**OS**

**1. 프로세스와 스레드의 차이**

프로세스는 운영체제로부터 자원을 할당받는 작업의 단위이고, 스레드는 프로세스가 할당받은 자원을 이용하는 실행의 단위 입니다. 프로세스는 운영체제로부터 메모리, 주소공간등을 할당받고 쓰레드는 할당받은 자원들을 내부 스레드끼리 공유하면서 실행됩니다.

쓰레드를 사용하는 이유는 운영체제에서 더 효율적으로 시스템 자원을 관리하기 위해 사용된다고 할 수 있습니다. 스레드는 프로세스에 비해 생성할 때 오버헤드도 적고, 공유된 자원에 대해서도 오버헤드가 적습니다.

멀티 프로세스로 진행되는 작업을 멀티 쓰레드로 수행하게 되면 시스템 콜이 줄어들기 때문에 자원을 효율적으로 관리 할 수 있고 프로세스의 통신비용보다 쓰레드간의 통신 비용이 적다는 이점도 있습니다.

단 쓰레드간의 자원공유는 전역변수를 이용하므로 동기화 문제에 신경을 써야하며 멀티스레드 프로그래밍은 프로그래머의 주의를 요구합니다.

**2. 뮤텍스와 세마포어**

프로세스 혹은 스레드 간의 통신 시에 shared memory 등을 쓰는 경우 하나의 자원에 두 개 이상의 프로세스 혹은 스레드가 접근하는 경우에 문제가 발생합니다. 뮤텍스는 공유된 자원의 데이터를 여러 스레드가 접근하는 것을 막는 것이고, 세마포어는 공유된 자원의 데이터를 여러 프로세스가 접근하는 것을 막는 것입니다.

뮤텍스 - 상호배제라고도 하며, Critical Section을 가진 스레드의 Running Time이 서로 겹치지 않도록 각각 단독으로 실행하게 하는 기술이다. synchronized 또는 lock을 통해 해결한다.

세마포어 – 리소스 상태를 나타내는 간단한 카운터이다. 공유 리소스에 접근할 수 있는 프로세스의 최대 허용치만큼 동시에 사용자가 접근하여 사용할 수 있다.

**2-1 뮤텍스와 세마포어의 차이**

(1) 관리하는 동기화 대상의 개수 : Mutex는 동기화 대상이 오직 하나뿐일 때, Semaphore는 동기화 대상이 하나 이상일 때 사용한다.

(2) Semaphore는 Mutex가 될 수 있지만 Mutex는 Semaphore가 될 수 없다. : Mutex는 상태가 0, 1 두 개 뿐인 binary Semaphore

(3) Semaphore는 소유할 수 없는 반면, Mutex는 소유가 가능하며 소유주가 이에 대한 책임을 가진다. : Mutex 의 경우 상태가 두개 뿐인 lock 이므로 lock 을 가질 수 있다.

(4) Mutex의 경우 Mutex를 소유하고 있는 스레드가 이 Mutex를 해제할 수 있다. 하지만 Semaphore의 경우 이러한 Semaphore를 소유하지 않는 스레드가 Semaphore를 해제할 수 있다.

(5) Semaphore는 시스템 범위에 걸쳐있고 파일시스템상의 파일 형태로 존재하는 반면 Mutex는 프로세스 범위를 가지며 프로세스가 종료될 때 자동으로 Clean up 된다.

**3. 스레드를 구현하기 위한 방법**

(1) Runnable 인터페이스 구현

run 함수를 반드시 구현해야 함.

Thread 생성자의 매개변수로 입력

(2) Thread 클래스를 상속

Thread 클래스를 상속 받아서 구현.

자바는 다중 상속이 안되므로 이외의 클래스를 상속 받을 수 없음

**4. synchronize, Asynchronous에 대해 설명하시오**

**동기 :** 다수의 개채들이 동일(일정)한 무언가를 가지는 것. 또는 무언가가 동일(일정)하게 되는 것. 어떤 작업을 요청했을 때 그 작업이 종료될 때까지 기다린 후 다음 작업을 수행한다.

- 데이터를 주고받는 '순서'가 중요할때 사용된다.

**비동기 :** 동기가 아닌 것. 어떤 작업을 요청했을 때 그 작업이 종료될 때까지 기다리지 않고(작업을 위임하고), 다음 작업을 수행한다. 요청했던 작업이 끝나면 결과를 받고, 그에 따른 추가 작업이 있다면 수행한다.

- 요청 순서에 상관 없이, 동시에 다수의 작업을 처리할 수 있다.

- I/O 작업이 잦고, 빠른 응답 속도를 요구하는 프로그램에 적합하다.

**4-1. 각각의 장단점에 대해 설명하시오**

동기 : 요청한 작업만 처리하면 되기 때문에 전체적인 수행 속도는 빠를 수 있지만, 한 작업에 대한 시간이 길어질 경우, 전체 응답이 지연될 수 있다.

비동기 : 작업이 끝날 때 따로 이벤트를 감지하고 결과를 받아 그에 따른 추가 작업을 해줘야 하기 때문에, 비교적 느릴 수 있다.

**5. 데드락이란**

데드락(교착상태)란 첫 번째 스레드는 두 번째 스레드가 들고 있는 객체의 락이 풀리기를 기다리고 있고, 두 번째 스레드 역시 첫 번째 스레드가 들고 있는 객체의 락이 풀리기를 기다리는 상황을 일컷는다. 모든 스레드가 락이 풀리기를 기다리고 있기 때문에, 무한 대기 상태에 빠지게 된다. 이런 스레드를 교착상태에 빠졌다고 한다.

**5-1 데드락의 조건 4가지**

(1) 상호 배제(mutual exclusion

한 번에 한 프로세스만 공유 자원을 사용할 수 있다. 좀 더 정확하게는, 공유 자원에 대한 접근 권한이 제한된다. 자원의 양이 제한되어 있더라도 교착상태는 발생할 수 있다.

(2) 들고 기다리기(hold and wait) = 점유대기

공유 자원에 대한 접근 권한을 갖고 있는 프로세스가, 그 접근 권한을 양보하지 않은 상태에서 다른 자원에 대한 접근 권한을 요구할 수 있다.

(3) 선취(preemption) 불가능 = 비선점

한 프로세스가 다른 프로세스의 자원 접근 권한을 강제로 취소할 수 없다.

(4) 대기 상태의 사이클(circular wait) = 순환대기

두 개 이상의 프로세스가 자원 접근을 기다리는데, 그 관계에 사이클이 존재한다.

**5-2 데드락을 방지하는 방법**

- 4가지 조건들 가운데 하나를 제거하면 된다.

- 공유 자원 중 많은 경우가 한 번에 한 프로세스만 사용할 수 있기 때문에(예를 들어, 프린트) 1번 조건은 제거하기 어렵다.

- 대부분의 교착상태 방지 알고리즘은 4번 조건, 즉 대기 상태의 사이클이 발생하는 일을 막는 데 초점이 맞춰져 있다.

**6. 외부 단편화와 내부 단편화**

내부단편화 : 작업에 필요한 공간보다 많은 공간을 할당받음으로서 발생하는 내부의 사용 불가능한 공간. 페이지 내부에서 공간이 허비되는 것으로, 페이징 기법일 때 발생한다.

외부단편화 : 프로그램을 할당하고 난 다음 아주 작은 크기로 남은 조각들이 생길 경우, 연속적인 공간이 아니라서 사용할 수 없는 상황을 외부 단편화라고 한다. 세그먼트 기법일 때 발생한다.

**7. CPU 선점, 비선점 스케줄러의 차이**

선점 : 우선 순위가 높은 다른 프로세스가 할당된 CPU를 강제로 빼앗을 수 있는 방법

- 실시간 처리, 대화식 시분할 시스템에 사용됨

- 선점으로 인한 많은 오버헤드를 초래함

비선점 : 비효율적, 비양보. 프로세스에게 이미 할당된 CPU를 강제로 빼앗을 수 없고, 사용이 끝날 때까지 기다려야 하는 방법

- 일괄 처리에 적합

- 실시간 처리가 안되므로 중요한 작업이 기다리는 경우가 발생

**8. 선점 스케줄러 종류와 설명**

(1) SRT (Shortest Remaining time First) : SJF 스케줄링을 선점 형태로 변경한 기법으로, 현재 실행중인 프로세스의 남은 시간과 준비상태 큐에 새로 도착한 프로세스의 실행 시간을 비교하여 가장 짧은 실행 시간을 요구하는 프로세스에게 CPU 할당

- 문제점 : starvation. 새로운 프로세스가 도달할 때마다 스케줄링을 다시하기 때문에 CPU 사용시간을 측정할 수 없음

(2) Round Robin : FCFS 스케줄링을 선점 형태로 변경한 기법으로, Ready Queue에 먼저 들어온 프로세스가 먼저 CPU를 할당받지만 각 프로세스는 할당된 시간 동안만 실행한 후 실행이 완료되지 않으면 다음 프로세스에게 CPU를 넘겨주고 Ready Queue의 가장 뒤로 배치됨

- 장점 : Response time이 빨라짐, 프로세스가 기다리는 시간이 CPU를 사용할 만큼 증가. 공정한 스케줄링

- 문제점 : 할당 시간이 너무 커지면 FCFS와 같아짐. 또 너무 작아지면 잦은 context switch로 ovrhead가 발생함. 적당한 시간을 설정하는것이 중요함!

(2) Priority Scheduling

- 선점형 : 더 높은 우선순위의 프로세스가 도착하면 실행중인 프로세스를 멈추고 CPU를 선점

- 비선점형 : 더 높은 우선순위 프로세스가 도착하면 Ready Queue의 Head에 넣음

- 문제점 : starvation, 무기한 봉쇄. 실행 준비는 되어있으나 CPU를 사용 못하는 프로세스를 CPU가 무기한 대기하는 상태

- 해결책 : Aging. 아무리 우선순위가 낮은 프로세스라도 오래 기다리면 우선순위를 높여 줌

**9. 비선점 스케줄러 종류와 설명**

(1) FCFS (First Come First Served) : Ready Queue에 도착한 순서에 따라 CPU 할당

- 장점 : 공평성 유지

- 문제점 : convoy effect. 소요 기간이 긴 프로세스가 먼저 도달하여 효율성을 낮춤

(2) SJF (Shortest-Job-First) : 실행시간이 가장 짧은 프로세스에 먼저 CPU를 할당

- 장점 : 가장 적은 평균 대기시간을 제공하는 최적의 알고리즘

- 문제점 : starvation. 특정 프로세스가 지나치게 차별 받음. 사용시간이 긴 프로세스는 거의 영원히 CPU를 할당받을 수 없음

(3) HRN (Hightest Response-ratio Next) : 우선순위 = (대기시간 + 서비스 시간) / 서비스 시간 실행 시간이 긴 프로세스에 불리한 SJF 기법을 보완하기 위한 것. 대기시간이 길수록 우선순위가 높게 나옴

**10. 페이지 교체 알고리즘에 대해 설명**

- 페이지 부재가 발생했을 때, 가상 기억장치의 필요한 페이지를 주기억장치에 저장해야 하는데, 이 때 주기억장치의 모든 페이지 프레임이 사용중이라면 어떤 페이지 프레임을 선택하여 교체할지 결정하는 기법

(1) OPT (OPTimal Replacement, 최적 교체) : 앞으로 가장 오랫동안 사용하지 않을 페이지를 교체

- 장점 : 가장 낮은 페이지 부재율

- 단점 : 각 페이지의 호출 순서와 참조 상황을 미리 예측해야 하므로 구현이 어려움

(2) FIFO (First In First Out) : 가장 먼저 들어와서 가장 오래 있었던 페이지를 교체

- 장점 : 쉽고 간단함

- 단점 : Belady의 모순 페이지를 저장할 수 있는 프레임의 갯수를 늘려도 되려 페이지 부재가 더 많이 발생하는 모순

(3) LRU (Least Recently Used) : 최근에 가장 오랫동안 사용하지 않은 페이지를 교체. 최적 알고리즘의 근사 알고리즘.대

- 대체적으로 FIFO 알고리즘보다 우수, OPT 알고리즘보다 그렇지 못함

(4) LFU (Least Frequently Used) : 사용 빈도가 가장 적은 페이지를 교체

- 단점 : 프로그램 실행 초기에 많이 사용된 페이지가 그 후로 사용되지 않을 경우에도 프레임을 계속 차지할 수 있음

(5) NUR (Not Used Recently) : 최근에 사용하지 않은 페이지를 교체

**10-1. 페이지 fault 횟수 구하는 문제**

**11. 페이징과 세그먼테이션에 대해 설명**

페이징

- 논리(가상) 메모리는 페이지(Page)이라 불리는 고정 크기의 블록으로 나누어지고, 물리 메모리는 프레임(Frame)라 불리는 페이지과 같은 크기의 블록들로 나누어짐. 보조 메모리 역시 프레임과 같은 크기의 블록들로 나누어짐.

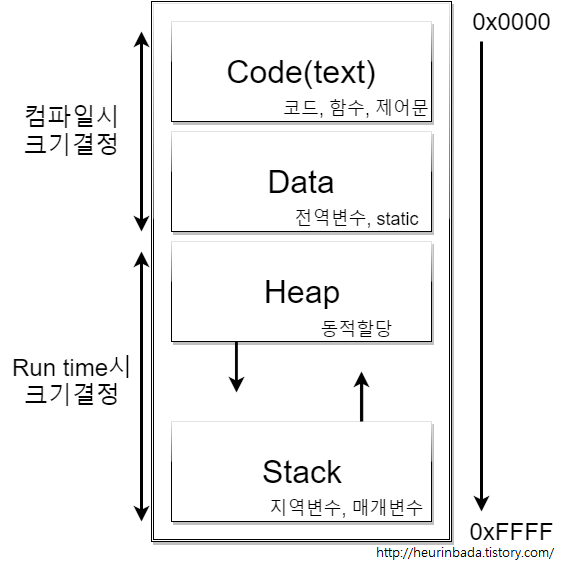
- 할당은 항상 프레임의 정수 배로 할당되는데, 이 때 프로세스가 페이지 경계와 일치하지 않는 크기의 메모리를 요구하게 되면 마지막 페이지 프레임은 전부 사용되지 않고 남아버리는 문제가 발생한다.(내부 단편화)

세그먼테이션

- 페이징에서처럼 논리 메모리와 물리 메모리를 같은 크기의 블록이 아닌, 서로 다른 크기의 논리적 단위인 세그먼트(Segment)로 분할

- 서로 다른 크기의 세그먼트들이 메모리에 적재되고 제거되는 일이 반복되다 보면, 자유 공간들이 많은 수의 작은 조각들로 나누어져 못 쓰게 될 수도 있다.(외부 단편화)

**12. 메모리 영역의 구조에 대해 설명**



프로그램이 실행되기 위해 프로그램이 메모리에 로드가 되어야합니다. 따라서 운영체제에서 프로그램의 실행을 위해 다양한 메모리 공간을 제공하고, 코드, data, stack, heap 영역을 할당합니다.

코드 영역 – 실행할 프로그램의 코드가 저장되는 영역. 컴파일 시 크기 결졍

data 영역 – 전역변수, 정적변수, 정적 배열, 정적 구조체 md 컴파일 시 크기가 결정되는 것들에 대한 영역

stack 영역 – 함수의 호출과 관계되는 지역변수와 매개변수가 저장되는 영역. 지역변수, 매개변수, 리턴 값과 같은 임시로 사용하는 것들에 대한 영역. 매개변수와 리턴값은 함수 호출시 생성하고 함수 수행이 끝나면 반환함. 런타임 시 크기가 결졍

heap 영역 – new, malloc 등 동적 할당 객체에 대한 영역. 런타임 시 크기 결정