unacademy

Operating System:
Threads & Multi-threading

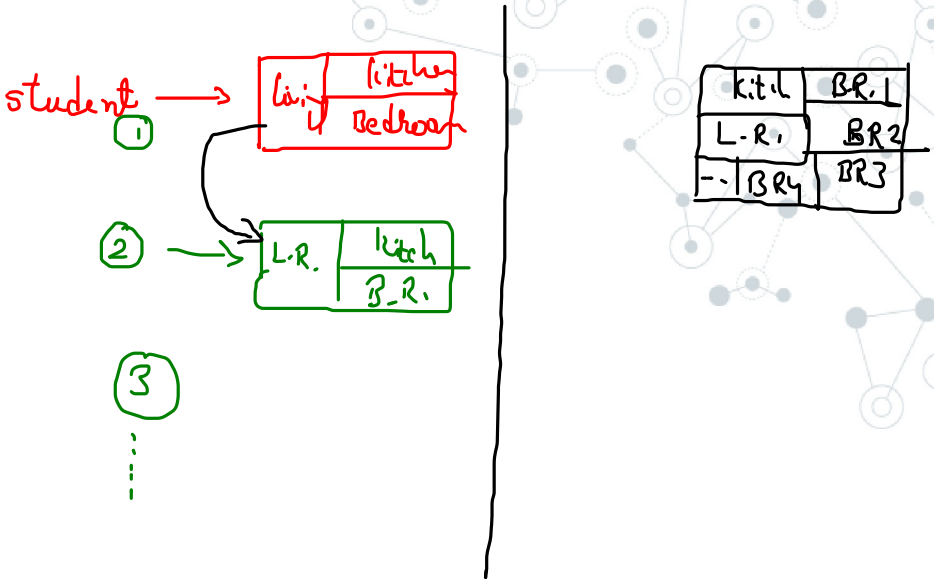
By: Vishvadeep Gothi

Thread

Component of process

or

Lightweight Process



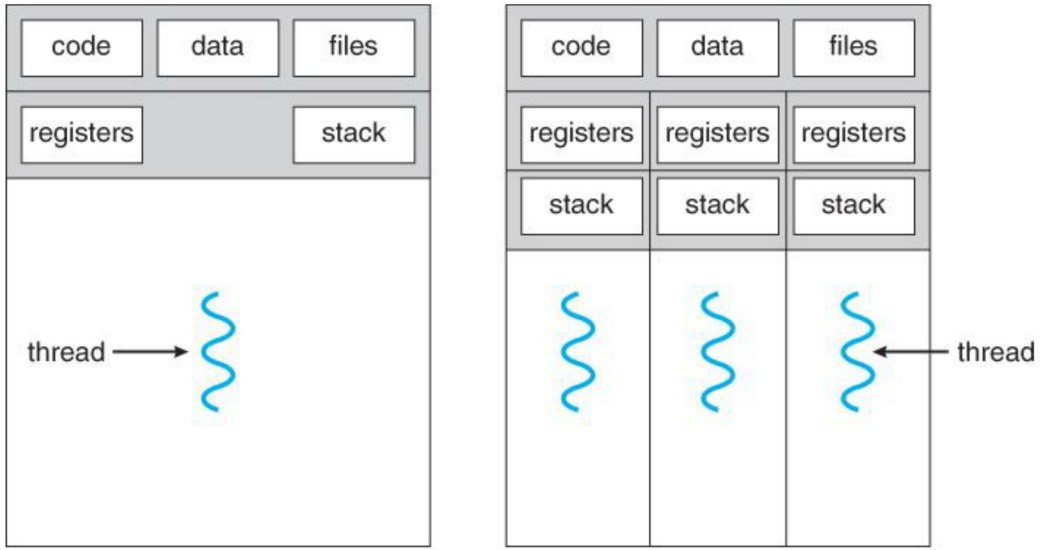
- Provide a way to improve application performance through parallelism

unacademy

Thread

Shared Among Threads	Unique For Each Thread
Code Section	Thread Id
Data Section	Register Set
OS Resources	Stack
Open Files & Signals	Program Counter

Thread



single-threaded process

multithreaded process

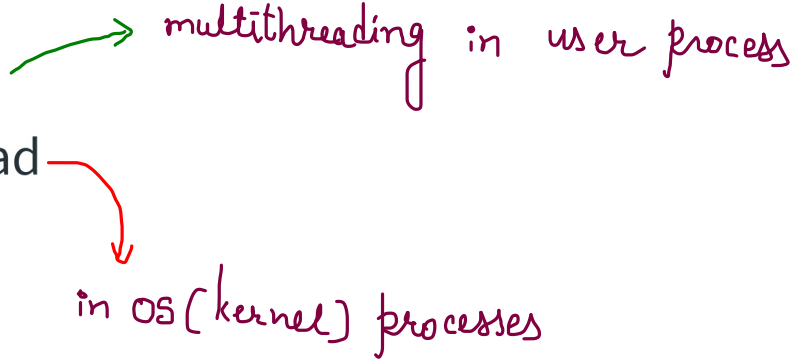
unacademy

Advantages of Multithreading

1. Responsiveness
2. Faster Context Switch
3. Resource Sharing
4. Economy
5. Communication
6. Utilization of Multiprocessor Architecture

Types of Thread

1. User Level Thread
2. Kernel Level Thread



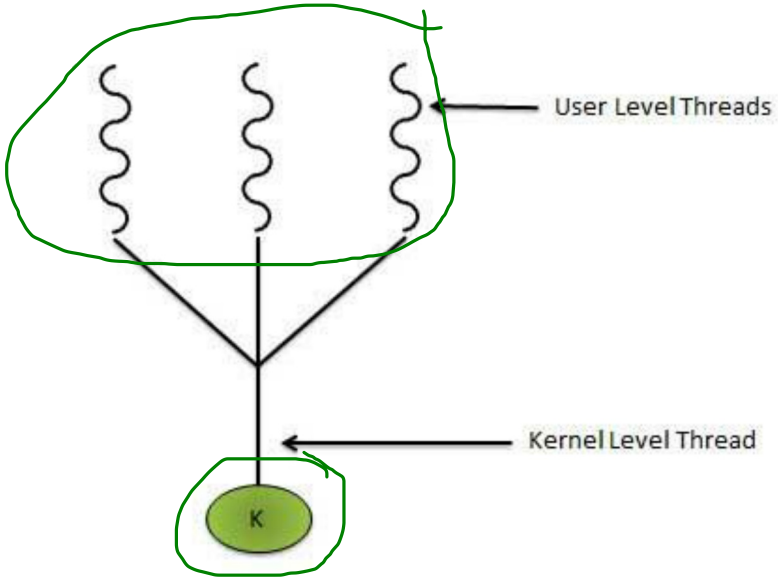
Types of Thread

User Threads	Kernel Thread
Multithreading in user process	Multithreading in kernel process
Created without kernel intervention	Kernel itself is multithreaded
Context switch is very fast	Context switch is slow
If one thread is blocked, OS blocks entire process	Individual thread can be blocked
Generic and can run on any OS	Specific to OS
Faster to create and manage	Slower to create and manage

Multithreading Model

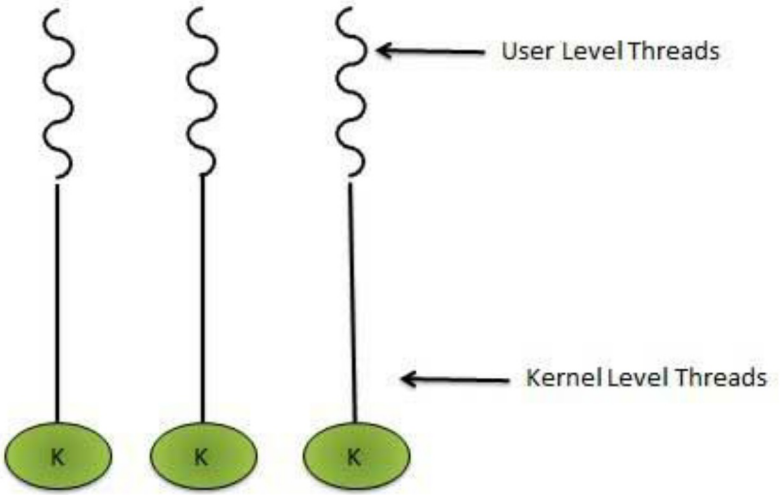
1. Many-to-One Model
2. One-to-One Model
3. Many-to-Many Model

Many-to-One



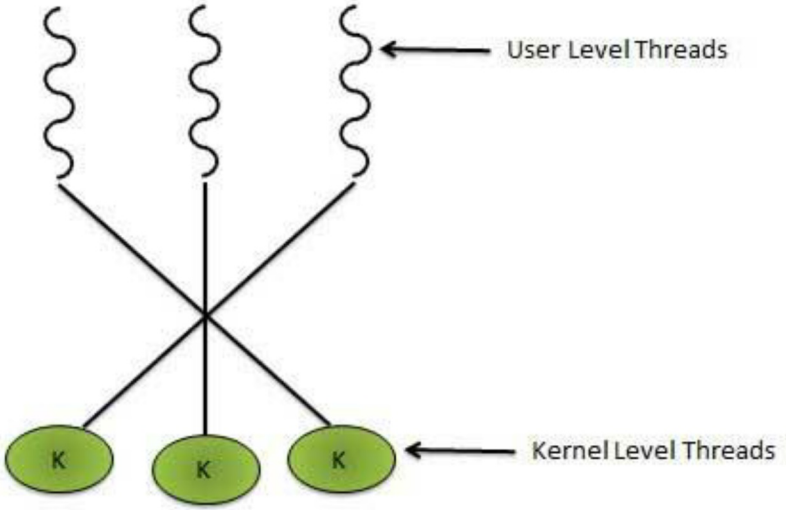
unacademy

One-to-One



unacademy

Many-to-Many



unacademy

unacademy

Operating System:
Process Synchonization

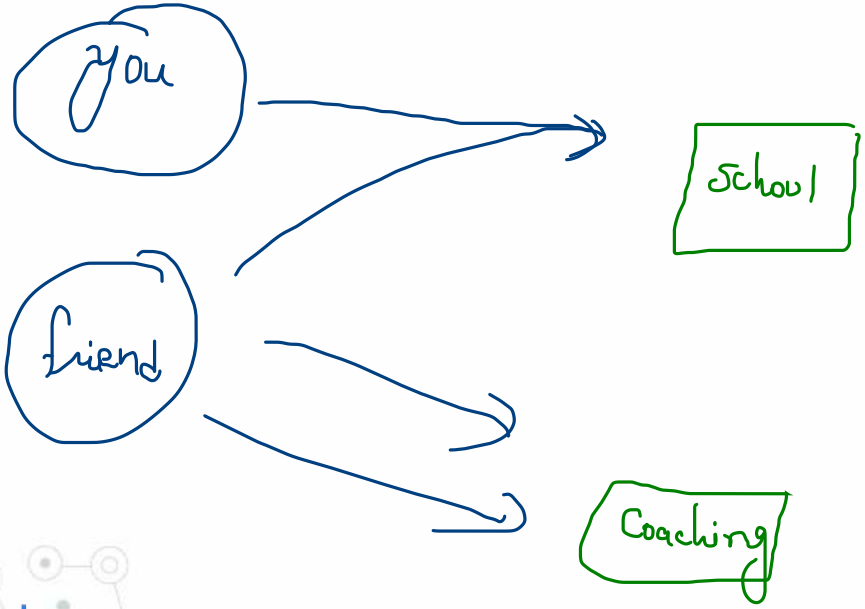
By: Vishvadeep Gothi

Types of Processes

1. Independent → no any communication with any other process → no synchronization
2. Cooperating/Coordinating/Communicating → Synch. required.
   - can affect other process
     or
     can be affected by other process.

Need Of Synchronization

↳ in communicating processes



7am - 1pm school

✗ 1PM - 4PM home

4PM — 6PM Coaching

6 — 8 PM Playing

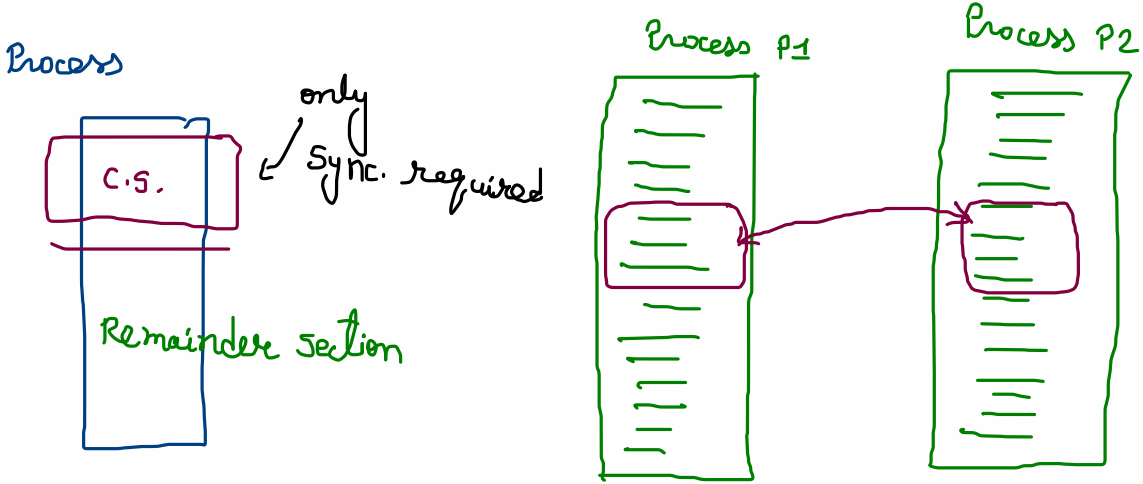
Problems Without Synchronization

- Problems without Synchronization:
  - Inconsistency
  - Loss of Data
  - Deadlock

Critical Section

Communication happens

The critical section is a code segment where the shared variables can be accessed .



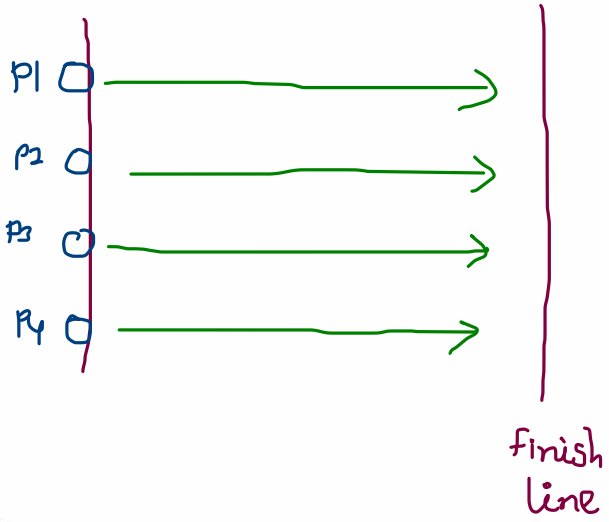
unacademy

Where Synchronization Required?

↳ only in critical section

Race Condition → because of lack of synchronization.

A race condition is an undesirable situation, it occurs when the final result of concurrent processes depends on the sequence in which the processes complete their execution.



Memory

a = 5

Process P1

R1 = a

R1 = R1+2

a = R1

R1 = 5

R1 = 7

Process P2

R3 = a

R3 = R3+5

a = R3

R3 = 5

R3 = 10

if P1 finishes first

a = ~~5~~ ~~7~~ 10

if P2 finishes first

a = ~~5~~ ~~10~~ 7

Solution of Critical Section Problem

- Requirements of Critical Section problem solution:
  1. Mutual Exclusion
  2. Progress
  3. Bounded Waiting

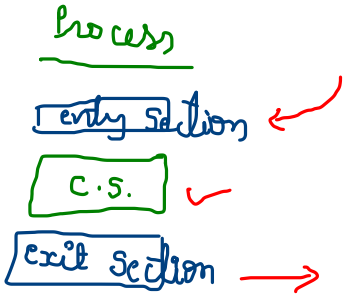
unacademy

Operating System:
Solution of Critical Section

By: Vishvadeep Gothi

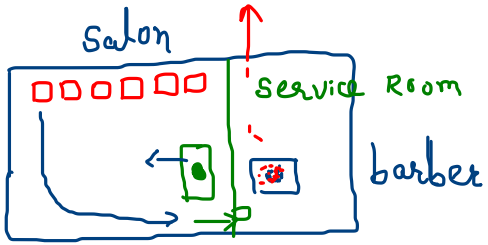
Critical Section

The critical section is a code segment where the shared variables can be accessed



Solution of Critical Section Problem

- Requirements of Critical Section problem solution:
  1. Mutual Exclusion → If one process is using c.s. then other process can not use that c.s.
  2. Progress → If no any process in c.s. & atleast one process wants to enter in c.s. then it should be allowed.
  3. Bounded Waiting → waiting of processes should be bounded.



unacademy

Solution 1: Using Lock

Lock = ~~false~~ ~~true~~ true

```
Boolean lock=false;
```

P0

```
while(true)
{
    while(lock);
    lock=true;
      CS
    lock=false;
      RS;
}
```

P1

```
while(true)
{
    while(lock);
    lock=true;
      CS
    lock=false;
      RS;
}
```

Does not satisfy mutual exclusion

↳ If a process preempted after while(lock); stmt

Progress is Satisfied.

Bounded waiting is not there

Solution 2: Using Turn

turn = 0 \ 1

```
int turn=0;
```

P0

```
while(true)
{
    while(turn!=0);
      CS
    turn=1;
      RS;
}
```

P1

```
while(true)
{
    while(turn!=1);
      CS
    turn=0;
      RS;
}
```

→ Bounded waiting is there

→ Mutual exclusion is satisfied

→ Progress is not satisfied

2 process will execute only in strict alternate manner.

Solution 3: Peterson's Solution

```
Boolean Flag[2];
int turn;
```

flag[0] = false
flag[1] = false

P0

```
while(true) {
    Flag[0]=true;
    turn=1;
    while(Flag[1] && turn==1);
        CS
    Flag[0]=False;
        RS;
}
```

P1

```
while(true) {
    Flag[1]=true;
    turn=0
    while(Flag[0] && turn==0);
        CS
    Flag[1]=False;
        RS;
}
```

- M.E. is satisfied
- Progress is also satisfied
- Bounded waiting satisfied

unacademy

Case 1:- Enter P0 in C.S. & try if P1 can enter

flag[0] = ~~f~~ T

flag[1] = ~~f~~ T

turn = ~~1~~ 0

P1 can not enter into C.S.

Checking for bounded waiting :-

P0 enters in C.S.

P1 waits outside C.S.

P0 comes out

flag[0] = ~~f~~ ~~t~~ ~~f~~ t

flag[1] = ~~f~~ t

turn = ~~1~~ ~~0~~ 1

Solution 3: Peterson's Solution

✓

All 3 solutions ⇒ S/W solutions

unacademy

Operating System:
Synchronization Hardware

By: Vishvadeep Gothi

Synchronization Hardware

1. TestAndSet()
2. Swap()

instruction support by CPU

⇓

can be used to provide synchronization

⇓

privileged instns

TestAndSet()

Returns the current value flag and sets it to true

lock =

TestAndSet (lock)

Busy waiting

a process executes on CPU but can not go further because of C.S. solution. Then the process is known to be in busy waiting

TestAndSet()

```
Boolean Lock=False;

boolean TestAndSet(Boolean *trg)
{
    boolean rv = *trg;
    *trg = True;
    Return  rv;
}
```

atomic

↳ you can't have preemption in between

P0

```
while(true)
{
  while(TestAndSet(&Lock));
        ↳ CS
  Lock=False;
}
```

false

P1

while (TestAndSet(&lock));

* ME ✓
* Progress ✓
* Bounded waiting ✗

Swap()

key = ~~True~~ ~~True~~ F

Lock = ~~False~~ T

Boolean Key, Lock=False;

```
void Swap(Boolean *a, Boolean *b)
{
   boolean temp = *a;
   *a=*b;
   *b=temp;
}
```

```
while(true)
{
   Key = True;
   while (key==True)
        Swap(&Lock, &Key);
   CS
   Lock=False;
      RS
}
```

P0

* M E ✓
* Progress ✓
* Bounded waiting ✗

No M.E. if preemption happens in process.

unacademy

unacademy

Operating System:

Synchronization Tool: Semaphore

By: Vishvadeep Gothi

Synchronization Tools

1. Semaphore
2. Monitor

Semaphore

- Integer value which can be accessed using following functions only
  - wait() / P() / Degrade() / lower() / down()
  - signal() / V() / Upgrade() / up()

atomic

wait() & signal()

```
wait(S)
{
   while(S<=0);
   S--;
}
```

```
signal(S)
{
   S++;
}
```

Types of Semaphore

Binary Semaphore

only 2 values → 0 or 1

Counting Semaphore

unlimited domain of values (any integer)

Types of Semaphore

Binary Semaphore	Counting Semaphore
It is used to implement the solution of critical section problems with multiple processes	It is used to control access to a resource that has multiple instances

↓

for mutual exclusion

Critical Section Solution

$S = 1$

P0

```
while(True)
{
  wait(S)
      C.S.
   signal(s)
}
```

P1

```
while(True)
{
  wait(S)
      C.S.
   signal(s)
}
```

```
wait(s)
{
while (s<=0);
s--;
}
```

Characteristics of Semaphores

- Used to provide mutual exclusion → binary
- Used to control access to resources → counting
- Solution using semaphore can lead to have deadlock
- Solution using semaphore can lead to have starvation ← indefinite waiting
- Solution using semaphore can be busy waiting solutions
- Semaphores may lead to a priority inversion
- Semaphores are machine-independent

Question 1

Counting

Consider a semaphore S, initialized with value 10. What should be the value of S after executing 6 times P() and 8 time V() function on S?

S = 10
- 6
4
+ 8
12

P() → wait()

signal() & V()

S = 10
− 6
4
+ 8
12
=

Question 2

Ans:- B, C, D

counting

Consider a semaphore S, initialized with value 27. Which of the following options gives the final value of S=12?

(A) Execution of 12 P() and 15 V() $27 - 12 + 15 = 30$

(B) Execution of 15 P() $27 - 15 = 12$

(C) Execution of 23 P() and 8 V() $27 - 23 + 8 = 12$

(D) Execution of 21 P() and 6 V() $27 - 21 + 6 = 12$

Question 3

Consider a counting semaphore S, initialized with value 1. Consider 10 processes P1, P2 …. P10. All processes have same code as given below but, one process P10 has signal(S) in place of wait(S). If all processes to be executed only once, then maximum number of processes which can be in critical section together ?

```
while(True)
{
  wait(S)
     C.S.
  signal(s)
}
```

P1 P2 P3 P4 P5 P6 P7 P8 P9 P10

P10
while(True)
}
signal(S)
CS
signal(S)
}

CS → P1 P2 P3

S = X X X X 0

Solution

S = ~~1~~ 2 3

→ Run s10 first

→ any 3 processes can enter into c.s..

P1, P2, ..., P9

```
while(True)
{
  wait(S)
      C.S.
   signal(s)
}
```

P10

```
while(True)
{
  signal(S)
      C.S.
   signal(s)
}
```

S = 1 → 0 → 1

Run P1 & make it into CS

Run P10 & S = 1 P10 in CS

Run any of P2 to P9 → CS.

unacademy

Operating System:
Questions on Semaphores

By: Vishvadeep Gothi

Question 1

S = 1

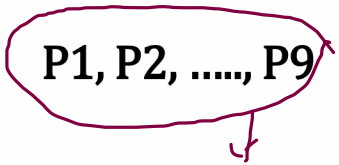
Counting

Consider a semaphore S, initialized with value 1. Consider 10 processes P1, P2 …. P10. All processes have same code as given below but, one process P10 has signal(S) in place of wait(S). If all processes can execute multiple times, then maximum number of processes which can be in critical section together ?

```
while(True)
{
  wait(S)
      C.S.
  signal(s)
}
```

Ans = 10

Solution 1




```
while(True)
{
wait(S)
    C.S.
 signal(s)
}
```

P10

```
while(True)
{
signal(S)
    C.S.
 signal(s)
}
```

S = 1

run P10 ⇒ S = 3

run P10 ⇒ S = 5

—11— ⇒ S = 7

—11— ⇒ S = 9

P10 in CS ⇒ S = 10

all P1 to P9 in C.S.

Question 2

Counting

Consider a semaphore S, initialized with value 1. Consider 10 processes P1, P2 …. P10. All processes have same code as given below but, one process P10 has signal(S) and wait(S) swapped. If all processes can execute only one time, then maximum number of processes which can be in critical section together ?

```
while(True)
{
  wait(S)
      C.S.
  signal(s)
}
```

Ans = 3

Solution 2

S = 12

run P10 & P1 in CS S = 2

run any 2 processes in C.S.

P1, P2, ..., P9

```
while(True)
{
  wait(S)
      C.S.
  signal(s)
}
```

P10

```
while(True)
{
  signal(S)
      C.S.
  signal(S)  wait(S)
}
```

Question 3

P() => wait
V() => signal

Given below is a program which when executed spawns two concurrent processes:

Semaphore X:=0;

/ Process now forks into concurrent processes P1 & P2 */*

P1 : repeat forever	*P2:repeat forever*
V(X);	*P(X);*
Compute;	*Compute;*
P(X);	*V(X);*

Consider the following statements about processes P1 and P2:

I. It is possible for process P1 to starve.

II. It is possible for process P2 to starve.

Question 4

Ans = 4

W X → 2 +
Y Z → 3 -

A shared variable x, initialized to zero, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the process W and X reads x from memory, increments by 2, stores it to memory and then terminates. Each of the processes Y and Z reads x from memory , decrements by 3, stores it to memory and then terminates. Each processes before reading x invokes the P operation (i.e., wait) on a counting semaphore S and invokes *the* V operation (i.e., signal) on the semaphore S after storing x to memory. Semaphore S is initialized to two. What is the maximum possible value of x after all processes complete execution?

x = 0 2 4 1 - 2

Solution

$x = 6 - 3 + 4$

$5 = 2 \times 0 + 2 \times 0 \times 2$

W

P(S)

R1 = x = 0

2 R1 = R1 + 2

x = R1

V(S)

X

P(S)

2 R2 = x

4 R2 = R2 + 2

x = R2

V(S)

Y

P(S)

R3 = x 0

−3 R3 = R3 − 3

x = R3

V(S)

Z

P(S)

2 R4 = x

−1 R4 = R4 − 3

x = R4

V(S)

W & Y run Concurrently, J writes on x first ⇒ x = 2

X & Z —11— , 3 writes on x first ⇒ x = 4 ← Ans

unacademy

Question 5

w & y run concurrently & w writes first ⇒ x = −3
x & z ——— // ——— & x writes first ⇒ x = −6

A shared variable x, initialized to zero, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the process W and X reads x from memory, increments by 2, stores it to memory and then terminates. Each of the processes Y and Z reads x from memory , decrements by 3, stores it to memory and then terminates. Each processes before reading x invokes the P operation (i.e., wait) on a counting semaphore S and invokes *the* V operation (i.e., signal) on the semaphore S after storing x to memory. Semaphore S is initialized to two. What is the minimum possible value of x after all processes complete execution?

Ans = −6

Question 6

Ans = 8

A shared variable *x*, initialized to zero, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the process W and X reads *x* from memory, increments by 2, stores it to memory and then terminates. Each of the processes Y and Z reads *x* from memory , decrements by 3, stores it to memory and then terminates. Each processes before reading *x* invokes the P operation (i.e., wait) on a counting semaphore S and invokes *the* V operation (i.e., signal) on the semaphore S after storing *x* to memory. Semaphore S is initialized to two. What is the total number of possible values of *x* after all processes complete execution?

$S = 2$

$x = p - 3$

W

P(s)

$x = x + 2$

V(s)

X

P(s)

$x = x + 2$

V(s)

Y

P(s)

$x = x - 3$

V(s)

Z

P(s)

$x = x - 3$

V(s)

Possible answers

$= -6$

$= 4$

$= -4$

$= -1$

$= 1$

$= -2$

$= 2$

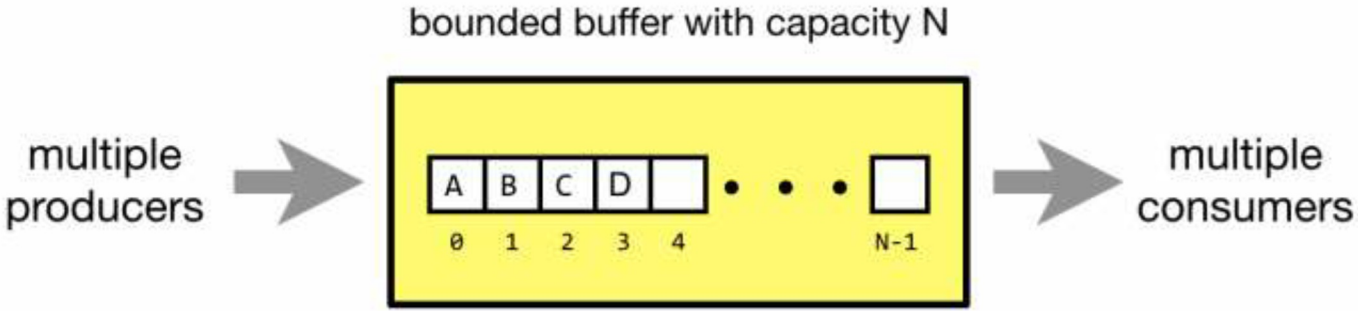
$= -3$

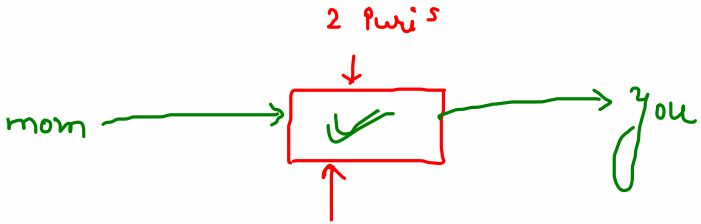
unacademy

Operating System:
Bounded-Buffer Problem

By: Vishvadeep Gothi

Bounded Buffer Problem





unacademy

Bounded Buffer Problem

- Known as producer-consumer problem also
- Buffer is the shared resource between producers and consumers

Bounded Buffer Problem: Solution

- Producers must block if the buffer is full
- Consumers must block if the buffer is empty

Bounded Buffer Problem: Solution

- Variables:
  - Mutex: Binary Semaphore to take lock on buffer (Mutual Exclusion)
  - Full: Counting Semaphore to denote the number of occupied slots in buffer
  - Empty: Counting Semaphore to denote the number of empty slots in buffer

- Initialization :- (Initially the buffer is empty)

  Mutex = 1

  Full = 0

  Empty = n

Producer()

```
Producer()
{
    wait(empty)        // to check if empty space available
    // produce an item
    wait(mutex)
    // add item on buffer
    signal(mutex)
    signal(full)
}
```

Consumer()

```
Consumer()
{
    wait(full)
    wait(mutex)
    // remove an item from buffer
    Signal(mutex)
    // Consume the item
    Signal(empty)
}
```

unacademy

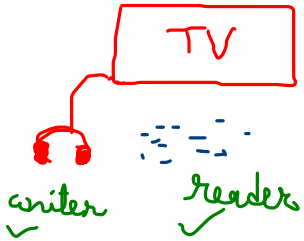
Operating System:
Reader-Writer Problem

By: Vishvadeep Gothi

Reader-Writer Problem

Consider a situation where we have a file shared between many people:

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her
- However, if some person is reading the file, then others may read it at the same time



Reader-Writer Problem: Solution

- If writer is accessing the file, then all other readers and writers will be blocked
- If any reader is reading, then other readers can read but writer will be blocked

Reader-Writer Problem: Solution

- Variables:
  - mutex: Binary Semaphore to provide Mutual Exclusion
  - wrt: Binary Semaphore to restrict readers and writers if writing is going on
  - readcount: Integer variable, denotes number of active readers
- Initialization:
  - mutex: 1
  - wrt: 1
  - readcount: 0

Writer() Process

```
writer()
{
    wait(wrt)
        // perform writing
    signal(wrt)
}
```

Reader() Process

```
Reader()
{
  wait(mutex)
  readcount++;
  if(readcount==1)
      wait(wrt)
  signal(mutex)
     //perform reading
  wait(mutex)
  readcount--;
  if(readcount==0)
      signal(wrt)
  signal(mutex)
}
```

→ a writer is writing

wrt = 0

→ a reader comes

mutex = 0 rc = 1

stuck at wait(wrt)

→ second reader comes

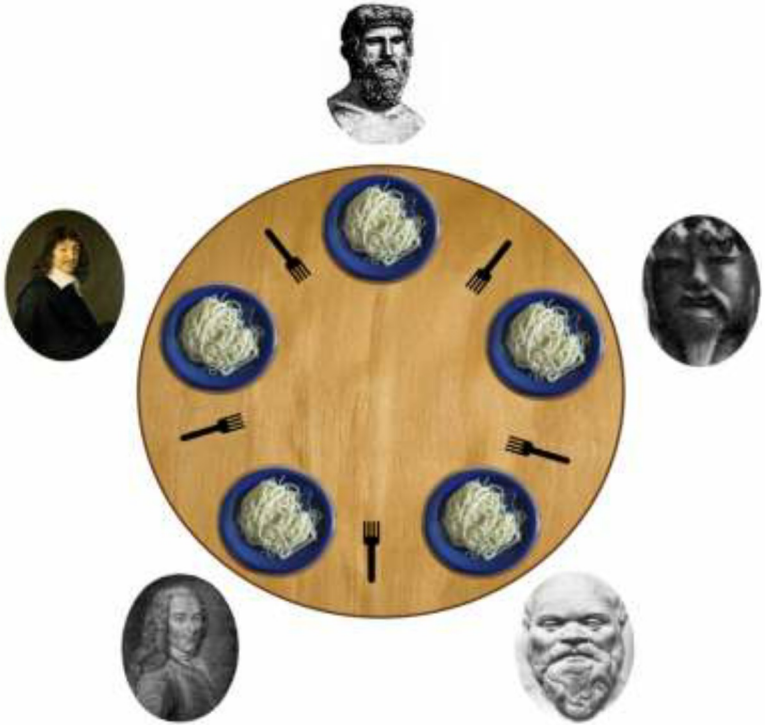
Stuck at mutex

unacademy

Operating System:
Dining-Philosopher Problem

By: Vishvadeep Gothi

Dining Philosopher Problem



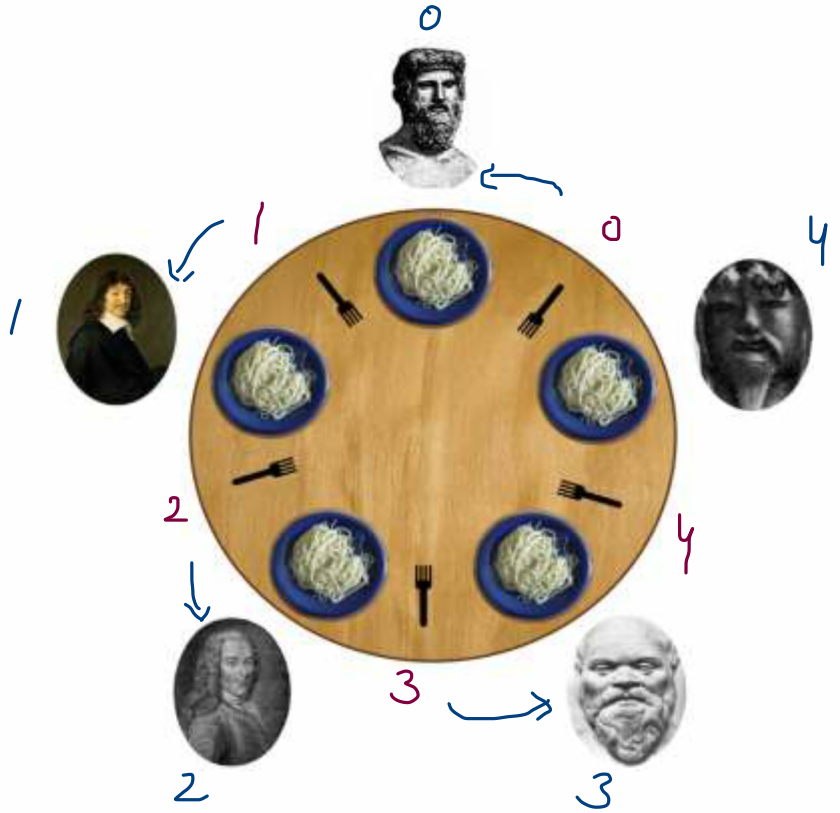
unacademy

Dining Philosopher Problem

- K philosophers seated around a circular table
- There is one chopstick between each philosopher
- A philosopher may eat if he can pick up the two chopsticks adjacent to him
- One chopstick may be picked up by any one of its adjacent followers but not both

Dining Philosopher Problem: Solution

Chopstick [5]



unacademy

Dining Philosopher Problem: Solution

Philosopher ⇒ i

```
pick (chopstick [i])
pick (chopstick [(i+1) % k])

//eat

release (chopstick [i])
release (chopstick [(i+1) mod 5])
```

Solution using Semaphore :-

Binary Semaphore array chopstick[5] = {1, 1, 1, 1, 1}

```
{
  wait(chopstick[i])
  wait(chopstick[(i+1) % k])

    //eat

  signal(chopstick[i])
  signal(chopstick[(i+1) % k])
}
```

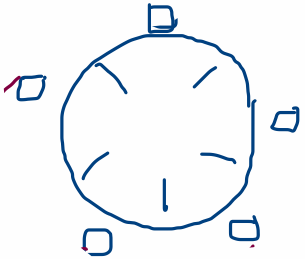
Dining Philosopher Problem: Solution

- This solution can lead to a deadlock

Dining Philosopher Problem: Solution

Some of the ways to avoid deadlock are as follows −

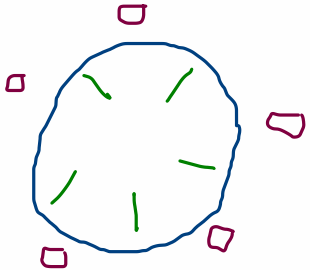
1. There should be at most (k−1) philosophers on the table



Dining Philosopher Problem: Solution

Some of the ways to avoid deadlock are as follows −

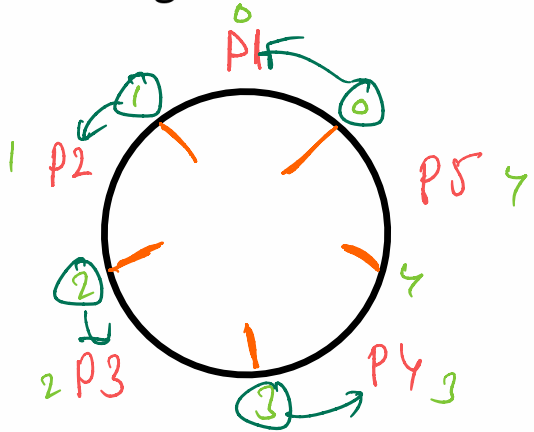
1. There should be at most (k−1) philosophers on the table
2. A philosopher should only be allowed to pick their chopstick if both are available at the same time

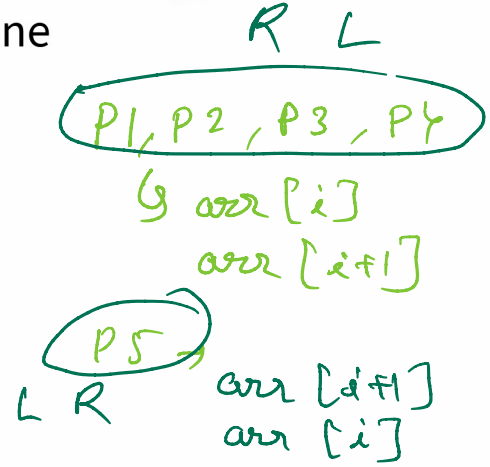


Dining Philosopher Problem: Solution

Some of the ways to avoid deadlock are as follows –

1. There should be at most (k–1) philosophers on the table
2. A philosopher should only be allowed to pick their chopstick if both are available at the same time
3. One philosopher should pick the left chopstick first and then right chopstick next; while all others will pick the right one first then left one





unacademy

Dining Philosopher Problem: Solution

all will run

↓

```
wait (chopstick[i])
wait (chopstick[(i+1) % k])
//eat
signal (chopstick[i])
signal (chopstick[(i+1) % k])
```

1 philosopher will run

```
wait (chopstick[(i+1) % k])
wait (chopstick[i])
//eat
signal (chopstick[(i+1) % k])
signal (chopstick[i])
```

unacademy

Operating System:
Deadlock

By: Vishvadeep Gothi

Operations on Resources

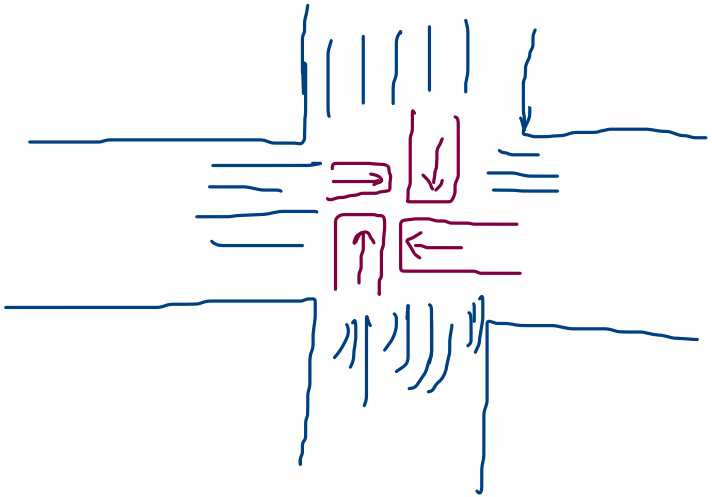
→ H/ω , S/ω

3 operations on resources:

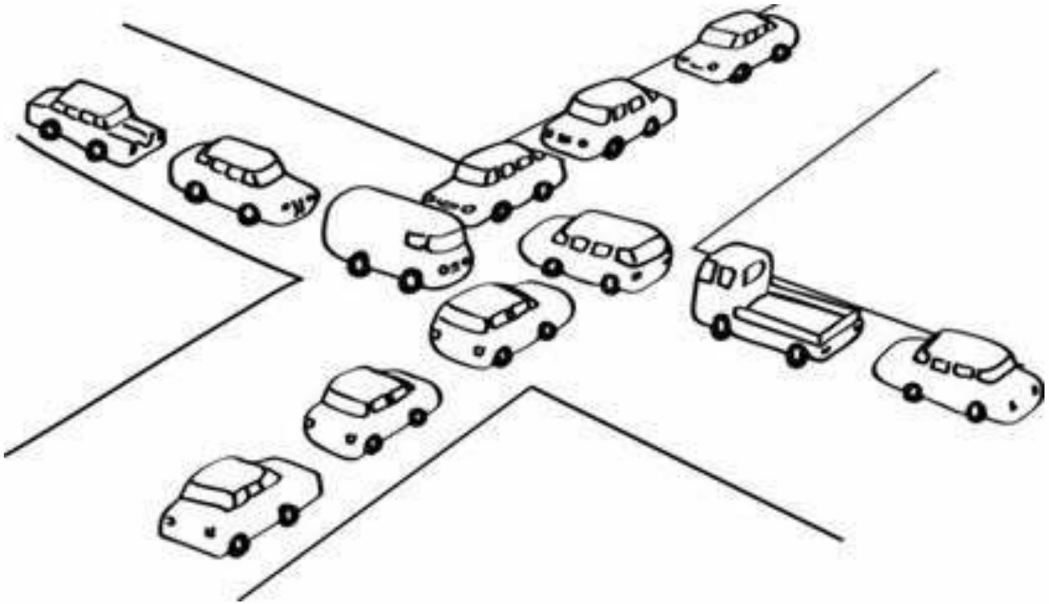
1. Request :- process requests for a resource to OS.
2. Use :- when OS allocates the resource to process, then the process can use it.
3. Release :- when use is completed then process releases the resource.

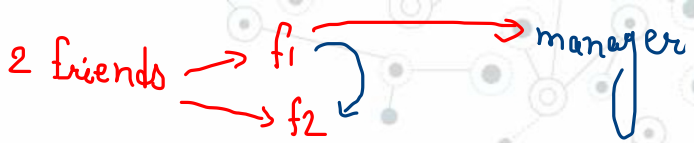
Deadlock

If two or more processes are waiting for such an event which is never going to occur.



Deadlock





unacademy

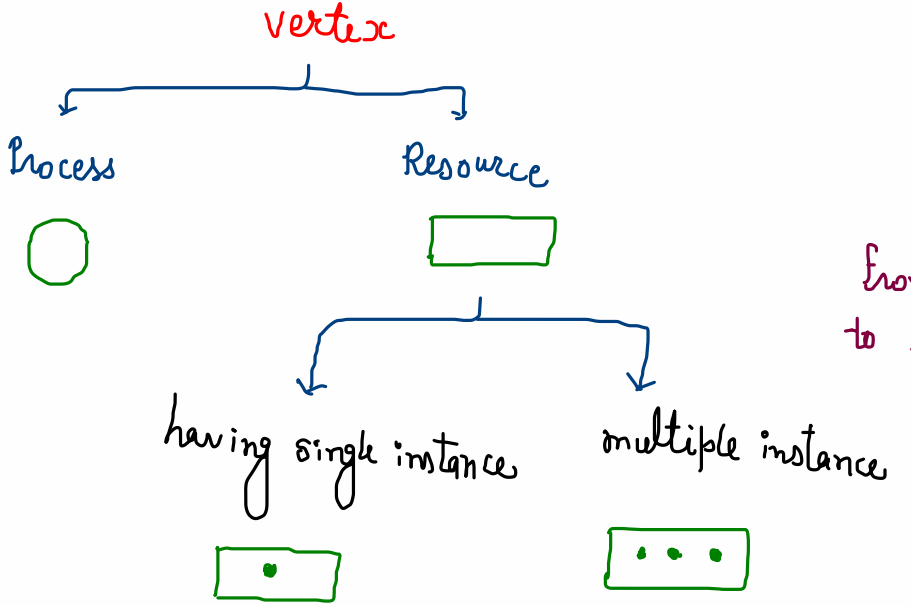
Necessary Conditions for Deadlock

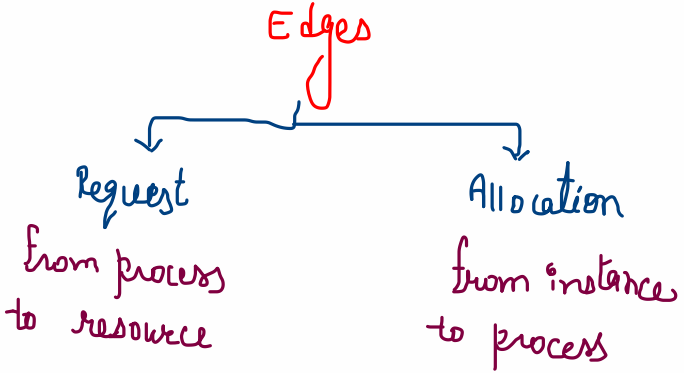
Deadlock can occur only when all following conditions are satisfied:

1. Mutual Exclusion :- if one process using a resource, then other process can not use it.
2. Hold & Wait :- each deadlocked process should hold atleast one resource & should wait for atleast one resource.
3. No-preemption :- no any forceful preemption of resources.
4. Circular Wait

Processes	Holds	Waits
P1	keyboard	Harddisk
P2	Harddisk	printer
P3	Printer	keyboard

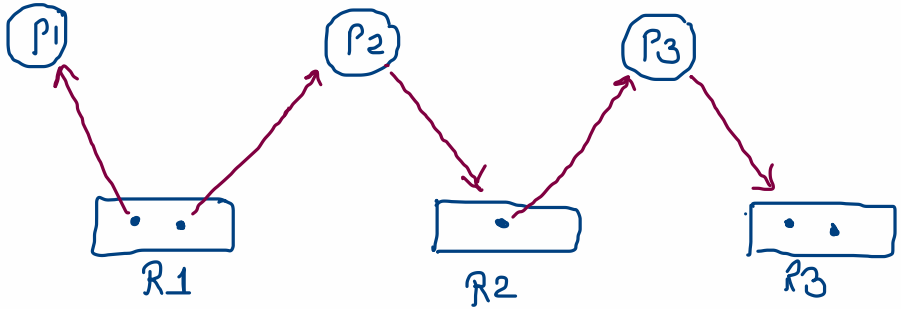
Resource Allocation Graph (directed)





Resource Allocation Graph

Ex:-



Recovery From Deadlock

1. Make Sure that deadlock never occur
   - Prevent the system from deadlock or avoid deadlock
2. Allow deadlock, detect and recover
3. Pretend that there is no any deadlock

unacademy

Operating System:
Deadlock Prevention

By: Vishvadeep Gothi

Recovery From Deadlock

1. Make Sure that deadlock never occur
   - Prevent the system from deadlock or avoid deadlock
2. Allow deadlock, detect and recover
3. Pretend that there is no any deadlock

This is resolved by Application programmer.
It is also called Ostrich Algorithm.

Deadlock Prevention

Prevent any of four necessary conditions to occur

1. Mutual Exclusion
2. Hold & Wait
3. No Preemption
4. Circular Wait

Preventing Mutual Exclusion

- Have enough resources to provide simultaneous execution
- Make all processes independent

required resources will be so many

↳ practically impossible

Preventing Hold & Wait

→ processes should either hold or only wait

Preventing Hold & Wait

- A process will either hold or wait but not together

1. If all resources are available then acquire or else just wait for all. (process may suffer from starvation.)
2. If process is trying to acquire a resource which is not available, while holding some resources; then process will release the allocated resources.
3. If a process holds resources but not using them, then there will be poor utilizatn of resources.

Preventing No Preemption

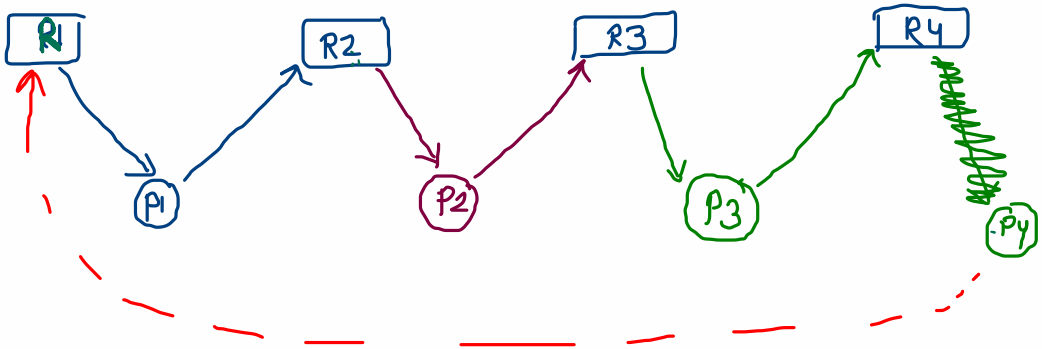
P1 requests for R1, & R1 is held by P2. P2 is also in wait for other process/resource.

OS may preempt R1 from P2 and will give it to P1.

Preventing No Preemption

- Preempt one or more resources from processes

Preventing Circular Wait



Preventing Circular Wait

1. All resources have been given sequence numbers (unique) $R_1, R_2, R_3, \ldots\ldots$
2. Any process while holding a resource R_i, can request for R_j only when $j > i$.
3. If a process is holding a resource R_i, and wants another resource R_j, when $j < i$. Then process will have to release R_i and will have to acquire R_j first.

unacademy

Operating System:
Deadlock Avoidance & Banker's Safety Algorithm

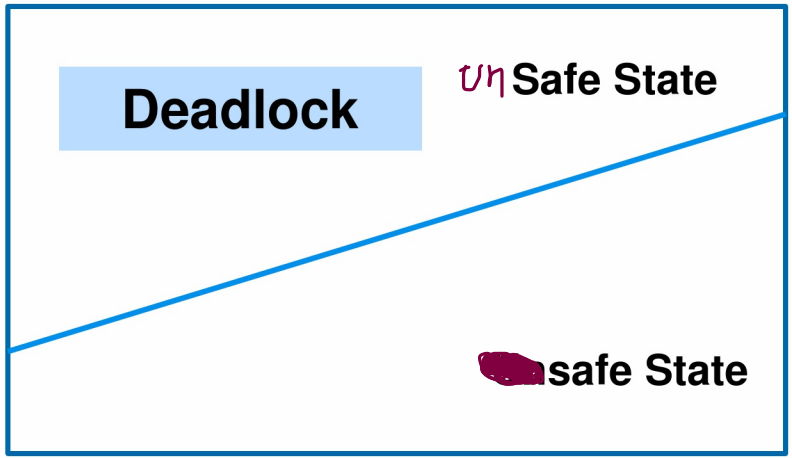
By: Vishvadeep Gothi

Recovery From Deadlock

1. Make Sure that deadlock never occur
   - Prevent the system from deadlock or avoid deadlock
2. Allow deadlock, detect and recover
3. Pretend that there is no any deadlock

Deadlock Avoidance

In deadlock avoidance, the OS tries to keep system in safe state



unacademy

Deadlock Avoidance

In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system.

Banker's Algorithm

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety

Banker's Algorithm

Process	Allocation	Max	Available
P1	1	3	~~1 4 5~~ to 12
P2	5	8	
P3	3	4	
P4	2	7	
	11		

Need = max - allocation

2

3

1

5

< P3, P1, P2, P4 > Safe sequence

System has total ⇒ 12 resources

1. Execute P3.
2. — " — P1
3. — " — P2
4. — " — P4

all processes can execute, hence ⇓ Safe state

Banker's Algorithm

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	~~3 3 2~~
P_1	2 0 0	3 2 2	~~5 3 2~~
P_2	3 0 2	9 0 2	~~7 4 3~~
P_3	2 1 1	2 2 2	~~7 5 3~~
P_4	0 0 2	4 3 3	~~10 5 5~~
	7 2 5		10 5 7

Need

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Safe sequence:-

$\langle P_1, P_3, P_0, P_2, P_4 \rangle$

unacademy

Banker's Algorithm

	Available A	B	C
	3	3	2
After P1	5	3	2
After P3	7	4	3
After P0	7	5	3
After P2	10	5	5
After P4	10	5	7

Banker's Algorithm

We have n number of processes & m number of resources.

1. Allocation: matrix of size $n \times m$
2. Max: —"— $n \times m$
3. Need: —"— $n \times m$
4. Available: array of size m
5. Finish :- —"— n Finish[0] ⋮

Banker's Algorithm

1. Let Work and Finish be vectors of length 'm' and 'n' respectively.
   Initialize: Work = Available
   Finish[i] = false; for i=1, 2, 3, 4….n
2. Find an i such that both
   (a) Finish[i] = false
   (b) Needi <= Work
   if no such i exists goto step (4)
3. Work = Work + Allocation[i]
   Finish[i] = true
   goto step (2)
4. if Finish [i] = true for all I
   then the system is in a safe state

Question (Homework)

Process	Allocation A B C D	Max A B C D	Available A B C D
P1	0 0 1 2	0 0 1 2	1 5 2 0
P2	1 0 0 0	1 7 5 0	1 5 3 2
P3	1 3 5 4	2 3 5 6	2 8 8 6
P4	0 6 3 4	0 6 5 2	2 14 11 10
P5	0 0 1 4	0 6 5 6	2 14 12 14
			3 14 12 14

Need

A B C D

0 0 0 0

0 7 5 0

1 0 0 2

0 0 2 0

0 6 4 2

$P_1 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5 \rightarrow P_2$

unacademy

unacademy

Operating System:

Banker's Algorithm
Resource Request Algorithm

By: Vishvadeep Gothi

Banker's Algorithm

Max - Allocation

Process	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	~~3 3 2~~ 2 3 0	7 4 3
P_1	~~3~~ 2 ~~0~~ 0 ~~2~~	3 2 2		~~1 2 2~~ 0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

5 3 2
7 4 3
7 5 3
10 5 5
10 5 7

<P1, P3, P0, P2, P4>

P1 → <1 0 2>

unacademy

Banker's Algorithm

1. Let Work and Finish be vectors of length 'm' and 'n' respectively.
   Initialize: Work = Available
   Finish[i] = false; for i=1, 2, 3, 4….n
2. Find an i such that both
   (a) Finish[i] = false
   (b) Needi <= Work
   if no such i exists goto step (4)
3. Work = Work + Allocation[i]
   Finish[i] = true
   goto step (2)
4. if Finish [i] = true for all I
   then the system is in a safe state

Resource Request Algorithm

Question

→ yes request can be granted.

What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?

$Request_{P_1} = \langle 1, 0, 2 \rangle$ → valid → available enough resources

1. Request is valid or not? $Request_i \leq Need_i$
2. Enough resources available? $Request_i \leq Available$
3. Allocate & check safety.
   → yes safe

Banker's Algorithm

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Allocate <1, 0, 2> to process P1.

1. $Allocation_i = Allocation_i + Request_i$
2. $Need_i = Need_i - Request_i$
3. $Available = Available - Request_i$

Question

$Request_0 = \langle 1, 0, 2 \rangle$ → valid → enough resources → allocate

What will happen if process P0 requests one additional instance of resource type A and two instances of resource type C?

	Need		
	A	B	C
P_0	~~7~~ 6	~~4~~ 4	~~3~~ 1
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

allocation: 1 1 2

available: ~~3 3 2~~ 2 3 0

System is in unsafe state

Request will not be granted.

Question

$Request_3 = \langle 0, 1, 0 \rangle$ → valid → enough resources → allocate

What will happen if process P3 requests one additional instance of resource type B?

	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	~~0~~ 0	~~1~~ 0	~~1~~ 1
P_4	4	3	1

Available

~~3 3 2~~

3 2 2

Allocation

P3	~~2~~ 2 ~~1~~ 1

System is in safe state

Request can be granted.

$\langle P_1, P_3, P_0, P_2, P_4 \rangle$

Resource Request Algorithm

1. *If $Request_i$ <= $Need_i$*
   Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. *If $Request_i$ <= Available*
   Goto step (3); otherwise, P_i must wait, since the resources are not available.
3. *Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:*

   Available = Available – Requesti
   $Allocation_i$ = $Allocation_i$ + $Request_i$
   $Need_i$ = $Need_i$– $Request_i$

4. Run safety algo, if safe state then request granted otherwise denied.

unacademy

Operating System:

Deadlock Detection & Recovery

By: Vishvadeep Gothi

Recovery From Deadlock

1. Make Sure that deadlock never occur
   - Prevent the system from deadlock or avoid deadlock
2. Allow deadlock, detect and recover
3. Pretend that there is no any deadlock

Deadlock Detection

1. When all resources have single instance → wait-for-graph
2. When resources have multiple instances
   ↳ detectn algo

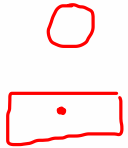
Deadlock Detection

When all resources have single instance:

Deadlock detection is done using wait-for-graph

Wait For Graph

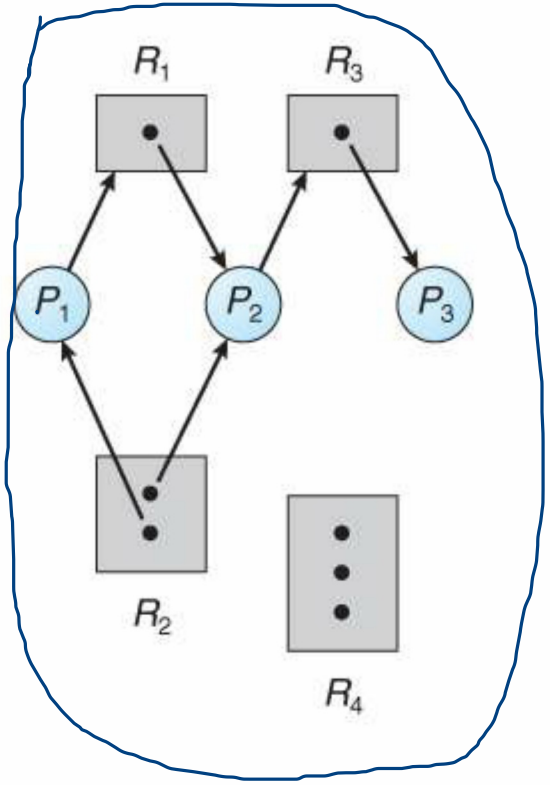
It is created from resource allocation graph



Req. edge process to resource

Allocat^n edge instance to process

Wait For Graph



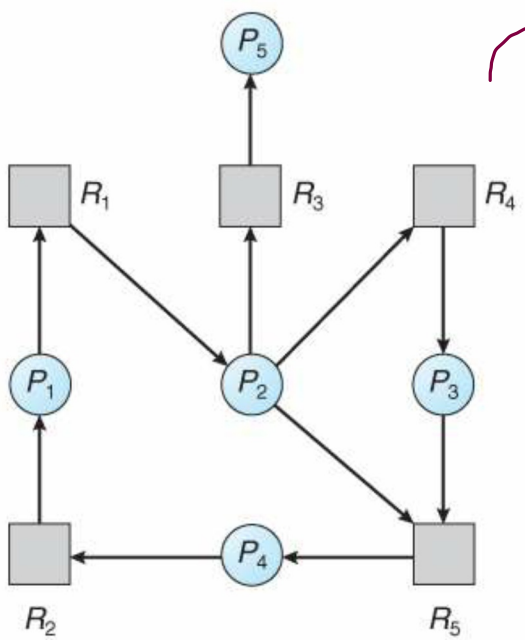
unacademy

Wait For Graph

→ vertices ⇒ only process

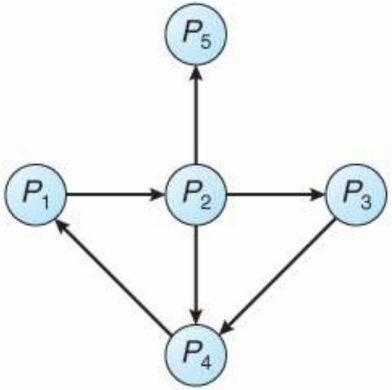
edge:- from p_i to p_j

process p_i is waiting for a resource which is held by p_j.



(a)

resource allocation graph



(b)

wait-for-graph

deadlock

Wait for graph :-

If a cycle exists then $\Rightarrow$ **deadlock**

Processes deadlocked $\Rightarrow$ P_1, P_2, P_3, P_4

$\Downarrow$

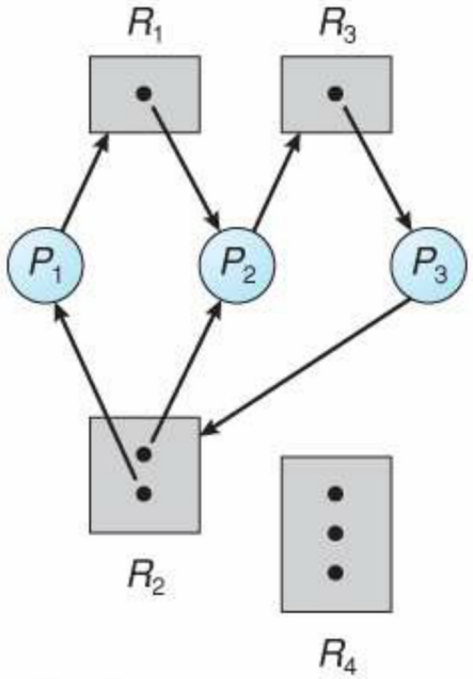
part of cycle.

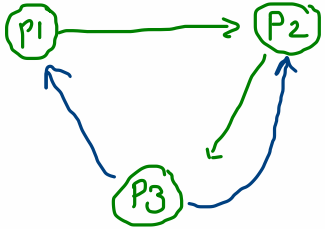
P_5 can completely execute.

Wait For Graph

If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the possibility of a deadlock, but does not guarantee one.

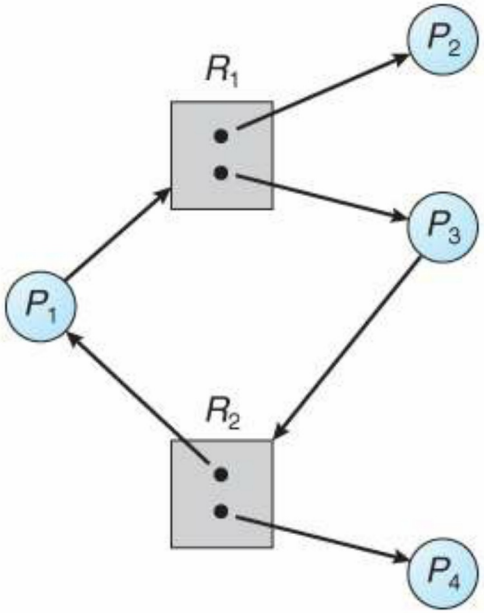
Wait For Graph: Example

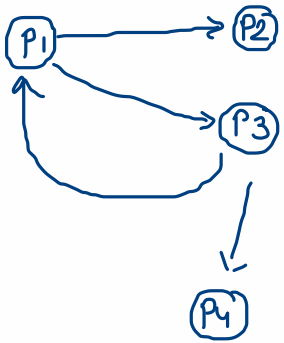




deadlock => yes

Wait For Graph: Example





cycle ⇒ yes

deadlock ⇒ no

unacademy

Deadlock Detection

When resources have multiple instance:

Deadlock detection is done using a specific algorithm

Deadlock Detection Algorithm

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

	available		
	A	B	C
after P_0	0	1	0
after P_2	3	1	3
after P_1	5	1	3
after P_3	7	2	4
after P_4	7	2	6

If all processes can execute ⇒ no deadlock.

unacademy

Deadlock Detection Algorithm

1. Let *Work* and *Finish* be vectors of length m and n respectively.
   Initialize *Work= Available*. For *i=0, 1, …., n-1*, if $Request_i$ = 0, then *Finish[i]* = true; otherwise, *Finish[i]*= false.
2. Find an index i such that both
   a) *Finish[i] == false*
   b) $Request_i$ <= *Work*
   If no such *i* exists go to step 4.
3. *Work= Work+* $Allocation_i$
   Finish[i]= true
   Go to Step 2.
4. If *Finish[i]== false* for some i, 0<=i<n,
   then the system is in a deadlocked state.
   Moreover, if *Finish[i]==false* the process P_i is deadlocked.

Detection-Algorithm Usage

When should the deadlock detection be done? Frequently, or infrequently?

Detection-Algorithm Usage

1. Do deadlock detection after every resource allocation
2. Do deadlock detection only when there is some clue

↳ if CPU performance goes on decreasing.

Recovery From Deadlock

There are three basic approaches to recovery from deadlock:

1. Inform the system operator and allow him/her to take manual intervention
2. Terminate one or more processes involved in the deadlock
3. Preempt resources.

Process Termination

1. Terminate all processes involved in the deadlock
2. Terminate processes one by one until the deadlock is broken

Process Termination

Many factors that can go into deciding which processes to terminate next:

1. Process priorities.
2. How long the process has been running, and how close it is to finishing.
3. How many and what type of resources is the process holding
4. How many more resources does the process need to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

Resource Preemption

Important issues to be addressed when preempting resources to relieve deadlock:

1. Selecting a victim → selecting those process or resource, that can break deadlock cycle when pre-empted.
2. Rollback → at what pt. process need to roll back so that it is stopped at a steady pt.
3. Starvation → If a process is victim for very very long time, it might be possible, that process may starve for very very long time.