Deadlock
- Problem
- System Model
- Conditions
- Prevention
- Avoidance
- Detection

1. THE DEADLOCK PROBLEM

- Deadlock : a set of blocked processes, each holding a resource and waiting to acquire a resource held by another process in the set

eg.

P_0	P_1
wait(A);	wait(B);
wait(B);	wait(A)

context switch

Semaphore A, B initialised to 1

B = 0 when either one of the wait(B) are entered
⇒ one process is holding B and another process is trying to access it, but it will never get released and the wait is atomic so it won't even get interrupted

2. SYSTEM MODEL

- Resource types $R_1 \ldots R_m$ ← memory space, I/O devices semaphores

- each resourcetype R_i has W_i identical instances

Resource
- request
- use
- release

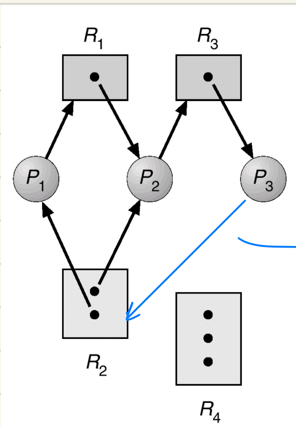
Resource Allocation Graph

V
- $P = \{P_1, P_2 \dots P_n\}$ (processes)
- $R = \{R_1, R_2, \dots, R_m\}$ (all resource types

E
- edge request $(P \rightarrow R)$
- assignment edge $(R \rightarrow P)$

} mutually exhaustive for a pair of P, R

eg.



sequence of process execution
$P_3 \rightarrow P_2 \rightarrow P_1$

yes deadlock, even with cycle

FACTS:

- no cycle ⇒ no deadlock
- cycle
  - one instance / resource ⇒ deadlock
  - many instances / resources ⇒ maybe deadlock

3. DEADLOCK CONDITIONS

↓

all 4 must be simultaneously true

1. Mutual exclusion : one process at a time can hold a resource instance

2. Hold and wait : A process holding at least 1 resource is is waiting to acquire additional resources held by other processes

3. No preemption : A resource can be released only voluntarily by the process holding it : after the process has completed it's task

4. Circular wait : each process in a set is waiting for a process held by the next one, and the last one waits for a process held by the first one

user handles deadcocks, not the os.
ignoring the deadlock is the simplest fastest method.

4. DEADCOCK PREVENTION

prevent 1 of the 4 conditions

Dining Philosopher's Problem:

1. Mutual exclusion : is necessary

2. Hold And wait : Allow philosopher to pick up both chopsticks only if both are present

3. No pre-emption : If a philosopher cannot get the second chopstick, tell him to put down the chopstick that is held.

4. Circular wait :
   - Allow at most 4 to be hungry together ← pigeonhole
   - Odd-even soln
   - All follow left → right, except for one (ordered picking up)

5. DEADLOCK AVOIDANCE

- this algorithm dynamically examines the resource allocation state to ensure that the system never goes into an unsafe state

request $\xrightarrow{\text{check}}$ (if safe) $\longrightarrow$ allocate immediately

$\downarrow$ else

wait

- safe? $\longrightarrow$ a safe completion sequence of all processes exists

- unsafe $\not\Rightarrow$ deadlock

  $\because$ a process can release a resource before completion too

- Banker's Algorithm

$n(P) = n$; $n(R) = m$

m	Available	$\longrightarrow$	Avaible $[R_j]$ = no of avaible instances of R_j
$n \times m$	Max	$\longrightarrow$	$Max[i,j]=k=$ P_i can have max k R_j instances
$n \times m$	Allocation	$\longrightarrow$	$Allo[i,j]=k$ P_i has k R_j instances
$n \times m$	Need	$\longrightarrow$	$N[i,j]=k$ P_i needs k R_j instances

$$N[i,j] = M[i,j] - A[i,j]$$

P1 Safety

Work → m
Finish → n

(1) work = Available
Finish [i] = false ∀i

(2) find i : (Finish[i] == false) && (Need[i,*] ≤ Work)
if none → (4)

loop (3) work = Work + Allocation [i,*] ; Finish[i] = true
(2)

(4) if(finish[i] == true) ∀ i → safe
else → unsafe.

P2 Resource Request

$Request_i$ ← for P_i
$Request_i[j]$ == k ⇝ process P_i wants k instances of resource type R_j

(1) if ($Request_i$ ≤ $Need_i$ ∀ j) → (2)
else → error

(2) if ($Request_i$ ≤ Available ∀ j) → (3)
else P_i → wait

(3) update : $Available_j$ = $Available_i$ − $Request_i$;
$Allocation_i$ = $Allocation_i$ + $Request_i$;
$Need_i$ = $Need_i$ − $Request_i$;

(4) Run P1