Synchronization Hardware

```
boolean Test_And_Set(boolean target){
    boolean rv = target;
    target = true;   // modifies the original
    return rv;       // value of target not
                     // a copy
                     (use address to modify
                      actual target)
}
```

atomic instruction (NOT a function, CPU instruction)

```
do{
    while (Test_And_Set(lock));   // Entry Section
    critical section;
    lock = false;   // Exit Section
    remainder section;
}while(1);
```

Mutual-exclusion Implementation with Test_And_Set

Another technique to overcome race-around condition — "Synchronization Hardware"

Solution to Critical Section Problem satisfies three requirements:

i) Mutual Exclusion — only one process should access critical section at a time

ii) Progress — the processes not executing remainder section decide which process enters critical section next

decision made by processes who're not in remainder section

P4 →
P1 → Entry Section
P2 → Critical Section
Exit Section
P3 → Remainder Section

iii) Bounded ~~Hardware~~ Waiting
↳ the processes should wait finitely

- If there are n processes, and 1 fails to get the CPU, it has to wait at most for (n-1) processes to execute.
- Synchronization Techniques have no control over which process enter the CPU next, so a process may end up waiting infinitely.

Common data structure used by $P_0, P_2, \ldots, P_{n-1}$ is boolean waiting [n], and ~~return~~ boolean lock;

// Code for a process P_i

```
do{
   waiting[1] = true;
   key = true;
   while (waiting[1] && key)                 } Entry Section
           key = TestAndSet (lock);
   waiting[1] = false;
      // Critical Section
   j = (i+1)% n;
   while ((j != i) && !waiting[i])
           j = (j+1)%n;                      } Exit Section
   if (j == 1)
           lock = false;
   else
        waiting[j] = false;
     // Remainder Section
} while(1);
```

* The problem with blocking wait : Deadlock and Starvation

Process P_0

```
   wait (s);
   wait (Q);
Critical section [ ...
   signal (s);
   signal (Q);
```

Process P_1

```
   wait(Q);
   wait (s);
Critical Section [ ...
   signal (Q);
   signal (s);
```

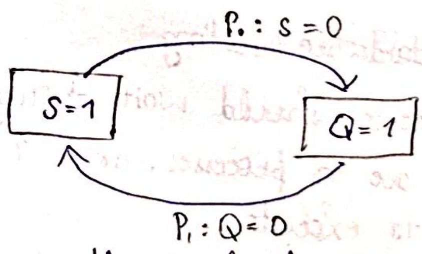
Execution Sequence when Deadlock situat

P_0 : wait (S);

P_1 : wait (Q);

P_0 : wait(Q);

P_1 : wait (s);



- The process P_0 and P_1 sharing two semaphore variables 'S' and 'Q'.
- Here, P_0 and P_1 both end up waiting for each other, unable to finish either. This situa is known as "Deadlock".

Solving Producer-Consumer Problem with bounded buffer using semaphore

```
int n;
int mutex = 1;
int empty = n;  } Counting
int full = 0;   } Semaphores
item buffer[n];
```

Producer

```
do{
    ...
    produce an item in next p;
    ...
    wait (empty);
    wait (mutex);
                 ...
    Critical   { add next P to buffer;
    Section    { ...
    signal (mutex);
    signal (full);
}while (1);
```

Consumer

```
do{
    wait (full);
    wait (mutex);
                 ...
    Critical   { remove an item from buffer
    Section    { to next pc;
                 ...
    signal (mutex);
    signal (empty);
    ...
    consume the item in next c;
}while (1);
```

Deadlock avoiding algorithm

	Maximum Need	Current Need	Need
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

Let there be 12 magnetic tape drivers are there in the system

- Out of 12, $(5+2+2) = 9$ are already in use, so we have $(12-9) = 3$ left.
- First we allocate 2 to P_1, i.e. 1 remaining
- When P_1 is done, we have $(4+1) = 5$ remaining
- Now, we allocate 5 to P_0, i.e. 0 remaining
- When P_0 is done, we have 10 remaining
- Now, we allocate 7 to P_2.
- Since there exists some order in which we can assign resources, the initial assignment is in a "safe state".

Now, consider that current need of $P_1 = 3$, i.e. Need of $P_1 = 1$ and no. of available resources = 2.

- We can still perform $P_1 \rightarrow P_0 \rightarrow P_2$, so it is still a "safe state"

Now, consider that current need of $P_2 = 3$, i.e Need of $P_2 = 6$ and no. of available resources = 2

- This is an "unsafe state" as no such sequence exists

Banker's Algorithm

- Whenever a request comes, it checks if the system is in a safe state before allocating resources (deadlock avoidance algorithm)
- It works like how a banker allocates loan — only granting loans if they are sure that they can satisfy all customers without anyone going bankrupt.

Safe State

- A state is safe if the system can allocate resources to each process (upto its max) in some order and still avoid deadlock
- A sequence of processes $\langle P_1, P_2, \ldots, P_n \rangle$ is a safe sequence for the current allocation state if, for each P_i the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all P_j with $j < i$.

Safety Algorithm (Banker's Algorithm)

1. Let Work and Finish be vectors of length m and n respectively.
   Initialize Work = available
   and, Finish[i] = false for all i = 1 to n
2. Find an i such that both:
   (a) Finish[i] = false
   (b) Need[i] $\leq$ 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] = true for all i, then the system is in safe state.

* Let m denote no. of resource types, and
n denote no. of processes

Available:

Printer	Mouse	Tabloid
3	5	6

Max:

	1	2	3	...	m
1	1	2	..	--	1
2	0	1	-	--	2
3					
⋮	⋮	⋮	⋮	⋮	⋮
n					

$n \times m$

Process	Allocation A B C	Max A B C	Available A B C	Need A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2	5 3 2	1 2 2
P_2	3 0 2	9 0 2	7 4 3	6 0 0
P_3	2 1 1	2 2 2	7 5 3	0 1 1
P_4	0 0 2	4 3 3	10 5 5 / 10 5 7	4 3 1

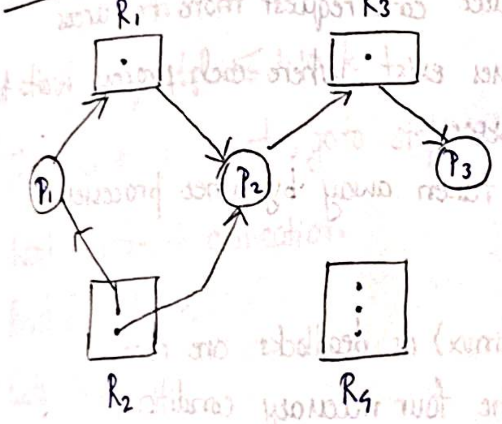
Initially Available:

A	B	C
10	5	7

need[i] = max[i] − allocation[i]

- $\langle P_1, P_3, P_0, P_2, P_4 \rangle$ is a safe sequence, hence the initial allocation state is a safe state

Deadlock

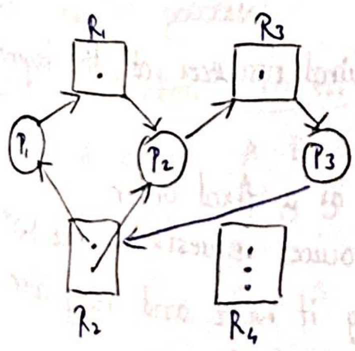


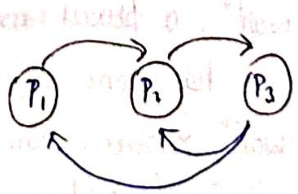
Resource Allocation Graph

Here, P_1 is waiting for P_2 to release R_1 and P_2 is waiting for P_3 to release R_3

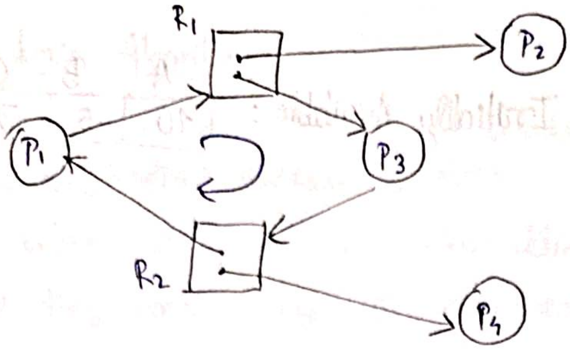
Wait-for Graph

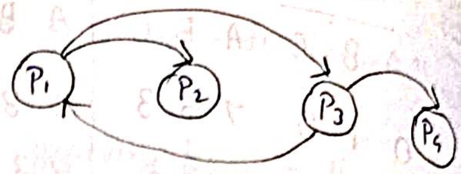
- Here, not all processes are waiting for each other, so no deadlock state.
- Now, if P_3 sends a request for R_2:





- This becomes a deadlock state.
- If a cycle is formed, and only one instance of each resource is there, it is a deadlock state.





- Here, P_1 and P_3 are NOT in a deadlock state even though there is a cycle because multiple instances of resources are available

* When multiple instances of a resource are there and a cycle is formed, it is NOT necessarily a deadlock.

* Deadlock — a situation in an OS where two or more processes are permanently blocked each waiting for a resource, the other processes are holding.

* Necessary Conditions for Deadlock:
  (i) Mutual Exclusion — at least one resource must be non-shareable
  (ii) Hold and Wait — a process holding one resource can request more resources
  (iii) Circular Wait — a circular chain of processes exist, where each process waits a resource held by next process.
  (iv) No Preemption — resources can't be forcibly taken away by other processes

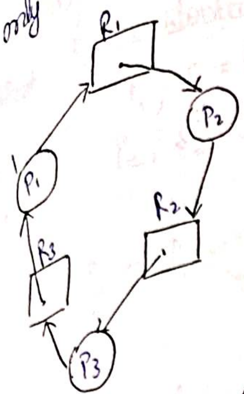
* Ways to handle Deadlock:
  (i) Ignore Deadlock — used in many OS (Windows, Linux) as deadlocks are rare
  (ii) Deadlock Prevention — prevents at least one of the four necessary conditions
  (iii) Deadlock Avoidance — uses algorithms like Banker's Algorithm to stay in safe state
  (iv) Deadlock Detection and Recovery — allows deadlock, detects it and recover (terminate/restart process)

* Solutions for Deadlock Prevention:
  (i) To prevent "mutual exclusion", make all resources shareable
  (ii) To prevent "hold and wait", a process must request all required resources at the beginning or only request when it has none allocated
  (iii) To prevent "circular wait", resources can only be requested in a fixed order.
  (iv) To prevent "no preemption", if a process holding some resources requests another NOT available, OS forces it to release its currently held resources, making it wait and retry later.

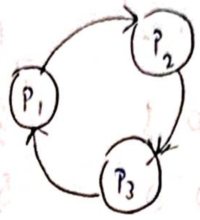
* Deadlock detection algorithm can be run when CPU utilization is below some threshold or after some fixed time interval or when some requests are made

Deadlock Detection and Recovery

If only 1 instance of each resource is there, and a cycle is formed it is a deadlock.



Resource Allocation Graph



Wait-for Graph

1. Let Work and Finish be vectors of length m and n respectively. Initialize Work = Available. for i=1 to n if $allocation_i \neq 0$, then Finish[i] = false, otherwise Finish[i] = true. ← if a process isn't holding any resource, it can't be part of deadlock

. Find an index i such that both:

ⓐ Finish [i] = false

ⓑ $Request_i \leq Work$

If no such i exists, goto step 4.

. Work = Work + allocation

Finish[i] = true

Goto Step 2

. If Finish[i] = false, for some i, $1 \leq i \leq n$, then the system is in a deadlock state.

Moreover, if Finish[i] = false, then process P_i is deadlocked

m – no. of resource types

n – no. of processes

Process	Allocation A B C	Request A B C
P_0	0 1 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

~~0 0 0~~ ← initial

~~0 1 0~~ (after P_0)

~~3 1 3~~ (after P_2)

~~5 1 3~~ (after P_1)

~~7 2 4~~ (after P_3)

7 2 6 ← final

$< P_0, P_2, P_1, P_3, P_4 >$ is a safe sequence, so it is NOT a dealock state.

Process	Allocation	~~Avail~~Request	Availble
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	~~0 0 0~~ ← initial
P_1	2 0 0	2 0 2	~~0 1 0~~ (after P_0)
P_2	3 0 3	0 0 1	~~2 1 0~~ (after killing P_1)
P_3	2 1 1	1 0 0	~~5 1 3~~ (after P_2)
P_4	0 0 2	0 0 2	~~7 2 4~~ (after P_3) 7 2 6 ← final

- P_0 will be executed, then Available = $<0,1,0>$ which isn't sufficient for any process, so it is a deadlock.
- Now, we can kill either all deadlocked processes simultaneously or one by one.
- If we kill P_1, $<P_0, P_2, P_3, P_4>$ is a safe sequence.
- After executing, we can restart P_1.

*Binary Semaphore

```
struct binary-semaphore{
        boolean value;
        queueType queue;
};
void wait (binary-semaphore s){
    if (s.value ==1)
        s.value = 0;
    else{
        place this process in s.queue;
        block this process;  // avoids busy waiting
    }
}
```

```
void signal (binary-semaphore s){
    if (s.queue in empty[])
        s.value = 1;
    else{
        remove a process from s.queue
        place it in the ready list;
    }
}
```

* Let $S_0 = 1$, $S_1 = 0$, $S_2 = 0$ are the binary semaphores

Process P_0	P_1	P_2 (Concurrent)
while (true) {	wait (S_1);	wait (S_2);
wait (S_0);	signal (S_0);	signal (S_0);
print '0';		
signal (S_1);		
signal (S_2);		
}		

Q:) How many min and max no. of 0's will be printed?

At least 2

At most 3

· P_1 and P_2 can't be run at the beginning as they are initialized with 0, blocking them
· After 1st loop: $S_0 = 0$ and O/P: 0 ; Also $S_2 = S_1 = 1$
· After that: P_0 is blocked, so we could run S_1 or S_2.
· If we run P_1, $S_1 := 0$ and $S_0 := 1$
· If we run P_2, $S_2 := 0$ and $S_0 := 1$

$$P_0 \to P_1 \to P_0 \to P_2 \to P_0$$
$$P_0 \to P_1 \to P_2 \to P_0$$

· $a=1, b=1, c=1, d=1$

X	Y	Z
P(a)	P(b)	P(c)
P(b)	P(c)	P(d)
P(c)	P(d)	P(a)
CS	CS	CS
V(a)	V(b)	V(c)
V(b)	V(c)	V(d)
V(c)	V(d)	V(a)

P: wait ()
V: signal ()

Q.) Sequence to execute P operations by the processes to avoid deadlock

· For deadlock avoidance, we can avoid circular wait by only accessing in a fixed order.

· Here,

$$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \arrow[r] & d \arrow[lll, bend left] \end{tikzcd}$$

, so a deadlock may occur.

· For the sequence:

X	Y	Z
P(b)	P(b)	P(a)
P(a)	P(c)	P(c)
P(c)	P(d)	P(d)

· Here,

$$\begin{tikzcd} b \arrow[r] \arrow[rr, bend right] & a \arrow[r] & c \arrow[r] & d \end{tikzcd}$$

, so no circular wait and hence deadlock not occur.

Q.) A system contains 3 programs and each requires 3 tape units for its operation. The min^{m} no. of tape units which the system must have such that deadlocks never arise is 7.

max : $n(k-1)+1$

n = no. of processes

k = no. of resources each process requires

P_1 P_2 P_3

After P_1 finishes execution, its resources can be released