

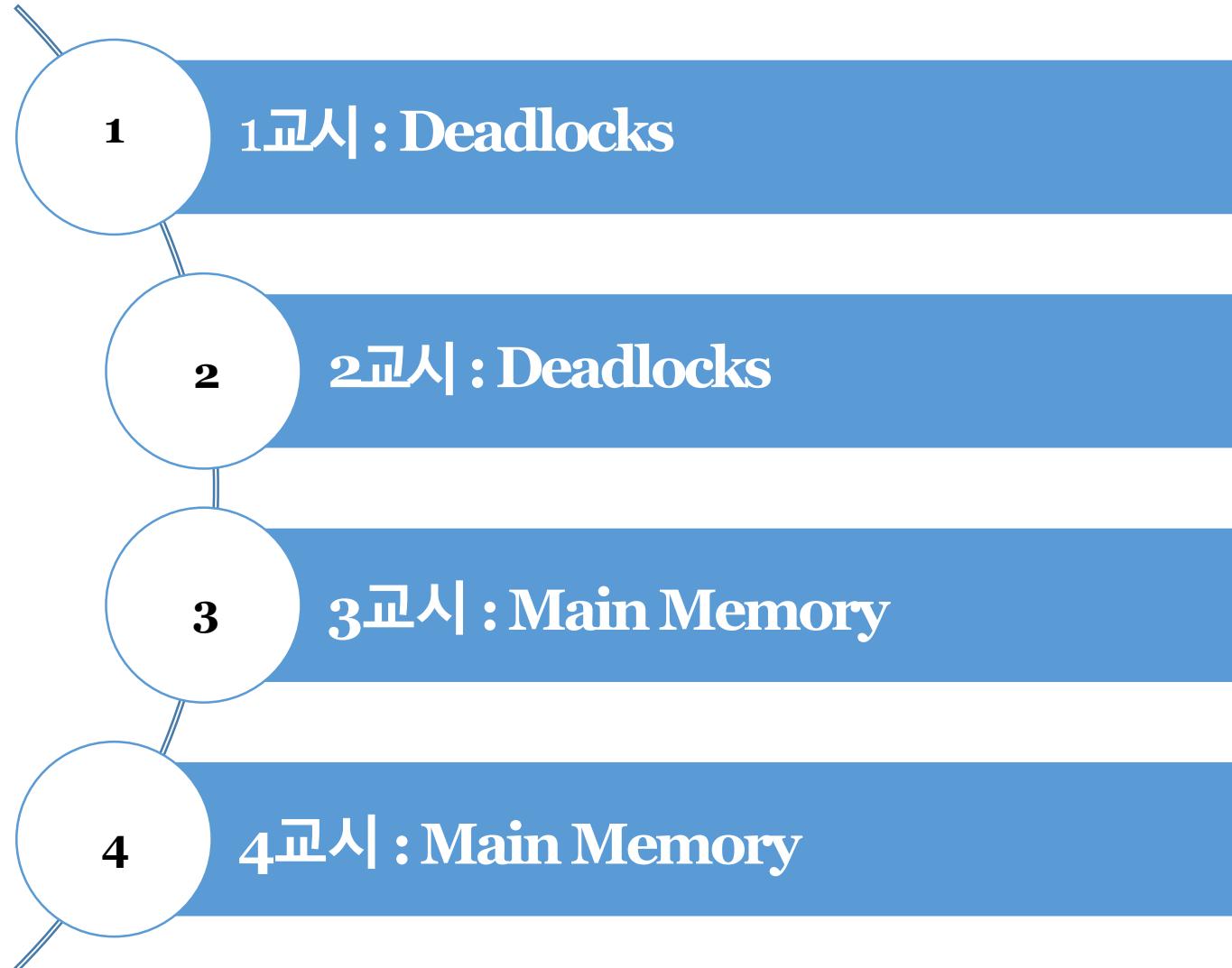
LG전자 BS팀

『3일차』: 오전

- ◆ 훈련과정명 : OS 기본
- ◆ 훈련기간 : 2023.05.30 ~ 2023.06.02



목차



『7과목』

1-2교시 :

Deadlocks





학습목표

- 이 워크샵에서는 뮤텍스 잠금mutex locks을 사용할 때 교착 상태가 발생할 수 있는 방법을 설명을 할 수 있다.
- 교착 상태를 특징 짓는 네 가지 필수 조건 정의을 할 수 있다.
- 리소스 할당 그래프에서 교착 상태 식별을 할 수 있다.
- 교착 상태를 방지하기 위한 4가지 접근 방식 평가를 할 수 있다.
- 교착상태 회피를 위한 뱅커 알고리즘 적용을 할 수 있다.
- 교착 상태 감지 알고리즘 적용을 할 수 있다.
- 교착 상태에서 복구하기 위한 접근 방식 평가를 할 수 있다.



눈높이 체크

- 시스템 모델을 알고 계신가요?
- 교착 상태 특성화를 알고 계신가요?
- 교착 상태 처리 방법을 알고 계신가요?
- 교착 상태 방지를 알고 계신가요?
- 교착 상태 회피를 알고 계신가요?
- 교착 상태 감지를 알고 계신가요?
- 교착 상태에서 복구를 알고 계신가요?



1. System Model

기본 개념

- 시스템은 리소스로 구성.
- 리소스 유형 R_1, R_2, \dots, R_m
 - CPU 주기, 메모리 공간, I/O 장치
- 각 리소스 유형 R_i 에는 W_i 인스턴스가 있다.
- 각 프로세스는 다음과 같이 리소스를 사용.

request

use

release



1. System Model

세마포어와의 교착 상태

Data:

A semaphore S_1 initialized to 1

A semaphore S_2 initialized to 1

Two threads T_1 and T_2

T_1 :

`wait(s_1)`

`wait(s_2)`

T_2 :

`wait(s_2)`

`wait(s_1)`



2. Deadlock Characterization

기본 개념

- 네 가지 조건이 동시에 성립하면 교착 상태가 발생할 수 있다.
- 상호 배제 Mutual exclusion : 한 번에 하나의 스레드만 리소스를 사용할 수 있다.
- 보류 및 대기 Hold and wait : 하나 이상의 리소스를 보유한 스레드가 다른 스레드가 보유한 추가 리소스를 획득하기 위해 대기 중.
- 선점 없음 No preemption : 스레드가 작업을 완료한 후에 리소스를 보유하고 있는 스레드에 의해서만 자원을 자발적으로 해제할 수 있다.
- Circular wait : T_0 은 T_1 이 보유한 리소스를 기다리고, T_1 은 T_2 , ..., T_{n-1} 이 보유한 리소스를 기다리는 대기 스레드 집합 $\{T_0, T_1, \dots, T_n\}$ 이 있다 T_n 이 보유한 자원을 기다리고 있고, T_n 은 T_0 이 보유한 자원을 기다리고 있다.



2. Deadlock Characterization

리소스 할당 그래프

- 꼭지점 세트 V와 모서리 세트 E.

V is partitioned into two types:

$T = \{T_1, T_2, \dots, T_n\}$, 시스템의 모든 활성 스레드로 구성된 세트.

$R = \{R_1, R_2, \dots, R_m\}$, 시스템의 모든 리소스 유형으로 구성된 집합

request edge – directed edge $T_i \rightarrow R_j$

assignment edge – directed edge $R_j \rightarrow T_i$

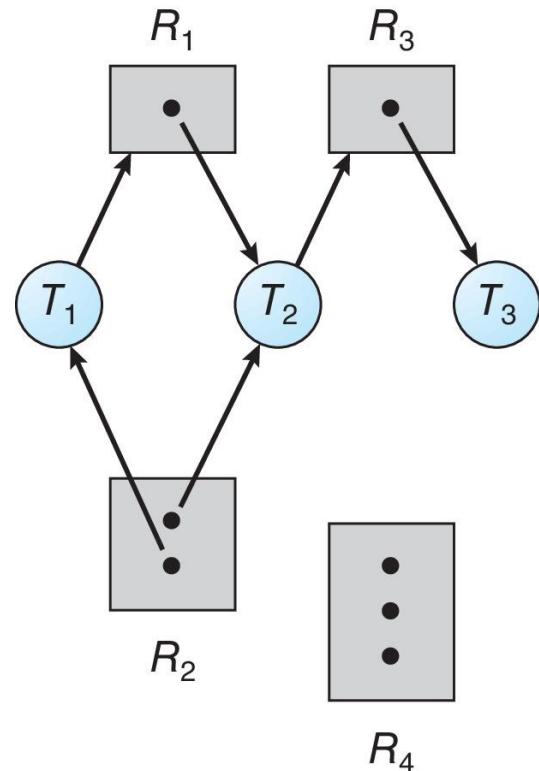
요청
할당

2. Deadlock Characterization



리소스 할당 그래프 예

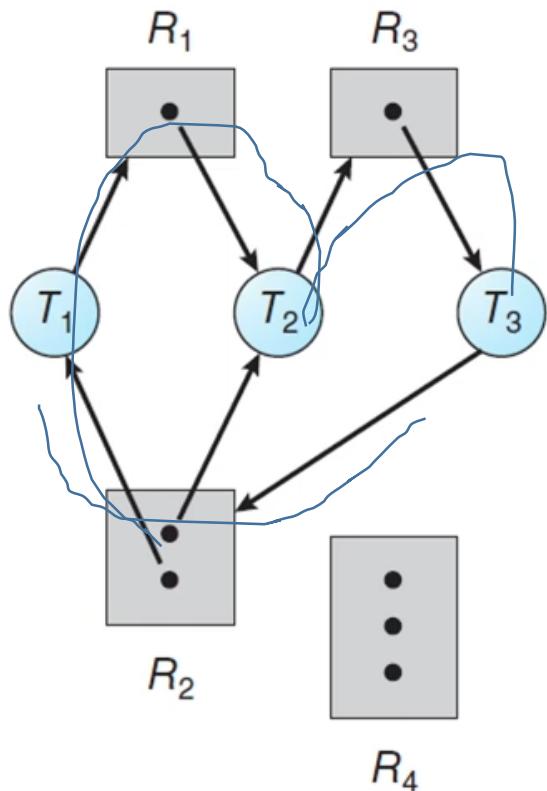
- R_1 의 한 인스턴스
- R_2 의 두 인스턴스
- R_3 의 한 인스턴스
- R_4 의 세 가지 인스턴스
 - T_1 은 R_2 의 인스턴스 하나를 보유하고 있으며 R_1 의 인스턴스를 기다리고 있다.
 - T_2 는 R_1 의 한 인스턴스와 R_2 의 한 인스턴스를 보유하고 있으며 R_3 의 인스턴스를 기다리고 있다.
 - T_3 는 R_3 의 한 인스턴스를 보유.
 - $T = \{T_1, T_2, T_3\}$
 - $R = \{R_1, R_2, R_3, R_4\}$
 - $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$



사이클이 1 교착 상태가 없는 그래프

2. Deadlock Characterization

- 사이클이 있고 교착 상태가 있는 그래프



Deadlock 발생

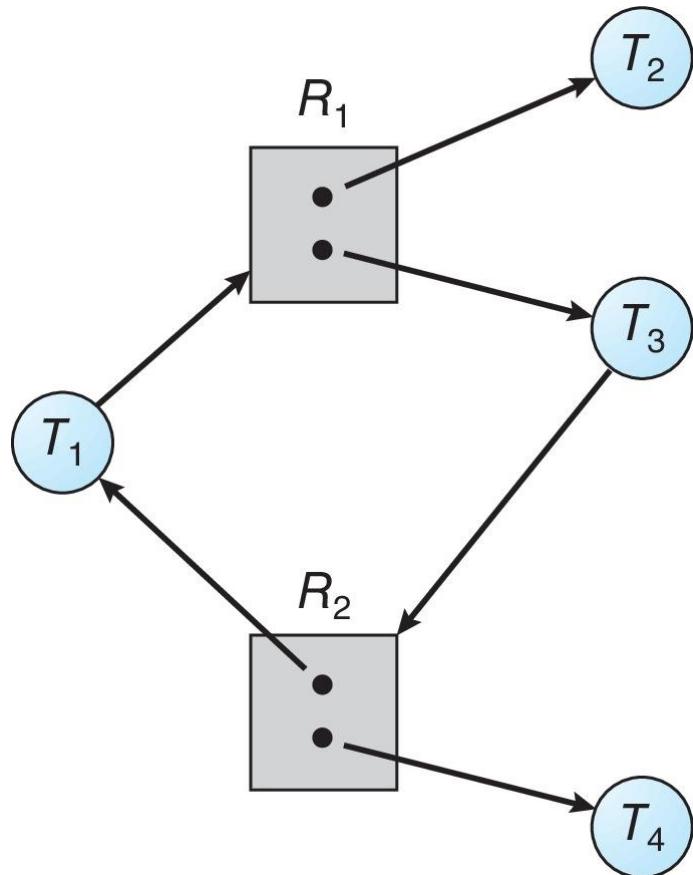
- Two cycles exist in this graph.
 - $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
 - $T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$

2. Deadlock Characterization



리소스 할당 그래프 예

- 사이클이 있지만 교착 상태가 없는 그래프



Deadlock 발생하지
않음

$T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$



2. Deadlock Characterization

기본 정보

- 그래프에 주기가 없는 경우 \Rightarrow 교착 상태가 없음
- 그래프에 주기가 포함된 경우 \Rightarrow
 - 리소스 유형당 하나의 인스턴스만 있는 경우 교착 상태
 - 리소스 유형당 인스턴스가 여러 개인 경우 교착 상태 가능성



3. Methods for Handling Deadlocks

기본 개념

- 시스템이 교착 상태에 빠지지 않도록 한다.
 - 교착 상태 방지 Deadlock prevention
 - 교착 상태 회피 Deadlock avoidance
- 시스템이 교착 상태에 들어간 후 복구하도록 허용
- 문제를 무시하고 시스템에서 교착 상태가 발생하지 않는 척한다.



4. Deadlock Prevention

Deadlock Prevention

- 교착 상태에 필요한 네 가지 조건 중 하나를 무효화.
- 상호 배제 Mutual Exclusion – 공유 가능한 리소스(예: 읽기 전용 파일)에는 필요하지 않습니다. 공유할 수 없는 리소스를 유지해야 한다.
- 보류 및 대기 Hold and Wait – 스레드가 리소스를 요청할 때마다 다른 리소스를 보유하지 않도록 보장해야 한다.
 - 스레드가 실행을 시작하기 전에 모든 리소스를 요청하고 할당받도록 요구하거나 스레드가 리소스를 할당하지 않은 경우에만 스레드가 리소스를 요청하도록 허용.
 - 낮은 자원 활용도, 기아 가능



4. Deadlock Prevention

Deadlock Prevention

- **선점 없음 No Preemption :**

- 일부 리소스를 보유하고 있는 프로세스가 즉시 할당할 수 없는 다른 리소스를 요청하면 현재 보유하고 있는 모든 리소스가 해제.
- 스레드가 대기 중인 리소스 목록에 선점된 리소스가 추가.
- 스레드는 이전 리소스와 요청 중인 새 리소스를 다시 얻을 수 있는 경우에만 다시 시작.

- **순환 대기 Circular Wait :**

- 모든 리소스 유형의 총 순서를 부과하고 각 스레드가 열거된 순서대로 리소스를 요청하도록 요구.

4. Deadlock Prevention

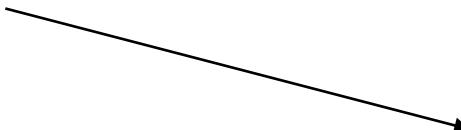
Circular Wait

- 순환 대기 조건을 무효화하는 것이 가장 일반적.
- 각 리소스(즉, 뮤텍스 잠금)에 고유 번호를 할당하기만 하면 된다.
- 자원은 순서대로 획득해야 한다.
- 만약에:

`first_mutex = 1`

`second_mutex = 5`

code for `thread_two` could not be
written as follows:



```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```



5. Deadlock Avoidance

기본 개념

- 시스템에 사용 가능한 추가 선형적 정보가 필요.
- 가장 간단하고 가장 유용한 모델은 각 스레드가 필요할 수 있는 각 유형의 최대 리소스 수 **maximum number** 를 선언해야 한다는 것.
- 교착 상태 회피 알고리즘은 자원 할당 상태를 동적으로 검사하여 순환 대기 상태가 절대 발생하지 않도록 한다.
- 리소스 할당 상태는 사용 가능한 할당된 리소스의 수와 프로세스의 최대 요구 사항으로 정의된다.



5. Deadlock Avoidance

안전한 상태

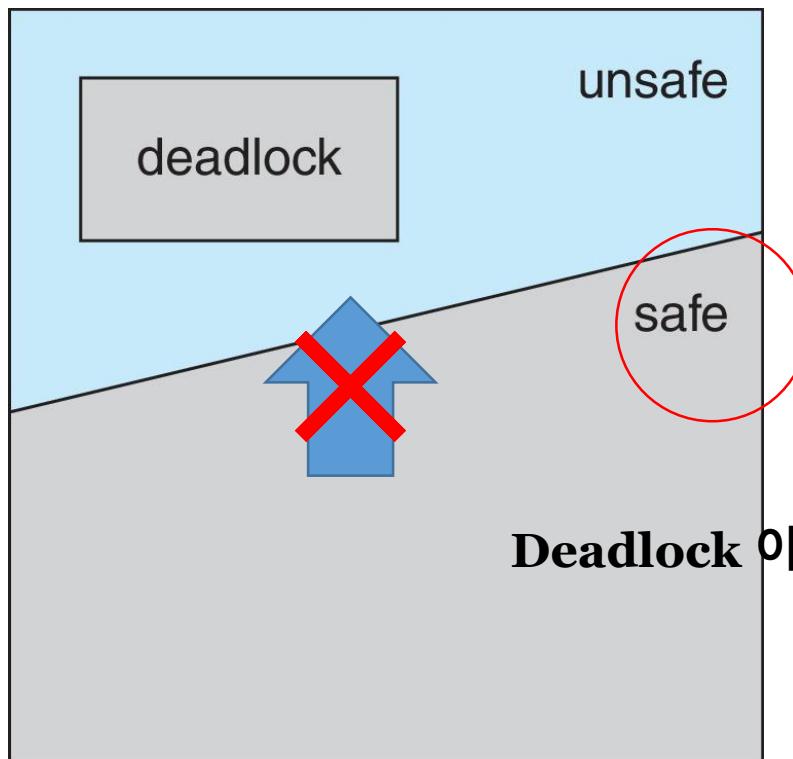
- 스레드가 사용 가능한 리소스를 요청할 때 시스템은 즉각적인 할당이 시스템을 안전한 상태로 유지하는지 여부를 결정해야 한다.
- 각 T_i 에 대해 T_i 가 여전히 요청할 수 있는 리소스가 현재 사용 가능한 리소스 + 보유 리소스로 충족될 수 있도록 시스템의 모든 스레드 시퀀스 $\langle T_1, T_2, \dots, T_n \rangle$ 이 존재하는 경우 시스템은 안전한 상태. 모든 $T_j, j < i$
- T_i 리소스 요구 사항을 즉시 사용할 수 없는 경우 T_i 는 모든 T_j 가 완료될 때까지 기다릴 수 있다.
- T_j 가 완료되면 T_i 는 필요한 리소스를 얻고 실행하고 할당된 리소스를 반환하고 종료할 수 있다.
- T_i 가 종료되면 T_{i+1} 은 필요한 리소스를 얻을 수 있다.



5. Deadlock Avoidance

Safe, Unsafe, Deadlock State

- 시스템이 안전한 상태인 경우 ⇒ 교착 상태가 없음
- 시스템이 안전하지 않은 상태인 경우 ⇒ 교착 상태 가능성
- 회피 voidance ⇒ 시스템이 절대 안전하지 않은 상태에 들어가지 않도록 한다.





5. Deadlock Avoidance

Avoidance Algorithms

- 자원 유형의 단일 인스턴스
 - 리소스 할당 그래프 resource-allocation graph 사용
- 리소스 유형의 여러 인스턴스
 - 뱅커 알고리즘 Banker's Algorithm 사용



5. Deadlock Avoidance

리소스 할당 그래프 체계

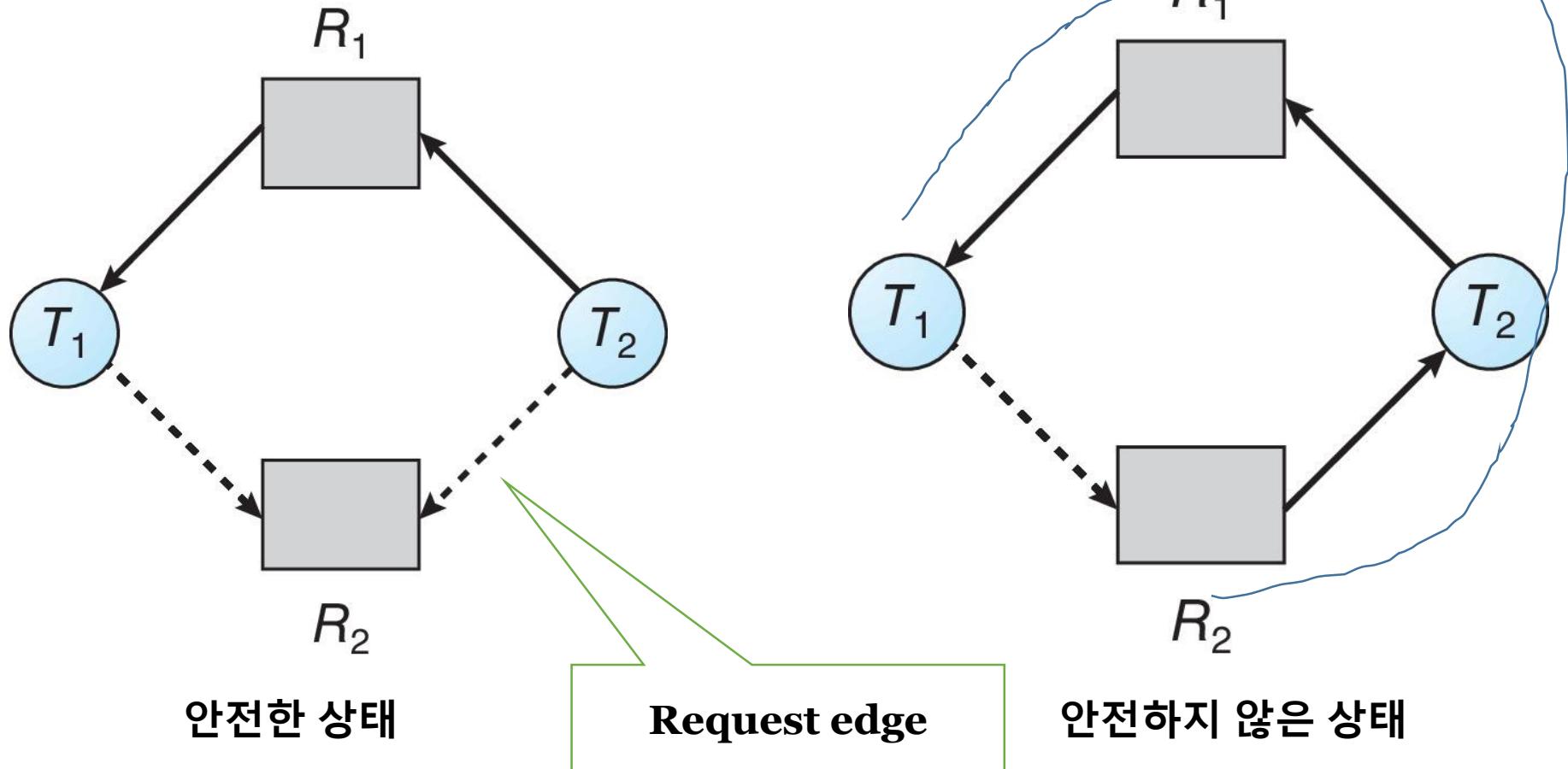
- 클레임 에지 Claim edge $T_i \rightarrow R_j$ 는 프로세스 T_j 가 리소스 R_j 를 요청할 수 있음을 나타낸다. 점선으로 표현
- 클레임 에지 Claim edge 는 스레드가 리소스를 요청할 때 요청 에지 request edge 로 변환.
- 스레드에 리소스가 할당되면 요청 엣지 Request edge 가 할당 엣지 assignment edge 로 변환됨
- 스레드에서 리소스를 해제하면 할당 에지 assignment edge 가 클레임 에지 claim edge로 다시 변환.
- 리소스는 시스템에서 선형적 priori 으로 요구되어야 함.



5. Deadlock Avoidance

Resource-Allocation Graph

- 리소스 할당 그래프와 리소스 할당 그래프의 안전하지 않은 상태





5. Deadlock Avoidance

자원 할당 그래프 알고리즘

- 스레드 T_i 가 리소스 R_j 를 요청한다고 가정.
- 요청 에지를 할당 에지로 전환해도 자원 할당 그래프에 주기가 형성되지 않는 경우에만 요청을 승인할 수 있다.



5. Deadlock Avoidance

Banker's Algorithm

- 리소스의 여러 인스턴스
- 각 스레드는 선형적으로 최대 사용을 요구해야 한다.
- 스레드가 리소스를 요청하면 대기해야 할 수 있다.
- 스레드가 모든 리소스를 가져오면 제한된 시간 내에 반환해야 한다.



5. Deadlock Avoidance

1. Banker's Algorithm 데이터 구조

- n = 프로세스 수, m = 리소스 유형 수
- 사용 가능 Available : 길이 m 의 벡터. 사용 가능한 $[j] = k$ 인 경우 사용 가능한 자원 유형 R_j 의 k 인스턴스가 있다.
- 최대 Max : $n \times m$ 행렬. $Max[i,j] = k$ 이면 프로세스 T_i 는 최대 k 개의 리소스 유형 R_j 인스턴스를 요청할 수 있다.
- 할당 Allocation : $n \times m$ 행렬. $Allocation[i,j] = k$ 이면 T_i 는 현재 R_j 의 k 인스턴스에 할당.
- 필요 Need : $n \times m$ 행렬. $Need[i,j] = k$ 이면 T_i 는 작업을 완료하기 위해 k 개의 R_j 인스턴스가 더 필요할 수 있다.

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



5. Deadlock Avoidance

2. Safety Algorithm

1. Work 및 Finish를 각각 길이 m 및 n의 벡터라고 한다.

초기화:

$Work = Available$

$Finish [i] = false$ for $i = 0, 1, \dots, n-1$

2. 다음과 같은 i를 찾는다.

(a) $Finish [i] = false$

(b) $Need_i \leq Work$

해당 i가 없으면 4단계로 이동.

3. $Work = Work + Allocation_i$

$Finish[i] = true$

go to step 2

4. $Finish [i] == true$ 이면 시스템은 안전한 상태.



5. Deadlock Avoidance

3. Process Ti를 위한 리소스 요청 알고리즘

- Request_i = 프로세스 T_i 에 대한 요청 벡터. $\text{Request}_i[j] = k$ 이면 프로세스 T_i 는 리소스 유형 R_j 의 k 인스턴스를 원한다.
 1. $\text{Request}_i \leq \text{Need}_i$ 인 경우 2단계로 이동. 그렇지 않으면 프로세스가 최대 청구를 초과했기 때문에 오류 조건을 발생.
 2. $\text{Request}_i \leq \text{Available}$ 이면 3단계로 이동. 그렇지 않으면 리소스를 사용할 수 없으므로 T_i 는 대기.
 3. 다음과 같이 상태를 수정하여 요청된 리소스를 T_i 에 할당하는 척 한다.

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

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

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

- 안전한 경우 \Rightarrow 리소스가 T_i 에 할당됨
- 안전하지 않은 경우 $\Rightarrow T_i$ 는 기다려야 하며 이전 리소스 할당 상태가 복원됨.

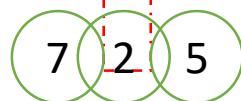


5. Deadlock Avoidance

뱅커 알고리즘의 예

- 5개의 스레드 $T_0 \sim T_4$;
3가지 리소스 유형:
A(인스턴스 10개), B(인스턴스 5개) 및 C(인스턴스 7개)
- 시간 T_0 의 스냅샷:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	7	5	3	3	3	2
T_1	2	0	0	3	2	2			
T_2	3	0	2	9	0	2			
T_3	2	1	1	2	2	2			
T_4	0	0	2	4	3	3			



B는 5-2는 3
A=10-7
C=7-5



5. Deadlock Avoidance

뱅커 알고리즘의 예

- 매트릭스 Need의 내용은 Max – Allocation으로 정의:
 $\text{Need } [i,j] = \text{Max}[i,j] - \text{Allocation } [i,j]$

	<i>Need</i>		
	A	B	C
T_0	7	4	3
T_1	1	2	2
T_2	6	0	0
T_3	0	1	1
T_4	4	3	1

- $\langle T_1, T_3, T_4, T_2, T_0 \rangle$ 순서가 안전 기준을 만족하므로 시스템은 안전한 상태.

5. Deadlock Avoidance

Example: T1 Request (1,0,2)가 새롭게 요청될 때

- 요청 \leq 사용 가능(즉, $(1,0,2) \leq (3,3,2)$) \Rightarrow 참인지 확인

T1	만족 조건	Allocation	Need	Available	Allocation	Need	Available
		A B C	A B C	A B C	A B C	A B C	A B C
	$Request_1 \leq Need_1$ $(1,0,2) \leq (1,2,2)$	T_0 0 1 0	T_1 3 0 2	T_2 3 0 2	T_3 2 1 1	T_4 0 0 2	T_0 0 1 0
	$Request_1 \leq Available$ $(1,0,2) \leq (3,3,2)$		T_1 0 2 0			T_1 2 0 0	T_1 2 0 0
	$(2,0,0) + (1,0,2) = (3,0,2)$			T_2 6 0 0		T_2 3 0 2	T_2 6 0 0
	$(1,2,2) - (1,0,2) = (0,2,0)$			T_3 0 1 1		T_3 2 1 1	T_3 0 1 1
	$(3,3,2) - (1,0,2) = (2,3,0)$			T_4 4 3 1		T_4 0 0 2	T_4 4 3 1

- 안전 알고리즘을 실행하면 시퀀스 $< T_1, T_3, T_4, T_0, T_2 >$ 가 안전 요구 사항을 충족함을 보여준다.
- T_4 가 $(3,3,0)$ 에 대한 요청을 승인할 수 있을까?
- T_0 의 $(0,2,0)$ 요청을 승인할 수 있을까?

$$\begin{aligned}
 & Request_4 \leq Need_4 \\
 & (3,3,0) \leq (4,3,1) \\
 & Request_4 \leq Available \\
 & (3,3,0) \not\leq (2,3,0)
 \end{aligned}$$

5. Deadlock Avoidance



Example: T₁ Request (1,0,2)가 새롭게 요청될 때

- 요청 ≤ 사용 가능(즉, (1,0,2) ≤ (3,3,2) ⇒ 참인지 확인

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
T ₀	0	1	0	7	4	3	2	3	0
T ₁	3	0	2	0	2	0			
T ₂	3	0	2	6	0	0			
T ₃	2	1	1	0	1	1			
T ₄	0	0	2	4	3	1			

To

Request₀ ≤ Need₀
(0,2,0) ≤ (7,4,3)

Request₁ ≤ Available
(0,2,0) ≤ (2,3,0)
(0,1,0) + (0,2,0) = (0,3,0)
(7,4,3) - (0,2,0) = (7,2,3)
(2,3,0) - (0,2,0) = (2,1,0)

- To의 (0,2,0) 요청을 승인할 수 있을까?



6. Deadlock Detection

기본 개념

- 시스템이 교착 상태 deadlock state 에 들어가도록 허용
- 탐지 알고리즘 Detection algorithm
- 복구 계획 Recovery scheme



6. Deadlock Detection

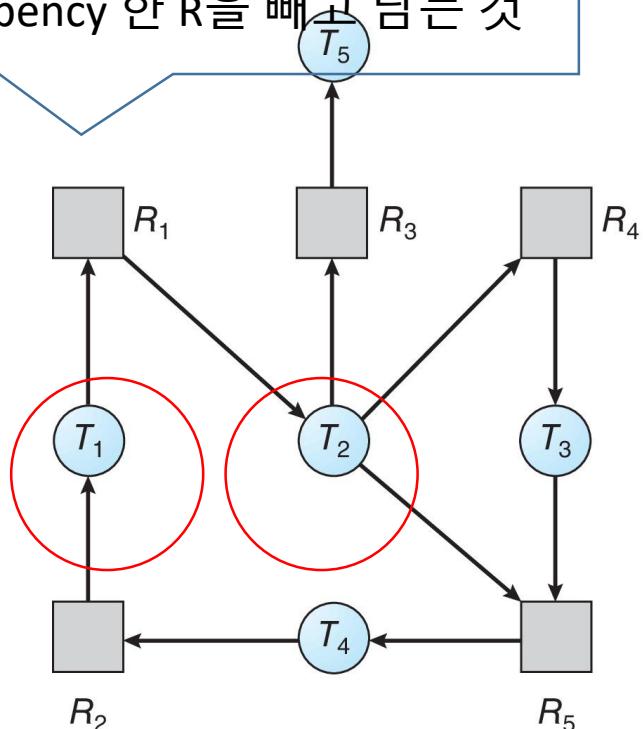
각 리소스 유형의 단일 인스턴스

- 대기 그래프 유지
 - 노드는 스레드.
 - $T_i \rightarrow T_j$ T_i 가 T_j 를 기다리는 경우
- 그래프에서 주기를 검색하는 알고리즘을 주기적으로 호출. 순환이 있으면 교착상태가 존재
- 그래프에서 주기를 감지하는 알고리즘에는 n^2 작업의 순서가 필요. 여기서 n 은 그래프의 정점 수.

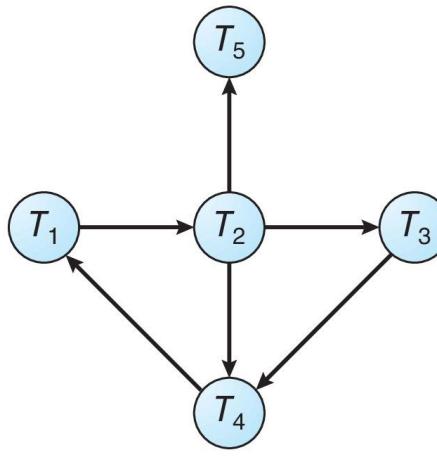
6. Deadlock Detection

리소스 할당 그래프 및 대기 그래프

Dependency 한 R을 빼고 남는 것



(a)



(b)

Resource-Allocation Graph

Corresponding wait-for graph



6. Deadlock Detection

리소스 유형의 여러 인스턴스

- 사용 가능 Available : 길이가 m인 벡터는 각 유형의 사용 가능한 자원 수를 나타냄.
- 할당 Allocation : $n \times m$ 행렬은 각 스레드에 현재 할당된 각 유형의 리소스 수를 정의.
- 요청 Request : $n \times m$ 행렬은 각 스레드의 현재 요청을 나타냄.
Request [i][j] = k이면 스레드 Ti는 자원 유형 Rj의 k개 이상의 인스턴스를 요청.



6. Deadlock Detection

Detection Algorithm

- Work 및 Finish를 각각 길이 m 및 n의 벡터로 설정.

초기화:

- a) $\text{Work} = \text{Available}$
 - b) For $i = 1, 2, \dots, n$, if $\text{Allocation}_i \neq \mathbf{0}$, then
 $\text{Finish}[i] = \text{false}$; otherwise, $\text{Finish}[i] = \text{true}$
-
- 다음과 같은 인덱스 i를 찾는다.
 - a) $\text{Finish}[i] == \text{false}$
 - b) $\text{Request}_i \leq \text{Work}$
 - 해당 i가 없으면 4단계로 이동.



6. Deadlock Detection

Detection Algorithm

$Work = Work + Allocation_i$

$Finish[i] = true$

go to step 2

- $Finish[i] == \text{false}$ 인 경우 일부 $i, 1 \leq i \leq n$ 에 대해 시스템은 교착 상태에 있다. 또한 $Finish[i] == \text{false}$ 인 경우 T_i 는 교착 상태.
- 알고리즘은 시스템이 교착 상태에 있는지 여부를 감지하기 위해 $O(m \times n^2)$ 작업 순서가 필요.



6. Deadlock Detection

검출 알고리즘의 예

- 5개의 스레드 T_0 에서 T_4 까지; 3개의 리소스 유형 A(인스턴스 7개), B(인스턴스 2개) 및 C(인스턴스 6개)
- 시간 T_0 의 스냅샷:

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

- 시퀀스 $\langle T_0, T_2, T_3, T_1, T_4 \rangle$ 는 모든 i 에 대해 $\text{Finish}[i] = \text{true}$ 가 된다.



6. Deadlock Detection

검출 알고리즘의 예

- T_2 는 유형 C의 추가 인스턴스를 요청.

	<u>Request</u>		
	A	B	C
T_0	0	0	0
T_1	2	0	2
T_2	0	0	1
T_3	1	0	0
T_4	0	0	2

T_2 는 유형 C의
추가되면
Deadlock 발생

- 시스템 상태?

- 스레드 T_0 이 보유한 리소스를 회수할 수 있지만 다른 프로세스를 수행하기에는 리소스가 부족. 요청
- 프로세스 T_1, T_2, T_3 및 T_4 로 구성된 교착 상태가 존재.



6. Deadlock Detection

탐지 알고리즘 사용

- 호출 시기 및 빈도는 다음에 따라 다르다.
 - 얼마나 자주 교착 상태가 발생할 가능성이 있을까?
 - 얼마나 많은 프로세스를 롤백해야 할까?
 - ✓ 분리된 주기마다 하나씩
- 감지 알고리즘을 임의로 호출하면 리소스 그래프에 많은 주기가 있을 수 있으므로 교착 상태에 빠진 많은 스레드 중 어떤 스레드가 교착 상태를 "일으켰는지" 알 수 없다.



7. Recovery from Deadlock

프로세스 종료

- 모든 교착 상태 스레드 중단
- 교착 상태 주기가 제거될 때까지 한 번에 한 프로세스씩 중단
- 중단하려면 어떤 순서로 선택해야 할까?
 1. 스레드의 우선순위
 2. 스레드가 계산된 시간 및 완료까지 걸리는 시간
 3. 스레드가 사용한 리소스
 4. 스레드가 완료해야 하는 리소스
 5. 종료해야 하는 스레드 수
 6. 스레드가 대화식입니까 아니면 배치일까?



7. Recovery from Deadlock

자원 선점

- 피해자 선택 Selecting a victim – 비용 최소화
- 롤백 Rollback– 안전한 상태로 돌아가서 해당 상태의 스레드를 다시 시작.
- 기아 Starvation– 동일한 스레드가 항상 피해자로 선택될 수 있으며 비용 요소에 롤백 횟수를 포함.

『8과목』

3-4교시 :

Main Memory





학습목표

- 이 워크샵에서는 메모리 하드웨어를 구성하는 다양한 방법에 대한 자세한 설명 제공을 할 수 있다.
- 다양한 메모리 관리 기술을 논의할 수 있다.
- 순수 분할과 페이징을 통한 분할을 모두 지원하는 인텔 펜티엄에 대한 자세한 설명 제공을 할 수 있다.



Syllabus

눈높이 체크

- 연속 메모리 할당을 알고 계신가요?
- 페이징을 알고 계신가요?
- 페이지 테이블의 구조를 알고 계신가요?
- 스와핑을 알고 계신가요?

1. Background

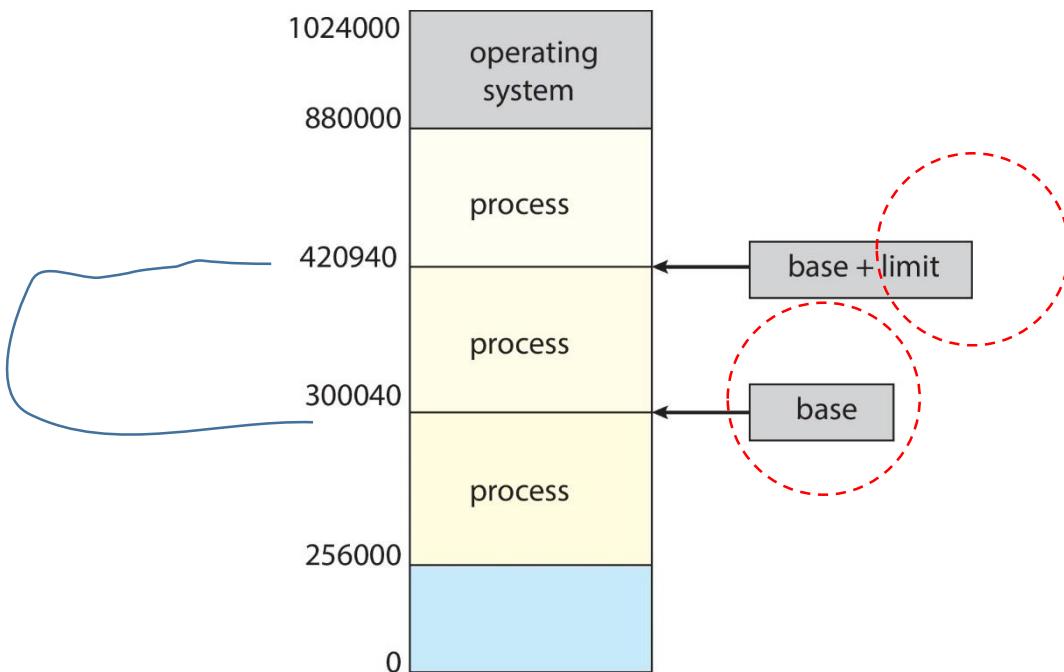
기본 개념

- 프로그램을 (디스크에서) 메모리로 가져와 실행하려면 프로세스 내에 배치해야 한다.
- 메인 메모리와 레지스터는 CPU가 직접 액세스할 수 있는 유일한 스토리지다.
- 메모리 장치는 다음의 스트림만 볼 수 있다.
 - 주소 + 읽기 요청 또는
 - 주소 + 데이터 및 쓰기 요청
- 레지스터 액세스는 하나의 CPU 클록(또는 그 이하)에서 수행.
- 메인 메모리는 많은 주기를 가져서 **스톨 stall** 을 일으킬 수 있다.
- 캐시 Cache 는 메인 메모리와 CPU 레지스터 사이에 있다.
- 올바른 작동을 보장하는 데 필요한 메모리 보호

1. Background

Protection

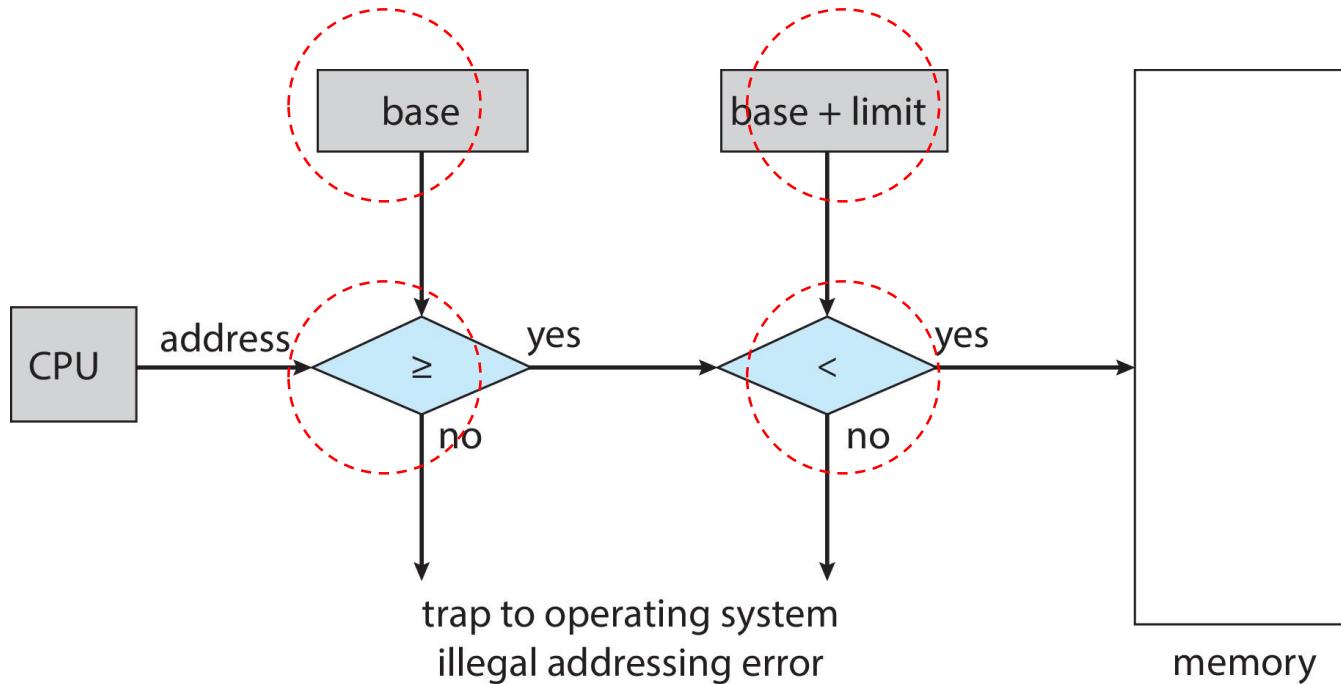
- 프로세스가 주소 공간에 있는 주소에만 액세스할 수 있도록 해야 한다.
- 프로세스의 논리 주소 공간을 정의하는 한 쌍의 기본 및 제한 레지스터를 사용하여 이러한 보호를 제공할 수 있다.



1. Background

하드웨어 주소 보호

- CPU는 사용자 모드에서 생성된 모든 메모리 액세스를 확인하여 해당 사용자에 대한 기본과 제한 사이에 있는지 확인해야 함.



- 베이스 레지스터와 리미트 레지스터를 로드하는 명령은 권한이 부여.



1. Background

주소 바인딩 Address Binding

- 입력 큐 **input queue**를 형성하기 위해 메모리로 가져올 준비가 된 디스크의 프로그램
 - 지원하지 않으면 주소 0000에 로드.
- 첫 번째 사용자가 항상 0000에서 물리적 주소를 처리해야 하는 불편함
 - 어떻게 그럴 수 없을까?
- 프로그램 수명의 다양한 단계에서 다양한 방식으로 표현되는 주소
 - 소스 코드 주소는 일반적으로 기호
 - 컴파일된 코드 주소는 재배치 가능한 주소에 바인딩.
 - ✓ 즉, "이 모듈의 시작 부분에서 14바이트"
 - 링커 또는 로더는 재배치 가능한 주소를 절대 주소에 바인딩.
 - ✓ 즉, 74014
 - 각 바인딩은 하나의 주소 공간을 다른 주소 공간에 매핑.



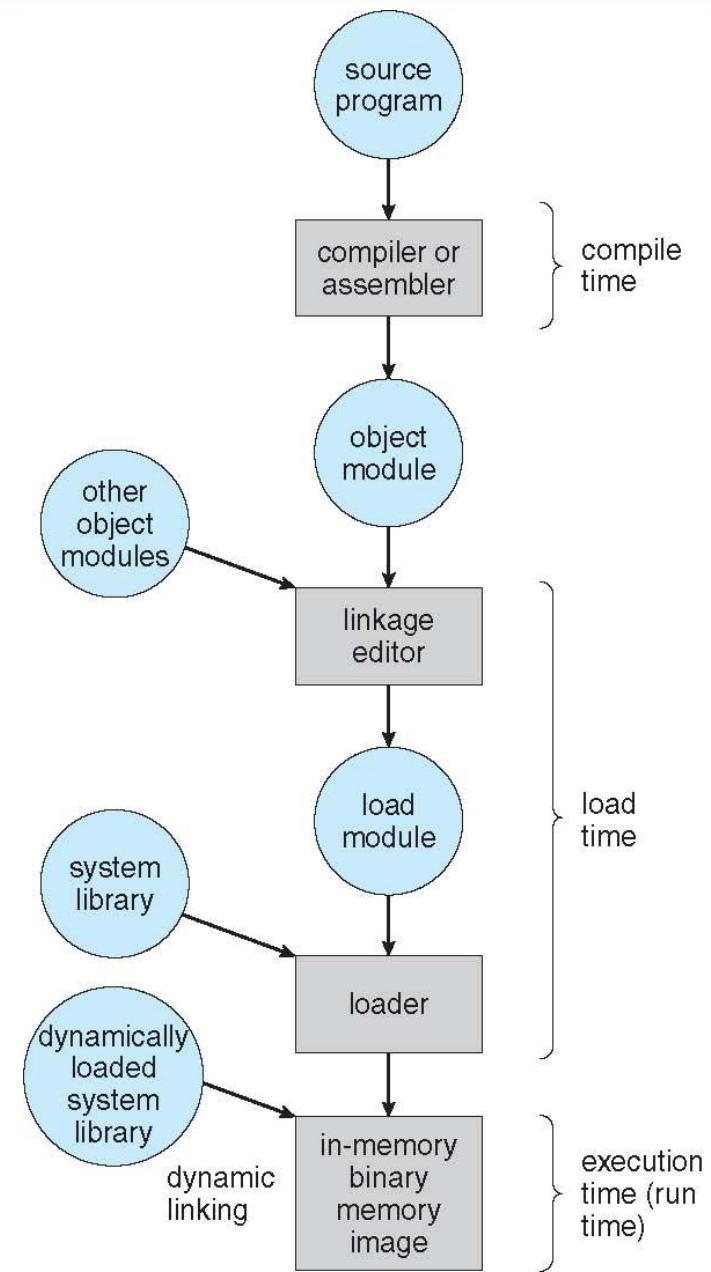
1. Background

명령 및 데이터를 메모리에 바인딩

- 메모리 주소에 대한 명령 및 데이터의 주소 바인딩은 세 가지 다른 단계에서 발생할 수 있다.
- 컴파일 시간 **Compile time** : 메모리 위치가 선험적으로 알려진 경우 절대 코드 **absolute code** 가 생성될 수 있다. 시작 위치가 변경되면 코드를 다시 컴파일해야 함
- 로드 시간 **Load time** : 컴파일 시간에 메모리 위치를 알 수 없는 경우 재배치 가능한 코드 **relocatable code** 를 생성해야 한다.
- 실행 시간 **Execution time** : 프로세스가 실행 중에 한 메모리 세그먼트에서 다른 메모리 세그먼트로 이동할 수 있는 경우 런타임까지 지연된 바인딩
- 주소 맵(예: 기본 및 제한 레지스터)에 대한 하드웨어 지원 필요

1. Background

사용자 프로그램의 다단계 처리





1. Background

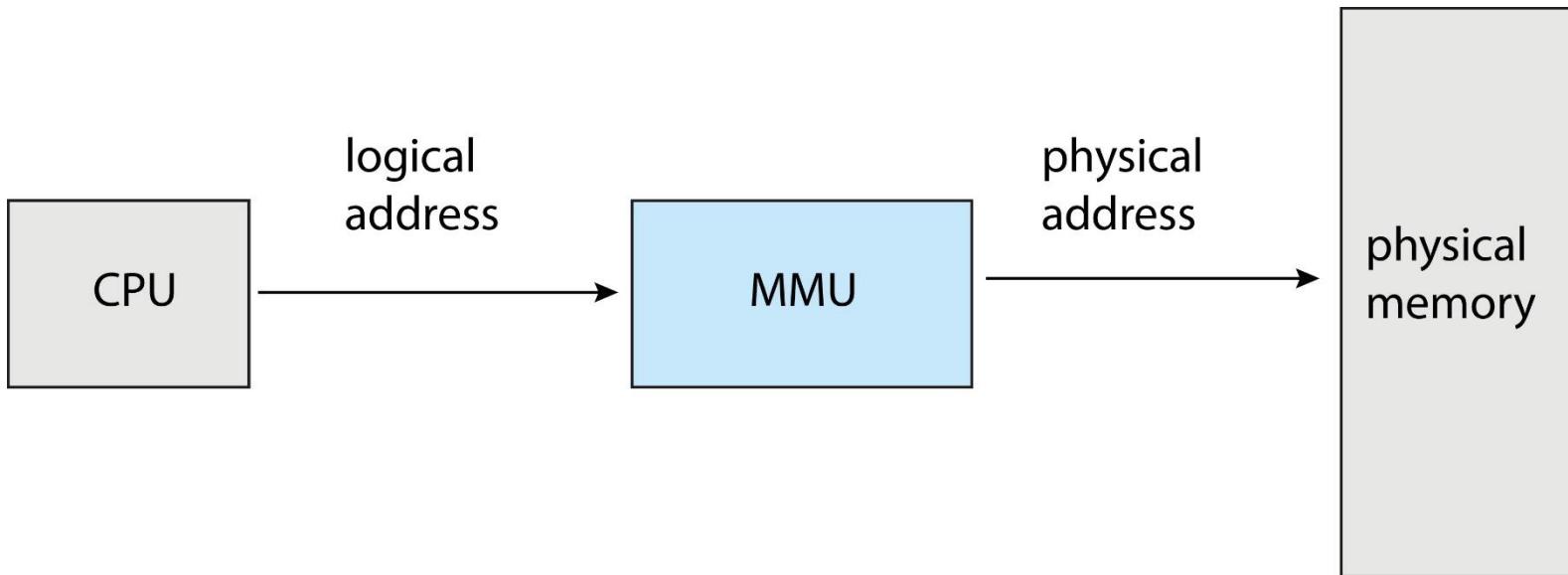
논리적 대 물리적 주소 공간

- 별도의 물리적 주소 공간 **physical address space**에 바인딩된 논리적 주소 공간 **logical address space**의 개념은 적절한 메모리 관리의 핵심.
 - 논리 주소 **Logical address** – CPU에서 생성. 가상 주소라고도 함
 - 물리적 주소 **Physical address** – 메모리 장치에서 볼 수 있는 주소
- 논리적 주소와 물리적 주소는 컴파일 시간과 로드 시간 주소 바인딩 체계에서 동일. 논리(가상) 및 물리 주소는 실행 시간 주소 바인딩 방식이 다르다.
- 논리 주소 공간 **Logical address space**은 프로그램에 의해 생성된 모든 논리 주소 집합.
- 물리적 주소 공간 **Physical address space**은 프로그램에 의해 생성된 모든 물리적 주소의 집합.

1. Background

메모리 관리 장치(MMU)

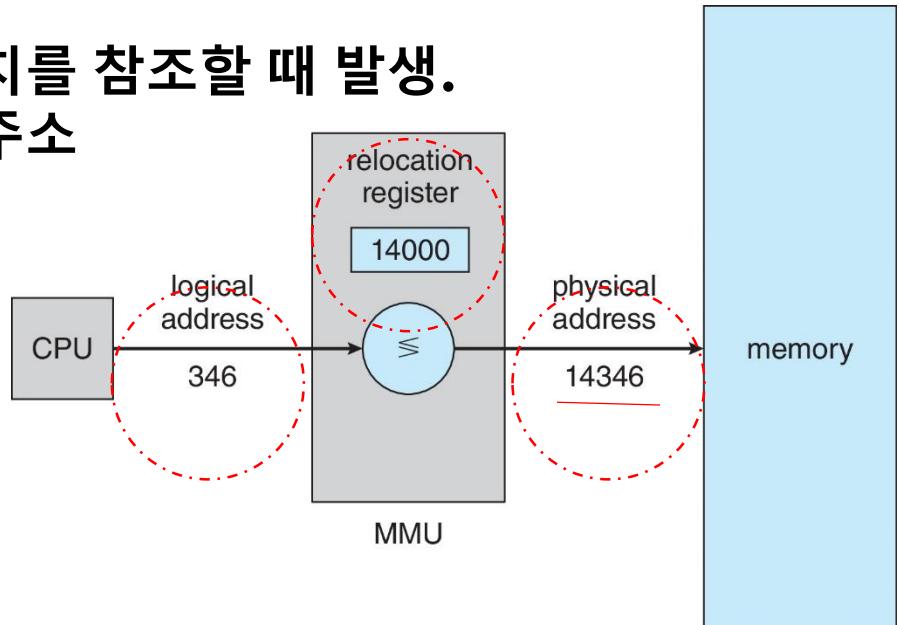
- **Memory-Management Unit** 런타임에 가상 주소를 물리적 주소로 맵핑하는 하드웨어 장치



1. Background

메모리 관리 장치(MMU)

- 베이스 레지스터 체계의 일반화.
- 이제 재배치 레지스터 **relocation register**라고 하는 기본 레지스터
- 재배치 레지스터의 값은 사용자 프로세스가 메모리로 보낼 때 생성된 모든 주소에 추가.
- 사용자 프로그램은 논리 주소를 처리. 실제 물리적 주소를 볼 수 없다.
- 실행 시간 바인딩은 메모리의 위치를 참조할 때 발생.
- 물리적 주소에 바인딩된 논리적 주소





1. Background

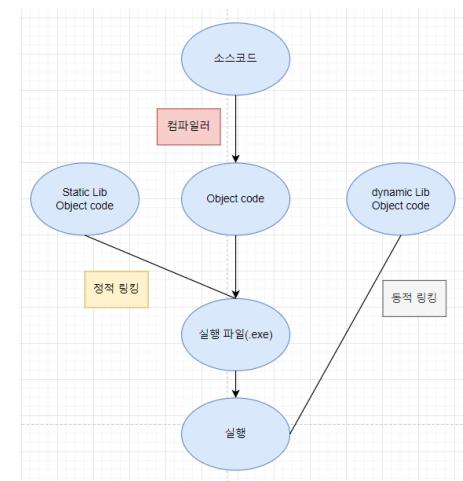
동적 로딩 Dynamic Loading

- 전체 프로그램을 실행하려면 메모리에 있어야 한다.
- 루틴은 호출될 때까지 로드되지 않는다.
- 더 나은 메모리 공간 활용; 사용하지 않는 루틴은 절대 로드되지 않다
- 재배치 가능한 로드 형식으로 디스크에 보관된 모든 루틴
- 드물게 발생하는 사례를 처리하기 위해 많은 양의 코드가 필요할 때 유용.
- 운영 체제의 특별한 지원이 필요하지 않다.
 - 프로그램 설계를 통해 구현
 - OS는 라이브러리를 제공하여 동적 로딩을 구현함으로써 도움을 줄 수 있다.

1. Background

동적 연결 Dynamic Linking

- 정적 연결 **Static linking** – 로더에 의해 바이너리 프로그램 이미지로 결합된 시스템 라이브러리 및 프로그램 코드
- 동적 연결 **Dynamic Linking** - 실행 시간까지 연기된 연결
- 적절한 메모리 상주 라이브러리 루틴을 찾는데 사용되는 작은 코드 조각, 스탑
- Stub은 자신을 루틴의 주소로 바꾸고 루틴을 실행합니다.
- 운영 체제는 루틴이 프로세스의 메모리 주소에 있는지 확인.
 - 주소 공간에 없으면 주소 공간에 추가
- 동적 연결은 라이브러리에 특히 유용.
- 공유 라이브러리 **shared libraries**
- 라고도 하는 시스템
- 패치 시스템 라이브러리에 대한 적용 가능성 고려
- 버전 관리가 필요할 수 있음





2. Contiguous Memory Allocation

연속 할당

- 메인 메모리는 OS와 사용자 프로세스를 모두 지원해야 한다.
- 제한된 리소스, 효율적으로 할당해야 함
- 연속 할당은 초기 방법 중 하나입니다.
- 메인 메모리는 일반적으로 두 개의 파티션으로 나뉜다.
 - 상주 운영 체제, 일반적으로 인터럽트 벡터와 함께 낮은 메모리에 보관됨
 - 그런 다음 사용자 프로세스는 높은 메모리에 보관됨.
 - 단일 연속 메모리 섹션에 포함된 각 프로세스



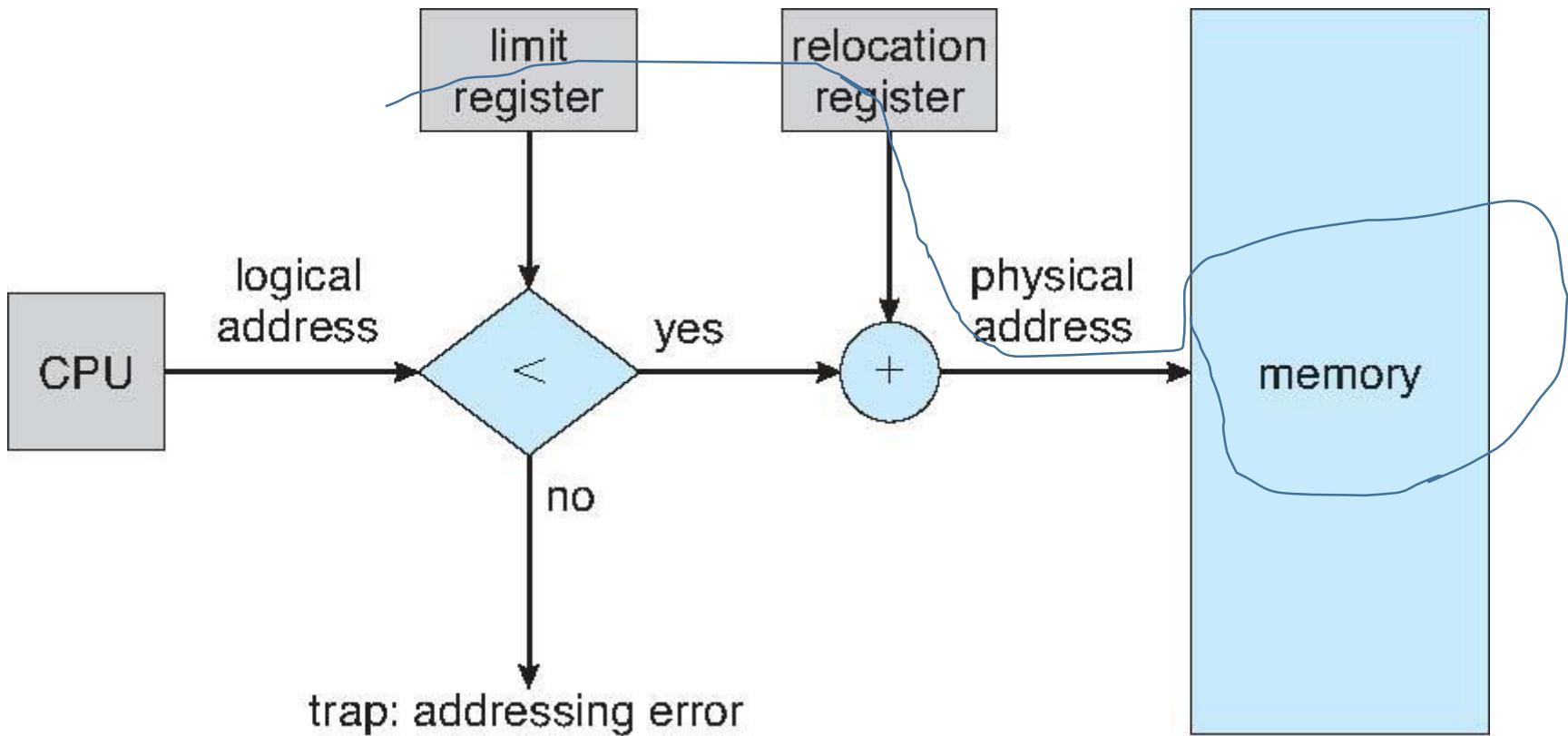
2. Contiguous Memory Allocation

연속 할당

- 사용자 프로세스를 서로 보호하고 운영 체제 코드 및 데이터 변경으로부터 보호하는데 사용되는 재배치 레지스터
 - 기본 레지스터에는 가장 작은 물리적 주소 값이 포함.
 - 제한 레지스터에는 논리 주소 범위가 포함됩니다. 각 논리 주소는 제한 레지스터보다 작아야 한다.
 - MMU는 논리 주소를 동적으로 매핑한다.
 - 그런 다음 커널 코드가 일시적이고 커널 크기가 변경되는 것과 같은 작업을 허용할 수 있다.

2. Contiguous Memory Allocation

재배치 및 제한 레지스터에 대한 하드웨어 지원





2. Contiguous Memory Allocation

가변 파티션

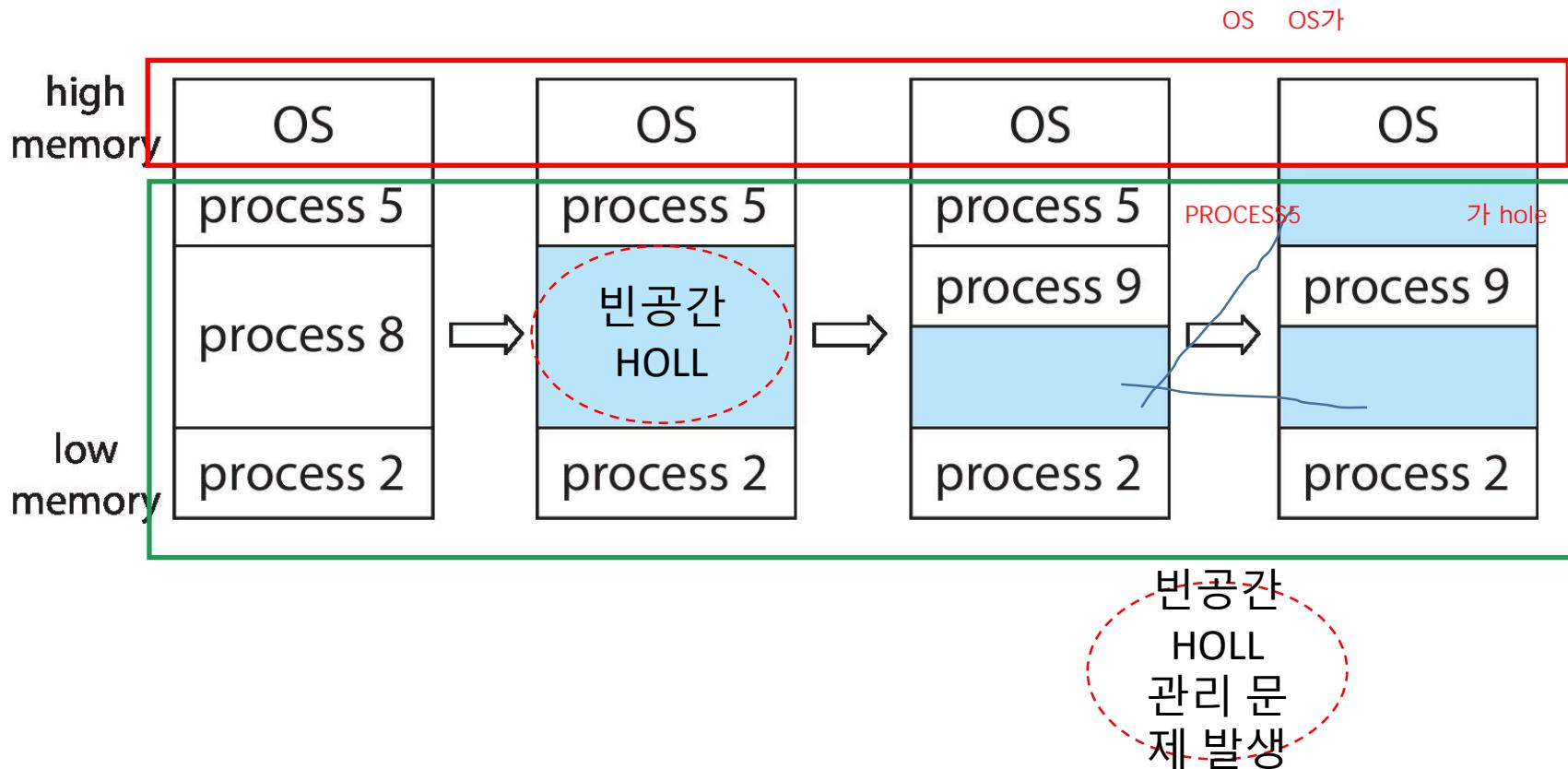
- **다중 파티션 할당**
 - 파티션 수에 의해 제한되는 다중 프로그래밍 정도
 - 효율성을 위한 가변 파티션 Variable-partition 크기(주어진 프로세스의 요구에 맞게 크기 조정)
 - 구멍 Hole : 빈공간 – 사용 가능한 메모리 블록 다양한 크기의 구멍이 메모리 전체에 흩어져 있다. >>
 - 프로세스가 도착하면 수용할 수 있을 만큼 큰 구멍에서 메모리가 할당.
 - 종료되는 프로세스는 파티션을 해제하고 인접한 여유 파티션을 결합.
 - 운영 체제는 다음에 대한 정보를 유지합니다.
 - a) 할당된 파티션 allocated partitions
 - b) 사용 가능한 파티션(구멍) free partitions (hole)

2. Contiguous Memory Allocation



가변 파티션

- 다중 파티션 할당





2. Contiguous Memory Allocation

동적 스토리지 할당 문제

- 빈 구멍 목록에서 크기 n 의 요청을 어떻게 충족시킬 수 있을까?
 - First-fit: 충분히 큰 첫 번째 구멍을 할당.
 - 최적 적합 Best-fit: 충분히 큰 가장 작은 구멍을 할당. 크기별로 정렬하지 않은 전체 목록을 검색해야 한다.
 - ✓ 남은 가장 작은 구멍 생성
 - 최악 적합 Worst-fit: 가장 큰 구멍을 할당. 또한 전체 목록을 검색.
 - ✓ 가장 큰 남은 구멍을 생성.
- 속도 및 스토리지 활용 측면에서 최악 적합보다 우선 적합 및 최적 적합이 더 우수함

2. Contiguous Memory Allocation



단편화 Fragmentation

30

10 10 10

- **외부 단편화 External Fragmentation** – 요청을 충족하기 위해 총 메모리 공간이 존재하지만 연속적이지 않다.
- **내부 단편화 Internal Fragmentation** – 할당된 메모리가 요청된 메모리보다 약간 클 수 있다. 이 크기 차이는 파티션 내부의 메모리 이지만 사용되지는 않다.
- First fit 분석 결과 N개의 블록이 할당된 경우 조각화로 인해 $0.5N$ 개의 블록이 손실되었음을 나타낸다.
- 1/3은 사용할 수 없음 -> 50% 규칙



2. Contiguous Memory Allocation

단편화 Fragmentation

- 압축으로 외부 조각화 감소

- 모든 여유 메모리를 하나의 큰 블록에 함께 배치하기 위해 메모리 내용을 섞는다.

- 압축은 재배치가 동적인 경우에만 가능하며 실행 시간에 수행.

- I/O 문제

- ✓ I/O에 관여하는 동안 메모리에 작업 래치
 - ✓ OS 버퍼로만 I/O 수행

- 이제 백업 저장소에 동일한 조각화 문제가 있음을 고려.



3. Paging

기본 개념

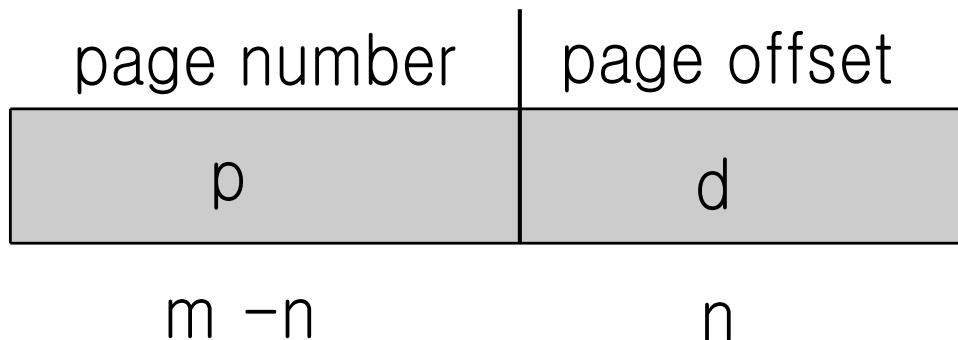
- 프로세스의 물리적 주소 공간은 비연속적일 수 있다. 후자가 사용 가능할 때마다 프로세스에 물리적 메모리가 할당한다.
- 외부 조각화 방지
- 다양한 크기의 메모리 청크 문제 방지
- 물리적 메모리를 프레임이라는 고정 크기 블록으로 나눈다.
- 크기는 512바이트에서 16MB 사이의 2의 거듭제곱.
- 논리 메모리를 페이지라고 하는 동일한 크기의 블록으로 나눕니다.
- 모든 무료 프레임 추적
- N 페이지 크기의 프로그램을 실행하려면 N 개의 여유 프레임을 찾아 프로그램을 로드해야 합니다.
- 논리 주소를 물리 주소로 변환하도록 페이지 테이블 page table 설정
- 백업 저장소도 마찬가지로 페이지로 분할.
- 여전히 내부 조각화가 있음



3. Paging

주소 변환 체계

- CPU에서 생성된 주소는 다음과 같이 나뉩니다.
 - 페이지 번호(p) Page number (p) – 물리적 메모리에 있는 각 페이지의 기본 주소를 포함하는 페이지 테이블에 대한 인덱스로 사용
 - 페이지 오프셋(d) Page offset (d) – 기본 주소와 결합하여 메모리 장치로 전송되는 물리적 메모리 주소를 정의.

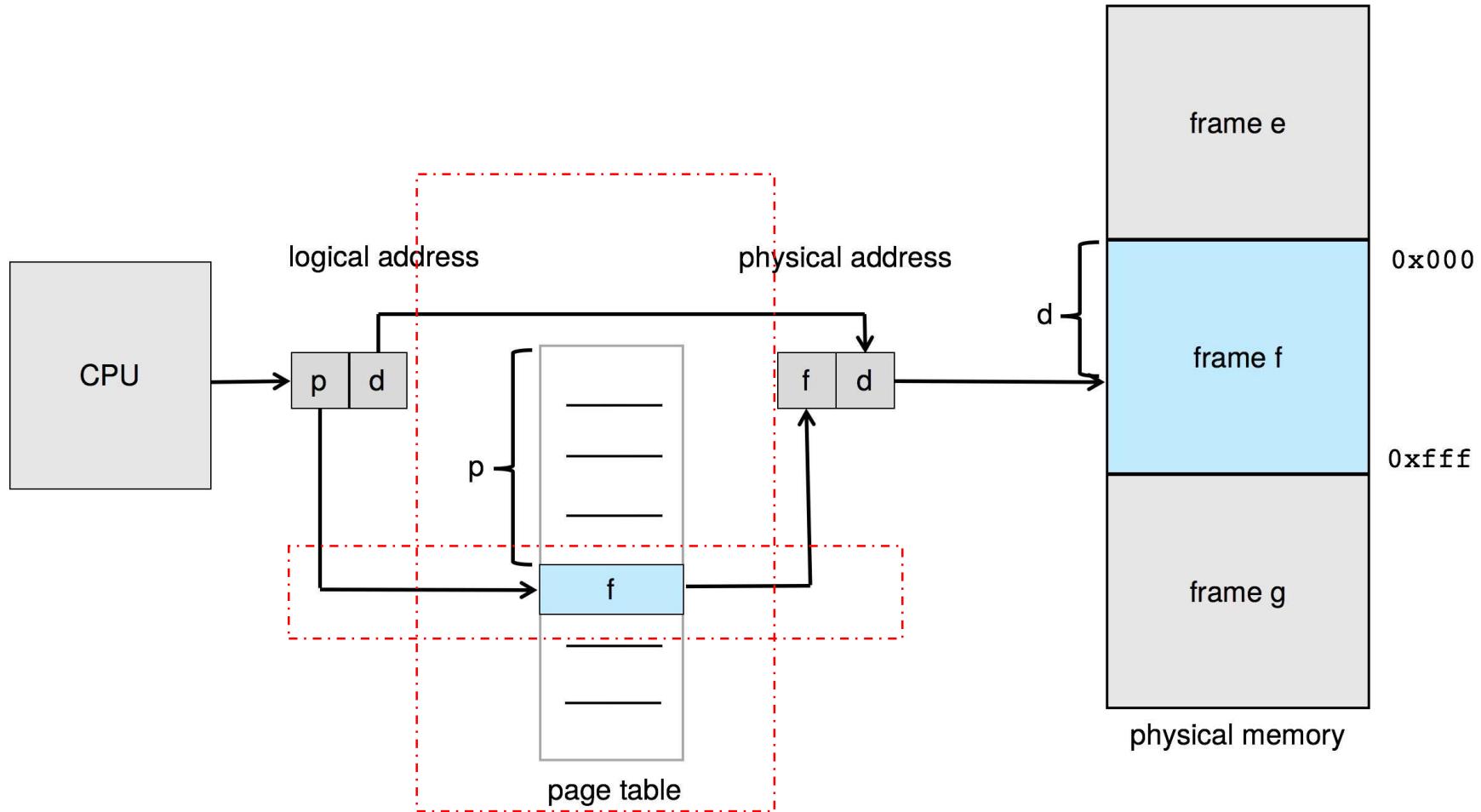


논리주소

- logical address space 2^m / page size 2^n

3. Paging

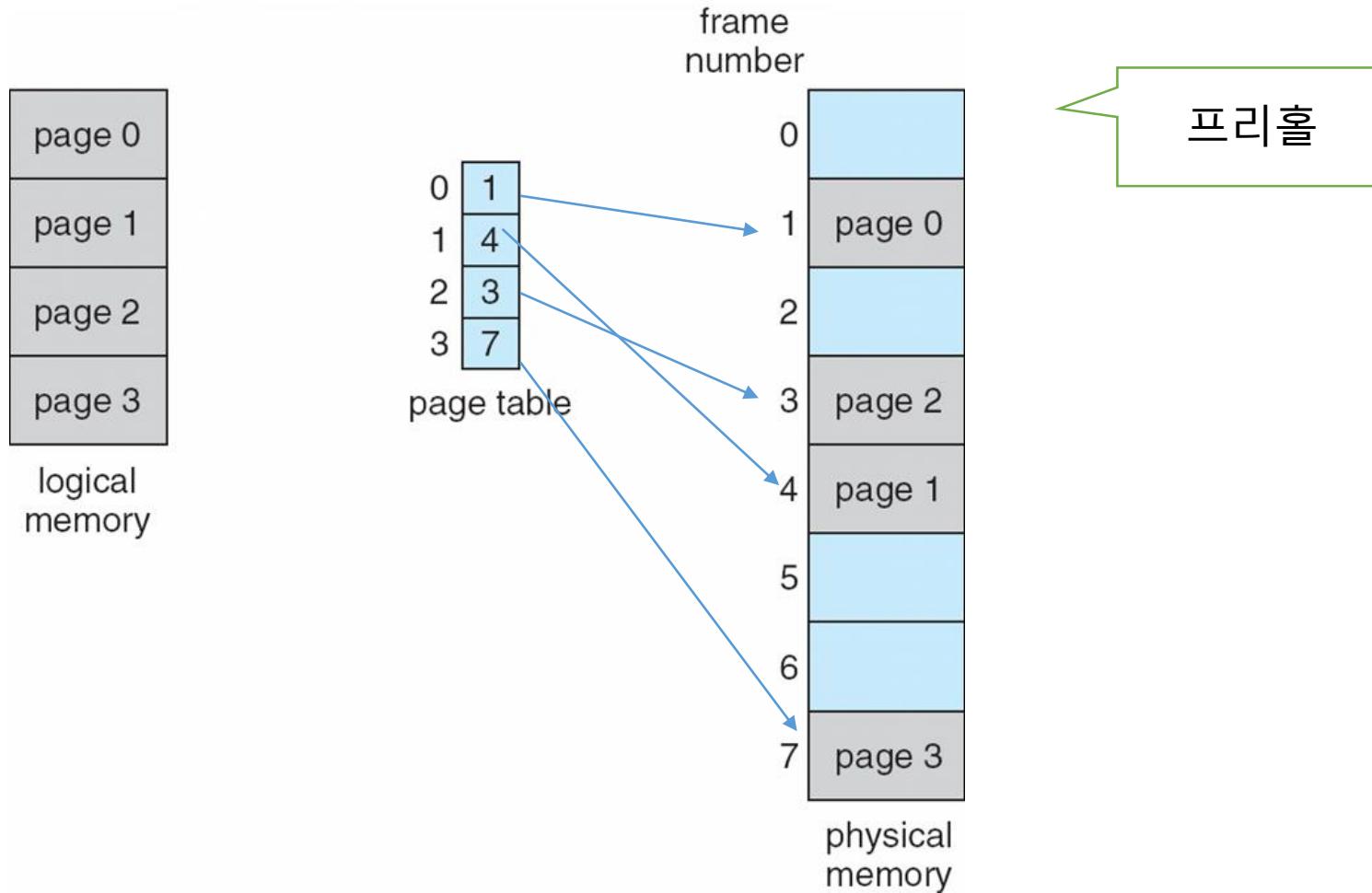
Paging Hardware



3. Paging

논리적 및 물리적 메모리의 페이징 모델

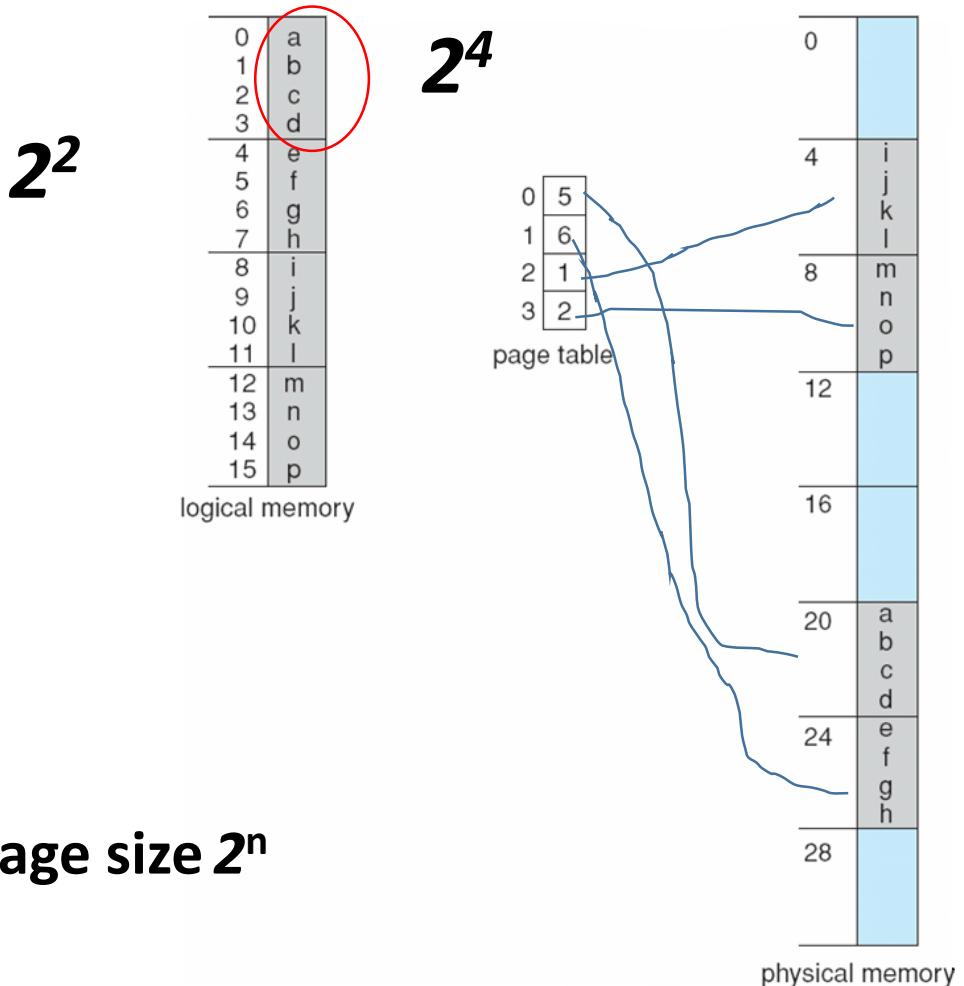
>>



3. Paging

Paging Example

- 논리 주소: $n = 2$ 및 $m = 4$. 페이지 크기 4바이트 및 물리적 메모리 32바이트 사용



- logical address space 2^m / page size 2^n



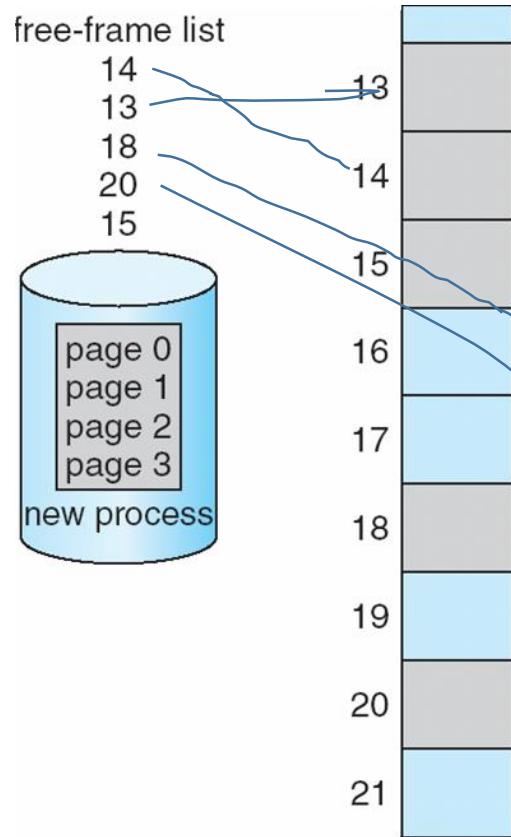
3. Paging

페이지 - 내부 단편화 계산

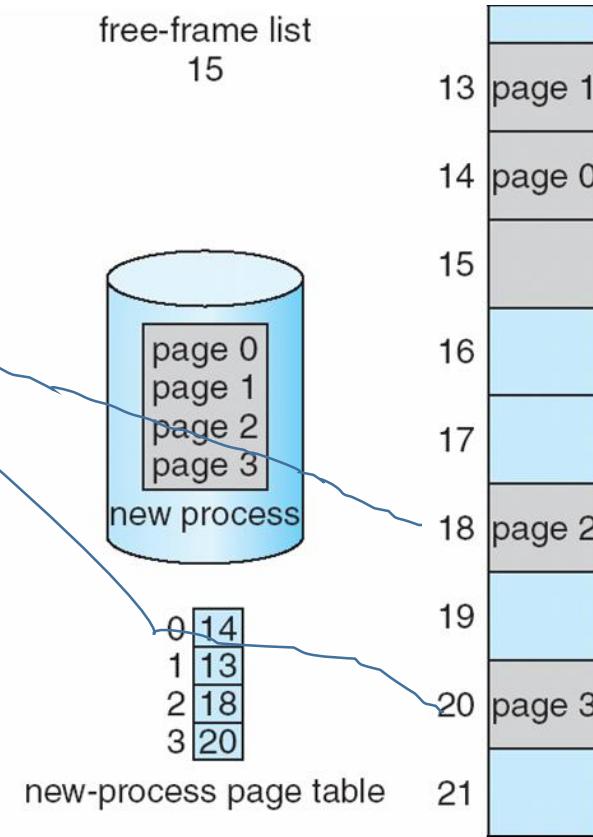
- 페이지 크기 Page size = 2,048바이트
- 프로세스 크기 Process size = 72,766바이트
- 35페이지 + 1,086바이트
- $2,048 - 1,086 = 962$ 바이트의 내부 단편화
- 최악의 단편화 = 1 프레임 - 1 바이트
- 평균 단편화 = 1/2 프레임 크기
- 작은 프레임 크기가 바람직합니까?
- 그러나 각 페이지 테이블 항목은 추적하는 데 메모리가 필요.
- 시간 경과에 따라 증가하는 페이지 크기
- Solaris는 8KB 및 4MB의 두 가지 페이지 크기를 지원.

3. Paging

Free Frames



(a)



(b)

Before allocation

After allocation



4. Structure of the Page Table

페이지 테이블 구현

- 페이지 테이블은 메인 메모리에 보관
 - 페이지 테이블 기준 레지스터(Page-table base register PTBR)는 페이지 테이블을 가리킨다.
 - 페이지 테이블 길이 레지스터(Page-table length register PTLR)는 페이지 테이블의 크기를 나타낸다.
- 이 체계에서 모든 데이터/명령 액세스에는 두 개의 메모리 액세스가 필요.
 - 하나는 페이지 테이블용이고 다른 하나는 데이터/명령어용.
- 2개 메모리 액세스 문제는 TLB(translation look-aside Buffers)(연관 메모리 associative memory 라고도 함)라고 하는 특수 고속 검색 하드웨어 캐시를 사용하여 해결할 수 있.



4. Structure of the Page Table

번역 참조 버퍼(TLB:Translation Look-Aside Buffer)

- 일부 TLB는 각 TLB 항목에 주소 공간 식별자 **address-space identifiers (ASIDs)** 를 저장. 각 프로세스를 고유하게 식별하여 해당 프로세스에 대한 주소 공간 보호를 제공.
 - 그렇지 않으면 모든 컨텍스트 스위치에서 플러시.
- 일반적으로 작은 TLB(64~1,024개 항목)
- TLB 미스 시 다음에 더 빠른 액세스를 위해 값이 TLB에 로드.
 - 교체 정책을 고려.
 - 영구적인 빠른 액세스를 위해 일부 항목을 연결할 수 있다.



4. Structure of the Page Table

하드웨어

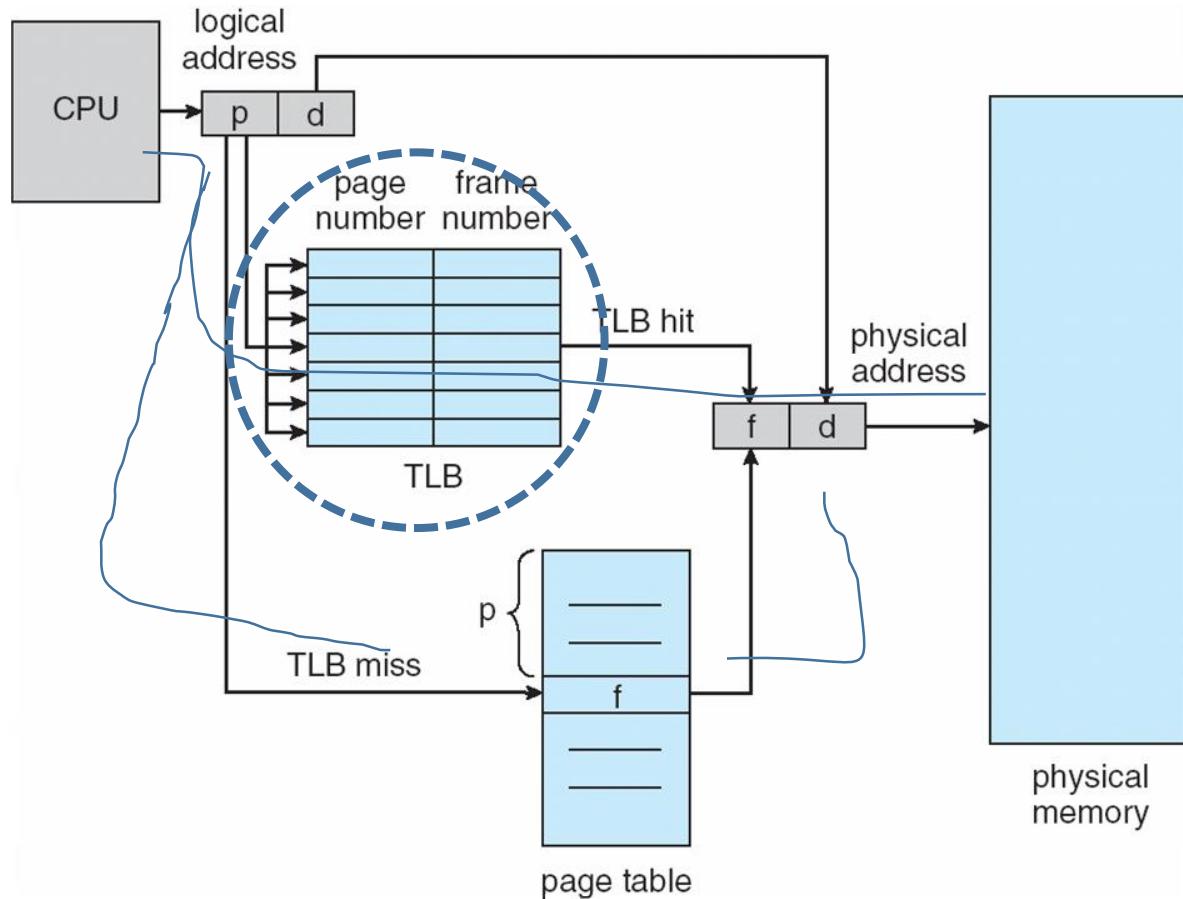
- 연상 메모리 **Associative memory** – 병렬 탐색

Page #	Frame #

- 주소 변환 **Address translation (p, d)**
 - p가 연관 레지스터에 있으면 프레임 #을 가져온다.
 - 그렇지 않으면 메모리의 페이지 테이블에서 프레임 번호를 가져온다.

4. Structure of the Page Table

TLB를 사용한 페이지 하드웨어





4. Structure of the Page Table

유효 액세스 시간 Effective Access Time

- 적중률 Hit ratio – TLB에서 페이지 번호가 발견된 횟수의 백분율
- 적중률이 80%라는 것은 TLB에서 80%의 시간 동안 원하는 페이지 번호를 찾았다는 것을 의미.
- 메모리에 액세스하는 데 10ns가 걸린다고 가정.
 - TLB에서 원하는 페이지를 찾으면 매핑된 메모리 액세스에 10ns가 걸립니다.
 - 그렇지 않으면 두 개의 메모리 액세스가 필요하므로 20ns.
- Effective Access Time ((EAT))
 $EAT = 0.80 \times 10 + 0.20 \times 20 = 12\text{ns}$
액세스 시간이 20% 느려짐을 의미
- 99%라는 현실적인 적중률을 고려하면,
 $EAT = 0.99 \times 10 + 0.01 \times 20 = 10.1\text{ns}$
액세스 시간이 1%만 느려진다는 것을 의미.



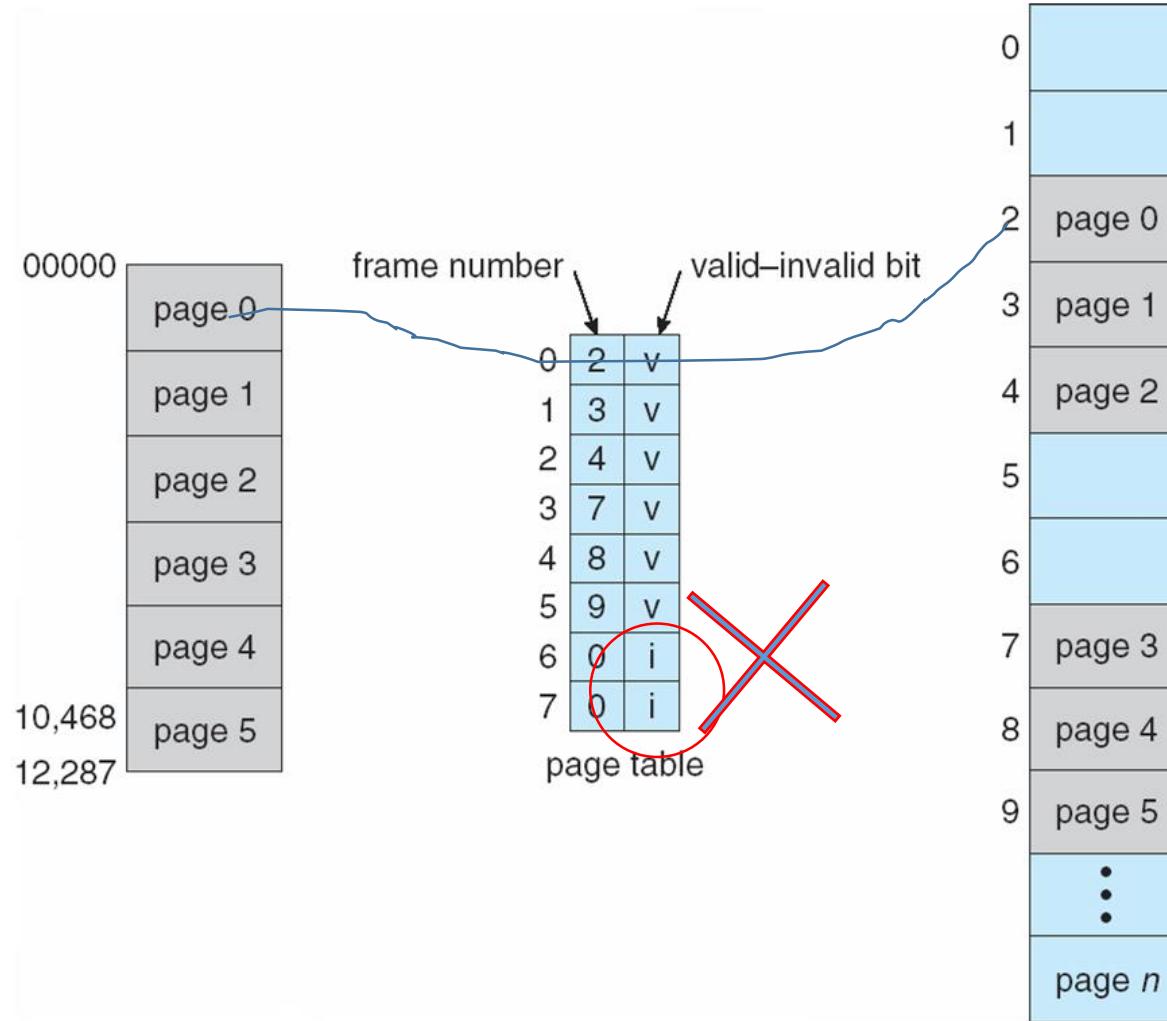
4. Structure of the Page Table

Memory Protection

- 읽기 전용 또는 읽기-쓰기 액세스가 허용되는지 표시하기 위해 보호 비트를 각 프레임과 연결하여 구현되는 메모리 보호
- 또한 페이지 실행 전용 등을 나타내기 위해 더 많은 비트를 추가할 수 있다.
- 페이지 테이블의 각 항목에 첨부된 유효-무효 Valid-invalid 비트:
 - "valid"는 관련 페이지가 프로세스의 논리 주소 공간에 있으므로 합법적인 페이지임을 나타낸다.
 - "무효 invalid"는 페이지가 프로세스의 논리 주소 공간에 없음을 나타낸다.
 - 또는 페이지 테이블 길이 레지스터(PTLR) 사용
 - 모든 위반으로 인해 커널에 트랩이 발생.

4. Structure of the Page Table

Valid (v) or Invalid (i) Bit In A Page Table





4. Structure of the Page Table

Shared Pages

- 공유 코드 **Shared code**

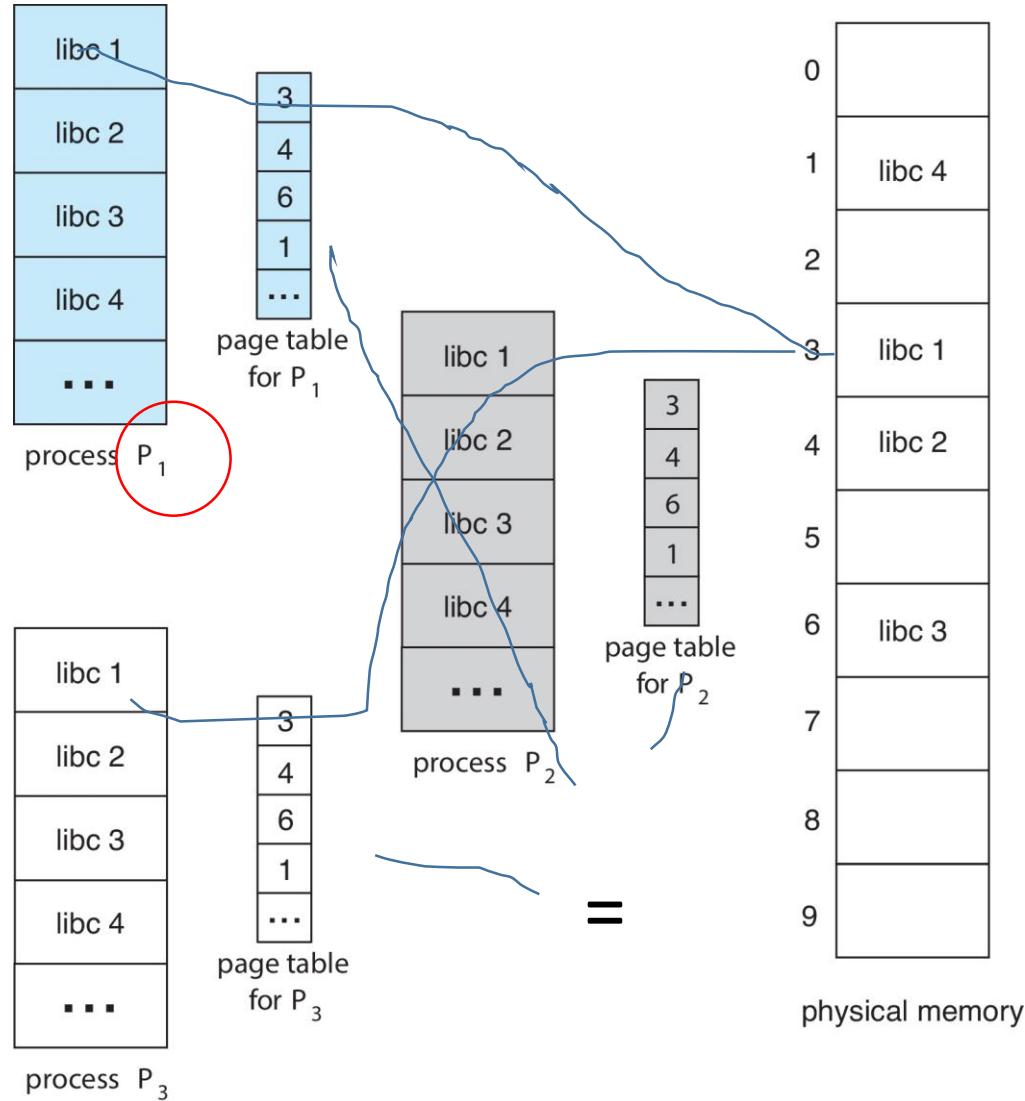
- 프로세스(예: 텍스트 편집기, 컴파일러, 윈도우 시스템) 간에 공유되는 읽기 전용(재진입 reentrant) 코드 사본 1개
- 동일한 프로세스 공간을 공유하는 여러 스레드와 유사
- 읽기-쓰기 페이지 공유가 허용되는 경우 프로세스 간 통신에도 유용.

- 비공개 코드 및 데이터 **Private code and data**

- 각 프로세스는 코드와 데이터의 별도 복사본을 유지합니다.
- 개인 코드 및 데이터에 대한 페이지는 논리 주소 공간의 어디에나 나타날 수 있다.

4. Structure of the Page Table

Shared Pages Example





4. Structure of the Page Table

페이지 테이블의 구조

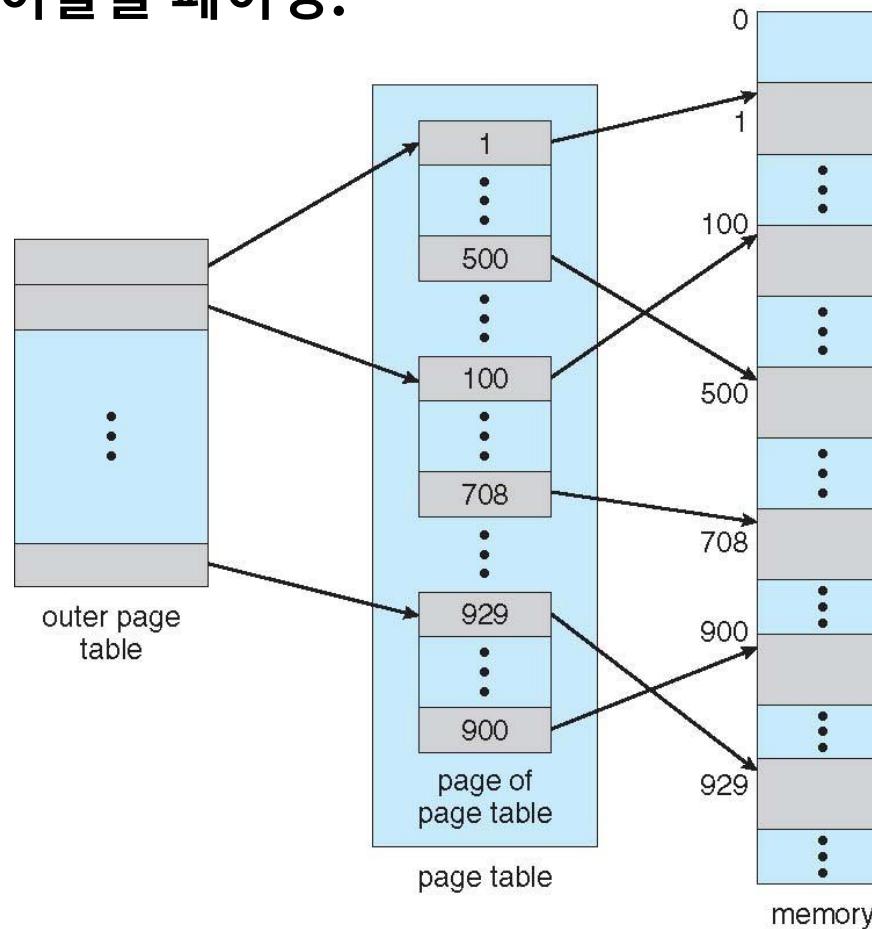
- 간단한 방법을 사용하면 페이징을 위한 메모리 구조가 커질 수 있다.
 - 최신 컴퓨터에서와 같이 32비트 논리 주소 공간을 고려.
 - 페이지 크기 4KB(2^{12})
 - 페이지 테이블에는 100만 항목($2^{32} / 2^{12}$)이 있다.
 - 각 항목이 4바이트인 경우 → 각 프로세스는 페이지 테이블만을 위한 4MB의 물리적 주소 공간
 - 메인 메모리에 연속적으로 할당하고 싶지 않다.
 - 한 가지 간단한 해결책은 페이지 테이블을 더 작은 단위로 나누는 것.
 - ① 계층적 페이징Hierarchical Paging
 - ② 해시된 페이지 테이블Hashed Page Tables
 - ③ 역 페이지 테이블Inverted Page Tables



4. Structure of the Page Table

Hierarchical Page Tables

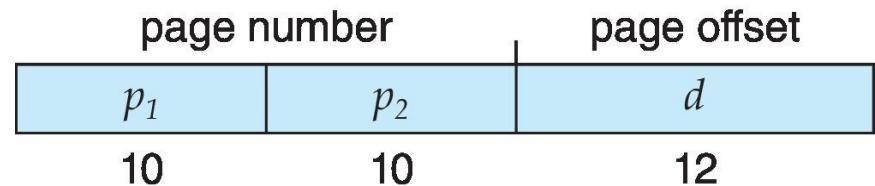
- 논리 주소 공간을 여러 페이지 테이블로 나눕니다.
- 간단한 기술은 2단계 페이지 테이블.
- 그런 다음 페이지 테이블을 페이징.



4. Structure of the Page Table

2단계 페이지 예

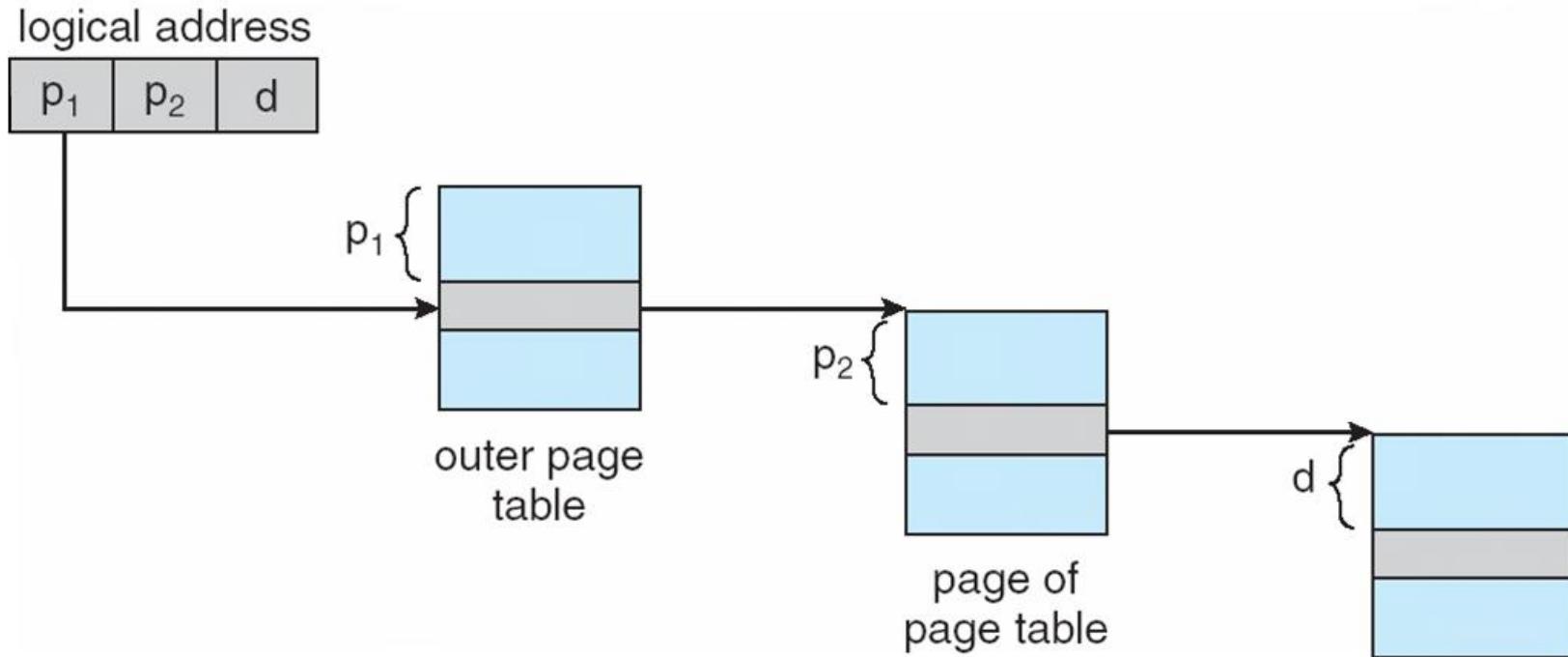
- 논리 주소(페이지 크기가 4K인 32비트 시스템에서)는 다음과 같이 나뉜다.
 - 20비트로 구성된 페이지 번호
 - 12비트로 구성된 페이지 오프셋
- 페이지 테이블이 페이지징되므로 페이지 번호는 다음과 같이 더 나뉜다.
 - 10비트 페이지 번호
 - 10비트 페이지 오프셋
 - 따라서 논리 주소는 다음과 같다.



- 여기서 p_1 은 외부 페이지 테이블에 대한 인덱스이고 p_2 는 내부 페이지 테이블의 페이지 내 범위.
- 정방향 매핑 페이지 테이블forward-mapped page table로 알려짐

4. Structure of the Page Table

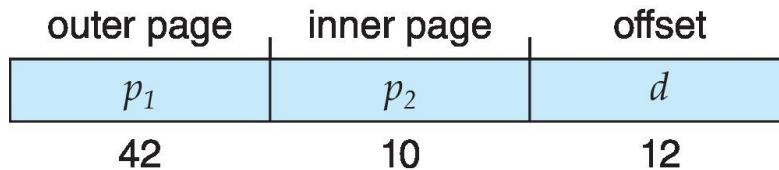
주소 변환 체계



4. Structure of the Page Table

64비트 논리 주소 공간 Page Tables

- 2단계 페이징 체계도 충분하지 않음
 - 페이지 크기가 4KB인 경우(2^{12})
 - 그런 다음 페이지 테이블에는 2^{52} 개의 항목이 있다.
 - 2단계 체계인 경우 내부 페이지 테이블은 2^{10} 개의 4바이트 항목이 될 수 있습.
 - 주소는 다음과 같다.

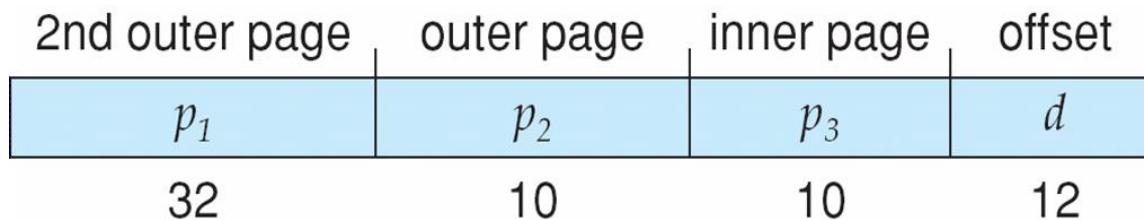
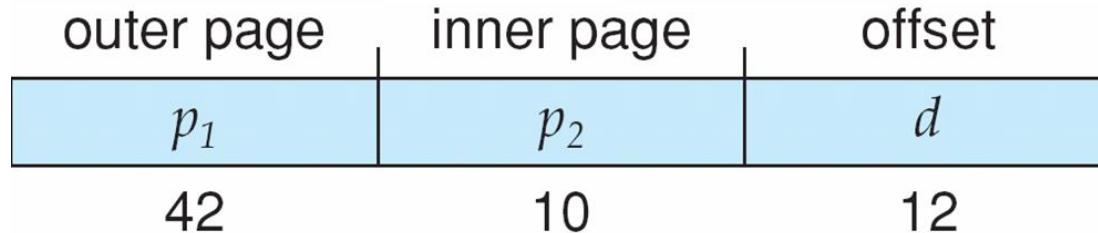


- 외부 페이지 테이블에는 2^{42} 개 항목 또는 2⁴⁴ 바이트가 있다.
- 한 가지 해결책은 두 번째 외부 페이지 테이블을 추가하는 것.
- 그러나 다음 예에서 두 번째 외부 페이지 테이블의 크기는 여전히 2^{34} 바이트다.
- 그리고 하나의 물리적 메모리 위치에 도달하기 위해 가능한 4개의 메모리 액세스



4. Structure of the Page Table

Three-level Paging Scheme





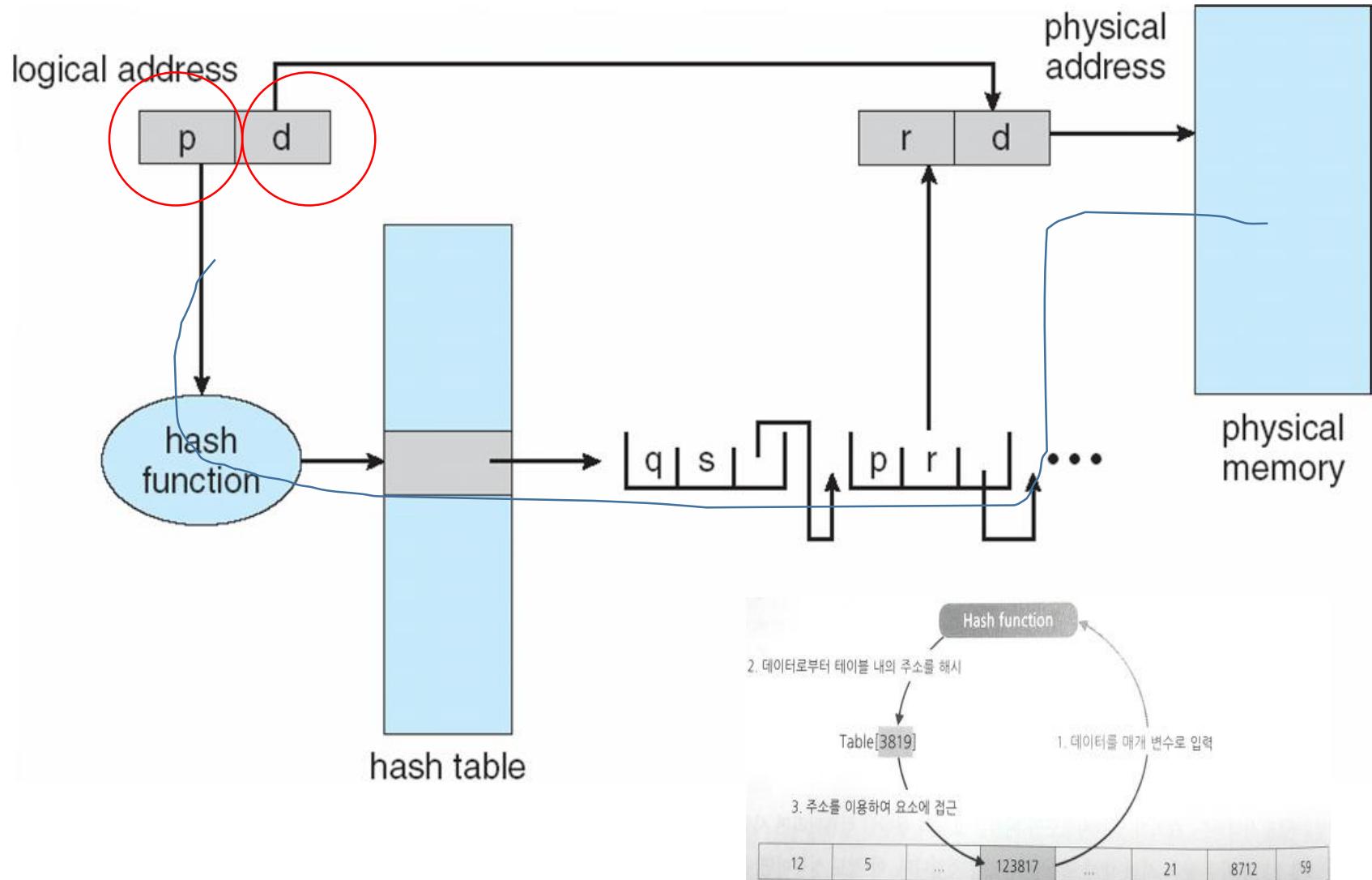
4. Structure of the Page Table

Hashed Page Tables

- 주소 공간 > 32비트에서 공통
- 가상 페이지 번호는 페이지 테이블로 해시.
 - 이 페이지 테이블에는 동일한 위치로 해싱되는 요소 체인이 포함 있다.
- 각 요소에는 (1) 가상 페이지 번호 (2) 매핑된 페이지 프레임의 값 (3) 다음 요소에 대한 포인터가 포함.
- 일치를 검색하는 이 체인에서 가상 페이지 번호가 비교.
 - 일치하는 항목이 발견되면 해당 물리적 프레임이 추출.
- 64비트 주소의 변형은 클러스터링된 페이지 테이블.
- 해시와 유사하지만 각 항목은 1이 아닌 여러 페이지(예: 16)를 참조.
- 스파스 주소 공간(메모리 참조가 비연속적이고 흩어져 있는 경우)에 특히 유용.

4. Structure of the Page Table

Hashed Page Tables





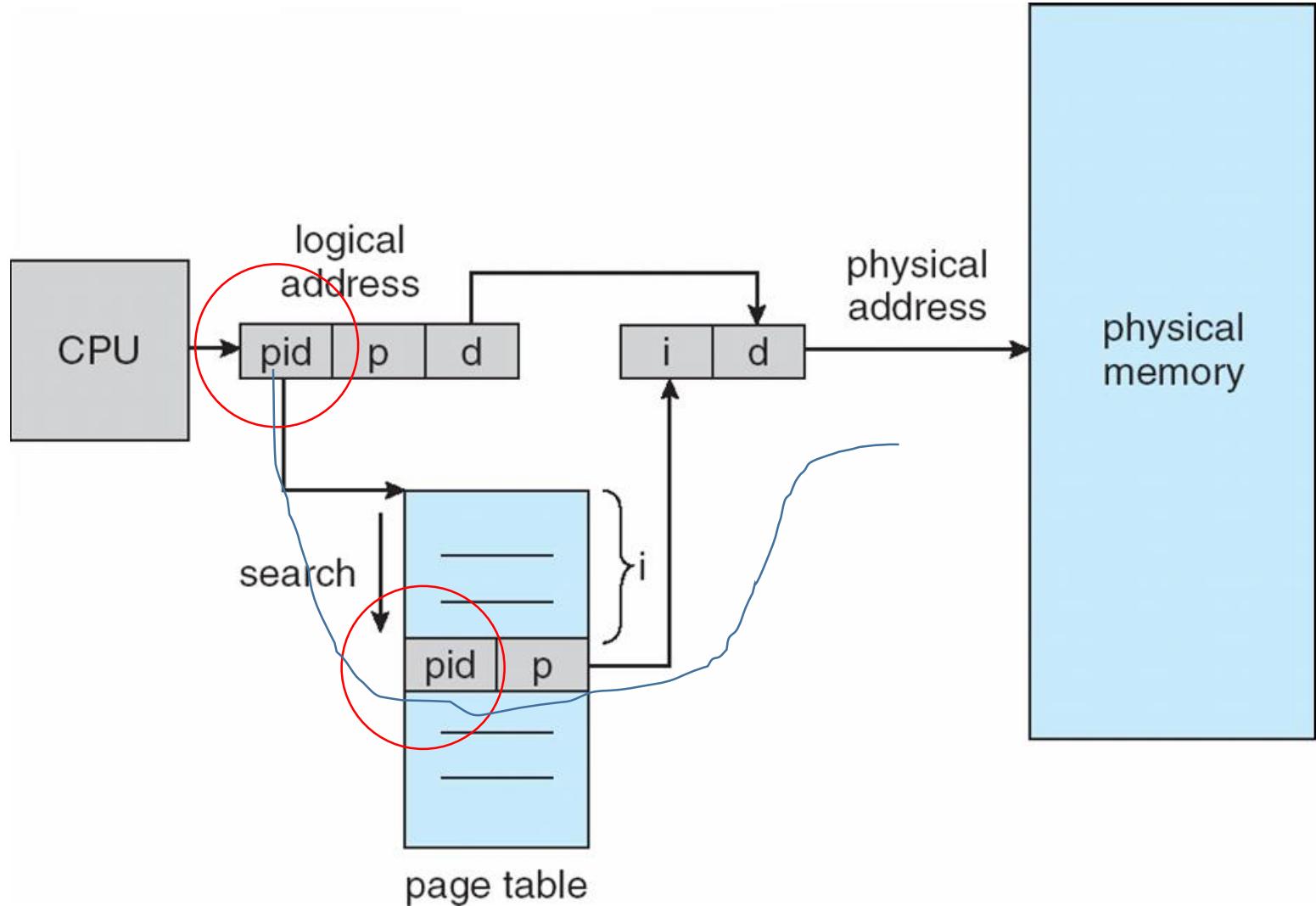
4. Structure of the Page Table

Inverted Page Table

- 반전된 페이지 테이블
- 각 프로세스가 페이지 테이블을 갖고 가능한 모든 논리적 페이지를 추적하는 대신 모든 물리적 페이지를 추적.
- 메모리의 각 실제 페이지에 대한 하나의 항목
- 항목은 해당 페이지를 소유하는 프로세스에 대한 정보와 함께 해당 실제 메모리 위치에 저장된 페이지의 가상 주소로 구성.
- 각 페이지 테이블을 저장하는 데 필요한 메모리는 줄어들지만 페이지 참조가 발생할 때 테이블을 검색하는 데 필요한 시간이 늘어납니다.
- 해시 테이블을 사용하여 검색을 하나 또는 최대 몇 개 페이이 테이블 항목으로 제한
 - TLB는 액세스를 가속화할 수 있다.
 - 그러나 공유 메모리를 구현하는 방법은 무엇일까?
 - 공유된 물리적 주소에 대한 가상 주소의 매핑 하나

4. Structure of the Page Table

Inverted Page Table





5. Swapping

기본 개념

- 프로세스는 일시적으로 메모리에서 백업 저장소로 교체된 다음 계속 실행을 위해 다시 메모리로 가져올 수 있다.
- 프로세스의 총 실제 메모리 공간은 실제 메모리를 초과할 수 있다.
- 백업 저장소 **Backing store** – 모든 사용자의 모든 메모리 이미지 복사본을 수용할 수 있을 만큼 충분히 큰 빠른 디스크. 이러한 메모리 이미지에 대한 직접 액세스를 제공.
- 롤아웃, 롤인 **Roll out, roll in** – 우선 순위 기반 스케줄링 알고리즘에 사용되는 스왑 변형. 우선 순위가 낮은 프로세스가 스왑 아웃되어 우선 순위가 높은 프로세스가 로드되고 실행될 수 있.
- 스왑 시간의 대부분은 전송 시간입니다. 총 전송 시간은 스왑되는 메모리 양에 정비례.
- 시스템은 디스크에 메모리 이미지가 있는 실행할 준비가 된 프로세스의 준비 대기열을 유지.



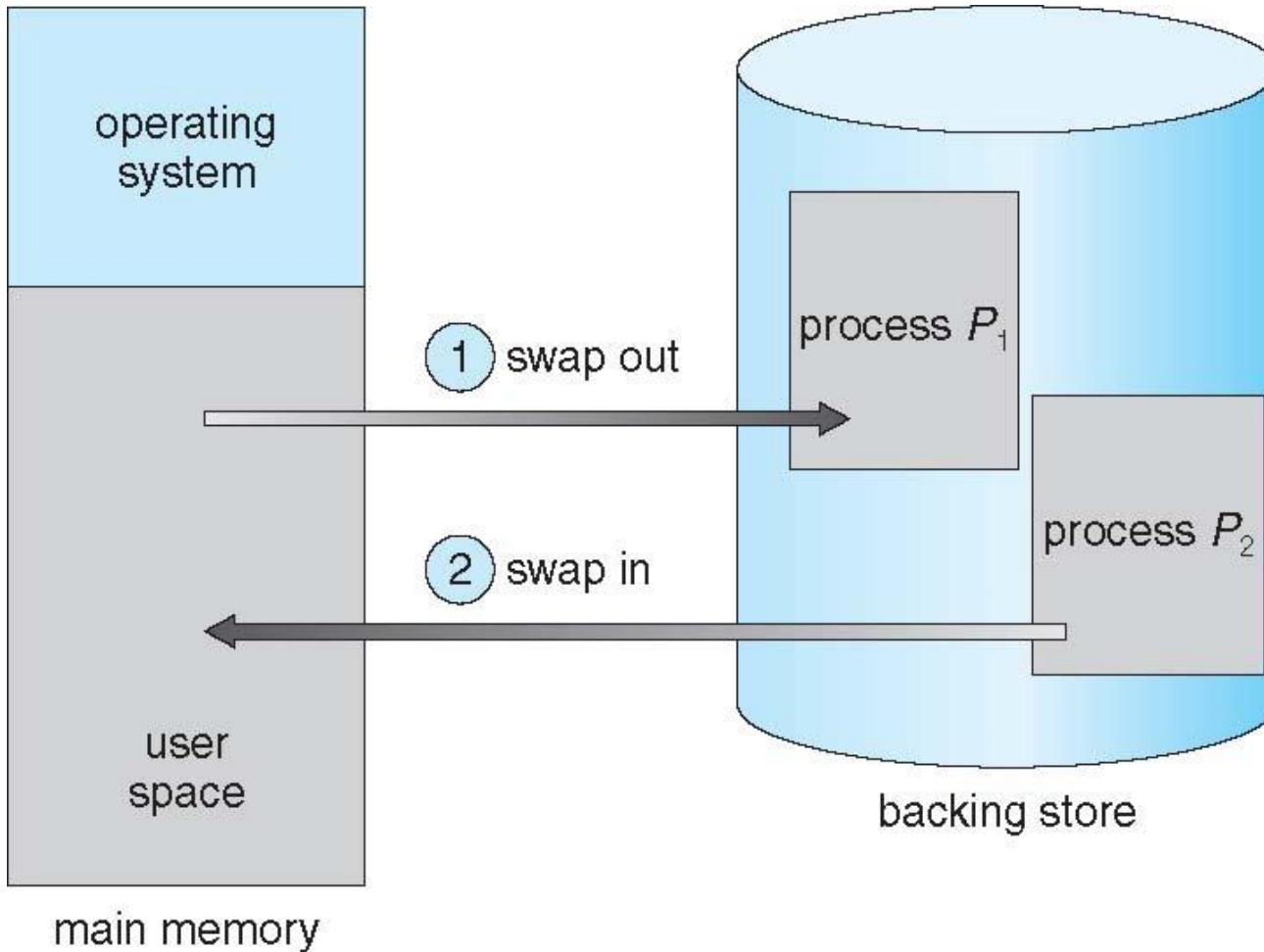
5. Swapping

기본 개념

- 교체된 프로세스가 동일한 물리적 주소로 다시 교체되어야 할까?
- 주소 바인딩 방법에 따라 다름
- 또한 프로세스 메모리 공간으로/에서 대기 중인 I/O를 고려.
- 수정된 버전의 스와핑은 많은 시스템(즉, UNIX, Linux 및 Windows)에서 발견.
- 스와핑은 일반적으로 비활성화됨
- 할당된 메모리의 임계값보다 많은 경우 시작됨
- 메모리 요구량이 임계값 아래로 감소하면 다시 비활성화됨

5. Swapping

스와핑의 개략도





5. Swapping

스와핑을 포함한 컨텍스트 전환 시간

- CPU에 올릴 다음 프로세스가 메모리에 없으면 프로세스를 교체하고 대상 프로세스에서 교체해야 함.
- 그러면 컨텍스트 전환 시간이 매우 길어질 수 있다.
- 전송 속도가 50MB/초인 하드 디스크로 100MB 프로세스 스와핑
 - 2000ms의 스왑 아웃 시간
 - 또한 동일한 크기의 프로세스에서 스왑
 - 총 컨텍스트 스위치 스와핑 구성 요소 시간 4000ms(4초)
- 스왑되는 메모리의 크기를 줄이면 줄일 수 있습니다. 실제로 사용되는 메모리 양을 알면 됩니다.
- `request_memory()` 및 `release_memory()`를 통해 OS에 메모리 사용을 알리는 시스템 호출



5. Swapping

스와핑을 포함한 컨텍스트 전환 시간

- 스와핑에 대한 기타 제약 사항

- Pending I/O - 잘못된 프로세스에 I/O가 발생하므로 교체할 수 없다.
- 또는 항상 I/O를 커널 공간으로 전송한 다음 I/O 장치로 전송

- 이중 버퍼링 double buffering 으로 알려진 오버헤드 추가

- 최신 운영 체제에서 사용되지 않는 표준 스와핑

- 그러나 수정된 버전 공통

- 사용 가능한 메모리가 매우 부족한 경우에만 스왑



5. Swapping

모바일 시스템에서 스와핑

- 일반적으로 지원되지 않음
 - 플래시 메모리 기반
 - 소량의 공간
 - 제한된 수의 쓰기 주기
 - 모바일 플랫폼에서 플래시 메모리와 CPU 사이의 낮은 처리량
- 대신 메모리가 부족한 경우 다른 방법을 사용하여 메모리를 확보.
 - iOS는 앱에 할당된 메모리를 자발적으로 양도하도록 요청.
 - 읽기 전용 데이터가 삭제되고 필요한 경우 플래시에서 다시 로드됨
 - 해제하지 않으면 종료될 수 있다.
 - Android는 사용 가능한 메모리가 부족하면 앱을 종료하지만 빠른 재시작을 위해 먼저 애플리케이션 상태를 플래시에 씁니다.
 - 두 OS 모두 아래에서 설명하는 페이징 paging 을 지원.



5. Swapping

Swapping with Paging

