**6강. 상호작용 프로세스와 동기화**

프로그램 내의 n개의 프로세스들이 상호작용 하면서 발생할 수 있는 문제들

1. **결정성** : 프로세스 각각의 실행 속도에 관계 없이 항상 주어진 초기값에 대하여 같은 결과를 낼 수 있는가?
2. **상호배제와 동기화** : 어떤 자원에 대하여 한번에 한 프로세스만이 그것을 접근할 수 있어야함
3. **교착 상태** : 프로세스들이 서로 상대방이 가지고 내놓지 않는 자원을 요구하여 아무도 진행되지 못하게 되는 상태
4. **기아** : 프로세스가 어떤 보장 없이 막연히 자원을 기다리는 것

프로세스 시스템은 **프로세스의 집합**과 이들의 **선행 제약**으로 정의되고, 프로세스 시스템 내의 프로세스들은 **서로 독립적이거나 선행제약 관계가 있다면 비순환적이여야** 한다.

1. 선행 제약 관계의 두 프로세스
   1. **부분 순서** 성질으로 **이행성**을 갖는다. ( aRb이고 bRc이면 aRc이다. )
   2. 정상적인 프로세스 시스템의 선행 그래프(Directed 그래프)는 **비순환적**이다. ( 순환이 내포된 선행 그래프는 선행 제약 관계의 판단이 불가능 )
2. 선행 제약 관계가 없는 두 프로세스 : **독립적**이다.
   1. 두 프로세스가 독립적인 관계일지라도 **공유 변수로 인한 간섭**이 발생할 수 있다.

**결정성** : 프로세스 시스템 내의 프로세스들 간의 코드 실행 순열이 매번 다를 수 있지만 **같은 조건과 입력이 주어진다면, 항상 같은 결과를 산출해야 한다**는 성질

프로세스 시스템에서 두 개의 프로세스가 있을 때,

한 프로세스가 **다른 프로세스를 선행**하거나, ( 두 프로세스가 **선행 제약 관계**거나 )

또는 두 프로세스가 **서로 독립적**일 경우

**한 프로세스의 출력 장소가 다른 프로세스의 입력 장소나 출력 장소가 아니면**,

두 프로세스는 **비간섭 관계**에 있다고 정의한다.

한 프로세스 시스템에서 모든 프로세스 쌍이 비간섭 관계가 성립하면,

그 시스템은 **비간섭 관계**를 만족한다고 한다.

**선행 제약, 독립적, 간섭, 결정성 관계 판단 방법**

|  |
| --- |
| Q1. 선행 제약이 있는가?  A1-1. 있다. (선행 제약 관계)  -> 결정적이다.  A1-2. 없다. (독립적 관계)  -> Q2. 공유변수간 간섭이 있는가?  A2-1. 있다. (간섭) -> 결정적이지 않다.  A2-2. 없다. (비간섭) -> 결정적이다. |

**결정성 지원 방향**

|  |
| --- |
| 1. **선행제약 관계를 명시**하도록 지원    1. **Parbegin, Parend** : 두 사이의 각 문장은 독립적 프로세스로 실행됨을 명시함    2. **Fork/Join** : 프로세스 분기와 합류를 명시함 ( 분기와 동시에 선행관계 생성 ) 2. **공유변수 간섭 문제 해결방법**을 지원    1. 공유 변수에 대한 **상호 배제 및 동기화 프로그래밍 방법**을 제공한다.   Ex ) 임계구역, 상호배제, 세마포어 등 |

ISSUE : **Race Condition**

* 생산자/소비자 스레드의 counter **공유 변수 동시 접근시** 발생하는 문제

( cf. 생산자/소비자 스레드 구현시 in, out 변수 2개로 코딩할 경우 공유 변수이나 한 쪽에서만 값을 변경하므로 비간섭 관계! But, counter 공유 변수 1개로 버퍼 내의 데이터 개수를 셀 경우 양 쪽에서 값을 변경해 간섭 문제가 발생)

* 고급어 -> 기계어로 변환되고, 생산자/소비자 스레드 각각 Time Slice 소진으로 코드가 번갈아 실행될 경우 공유 변수의 결과 값이 엉뚱하게 나올 수 있다.
* SOLUTION : **원소적(Atomic) 실행** -공유변수를 통해 상호작용하는 프로세스 혹은 쓰레드 간에 **문맥교환이 언제 일어나도 간섭이 없는 실행이 보장**되는 것

**원소적 실행 ( Atomic 실행 )**

|  |
| --- |
| 1. 공유변수를 사용하는 코드 영역에 **임계구역을 설정**    1. 각 프로세스 혹은 쓰레드가 공유변수, 자료구조, 파일 등을 배타적으로 읽고 쓸 수 있도록 설정한 코드 세그먼트    2. **동기화 (Synchronization)** : 코드 상에 임계 구역을 설정하여 **진입/진출을 순서화 (Serialize)** 시키는 것       1. 코드 상으로는 **독립적**이지만 **실행 상황에서 선행제약**을 만듦 2. 한 쓰레드가 먼저 임계구역 내에 진입하여 실행 중 문맥 교환이 발생하여 상대 쓰레드에 선점되더라도 그 쓰레드가 임계 구역에 진입하는 것을 불허하고 대기하도록 함 (동기화) |

**임계구역 Atomic 실행의 예제 2가지**

생산자/소비자 문제에서의 Race Condition을 해결하려면 :

1. 생산자/소비자 코드 각각에서 counter(공유 변수)를 접근하는 코드 1줄을

Enter\_Mutex(lock) 과 Exit\_Mutex(lock)으로 둘러싸 “임계구역” 임을 표시한다.

1. Lock을 mutex 변수로 선언했으므로, Enter\_Mutex(lock) 이라는 코드를 실행시키기 위해서 (임계구역에 진입하기 위해서) 생산자/소비자 스레드가 mutex 변수인 lock을 얻기 위해 경쟁한다.
2. 먼저 mutex 변수를 얻은 스레드가 먼저 실행되고, 프로세스의 실행 과정에서 생산자-소비자 스레드 간에 선행제약이 생긴다.

우선순위 역전 문제 : ( 우선 순위 T1 > T2 > T3 )

1. 우선 순위가 제일 낮은 프로세스 T3에서 자원(R)을 lock해준다.

= Enter\_Mutex(lock)으로 특정 영역을 임계구역으로 설정해준다. 공유 변수 R은 이제 T3 외에서는 변경될 수 없다.

1. T1이 임계구역에 진입을 시도하지만, mutex변수 lock을 획득하는 것에 실패, 임계구역에 들어가지 못한다. 결국, T3에서 exit\_mutex(lock)되어 임계구역을 벗어나기 전까지 T1은 실행될 수 없다. ( 우선순위 역전 )

**임계구역에 대한 요구사항 3가지**

1. **상호 배제 ( mutual exclusion )**

: 한 프로세스가 임계 구역을 실행 중일 때, **다른 어떤 프로세스도 임계 구역을 실행할 수 없다.**

1. **진행 ( progress )**

: 임계 구역을 실행하는 프로세스가 없고 여러 개의 프로세스들이 임계 구역에 들어오고자 하는 상황에서는 **반드시 하나의 프로세스를 선택하여 진입시키는 올바른 결정 기법**이 있어야 하고, 이러한 **결정은 무한정 미루어 져서는 안 된다**.

1. **제한된 대기 ( bounded waiting )**

: 한 프로세스가 임계 구역에 대한 진입 요청 후부터 요청의 수락까지의 기간 내에, **다른 프로세스가 임계 구역을 실행할 수 있는 횟수에는 제한이 있어야** 한다. ( 이 요구사항이 없다면 **기아현상이 발생**한다. )

**Mutex의 구현**

|  |  |
| --- | --- |
| enter\_mutex(lock){ //entry section  while (lock == true) ;  lock = true; } | exit\_mutex(lock) { // exit section  lock = false;  } |

ISSUE : enter\_mutex 내에서 while (lock == true) ; 까지 실행되고 문맥교환이 발생할 수 있음

* Lock=true;로 설정되기 전에 문맥 교환이 일어나버려 두 프로세스가 동시에 lock을 테스트하는 경우 모두 false로 인식 > 경쟁 관계의 다른 프로세스의 while문이 수행되는 경우가 발생 : **상호 배제가 성립되지 않음**
* 공유변수 때문에 발생한 임계 구역 문제를 또다시 공유변수로 해결하려 하기 때문
* SOLUTION에는 S/W SOLUTION 2가지 : **Peterson’s Solution , Bakery 알고리즘**

H/W SOLUTION 3가지 : **Test-And-Set, Swap, 인터럽트 통제**

* S/W SOLUTION 1 : **Peterson’s Solution**

|  |  |
| --- | --- |
| int flag = 0,0; // 공유 변수  int turn = 0; // 공유 변수  enter\_mutex {  flag[i] = 1; // i는 나, j는 너의 프로세스 id  turn = j;  while (flag[j] == 1 && turn == j) ; } | exit\_mutex(lock) { // exit section  lock = false;  } |

* 경쟁 관계의 두 프로세스가 **반드시 한번씩** 서로에게 양보를 함 (turn = j)
* 서로에게 한번씩 양보했기 때문에 최종적으로는 먼저 양보를 한 프로세스( “turn = j”를 먼저 수행한 프로세스 )가 먼저 실행되게 된다.
* 변수 turn은 동시에 임계 구역 진입을 원하는 경우 tie-breaking 역할을 한다.

|  |  |
| --- | --- |
| Process 1 | Process 2 |
| enter\_mutex {  flag[i] = 1;  **(1)** turn = j;  **(3)** while (flag[j] == 1 && turn == j) ;  } | enter\_mutex {  flag[j] = 1;  **(2)** turn = i;  **(4)** while (flag[i] == 1 && turn == i) ;  } |

* 두 프로세스 모두 flag[i]까지 수행했다면, 실행되는 순서는 **(1)-(2)-(3)-(4)** 이다.
* S/W SOLUTION 2 : **Bakery 알고리즘 (Lamport)**

|  |  |
| --- | --- |
| Entry | 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) ); }** |
| Exit | number[i] = 0; |

* 대기하는 각 프로세스는 우선 번호표를 하나씩 뽑는다.
  + 각 프로세스는 **자신의 번호가 가장 작은 번호가 될 때까지 기다린다**.
  + 만약 발급 받은 숫자가 같을 경우 **PID가 작은 것을 먼저 처리**

**: (while ( number[j] != 0 && (number[j],j) ＜ (number[i],i) );))**

* H/W SOLUTION 1 **: Test-And-Set**

|  |  |
| --- | --- |
| int Test-and-Set (int \*target) // 또는 {  int temp;  temp = \*target ;  \*target = 1 ; //true  return temp;  } | … /\* before critical section \*/  while ( Test-and-Set ( &lock ) ) ;  <<<< critical section; >>>>  lock = 0;  … /\* after critical section \*/ |

* 전역변수 lock을 0으로 initialize, 프로세스A가 처음으로 Test-and-Set(&lock)을 실행시켰다면 Test-And-Set에서 lock은 1로 초기화되고 리턴은 lock이 0이였던 한 번을 제외하고는 계속 1이 된다. 고로, 문맥교환이 일어나 다른 프로세스에서 while문을 실행시켰다고 하더라도 그것은 리턴 값이 1이 되기 때문에 임계 구역에 진입하지 못하고 계속 while문을 도는 대기 상태가 된다. 현재로서는 **Return 0을 할 수 있는 경우는 오로지 프로세스A**밖에 없으므로 임계구역이 지켜질 수 있는 것!
* Lock에 대한 test 혹은 copy를 CPU의 명령어로 제공 ( CPU 명령어 하나는 **문맥교환 없이 원소적으로 수행됨** )
* 프로세스P가 임계구역을 빠져나가면서 lock을 0으로 바꾸면 대기하고 있던 프로세스 중 가장 먼저 test\_and\_set을 수행시킨 **딱 하나의 프로세스가 딱 한번만 0의 값을 리턴받게 됨**
* H/W SOLUTION 2 : **Swap**

|  |  |
| --- | --- |
| void Swap (int \*a, int \*b) {  int temp;  temp = \*a ;  \*a = \*b ;  \*b = temp ;  } | key = 1;  do {  Swap( &lock, &key) ; }  while (key == 1);  <<<< critical section; >>>>  lock = 0 ; // false  // … remainder section; |

* 프로세스A가 처음으로 Swap을 실행시켰다고 했을 때, **lock = 1 & key = 0 이 되어 임계구역으로 진입한다**. 이 때 다른 프로세스가 Swap을 실행시킬 경우 **lock = 1 & key = 1이 되어 while문을 빠져나갈 수 없고, 대기 상태에 머무르게 된다.**
* CPU의 명령어로 두 Boolean 변수 값을 문맥교환 없이 교체
* 프로세스A가 임계구역을 빠져나가며 lock을 0으로 리셋하면, 대기하고 있던 프로세스 중 가장 먼저 swap 명령어를 수행시킨 **딱 하나의 프로세스만 딱 한번만 lock =1 & key = 0이 될 수 있어 while문을 빠져나가 임계구역으로 진입할 수 있다.**
* H/W SOLUTION 3 : **인터럽트 통제**

|  |
| --- |
| interrupt-disable;  <<<< critical section; >>>>  interrupt-enable;  … remainder section; |

* Interrupt-disable 과 interrupt-enable
  + **임계 구역 내의 문맥 교환 자체를 방지**
  + **특권 명령어**이기 때문에 **사용자 모드에서는 사용 불가능**
  + 커널 내의 시스템 호출 부분과 인터럽트 처리기의 임계 구역 보호를 위해 사용

Mutex의 SW적 구현들과 Test-and-Set, Swap 등은 모두 **“busy waiting”** 알고리즘

임계구역 진입 시 **이미 다른 프로세스가 있으면** busy-waiting loop 실행 -> **타임 슬라이스 낭비**

1. 프로세스A가 스케쥴링에서 선택받음 : **준비->실행(진입 대기) 상태**
2. 공유변수 lock을 얻을 때까지 Busy-waiting loop를 돌다가 못 얻고 타임 슬라이스 전부 소진 : **실행->준비 상태**
3. Lock을 얻었을 경우 Busy-waiting loop를 끝내고 임계구역 진입 : **실행 상태**

Cf. **Spinlock** : 상호 배제 관계에 있는 임계구역들의 계산량이 현저히 적다는 것을 알고 있는 경우 **문맥 교환의 오버헤드를 줄일 수 있게 busy waiting 방식의 lock을 사용**하는게 효과적 ( OS 커널에서 주로 활용 )

**세마포어 : block/wakeup 알고리즘**

* 임계 구역으로 진입이 불가능할 때 **대기상태로 전환**
* 임계 구역을 떠나는 프로세스가 **대기 프로세스를 준비 상태로 깨워줌**
* 구성 : **정수 변수 & 프로세스 대기 큐**
* (Semaphore) S.value : 자원 활용 현황
  + **양수 : 남아있는 자원 수 / 음수 : 부족하여 대기하고 있는 대기 프로세스 수**
  + S.value의 초기값 n : **자원의 개수**
* 정수에 대한 3가지 연산 : **init, wait (P연산), signal (V연산)**

|  |  |  |
| --- | --- | --- |
| INIT | WAIT | SIGNAL |
| init (S, n) {  S.value = n;  } | wait(S){ // P 연산  S.value = S.value - 1 ;  if ( S.value < 0 ) {  block this process;  and add it to wait queue;  }  } | signal(S) { // V 연산  S.value = S.value + 1;  if ( S.value <= 0 ) {  remove a process from wait queue;  add it to ready queue;  } } |

* **Init** : 세마포어의 **자원의 개수**를 initialize해주는 연산
* **Wait : 자원 요청 연산**
  + - * 남아있는 자원의 개수가 양수일 경우 해당 프로세스에게 자원을 할당 ( -1 )
      * **남아있는 자원의 개수가 음수일 경우 해당 프로세스를 대기큐에 추가**
* **Signal : 자원 반납 연산**
* 남아있는 자원의 개수가 양수일 경우 그냥 자원 반납 ( +1 )
* **남아있는 자원의 개수가 음수 또는 0일 경우 ( 자원을 기다리고 있던 프로세스가 존재하는 경우 ) 해당 프로세스를 대기큐에서 빼고 준비큐로 삽입**

세마포어와 Mutex의 차이점

* 세마포어 : **Block/wake up 알고리즘** ( 준비상태와 실행, 대기 상태를 넘나듦 )
* Mutex : **Busy Waiting 알고리즘** ( 준비상태와 실행상태를 넘나듦 )

세마포어는 다양한 목적으로 활용 가능

1. n > 1 : 자원이 여러 개 있을 경우 **자원과 대기자를 동시에 관리** 가능

ex) 프린터가 동시에 3개까지 돌아갈 수 있다고 할 때, ( value = 3 ) 프린터를 쓰려는 모든 프로세스들이 순서대로 자원을 사용할 수 있게끔 세마포어가 관리해줌

1. n == 1 : **공유변수에 대해 mutex 구현**

|  |  |
| --- | --- |
| Process A | Process B |
| wait (Mtx);  <<<< critical section; >>>>  signal (Mtx);  remainder section | wait (Mtx);  <<<< critical section; >>>>>  signal (Mtx);  remainder section |

* + 자원이 1개이므로 Process A의 자원요청(wait연산)을 수행하고 나면 Mtx가 0이 되어 Process B는 wait연산을 수행하지 못하고 대기큐에 진입, Process A가 자원반환(signal)연산을 마쳐야만 Process B의 wait연산이 수행될 수 있다. ( Mutex와 동일 )
  + **공유변수가 곧 공유자원**

1. n == 0 : event에 의한 **다중 쓰레드 간의 serialization**에 사용
   * Process A -> Process B 순으로 수행되어야 한다고 할 때, 역으로 Process B에서 먼저 자원 요청(wait 연산)을 하고 Process A에서 Signal 연산을 수행한다.
   * Process B의 wait연산을 수행하면 자원이 -1로 음수가 되어 임계구역이 실행되지 못하고 대기큐로 들어가게 된다. 후 Process A가 먼저 실행되고 나서 Signal 연산을 수행하면 value = 0이 되어 Process B가 대기큐에서 빠져나와 실행된다.

Producer-Consumer 문제에서 발생할 수 있는 ISSUE :

* 사용할 수 있는 item이 없어 대기큐에 존재하던 Consumer 스레드는 Producer스레드가 item을 생성하고 signal요청을 할 경우 깨어나야 정상이나, mutex로 임계구역이 설정되어있는 경우에는 Consumer스레드가 깨어날 수 없음
* SOLUTION : pthread\_cond\_wait()
* Pthread\_cond\_wait() : 스레드가 휴면상태가 되도록 하며, 휴면상태는 pthread\_cond\_signal() 이라는 함수를 통해 깨어난다. 내부적으로 mutex를 잠금 해제하고 스레드 실행을 중지시킨다.

Readers-Writers Problem : 여러 개의 스레드가 읽기와 쓰기를 할 경우, 하나의 읽기 프로세스가 임계구역 내에 있을 때 쓰기 프로세스는 대기 BUT 다른 읽기 프로세스마저 대기하는 것은 병행성을 낮춤 – 성능저하

* 쓰기 프로세스는 모든 프로세스에 배타적 ( exlcusive lock 또는 writer’s lock) / 읽기 프로세스끼리는 공유 (shared lock 또는 reader’s lock) > 읽기/쓰기 모드를 명시하도록 함
* 읽기 프로세스가 계속 lock을 차지하면 쓰기 프로세스의 기아현상(무기한 연기) 발생, aging 으로 해결

**7강 교착 상태 ( DeadLock )**

**교착상태**

1. **프로세스 A가 자원을 요청하였으나, 프로세스 B가 선점해 사용할 수 없을 때 대기 상태로 진입**
2. **그 프로세스 B도 다른 프로세스에 의해 선점된 자원의 요청으로 인해 대기 상태로 진입**
3. **1-2 형태의 대기가 환형을 이루게 되면 대기 상태의 프로세스들은 영원히 대기 상태에서 벗어날 수 없음**

**교착상태 특징**

1. **상호 배제** : 할당 후 반환까지 한 프로세스만 사용하는 자원이어야 함
2. **점유와 (동시에) 대기 :** 적어도 하나의 자원을 보유하고 현재 다른 프로세스에게 할당된 자원을 얻기 위해 대기 중인 프로세스가 존재해야 함
3. **비선점 :** 자원은 강제로 빼앗을 수 없고 그 자원을 점유하고 있는 프로세스가 끝나야 해제됨
4. **환형 대기 :** 대기 프로세스의 집합이 있어서 P0은 P1이 가진 자원을 기다리고, P1은 P2, … Pn-1은 Pn, Pn은 P0가 가진 것을 기다림

**시스템 모델**

자원을 사용하려면 우선 요청을 하고, 사용 후에는 반드시 해제 ( 요청이 받아들여지지 않으면 대기 )

시스템호출을 통하게 되어 있어 **운영체제는 시스템 테이블에 자원할당 상황을 기록**

**자원 할당 그래프**

사이클이 없다면 -> 교착상태 발생 X

**사이클**이 있다면 -> 각 자원 종류마다 **하나씩의 자원밖에 없으면 반드시 교착상태** 발생

( 교착상태의 필요충분 조건 )

-> 각 자원 종류마다 여러 개의 자원이 있다면 교착상태가 발생할 가능성이 있음

( 교착상태의 필요조건 )

**교착상태 처리 방법**

* **교착상태가 되지 않도록 사전에 조치 ( 예방, 회피 )**
* **교착상태 방치 후 탐지하여 복구 ( 탐지, 복구 )**

**교착상태 예방**

1. ‘상호 배제’ 조건을 부정 (불가능)
2. **‘점유와 (동시에) 대기’ 조건을 부정**
   1. 모든 자원을 실행 전에 확보하도록 함 ( 대기 없음 )
      1. 자원이 할당된 후에도 한 동안 사용되지 않을 수 있어 **자원의 활용도가 낮음**
   2. 자원을 가지고 있지 않은 경우에만 요청하도록 함 ( 점유 없음 )
      1. 자주 쓰이는 자원을 여러 개 필요로 하는 프로세스의 경우 적어도 하나가 다른 프로세스에 할당되어 있는 경우가 빈번히 발생해 **기아 상태 발생의 가능성이 증가**
3. **‘비선점’ 조건을 부정**
   1. 자원을 보유하고 있는 임의 프로세스가 지금 할당 받을 수 없는 자원을 요청한다면 대기상태로 들어가면서 현재 할당 받은 것을 **강제로 반환토록 함**
      1. 대기상태에 있는 프로세스의 반환한 자원에 대해서는 선점이 가능
      2. 자원을 선점 당한 프로세스가 재시작되려면 원래 가지고 있던 자원 + 대기상태에 들어갈 때 요청했던 자원을 모두 할당 받아야함 ( **상태의 보존과 복구가 필요** )
4. **‘환형대기’ 조건을 부정** 
   1. 각 자원에 증가하는 순서의 번호를 부여한 뒤, 순서대로만 자원을 요청하도록 강제함
   2. 한 프로세스가 가진 자원 중 제일 번호가 큰 자원이 Ri면 그 프로세스는 Ri+1부터 Rn까지만 요청 가능
   3. 현재 요청하는 자원 Ri보다 번호가 같거나 큰 자원 Rj를 소유하고 있었다면 이를 모두 해제한 후 Ri를 요청

**Dining Philosophers Problems**

ISSUE : 각 스레드가 포크 세마포어 하나씩을 소유하고 다음 포크 세마포어를 wait할 경우 교착상태 발생

**SOLUTION1. AND 동기화**

* 필요한 모든 세마포어(자원)를 한꺼번에 모두 차지하거나 아예 하나도 갖지 않도록 하는 데 사용하는 P연산 ( wait연산 ) 필요
* ‘점유와 (동시에) 대기’를 부정

**SOLUTION2. 비대칭 해결안 : 번호 증가 방식**

* 비대칭 해결안 : 공평하지 않은 해결책
* 포크의 번호가 증가하는 순으로만 포크를 집도록 함
* ‘환형 대기’를 부정

**SOLUTION3. 비대칭 해결안 : 홀짝 방식**

* 짝수의 철학자는 오른쪽 포크를 집은 후 왼쪽 포크를, 홀수의 철학자는 먼저 왼쪽 포크를 집고 오른쪽 포크를 집도록 함.
* ‘환형 대기’를 부정

교착상태의 예방 대책은 **자원의 활용도를 낮게 하고, 시스템의 처리량을 감소시키거나, 기아상태 발생 가능**

**교착상태 회피**

* 현재 사용 가능한 자원, 각 프로세스에 할당된 자원, 각 프로세스의 미래 요구와 해제 정보를 활용해 **자원을 할당하지 않고 기다려야 할지 말지를 결정** ex) Banker’s Algorithm : 각 자원에 대한 최대 요구량을 선언토록 요구
* <P1, P2, … , Pn> 에서 n!개의 프로세스 순열 중 모든 Pi가

( 앞으로 요청할 수 있는 최대량의 자원 ) <= (현재 남은 자원 개수) + ( 프로세스가 반환할 자원 ) 일 때,

이 순열을 **안정 순열**이라 하고, **안정 순열이 존재하면 안정상태**라고 함

* 교착상태 회피는 **자원 할당시 시스템이 항상 안정상태에 있도록 유지하는 것**이 관건
  + 안정상태가 유지된다는 보장이 되면 바로 할당, 그렇지 않으면 대기
* 현재 유휴 자원이 있어도 기다려야 하는 상황이 발생, 시스템 이용도가 낮아짐

**교착상태 회피 : Banker’s 알고리즘**

* 구성 요소
  + Available[j]=k : **각 자원별 사용 가능 개수**를 기록하는 길이 m의 벡터
  + Max[i,j]=k : **Pi가 최대 k개의 Rj 를 요청할 수 있음**을 뜻하는 n\*m 행렬
  + Allocation[i,j]=k : **k는 Pi가 Rj 를 k개 할당받고 있음**을 뜻하는 n\*m 행렬
  + Need : **Max – Allocation**, 최대 요청 가능 개수를 뜻하는 n\*m 행렬
  + Request[i,j] = k : k는 Pi가 Rj 중 k개 할당을 요청하고 있다는 뜻
* 알고리즘
  + If ( Request i <= Need i ) {

If ( Reuqest i <= Available i ) {

Available i = Available i – Request i

Allocation i = Allocation i + Requst i

Need i = Need i – Request i

} else *Pi.wait()*

} else *illegal request*

* SAFETY Algorithm
  + 남은 자원으로 최대 가능 요구량이 만족되는 프로세스를 찾아 가진 것을 남은 자원에 더함, 모든 프로세스에 대해 이 과정을 반복할 수 있다면 안정상태

|  |
| --- |
| Work = Available;  Finish[i] = false;  for(j=0; j<n; j++){  for(i=0; i<n; i++){  if (**Finish[i]==false && Needi <= Work**)  break; }  Work = Work + Allocationi;  Finish[i]=true; } |

**모든 i에 대하여 Finish[i]=true면 안정상태**

**교착상태 탐지**

* 교착상태 예방, 회피 등을 사용하지 않을 경우 교착상태를 검사하는 알고리즘을 주기적으로 수행

|  |
| --- |
| Work = Available;  Finish[i] = (Allocationi != 0) ? False : true;  for(j=0; j<n; j++){  for(i=0; i<n; i++){  if (Finish[i]==false && Requesti<= Work)  break; }  Work = Work + Allocationi;  Finish[i]=true; } |

**임의의 i에 대하여 Finish[i] = false면 시스템이 교착상태 (Pi가 deadlock 되어있음)**

**‘회피’는 할당을 가정하고 안정순열 존재 여부를 통해 교착상태를 예측해보는 것이고,**

**‘탐지’는 현재의 Allocation과 Request 간에 교착상태가 있는지를 체크하는 것**

**교착상태 복구**

* **종료** : **교착상태 프로세스를 모두 중지하거나 교착상태가 제거될 때까지 한 프로세스씩 중지**
* **선점 : 교착상태에 있는 프로세스가 갖고 있는 자원을 계속적으로 선점하여 다른 프로세스에게 줌**
  + **후퇴 필요 :** 안정상태로 rollback 해야하니 각 프로세스의 상태 정보를 더 많이 관리해야 함
  + **기아 발생 고려 :** 비용에 근거한 선정시 동일 프로세스가 항상 선택될 가능성 있음

**8강 메모리 경영**

**주소 결속 : 논리주소가 물리주소로 확정되는 것**

* 논리주소 : CPU가 프로그램을 수행하면서 변수 등에 해당하는 특정 메모리를 접근하기 위하여 사용하는 주소
* 논리주소는 링크시점에 확정 ( 프로그램 시작 지점 : 0번지 )
* 주소결속 시점은 다양
  + 적재 시점에 결속할 경우

코드 내 논리주소들을 물리주소로 변환하여 적재 ( 적재 후 프로그램의 재배치가 불가능 )

* + 실행 시점에 결속할 경우

먼저 논리주소로 적재된 후 실행 중에 물리주소 확정 ( 재배치 및 부분적재 가능 )

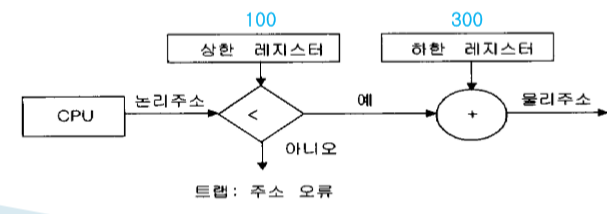
실행 시 하드웨어(MMU)의 지원을 받아 논리주소를 물리주소로 변환

**결속 방식에 따른 메모리 경영 기법 분류**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 분류 | 공간 크기 | 사상 단위 | 적재 단위 | 장단점 |
| 분할 방법 | 논리=물리 | 전체 프로그램 | 전체 프로그램 | 단점 : 외부 단편화 발생 |
| 페이징 | 논리=물리 | 페이지  (세그먼트 기법은 가변 크기) | 전체 프로그램 | 단점 :  사상테이블인 페이지 테이블 필요 |
| 가상 메모리 | 논리>물리 | 페이지 | 프로그램 일부 적재 | 장점 : 오버레이 가능, 부분 메모리 활용도 증대 |

**메모리 경영 기법 1. 분할 방법**

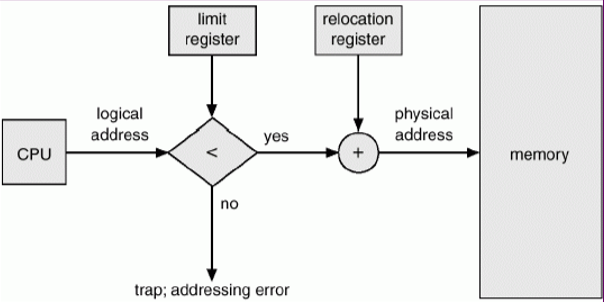
* 여러 개의 프로그램을 적재하기 위해 메모리를 여러 개로 분할
* 분할의 수만큼 다중 프로그래밍
  1. **고정 분할**
* 분할의 수와 각각의 크기가 고정
* 각 분할마다 별도의 큐를 두어 프로그램이 들어오면 해당 큐에 고정적으로 배정
* 비어 있는 분할의 크기에 가장 잘 맞는 프로그램을 먼저 할당하는 방법이 좋을 거라는 아이디어가 등장하면서 가변분할이 등장



* 상한 레지스터 : 분할의 크기 / 하한 레지스터 : 시작 주소
  1. **가변 분할**
* 메모리를 고정 분할하는 대신 다중 프로그래밍의 정도만 결정하고, **메모리 전체를 하나의 연속된 공간으로 보고 적절한 빈 장소에 프로그램을 적재**
* First Fit / Best Fit / Worst Fit 이 존재
  + First Fit : FCFS, 도착 순서대로 메모리에 프로그램을 적재
  + Best Fit이라고 항상 좋은 것은 아님 : Worst Fit은 잉여 공간이 많이 생기기 때문에 (확률적으로) 넒은 잉여 공간에 다른 프로그램을 적재할 수 있지만, Best Fit은 잉여 공간이 적어 재활용 불가 / 그렇다고 Worst Fit이 좋은 건 또 아님…
* **외부 단편화** 발생 : 잉여 메모리가 적재되어야 할 프로그램이 요구하는 용량보다 큼에도 불구하고 연속된 메모리가 아니라 프로그램을 수용하지 못하는 현상

**50% 규칙** : First Fit의 경우 통계적으로 N**개의 블록이 할당되었을 때 다른 0.5N개의 블록이 단편화 현상 때문에 사용할 수 없음** – 이론적으로 용량의 1/3에 해당하는 불용 공간이 발생

* **압축** : 잉여 공간들을 합해 하나의 큰 공간을 만든다. 상, 하한 레지스터 값을 변경하고 프로그램을 이동 ( 압축의 비용 : 압축을 위하여 움직여야 하는 프로그램 크기의 합 )



* CPU가 생성하는 **논리주소에 대해 상한 레지스터 값으로 영역을 체크**한 후, (분할 영역 보호)

**하한 레지스터 값을 더하여 물리 주소로 맵핑**

* **분할 방법의 단편화 문제의 근본적 원인인 ‘프로그램이 연속된 메모리에 적재되어야함’ 을 해결하기 위해서 페이징 등장**

**메모리 경영 기법 2. 페이징**

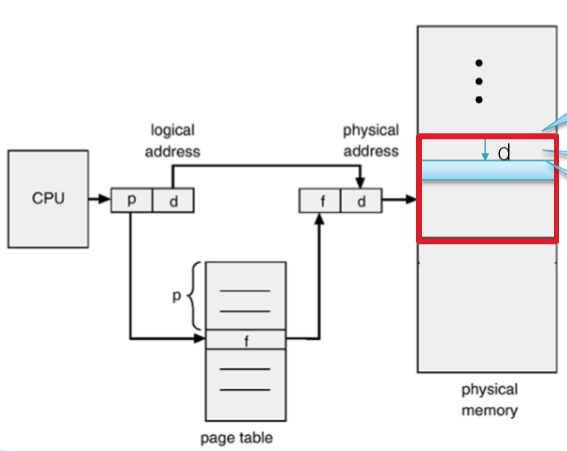
* 연속된 물리 공간이 필요하지 않고 실행 시간 주소 결속이 가능함
* 페이징의 구성 요소

1. **프레임** : 물리적 공간
2. **페이지** : 논리적 공간을 같은 크기로 나눈 것
3. **페이지 테이블** : 페이지와 프레임의 대응 관계를 나타내는 주소 변환 테이블, 임의 페이지에 해당되는 프레임의 시작 주소를 갖고 있음

* 주소 표현

1. 물리적 주소 : ( f , d ) = f \* S + d
   * f : 페이지 p에 해당하는 프레임의 번호 / S : 페이지 크기
2. 논리적 주소 : ( p, d )
   * + p : 페이지 번호 / d : 페이지 내의 상대 주소 (offset)
3. (기타) 오프셋(d) 표현을 위해 log2(S) 비트 사용

* **내부 단편화** : 외부 단편화는 일어나지 않으나 페이지 크기보다 페이지에 할당된 프로그램의 용량이 더 작을 경우 여전히 잉여 공간 발생 (평균적으로 각 프로세스에 대하여 프레임 크기의 반)



ISSUE : 논리적 주소 공간이 더 커진다고 했을 때, 페이지 테이블의 개수가 많아져 페이지 테이블이 차지하는 주소 공간이 너무 커지면, 성능이 떨어짐.

SOLUTION : 페이지 테이블의 한 entry들이 연속된 주소 공간에 존재할 필요가 없음 => 페이지 테이블을 나누자!

1. 2단계 페이징 주소 변환
   1. 논리주소의 구성 요소 : ( 루트 페이지테이블#, 4-K바이트 페이지테이블#, Offset )
      1. 루트 페이지 테이블의 시작 포인터 + 루트 페이지 테이블#
      2. i에서 얻은 값 + 4-K바이트 페이지 테이블# -> 해당되는 프레임의 시작 주소
      3. ii에서 얻은 값 + Offset = 최종적으로 참조하려는 물리적인 주소
   2. 페이지 테이블이 2개이기 때문에, 용량당 표현할 수 있는 물리 주소의 범위가 넓어짐
      1. 루트 페이지 테이블의 entry가 1024개이고 각 PTE당 크기가 4바이트
      2. 4-K바이트 페이지 테이블의 entry (1024 \* 4바이트)가 1024개이면 1024\*1024\*4바이트
      3. 최종적으로 표현할 수 있는 물리주소는 1024 \* 1024 \* 1024 \* 4바이트
2. 해쉬 페이지 테이블
   1. 논리주소 : ( 페이지#, Offset )
      1. 해쉬 테이블의 “페이지#를 해싱한 값”으로 맵핑
      2. 해시 테이블에서 연결 리스트의 첫번째 원소와 페이지 번호를 매칭한다.
      3. 매칭이 일어났다면, 그에 대응하는 페이지 프레임 번호 ( 두번째 필드 값 )을 가져와 물리적 주소를 얻는다.
3. 역 페이지 테이블 ( Inverted Page Table )
   1. 논리 주소 : ( 프로세스 PID, 페이지#, Offset )
   2. Hashing 없이
      1. 각 역 페이지 테이블의 항목은 ( PID, 페이지# ), 매칭을 시도할 때는 갖고있는 PID와 페이지#를 and조건으로 연결해서 매칭한다.
      2. 만일 매칭이 일어났다면, 그에 대응하는 페이지 프레임 번호를 가져와 물리적 주소를 얻는다.
   3. Hashing 있이
      1. N bits의 가상공간 (PID&페이지 번호)를 해쉬함수를 사용해 m bits의 물리공간 프레임번호로 맵핑함
      2. 해싱된 값이 같고 PID도 같다면, Frame#를 얻어 물리 주소를 얻는다. 반면 해싱된 값이 같 같은 PID가 다를 경우, Chain을 사용해 같은 해싱 값을 가지지만 다른 PID를 갖는 프레임 정보로 이동해 그곳의 PID 값과 비교한다.
   4. 역 페이지 테이블의 프로세스 식별자 : 해당 페이지를 소유한 프로세스 식별자 ( PID & Page# )
   5. 제어비트 : 유효, 참조, 변경, 보호, 잠금 등의 정보를 나타내는 필드
   6. 체인 포인터 : 다음 항목이 가리키는 인덱스 값이 null이면 다음 항목이 없음

* ISSUE : 페이지 테이블의 속도 개선 : 페이지 테이블 구현을 위해 메모리를 사용할 경우 모든 addressin에 2번의 메모리 접근이 필요 (페이지 테이블 한번, 물리 주소 한번) -> 속도 저하

SOLUTION : **연관 레지스터(TLB)**를 이용

* + - 연관 레지스터에서 사상 정보를 찾은 후, 없을 경우에 메모리 상의 페이지 테이블에서 찾음
* **페이징의 장점**

1. **단편화 문제를 해결**할 수 있음
2. **페이지의 공유**가 가능
   * + 프로그램이 수행 중 변경되지 않는다면, 여러 프로세스들이 동시에 사용가능
3. **페이지의 보호**가 가능
   * + 페이지 테이블에 보호용 비트(valid, invalid key)를 추가할 수 있음

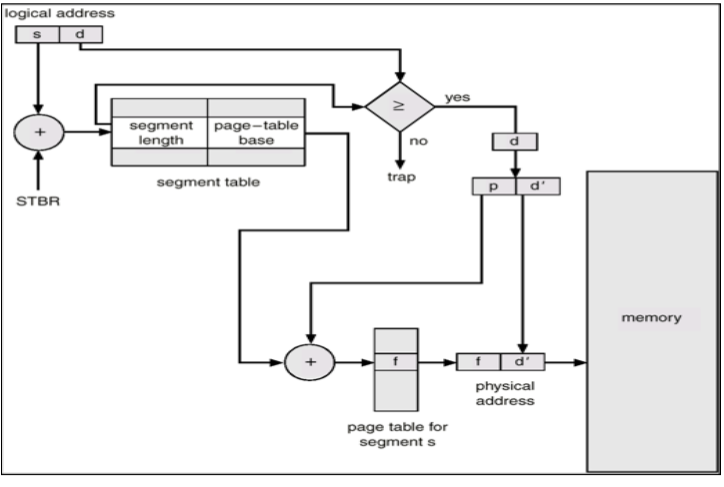
* 페이지의 단점 : 페이지 테이블 사용으로 분할 방식에 비해 주소결속에 따른 오버헤드 발생

**메모리 경영방식 2-1. 세그멘테이션**

* 주소결속 방식은 페이징과 동일하나 세그먼트 테이블에는 세그먼트 시작 주소와 **세그먼트의 크기 정보가 필요** (세그먼트마다의 크기가 전부 다름)
* 페이지와 마찬가지로 다른 프로세스들은 같은 세그먼트를 공유할 수 있음
* 페이징에서 해결된 **외부단편화 문제**가 다시 발생

**메모리 경영방식 2-2. Paged Segmentation**

* 세그먼트의 외부단편화를 해결하기 위한 방법
* 세그먼트 테이블이 페이지 테이블의 시작주소를 알려줌



**9강 가상 메모리**

**가상 메모리 : 물리 메모리에 일부의 페이지만 적재하고 적재 여부를 페이지 테이블에 표시**

* 구성 요소
  + **유표(Valid)/무효(Invalid) 비트**
    - **페이지가 메모리에 있는 경우**를 표시할 때는 유효(Valid) 표시
    - **페이지가 메모리에 없는 경우**를 표시할 때는 무효(Invalid)표시
  + **오염비트(dirty bit, modify bit)**
    - 페이지 대치 시 **디스크로 다시 출력(write back) 필요 여부 판정**이 필요
    - **페이지 적재 후 변경 유무**를 표시함

**요구 페이징**

CPU가 페이지A를 요구했을때, 페이지 테이블에 페이지A의 물리 주소 적재 여부가 적혀 있지 않다면( valid-invalid 비트가 invalid라면 )

-> “**페이지 부재(page fault)**가 발생!” -> 트랩을 걸어 해당 페이지를 적재함

|  |
| --- |
| 1. CPU가 페이지A를 요구, 페이지 테이블 참조 2. 페이지 부재가 발생했을 때 **트랩(ISR)을 발생시켜** 프로세스를 wait상태로 만들고 **요구된 페이지를**   **하드디스크(보조 저장 장치)에서 찾음**   1. 페이지 부재가 발생했던 페이지를 옮겨와 메모리에 적재함 2. 페이지 테이블을 갱신 ( valid-invalid 및 적재여부 기록 ) 3. 해당 프로세스를 다시 ready->running하여 명령어를 시작함 |

**페이지 대치**

Q. 물리 메모리에 여유가 없을 때 페이지 부재가 발생했다면, 페이지 적재할 곳이 없는데 이 때는 어떡하나요!?!?

A. ‘희생될 페이지’를 찾아서 대치해야한다! ( 페이지 대치 )

|  |
| --- |
| 1. 희생될 페이지를 내보냄  2. 유효/무효 비트( Valid-invalid bit ) 수정  3. 원하는 페이지를 적재. 이때, 오염비트를 참조하여 write시 변경사항이 있다면 write\_back을 호출한다. 적재가 완료된 후에는 오염비트에 변경사항이 있음을 표시한다.  4. 페이지 테이블 재설정 |

**“프로세스가 사용할 수 있는 최대한의 프레임 개수”**와 **“페이지 대치 알고리즘”**이 가상메모리의 성능을 결정함

* + 프로세스가 사용할 수 있는 최대한의 프레임 개수가 클수록 페이지 폴트 발생률은 떨어짐

**프로그램 부분 적재의 타당성**

* Locality of Reference 성질 : 프로그램의 어느 한 특정 작은 부분만 한동안 집중적으로 참조하는 현상
* 부분 적재의 이점
  + 프로그램의 크기가 물리적인 용량에 무관
  + 더 많은 프로그램이 메모리를 공유 – CPU 사용률과 처리율을 향상함
  + 주소 결속을 실행 시간에 할 수 있음
* 이는 곧 ‘Page-fault 발생률을 낮춰 성능을 높이는 것이 좋다’ 라는 결론을 낼 수 있음

**페이지 대치 알고리즘 1. 최적 대치 알고리즘 (Belady 알고리즘, B0 알고리즘)**

* “앞으로 가장 오랫동안 사용되지 않을 페이지를 찾아서 교체”하는 알고리즘
* Lookahead 알고리즘 (미래의 정보가 필요한 알고리즘)으로 실현이 어려움
* 대치 알고리즘의 상한선

Cf. 최악 대치 알고리즘 – 랜덤 알고리즘 (대치 알고리즘의 하한선)

**페이지 대치 알고리즘 2. FIFO**

* 메모리에 올라온 지 **가장 오래된 페이지를 택하여 대치**하는 알고리즘 (페이지가 프레임으로 적재된 순서 이용)
* Belady’s Anomaly(벨러디 변이) 발생 – 페이지 오류율이 할당되는 프레임의 수가 증가함에도 오히려 증가할 수도 있다!

**페이지 대치 알고리즘 3. LRU (Least Recently Used)**

* 가장 오랫동안 사용하지 않은 페이지를 대치하는 알고리즘 (적재된 순서가 아닌 참조된 시간 정보가 필요)
* 스택 알고리즘의 일종
* B0 알고리즘의 경우 미래의 정보가 필요한 알고리즘이지만 현실적으로 힘듦
  + “가장 최근의 과거”를 사용, 오랜기간 사용되지 않는 페이지를 대치함

(Locality of Reference : 가장 오랫동안 사용하지 않은 페이지는 앞으로도 상당한 기간 사용되지 X)

* B0과 LRU의 대칭성에 따른 성능의 유사성이 존재
* ISSUE : LRU는 개념적으로는 좋은 기법이지만 각 프레임들의 최후 사용시간에 따른 순서를 정하는 데에 하드웨어의 지원이 필수
  + 1. 참조가 일어날 때마다 페이지 테이블에 있는 사용시간 레지스터에 CPU의 논리시계 혹은 계수기 값을 기록
  + 2. 페이지 번호의 LRU스택을 유지. 마이크로 코드로 구현하여 오버헤드를 최소화
* SOLUTION : 참조비트 지원 (Additional Reference Bit 알고리즘)

**페이지 대치 알고리즘 4. Additional Reference Bit 알고리즘**

* LRU는 하드웨어의 지원이 필요 : 참조비트 등장
* 참조된 각 페이지의 참조비트가 하드웨어에 의해 1로 설정
  + 사용 순서는 모르지만 해당 기간 동안의 사용여부를 판별함으로써 부분적 순서정보 추출이 가능
* 일정 간격으로 참조비트를 참조바이트에 복사
* 최근의 8번 기간동안 그 페이지 사용 기록을 간직 ( 8회 이상 이전의 참조상황은 비교불가)
* 참조바이트 값이 가장 작은 페이지를 대치

**페이지 대치 알고리즘 5. Second Chance 알고리즘 (2차 기회 알고리즘)**

* 참조비트만 제공될 경우에 사용
* LRU에 근접함
* FIFO 형태로 빈 프레임 찾기를 계속함
  + 참조비트가 0이면 그 프레임의 페이지를 대치
  + 1이면 0으로 변경하여 다시한번 기회를 부여, FIFO 찾기 계속
* 프레임을 할당 받은 페이지는 4개 등급 중 하나가 되고, 등급 번호가 가장 낮은 것이 대치됨
  + (0,0) : 참조 x 변경 x
  + (0,1) : 참조 x 변경 o
  + (1,0) : 참조 o 변경 x
  + (1,1) : 참조 o 변경 o
* 가장 낮은 등급의 페이지가 여러 개 있으면 FIFO를 사용

Q. 페이지 오류를 발견하면 많은 오버헤드가 생성 -> 애초에 항상 빈 프레임을 갖고 있을 순 없음?

A. 페이지 대치 알고리즘을 사전에 실행시키자!

사전 적재

* 페이지 부재가 발생했을 때 한 페이지만 적재x -> 여러 페이지를 한 번에 적재
* 컴파일러에 의해 참조되는 서브루틴 및 행렬 정보까지 전부 적재

페이지 크기

* 사이즈가 작아지면 처음에는 페이지 수가 많으나 나중엔 locality에 의해 인접 부분 모두 포함
  + 시간이 지나면 페이지 폴트 발생이 낮아짐
* 사이즈가 커지면 필요한 페이지는 적어지나 locality 효과가 약화되어 페이지 폴트가 증가

프레임 할당

1. 동등 할당 : 모든 프로세스에게 똑 같은 개수의 프레임을 할당

2. 비례 할당 : 프로그램의 크기에 비례하여 프레임을 할당

3. 우선순위 할당 : 프로그램의 크기가 아니라 우선순위에 따라 프레임 할당

쓰래싱 (Thrashing)

* 프로세스들이 실행되는 시간보다 페이지를 교체하는 데에 더 많은 시간이 소요되는 현상
* 방지 : 각 프로세스가 필요로 하는 최소한의 프레임 개수를 알기 위해 “작업세트 (Working Set)” 등장
* 작업 세트 모델
  + Locality에 근거해서 가장 최근의 d개 페이지 참조에 들어 있는 페이지의 집합에서 d를 선택
    - d가 작으면 지역을 충분히 포함할 수 없고, d가 크면 지역이 서로 겹침
  + 각 프로세스의 작업세트를 계속 감시해서 각 프로세스에게 작업세트의 크기에 맞는 충분한 프레임을 할당한 뒤 여분이 있을 시 다른 프로세스들도 수행함
    - 각 작업 세트 크기의 총 합이 유효 프레임보다 큰 경우 – 스래싱 발생 감지
    - 해당 프로세스를 중지하던지의 방식으로 쓰래싱을 예방
  + 임의 프로세스에 대하여 오류율의 상한선과 하한선을 정하고, 상한선을 넘을시 더 많은 프레임을 할당하고 하한선을 넘으면 프레임을 회수하도록 함

커널 메모리 할당

* 커널 메모리 : 사용자 모드 프로세스에게 할당되는 페이지와는 별도의 메모리 풀을 사용함
* Buddy System 알고리즘
  + 세그먼트로부터 메모리 할당 시 2의 거듭제곱 단위로 할당함.
  + 반납시 병합(하나의 큰 세그먼트로 합쳐짐)
  + 장점 : 속도가 빠름 / 단점 : 단편화 발생
* Slab Allocation (슬랩 할당) 알고리즘
  + 슬랩 : 하나 또는 그 이상의 연속된 페이지
  + 캐시 : 하나 또는 그 이상의 슬랩들로 구성됨
  + 캐시가 생성되면 free로 표시된 커널 객체들을 캐시 내에 미리 마련해 놓음
  + 커널 객체가 요청되면 해당 캐시의 free 객체들 중에서 하나를 할당, 해당 객체를 used로 표시함
  + 커널 객체 할당 요청이 올 경우
    - Partial 슬랩의 free 객체를 이용해 할당
    - Partial 슬랩이 없으면 empty 슬랩의 free 객체를 할당
    - Empty 슬랩도 없으면 새로운 슬랩 하나를 캐시에 할당
  + 장점
    - 단편화 발생x (각 커널 객체마다 해당 캐시가 존재하기 때문)
    - 빠름 ( 객체들이 미리 생성되어 있고 캐시에서 쉽게 할당 가능함 )

**10강 파일 시스템**

**파일 : 정보의 집합체**

* 보조기억 장치 속에 저장되어 있고, 주로 프로그램 파일 또는 데이터 파일
* 하나의 파일은 디스크 내 **여러 개의 섹터**로 구성 – **논리적으로 블록에 해당**
* 파일의 추상화
* 가상메모리 : exe 파일을 구성하는 블록들이 프로세스의 페이지로 맵핑 되어지는 것
* 장치 파일 : 모든 입출력 장치를 파일이라는 개념으로 **일관성** 있게 추상화

**파일 시스템**

* 파일의 추상화를 위해 등장
* 저장장치는 선형적인 주소를 가진 바이트의 블록만을 저장
* 파일 시스템은 **저장장치와 응용프로그램 간의 자료구조 차이점을 해결**함
* Steam-block Translation : 기억장치의 블록을 음수가 아닌 정수 값을 갖는 연속된 주소를 가진 바이트의 집합으로 변환
  + File descriptor 생성 : 각 파일에 대한 상세 정보 (이름, 상태, 권한, 크기 등)을 저장한 자료구조
    - 저장 장치에 그 파일 내용과 함께 수록됨
    - 파일이 open된 경우 File descriptor에 추가 정보(상태, position 등)가 수록되고 file descriptor의 일부가 메모리에 로딩됨
    - 분산 시스템의 경우 일관성 유지를 위해 다양한 문제를 해결해야함
* Record-stream Translation : 응용프로세스의 자료구조로 변환

파일의 조작

1. 파일 생성 (create) : 파일공간을 할당한 후 디렉토리 생성, 파일의 이름과 파일 시스템 내의 위치를 기록
2. 파일 열기 (open) : File descriptor의 내용을 메모리 테이블에 복사, 이후 주소로 파일 참조
3. 파일 읽기 (read) : 시스템 호출 (파일의 이름과 기억장소의 위치 명시), 디렉토리 탐색
4. 파일 기록 (write) : 디렉토리 탐색 후 파일 포인터로 다음 블록의 주소 계산, 정보 기록
5. 파일 삭제 (delete) : 지명된 파일을 디렉토리에서 찾은 후 파일이 사용하는 모든 공간을 해제, 디렉토리 항목 무효화

**디렉토리 시스템**

* 디렉토리 : 자신에게 속하는 모든 파일과 부 디렉토리의 목록을 가진 자료구조의 파일 (폴더)
* **단단계 디렉토리** : 유지와 이해가 쉽지만, 모든 파일이 한 디렉토리 내에 있으므로 유일한 파일명을 가져야함
* **그래프 디렉토리** : 임의 파일이나 디렉토리가 서로 다른 두 개 이상의 디렉토리를 부모로 가질 수 있고 사이클을 형성할 수도 있음 ( 사이클이 없다면 트리형태 )
  + 융통성은 좋으나 구조가 복잡함
  + 경로가 여러 개이므로 서로 다른 경로 이름이 같은 파일을 가리키는 경우도 있음
  + 파일의 삭제 과정에서 **고아가 된 포인터**(dangling pointer)의 문제 발생
    - 경로 A의 파일 (혹은 디렉토리)을 삭제했을 경우, 같은 파일을 가리키는 경로 B는 고아가 된 포인터가 됨
* **계층 구조 디렉토리** : 하나의 root 디렉토리가 다른 디렉토리나 파일을 포인트
  + 한 디렉토리는 부 디렉토리를 포인트하며 계층이 내려가기를 반복
  + 한 디렉토리의 부모 디렉토리가 여러 개일 수 있어, 관리 및 사용의 복잡성이 여전히 존재함
* **트리 구조 디렉토리** : root 디렉토리는 부모 디렉토리가 없고, 모든 디렉토리나 파일은 한 개의 상위 디렉토리에 의해서 포인트됨 ( 경로가 유일 )

절대경로와 상대경로

* 절대경로 : root에서부터 지정된 파일까지의 모든 경로가 명시된 것
* 상대경로 : 현재 디렉토리의 위치를 기준으로 목적하는 파일까지의 경로를 나타낸 것

디스크 공간할당

* 디스크 공간할당 이전에 빈 공간 관리 방법 필요
* **효율적으로 디스크 공간을 이용**하며 **얼마나 빠르게 파일에 접근**할 수 있도록 할 것인가를 결정
* **연속할당** : 임의의 한 파일을 위해 디스크 내에 **선형적으로 연속된 블록을 할당**하는 방법
* **불연속할당** : 임의의 한 파일을 위해 디스크 내에 **산재된 블록을 연결하여 할당**하는 방법
  + 블록이 디스크 내 어느 곳에 있어도 액세스가 가능
  + 연결 할당, 색인 할당, 간접 색인 등이 있음

**연속 할당**

* 임의의 한 파일을 위해 디스크 내에 선형적으로 연속된 블록을 할당하는 방법
* 장점
  + 연속하는 논리적 블록들이 물리적으로 인접 -> **빠른 액세스가 가능함**
  + 디렉토리는 시작블록의 주소와 블록 개수만 유지 -> **디렉토리 구현의 단순화**
* 단점
  + 원하는 만큼 연속된 공간이 확보되지 않으면 파일 생성이 불가능 -> 필요한 공간의 크기를 미리 명시
  + 한 파일이 할당된 공간보다 커질 때, 그 파일이 들어갈 수 있는 새로운 영역으로 옮겨가야 함
  + **외부 단편화 : 파일의 생성과 삭제가 반복되면서 가용공간의 단편화 발생 -> 주기적 압축 필요**
  + **내부 단편화** : 확장을 대비하기 위해 필요한 공간보다 더 많은 여분의 공간을 확보하려고 하기 때문에 기억공간의 낭비 초래
  + **파일의 크기가 계속 변하는 경우에는 구현이 복잡 ( 연속된 공간을 할당해야 하기 때문에 발생 )**

**연결 할당 ( Linked Allocation )**

* 파일에 속해 있는 여러 섹터들이 서로 **연결된 리스트**로 되어 있고 **각 섹터 간에는 연결을 위한 포인터**를 갖고 있는 형태 ( 섹터는 논리적으로 블록에 해당 )
* 파일을 디스크 내의 산재된 블록을 연결하여 할당하는 방법
* 한 파일의 저장을 위한 블록들이 **디스크 내의 어느 곳에 흩어져 있어도 액세스**가 가능
* 장점
  + 압축이 필요하지 않음 : 파일이 더 확장될 경우 가용 공간 리스트로부터 가용 섹터들을 요청
  + 단편화 발생 X : 파일이 축소되면 섹터를 빈 공간 리스트로 반납
* 단점
  + 논리적으로 연속된 블록들의 검색에 긴 시간이 필요 : 파일의 블록들이 분산되어 있기 때문
  + 연결 리스트 구조를 유지하는 데 추가적인 시간이 필요
  + 연속할당 방법보다 기억장소의 낭비 : 리스트 내에 포인터 정보가 필요하기 때문

**색인할당 ( Indexed Allocation )**

* **색인 블록에 포인터(섹터주소)를 모아둠**으로써 포인터가 산재되어 비효율적인 연결할당의 단점을 해결
* 디렉토리는 그 디렉토리에 속한 각 파일의 색인 블록에 대한 포인터를 소유
* 메모리 경영에서의 페이지 테이블과 같은 기능
* 장점
  + **탐색이 빠름** : 탐색은 색인 블록 자체에서 일어남 ( 색인 블록은 주기억 장치에 유지됨 )
* 단점
  + 삽입하고자 할 때 **색인 블록을 완전히 재구성**해야 함
  + 작은 파일에도 하나의 색인 블록이 필요하므로 **기억 장치가 낭비**될 수 있음
  + 하나의 색인 블록으로 모든 블록을 색인할 수 없을 정도의 **큰 파일은 저장할 수 없음**

혼합색인

* direct block과 single, double, triple indirect까지 사용
  + direct block : 간접색인을 하지 않은 블록
  + single, double, triple indirect : 간접색인 1단계 / 2단계 / 3단계
* 간접색인 : 상위의 색인 파일이 하위의 색인 파일을 indexing
* Direct block : 파일의 크기가 작거나 자주 접근하는 파일이 직접 블록에 저장
  + 파일의 크기가 작다면 간접 블록에 저장하는 것은 기억 장치의 낭비
  + 파일의 사용이 잦다면 직접 블록에 저장하여 탐색 시간을 줄임
* Indirect block : 파일의 크기가 큰 파일은 간접 블록에 저장

빈 공간 관리

* **디스크 공간을 할당하고 회수**하여 관리하는 작업 ( 가변 분할 기법에 의한 메모리 할당 방법과 유사 )
* 파일들이 디스크 내에 인접해 있거나 연속된 장소에 저장되도록 하려면 작은 공간들을 서로 모아야 함
  + 남은 공간을 비워둔 채로 방치하게 되면 기억공간들의 단편화가 발생 -> 주기적 압축 필요
* 연결리스트
  + 사용가능 공간 리스트의 헤드를 운영체제가 알고 있고, 그 다음은 연결리스트로 구현
  + ‘적당한 위치’의 빈 블록을 찾아내기 위한 Search 성능이 안 좋음
* 그룹핑
  + 맨 처음 빈 블록에 n개의 블록에 대한 주소를 기입, n-1개는 빈 공간 블록의 주소, n번째에는 다른 n개의 빈 공간의 주소를 포함하는 블록의 주소를 기입하여 연결하는 방식
  + 많은 양의 사용 가능 공간을 빨리 찾을 수 있음
* 비트맵 (비트벡터)
  + 빈 공간 목록을 비트맵으로 구성하고 블록을 하나의 비트에 대응
  + 속도가 빠르나 비트맵이 메인 메모리에 로딩되어 있어야함
  + 2, 3, 5, 6 이 사용중이면 011011 저장
* 계수
  + 공간이 할당되거나 회수될 때 연속성이 크다는 점을 이용
  + 모든 사용 가능 블록에 대한 주소를 갖지 않고, 임의 빈 블록의 주소와 그로부터 연속하여 할당될 수 있는 빈 블록의 개수만 기록한 리스트를 관리
  + 5 6 7 의 블록이 비었다면 ( 5,3 )으로 저장

**11강 디스크와 RAID 경영**

디스크의 구성요소

* Platter상에는 Spindle을 중심으로 하여 Sector들이 원을 이루고 있음 : Track이라고 부름 ( 한 트랙의 섹터의 수는 동일! )
* Sector : 디스크가 입출력하는 단위
* Cylinder : 각 Platter 상의 동일 track들을 모아 Cylinder이라고 부름
* Track : Read-write head : sector을 read하거나 write하는 역할
* Arm : read-write할 track을 찾아가는 역할

섹터 접근 : 헤드를 트랙으로 움직이고 회전하는 트랙 상에서 원하는 섹터가 올 때까지 기다린 다음 작업 (read/write) 수행

시간지연요소

1. 탐색시간 (seek time)
2. 회전지연시간 (rotational latency)
3. 전송 시간 (transfer time)

디스크 스케줄링 : 어느 프로세스의 요청을 먼저 선택하여 실행할 것인지 순서를 결정하는 것

* 섹터는 디스크 상에 물리적으로는 순차적으로 저장되지 않음
* 동시에 수행 중인 프로세스들에 의해 발생되는 요청이 수시로 디스크 입출력 요청 큐에 도착
* 처리율의 극대화
  + 평균 반응시간 줄이기 : 어떤 요청이 디스크 구동기에 전해질 때부터 그 결과가 나올 때까지의 시간의 평균값을 줄이는 것
  + 반응시간 분산 줄이기 : 디스크 입출력 시간의 예측성을 향상시키는 것

FCFS 스케줄링

* 먼저 도착한 요청이 우선적으로 처리
* 특징
  + 대기 큐를 재배열 하지 않음
  + 탐색 패턴을 최적화하려는 시도는 없음
  + 요청의 우선순위와 상관없이 요청의 순서가 유지
* 장점
  + 공평성 보장, 프로그래밍 용이 : 요청 큐에 먼저 도착하면 먼저 처리
* 단점
  + 요청이 많아질 경우 평균 응답 시간이 길어짐
* 효율이 낮은 디스크 스케줄링 방법

SSTF 스케줄링

* 탐색거리가 가장 짧은 요청이 먼저 처리
* 특징
  + 헤드가 요청을 처리하기 위해 먼 곳으로 이동하기 전에 현재의 헤드 위치에서 최소의 탐색 시간을 요하는 요청을 골라서 먼저 처리
* 장점
  + FCFS보다 처리량이 많고 평균 응답시간이 짧음
* 단점
  + 안쪽이나 바깥쪽의 트랙이 가운데에 있는 트랙보다 처리를 덜 받을 수 있어 응답시간에 편차가 생김 – 안쪽이나 바깥쪽 트랙에 대한 요청은 기아 현상 발생 가능
* 처리량이 주요 목표인 일괄 처리 시스템에 유용하고, 응답시간의 편차가 크므로 대화형 시스템엔 부적합

SCAN 및 LOOK 스케줄링 (엘리베이터 알고리즘)

* SSTF와 비슷하지만 헤드 진행 방향 상의 가장 짧은 거리에 있는 요청을 먼저 처리
* 차이점
  + SCAN : 움직이는 방향의 앞으로의 더 요청이 없으면 인위적으로 헤드를 디스크 끝까지 이동한 후 방향을 바꿈
  + LOOK : 움직이는 방향의 앞으로의 더 요청이 없으면 디스크 끝까지 가지 않고 그 자리에서 바로 반대 방향으로 헤드를 움직임
* 특징
  + 대기큐의 동적인 특성을 효율적으로 방향
  + 처리량과 평균 응답시간을 개선했다는 SSTF의 장점을 가지지만 SSTF에서 발생하는 편차를 줄임
  + BUT, 제일 안쪽과 바깥쪽의 트랙에 대한 응답시간 편차는 여전히 있음
  + 헤드가 높은 편차를 갖고 움직이는 경우는 없어짐

C-SCAN 스케줄링

* 헤드가 마지막 트랙으로 이동하게 되었을 때, 반대 방향으로 가지 않고 다시 0트랙부터 움직임
* 복귀시간이 필요하나 처리시간이 공평함
* 대기시간이 균등해짐

회전지연시간 최적화를 위한 알고리즘

1. SLTF 스케줄링 ( Shortest-Latency-Time-First )

* 모든 요청들을 도착 순서와 상관 없이 섹터위치에 따라 큐에 넣고 회전 지연 시간이 가장 짧은 요청을 (가장 가까운 섹터에 대한 요청을) 먼저 서비스
* 구현하기 쉽고 최적 전략에 근접함

1. SPTF 스케줄링 ( Shortest-Positioning-Time-First )

* **탐색시간과 회전지연시간**의 합이 가장 짧은 요청을 다음 서비스 대상으로 선택

**(위치 결정시간)**

* 처리량이 많고 평균반응시간이 짧으나, 안쪽과 바깥쪽 실린더에 대한 요청에 기아 현상 발생 가능

1. SATF 스케줄링 ( Shortest-Access-Time-Fist )

* 탐색시간과 회전지연시간과 전송시간의 합이 가장 짧은 요청을 다음 서비스 대상으로
* 3가지 시간지연 요소를 모두 고려하는 스케줄링 방법
* SPTF와 SATF 모두 지연시간, 트랙간 탐색시간, 섹터의 상대적 위치 등 디스크에 관한 지식이 필요

직접적인 성능 강화로 디스크의 회전속도**(RPM)**를 높여 회전 지연 시간을 줄일 수 있으나, RPM을 높이려면 많은 전력 소비 및 열, 소음 문제가 발생 -> 현실적으로 어려움

RAID

* 좋은 성능과 신뢰성을 동시에 제공하기 위해 여러 디스크를 사용
* 하드 디스크의 전송률은 느린 속도로 발전, 요구 속도를 따라잡지 못해 처리량과 전송률 개선을 위해 여러 디스크를 동시에 접근할 수 있는 디스크 배열 사용
* 디스크 배열 사용 -> 성능 향상 & 신뢰성 향상
  + 데이터 스트라이핑
    - 스트립 : 디스크를 고정크기 블록으로 나눈 것
    - 스트라이프 : 디스크 배열 내 각 디스크 상에 동일한 위치에 있는 스트립의 집합
    - 파일을 스트립으로 분할하고 연속된 스트립을 스트라이프로 배치해 파일에 대한 요청이 한번에 여러 디스크를 통해 동시에 이루어짐 ( 병렬화 -> 접근 시간 개선 )
    - 주로 block-level striping
  + 중복성
    - 전송률은 높아지지만 배열의 디스크 수가 증가할수록 디스크가 고장 날 확률도 증가
      * 배열 중 한 디스크가 고장나면 그 디스크에 포함된 스트립을 갖고 있는 파일은 손상됨
    - 디스크 미러링(Dist Mirroring)으로 장애내구성을 높임
      * 하나의 데이터 항목들을 디스크 두 개에 저장
      * 전체 디스크 공간을 1/2만 사용하지만 장애 내구성은 높아짐
* RAID 컨트롤러
  + 데이터 스트라이핑 및 중복성 실현을 CPU가 하면 시스템 성능 저하가 우려되기 때문에, RAID 컨트롤러가 이를 담당

RAID Level 0

* 중복성 없이 스트라이프 된 디스크 배열을 사용
* 장애 내구성을 높이는 구조가 아니므로 진정한 RAID Level은 아님
* 스트립 크기에 따라 한 디스크만 손실되어도 모든 데이터가 손실될 수 있음
* 저렴한 비용으로 속도를 내야하는 경우 사용, 중복성 오버헤드는 없음

RAID Level 1

* 디스크 미러링을 사용해 중복성 실현, 디스크들이 미러 쌍으로 이루어 이중화됨
* 데이터 일관성을 위해 쓰기 요청은 미러링된 두 디스크에 한번에 써야 함
* 각 데이터 블록을 디스크 쌍에 저장, 디스크가 하나만 손상되어도 데이터 손실을 막을 수 있음
* 장점
  + 높은 장애내구성 : 여러 드라이브의 고장에도 데이터 손실이나 가용성 손실을 견딜 수 있음
  + 빠른 Read : 미러 쌍의 디스크에 동시 접근하여 서로 다른 데이터를 read할 수 있음
* 단점
  + 저장소 용량당 비용이 높은 고비용 방식 : 디스크 용량의 절반만 사용할 수 있어 많은 저장소 오버헤드 야기
  + 느린 Write : 미러 쌍에 대한 쓰기 요청은 한번에 하나씩 수행해야 함

RAID Level 2 (비트수준 해밍 ECC 패리티)

* 바이트 혹은 워드 단위로 스트라이프하고, 비트 단위의 패리티 사용 -> 패리티 디스크 필요
* 미러링이 없으므로 저장소 오버헤드 문제가 줄어듦
* 해밍 ECC를 패리티 비트로 사용
* RAID 컨트롤러가 쓰기 작업을 수행하려면 배열의 모든 데이터 디스크에서 읽어야 하고 모든 패리티 디스크에 써야 함
  + 읽기-수정-쓰기 사이클로 인해 성능이 저하됨 (매 쓰기 작업마다 디스크 배열에 2번 접근 필요)

RAID Level 3 (비트 수준 XOR ECC 패리티)

* 패리티 생성을 위해 XOR ECC를 사용해 해밍 ECC보다 훨씬 간단한 연산을 수행
* 장점
  + XOR 의 특성상 구현이 쉽고, Level 2와 비교했을 때 적은 저장소 오버헤드로 비슷한 신뢰도
  + 큰 파일을 읽거나 쓸 때 전송률이 높음
* 단점
  + 한번에 한 요청에 대하여 서비스 가능
  + 비트 단위의 스트라이핑 때문에 대부분의 읽기 요청은 배열 전체에 접근하게 되며, 패리티 생성 때문에 한번에 하나의 쓰기 요청만 수행 가능

RAID Level 4 (블록 수준 XOR ECC 패리티)

* RAID Level 3에서 Block-level striping을 하는 경우
* 블록 단위로 XOR ECC 패리티를 생성하여 패리티 디스크에 저장
* 해당 블록과 관련된 패리티 블록만 간단히 계산 가능해 배열 전체를 접근해야 하는 오버헤드 문제 해결
* 여러 읽기 요청을 동시 처리 가능 : 한 스트립 내에 더 많은 비트를 저장할 수 있어 임의 데이터가 같은 스트립에 저장
* 쓰기 요청시 패리티 갱신을 위해 하나의 패리티 디스크를 접근, 한번에 하나씩 수행 (쓰기 병목문제)

RAID Level 5 (블록 수준 분산 XOR ECC 패리티)

* RAID Level 4와 비슷하지만 패리티 블록들이 배열의 디스크 전체에 분산됨
* 패리티 블록이 여러 디스크에 분산되어 있어 여러 패리티 스트립을 동시에 접근할 수 있음, 쓰기 병목현상 제거
* 각 쓰기 요청에 대하여 읽기-수정-쓰기 사이클을 수행해야 하므로 한번 쓰기에 4번의 접근이 필요 (패리티 로깅, AFRAID 등의 개선 방법이 있지만 이 과목에선 다루지 않음)
* 장점
  + 성능과 비용, 신뢰성의 균형을 맞출 수 있어 범용 RAID로 인식됨, 서버나 비즈니스 시스템에서 자주 보임
* 단점
  + Level 2~4에 비해 성능은 높지만 구현이 어렵고 비용이 많이 듦
  + 패리티가 디스크 배열 전체에 흩어져 있기 때문에 다른 수준보다 데이터 재구축이 복잡

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| RAID 수준 | 읽기 병행 | 쓰기 병행 | 중복성 | 스트라이핑 수준 |
| 0 (스트라이핑) | YES | YES | 없음 | 블록 |
| 1 (미러링) | YES | NO | 미러링 | 없음 |
| 2 (비트 수준 해밍 ECC 패리티) | NO | NO | 해밍 ECC 패리티 | 비트 |
| 3 (비트 수준 XOR ECC 패리티) | NO | NO | XOR ECC 패리티 | 비트/바이트 |
| 4 (블록 수준 XOR ECC 패리티) | YES | NO | XOR ECC 패리티 | 블록 |
| 5 (블록 수준 분산 XOR ECC 패리티) | YES | YES | 분산 XOR ECC 패리티 | 블록 |