Chapter 2
Processes and Threads

Part 1

Topics : Process, Process States, PCB

-- By R.G.B

PROCESS

- Process is an instance of a **program in execution**
- Program is a ***passive entity*** -- file stored on disks
- Process is an **active entity**, with a **program counter** specifying the next instruction to execute and **a set of associated resources**.

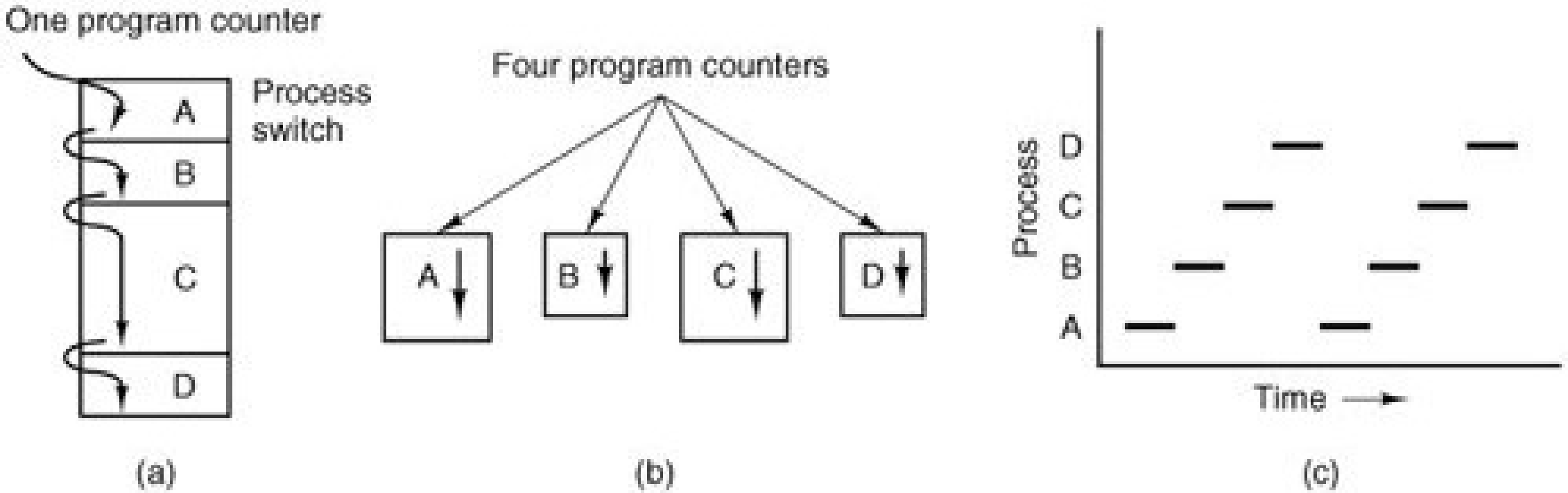
Program→ loaded into memory→ Process

- Process is identified and managed using **PID (Process IDentifier)**

PROCESS MODEL

- Number of **sequential processes**
- A process is just an instance of an executing program, including the current values of the **program counter, registers, and variables**
- Switches back and forth from process to process**(multiprogramming)**
- Collection of processes running in (pseudo) parallel

PROCESS MODEL



Fig(a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at any instant

PROCESS CREATION

- There are four principal events that cause processes to be created:

1. System initialization
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

- **fork –** system call to create process– clone of calling process
- **execve --** Change memory image and run new program

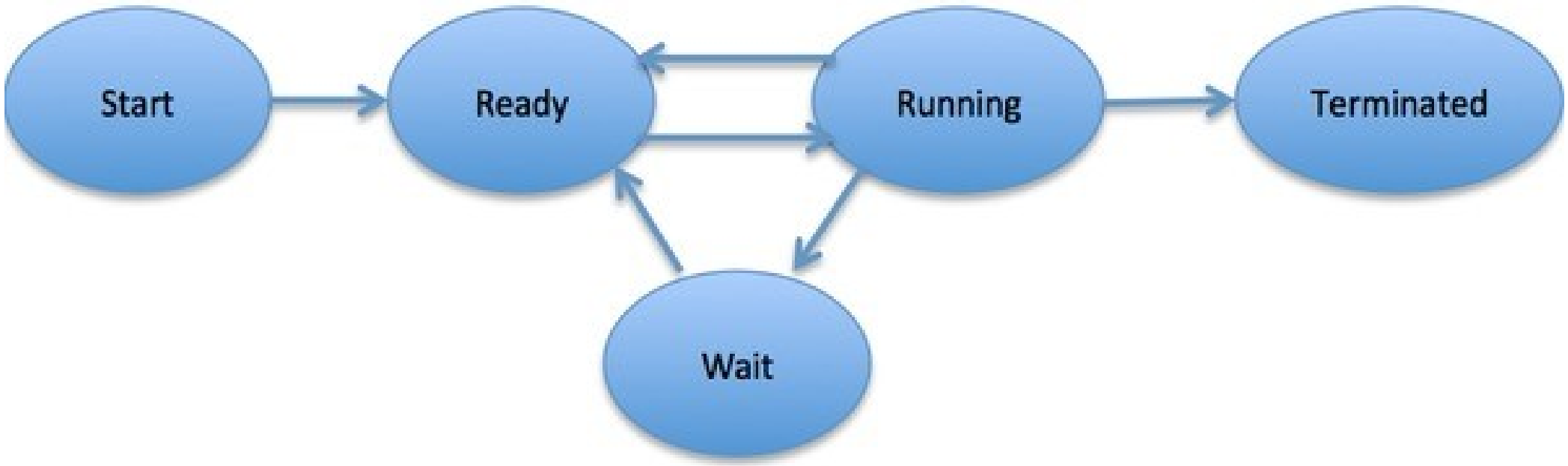
PROCESS TERMINATION

The process has **finished** execution.

Terminate due to one of the following conditions.

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Process Life Cycle (Process States)



Process State 1 Start

This is the initial state when a process is first started/created

Process State 2 Ready

The process is waiting to be assigned to a processor

Ready processes are waiting to have the processor allocated to them by the operating system so that they can run

Process may come into this state after Start state or while running it by but interrupted by the scheduler to assign CPU to some other process.

Process State 3 Running

Instructions are being Executed

Process is assigned to a processor by OS Scheduler

Process State 4 Waiting

Waiting for some event to occur or recourses to become avaliable

Wait for file to become available /Wait for Input

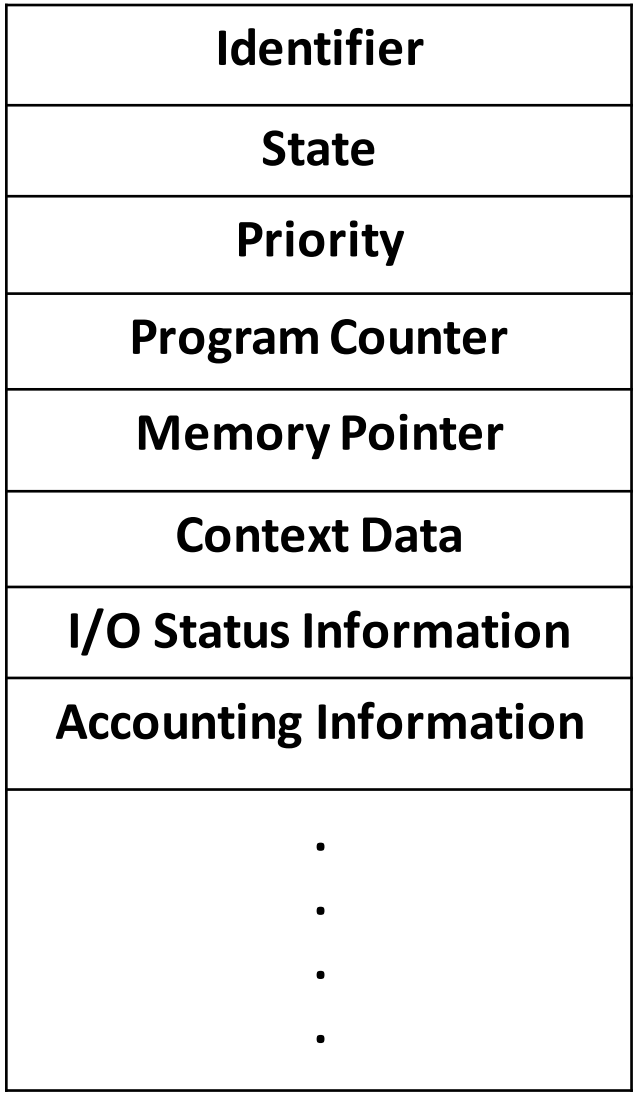
Process State 5 Terminated

Process has Finished Execution (Process Exit)

Removed from main memory

PROCESS CONTROL BLOCK

- Each process represented by PCB/Task CB
- Information regarding **A** process
- Data Structure that represent process in memory
- Fields in PCB are Shown in Figure



PROCESS CONTROL BLOCK (Fields)

- **Identifier**: A unique identifier associated with this process, to distinguish it from all other processes.
- **State**: running /blocked/waiting/new/terminated
- **Priority**: Priority level relative to other processes.
- **Program counter**: Address of the next instruction
- **Memory pointers:** Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
- **Context data**: These are data that are present in registers in the processor while the process is executing.

PROCESS CONTROL BLOCK (Fields)

- **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., tape drives) assigned to this process, a list of files in use by the process, and so on.
- **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

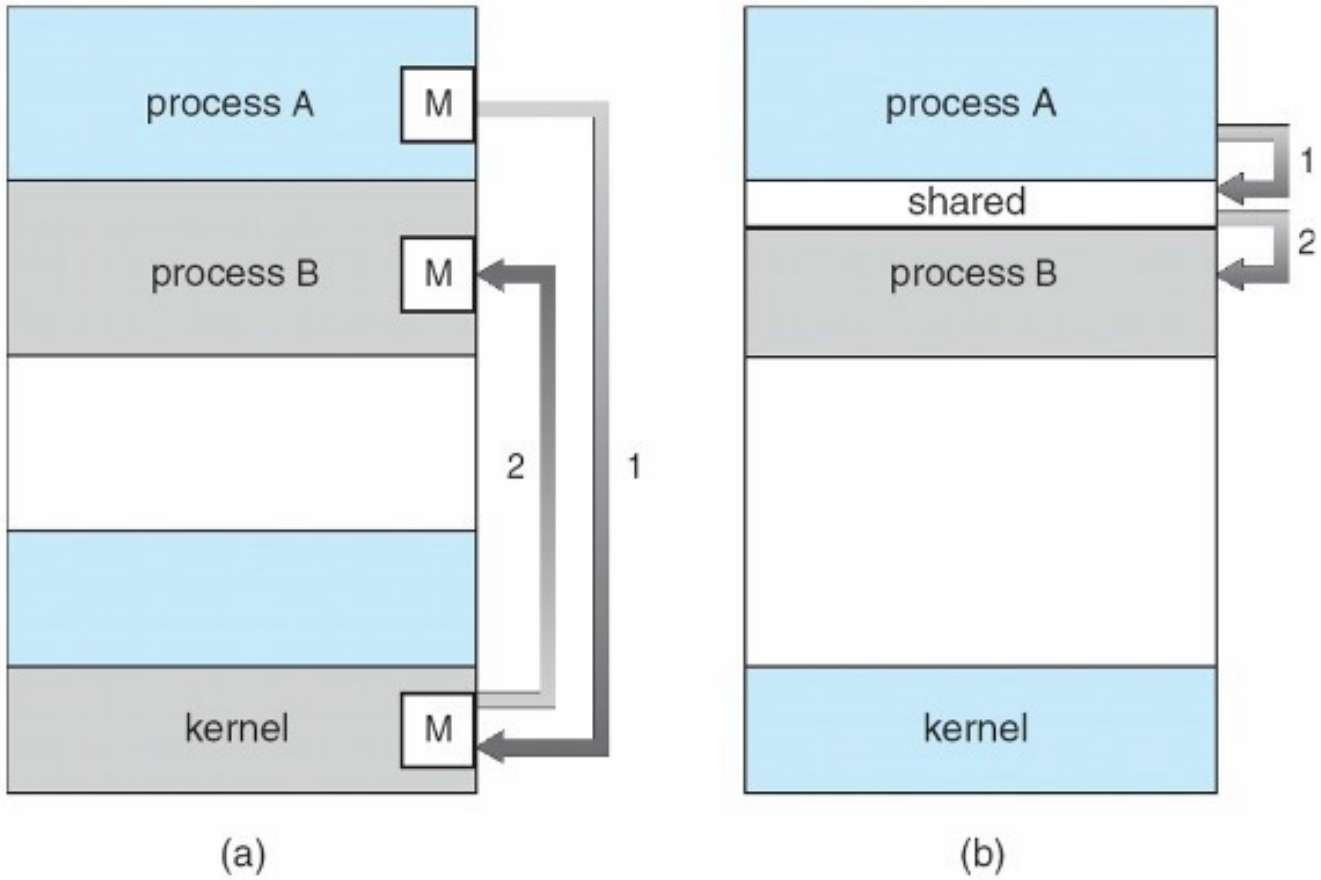
INTER PROCESS COMMUNICATION(IPC)

- Techniques for the **exchange of data** among multiple threads in one or more processes
- Processes may be running on one or more computers connected by a network
- one application to control another application, and for several applications to share the same data without interfering with one another
- Different techniques
  1. **Message Passing**
  2. **Shared Memory**

- A process is independent if it can't affect or be affected by another process.
- A process is co-operating if it can affects other or be affected by the other process.
- Any process that shares data with other process is called co-operating process.
- There are many reasons for providing an environment for process co-operation.
    1. **Information Sharing:** Several users may be interested to access the same piece of information(for instance a shared file).
    2. **Computation Speedup**: Breakup tasks into sub-tasks
    3. **Modularit**y: construct a system in a modular fashion.
    4. **convenience: co-operating** process requires IPC

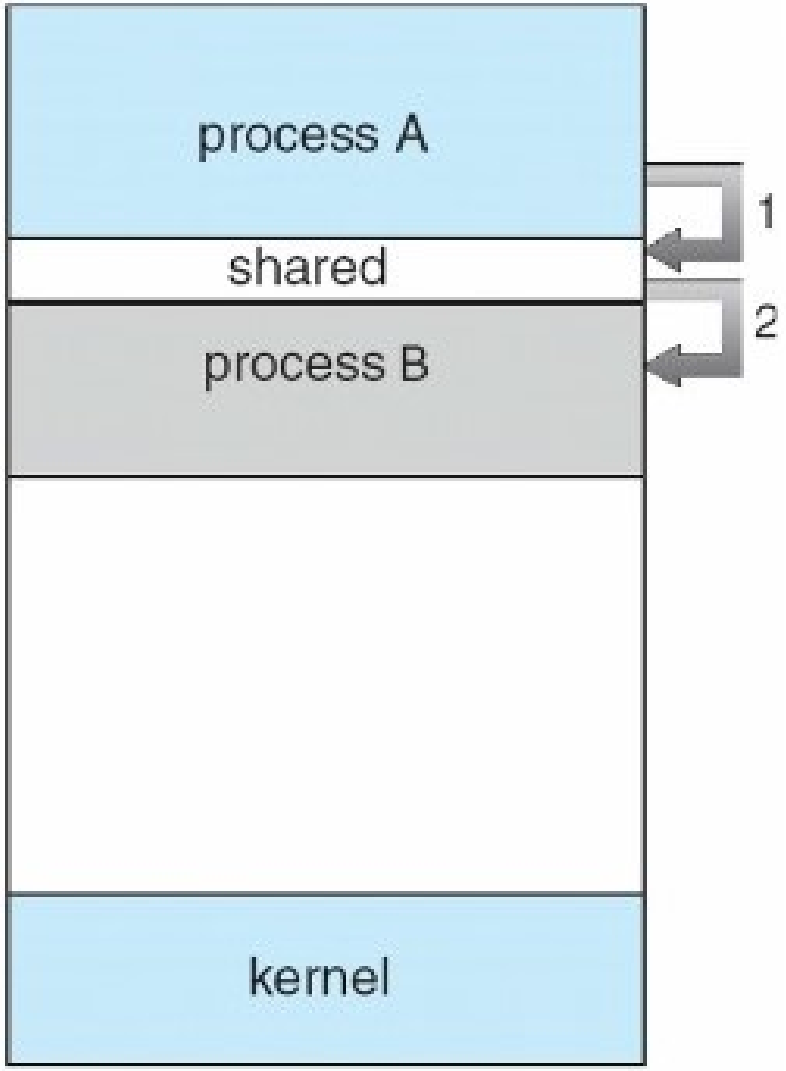
Two fundamental ways of IPC

Message Passing **Shared Memory**



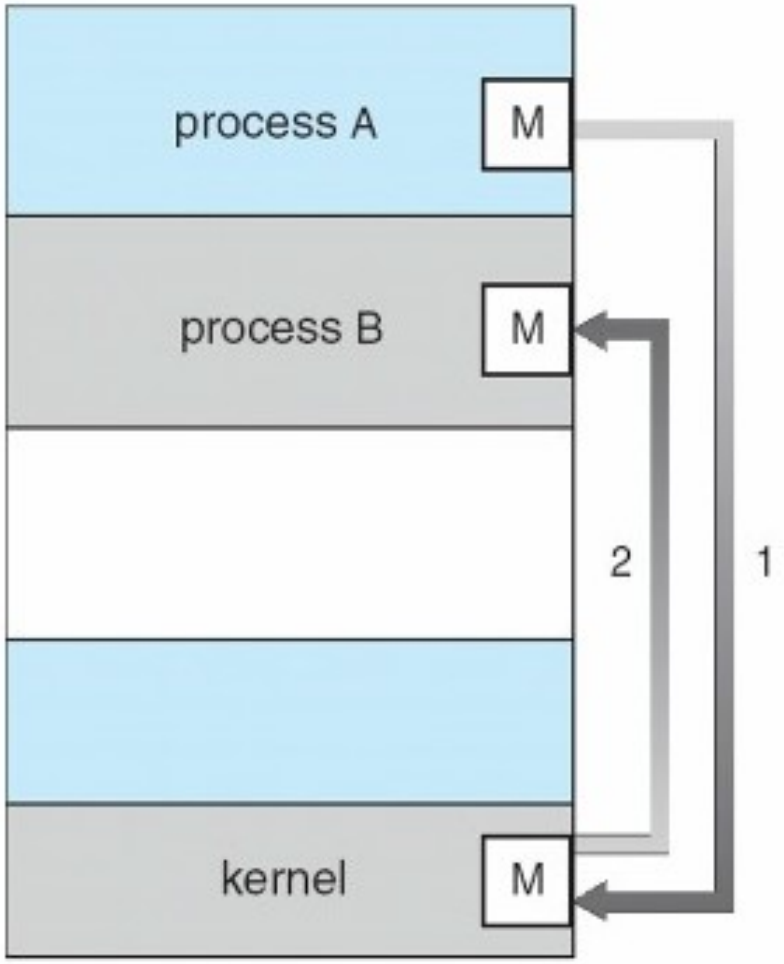
Shared Memory

- Region of memory that is shared by co-operating process is established.
- Process can exchange the information by reading and writing data to the shared region.
- Shared memory allows maximum speed and convenience of communication as it can be done at the speed of memory within the computer
- System calls are required only to establish shared memory regions.
- Once shared memory is established no assistance from the kernel is required, all access are treated as routine memory access.



MESSAGE PASSING

- Communication takes place by means of messages exchanged between the co-operating process
- Message passing is useful for exchanging the smaller amount of data since no conflict need to be avoided.
- Easier to implement than shared memory.
- Slower--implemented using System call which requires more time consuming task of Kernel intervention.



RACE CONDITION

A **race condition** is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly

Race Condition → More than one process reading/writing @ same time

RACE CONDITION

A race condition occurs when a device or system makes an attempt to perform two or more operations at the same time, **but not in the proper sequence**

Concurrent access may leads to data inconsistencies

Result may be : a computer crash, an ***"illegal operation,"*** notification and shutdown of the program, errors reading the old data, or errors writing the new data.

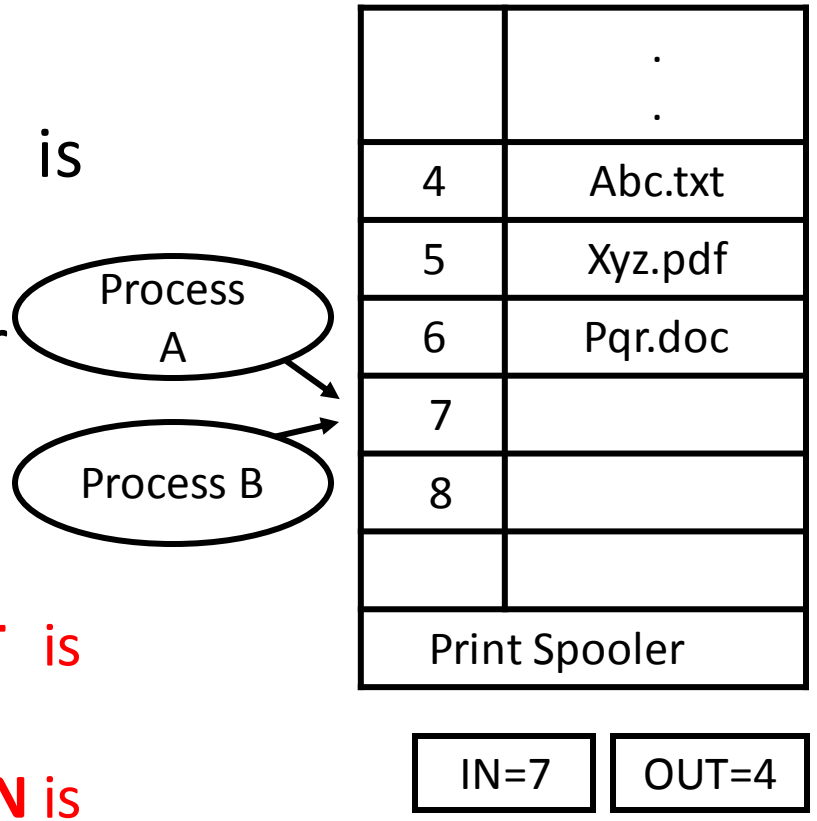
Printer Example for RACE Condition

Scenario

Print Spooler : Queue of file to be printed is stored

IN : Next Free slot available in the print spooler

OUT : Next file to be printed



Note:

- If the file is take for printing value of **OUT** is incremented
- If a new file is inserted to print spooler value of **IN** is incremented

Printer Example for RACE Condition

- Consider 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)
- **IN = 7** → Next free slot is 7 means next file given for printing will be inserted at position 7 in spooler
- **OUT = 4** → Next file to be printed from the spooler

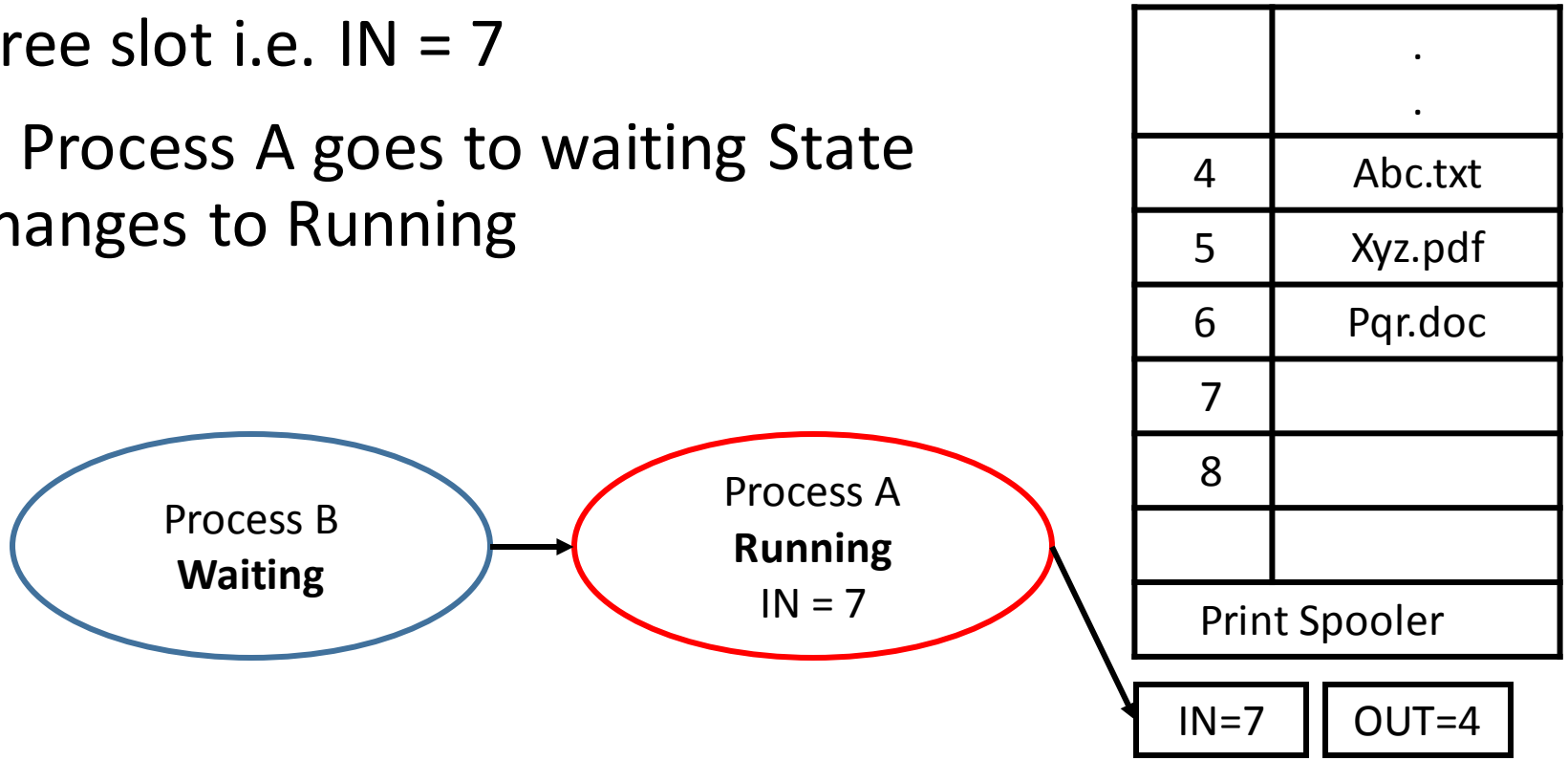
2 Process (A & B) tries to print different files

Printer Example for RACE Condition

Process A access print spooler

Reads the next free slot i.e. IN = 7

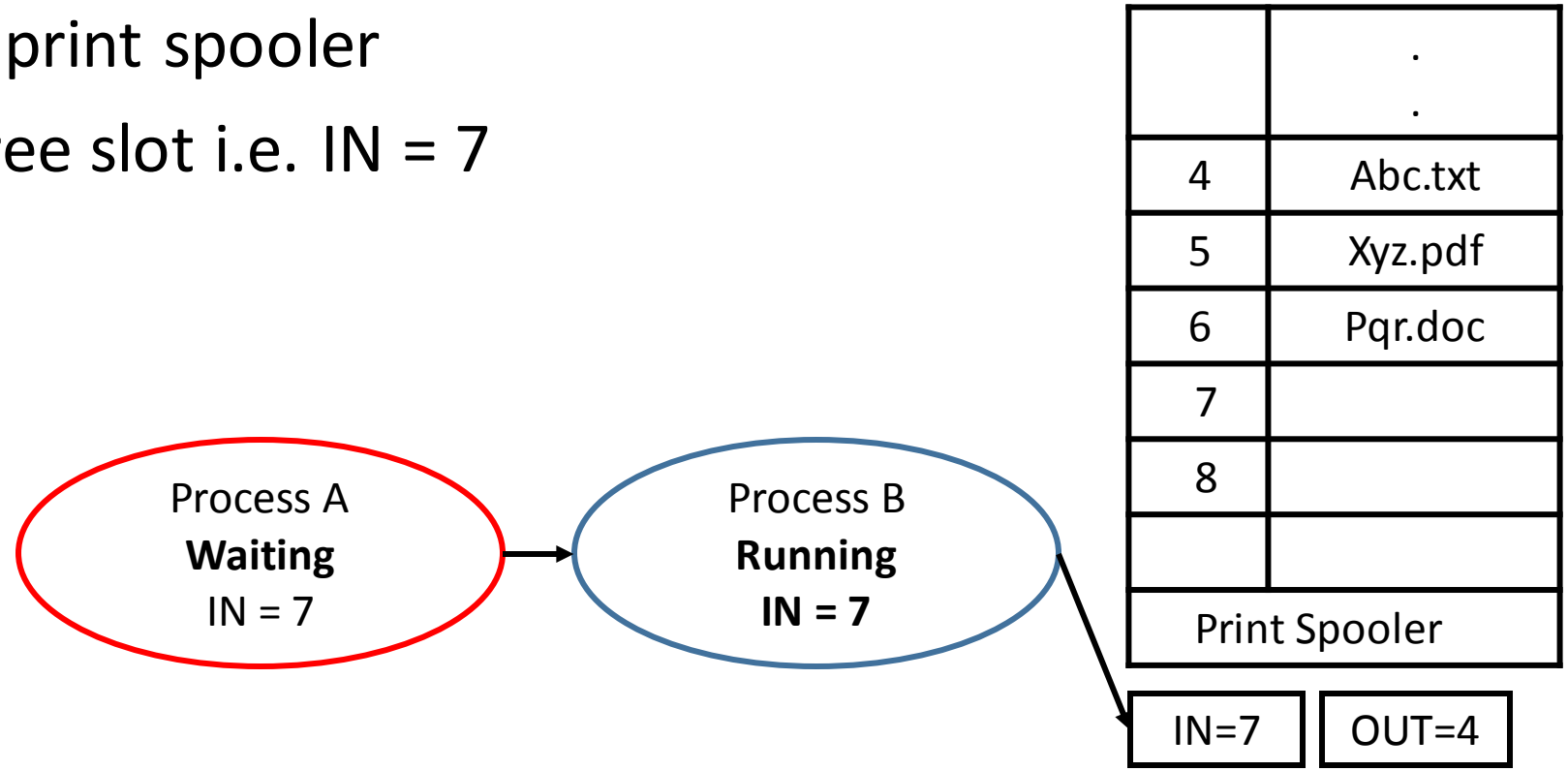
Due to interrupt Process A goes to waiting State and Process B Changes to Running



Printer Example for RACE Condition

Process B access print spooler

Reads the next free slot i.e. IN = 7

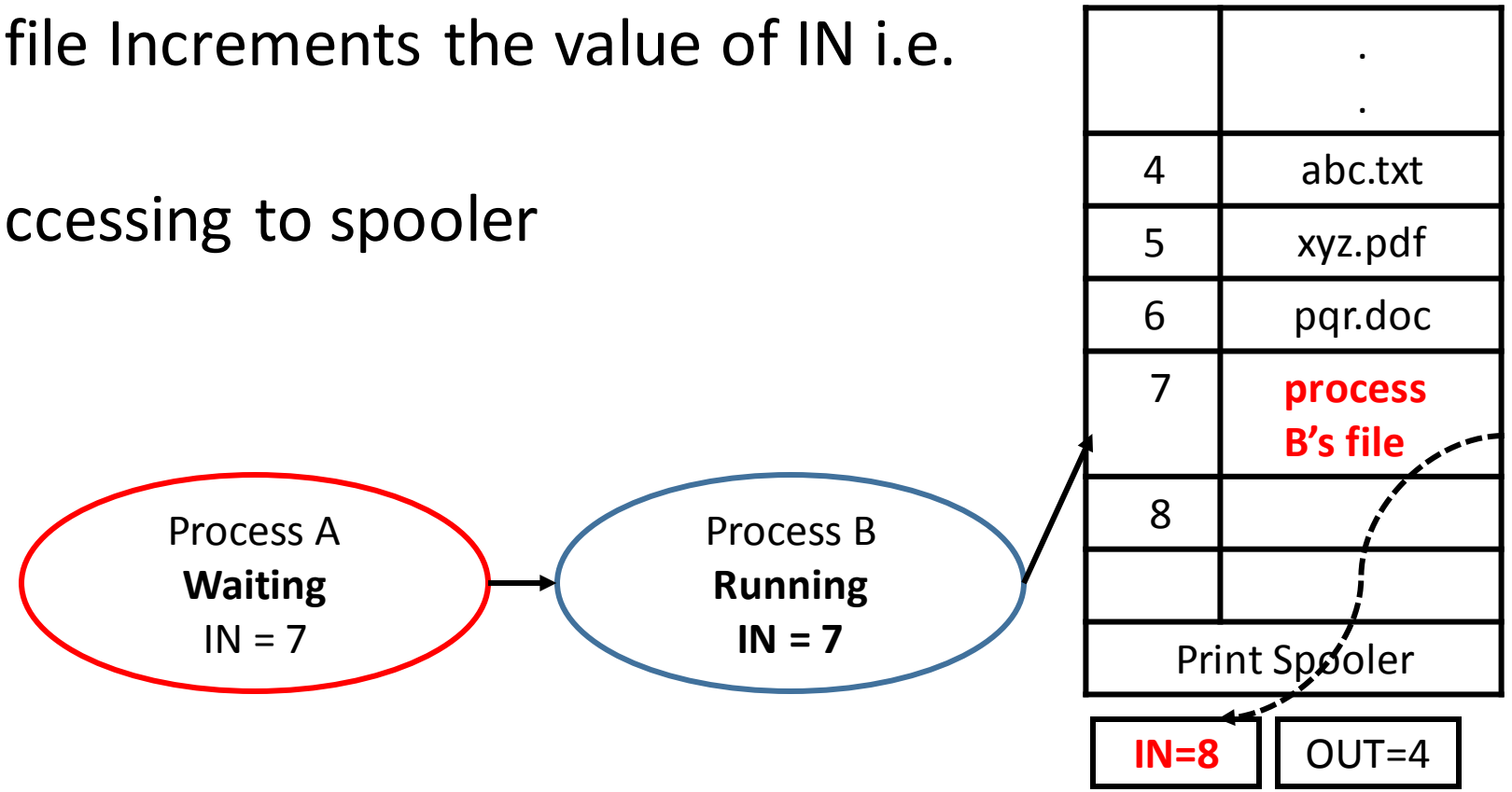


Printer Example for RACE Condition

Process B Writes the file in the free slot i.e. **IN = 7**

After writing the file Increments the value of IN i.e. now IN = 8

Process B stops accessing to spooler



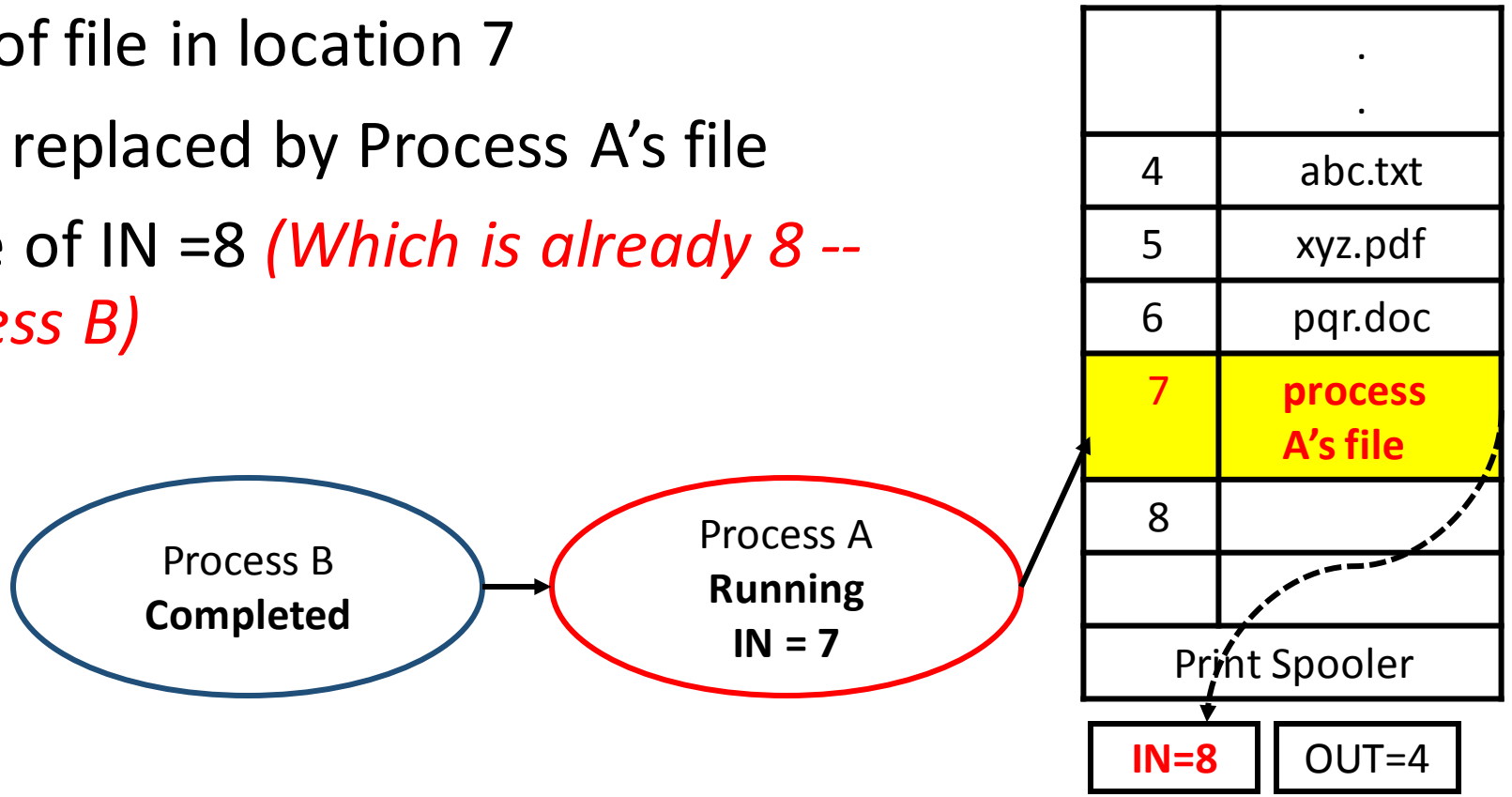
Printer Example for RACE Condition

Process A already have IN value 7

Writes the value of file in location 7

Process B's File is replaced by Process A's file

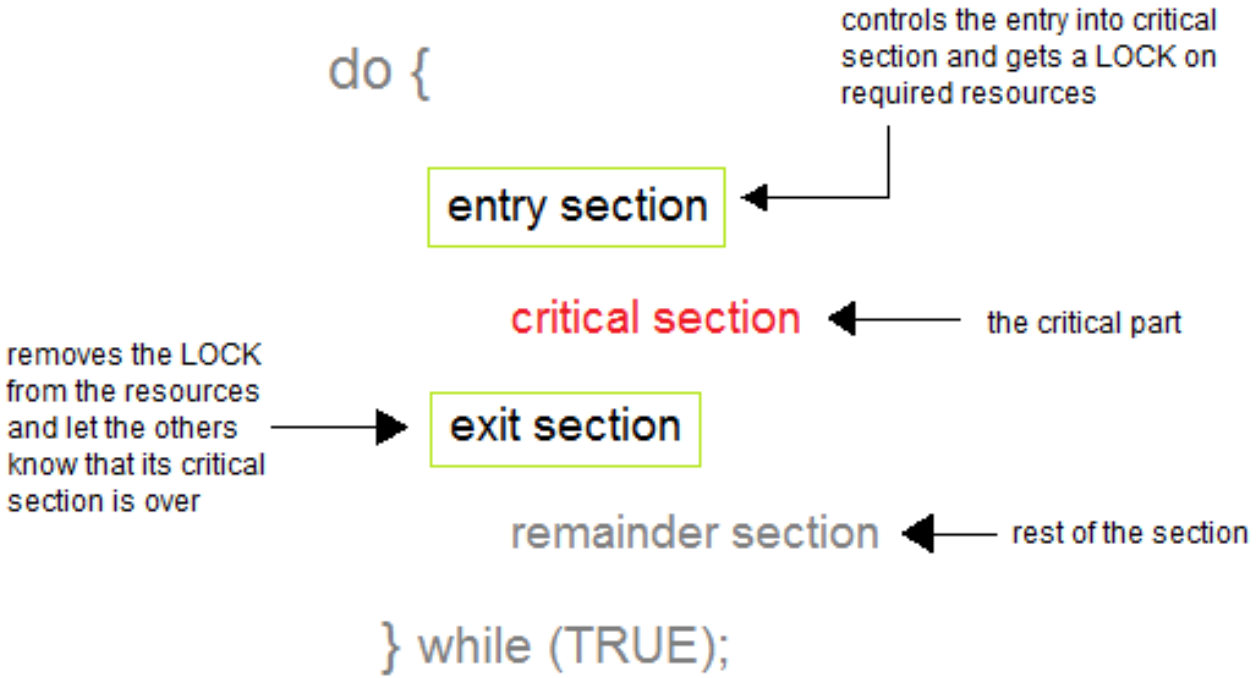
And update value of IN =8 *(Which is already 8 -- Updated by process B)*



Critical Section /Critical Section

Part/s of the program where the shared **resource** is accessed

Resources maybe hardware/software/Memory/Files



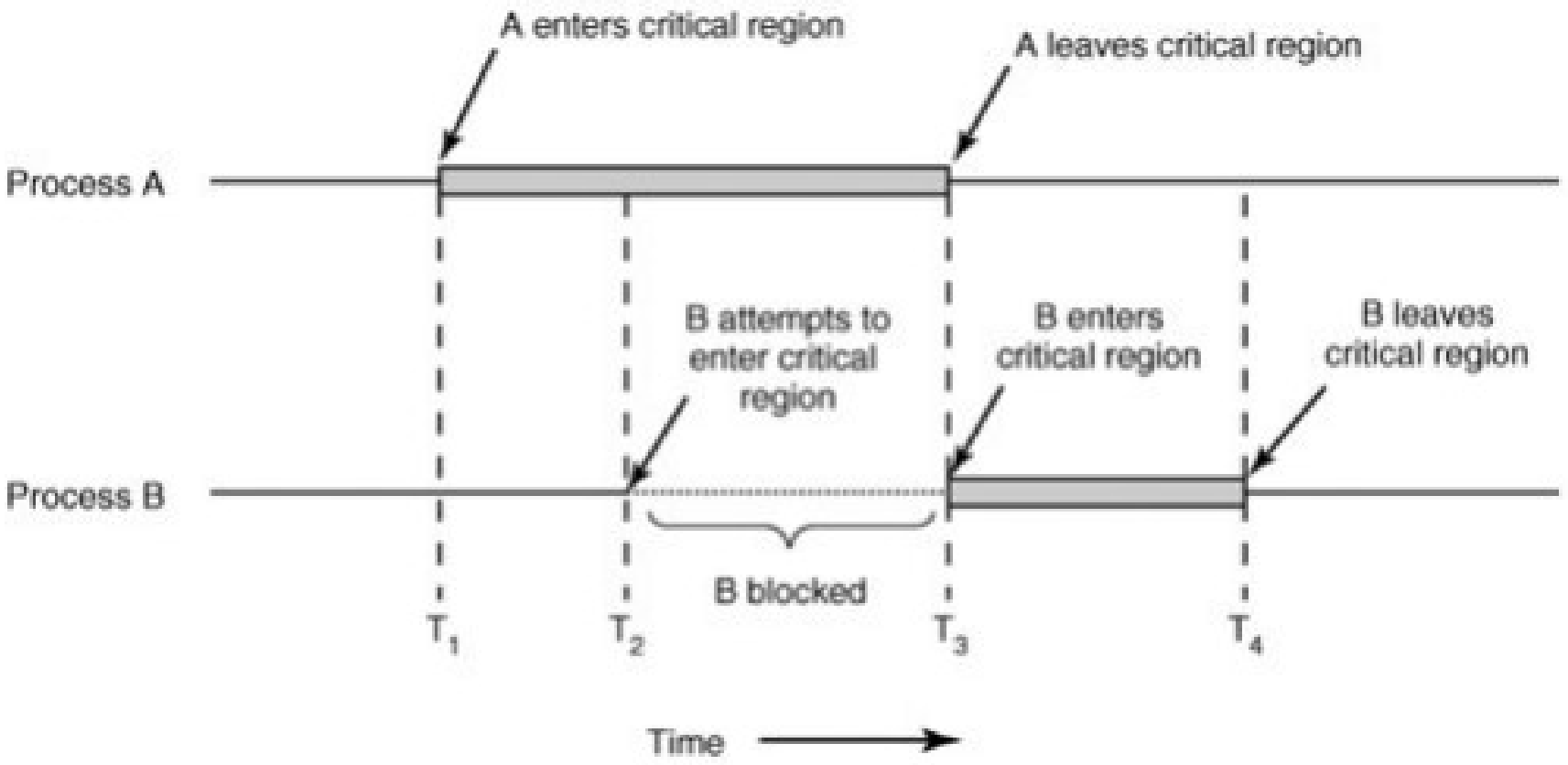
Solution to Critical section problem(Avoiding Race condition):

1. No two processes may be simultaneously inside their critical regions
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

Mutual Exclusion

Mutual exclusion is some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing

Mutual Exclusion



Mutual Exclusion

In above figure, process A enters its critical region at time T1. A little later, at time T2 process B attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, B is temporarily suspended until time T3 when A leaves its critical region, allowing B to enter immediately. Eventually B leaves (at T4) and we are back to the original situation with no processes in their critical regions.

Busy Waiting

Continuously testing/Checking a variable until some value appears is called busy waiting

Techniques for avoiding Race Condition (Critical section)problems

1. *Disabling Interrupts*
2. *Lock Variables*
3. *Strict Alternation*
4. *Peterson's Solution*
5. *TSL Instruction*

Disabling Interrupts

- Disable all interrupts just after entering its critical region and re-enable them just before leaving it.
- With interrupts disabled, **no clock interrupts** can occur
- *No Process will stops execution in middle of critical region access*
- CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process.
- Process can examine and update the shared memory without fear that any other process will intervene.

Disabling Interrupts

Disadvantages:

- Unwise to give user processes the power to turn off interrupts
- If system is a multiprocessor, disabling interrupts affects only the CPU that executed the disable instruction.

```
{
……
disable_interrupt();
enter C_Region();
……
……
enable_interrupt();
leave C_Region();
}
```

Lock variables

- Single, shared (lock) variable
- Initially value is 0
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0.
- 0 means that no process is in its critical region
- 1 means that some process is in its critical region

Lock variables

Drawbacks:

- Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory
- Lock Variable is also shared, So Probability of race condition

```
int lock =0;
```

```
fun()
{
……
lock=1;
enter C_Region();
……
……
lock=0;
leave C_Region();
}
```

Strict Alteration

- Integer variable **turn** is initially 0.
- **Turn** keeps track of whose turn it is to enter the critical region and examine or update the shared memory.
- Only when there is a reasonable expectation that the wait will be short is busy waiting used.
- A lock that uses busy waiting is called **a spin lock**.

Strict Alteration

- When process 0 leaves the critical region, it sets turn to 1, to allow process 1 to enter its critical region.
- This way no two process can enters critical region simultaneously.

Drawbacks:

- Taking turn is not a good idea when one of the process is much slower than other. This situation requires that two processes strictly alternate in entering their critical region

Strict Alteration

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

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

TSL(Test Set Lock)

- It reads the contents of the memory word LOCK into register RX and then stores a nonzero value at the memory address LOCK.
- The operations of reading the word and storing into it are guaranteed to be indivisible no other processor can access the memory word until the instruction is finished
- The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

Algorithm for entering c_region

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
```

Algorithm for leaving c_region

leave_region:

```
MOVE LOCK, #0 //store a 0 in LOCK
RET //return to caller
```

Peterson's Solution

Enter Critical Region

Before using the shared variables each process calls ***enter_region()*** with its own process number as parameter.

Call will cause it to wait, if need be, until it is safe to enter

Exit Critical Region

After it has finished with the shared variables, the process calls ***leave_region()*** to indicate that it is done and to allow the other process to enter, if it so desires

Peterson's Solution (Variables)

```
#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)*/
```

Peterson's Solution (Enter Region)

```
void enter_region(int process) /* process is 0 or 1 */
{
int other; // Indicates number of the other process
other = 1 - process; // other process
interested[process] = TRUE; // show that you are interested
turn = process;   // set flag
while (turn == other && interested[other] == TRUE); //Wait
}
```

Peterson's Solution (Leaving Region)

```
void leave_region(int process)
/* process: who is leaving */
{
interested[process] = FALSE;
/* indicate departure from critical region */
}
```

Peterson's Solution (Explation)

- A process that is about to enter its critical region has to call enter_region. At the end of its critical region it calls leave_region.

- Initially, both processes are not in their critical region and the array *interested* has all (both in the above example) its elements set to false.

Peterson's Solution (Explation)

- Assume that process 0 calls enter_region. The variable *other* is set to one (the other process number) and it indicates its interest by setting the relevant element of *interested*. Next it sets the *turn* variable, before coming across the while loop. In this instance, the process will be allowed to enter its critical region, as process 1 is not interested in running.
- Now process 1 could call enter_region. It will be forced to wait as the other process (0) is still interested. Process 1 will only be allowed to continue when *interested*[0] is set to false which can only come about from process 0 calling leave_region.

Peterson's Solution (Explation)

- If we ever arrive at the situation where both processes call enter region at the same time, one of the processes will set the turn variable, but it will be immediately overwritten.

- Assume that process 0 sets turn to zero and then process 1 immediately sets it to 1. Under these conditions process 0 will be allowed to enter its critical region and process 1 will be forced to wait.

Decker's Solution

- Achieved with the use of **"flags"** and **"token"**
- The flags indicate whether a **process wants to enter the critical section (CS) or not**
- Value of 1 means TRUE that the process wants to enter the CS, while 0, or FALSE, means the opposite.
- The token, which can also have a value of 1 or 0, **indicates priority** when both processes have their flags set to TRUE.

Decker's algorithm

Disadvantages

- Constantly test whether the critical section is available and therefore wastes significant processor time.
- It creates the problem known as lockstep synchronization, in which each thread may only execute in strict synchronization.
- It is also non-expandable as it only supports a maximum of two processes for mutual exclusion.

Problems with mutual Exclusion With Busy Waiting

- The above techniques achieves the mutual exclusion using busy waiting. Here while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.
- Mutual Exclusion with busy waiting just check to see if the entry is allowed when a process wants to enter its critical region, if the entry is not allowed the process just sits in a tight loop waiting until it is

1. Waste CPU time
2. priority inversion problem

- **Priority Inversion Problem:**
- Consider a computer with two processes, H, with high priority and L, with low priority, which share a critical region.
- At a certain moment, with L in its critical region, H becomes ready to run (e.g., an I/O operation completes).
- H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever. This situation is sometimes referred to as the priority inversion problem.
- Let us now look at some IPC primitives that blocks instead of wasting CPU time when they are not allowed to enter their critical regions. Using blocking constructs greatly improves the CPU utilization

Sleep and Wakeup

- Peterson's and TSL solutions have the defect of requiring **busy waiting**
- when a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a **tight loop waiting until it is allowed**
- **Sleep is a system call** that causes the caller to block, that is, be suspended until another process wakes it up.
- The **wakeup call** has one parameter, the process to be awakened.

Sleep and Wakeup

- Peterson's and TSL solutions have the defect of requiring **busy waiting**
- when a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a **tight loop waiting until it is allowed**
- **Sleep is a system call** that causes the caller to block, that is, be suspended until another process wakes it up.
- The **wakeup call** has one parameter, the process to be awakened.

Producer-Consumer problem

- Also known as the **bounded-buffer** problem

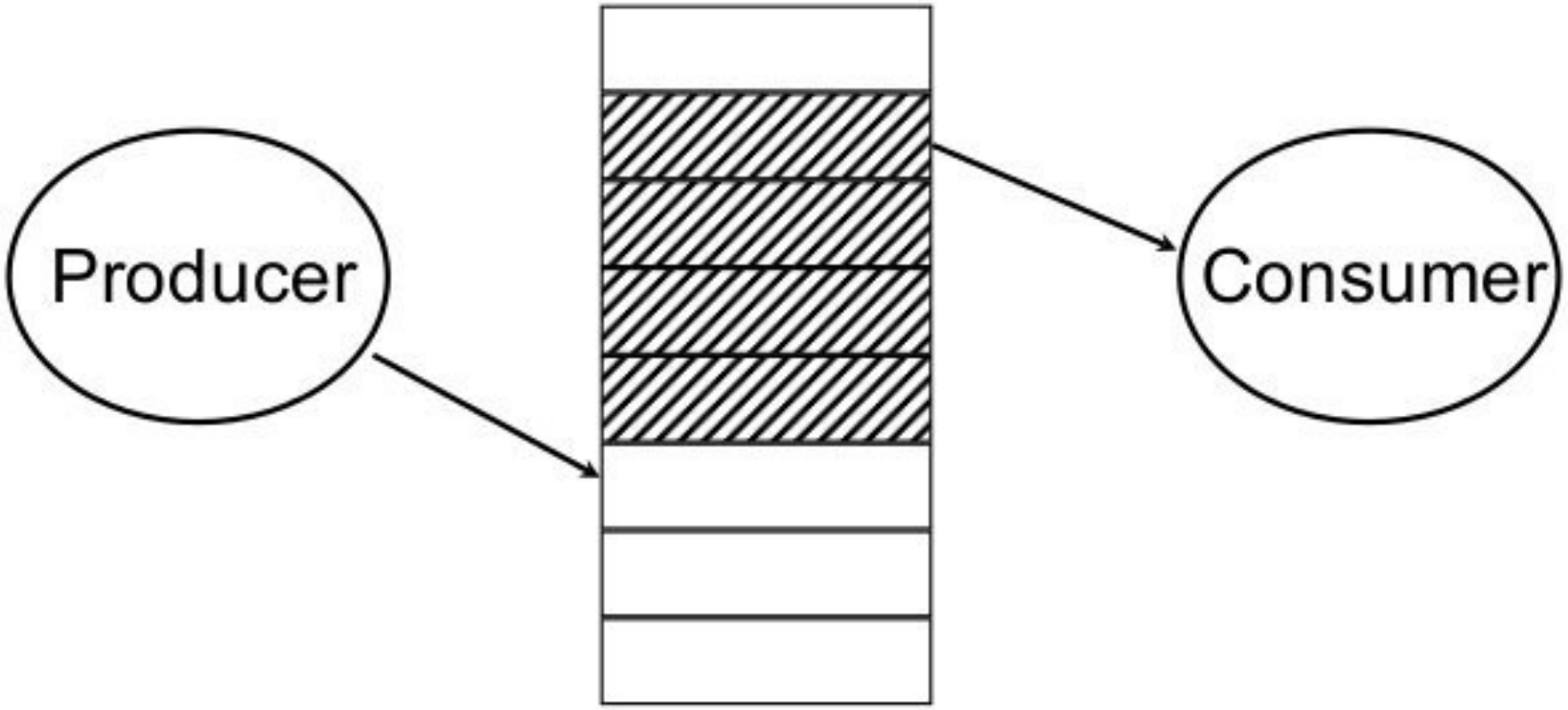
General Situation:

- Two processes share a common, fixed-size buffer
- The producer -puts information into the buffer
- The consumer-takes it out

Problem:

- The producer wants to put a new data in the buffer i.e. already full.
- The consumer wants to remove data the buffer i.e. already empty.

Producer-Consumer problem



Producer-Consumer problem

Solution

- Producer goes to sleep and to be awakened when the consumer has removed data
- Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up

Producer-Consumer problem

```
#define N 100
int count = 0;
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);
}
}
```

Producer-Consumer problem

```
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

Producers code:

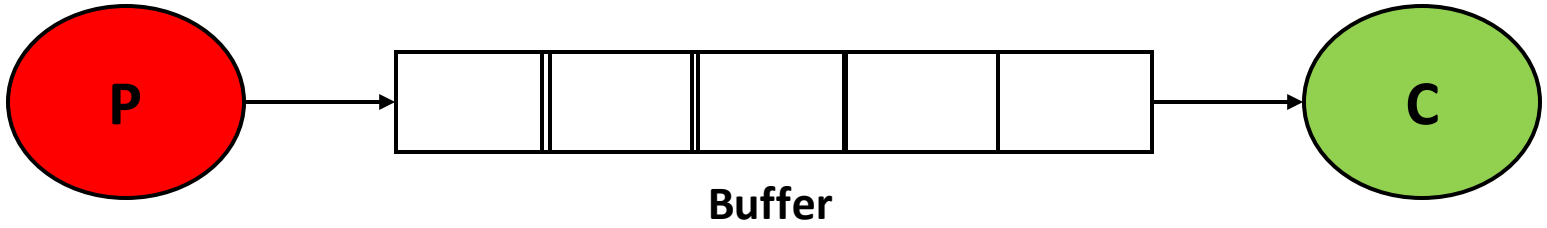
The producers code is first test to see if count is N. If it is, the producer will go to sleep ; if it is not the producer will add an item and increment count.

Consumer code:

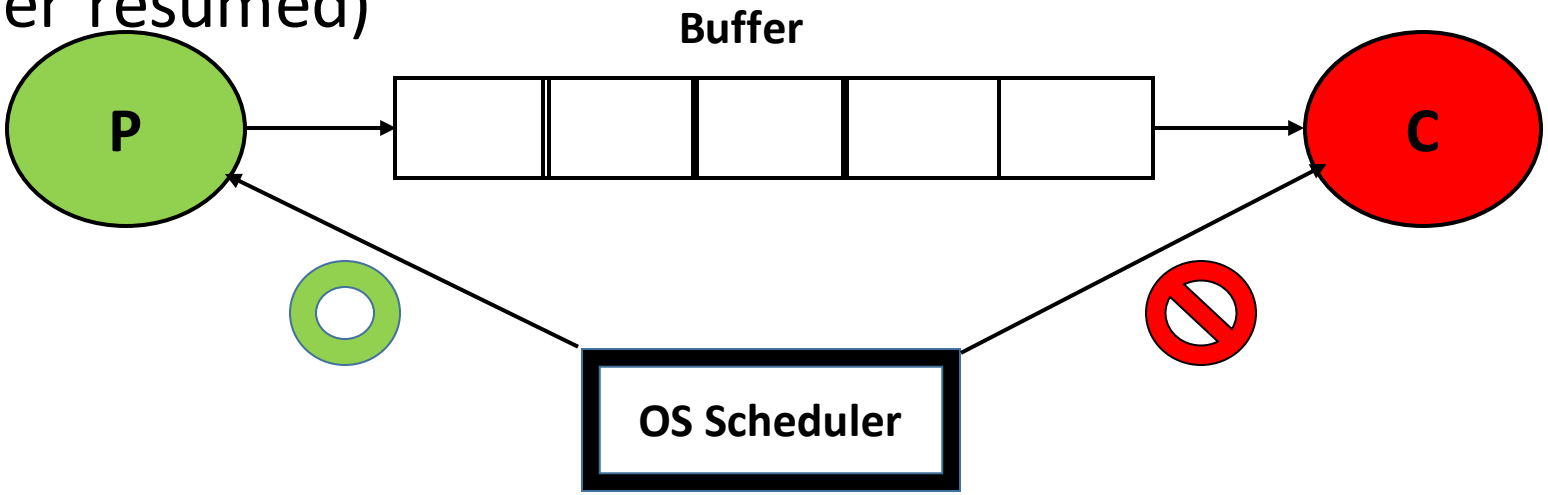
It is similar as of producer. First test count to see if it is 0. If it is, go to sleep; if it nonzero remove an item and decrement the counter.

Producer-Consumer problem

1. The buffer is empty and the consumer has just read count to see if it is 0.

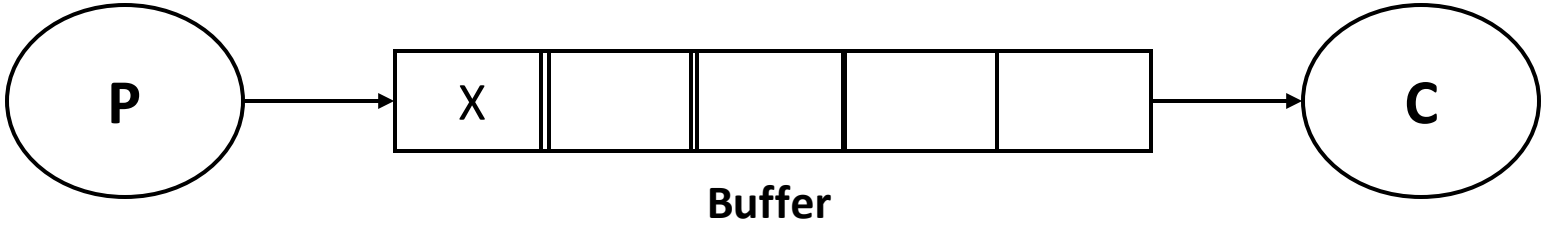


2. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. (Consumer is interrupted and producer resumed)

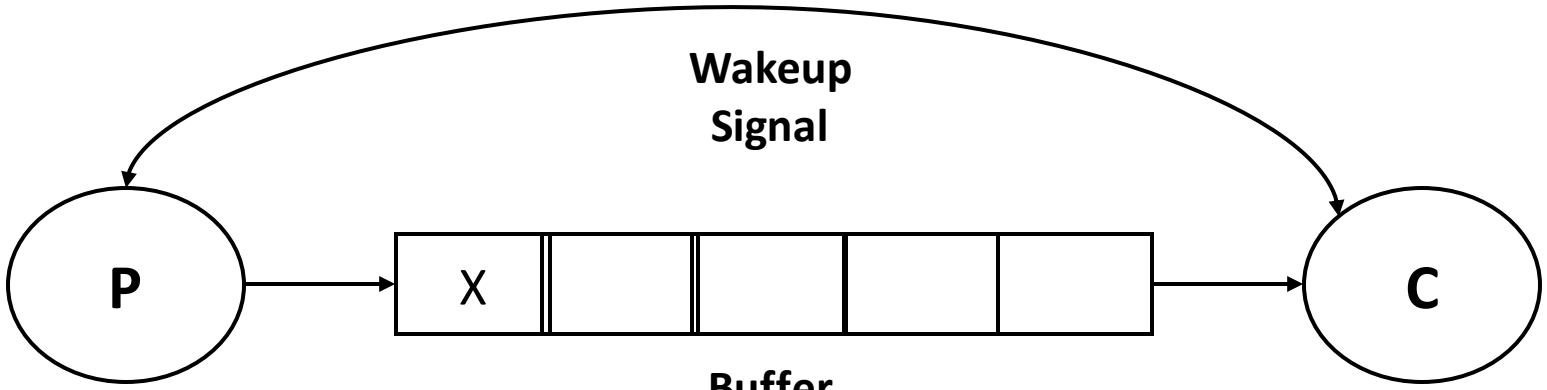


Producer-Consumer problem

3. The producer creates an item, puts it into the buffer, and increases count.

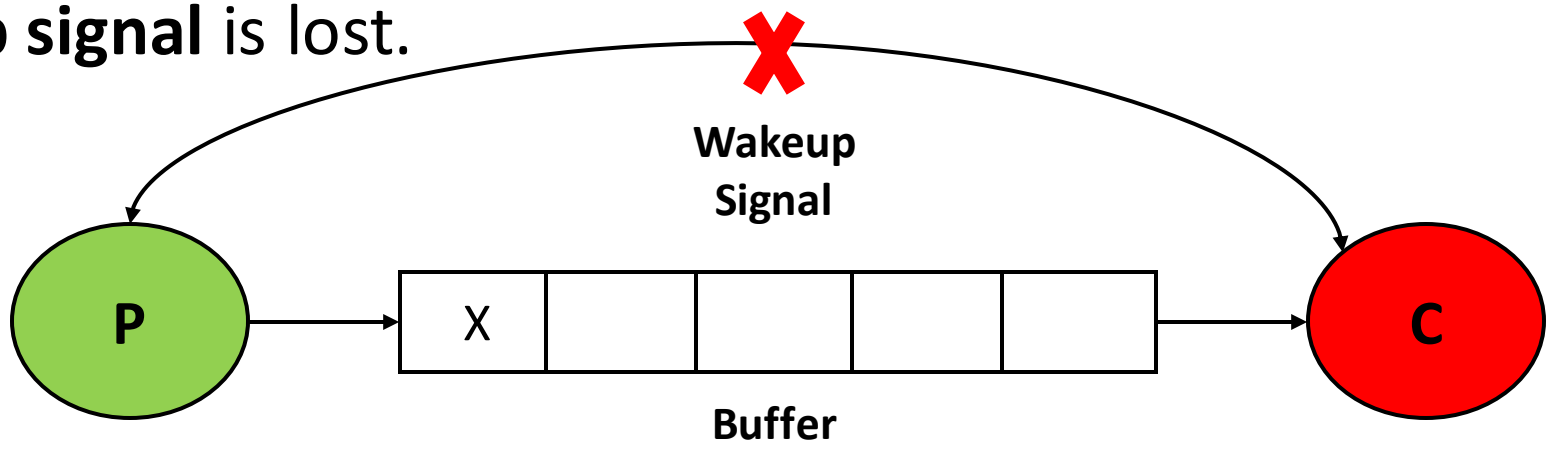


4. Because the buffer was empty prior to the last addition (count was just 0), the producer tries to wake up the consumer.

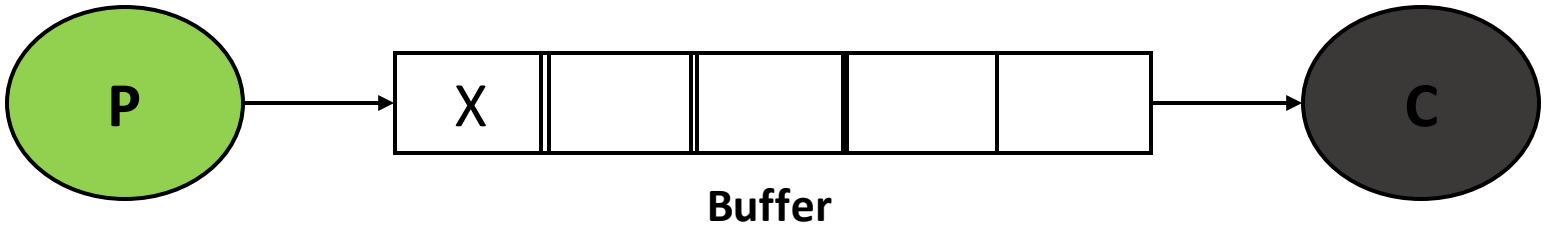


Producer-Consumer problem

5. Unfortunately, the consumer is not yet logically asleep, so the **wakeup signal** is lost.

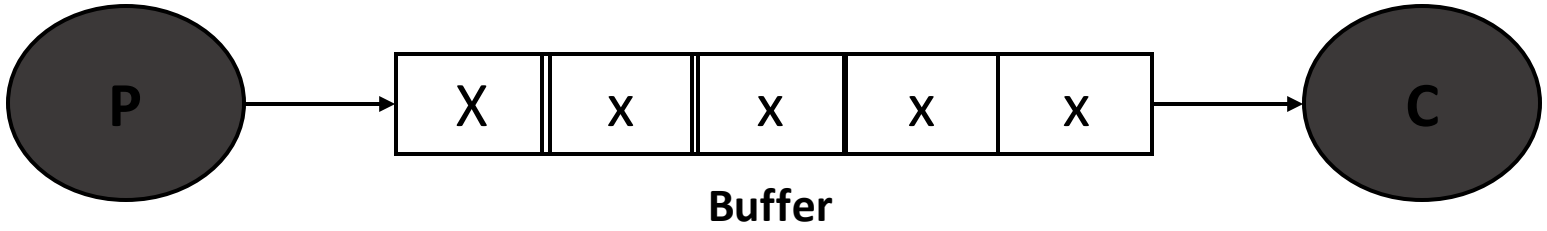


6. When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep



Producer-Consumer problem

8. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.



Semaphores

- A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending
- Synchronization tool that does not require busy waiting
- A semaphore is a special kind of integer variable which can be initialized and can be accessed only through two **atomic operations**
- **2 Operations**
  - **P – down operation**
  - **V – up operation**

Semaphores (P Operation)

P (S):

S=S-1;

IF (S<0)

wait for S

Semaphores (V Operation)

P (S):

S=S-1;

IF (Process / Thread is WAITING FOR S)

Wake up (Process / Thread) ;

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1; can be simpler to implement Also known as mutex locks

```
#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){                              /* TRUE is the constant 1 */
          item = produce_item();                /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}

void consumer(void)
{
     int item;

     while (TRUE){                              /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item();                 /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

- **This solution uses three semaphores:**
- **one called *full* for counting the number of slots that are full,**
- **one called *empty* for counting the number of slots that are empty**
- **one called *mutex* to make sure the producer and consumer do not access the buffer at the same time.**

Mutex Lock

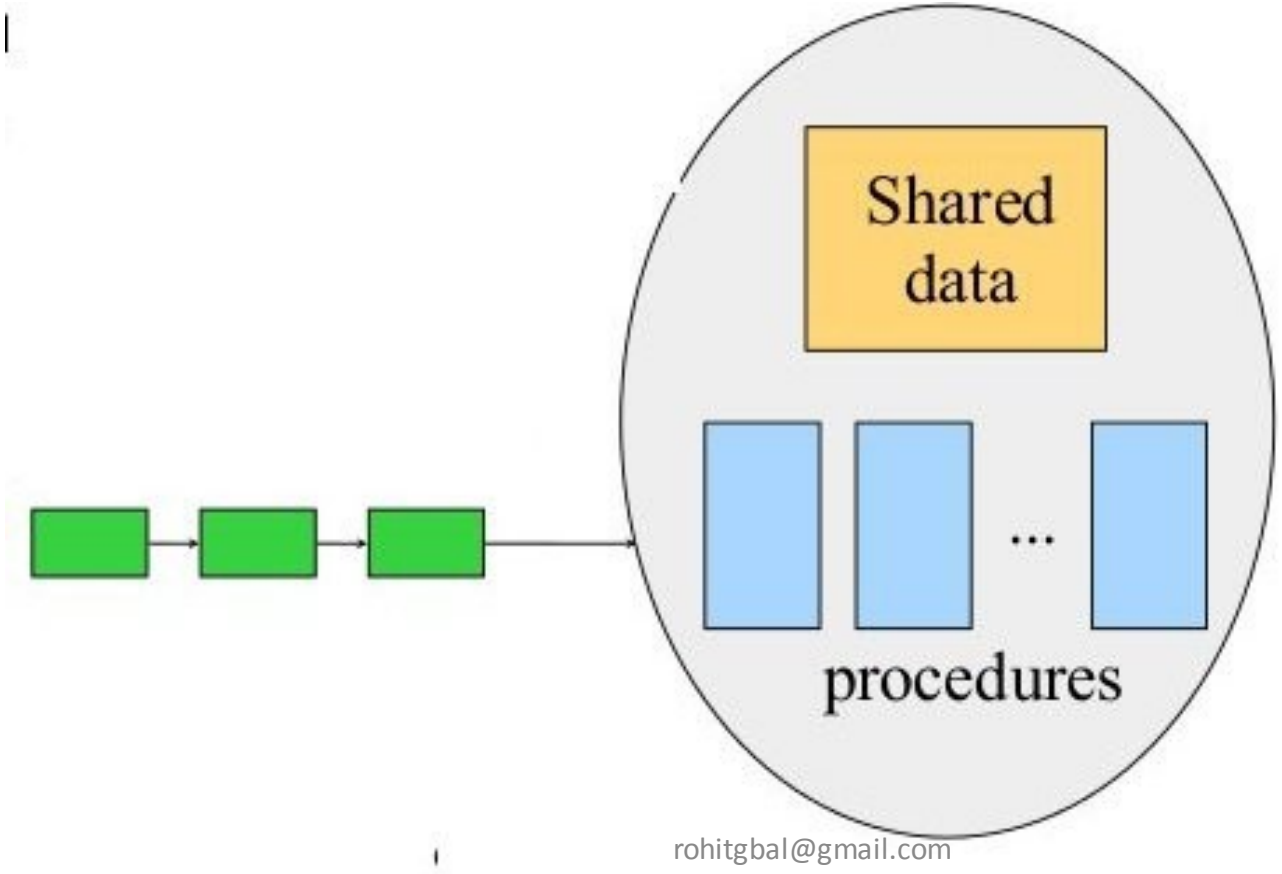
- A mutex is a variable that can be in one of two states: unlocked or locked
- Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked
- Two procedures are used with mutexes. When a process (or thread) needs access to a critical region, it calls mutex_lock
- If the mutex is currently unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region

Monitors

- A higher level synchronization primitive.
- A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of **module or package**.
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.
- A **monitor** consists of a mutex(lock)object and **condition variables**.
- A **condition variable** is basically a container of threads that are waiting on a certain condition

Monitors

- A **monitor** is an object or module intended to be used safely by more than one thread
- characteristic - Its methods are executed with mutual exclusion



Message Passing

- Form of communication in interprocess communication
- Processes can send and receive messages to other processes.
- Forms of messages - **method invocation, signals, and data packets**
- Two primitives- **send and receive**
- send(destination, &message)
- receive(source, &message)
- **Synchronous message** passing systems requires the sender and receiver to wait for each other to transfer the message
- **Asynchronous message** passing systems deliver a message from sender to receiver, without waiting for the receiver to be ready

Message Passing

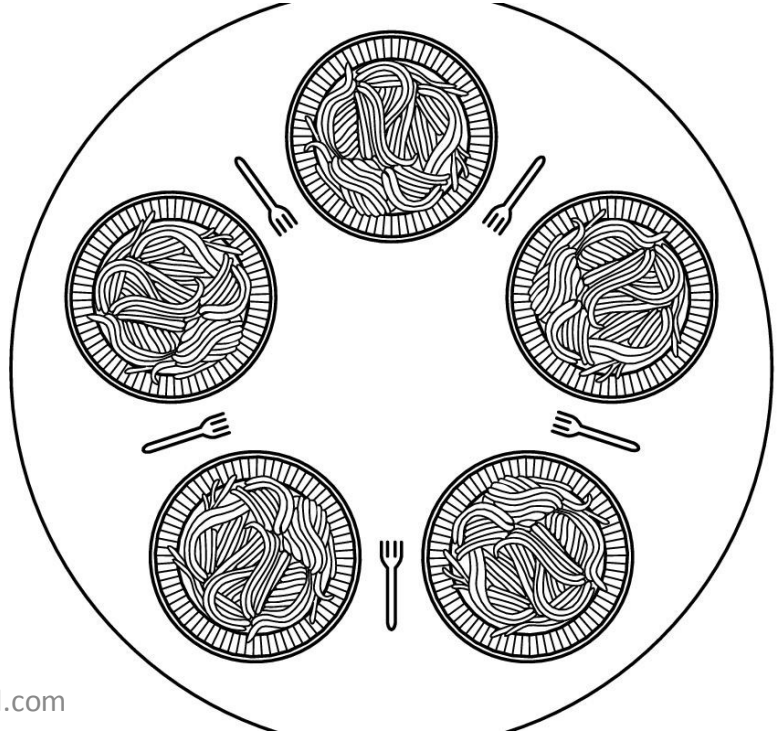
- Designing a message passing
  - Whether messages are transferred reliably
  - Whether messages are guaranteed to be delivered in order
  - Whether messages are passed one-to-one, one-to-many (multicasting or broadcasting), or many to-one (client–server)
  - Whether communication is synchronous or asynchronous.

Classical IPC Problems

1. Dining Philosophers Problem
2. The Readers and Writers Problem
3. The Sleeping Barber Problem

Dining Philosophers Problem

- N philosphers sitting around a circular table eating spaghetti and discussing philosophy
- **problem** - each philosopher needs 2 forks to eat, and there are only N forks, one between each 2 philosophers
- Design an algorithm – no one starves, max no philosopher eat at one
  - Philosophers eat/think
  - Eating needs 2 forks
  - Pick one fork at a time
  - How to prevent deadlock



Dining Philosophers Problem(Possible Solutions)

- One idea is to instruct each philosopher to behave as follows:
  - think until the left fork is available; when it is, pick it up
  - think until the right fork is available; when it is, pick it up
  - eat
  - put the left fork down
  - put the right fork down
  - repeat from the start
- **Incorrect Solution -** it allows the system to reach deadlock
- All five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

Dining Philosophers Problem (Possible Solution)

- After taking the left fork, the program checks to see if the rightfork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process
- **Solution Fails:** Suppose all the philosophers could start the algorithm simultaneously

Dining Philosophers Problem(Solution)

- Deadlock-free and allows the maximum parallelism
- Uses an array state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks)
- eating state only if neither neighbor is eating

Dining Philosophers Problem

Solution:

```
#define N 5 /* number of philosophers */
#define LEFT (i+N-1)%N /* number of i's left neighbor */
#define RIGHT (i+1)%N /* number of i's right neighbor */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is trying to get forks */
#define EATING 2 /* philosopher is eating */
typedef int semaphore; /* semaphores are a special kind of int */
int state[N]; /* array to keep track of everyone's state */
semaphore mutex = 1; /* mutual exclusion for critical regions */
semaphore s[N]; /* one semaphore per philosopher */
```

Dining Philosophers Problem

```
void philosopher(int i) /* i: philosopher number, from 0 to N1 */
{
while (TRUE){ /* repeat forever */
think(); /* philosopher is thinking */
take_forks(i); /* acquire two forks or block */
eat(); /* yum-yum, spaghetti */
put_forks(i); /* put both forks back on table */
}
}
```

Dining Philosophers Problem

```
void take_forks(int i) /* i: philosopher number, from 0 to N1 */
{
down(&mutex); /* enter critical region */
state[i] = HUNGRY; /* record fact that philosopher i is hungry */
test(i); /* try to acquire 2 forks */
up(&mutex); /* exit critical region */
down(&s[i]); /* block if forks were not acquired */
}
```

Dining Philosophers Problem

```
void put_forks(i) /* i: philosopher number, from 0 to N1 */
{
down(&mutex); /* enter critical region */
state[i] = THINKING; /* philosopher has finished eating */
test(LEFT); /* see if left neighbor can now eat */
test(RIGHT); /* see if right neighbor can now eat */
up(&mutex); /* exit critical region */
}
```

Dining Philosophers Problem

```
void test(i) /* i: philosopher number, from 0 to N1* /
{
if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
{
state[i] = EATING;
up(&s[i]);
}
}
```

Sleeping Barber Problem

Sleeping barber problem

- Customers arrive to a barber
- If there are no customers the barber sleeps in his chair.
- If the barber is asleep then the customers must wake him up
- The barber has one barber chair and a waiting room with a number of chairs in it.
- When the barber finishes cutting a customer's hair, he dismisses the customer and then goes to the waiting room to see if there are other customers waiting
- If there are, he brings one of them back to the chair and cuts his hair.
- If there are no other customers waiting, he returns to his chair and sleeps in it

Sleeping Barber Problem

- Each customer, when he arrives, looks to see what the barber is doing
- If the barber is sleeping, then thecustomer wakes him up and sits in the chair
- If the barber is cutting hair, then the customer goes to the waiting room.
- If there is a free chair in the waiting room, the customer sits in it and waits his turn
- If there is no free chair, then the customer leaves.

Sleeping Barber Problem(Problem 1)

- a customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room.
- While he is on his way, the barber finishes the haircut he is doing and goes to check the waiting room.
- Since there is no one there (the customer not having arrived yet), he goes back to his chair and sleeps.
- The barber is now waiting for a customer and the customer is waiting for the barber

Sleeping Barber Problem(Problem 2)

- two customers may arrive at the same time when there happens to be a single seat in the waiting room.
- They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair

Sleeping Barber Problem(Solution)

- The key element of each is a mutex, which ensures that only one of the participants can change state at once.
- The barber must acquire this mutex exclusion before checking for customers and release it when he begins either to sleep or cut hair.
- A customer must acquire it before entering the shop and release it once he is sitting in either a waiting room chair or the barber chair.
- This eliminates both of the problems mentioned in the previous section
- A number of semaphores are also required to indicate the state of the system

Readers Writer Problem

- **Example**- an airline reservation system, with many competing processes wishing to read and write it
- It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other process may have access to the database

Solution to Readers Writer problems

```
typedef int semaphore; /* semaphores are a special kind of int */
semaphore mutex = 1; /* controls access to 'rc' */
semaphore db = 1; /* controls access to the database */
int rc = 0; /* # of processes reading or wanting to */
```

Readers Writer Problem

```
void reader(void)
{
while (TRUE){ /* repeat forever */
down(&mutex); /* get exclusive access to 'rc' */
rc = rc + 1; /* one reader more now */
if (rc == 1) down(&db); /* if this is the first reader ... */
up(&mutex); /* release exclusive access to 'rc' */
read_data_base(); /* access the data */
down(&mutex); /* get exclusive access to 'rc' */
rc = rc 1; /* one reader fewer now */
if (rc == 0) up(&db); /* if this is the last reader ... */
up(&mutex); /* release exclusive access to 'rc' */
use_data_read(); /* noncritical region */
}
}
```

Readers Writer Problem

```
void writer(void)
{
while (TRUE){ /* repeat forever */
think_up_data(); /* noncritical region */
down(&db); /* get exclusive access */
write_data_base(); /* update the data */
up(&db); /* release exclusive access */
}}
```

- First reader to get access to the data base does a down on the semaphore db. Subsequent readers merely have to increment a counter
- As readers leave, they decrement the counter and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in

Deadlock

- **Resources** are the passive entities needed by processes to do their work
- A resource can be a **hardware device or a piece of information**
- Two types of resources :
- **Preemptable** – A Preemptable resources is one that can be taken away from the process owing it with no ill effect (Ex: Read only files)
- **Non-preemptable** – A non-preemptable resources in contrast is one that cannot be taken away from its current owner without causing the computation to fail.(Ex CD- rom, printer)
- The sequence of events required to use a resource is:
    1. Request the resource
    2. Use the resource
    3. Release the resource

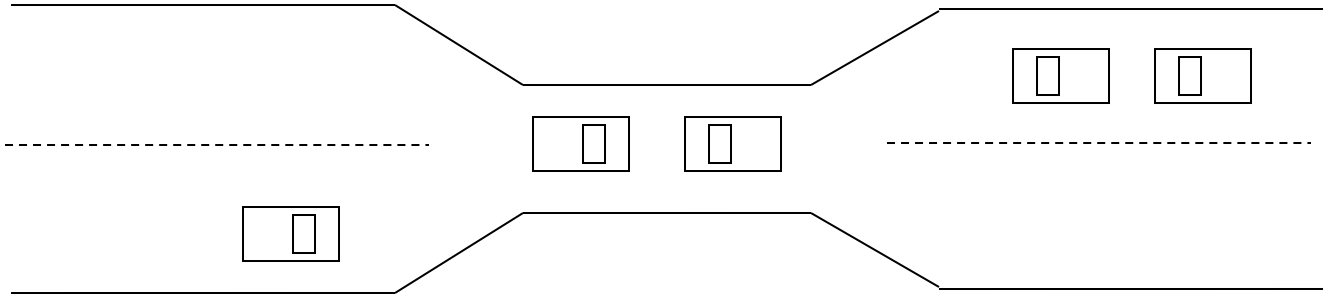
What is Deadlock?

- Set of process is said to be in deadlock if each process in the set is waiting for an event that only another process in the set can cause
- Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes **continue to wait forever**.
- Staying Safe: Preventing and Avoiding Deadlocks
- Living Dangerously: Let the deadlock happen, then detect it and recover from it.

Cause of deadlocks

- Each process needing what another process has. This results from sharing resources such as memory, devices, links.

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

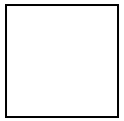
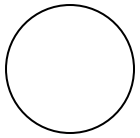
Necessary Conditions

ALL of these four **must** happen simultaneously for a deadlock to occur:

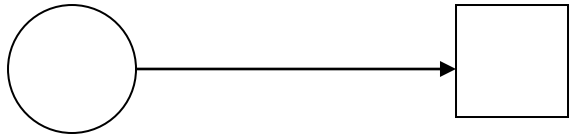
1. **Mutual exclusion:** One or more than one resource must be held by a process in a non-sharable (exclusive) mode.
2. **Hold and Wait:** A process holds a resource while waiting for another resource.
3. **No Preemption:** There is only voluntary release of a resource - nobody else can make a process give up a resource.
4. **Circular Wait:** Process A waits for Process B waits for Process C .... waits for Process A.

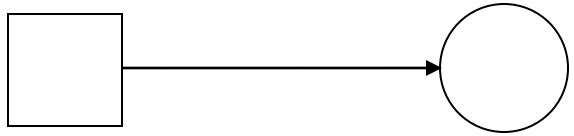
Resource Allocation Modelling using Graphs

Nodes :

resource  process 

Arcs :

resource requested : 

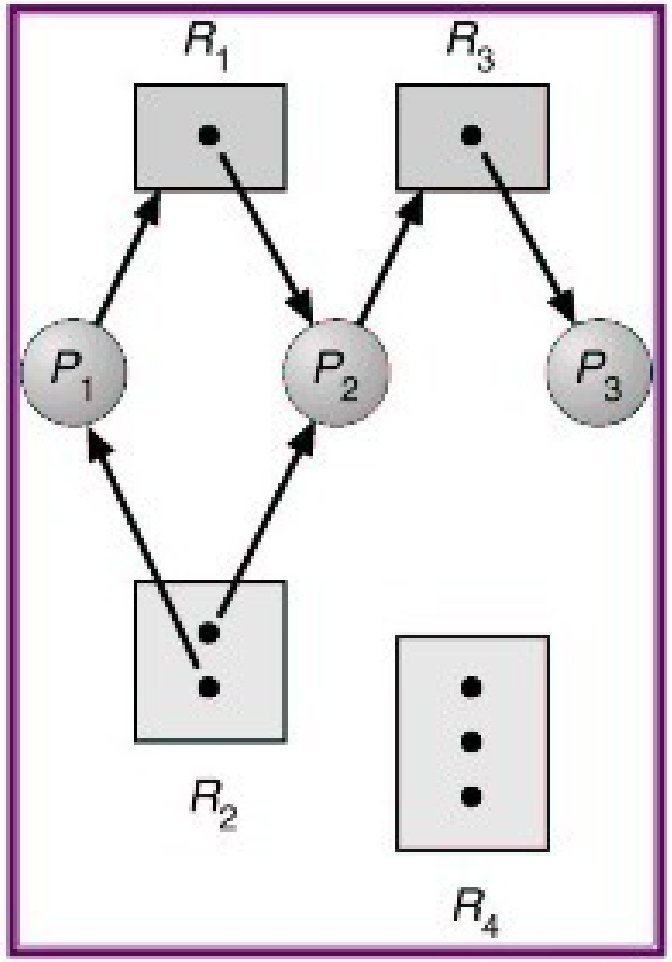
resource allocated : 

Resource Allocation Graph

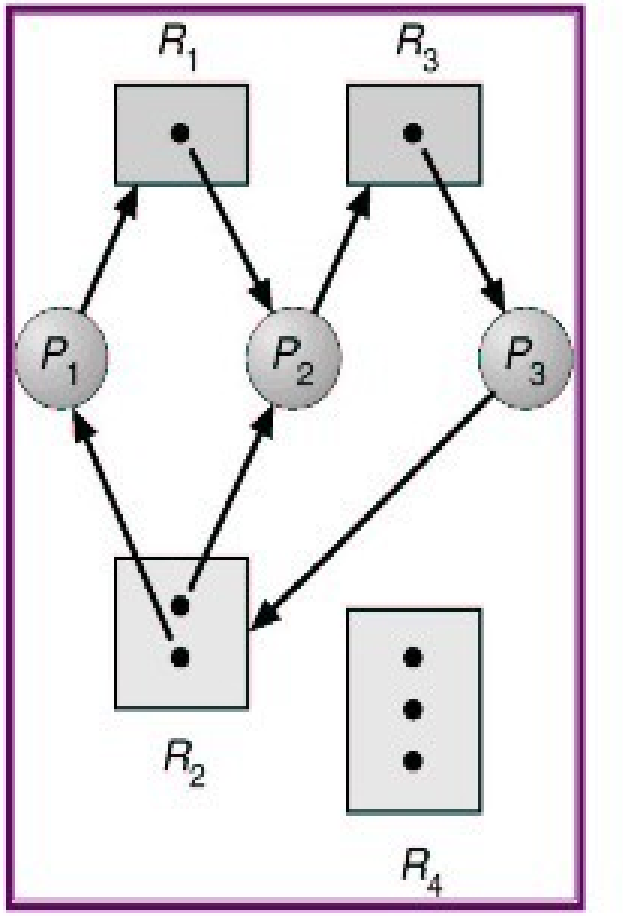
- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$
- **Basic Facts:**

1. If graph contains no cycles ⇒ no deadlock.
2. If graph contains a cycle ⇒
   a) If only one instance per resource type, then deadlock.
   b) If several instances per resource type, possibility of Deadlock

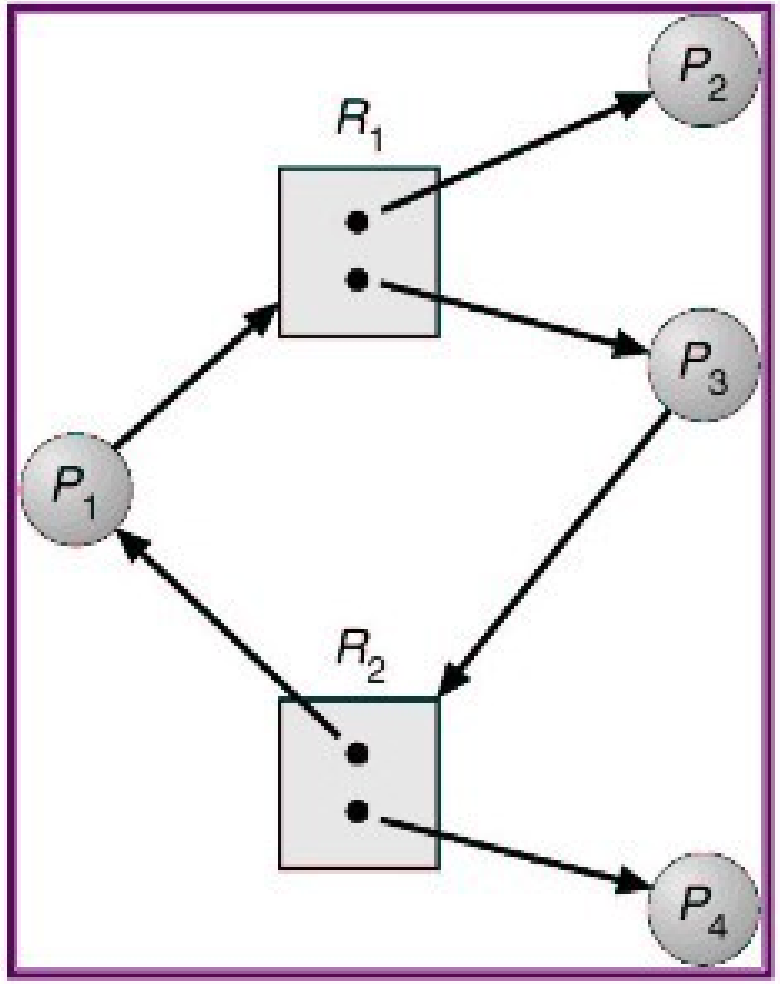
Resource Allocation Graph



Resource Allocation Graph With Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



Methods for Handling Deadlock

1. Allow system to enter deadlock and then recover
   - Requires deadlock detection algorithm
   - Some technique for forcibly preempting resources and/or terminating tasks
2. Ensure that system will never enter a deadlock
   - Need to monitor all lock acquisitions
   - Selectively deny those that might lead to deadlock
3. Ignore the problem and pretend that deadlocks never occur in the system
   - Used by most operating systems, including UNIX(Ostrich algorithm)

Deadlock Prevention

- To prevent the system from deadlocks, one of the four discussed conditions that may create a deadlock should be discarded.
- The methods for those conditions are as follows:

1. Mutual Exclusion:

- In system all resources are non-sharable.
- Some resources like printers, processing units are non-sharable.
- So it is not possible to prevent deadlocks by denying mutual exclusion

2. Hold and Wait:

- One protocol to ensure that hold-and-wait condition never occurs says *each process must request and get all of its resources before it begins execution*.
- Another protocol is *"Each process can request resources only when it does not occupies any resources."*
- The second protocol is better. However, both protocols cause low resource utilization and starvation.

3. No Pre-emption:

- Release any resource already being held if the process can't get an additional resource.
- Allow preemption - if a needed resource is held by another process, which is also waiting on some resource, steal it,Otherwise wait.

4. Circular Wait:

- One protocol to ensure that the circular wait condition never holds is “Impose a linear ordering of all resource types.” Then, each process can only request resources in an increasing order of priority.
- For example, set priorities for r1 = 1, r2 = 2, r3 = 3, and r4 = 4. With these priorities, if process P wants to use r1 and r3, it should first request r1, then r3.

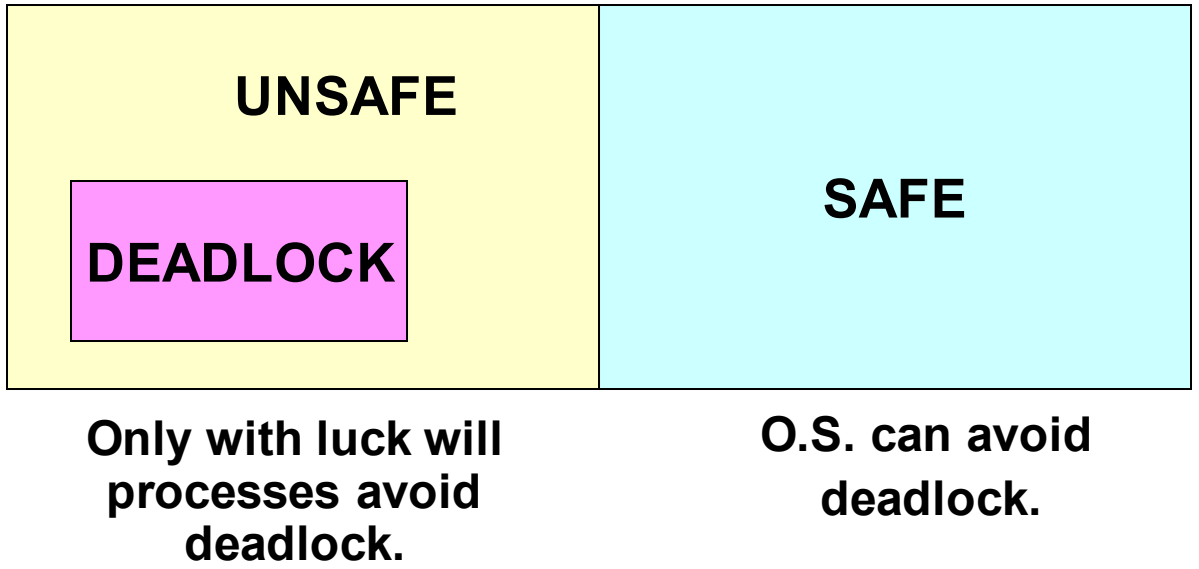
Deadlock Prevention Conclusion

- EACH of these prevention techniques may cause a decrease in utilization and/or resources.
- For this reason, prevention isn't necessarily the best technique.
- Prevention is generally the easiest to implement.

Deadlock Avoidance

- If we have prior knowledge of how resources will be requested, it's possible to determine if we are entering an "unsafe" state.
- Possible states are:
  1. **Deadlock** No forward progress can be made.
  2. **Unsafe state** A state that **may** allow deadlock.
  3. **Safe state** A state is safe if a sequence of processes exist such that there are enough resources for the first to finish, and as each finishes and releases its resources there are enough for the next to finish.
- The rule is simple: If a request allocation would cause an unsafe state, do not honor that request.
- **NB: All deadlocks are unsafe, but all unsafe are NOT deadlocks.**

Deadlock Avoidance



Two deadlock avoidance algorithms:

1. Resource-allocation Graph Algorithm
2. Banker's Algorithm

Resource-allocation Graph Algorithm

- Only applicable when we only have 1 instance of each resource type
- Claim edge (dotted edge), like a future request edge
- When a process requests a resource, the claim edge is converted to a request edge
- When a process releases a resource, the assignment edge is converted to a claim edge

Bankers Algorithm

- Resource allocation state is defined by the number of available and allocated resources and the maximum demand of the processes.
- Checks if immediate allocation n leaves the system in a safe state.
- The system is in a safe state if there exists a safe sequence of all processes
- A state is safe if the system can allocate resources to each process in some order avoiding a deadlock.

➢Customer =Processes

➢Units = Resources

➢Bankers=OS

- The Banker algorithm does the simple task

1. If granting the request leads to an unsafe state the request is denied.
2. If granting the request leads to safe state the request is carried out.

- Basic Facts:

1. If a system is in safe state ⇒ no deadlocks.
2. If a system is in unsafe state ⇒ possibility of deadlock.
3. Avoidance ⇒ ensure that a system will never enter an unsafe state.

- Bankers Algorithms for a single resource:

Customer	Used	Max
A	0	6
B	0	5
C	0	4
D	0	7

- Available units: 10
- In the above fig, we see four customers each of whom has been granted a certain no. of credit units
- The Banker reserved only 10 units rather than 22 units to service them since not all customer need their maximum credit immediately.

- At a certain moment the situation becomes:

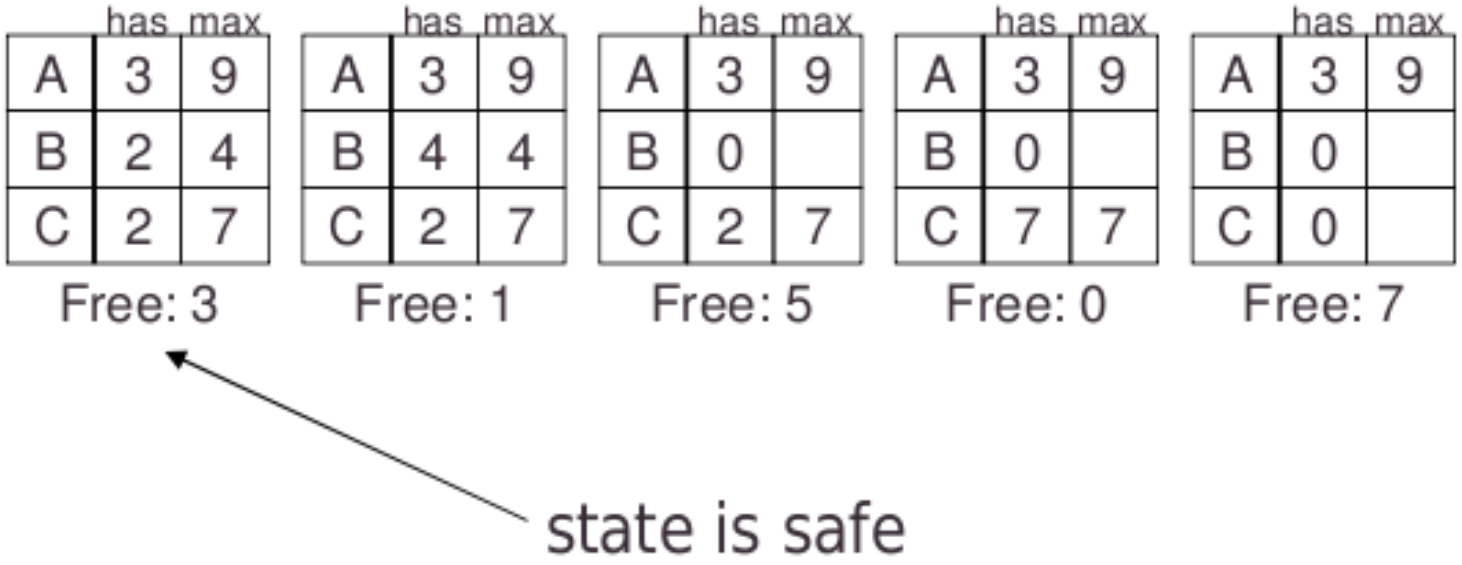
Customer	Used	Max
A	1	6
B	1	5
C	2	4
D	4	7

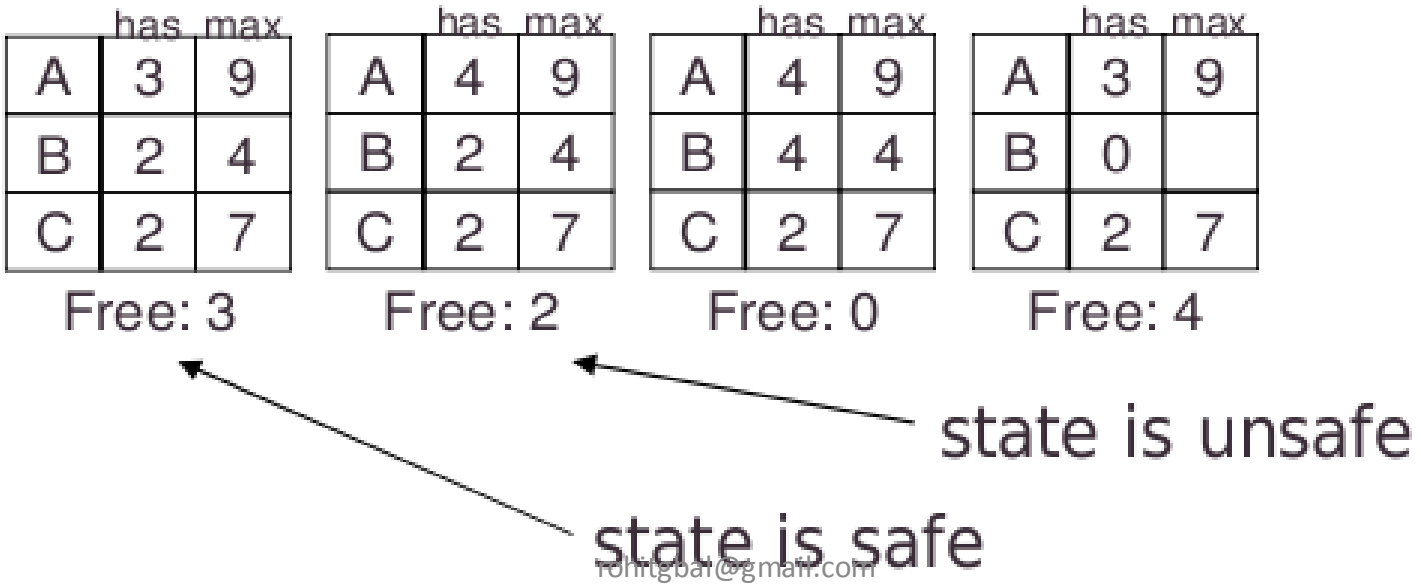
- Available units: 2
- Safe State: With 2 units left, the banker can delay any requests except C's, thus letting C finish and release all four of his resources. With four in hand, the banker can let either D or B have the necessary units & so on.
- Unsafe State: B requests one more unit and is granted.

Customer	Used	Max
A	1	6
B	2	5
C	2	4
D	4	7

- Available units: 1
- This is an unsafe condition. If all of the customer namely A, B,C & D asked for their maximum loans, then Banker couldn't satisfy any of them and we would have deadlock.
- It is important to note that an unsafe state does not imply the existence or even eventual existence of a deadlock.
- What an unsafe does imply is that some unfortunate sequence of events might lead a deadlock.

- **D**

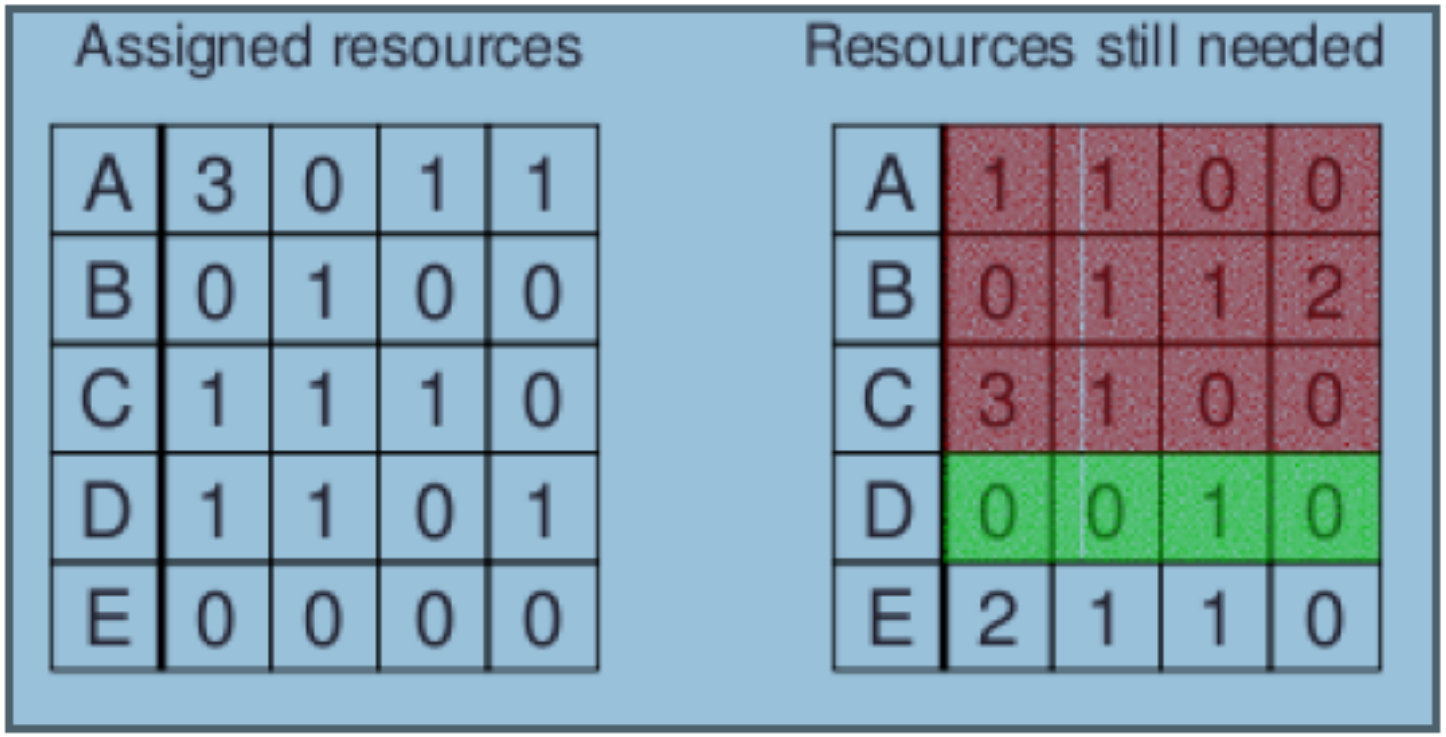

has max
A 3 9
B 2 4
C 2 7
Free: 3
has max
A 3 9
B 4 4
C 2 7
Free: 1
has max
A 3 9
B 0
C 2 7
Free: 5
has max
A 3 9
B 0
C 7 7
Free: 0
has max
A 3 9
B 0
C 0
Free: 7
state is safe


has max
A 3 9
B 2 4
C 2 7
Free: 3
has max
A 4 9
B 2 4
C 2 7
Free: 2
has max
A 4 9
B 4 4
C 2 7
Free: 0
has max
A 3 9
B 0
C 2 7
Free: 4
state is unsafe
state is safe

Bankers Algorithms for Multiple Resources

- The algorithm for checking to see if a state is safe can now be stated.

1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion.
2. Assume the process of the row chosen requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to the A vector.
3. Repeat steps 1 and 2 until either all processes are marked terminated, in which case the initial state was safe, or until a deadlock occurs, in which case it was not.



a). Current Allocation Matrix b). Request Matrix

- E (Existing Resources): (6 3 4 2)
- P (Processed Resources): (5 3 2 2)
- A (Available Resources): (1 0 2 0)

- **Solution:**
- Process A, B & C can't run to completion since for Process for each process, Request is greater than Available Resources. Now process D can complete since its requests row is less than that of Available resources.
- **Step 1:**
- When D run to completion the total available resources is:
- A = (1, 0, 2, 0) + (1, 1, 0 , 1)= (2, 1, 2, 1)
- Now Process E can run to completion
- **Step 2:**
- Now process E can also run to completion & return back all of its resources.
- A = (0 , 0, 0, 0) + (2, 1, 2, 1) = (2, 1, 2, 1)

- **Step 3:**
- Now process A can also run to completion leading A to
- (3, 0, 1, 1) + (2, 1, 2, 1) = (5, 1, 3, 2)
- **Step 4:**
- Now process C can also run to completion leading A to
- (5, 1, 3, 2) + (1, 1, 1, 0) = (6, 2, 4, 2)
- **Step 5:**
- Now process B can run to completion leading A to
- (6, 2, 4, 2) + (0, 1, 0 , 0) = (6, 3, 4, 2)
- This implies the state is safe and Dead lock free.

- A system has three processes and four allocable resources. The total four resource types exist in the amount as E= (4, 2, 3, 1). The current allocation matrix and request matrix are as follows: Using Bankers algorithm, explain if this state is deadlock safe or unsafe.

Current Allocation Matrix				
Process	Ro	R1	R2	R3
P0	0	0	1	0
P1	2	0	0	1
P1	0	1	2	0

Allocation Request Matrix				
Process	Ro	R1	R2	R3
P0	2	0	0	1
P1	1	0	1	0
P1	2	1	0	0

- **Avaliable(2,1,3,1)**

Before Executing Process P2

	R0	R1	R2	R3
P0	0	0	1	0
P1	2	0	0	1
P2	**2**	**2**	**2**	**0**

A(0,0,3,1)

After Executing Process P2

	R0	R1	R2	R3
P0	0	0	1	0
P1	2	0	0	1
P2	**0**	**0**	**0**	**0**

A(2,2,5,1)

Before Executing Process P0

	R0	R1	R2	R3
P0	**2**	**0**	**1**	**1**
P1	2	0	0	1
P2	0	0	0	0

A(0,2,5,0)

After Executing Process P0

	R0	R1	R2	R3
P0	**0**	**0**	**0**	**0**
P1	2	0	0	1
P2	0	0	0	0

A(2,2,6,1)

Before Executing Process P1

	R0	R1	R2	R3
P0	0	0	0	0
P1	**3**	**0**	**1**	**1**
P2	0	0	0	0

A(1,2,5,1)

After Executing Process P1

	R0	R1	R2	R3
P0	0	0	0	0
P1	**3**	**0**	**1**	**1**
P2	0	0	0	0

A(4,2,6,2)

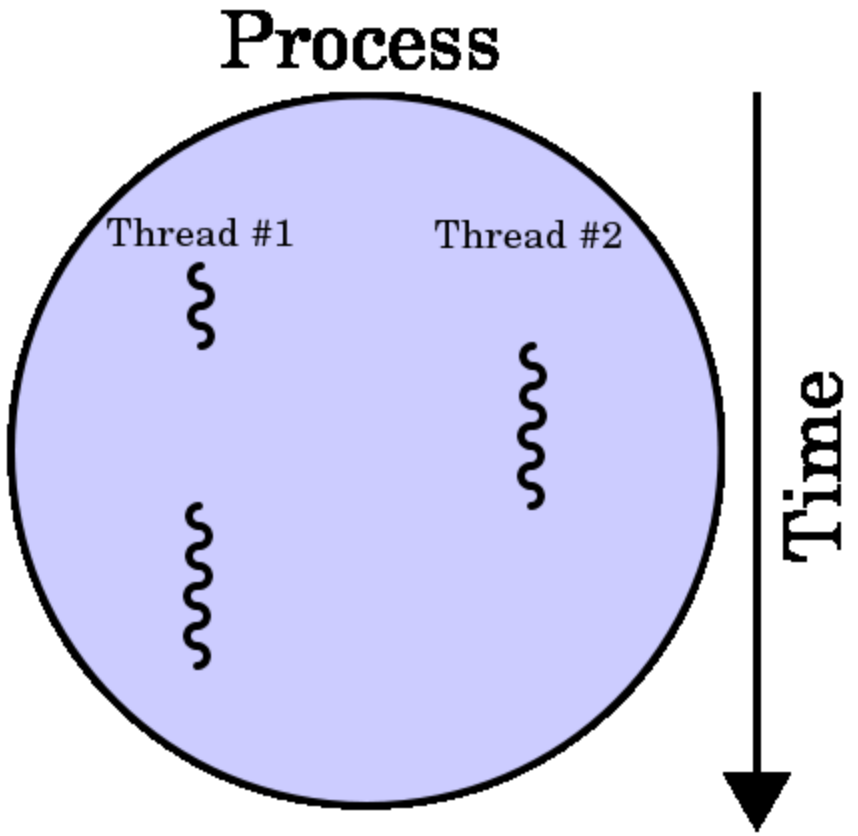
Starvation vs. Deadlock

- **Starvation:** thread waits indefinitely
- Example, low-priority thread waiting for resources constantly in use by high-priority threads
- **Deadlock:** circular waiting for resources
- Thread A owns Res 1 and is waiting for Res 2 Thread B owns Res 2 and is waiting for Res 1
- Deadlock is Starvation but not vice versa
- Starvation can end
- Deadlock can’t end without external intervention

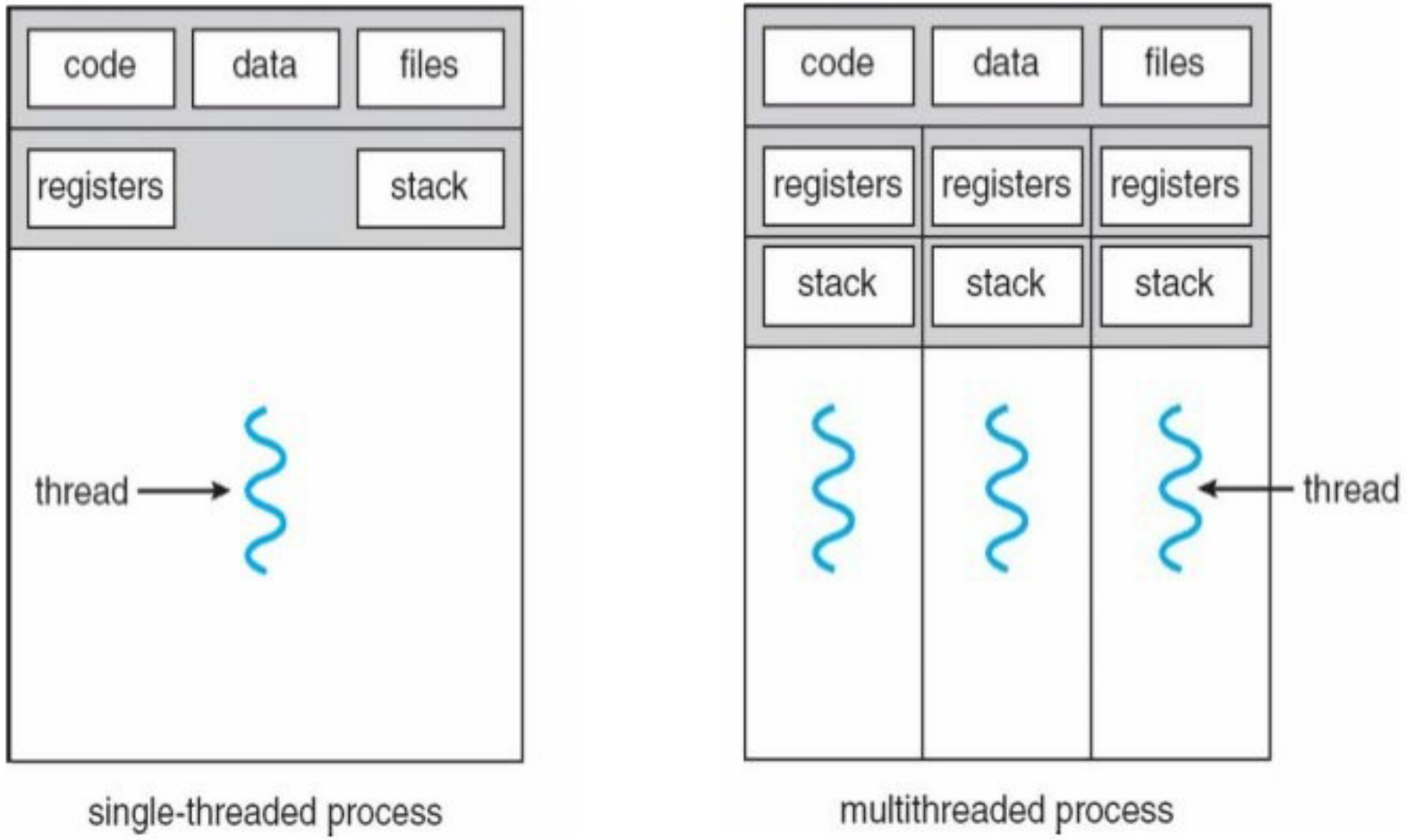
THREADS

- Light Weight Process
- Process may contain one or more thread
- A thread is a basic unit of CPU utilization
- Separate --Thread ID, program counter, register set and stack.
- Shares -- code section, data section, and other operating system resources, such as open files and signals
- It improves the performance in execution of program with parallelism
- Web Server

THREADS



THREADS


code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

ADVATAGES OF THREADS

- Thread **minimize** context switching time.
- Use of threads provides **concurrency** within a process.
- **Efficient** communication.
- **Economy**- It is more economical to create and context switch threads.
- Utilization of multiprocessor architectures to a greater scale and efficiency

TYPES OF THREAD

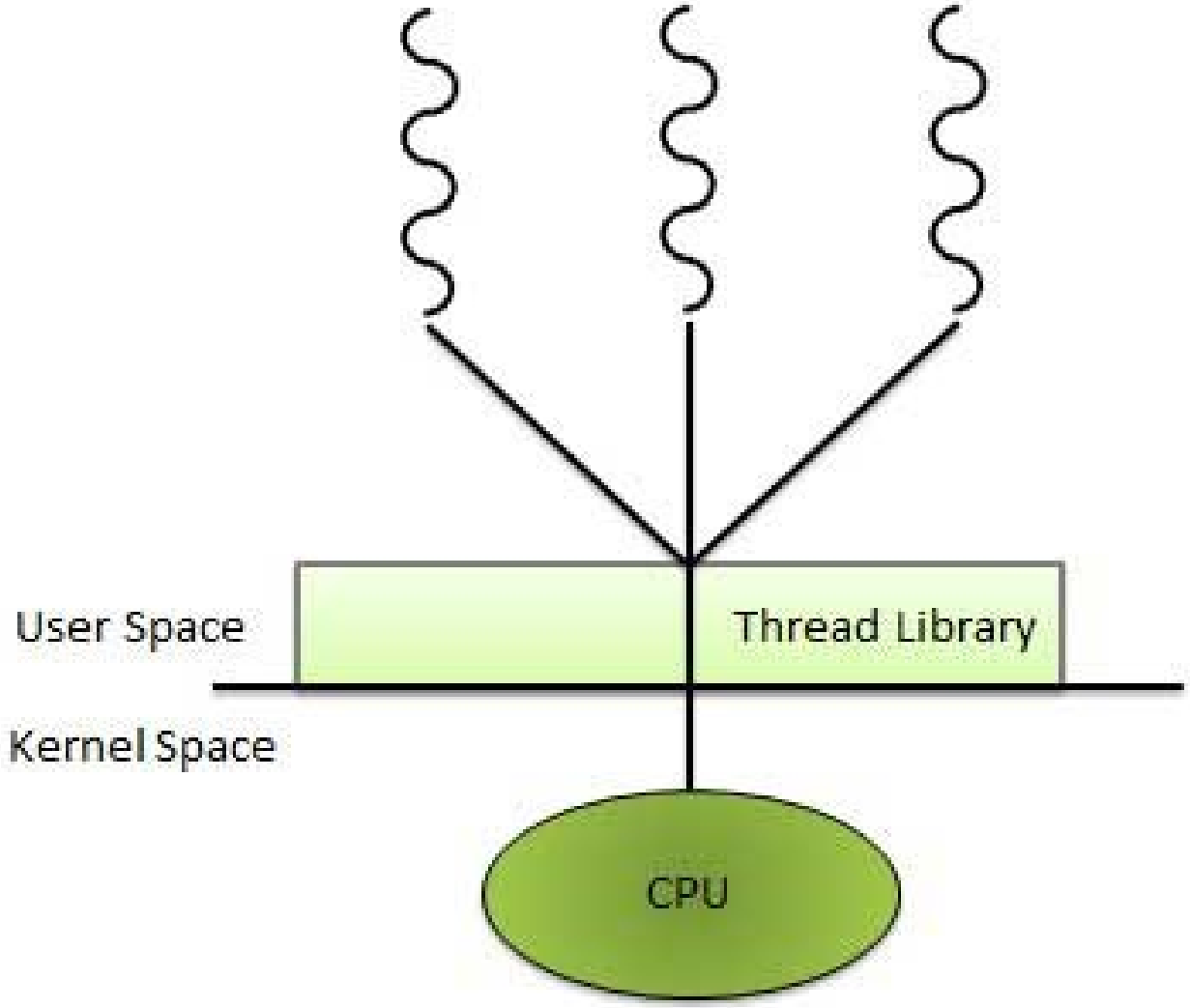
Threads are implemented in following two ways

1. **User Level Threads** -- User managed threads
2. **Kernel Level Threads** -- Operating System managed threads acting on kernel

USER LEVEL THREADS

- Application manages thread(user)
- Kernel is not aware of the existence of threads.
- The **thread library contains code** for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts.
- The application begins with a single thread and begins running in that thread.

USER LEVEL THREADS



USER LEVEL THREADS

ADVANTAGES

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

DISADVANTAGES

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

KERNEL LEVEL THREADS

- Thread management done by the Kernel.
- There is no thread management code in the application area.
- Kernel threads are supported directly by the operating system.
- The Kernel maintains context information for the process as a whole and for individuals threads within the process.
- Scheduling by the Kernel is done on a thread basis.
- The Kernel performs thread creation, scheduling and management in Kernel space.

KERNEL LEVEL THREADS

ADVANTAGES

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can multithreaded.

DISADVANTAGES

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

MULTITHREADING MODELS(THREAD MODEL)

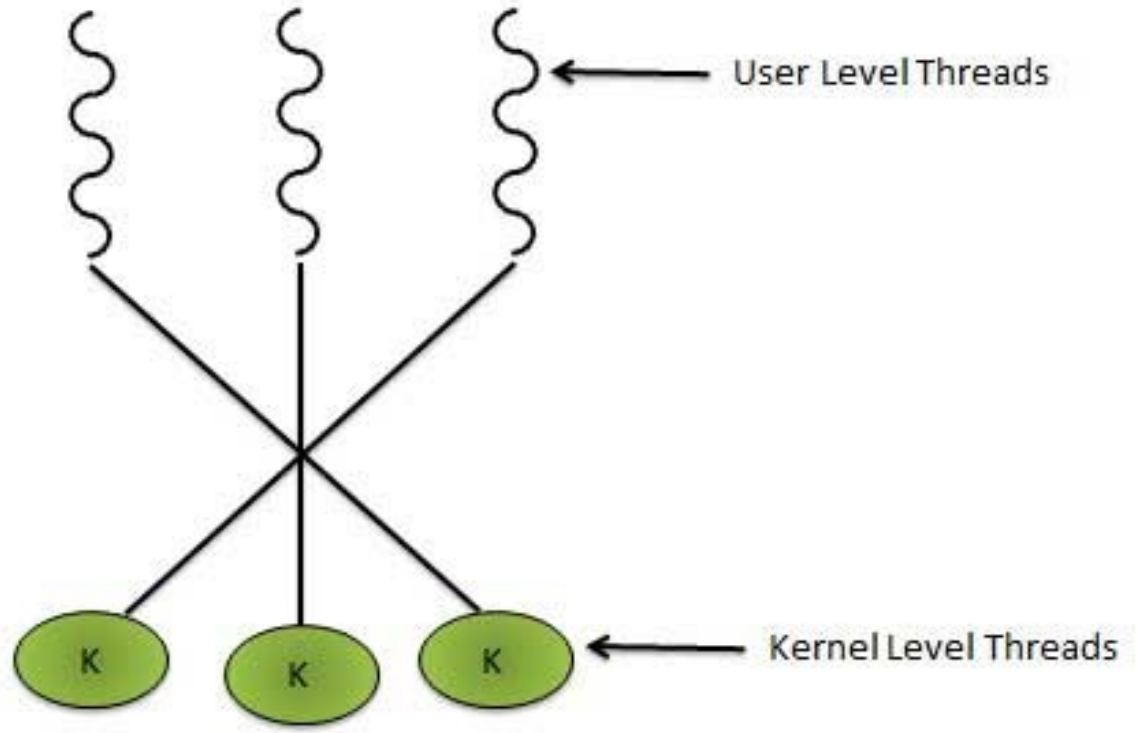
- Some operating system provide a combined user level thread and Kernel level thread facility.
- Solaris is a good example of this combined approach.
- Multiple threads within the same application can run in **parallel** on multiple processors and a **blocking system call need not block the entire process**.

Multithreading models are three types

1. Many to many relationship.
2. Many to one relationship.
3. One to one relationship.

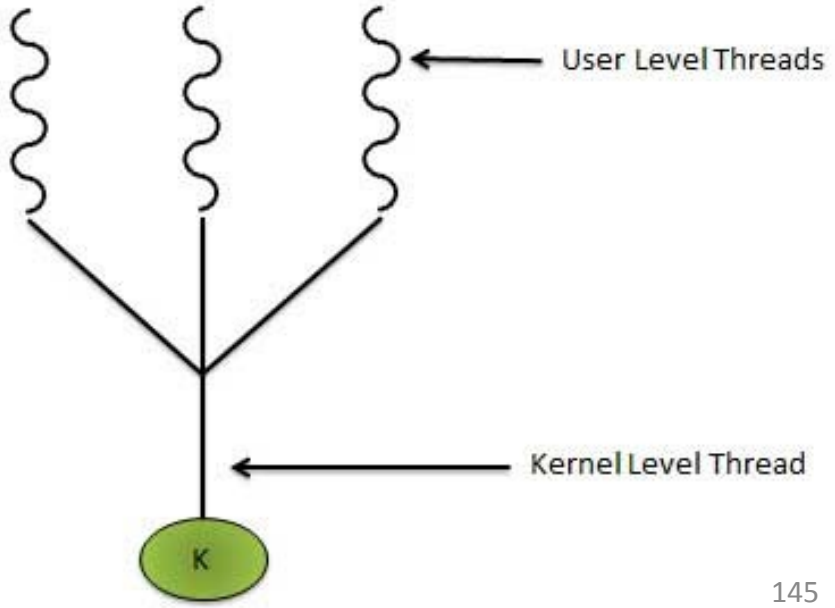
MANY TO MANY MODEL

- Many user level threads multiplexes to the Kernel thread of smaller or equal numbers.
- Number of Kernel threads may be specific to either a **particular application or a particular machine.**
- **Examples:**
  - Solaris Green Threads
  - GNU Portable Threads



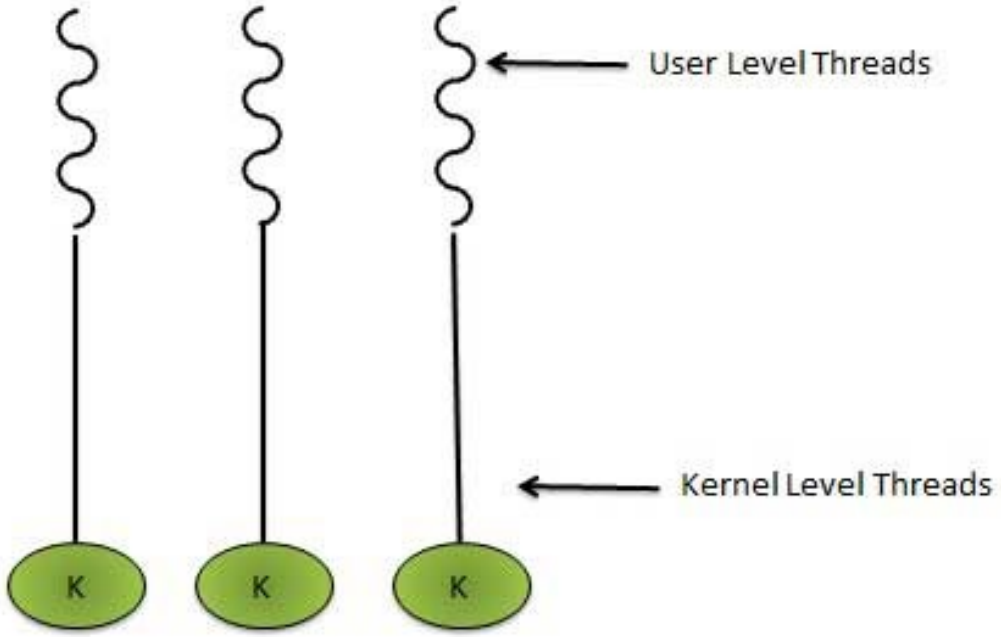
MANY TO ONE MODEL

- Many to one model maps many user level threads to one Kernel level thread
- Thread management is done in user space
- Blocking system call, the entire process will be blocks
- **Unable to run in parallel on multiprocessors**
- **Examples**
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

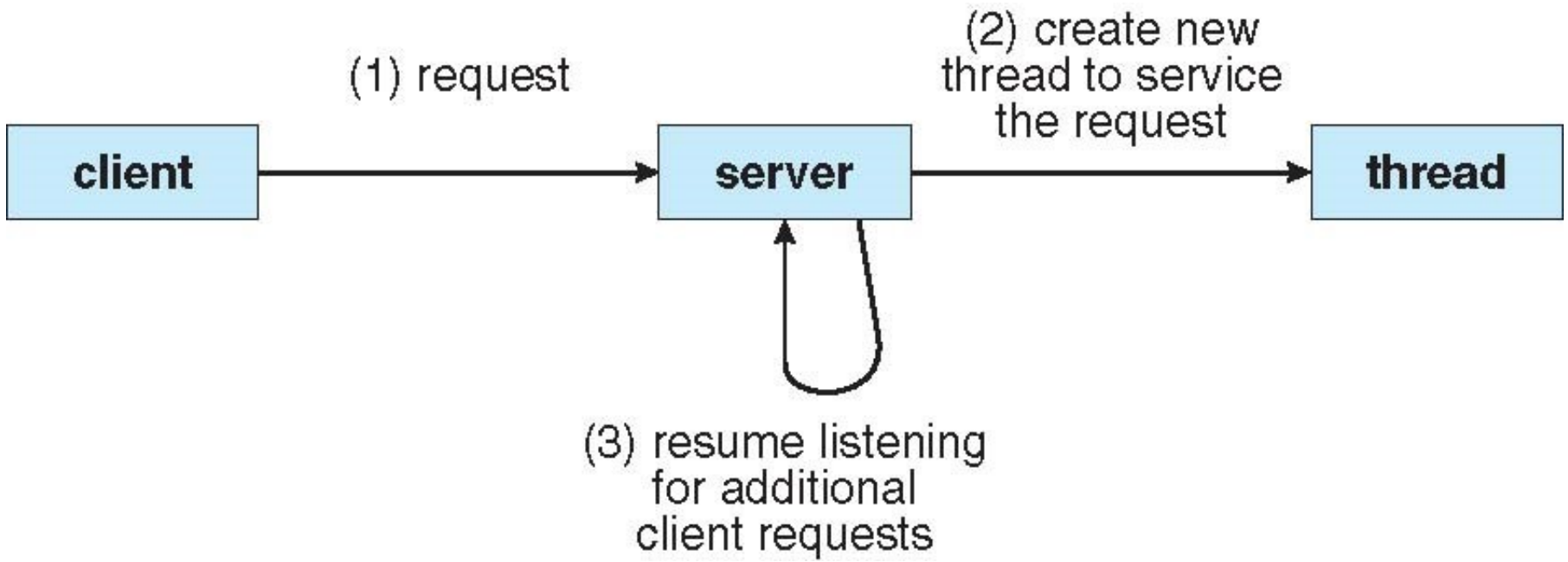


ONE TO ONE MODEL

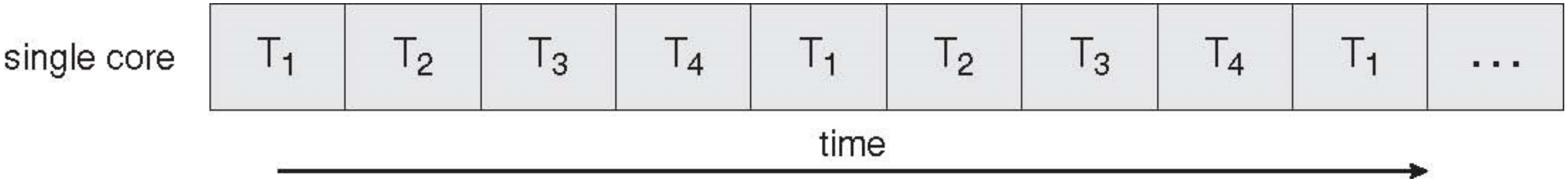
- One to one relationship of user level thread to the kernel level thread.
- Model provides more concurrency than the many to one model.
- When a thread makes a blocking system call
- It support multiple thread to execute in parallel on microprocessors
- **Examples**
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



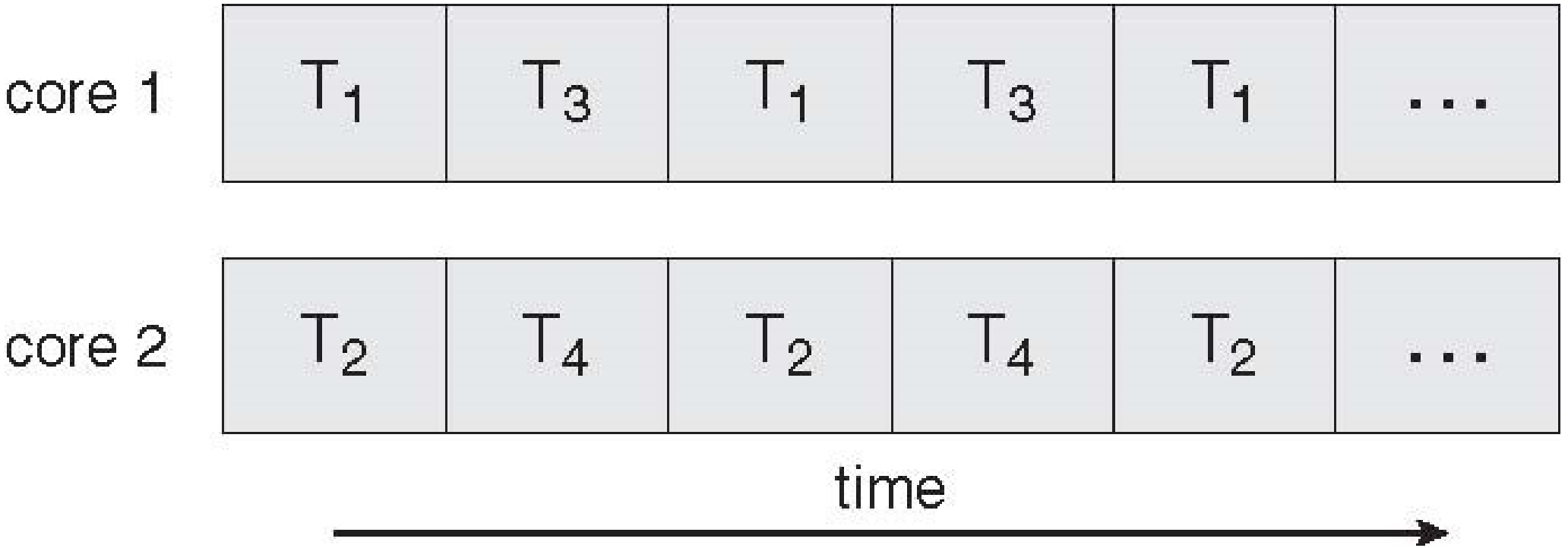
Multithreaded Server Architecture



CONCURRENT THREAD EXECUTION ON A SINGLE-CORE SYSTEM



CONCURRENT THREAD EXECUTION ON A MULTI-CORE SYSTEM



THREAD USAGE

- The main reason for having threads is that in many applications, **multiple activities are going on at once**. Some of these may block from time to time.
- By decomposing such an application into multiple sequential threads that run in **quasi-parallel**, the programming model becomes simpler.
- Ability for the parallel entities to **share an address space and all of its data among themselves**.
- This ability is essential for certain applications, which is why having multiple processes (with their separate address spaces) will not work

- They are **lighter weight than processes**, they are easier (i.e., faster) to create and destroy than processes
- Creating a thread goes **10-100 times faster** than creating a process.
- When the number of threads needed changes dynamically and rapidly, this property is useful

A third reason for having threads is also a performance argument.

Threads yield no performance gain when all of them are CPU bound, but when there is **substantial computing and also substantial I/O**, having threads allows these activities to overlap, thus speeding up the application.

PROCESS	THREAD
• Process is heavy weight or resource intensive	• Thread is light weight taking lesser resources than a process
• Process switching needs interaction with operating system	• Thread switching does not need to interact with operating system
• In multiple processing environments each process executes the same code but has its own memory and file resources	• All threads can share same set of open files, child processes
• If one process is blocked then no other process can execute until the first process is unblocked	• While one thread is blocked and waiting, second thread in the same task can run
• Multiple processes without using threads use more resources	• Multiple threaded processes use fewer resources
• In multiple processes each process operates independently of the others.	• One thread can read, write or change another thread's data