**운영체제 과제 7(5-7장)**

2015253039 권진우

5장

**1.** (1) Spin lock이란? 어떤 문제? 해결법?

- Spin lock이란 Lock변수 하나를 갖고 여러 프로세서들이 Lock을 점유했을 시에만 CPU를 점유하고 실행되고 Lock을 얻지 못한 프로세스는 Busy-waiting을 하며 기다리는 방식이다. 따라서 Lock은 1개이므로 동시에 여러 프로세스가 사용할 수 없으며, 싱글 프로세서 시스템에서는 Spin Lock 방식이 오히려 오버헤드를 일으킬 수 있다. 싱글(단일) 프로세서는 어차피 1개의 CPU를 사용해서 Context Switching을 통해 작업을 수행하기 때문에 Spin Lock을 검사하는 루틴 자체가 사용자에게 있어서 오버헤드이기 때문에(cpu cycle낭비) 필요치 않고 다중 프로세서 시스템에서는 짧은 시간내에 Lock획득이 예상된다면 Spinlock이 유용하다.(Context Switching이 필요없음) 하지만 길어지면 프로세스가 노는 시간이 길어지기 때문에 효율이 떨어진다. Block과 Wake-up을 이용한 No busy waiting을 하는 스핀락으로 사이클 낭비를 줄일 수 있으며 Lock은 1개를 여러 프로세스를 돌며 사용하기 때문에 Mutex Lock보다 더욱 정교한 동기화 도구인 세마포를 사용한다.

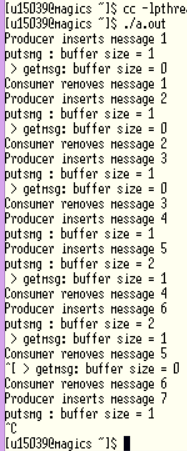
(2) 세마포란 무엇이고 세마포의 두가지 기본연산?

세마포란 뮤텍스락의 개념을 포함한 특수한 자료구조이다. 뮤텍스락보다 정교하고 강력한 프로세스 동기화 도구이다. 세마포의 두가지 기본연산은 P(test)와 V(increment)가 있다. 먼저 최대 가용자원의 개수를 S변수의 초기값으로 설정하고 프로세스가 자원을 사용시마다 하나씩 감소시킨다. P(test)연산은 대기, 진입 2가지 역할을 수행하고 세마포 자원사용 요청을 한다. 자원이 남아있다면 세마포 값 S를 하나 감소시키고 Lcok을 획득한다. V(increment)연산은 반납연산으로 자원 사용 후 이 연산을 통해 자원을 반납시킨다. 이 두 연산은 세마포 값(공유하는 변수)을 변동시키므로 원자적으로 수행되어야 한다.

(3) 세마포의 전형적인 사용법

세마포 값을 초기화하여 유한자원으로 제한시키고 P(test)연산 시 세마포 값 S = S-1을 을 통해 자원의 갯수를 1개 감소시키고 자원을 사용한다. 자원을 사용하고 난 후 V(increment)연산을 통해 S = S+1을 수행하여 자원의 갯수를 1개 증가시킴으로써 자원을 반납한다.

**2.** . (1) bounded buffer를 사용하여 데이터를 주고 받는 producer와 consumer 문제에 대해서 producer와 consumer의 동작을 세마포를 사용하여 구현하시오.



(2) 공유 데이터를 읽기만 하는 프로세스들과 쓰기도 수행하는 프로세스들로 구성되는 reader-writer 문제에 대해서 reader와 writer의 동작을 세마포를 사용하여 구현하시오.

Semaphore mutex = 1

Semaphore rw\_mutex=1;

int read\_count = 0

// mutex : for updating read\_count

// rw\_mutex : for updating shared

database (common to reader/writer)

Reader

// acquireReadLock

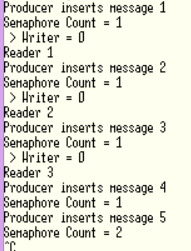
P(mutex);

read\_count++;

if (read\_count==1) // first reader

P(rw\_mutex);

V(mutex);



reading is performed

...

P(mutex);

read\_count --;

if (read\_count==0)

v(rw\_mutex);

V(mutex);

Writer

P(rw\_mutex); // acquireWriteLock

...

writing is performed

...

V(rw\_mutex); // releaseWriteLock

(3) (2)번의 방법은 어떤 문제가 발생할 수 있는가?

- 공유 데이터를 읽기만 하는 프로세스들은 동시에 공유데이터에 접근할 수 있고 여러 프로세스들이 동시에 접근하여 사용한다 하더라도 공유 데이터 내용을 수정하거나 건드리지 않기 때문에 문제가 되지 않지만 Read-Write동작을 모두 하는 프로세스는 공유데이터를 사용할 때 공유데이터를 변경 시킬 수 있기 때문에 여러 프로세스가 동시에 접근하여 사용하게 된다면 한 프로세스가 사용하는 데이터가 다른 프로세스에 의해서 변경되는 데이터 일관성 문제가 발생할 수 있다.

또한 Writer는 Lock을 가지고 Lock을 걸고 Write를 수행하고 Write를 수행하는 중에는 Read가 일어나지 못하도록 막는다. Write가 끝나면 Lock을 반환하고 Reader가 권한을 얻어서 Read를 수행할 수 있게 해준다. 첫 Reader는 공유 자원 사용 권한을 얻고 마지막 Reader는 자원 사용 권하늘 반납한다.

**3.** monitor란 무엇이며 monitor에서 프로세스 간의 동기화는 어떻게 구현되는가?

- Monitor란 프로그래머가 세마포를 사용할 때 P와 V의 연산 순서를 바꾸거나(상호배제위반), V대신 P를 사용하거나(교착상태발생), P또는 V를 누락(CS진입X, 상호배제 위반) 등의 실수를 하게되면 교착상태나 기아 등과 같은 상황이 발생 할 수 있기 때문에 모니터는 병행 프로그래밍에 있어서 프로그래머의 실수를 예방하기 위해 병행 프로그래밍에서 고수준의 상호배제 지원을 통해서 (언어차원에서 지원) 프로그램이 세마포 기능을 대신 지원하는 방식으로 상호배제를 보장하여 준다.

모니터는 조건변수를 사용하는데 세마포와는 다르게 코드에서 Wait를 만나면 무조건 Signal이 호출할 때까지 기다림

**4.** pthread 라이브러리의 mutex lock, semaphore에 대한 기본 연산을 수행하는 함수 이름과 기능 은 무엇인가?

- Pthread 라이브러리

Pthread \_cond\_t\_cond : pthread initializer

Init pthread\_cond\_init() : pthread 생성 및 실행

Init pthread\_cond\_signal() : 조건변수 cond에 신호를 보낸다 Wait쓰레드가 있다면 깨우게 된다.

Init pthread\_cond\_broadcast() : 조건변수 **cond**에서 기다리는(wait) 모든 쓰레드에게 신호를 보내서, 깨운다

Init pthread\_cond\_wait() : Signal신호가 올 때 까지 기다리는 Wait

Init pthread\_cond\_destroy() : 조건변수 자체를 없앨 때 사용

sem\_init() = 세마포 초기 값 지정

sem\_wait() = 세마포를 얻기까지 기다림

sem\_post() = 세마포 반납

pthread\_mutex\_init() = 뮤택스락 초기화

6장

**1.** deadlock이란 무엇이며, 어떤 조건에서 발생하는가?

- Deadlock=교착상태를 뜻하며, 교착상태란 각 프로세스들이 자원을 1개씩 점유하면서 서로 상대방의 자원을 필요로 할 때 무한히 기다리는 상황이 발생하는 무한대기 상태를 Deadlock이라고 한다.

- 교착상태를 일으키는 4가지 필요조건

1. **상호배제** : 한번에 한 프로세스만 사용할 수 있는 자원(*CS*)이 존재

2. **점유하며 대기** : 적어도 1개의 자원을 점유하고 타 프로세스의 점유된 자원 대기

3. **비선점** : 자원을 강제로 뺏을 수 없음, 자발적으로 방출된 자원만 사용 = 대기

\*선점 : 상대방의 자원을 강제로 빼앗아 사용

4. **순환대기** : 순환적으로 대기 P0->P1->P2->P3->P0...(**식사하는 철학자**)

**2.** deadlock 예방(prevention)과 회피(avoidance) 방법에 대해서 비교 설명해보시오.

- 교착상태 예방 방법은 위 4가지 원인 중 1가지만 제거하여도 가능하고 교착상태 회피 방법은 프로세스가 실행 전에 해당 자원들을 모두 사용 가능한지 추가정보를 필요로 하게 됩니다. 따라서 자원이 사용 가능하지 않을 시 프로세스를 대기시키며 자원이 모두 반납되었을 때 실행합니다. 하지만 회피 방식을 사용하면 사용 가능한 자원이 있는대도 불구하고 사용되지 않는 장치 이용률 저하와 시스템 처리율이 감소하는 비효율적인 상황이 발생 할 수 있습니다.

**3.** deadlock 예방 방법에 대해서 설명하시오.

1. **상호배제 조건 거부** : 자원을 동시에 같이 쓰게 함[EX. 읽기 전용 파일은 동시에 읽어 도 문제 발생X ], **(*단점* : 동시 공유가 불가한 자원에 대해서는 불가능)**

2. **점유 대기 조건 거부** : 실행 전 모든 자원 할당

- **실행 전** 필요한 **모든 자원 요청** 및 할당

- 점유 자원을 반납 한 후에 반납한 자원과 추가 자원 요청

- *단점* : 자원 이용률이 낮아진다.(비효율) + 기아상태 가능성

3. **비선점 조건 거부** : 선점 허용

- 현재 점유한 모든 자원을 반납하고 대기 => 프로세스는 반납한 자원과 추가요청 자원 의 대기리스트에 추가 => **모든 자원을 할당 받을 수 있을 때 재개**

- **요청한 자원을 이용할 수 없을 때 대기상태가** 됨 => 자원이 타 프로세스에게 선점됨 => **선점 된 자원과 요청한 자원을 할당 받을 수 있을 때 재개**

- *단점* : 자원 저장/복원이 용이한 경우만 적용가능(CPU레지스터, 메모리)

4. **순환 대기 조건 거부** : **여러 자원이 필요한 경우 오름차순으로 자원을 요청** => 자원 을 얻는 순서가 일정하게 적용 됨 => **0,1,2,3,4...(자원) 순으로 자원을 할당 요청(자원 3 을 할당 받은 상태에서 추가요청은 4,5,6,7... 순으로 자원 요청, 0,1,2 자원 요청하려면 반납후 대기)**

7장

**1.** (1) linker와 loader의 역할은 무엇인가?

- Linker : 여러 모듈(obj파일, 라이브러리 등)을 함께 묶어서 실행파일을 생성하는 역할

- Loader : 실행을 위해 프로그램의 일부/전부를 메모리에 적재하는 역할

(2) 주소 바인딩이란 무엇인가? 주소 바인딩 시점에 대해서도 논의해보시오.

- 주소 바인딩이란 물리적 메모리에 데이터나 코드와 같은 파일이 적재되어 결정되는 시점을 말합니다.

주소 바인딩 시점으로는 3가지가 있는데 컴파일시간, 적재시간, 실행시간 바인딩이 있습니다.

- 컴파일 시간 - 프로그램이 적재될 메모리 위치를 미리 알 때, 컴파일러가 절대주소 생성

적재 시작 위치가 변경된다면, 코드를 다시 컴파일 해야 함

- 적재 시간 - 컴파일러가 relocatable code(맵핑{사상}된 주소) 생성 🡪 Load time에 주소위치 결정

- 실행 시간 – 프로세스가 실행 도중 메모리 주소위치 이동 가능 <- 바인딩이 실행 중 일어남

- 주소 맵핑을 위한 하드웨어 지원 필요 -> MMU(메모리 관리 장치)

정적 바인딩 : 컴파일 시간, 적재 시간 바인딩

동적 바인딩 : 실행시간 바인딩 = 바인딩이 실행시간에 동적으로 일어남

**2.** 공유 라이브러리는 언제 링크가 이루어지며 어떻게 호출되어 실행되는가?

- 공유 라이브러리는 소스 프로그램(코드)에서 컴파일러를 통해 주소위치가 재지정되고 링커와 로더를 통해 라이브러리가 적재된다. 재지정코드와 라이브러리코드가 링커에 의해서 묶어져서 로더에 의해 메모리에 적재된다. 라이브러리 중 동적 라이브러리는 프로세스가 Library 함수를 호출 할 때 Stub코드를 통해서 라이브러리가 메모리상에 위치하여 있는지 검사하고 없다면 디스크에서 공유 라이브러리를 가져와서 적재 및 프로세스들이 공유, Stub코드는 library 함수 주소를 반환값으로 받아서 대체되어 다음 호출부터는 바로 메모리에서 참조해서 실행

**3.** (1) 메모리의 외부 단편화와 내부 단편화란 무엇이며 언제 발생하는가?

- 외부 단편화 : 실행하려고 요청한 프로세스의 크기보다 작은 메모리 블록만 존재하는 것

- 내부 단편화 : 요청 받은 프로세스를 적재하는데 더 큰 메모리 Part를 할당하는 것

(2) 외부 단편화 문제에 대한 해결 방안은 어떠한 것들이 있는가?

- 해결방안으로는 압축과 비연속 물리 메모리 공간을 허용하는 것이 있는데 압축은 모든 Free 메모리 블록 공간을 하나의 큰 블록으로 재배치 하는 것이다. 하지만 이는 시간이 매우 많이 걸리고 모든 적재되어있는 프로세스들이 동적이어야 가능하기 때문에 사용이 어렵다. 비연속 물리 메모리 공간을 허용 하는것은 페이지방식과 세그먼트방식이 있는데 페이징은 할당할 1개의 프로세스를 연속된 메모리 공간이 없을 때 메모리에 분할되어있는 Free partition(hole)들의 크기로 프로세스를 분할하여 맵핑하는 방법이다. 메모리는 Page frame단위로 분할하고 page단위로 분할된 프로세스가 1:1 적재

**4.** (1) 메모리 성능을 고려한 페이징을 지원하는 하드웨어 구조와 동작에 대해서 설명하시오.

- 메모리와 CPU 사이간에는 속도차이가 발생하고 주 메모리공간에 Page table을 만들어서 Page table을 액세스한 뒤 해당 Page frame을 찾아가서 다시 액세스하기 위하여 2번의 액세스가 필요한데 이것은 불가능하므로 완전연관사상 캐시인 TLB(Translation Look-aside Buffer)캐시를 사용하여 Page table내용을 캐시에 저장하고 캐시를 액세스 후 주 메모리의 해당 Page frame을 찾아 액세스하는 방법이다.

(2) 페이징 방법은 어떻게 메모리를 보호하는가?

- 페이징이 메모리를 보호하는 방법은 각 프로세스는 독립적으로 프로세스마다 1개씩의 Page table을 가지기 때문에 각 Page table마다 지정하는 메모리의 영역이 다르기 때문에 한 프로세스가 다른 프로세스나 OS에 대해 접근하거나 수정을 방지할 수 있게 메모리를 보호할 수 있다.

또한 Protection bit(RW, RO, EO)와 Valid bit를 통해서 페이지 단위에서 보호가 가능하다.

(3) 계층적 페이징 방법에 대해서 설명하시오. 이 방법이 필요한 이유와 단점은 무엇인가?

- Hierarchical paging 방법은 32비트 주소공간에서는 계층적 페이징이 필요치 않지만 그보다 더 큰 주소 공간을 사용하게 되면 Page table의 크기가 Page frame 갯수의 증가로 인해서 한 Page 사이즈보다 더 커지게 되므로 Page table을 하나의 연속적 메모리 공간이 아닌 1page단위 크기로 여러개의 조각들로 나눈 계층적 페이징을 통해서 더 많은 주소공간 사용 가능

상위비트(page table)와 하위비트(page frame에서의 Offset)로 나뉘어있는 논리 주소를 상위비트를 더 많은 단계로 나누어서 페이지 테이블을 메모리의 여러 주소공간에 저장하는 방법이다.

이러한 방법으로 더 많은 주소공간을 페이징 단위로 나누어 사용할 수 있지만 계층이 2개가 넘는(3번 이상) 구조에서는 메모리 액세스가 3단계 계층 Paging일 경우 4번의 액세스가 필요하므로 너무 큰 오버헤드가 발생하게 됩니다.

(4) 논리주소의 비트수가 큰 경우에 계층적 페이징 방법은 어떤 문제가 있으며 이에 대한 해결 방안은 무엇인가?

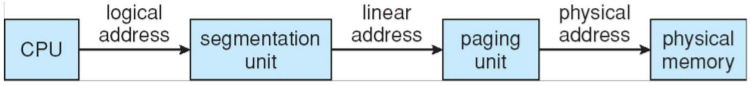
- 논리 주소의 비트수가 큰 경우 계층적 페이징을 이용하게 되는데 이렇게 되면 단계가 늘어날수록 메모리를 액세스하는 횟수가 늘어나는 것이기 때문에 최근 64bit 시스템에서는 해시함수를 사용하여 Hash page table을 구현하여 사용합니다. Hash page table은 해시 함수를 통해 나온 값으로 hash table의 검색대상 범위를 줄일 수 있고 Linked list를 사용하여 검색대상이 된 여러 범위를 List 순차적으로 탐색하여 검색된 Page frame으로 액세스한다.

**5**. (1) base와 limit 레지스터를 사용한 메모리 보호 방식을 설명하시오.

Base와 limit레지스터는 각각 Base는 프로세스의 시작주소, Limit는 프로세스의 길이 정보를 저장하는 레지스터이며 CPU가 발생하여 내보낸 주소로 메모리를 액세스하기 전 Base<=발생주소<Base+Limit 인지 검사를 통해 합법적 주소 범위 내에서만 사용할 수 있도록하여 타 프로세스나 OS를 건드리지 못하게 메모리 보호를 할 수 있게 됩니다. Base와 Limit 레지스터값은 OS만 접근 가능한 특권 명령어 사용

(2) segmentation을 지원하는 하드웨어 구조와 주소 변환 과정을 설명하시오.

-Segmentation은 프로그램을 논리적 단위로 분할 한 뒤 프로그램이 분할된 크기에 따라 메모리에 적재되는 방식이다. CPU에서 발생한 프로세스의 2차원 주소(seg num, 해당 segment offset)인 논리 주소가 Segmentation unit을 통해서 선형주소로 바뀌고 Paging unit을 통해서 물리적 주소인 절대주소로 바뀌어 해당 메모리 주소에 액세스한다.



**6.** Intel Pentium 프로세서(IA-32)의 주소 변환 기법을 설명하시오.

- CPU에서 발생한 2차원 논리 주소(Segment 번호, 해당 Segment Offset)를 Segmentation을 이용하여 Segment table을 통해 주소를 발생시키고 기존 하위비트와 합하여서 선형주소로 내보낸다. 32비트 선형 주소는 상위 20비트(10+10비트)와 하위 12비트로 나뉘는데 하위 12비트는 2^12(4KB)인 1페이지 단위를 나타낸다. 최상위 10비트를 통해서 Page directory에서 Page table을 탐색하고 탐색된 Page table로 가서 상위 10비트를 통해 물리적 메모리 상의 어느 Page frame으로 접근할지 탐색한다. 해당 Page frame으로 가서 하위12비트 Offset 주소를 통해 절대주소에 접근한다.

또한 복수 페이징크기를 지원하는 경우 Page Directory에서 상위 10비트를 이용해서 Page table을 찾아가서 하위 12비트(4KB)의 Page frame를 찾아갈 수도 있고 Page Directory에서 바로 하위 22비트를 Offset으로 사용하여 22비트(4MB)의 Page frame을 찾아갈 수도 있게 4KB, 4MB 두가지 크기의 페이지를 지원할 수도 있다.

