



Operating System Revision: Part 3

Special class

Operating System: Process Synchronization

By: Vishvadeep Gothi

▲ 1 • Asked by Vaishnavij...

Please help me with this doubt

5.15.40 Process Scheduling: GATE IT 2006 | Question: 12 top

https://gateoverflow.in/38

In the working-set strategy, which of the following is done by the operating system to prevent thrashing?

- I. It initiates another process if there are enough extra frames.
 - II. It selects a process to suspend if the sum of the sizes of the working-sets exceeds the total number of available frames.
- A. I only
B. II only
C. Neither I nor II
D. Both I and II

Types of Process

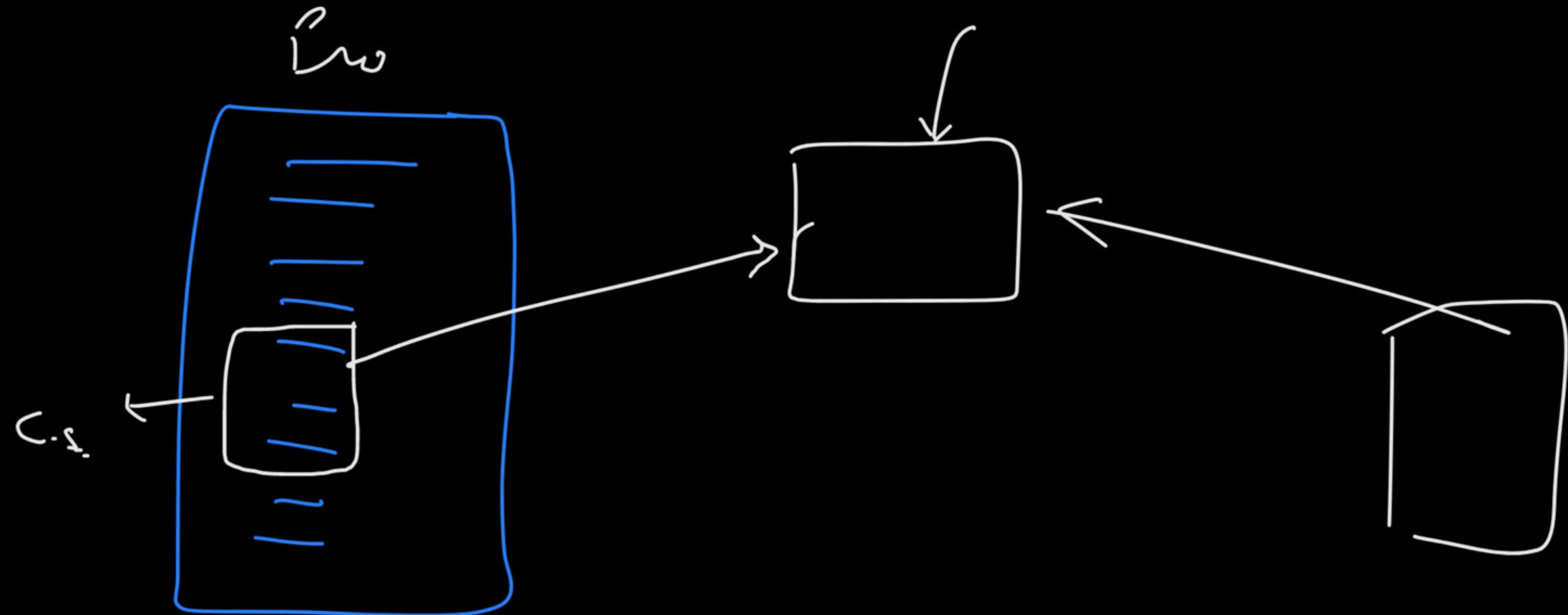
1. Independent
2. Cooperating/Coordinating/Communicating

Lack of Synchronization

Problems without Synchronization:

- Inconsistency
- Loss of Data
- Deadlock

Critical Section



Race Condition

Solution for Critical Section Problem

Solution for Critical Section Problem

Requirements of Critical Section problem solution:

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

Solution 1

Boolean lock=false;

while(true)

{

while(lock);

lock=true;

CS

lock=false;

RS;

}

X M. E.

✓ Progress

X B. W.

Solution 2

```
int turn=0;  
while(true) P0  
{  
    while(turn!=0);  
    CS  
    turn=1;  
    RS;  
}
```

strict alternation

```
P1 while(true)  
{  
    while(turn!=1);  
    CS  
    turn=0;  
    RS;  
}
```

✓ Η.Ε.
✗ Progress
✓ Η.Ω.

Peterson's Solution

```

Boolean Flag[2] = (False, False);
int turn;
    i
while(true) {
    Flag[i]=true;
    turn=j;
    while(Flag[j] && turn==j);
        CS
    Flag[i]=False;
        RS;
}

```

j

```

while(true){
    Flag[j]=true;
    turn=i;
    while(Flag[i] && turn==i);
        CS
    Flag[j]=False;
        RS;
}

```

\checkmark T.E.
 \checkmark Progress

\checkmark B.W.

$$F(i) = \cancel{FT} \cancel{F_+}$$

$$F(j) = \cancel{FT}$$

turn = j \neq j

Synchronization Hardware

1. TestAndSet()

2. Swap()

instⁿs in Comp. (CPU)



TestAndSet()

Returns the current value flag and sets it to true.

TestAndSet()

```
Boolean Lock=False;  
while(true)  
{  
    boolean TestAndSet(Boolean *trg){  
        boolean rv = *trg;  
        *trg = True;  
        CS  
        Lock=False;  
        return rv;  
    }  
}
```

Umesh.
Kroj.
D.W.

local for each process Swap()

✓ global & shared among all processes

Boolean Key, Lock=False;

~~X X F T~~

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

}

P₁

key = X
F

P₂

key = T
F

Semaphore

Binary $\Rightarrow 0, 1$

Counting $\Rightarrow 0, 1, 2, 3, \dots$

Atomic

wait(S)

{

while(S<=0);

S--;

}

Signal(S)

{

S++;

}

P |

S = 6

wait(s)

Critical Section Solution using Semaphore

Binary Semaphore $s = 1$

wait(s)

c.s.

signal(s)

Question GATE-2005

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

Semaphore X:=0; ~~X~~

/ 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 GATE-2004

- Consider two processes P1 and P2 accessing the shared variables X and Y protected by two binary semaphores S_x and S_y respectively, both initialized to 1. P and V denote the usual semaphore operators, where P decrements the semaphore value and V increments the semaphore value. The pseudo-code of P1 and P2 is as follows:

P1:

```
while true do {
```

```
L1: .....
```

```
L2: .....
```

```
X = X + 1;
```

```
Y = Y - 1;
```

```
V(Sx);
```

```
V(Sy); }
```

P2:

```
while true do {
```

```
L3: .....
```

```
L4: .....
```

```
Y = Y + 1;
```

```
X = Y - 1;
```

```
V(Sy);
```

```
V(Sx); }
```

P1

P(S_x)

P(S_y)

P2

P(S_x)

P(S_y)

In order to avoid deadlock the correct operators at L_1 , L_2 , L_3 and L_4 are respectively

- (A) P(S_y), P(S_x); P(S_x), P(S_y)
 (B) P(S_x), P(S_y); P(S_y), P(S_x)

- (C) P(S_x), P(S_x); P(S_y), P(S_y)
 ✓(D) P(S_x), P(S_y); P(S_x), P(S_y)

Question GATE-2013

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 one, stores it to memory and then terminates. Each of the processes Y and Z reads x from memory , decrements by two, 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?

(A) -2

(B) -1

(C) 1

(D) 2

W X Y Z

$$x = x + 1 \quad x = x + 1 \quad x = x - 2 \quad x = x - 2$$

$x = \cancel{2} \leftarrow 2$

Busy Waiting (Spin-lock)

Solutions Without Busy Waiting

```
wait(Semaphore s){  
    s=s-1;  
    if (s<0) {  
        // add process to queue  
        block();  
    }  
}  
  
signal(Semaphore s){  
    s=s+1;  
    if (s<=0) {  
        // remove process p from queue  
        wakeup(p);  
    }  
}
```

Classical Problems on Synchronization

1. Producer-Consumer Problem or Bounded Buffer Problem
2. Reader-Writer Problem
3. Dining Philosopher Problem

Producer – Consumer

Both share a buffer of bounded size (Size = n).

Producer can produce any item until any empty place is available in buffer.

Consumer can consume any item until there is any item in buffer.

Variables used in Algorithm:

Empty: Semaphore to denote number of empty places in buffer (Initialized to n)

Full: Semaphore to denote number of occupied places in buffer (Initialized to 0)

Mutex: Semaphore to provide mutual execution (Initialized to 1)

Producer – Consumer

Consumer:

Producer:

```
while(true)
{
    wait(Empty);
    wait(Mutex);
    //Produce an item in buffer;
    signal(Mutex);
    signal(Full);
}
```

```
while(true)
{
    wait(Full);
    wait(Mutex);
    //Consume an item from buffer;
    signal(Mutex);
    signal(Empty);}
```

Reader-Writer

Multiple readers can read

If one writer is writing then no any writer or reader can perform their task.

If reader is reading then writer can not perform its task.

Variables used in Algorithm:

Wrt: Semaphore to provide mutual exclusion on write

Mutex: Semaphore to provide mutual exclusion on shared variable

ReadCount: Semaphore to provide mutual execution (Initialized to 1)

Reader-Writer

Writer:

```
while(true)
{
    wait(Wrt);
    //Perform
    Writing;
    signal(Wrt);
}
```

Reader-Writer

Reader:

```
while(true)
{
    Wait(Mutex);
    ReadCount++;
    if(ReadCount==1)
        Wait(Wrt);
    Signal(Mutex);
    //Perform Reading
```

```
Wait(Mutex);
ReadCount--;
if(ReadCount==0)
    Signal(Wrt);
Signal(Mutex);
}
```

Dining Philosopher

- n philosophers are sitting on a round table.
- They can either eat or think.
- n chopsticks are shared among them.

Dining Philosopher

```
while(true)
{
    wait(chopstick[i]);
    wait(chopstick[ (i+1) % 5 ]);

    // Eat
    signal(chopstick[i]);
    signal(chopstick[ (i+1) % 5 ]);

    // Think
}
```

Dining Philosopher

- Given solution is causing Deadlock
- So, Deadlock free solution should be like:
 1. Allow $n - 1$ philosophers at a time to sit and eat
 2. Allow a philosopher to pick chopsticks only if both are available
 3. Use an asymmetric solution

Deadlock

Deadlock

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

Deadlock

3 operations on resources:

1. Request
2. Use
3. Release

Necessary Conditions for Deadlock

Resource Request Graph

Deadlock Recovery

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

Deadlock Prevention

Do not let occur any of the necessary conditions for deadlock

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

Deadlock Avoidance

Bankers Algorithm:

1. Safety Algorithm
2. Resource-Request Algorithm

Banker's Algorithm: Safety Algorithm

- Available
- Max
- Allocation
- Need

Banker's Algorithm: Safety 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			

Banker's Algorithm: Safety 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) $\text{Need}_i \leq \text{Work}$

if no such i exists goto step (4)

3) $\text{Work} = \text{Work} + \text{Allocation}[i]$

Finish[i] = true

goto step (2)

4) if Finish [i] = true for all i

then the system is in a safe state

Banker's Algorithm: Resource Request Algorithm

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

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			

Banker's Algorithm: Resource Request Algorithm

1) If $\text{Request}_i \leq \text{Need}_i$

Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim

2) If $\text{Request}_i \leq \text{Available}$

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:

$$\text{Available} = \text{Available} - \text{Request}_i$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i$$

Deadlock Detection

Single Instances of Resources

Multiple Instances of Resources

Deadlock Detection

Available
Allocation
Request

Deadlock Detection

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Process Termination

1. Abort all processes
2. Abort one process at a time until deadlock cycle is eliminated

Resource Preemption

Issues with resource preemption:

1. Selecting a victim
2. Rollback
3. Starvation

Question

Consider a system with 3 processes A, B and C. All 3 processes require 4 resources each to execute. The minimum number of resources the system should have such that deadlock can never occur?

Question

Consider a system with 3 processes that share 4 instances of the same resource type. Each process can request a maximum of K instances. Resource instances can be requested and released only one at a time. The largest value of K that will always avoid deadlock is ____.

Question

Consider a system with n processes with single resource R. Each process requires k instances to execute. What is the maximum instances of R to cause a deadlock?

Question

A computer has six tape drives, with n processes competing for them. Each process may need two drives. What is the maximum value of n for the system to be deadlock free?

Happy Learning.!