);
```

Solution using Swap

```
void Swap(int register,
int memory)
{
int temp;
temp = memory;
memory = register;
register = temp;
}
```

- Method:

1. Shared Boolean variable lock initialized to FALSE;
2. Each process has a local Boolean variable key

Solution:

```
Process - I
do {
        key = TRUE;
        while ( key == TRUE && lock == FALSE)
                Swap (&lock, &key );

                //    critical section

                lock = FALSE;

               //      remainder section

         } while (TRUE);
```

Mutual Exclusion Machine Instructions

- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections

Mutual Exclusion Machine Instructions

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one processes are waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

Outline

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem