

# Chapter 6. Concurrency: Deadlock and Starvation

6.1 교착상태 원리

6.2 교착상태 예방

6.3 교착상태 회피

6.4 교착상태 발견

6.5 통합 교착상태 전략

6.6 식사하는 철학자 문제

6.7 사례연구

유닉스 병행성 기법

리눅스 커널 병행성 기법

솔라리스 스레드 동기화 프리미티브

윈도우즈 병행성 기법

Android의 프로세스 간 통신

## 6장의 강의 목표

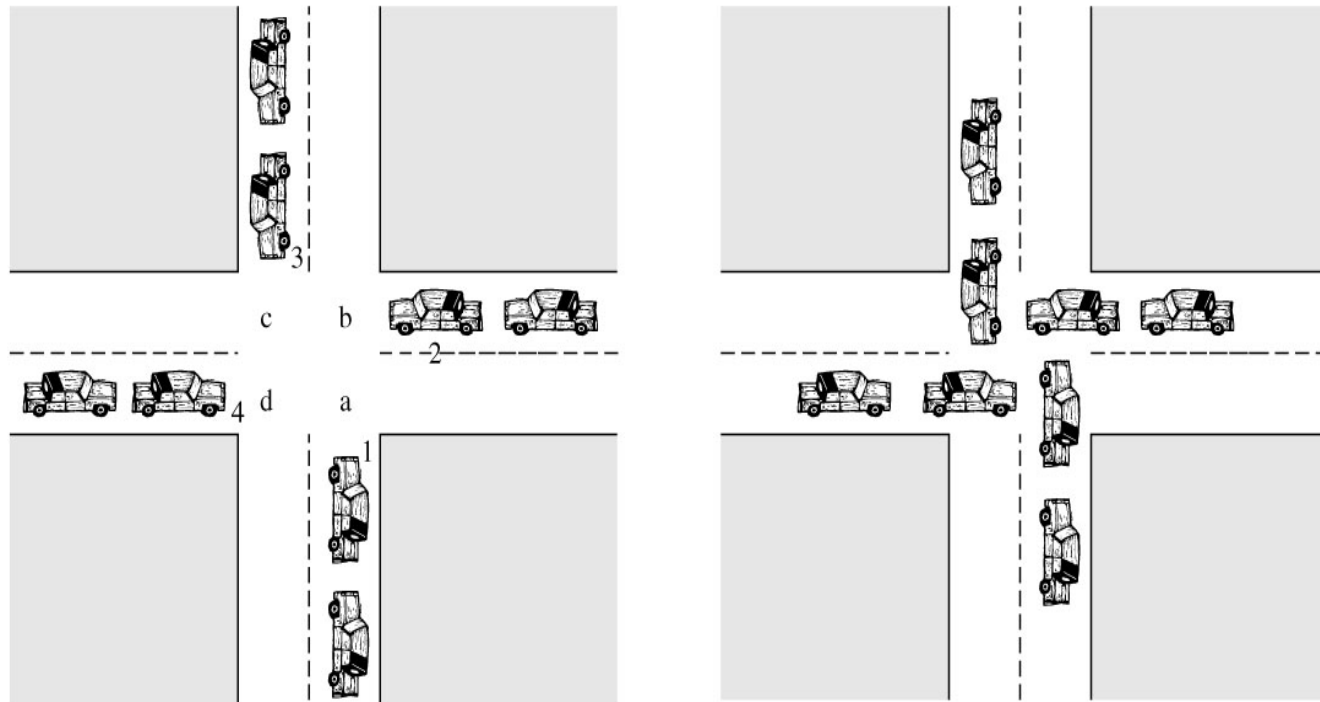
---

- 교착상태(deadlock)의 원리를 이해한다.
- 교착상태에 자원할당 그래프가 어떻게 이용되는지 이해한다.
- 교착상태가 발생하기 위한 필요.충분 조건을 이해한다.
- 교착상태 예방 기법들을 이해한다.
- 교착상태 회피 기법들을 이해한다.
- 교착상태의 발견과 복구 기법들을 이해한다.
- 식사하는 철학자 문제를 이해하고 해결 방법을 이해한다.
- UNIX, LINUX, Solaris, Windows 운영체제에서 제공하는 병행성 기법들을 이해한다.

## 6.1 교착상태 원리

### ■ 교착상태 정의

- ✓ 영속적인 블록 상태
- ✓ 2개 이상의 프로세스들이 공유 자원에 대한 경쟁이나 통신 중에 발생 (다른 블록된 프로세스의 자원을 기다리면서 자신도 블록)



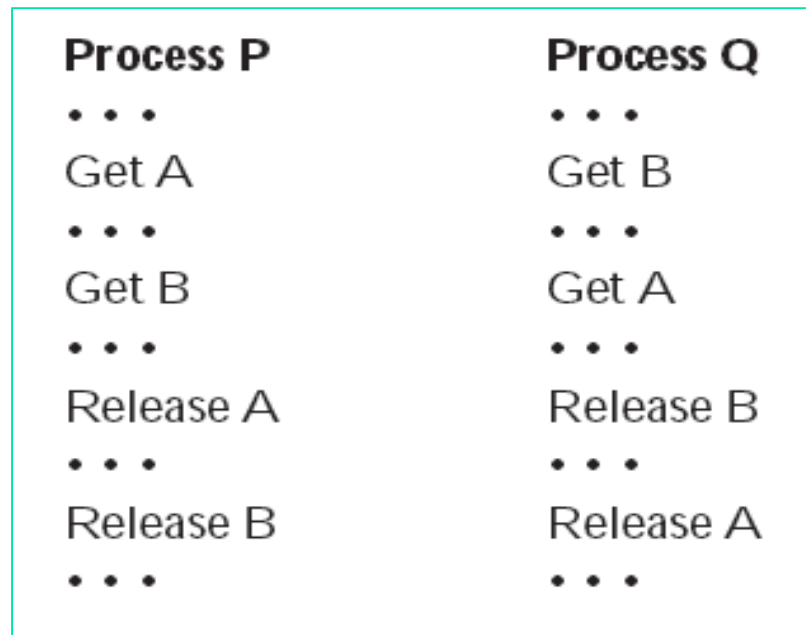
(a) 교착상태 가능성 존재

(b) 교착상태 발생

# 교착상태 원리

## ■ 결합 진행 다이어그램

- ✓ Lets look at this with two processes P and Q
- ✓ Each needing exclusive access to a resource A and B for a period of time



# 교착상태 원리

## ■ 결합 진행 다이어그램

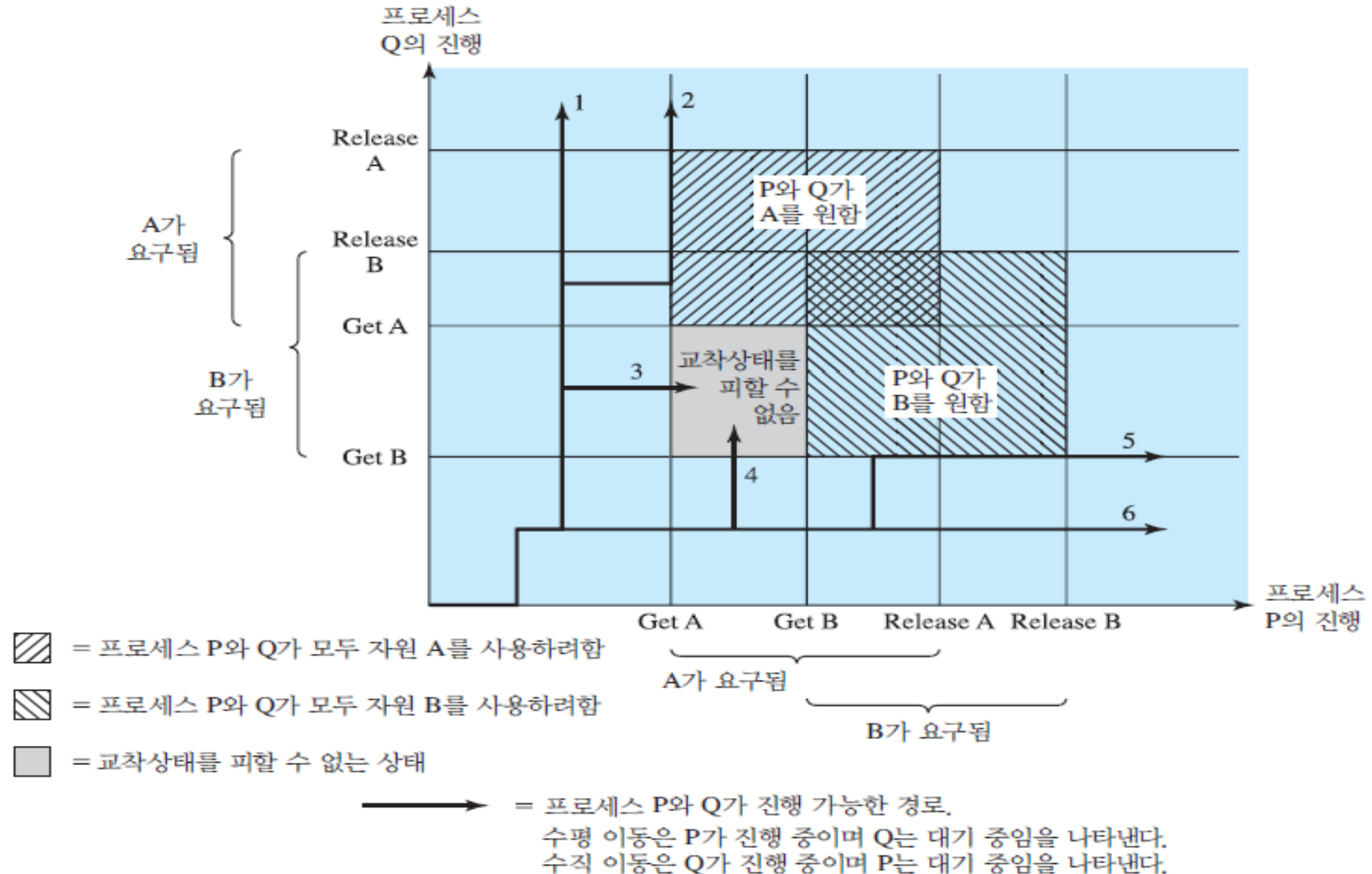


그림 6.2 교착상태 예

# 교착상태 원리

## ■ 결합 진행 다이어그램: alternative

Process P

...

Get A

...

Release A

...

Get B

...

Release B

...

Process Q

...

Get B

...

Get A

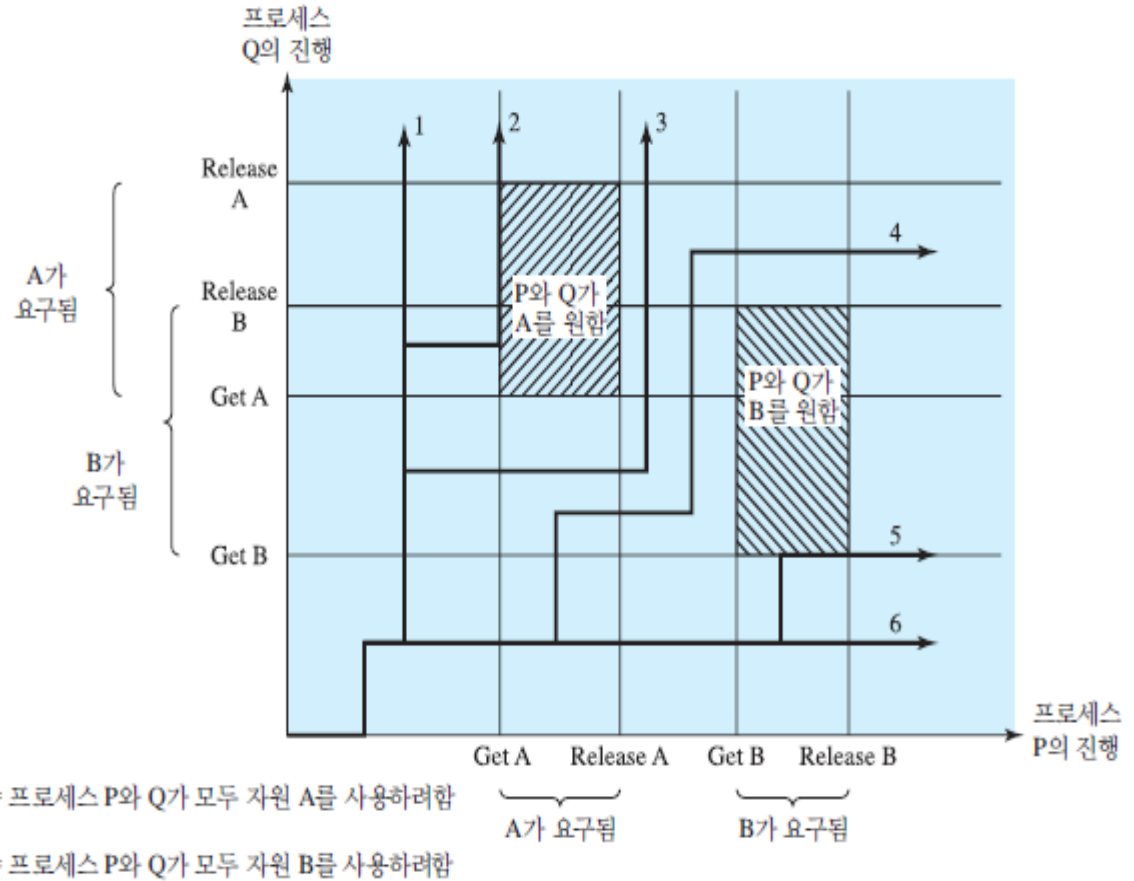
...

Release B

...

Release A

...



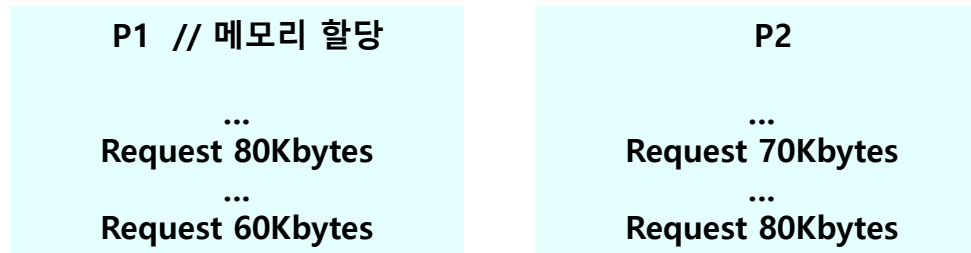
→ = 프로세스 P와 Q의 진행 가능한 경로.  
수평 이동은 P가 진행 중이며 Q는 대기 중임을 나타낸다.  
수직 이동은 Q가 진행 중이며 P는 대기 중임을 나타낸다.

그림 6.3 교착상태가 발생하지 않는 예 [BACO03]

# 교착상태 원리

## ■ 교착상태 다른 예들

- ✓ 메모리 할당 (가용 메모리 크기 = 200Kbytes)



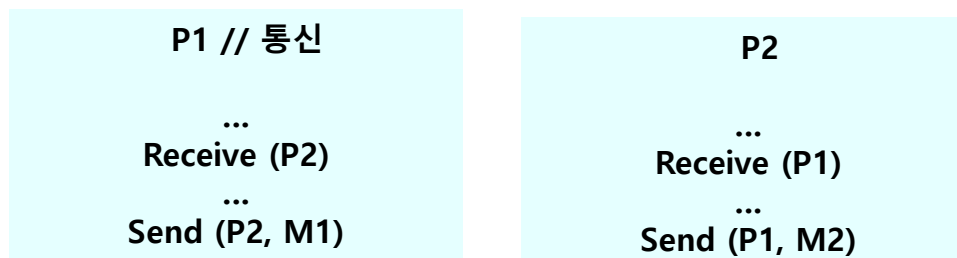
- ✓ 디스크와 테이프를 이용한 백업

Process P		Process Q	
Step	Action	Step	Action
p <sub>0</sub>	Request (D)	q <sub>0</sub>	Request (T)
p <sub>1</sub>	Lock (D)	q <sub>1</sub>	Lock (T)
p <sub>2</sub>	Request (T)	q <sub>2</sub>	Request (D)
p <sub>3</sub>	Lock (T)	q <sub>3</sub>	Lock (D)
p <sub>4</sub>	Perform function	q <sub>4</sub>	Perform function
p <sub>5</sub>	Unlock (D)	q <sub>5</sub>	Unlock (T)
p <sub>6</sub>	Unlock (T)	q <sub>6</sub>	Unlock (D)

**Deadlock:**  
P0-P1-q0-q1-  
p2-q2

- ✓ 통신

Figure 6.4 Example of Two Processes Competing for Reusable Resources



**Receive** 함수가 블럭킹 타입  
(수신 프로세스가 메시지가 올  
때까지 대기하는 유형) 이면  
**deadlock** 발생



# 교착상태 원리

## ■ 자원 종류

### ✓ 재사용 가능 자원 (reusable)

- can be safely used by only one process at a time and ***is not depleted*** by that use.
- 예: Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other (메모리 할당, 디스크와 테이프 사례)

### ✓ 소모성 자원 (consumable)

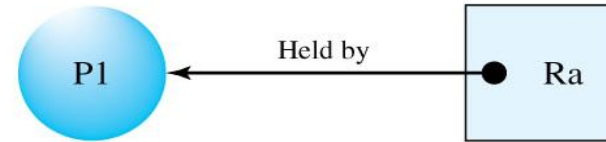
- one that can be created (***produced***) and destroyed (***consumed***).
- 예: Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking (통신사례)

# 교착상태 원리

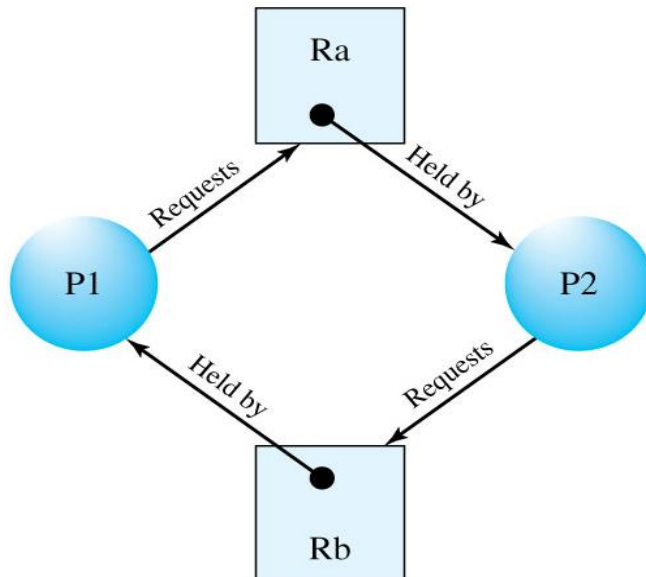
## ■ 자원 할당 그래프(Resource Allocation graph)



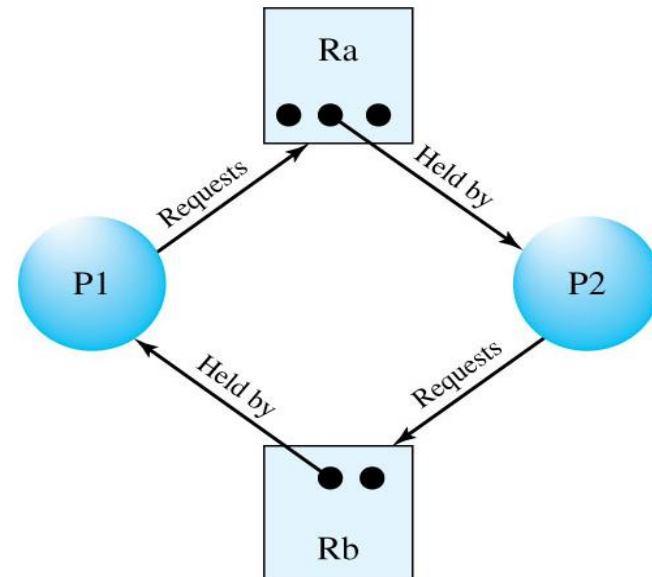
(a) 프로세스가 자원을 요청하고 있다.



(b) 자원이 프로세스에게 할당되었다.



(c) 환형 대기



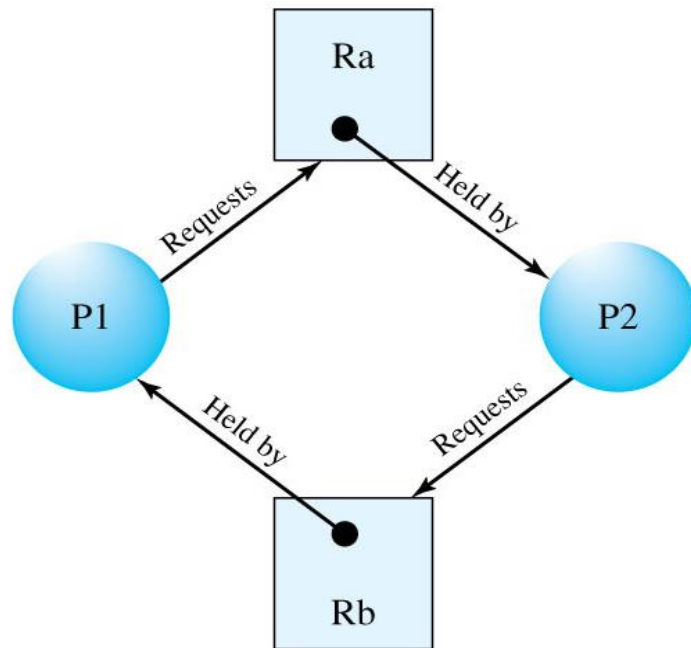
(d) 교착상태가 아닌 경우

# 교착상태 원리

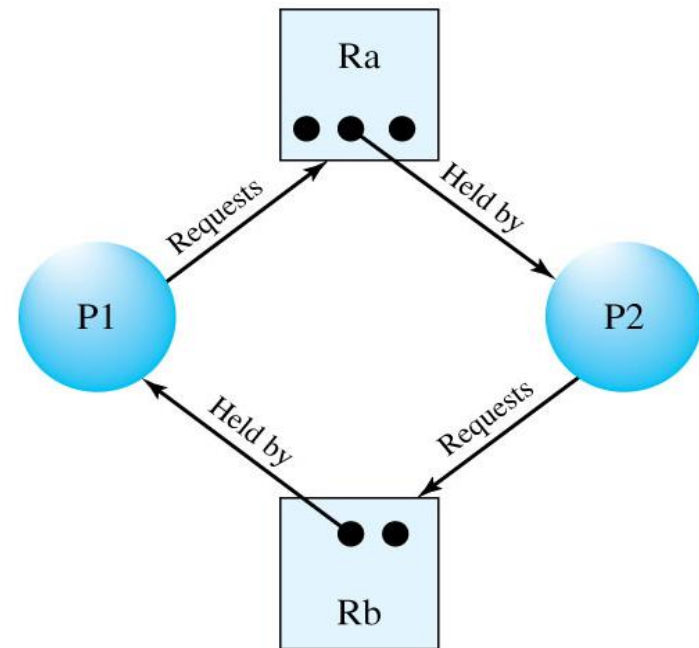
## ■ 교착상태 4가지 조건 : 필요충분조건

- ✓ 상호배제 (mutual exclusion)
- ✓ 점유대기(hold and wait)
- ✓ 비선점 (no preemption)
- ✓ 환형대기(circular wait)

## ☞ 교착상태 가능 vs 교착상태 발생



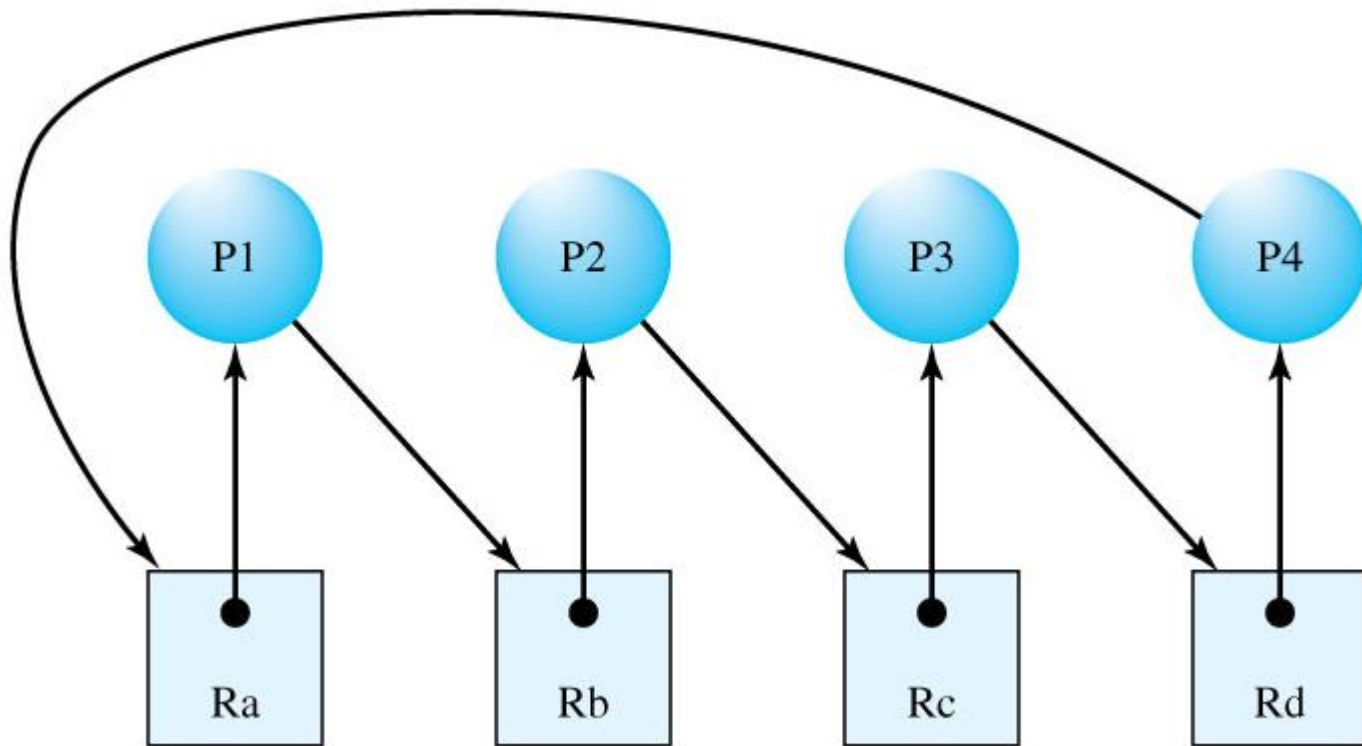
(c) 환형 대기



(d) 교착상태가 아닌 경우

# 교착상태 원리

## ■ 자원 할당 그래프의 유용성



## 6.2 교착상태 예방 (deadlock prevention)

- 교착 상태 예방
  - ✓ 교착 상태가 발생하기 위한 4가지 필요충분 조건 중 하나를 설계 단계에서 배제하는 기법
- 상호배제
  - ✓ 운영체제에서 반드시 보장해 주어야 함
- 점유 대기
  - ✓ 프로세스가 필요한 모든 자원을 한꺼번에 요청
- 비선점
  - ✓ 프로세스가 새로운 자원 요청에 실패하면 기존의 자원들을 반납한 후 다시 요청 or 운영체제가 강제로 자원을 반납시킴
- 환형 대기
  - ✓ 자원 할당 순서(자원 유형) 정의

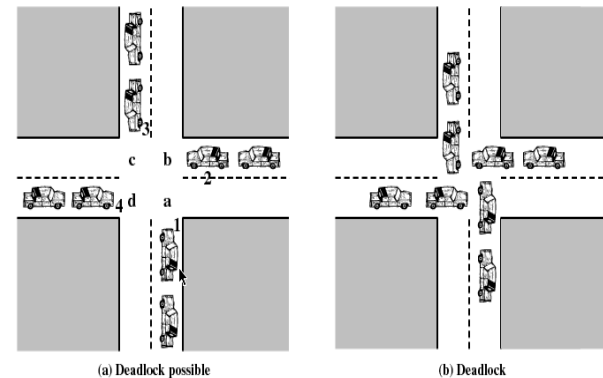


Figure 6.1 Illustration of Deadlock

# Deadlock Prevention

## ■ Simple examples of Deadlock

- ✓ Two or more threads (processes) access two data sets

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

// account information
int a[4] = {150,100,200,50};    // amount
float b[4] = {0.2, 0.2, 0.3, 0.1}; // interest
```

```
void *func1() {                // deposit
    int tmp1; float tmp2;

    ... // pre processing
    entercritical(a)
    tmp1 = a[1];
    tmp1 = tmp1 + 50;
    if (tmp1 >= 200) {
        entercritical(b)
        tmp2 = b[1];
        tmp2 += 0.1;
        b[1] = tmp2;
        exitcritical(b)
    }
    a[1] = tmp1;
    exitcritical(a)
    ... // post processing
}
```

☞ Prevention

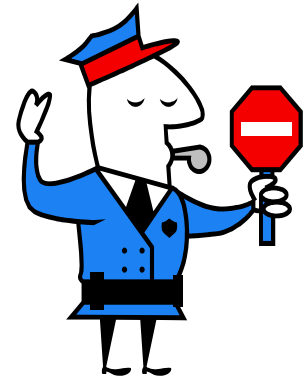
```
void *func2() {                // interest earned
    int tmp1; float tmp2;

    ... // pre processing
    entercritical(b)
    tmp2 = b[1];
    if (tmp2) {
        entercritical(a)
        tmp1 = a[1];
        tmp1 += tmp1 * tmp2;
        a[1] = tmp1;
        exitcritical(a)
    }
    exitcritical(b)
    ... // post processing
}
```

## 6.3 교착상태 회피 (deadlock avoidance)

### ■ 교착 상태 회피

- ✓ 교착상태 예방은 자원의 사용과 프로세스 수행에 비효율성을 야기할 수 있다.
- ✓ 교착상태 회피 기법은 교착 상태 발생을 위한 4가지 조건 중 1, 2, 3을 허용하며, 4를 처리할 때 처럼 자원 할당 순서를 미리 정하지 않는다.
- ✓ 그대신 자원을 할당할 때 교착 상태가 발생 가능한 상황으로 진행하지 않도록 고려한다.
  - 미래의 자원 요청 정보 필요



# 교착상태 회피 (deadlock avoidance)

---

## ■ 시스템 상태 구분

- ✓ 안전한 상태(safe state): 교착상태가 발생하지 않도록 프로세스에게 자원을 할당할 수 있는 진행 경로가 존재
- ✓ 안전하지 않은 상태(unsafe state): 경로가 없음
- ✓ 자원 할당 거부 (Resource Allocation Denial)
  - 자원을 할당할 때 교착상태가 발생할 가능성이 있는지 여부를 동적으로 판단
  - 교착상태의 가능성이 없을 때 자원 할당 (안전한 상태를 계속 유지할 수 있으면 자원 할당)
  - 각 프로세스들이 필요한 자원들을 미리 운영체제에게 알려야 함
- ✓ 프로세스 시작 거부 (Process Initialization Denial)
  - 교착상태가 발생할 가능성이 있으면 프로세스 시작 거부



- Granting the request leads to an unsafe state?
  - True: deny the request
  - False: carry the request out

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10  
Safe

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2  
Safe

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1  
Unsafe!

# 교착상태 회피 (deadlock avoidance)

## ■ 사용하는 벡터와 행렬

자원 = $\mathbf{R} = (R_1, R_2, \dots, R_m)$	시스템에 존재하는 자원의 전체 개수
가용 = $\mathbf{V} = (V_1, V_2, \dots, V_m)$	시스템에 존재하는 자원 중 현재 사용가능한 자원의 개수
요구 = $\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{pmatrix}$	$C_{ij}$ 는 프로세스 $i$ 가 자원 $j$ 를 요구하고 있음을 의미
할당 = $\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{pmatrix}$	$A_{ij}$ 는 프로세스 $i$ 가 자원 $j$ 를 할당받아 점유하고 있음을 의미

# Deadlock Avoidance

## ■ Safe State 예 (Banker's Algorithm)

	R1	R2	R3		R1	R2	R3		R1	R2	R3		R1	R2	R3		R1	R2	R3		R1	R2	R3
P1	3	2	2	P1	1	0	0	P1	2	2	2	P1	3	2	2	P1	1	0	0	P1	2	2	2
P2	6	1	3	P2	6	1	2	P2	0	0	1	P2	0	0	0	P2	0	0	0	P2	0	0	0
P3	3	1	4	P3	2	1	1	P3	1	0	3	P3	3	1	4	P3	2	1	1	P3	1	0	3
P4	4	2	2	P4	0	0	2	P4	4	2	0	P4	4	2	2	P4	0	0	2	P4	4	2	0
Claim matrix C				Allocation matrix A				C - A				Claim matrix C				Allocation matrix A				C - A			

R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
9	3	6	0	1	1	9	3	6	6	2	3
Resource vector R			Available vector V			Resource vector R			Available vector V		

(a) Initial state

(b) P2 runs to completion

	R1	R2	R3		R1	R2	R3		R1	R2	R3		R1	R2	R3		R1	R2	R3		R1	R2	R3
P1	0	0	0	P1	0	0	0	P1	0	0	0	P1	0	0	0	P1	0	0	0	P1	0	0	0
P2	0	0	0	P2	0	0	0	P2	0	0	0	P2	0	0	0	P2	0	0	0	P2	0	0	0
P3	3	1	4	P3	2	1	1	P3	1	0	3	P3	0	0	0	P3	0	0	0	P3	0	0	0
P4	4	2	2	P4	0	0	2	P4	4	2	0	P4	4	2	2	P4	0	0	2	P4	4	2	0
Claim matrix C				Allocation matrix A				C - A				Claim matrix C				Allocation matrix A				C - A			

R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
9	3	6	7	2	3	9	3	6	9	3	4
Resource vector R			Available vector V			Resource vector R			Available vector V		

(c) P1 runs to completion

(d) P3 runs to completion

# Deadlock Avoidance

## ■ Unsafe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	1	1	2

Available vector V

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	0	1	1

Available vector V

(a) Initial state

(b) P1 requests one unit each of R1 and R3

- ➡ Resource Allocation Denial (Banker's algorithm)
- ➡ Process Initialization Denial

# Deadlock Avoidance

```
struct state {  
    int resource[m];    /* 자원 벡터 */  
    int available[m];   /* 가용 벡터 */  
    int claim[n][m];    /* 요구 행렬 */  
    int alloc[n][m];    /* 할당 행렬 */  
}
```

(a) 전역 자료 구조

```
if (alloc [i,*] + request [*] > claim [i,*])  
    <에러(error)>;                /* 전체 요청이 요구한 것보다 큼 */  
else if (request [*] > available [*])  
    <프로세스 일시 중지>;  
else {                            /* 자원 할당 부분 */  
    <새로운 상태(newstate)로 전이:  
    alloc [i,*] = alloc [i,*] + request [*];  
    available [*] = available [*] - request [*]>;  
}  
if (safe (newstate))  
    <자원 할당 수행>;  
else {  
    <원래 상태로 복구>;  
    <프로세스 일시 중지>;  
}
```

(b) 자원 할당 알고리즘

```
boolean safe (state S)  
{  
    int currentavail[m];  
    process rest[<number of processes>];  
    currentavail = available;  
    rest = {all processes};  
    possible = true;  
    while (possible)  
    {  
        <find a process  $P_k$  in rest such that  
        claim [k,*] - alloc [k,*] <= currentavail;>  
        if (found)                            /* simulate execution of  $P_k$  */  
        {  
            currentavail = currentavail + alloc [k,*];  
            rest = rest - { $P_k$ };  
        }  
        else  
            possible = false;  
    }  
    return (rest == null);  
}
```

(c) test for safety algorithm (banker's algorithm)

# 교착상태 회피


## ■ 교착상태 회피 장점

- ✓ 교착상태 발견에 비해: 선점이나 롤백의 필요가 없다
- ✓ 교착상태 예방에 비해: 덜 제한적이다.

## ■ 교착상태 회피 단점



- Maximum resource requirement for each process must be stated in advance



- Processes under consideration must be independent and with no synchronization requirements



- There must be a fixed number of resources to allocate



- No process may exit while holding resources

## 6.4 교착상태 발견 (deadlock detection)

교착상태 예방은 매우 보수적임(Deadlock prevention strategies are very conservative)

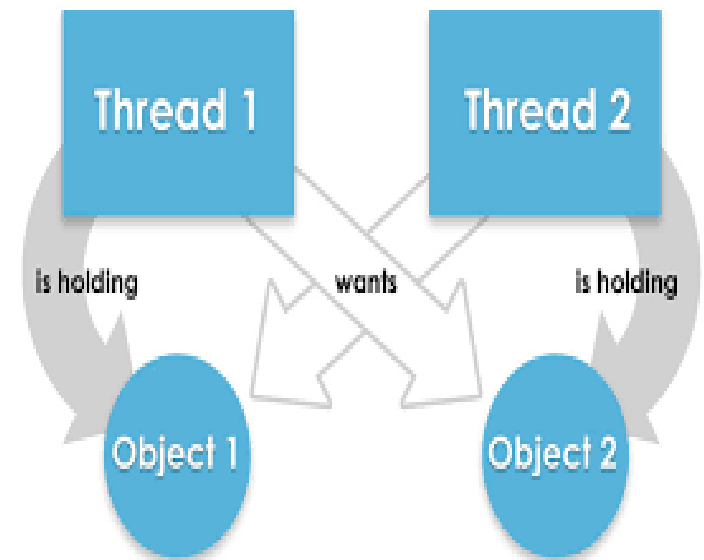
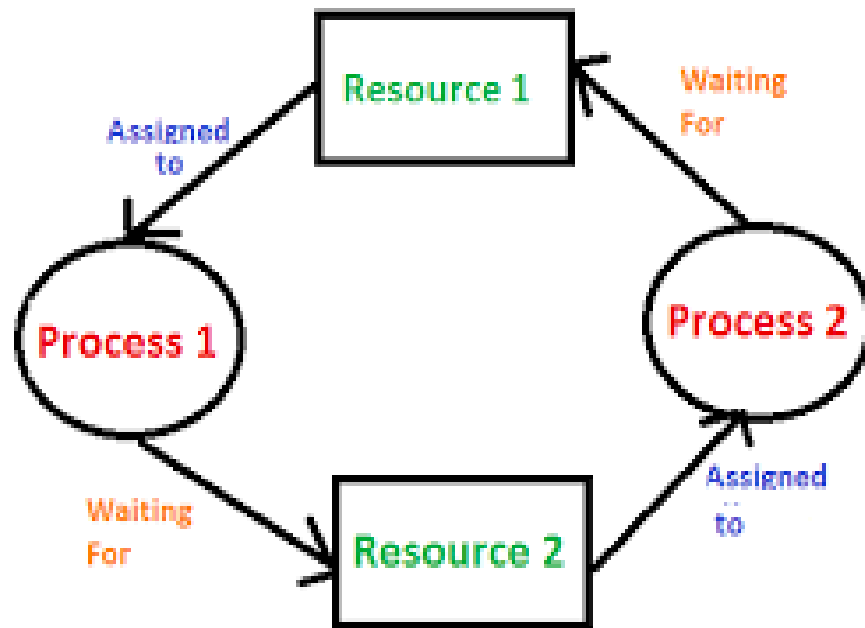
- limit access to resources by imposing restrictions on processes

교착상태 발견은 반대로 매우 낙관적임

- resource requests are granted whenever possible

### ■ 교착상태 발견 주기가 중요

- ✓ 짧으면 빠른 발견가능. 발견 기법도 간단, but 부하가 크다
- ✓ 길면 부하는 적지만 발견이 늦어지고 발견 기법도 복잡해 진다.





# Deadlock Detection

## ■ 교착상태 발견 알고리즘

1. 할당 행렬 **A**에서 행의 값이 모두 0인 프로세스를 우선 표시한다.
2. 임시 벡터 **W** 를 만든다. 그리고 현재 사용 가능한 자원의 개수(결국 가용 벡터 **V** 의 값)를 벡터 **W** 의 초기값으로 설정한다.  $W = (0\ 0\ 0\ 0\ 1)$
3. 표시되지 않은 프로세스들 중에서 수행 완료 가능한 것이 있으면 (요청 행렬 **Q** 에서 특정 행의 값이 모두 **W**보다 작은 행에 대응되는 프로세스) 이 프로세스를 표시한다.  $\langle P3 \rangle\ W = W + (0\ 0\ 0\ 1\ 0) = (0\ 0\ 0\ 1\ 1)$   
만일 완료 가능한 프로세스가 없으면 알고리즘을 종료한다.
4. 단계 3의 조건을 만족하는 행을 **Q**에서 찾으면, 할당 행렬 **A**에서 그 행에 대응되는 값을 임시 벡터 **W**에 더한다. 그리고 3 단계를 다시 수행한다.

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

요청 행렬 Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

할당 행렬 A

R1	R2	R3	R4	R5
2	1	1	2	1

자원 벡터 R

R1	R2	R3	R4	R5
0	0	0	0	1

가용 벡터 V

알고리즘 종료 후에도 P1,P2가 표시되지  
않은 상태여서, P1,P2는 교착상태 !!

# 교착상태 발견

## ■ 회복 (recovery)

- ✓ 모든 프로세스 종료
- ✓ 교착상태에 연관된 프로세스들을 체크포인트 시점으로 롤백한 후 다시 수행 시킴 (교착상태가 다시 발생할 가능성 존재)
- ✓ 교착상태가 없어질 때까지 교착상태에 포함되어 있는 프로세스들 하나씩 종료
- ✓ 교착상태가 없어질 때까지 교착상태에 포함되어 있는 자원을 하나씩 선점

## ■ 종료 (또는 선점될) 프로세스 선택 기준

- 지금까지 사용한 처리기 시간이 적은 프로세스부터
- 지금까지 생산한 출력량이 적은 프로세스부터
- 이후 남은 수행시간이 가장 긴 프로세스부터
- 할당받은 자원이 가장 적은 프로세스부터
- 우선 순위가 낮은 프로세스부터

## 6.5 Integrated Deadlock Strategy

표 6.1 | 운영체제에서 교착상태 예방, 회피, 발견 기법 정리[ISL080]

접근 방법	자원 할당 정책	구체적인 기법	장점	단점
예방	보수적 (자원 할당이 가능하더라도 조건에 따라 할당하지 않을 수 있다.)	모든 자원을 한꺼번에 요구	<ul style="list-style-type: none"> <li>순간적으로 많은 일을 하는 프로세스에 적합</li> <li>선점이 불필요</li> </ul>	<ul style="list-style-type: none"> <li>효율이 나쁨.</li> <li>프로세스 시작을 지연시킬 가능성 있음.</li> <li>프로세스는 사용할 모든 자원을 미리 알고 있어야 함.</li> </ul>
		선점 가능	<ul style="list-style-type: none"> <li>자원 상태의 저장과 복구가 간단한 자원에는 적용하기 쉬움.</li> </ul>	<ul style="list-style-type: none"> <li>선점이 필요보다 자주 일어남.</li> </ul>
		자원 할당 순서	<ul style="list-style-type: none"> <li>컴파일 시점에 강제할 수 있음</li> <li>시스템의 설계 시점에 문제를 해결했기 때문에 동적 부하가 없음.</li> </ul>	<ul style="list-style-type: none"> <li>점진적인 자원 할당이 안 됨.</li> </ul>
회피	예방과 발견의 중간 정도	교착상태가 발생하지 않는 안전한 경로를 최소한 하나는 유지	<ul style="list-style-type: none"> <li>선점이 불필요</li> </ul>	<ul style="list-style-type: none"> <li>운영체제는 자원에 대한 미래 요구량을 미리 알고 있어야 함.</li> <li>오랜 기간 지연 발생의 가능성 있음.</li> </ul>
발견	적극적 (자원 할당이 가능하면 즉시 할당한다.)	주기적으로 교착상태 발생 여부 파악	<ul style="list-style-type: none"> <li>프로세스 시작을 지연시키지 않음</li> <li>온라인 처리 가능</li> </ul>	<ul style="list-style-type: none"> <li>선점에 의한 손실 발생</li> </ul>

# Dining Philosophers Problem

공유자원을 통한 협동의 대표적인 문제이며, 이러한 협동 문제는 실제 병행 스레드 기반 응용에서 자주 발생

## ■ 문제 정의

- ✓ No two philosophers can use the same fork at the same time (mutual exclusion)
- ✓ No philosopher must starve to death (avoid deadlock and starvation ... literally!)

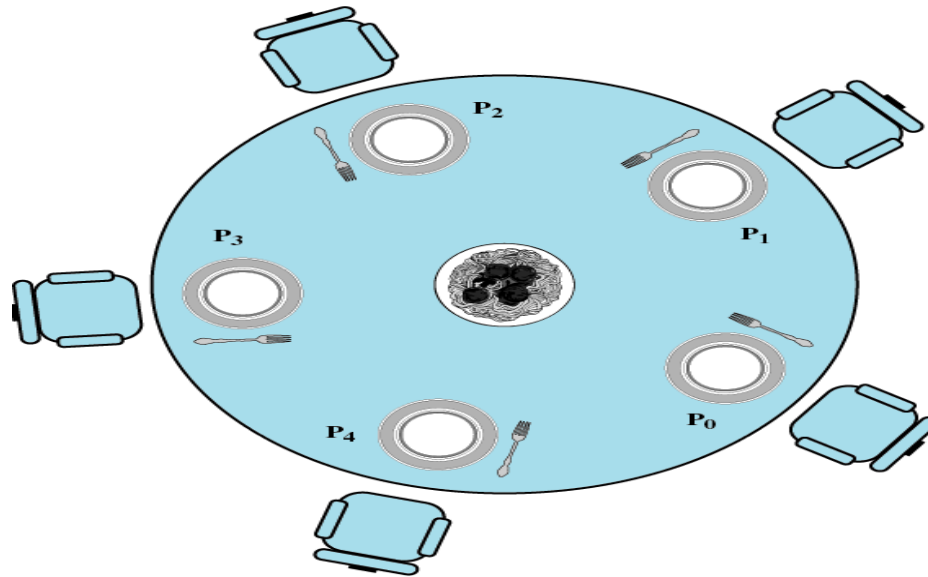


Figure 6.11 Dining Arrangement for Philosophers

# Dining Philosophers Problem

- Solution using Semaphore -> 동시에 왼쪽 포크를 잡으면?

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true)
    {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
```

**Figure 6.12** A First Solution to the Dining Philosophers Problem

# Dining Philosophers Problem

- Solution using Semaphore -> 한번에 최대 4명까지 식탁에 앉도록 제한

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int I)
{
    while (true)
    {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

# Dining Philosophers Problem

```
monitor dining_controller;
cond ForkReady[5];
boolean fork[5] = {true};

void get_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /* 왼쪽에 놓인 포크 접근 */
    if (!fork(left))
        cwait(ForkReady[left]);
    fork(left) = false;
    /* 오른쪽에 놓인 포크 접근 */
    if (!fork(right))
        cwait(ForkReady[right]);
    fork(right) = false;
}
void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /* 왼쪽에 놓인 포크 반납 */
    if (empty(ForkReady[left]))
        fork(left) = true;
    else
        csignal(ForkReady[left]);
    /* 오른쪽에 놓인 포크 반납 */
    if (empty(ForkReady[right]))
        fork(right) = true;
    else
        csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]
{
    while (true) {
        <think>;
        get_forks(k);
        <eat spaghetti>;
        release_forks(k);
    }
}
```

# Dining Philosophers Problem

---

## ■ ForkReady

- 5개의 각 조건변수는 각 **fork**에 대응
- 철학자들이 **fork**가 사용 가능할 때까지 대기할 때 이용

## ■ get\_forks proc

- 철학자가 2개의 **fork**를 요청할 때 사용되며, 2개중 1개라도 허용되지 않으면 연관된 조건변수에서 **wait**

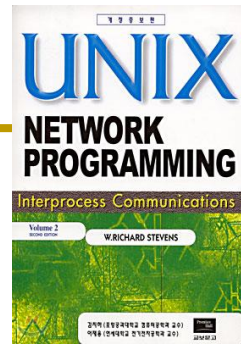
## ■ release\_forks proc

- 대기하고 있는 프로세스가 있으면 깨워주는 역할

- 철학자가 모니터에 진입했다면, 그 철학자는 왼쪽 **fork**를 잡은 다음에 오른쪽 **fork**를 잡게 되며, 그 사이에 오른쪽에 앉은 철학자가 간섭하지 않음을 보장받게 됨



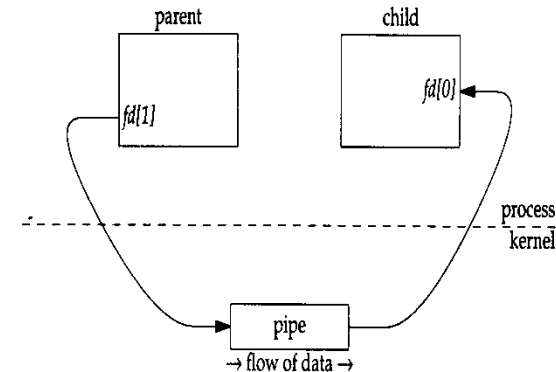
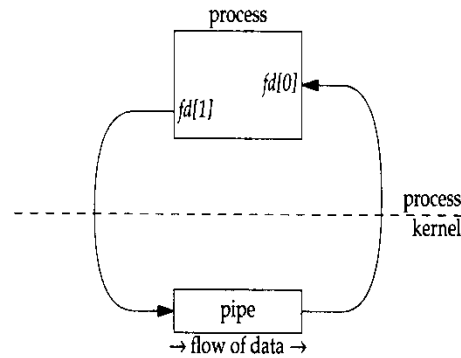
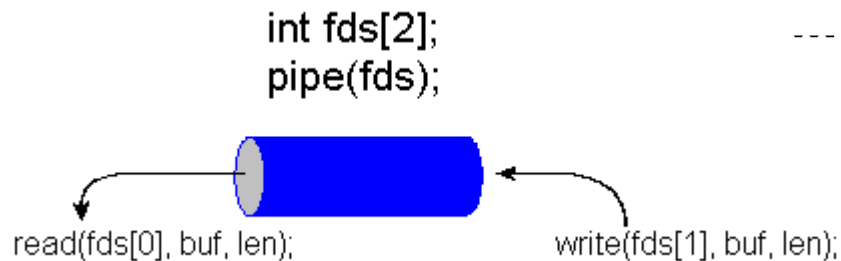
# 6.7 UNIX Concurrency Mechanisms



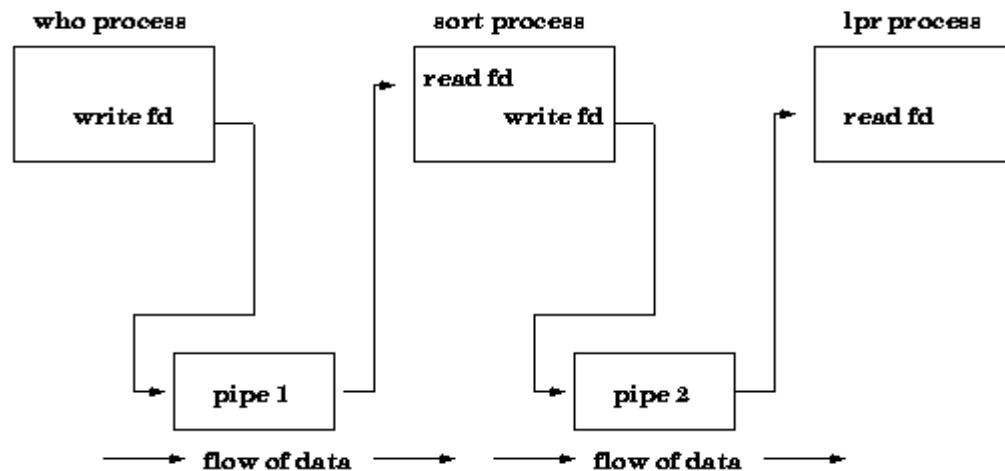
## ■ IPC (InterProcess Communication)

### ✓ Pipes

- Named, unnamed

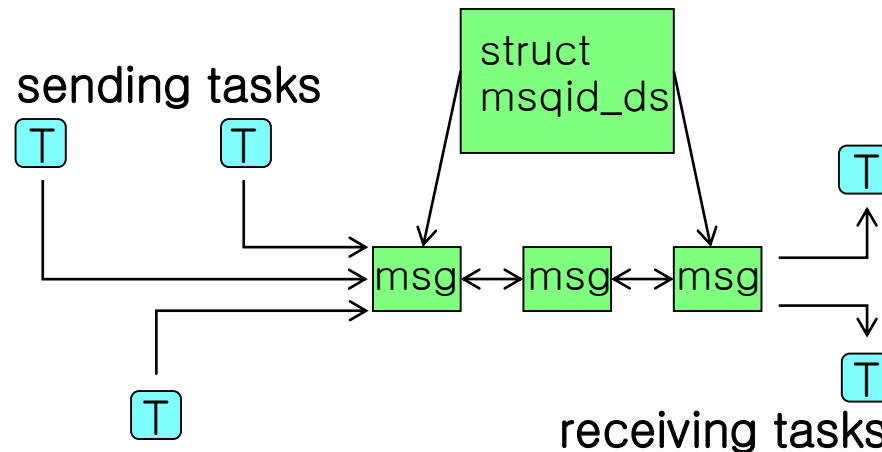
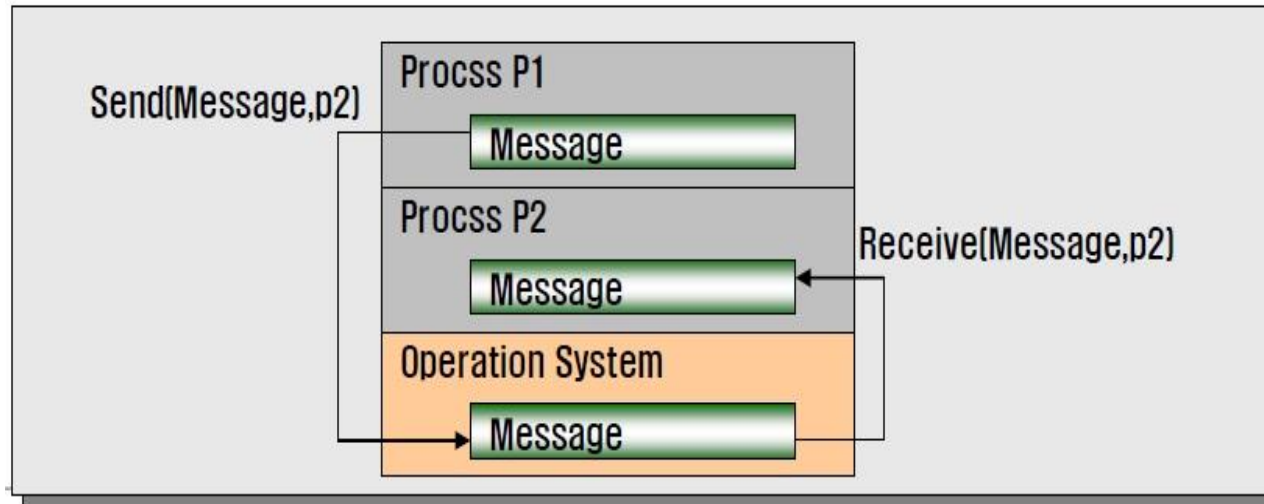


- Who | sort | lpr



## 6.7 UNIX Concurrency Mechanisms

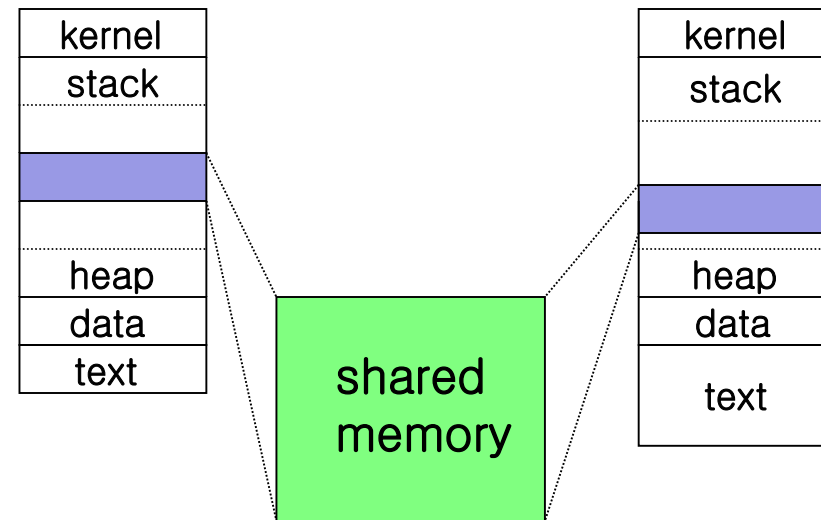
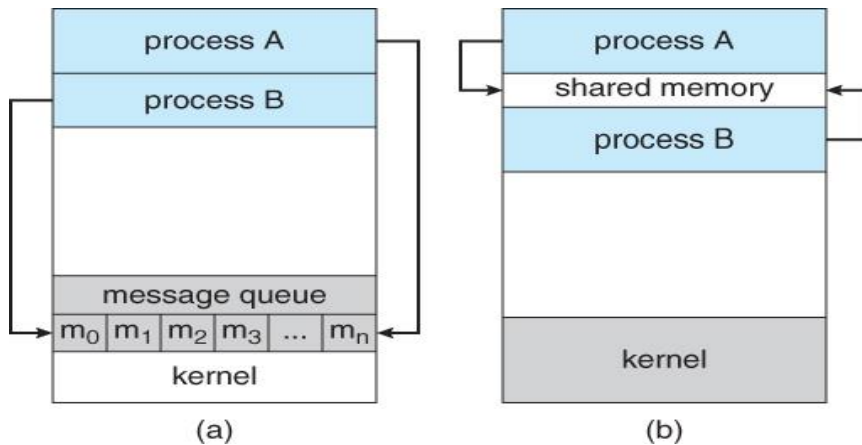
- IPC (InterProcess Communication)
  - ✓ Messages



## 6.7 UNIX Concurrency Mechanisms

### ■ IPC (InterProcess Communication)

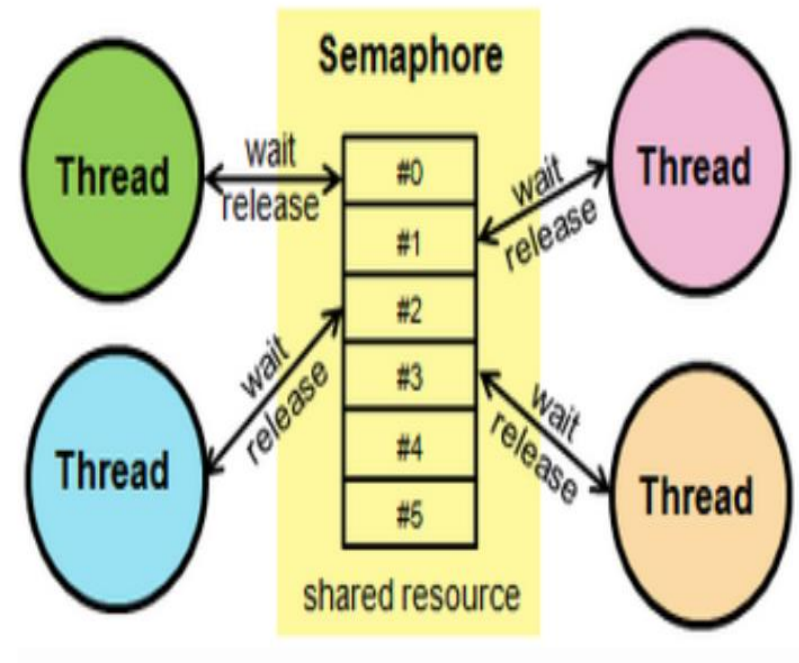
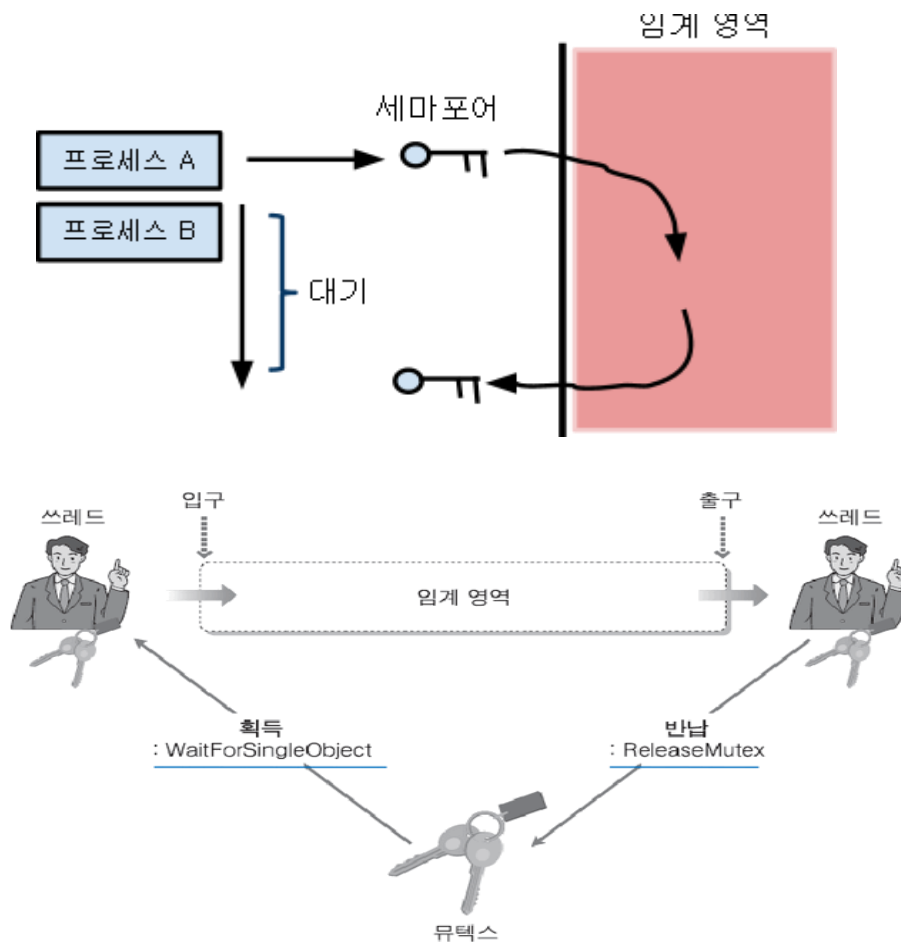
- ✓ Shared memory



## 6.7 UNIX Concurrency Mechanisms

### ■ IPC

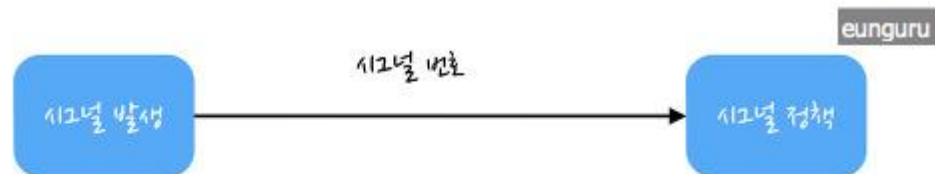
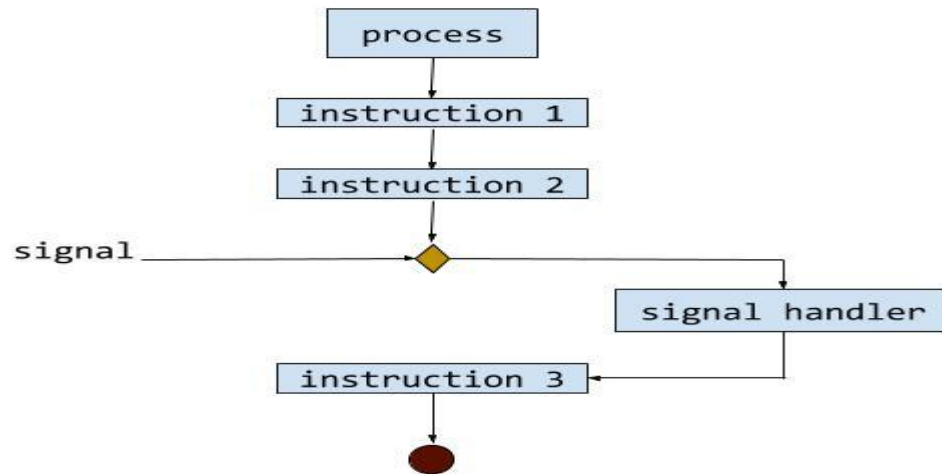
#### ✓ Semaphore



## 6.7 UNIX Concurrency Mechanisms

### ■ IPC (InterProcess Communication)

- ✓ Signal(시그널) : 비동기적인 사건의 발생을 프로세스에 알리는 SW적인 방법
- ✓ HW 인터럽트와 비슷하지만, 우선순위는 없음



- 예외 상황: 커널
- 외부 상황: 키보드 입력
- 이벤트 발생: alarm() 함수, 프로세스 종료
- 인위적 발생: kill() 함수(System call), Kill 명령어

- 무시
- 보류
- 특정 함수 호출
- 디폴트(일반적으로 종료)

# UNIX Concurrency Mechanisms

## ■ Signals

표 6.2 | UNIX 시그널 정의

시그널 번호	시그널 이름	설명
01	SIGHUP	Hang up. 프로세스가 사용자와 단절되어 특별한 작업을 수행하지 않고 있는 상태로 파악되면 운영체제가 프로세스에 이 시그널을 보낸다.
02	SIGINT	인터럽트
03	SIGQUIT	Quit. 사용자가 특정 프로세스를 종료시키고 코어 덤프를 생성시킬 때 사용
04	SIGILL	불법 명령어 수행
05	SIGTRAP	트레이스 트랩. 프로세스의 수행 단계를 추적할 수 있다.
06	SIGIOT	IOT 명령어
07	SIGEMT	EMT 명령어
08	SIGFPE	부동소수점 예외
09	SIGKILL	Kill 프로세스 종료
10	SIGBUS	버스 에러
11	SIGSEGV	세그멘테이션 오류. 프로세스가 자신에게 할당된 이외의 주소 참조
12	SIGSYS	시스템 호출에서 잘못된 인자 사용
13	SIGPIPE	파이프 오류, 파이프에서 데이터를 읽으려는 프로세스가 없는데, 그 파이프 데이터를 쓰려고 할 때 발생
14	SIGALRM	알람. 프로세스가 일정 시간 이후에 시그널을 받으려고 할 때 사용
15	SIGTERM	소프트웨어 종료
16	SIGUSR1	사용자 정의 시그널 1
17	SIGUSR2	사용자 정의 시그널 2
18	SIGCHLD	자식 프로세스 종료
19	SIGPWR	전원 결합

# Unix Signal

## ■ Signal

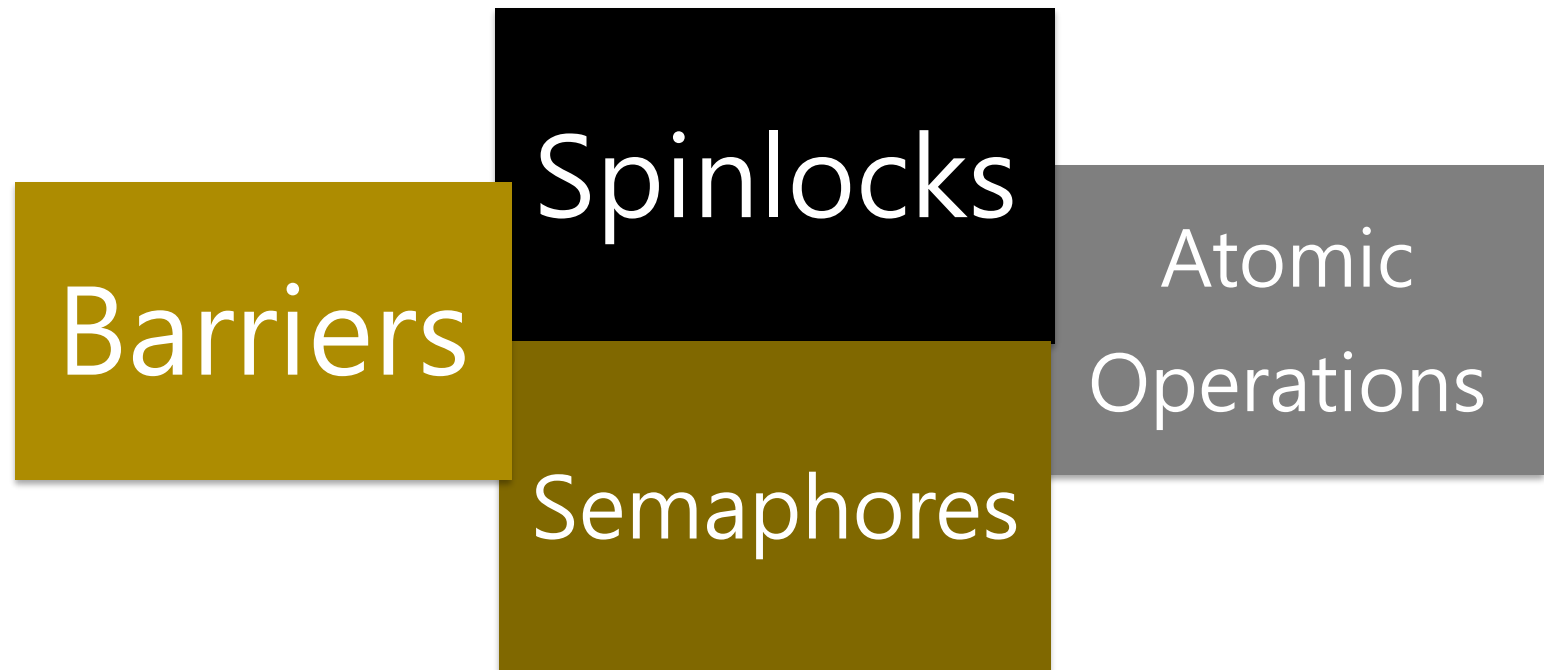
```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
Void ouch(int sig) {
    printf (" OUCH – I got Signal %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}
Int main() {
    (void) signal(SIGINT, ouch);
    while(1) {
        printf("Hello Operating Systems !!!\n");
        sleep(1);
    }
}
```

=> 실행시 CTRL+C, 잠시후 CTRL+C

## 6.8 리눅스 커널 병행성 기법

---

- All UNIX mechanisms + additional features





## 6.8 리눅스 병행성 기법

---

- 유닉스 병행성 기법 지원
- 원자적 연산
  - ✓ Atomic operations execute without interruption/interference
- 스핀 락
  - ✓ Only one thread at a time can acquire a spinlock.
  - ✓ Any other thread will keep trying (spinning) until it can acquire the lock.
- 세마포어
  - ✓ Binary semaphores, counting semaphores, reader-writer semaphores
- 장벽 (barrier)
  - ✓ To enforce the order in which instructions are executed

# Linux Kernel Concurrency Mechanisms

- Includes all the mechanisms found in UNIX
- Atomic operations execute without interruption and without interference

Atomic Integer Operations	
ATOMIC_INIT (int i)	At declaration: initialize an atomic_t to i
int atomic_read(atomic_t *v)	Read integer value of v
void atomic_set(atomic_t *v, int i)	Set the value of v to integer i
void atomic_add(int i, atomic_t *v)	Add i to v
void atomic_sub(int i, atomic_t *v)	Subtract i from v
void atomic_inc(atomic_t *v)	Add 1 to v
void atomic_dec(atomic_t *v)	Subtract 1 from v
int atomic_sub_and_test(int i, atomic_t *v)	Subtract i from v; return 1 if the result is zero; return 0 otherwise
int atomic_add_negative(int i, atomic_t *v)	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
int atomic_dec_and_test(atomic_t *v)	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
int atomic_inc_and_test(atomic_t *v)	Add 1 to v; return 1 if the result is zero; return 0 otherwise

Atomic Bitmap Operations	
void set_bit(int nr, void *addr)	Set bit nr in the bitmap pointed to by addr
void clear_bit(int nr, void *addr)	Clear bit nr in the bitmap pointed to by addr
void change_bit(int nr, void *addr)	Invert bit nr in the bitmap pointed to by addr
int test_and_set_bit(int nr, void *addr)	Set bit nr in the bitmap pointed to by addr; return the old bit value
int test_and_clear_bit(int nr, void *addr)	Clear bit nr in the bitmap pointed to by addr; return the old bit value
int test_and_change_bit(int nr, void *addr)	Invert bit nr in the bitmap pointed to by addr; return the old bit value
int test_bit(int nr, void *addr)	Return the value of bit nr in the bitmap pointed to by addr

# Linux Kernel Concurrency Mechanisms

## ■ 스핀 락(Spinlocks)

- ✓ Busy-waiting
- ✓ Used for protecting a critical section
- ✓ `basic spinlock(plain, irq, irqsave, bh)`, reader-writer spinlock

**Table 6.4 Linux Spinlocks**

<code>void spin_lock(spinlock_t *lock)</code>	Acquires the specified lock, spinning if needed until it is available
<code>void spin_lock_irq(spinlock_t *lock)</code>	Like <code>spin_lock</code> , but also disables interrupts on the local processor
<code>void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)</code>	Like <code>spin_lock_irq</code> , but also saves the current interrupt state in flags
<code>void spin_lock_bh(spinlock_t *lock)</code>	Like <code>spin_lock</code> , but also disables the execution of all bottom halves
<code>void spin_unlock(spinlock_t *lock)</code>	Releases given lock
<code>void spin_unlock_irq(spinlock_t *lock)</code>	Releases given lock and enables local interrupts
<code>void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)</code>	Releases given lock and restores local interrupts to given previous state
<code>void spin_unlock_bh(spinlock_t *lock)</code>	Releases given lock and enables bottom halves
<code>void spin_lock_init(spinlock_t *lock)</code>	Initializes given spinlock
<code>int spin_trylock(spinlock_t *lock)</code>	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
<code>int spin_is_locked(spinlock_t *lock)</code>	Returns nonzero if lock is currently held and zero otherwise

# Linux Kernel Concurrency Mechanisms

## ■ Semaphores

- ✓ 기본 세마포어(up,down), 읽기-쓰기 세마포어

표 6.5 | 리눅스 세마포어

일반 세마포어	
<code>void sema_init(struct semaphore *sem, int count)</code>	세마포어 동적 생성. 세마포어의 초기 값을 <code>count</code> 로 설정
<code>void init_MUTEX(struct semaphore *sem)</code>	세마포어 동적 생성. 세마포어의 초기 값을 1로 설정(결과적으로, 세마포어를 사용가능한 상태로 초기화)
<code>void init_MUTEX_LOCKED(struct semaphore *sem)</code>	세마포어 동적 생성. 세마포어의 초기 값을 0으로 설정(결과적으로, 세마포어를 사용 중인 상태로 초기화)
<code>void down(struct semaphore *sem)</code>	세마포어 획득을 시도. 세마포어 획득이 불가능하면 프로세스는 인터럽트 불가능한 수면 상태로 전이.
<code>int down_interruptible(struct semaphore *sem)</code>	세마포어 획득을 시도. 세마포어 획득이 불가능하면 프로세스는 인터럽트 가능한 수면 상태로 전이. 수면 상태에서 깨어났을 때 수신한 이벤트가 <code>up</code> 이 아닌 시그널이라면 <code>-EINTR</code> 을 리턴한다.
<code>int down_trylock(struct semaphore *sem)</code>	세마포어 획득을 시도. 세마포어 획득이 불가능하면 0이 아닌 값으로 리턴
<code>void up(struct semaphore *sem)</code>	세마포어를 반납
읽기/쓰기 세마포어	
<code>void init_rwsem(struct rw_semaphore, *rwsem)</code>	세마포어 동적 생성. 세마포어의 초기 값을 1로 설정
<code>void down_read(struct rw_semaphore, *rwsem)</code>	읽기를 시도하는 프로세스가 사용하는 down 연산
<code>void up_read(struct rw_semaphore, *rwsem)</code>	읽기를 시도하는 프로세스가 사용하는 up 연산
<code>void down_write(struct rw_semaphore, *rwsem)</code>	쓰기를 시도하는 프로세스가 사용하는 down 연산
<code>void up_write(struct rw_semaphore, *rwsem)</code>	쓰기를 시도하는 프로세스가 사용하는 up 연산

# Linux Kernel Concurrency Mechanisms

## ■ 메모리 Barriers

표 6.6 Linux 메모리 장벽(barrier) 연산

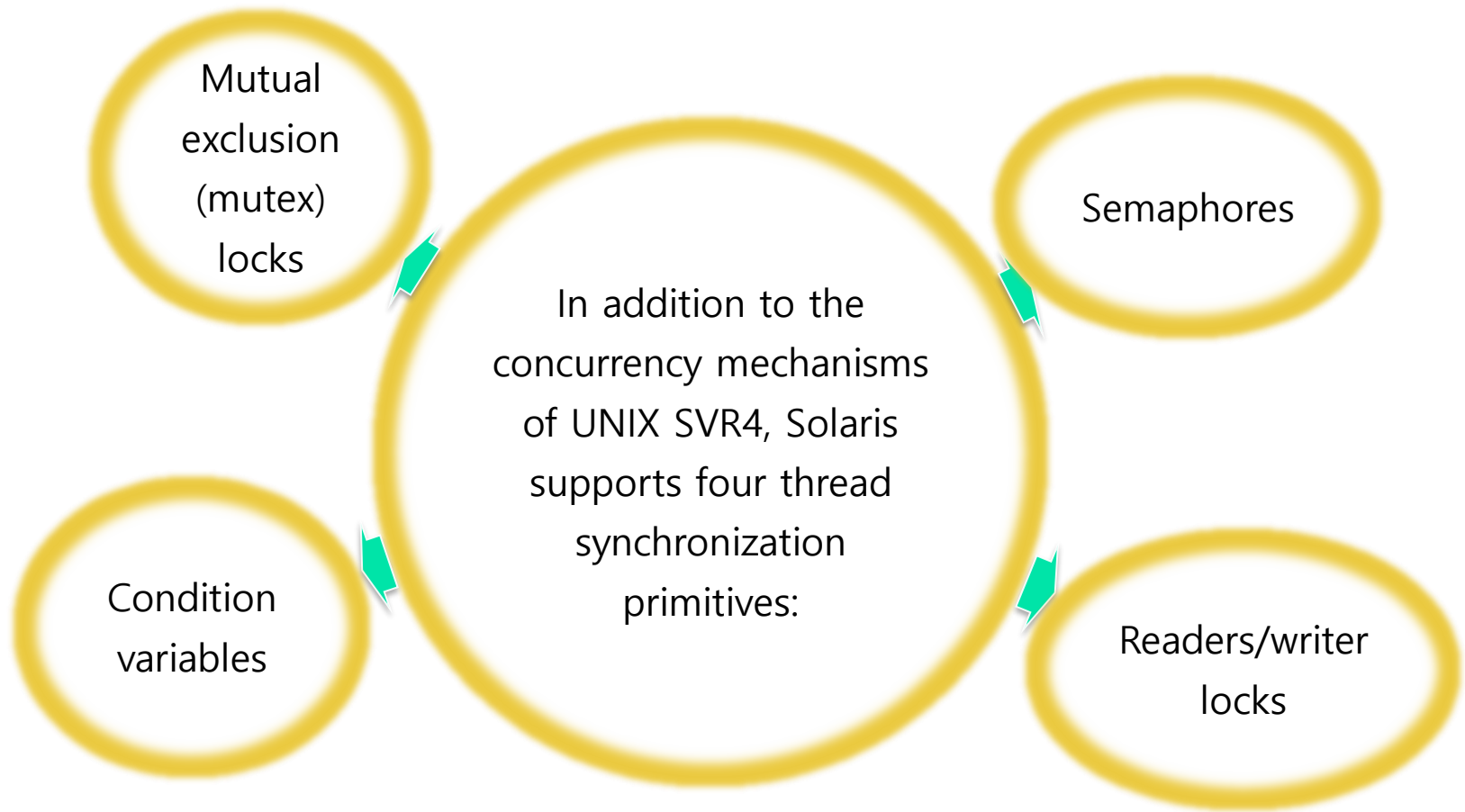
<code>rmb()</code>	메모리 읽기(load) 명령들의 실행 순서 변경이 장벽을 넘는 것을 방지
<code>wmb()</code>	메모리 쓰기(store) 명령들의 실행 순서 변경이 장벽을 넘는 것을 방지
<code>mb()</code>	메모리 읽기/쓰기 명령들의 실행 순서 변경이 장벽을 넘는 것을 방지
<code>barrier()</code>	컴파일러가 명령들의 실행 순서 변경할 때 장벽을 넘는 것을 방지
<code>smp_rmb()</code>	SMP 시스템에서는 <code>rmb()</code> , UP 시스템에서는 <code>barrier()</code>
<code>smp_wmb()</code>	SMP 시스템에서는 <code>wmb()</code> , UP 시스템에서는 <code>barrier()</code>
<code>smp_mb()</code>	SMP 시스템에서는 <code>mb()</code> , UP 시스템에서는 <code>barrier()</code>

SMP = 대칭형 멀티프로세서

UP = 단일처리기

## 6.9 솔라리스 스레드 동기화 프리미티브

---



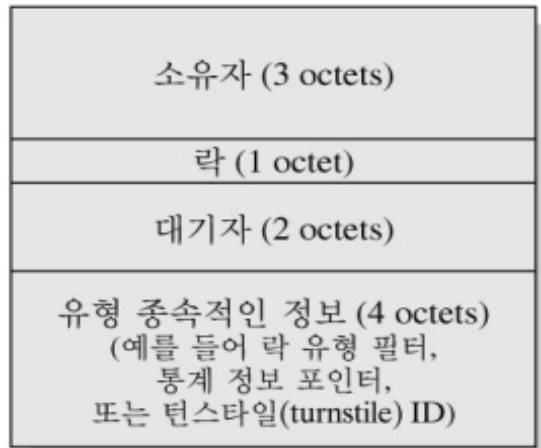
## 6.9 Solaris 스레드 동기화 프리미티브

---

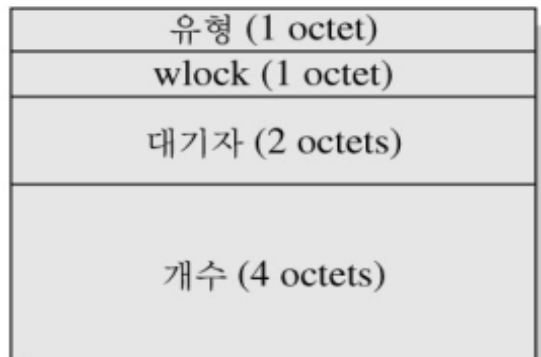
- 상호배제 (mutex) 락
  - ✓ A mutex is used to ensure only one thread at a time can access the resource protected by the mutex.
  - ✓ 락(locks)을 획득한 스레드는 반드시 락을 반납(unlocks) 해야한다.
    - `mutex_enter()`, `mutex_exit()`
- 세마포어 (semaphore)
  - ✓ Solaris provides classic counting semaphores.
- 읽기-쓰기 락 (readers/writers lock)
  - ✓ The readers/writer lock allows multiple threads to have simultaneous read-only access to an object protected by the lock.
- 조건 변수 (conditional variables)
  - ✓ A condition variable is used to wait until a particular condition is true.
  - ✓ Condition variables must be used with a mutex lock
    - `cv_wait()`, `cv_signal()`

# 솔라리스 스레드 동기화 프리미티브

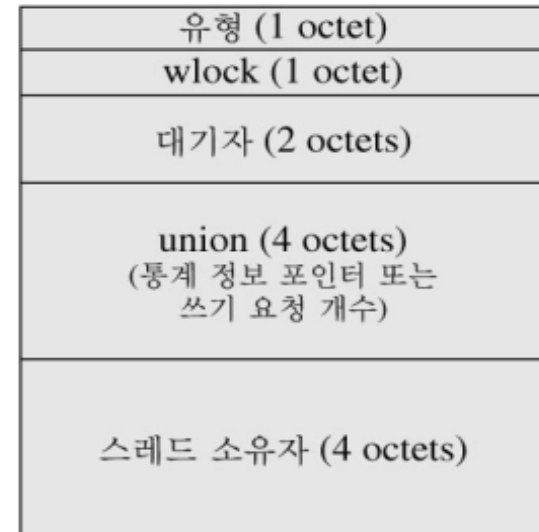
## ■ 구조



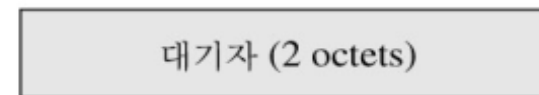
(a) 상호배제(mutex) 락



(b) 세마포어



(c) 읽기/쓰기 락



(d) 조건 변수



## 6.10 윈도우즈 7 병행성 기법

---

- 객체 아키텍처의 일부분으로 동기화 기법 제공

Most important methods are:

- executive dispatcher objects
- user mode critical sections
- slim reader-writer locks
- condition variables
- lock-free operations

# 윈도우즈 7 병행성 기법

- 대기 함수 (Wait functions) 이용
  - wait functions 수행중에 스레드의 블록을 허용하는 함수
  - wait functions 가 호출되면 특정 조건이 만족될때 까지 리턴하지 않는다
- 대표적인 (동기화)디스패처 객체

표 6.7 WINDOWS 동기화 객체

객체 유형(object type)	정의	시그널 받는 시점	대기 스레드의 영향
알림 사건 (notification event)	시스템 사건의 발생을 알림	스레드가 사건을 설정	모두 깨어남
동기화 사건 (Synchronization event)	시스템 사건의 발생을 알림	스레드가 사건을 설정	한 스레드가 깨어남
상호 배제 (Mutex)	상호 배제 기능 제공. 이진 세마포어와 동일	소유자(또는 다른 스레드)로부터 mutex 반납	한 스레드가 깨어남
세마포어	자원을 사용할 수 있는 스레드의 개수를 조절하는 카운터	세마포어 카운터의 값이 0으로 됨	모두 깨어남
대기가능 타이머 (waitable timer)	시간의 흐름을 기록하는 카운터	타임 설정 또는 타임 인터벌의 소멸	모두 깨어남
파일	오픈된 파일 또는 I/O 장치의 인스턴스	I/O 연산의 종료	모두 깨어남
프로세스	프로그램 시작, 주소 공간과 프로그램 수행을 위한 자원이 포함됨	마지막 스레드 종료	모두 깨어남
스레드	프로세스 내부의 수행 객체	스레드 종료	모두 깨어남

참고: 음영 처리된 행(1~5행)의 객체는 동기화를 목적으로 제공되는 객체이다.

# 윈도우즈 7 병행성 기법

---

- 임계 영역
  - ✓ Similar mechanism to mutex (except that critical sections can be used only by the threads of a single process. )
  - ✓ If the system is a multiprocessor, the code will attempt to acquire a spin-lock.
- Slim 읽기-쓰기 락
  - ✓ Windows Vista added a user mode reader-writer.
  - ✓ 'Slim' as it normally only requires allocation of a single pointer-sized piece of memory.
- Condition Variable
  - ✓ Windows Vista also added condition variables.
  - ✓ Used with either critical sections or SRW locks
- Lock free 동기화
  - ✓ Synchronizing without taking a SW lock.
  - ✓ A thread can never be switched away from a processor while still holding a lock

## 6.11 Android 프로세스 간 통신

---

- Android adds to the kernel a new capability known as Binder
  - ✓ 바인더(Binder)는 경량 RPC (remote procedure call) 능력을 제공하며, 메모리와 프로세서 사용측면에서 효율적
  - ✓ 2개의 프로세스간에 모든 상호작용을 중재
- The RPC mechanism works between two processes on the same system but running on different virtual machines
- The method used for communicating with the Binder is the *ioctl* system call
  - ✓ the *ioctl* call은 특정 장치에 해당하는 I/O 연산들을 처리하기 위해 제공되는 범용 시스템 호출(general-purpose system call )

# Android 프로세스 간 통신

## ■ 바인더 연산 예

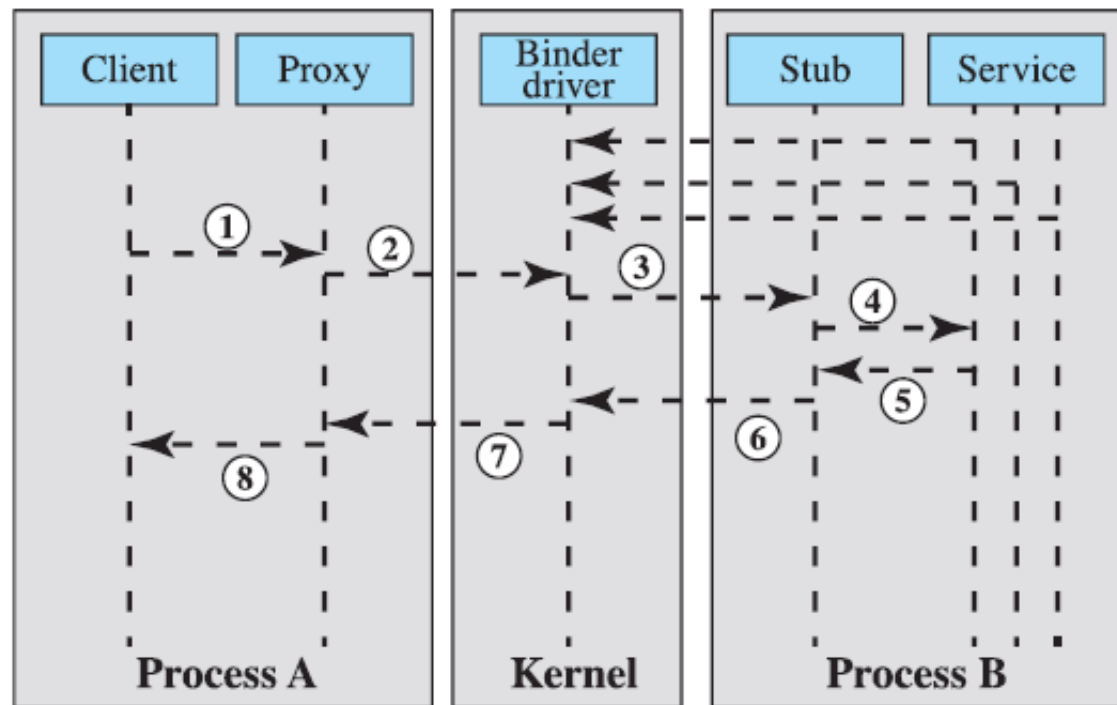


그림 6.16 바인더 연산

# Summary

---

- Principles of Deadlock
  - ✓ 4 conditions for deadlock
  - ✓ Resource allocation graph
- Deadlock handing
  - ✓ Prevention
  - ✓ Avoidance
  - ✓ Detection
- Dining Philosophers Problem
- Case Study
  - ✓ Unix, Linux Kernel, Solaris, Windows 7, Android