# 7. Deadlock

OS는 `resource` 를 관리한다.

- Process는 OS에 `resource`를 요청하며, OS는 process에 `resource`를 제공해야 한다.
- Process는 기본적으로 실행을 위해서 `Resource`가 필요하다.
  1. Hardware: CPU, BUS, MEMORY
  2. Sofware: Semaphore, Lock
- 필요한 `Resource` 중 하나라도 얻지 못 하면, Process는 기다리게 된다.

## Deadlock

- 한 Process가 원하는 Resource를 얻기 위해 대기 중일 때, 해당 Resource를 갖고 있는 다른 process도  같이 대기하고 있는 상태

예시는 아래와 같다.

Example
- System has 2 tape drives
- $P_1$ and $P_2$ each hold one tape drive and each needs another one

Example
- semaphores *A* and *B*, initialized to 1

| $P_0$ | $P_1$ |
| --- | --- |
| *P (A);* | *P(B)* |
| *P (B);* | *P(A)* |

- 이 경우, 두 번째 줄의 P(B)에서 P_0 가 대기하게 되고, P(A)에서 P_1이 대기하게 된다.

# Deadlock 발생 조건 (4가지)

1. Mutual exclusion : 한 Resource는 한 시점에 하나의 process만이 사용 가능하다.
2. No preemption : 다른 Process가 사용 중인 Resource를 뺐을 수 없다.
3. Hold and Wait : 한 Process가 waiting중 일 때, 이미 확보한 Resource를 다시 내어주지 않는다.
4. Circular wait : {P_0, P_1, ... , P_N, P0} 의 wating process set이 존재해야 한다.
    a. 이 경우에는 P_0는 P_1이 가지고 있는 resource를 기다리고 P_N은 P_0가 가지고 있는 resource를 기다린다.

위 4가지 조건이 동시에 발생해야 Deadlock이 발생한다.

# Deadlock을 시각화 해보자

Resource-Allocation Graph

1. **VERTEX**

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

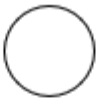
2. **EDGE**

request edge – directed edge $P_1 \rightarrow R_j$

assignment edge – directed edge $R_j \rightarrow P_i$

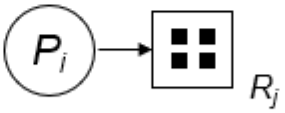
구성 요소는 아래와 같다.

- Process

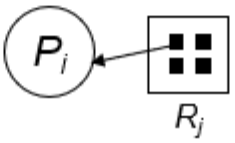


- Resource Type with 4 instances

- $P_i$ requests an instance of $R_j$



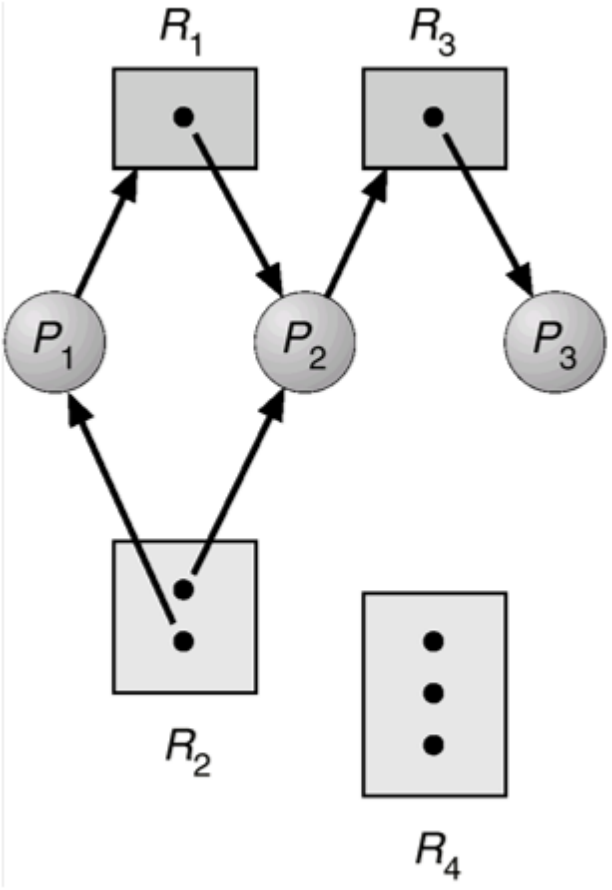

- $P_i$ is holding an instance of $R_j$




- 하나의 Resource는 여러 개의 Instance를 가질 수 있다.
  - Memory에서 page를 생각하면 된다.
  - 하나의 Process에 같은 Resource instance 여러 개를 줘도 된다.
- 할당되는 것은 Resource 전체가 아니라, Resource 속 Instance 이다.

우리는 위 그래프에서 Deadlock이 발생하는지 확인해야 한다.

- Deadlock에 4가지 조건이 Graph에서 발생하는지 확인하자.



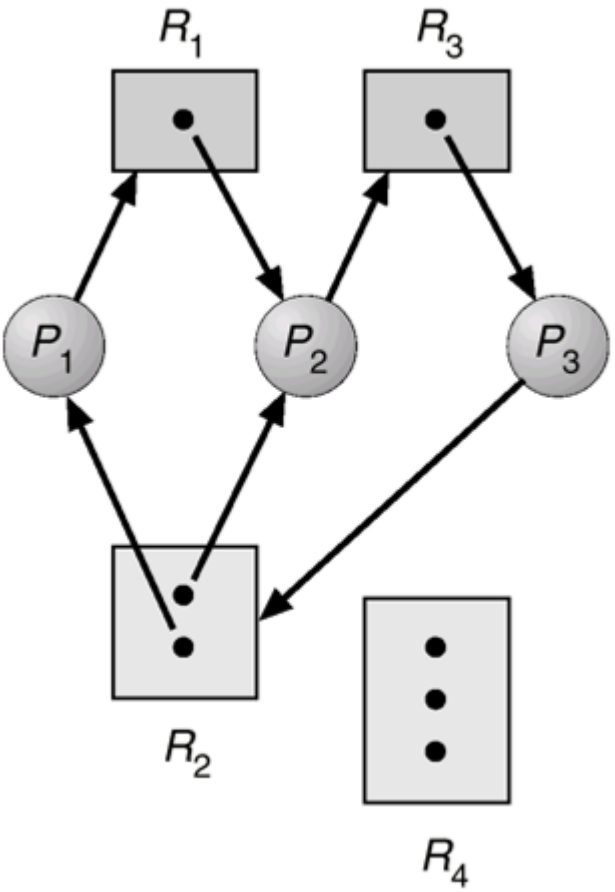

- 위 예시는 Deadlock이 발생하지 않는다.
- Mutual exclusion, No preemption, Hold & wait은 Graph에서 확인할 수 없다.
  - 위 3가지 조건은 알아서 보장된다고 생각하면 된다.
- 따라서 이 Graph에서는 Cycle이 존재하는 지만 확인하면 된다.

Deadlock 발생 여부를 아래와 같이 정리할 수 있다.

1. **Cycle이 존재하지 않으면 Deadlock이 발생하지 않는다.**
2. **Cycle이 존재한다면, Deadlock 발생 가능성이 존재한다.**
   a. 각 Resource type마다 하나의 instance만 존재한다면, 무조건 Deadlock
   b. 각 Resource type마다 여러 개의 instance가 존재한다면, dealock의 가능성이 있으니 확인해보아야 한다.

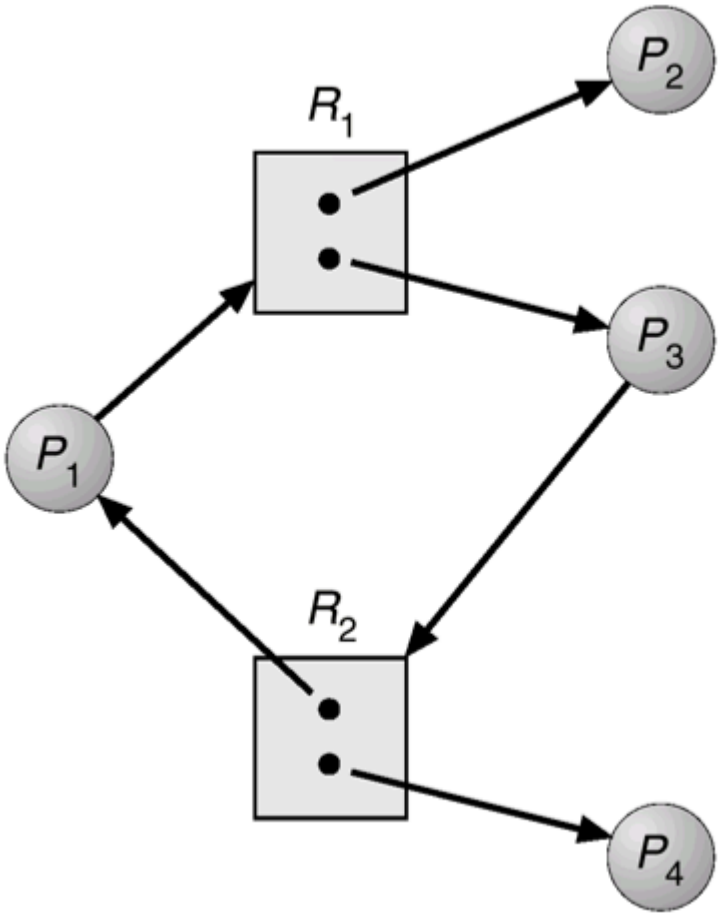
예시를 살펴보자

1. Cycle + Deadlock




- 위 예시에서는 P1가 P2를 기다리고, P2가 P3를 기다리고, P3가 P1를 기다린다. (Cycle)
- R1, R3는 Instance가 하나 뿐이므로 살펴볼 필요가 없고, R2의 경우에는 모든 Resource가 process에 할당되어 있다.
- 따라서 이 경우는 Deadlock이다.

2. Cycle + No Deadlock




- 우선 P1이 P2와 P3를 기다리고, P3도 P1과 P4를 기다리고 있으므로 cycle이 존재한다.
- 그러나 이 경우, P2와 P4는 원하는 모든 Resource를 할당 받은 상태이다.
- 따라서 언젠가 P2 또는 P4가 종료되므로, P1, P3는 resource를 할당 받아 실행될 수 있다.
  - 즉, 모든 process가 언젠가 실행될 수 있다.
- 따라서 이 경우, Deadlock이 아니다.

이처럼, Graph 상에서 cycle이 존재하더라도, Deadlock 여부를 추가적으로 판단해야 한다.

따라서 우리는 Graph로 확인하는 대신에, `Algorithm` 을 이용한다.

## OS가 Deadlock에 대처하는 방법

1. System에서 Deadlock이 절대 발생하지 않도록 하는 방법
    a. Prevent, Avoid

2. System에 Deadlock을 허용하되, Deadlock을 감지하고 처리 (Recover)하는 방법
    a. 지속적으로 Deadlock detection 해야 한다.

3. 그냥 Deadlock이 발생하던 말던 무시하고, Deadlock이 발생하면 그냥 껐다 키는 방법
    a. Deadlock이 OS 입장에서는 굉장히 드물게 발생하기 때문에 사용하는 방법
        i. Deadlock은 크게 Databasem Semaphore, Lock의 경우에서만 발생한다.
    b. 지속적으로 Detecting, Prevention 하는 것(Overhead)보다 위 방법이 효율적이다.
    c. 대부분의 OS가 사용하는 방법이다.

우리는 1번 방법부터 살펴보자

**Deadlock Prevention**

- Deadlock이 발생하기 위한 조건 네 가지 중 하나를 제거한다.

1. **Mutual exclusion을 제거**
    a. Non-Sharable한 변수에 대해서는 필수적인 요소라 제거할 수 없다.

2. **Hold and Wait을 제거**
    a. 필요한 Resource 중 하나라도 확보하지 못 할 경우면 아무 것도 확보하지 않도록 한다.
    b. 필요한 Resource를 모두 동시에 확보하거나, 아무것도 확보하지 않거나 한다.
    c. Low resource utilization, starvation possible 문제 발생

i. 1, 2, 3 Resource가 있을 때, 먼저 3번이 Non-available하여 Resource를 확보하지 못 하였다가, 3번이 Avilable해졌을 때에는 1번이 Non-available해져서 확보하지 못하는 경우에 Starvation 발생

3. **No Preemption을 제거**
   a. Hold & wait을 허용했을 때, 다른 Resource를 확보하지 못하여 대기 중인 Process의 Resource 일부를 뺐을 수 있도록 허용
   b. 일부 Resource를 확보한 process가 다른 Resource 때문에 wait 해야 하는 경우에 이미 확보한 자원은 release하게 한다.
   c. 이후, Wait 중이던 process는 자신이 빼앗긴 resource와 waiting하던 resource 모두를 확보해야 재시작 가능
   d. Hold&Wait의 경우와 마찬가지로 Low resource utilization, starvation possible 문제

4. **Circular wait을 제거**
   a. 각 Resource type에 순서를 매기고(Total ordering), 순서대로만 resource request를 할 수 있도록 한다.
   b. 이 경우, p1→p2→p3에서 p1이 p2가 확보한 resource를 확보하고 싶을 때, p1이 확보하고 있는 자원의 번호는 p1이 지금 요청하는 (p2가 확보하고 있는) resource의 number보다 반드시 작다.
   c. 구조적으로 Circular wait이 발생할 수가 없다.
   d. 더 번호가 큰 Resource가 확보 가능한 상황일 때, 번호가 작은 Resource 때문에 확보할 수 있는 자원을 확보하지 못 하고 대기하게 되어, Resource utilization이 떨어진다.
   e. 그래프 상에서 back edge가 만들어질 수가 없다.

- No preemption 조건을 깨는 방법과 Hold&Wait을 깨는 방법은 과정이 다른 것이지 결과는 비슷하다.

Deadlock prevention 방법은 resource utilization을 낮추고, system throughput을 낮춘다.

- 본질적으로 좋은 방법이 아니다.

Deadlock Avoidance

- **Deadlock 발생 조건 4가지를 모두 허용하되, 동작 과정에서 Deadlock이 발생하지 않도록 조정한다.**
- 미래에 Deadlock이 발생할 가능성이 존재한다면, 해당 동작을 하지 않도록 방지
- 이 방법을 사용하기 위해서는 미래를 예측할 수 있어야 한다.

미래를 예측하기 위해 어떤 정보를 사용할 수 있을까?

1. 각 Process가 각 resource type에서 최대 몇 개의 instance**(Maximum number)**를 사용하는지 기록한다.
2. 현재 각 Proecss에 할당된 resource의 상태

**즉, 이 방법을 사용하기 위해서는 사전에 각 Process가 어떤 resource를 얼만큼 사용하는지 반드시 알아야 한다.**

위 2가지 정보를 활용하여, Deadlock-avoidance algorithm이 동작 과정에서 절대 circular-wait condition이 발생하지 않는 동작만 하도록 한다.

- OS는 process가 resource request를 했을 때, 즉시 process에 자원을 할당해줄 수도 있고, 나중에 자원을 할당해줄 수도 있다.
- 각 동작에 따라 동작 방식이 확연히 달라지는데, 이 중 Circular-wait이 절대 발생하지 않는 방식으로 동작하게 한다.

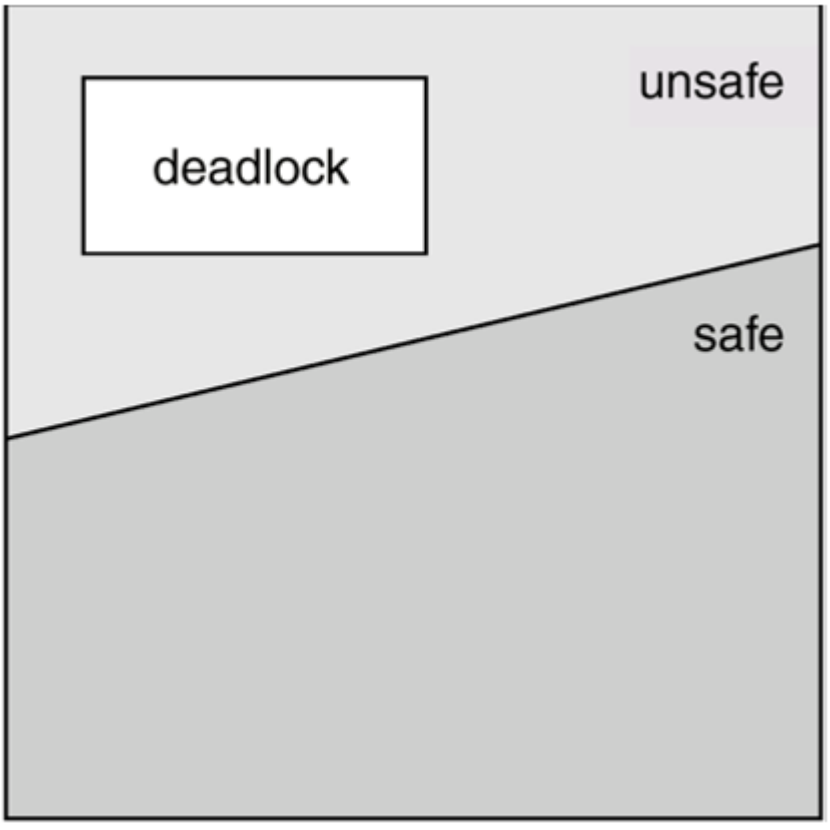
Safe state

- OS가 바로 process에게 resource를 할당해줄 수 있는 안전한 상태
- 요청한 자원을 할당해줘도 Deadlock이 발생할 가능성이 없다.
- **Sequence <P1, P2, ..., Pn>에서, Pi (i = 1 ~ n)에 sequnce 상에서 Pi 앞에 존재하는 (Pj, j < i) process가 갖고 있는 모든 resource와 현재 available한 모든 resource를 Pi에게 할당했을 때, Pi가 원하는 작업을 수행하고 종료할 수 있다면 Safe state이다.**
- System은 모든 process에 대해 Sage state를 형성하는 process seqeunce 하나만 존재해도 safe state라고 판단한다.

Safe state라면 Deadlock은 절대 발생하지 않는다.

Unsafe state라면 Deadlock이 발생할 가능성이 있다.

- 이 경우, Deadlock avoidance 는 Deadlock의 일말의 가능성이 있기 때문에 다른 길을 선택한다.
- Unsafe state라도 safe state에 대한 판단은 Maximum number of  resources of each type으로 하기 때문에 deadlock이 발생하지 않을 수도 있다.
    - 1, 2, 3번의 resource를 하나의 process가 필요로 할 때, safe state는 1, 2, 3번 resource를 동시에 사용하는 것으로 알고 판단하지만, 실제로는 그렇지 않을 수도 있다.
    - 1번을 사용하고 반납 → 2번 사용 → 반납 → 3번 사용




**결국 Deadlock avoidance는 system이 절대 Unsafe state가 될 수 없도록 보장한다.**

Avoidance algorithm

Maximum 사용량은 항상 1이므로, 각 process는 해당 resource를 사용할지 여부만 선언한다.

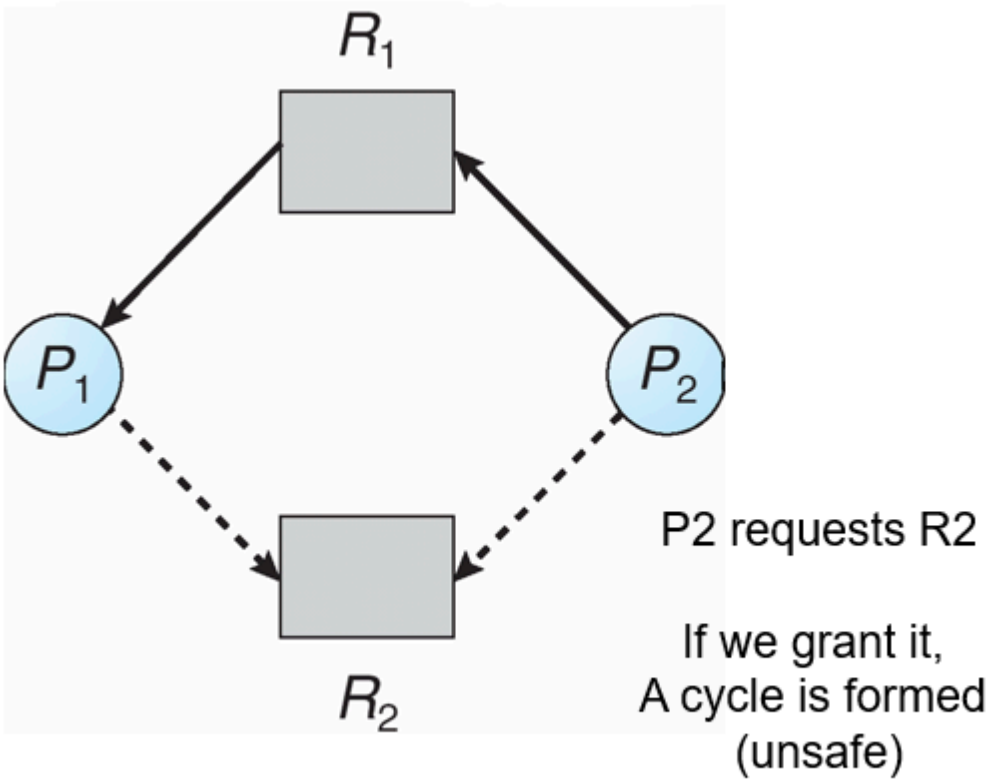
- Single instance of a resource type
  - Resource-allocation graph에서 cycle이 발생하는 지만 확인하면 된다.
  - Resource-allocation graph 사용

- Multiple instances of a resource type
  - Banker's algorithm 사용

**Case A: One instance per resource types**

- Claim edge
  - 점선으로 표시하고, 미래에 사용할 (request 할) 가능성이 있다.
  - 우리가 사전에 알아야 하는 정보이다.
- Request edge
  - Claim edge가 있는 것에서만 사용 가능
  - Claim edge가 실선으로 변경, process가 resource를 요청
- Assignment edge
  - Request edge가 Assingment edge로 바뀌면서 자원 할당
  - 자원이 반납되면 Claim edge로 변경된다.

- Algorithm
  - Request edge를 Assignment edge로 바꾸었을 때, Cycle이 발생한다면 Deadlock detected!!




- 이 경우에서, R2→P2 Assignment edge가 생기면 Cycle이 발생한다.
- Claim edge는 나중에 request edge로 바뀔 가능성이 있기 때문에, 점선도 포함한다.
- Cycle 이 발생했기 때문에, System은 P2의 요청을 거절한다.

**Case B: Multiple instances per resource types**

가정)

- 각 process는 미리 maximum use를 선언한다.
- Process는 resource 때문에 대기해야 할 수 있다.
- 각 Process가 모든 resource를 확보한 이후에는 제한된 시간 내에 모든 resource를 반납해야 한다.

**Banker's algorithm을 위한 자료구조**

Let $n$ = number of processes, and $m$ = number of resources types

**vector**

- **Available**:

  Available[$j$] = $k$ : $k$ instances of resource type $R_j$ are available

***n x m* matrix**

- **Max**: $Max[i,j] = k$ : $P_i$ may request at most $k$ instances of $R_j$.
- **Allocation**:

  Allocation[$i,j$] = $k$ : $P_i$ is currently allocated $k$ instances of $R_j$
- **Need**: If $Need[i,j] = k$ : $P_i$ may need $k$ more instances of $R_j$

  $Need[i,j] = Max[i,j] - Allocation[i,j]$.

**Safety algorithm**

1. Let *Work* and *Finish* be vectors of length $m$ and $n$, respectively. Initialize:

   $Work := Available$

   $Finish[i] = false$ for $i = 1,2, \ldots, n$.

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 a safe state.

1. Work = 시스템에 남아있는 자원의 양, Finish = 각 process가 끝났는지 여부
2. 아직 끝나지 않았으면서, 현재 시스템에 남아있는 자원의 양이 해당 process가 필요로 하는 자원의 양보다 많은 경우, (3)으로 이동한다.

a. 이 과정에서 Need가 work보다 작은 지를 확인한다. (최악의 경우 가정)

3. Work를 해당 process에 할당되어져 있던 자원의 개수만큼 늘리고 (Finished 되었다고 생각), 해당 process를 finished
4. (2)의 조건을 만족하는 process가 더는 없을 때, 모든 process가 종료된 상태라면 safe state이다.

**Resource request algorithm**

$Request_i$ = request vector for process $P_i$.

If $Request_i[j] = k$ then process $P_i$ wants $k$ instances of resource type $R_j$

1. If $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \le Available$, go to step 3. Otherwise $P_i$ must wait, since resources are not available.
3. Pretend to allocate requested resources to $P_i$ by modifying the state as follows:

$$Available := Available - Request_i;$$
$$Allocation_i := Allocation_i + Request_i;$$
$$Need_i := Need_i - Request_i;$$

- *If safe* $\Rightarrow$ *the resources are allocated to* $P_i$
- *If unsafe* $\Rightarrow$ $P_i$ *must wait, and the old resource-allocation state is restored*

1. 요청하는 자원 개수가 필요한 양 (maximum)보다 크면 error 발생
2. 요청하는 자원 개수가 system에 남아있는 자원 개수보다 작지 않으면 available하지 않은 상태이다.
3. (3) 에서의 state 처럼 변경하여, Pi에 요청한 자원을 할당해준 것처럼 보이도록 만든다.
   a. 이렇게 바꾸고 safety를 확인한다.
   b. Safe라면 resource를 할당한다.
   c. Unsafe(Pi는 wait)의 경우에는 (3)에서 바꾼 state를 복구한다.

***Banker's algorithm***

Request 처리는 Resource reqeust algorithm으로, Deadlock 판단은 Safety algorithm으로 한다.

- Request 처리 후, 준 것처럼 상태를 변화한 후에 safety algorithm으로 확인한다.

**Example**

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ *true*)

| | *Allocation* | *Need* | *Available* |
|---|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 7 4 3 | 2 3 0 |
| $P_1$ | 3 0 2 | 0 2 0 | |
| $P_2$ | 3 0 2 | 6 0 0 | |
| $P_3$ | 2 1 1 | 0 1 1 | |
| $P_4$ | 0 0 2 | 4 3 1 | |

- Executing safety algorithm shows that sequence <$P_1$, $P_3$, $P_4$, $P_0$, $P_2$> satisfies safety requirement.

이 경우에, 위의 알고리즘을 적용했을 때, <P1, P3, P4, P0. P2> sequence라면 safety state가 된다.

2번 방법을 살펴보자

Deadlock detection

Deadlock이 발생하도록 냅두고, Deadlock이 실제로 발생하면 그때 처리(Recovery)하자

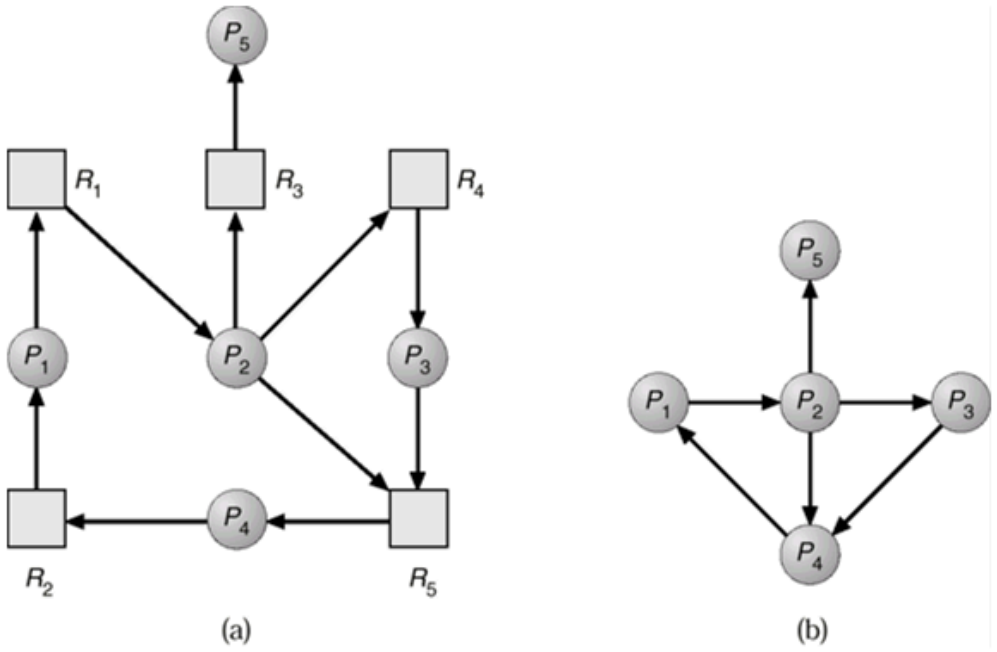
- Detection algorithm
- Recovery scheme

1번 방법과 동일한 2가지 케이스가 있다.

## Case A: Single instance per resource type

### Wait-for graph

- 기존 Resource-allocation graph에서 resource vertex 제거
- Cycle detection은 vertex에 비례한 time complexity 존재
  - # of vertexs = N이면 O(N^2)
  - 특정 process가 다른 process를 기다리는 지를 표시한다.




  - Single instance이기 때문에, cycle이 존재하면 deadlock이 발생한다.

## Case B: Multiple instance per resource type

Deadlock detection algorithm에 사용하는 자료구조는 다음과 같다.

Data structures

- *Available*: vector of length m indicates the number of available resources of each type
- *Allocation*: *n* x *m* matrix defines the number of resources of each type currently allocated to each process
- *Request*: *n* x *m* matrix indicates the current request of each process. If *request[i,j]=k*, then process $P_i$ is requesting *k* more instances of resource type $R_j$

- Deadlock avoidance와 비슷하지만, Deadlock detect는 미래를 예측하는 것이 아니기 때문에, Need나 Claim 같은 정보가 필요하지 않다.

**Detection algorithm**

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize
   *Work* := *Available*
   For *i* = 1,2, …, *n*
   *Finish[i]* = *false*, *if* $Allocation_i$ *is not 0*
   *Finish[i]* = *true*, *otherwise*
2. Find an index *i* such that both:
   (a) *Finish* [*i*] = *false*
   (b) $Request_i \leq Work$
   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*] = *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.

1. 초기화: Work = System에 현재 존재하는 남은 자원, Finish = 할당된 것이 없으면 true
   a. 실제로 종료되지 않았더라도, 할당된 것이 없다면 해당 process를 기다리는 다른 process가 없으므로 종료된 것처럼 명시

2. 아직 종료되지 않고, 요청하는 자원의 개수가 현재 System에 남아있는 자원 개수보다 작은 index를 찾는다.
3. 해당 process가 종료되었다고 가정하고 allocation만큼 work를 늘린다.
4. (2) 조건에 해당하는 process가 없을 때, 모든 process가 finish라면 deadlock이 존재하지 않는다.
    a. 이 경우 i번째 process가 not finished라면 해당 process가 deadlocked

**Example**

- Five processes $P_0$ through $P_4$; three resource types A (7 instances), $B$ (2 instances), and $C$ (6 instances)
- Snapshot at time $T_0$:

| | *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 | |

- Sequence <$P_0$, $P_2$, $P_3$, $P_1$, $P_4$> will result in *Finish*[*i*] = true for all *i*

- <P0, P2, P3, P1 P4> 의 순서라면 deadlock이 존재하지 않는다.
- 나중에 각 process p0가 reqeust하여 deadlock이 발생하면 s
    - 미래 일은 무시하고 현 시점만 가지고 판단한다.
    - 긍정적인 시점에서 확인

**Problem**

이 알고리즘은Deadlock을 감지하기 위해 O(m*n^2)의 시간 복잡도를 요구한다.

- m = resource type, n = process number

- Wait-for graph에서 O(N^2)를 m개의 resource type에 대해 진행해야 함.

그렇다면, 이런 Detection algorithm을 얼마나 자주 호출해야 하는가?

1. 모든 request마다
    a. Overhead가 너무 크다.
2. Allocated가 즉시 되지 않은 경우마다?
3. 주기적으로 검사
    a. Deadlock 발생 확률이 낮기 때문에 가능하다.

Deadlock을 detect 했다면 어떻게 recovery하는가???

1. Deadlock에 걸린 모든 process를 abort
2. Deadlock에 걸린 process 중 하나씩 Abort해보자

(2)번 방법의 경우 어떤 순서로 Abort 해야하는가?

1. Process의 Priority가 낮은 것부터
2. 시작한 지 오래되지 않은 process부터
3. Resource 사용이 적은 것부터
4. 다른 Process가 사용해야 할 Resource를 가진 것부터
5. Terminated 되는 최소한의 process 개수를 찾음
6. Batch job (CPU Bound job) 우선

**Recovery from deadlock: Resource preemption**

1. 가장 cost가 작은 victim을 선택
2. Rollback
    a. Safe state 상태로 되돌림
    b. 해당 Safe state부터 process를 재시작

c. Deadlock이 굉장히 특수한 상황이기에 이런 방법을 사용해서 Deadlock이 다시 발생하지 않는다.

문제점)

- 동일한 process가 계속 victim으로 선택되다 보면, Starvation이 발생
- Cost factor에 rollback 횟수를 넣는 방식으로 해결 가능

## Aviodance vs Detection

- `Avoidance` : 모든 process가 worst case로 동작한다고 가정하고 Deadlock 여부 확인
  - Waste of resource 발생 가능
  - 모든 Process가 자신의 최대 사용량을 모두 요청할 때에도 Deadlock이 발생하지 않는 지를 확인한다.
- `Detection` : 어떤 process도 현재 요청한 개수 이상의 자원을 요청하지 않는다고 가정한다.
  - 현재 상태에 기반해서 Deadlock 판단
  - Best case assumption