6장 병행성: 교착상태와 기아

6장의 강의 목표

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

목 차

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 유닉스 병행성 기법

6.8 리눅스 커널 병행성 기법

6.9 솔라리스 스레드 동기화 프리미티브

6.10 윈도우즈 병행성 기법

6.1 교착상태 원리(deadlock principles)

- 교착상태 예

```
/* DeadLock example.c */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int a[4] = {150,100,200,50};          // account
int b[4] = {1, 1, 2, 0};              // credit status
```

☞ 상호배제를 위해 5장에서 배운 기법을 사용하면..

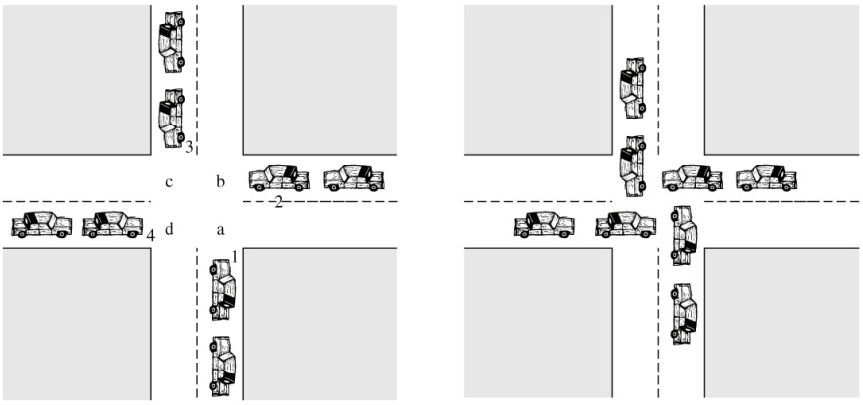
☞ 이때 발생하는 문제는?

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

    ...  // pre processing
    entercritical(a)
    tmp1 = a[1];
    tmp1 = tmp1 + 10;
    if (tmp1 > 200) {
        entercritical(b)
         tmp2 = b[1];
         tmp2 += 1;
         b[1] = tmp2;
        exitcritical(b)
    }
    a[1] = tmp;
    exitcritical(a)
    ... // post processing
}
```

```
void *func2() {              // credit evaluation
    int tmp1, tmp2;

    ...  // pre processing
    entercritical(b)
    tmp2 = b[1];
    if (tmp2 >= 2) {
        entercritical(a)
         tmp1 = a[1];
         tmp1 =  tmp1 * 0.05;
         a[1] = tmp1;
        exitcritical(a)
    }
    exitcritical(b)
    ... // post processing
}
```

교착상태의 정의

- 영속적인 블록 상태
- 2개 이상의 프로세스들이 공유 자원에 대한 경쟁이나 통신 중에 발생
- 사거리에서의 교착상태 예



(a) 교착상태 가능성 존재 (b) 교착상태 발생

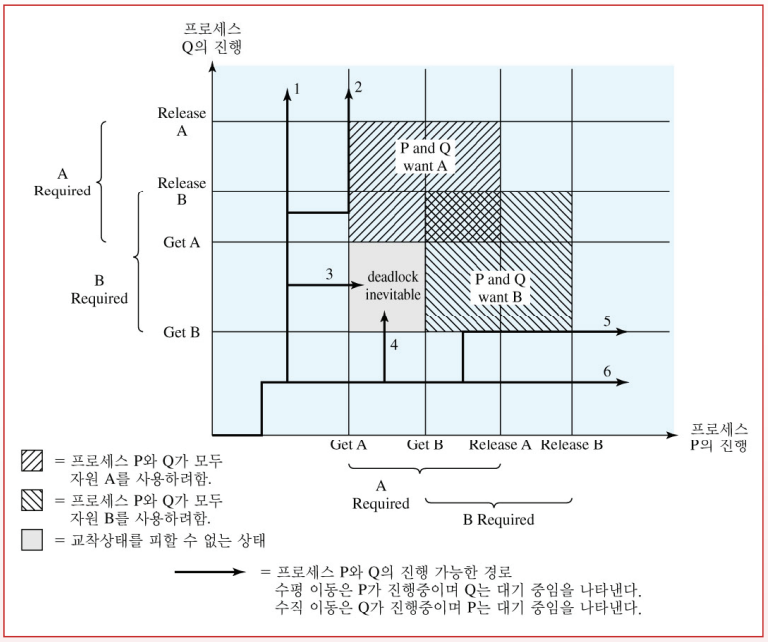
결합 진행 다이어그램

- 다음과 같이 실행하는 두 프로세스 P와 Q가 있다면,
  - 둘 다 자원 A와 B에 대해 배타적 사용을 원하고 있음
  - 단, 자원 사용 기간은 유한함.

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Get B	Get A
• • •	• • •
Release A	Release B
• • •	• • •
Release B	Release A
• • •	• • •

결합 진행 다이어그램 (계속)

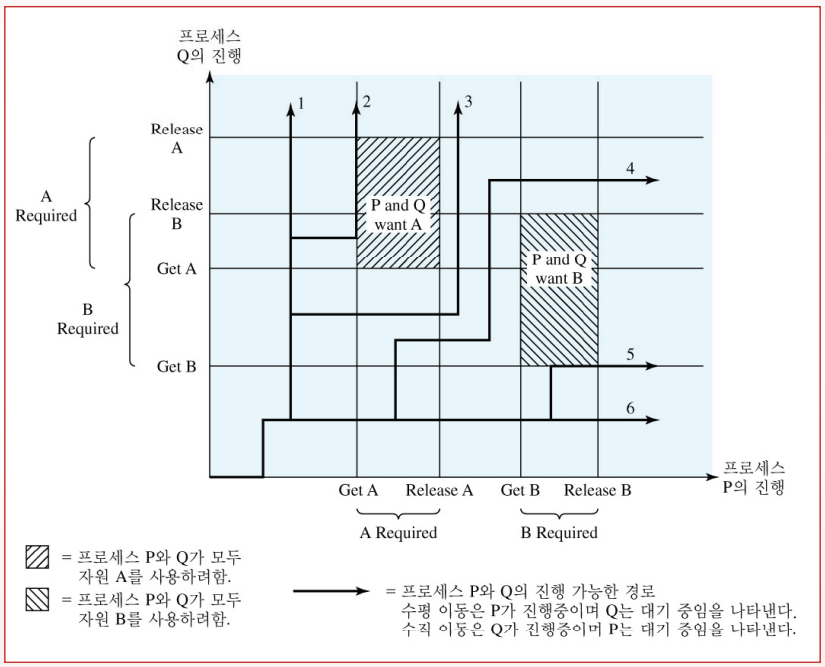
- 교착 상태를 쉽게 이해하기 위한 결합 진행 다이어그램 예



결합 진행 다이어그램: alternative

- 교착상태가 발생하지 않는 예

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Release A	Get A
• • •	• • •
Get B	Release B
• • •	• • •
Release B	Release A
• • •	• • •



재사용 가능 자원 (reusable)

- 프로세스가 사용했다고 없어지지는 않는 자원
- 사용 후 반납해야 함.
- 예: 처리기, I/O channels, 주/보조 메모리, 장치, 커널 자료구조(파일, 데이터베이스, 세마포어 등)
- 프로세스가 한 자원을 갖고 있으면서 다른 자원을 요구하면 교착상태가 발생할 수 있음
- 예
  - 메모리 할당 (가용 메모리 크기 = 200Kbytes)
  - 디스크와 테이프를 이용한 백업

P1 // 메모리 할당	P2
... Request 80Kbytes ... Request 60Kbytes	... Request 70Kbytes ... Request 80Kbytes

Process P		Process Q	
Step	Action	Step	Action
p_0	Request (D)	q_0	Request (T)
p_1	Lock (D)	q_1	Lock (T)
p_2	Request (T)	q_2	Request (D)
p_3	Lock (T)	q_3	Lock (D)
p_4	Perform function	q_4	Perform function
p_5	Unlock (D)	q_5	Unlock (T)
p_6	Unlock (T)	q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

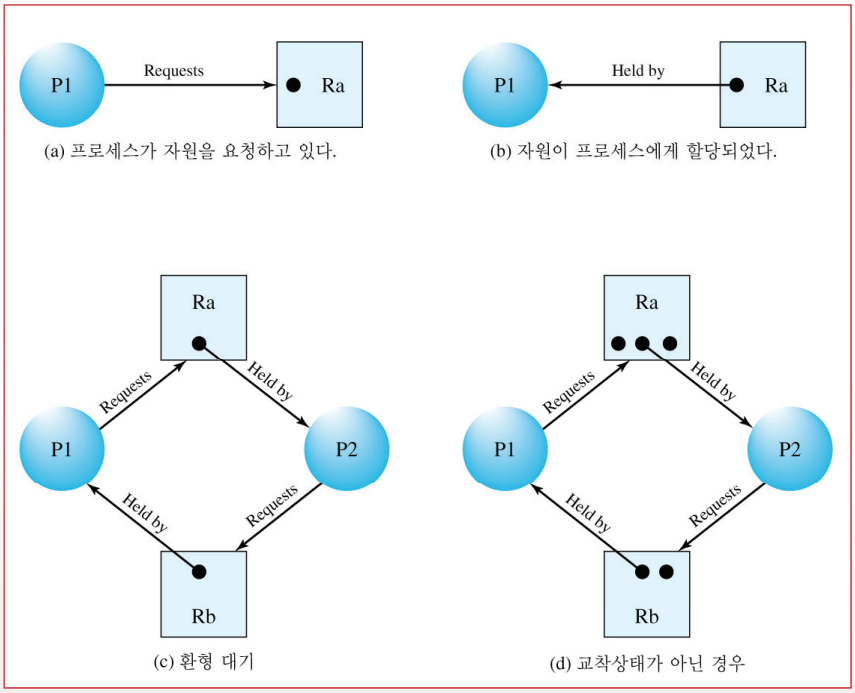
소모성 자원 (consumable)

- 생성되었다가 사용된 후 소멸되는 자원
  - 생산자와 소비자 프로세스
- 예: 인터럽트, 시그널, 메시지, I/O 버퍼에 있는 정보
- 메시지 수신이 blocking 방식으로 이루어지면 교착상태가 발생할 수 있음
- 예: 통신 시 발생하는 교착상태

P1 // 통신	P2
... Receive (P2) ... Send (P2, M1)	... Receive (P1) ... Send (P1, M2)

자원 할당 그래프(Resource Allocation graph)

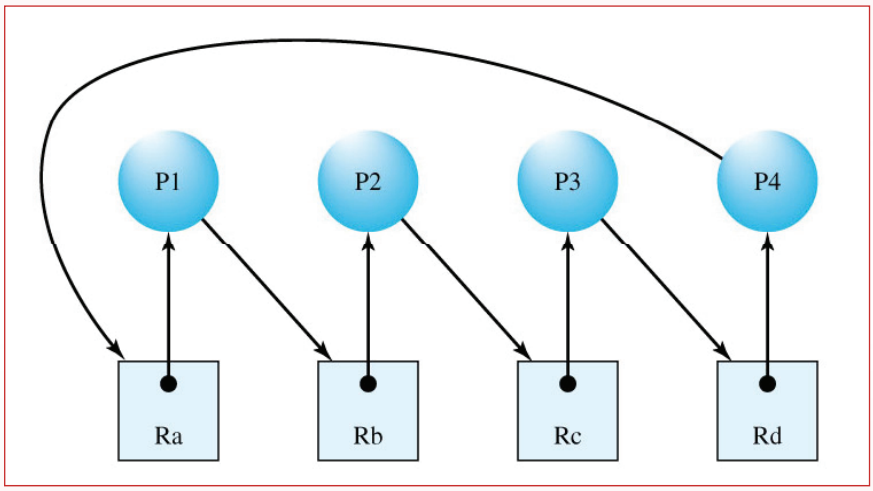
- **자원 할당 그래프 예**



(a) 프로세스가 자원을 요청하고 있다.
(b) 자원이 프로세스에게 할당되었다.
(c) 환형 대기
(d) 교착상태가 아닌 경우

자원 할당 그래프(Resource Allocation graph)

- **교차로 교착상태의 RAG 표현**

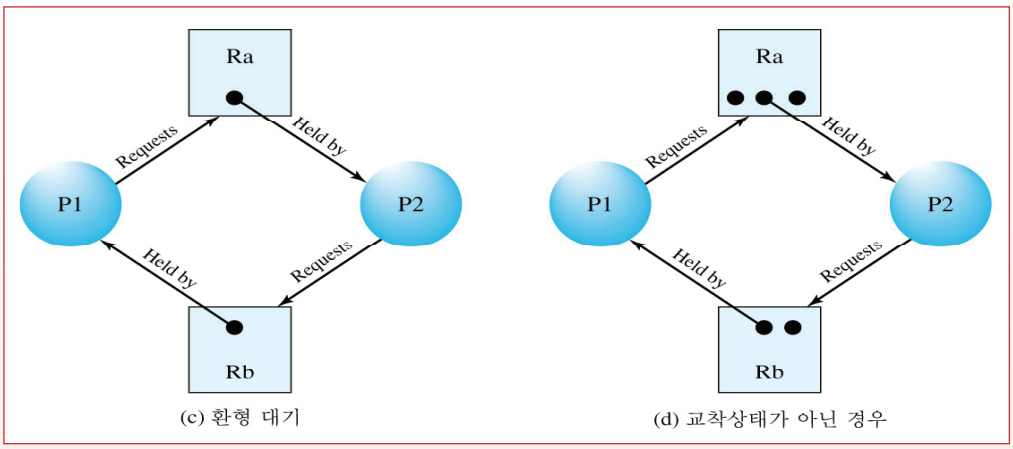


– Cycle이 발생하고 있음

교착상태 조건

- **교착상태가 발생하기 위한 필요충분조건 4가지**
  - ***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

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 강제적으로 자원을 반납시킴
- **환형 대기**
  - 자원 할당 순서(자원 유형)를 미리 정해두면 없앨 수 있음.

교착상태 예방의 예

```
/* DeadLock example.c */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int a[4] = {150,100,200,50};          // account
int b[4] = {1, 1, 2, 0};              // credit status
```

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

    ...  // pre processing
    entercritical(a)
    tmp1 = a[1];
    tmp1 = tmp1 + 10;
    if (tmp1 > 200) {
        entercritical(b)
         tmp2 = b[1];
         tmp2 += 1;
         b[1] = tmp2;
        exitcritical(b)
    }
    a[1] = tmp;
    exitcritical(a)
    ... // post processing
}
```

```
void *func2() {              // credit evaluation
    int tmp1, tmp2;

    ...  // pre processing
    entercritical(b)
    tmp2 = b[1];
    if (tmp2 >= 2) {
        entercritical(a)
         tmp1 = a[1];
         tmp1 =  tmp1 * 0.05;
         a[1] = tmp1;
        exitcritical(a)
    }
    exitcritical(b)
    ... // post processing
}
```


6.3 교착상태 회피 (deadlock avoidance)

- **예방과 회피의 차이점**
  - 교착상태 예방은 자원의 사용과 프로세스 수행에 비효율성을 야기할 수 있다.
  - 교착상태 회피 기법은 교착 상태 발생을 위한 4가지 조건 중 1, 2, 3을 허용
  - 자원 할당 순서를 미리 정해놓지도 않음
  - 그 대신, ***자원을 할당할 때 교착 상태가 발생 가능한 상황으로 진행하지 않도록 고려***한다.
    - 프로세스 시작 시 요구하는 자원할당이 교착상태를 발생시킬 가능성이 있으면, 프로세스를 시작시키지 않는다.
    - 수행 중인 프로세스가 요구하는 추가적인 자원할당이 교착상태를 유발할 수 있으면 거부한다.

시스템 상태 구분

- **안전 상태(safe state): 교착상태가 발생하지 않도록 프로세스들에게 자원을 할당할 수 있는 할당 경로가 존재**
- **불안전 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. 즉, 안전한 상태를 계속 유지할 수 있을 때에만 자원 할당
  - 각 프로세스들이 필요한 자원들을 미리 운영체제에게 알려야 함
- **프로세스 시작 거부 (Process Initialization Denial)**
  - 교착상태가 발생할 가능성이 있으면 프로세스 시작 거부

Banker's Algorithm

- **안전 상태 판별 과정**

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

요구 행렬 C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

할당 행렬 A

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

C – A

R1	R2	R3
9	3	6

자원 벡터 R

R1	R2	R3
0	1	1

가용 벡터 V

(a) 초기 상태

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

요구 행렬 C

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

할당 행렬 A

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

C – A

R1	R2	R3
9	3	6

자원 벡터 R

R1	R2	R3
6	2	3

가용 벡터 V

(b) P2 수행 완료

Banker's Algorithm (계속)

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

요구 행렬 C

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

할당 행렬 A

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

C – A

R1	R2	R3
9	3	6

자원 벡터 R

R1	R2	R3
7	2	3

가용 벡터 V

(c) P1 수행 완료

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

요구 행렬 C

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

할당 행렬 A

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

C – A

R1	R2	R3
9	3	6

자원 벡터 R

R1	R2	R3
9	3	4

가용 벡터 V

(d) P3 수행 완료

Banker's Algorithm (계속)

- **불안전 상태 판별 예**

☞ **자원 할당 거부 (Banker's algorithm)**

☞ **프로세스 시작 거부**

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

요구 행렬 C

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

할당 행렬 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
9	3	6

자원 벡터 R

R1	R2	R3
1	1	2

가용 벡터 V

(a) 초기 상태

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

요구 행렬 C

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

할당 행렬 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

자원 벡터 R

R1	R2	R3
0	1	1

가용 벡터 V

(b) P1이 자원 R1과 R3을 하나씩 할당받은 상태

Banker's Algorithm (계속)

- 은행원 알고리즘 구현 예

```
struct state {
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

(a) 전역 자료구조

```
if (alloc [i,*] + request [*] > claim [i,*])
      < error >;                    /* 전체 요청이 요구한 것보다 큼 */
else if (request [*] > available [*])
      < suspend process >;
else   {                                /* 자원 할당을 시뮬레이션  */
      < define newstate by:
      alloc [i,*] = alloc [i,*] + request [*];
      available [*] - available [*] - request [*] >;
;
if (safe (newstate))
      < carry out allocation >;
else {
      < restore original state >;
      < suspend process >;
;
```

(b) 자원 할당 알고리즘

Banker's Algorithm (계속)

- 은행원 알고리즘 구현 예 (계속)

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

(c) 안전한 상태 확인 알고리즘(은행원 알고리즘)

그림 6.9 교착상태 회피 논리

6.4 교착상태 발견 (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.
  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. 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

교착상태 회복(recovery) 알고리즘

① **교착상태 관련 모든 프로세스 종료**

② **교착상태에 연관된 프로세스들을 체크포인트 시점으로 롤백한 후 다시 수행 시킴 (교착상태가 다시 발생할 가능성 존재)**

③ **교착상태가 없어질 때까지 교착상태에 포함되어 있는 프로세스들 하나씩 종료**

④ **교착상태가 없어질 때까지 교착상태에 포함되어 있는 자원을 하나씩 빼앗음**

⑤ **종료 (또는 선점될) 프로세스 선택 기준**
- 지금까지 사용한 처리기 시간이 적은 프로세스부터
- 지금까지 생산한 출력량이 적은 프로세스부터
- 이후 남은 수행시간이 가장 긴 프로세스부터
- 할당 받은 자원이 가장 적은 프로세스부터
- 우선 순위가 낮은 프로세스부터

6.5 통합적인 교착상태 전략

- **교착상태 예방, 회피, 발견**

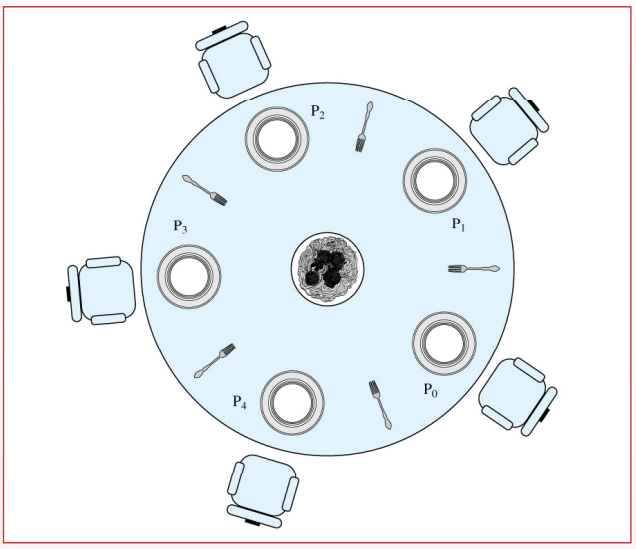
표 6.1 | 운영체제에서 교착상태 예방, 회피, 발견 기법 정리[ISL080]

접근 방법	자원할당 정책	구체적인 기법	장점	단점
예방	보수적 (자원할당이 가능하더라도조건에 따라 할당하지 않을 수 있다.)	모든자원을 한꺼번에 요구	• 순간적으로 많은 일을 하는 프로세스에 적합 • 선점이 불필요	• 효율이 나쁨. • 프로세스 시작을 지연시킬 가능성 있음. • 프로세스는 사용할 모든 자원을 미리 알고 있어야 함.
		선점 가능	• 자원 상태의 저장과 복구가 간단한 자원에는 적용하기 쉬움.	• 선점이 필요보다 자주 일어남.
		자원할당 순서	• 컴파일 시점에 강제할 수 있음 • 시스템의 설계 시점에 문제를 해결했기 때문에 동적 부하가 없음.	• 점진적인 자원 할당이 안 됨.
회피	예방과 발견의 중간 정도	교착상태가 발생하지 않는 안전한 경로를 최소한 하나는 유지	• 선점이 불필요	• 운영체제는 자원에 대한 미래 요구량을 미리 알고 있어야 함. • 오랜 기간 지연 발생의 가능성 있음.
발견	적극적 (자원할당이가 능하면 즉시 할당한다.)	주기적으로 교착상태 발생 여부 파악	• 프로세스 시작을 지연시키지 않음 • 온라인 처리 가능	• 선점에 의한 손실 발생

6.6 식사하는 철학자 문제

- **문제 정의**
  - 철학자들은 반드시 포크 2개를 사용해야 함.
  - 포크는 여러 철학자들에 의해 공유될 수 없음(mutual exclusion)
  - 어떤 철학자도 굶어 죽어서는 안 된다.

➔ ***교착상태도 없어야 하고, 굶어 죽지도 않아야 한다.***



세마포어를 이용한 해결 방법

```
/* 식사하는 철학자 프로그램 */
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));
     }
```

그림 6.12 세마포어를 이용한 식사하는 철학자 문제 해결 방법: 첫 번째 버전

☞ 이 방법에 내재되어 있는 문제점은?

세마포어를 이용한 다른 해결 방법

- **교착상태 발생하지 않는 버전**
  - 한 번에 최대 4명까지 철학자가 식탁에 앉을 수 있게 제한

```
/* 식사하는 철학자 프로그램 */
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));
}
```

그림 6.13 세마포어를 이용한 식사하는 철학자 문제 해결 방법: 두 번째 버전

모니터를 이용한 해결 방법

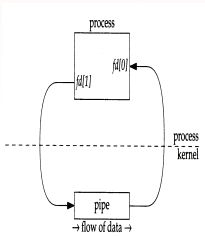
```
monitor dining_controller;
cond ForkReady[5];                          /*  동기화를 위한 조건 변수  */
boolean fork[5] = {true};                   /*  각 포크의 사용가능 여부  */

void get_forks(int pid)                     /*   pid는 철학자 id의 수   */
{
   int left = pid;
   int right = (++pid) % 5;
   /*  왼쪽에 놓인 포크 접근  */
   if (!fork(left)
      cwait(ForkReady[left]);                   /*  조건 변수에서 대기 */
   fork(left) = false;
   /* 오른쪽에 놓인 포크 접근 */
   if (!fork(right)
      cwait(ForkReady(right);                   /*  조건 변수에서 대기 */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*  왼쪽에 놓인 포크 반납  */
   if (empty(ForkReady[left])              /*  대기하는 프로세스가 없음  */
      fork(left) = true;
   else                                    /*  대기하던 프로세스를 깨움  */
      csignal(ForkReady[left]);
   /* 오른쪽에 놓인 포크 반납 */
   if (empty(ForkReady[right])             /*  대기하는 프로세스를 깨움  */
      fork(right) = true;
   else                                    /*  대기하던 프로세스를 깨움  */
      csignal(ForkReady[right]);
}
```

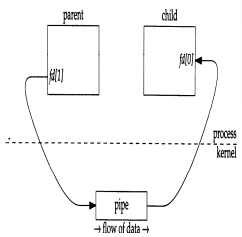
```
void philosopher[k=0 to 4]                     /*  5명의 철학자 존재  */
{
   while (true) {
      <think>;
      get_forks(k);            /*  철학자가 모니터를 통해 두 개의 포크 요청  */
      <eat spaghetti>;
      release_forks(k);        /*  철학자가 모니터를 통해 두 개의 포크 반납  */
   }
}
```

그림 6.14 모니터를 이용한 식사하는 철학자 문제 해결 방법

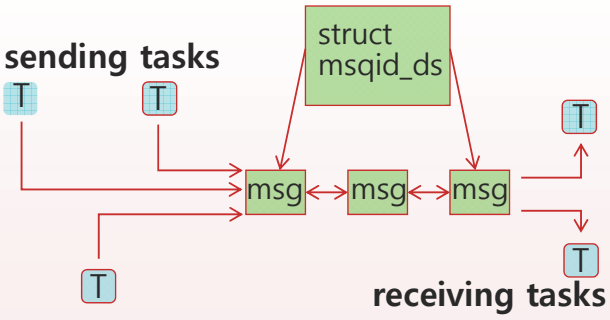
유닉스 병행성 기법

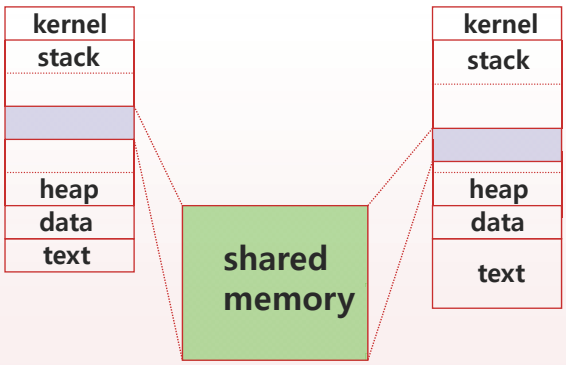
- **프로세스간 통신 (IPC: InterProcess Communication)**
  - 시그널(Signal)
  - 파이프(Pipe)





  - 메시지 전달(Message passing)
  - 공유 메모리(Shared memory)
  - 세마포어(Semaphore)





유닉스 병행성 기법

- **시그널**

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	세그먼테이션 오류. 프로세스가 자신에게 할당된 이외의 주소 참조
12	SIGSYS	시스템 호출에서 잘못된 인자 사용
13	SIGPIPE	파이프 오류. 파이프에서 데이터를 읽으려는 프로세스가 없는데, 그 파이프로 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 결함

☞ 인터럽트와 유사점 및 차이점은?

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

리눅스 커널 병행성 기법

- **원자적 연산 (atomic operation)**

표 6.3 | 리눅스의 원자적 연산

원자적 정수 연산	
ATOMIC_INIT (int i)	i변수가 원자성을 갖도록 선언
int atomic_read(atomic_t *v)	v에서 정수형 값을 읽음
void atomic_set(atomic_t*v, int i)	정수형 변수 i에 값 v를 설정
void atomic_add(int i, atomic_t *v)	i를 v에 더함
void atomic_sub(int i,atomic_t *v)	v에서 i를 뺌
void atomic_inc(atomic_t *v)	v를 1만큼 증가시킴
void atomic_dec(atomic_t *v)	v를 1만큼 감소시킴
int atomic_sub_and_test(int i, atomic_t *v)	v에서 i를 뺌. 그 결과가 0이면 1을 리턴, 아니면 0을 리턴
int atomic_add_negative(int i, atomic_t *v)	v에서 i를 더함. 그 결과가 음수이면 1을 리턴, 아니면 0을 리턴 (세마포어를 구현할 때 사용)
int atomic_dec_and_test(atomic_t *v)	v에서 1을 뺌. 그 결과가 0이면 1을 리턴, 아니면 0을 리턴
int atomic_inc_and_test(atomic_t *v)	v에 1을 더함. 그 결과가 0이면 1을 리턴, 아니면 0을 리턴
원자적 비트 연산	
void set_bit(int nr, void *addr)	addr이 가리키는 비트맵에서 nr 비트를 설정(set)
void clear_bit(int nr, void *addr)	addr이 가리키는 비트맵에서 nr 비트를 설정 해제(clear)
void change_bit(int nr, void *addr)	addr이 가리키는 비트맵에서 nr 비트를 반전(invert)
int test_and_set_bit(int nr, void *addr)	addr이 가리키는 비트맵에서 nr 비트를 설정하고, 그 이전의 비트 값을 리턴
int test_and_clear_bit(int nr, void *addr)	addr이 가리키는 비트맵에서 nr 비트를 설정 해제하고, 그 이전의 비트 값을 리턴
int test_and_change_bit(int nr, void *addr)	addr이 가리키는 비트맵에서 nr 비트를 반전하고, 그 이전의 비트 값을 리턴
int test_bit(int nr, void *addr)	addr이 가리키는 비트맵에서 nr 비트의 값을 리턴

리눅스 커널 병행성 기법

- **스핀 락 (Spinlocks)**
  - 기본 스핀 락 (plain, irq, irqsave, bh), 읽기-쓰기 스핀 락

표 6.4 | 리눅스 스핀 락

void spin_lock(spinlock_t *lock)	스핀 락 획득 시도. 만일 가용하지 않으면 바쁜 대기 수행
void spin_lock_irq(spinlock_t *lock)	spin_lock과 기본적으로 동일. 추가적으로 수행되는 처리기에서 인터럽트를 허용하지 않음
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)	spin_lock_irq와 기본적으로 동일. 추가적으로 현재 인터럽트 상태를 저장
void spin_lock_bh(spinlock_t *lock)	spin_lock과 기본적으로 동일. 추가적으로 리눅스의 하반부(bottom half) 처리를 허용하지 않음
void spin_unlock(spinlock_t *lock)	스핀 락을 반납
void spin_unlock_irq(spinlock_t *lock)	스핀 락을 반납하면서 인터럽트 허용
void spin_unlock_irqrestore (spinlock_t *lock, unsigned long flags)	스핀 락을 반납하면서 저장했던 인터럽트 상태를 복구
void spin_unlock_bh(spinlock_t *lock)	스핀 락을 반납하면서 리눅스의 하반부(bottom half) 처리를 허용
void spin_lock_init(spinlock_t *lock)	스핀 락 초기화
int spin_trylock(spinlock_t *lock)	스핀 락 획득 시도. 얻을 수 있으면 0을 리턴, 얻을 수 없으면 0이 아닌 값을 리턴
int spin_is_locked(spinlock_t *lock)	현재 스핀 락 획득이 가능하면 0을 리턴. 아니면 0이 아닌 값을 리턴

리눅스 커널 병행성 기법

- **세마포어 (Semaphores)**
  - 기본 세마포어 (up, down), 읽기-쓰기 세마포어

표 6.5 | 리눅스 세마포어

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

리눅스 커널 병행성 기법

- **장벽(Barriers)**

표 6.6 | 리눅스 메모리 장벽 연산

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

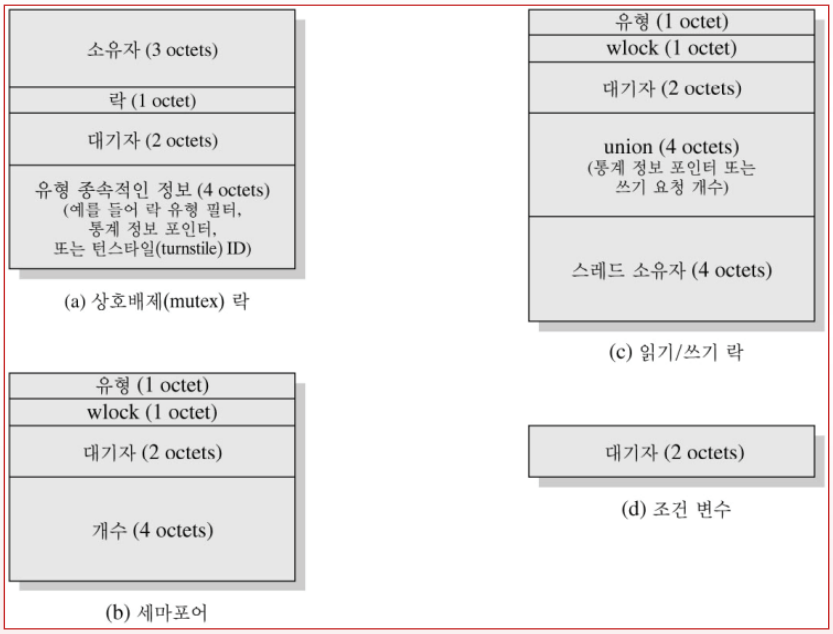
SMP = 대칭적 다중처리기
UP = 단일처리기

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.
  - The thread that locks the mutex must be the one that unlocks it
- **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

솔라리스 스레드 동기화 프리미티브

- **구조**



윈도우즈 커널 병행성 기법

- **객체 아키텍처의 일부분으로 동기화 기법 제공**
  - Executive 디스패처 객체 (대기 함수(Wait functions) 사용)

표 6.7 | Windows 동기화 객체

객체 유형	정의	시그널 받는 시점	대기 쓰레드의 영향
알림 사건 (notification event)	시스템 사건의 발생을 알림.	쓰레드가 사건을 설정	모두 깨어남
동기화 사건	시스템 사건의 발생을 알림.	쓰레드가 사건을 설정	한 쓰레드가 깨어남
상호 배제	상호 배제 기능 제공. 이진 세마포어와 동일	소유자(또는 다른쓰레드)로부터 mutex 만남	한 쓰레드가 깨어남
세마포어	자원을 사용할 수 있는 쓰레드의 개수를 조절하는 카운터	세마포어 카운터의 값이 0으로 됨	모두 깨어남
대기가능 타이머	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 쓰레드 종료	모두 깨어남
쓰레드	프로세스 내부의 수행 객체	쓰레드 종료	모두 깨어남

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

  - 임계영역 (Critical Section)
  - Slim 읽기-쓰기 락
  - 조건 변수

윈도우즈 커널 병행성 기법

- **대기 함수 (Wait functions)**
  - The wait functions allow a thread to block its own execution.
  - The wait functions do not return until the specified criteria have been met.
- **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

윈도우즈 리눅스 비교

WINDOWS/LINUX 비교	
Windows Vista	**Linux**
세마포어, 상호 배제, 스핀 락, 타이머 같은 공통 동기화 프리미티브가 대기/신호 메커니즘을 기반으로 한다.	세마포어, 상호 배제, 스핀 락, 타이머 같은 공통 동기화 프리미티브가 sleep/wakeup 메커니즘을 기반으로 한다.
"많은 커널 객체는 디스패처 객체이다"라는 것은 쓰레드가 공통 이벤트 메커니즘에 의해 동기화된다는 것을 의미한다. 이벤트 메커니즘은 사용자 수준에서도 사용가능하다. 프로세스와 쓰레드의 종료, I/O 완료 모두 이벤트이다.	
쓰레드는 한 순간에 여러 개의 디스패처 객체를 기다릴 수 있다.	프로세스는 select( ) 시스템 호출을 사용하여 최대 64개의 파일 디스크립터에서 I/O를 기다릴 수 있다.
사용자 수준 읽기/쓰기 락과 조건 변수가 지원된다.	사용자 수준 읽기/쓰기 락과 조건 변수가 지원된다.
원자적 증가/감소, compare-and-swap 등 다양한 하드웨어 원자적 연산이 지원된다.	원자적 증가/감소, compare-and-swap 등 다양한 하드웨어 원자적 연산이 지원된다.
Compare-and-swap을 이용하여, 비블록킹 원자적 LIFO 큐를 지원한다. 이 큐는 SLIST라고 불린다. OS와 사용자 프로그램이 이것을 자주 사용한다.	

윈도우즈 리눅스 비교

확장성을 증가시키기 위한 다양한 동기화 기법이 커널 수준에 존재한다. 대부분은 compare-and-swap 메커니즘을 기반으로 하며, 대표적인 예로, 푸시 락(push-lock), 객체의 빠른 참조 등이 있다.	
이름 있는 파이프, 소켓 등은 원격 프로시저 호출(RPC: Remote Procedure Call)을 지원한다.이것은 마치 지역 시스템 내부에서 사용하는 ALPC(An efficient Local Procedure Call)처럼 동작한다. ALPC는 고객과 지역 서비스 간의 통신을 위해 많이 사용된다.	이름 있는 파이프, 소켓 등은 원격 프로시저 호출(RPC: Remote Procedure Call)을 지원한다.
APCs(Asynchronous Procedure Calls)도 커널에서 주로 쓰레드들의 상호작용에 많이 사용된다(예를 들어, 종료나 I/O 완료를 APCs로 구현하는데, 이 사건들은 쓰레드들 간의 문맥보다는 한 쓰레드 문맥에서 구현하는 것이 더 간결하기 때문이다). APCs는 또한 사용자 수준에서도 사용가능한데, 사용자 수준 쓰레드가 커널에서 블록되었을 때만 APCs가 전달된다.	UNIX는 프로세스 간 통신을 위해 범용 시그널(signal) 메커니즘을 제공한다. 시그널은 하드웨어 인터럽트와 유사하다. 또한 언제든 전달 가능하며, 수신 프로세스에 의해 블록 당하지 않는다. 하드웨어 인터럽트처럼 시그널 의미는 다중쓰레딩에 의해 상당히 복잡해진다.
인터럽트 레벨이 내려갈 때까지 인터럽트 처리를 미루는 하드웨어 지원이 DPC(Deferred Procedure Call) 제어 객체에 의해 지원된다.	인터럽트 레벨이 내려갈 때까지 인터럽트 처리를 미루는 기능이 *tasklet*에 의해 지원된다.

요 약

- **교착상태 원리**
  - 교착상태 4가지 조건
  - 자원할당 그래프
- **교착상태 해결**
  - 예방(Prevention)
  - 회피(Avoidance)
  - 발견(Detection)
- **식사하는 철학자 문제**
- **사례 연구**
  - 유닉스, 리눅스 커널, 솔라리스, 윈도우즈