운영체제론

운영체제란?
컴퓨터 하드웨어를 관리하는 소프트웨어이다.
컴퓨터 유저와 하드웨어 간 중재자 역할을 한다.
항상 실행되고 있는 프로그램
통상적으로 커널이라고 불림.

컴퓨터 시스템은 크게 4가지의 요소로 구성된다.
1. 하드웨어
2. 운영체제
3. 응용프로그램
4. 유저

H/W자원 : cpu, 메모리 , i/o

구조
유저 – 시스템, 응용프로그램 – 운영체제 – 하드웨어

프로그램의 정의 : a set of instruction
프로세스의 정의 : 실행 중인 프로그램(프로그램인데 메모리에 적재되어있는 상태!!!)

프로그램의 두가지 종류
1. 응용프로그램
2. 시스템 프로그램

부팅프로그램(부트스트랩 프로그램)
부팅과정
1. 전원의 인가 > cpu에 전기신호가 제일 처음 도달된다.
2. **cpu의 기본 동작은 메모리에 접근하여 프로그램을 실행**하는 것!
(*램은 휘발성 메모리이므로 전원을 인가했을때 아무정보도 남아있지 않다.)
3. 따라서 eeprom에서 부트스트랩 프로그램이 실행된다. 각종 디바이스(hdd)를 접근, 사용할 수 있는 초기화 코드(대충 초기화)가 실행된다.
4. hdd에 접근하여 부트로더(hdd내 mbr에 위치한 코드!)가 메모리에 운영체제 커널 이미지를 로딩해주는 역할을 해야한다. > (unzip하여 커널이미지를 램 카피해줌.)
5. 운영체제의 커널 엔트리 포인트(entry point)로 이동, 명령어를 실행(boot trapping code : 정교하게 초기화!)한다.
6. shell 혹은 윈도우 매니저를 띄움.

인터럽트
Cpu 주변디바이스간 통신방법 중 하나

저장장치 계층구조
1. 용량
2. 접근시간에 따라

디바이스(i/o) 컨트롤러를 만드는게 운영체제의 매우 큰 포션을 차지하고있다.

메모리(명령어와 데이터 저장되어있음) – cpu : 명령어 실행 사이클 , 데이터 이동
장치 – cpu : 데이터(io 리퀘스트(from cpu) , 인터룹트(from io))
메모리 – 장치 : 데이터 이동(dma)

Smp(symmetric multiprocessing)
메모리에 연결되어있는 여러개의 cpu가 각각의 레지스터, 캐시를 갖는다.

Multi-core 디자인
Cpu를 여러개 갖는 건 비용이 비싸니, cpu내에 코어를 여러개 갖게하자!
코어 내에 독립적인 레지스터와 캐시가 존재하는건가? 아니면 공유하는건가?

Multi programming(매우 중요)
프로그램의 정의 : a set of instruction
프로세스의 정의 : 실행 중인 프로그램
과거에는 메모리에 한개의 프로그램만 적재해두고 실행했다.
여러 개의 프로그램을 메모리에 동시에 올려두고// 동시에 실행시켜주면 되지!
한 개 이상의 프로그램이 동시에 실행되는 것이 멀티프로그래밍.
동시에 여러 프로세스들이 메모리에 유지되고 있으면 cpu utilization이 증가한다.

Multitasking(=multiprocessing)
하나의 cpu가, 여러개의 job들을 자주자주 switch해주는 것.
(시간을 분할하여 switch = time sharing == concurrency)
유저는 여러 개의 job들과 동시에 interact할 수 있다.

그래서 cpu scheduling이 필요하다!(5장)
램에는 여러개의 프로그램이 적재되어있다. 어떤 프로세스를 다음에 실행할 것인지에 대한 문제.
목표는 cpu utilization을 높이는 것!

Operation mode
1. 유저모드
2. 커널모드

System call
유저모드에서 o/s에게 서비스를 요청하는 것.

가상화 기술
Cpu가 멀티프로세싱을 하네?
그럼 하드웨어 자원을 이용해서 여러개의 o/s를 돌리는 것도 가능하지 않을까?
구조 예시 : o/s – virtual machine manager(monitor) – h/w

운영체제는 프로그램 실행을 위한 환경을 제공해 줘야한다.
Ui, 프로그램 실행, io operation , 파일시스템제어, 통신, 에러탐지, 리소스 할당, 로깅, 보안

가장 중요한 개념!
프로세스, 쓰레드(프로세스의 하위개념)
Multiprocessing을 함에 있어서 synchronization(동기화)문제가 발생한다. 이 문제를 제대로 해결하지 못하면 deadlock이 발생한다.
프로그램의 정의 : a set of instruction'

유저 – os간의 interface방법
1. cli : shell과 같이 커맨드라인을 통한 방법
2. gui : 그래픽 ui
3. 터치스크린

그럼 프로그램- os간 interface는 어떻게 진행되는가?
바로 system call (=os API)이다!!!
API: application programming interface

일일히 시스템콜을 나열해서 작성하기 어려우니 라이브러리를 제공해줌!! (ex : printf)

Concurrency vs parerell

프로세스의 이해

프로세스 : 실행 중 프로그램, 운영체제에서는 작업의 단위를 프로세스라고 본다.

프로세스의 필요 자원
1. cpu타임
2. 메모리에 적재되어있어야함.
3. 파일, i/o디바이스(리소스)들을 관리할 수 있어야한다.

프로세스 메모리 레이아웃 (여러개의 section으로 나뉘어져있음)

1. text section(실행가능한 코드)<<<< logically 프로그램 시작번지(0번지)
2. data section (전역 변수 : 초기화 된 데이터, 초기화 되지 않은 데이터)
3. heap영역(runtime시 동적 할당 된 메모리 영역)
4. stack 영역 (함수호출 시 쌓이는 영역, 함수 파라미터, 리턴 주소, 지역변수)<<<밑에서 부터 쌓아 올림.
5. shell 커맨드에서 주는 파라미터값들 argc, arg**v**

프로세스는 5개의 상태를 갖는다(생명주기)
1. new : 생성된 상태
2. running : cpu를 점유해서 프로세스의 명령어를 실행하고 있는 상태
3. waiting : 1. Cpu time을 모두 썼지만 수행이 모두 끝나지 않았을때, 2. Io interrupt대기?
4. ready : io대기 하고 있다가 io가 완료되었을때 ready로 간다.
5. terminated : 모든 것을 다 끝냈을때. : 1. 마지막 문장을 실행하여 끝냈을때, 혹은 2. 시스템콜로 exit()호출시 - 이때는 os가 모든 자원들을 회수해줘야 한다.

Fork()라는 시스템콜을 통해 새로운 프로세스가 생성되면, new상태로 진입
프로세스 초기화 다 끝나면 ready큐로 진입
Scheduler dispatcher에 의해 ready큐에서 끌어와 running 상태로 진입
1. 만약 이 프로세스가 i/o처리나 event를 해야한다. : waiting으로 진입 > io끝나면 ready큐로 진입
2. 만일 cpu타임을 다 썼다. Interrupt시그널 받고 ready큐로 진입
3. 모두 수행하면 terminated된다. (return 호출 혹은 exit())

Pcb(process control block) : 프로세스 관리 구조체
-프로세스 상태
-program counter : 다음 수행해야할 명령어의 주소
-cpu register(instr reg)
이 레지스터 정보들을 context라고 부른다!!!!!
-Cpu스케쥴링 정보
-메모리관리 정보
-계정정보(어떤 유저가 생성했는지?
-i/o 상태 정보

Pcb는 프로세스마다 갖는다! 이 pcb를 os가 관리해줘야한다.

기본적 프로세스
실행중인 프로그램인데,
single thread(흐름, 운체에서 얘기하는 thread아님) of execution으로 수행된다.
그런데, 프로세스 내에서도 single thread of execution으로는 부족하다.
따라서, multiple thread(흐름) of execution으로 수행되게 하자!

Thread 개념의 등장. (챕터4)
스레드는 lightweight 프로세스이다.
Multithreading으로 동작하는 것이 장점이 많다.

Multiprogrammig의 목적
프로세스를 동시에 실행시키자.
Cpu utilization을 높이자.

Time sharing 의 목적
Cpu 코어가 여러 프로세스들을 자주자주 switch해주는 것.
타임스냅샷을 찍었을때 실제로는 프로세스 하나만 처리하는 것이지만,
이는 유저가 느끼기엔 동시에 처리되어지는 것 처럼 느껴짐.
(반응 속도측면에서 좋다.)= can interact

At the same time = simultaneously = concurrency = time sharing
<>
parerell

여러개의 프로세스를 time sharing하여 실행하기 위해서는 스케쥴링 queue가 필요하다.
1. ready queue
2. wait queue

Child를 fork()했을때 child는 new상태로 진입 후 ready큐에 들어감. Parent는 child process가 종료될 때까지 wait queue에서 대기 후 child가 terminated되면 ready큐로 들어감.

Context switch를 os가 해주어야한다.
프로세스의 context는 pcb로 나타내어진다.

Interrupt발생 혹은 시스템콜 발생 시 ,
시스템은 실행 프로세스의 현재 context를 저장해두었다가, 이후 resumed됐을 때, context를 restore해준다.

Context switch라는 것은 cpu코어를 다른 프로세스에게 switch해주는 task이다. (== 동시성 제어)
현재프로세스의 상태 저장, 그리고 다른 프로세스의 상태를 restore해주면 switching이 되는 것이다.

실행 관점에서의 parent process – child process 의 두가지 가능성
1. concurrently execution(부모가 child를 wait하지 않고 실행)
2. child가 terminated 될 때까지 parent가 wait(if(pid>0)으로 구분)

Address 관점에서의 parent process – child process의 두가지 가능성
1. 만일 child와 parent가 동일한 text를 수행하는 경우에는 pcb만 따로 생성하면 되는거다. Duplicate하는 경우
2. 새로운 프로그램을 갖는 경우

Zombie process : parent가 남아있지만 wait를 invoke하지 아니할때
Orphan process : parent가 죽어서 wait호출을 해주지 못할 경우

Wait()는 무슨일을 하길래?
Process종료 > 자원 반납 > scheduler를 불러 다음 실행 할 process를 선택!!
그럼 child process가 완료돼서 죽으면 scheduler를 누가 불러주냐? Parent process가 wait()를 반드시 콜 해줘야한다.

Fork()의 동작
Parent의 address(process memory layout== text section , data sec , heap , stack 등)을 그대로 복사
그럼 parent의 pcb도 그대로 복사해오는 것인가? : 맞다. 부모의 pcb도 통째로 물려받음.
Pcb는 어디 저장되는가? : 커널 내 보호된 메모리에 저장되며 stack으로 관리된다.

Parent가 fork하면 ? : child의 pid를 return
Child가 fork하면 ? : 성공했는지 여부를 알려주는 0 return

If(pid>0){
Wait()
}
여기서부터 child를 기다려주는 상태. 그 전까지는 child에 선행하여 parent가 실행하는 상태
Wait를 하지 않는다면, 통상적으로 parent가 먼저 실행되고 이 후에 child가 실행된다. (100% 보장은 못함. 하필이면 context switch가 일어나서 child가 먼저 실행 될 수도 있다.)

프로세스의 생성04 23분 25초 코드 실행 결과 따로 생각해보자.

Execlp() : fork()된 주소에 실행할 코드(파일)를 복사한다. 따라서 parent의 코드 중 execlp 다음의 state들은 아무런 의미가 없어진다.(child가 execlp 다음의 state를 실행을 하지 않는(못한다는) 다는 의미.)

Getpid()는 자신의 pid를 반환하는 함수

프로세스간 통신(IPC)

프로세스가 concurrently 실행되는 경우
1. independent process : independent는 다른 process와 share data가 없다라는 의미
2. cooperating process(여기서 문제 생김) : 다른 process와 share data가 존재하는 경우 또는 메시지를 주고 받을때

IPC의 두가지 모델
1. shared memory :
2. message passing : kernel이 중재하여 메시지를 패싱

Consumer-producer의 문제를 고려해보자!
: **cooperating process를 묘사하기 위함**

Producer는 정보를 생산하고 consumer는 정보를 소비한다.
예) 컴파일러가 어셈블리 코드를 생산하면, 어셈블러는 이를 소비한다.
예) 웹서버는 html파일을 생산하고, 브라우저는 이를 소비한다.

1) Shared memory solution
생산자프로세스와 소비자 프로세스가 concurrently 실행되는 것을 허용한다. (cpu를 time sharing한다는 의미)

버퍼(=shared memory)를 이용하게 하면 된다.
생산자는 버퍼를 채우고, 소비자는 버퍼를 비운다.
Bounded buf에서 buf가 가득차면 생산자프로세스는 wait, buf가 비워져있다면 소비자프로세스가 wait

프로세스의 메모리영역의 기본 철학 : 프로세스의 메모리 영역은 프로세스 자신만 접근해야한다. 다른 프로세스가 침범하게 만들면 절대 안됨. 따라서 통신하는 프로세스들이 접근할 수 있는 영역(= shared memory)을 메모리내에 할당하여 관리하자.(관리 = os가 담당)

생산자프로세스 : (in+1)%bufsize == out 이면 대기(버퍼가 가득 찼다는 의미)
소비자프로세스 : in == out 이면 비어있는 버퍼

Shared memory의 문제?
응용프로그래머가 명시적으로 구현해주어야한다.
그럼 os에게 맡겨보자!!! == message passing

Message passing
시스템콜 이용
1. send(msg)
2. receive(msg)

Communication link
1) direct : 각 프로세스가 보낼 프로세스를 알고 있는 상황. Send receive에 대해 명시적으로 이름을 붙여 comm// 제일 중요한 것 : 누구에게 보낼지 누구에게 받을지를 명시해주는 것
링크 자동생성, 프로세스 간 정확히 one link만 생성됨.

2)indirect : 중간의 메일박스, 포트가 존재 // send 와 receive콜에 A(메일박스)를 명시해줌.
두개의 프로세스가 shared mailbox를 갖는다.
이 링크는 두개 이상의 프로세스와 linking하는 것에 문제가 없다.
여러개의 different링크가 존재할 수 있다.

o/s는 메일박스를 생성, 삭제해주면 된다. 메일박스에 read, write(send, receive)해줄 수 있도록 관리

구현 상의 다양한 디자인 옵션(block(synch), non-block(asynch))
Blocking send : 데이터를 모두 send할 때까지 다른 일은 못함, block당함
Non-blocking send : os가 보낼 데이터들을 가지고 있고, 이를 보낼 책임은 o/s에게 전가. Sender 프로세스는 자신의 다른일을 계속 하고있음.

Blocking receive : 데이터를 모두 받을 때까지 block됨
Non-blocking receive : 리시버가 유효한 메시지를 받거나 null값을 받거나(리턴하고 자기의 일을 한다).

Non-blocking 은 인터럽트나 시그널방식으로 구현된다.

Block==synch인 이유?
Send()다음 문장으로 넘어갔다는 의미는 receiver가 모두 받았다는 것을 의미한다. 따라서 프로세스 간 동기화가 이루어졌다는 의미이므로!!

Asynch는 상대방이 모두 받았다는 것을 확신할 수 없다. 다만 효율적이고, 빠르다.

IPC 의 실제

1. shared memory : POSIX shared memory
2. message passing : pipes

Posix : 유닉스 운영체제의 표준화를 기술

보통의 fd 에 fopen 을 하게 되면 하드디스크의 메모리 영역을 잡는다. 하지만, posix shared memory 는 메모리에 파일을 생성한다.(== memory mapped : 빠른 접근 가능)

1. fd 에 shm_open 을 통해 이름, 권한부여 및 생성 > 이름(name 파라미터)으로 생산자와 소비자가 같은 shm_fd 를 갖게함.
2. ftruncate 를 통해 fd 사이즈 부여
3. mmap 을 이용하여 fd 를 메모리 mapped 시켜줌. (메모리에) 이를 포인터가 가리킴

생산자는 1-3 이후 sprintf 를 통해 포인터의 이동을 통해 write 진행

소비자는
1-3 모두 이후 printf 로 포인터가 가르키는 shm 를 읽어들임.
모두 읽은 후 shm_unlink(name) 을 통해 shm 을 제거해준다.

Shared memory 의 단점은 open read write 를 프로그래머가 모두 기술 해줘야하는 번거로움이 있다. 따라서 이번에는 message passing 방식을 알아보자.

Pipes(msg passing 방식)== 파일 디스크립터로 구현
두개의 프로세스가 커뮤니케이션하는 도구(conduit)처럼 사용된다.

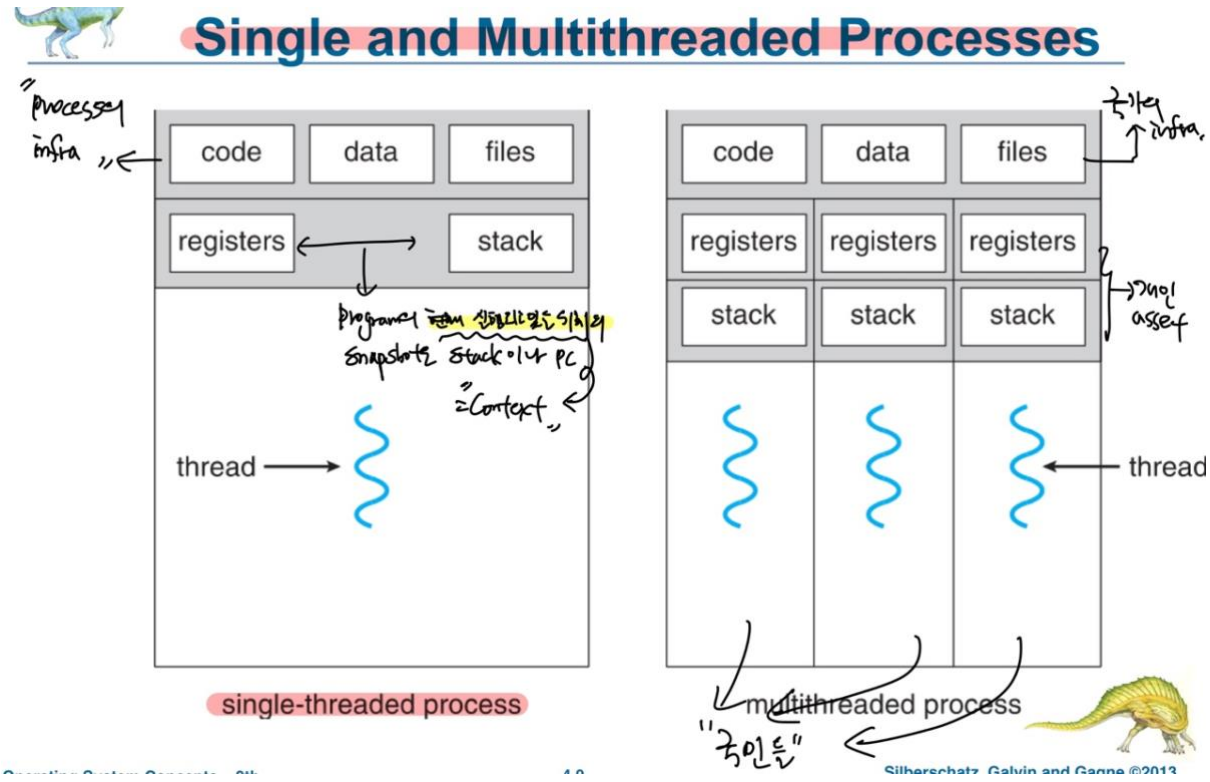
구현 시 네가지 이슈
1. uni-direct//bi-direct 구현
2. 2-way comm 이 가능한가? (half-duplex, full-duplex(왔다 갔다 할 수 있는가?))
3. comm process 간 relationship 이 구현의 편의상 parent-child 관계를 가져야한다.
4. 네트워크에서 동작할 수 있냐?파이프는 어렵다. (== 소켓)

Pipe 의 두가지 타입
1. ordinary pipe (:parent 가 pipe 를 생성하면 이는 반드시 child 와 comm 하는데 사용)
- 두개의 fd(각각은 uni-direct)를 이용해 두개의 파이프를 생성하여 two-way comm 이 가능하다.
2. named pipe (: 이름을 부여해서 Relationship 이 없어도 가능함) : mkfifo 를 통해 pipe object 를 생성한다. 양방향가능함.

RPC(remote procedure call)[자바에서는 RMI remote method in or COM]
Ipc(컴퓨터 내부)의 확장개념이다!!
"원격에 있는 함수를 호출"
Stub 을 통해 상대 컴퓨터의 함수, 스켈레톤을 알아낼 수 있다?
Marshal 을 통해 파라미터 등 데이터에 대한 형식을 맞게 직렬화? 해준다.
라이브러리 함수를 자신이 갖고있지 않고, 다른 컴퓨터가 갖고있다가 내 컴퓨터가 의뢰하면 networking 을 통해 함수 호출을 의뢰하는 방식 : 메모리 효율 측면에서 좋다, 다만 네트워킹상황이 불안정하면 담보할 수 없다.

Rpc 메커니즘 : double hand shaking

Thread 의 이해
멀티쓰레딩이 제공이 되면 Cpu 의 기본적인 점유 단위가 thread 가 된다.
Pid 내의 thread id (tid)가 cpu 를 점유하고 있다고 보면 된다. Pc 나 레지스터 set, 스택은 스레드별로 별도로 관리, 달라져야한다.
Pc 와 같은 레지스터셋 정보만 별도로 유지를 한다면 프로세스도 여러개의 쓰레드로 실행될 수 있다.
lightweight process 라고도 말한다.



멀티스레딩의 동기
예) 클라이언트-서버 소켓 통신일때 새로운 연결이 요청되면 기존 요청(통신)이 모두 처리 된 후 다음의 새로운 요청을 처리해야됨. 처리 될 때까지 대기해야되는 상황. 대신 스레드에게 이 일을 넘기고 요청을 non-blocking 으로 resume 하게 되면 스레드 생성 한도까지 새로운 요청을 계속 받고 처리할 수 있다.

멀티스레딩의 장점
1. responsiveness : 어떤 ui 를 처리를 할때 block 되어있을 필요없이 계속 실행 가능하게함
2. 리소스 sharing : 프로세스 간 메모리 공유는 overhead 가 큼. 다만 스레드들은 프로세스 내의 리소스를 별도의 overhead 없이 공유할 수 있다.
3. 경제적 : pcb 의 context switch 와 비교하여 thread switch 가 훨씬 적은 overhead 를 지닌다.
4. scalability(확장성) : 멀티프로세서 구조에서 각각의 스레드들을 붙여 병렬처리까지 가능해진다.

Jave thread
3 가지 방법
1. inheritance
2. implement
3. 람다방식 anonymous thread(클래스 생성을 안하는 방식)

스레드의 wait()는 Join()을 이용한다.
스레드의 종료는 interrupt()를 이용

멀티코어시스템에서의 멀티스레딩
: concurrency 가 훨씬 향상됨.

Single core : 스레드들이 시간별로 interleaved 된다.(시간을 기준으로 파티셔닝되어 끼워넣어진다는 의미)

Multiple-core : 각 코어들은 시분할을 통해 실행이 되면서 코어간에는 병렬적으로 처리됨.

멀티코어시스템에서의 challenge 들
1. identifying task : 병렬 처리 될 수 있는 task 들을 분리할 수 있어야 함.
2. balance :
3. data splitting : 데이터 또한 분할 되어야만 함
4. data dependency : task 의 수행이 올바르게 synch 되도록 보장되어야함
5. testing &debugging : 여러개의 스레드에서 어디서 문제가 발생했는지 찾기가 어려워짐.

암달의 법칙 : cpu 코어는 무조건 많으면 좋은가? Cpu 코어수가 아무리 많아져도 병렬 처리가 가능한 부분에 따라 speedup 에 한계가 있다.

멀티스레딩(pthread)

스레드의 두개의 타입
1. 유저 스레드
2. 커널 스레드

유저스레드 : 유저모드에서 스레딩
커널스레드 : os 가 직접 관리하는 스레드(코어에 직접 다이렉트로)

유저스레드와 커널스레드의 관계
1. Many to one model 감당이 어려움
2. One to one model 낭비인듯
3. Many to many model

스레드 라이브러리
스레드 생성과 관리를 위한 api 임

3 가지 메인 스레드 라이브러리
1. pthread
2. windows thread
3. java thread

Pthreads

Implicit threading
알아서 좀 해주라!의 개념
컴파일러가 알아서, 런타인 라이브러리가 해줘! 라는 개념.
1. thread pools : 스레드 풀을 만들어 풀에 스레드를 저장해두고 필요하면 꺼내서 쓴다.
2. openMP : 컴파일러 지시어를 통해 그 부분을 스레드생성해줌
3. grand central dispatch(맥)

OpenMp 의 #pragma omp parallel 다음의 블락은 하드웨어 스레드 개수만큼 병렬처리(스레드생성)된다.

프로세스 생성의 의미는 pcb 를 생성한다는 것이고 스레드를 생성한다는 의미는 라이브러리에 스레드 구조체를 생성한다는 의미이다.

Cpu 스케쥴링 (디스패쳐(스위치), 스케쥴러(선택))

Multiprogrammed os 에서는 필수 basis 가 된다.
Multiprogramming , processing , tasking 은 메모리에 여러개의 프로세스들을 올려 cpu 를 time slice 하여 선점하며 concurrent 하게 실행되는 것이다.

스케쥴링의 목적은 : cpu utilization 을 높이는 데 목적이있다.

스케쥴링 전제
Cpu burst 는 연산처리하는 구간, i/o burst 는 io 대기하는 구간. 이를 왔다갔다 하면서 프로세스가 실행된다. 이를 기준으로 cpu bound i/o bound 로 나눈다.

Cpu 스케쥴러
메모리에 있는 프로세스들로부터 다음 실행할(cpu 를 할당해줄) 프로세스를 선택함.

Fifo 레디큐 : 그냥 온 순서대로 처리함.
Priority 레디큐 : 여러가지 방법이 있다.

Preemtive (선점형)vs non-preemtive(비선점형)
쫓아낼수있는거 vs 못쫓아냄

Non : 프로세스가 자발적으로 나오기 전까지는 cpu 를 쓰도록 냅둔다.
Preem: 스케쥴러에 의해 프로세스가 쫓아 낼 수 있다. (or 선점할 수 있다.)

프로세스 상태

1. Running > waiting
2. running > ready
3. waiting > ready
4. terminate

1, 4 에는 non
2&3에 non 인지 preem인지 적용됨.

Dispatcher : context switch를 해주는 모듈, (running 상태를 쫓아내고 레디큐에서 끌어오는 모듈)
역할 1. Context switch
역할 2. Switching user mode
역할 3.

스케쥴링 기준
1. cpu utilization 이 목적
2. throughput 이 목적
3. turnaround time 이 목적 : 레디큐 도착에서 종료(completion)까지의 time
4. wating time(레디큐의 대기시간의 합)이 목적 : 레디큐에서 대기하고 있는 시간을 최소화 시키자. : 이를 해결하면 1,2,3 번도 해결되나봄.
5. response time 이 목적 : ui 같은. 게임같은

Cpu 스케쥴링 문제 : 레디큐 내의 프로세스 중 어떤것을 선택해서 cpu 코어를 할당해줄 것인가?

Fcfs (non-pre): first come first served == fifo q
- Convoy effect : short process(io bound) behind long process(cpu bound)

Sjf : shortest job first (srtf : shortest remaining time first)
- Fcfs 보다 Waiting time 을 감소시킬 수 있다.
- Cpu burst time 이 작은것부터 실행 시켜주자
- Waiting time 과 turnaround time 을 감소 시킬 수 있었다.
- Preem 할수도 있고 non 할수도있다.

Rr : round robin (시분할을 하여 정해진 시간만큼만 써라)

Priority based(다음 프로세스 선택시 우선순위 부여)
Mlq : multi level q
Mlfq : multi level feedback q

Sjf 는 구현이 너무 어려움. 왜냐? Next cpu burst time 을 알 수 있는 방법이 없으니까!!!
과거를 통해 근사적으로 구해보자(예측)

Srtf : shortest remaining time first : preemtive sjf

Round robin(: preemtive)방식
Preemptive fcfs 라고 보면됨. With time quantum(time slicing)
레디큐는 circular q 로 구현

Time quantum 을 얼마로 줄지가 중요하다. 너무 빈번하면 context switch 가 부담(디스패치 레이턴시 발생)으로 온다.
Q 를 무한대로 주면 fcfs 와 같다.

Priority – base 스케쥴링
Sjf 는 하기 어려움(예측해야하니까) 우선순위를 부여해서 next 를 결정해보자.

1. preemtive 방식
: starvation(infinite blocking) 문제가 발생함.
문제 해결 : starvation aging 으로 해결하자.

Multi level queue 스케쥴링
우선순위에 따라 큐를 따로 두자.

큐에 따라 starvation 이 발생할 수 있음.

Multi level feedback q : 퀀텀짧게 > 갈 수록 퀀텀 길게 > 마지막 큐는 fcfs 큐

(커널)스레드 스케쥴링

스케쥴러는 프로세스가 아니라 커널스레드를 대상으로 스케쥴링함.
Process contention scope : matching 할 스레드를 선택 하는 문제 (라이브러리에 의해)
System contention scope : 커널스케쥴링임. : 어떤 couple 들을 cpu 에 할당 해줄 것인가에 대한 문제

Real time os 에서의 스케쥴링
1. soft rt : 보장은 아니지만 critical process 가 우선 실행되어야함
2. hard rt : 데드라인을 반드시 지켜 실행되어야하는 것

우선순위역전 해결방법?

Edf :
Propotional share scheduling

프로세스 동기화

Concurrent access to shared memory 에서 항상 data inconsistancy 가 발생한다.
순서대로 실행되는 것을 보장 해줘야 이 문제를 해결할 수 있다.

Producer(p1) – consumer(p2`) problem 의 concurrent 실행 고려해보자.
Race condition (여러 스레드를 사용하면 어떤 순서로 액세스가 일어나느냐에 따라 결과가 달라짐) – 경쟁상황!

해결 방안 : 한개의 프로세스만 shared memory 를 수정하게 하자.
The critical section problem

Critical section : 프로세스가 접근, 수정하고, 이를 적어도 하나 이상의 프로세스와 공유한다면 이를 critical sec 이라고 한다.

하나의 프로세스가 이 sec 을 실행하고 있을때 다른 프로세스가 접근, 수정하지 못하게하자.

코드 영역 구분
Entry
Critical
Exit
Remainder

해결책(synchronization)의 3 가지 요구사항
1. 뮤텍스(상호 배제) : 단, 데드락을 야기 할 수 있음
2. progress(데드락 : 서로 눈치보는 상황을 회피)
3. bounded waiting(starvation) : 진입못하는 놈이 있을 수 있음 .

Single core 에서는 interrupt 를 막으면 방지할 수 있다.

해결책 두가지

Non- preem 은 문제 생기지 않음
Preem 은 문제 생김

Peterson solution(소프트웨어 해결책)
완벽하게 해결하지 못함.
1.자신의 깃발을 true 로 설정
2. Turn 을 양보

자신의 깃발과 turn 모두 true 면 critical 진입

3. 자신의 깃발을 false 로 변경

h/w 해결책
atomicity 보장: 원자성 (더이상 쪼갤 수 없는 operation 단위)
이 성질을 이용한
1. test and set
2. compare and swap

Atomic var : compare and swap 을 이용하여 atomic 한 변수를 만들 수 있다.

뮤텍스와 세마포어

뮤텍스 락 : 가장 간단한 synch tool (두개의 프로세스만 제어 가능)
세마포어: n 개의 프로세스 제어 가능
Monitor : 뮤텍스 세마포어의 단점을 극복

세마포어 s 변수 n 으로 초기화 : n 개의 lock key 라고 생각하면됨. S—로 key 갖고가고(wait) s++로 반납(signal) s 가 0 이면 모두 쓰고 있는 상황

S 가 1 이면 뮤텍스락과 같다. (= binary semaphore)

S 를 0 으로 초기화 하면?
P1 이 state 를 실행 한 후 signal 을 줌 > s++
P2 의 state 전에 wait 를 놓으면 p1 이 signal 준 후 에 순차 실행 할 수 있음.