

## 3장. 병행성(Concurrency)

- 목 차 -

**3.1 개요**

**3.2 임계영역**

**3.3 프로세스간 통신**

**3.4 리눅스 병행성**

## 3.1 개 요

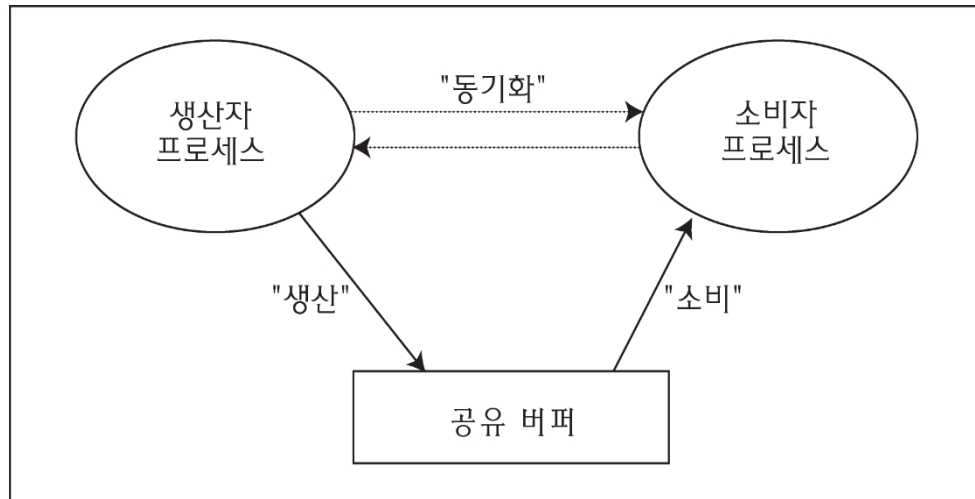
### ❖What is a process concurrency?

“여러 개의 프로세스들이 동시에 실행되는 것”

- Multiprogramming : The management of multiple processes within a uniprocessor system.
- Multiprocessing : The management of multiple processes within a multiprocessor system.

### ❖ 프로세스의 결정성(determinacy) : 동일한 입력에 대한 프로세스의 실행결과는 항상 일정하여야 한다

## 생산자/소비자 프로세스



- ✓ 두 프로세스는 상호 영향을 주고 받는다.
  - ✓ 두 프로세스에 대한 결정성 보장이 요구됨.
- ===> 동기화(synchronization) !!!

### 공유 변수

```
char buffer[N]; /* 환형 큐 구조의 공유 버퍼 */  
int counter = 0; /* 공유 버퍼에 저장된 데이터 개수 */
```

### 생산자 프로세스

```
for(;;){  
    while(counter == N) /* 공유 버퍼가 꽉 차면 기다림 */  
        ;  
    buffer[in] = nextp; /* 공유 버퍼에 데이터 저장 */  
    in = (in + 1) % N;  
    counter = counter + 1; /* 공유 버퍼에 저장된 데이터 개수 증가 */  
}
```

### 소비자 프로세스

```
for(;;){  
    while(counter == 0) /* 공유 버퍼가 비어 있으면 대기 */  
        ;  
    nextc = buffer[out]; /* 공유 버퍼로부터 데이터 제거 */  
    out = (out + 1) % N;  
    counter = counter - 1; /* 공유 버퍼에 저장된 데이터 개수 감소 */  
}
```

counter = counter + 1;

(단계-1) mov reg, counter ;변수 counter 값을 reg으로 읽기  
(단계-2) add reg, 1 ;reg 값에 1 더하기  
(단계-3) mov counter, reg ;reg 값을 변수 counter에 쓰기

counter = counter - 1;

(단계-1) mov reg, counter ;변수 counter 값을 reg으로 읽기  
(단계-2) sub reg, 1 ;reg 값에서 1 빼기  
(단계-3) mov counter, reg ;reg 값을 변수 counter에 쓰기

## 경쟁 조건(race condition)이 발생할 경우

(과정-1) mov reg, counter (reg = 4, counter = 4)  
(과정-2) add reg, 1 (reg = 5, counter = 4)  
/\*\* 문맥 교환 발생 \*\*/  
(과정-3) mov reg, counter (reg = 4, counter = 4)  
(과정-4) sub reg, 1 (reg = 3, counter = 4)  
(과정-5) mov counter, reg (reg = 3, counter = 3)  
/\*\* 문맥 교환 발생 \*\*/  
(과정-6) : mov counter, reg (reg = 5, counter = 5)

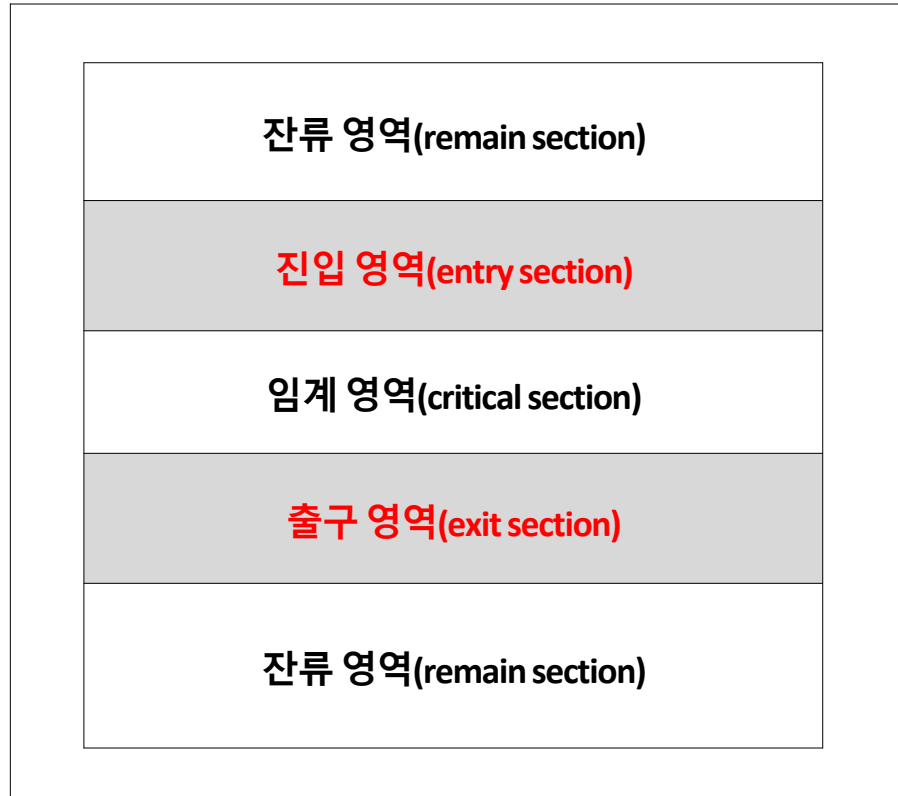
## ❖ 프로세스의 경쟁조건(Race Condition)이란?

“다수의 프로세스들이 공유 메모리를 병행적으로 접근할 때,  
그 공유 메모리에 대한 접근이 발생하는 순서에 따라  
프로세스들의 실행 결과가 달라질 수 있는 상황”

When the several processes **access the same data concurrently**, the outcome of the execution depends on the particular order in which the access takes place is called a **race condition**.

## 3.2 임계영역(Critical Section)

“공유 자원을 접근하는 프로세스의 **코드영역**”



[그림 3.3] 프로세스의 코드 영역

## ❖ 임계영역 문제 해결책의 조건

### ① 상호배제(mutual exclusion) :

- ✓ 두 개 이상의 프로세스가 임계영역에 존재해서는 안 된다.

### ② 진행(progress) :

- ✓ 임계영역에 존재하는 프로세스가 없을 경우 임계영역에 진입할 수 있어야 한다.

### ③ 제한된 대기(bounded waiting) :

- ✓ 임계영역 진입하려는 프로세스가 무한정 기다리게 해서는 안 된다.



### 3.2.1 소프트웨어에 의한 방법

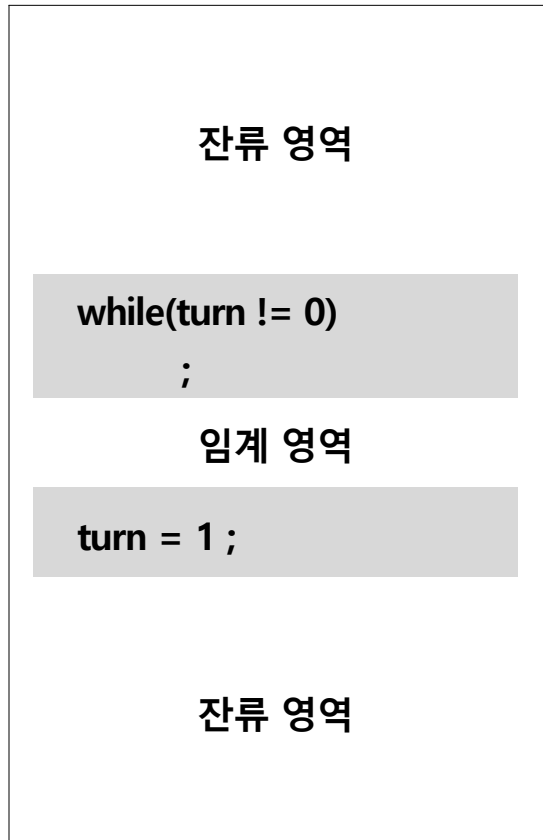
#### (1) 두 프로세스( $n=2$ )

1. Dikker's 알고리즘(1)
2. Dikker's 알고리즘(2)
3. Peterson's 알고리즘

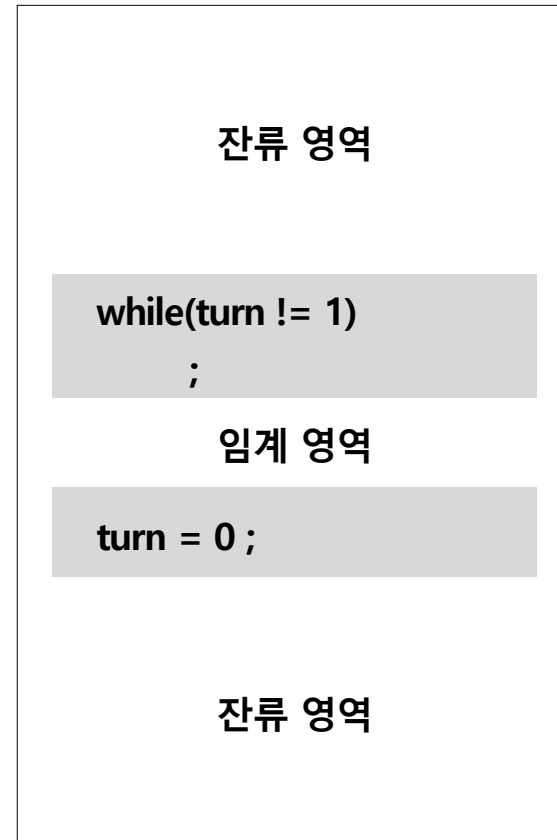
#### (2) 다중 프로세스( $n>2$ )

“Bakery 알고리즘”

## 1. Dikker's 알고리즘(1)

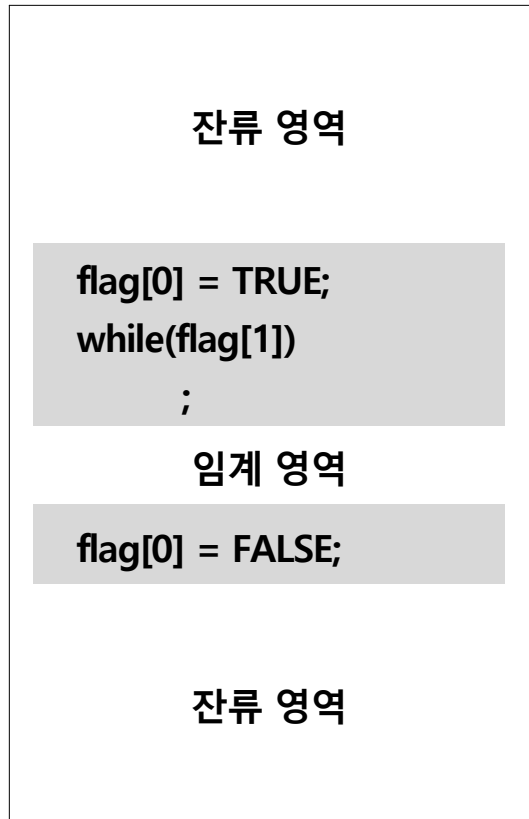


(a) 프로세스( $P_0$ )

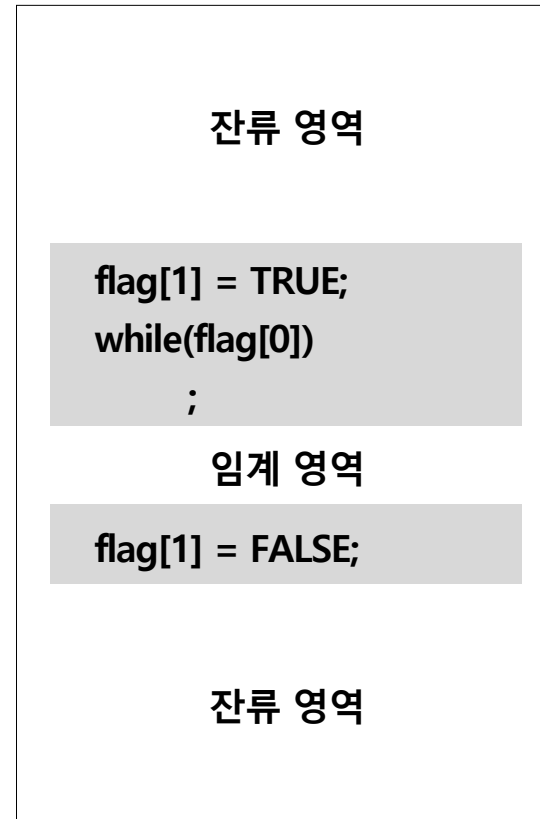


(b) 프로세스( $P_1$ )

## 2. Dikker's 알고리즘(2)

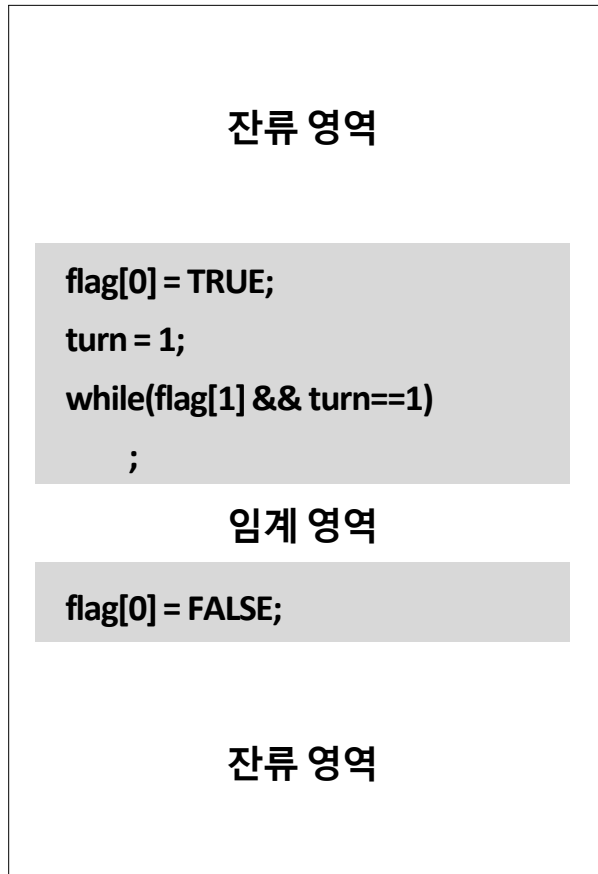


(a) 프로세스( $P_0$ )

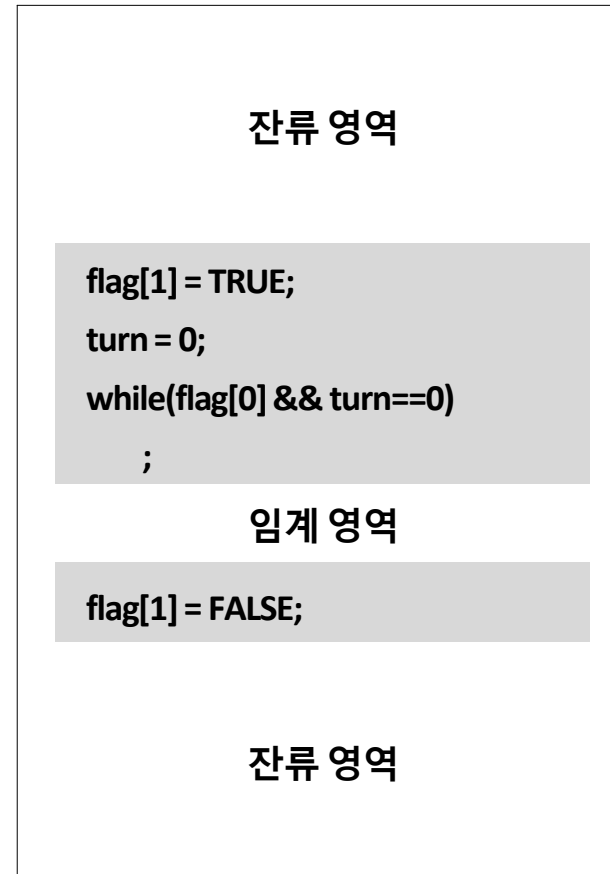


(b) 프로세스( $P_1$ )

### 3. Peterson 알고리즘



(a) 프로세스( $P_0$ )



(b) 프로세스( $P_1$ )

### 3.2.2 하드웨어에 의한 방법

“MOV, ADD와 같은 CPU 명령어가 처리되고 있는 동안에는 인터럽트 혹은 어떤 사건에 의한 문맥교환이 발생하지 않는다.”

이러한 하드웨어적인 기능을 이용하여 임계영역의 문제를 해결하는 방법이다. 이를 위하여 CPU 명령어가 필요하다.

- (1) Test-and-Set 명령어
- (2) Swap 명령어

## (1) Test-and-Set 명령어

```
bool Test_and_Set(bool *lock) {  
    bool ret = *lock;  
    *lock = TRUE;  
    return ret;  
}
```

잔류 영역

```
while(Test_and_Set( &flag ) ) ;
```

임계 영역

```
flag = FALSE;
```

잔류 영역

## (2) Swap 명령어

```
Swap(bool *a, bool *b) {  
    bool temp;  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

잔류 영역

```
key = TRUE;  
while( key ) Swap( &flag, &key );
```

임계 영역

```
flag = FALSE;
```

잔류 영역

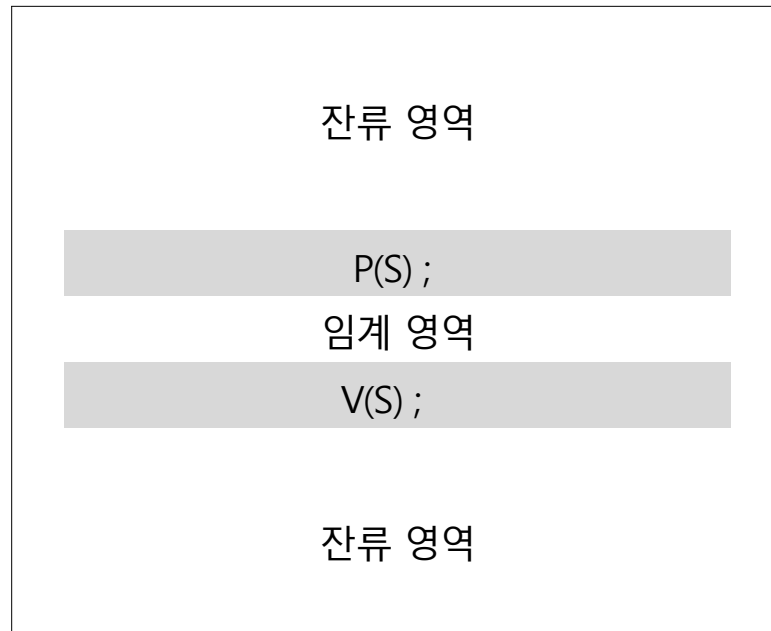
### 3.2.3 세마포어(Semaphore) 방법

- 세마포어(Semaphore) 란?

초기값이 부여한 후, 오직 P, V 연산자에 의해서만 변경할 수 있는 정수형 변수.

```
P(S):  while(S <= 0) ;  
        S = S - 1 ;
```

```
V(S):  S = S + 1 ;
```





### 3.2.3 세마포어(Semaphore) 의한 방법

- 바쁜대기(busy waiting)을 해결하기 위한 세마포어 변수 구조체 및 P, V 연산자

```
struct {  
    integer value;  
    struct pcb *ptr;  
} S;  
  
P(S): S.value = S.value - 1 ;  
    if(S.value < 0) {  
        이 프로세스를 S.ptr에 등록;  
        block();  
    }  
  
V(S): S.value = S.value + 1 ;  
    if(S.value <= 0) {  
        S.ptr로부터 프로세스(P)를 제거;  
        wakeup(P);  
    }  
}
```

### 3.3 프로세스간 통신(IPC: Inter-process Communication)

“서로 다른 프로세스들 사이에 데이터를 주고 받는 기법”

➔ 공유 메모리가 필요하다 !!!

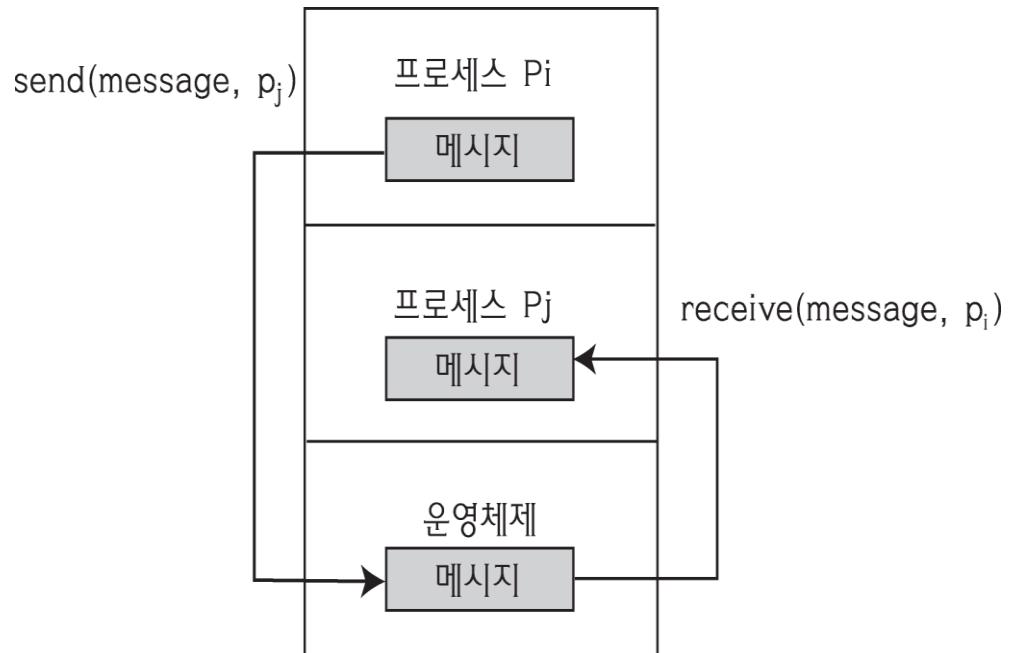
#### ❖ 프로세스간 통신 방식

(공유 메모리를 이용하는 방법에 따라)

1. 메시지 전송(message passing) : 커널 영역을 이용
2. 공유 메모리(shared memory) : 사용자 영역을 이용

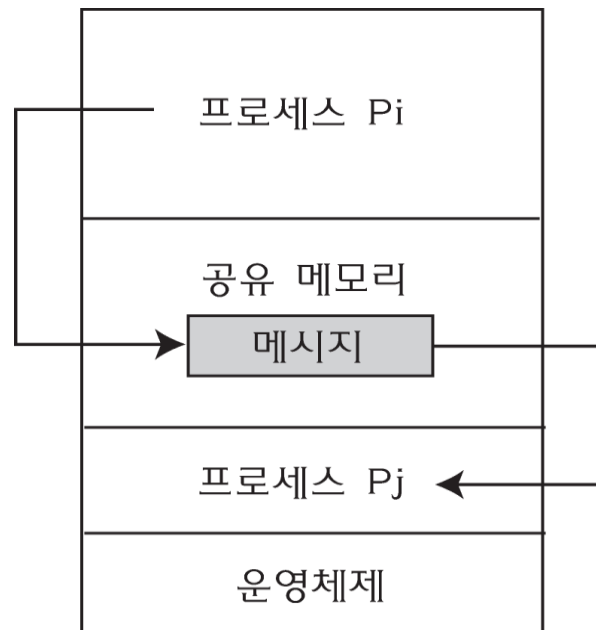
### 3.3.1 메시지 전송 방식

- ✓커널 영역을 공유 메모리로 이용한다.
- ✓메시지가 세 곳에 존재한다.



### 3.3.2 공유 메모리 방식

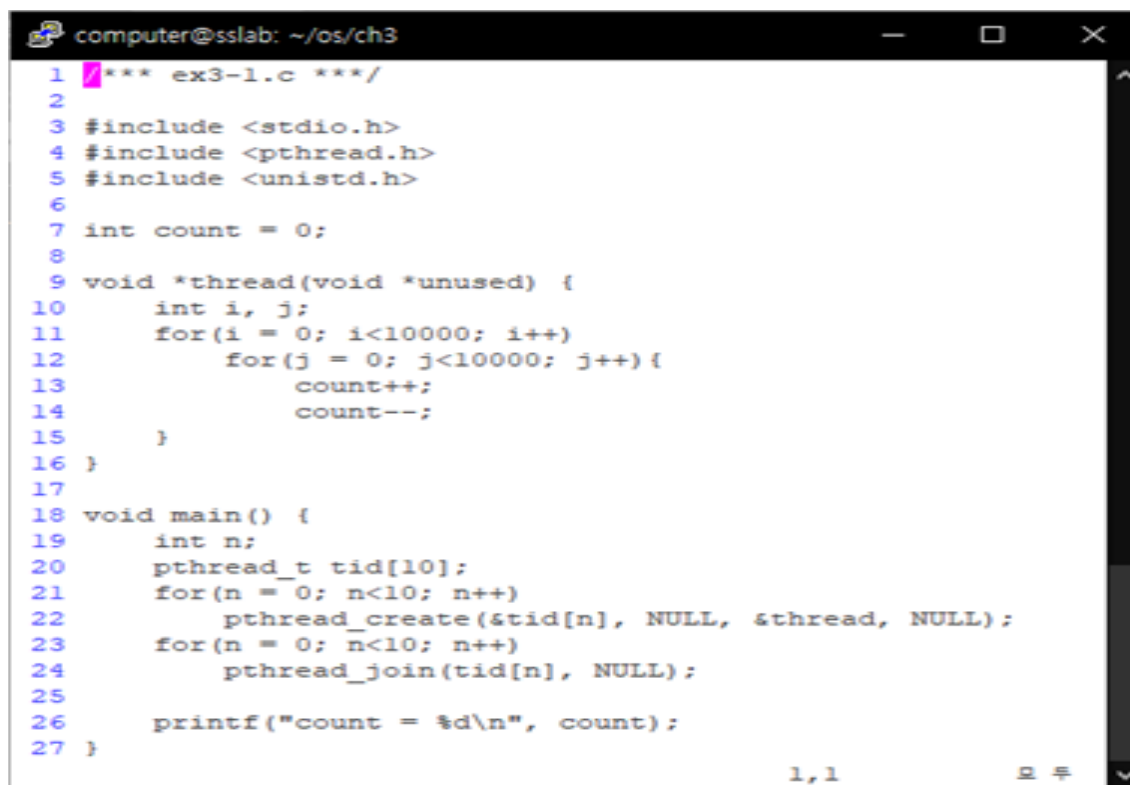
- ✓사용자 영역을 공유 메모리로 이용한다.
- ✓메시지가 한 곳에만 존재한다.



## 3.4 리눅스 병행성

### 3.4.1 스래드

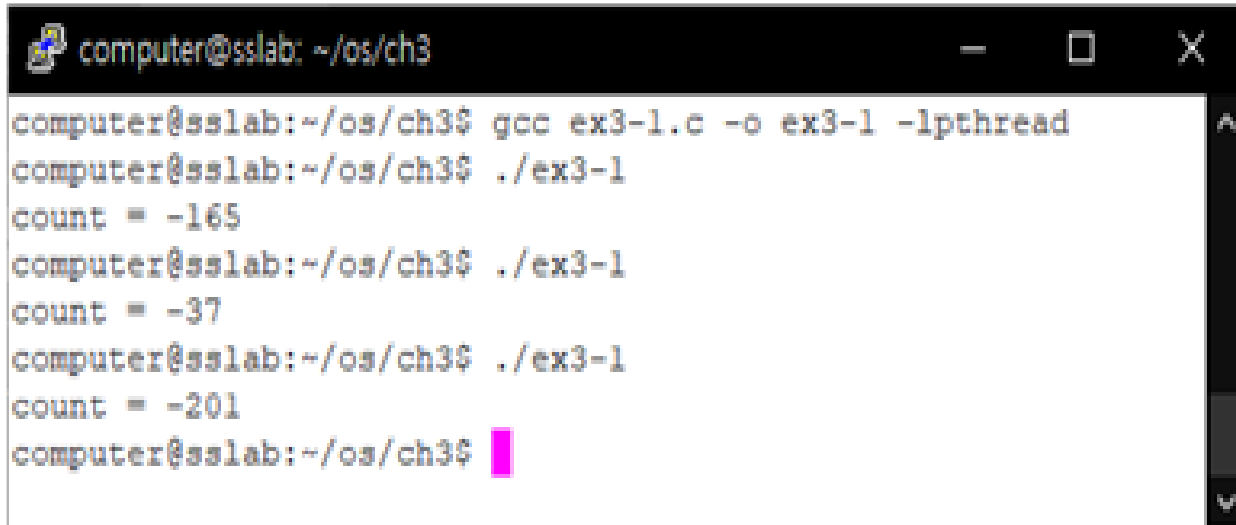
- ① pthread()에 의해 생성된 10개의 자식 스래드들이 병행적으로 실행된다.
- ② 각 스래드는 전역변수(count) 값을 1씩 증가와 감소를 반복한다.
- ③ 프로그램의 실행 결과 값은 항상 0이 되어야 할 것이다.



```
computer@sslab: ~/os/ch3
1  *** ex3-1.c ***/
2
3  #include <stdio.h>
4  #include <pthread.h>
5  #include <unistd.h>
6
7  int count = 0;
8
9  void *thread(void *unused) {
10     int i, j;
11     for(i = 0; i<10000; i++)
12         for(j = 0; j<10000; j++){
13             count++;
14             count--;
15         }
16 }
17
18 void main() {
19     int n;
20     pthread_t tid[10];
21     for(n = 0; n<10; n++)
22         pthread_create(&tid[n], NULL, &thread, NULL);
23     for(n = 0; n<10; n++)
24         pthread_join(tid[n], NULL);
25
26     printf("count = %d\n", count);
27 }
```

### 3.4.1 스래드

다음과 같이 컴파일링하여 실행시켜 본다.



```
computer@sslab: ~/os/ch3
computer@sslab:~/os/ch3$ gcc ex3-1.c -o ex3-1 -lpthread
computer@sslab:~/os/ch3$ ./ex3-1
count = -165
computer@sslab:~/os/ch3$ ./ex3-1
count = -37
computer@sslab:~/os/ch3$ ./ex3-1
count = -201
computer@sslab:~/os/ch3$
```

✓ 실행 결과가 일정하지 않다. 즉 결정성이 보장되고 있지 않다.

✓ 동기화가 필요함!!!

## (1) 스래드 뮤텝스

- 스래드 뮤텝스 사용법은 다음과 같다.

- ① 뮤텝스 타입 변수를 선언 및 속성을 초기화 한다.  
`pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- ② 뮤텝스 변수에 대한 잠금 연산을 시도한다.  
`pthread_mutex_lock(&mutex);`
- ③ 뮤텝스 변수에 대한 해제 연산을 시도한다.  
`pthread_mutex_unlock(&mutex);`

## (2) 스래드 세마포어

- 스래드 세마포어 사용법은 다음과 같다.

- ① 세마포어 타입 변수를 선언한다. 예) `sem_t s;`
- ② 세마포어 변수에 초기 값을 부여한다. 예) `sem_init(&s, 0, 5);` /\* 초기값 = 5 \*/
- ③ 세마포어 변수에 대한 P, V 연산을 시도한다.  
예) `sem_wait(&s);` /\* P 연산 \*/, `sem_post(&s);` /\* V 연산 \*/

```
sem_wait(S) : if(S == 0) then block();  
              S := S - 1;
```

```
sem_post(S) : S := S + 1;  
              if( (S == 1)&&(waiting_threads) then wakeup(t);
```



## 3.4.2 프로세스

리눅스에서 제공하고 있는 대표적인 IPC 기법과 동기화 기법.

- 파이프(PIPE)
- 메시지 큐(Message Queue)
- 공유 메모리(Shared Memory)
- 세마포어(semaphore)

※ 기본적으로 다음과 같은 과정을 거치게 된다.

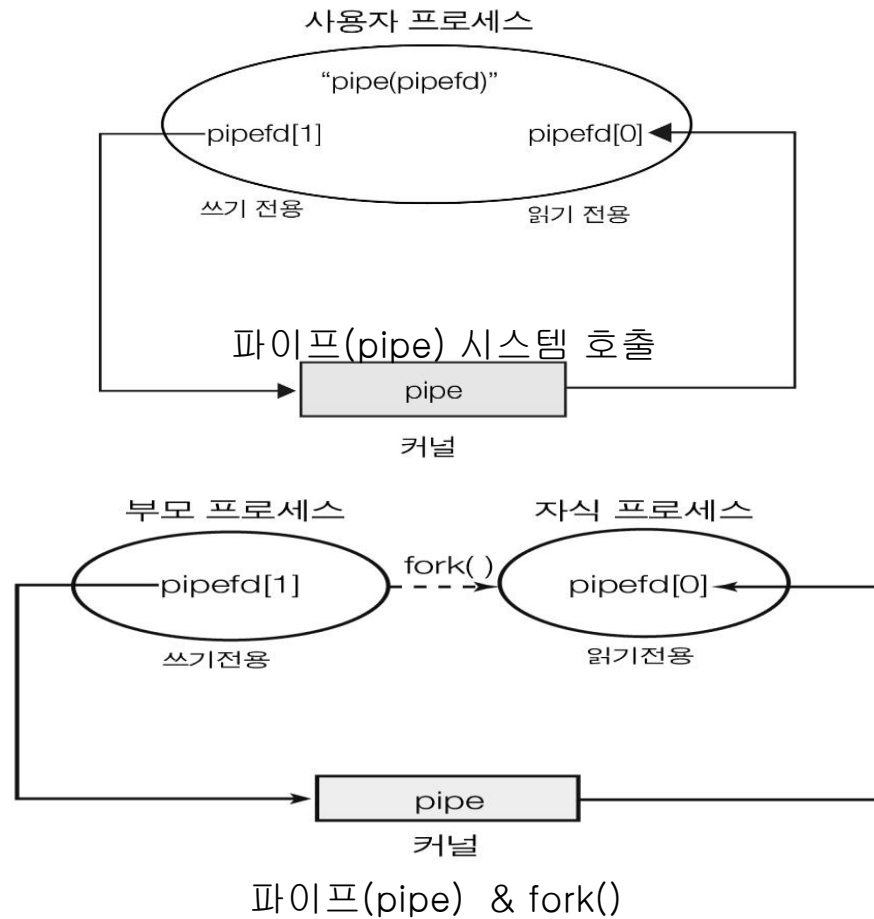
[과정-1] 공유 메모리 확보

[과정-2] 공유 메모리 접근

[과정-3] 공유 메모리 반납

# (1) 파이프(PIPE)

## 1) 파이프(unnamed PIPE)



## 1) 파이프(unnamed PIPE)

- 이름 없는 파이프를 통한 IPC 기법의 기본적인 과정은 다음과 같다.

### [과정-1] 공유 메모리 확보

- ① 읽기/쓰기 전용 변수를 선언한다.  
`int pipefd[2]; /* pipefd[0] 읽기, pipefd[1] 쓰기 */`
- ② 파이프 채널을 생성한다.  
`pipe(pipefd); /* 커널 내부에 공유 메모리 확보 */`
- ③ 자식 프로세스를 생성한다.  
`fork(); /* 공유 메모리의 주소를 공유 */`

### [과정-2] 공유 메모리 사용 : read(), write() 함수를 이용하여 데이터를 수신/전송한다.

- ① `read(pipefd[0], msgbuf, msg_length);`
- ② `write(pipefd[1], msgbuf, msg_length);`

### [과정-3] 공유 메모리 반납: 부모 프로세스가 종료될 때 반납된다.

## 2) 이름 있는 파이프(named pipe)

- 이름 있는 파이프를 통한 IPC 기법의 기본적인 과정은 다음과 같다.

[과정-1] 공유 메모리 확보 : FIFO 파일("fifo")을 생성한다.

```
mkfifo("fifo", 0666)  
/* 접근모드 0666인 "fifo"이름의 FIFO 파일생성 */
```

[과정-2] 공유 메모리 사용

① FIFO 파일("fifo")을 읽기/쓰기용으로 연다.

```
fd = open("fifo", O_RDONLY); /* 읽기용 */  
fd = open("fifo", O_WRONLY); /* 쓰기용 */
```

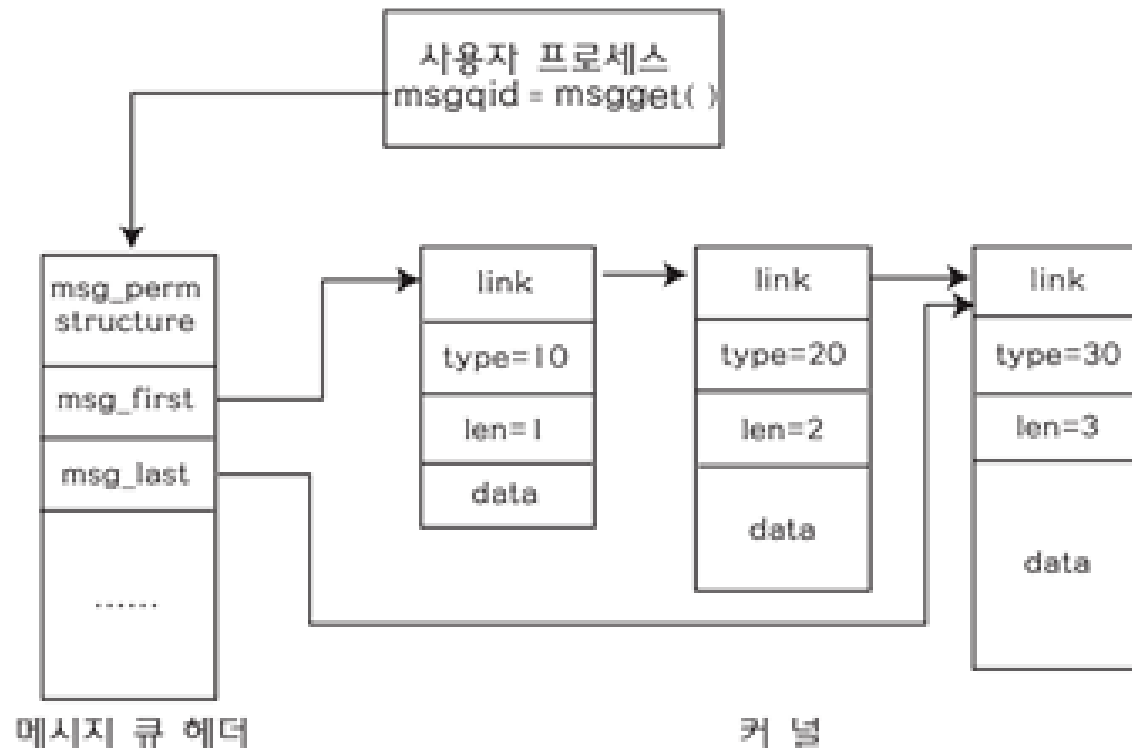
② read(), write() 함수를 이용하여 데이터를 수신/전송한다.

```
read(fd, msgbuf, msg_length); /* 메시지 수신 */  
write(fd, msgbuf, msg_length); /* 메시지 전송 */
```

[과정-3] 공유 메모리 반납: FIFO 파일("fifo")을 삭제한다.

```
unlink("fifo");
```

## (2) 메시지(Message)



## (2) 메시지 큐(Message Queue)

- 메시지 큐를 통한 IPC 기법의 기본적인 과정은 다음과 같다.

[과정-1] 공유 메모리 확보 : msgget() 함수를 이용하여 메시지 큐를 생성한다.

```
msgqid = msgget(key, msgflg) /* 메시지 큐 생성 */
```

[과정-2] 공유 메모리 사용 : msgrcv(), msgsnd() 함수를 이용하여 데이터를 송수신한다.

```
msgsnd(msgqid, *msgp, msgsz, msgflg);  
msgrcv(msgqid, *msgp, msgsz, msgty, msgflg);
```

[과정-3] 공유 메모리 반납: msgctl() 함수를 이용하여 메시지 큐를 삭제한다.

```
msgctl(msgqid, IPC_RMID, 0); /* 메시지 큐 삭제 */
```

### (3) 공유 메모리(Shared Memory)

- 공유 메모리를 통한 IPC 기법의 기본적인 과정은 다음과 같다.

[과정-1] 공유 메모리 확보 : 사용자 영역에 공유 메모리를 생성한다.

```
shmld = shmget(key, size, shmflg)
/* key 값에 해당하는 size 크기의 공유 메모리 생성 */
```

[과정-2] 공유 메모리 사용 : shmat() 함수를 이용하여 공유 메모리의 주소를 프로세스의 변수(local)에 연결한 후 변수에서 직접 읽기/쓰기 한다.

```
local = shmat(shmid, *shmaddr, shmflag);
/* shmid 공유 메모리를 프로세스의 변수(local)에 연결 */
```

[과정-3] 공유 메모리 반납:

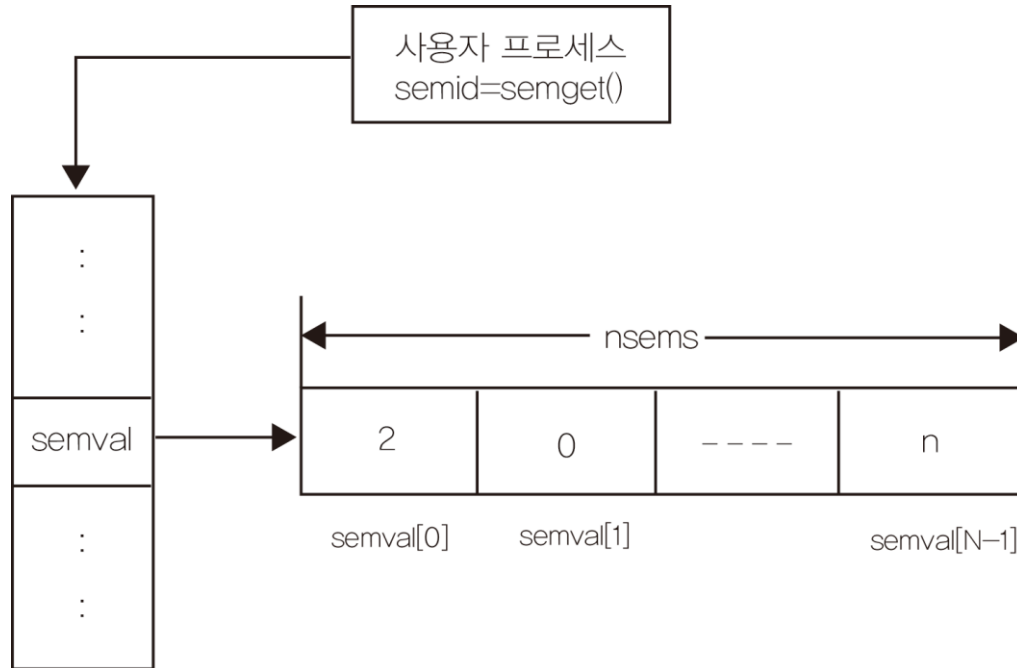
- ① shmdt() 함수를 이용하여 공유 메모리의 주소를 프로세스의 변수로부터 분리한다.

```
shmdt(local); /* 공유 메모리를 local로부터 분리 */
```

- ② shmctl() 함수를 이용하여 공유 메모리를 삭제한다.

```
shmctl(shmid, IPC_RMID, 0); /* 공유 메모리 삭제 */
```

## (4) 세마포어(Semaphore)





## (4) 세마포어(Semaphore)

- 세마포어를 이용한 프로세스 동기화 기법의 기본적인 과정은 다음과 같다.

[과정-1] 세마포어 생성: semget()을 이용하여 세마포어를 생성한다.

```
semid = semget(key, nsems, semflg)
/* key 값으로 nsems 개의 세마포어를 생성 */
```

[과정-2] 세마포어 초기 값 설정과 P,V 연산:

- ① semctl() 함수를 이용하여 세마포어 배열 semval[snum]의 초기 값(val)을 부여한다.

```
semctl(semid, snum, SETVAL, val);
```

- ② semop() 함수를 이용하여 P,V 연산을 수행한다.

```
struct sembuf p_op = {0, -1, SEM_UNDO};
struct sembuf v_op = {0, 1, SEM_UNDO};
semop(semid, &p_op, nsops); /* P 연산 */
semop(semid, &v_op, nsops); /* V 연산 */
```

[과정-3] 공유 메모리 반납: semctl() 함수를 이용하여 세마포어를 삭제한다.

```
semctl(semid, IPC_RMID, 0);
/* 세마포어(semid) 삭제(IPC_RMID) */
```