Operating System

Table of contents generated with markdown-toc

—

1. 운영체제란?

운영체제란 하드웨어와 사용자 응용프로그램 사이에서 인터페이스 역할을 해주는 것이다.

사용자 응용프로그램이 CPU, 메모리, I/O 입출력장치 등 자원을 효율적으로 사용할 수 있도록 관리해주는 것이다.

---

2. 운영체제의 역할

2-1. 프로세스 관리

- 프로세스/스레드

프로세스란 ?

메모리 영역에서 실행중인 프로그램을 프로세스라고 한다.

프로세스는 DATA, CODE, HEAP, STACK으로 구성되어있다.

- DATA : 전역변수 영역
- CODE : 프로세스의 작업 코드
- HEAP : 동적할당시에 사용되는 영역(malloc)
- STACK : 스레드의 작업 저장 (지역변수)

스레드란 ?

프로세스 안에서 실행되고있는 작은 단위의 일들을 말한다.

- 멀티프로그래밍/멀티프로세싱/멀티스레드

멀티프로그래밍이란?
CPU 하나가 여러 프로세스의 작업을 번갈아가면서 수행함으로써 CPU의 효율성을 높이는 것이다.

멀티프로세싱이란?

CPU의 여러대가 동시적으로 여러개의 프로세스의 작업을 수행하여 작업 효율을 높이는 것이다.

멀티프로그래밍과 멀티프로세싱의 차이점.

CPU가 여러 프로세스의 작업을 수행함으로써 작업 효율을 높이는 것은 동일하지만

멀티프로그래밍은 사실상 CPU 하나가 여러 프로세스를 번갈아가면서 작업을 수행하기때문에 같은 시간에서 보았을때 프로세스 하나만이 CPU를 점유하고 있다.

하지만, 멀티프로세싱은 CPU가 2대가 있다고 생각하면된다. 그렇기때문에 각 CPU는 프로세스 작업을 하나씩 맡아 수행하고 있기때문에 2개의 프로세스는 같은 시간에 동시적으로 수행되고 있는 것이다. (병렬처리 개념과 같다고 보면 될 것 같다)

멀티스레드란?

프로세스는 기본적으로 하나의 메인스레드를 포함하고있다.

프로세스는 앞에서 말했듯이 DATA, CODE, HEAP, STACK으로 구성되어있다.

멀티스레드는 프로세스 하나 안에서 스레드 여러개를 생성하여 작업을 더 효율적으로 하는 것에 목표를 두고 있다.

2개이상의 스레드는 DATA, CODE, HEAP영역은 공유하지만 자신의 작업을 독립적으로 저장하여 수행할 수 있도록 STACK만 개별적으로 갖고있다. DATA영역의 전역변수를 공유하고 있기때문에 Race Condition이 발생할 수 있다.

공유 변수에 대한 동시 접근으로 생기는 문제(Race Condition)를 막기 위해서 스레드들간의 동기화가 필요하다.

- Context Switching

Context Switiching이란 CPU가 현재 작업하던 프로세스 정보를 PCB에 저장하고 대기큐에있는 다음 실행되어야하는 프로세스의 PCB를 읽어와 수행하는 과정을 말한다. 즉, CPU의 점유하고 있는 프로세스가 바뀌는 것을 Context Switching이라고 한다.

PCB란?

Process Control Block, 말그대로 프로세스를 컨트롤 하기위한 정보를 담고있는 Block이다.

디스크에 있던 프로그램 실행파일이 메모리에 올라와 실행될때 운영체제는 커널 영역에 프로세스마다 PCB를 생성하여 정보를 기록하고

자원 스케줄링에 활용한다.

- Process ID : 프로세스의 고유 번호
- Process state : 프로세스의 상태 (running, ready, waiting, terminate)
- Process 우선 순위
- PPID, CPID : 부모 프로세스, 자식 프로세스
- Program counter : 다음 실행 될 명령의 위치

- CPU 스케줄링

CPU 스케줄링의 목적은 공평성, 효율성, 안정성, 확장성, 반응 시간 보장, 무한 연기 방지에 있다.

- 공평성 : 모든 프로세스에게 공평하게 CPU가 할당되어야한다. (Starvation이 없어야한다)
- 효율성 : 시스템 자원이 유휴시간이 없도록 효율적으로 스케줄링이 되어야한다.
- 안정성 : 우선순위가 높은 프로세스가 먼저 자원을 할당받도록 하여 중요한 작업과 자원을 보호해주어야한다.
- 확장성 : 프로세스의 개수가 늘어나더라도 안정적으로 시스템이 작동해야한다.
- 반응 시간 보장 : 적절한 시간내에 프로세스 요구에 응답하여야한다.
- 무한 연기 방지 : 특정 프로세스의 작업이 무한히 미뤄져서는 안된다.

CPU 스케줄링에는 크게 2가지가 있다.

- 비선점 스케줄링
- 선점 스케줄링

선점 스케줄링이란?

프로세스가 CPU를 점유하고 있다면 다른 프로세스는 CPU의 점유권을 뺏을 수 없는 스케줄링 방식이다.

- FCFS (First Come First Serve)
  - 대기큐에 먼저 들어온 프로세스부터 CPU를 점유하여 사용한다. 이 스케줄링의 가장 큰 문제점은 CPU를 점유하고 있는 프로세스의 작업이 길어지면 뒤에 대기큐에서 CPU 할당을 기다리고 있는 프로세스의 대기시간이 길어진다는 것이다. 이것을 Convoy Effect라고 한다.
- SJF (Shortest Job First)
  - 앞에 FCFS 스케줄링의 방식의 문제점은 Convoy Effect였다. 이를 해결하기 위해서 CPU를 점유하고 있던 프로세스의 작업이 끝나고 대기큐에 있는 프로세스들 중에서 가장 CPU 점유 시간이 짧을 것으로 예상되는 프로세스에게 CPU를 할당해 주는 방식이다. 이는 매우 효율적으로 보이겠지만 CPU 점유 시간이 길것이라고 예측되는 프로세스는 CPU 할당을 받지 못할 가능성이 매우 커지기 때문에 Starvation 현상이 나타나는 문제가 있다. 또한, CPU 점유시간을 예측하는 것은 사실상 매우 어려운 일이다.
- HRN (High Response ratio Next)
  - SJF 스케줄링 방식은 Convoy Effect를 해결해 주지만 Starvation 문제를 가져온다. Starvation 문제를 해결하기 위해서 Aging기법을 활용하는 스케줄링 방식이 HRN이다. HRN은 Response ratio가 가장 큰 프로세스에게 CPU를 할당해 주는 것이다.
  - Response ratio = (대기시간+CPU 점유 예측 시간) / CPU 점유 예측 시간
  - Response ratio는 대기시간이 길어지는 프로세스에게도 우선권이 갈 수 있도록 하는 방식이다. 하지만 SJF랑 비교해본다면 CPU 점유 예측시간이 긴 프로세스가 우선권이 커질 수 있는 문제가 다시 발생할 수 있다.

비선점 스케줄링이란?

프로세스가 CPU를 점유하고 있다하더라도 다른 프로세스가 CPU 점유권을 뺏을 수 있는 스케줄링 방식이다.

- RR (Round Robin)
  - 라운드 로빈 스케줄링 방식은 모든 프로세스에게 CPU를 점유할 수 있는 시간을 공평하게 나누어 주는 것이다.
  - 프로세스가 고정적으로 할당된 CPU 점유시간이 지나면 Time out으로 자신의 작업 상태를 PCB에 저장하고 대기큐로 옮겨서 대기한다. 그리고 다음 대기큐에 있는 프로세스가 똑같이 할당된 고정 시간동안 CPU를 점유한다.
  - RR 스케줄링의 중요한 것은 고정시간을 얼마로 할 것인가이다. 고정시간을 길게 잡으면 사실상 FCFS와 다를바가 없어진다. 하지만 고정시간을 짧게 잡으면 Context Switching이 빈번히 일어나 CPU의 작업 효율성을 떨어트릴 수 있다.
- SRT (Shortest Remaning Time)
  - SRT 스케줄링은 기본적으로 RR 스케줄링 방식을 따른다. SRT는 주기적으로 CPU가 작업중이던 프로세스이 남은 시간과 대기큐에 있는 프로세스들의 CPU 작업 예측시간을 비교하여 대기큐에 있는 프로세스의 작업 예측 시간이 작업중이던 프로세스의 남은 시간보다 짧다면 Context Switching하여 CPU 점유권을 뺏는 스케줄링 방식이다. 비선점 스케줄링에서 SJF의 방식과 마찬가지로 CPU 작업 시간을 예측하는 것은 매우 어려우며 작업 예측시간이 긴 프로세스는 Starvation 현상이 일어날 수 있다.
- 우선 순위 큐
  - 프로세스마다 우선순위를 부여하여 우선순위가 높은 순서대로 CPU를 할당받는다.
  - 보통 커널 프로세스가 사용자 프로세스보다 우선순위가 높다.
  - 우선순위가 낮은 프로세스들은 Starvation이 생길 수 있다.
- Multilevel Queue

    - Multilevel Queue 스케줄링 방식은 대기큐를 여러개로 생성하고 각 큐는 우선순위가 정해져있다.
    - 각 큐마다 스케줄링 방식을 달리할 수 있으며 Time slice도 달리 할 수 있다.
    - 보통 우선순위가 높은 큐는 반응속도를 높이기 위해서 Time slice를 작게 잡으며 우선순위가 낮은 큐는 FCFS 스케줄링 방식을 따른다.
    - CPU는 항상 우선순위가 높은 대기큐의 작업을 가져다가 수행한다. 우선순위가 가장 높은 대기큐에 프로세스가 없다면 다음 우선순위의 대기큐에 있는 프로세스에 할당되어 작업을 수행한다. 하지만, 수행 도중에 우선순위가 높은 대기큐에 프로세스가 대기하고있다면 하던 작업을 저장하고 우선순위가 높은 프로세스를 우선으로 작업을 수행한다.
    - 우선순위가 낮은 프로세스의 Starvation이 생길 수 있다.
- Multilevel Feedback Queue
    - Multilevel Feedback Queue 스케줄링에서는 프로세스가 CPU를 한번씩 할당받아 실행이 될 때마다 우선순위를 하나씩 낮은 대기큐에 들어간다. 물론 낮은 우선순위 대기큐에서 대기시간이 길어지면 높은 우선순위 대기큐로 이동할 수 있다.
    - Multilevel Queue 스케줄링과 마찬가지로 우선순위가 높은 큐에서 낮은 큐로 갈수록 Time Slice는 커진다. (높은 우선순위 큐에서는 반응속도를 짧게 함)
    - Starvation의 문제를 막을 수 있다.

- 교착상태

- 교착 상태가 일어나야하는 조건

    - 상호배제 : 자원 하나당 단 하나의 프로세스만 사용할 수 있다. (동시 점유 불가)
    - 비선점 : 자원을 어떤 프로세스가 점유하고 있다면 그것을 뺏을 수 없다.
    - 점유대기 : 다음에 필요한 자원을 얻기 전까지 현재 점유하고 있는 자원을 내려놓지 않는다.
    - 원형대기 : 자원을 얻기위해 대기하고있는 프로세스가 원형을 이루워서 대기하고있다.

- 교착 상태 예방

  교착 상태를 예방하는 방법은 교착상태가 일어나야하는 조건을 없애는 것이다.

    - 상호 배제 없앰 -> 모든 자원을 공유자원으로... (자원을 모두 공유한다는 것은 Race Condition 문제발생)
    - 비선점 없앰 -> 선점으로 바꾼다. (중요한 작업중에 자원을 뺏기면 문제발생)
    - 점유대기 없앰 -> 다음 자원을 얻기 위해서는 자신이 점유한 자원을 내려놓는다, 처음부터 프로세스가 필요한 자원을 모두 할당해준다. 하지만 프로세스가 자신이 필요할 자원을 미리 다 아는 것은 어려운 일이다. (자원 활용 효율 떨어짐, 많은 자원을 얻어야하는 프로세스의 Starvation)
    - 원형대기 없앰 -> 자원에 번호를 매긴 후 자원의 번호가 큰쪽으로만 요구할 수 있다. (자원 활용 효율 떨어짐)

- 교착 상태 회피

  이것은 은행원 알고리즘에서 온 것이다. 자원을 안전하게 배분할 수 있도록 하기 위해 어느정도 확보해 둔 상태에서 프로세스 요구에 자원을 할당해준다.

  은행원 알고리즘 예시를 들어본다면 대출을 원하는 사람들에게 돈을 빌려줄때 은행은 자원의 안정성을 위해서 기본적으로 확보해둘 자원을 고정하고 그 아래로는 떨어지지 않도록 돈을 빌려주는 방식이다.

    - 프로세스의 기대자원 : 앞으로 할당 받을 것이라고 예측되는 자원 수
    - 가용자원 : 할당해 줄 수 있는 자원 수
    - 기대자원<=가용자원 : 할당해준다. (안정상태)
    - 기대자원>가용자원 : 할당하지 않는다. (불안정상태)

- 교착 상태 검출

    - 타임아웃 : 일정 시간 동안 작업이 진행되고있지 않은 프로세스를 교착상태로 여기고 강제 종료 시킨다. (이유가 교착상태가 아니더라도 종료될 가능성이 크다, 분산네트워크 관계에서는 네트워크 문제일 수 도 있기때문)
    - 자원할당 그래프 : 자원 할당한 그래프를 통해서 원형이 이루어지고 있는지 확인 (사이클을 감시함으로써 오버헤드가 발생할 수 있다)

- 교착 상태 회복

  교착상태를 일으킨 모든 프로세스를 종료하거나 하나씩 순서대로 종료해봄으로써 교착 상태가 회복되는지 확인하는 방법이다.

    - 모든 프로세스를 종료하는 것은 다시 모든 프로세스가 재실행되었을때 다시 교착 상태를 일으킬 확률이 크다. 그렇기 때문에 다시 실행시킬 경우에는 하나씩 실행시켜야한다.

    - 교착 상태를 일으킨 프로세스 하나씩 종료시키는 것은 우선순위가 낮은 프로세스부터 종료시킨다. 우선순위가 같을 경우에는 작업 시간이 짧은 프로세스 먼저 종료하며 여기까지 조건이 같을 경우에는 자원을 많이 사용하는 프로세스를 먼저 종료한다.

  이 방법은 프로세스를 종료시키기전에 작업 상태를 저장해야하는 일이 필요하다. (복구를 위해서) 그렇기 때문에 시스템 부하가 올 수도 있는 문제이다.

- 프로세스 통신

- 프로세스 내부에 있는 스레드들은 전역변수(DATA) 영역을 공유함으로써 통신한다.
- 프로세스 간 통신은 파일 입출력이나 운영체제가 제공하는 파이프 방식을 사용한다.

- 네트워크 이동이 필요한 프로세스들 간의 통신은 소켓 통신을 한다.

- 전역변수를 이용한 통신은 Race Condition 관리가 중요하다.

  공동으로 관리하는 메모리를 사용하여 전역변수의 값을 스레드나 프로세스들이 함께 사용한다.

  스레드는 DATA 영역을 공유함으로써 전역변수를 이용한 통신을 하며 프로세스는 부모 프로세스가 전역변수를 선언한 뒤 fork()로 자식 프로세스를 만들면 같은 메모리 영역을 공유할 수 있다.

    - 전역변수가 언제 바뀔지 모르기때문에 마냥 기다려야하는 Busy waiting 문제가 발생한다.

    - Race Condition 관리

  Critical Section : 프로세스나 스레드가 동시로 공유 데이터에 접근하는 코드 부분

    - mutex : wait()/release(), lock()/unlock(), binary semaphore와 같다
    - semaphore : wait()/signal(), 동시에 접근할 수 있는 자원의 수만큼 설정
    - monitor : wait()/notify() 자바 동기화에 사용되는 방법

- 파일 입출력을 통한 통신

  open()을 하여 파일을 생성하고 write()를 하여 파일에 데이터를 입력한 뒤 디스크에 저장한다. 파일에 생성된 데이터 read()하여 디스크에서 파일을 읽어와 공유한다. 이 경우는 보통 부모-자식 프로세스에서하는 통신 방법이며 운영체제가 동기화에 관여하지 않는다. 동기화가 필요하다면 wait()을 통해서 순차적으로 작업

이 이루어질 수 있도록 한다.

- 파이프를 이용한 통신

  공유 메모리 사용과 달리 운영체제가 동기화를 제공해준다. 공유 메모리는 프로세스 내부에 heap영역에 공유 메모리 영역을 생성하거나 스레드들간에는 DATA 영역의 전역변수로 공유한다. 하지만 파이프를 이용한 통신은 커널 영역에 공유하는 공간을 따로 할당 받는다.

  - 파이프를 이용한 통신은 단방향 통신이다. 양방향으로 통신하기 위해서는 두개의 파이프를 생성해야한다.
  - 부모 자식간의 관계의 프로세스 통신 (익명 파이프 사용)

  ex) 부모 -> 자식

  부모 프로세스가 파이프를 생성하고 fork() 하여 자식 프로세스가 exec() 하여 파이프를 연결한다.

  - 부모 자식간의 관계가 아닌 프로세스들 간의 통신 (이름 있는 파이프 사용)

  서로 다른 프로세스가 파이프 이름을 알고 있다면 파이프 통신이 가능하다.

- 소켓을 이용한 통신

  1. 서버, 클라이언트- socket() : 양쪽 소켓 생성, 사용할 프로토콜 결정
  2. 서버- bind() : 서버가 특정 포트와 IP주소 병합
  3. 서버- listen() : 서버가 클라이언트 요청을 받을 준비
  4. 클라이언트- connect() : 연결 요청
  5. 서버- accept() : 연결 승낙 (접속한 클라이언트와 통신할 수 있도록 새로운 소켓을 생성한다)
  6. 서버, 클라이언트 - read(), write() : 데이터 송수신
  7. 서버, 클라이언트 - close()

2-2. 메모리 관리

- 절대주소와 상대주소

절대 주소는 메모리에 프로세스가 실제로 올라온 주소를 말하며 상대 주소는 프로세스가 주소지 0부터 시작한다고 가정하였을때 배정받은 주소값이다.

1. 사용자 프로세스가 참조하고싶은 데이터 상대주소를 요청한다.
2. 메모리 관리자는 요청한 데이터가 물리적으로 어느 주소에 있는지 확인 후 응답한다.

상대주소를 사용하는 이유는 커널 영역과 사용자 영역이 어떻게 변화할지 모르며 프로세스가 물리적 메모리에 적재되는 위치가 매번 바뀔 수 있기 때문이다.

- 가상 메모리

만약 프로그램 전체를 메모리에 올려 작업 실행을 한다면 물리적으로 메모리보다 큰 프로그램은 실행될 수 없다.

이를 해결하기위해 만들어진 개념이 가상 메모리이다.

현재 메모리에 남은 공간이 실행시키고자 하는 프로그램보다 작을 경우에 메모리 관리자는 디스크 영역의 일부분을 메모리 영역으로 지정하여 빌려다 사용한다.

(메모리는 용량이 작고 비싸지만 디스크 영역은 메모리보다 용량이 크며 싸다. 그래서 메모리를 늘리는 것보다 응답 속도가 조금 느려진다해도 디스크 영역을 활용하여 쓰는것이 더 효율적이라고 할 수 있다)

메모리 관리자는 실행이 필요한 부분은 메모리에 올린다. 이때 메모리 영역을 확보하기 위해서 불필요한 프로세스를 잠시 디스크에 빌린 가상의 메모리 공간으로 보낸다. 그 공간을 swap 영역이라 한다.

- 필요한 프로세스 -> 메모리 : swap in
- 불필요한 프로세스 -> 디스크 : swap out

- 메모리 분할 방식

프로세스의 크기들은 모두 제각각이기에 메모리 영역을 어떻게 할당해 주는 것인가도 매우 중요한 문제이다.

- 가변 분할 방식 : 프로세스 크기에 따라 메모리를 나누는 방식
- 고정 분할 방식 : 프로세스 크기와 상관없이 고정된 크기로 메모리를 나누어 할당하는 방식

- 외부 단편화 / 내부 단편화

- 세그먼테이션 기법(가변 분할 방식) - 외부 단편화 문제

  가변 분할 방식을 세그먼테이션 기법이라고 한다. 하나의 프로세스를 메모리에 연속되게 할당해주는 방식이다.

  - 연속 공간에 배치하였기 때문에 프로세스내에서 데이터 참조는 매우 편리하다.
  - 하지만 외부 단편화가 생기기때문에 프로세스를 재배치해야하는 복잡한 일이 생기는 것이 단점.
  - 프로세스 배치 방법
  - first fit : 프로세스가 할당될 수 있는 첫번째 메모리 공간에 할당
  - best fit : 단편화를 가장 적게 일으키는 최적의 공간에 할당
  - worst fit : 프로세스가 할당받을 수 있는 공간중 가장 큰 메모리 공간에 할당

  first fit은 가능한 공간을 모두 search할 필요는 없지만 best fit보다는 단편화가 많이 일어난다.

  best fit은 단편화를 줄일 수 있지만 최적의 공간을 search해야한다.

  worst fit은 배치하고 나서도 남는 공간이 크기때문에 효율적이라고 볼 수 있겠지만 적재되는 프로세스가 많아진다면 결국 단편화문제와 탐색 문제를 일으킨다.

    - Garbage collector

- 페이징 기법(고정 분할 방식) - 내부 단편화 문제

  메모리를 고정된 크기로 나누어 관리한다.

    - 가변 분할 방식처럼 프로세스를 재배치하지않아도 되는 장점이 있다.
    - 내부 단편화 문제가 발생할 수 있다. (낭비되는 메모리 공간이 생겨버림)
    - 프로세스는 각각 자신의 페이지 테이블을 가지고 있다. VA=<P,D> 가상주소를 PA=<F,D>로 매핑시킨다.

  프로세스가 늘어날 수록 커널영역 메모리에 적재되는 페이지 테이블도 늘어난다. 따라서 사용자 영역의 크기가 줄어든다.

  물리 메모리가 부족할 경우 프로세스만 스왑영역에 옮겨지는것이 아니라 일부 페이지 테이블도 스왑영역으로 옮겨진다.

  프로세스의 페이지 테이블을 빠르게 참조하기 위해서 프로세스의 PCB에 PTBR(process table base register)에 페이지 테이블 시작 주소를 저장해둔다.

    - 페이지 테이블 매핑 방식
    - 직접 매핑
    - 연관매핑
    - 집합-연관 매핑
    - 역매핑

  직접 매핑은 페이지 테이블 전체가 물리 메모리에 있는 것이다. 이는 메모리가 꽉차게 만드는 것이 단점이다.

  연관 매핑은 페이지 테이블 전체를 스왑영역에서 관리하는 것이다. 그리고 일부만 물리 메모리에 가지고 있는 방식이다. 연관 매핑은 페이지가 물리영역에 있는지 전체를 탐색해야하는 단점이 있다. 또한, 만약 없을 경우 스왑 영역에서도 다시 탐색해야하기 때문에 시간 낭비가 생긴다. (Table hit, Table miss)

  직접-연관 매핑 방식은 전체 페이지 테이블은 스왑 영역에 있지만 전체 페이지 테이블을 집합단위로 잘라 테이블을 생성하여 물리메모리에 적재해놓는 방식이다. 이것의 장점은 연관 매핑처럼 페이지가 스왑 인이 되어있는지 전체 탐색하지않고 바로 확인이 가능하다는 점이다. VA=<P1, P2, D> : P1은 탐색 페이지가 속한 집합 페이지, P2는 탐색 페이지, D는 탐색 변수

  역매핑은 프레임 기준으로 페이지 테이블을 생성한 것이다 <F, PID, P> F프레임에 PID 프로세스의 P 페이지가 적재되어있다는 뜻이다. 원하는 페이지가 없을 경우 스왑 영역을 다 탐색해야하는 안좋은 점이 있다.

현재 운영체제는 가변 분할과 고정분할을 혼합하여 사용하고 있다.

- 프로세스에 권한은 읽기/쓰기/실행 3가지이다. 이를 반영하기 위해서는 사실상 2^3=8bit가 필요하다. 하지만 읽기 없이 쓰기는 일어나지 않으므로 보통 경우의 수는 6가지이다.
- 데이터 영역은 읽기/쓰기 권한을 얻고 코드 영역은 읽기/실행만 허용한다.
- 권한 비트가 페이지마다 추가가 된다면 페이지 테이블의 크기는 커지기 마련이다.
- 세그먼테이션 테이블을 추가하여 <권한비트, 페이지 집합들의 첫 주소>

페이지 테이블 구성 요소

<페이지 번호, access, modify, valid, read, write, 프레임 주소필드>

- 요구 페이징

프로세스 요구에 응답 속도를 높이기 위해서 프로세스의 필요한 부분만 메모리에 올려 실행하는 기법을 가상 메모리, 페이징 기법이라고 앞에서 설명하였다. 그렇다면 메모리가 꽉차있는데 필요한 페이지를 스왑 인 해야하는 경우 어떤 페이지를 스왑아웃 시킬 것인가.

프로세스는 지역성을 띄고 있다

- 공간의 지역성

  현재 참조하고 있는 위치에서 주변 데이터를 참조할 가능성이 크다.

- 시간의 지역성

  현재 참조하고 있는 것은 가까운 시간내에 또 다시 참조되어질 가능성이 크다.

- 순차적 지역성

  코드는 순서대로 수행될 가능성이 크다.

- 페이지 교체 알고리즘

- FIFO

  가장 먼저 메모리에 적재되어있던 페이지를 스왑 아웃 시킨다.

    - 큐로 쉽게 구현할 수 있따는 것은 장점이다.
    - 자주 참조되는 페이지의 경우 다시 스왑 인을 시켜야하는 번거로움이 있다. (시간의 지역성으로 인해)

- 최적 페이지 교체 알고리즘

  앞으로 사용되지 않을 페이지르 스왑 아웃 시킨다.

  아주 best한 방법이지만 구현이 어렵다. 예측하는 것은 어렵기 때문에 이는 다른 구현 가능한 알고리즘과의 성능 비교대상으로 많이 활용된다.

- LRU (Least Recently Used page)

  가장 오래전에 참조된 페이지를 스왑 아웃 시킨다.

    - 페이지 별로 참조 된 시간을 저장한다. (카운터)
    - 시간이 지날수록 카운터의 숫자가 커지기 때문에 추가적인 메모리 공간이 필요할 수도 있다.

    - 카운터(시간) 대신에 참조 비트를 사용할수도 있다. 참조가 되면 참조비트를 하나씩 shift하면서 확인하는 방식이다. 이는 접근 시간이나 참조비트를 위해 추가적인 공간이 필요하다.

- LFU (Least Frequently Used page)

  최소 빈도로 참조된 페이지를 스왑 아웃 시킨다. 참조된 횟수를 카운트 한다.

    - 접근 횟수를 위한 메모리 공간이 추가적으로 필요하다. 하지만 이또한 공간 낭비를 줄일수는 있다.

- NUR (Not Used Recently)

  최근 미사용된 페이지를 스왑아웃시킨다.

    - 참조비트/변경비트 사용 (0,0)으로 초기화.
    - 스왑 아웃 우선 순위 (0,0)>(0,1)>(1,0)>(1,1)
    - 모든 페이지가 (1,1)이되면 모두 (0,0)으로 초기화

- FIFO 변형 알고리즘

    - 2차 기회를 주는 방식이다. page hit가 발생하면 큐의 마지막으로 이동되어 추가된다.
    - 큐 마지막으로 이동될때 추가타임이 생기는 것이 단점이다.

- 스레싱

프로세스를 많이 사용하다보면 페이지 부재로인해 스왑인과 스왑아웃이 자주 발생하여 작업이 멈춘거같이 느껴지게되는 현상을 스레싱이라고 한다. (멀티 프로그래밍이 커질 수록 효율성이 증가하다가 떨어지는 현상)

떨어지는 시점을 스레싱 포인트라고한다.

스레싱은 프레임 할당과 연관이 있다. 실행 중인 여러 프로세에 프레임을 얼마나 할당해주는가도 중요하다.

- 정적 할당

    - 균등할당
    - 크기가 작은 프로세스는 프레임이 남는 경우가 생겨버린다.
    - 크기가 큰 프로세스는 페이지 부재가 많이 발생한다.
    - 비례할당
    - 프로세스 크기에 비례하여 프레임을 할당하는 방식이다.
    - 요구 페이징 기법에서 페이지가 크더라도 사용 부분만 메모리에 적재하는데 페이지를 미리 할당해줘야하는 유동성이 없다.

- 동적할당

  페이지 부재 빈도가 잦은 프로세스에게 페이지를 더 할당해주는 방식이다.

- 전역교체 / 지역교체

  전역교체 : 전체 페이지에서 참조가 제일 낮은 페이지를 스왑아웃

  지역교체 : 요청한 페이지와 같은 프로세스 영역에서 참조가 제일 낮은 페이지를 스왑아웃

  전역 교체는 전체 페이지를 탐색해야하고 다른 프로세스 영역이 스왑아웃이 될 경우 다른 프로세스 스레싱을 일으킬 수 있다, 지역교체는 자신 프로세스 페이지 영역만 비교하면되는 장점이 있지만 다른 프로세스가 참조율이 더 낮을 때 오히려 요구 프로세스가 스레싱이 일어날 수 있다. 전체적으로 보았을때 전역교체가 더 시스템 효율적일 수 있다.

2-3. 입출력장치 관리

- 폴링

커널이 주기적으로 하드웨어의 입출력 상태를 체크하여 입출력을 관리한다. 시스템 부하가 일어날 수 있다.

- 인터럽트

CPU가 입출력이 필요한 경우나 디스크에서 데이터가 필요한 경우 시스템 콜하여 커널모드에서 처리 요청을 한다. 하드웨어가 입출력 작업이 끝나면 커널의 인터럽트 관리자에게 신호를 보낸다.

DMA 컨트롤러:

DMA 컨트롤러를 사용하여 CPU가 입출력장치와 직접 데이터 교환을 하는 것이 아니라 메모리와 입출력 장치가 데이터를 교환할 수 있도록 한다.

- IRQ (Interrupt Request)

  각 주변 장치들은 고유 인터럽트 IRQ 번호를 갖는다.

- 인터럽트 핸들러

  인터럽트 종류에 따라서 수행해야하는 함수를 벡터로 연결해놓은 것을 인터럽트 벡터라고 한다.

1. 주변장치가 인터럽트 발생을 일으킨다.
2. 커널의 인터럽트 처리 시스템이 인터럽트 핸들러에서 맞는 수행 함수를 연결하여 수행한다.

- 하드웨어 인터럽트 (외부 인터럽트)
    - 키보드 입출력
    - 전원 이상
    - 기계적 오류
- 소프트웨어 인터럽트 (내부 인터럽트) = Trap
    - 0으로 나누기
    - 허용하지 않은 참조영역 요청

    - 프로세스 예외상황
- 시그널 (자발적 인터럽트)
    - Ctrl+C 사용자 프로세스 강제종료

2-4. 저장장치 관리

저장 장치는 C부터 순서대로 이름을 부여한다. 파티션별로 파일 테이블을 만든다.

하드디스크가 많아지면 파일 테이블 관리가 어려워지기때문에 여러 파티션들을 하나로 통합하는 마운트 기술을 이용한다.

- C드라이브 : 운영체제와 각종 응용 프로그램 저장 (가상메모리의 스왑 영역도 C드라이브이다)
- D드라이브 : 문서, 사진 등을 저장
- E드라이브 : D드라이브 백업용으로 사용

페이지 테이블은 가상 주소가 디스크 물리적 영역에 어디 저장되어있는지 매핑시켜주는 역할을 한다.

- 포맷팅이란

  빈 저장 장치에 파일 테이블을 생성하는 것을 말한다. 파일 테이블은 각 파티션의 상단에 생성하고 모든 섹터를 0으로 초기화한다.

- 디스크 스케줄링

- FCFS (First Come First Service)

  요청이 들어온 순서대로 트랙을 탐색하여 응답한다.

  헤드가 움직이는 것이 비효율적이지만 단순한 방법이다.

- SSTF (Shortest Seek Time First)

  현재 헤드가 있는 위치에서 가장 가까운 트랙부터 방문한다.

    - 효율성은 FCFS보다 좋지만 멀리 있는 곳은 Starvation이 일어날 수 있다.
    - 일괄 시스템에서는 효율적일 수 있다.

- Block SSTF

  탐색 요청 트랙을 블럭으로 잘라 블럭 안에서만 SSTF를 수행한다.

- SCAN

  헤드가 한방향으로 움직이면서 트랙을 방문한다.

    - 중간에 있는 트랙들은 2번 방문할 수 있기 때문에 바깥 트랙들이 불공평해진다.

- C-SCAN

  SCAN처럼 한방향으로만 헤드를 움직이면서 탐색하지만 되돌아 올때는 서비스를 하지 않고 되돌아온다.

    - 공평해지는 장점은 있다.
    - 작업없이 헤드만 움직이는 것은 비효율적일 수 있다.
    - 동일한 트랙이 반복요청인 경우 매우 비효율적이다.

  그래서 C-SCAN은 잘 사용하지 않는 방식이다.

- LOOK

  SCAN방식에서 헤드가 트랙끝까지 가지않아도 요청이 없을 경우에는 헤드의 방향을 바꾼다.

- C-LOOK

  C-SCAN과 마찬가지지만 더이상 서비스 할 트랙이 없으면 중간에 헤드방향을 바꾼다.

- SLTF (Shortest Latency Time First)

  헤드가 움직이는 것이 아니라 헤드는 고정되어있고 디스크가 움직인다.

    - 디스크가 회전하는 방향대로 트랙을 방문한다.

2-5. 파일시스템 관리

---

3. 운영체제의 구조

3-1. 커널 영역 / 사용자 영역

- 경계 레지스터

사용자 영역이 커널 영역을 침범하는 것을 막기 위해 커널 영역과 사용자 영역을 구분하는 메모리 주소를 레지스테 저장해둔다.

사용자 응용프로그램이 참조 요청시 경계 레지스터로 참조 허용 범위인지 확인한다.

- 시스템 콜(System Call)

커널모드 0 / 유저모드 1

- 유저모드 > 커널모드

  사용자 응용프로그램이 입출력이 필요하거나 자원할당이 필요한 경우 System Call로 커널에 요청한다.

  ex) 파일 요청, I/O작업 요청, 통신 요청, 프로세스 복제 요청 등

- 커널모드 > 유저모드

  사용자 응용프로그램이 결과값을 System Call로 리턴해준다.