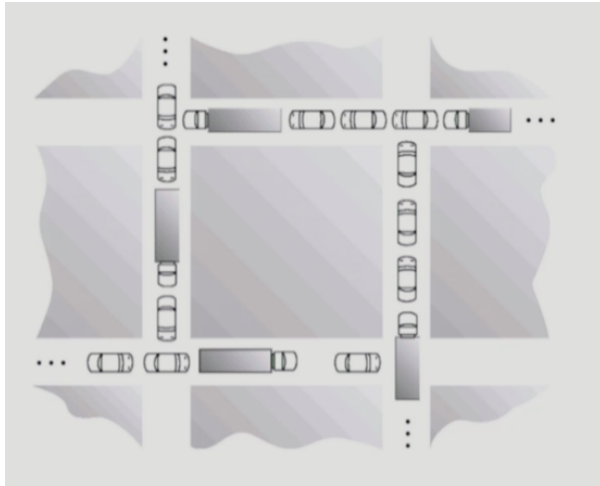


DeadLock (교착상태)

일련의 프로세스들이 서로가 가진 자원을 기다리며 Block 된 상태

자원 => 하드웨어/소프트웨어등 포함하는 모든 자원



Deadlock Example 1

- ✓ 시스템에 2개의 tape drive가 있다
- ✓ 프로세스 P_1 과 P_2 각각이 하나의 tape drive를 보유한 채 다른 하나를 기다리고 있다

Deadlock Example 2

- ✓ Binary semaphores A and B

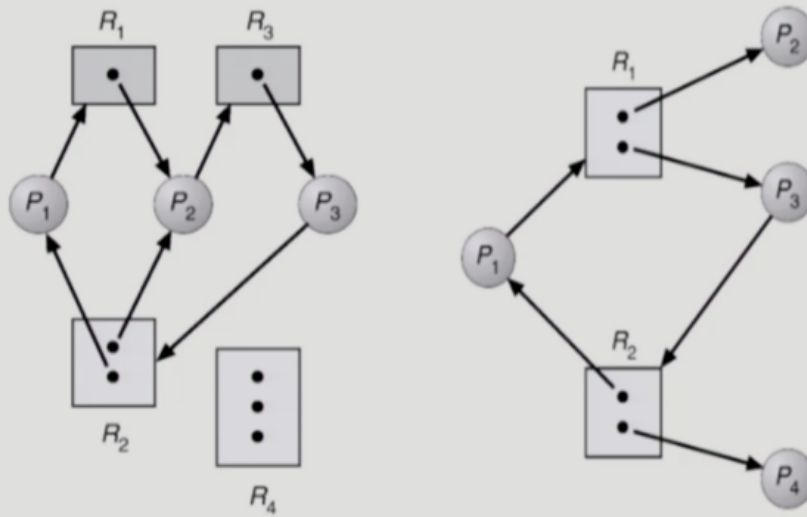
P_0	P_1
$P(A);$	$P(B);$
$P(B);$	$P(A);$

Deadlock 발생 조건 4 (필요충분조건)

- 상호배제(Mutual exclusion) - 프로세스 혼자만 독점적으로 자원을 사용해야함
- 비선점 (No Preemption) - 자원선점 후 빼앗기지 않음
- 보유대기 (Hold and wait) - 프로세스가 보유자원을 놓지 않고 추가적인 자원을 기다림
- 순환대기 (Circular wait) - 자원을 기다리는 프로세스간 사이클 형성

<순환대기 모형>

Resource-Allocation Graph



- 그래프에 cycle이 없으면 deadlock이 아니다
- 그래프에 cycle이 있으면
 - ✓ if **only one instance** per resource type, then **deadlock**
 - ✓ if several instances per resource type, possibility of deadlock

R -> P : 자원이 프로세스에 할당됨

P -> R : 프로세스가 자원에 할당을 요청함

Deadlock 처리방법

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection and recovery
- Deadlock Ignorance

1. DEADLOCK PREVENTION

= Deadlock이 발생하는 조건을 알고 미연에 방지함

Deadlock Prevention

→ Mutual Exclusion

- ✓ 공유해서는 안되는 자원의 경우 반드시 성립해야 함

→ Hold and Wait

- ✓ 프로세스가 자원을 요청할 때 다른 어떤 자원도 가지고 있지 않아야 한다
- ✓ 방법 1. 프로세스 시작 시 모든 필요한 자원을 할당받게 하는 방법
- ✓ 방법 2. 자원이 필요할 경우 보유 자원을 모두 놓고 다시 요청

→ No Preemption

- ✓ process가 어떤 자원을 기다려야 하는 경우 이미 보유한 자원이 선점됨
- ✓ 모든 필요한 자원을 얻을 수 있을 때 그 프로세스는 다시 시작된다
- ✓ State를 쉽게 save하고 restore할 수 있는 자원에서 주로 사용 (CPU, memory)

→ Circular Wait

- ✓ 모든 자원 유형에 할당 순서를 정하여 정해진 순서대로만 자원 할당
- ✓ 예를 들어 순서가 3인 자원 R_i 를 보유 중인 프로세스가 순서가 1인 자원 R_j 를 할당받기 위해서는 우선 R_i 를 release해야 한다

➡ Utilization 저하, throughput 감소, starvation 문제

- circular wait ex) 1,3,5 자원이 필요한 경우 낮은 번호 부터 자원을 획득하는 순서를 정해놓음

=> 미리 데드락을 방지하는 방법은 비효율적이다.

2. DEADLOCK AVOIDANCE

= 프로세스가 시작 될 때 프로세스 평생동안 쓸 자원의 최대량을 알고 자원할당을 결정함

Deadlock Avoidance

→ 시스템이 safe state에 있으면

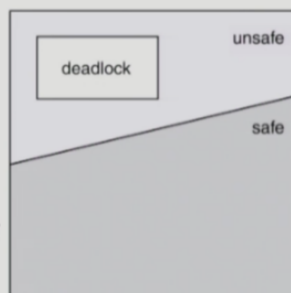
⇒ no deadlock

→ 시스템이 unsafe state에 있으면

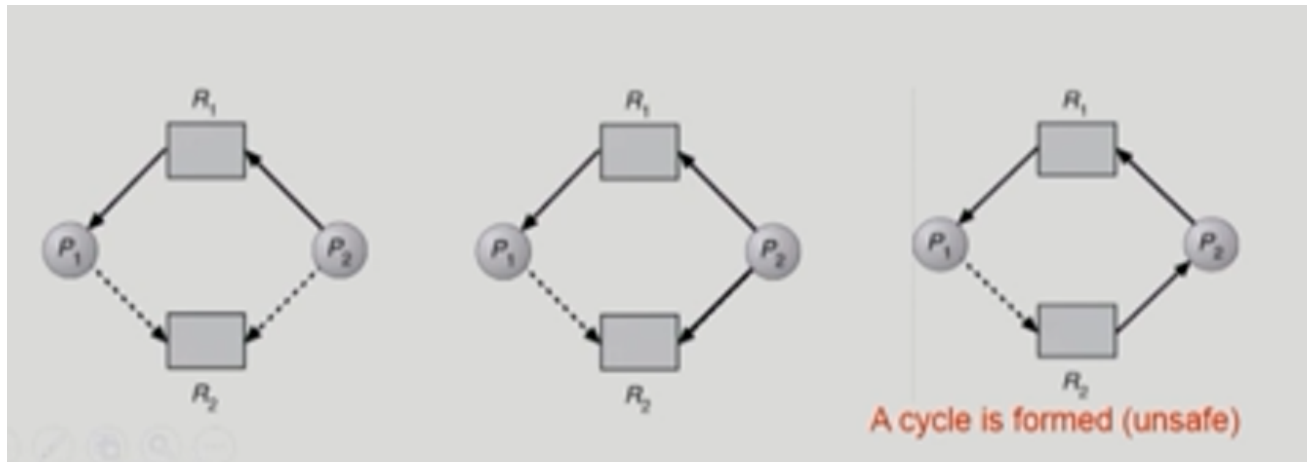
⇒ possibility of deadlock

→ Deadlock Avoidance

- ✓ 시스템이 unsafe state에 들어가지 않는 것을 보장
- ✓ 2가지 경우의 avoidance 알고리즘
 - Single instance per resource types
 - Resource Allocation Graph algorithm 사용
 - Multiple instances per resource types
 - Banker's Algorithm 사용



1. 자원당 하나의 인스턴스가 있는 경우



2. 자원당 여러개 인스턴스

Banker's Algorithm

Example of Banker's Algorithm

→ 5 processes P_0, P_1, P_2, P_3, P_4

→ 3 resource types A (10), B (5), and C (7) instances. 10 5 7

→ Snapshot at time T_0

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u> (Max - Allocation)
	A B C	A B C	A B C	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		1 2 2
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

* sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ 가 존재하므로 시스템은 **safe state**

보수적, 최악의 상황을 가정하는 알고리즘

가용자원(Available)보다 많을 가능성이 있는 요청 (Need Max-Allocation) 은 무조건 거부함

ex) available 332 일 때 P_0 가 1만 요청해도 안들어줌

가용자원이 남아도 최대가능요청자원을 넘어설 때까지 안줌

프로세스의 최대 요청자원을 순서대로 다 처리할수 있는 시퀀스가 존재하면 Safe state

Deadlock Avoidance

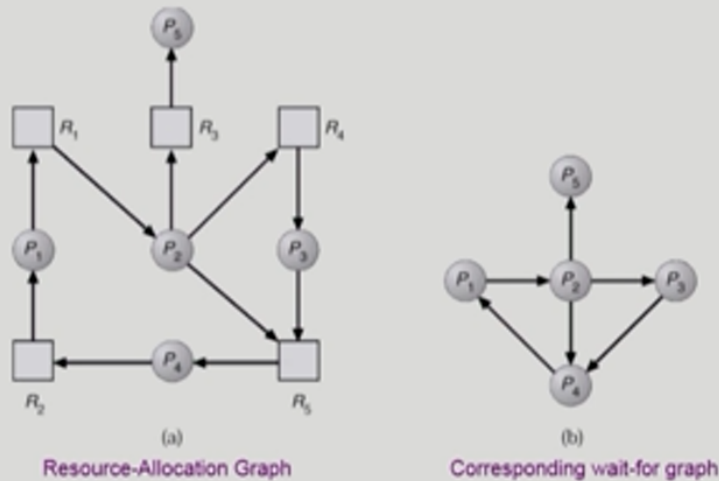
- ➔ Deadlock avoidance
 - ✓ 자원 요청에 대한 부가정보를 이용해서 자원 할당이 **deadlock**으로 부터 안전(**safe**)한지를 동적으로 조사해서 안전한 경우에만 할당
 - ✓ 가장 단순하고 일반적인 모델은 프로세스들이 필요로 하는 각 자원별 최대 사용량을 미리 선언하도록 하는 방법임
- ➔ **safe state**
 - ✓ 시스템 내의 프로세스들에 대한 **safe sequence**가 존재하는 상태
- ➔ **safe sequence**
 - ✓ 프로세스의 **sequence** $\langle P_1, P_2, \dots, P_n \rangle$ 이 **safe**하려면 P_i ($1 \leq i \leq n$)의 자원 요청이 "**가용 자원 + 모든 P_j ($j < i$)의 보유 자원**"에 의해 충족되어야 함
 - ✓ 조건을 만족하면 다음 방법으로 모든 프로세스의 수행을 보장
 - P_i 의 자원 요청이 즉시 충족될 수 없으면 모든 P_j ($j < i$)가 종료될 때까지 기다린다
 - P_{i-1} 이 종료되면 P_i 의 자원요청을 만족시켜 수행한다

3. DEADLOCK DETECTION AND RECOVERY

1) Detection

- detection 은 그래프/뱅크스 둘다 이용해서 찾을 수 있음

Deadlock Detection and Recovery



자원의 최대 사용량을 미리 알릴 필요 없음 → 그래프에 점선이 없음

- 그래프는 resource를 빼고 process들만으로 연결한 축약형 그래프로도 표현가능

Deadlock Detection and Recovery

- Deadlock Detection
 - ✓ Resource type 당 single instance인 경우
 - 자원할당 그래프에서의 cycle이 곧 deadlock을 의미
 - ✓ Resource type 당 multiple instance인 경우
 - Banker's algorithm과 유사한 방법 활용
- Wait-for graph 알고리즘
 - ✓ Resource type 당 single instance인 경우
 - ✓ Wait-for graph
 - 자원할당 그래프의 변형
 - 프로세스만으로 node 구성
 - P_i 가 가지고 있는 자원을 P_k 가 기다리는 경우 $P_k \rightarrow P_i$
 - ✓ Algorithm
 - Wait-for graph에 사이클이 존재하는지를 주기적으로 조사
 - $O(n^2)$

=> DFS/BFS 처럼 최대 $O(n^2)$ 만큼 걸림

- 프로세스당 자원이 여러개 일 때

Deadlock Detection and Recovery

→ Resource type 당 multiple instance인 경우

- ✓ 5 processes: P_0, P_1, P_2, P_3, P_4
- ✓ 3 resource types: A (7), B (2), and C (6) instances
- ✓ Snapshot at time T_0 :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	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			

- ✓ No deadlock: sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will work!

☞ "Request"는 추가요청가능량이 아니라 현재 실제로 요청한 자원량을 나타냄

2) Recovery

Deadlock Detection and Recovery

→ Recovery

- ✓ **Process termination**
 - Abort all deadlocked processes
 - Abort one process at a time until the deadlock cycle is eliminated
- ✓ **Resource Preemption**
 - 비용을 최소화할 victim의 선정
 - safe state로 rollback하여 process를 restart
 - Starvation 문제
 - 동일한 프로세스가 계속해서 victim으로 선정되는 경우
 - cost factor에 rollback 횟수도 같이 고려

- process termination - 프로세스 죽이기

1. 전부 죽이기

2. 하나씩 죽여보기

- Resource preemption

희생양 프로세스를 선정하고 자원을 빼앗음

이 때, Starvation 문제 (우선순위가 낮은 프로세스는 오랫동안 자원할당을 받지 못함)

=> 대기시간과 뺏긴 횟수등 다양한 요소를 고려해서 순서 정함

4. Deadlock Ignorance

Deadlock이 일어나든 말든 무시하는

prevention 은 오버헤드가 너무 큼

detection 역시 조금만 느려져도 detection 루틴이 실행되면 이것역시 시스템 오버헤드가 너무 큼

사용자에게 핸들링을 맡김