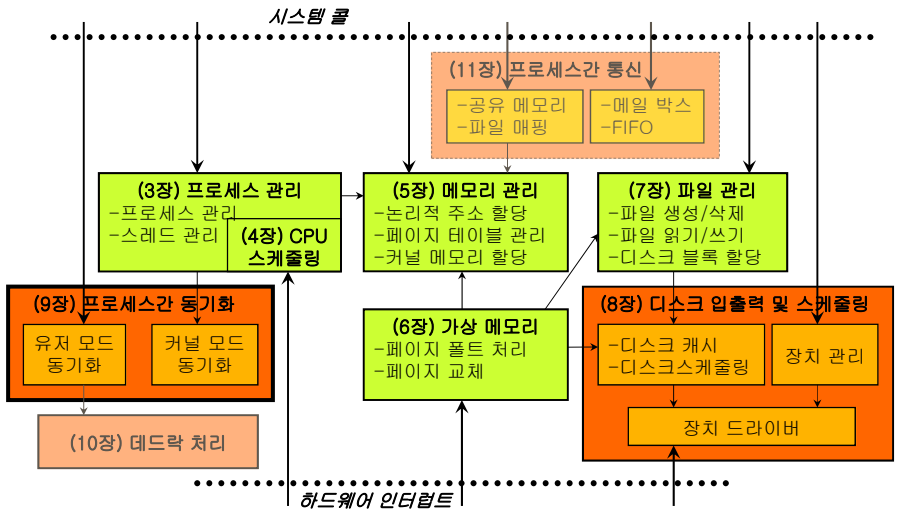
프로세스 간의 동기화

- ✓공유 데이터의 일관성 유지 문제
- ✓크리티컬 섹션
- ✓커널 내부만을 위한 기능
- ✓뮤텍스를 이용한 해법
- ✓세마포를 이용한 해법
- ✓Peterson의 알고리즘
- ✓유한버퍼 문제와 세마포의 활용
- ✓Readers-Writers 문제
- ✓크리티컬 리전
- ✓모니터와 컨디션 변수
- ✓picoKernel의 뮤텍스/세마포/컨디션 변수 구현

관련 운영체제 구성 모듈



이해하고 넘어가야 할 내용들

- ✓ 프로세스간의 공유 데이터 사용에서 발생하는 일관성 유지 문제의 이해와 크리티컬 섹션의 개념 이해
- ✓ 크리티컬 섹션 문제 해결을 위한 방안으로서 커널 내부에서만 사용 가능한 방법들과 유저 모드에서도 사용 가능한 방법들에 대한 이해
- ✓ 크리티컬 섹션 이외의 동기화 문제들과 이를 위한 세마포의 활용에 대한 이해
- ✓ 동기화 문제를 위해서 프로그래밍 언어 차원에서 제공하는 기능들에 대한 이해
- ✓ 뮤텍스, 세마포 및 컨디션 변수의 구현 방법에 대한 이해

공유 데이터의 일관성 유지 문제

- ✓ **공유데이터의 일관성**
  - ✓ 프로세스들이 공유하는 데이터를 변경하는 과정에서 데이터가 예상 밖의 상태로 바뀌는 경우가 발생할 수 있음

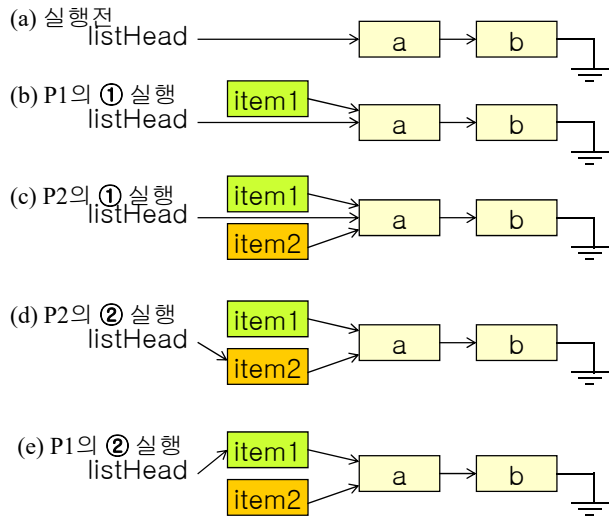
  ➔ 프로세스간의 동기화 (synchronization) 로 해결
- ✓ **공유 리스트에 항목 삽입 (프로세스 P1, P2)**

```
insertItem(item)
{
  ①   item->next = listHead;
  ②   listHead = item;
}
```

- ✓ **공유 리스트의 오류 발생 시나리오**
  - ✓ P1 이 ① 실행 → 문맥교환 → P2가 ①, ② 실행 → 문맥교환 → P1 이 ② 실행

공유 리스트의 오류 발생

```
① item->next = listHead;
② listHead = item;
```



간단한 프로그램의 데이터 공유

✓ **C 언어 표현**

```
increment()
{    counter = counter + value;
}
decrement ()
{    counter = counter – value;
}
```

✓ **어셈블리 언어 표현**

increment :

```
①  movl counter, %ecx      # ECX ← counter
②  addl value, %ecx        # ECX ← ECX + value
③  movl %ecx, counter      # counter ← ECX
```

decrement :

```
④  movl counter, %edx      # EDX ← counter
⑤  subl value, %edx        # EDX ← EDX – value
⑥  movl %edx, counter      # counter ← EDX
```

✓ **일관성 문제 발생 시나리오**

✓ P1 ① 실행 → P2 ④ ⑤ ⑥ 실행 → P1 ② ③ 실행

동기화 처리

- ✓ **프로세스간의 공유데이터 (공유자원, 공유메모리)**
  - ✓ 어느 프로세스가 공유데이터를 사용하는 부분을 실행 중이면 이 부분을 완료하기 전에는 다른 프로세스가 동일한 공유데이터를 사용하는 부분을 실행하지 못하게 처리

  ➔ 뮤텍스나 세마포 활용
- ✓ **커널 데이터의 일관성 문제**
  - ✓ 커널은 많은 데이터들을 관리함
  - ✓ 시나리오1: 커널을 실행 중에 (데이터를 변경 중에) ➔ 인터럽트 발생 ➔ 인터럽트 처리루틴에서 데이터 변경
  - ✓ 시나리오2 (*선점형 커널일때*): 커널을 실행 중에 (데이터를 변경 중에) ➔ 문맥교환에 의해 다른 프로세스로 변경 ➔ 시스템 콜 호출로 커널 부분을 실행하면서 데이터 변경
- ✓ **㈜ 레이스 컨디션 (race condition)**
  - ✓ 실행순서는 미리 예측할 수 없고, 실행 순서가 달라지면 그 결과에도 영향을 미치는 경우

크리티컬 섹션

- ✓ **크리티컬 섹션 (critical section)**
  - ✓ 프로그램에서 공유데이터를 사용하는 코드 부분들
  - ✓ 프로세스들 간에는 여러 크리티컬 섹션들이 복잡하게 얽혀 있을 수 있음
- ✓ **상호배제 (mutual exclusion)**
  - ✓ 동일한 공유데이터를 사용하는 크리티컬 섹션들 간에는 실행과정이 상호배제되어야 함
  - ✓ 크리티컬 섹션의 시작부분과 끝부분에는 허가를 획득하고 반납하는 코드 추가 (entry section 및 exit section)
- ✓ **Entry / Exit Section 의 추가**

```
increment() {
    entry section of counter;
    counter = counter + value;      // critical section of P1
    exit section of counter;
}
decrement() {
    entry section of counter;
    counter = counter – value;      // critical section of P2
    exit section of counter;
}
```

Entry/Exit 섹션의 조건

- ✓ **상호배제 (mutual exclusion)**
  - ✓ 공유데이터 S 와 관련된 크리티컬 섹션을 한 프로세스가 실행중이면 어떤 프로세스도 S와 관련된 크리티컬 섹션을 실행불가
- ✓ **진행 (progress)**
  - ✓ S와 관련된 크리티컬 섹션을 아무도 실행중이지 않으면, S와 관련된 크리티컬 섹션을 실행하고자 하는 프로세스들 중 하나는 일정한 시간 내에 실행할 수 있어야
- ✓ **제한된 대기시간 (bounded waiting)**
  - ✓ S와 관련된 크리티컬 섹션을 실행하고자 하는 프로세스는 일정한 시간 내에 자신의 크리티컬 섹션을 실행할 수 있어야
- ➔ **Entry / Exit 섹션의 구현방안**
  - ✓ 인터럽트 금지, 스핀락, 뮤텍스, 세마포 등

인터럽트 금지를 이용한 해법

- ✓ **구현방범**
  - ✓ Entry 섹션: 인터럽트 금지 명령어
  - ✓ Exit 섹션: 원상 복구 (*주의: 인터럽트 허용이 아니라 원상복구임*)
- ✓ **적용 근거**
  - ✓ 크리티컬 섹션들간의 실행 중복은 중간의 문맥교환 방지로 해결
  - ✓ 커널에서 문맥교환을 실행하는 계기는 인터럽트로부터 발생
- ✓ **사용제한**
  - ✓ 인터럽트 금지는 특권명령이므로 커널모드에서만 실행가능
  - ✓ 크리티컬 섹션에는 문맥교환을 유발하는 시스템 콜 사용 금지
  - ✓ 크리티컬 섹션 실행시간이 짧아야 ← 하드웨어 동작에 문제발생 가능성
  - ✓ 멀티프로세서/멀티코어 시스템에서는 상호배제를 보장 못함
  - ✓ 크리티컬 섹션과 관련이 없는 프로세스로도 문맥교환이 제한됨

스핀락(Spin Lock)을 이용한 해법

✓ **구현방법 (lock 변수 도입)**
- ✓ Entry 섹션: lock 변수가 0이 될 때까지 기다렸다가 lock 변수를 1로 설정
- ✓ Exit 섹션: lock 변수를 0으로 설정

Entry 섹션	Exit 섹션
`while (counterLock == 1)` `  ;` `counterLock = 1;`	`counterLock = 0;`

✓ **문제점은?**
- ✓ 단일 프로세서 시스템
- ✓ 멀티 프로세서 시스템

스핀락(Spin Lock)을 이용한 해법

✓ **Read-Modify-Write 사이클 명령어 이용**
- ✓ XCHG (exchange), SWP (swap), TAS (test and set) 등 프로세서마다 다름

	Entry 섹션	Exit 섹션
알고리즘 표현	`int flag = 1;` `while (flag != 0)` `  XCHG(counterLock, flag);`	`counterLock = 0;`
어셈블리 언어 표현	`  movl $1, %eax` `loop:` `  xchg %eax, counterLock` `  cmp $0, %eax` `  jne loop`	`movl $0, counterLock`

스핀락을 위한 함수 제공

✓ 스핀락 제공함수

Entry 섹션	Exit 섹션
spinLock(int *lock) { int flag = 1; while (flag != 0) XCHG(*lock, flag); }	**spinUnlock(int *lock)** { *lock = 0; }

✓ 사용제한

- ✓ 비지 웨이팅 (busy waiting)이므로 긴 크리티컬 섹션의 경우 시간 낭비 초래
- ✓ 단일프로세서 시스템에서는 적용 불가

➔ 멀티프로세서 시스템에서 짧고 중간에 문맥교환 없는 크리티컬 섹션에 사용

뮤텍스를 이용한 해법

✓ 뮤텍스 (mutex 또는 mutual exclusion)

- ✓ 상호배제를 위해 **프로세스가 사용**할수 있도록 운영체제에서 제공하는 기능
- ✓ 조건이 만족되지 않으면 lock을 요청한 프로세스를 waiting 상태로 전환 ← 비지 웨이팅 제거
- ✓ 뮤텍스는 lock을 한 프로세스를 소유자로 기록 ← 소유자만 unlock, 데드락 문제/우선순위 역전 문제 해결에 활용

Entry 섹션	Exit 섹션
mutexLock(int *lock) { int flag = 1; XCHG(*lock, flag); while (flag != 0) { *인터럽트 금지 또는 spinLock;* change state to *WAITING*; context switch to another *READY* process; *인터럽트 복구 또는 spinUnlock;* XCHG(*lock, flag); } }	**mutexUnlock(int *lock)** { *lock = 0; *인터럽트 금지 또는 spinLock;* wake-up all processes waiting for *lock*; *인터럽트 복구 또는 spinUnlock;* }

세마포를 이용한 해법

- ✓ **세마포 (semaphore)**
  - ✓ E. W. Dijkstra (1965)에 의해 제안
  - ✓ 프로세스가 사용할 수 있도록 운영체제에서 제공
  - ✓ 상호배제뿐만 아니라 범용으로 사용 가능
  - ✓ Wait 와 Signal 함수 사용 (P operation 과 V operation)

	Wait 동작	Signal 동작
Dijkstra의 정의	`while (S <= 0)` `  ;` `S = S - 1;`	`S = S + 1;`
비지웨이팅이 없는 구현	`semWait(int *sem)` `{` `  인터럽트 금지 또는 spinLock;` `  while (*sem <= 0) {` `  change state to WAITING;` `  context switch to another` `  READY process;` `  }` `  *sem = *sem - 1;` `  인터럽트 복구 또는 spinUnlock;` `}`	`semSignal(int *sem)` `{` `  인터럽트 금지 또는 spinLock;` `  *sem = *sem + 1;` `  wake-up all processes` `  waiting for sem;` `  인터럽트 복구 또는 spinUnlock;` `}`

Peterson의 알고리즘

- ✓ **Dekker의 알고리즘 (T. Dekker, 1965)**
  - ✓ Entry 섹션과 exit 섹션을 순수한 알고리즘만으로 구현
  - ✓ 두개의 프로세스 간의 문제
- ✓ **Peterson의 알고리즘 (G. L. Peterson, 1981)**
  - ✓ Dekker의 알고리즘을 단순하게 개선 (스핀락과 유사)
  - ✓ 프로세스 0과 1간의 처리 알고리즘
  - ✓ ㈜ 실제 운영체제 구현에서는 XCHG 등의 특수 명령어로 간편하게 해결

```
int turn, flag[2];

Lock(i)                 // entry section of process i
    j = (i+1) % 2;
    flag[i] = 1;
    turn = j;
    while (flag[j] == 1 && turn == j)
        ;

Unlock(i)               // exit section of process i
    flag[i] = 0;
```

- ✓ (주) 베이커리 알고리즘: 다수 프로세스들 간의 문제로 확장한 알고리즘

(참고) Bakery 알고리즘

- ✓ **L. Lamport (1974)**
- ✓ **기본 개념**
  - ✓ 제과점에서 손님들이 번호표를 뽑고 기다리는 것에 비유
  - ✓ 번호는 이미 대기중인 손님보다 큰 번호 부여
  - ✓ 자신보다 번호가 작은 손님이 없으면 권한확보
  - ✓ 동시에 도착한 (번호가 동일한) 경우 ID 번호가 작은 쪽 우선
- ✓ **활용 예: AJAX (Asynchronous JavaScript and XML)**

```
int choosing[n], number[n];

Lock(i):                              // entry section of process i
  choosing[i] = true;
  number[i] = max(number[0],number[1],...number[n-1]) +1 ;
  choosing[i] = false;
  for (j=0; j < n; j++) {
    while ( choosing[j] ) ;
    while ( number[j] != 0 && (number[j],j) < (number[i],i) );
  }

Unlock(i):                            // exit section of process i
    number[i] = 0;
```


크리티컬 섹션 문제 (뮤텍스 활용)

```
insertItem(item){
    mutexLock(itemLock);
    item->next = listHead;
    listHead = item;
    mutexUnlock(itemLock);
}
increment() {
    mutexLock(counterLock);
    counter = counter + value;
    mutexUnlock(counterLock);
}
decrement(){
    mutexLock(counterLock);
    counter = counter - value;
    mutexUnlock(counterLock);
}
```

유한버퍼 문제

- ✓ **유한버퍼 (bounded buffer) 문제**
  - ✓ 버퍼의 용량 제한
  - ✓ 프로듀서 프로세스는 버퍼에 여분이 있을 때 기록
  - ✓ 컨슈머 프로세스는 버퍼에 데이터가 있을 때 읽기
  - ✓ 조건이 만족되지 않으면 대기
- ✓ **세마포를 이용한 알고리즘**
  - ✓ 프로그램 9.1
  - ✓ Full 세마포: 버퍼에 데이터가 들어있음을 표시
  - ✓ Empty 세마포: 버퍼에 빈 공간이 있음을 표시
  - ✓ ㈜ 이 알고리즘에서 세마포 대신에 뮤텍스는 사용 불가, 뮤텍스를 사용하려면 알고리즘을 일부 변경해야 함

유한버퍼 문제 (프로그램 9.1)

```
struct dataItem buffer[N];          // 데이터를 저장할 순환 버퍼
int head = 0, tail = 0;             // 버퍼의 head/tail을 표시하는 인덱스
semaphore Full = 0;                 // 버퍼에 데이터가 들어 있음을 표시
semaphore Empty = N;                // 버퍼에 빈공간이 있음을 표시
void producer() {
   struct dataItem item;
   while (1) {
        프로듀서의 목적에 따라 item 생성;
        semWait(Empty);             // 순환 버퍼에 빈 공간이 생길 때까지 대기
        buffer[tail] = item;        // 순환 버퍼에 item을 복사하여 보관
        tail = tail + 1;            // 순환 버퍼의 tail 인덱스 증가
        if (tail == N)              // 버퍼의 끝이면 처음으로 변경
                tail = 0;
        semSignal(Full);            // 순환 버퍼에 데이터가 기록되었음을 알림
   }
}
void consumer() {
   struct dataItem item;
   while (1) {
        semWait(Full);              // 순환 버퍼에 데이터가 들어올 때까지 대기
        item = buffer[head];        // 순환 버퍼로부터 데이터를 item에 복사
        head = head + 1;            // 순환 버퍼의 head 인덱스 증가
        if (head == N)              // 버퍼의 끝이면 처음으로 변경
                head = 0;
        semSignal(Empty);           // 순환 버퍼에 빈 공간이 생겼음을 알림
        컨슈머의 목적에 따라 읽어낸 item 사용;
   }
}
```

Readers-Writers 문제

- ✓ **Readers-Writers 문제**
  - ✓ 읽기 작업은 여러 프로세스가 동시에 접근 가능하도록 처리
  - ← 단순히 크리티컬 섹션으로 처리하면 읽기 작업도 프로세스 하나씩만 접근가능
    - ✓ Writer: mutexLock; write; mutexUnlock;
    - ✓ Reader: mutexLock; read; mutexUnlock;
- ✓ **세마포와 뮤텍스를 활용한 해법 (프로그램 9.2)**
  - ✓ readerLock: 첫번째 reader이면 lock semData
  - ✓ readerUnlock: 마지막 reader이면, unlock semData
  - ✓ 세마포 semData를 사용한 이유: 뮤텍스는 lock owner가 unlock
- ✓ **POSIX 표준에서 정의한 관련 함수**
  - ✓ pthread_rwlock_rdlock / pthread_rwlock_wrlock
  - ✓ pthread_rwlock_unlock
  - ✓ pthread_rwlock_init / pthread_rwlock_destroy

Linux Kernel의 Readers-Writers Lock

```
#read_lock(rwlock)
   lea rwlock, %eax
   lock; decl (%eax)
   jns c
a: lock; incl (%eax)
b: cmpl $1, (%eax)
   js b
   lock; decl (%eax)
   js a
c:

#read_unlock(rwlock)
   lock; incl rwlock
```

```
#write_lock(rwlock)
   lea rwlock, %eax
   lock; subl $0x01000000 (%eax)
   jns c
a: lock; addl $0x01000000 (%eax)
b: cmpl $0x01000000, (%eax)
   jne b
   lock; subl $0x01000000 (%eax)
   jnz a
c:

#write_unlock(rwlock)
   lock; addl $0x01000000, rwlock
```

```
Unlock 상태        : 0x01000000
Write-Lock 상태    : 0x00000000
Read-Lock 1회 상태 : 0x00ffffff
Read-Lock 2회 상태 : 0x00fffffe
```

크리티컬 리전

- ✓ **프로그래밍 언어 차원의 동기화 기능**
  - ✓ 뮤텍스나 세마포를 직접 사용하는 것 보다 오류를 줄이고 표현이 체계적임
  - ✓ 크리티컬 리전 (Hoare, 1972), 모니터 (Brinch Hansen, 1973) 등
  - ✓ ➔ 컴파일러에 의해서 자동으로 뮤텍스나 세마포를 활용하여 변환
- ✓ **크리티컬 리전 (critical region 또는 conditional critical region)**
  - ✓ 사용 문법 (Pascal 문법의 확장)

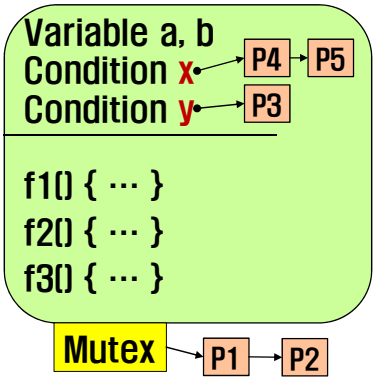
```
var v : shared T;
region v when B do S;
```

  - ✓ 공유 변수 v를 사용하는 region들 (S 부분) 간에는 상호 배제 보장
  - ✓ 조건 B가 참일 때만 S 실행, 거짓이면 참이 될 때까지 대기
- ✓ **유한 버퍼 문제에 사용한 예 (프로그램 9.3)**

모니터와 컨디션 변수

- ✓ **모니터 (monitor)**
  - ✓ 공유변수들은 모니터 속에 선언되고 모니터 속의 procedure들에서만 사용 가능
  - ✓ 모니터의 procedure들을 실행할 때 크리티컬 섹션의 조건들을 만족하도록 보장
  - ✓ Procedure를 실행중에 특정 조건이 만족될 때까지 대기할 필요가 있으면 컨디션 변수 사용
  - ➔ Java, Concurrent Pascal, Modular-2 등에 응용됨
- ✓ **컨디션 변수 (condition variable) x 에 대하여**
  - ✓ x.wait() 호출 ➔ 이후에 다른 프로세스가 x.signal() 호출시까지 무조건 대기
  - ✓ x.signal() 호출 ➔ x에 대기 중인 프로세스들 중에서 하나만 실행재개 (대기 중인 프로세스가 없으면 그냥 리턴)
  - ✓ 모니터를 사용중인 프로세스가 wait 호출로 대기 상태로 전환되면 이 모니터는 다른 프로세스가 procedure를 실행할 수 있도록 처리
  - ✓ signal을 호출한 프로세스와 이 호출에 의해 실행을 재개하는 프로세스 중 하나만 실행
- ✓ **유한 버퍼 문제에 사용한 예 (프로그램 9.4)**
  ➔ 컴파일러에 의해 변환한 예 (프로그램 9.5)

모니터의 개념적 구조



Java의 모니터와 컨디션 변수

- ✓ 단순화한 형태의 모니터 기능과 컨디션 변수 제공
- ✓ Method 단위나 method 내의 일정한 블록을 synchronized로 선언
- ✓ 동일한 객체의 synchronized 부분들 간에는 상호배제 보장
- ✓ 컨디션 변수는 클래스마다 하나씩 만 사용
  - ✓ ➔ 컨디션 변수의 이름이 없고 wait()와 notify() 사용
- ✓ 은행계좌 이체를 위한 프로그램 예

```
class Bank {
  private int[] accounts = new int[1000];

  public synchronized transfer(int from, int to,
   int amount){
    while (accounts[from] < amount)
      wait();
    accounts[from] -= amount;
    accounts[to] += amount;
    notify();
  }
}
```

Java 객체의 Synchronized 처리

✓ **Synchronized { … } 부분**

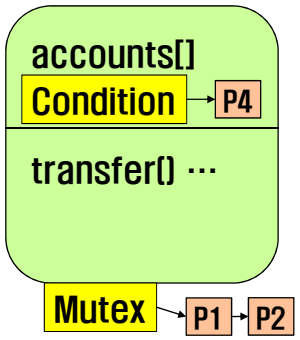
```
MutexLock(this.Mutex);
{ … };
MutexUnlock(this.Mutex);
```

✓ **Wait()**

```
CondWait(this.Cond, this.Mutex);
```

✓ **Notify()**

```
CondSignal(this.Cond);
```



Java와 Monitor의 문법 비교

Java 프로그램

```
class Bank {
  private int[] accounts = new int[1000];

  public synchronized transfer(int from,
   int to, int amount){
    while (accoints[from] < amount)
      wait();
    accounts[from] -= amount;
    accounts[to] += amount;
    notify();
  }
}
```

Brinch Hansen의 monitor를 적용한 프로그램

```
monitor Bank {
  int accounts[1000];
  condition cond;

  procedure transfer(int from, int to,
   int amount) {
    while (accounts[from] < amount)
      cond.wait();
    accounts[from] -= amount;
    accounts[to] += amount;
    cond.signal();
  }
}
```

picoKernel의 뮤텍스/세마포/컨디션변수

- ✓ 뮤텍스의 구현 (프로그램 9.7)
- ✓ 세마포의 구현 (프로그램 9.8)
- ✓ 컨디션 변수의 구현 (프로그램 9.9)

picoKernel 시스템 콜 함수

✓ **뮤텍스**

```
int mutexCreate(void);
int mutexLock(int mutex);
int mutexTryLock(int mutex);
int mutexUnlock(int mutex);
```

✓ **세마포**

```
int semCreate(int value);
int semWait(int sem);
int semSignal(int sem);
int semValue(int sem);
```

picoKernel 시스템 콜 함수

✓ **컨디션 변수**

```
int condCreate(void);
int condWait(int cond, int mutex);
int condSignal(int cond);
```


POSIX 시스템 콜 함수

✓ **뮤텍스**

```
#include <pthread.h>
int pthread_mutex_init (
  pthread_mutex_t *mutex,  // 초기화할 뮤텍스의 주소
  pthread_mutexattr_t *attr  // 뮤텍스 초기화에 필요한 값들
);
int pthread_mutex_lock (
  pthread_mutex_t *mutex   // lock을 원하는 뮤텍스의 주소
);
int pthread_mutex_unlock (
  pthread_mutex_t *mutex   // unlock을 원하는 뮤텍스의 주소
);
```

POSIX 시스템 콜 함수

✓ **컨디션 변수**

```
#include <pthread.h>
int pthread_cond_init (
  pthread_cond_t *cond,   // 초기화할 컨디션 변수의 주소
  pthread_condattr_t *attr  // 컨디션 변수 생성에 필요한 값들
);
int pthread_cond_wait (
  pthread_cond_t *cond,    // wait을 원하는 컨디션 변수의 주소
  pthread_mutex_t *mutex  // 연관된 뮤텍스의 주소
);
int pthread_cond_signal (
  pthread_cond_t *cond    // signal을 원하는 컨디션 변수의 주소
);
```


POSIX 시스템 콜 함수

✓ **세마포**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget (
  key_t key,          // 세마포 집합 고유의 key 값
  int nsems,          // 생성될 세마포의 개수
  int semflag         // 여러 가지 플래그 비트
);
int semop (
  int sem,            // 세마포 집합의 고유번호
  struct sembuf *sops,// 세마포 동작을 지정하는 목록
  unsigned nsops   // sops 배열의 항목 수
);
```

POSIX 세마포 사용

✓ **Wait(semid) 처리**

```
sembuf sop;
sop.sem_num = 0;
sop.sem_op = -1;
sop.sem_flg = 0;
semop(semid, &sop, 1);
```

✓ **Signal(semid) 처리**

```
sembuf sop;
sop.sem_num = 0;
sop.sem_op = 1;
sop.sem_flg = 0;
semop(semid, &sop, 1);
```


Java의 세마포 기능 구현

```
Class SemClass {
  private int sem;

  public synchronized MySemInit(int value)
  { sem = value; }
  public synchronized MySemWait(void)
  {
    while (sem <= 0)
      wait();
    sem = sem - 1;
  }

  public synchronized MySemSignal(void)
  {
    sem = sem + 1;
    notify();
  }
}
```

POSIX 스레드 간의 세마포 기능 구현

```
int sem;
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;

MySemInit(int value) { sem = value; }
MySemWait(void) {
   pthread_mutex_lock(&lock1);
   while (sem <= 0)
      pthread_cond_wait(&cond1, &lock1);
   sem = sem - 1;
   pthread_mutex_unlock(&lock1);
}
MySemSignal() {
   pthread_mutex_lock(&lock1);
   sem = sem + 1;
   pthread_cond_signal(&cond1);
   pthread_mutex_unlock(&lock1);
}
```


과제 #2: 스레드 동기화 문제

- ✓ 동기화가 필요한 문제 프로그램
  - ✓ 동기화가 되지 않으면 실행 결과에 오류가 발생하는 문제를 정의
  - ✓ 가급적 과제 #1에서 자신이 작성한 프로그램을 보완하는 방향으로
  - ✓ threadYield를 적절히 넣어서 문제발생 유도
- ✓ 동기화 적용한 프로그램
  - ✓ Mutex나 Semaphore를 활용하여 문제 해결
- ✓ 동기화 처리전과 후의 비교를 위해
  - ✓ 실행1: 동기화를 적용하지 않은 상태의 실행 결과 검토
  - ✓ 실행2: 동기화를 적용한 후의 정상적인 실행 결과 검토
- ✓ 적용 운영체제
  - ✓ 가급적 picoKernel 의 mutex, semaphore 사용
  - ✓ 또는 일반 운영체제의 POSIX 함수 사용