1장 multiprogramming

system call 은 핵심중의 핵심. 아주 매우 중요 / os에 의해 제공되는 다양한 서비스들을 받기 위해서 여러분들이 만든 program이 사용할 수 있는 interface가 system call
프로그램이 운영체제의 서비스를 받고싶으면 system call의 형태로 서비스를 요청해야 한다.
대부분의 cpu kernel 와 user 모드
psw bit이 커널모드에서 작동중인지 유저모드에서 작동중인지 컨트롤하는 비트
cpu가 kernel모드에서 작동중일 때는 , 명령어 집합(instruction set) 에 있는 어떤 명령어도 수행 가능. 하드웨어의 모든 feature 수행 가능. 운영체제가 kernel 모드에서 일반적으로 수행. 하드웨어에 대한 완벽한 access 가능
user 프로그램은 user 모드에서 수행. 전체 명령어 집합 중 부분집합만 수행 허용. 하드웨어 feature 들도 일부분만 사용 가능 .
일반적으로 , i/o instruction이나 memory protection에 관한 instruction들은 보통 유저 모드에서 수행 허용되지 않음.

유저 프로그램이 운영체제로부터 서비스를 받기 위해서는 , 유저 프로그램은 시스템 콜을 해야한다. kernel 로 trap이 일어난다. Trap into the kernel (kernel)로 들어가는 것
Os invoke(operating system 수행 이 일어난다)
Trap instruction 이 kernel로 trap을 일어나게 하는 명령어라고 볼 수 있다.
trap instruction이 수행 -> user mode 에서 kernel 모드로 바뀌고 운영체제가 수행된다. (운영체제의 특정 주소부터 수행된다)
서비스가 완료, 제어가 다시 user program 으로 돌아간다 . (보통은 system call 한 다음 instruction)
trap instruction 수행하면 들어가기도 하지만 하드웨어적으로 발생하기도 한다. divide by 0 (제대로 수행할 수 없다 - > 특수한 상황 , trap 일어난다)
trap 은 trap instruction 수행했을 때 뿐만 아니라 예외적인 상황에서도 kernel로의 trap이 있다.

c program 으로부터 system call을 하기 위해서 보통은 library procedure call 을 한다. 실질적으로 system call 하는것이 machine dependent하기 때문에 assembly code 로 작성된 system call을 일으키는 부분을 call 해주는 (수행하는) library procedure을 call 하게 된다. library procedure 안에 assembly code 로 작성된 부분이 있다. 거기서 실질적으로 trap instruction 이런 부분이 assembly code로 작성되어 있다. 그것들이 수행되면서 실질적 system call이 일어난다.
system call을 하는 step
User mode에서 user program 을 수행하고 있으면 ,system service를 필요로 하는(예를 들어 파일로부터 데이터를 읽는)trap 또는 system call instruction 을 수행해서 trap instruction 수행 . Os 로 control 이 넘어감 . 모드가 user mode 에서 kernel mode. Os 가 수행.
Os가 system call을 한 process가(calling process) 무엇을 원하는지 파라미터를 보고 체크한다.
system call을 os가 수행해주고 제어를 처음에 user program에서 system call을 한 후 그 다음 부분으로 돌려준다.

system call은 procedure call과 유사한데 차이점은 system call 은 kernel 로 들어가는 상황이 발생하는데 procedure call은 그렇지 않다. (procedure call이나 fuction call 은 같은얘기. procedure call이나 fuction call은 호출한다고 해서 mode가 kernel mode로 바뀌지 않는다. 계속 user mode.

반면, system call을 하게되면 cpu모드가 user mode에서 kernel mode로 바뀌고 kernel의 코드가 수행된다.os가 수행된다.

system call이 일어나는 메커니즘
read system call을 실질적으로 해주는 library procedure을 call // count=read(fd,buffer,nbytes) read library processor 를 call
fd: file buffer : buffer에대한 포인터 nbytes : 읽을 bytes 수 read의 return값으로는 실제적으로 읽혀진 byte 수
read procedure call 을 하면 c 프로그램에서 일어나는 일(그림) ->메모리 전체를 나타냄
kernel space (os의 코드와 data)
큰 흰 부분은 user space = 메모리상에서 user program의 data,code,stack 등이 있는부분
c에서 read procedure call이 있는 line 작성. 컴파일되면 기계어로 바뀌는데 5개 정도의 instruction으로 바뀐다.
밑에서부터 수행된다. 나중에 sqw파일 에 들어가서 수행이 된다. user program calling read 가 바로 c 프로그램에서 read library procedure을 컴파일 했을 때 기계어 부분이다.
argument를 처음에 stack에 push한다. 그 다음에 실질적으로 read 라는 library procedure을 call하는게 온다. Call read (4번)
4번화살표로 가면 오른쪽 끝에 library procedure read라고 되어있다. 이 안에서 실질적으로 system call하는 코드가 들어있다.
5번 스텝에서 code ==system call number 이다. system call이 os에 여러개가 있기때문에 나타내는 number가 있다 . 그것이 system call number 라고 한다.
여기서는 간단하게 code라고 표현. system call number 즉 code를 register에 넣는다. (read 에 해당하는 system call number)
Trap to the kernel (kernel 로의 trap) 이 일어난다. kernel 로 들어간다. 6번 화살표로 kernel space로 들어간다.
이때 cpu 모드가 user mode에서 kernel mode로 바뀌게 되고 ,os 의 특정 주소부터 수행이 시작된다. trap instruction이 수행된다. 운영체제의 특정 주소부터 주소로 점프해 가서 수행이 되는데 특정 주소에 dispatch code가 기다리고 있다. dispatch 코드가 하는일은 dispatch 오른쪽에 테이블이 있는데(7번) 그 테이블은 read에 해당하는 system call number 을 레지스터에 넣어놨는데, 그걸 가지고 테이블의 인덱스로 사용한다. 테이블 엔트리 하나하나에는 각각의 system call을 처리해주는 루틴. system call handler .function.procedure 루틴이라고 볼수 있다. read system call handler 등등 이렇게 read 와 write system call을 처리하는 루틴 fuction procedure에 대한 주소들이 각 entry 에 들어있다. dispatch가 하는일은 레지스터에 저장된 system call number . Read 에 해당하는 system call number 가 20이라고 가정하면, 20을 가지고 테이블에 인덱스로 써서 20번째 엔트리를 딱 보면 그 안에 read system call을 처리해주는 루틴 즉 sys call handler 가 read system call handler의 주소가 있다. 그러면 그걸 가지고 그쪽으로 점프. read sys call handler 가 수행 (8번)

수행이 다 끝나면 return (9번) library procedure read에서 trap to the kernel 한 그 다음 명령어로 간다. <return to caller . Caller 는 user program>
거길로 돌아가게 된다 (10) 그것이 바로 call read 의 다음 instruction 단계. increment sp (stack point 를 증가시킨다) 그것은 아까 push push push를 1,2,3에서 했다. 그것을 stack에서 뽑아내는 것임.

unix에서는 각 system call에 대응해서 한개 정도의 library procedure 가 있다. read system call 에 대응하는 read library procedure 가 있다.
posix standard는 system call에 대응하는 library procedure ,한 100여개 에 대한 표준 제시 (프로시저의 이름, argument type 같은) system call 자체에 대한 표준이 아닌. System call을 일으키게 해주는 library procedure의 syntax 에 대한 표준이 posix 표준
os design 할 때 posix 에서 제공하는 100여개 procedure call을. 표준에 맞게 제공. 각각의 procedure에 대해서는 그것을 내부적으로 일반적으로 system call 을 하는 형태로 구현.필수적인 것은 아님 . system call을 안 써도 posix의 procedure call 구현가능하면 안써도 됨
posix에서 제시하는 procedure call 에 의해 제공되는 서비스는 보통 운영체제가 해야될 대부분의 일을 규정.
그러한 서비스에는
프로세스를 생성하고 종료
파일을 생성 삭제 ,read write , 디렉토리 관리, 입출력 수행
등이 포함된다.
주요 posix system call의 예 (엄밀하게 말하면 system call을 해주는 library procedure) [아래는 예시를 들어서 설명한 것]
pid = fork() child process를 생성해주는 system call 부모와 똑같은 child process
Fd=open
S=mkdir create a new directory
S= chdir change the working directory

프로세스란 , 수행중인 프로그램 . 실행중인 프로그램. 멀티프로그래밍의 컨셉. 메모리에 여러 개의 프로그램 넣어놓고 이 프로그램 수행하다 다른 프로그램 수행하는것 - 멀티프로그래밍. cpu가 process와 process switch 하면서 수행. 그림 a를 보면 메모리에 프로그램 4개가 올라와있다. cpu 하나로 가정. cpu 하나가 a를 수행했다가 b를 수행했다가 c를 수행했다가 d 수행 .이때 그림 a에서 프로그램 counter는 cpu가 하나이니까 1개이다. cpu 속에 있는 현재 수행중인 instruction의 주소를 담고 있는것이. program counter . Program counter는 하나인데 a의 instruction을 수행하다가 a의 instruction을 가리킴 b를 수행할때는 프로그램 카운터 값을 b에 있는 instruction 쪽으로 바꿔야함. 그림 b 보면 개념적으로는 그렇지 않지만 a,b,c,d 가 고유의 program counter를 가지고 있는 컨셉. 하드웨어적으로는 하나의 프로그램 카운터이지만 a가 하드웨어적으로 하나의 프로그램 카운터를 자기가 이용 . 그러다가 b로 제어가 넘어가면 a가 쓰고있던 프로그램 카운터 값을 어딘가에 저장 (메모리) 그리고 b 수행. 그러다가 a로 돌아오면 메모리에 저장해둔 a의 program counter 값을 다시 program counter에 load(적재)를 하고, 수행 . 그림 c를 보면 cpu는 하나이고 멀티프로그래밍 하는 상황을 봤을 때 프로세스 4개가 있다. 시간이 오른쪽으로 흘러감에 따라서 a가 먼저 수행된다 . 그러다가 b로 제어 넘어가 b 수행 .cpu는 하나이기 때문에 cpu는 동시에 a,b 수행할 수 없는 상황. 시간을 쪼개서 하는것 . 어느 한 순간에는 하나밖에 수행이 안된다. a 수행하다 b 수행하다 c .. 시간이 흘러가는 쪽으로 본다. 그다음에 다시 a,b,c,d를 수행한다.cpu가 하나인 이상은 한번에 하나의 프로그램 밖에 수행할 수 없다. 이런식으로 멀티프로그래밍이 가능하다. a를 수행하다가 b로 수행이 넘어갔을 때 , 멀티프로그래밍이 한 프로그램에서 다른프로그램 수행할 때 보통 원래 프로그램이 입출력을 하는 동안에 시간이 걸리니까 다른 프로그램을 수행한다고 했다. (Cpu)에서. 그런 경우에도 한 프로그램 수행하다가 다른 프로그램 수행할 수 있지만 , 예를 들어 입출력과는 상관없이 여러 개의 프로그램을 cpu가 하나이니까 그냥 골고루 나눠서 쓰게 한다 . 시간으로 쪼개서. 예를 들어 a프로그램으 어느 시간동안 수행하다가 시간이 다 지나면 제어를 b프로그램으로 넘겨서 b를 수행한다. a를 수행하다가 b를 수행한다. 입출력이 발생할 때도 a를 수행하다 b를 수행할 수 있지만. 입출력 발생 안하는 경우에도 a를 수행하다가 b를 수행할 수 있다 . a를 수행하는 cpu의 시간 할당량이 다 찼을 경우에. 시간 할당량을 프로그램마다 조금씩 줘서 할당량 만큼씩만 cpu를 사용하게 해준다. 멀티프로그래밍에서 한 프로세스에서 다른 프로세스로 수행이 넘어가는 상황이 된다. 프로세스= 실행중인 프로그램

구체적으로, implementation 측면에서 봤을 때 process는 실행중인 프로그램이기 때문에 메모리 상에 자기의 address. 자기의 process image를 가지고 있다. core image라고 함. 메모리, 주소 공간에서 자기만의 공간을 차지. 각 프로세스는 core image를 가지고 있다(==address space 주소 공간을 가지고 있다.)
process는 process table 상에 하나의 entry를 차지하고 있다. process table은 운영체제가 관리하는 data structure. Table 이다.여기의 한 entry를 차지하게 된다. entry에는 해당 프로세스가 나중에 수행될 때를 대비해서 현재의 상태 를 저장해둔다. A프로세스를 수행하다 b프로세스를 수행하는 상황이 발생하면. a 프로세스의 현재 상태(레지스터 값들이나 그런것) 이런 것들을 a프로세스에 응하는 process table entry에 저장해놓고 b 프로세스를 수행한다. 나중에 a프로세스를 수행하게 될 때가 오면 process table 상의 entry 값을 loading 한다(레지스터 값들) 그 후 실행을 계속 한다. 그래서 결론적으로는 프로세스는 core image라고 불리는 address space(자기만의 주소 공간) 주소 공간에는 code 부분(==text부분) , data 부분, stack 부분이 사실은 있다. Process table 상의 entry 를 차지한다.

process state => 굉장히 중요합니다.. +그림 이해
state diagram 이 특히 중요 . process 가 거치는 state를 나타낸다. 상태간의 전이도 보여준다 (transition)
Running , ready, blocked 상태를 가질 수 있다. 어떨 때 running 상태가 되는가? == 현재 cpu를 사용하고 있을 때 . 지금 cpu를 사용해서 수행되고 있는 놈. 지금 cpu를 잡고 사용하고 있는 놈. ready상태로 가는 경우. runnable == 실행 가능한 상태. 보통 ready 라고 부름. cpu를 잡고 수행되고 있지는 않지만 언제든지 자기 차례가 되면 수행할 수 있는 process. cpu를 여러개의 프로세스에 돌아가면서 사용하게 되는 경우, 그 때 a프로세스가 running 상태에 있는데 다른 프로세스에도 기회를 줘야 되니까 a프로세스에게 일정한 시간만 할당을 해주고 (10 ms)할당된 시간이 다 지나면 running상태의 a프로세스를 ready 상태로 끌어내리고 그다음에 ready 상태에서 대기하고 있던 프로세스 중에 한놈을 running 상태로 올린다. (2, 3 번 transition) 이렇게 함으로서 cpu를 여러 프로세스가 돌아가면서 사용.
running 상태에 있는 프로세스가 ready 상태로 자기에게 주어진 시간 할당량을 다 써버렸기 때문에 ready 상태로 내려오는 경우에는 보통 ready 상태의 프로세스들이 줄을 서있다. (queue) 끝에다가 보통 세운다. ready 상태에 줄서있는 queue에 있는 프로세스 들 중에. 맨 앞에 있는 애를 running 상태로 보내는 경우가 많다 .
Blocked 상태는 running 상태에 있는 프로세스가 i/o 요구같은 것을 했을때, 파일을 read 한다던가 키보드로부터 input을 읽는다던가 입출력 요구를 했을때 입출력을 수행하는 동안에 cpu를 사용하지 못하니까 (입출력을 기다려야 되니까) 현재 수행중인 프로세스는 (현재 입출력을 요구했던 프로세스는)blocked 상태로 보내버린다. 그러면 cpu의 running 자리가 빈다. 그러면 ready 상태에 있는 프로세스 중에 맨 앞에 있는 애를 running 상태로 보낸다. ready 상태에 있는 프로세스들 끼리 돌아가면서 수행한다. 시간이 흐르면 요구했던 입출력이 완성이 되었을 때 , blocked 상태로 갔던 프로세스가 원하는 파일을 읽었으니까 처리를 계속할 수 있다. blocked를 ready 로 보내진다. ready의 줄 맨 뒷줄에 설 수 있다. 언젠가 자기 차례가 오면 running 으로 간다. blocked 에서 ready로 갈때(4번) input becomes available. 입력을 요구했는데 입력이 들어왔다. blocked 에서 ready로 가서 나중에 running으로 자기 차례가 되면 간다. running 에서 blocked로 가는 (1번) 입출력을 요구했을 때 blocked 로 가게된다. state diagram + transition 기억. 4번이 입력이 왼료되었을 때. disk부터 파일을 읽는 . 키보드로부터 char을 타입 완료되었을때. Input이 사용가능하게 되어지면 blocked를 풀어준다. ready 상태로 .

process는 자신만의 core image == address space를(주소공간)을 가진다. 코어 이미지는 3 segment로 구성. text , data ,stack
Text 는 code 부분 data 는 data영역 . stack 이 있다. data는 static variable(정적 변수)들이 저장되는 부분이다. stack은 automatic varible들이 저장되는 영역이다. (function 안에서 variable 선언 == automatic varible) function 밖에서 선언하면 static. =>data segment에 저장된다. stack은 이제 보면 주소를 아래쪽이 낮은 주소 위에쪽이 높은 주소로 해뒀다. data segment와 stack segment는 서로를 향해서 자라나는 . Stack에 push가일어나면 stack은 밑으로 커지고 data segment가 커지면 위로 커진다. 사이에는 gap이 있다. stack 은 push하면 점점 자라나고 data segment는 malloc() calloc() 동적으로 메모리 할당 call했을 때 그 부분이 자라난다. 동적인 데이터를 할당해주기 위해서. 자라나고 있는 부분을 heap이라고 부른다. Heap의 data segment를 잡고 delimeter를 늘려가지고 heap부분을 malloc()으로 call 한 variable , memory 할당 이런것을 사용한다.

process hierarchies
Parent 프로세스가 child 프로세스를 보통 만들게 된다. child 프로세스도 자기의 child 프로세스를 만들 수 있다. 그림을 보면 a프로세스가 child 두개 b,c를 create 한 것이고 b가 자기의 child d,e,f 를 create 한 것이다 . process tree 라고 한다. Unix os에서는 프로세스와 자식들을 다 합쳐서 process group이라고 부른다. 보통은 해당 윈도우 터미널 띠웠을 때 생성된 프로세스를 보통 한 process group에속하게 된다 . 그에 반해 windows os는 process hierarchies 개념이 존재하지 않는데 모든 process가 equal한 개념으로 존재한다.

protection
Process를 한번 시작하면 해당 프로세스를 시작한 사용자(user)의 user id를 부여받게 된다. user id는 시스템을 사용하도록 허락된 사람은 user id를 부여받게 된다. system admin으로 부터. 그 사용자가 자신의 user id를 가지게 된것이고 그러면 그 사용자가 이제 시스템에서 로그인해서 process 시작을하면 process는 자신을 시작시킨 사용자의 user id를 갖게 된다.user들은 그룹의 멤버가 될 수 있다. 각 그룹을 나타내는 id는 group id 라는 뜻으로 gid 라고 부른다.

process table
Process table은 운영체제가 관리하는 data structure 중의 하나. 프로세스 마다 한 entry 차지하게 된다. 그때 그것을 process control block 이라고도 부른다.
process table에는 해당 process의 state. State는 running ready blocked 가 있다. program counter 값. stack pointer 값 . memory allocation 정보. 오픈 파일에 대한 정보 acounting scheduling information . 그리고 그 밖의 정보들 . 즉 프로세스가 자기가 cpu를 쓰고있다가 다른 프로세스에게 cpu를 넘겨주는 상황에서 나중에 다시 자기 차례로 돌아왔을 때 자기 상태가 save되어 있어야 그걸 가지고 다시 시작 가능. save를 해야할 정보가 여기 저장됨. process가 running 상태에 있다가 ready 나 blocked 로 빠져 버리는 상황에서 저장해야 될 정보 가 들어감. (상태 , program counter 값 ~ 오픈 파일 관한 정보 다 일단 저장) 그리고 다른 프로세스 실행. 다시 자기 차례가 되면 저장했던 정보 process entry table로부터 읽어가지고 loading을 하는. 표로 process entry table 정보가 나와있다.
Process management 정보에는 레지스터 값 . (나중에 다시 수행할 때 loading) program status word == bit 하나가 kernel 모드냐 user 모드냐 나타내는 bit가 하나. stack pointer 현재 스택이 어디까지 차있는지. Process state ==running ready blocked

중간에 memory management쪽은 해당 process의 core image. Address space 에 관한 정보가 있다. 맨 오른쪽 file management 여기에 root directory. File descriptor는 이 프로세스가 오픈한 파일에 대한 정보. user id는 프로세스를 실행시킨 사용자의 id.

Process creation
프로세스가 언제 created 되는지에 대한 얘기. 일반적으로 unix system. 에서는 새로운 프로세스가 기존에 존재하는 프로세스가 process creation system call을 수행 함으로써만 새로운 프로세스가 생성될 수 있다. unix에서 fork() 이다. process creation 하는 주요 상황 (4가지 정도)
프로세스가 생성되는 메카니즘 자체는 기존의 프로세스가 process creation system call을 해야지만 새로운 프로세스가 생성. 그게 process를 create하는 메커니즘. 어떤 상황에서 process가 create되냐 하면
1. system initialization
daemons - activity를 handle해주는 background에서 도는 프로세스. email , web page, news, printing 등등 처리해주는 프로세스. 시스템 시작할때 보통 시작
2. 응용 프로그램을 수행하다가 fork system call이 있어서 그것을 수행했을 때 새로운 process create.
3. unix shell에서 새로운 process create해달라고 user가 request할 때 .icon 클릭할때 gui interface에서 . windows에서는 더블클릭 -> 커맨드 내리는것과 같은 상황 .
4. batch job 을 시작할 때 process가 만들어진다. Batch 는 아주 예전 system
결과적으로 이 4가지 상황에서도 process가 실제로 create되는 메카니즘은 fork system call 이 다 관여한다.

Process termination
1.normal exirt(voluntary) 정상적으로 수행하고 (ex.c program) return을 만난다거나. 정상 종료 시 (끝까지 가서)
2.error exit(voluntary) 잘못된 입력. 종료하게 만드는 그런 경우.
3. fatal error(involuntary) 프로그램 버그. 존재하지 않는 메모리 쪽을 access . Divide by 0 상황이나 illegal instruction. 비자발적 종료
4. kill 다른 프로세스에게 kill 당하는 경우

process management를 위한 system call
Process creation / termination 하는 system call. 프로그램을 다른 걸로 overlay . (Core image를 다른걸로 overlay) 덮어씌우는것. child process 가 terminate 되기를 기다리는 system. 동적으로 Memory allocation, request 하는 system call

Shell . Unix에서 shell prompt. 명령어 처리. shell 자체의 source code shell은 크게 보면 while 루프 돈다. 명령 받아들이고 처리하고 명령 받아들이고 실행하고 반복하는것이 shell . While(true)에서 무한루프 돌면서 type_prompt(); // shell prompt 출력하고 read_command(command,parameters)사용자가 입력한 command를 읽고 . if(fork()!=0) . 지금은 shell process가 수행되고 있는 상태. shell code이니까. shell process가 수행되면서 사용자가 입력한 것을 받아들임. read command로. 사용자가 입력한 command는 read command의 argument 보면 (command,paramters) command와 파라미터들이 argument로 들어감. 파라미터 없이 hello라고 치면 , shell에 read command가 수행되면서 첫번째 argument에 hello라는 string이 들어간다. if(!fork()=0)fork system call을 하면 fork system call이 바로 child process. 현재의 process 로부터 child process 하나 만들어준다. 현재 프로세스는 parent. 새롭게 만들어진 프로세스는 child process 이다. fork system call을 shell process 가 했다. Fork system call이 운영체제에서 수행이 되면서 다시 child process 가 만들어진다. child process를 만들어주긴 한데 parent 와 똑같은 걸 만들어준다.clone을 만드는것. fork system call이 수행되고 있는 중간에 child process 가 생기고 그러면 shellprocess 가 parent process고 새로 생긴 child process 도 shell process 가 된다. 둘다 fork를 call 했다고 생각하는 상황. fork로부터 return. Parent 인 shell process 도 fork 로부터 return 하고 child. Process 도 fork 로부터 return. Return 했을 때 . parent process 가 fork로부터 return하면 return 값이 지금 새롭게 clone으로 만들어진 child process의 (pid =process id) 그것을 return 값으로 받는다. child process 인 경우에는 fork로부터 return 할때 return 값을 0을 받는다. 따라서 if fork에서 불러가지고 처리할때 (운영체제에서) 그러면서 child process가 부모랑 똑같은게 생겨난다. 두개의 프로세스가 return. 부모의 경우에는 child process 의 pid. 256이라고 치면 fork는 return 값이 256이니까 조건문이 0이 아님. if 속으로 들어간다. waitpid를 call. 자식인 경우에는 else로 들어간다 .fork의 return 값이 0이니까. execve를 부른다. exec system call의 variation 중 하나인데 exec system call은 현재 process의 core image를 다른 걸로 덮어쓰는것. overwrite. 그 때 execve(command ,parameters,0)인데 예시로 hello. Hello sqw파일 내용 가지고 exec을 수행하니까 shell process의 image를 hello의 image로 덮어쓴다. exec system call로부터 return 했을때는 이 프로세스는 shell의 clone인 child process가 아니고 image가 hello로 바뀐 그런 child process. 이 프로세스가. hello 라는것이 수행이 끝나면 . 프로세스가 종료를 하게되면 child process인 hello process가. 종료를 하게되면 parent process가 waitpid를 call하고 있었는데 waitpid는 맷 첫번째 argument로 준 것은 내가 기다려야 할 process의 id.waitpid(300) 이면 pid가 300인 process를 종료할때까지 기다림. -1을 넣으면 , 나의 child process가 끝날 때까지 기다림. Hello가 끝나면 waitpid로부터 return. if문을 빠져나가고 큰 while 루프의 끝으로 가서 다시 돌아가게 된다. 그래서 다시 type_prompt(); shell prompt를 출력한다.사용자가 입력을 하기를 기다린다. 이렇게 반복을 한다. shell code의 뼈대이다.

brk system call은 data segment를 키워주는 것이다. (heap)부분을 만들어주는것. 동적 메모리 할당을 할 때 brk system call이 보통 할당. data segment의 끝나는 그 부분에서 물려준다. 끝나는게 주소 1000번지면 2000번지로 한다든가 해서 data segment를 자라나게 한다. c에서 malloc, calloc dynamic memory allocation. 그게 call되었을때 brk system call으로 segment 늘림으로 heap 부분 만들고 program에 할당된다.

process state diagram 중요. 어떤 process가 running 상태에 있다가 read system call 하면 blocked 상태로 간다. 입력을 요구하게 되면 blocked 상태로 간다. 운영체제 system call을 운영체제가 처리하면서 system call handler가 read system call을 처리하는데 그러면서 결과적으로 운영체제는 그 프로세스를 blocked상태로 보낸다. 운영체제의 read system call은 결국은 disk로부터 파일을 읽어야하니까 disk에게 read 요청. 어떤 특정 디스크 블록을 읽어라. read system call을 하면 blocked 상태로 바뀌고 , 운영체제는 disk(하드웨어)한테 읽어달라고 요청함. 결과적으로 running이 있다. running 은 물론 ready 에 있는 애들 중 하나가 running으로. os scheduler에 의해 ready 에 있는 애들 중 하나가 running으로 가서 빈자리 메꾼다. running에 있다가 시간 할당이 다 되면 ready 로 끌어내리고 ready 중에 한 얘를 running으로 보낼수도 있다. 이런식으로 running의 자리가 계속 바뀔수 있다. 그러다가 언젠가는 blocked로 간 프로세스가 처음에 요구했던 disk에서 파일 read 하는게 완료되면 , disk controller 하드웨어는 결과적으로 cpu에게 interrupt를 건다. 자기의 device number 도 알려준다. 그러면 device number가 interrupt number라고 할 수 있다. cpu는 running으로 어떤 process 수행중일때 그러다가 갑자기 하드웨어 적으로 disk controller쪽으로부터 interrupt signal, number 가 온다. interrupt number 4번이라 치면. 지금 요청받았던 파일의 그 부분을 읽었다. 이런 것을 의미. Os는 cpu를 이용해서 어떤 프로세스를 수행 중인 상황이었을것이고, 그런데 interrupt가 걸리게 되면 하드웨어적으로 걸린것이기 때문에 신호가 가서 일단은 현재 수행하고 있는 프로세스가 중단된다.그리고 8개의 step이 수행된다.

Interrupt 발생
첫번째. 하드웨어적으로 cpu가 현재 process의 수행을 중단하고 상태를 저장한다. program counter, 레지스터 값들. 다 저장한다. 어디에 저장하면 현재 process의 current stack
두번째. cpu가 하드웨어적으로 interrupt vector로부터 program counter를 loading한다. interrupt가 disk controller로부터 cpu쪽으로 걸릴 때 device number ==interrupt number가 같이 온다. 그래서 5번이라고 치면 5번을 cpu는 여러가지 device가 cpu쪽으로 interrupt걸 수 있는데 disk인 경우에는 자기를 나타내는 번호가 5번인것이다. disk device인 경우에는 5번을 cpu에게 보낸 상태(interrupt 걸면서) cpu는 5번이라는 interrupt number를 interrupt vector table이 메모리에 있는데 interrupt vector 는 interrupt handler 루틴의 주소. 각 device 마다 device 에 의해 interrupt 가 걸렸을때 그것을 처리해주는 routine 그것을 interrupt service routine. Interrupt handler routine이라고 부른다. 주소를 interrupt vector라고 하며 그 벡터들이 모여있는 것이 interrupt vector table. interrupt number를 interrupt vector table에 대한 index로 쓴다. 5번이 왔다 하면 (disk controller로부터)5번을 interrupt vector table에 인덱싱하면 거기에 interrupt vector가 있다.(disk interrupt를 service.handle 해주는 루틴의 주소 가 있다) 그 주소로 가서 수행하면 된다. 그 주소를 cpu program counter 에 load 하는것임.
3. 벡터가 나타내는 주소로 갔을것이다. 시작주소는 보통 asssembly code로 작성. (나중에 기계어로 번역) 3,4번이 어셈블리 코드로. 3번에는 레지스터 값을 save한다.아까 1번 단계에서 하드웨어적으로 cpu가 current stack에 저장 레지스터 값들을. 그 값을 읽어서 os가 관리하는 process table에다가 저장을 한다. (entry) 어느 entry냐면 수행하다가 중단된 process. Interrupt 맞아서. process table 속에 entry에 레지스터 값을 저장을 한다. Current stack 에 저장되어있던 레지스터 값을 읽어서 pop() 한다. 그다음에 4번 단계에서는 temporary stack을 set up 한다. interrupt 서비스 루틴이 수행되는거니까 수행되면서 사용할 수 있는 stack 을 set up 한다. stack pointer 값에다가 temporary stack의 시작 주소를 setting 해준다는 것 기존의 user process 의 stack 이용하는게 아니라 os 의 stack을 이용하는것. os 의 스택을 setup 하는것. 5번 본격적으로 interrupt service 루틴이 실행. 예를 들어. disk로부터 read가 끝나서 interrupt가 걸린것이었다면 , 5번에서는 read한 값을 kernel buffer로 옮긴다. 그것은 결과적으로 user program의 버퍼로 결과적으로 전달된다.

이렇게 되면, 놓치지 말아야 하는것이..
read system call 한 process가 있었다. 걔가 running 에서 blocked로 갔다. 그상태에서 ready에 있는 애를 running으로 놓고 스케줄러가 ready에 있는 애들 중에 하나를 선택하고, 그다음에 그게 결과적으로 running 으로 가서 수행. 걔가 또 ready로 물러나고 따른에가 ready에서 running으로 갔을 수 있다 시간이 흐르다가. 그러다가 언젠가는 disk로부터 interrupt 걸린것. 스텝이 5까지 옴. blocked로 간 process는 read 가(io가) 완료. 버퍼로 값이 들어옴. 더이상 blocked에 있지 않고 os는 ready의 큐의 줄에다가 세움. 4번 transition(process state 상태 ppt). -> interrupt ppt (15page 의 5번 정도에서 일어남.)interrupt service 해주고 blocked 에서 ready 로 보내준다.5번 쯤. 6번 스텝에서는 os scheduler가 누구를 수행할까를 선택. 일반적으로는 아까 처음에 수행하고 있다가 interrupt를 맞은 얘. 걔를 선택하는 경우가 많다. 걔는 상태가 save되었으니까 1,2,3번에 의해서. 결과적으로 다시 수행 가능. 6번에서 os의 스케줄러는 누구를 수행할까 선택하는데 보통은 interrupt를 맞았던 프로세스를 수행하는 경우가 많다. 7번은 5번같은 경우가 c로 짜여있기 때문에 다시 어셈블리 코드로 돌아와서 8번 스텝에 어셈블리 프로시저가 새로운 프로세스를 스타트한다. 6번에서 os 스케줄러가 프로세스를 선택(running)으로 둘 프로세스. 결과적으로 8번에서 선택된 프로세스를 set up 후 start. 그것을 보통 어셈블리 코드로 짠다, 왜냐면 새로운 프로세스 스타트하기 위해서는 process table 에 있는 값들을 다 읽어서 레지스터에 loading 해야하기 때문에. memory image. Text, data, stack에 대한 포인터 세팅 -> 어셈블리 언어로. 그렇게 하면 process 가 수행하게 된다.

ipc interprocess communication
프로세스 간의 통신.프로세스 간의 communicate . 이슈가 3가지정도있는데 하나는 하나의 프로세스가 다른 프로세스에게 어떻게 information을 전달할것인가.
또 하나는 두 개 이상의 프로세스가 중요한 activity에 관여하고 있을 때 어떻게 서로의 방해를 하지 않도록 할것인가.
또 서로 의존적인 관계가 존재할 때 어떻게 순서를 제대로 맞춰줄 것인가. 예를 들어 프로세스 a가 데이터를 생산하고 b가 그 데이터를 print하는 역할을 한다면 b는 프린트를 하기 전에 뭔가 a가 데이터 생산하기 까지 기다려야한다.진행하지 말고.. 이런 sequence 순서를 말한다.

race conditions
굉장이 중요한 컨셉 . 매우 중요하다 번역하자면 경쟁 조건 정도.
Race condition은 두개 이상의 프로세스들이 shared data(공유 데이터를) read나write를 할때(동시에) 최종 결과가 어느 프로세스가 언제 수행되었느냐에 따라서 .
Ppt 번역 -> 둘 이상의 프로세스가 동시에 일부 공유 데이터를 읽거나 쓰는 상황. 최종 결과는 정확히 언제 실행하는지에 따라 다르다. 예를 들면 print a daemon. Daemon은 background process . System에 있는 .printing을 해주는 서비스인데 user process들이 파일을 출력해 달라고 요청을 하면 print a daemon process가 그 요청받은 파일을 contents를 disk에서 읽어가지고 출력을 해준다던가 한다. 그런 상황을 생각. 메모리 같은데 spooler directory 가 있다고 가정. 출력해야할 파일 이름들이 쭉 줄서있다. user process들은 여기에다가 자기가 출력할 파일이름을 쓴다. 그리고 print a daemon process 는 여기에 쓰인 파일을 하나하나 이름을 읽어서 content를 disk에서 파일을 읽어서 print로 출력한다고 생각. 좀더 구체적으로는 출력하고자 하는 user process는 일다는 out 과 in이라는 shared data가 있다. 공유 데이터인데 이것을 읽는다. User process 입장에서는 in의 값을 읽는다. in의 값을 읽는다 이경우에는 7. 7번 슬롯 거기에다가 자기가 출력하고 싶은파일의 이름을 쓰고 in의 값 증가 -> 8로. in은 출력하고 싶은 파일의 이름을 가리키는 인덱스. Out은 print a daemon process가 사용. 현재 출력해야 될 파일의 이름을 가리킨다. 4니까 abc이다.

print a daemon process가 out의 값을 읽어서 4이니까 4번 슬롯에 있는 abc의 content 를 disk 에서 읽어서 모니터로 출력. out 1 증가 4->5 . 이때 , . race condition 이 발생 가능. process a가 출력하고 싶은 파일이 있다 a.txt . 공유 데이터인 in의 값을 읽어서 자기의 local variable에 넣었다고 가정. Local variable in의 값에 7이 들어감. 그다음에 공교롭게도 context switch가 발생. context switch라는 것은 수행중인 process를 다른 process로 바꾸는것. process a가 수행중이었는데 (running) context switch 가 발생해서 다른 프로세스가 running 상태로 온다. 프로세스 a는 running 상태에서 일반적을 ready 상태로 쫓겨남 . 프로세스 a는 in의 값을 읽어서 local variable에 넣고 거기에 context switch 당해서 ready가 되었다고 가정. process b가 자기 차례라고 가정. process b도 b.txt 출력하고 싶다. 얘도 이제 in의 값을 읽어서 local variable 에 넣는다. b의 local variable에 7이 들어간다. b는 context switch 당하지 않고 수행 계속 일어난다고 치면 , (race condition 보여주기 위한 시나리오) b는 shared data인 in의 값을 local variable에 넣고 , local variable이 갖고 있는 값의 슬롯. spooler directory 의 7번 슬롯 . 거기에다가 자기가 출력하고 싶은 파일의 이름 b.txt를 썼다. local variable 7 ->8 으로 증가. 그것을 shared data 인 in에다가 쓴다. in의 값은 8로 업데이트. 나중에 context switch 발생하면 b->c 로 running이 넘어갈 수 도 있다. 언젠가 process a의 차례가 오면 a는 이제 자기가 하던 다음부터 수행. 어디까지 했냐면 in의 값을 읽어서 자기의 local variable 에 저장까지 해둠. local variable이 7 인 상태에서 switch 당함. 거기부터 시작. a는 local variable 이 가리키는 것이 7이니까 spooler directory의 7번 슬롯에 자기가 출력하고 싶은 파일 이름을 쓴다. a.txt .. (문제 발생)b가 7번 슬롯에 b.txt 라는 파일 이름을 썼는데 overwrite 해버림. 7번 슬롯에서 b가 쓴 데이터는 a.txt 에 의해 overwrite. a는 수행을 계속한다고 가정하면 local variable 의 값은 7->8이 된다. 그다음에 8의 값을 shared data인 in에다가 쓴다. in 은 아까 process b가 8로 업데이트 해둠. 또다시 8로 업데이트.그럼 이제 process b 입장에서는 자기가 썼던 b.txt가 출력되기를 기다린다.(process b를 수행했던 user b) printer 앞에 가서 b.txt 기다리고 있다 . 아무리 기다려도 영원히 나오지 않는다. A.txt 만 나옴. 7번 슬롯에 있는 파일이름은 a.txt로 바뀌어있으니까. 프로세스 a와 b가 수행되는 순서를 context switch를 적절하게 가정하면서 했더니 결과적으로는 비극적인 결과 초래 (b 입장에서는) 시나리오 바꿔서, 프로세스 a가 다 수행을 끝내고 in의 값을 다 업데이트 하고 그다음 프로세스 b가 출력 하면 문제 없다. 프로세스 a는 7번 슬롯에 a.txt 쓰고 프로세스 b는 8번 슬롯에 b.txt 쓴다. 프로세스 a가 in의 값을 local에 넣고 local varible이 가리키는 곳에다가 자기가 출력할 파일이름 쓰고. 그다음에 local variable 하나 증가시키고 그다음에 증가된 local variable 값을 shared data in에다가 쓰고 . 그때까지 context switch가 안일어나면 아무 문제가 없다.그다음에 프로세스 b가 와가지고 in의 값을 local에. 8에다가 쓰고 9를 만들어서 in에다가 overwrite. 이렇게 하면 문제가 없다. 첫번째 시나리오에서는 b 비극. 두번째는 a,b 가 happy . 이게바로 race conditions 두개이상의 프로세스가 shared data. 이경우에는 in. 그것을 동시에 읽거나 쓸때 최종 결과가 어느놈이 정확히 언제 수행되는가에 따라서 결과가 첫번째 시나리오와 두번째 시나리오로 달라질 수 있다.

race condition이 발생하지 않게 하려면 process a가 shared data인 in을 읽고 증가시켜서 다시 업데이트 하고 이러는 동안에 process b가 똑같은 일을 안하게 하면 된다. 그게 바로 mutual exclusion (상호 배제)- 한개의 프로세스가 shared data. 즉 메모리상의 shared variable 또는 shared file을 사용(읽거나. 쓸 때) 다른 process가. 같은 짓을 하지 못하도록 막는것, 배제하는것 . 그게 바로 mutual exclusion이다. 상호 배제를 제공하면 아까와 같은 race condition은 발생하지 않을것이다.
mutual exclusion 아주 중요하다.

critical region 아주 중요한 개념.
shared data가 access되는 프로그램의 부분. 헷갈리는 경우. critical region이 뭐냐고 했을때 shared data 그 자체라고 헷갈리는 사람이 꽤 있다. 그게 아니고 shared data를 access하는 program의 코드 부분. shared data를 읽고 쓰는. 코드 부분. in 을 읽는다던지 in에다가 증가된 값을 쓴다던지. (critical section)

Critical -section problem 에 대한 좋은 솔루션
race condition을 방지하기 위해서는 mutual exclusion을 제공하면 되는데 mutual exclusion만 제공해서는 원활한 문제해결이 되진 않는다. 효율적이진 못함. 그래서, 아주 좋은 solution. 은 mutual exclusion 뿐만 아니라 몇가지 더 제공해야 한다.그 이야기. 4가지
1. mutual exclusion이 된다. 어느 두개의 프로세스도 동시에 critical region에 있으면 안된다.
2. cpu의 스피드나 넘버에 대해서 어떠한 가정도 해서는 안된다.
3. critical region 밖에서 수행중인 process가 critical region 속으로 들어가려는 다른 프로세스를 막아서도 안된다.
4. critical region에 들어가려는 어떤 프로세스도 영원히 기다려서는 안된다.

page 4. Mutual exclusion을 그림으로 그린것
오른쪽으로 갈수록 시간이 흐름. process a가 t1에 critical region에 들어감. 아무도 안들어가 있는 상태. process b가 t2에 들어가려고 시도. mutual exclusion의 경우에는 못들어가게 막음. b blocked. a가 t3에 critical region 떠남. 그때 b는 비로소 critical region 들어가는게 허용. 이게 바로 mutual exclusion. 시간 t4에 b가 critical region 떠남 . 결과적으로 mutual exclusion 말하는 것.

mutual exclusion을 어떻게 구체적으로 제공?
1. disabling interrupts
cpu의 interrupt가 외부에서 자기의. interrupt pin쪽으로 들어오는데 그것에 대해서 반응을 못하게 할 수 있다, disable interrupt instruction을 cpu가 수행하게 하면 된다. 그러면 interrupt signal이 들어오더라도 cpu는 그 interrupt에 대해서 반응하지 않게 된다.다시 반응을 하게 하려면 enable instruct 명령어를 수행하게 해야한다. interrupt를 disable하는것에 대해 이야기를 하는 이유는. 이게 바로 mutual exclusion의 한 방법이 될 수 있다. mutual exclusion은 한 프로세스가 공유 데이터를 사용하고 있을때 다른 process 가 똑같은 짓을 못하게 하는 것. 인데 이게 disabling interrupt 랑 무슨 관계.. 한 프로세스가 공유 데이트를 읽거나 쓰는것. critical region에 들어가있는것. 공유 data를 access 하는 program의 code부분을 수행하고 있는 상황. 그때 한 프로세스가 critical region에 들어 있을때 context switching이 일어나지 못하게 막으면 되지 않겠어요 그러면 다른 process가 critical region에 못들어옴 context switching 자체가 안일어남. 바로 그 방법.mutual exclusion 제공. disabling interrupt 라는 방식은 한 프로세스가 critical region 에 들어간 직후에 interrupt를 disable시킨다. disable interrupt 명령어로 . 그래서 critical region에서 할일을 하고 critical region에서 나올때 interrupt를 enable 시킴.결과적으로 이 프로세스가 critical region에 있는 동안에는 interrupt 걸지 못하니까 clock interrupt도 못건다. clock interrupt는 context switching에 이용 , process를 수행하다가 할당된 시간이 되어서 다른 process에게도 기회를 주고 싶을 때 context switching을 하게 되는데 그때 그것을 어떤 메커니즘으로 하냐면 clock interrupt로 한다. 즉 process 하나 수행하다가. 예를들어 30 ms (한 프로세스 허용시간) 30ms 후에 clock interrupt 통해가지고

30ms 가 걸린 것을 알 수 있으니까clock interrupt 가 걸리면. 30ms 인 경우에는 clock interrupt handler 통해서 결과적으로는 os scheduler 이용해서 running 상태 프로세스를 다른 프로세스로 context switching. 그런데 이것 자체의 출발을 clock interrupt를 받아들일 수 있어야 context switching을 할 수 있다. interrupt 자체를 disable 시켜버리면 clock interrupt가 cpu에 걸려고 해도 반응을 안한다.그렇게 함으로써 mutual exclusion 제공하겠다. 단점은 user process가 interrupt 를 disable 하고 enable 하고 이런 것을 허용하게 하면 위험할 경우도 있다. 왜냐면 user program은 user 가 졸다가 프로그래밍 할수도 있고. critical region에 들어가기 전에 disable interrupt 명령어를 승인 했는데 critical region 다음에 enable interrupt 코드를 안넣으면 ? 프로그램 수행 되면 interrupt가 disable 된 채로 cpu는 interrupt를 영원히 못받아들임 .context switching 영원히 불가. 안좋은 approch. disabling interrupt 쓸 때는 kernel code에서 이런 걸 쓰는 경우가 있다. kernel code는 os designer가 coding, os designer는 신중하게 코딩. kernel 이 os의 data structure같은걸 access 해서 update 한다거나 그럴 때 잠시 interrupt에 disable 하고 os data structure update 하고 다시 re enable interrupt 하는 경우 있다. 이런경우 에는 보통 kernel code 에서 사용한다.

race condition 방지하기 위해서는 mutual exclusion 제공해야 한다. mutual exclusion 제공하는 방법들 소개 -> . 알고리즘. 메커니즘들. disabling interrupt. Mutual exclusion을 제공하려면 한 프로세스가 critical region에 있을때 다른 프로세스가 끼어들게 하는 문제 발생. process switching 또는 context switch가 일어나서 다른 process가 끼어들면 문제가 발생. 한 프로세스가 수행중일 때 다른 프로세스가 못 끼어들게. 어떻게? 한 프로세스가 수행하다가 다른프로세스가 수행되기(running) 위해서는 . Context switch가 발생하는 메커니즘은 clock interrupt를 통해서 가능. clock interrupt를 이용.context switch 가 못일어나게 하면 mutual exclusion 제공될 수 있다. context switch 못일어나게 하려면 cpu에 clock interrupt를 거는것에 대해서 반응 하지 않으면 된다. interrupt 받아들이지 않으면 된다. Interrupt disable. Cpu가 interrupt를 disable하려면 disabling interrupt 라는 명령어 수행하게 한다. 어셈블리 언어의 instruction 중에 있다. cpu는 interrupt에 반응하지 않는다. disabling interrupt 방식은 프로세스가 critical region에 있을때 disabling interrupt 함으로써 interrupt들을 다 disable. Interrupt signal 에 cpu가 반응하지 않게 한다. critical region 다 수행하고 나선 enable interrupt. 명령어 수행. user process가 이방식을 쓰는 것은 현명하지 못함 -> 전 시간에 설명한것 . Disalbe interrupt 하고 critical region 수행하고 enable을 안해버리니까. 결국 clock interrupt 반응 안함. context switch 일어나지 않음. -> 시스템 오류. kernel 프로그래머가 이방식 종종 쓴다. (위에 설명) mutual exclusion을 제공하는 그 외 다른 방식

lock variable -software solution
공유(shared) lock variable을 쓰는 것 . 알고리즘 -> 프로세스가 critical region에 들어가길 원하면 공유 lock variable 줄여서 lock을 테스트한다. lock이 0이면 프로세스 1로 세팅한다음 critical region에 들어감. 이미 lock이 1이면 ? 그 프로세스는 lock 이 0이 될때까지 기다린다. 실패한 알고리즘 .Mutual exclusion 제공 못함
-> 프로세스 a 가 먼저 critical region에 들어가려고 lock 체크 했는데 0이었음. 1로 바꾸려고 하는데 context switch(process switch)가 발생했다고 가정 .프로세스 b 차례. critical region에 들어가고 싶어서 rock 체크. rock이 0이니까 1로 세팅하고(b는 cpu차지하고 수행한다고 가정) critical region에 들어감. critical region에서 수행. 그 다음에 context switch가 수행되서 c가 수행되고 …d가 수행되다 a차례가 돌아와서 아까 한것 다음부터 수행. lock의 값을 1로 세팅. 이미 process 가 1로 세팅해논 후임. a는 알고리즘 단계에서이니까 세팅하는 것 뿐 . 그다음에 critical region으로 들어감.사실상 process a는 1로 된 lock을 1로 overwrite 하고 critical region에 들어감. critical region에는 b가 들어와있음. 그런데 프로세스 a로 들어가는 상황 발생.critical region에 두마리의 프로세스가 활동 .mutual exclusion 위배 -> 실패한 알고리즘이다. spooler directory 알고리즘 과 똑같은 race condition이 발생

세번째 알고리즘 -strict alternation
critical region에 들어가는 것을 엄격하게. 프로세스 a,b 가 있다고 가정. 프로세스 a는 a 코드 수행하고, 프로세스 b는 b의 코드를 수행 (그림)
프로세스 a가 먼저 수행된다고 가정. 공유 lock variable인 tum이 먼저 수행. tum이라는 variable 값이 0으로 initialize . 프로세스 a가 먼저 while 루프에 들어감. while(true) 무한루프. 프로세스 a가 while(tum!=0) ;조건 체크. tum은 0이니까 while(); // null statement 조건 거짓이니까 바로 빠져나옴. critical region();에 들어간다. 이때 b 프로세스가 아래 b 코드를 실행하면서 while(tum!=1); 체크. tum의 값은 0이니까 1이 아니라 참. null statement 수행하고 다시 조건 검사하러 돌아옴. 루프를 돈다 (뺑뺑이로) 프로세스 a만 critical region에 있는 상태이고 프로세스 b는 못들어감. mutual exclusion 제공. 그러다가 a가 critical region 다 끝내고 tum=1로 세팅. 그 순간 process b 코드에서 1!=1 거짓으로 조건문 빠져나오면서 critical region에 들어간다. 그러나 프로세스 a는 이미 critical region을 빠져나옴. b만 critical region에 있으니까 mutual exclusion이 여전히 제공. process a는 아까 tum을 1로 바꾸고 noncritical region에서 수행중임. Critical region이 아닌 프로그램 코드. process b는 critical region 에서 수행.
Busy waiting. 굉장히 중요함.
While(tum) 에서 tum의 값을 계속 체크. tum의 값이 b인 경우에는 1이 아닐 때 참이니까 루프를 돈다. tum의 값이 1이 되길 기다리면서 .이런것을 busy waiting. 어떤 variable을 계속 루프를 돌면서 계속 테스트하는것. 어떠한 값이 나타날때까지. 값이 나타나면 루프를 빠져나감. 값이 나타날때까지 루프를 계속 돌면서 값이 나타나는지 체크. busy waiting은 좋지 못한 방식 . 쓰이는 것이 추천되지 않음 피하는게 좋음. Cpu time을 너무 단순한 루프를 도는데 낭비. 어떨 때 사용되냐면 waiting time이 굉장히 짧을 것으로 예상될 때. 루프를 도는 waiting 시간이 짧을 것으로 예상될 때. busy waiting에 사용되는 lock을 spin lock이라고 한다. Busy waiting에 사용되는 lock variable. 중요함.
strict alternation 방식은 race condition 예방. mutual exclusion 제공해주니까. condition3를 위반한다 race condition을 예방하기 위해서는 mutual exclusion을 제공하는 알고리즘을 제공해야한다고 했는데, 효율적인 알고리즘 이라면 mutual exclusion + 몇가지를 더 제공하면 좋다. (four condition == 강의록 3페이지 race condition 예방하는 효율적인 알고리즘의 조건) . 3번째 condition 위배. strict alternation 알고리즘이. 3번째 조건은 critical region 밖에서 수행중인 프로세스가 critical region에 들어가려는 프로세스를 막아서는 안된다. 프로세스 a가 a코드를 수행하다가 critical region 수행하고 tum = 1로 바꿈. non critical region으로 감. b는 while 루프에서 기다리다가 spin lock 루프 돌다가 tum 1로 바꿔주는 바람에 while루프 빠져나와 critical region 들어감. tum=0으로 바꿈. 그다음에 non critical region 수행. 그다음에 b가 계속 수행된다고 가정, 다시 루프를 돌아야함. while(tum!=1) 체크. tum은 0임 b가 tum을 0으로 바꾸고 왔기 때문에. 프로세스 b가 while루프 두번째 돌때 조건이 참이라 계속 busy waiting 수행. condition. 3 violate. 이때 프로세스 a는 noncritical region에 있다. critical region에 못들어가는 상황 발생 . 프로세스 a는 non critical region에 있는데 critical region에 들어가려는 프로세스 b를 막고 있는 셈. 왜냐하면 프로세스 b가 들어가려면 프로세스 a가 non critical region 수행하고 나와서 다시 while 루프 체크 해서 . Tum이 0이어서 a 입장에서는 0!=0 거짓. critical region에 들어가서 tum =1 로 해야 프로세스 b가 critical region에 들어갈 수 있다. a가 그것을 안해줘서 b가 못들어감. a책임. a가 non c.r(critical region)에서 꾸물거리고 있어서 b가 아무도 없는 critical region에 들어가는 것 막고있다. condition 3 위반. 따라서 이 알고리즘은 mutual exclusion 제공은 해도 4가지 다 만족시키지는 않아서 완벽한 알고리즘은 아니다. 한프로세스가 다른 프로세스보다 매우 느릴때 서로 돌아가면서 수행하는 . 것은 바람직하지 않다. 이 알고리즘은 두 프로세스가 번갈아가면서 critical region에 들어갈 때 mutual exclusion이 잘 제공 . 한프로세스가 빨라서 다른 프로세스 보다 critical region에 나와서 또 들어가려고 하고 그러면 못들어간다 . spooler directory 상황에서 이 알고리즘 쓰면 ,파일을 하나가 print하고 상대방 프로세스가 자기 파일을 print해야지 내 파일을 print 가능. 답답한 상황 발생. 번갈아가면서 할때만 제대로 작동.

strict alternation 방식 완벽하지 못하다. dekker라는 사람이 교대하지 않고도 돌아가는 알고리즘을 개발 . peterson이 좀 더 simple한 알고리즘 개발. peterson's solution. 엄격하게 교대하지 않더라도 mutual exclusion을 제공. 프로세스 2개를 가지고 알고리즘 보여주고 있다. 두개의 프로세스 가정하고 , 프로세스 0, 프로세스 1 두 개의 프로세스가 돌고 있다고 가정. 각 프로세스는 코드가 어떤 코드를 수행하고 있냐면 enter_region 코드와 critical region 코드가 있고 leave_region 코드가 있다. 프로세스 0,1. 프로세스 0는 enter_region call 할때 argument를 0으로 주고 leave_region call 할때 argument 0으로 준다. process 1 인 경우 각각 argument 1로 주는 것으로 가정. 프로세스 0가 enter_region(0) call 하면 enter _region procedure로 와서. local varible other은 other=1-0 으로 1; 이된다. 상대방 프로세스 번호이다. interested[0]=true; interested는 int interested[n] . 배열로서 공유 variable이다. int turn도 공유 variable이다. interested[process==0]=true; critical region에 들어가는 데 관심있다. 들어가겠다. Turn = process; turn 값이 0이 된다. while을 수행. while(turn == process && interested[other] ==true) ;trun의 값은 0이다. Process도 0이다. 참. interested[1]==true 는 거짓. interested 배열값이 0으로 초기화되어있다고 가정. false. interested [1]==true 이것은 거짓. 초기값이 false 이다. 그래서 아직 process 1는 enter_region call 안했다고 가정. 전체 논리조건이 거짓. while을 빠져나옴. return. 그다음에 process 0의 코드가 enter_region call하고 그다음에 critical region 수행하고 leave_region call한다. enter_region return 했으니까 critical region 수행. 이때 process 1이 enter_ region call했다고 치면 enter_region(1)이다. other =1-1; 0; other은 0이다. interested[1]=true; 로 세팅 하고. trun = process // =1; 로 세팅. while루프로 가서 trun==process 는 참. interested[other] == true other은 0. 상대방은 true이다. 상대방이 아까 true로 세팅 . 그러니까 상대방은 critical region에 들어가있는 상황 . 프로세스 1이 while 루프에서 참. null statement 수행하면서 루프를 돈다. mutual exclusion이 제공. 프로세스 0가 critical region 다 수행하고 나서 leave_region 0을 call. interested[0]=false; 자기의 관심을 false; 프로세스 1 입장에서 while루프에서 돌고 있는데(busy waiting) 프로세스 0 덕분에 논리곱 뒷부분이 interested[0] = false로 바뀌어서 거짓이 됨. 프로세스 1이 while에서 빠져나옴. enter_region 에서 return. critical region으로 들어간다. 프로세스 a는 이미 critical region에서 나온상태. b가 들어가도 문제 없다. mutual exclusion. race condition이 발생하는지 살펴봄. 두개의 프로세스가 동시에 enter_region call을 하면 , 프로세스 0이 enter region 0을 call함. other는 1로 세팅하고 interested[0] = true; 상대방 프로세스 1도 enter_region call 해서 other = 0으로 해서 interested[1]=true; . 0과 1이 관심을 true 로 해둔 상황. cpu 하나라고가정 둘다 interested[true] 간발의 차이로. 프로세스 0가 turn = process 수행하면 공유 variable인 turn 값이 0이 된다. turn 은 while루프로 간다. 앞의 while문은 참이고 interested[other]은 상대방이 해놔서 true. 프로세스 0는 간발의 차이로 먼저 while루프에 왔지만 루프를 돈다(조건이 다 참이라서). 약간 늦어진 프로세스 1은 프로세스 0이 루프를 돌고있는데. turn =process로 세팅. turn이 1로 바뀐다. 프로세스 1 이 turn 을 1을 바꾸는 바람에 프로세스 0가 turn 1로 바꿔서 while 조건 앞의 부분이 거짓. 조건문이 거짓 -> 루프 빠져나와서 enter_region에서 리턴하고 프로세스 0는 critical region으로 들어간다. 프로세스 1은 turn=process 바꾼 후 while조건체크. turn==process 참이고 1==1 . interested[other] ==true; 이것도 참. 간발의 차이로 온 1은 루프를 돈다. 그래서 프로세스 0만 critical region에 들어가서 mutual exclusion 제공. 프로세스 1은 프로세스 0이 critical region 끝내면 leave_region을 call하는데 leave_region(0) interested[0]=false. 프로세스 1은 busy waiting 중인 while루프에서 드디어 interested[other]이 false가 되어서 거짓. while문 빠져나와서 critical region에 들어갈 수 있다. 프로세스 0은 leave_region call한 상태. mutual exclusion이 잘 제공.

disabling interrupt, lock varible 사용 . lock variable 알고리즘 대로 사용하면 실패. strict alternation. peterson solution
tsl instruction 사용해서 mutual exclusion 제공. race condition 예방 해보겠다.tsl rx,clock (test and set lock 의 약자) lock 변수를 test 하고 setting 한다. lock 변수 값을 테스트하고 1로 세팅한다. tsl rx lock. rx 가 register 연산자 lock이 lock 변수 tsl이 하는 구체적인 일은 memory word인 lock 변수의 값을 읽어서 어디로 읽어들이느냐 register에 읽어들어서 이경우에 tsl r1 .r1에다가 읽어들임. 그다음에 non zero value를 lock 변수에 저장한다. 보통 1을 lock 변수에 저장한다. ->tsl이 하는 일 내부적으로 두가지 operation이 있다. 레지스터 값을 lock변수로 읽어들인다. 또하나는 lock변수의 값을 1로 세팅한다, 중요한 것은 , 이 tsl은 indivisibl 쪼갤 수 없다. 내부적으로는 lock의 값을 레지스터로 읽어들이는 operation과 lock변수의 값을 1로 세팅하는 operation 두개가 있는데 처음 operation을 하고 나서 context switch가 일어난다거나 이런게 불가능 .indivisable. 이 안의 내부적인 operation이 다 일어나든지 하나도 안일어나든지. tsl 이 instruction 하나임 (명령어 하나) 명령어 내부에사 context switch나 process switch가 일어날 수는 없다. 이것을 이용해서 mutual exclusion 제공하는 코드. enter _region procedure와 leave_Region procedure. critical region 코드가 있으면 그거 앞부분에 enter_region을 call하는 부분이 있어야 하고 그다음 critical region. 그 다음에 leave_Region을 call 하는 proocedure가 있어야 한다 . 그렇게 하면 critical region에 의해서 mutual exclusion이 제공된다. 여러개의 process들이 그런식으로 코딩이 되어 있으면 오직 여러개의 프로세스 중 하나만이 critical region에 들어갈 수 있다 .mutual exclusion제공. enter_region: 코드. leave_region의 assembly 코드는 굉장히 중요하다. 굉장히 굉장히 중요
TSL REGISTER, LOCK . lock의 값이 레지스터로 copy되고 그다음에 lock이 1로 세팅. 현재 lock을 shared variable로 가정. lock. 공유 변수 lock이 0이라고 치자. 그러면 TSL REGISTER,LOCK 하면 lock의 0의 값이 레지스터로 copy되고 그다음에 lock 이 1로 세팅. CMP REGISTER, #0 그다음에 레지스터를 상수 0과 비교. 레지스터는 0이니까 , 0과 0을 비교. 그다음에 JNE enter_region JNE = jump if not equal. 레지스터와 0이 같다. not equal 이 아니다 . 따라서 not equal이었으면 enter_region으로 점프해갔을텐데 같으니까 다음 명령어로 return. RET enter_region procedure로부터 return. 그럼 이제 critical region 코드가 기다린다. lock의 값이 0이면 enter_region call했을때 자연스럽게 return 되고 critical region으로 들어간다. lock의 값이 1이라면 (만약), TSL lock이니까 TSL REGISTER,LOCK lock의 값을 레지스터로 copy. lock 이 1이니까 레지스터는 1. 그다음에 lock의 값 1 -> 1 로overwrite. CMP REGISTER, #0 레지스터 1이고 0과 비교. not equal. 같지 않으면 enter region으로 가니까 다시 위로 올라감. 다시 TSL 수행. 계속 루프 (lock이 1일때는) .즉 다른 놈이 먼저 enter_region을 call해서 lock의 값을 0을 1로 세팅해둔 후에는 다른 프로세스가 enter _region을 call하더라도 loop에서 돌게 된다. 한 프로세스만 critical region을 수행. 그러다가 먼저 critical region에 들어가있던 프로세스가 process가 끝나면 leave_region을 call 하게 코딩을 해야한다. leave_region이 call되면 MOVE LOCK,#0 lock의 값이 0으로 바뀐다. 루프 돌던 프로세스는 TSL REGISTER,LOCK에서 lock이 0이니까 레지스터에 0을 copy하고 그다음에 lock은 다시 1로 세팅 . CMP REGISTER,#0 에서 0과 0을 비교해 equal. JNE를 만나도 같으니까. return 한다. 이프로세스도 critical region에 들어간다 . 아까 프로세스는 이미 나와있는 상태 (critical region에서) mutual exclusion 보장. critical region을 enter_region과 leave_region으로 감싸면 어느 순간에는 한 프로세스만 들어가 있다 ->mutual exclusion이 제공 된다.

지금까지 사용한 방법은 다 busy wating 을 사용한 방법임. tsl도 루프도는 명령 - > busy waiting. 그런 것들이 다 busy waiting, 을 쓰는 경우의 아쉬운 문제점은 cpu을 너무 단순한 루프 도는데 사용함. cpu time을 낭비. 그리고 priority inversion problem(참고로만 알고있기) 간단히 말하면 현재 os가 priority scheduling을 하고 있다고 가정. 뭔가 하면 process state diagram. (running., blocked , ready). running으로 올라갈 프로세스를 선택하는 것이 scheduling 알고리즘 . scheduling 알고리즘은 다양하게 있는데 그중 하나가 priority scheduling.이건 뭔가 하면 ready 에 있는 애중에 제일 우선순위가 높은 애를 running으로 보내는 것. 정확히 말하면 running에 있는 애랑 ready에 있는 애랑 다 합쳐서 제일 우선순위가 높은 애가 언제든지 cpu 차지하게 해주는것. 그러한 scheduling 에 쓴다고 가정 했을 때 priority intervion problem이 발생 가능. busy waiting을 쓰게 되는 경우 예를 들어서, low priority process 가 있고 high priority process 가 있다.

high priority process가 running상태에서 수행되다가 read system call 만나면 blocked로 빠짐. running에서 내려옴. 그러면 이제 ready에 있는 애 중에 제일 priority 높은 애가 running으로 간다. 수행하고 있던 low priority process가 critical region에 가고싶다. enter_region call 하고 critical region에 들어감. blocked로 빠졌던 high 는 i/o가 완료(read)완료해서 ready 상태로 간다. 그러면 현재 priority scheduling을 가정했으니까 얘가 인제 ready로 온 이상 현재 수행중인 low priority는 우선순위에서 밀린다. high priority process가 ready로 오면 다시 cpu를 차지하게 된다. low prority는 ready로 빠진다. high priority process도 같은 criitical region에 들어가려고 enter region을 call. busy wating을 한다.문제는 busy waiting 에서 풀려나려면 lock이 1에서 0으로 바뀐다던가 해야한다.lock을 풀어주려면 low priority process가 걔가 아까 critical region에 있었음 . 밀려난 것임 . 그런데 얘가 lock 을 풀어줌으로써 high priority가 들어가게 해줘야하는데 lock을 풀어주려면 cpu를 running상태로 사용해야지만 critical region에서 나와서 leave region같은거 call해야 high priority가 enter region에 busy waiting에서 풀려나서 들어간다. 그런데 그게 일어날 수가 없다 . 왜냐하면 priority scheduling을 가정했기 때문에. low priority process는 running 상태로 가는것이 아예 불가능함. 자기보다 우선순위가 높은 high priority process가 있는 한. high priority는 cpu를 진짜 단순하게 쓴다 .(busy waiting)에 -> lock의 값이 0으로 바뀌길 기다리면서 . 그러면서 cpu계속 쓴다 물러나지 않고. busy waiting을 계속 무한루프 돈다 풀어줄수 있는 프로세스는 low priority인데 걔는 cpu를 차지 못한다. running상태로 갈 수 없다. 그렇기 때문에 시스템이 이상태로 계속 간다 결과적으로는 high priority process가 critical region에 아예 못들어간다 priority inversion problem . 참고로만 알고있자. busy waiting cpu의 문제 cpu time을 낭비한다. cpu time 낭비하니까 생각한 다른 방법 sleep and wakeup
sleep은 system call의 일종으로서 이것을 call 한 프로세스를 blocked 상태로 보낸다. 언제까지 냐면 다른 프로세스가 깨워줄때까지 (wakeup system call을 함으로서)
sleep and wake up 사용의 예

producer-consumer problem 예로 사용 (생산자 소비자 문제 -> bounded buffer problem) 두개의 프로세스가 공통의 제한된 크기의 buffer를 공유하는 상황. producer process와 consumer process가 있다고 가정, producer process는 공유 버퍼에 information을 생산해서 (data,item)등 공유 버퍼 넣는 일을 하는 프로세스. consumer process는 공유 버퍼에서 데이터를 (information,item)을 꺼내서 소비를 하는 역할을 하는 프로세스. 이때 producer가 버퍼에 새로운 item을 넣기를 원하는데 버퍼가 꽉 차있을 때 (제한된 크기) producer process는 sleep (blocked) 상태로 간다. consumer process가 공유 버퍼에서 item 하나 제거하고(꺼내고)깨워줄 때 까지. 마찬가지로,consumer process도 buffer로부터 item을 하나 뽑아내려고 (remove)하려고 하는데 버퍼가 텅 비어있다 .그러면 sleep(blocked)상태로 간다. producer 프로세스가 버퍼에 뭔가를 넣고 깨워줄때까지. sleep and wakeup code 내용
define N 100 //버퍼 슬롯의 개수. count=0 //버퍼의 아이템의 개수. count는 공유 변수이다. producer process는 producer function을 call. consumer process는 consumer function을 call 해서 수행. producer procedure의 코드를 보면 while루프를 돈다. item=rpoduce_item(); item을 일단 하나 생산한다 .그다음에 if는 일단 넘어가고 그 생상된 item을 insert_item(item); //insert_item call할때 item을 argument를 준다. insert_item이 하는 일은 공유 버퍼에다가 argument로 넘어온 item을 넣는 역할. count = count+1 .count는 버퍼에 차있는 아이템의 수 . 체크하는것. 마찬가지로 consumer도 while루프 돌면서 (if는 일단 빼고) item =remove_item. 공유 버퍼에서 item 하나 뽑아낸다 (제거한다) item으로 return 받아서 count는 하나 감소시키고 (공유 변수 count) if는 넘기고 consume_item(item) 공유 버퍼에서 꺼낸 아이템을 소비한다. 그러면 다시 producer로 가서while루프 속을 다시 들여다보면 (좀더 자세히) produce_item으로 item 생산 하고 나서 if(count ==N) 즉 버퍼가 꽉차있으면 -> sleep(); sleep system call 해서 blocked 상태로 들어간다. consumer의 while 루프에서 꽉차있는 버퍼에서 remove _ item 가능 . 그다음에 count=count-1; 100에서 99로. if(count==N-1) N-1 도 99이므로, wakeup(producer); 좀전에 sleep으로 가서 blocked 상태로 있는 producer process를 깨운다 -> ready 상태로 간다. 마찬가지로 , 버퍼가 텅 비어있는 경우에는 consumer가 while 루프에서 if(count==0)sleep; 으로 간다. 그러면 producer 쪽에서 while루프 쪽에서 item 하나 produce해서 insert_item(item) 공유 버퍼에 넣고 count = count+1로 count를 1로 만들어준다. if(count ==1)wakeup(consumer); 로 consumer가 깨어난다. 그럼 sleep으로 갔던 consmer가 깨어나서 remove item을 call한다.

producer - consumer problem이 sleep and wakeup call을 써서 자 문제가 풀렸는가? 문제가 있다. race condition이 발생할 수 있는데 현재 버퍼가 비어있다고 치자. consumer을 보면 거기에 while안에 if(count==0) zero 맞다 버퍼가 비어있어서 . count 가 0인것까지 확인했다고 가정하고 sleep call 하기 전에 context switch (process switch)가 발생해서 consumer은 지금 0에서 ready로 가고 producer process가 running으로 갔다. 현재 버퍼는 비어있다. produce_item으로 item하나 만들어서 insert_item 아이템 하나 공유버퍼에 넣고 count를 하나 증가시키니까 0에서 1로 바뀐다 . if(count==1) wakeup(connsumer) consumer을 깨움 -> consumer는 아직 sleep을 call 안함. wakeup 해봤자 효과가 없다.consumer가 이제 cpu 차지해서 running 상태. 로 바뀜. 아까 consumer는 count==0 까지 확인했다. 그다음인 sleep을 call을 함. 비로소 consumer가 sleep을 call. blocked 상태로 간다. 더이상 진행할 수 없다. producer는 수행 가능. 아이템 계속 produce 해서 공유버퍼에 넣고 count 1 증가시킨다. 언젠가는 공유버퍼가 꽉 찬다. while루프에서 producer가 item을 produce하고 난다음에 if(count==N) 을 확인하는데 꽉차있으니까 참. 그러면 얘도 sleep으로 간다. 그러면 consumer도 sleep. producer 도 sleep. 상대방을 깨워줄 수 없다. 그래서 등장 한 것이 semaphore 라는 컨셉

Semaphore
1965년도 다익스트라가 소개된 . 새로운 variable type .
사용은 down operation call 하고 critical region. up operation . mutual exclusion 제공할 때 이렇게 사용 .나중에 코드에서 사용하는 것이 나온다. 어떤 특정 semaphore variable에 대해 down operation call하고 critical region 코드 넣고 특정 semaphore에 대해 up operation 을 call 하는 순서대로 일반적으로 사용한다. 두 개 opreation 연관
down operation (P operation). P는 test를 의미 up(V) operation increment를 의미. 어떤 semaphore 변수에 대해 down operation을 하면 어떤 일이 발생하는가.
해당 semaphore가 0보다 크면 , 값을 하나 감소하고 계속 진행한다. 세마포어가 0이라면 down을 call한 process는 put to sleep(blocked)상태로 가게된다. down operation은 일단은 완료하지 못한 상태로 sleep하게 된다 . sleep에 처해진다.
up operation 하게 되면 하나 이상의 semaphore에 대해서 sleep하고 있던. 그중의 하나가 깨워진다. 그리고 down operation을 마무리하게 해준다. down에서 return을한다. 그러나, (그렇지않으면) 하나이상의 process가 semaphore에 대해 sleep 하고있지 않으면(아무도 잠자고 있지 않으면) semaphore의 값을 하나 증가시킨다.
operation. 은 쪼갤 수 없는 (indivisible) 원자적 action(atomic action)이다. 예를 들어 down operation 을 들여다보면 if semaphore is greater than 0. semaphore가 0보다 큰 것 체크. semaphore 값을 하나 감소시키는 operation. down속에 있는 두개의 operation을 쪼갤 수가 없다. 즉 semaphore가 0보다 큰 것을 체크하고 난다음에 context switch가 일어날 수 없다.-> 핵심 up operation도 indivisible atomic action. 굉장이 중요한 operation 굉장히 중요.

producer-consumer problem을 semaphore를 써서 해결. code 보기 전 설명. for mutual exclusion . for synchronization 두가지가 있다.semaphore를 사용하는데 상호배제를 목적으로 사용하는 semaphore가 하나 등장하고 이름을 mutex라고 붙임. synchronization(동기화) 목적으로 사용하는 semaphore가 두개 등장. 하나는 full, 하나는 empty. total 3 semaphore. mutex 는 mutual exclusion 목적. full,empty는 synchronization 목적. mutual exclusion은 오직 하나의 프로세스만이 한 번에 공유 버퍼를 읽거나 쓰는 . 관련 버퍼를 읽거나 쓰는 . 쓰도록 보장해 준다. mutex semaphore 등장. producer와 consumer process가 동시에 공유 buffer를 access하는 것을 막아줌 이때 mutex라는 세마포어는 binary semaphore로 볼 수 있다. 1로 초기화 해놓고 두개 이상의 process중에 critical region에 동시에 들어갈 수 있는 것은 오직 하나. 를 보장. 그러한 목적으로 사용되는 세마포어가 binary semaphore. mutual exclusion 제공. full이랑 empty 라는 이름의 semaphore는 synchronization (동기화 목적 사용)
producer process가 수행을 멈추게 (언제 .buffer가 full일때) consumer process가 stops running(수행을 멈추게) 언제 .공유 버퍼가 비어잇을 때. full semaphore는 차 있는 슬롯의 개수를 counting 하게 되고 empty semaphore는 비어있는 슬롯의 개수를 counting 하는 데 사용.

#define N 100 // 공유 버퍼 slot의 개수. sema'phore mutex , empty, full이 등장. mutex 는 1로 초기화 (binary semaphore) 에서 .mutual exclusion 제공 위해. empty와 full이 각각 N과 0으로 초기화. synchronization 목적.producer 와 consumer function(procedure)의 형태. producer는 while루프에서 item=produce_item() 아이템을 하나 생산해 낸 다음에 down은 일단 제끼고 insert_item(item) 공유 버퍼에 넣는 일. insert_item이 critical region 이라고 할 수 있다. 공유 버퍼에 item을 넣는다. up 제끼고 consumer process쪽 보면 무한루프 돌면서 item=remove_item(); 이게 바로 critical region 부분 .공유 버퍼로부터 item 하나 꺼내는. 제거하는.. 그래서 return 해주는. 맨 아래 consume_item(item)뽑아낸 아이템을 소비하는 부분.
자세히 ->
producer 부분을 보면 while루프로 들어가서 item=produce_item()아이템 만든 다음에 down(&empty); down(&mutex); down (&mutex) 부터 보면 insert_item 위에는 down mutex 밑은 up mutex. down mutex와 up mutex로 insert_item을 감쌌다. producer code에서. insert_item이 공유 버퍼에 item을 넣는건데 이부분이 바로 critical region. 그렇기 때문에 mutual exclusion 제공될 필요가 있다.그래서 mutex binary semaphore를 사용 . binary semaphore가 mutual exclusion 제공. 처음에 1로 세팅 . down(&mutex); insert_item(item); up(&mutex); 이런식으로 사용을 하면 insert_item이라는 critical region에 한놈만 들어간다. consumer쪽을 보면 down(&mutex); item=remove_item();//critical region 공유 버퍼에 데이터 하나 뽑아내는 코드 . up(&mutex); mutual exclusion 제공. 예를 들어 mutex=1이라고 했을 때 producer가 먼저 down(&mutex); mutex가 현재 1이니까 down operation이 . semaphore가 0보다 클때는 value를 0으로 감소 시키고 진행한다. 그다음 insert_item 수행. 이때 consumer가 또 수행되었다고 가정, while 루프에서 down(&mutex); mutex값이 producer에 의해 0으로 세팅 . down operation에서 semaphore가 0이면 sleep에 처해짐 (blocked). consumer는 일단은 sleep. 그러다가 producer가 insert _ item. critical region 다 수행하고 공유버퍼에다가 item 넣고 up(&mutex). 부름 . producer가. mutex는 0인데 up operation은 하나 이상의 프로세스가 semaphore대해서 잠들어 있다면 ->consumer가 잠들어 있음. 걔를 깨우고 down operation을 마무리 할 수 있게 함. consumer process 는 down(&mutex)에서 리턴해서 item=remove_item(); 공유버퍼에서 아이템 하나 제거. up(&mutex); mutex값은 아직0. 아무도 자고 있지 않음. up operation에서 semaphore 값을 0에서 1로 증가시킨다. down(&empty) 부터는 내일.

down , up operation 매우 중요. producer- consumer problem 세마포어로 해결. mutual exclusion 용도로는 mutex, synchronization용도로는 full,empty 세마포어 producer가 while루프 속에서 item=produce_item 아이템 하나 생산해내고 insert_item 아이템 공유 버퍼에 넣는다. consumer는 공유버퍼에서 remove_item item뽑아내고 consume으로 소비. producer로 다시 가서 producer에서 insert_item(item)하기 전에 down(&mutex) 하고 나서는 up(&mutex) 가 있다. 이것이 바로 세마포어를 mutual exclusion사용 하는것.mutex값이 처음에 1로 초기화. semaphore 변수를 1로 초기화해노면 critical region의 앞에는 down 밑에는 up. 보통 binary semaphore. 라고 한다. 1로 초기화 해놓고 하나의 프로세스만 들어가는. mutex는 binary semaphore로 사용되었다 .그래서 producer가 먼저 down(&mutex) call하면 mutex가 1이니까 down 알고리즘에 의해 세마포어값이 0으로 바뀌고 진행. insert_item(item)으로 아이템 공유버퍼에 넣는다. 그때 consumer process가 작동했다고 봤을때 down(&mutex) mutex가 0이기 때문에 down operation에 의해 sleep. (blocked) 결국 consumer는 sleep. remove_item() 하는 critical region에 못들어간다 . 그러면 결국 producer process가 insert_item 하고 up(&mutex); up operation은 mutex에 자고있는 애가 있다. 그럼 그중의 하나를 깨운다. consumer process가 깨어난다. (blocked -> ready) 그리고 걔는 remove_item() 을 call 한다. // critical region에 들어감. consumer가 critical region에 들어갈 때는 producer process는 이미 critical region에서 빠져나온 상태. 그다음에 볼거는 empty.full

full, empty는 synchronization용도. (동기화) 버퍼가 꽉 찼을때는 producer process stop. 버퍼가 텅 비었을 때는 consumer process를 stop하게 해주는 그런 용도로 사용. mutex semaphore 는 for mutual exclusion. binary semaphore로 사용.
code ->
처음에 consumer process 가 버퍼가 텅 비어있다. down(&full) call하면 , full semaphore가 값이 현재 0임. down 알고리즘은 semaphore 값이 0이면 process is put to sleep . sleep 상태에 빠진다. 그러면 나중에 producer가 수행될 때 item=produce_item() ;아이템 하나 생산해내고 down(&empty); empty는 처음에 버퍼슬롯 수. N. 위에 N으로 초기화됨. down 알고리즘이 semaphore 값 하나 감소 (0이 아닐때) N-1 로 하고 다음으로 진행 . down(&mutex); mutex값이 1이니까 0으로 감소하고 진행 .insert_item(item) 공유 버퍼에 또 처음으로 item을 집어넣는다. 거기가 critical region. 그다음에 up(&mutex) up 알고리즘에 의해 mutex가 0으로 바뀌어있으니까 up 알고리즘이 mutex를 1로 증가 (아무도 자고있지 않으니까). 그다음에 up(&full); up알고리즘이 수행되면 full에 대해 잠든애가 있는지 본다. -> 아까 consumer가 down(&full)해서 잠듬. 걔를 깨운다 producer의 up(&full)이. 잠들었던 consumer process가 blocked에서 ready로 보내진다. consumer가 running상태로 가게되면 down으로부터 return. down(&mutex) call해서 1-> 0 으로 바꾸고 진행. item=remove_item(); 한 후 up(&mutex); mutex값을 0을 다시 1로 올리고 up(&empty) 이런 식으로 진행을 한다. 이게 바로 full과 empty라는 semaphore가 synchronization 용도로 사용되는 부분이다.
mutexes
mutex는 semaphore 는 아니다 .. -> 앞의 예제에서 semaphore variable 이름을 mutex라고 붙인 게 있는데 mutex는 세마포어이지 얘가 mutex는 아니다. 변수명으로만 쓴거임. mutex는 not semaphore. 아까 (mutex는 세마포어이다)
not semaphore이지만 세마포어 기능을 단순화 시킨 놈. mutex는 두 가지 state 중에 하나가 될 수 있다. unlocked , locked . unlocked는 mutex값이 0이다. mutex가 locked상태면 non-zero 보통 1을 가진다. mutex는 unlocked와 locked (0과 1)을 보통 왔다갔다한다. 보통 어떻게 사용하냐면 critical region code 위에 mutex lock을 call하고 밑에는 mutex unlock을 call한다. 편의상 mutex_lock mutex_unlock이라고 붙임 (lock,unlock이라고 간단히 할수도 있음) 이렇게 하면 mutual exclusion 제공할 수 있다 .mutex 구현한 예 (mutex lock, mutex unlock 어셈블리 언어로 구현)
thread환경에서 사용되는 예. 이해가 안가도 당황할 필요 없음 . 개념적인 것만 알고있기. (thread는 프로세스 내에서 실행되는 흐름의 단위를 말함) thread환경에서 mutex lock을 구현. TSL REGISTER,MUTEX . tsl명령어 사용 test and set lock . mutex라는 메모리상의 변수에 대해서 TSL 부르니까 mutex가 0이라고 치면 mutex0이 일단 register로 copy. register가 0. 그리고 mutex 값을 1로 세팅. test and set이니까. CMP REGISTER,#0 레지스터랑 0 비교. 레지스터는 현재 0. JZE ok jump if zero 는 앞 compare 명령에서 두개의 피연산자 레지스터 와 0가 있는데 그걸 뺀 결과값 레지스터 - 0 이 0(zero) 냐 그러면 점프해라. 이 얘기. 0-0=0 ok로 jump ok:RET return 0 따라서 mutex lock으로부터 return . mutex lock다음에 critical region 기다리고 있어서 critical region으로 들어감 . 이때 다른 process가 mutex_lock을 call 아까 프로세스는 critical region에 있음. 다른 process가 mutex _lock을 call해서 TSL 명령어 수행. mutex 1이니까 1이 레지스터로 들어가고 mutex값은 또 1. CMP 수행하면 1-0=1 . JZE? 아니라서 ok로 안간다. CALL thread_yield thread_yield를 call하는데 thread배울 때 배우는 procedure인데 이걸 call한 thread는 cpu를 내놓는다. running->ready. os의 스케줄러가 ready에 있는 프로세스 하나 뽑아서 running으로 . 수행이 중단되는 것. ready 상태에 있다가 os 스케줄러가 언젠가 뽑아서 running으로 가면 jmp mutex_lock을 call한다. mutex_lock이 맨 위에 label. 또다시 돌아가서. TSL 명령 반복. 단순한 busy waiting 아니고 cpu를 양보하면서 하기 때문. cpu 차지하면서 loop도는것은 아닌다. original thread가 critical region에서 나왔다고 가정 . mutex_unlockd을 call함. MOVE MUTEX,#0 0를 mutex에 넣는다. thread_yield에서 언젠가 ready상태로 간 얘가 언젠가 running으로 되면 jmp mutex_lock을 call하고 TSL 명령어 call 하면 mutex가 0이므로 레지스터가 0, mutex 1. CMP 명령어 수행하면 ok로 수행해서 return 하고 critical region에 들어가게 된다.

semaphore와 같은 shared data structure는 kernel에 보통 저장되고 ,system call로 보통 access . up ,down operation은 system call로 제공이 되고 user program 이 call하면 os kernel이 up과 down을 수행. 그러는 도중에 semaphore 값을 증가시키도 하고 감소시키기도 하고. kernel이 하는 역할. os kernel의 data segment에 semaphore 같은 변수들이 들어있다. 일반적인 shared data. 공유 버퍼 자체가 shared data가 될 수 있고 그것의 차있는 slot의 개수를 counting 하는 conunt 변수같은 것도 공유 data가 될수 있는데 프로세스가 이런 공유 data를 어떻게 공유하냐 . 메모리상의 shared data를 어떻게 공유하냐. process마다. unix같은 경우에 text data stack segment가 있는데 unix os system은 자신의 address space를 core image에 some portion일부를 공유할 수 있는 메커니즘이 있다. 그것을 사용하면 된다. 그런게 운영체제에서 제공 안해줘도 data 공유할 수 있는 방법은 file사용 . file에다가 data를 쓰고. 다른 프로세스가 그 file을 읽고 update 하면 된다. file을 사용하면 프로세스간의 data 공유 가능

monitors
semaphore는 코딩하는 게 까다롭다. 잘못 사용하면 큰 사고 발생. semaphore 예제에서 (16page) consumer code에서 코딩하다 졸았음. 조는 바람에 첫줄에 down(&mutex)가 오고 그 다음줄에 down(&full) 이 오게 코딩 -> 엄청난 사고가 발생. consumer 먼저 수행했다고 가정하면 while루프로 먼저 들어가면 down(&mutex)부터 수행. mutex값이 1이었으니까 0으로 바뀌고 return해서 진행. down(&full); full값은 0이니까 (버퍼는 비어있고) down 알고리즘에서 세마포어가 0이면 sleep. 더이상 진행을 할 수 없다. blocked 상태. producer의 차례가 와서 while루프에서 item하나 생산 후 down(&empty) empty 초기값은 N이므로 N-1로 감소 . down 알고리즘. 그다음에 down(&mutex) producer 코드는 스위치 안하고 그대로임 . down(&mutex)에서 mutex 에 대해 consumer가 아까 down(&full)해서 잠들어있다. consumer 아까 down(&full 하기 전에 down(&mutex)에서 mutex 1->0으로 바꿔둔 상태. 그래서 consumer가 아까 down(&mutex)먼저 call해서 mutex 0으로 감소시켜 놓고 down full해서 full이 0이니까 잠들어있고 producer로 차례 넘어와서 down(&mutex)했는데 mutex가 0이라서 down수행하다가 sleep(blocked). consumer도 down full에 의해 sleep, producer도 down mutex에 의해 sleep. 둘다 sleep. 상대방이 서로 자고있어서 못깨워줌. 자고있는상태로 계속 간다. 큰 사고 발생 가능 사용하기가 까다롭다. 그래서 monitor 등장

monitor 는 high-level synchronization primitive이고 procedure,variable,data structure들이 모여 있는것 . 특별한 모듈이나 패키지에 그룹화. 프로세스들은 모니터의 내부적인 data structure를 직접적으로 access할 수 없다. monitor가 mutual exclusion 기본적으로 제공하는게 중요한 포인트. 어느 하나의 프로세스만 어느 순간에 monitor에서 active 할 수 있다. 컴파일러가 mutual exclusion 제공. 컴파일러가 complie(기계어로 바꾸면서) mutual exclusion에필요한 instruction들을 적재적소에 배치. 그렇기 때문에 프로그래머는 신경 안써도 된다. monitor 정리하고, 사용하면 된다.

monitor example
monitor는 기본적으로는 이런식으로 생겼다 이안에 일반적 변수 정의 가능 .condition variable 정의 가능. 모니터 안에 프로시저 정의 가능. 그래서 monitor에는 condition variable이라는 중요한 concept이 있다.굉장히 중요 여기에 관련된 두개의 operation 굉장히굉장히 중요. (wait,signal) 내용을 알고있어야 한다 굉장히 중요하다.
condition variable에 대해서 wait를 call했을 때 (wait on a condition variable)
wait를 call한 process가. 이 해당 condition variable에 대해서 blocked을 하게 만든다. call한 프로세스가 blocked 상태로 간다. 이렇게 되면 다른 프로세스가 monitor 안에 들어갈 수 있다.

signal on a condition variable
해당 condition variable에 대해 sleep 하고 있던 process를 깨워준다. 아까 wait한 놈. 깨워준다는 얘기. mutual exclusion을 제공하고, signal은 accumulated 되지 않는다. 즉 condition variable에 대해 signal을 call했는데 wait를 call 한 애가 없다. 아무도 blocked 되어있지 않다 (condition variable에 대해서) 그러면은 그 signal은 그냥 없어진다. 아무런 효과가 없다.
monitor의 문제는 컴파일러에 의해 support되어야 쓸수 있다. language 차원에서 supprot해야한다. 분산환경에서도 적용할 수 없다.mutiple cpu가 서로 priviate 메모리를 갖고 있으면 local network으로 연결된. 그런 시스템에서 사용할 수 없다. c에서 지원하지 않는다. java에서는 제공.
monitor code c는 아니고 다른 언어.
왼쪽에는 monitor를 정의한것 (producerconsumer) producer process는 오른쪽 상단에 있는 procedure producer 이고 consumer procedure는 오른쪽 하단에 있는 procedure consumer를 수행하면 된다. 오른쪽 상단의 procedure producer는 while루프 수행. begin에서. item=produce_item item 하나 만들고. 그 아이템을 producerconsuer.insert(item) 왼쪽에 있는 정의된 모니터 . insert insert는 왼쪽에 정의된 모니터 속의 procedure insert를 call한것. arguement는 item. 오른쪽 하단의 consumer 코드는 while루프를 돌면서 item=ProducerConsumer.remove; ProducerConsumer라는 왼쪽의 모니터 . remove .. remove는 왼쪽의 모니터 아래에 fuction remove를 call하는것. 아이템 하나 뽑아서 return 받아서. 그 아이템 받아 consume하는 것 . 코드 단순화 -> 프로그래머 입장에서 편함. 모니터 정의한 왼쪽 부분을 보면 condition full,empty ; 중요한 condition variable이 등장. integer count ; 정수 변수. 버퍼에 차 있는 슬롯의 개수 count. 그다음에 procedure insert와 function. return을 하는것을 function으로 정의한것. insert는 return이 없어 procedure로 정의. procedure insert를 보면 ,if 건너뛰고 insert_item(item) 공용 버퍼의 item insert하고 count는 count+1 공유 변수의 카운터 하나 증가. 마찬가지로 function remove 하단에 보면 remove =remove_item ;공유 버퍼에서 아이템 하나 뽑아내고 count 는 count -1 공유 변수인 카운터 하나 감소. 이게 monitor이기 때문에 오직 한 놈만이 활동 가능. 예를 들어 producer 코드 수행되면서 ProducerConsumer.insert(6); //오른쪽 상단 코드 procedure insert가 수행 중이면 remove call해서 monitor의 remove function으로 들어오려고 해도 일단은 못들어온다. monitor 가 막아준다. 그래서 compiler 가 명령어를 만들어서 막아줌 .mutual exclusion. producer가 먼저 monitor 속으로 들어왔으면 다른 프로세스는 monitor에 있는 어떤 procedure 나 function으로 들어올 수 없다.
현재 버퍼가 비어있다고 치고 consumer가 먼저 수행되었다고 가정 .consumer는 while루프에서 item=ProducerConsumer.remove를 call한다. 왼쪽 monitor의 function remove로 들어온다. if count = 0 count는 0이다 지금 텅 비어있으니까 (count는 버퍼에 차 있는 슬롯 개수) then wait(empty); wait를 call한다. empty라는 condition에 대해서 . 알고리즘이 wait on a condition variable 하면 이걸 call 한 process 는 condition variable 에 대해서 block. 따라서 empty condition variable 에 대해서 block. blocked 상태로 가게 된다. (sleep). 버퍼가 비어있는 경우 count 가 0이니까 그렇게 된다. 그럼 기능을 못하게 된다. consumer process가 remove function 수행 도중에 if count =0 then wait (empty) wait (empty) 에서 wait call했으니까 blocked 된다. 그럼 consumer process가 condition variable에 대해 block. wait 알고리즘을 보면 allows another process to enter the monitor. 다른 프로세스가 monitor 에 들어가게 허용을 해준다 당사자는 blocked이기 떄문이다. 그러면 오른쪽 위에서 ProducerConsumer.insert(item)하면 들어갈 수 가 있다. consumer는 block이기 때문에 . insert로 들어가서 if count =N . N이 아니다 지금. 0이라서 . 넘어가고 insert_item. 공유 버퍼에 아이템 넣고 //critical region임 . count = count+1 count 하나 증가시키고 if count= 1 1 맞다 then signal(empty) empty condition variable에 대해서 signal 알고리즘. -> condition varible 의해 sleep 하고 있는 프로세스 깨움. sleep 하고 있던건 consumer . consumer가 wait (empty) 해서 empty 에 의해 block -> 깨어나서 ready 상태로 간다.

consumer 가 깨어남. 다음에 remove= … 수행 . producer process는 blocked되어 있지 않다. 활동중. consumer process도 활동을 시작. 그럼 동시에 두 개 프로세스가 monitor 속에 있다 , monitor rule 위반 (한놈만 활동) 이걸 해결하기 위한 메커니즘이 두가지
1. hohre 제시 -> signal을 한 프로세스는 block이 되게 한다. 그래서 wait에서 풀려난 애가 monitor 에서 활동하고 걔가 monitor 밖 빠져나오면 blocked 된 놈이 block에서 풀리는것 .
2. signal을 call 할 때 가장 마지막에 call -> 23페이지 가 두번째 방법을 쓴것. insert procedure 안에서 signal이 제일 마지막에 있다. 따라서 signal을 call하고 insert procedure에서 빠져나오는거니까 모니터에서 빠져나오는것이다, 결국은 consumer process만 monitor에 남게 된다.
버퍼가 N개 슬롯 가지고 있다고 가정 .count는 슬롯의 개수를 세는 공유 변수임.

interprocess communication. mutual exclusion 제공하는 . synchroniazation 제공하는 방법들에 대해 얘기를 했다. 다양한 알고리즘들이 공유 메모리를. 메모리에 있는 변수를 공유한다. 동일한 메모리를 access. 컴퓨터가 한대가 있고 그안에 cpu가 있고 cpu의 프로세스가 여러개 돌고있고 그다음 메모리가 cpu 옆에있고 그래서 프로세스들은 메모리상에 있는 공유 변수들을 access 하는 것을 가정해서 나오는 알고리즘. 만약에 프로세스가 그렇게 메모리를 공유할수가 없는 상황이라면 . 메모리를 공유할 수 있는 상황이 아니라면 interprocess communication 문제르 어떻게 해결하나. 어떻게 mutual exclusion과 synchronization을 제공하나. 프로세스들이 서로 메모리를 공유하고 있지 못하면 . 어떤 경우냐면 서로 다른컴퓨터에있을때. 프로세스 a는 1번컴퓨터에서 수행중이고 프로세스 b는 2번컴퓨터 수행중. 서로 네트워크로 연결. 그러면 메모리를 공유하고 있진않아서 공유변수 사용 불가능. 이때는 message passing 방식. interprocess communication 방식 중 하나인 message passing 방식. send와 receive system call사용 .
send(destination,&message) message를 des에 보내는 시스템
receive(source,&message) message를 주어진 source로부터 받는다. 여기서 des 나 source는 상대방 프로세스를 말할 수 있다. 서로 다른 컴퓨터에 존재하는 프로세스가 될 수 있다. 물론 같은컴퓨터에 있을 수 있지만 다른컴퓨터에 존재해도 작동한다 . 이때 receive system call이 message가 available하지않다 . receive call해놓고. receive를 call 한 프로세스는 메세지가 도착할때까지 block상태로 간다. 또는 receive system call이 즉각 리턴한다. error code와 함께. (구현하기 나름) 보통은 전자의 방법을 많이 사용. 책에도 그 방법을 쓰고 있다. (메세지 올때까지 block) message passing은 design issues가 있다. 하나는 network을 통해 message가 가는거니까 network에서 유실될 수 있다 (lost) 이것을 해결하는 메커니즘 -> acknowledement. (ack) sender가 receiver 한테 메세지 보낸 후에 acknowledement 메세지 받는것. 메세지를 받았다는 것을 알게된다. sender가 receiver에게 메세지 보내고 일정시간 기다렸는데 메세지가 안오면 유실되었다고 보고 똑같은 메세지를 다시 보낼 수 있다. 그렇게하면 메세지가 lost가 되어도 다시 보내는것. 이렇게 하면 문제가 sender가 메세지 보내고 일정시간 기다렸는데 메세지가 안왔다. 그래서 메세지 다시 보냄(같은) 그런데 알고보니 sender가 receiver로부터 ack를 못받은 이유는 original 메세지가 유실된게 아니라 ack가 유실되었던 것. 그때 sender입장에서는 ack이 안오니까 (유실) . 일정시간 기다렸는데 안와서 똑같은 메세지를 또보냄 . 이런 경우에는 sender가 똑같은 메세지를 또보내니 receiver는 같은메세지를 또 받는다. receiver는 같은메세지를 두번 받았기때문에 하나는 버려야한다. 이것을 하려면 메세지마다 고유의 넘버. sequence number를 붙여야한다. 첫메세지는 1번. 두번째는 2번. 이렇게 넘버를 붙여서 메세지를 보낸다. 301번 메세지를 보냈는데 일정시간 후 ack가 안오면 일정시간 후 sender가 301번 메세지를 다시 보내긴 하겠지만 receiver입장에서는 301번 메세지를 받았던 거라서 두번째 301번이 오면 버린다 (참고로만 알기) 나머지도 참고로만 ->how process are named 네트워크 저편에 있는 상대방 프로세스를 어떻게 naming할것인가 .뭐라고 읽을것인가.
authentication . 네트워크 저편에 있는 상대방이 내가 생각하는 상대방이 맞는지. 지금 이말을 한 상대방은 누구냐. 이것을 밝히는게 authentication. 상대방이랑 내가 채팅을 한다고 했을 때 authentication 자체의 concept은 내가 상대방이랑 채팅을하는데 상대방이 정말 영철이가 맞는지. 이것을 파악하는게 authentication. performance issue when sender 와 receiver가 같은 머신. 같은머신에서 수행되면 네트워크 타고 메세지 가는건 아니고 같은컴퓨터로 가는거라서 performance 높이는 방법 제공되면 좋겠다.

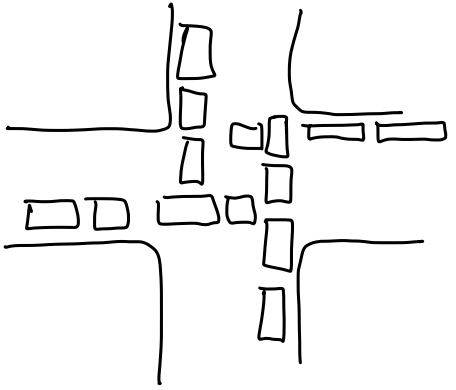
message passing이용해서 producer-consumer problem 해결한 code.
#define N 100 버퍼의 슬롯의 개수. producer process가 1번 머신에서 수행 . 네트워크 건너편 2번 컴퓨터에서는 consumer process가 아래 코드를 수행한다. 어떻게 작동하는가 하면. consumer의 for 루프부터 수행 .N 번동안 루프를 돌면서 send(producer,&m) 프로듀서 프로세스에게 empty 메세지를 N개(100개)를 보낸다. while루프로 넘어간다. producer는 처음에 while (True) 여기서 item=produce_item() 아이템 하나 생산한다. receive(consumer,&m) consumer process로부터 빈 메세지를 받는다. 하나 받아서 build_message(&m,item) 빈 메세지에다가 produce(생산한) 아이템을 넣는다. send(consumer,&m) 꽉찬 메세지를 consumer process에게 보낸다. consumer 입장에서는 receive(producer,&m) producer process로부터 차있는 메세지를 받는다. item=extract_item(&m) item을 뽑는다. send(producer,&m) producer에게 빈메세지를 보낸다. 그리고 consume_item(item) 방금 뽑아낸 아이템 소비. 이렇게 하면 producer -consumer problem 구현한 것. synchronization 같은 이슈도 여기서 해결.

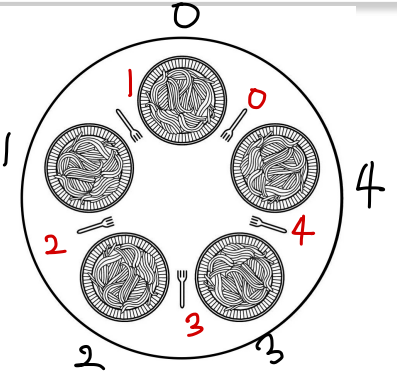
예를 들어서 , producer가 아직 메세지를 build _message에 message채워가지고 꽉찬 메세지 아직 안보내면 consumer가 while루프 처음 들어가면 receive한다. receive system call해봤자 producer가 메세지 아직 안보낸 상태라면 receive call했을때 block이 된다. (receive 알고리즘에 의해). synchronization이 되고있다 .진행안하고. block이 되게 되어있다. 메세지가 producer로부터 안온다. 동기화가 잘 되었다. 그다음에, 마찬가지로 producer도 receive(consumer)이 코드에서 consumer로부터 메세지가 안오면 block이 될수밖에 없다. 그러면 producer-consumer problem 컨셉이. 생산자(producer)는 아이템 계속 생산하고 consumer(소비자)는 아이템을 계속 소비하는 개념. 공유 버퍼가 있어 N개의 고정된 개수의 슬롯이 있다. 고정된 크기의 버퍼 . producer는 N개보다 더 많은 item 생산 못함. 버퍼가 꽉 차게 되니까 producer가 더 진행을 못하게 하는게 컨셉. consumer도 버퍼가 텅 비어있으면 더이상 진행 못하게 하는게 producer-consumer problem의 컨셉. 그게 여기서 구현. producer로부터 메세지가 하나도 안오면(build_message했는데도) 첫번째 build message에 message를 채워서 보내지도 않은 경우는 consumer가 receive에서 block. 진행을 못한다. 그런부분이 바로 synchronization이 잘 구현된것. message passing을 이용해서 해결하는 것을 보여준다.

Barrier (s)
barrier의 보통 여러 프로세스가 동기화를 할 때 보통 사용. 시간이 오른쪽으로 흐르고 있는데 그림 a에 보면 프로세스 a,b,c,d 가 있다. 시간이 흐르면서 수행. 그래서 시간이 좀 흘러 그림 b를 보면 a,b,d 는 barrier primitive를 call하게 된다. 그런데 barrier primitive라는 것은 특징은 모든 a,b,c,d 그러니까 모든 프로세스가 barrier primitive를 call해야지 barrier primitive로부터 return 하면서 다음으로 넘어갈 수가 있다, 그렇지 않고 하나가 call하지 않으면 먼저 barrier primitive를 call한 a,b,d 얘네들은 block이 된다. 그림 (b) 경우가 그런것임 . c가 아직 barrier primitive call하지 않음. a,b,d만 call한상태. a,b,d는 barrier primitive call하고 return 못하고 block. 나중에 시간 흘러서 c가 barrier primitive call하면 그림 (c) a,b,c,d 가barrier primitive 로부터 return하면서 (block풀리면서) 다음으로 진행할 수 있다. synchronization 방법으로 사용되는 barrier라고 하는 primitive. 이런것들이 실제 문제를 풀 때 사용된다.간단하게 책에서는 example 얘기. 참고로만 알고있자. -> 병렬적으로 행렬의 원소값 계산하지만 자기의 계산이 끝났다고 해서 그다음 next 1초 계산 못한다. 다른 원소의 값이 계산 끝나야 해서 .이 때 barrier 사용할 수 있다. 자기의 온도 계산하고 barrier system은 barrier primitive를 call 하게 코딩. barrier primitive call했을때 다른 원소 값 계산 안했으면 진행 못하고 block. 마지막까지 계산 끝나면 next 계산하러 간다 (진짜 참고만 하기)

Dining Philosophers 식사하는 철학자. 다익스트라(세마포어 개발한사람)가 1965년에 제시한 synchronization problem. problem을 다익스트라가 제시하고 자신의 알고리즘 가지고 해결. 자기의 알고리즘이 synchronization 이슈를 잘 해결했다. 이렇게 한것임. 다른 과학자들도 자기의 알고리즘을 개발했을 때 dining philosopher을 해결함으로써 내 알고리즘이 mutual exclusion과 synchronization을 해결한다. 이렇게 자랑했다.

철학자가 5명이 있다. 그리고 접시가 5개가 있다. 철학자들이 하는 일은 먹거나 / 생각하거나. 먹다가 생각하다가 이런다. 먹을때는 포크를 반드시 두개 사용 한번에 포크를 잡을 때 한포크씩 잡는다. 어떻게 deadlock을 방지할 수 있는가.deadlock이라는 issue는 . 현실같은 곳에서 deadlock은





서로 다 막고있어서 아무도 진행 못함 . 위의 그림이 현실 세계에서의 deadlock. 운영체제에서 deadlock은 a프로세스가 b프로세스가 가진 자원을 기다리고있고 b프로세스는 c프로세스가 가진 자원을 기다림. c프로세스는 다시 a프로세스가 가진 자원 기다리면서 대기하고 있을 경우 아무도 진행 못함. 이런 경우를 운영체제에서 deadlock이라고 한다. dining philosopher에서 deadlock 발생가능. (simulation하다보면) 그것을 막을 줄알아야한다. dining philosopher 해결하는 솔루션 -> 첫번째 코드
첫번째 solution은 deadlock을 유발한다. philosopher가 N이 일단 5임. #define N 5. philosopher가 5명. 각 philosopher는 argument가 i임. philosopher 프로세스가 5개가 돌아가는 것임. 그래서 simulation 하는것. philosopher process 0은 philosopher function call 할때 philosopher(0)을 call한다. 그런식으로 5개philosopher 프로세스가 돌고있다. dining philosopher를 simulation 하는 상황. 이 알고리즘 쓰면 deadlock 발생. while 루프에서 일단. 철학자들의 넘버를 구분해서 생각한다. 그다음에 fork가 등장하는데 위 그림에 표시함. 각 philosopher는 while루프에 들어가서 think(); . take_fork(i); philosopher가 3이라고 치면 philosopher 3을 call해서 루프를 돌고있다. 그러면 think생각하다가 take_fork(3)왼쪽 포크. 선택해서. 그다음에 take_fork(i+1 이면 take_fork(4)로 오른쪽 포크를 잡고 eat(); 둘다 잡아서 먹고 put_fork로 왼쪽 포크 놓고. 오른쪽 포크를 놓는다.

3 philosopher example

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

down 3 4 5 statement 3 up 4

이경우,deadlock이 발생한다. 어떤 경우 -> 동시에 시작을 할 때, while루프에 동시에 들어가서 think 똑같이 하고.왼쪽포크 3 잡았는데 오른쪽 포크 4를 잡으려고 하는데 이미 4 프로세스가 잡고있다. 4번 프로세스는 또 0번 포크를 잡고싶은데 0번 프로세스가 이미 잡고있다. 쭉 돌다가 마지막 프로세스는 처음 프로세스의 포크를 기다린다. -> 서로 프로세스가 상대방 자원을 기다리는데 결과적으로 갖지 못하고 기다린다. 원형을 이룬다. circle을 이룬다. deadlock을 발생시킨다.

2nd solution
왼쪽 포크를 잡고, 오른쪽 포크를 바로 잡지 말고 오른쪽 포크가 available 한지 체크한다. available 하지 않으면 왼쪽포크마저 다시 그냥 테이블에 놓는다. wait for some time(특정시간만큼 기다린다) 0.1초라고 가정 그다음에 다시 repeats the whole process ->다시 왼쪽 포크를 잡는다. 반복.
->deadlock은 생기지 않겠지만. 즉 ,deadlock은 상대방의 자원을 계속 기다리면서 끊임없이 기다리는 것. 이거는 끊임없이 기다리는 deadlock은 아니고 뭔가는 일을 한다. 진전이 없다 ->starvation. 동시에 시작했다고 가정하고 각 philosopher입장에서 왼쪽을 잡고 오른쪽 포크를 봤는데 available 하지 않는다. 그러면 다시 왼쪽포크를 놓는다. 그러면 0.1초 기다리면 똑같이 또 0.1초 기다리고 또 왼쪽포크 잡고 오른쪽 포크 available한지 보는데 아니라서 또 왼쪽 포크 놓고 또 0.1초 기다리고.. 똑같이 시작했으면 똑같이 될거고 뭔가 열심히 일을 하는데 result는 없다. 이게 starvation.

3rd solution ''
똑같이 0.1초 기다리니까 문제가 아니냐. random time을 각자 기다리게 하자. -> 다른 알고리즘에 쓰이고 있는데(예를들어 이더넷. 이더넷 local area network(lan)에서 이런 computer들이 lan에 몰려있을 때 배선실로 보낸다. 메세지가 거의 동시에 두개의 컴퓨터가 같은 lan에 물려있는 두대의 컴퓨터가 보내면 메세지충돌발생 그때 각 컴퓨터의 프로세스는 랜덤 타임 기다린다음에 다시 보냄. 이더넷 프로토콜에 있음. 실제로 사용하는 알고리즘. 메세지보낼 때는 심각한 상황이 아니라 써도되는데 중요한 상황에서 (원자력) 쓰면 오류날수도.. random time이 우연히 똑같으면 ..? 큰 문제가 발생가능. 3번 알고리즘은 재수없을 때는 문제 -> 완벽한 알고리즘 아님
4th solution 이제 첫번째에서 5 statement의 앞과 뒤에 세마포어를 둔다. 즉, down 세마포어 랑 put_fork 맨 마지막 다음에 up 세마포어. binary semaphore이니까 이름을 b라고 하면 , 처음에 초기화를 1로. 이렇게 하면 down b , up b로 5 개의 statement를 감싼다. 오직 한개의 프로세스만 critical region에 들어갈 수 있다(5개의 statement 속으로) 한 철학자 프로세스가 5개의 statement 수행하는 동안에는 나중의 4개 철학자 프로세스는 이 안에 못들어온다. 문제를 해결하긴 함 . 그러나 사용하기에는 아쉬움 ->performance 가 느리다.. 한번에 한 철학자밖에 못먹음 .잘 보면 두명이 동시에 먹어도 된다. 0번 철학자랑 2번 철학자 동시에 먹어도 됨. 퍼포먼스가 안좋다 . 느림.

```
#define N                   5                       /* number of philosophers */
#define LEFT                (i+N−1)%N               /* number of i's left neighbor */
#define RIGHT               (i+1)%N                 /* number of i's right neighbor */
#define THINKING            0                       /* philosopher is thinking */
#define HUNGRY              1                       /* philosopher is trying to get forks */
#define EATING              2                       /* philosopher is eating */
typedef int semaphore;                              /* semaphores are a special kind of int */
int state[N];                                       /* array to keep track of everyone's state */
semaphore mutex = 1;                                /* mutual exclusion for critical regions */
semaphore s[N];                                     /* one semaphore per philosopher */

void philosopher(int i)                             /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                                  /* repeat forever */
        think( );                                   /* philosopher is thinking */
        take_forks(i);                              /* acquire two forks or block */
        eat( );                                     /* yum-yum, spaghetti */
        put_forks(i);                               /* put both forks back on table */
    }
}

void take_forks(int i)                              /* i: philosopher number, from 0 to N−1 */
{
    down(&mutex);                                   /* enter critical region */
    state[i] = HUNGRY;                              /* record fact that philosopher i is hungry */
    test(i);                                        /* try to acquire 2 forks */
    up(&mutex);                                     /* exit critical region */
    down(&s[i]);                                    /* block if forks were not acquired */
}

void put_forks(i)                                   /* i: philosopher number, from 0 to N−1 */
{
    down(&mutex);                                   /* enter critical region */
    state[i] = THINKING;                            /* philosopher has finished eating */
    test(LEFT);                                     /* see if left neighbor can now eat */
    test(RIGHT);                                    /* see if right neighbor can now eat */
    up(&mutex);                                     /* exit critical region */
}

void test(i)                                        /* i: philosopher number, from 0 to N−1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

maximum 최대 2명의 철학자 먹을 수 있다. void philosopher(int n) . argument 가 있는데 philosopher process 0 부터 4까지 5개가있다. 각 프로세스는 philosopher을 call할 때 process number 을 주면된다. 안으로 들어가서 보면 while(true) . think() 각 philosopher process는 while루프 돌면서 생각하고 포크들을 잡고 먹고 포크를 놓는다. 함수들은 아래 구현되어 있다.
예를 들어 철학자 2번을 가지고 본다. void philosopher(2)를 call한다.
맨 위의 코드를 보면 N은 전체 철학자의 수. 명수. LEFT는 이제 i값을 2로 한다고 하면 LEFT가 1이고 RIGHT가 3이된다. THINKING은 philosopher의 상태. 0 HUNGRY는 1 EATING은 2. int state[N]; 은 공유 변수인데 배열 . 각 철학자의 상태를 나타내서 원소의 개수가 N. 각 철학자의 상태는 thingikng eating hungry 중에 하나. semaphore가 두개. mutex가 1로 세팅. binary semaphore로 사용 . semaphore s[N]. 세마포어 s라는 배열 요소의 개수가 N(5)개. 각 원소가 각 philosopher 당 하나의 세마포어 인 것. 이경우에는 s[5]. 5개 원소 가진 배열이고 이것 자체가 세마포어. s배열의 각 원소는 각 철학자 당 하나씩 있다. 하나씩의 semaphore가 주어진다. i== 2라고 가정. think(); take_fork(2);
그러면 down(&mutex); mutex는 현재 1이었음. 그러니까 down하면 1->0으로 감소 후 진행. state[2]= hungry. 철학자 2는 배고프다고 자기의 상태 세팅. 그다음에 test(2); test를 call. test에서 if문을 보면
state[i]==hungry 이것은 참. 위에서 세팅. true
state[LEFT]!=EATING 왼쪽은 state[1]. 1번철학자도 EATING이 아니고 (초기화로 생각. state가 0이면 thinking). eating이 아니다. true. 뒤에도 그래서 true).
그럼 if문 안으로 들어가서 state[i]=EATING . state[2]=EATING. 2번철학자가 먹고있다. up(&s[i]); s는 처음에 0으로 다 초기화. 각 철학자가 하나씩 원소가 mapping되는건데 세마포어 배열인데 처음에 0으로 초기화. up(&s[2]). 하면 s[2]가 0일테니까 up 했는데 세마포어 값이 0이면 하나 증가해서 s[2]의 값이 0에서 1로 바뀌고 test.에서 return. 그다음 tast_forks에서 up(&mutex)함.쫌전에 down에서 1에서 0으로 바꿨으니까 0에서 1로 바뀐다. down(&s[2]); s[2]를 1로 바꿔놔서 1을 0으로 바꿈. take_fork에서 return 하고, eat(); 맛있게 먹는다. 먹고 있는 동안에 다른 철학자인 3번 철학자가 가동했다고 봤을 때 걔는 take_fork(3); 으로 들어간다.

```
#define N                   5                   /* number of philosophers */
#define LEFT                (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT               (i+1)%N             /* number of i's right neighbor */
#define THINKING            0                   /* philosopher is thinking */
#define HUNGRY              1                   /* philosopher is trying to get forks */
#define EATING              2                   /* philosopher is eating */
typedef int semaphore;                          /* semaphores are a special kind of int */
int state[N];                                   /* array to keep track of everyone's state */
semaphore mutex = 1;                            /* mutual exclusion for critical regions */
semaphore s[N];                                 /* one semaphore per philosopher */

void philosopher(int i)                         /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {                              /* repeat forever */
        think( );                               /* philosopher is thinking */
        take_forks(i);                          /* acquire two forks or block */
        eat( );                                 /* yum-yum, spaghetti */
        put_forks(i);                           /* put both forks back on table */
    }
}

void take_forks(int i)                          /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                               /* enter critical region */
    state[i] = HUNGRY;                          /* record fact that philosopher i is hungry */
    test(i);                                    /* try to acquire 2 forks */
    up(&mutex);                                 /* exit critical region */
    down(&s[i]);                                /* block if forks were not acquired */
}

void put_forks(i)                               /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                               /* enter critical region */
    state[i] = THINKING;                        /* philosopher has finished eating */
    test(LEFT);                                 /* see if left neighbor can now eat */
    test(RIGHT);                                /* see if right neighbor can now eat */
    up(&mutex);                                 /* exit critical region */
}

void test(i)                                    /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

take_fork(3) 에서 mutex가 1이므로 down 에서 1->0으로 바꾸고. return 하고 state[3]=HUNGRY. test(3)을 call.
아래 test if문 조건 체크.
state[3]=HUNGRY 는 맞다. 그런데 state[2]!= EATING eating이 아니어야 하는데 eating이다. 2번이 먹고있다. 그래가지고 false. if문속으로 못들어가서 state[3]=EATING으로 못바꿈. hungry.임. up(&s[i]) 도 못수행. 그런 상태로 return .take_fork로 다시 간다 up(&mutex) mutex를 0에서 1로 올리고 down(&s[3]) s[3]은 아직 0이라서 down했는데 세마포어 값이 0이면 잠든다. block이 된다. 마찬가지로 프로세스 3이 block. 프로세스 1도 block(2번이 먹고있는 동안). 철학자 2번이 eat을 끝내고 put_fork(2)를 하면 오른쪽 왼쪽의 철학자 3번의 1번 (down s) 해서 잠들어 있던 애들을 깨워주게 된다.

철학자가 먹고 사색하고 이러면서 보냄. 먹을 때 두개의 포크 사용. 포크를 처음에 잡을 때 한번에 하나의 포크 잡는다. 두개의 포크 동시에 잡지는 않는다. 그때 이러한 상황을 deadlock 없이 어떻게 해결 -> mutual exclusion. synchronization해결하는데 사용 dining philosopher 문제를 자신의 알고리즘으로 풀면서 증명하곤 함. 좀더 현실적으로 만들자면 스파게티를 짜장면으로 바꾸고 포크를 젓가락으로 바꾼다. 젓가락은 한 쌍이 있어야 짜장을 먹으니까. 4가지 solution은 좀 엉터리. 4번째 solution은 어느 순간에 한명의 철학자밖에 못먹음. 사실 두명이 먹을수있는데. (포크들이 겹치지 않음) maximum 2명까지 먹게 해주는 알고리즘이 왼쪽 알고리즘.

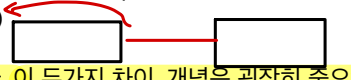
N 은 5. 철학자가 총 5명. 5명의 철학자 simulation하면서 5개의 process가 사용. process 0부터 4까지. 맨 밑에 있는 philosopher라는 function이 각 process가 수행하는 function. philosopher process 0은 philosopher(0) 으로 call. 파라미터 i에 0이 넘어옴. 그런식으로 5개의 프로세스가 돌고 있다. LEFT. 철학자의 왼쪽 번호는 결국 i-1이다. 철학자 2번이면 LEFT는 1이고 RIGHT는 3이다. right는 i+1과같다.
THIKING ,HUNGRY,EATING 은 철학자의 상태. state값 0,1,2 . int state[N] 프로세스 5개가 공유하는 공유 변수. 배열. 각각의 element(원소)는 철학자 i의 상태를 나타낸다. 이때 상태값은 위에 있는 3개중의 하나가 될 수 있다. semaphore mutex, s[N]. mutex 세마포어는 1로 초기화. binary semaphore . mutual exlusion 제공. s[N] .배열의 형태. 철학자당 하나 씩 할당되어있는 세마포어.

philosopher 2를 예를 들어 설명. while루프로 들어가서, think(); 일단 생각. take_forks(2) 를 call한다. 그럼 이제 take_forks 코드로 가서 (i값에 2가 넘어옴) down(&mutex); down(&mutex)는 up(&mutex)와 쌍임. 그 사이에 두개의 statement가 있는데 두개의 statement가 critical region이기 때문에 위 아래를 binary semaphore 로 감쌈. mutual exclusion 제공하기 위해서. down(&mutex). mutex는 1로 세팅인데 down 알고리즘에 의해 1->0 으로 감소시키고 진행. state[2]==HUNGRY. 포크를 잡으려고 한다는 것. 그다음에 test(2)가 호출. test(2)로 와서 if(state[2]==HUNGRY 는 맞다. 방금 위에서 세팅함. state[LEFT] . i 값이 2인데 left는 i-1 하면 1이다. state[1]은 초기에는 thinking이기 때문에 . state[N] 의 초기값은 0. THINKING의 상수값은 0이다. right은 i가 2니까 3이다. 3번 철학자도 초기이니까 thinking이다. 결국 이것도 참. state[2]=EATING. 2번 철학자는 EATING상태로 바꿈. up(&s[2]). s는 세마포어 배열이고 각각의 원소는 철학자 한명에 대응. 철학자 프로세스마다 s배열의 한 원소에 대응. 각각의 원소는 세마포어가 된다. 처음에 세마포어 s는 0으로 초기화. up(&s[2]) 는 s[2]는 0이니까 up 알고리즘은 값을 증가시켜 0->1로 up을한다. 물론 자고있는 프로세스가 있으면 깨워주지만 자고있는 프로세스가 없기때문에 값을 증가시킨다. up 알고리즘 . s[2]의 값이 0->1이 된다. up(&mutex)를 call해서 mutex를 0에서 1로 올린다. down(&s[2]) . 좀전에 s[2]의 값을 0->1로 올려놨는데 down operation에 의해 1->0으로 감소한다. 그다음에 return. take_fork 자체로부터 return. 그다음에 eat(); 스파게티를 먹고있는 것.

왼쪽에 있는 프로세스 1과 3은 ,현재 2가 먹고있는데 먹으려고 하면 어떻게 될까?
process 1은 philosopher(1)을 call한다. whie(true)에서 think한 후 take_fork(1) 을 call한다. take_fork(1). 에서 mutex를 down operation 에 의해 1->0으로 내리고 state[1]==HUNGRY. test(1)을 call한다. if(state[1]==HUNGRY 이건 참. state[LEFT] left는 0이된다.철학자 0은 초기상태여서 thinking이므로 참으로볼수있는데 state[right]!=eating. right는 2인데 2번 철학자는 EATING. 마지막은 거짓이다. if의 조건이 거짓이다. 그 안의 두개의 statement 수행 안하고 return. 그다음에 up(&mutex) . mutex는 0으로 바뀌었으니 다시 1로 올린다. down(&s[1]) 을 call. s[1]은 초기값인 0이다. if 조건이 거짓인 바람에 그대로 초기값인 0이다. down 알고리즘에 의해 sleep이된다. blocked상태로 들어간다. down(&s[1])을 call하면 s1이 0이기 때문에 sleep상태로 간다. 마찬가지로 프로세스 3도 sleep으로 간다.

프로세스2는 맛있게 먹고있는데 다 먹고나면, eat() 중이었는데 put_forks(2)를 부른다.
down(&mutex) mutex를 1->0으로 바꾸고 state[2]=THINKING. 다시 생각을 하겠다. test(LEFT)와 test(RIGHT)을 차례로 부른다. test(LEFT) 하면 i가 2니까 left는 1이다. 즉 왼쪽 철학자 . 철학자 1번을 말하는것. 그러면 test(1)로 들어가서 state[1]==HUNGRY. 아까 hungry로 세팅. state[left]!=eating left는 0이다. 0은 eating이 아니다(참) 초기이니까 thinking. state[right] right은 2이다. 참. 철학자 2는 다 먹고 THINKING 으로 만들었기 때문에. 참이다. if의 조건 3가지가 참이라서 state[1]=EATING으로 만들고 up(&s[1])을 부른다. 철학자 1번은 down(&s[1]) 을 해서 sleep하고 있었는데 철학자 2번 프로세스가 test(1) call해서 up(&s[1])을 부른다. up(&s[1]) 에 의해 s[1]에 대해 잠자고 있던 프로세스인 철학자 1을 깨운다. up 알고리즘은 하나이상의 프로세스가 해당 세마포어에 대해 자고있으면 awake(깨워준다) 철학자 프로세스 1은 깨어나게 된다. down에서 깨어났으니까 철학자 1은 take_fork(1)로부터 return한다. 그러고 걔는 eat(); 로 들어가게 된다. 맛있게 먹기 시작. 마찬가지로 프로세스 3도 프로세스1로 down(&s[3])에서 put to sleep에 처했는데 결과적으로는 프로세스 2번이 test(RIGHT)을 call하는 바람에 down(&s[3])에서 sleep 되어있다가 풀려난다(awake). blocked에서 ready로 가게된다. right는 3이다. blocked 상태에서 풀린다. 결과적으로 프로세스 1,3(철학자 1,3)은 eating으로 들어간다. 실제상황에서는 최대 2명까지 먹을수있어야하므로 그게 잘 구현된 것임.

Scheduling
process scheduling을 말함. 옛날 batch system. timesharing system. 그당시 오래된 machine이 있을 때는 cpu time이 희소 자원이었다. scarce resource.
scheduling알고리즘 -> 다음에는 어느 프로세스를 선택해서 실행시킬가 를 결정하는 스케줄링 알고리즘이 효율적이도록 노력을 많이 했다.
personal computer 시대 도래.
초기에는 pc에서 프로세스를 별로 동시에 여러개 안불렀다. 스케줄링이 중요하게 부각되지 않음. pc 세계에서는 . 동시의 여러 프로세스가 돌지 않아 스케줄링 중요시되지 않음.
high-end networed workstation and server
여러개의 프로세스 동시에 돌리기 때문에 process scheduling 굉장히 중요.
process switching is expensive.
프로세스 스케줄링을 하게 되면 a프로세스가 수행하다가 b 프로세스가 수행. 스케줄러가 b를 선택해서 a대신 b를 돌린다 cpu사용해서. a -> b context switch 또는 process switch. a가 running상태에서 cpu 차지해 돌고있다가 스케줄러에 의해 b 선택. b가 running상태가 되어 cpu 차지한다. 프로세스 a에서 b로 바꾸는 것을 context switch. 라고 한다. 운영체제에서 expensive하다는 의미는 a프로세스 수행하다 도중에 b프로세스 수행하기 위해서는 a프로세스의 상태를 저장해야 한다. 이어서 수행해야하니까 a프로세스의 상태(레지스터 값, 메모리 맵 정보. address space의 주소) 같은걸 저장해야 수행하던 바로 그 지점 바로 다음부터 수행할 수 있다.b 프로세스를 running으로 올려야함. b도 어딘가에 save되어 있을것. 그상태를 loading해야함. 레지스터 등의 값을 loading 결과적으로 시간이 걸린다. 기존의 프로세스를 상태 저장해야하고 스케줄러에 의해 선택된 b가 running상태 가 되기 위해서는 과거 저장된 상태를 loading해야함. 시간이 걸리기 때문에 context switch ==process switch는 시간이 많이 걸린다. expensive
process behavior (행동 양태)
process behavior에 따라 a타입과 b타입으로 나뉨.
a타입 : cpu-bound process(compute- bound process)
b타입 : I/O bound process
cpu-bound process는 cpu burst 부분이 길다. 오른쪽으로 갈수록 시간이 흐르는데(그림에서), 아주 얇은 직사각형이 바로 cpu burst이다. cpu를 연속적으로 사용하는 부분이다. cpu - bound process는 cpu burst가 길다. I/O-bound process는 cpu burst 가 짧다. 선만 있는 것(박스와 박스 사이의 줄)은 cpu를 사용하지 않고 I/O를 요구하고 기다리고 있는것. 즉 blocked상태로 가서 기다리고 있는 부분.



cpu-bound process와 i/o bound process의 차이는 cpu-bound는 cpu burst가 길고 i/o-bound process는 cpu burst가 짧다. 이 두가지 차이, 개념은 굉장히 중요하다.
b 타입의 I/O-bound process는 I/O 가 길다고 착각하는데 그건 아니고 cpu burst 가 짧은 프로세스. 선 부분은 I/O를 요청하고 기다리는. blocked 되어있는 시간.

10/1 운영체제
when to schedule
process 가 create 되었을 때, exit 할때 block 할때 iterrupt가 발생할때 등 4종류. 4가지 상황에서 스케줄링이 발생
첫번째는 프로세스가 create될때. 프로세스는 보통 folk시스템 콜을 함으로써 프로세스가 하나 생겨난다.어떤 프로세스가 folk system call 하면 child process 가 하나 만들어진다. folk system call 한 프로세스는 parent. 새로 생겨난 프로세스는 child. 새롭게 child가 생겨나면 누구를 running으로 놓고 쓸까 ? parent or child . 요때 스케줄링 dicision을 내려야 한다. 두번째 상황은. process가 exit할때. 프로세스가 이제 완전히 종료하게 되면 running상태에 있는 프로세스는 종료했으니까 다른 상태에 있는 프로세스를 running으로 해야함. 보통은 ready 에 있는 프로세스 중 하나를 running에 넣는다.ready 상태에 있는 프로세스들이 줄서있으면 맨 앞에있는 얘 running상태로. 또 스케줄링 발생하는 3번째 경우는 프로세스가 block될 때. 프로세스가 i/o 요청하면 blocked상태로 가는데running상태에 있는 프로세스가 내려갔으니까 누군가는 또 running에 가야함. 그러면 이때 스케줄링이 일어나서 ready상태에 있던 놈중 하나를 running 상태로 놓는 스케줄링이 발생. 그다음에 4번쨔 스케줄링 ->i/o interrupt . 이건 언제일어나나면 과거의 어떤 프로세스가 i/o를 요청했는데 i/o device가 요청된 일을 처리. 요청된 일이 완료되면 , i/o device로부터 cpu쪽으로 interrupt signal이 간다. i/o가 완료되었을 때 i/o interrupt 가 걸린다. cpu는 현재 수행하고 있던 프로세스를 잠시 멈추고, 일시정지하고 interrupt handler 를 수행. interrupt handelr 가 수행되면 i/o 완료된 거의 후속작업을 한다. i/o controller에 있는 읽혀진 데이터를 커널 쪽으로 옮기고. 커널 버퍼로 옮겨진 데이터를 유저 버퍼로 옮긴다. 이렇게되면 결과적으로 처음에 i/o 요청하는 바람에 blocked 상태로 간 프로세스. 걔는 이제 ready 상태로 풀려난다. 걔를 ready 상태로보낸다. 이런식으로 interrupt handler가 수행되는데. 수행 다끝났으면 어느 프로세스를. 또 running상태에 놓을것인가. 아까. interrupt때문에 수행이 일시중단된 프로세스를 다시 running으로 놀것인가 아니면 interrupt로 인해 interrupt handler가 수행되어서 이제 blocked에서 ready로 가게된 이 프로세스를 running까지 할것인가. 아니면 제3의 프로세스를 running으로 할것인가 이런 scheduling decision을 하게된다. 일반적으로 가장 자연스러운 결정은 원래 수행하고 있다가 interrupt를 맞아서 억울하게 일시적으로 중단된 프로세스를 running으로 올리고 수행하는 것이 좋다. 그다음에 하단부에 나오는 스케줄링 알고리즘은 두개의 카테고리로 나눌 수 있다. clock interrupt를 어떻게 처리하느냐에 따라서
nonpreemptive
수행할 프로세스를 선택해서 그것을 수행되게 하는데 언제까지 수행되게 놔두느냐 그 프로세스가 block이 되거나 자발적으로 cpu를 내놓거나 그럴 경우 까지 그냥 주구장창 수행되게 놔둔다.block이 되는 경우는 I/O request했을 때 read system call 한다던가.
preemptive
수행할 프로세스를 선택한다음에 그것을 수행되게 하는데 언제까지 -> 어떤 고정된 최대 시간까지만 수행되는 것을 허용. 10ms 다 하면 어떤 프로세스를 선택해서 얘를 수행하게 놔두는데 최대 10ms 까지만 수행하는 것을 허용. 만약에 10ms 이 되었는데 이 프로세스가 아직도 수행중이다 그러면 running에서 끄집어 내린다. 보통 ready 상태로 보내버린다. 그리고 ready에 있는 애들중 하나를 running으로 보낸다. 이게 preemptive 알고리즘. 두 가지 개념은 중요하다.
스케줄링 알고리즘의 카테고리에 대해서.(참고로)
batch system의 스케줄링 알고리즘은 보통 nonpreemptive,preemptive 한 알고리즘 둘다 사용. 각 프로세스 당 시간을 길게 할당을해준다. preemptive알고리즘 같은 경우에는 maximum of some fixed time해서 최대 허용되는 시간을 준다. 보통 이것을 길게 준다 .batch에서는 . 한번 프로세스를 cpu가 잡으면 batch에서 오래 수행되게 해준다. 그렇게 되면 process switche 가 드문드문 일어나서 성능을 향상시킨다. process switch 자주 발생하지 않아 cpu가 프로세스 스위치에 낭비하는 시간이 줄어들게 된다. cpu에 대부분의 시간이. 실제로 프로세스 수행하는데 사용.

interactive . 수행되는 동안 ' 에 사용자와 intera'ction하는 상황. interactive system에서는 preemptive 알고리즘 사용. 여러개의 프로세스가 돌고있고 각 사용자가 자기의 프로세스를 그중에 돌리고있다. 이때 scheduling 알고리즘은 각 사용자는 자기의 job이 계속 interaction이 원활하게 되는 것을 원한다. 그러기 때문에 preemptive 알고리즘을 써서 일정시간씩만 각 사용자의 process 에 시간을 주고 그걸 다쓰면 이제 다음 사용자의 프로세스에게 cpu를 넘긴다. 그렇게함으로써 돌리는 것. 그렇게되면 결국 한 프로세스가 cpu를 점유하는 것을 막을 수 있다. 프로세스에 할당된 시간을 길지 않게 해야한다. 사용자 a의 프로세스 1. 사용자 b의 프로세스 2 사용자 c의프로세스 3이 있을때 각 프로세스에게 조금씩 시간을 줘서 계속 switching을 한다. 그래야 사용자 입장에서는 자기 차례가 interaction하는데 딜레이가 안생긴다고 느낀다, 차례가 빨리빨리 오니까, 시간을 많이줘버리면 각 사용자의 프로세스에게. 사용자입장에서는 자기 차례가 늦게 들어와서 interaction이 늦어져 답답함을 느낀다. 할당된 시간을 좀 짧게줘야한다(batch system에 비해서
real time 인 경우에는 특징이 (참고) preemption이 어떨때는 필요하지 않다 . 프로세스들은 자신이 오래 수행되지 않을거라는걸 알고있다. 자신의 일을 하고 빨리 block을 한다. 참고로만 알고있기.

scheduling algorithm goal(목표)
모든 시스템은 goal 중의 하나가 fairness -공평함 . 모든 프로세스에게 cpu 공평하게 할당 . policy enforcement 공평함이 목표라면 그것이 잘 지켜지고 있는지. 그것을 지켜보는것 balance- 균형. 시스템의 모든 파트가 바쁘게 유지하는것.
batch system에서는 , 1. throughput은 굉장히 중요!!!!!!!!! Throughput은 단위 시간에 완료된 job의 개수. 예를들어 한시간에 완료된 job의 개수 가 throughput. 중요하다!!!! 이것을 maximize하는게 목표가 되어야한다.. turnaround time도 중요.job이 system에 submit된다음에(제출된 다음) terminate(종료될때) 까지의 시간. job,process가 시스템에 submit 된 시간부터 terminate(종료될때) 걸린시간. minimize 하는게 중요. cpu utilization(cpu이용률) - cpu를 바쁘게하는것 이 목표가 되어야한다. 놀게하지많고
interactive system response time (응답시간) - interactive system은 사용자와 계속 interaction하는 system. job이 수행되는도중에. 응답시간이 빠른게 좋다. 요청에 빨리 반응하는것. proportionality(참고로만.) - 사용자의 기대를 만족시켜주는 목표
real-time system 에서는 meeting deadline -deadline을 충족하는 게 중요하다 . 데이타 잃어버리거나 사고 방지. predictability - 멀티미디어 시스템에서 품질 저하quality degradation을 막는 것.

scheduling algorithm을 하나씩 본다. (필기 시작)
batch system의 scheduling algorithm을 먼저 본다.
first-come first -served (fcfs) 이 말에서 concept이 나와있다. 먼저온놈이 먼저 서브받는다. 서비스를 받는다. 프로세스들은 요청한 순서대로 cpu를 할당받는다. 이렇게 하기 위해서는 프로세스들을 요청한 순서대로 줄을 세워야한다. 그래서 single queue하나의 queue를 만든다. 그리고. 이 알고리즘은 nonpreemptive하다. 그래서 한번 수행을 하면 계속 수행한다. block이 되거나 자발적으로 내놓을때까지. 알고리즘 하나하나가 nonpreemptive한지 preemptive한지 잘 보기. 그리고, process가 수행되고 있다가 i/o 요청해서 block상태로 가면 ready 상태에 있는 프로세스들(queue에있는). 제일앞에있는 first process가 running상태로 가게된다. 이 얘기임. 그리고 blocked 상태로 간 프로세스가 언젠가 i.o가 완료되어서 (interrupt가 걸리고 interrupt handler 수행되고 i/o 완료 후 읽혀들어온 data가 device controller 같은 데 들어오면 interrupt handler 쪽에서 kernel 에서 user 쪽으로 보낸다) 이런게 되면서 block된 프로세스는 이제 수행할 수 있는 상황이라서 ready상태로 가게된다. ready 상태로 보내질 때 구체적으로는 queue의 마지막에다가 붙인다. 이 알고리즘은 이해,프로그래밍 기 쉽다. disadvantage(단점 - 참고로만) 하나의 compute-bound process가 있고 여러개의 i/o bound process가 있다고 가정.

얘네들이 system에서 돌고있는데, compute-bound process는 1초 수행되고 그다음에 disk block 하나를 read 하는 특성 . 그다음에 또 1초 cpu 사용해서 수행되고 그다음에 disk block 하나 읽어오는 i/o 요청을 한다. 이것을 반복하는 process. (cpu-bound proces라고도함). i/o bound process는 여러개가있다. 공통된 특징이 cpu time은 거의 안쓴다. 그러나 I/O를 많이하는데 1000번의 disk read를 끝내야지만 하나의 i/o bound process가 끝난다. 그런것들이 여러개 있다(I/O bound process가 여러개). 이런 상황을 가정한 것 fcfs로 차례로 수행한다고 봤을 때 시간이 흐르고 있을때 cpu바운드 프로세스가 1초를 사용. 그다음에 I/O를 요청 .줄서있던 i/o-bound process가 cpu를 조금씩 조금씩 쓴다. 조금쓰고 i/o요청하고 조금쓰고 i/o요청하고 반복. 그다음에 cpu-bound process가 1초 사용. 결과적으로 이런식으로 수행되면 i/o bound process는 완료되려면 사이사이에 cpu-bound process가 1초씩 끼니까 i/o bound process 하나 입장에서는. 결국은 천번 i/o를 해야한다 그사이에 1초씩 cpu bound process가 껴서 1000초가 걸려야지 i/o bound process 하나가 끝난다. i/o bound process 하나의 입장에서는 1000초가 걸릴것이다. 참고로만 알고있기.

shortest job first. SJF
nonpreemptive 짧은 job을 먼저 수행한다. 실행시간이 작은 프로세스부터 먼저 수행한다. 프로세스들의 실행시간 (runtime이) 미리 알려져 있는 상황을 가정. 여러개의 프로세스 들 중에 짧은 작업(job)부터 차례대로 수행. optimal 한 average turnaround time이 보장된다. turnaround time의 평균이 최적화되어있다. (최소화되어있다) 조건은 모든 job이 동시에 available- > 동시에 system에 도착했을 경우. 예를 들어보면 , average turnaround time 계산.
a가 아무렇게나 스케줄링. b가 SJF 로 스케줄링. A B C D 가 있는데 runtime이 A는 8 B 는 4 C 는 4 D는 4. a그림은 그냥 무식하게 한것. 알파벳순서로 했다고 가정.
각각의 turnaround time은. 계산한다음에 평균을 내보면 b그림은 SJF 알고리즘으로 스케줄링한건데 average turnaround time 계산해서 비교해보자. a그림은 시간이 왼쪽이 0이고 오른쪽으로 갈수록 흘러가는것. 0에 모든 job이 도착. A,B,C,D 가 동시에 시스템에 submit .스케줄링은알파벳순서로. a그림부터 본다. A는 시간 0에 submit되어서 수행하는데 8 걸림. 시간 8에서 끝남. A의 turnaround time은 8-0=8(terminate-submit된 시간) A는 8이다. B는 0에 도착했는데 12에서 끝남. 12-0=12 C는 submit가 0이고 끝난게 16이므로 16-0=16 D는 20-0=20 다 더해서 4로 나오면 14minutes 가 나온다. SJF 로 하면 , 짧은것부터 수행한다. A,B,C,D 의 런타임 순으로 B,C,D,A 순으로 했다. b는 4-0 =4 이다. c는 길이가 4이니까 8에끝난다. 8-0= 8이다. 도착은 0에 다 도착. D는 12-0=12. A는 마지막으로 20-0=20 이다. 4+8+12+20을 4로 나누면 average turnaround time이 나오는데 11minutes 가 나온다. b가 a보다 작다. average turnaround time이 optimal(최소값)으로 나온다. 중요한 것은 모든 job이 동시에 available 해야한다. -> 동시에 도착해야한다. 동시에 도착을 해야 모든 job이 available하니까 각각의 실행시간을 미리 알수있다. 얘내들중에 도착한 애들 가지고 실행시간 짧은것부터 수행한다. 모든job이 동시에 available 해야한다는 것이 기본 가정. 동시에 available 하지 않으면 문제가 발생 ->turnaround time이 optimal하지 않음.
cpu 스케줄링 .(스케줄링을 구체적으로) 스케줄링- 어느 프로세스를 수행할것인가. running상태로 놓고. 그것을 지에 결정하는 알고리즘이 스케줄링 알고리즘이라고 할 수 있다. 스케줄링 알고리즘이 수행할 프로세스를 선택하면 그 프로세스는 running상태로 가게되고 cpu를 차지하게 된다. 스케줄링 알고리즘 보고있는 batch system 스케줄링 알고리즘 보고있었음. fcfs 와 sjf를 다 봄. SJF는 짧은 job 먼저 수행. runtime이 짧은 놈 먼저 수행한다. 이 알고리즘은 특징이 optimal average turnaround time 보장한다. 단 조건이 , 모든 job이 동시에 available할때. 모든 job이 동시에 도착했을 때 가능하다. 보장한다. A,B,C,D 4개의 프로세스가 시간 0에 동시에 도착했다고 봤을 때 A의 런타임은 8, B의 런타임은 4 ,C,D도 모두 4. 이때, SJF 로 스케줄링 하면 그림 b처럼 짧은 것 부터 수행한다. BCDA. BCD 는 동률이라 상관없음 바뀌어도. SJF 하면 optimal average turnaround time 보장. turnaround time이 최솟값이 나온다. 계산했을 때 4로 나눴더니 11분 . 제일 짧은 average turnaround time이 나옴. 추가적으로 얘기. optimal average turnaorund time 보장 위해서는 모든 job이 동시에 available 해야한다. 동시에 availalbe 하지 않는 걸로 따져봄. 5개의 job이 있다고 친다. A,B,C,D,E runtime은 각각 2,4,1,1,1 이다(필기) 도착시간은 (0,0,3,3,3) .5개의 job이 동시에 available 하지 않다. 이것에 주목! 이런상황에서 average turnaround time을 계산해본다. SJF 로 스케줄링을 해본다.

A,B가 먼저 0에 도착했다 '. 시간 0에 A,B가 도착했다. 시간 0에는 두개밖에 available 하지 않는다. SJF 로 한다고 가정하고, A,B 중에 할 ' ' 수밖에 없다. 짧은것부터 해야 하니까 A부터 하고, B를 수행한다. A 는 runtime 2 로 해서 0부터 2까지. B는 런타임이 4라 6에 끝난다(그림) 그래서 A,B는 그렇게 할수밖에 없고 6이 되었을 때 누구를 스케줄링 하느냐 C,D가 나오는데 C,D는 arrival time이 3이었다. 이제 얘네들은 시간 3에 C,D,E가 도착했다. 그런데 A,B가 먼저 스케줄링 되어서 B가 6에 끝났다. 그다음부터는 C,D,E를 스케줄링 할 수 있다. C,D,E가 3에 도착했을 때 결과적으로 얘네들은 B가 끝난시간부터 수행해야 할텐데 크기가 다 똑같다. runtime이 다 111. runtime이 같아서 그냥 CDE순서대로 한다. 이제 average turnaround time을 계산. turnaround time은 어떤 job이 submit된 후 terminate 될때까지 걸리는 시간. arrive 한 시간 A의 경우는 2-0 하면 2이다. B는 0에 도착해서 6에 끝남. 6-0=6이다. C는 도착은 3, 끝나는것 7 . 7-3=4 D는 3에 도착해서 8에 끝났으니 5. E는 3에도착 9에 끝났으니 6 (2+6+4+5+6)/5=23/5
이므로 4.6 정도가 된다. 단 모든 job이 동시에 available 하지 않았다. 모든job이 동시에 available 하지 않은 상태에서 SJF 적용해봤자 optimal 한값이 보장이안된다. SJF로 했을때 average turnaround time이 존재하는지는 보여주기
BCDEA로 스케줄링 했다고 가정. (SJF무시)
시간 왼쪽은 0 . 0에 A,B가 도착. BCDEA로 스케줄링 하는 것을 가정 .B부터 수행. B는 runtime이 4. 4에 끝난다. C,D,E 다 1짜리. E가7에 끝난다. A는 runtime이 2라서 9에 끝난다. avg turnaround time을 계산하면 , 시간 3에 C,D,E가 도착했다는 걸 명심.
B는 0에 도착해서 4에 끝난다. 4 이고 C는 3에 도착해서 5에 끝난다. 5-3=2 이다. D는 3에 도착해서 6에 끝난다. 6-3=3 이다. E는 3에 도착해서 7에 끝난다. 7-3=4
A는 0에 도착해서 9에 끝난다. 9이다
(4+2+3+4+9)/5=22/5 이다 따라서SJF로 했을때보다 average turnaround time이 작다. 즉 ,SJF 하는게 최소를 보장을 못함 -> 그 이유는 동시에 다섯개의 job이 도착한 것이 아니기 때문이다.
Shortest remaining Time Next
방금 SJF의 preemptive version 정도. 한마디로 remaining runtime이 제일 짧은 프로세스를 선택한다. 남아있는 실행시간이 제일 짧은 프로세스를 뽑는다. cpu를 running 상태로 놓고 사용하게 해줌. 구체적으로는 새로운 job이 도착을 하면 그 job에 total runtime을 running 상태로 수행되고 있는 current process의 remaining time.현재 수행중인 프로세스의 잔여 수행시간과 비교한다. 새로운 job의 total runtime이 현재 수행중인 job의 잔여 수행시간보다 더 짧으면 process switch. context switch를 한다. 즉, current process를 running 상태에서 ready 상태로 끌어내리고 새로운 job을 running으로 논다.

Three level scheduling은 cpu scheduler그 정도만 생각. 지금 배우는 스케줄링이 다 cpu 스케줄러가 하는 스케줄링 알고리즘 정도.나머지는 몰라도 됨.
메인 메모리에 있는 ready 상태에 있는 process 들 중에 누구를 이제 뽑아서 수행할것인가. 그게 cpu 스케줄러가 하는 일이다.

interactive system의 스케줄링 알고리즘 소개(좀전까지는 batch system)
첫번째 등장이 round-robin scheduling. round robin이 영어에서 돌아가면서 순서를 주는것이다. 프로세스에게 돌아가면서 순서를 준다. round robin scheduling의 핵심은 quantum.이다. 각 프로세스는 quantum을 할당받는다.quantum이라는 것은 프로세스가 수행되는게 허락된(허용된) time interval이다.10ms, 20ms이렇게 수행하는게 허락된. 허용된 time interval. 그게 바로 quantum이다. 예를 들어서 현재 시스템의 quantum은 10ms라고 가정.round robin 스케줄링 하는데 quantum이 10ms라고 가정. 돌아가면서 수행하는데 먼저 수행되는 프로세스가 10ms까지 수행하는 것이 허용. 그림 a를 보면 BFDGA 프로세스가 ready 상태에 있는프로세스들이 줄서있다. 줄서있는 순서는 프로세스가 처음에 생성될 때 생성된 순서대로 줄의 끝에 달라붙어서 형성되었다고 생각해도 된다.

그러면은이제, 줄을 서있 ' 는데 얘네들한테 맨 앞에있는 애부터 수행을하는것. round robin은. B가 맨앞에있어서 걔가 수행을하고있다.quantum이 10ms라고 했는데 B프로세스가 수행을 하고 있는데 수행 시작한지 10ms 됫는데도 아직도 수행 중이면 할당된 quantum은 끝났으니까 B를 running 상태에서 ready 상태로 끌어내린다. 줄의 끝부분에 붙인다. 그림 b의 상황이된다. 그다음에는 이제 다음에 서있던 F가 선택이 된다.running 상태로. cpu를 차지하게 된다. quantum까지만 수행이 허용이 된다.그래서 이런식으로 수행하는게 round robin scheduling. interactive system에서 아주 많이 사용되는 스케줄링 알고리즘. quantum이 중요한 핵심이다. 여기서 주목할 만한것은 quantum의 길이. quantum의 길이 too short하거나 too long하면 어떤 문제가 발생하느냐. 너무 짧으면 process를 조금 수행하고 바꿔서 다른 프로세스 수행하고. 계속프로세스를 조금 수행하고 다른 프로세스를 수행한다. context switch(process switch)가 너무 자주 발생한다. 이렇게되면 cpu효율이 떨어진다. 내려간다. 결과적으로 cpu는 process switch를 할때마다 시간을 쓴다. expensive.시간을 많이사용한다. 그림에서처럼 b의 프로세스를 running에서 ready로 끌어내리면서 b프로세스의 상태를 process table entry에 저장을해야한다. 그리고 F를 running상태로 놓고 수행하려면 f의 저장되었던 상태를 다시loading해야한다.register라던가 core image. 각각의 segment에 대한 포인터 값,memory map 등등.cpu가 시간을 필요로 함. 이런것들이 process switch마다 일어나는데 quantum이 너무 짧으면 자주 해야함.그러면 정작 cpu가 프로세스 자체를 수행할 때 쓰는 시간에 프로세스 자체에만 시간을 많이쓰지 못하고 process switch에 시간을 낭비한다. cpu효율이낮아진다. too long 일때. 너무 길때는 짧은 interactive request에 대해서 response 가 나빠진다. poor response. 응답시간이 길어진다. 즉, quantum이 길면, process 하나를 굉장히 길게 수행하고 context switch 하고 다음 프로세스 수행하고. efficiency는 올라가는데 줄서있는 프로세스중에 끝으로 갈수록 자기 차례가 올때까지 시간이 많이 걸린다. process 들이 각각 interactive한 프로세스들이어서 사용자와 interaction을 하는 프로세스면은 자신의 프로세스가 사용자중에 자신의 프로세스가 끝에 쪽에있으면 그러한 프로세스는 한마디로 interaction을 하는데 차례가 오기가 굉장히 시간이 많이걸린다. 그래서 따라서 response가 느려질수밖에없다. quantum이 너무 길면 poor response. 각각의 문제가 있다 ,굉장히 중요하다.

interactive system의 두번째

priority scheduling. 말그대로 각 프로세스에게 우선순위를 부여한다. 그다음에 runnable process중에 (ready 상태에 있는 프로세스) 제일 우선순위가 높은 놈이 cpu를 차지하고 running상태로 되는것이바로 priority scheduling이다. 이렇게 할 경우에는 high priority process들이 주구장창 cpu를 차지하고 계속 수행될수 도 있다. 그런것을 해결하는 방법 두가지 제시

첫번째 방법은, 현재 수행중인 프로세스의 priority를 clock이 발생할 때 마다(clock tick) 깍는다. decrease. 낮춘다. 그러다보면 시간이 흐르면 현재 수행중인 프로세스의 priority가 두번째로 우선선위 높은 프로세스보다도 낮아질 수가 있다 ,그러면 context switch를 한다. 이런 컨셉

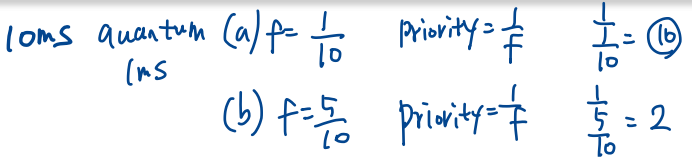
두번째 방법은, 각 프로세스에게 최대 타임 quantum을 할당해서 그때까지만 수행하는 걸 허용한다. 계속 수행하는 것을 막겠다.

priority를 할당하는 방법이 두가지로 나오는데 정적,동적

정적으로 할당하는 경우는 프로세스를 수행하기 전에 priority를 정적으로 할당하는것이다.그러면 무엇을 기준을 priority를 주는가. 예를 들면 사용자가 누구냐에 따라서. 예를 들어 대학같은 환경에서는 사용자가 학생, 교수, 직원 , 총장 등등 사용자가 있는데 사용자에 따라 priority를 할당 한다던가 또다른 정적 priority 할당 방법은 price를 기준으로 . 돈을 기준으로 commercial computing center 같은데서 돈 내고 컴퓨터 쓰는데서 시간당 돈을 많이내는 사람일수록 process의 priority를 높게 준다. nice는 유닉스의 커맨드 중 하나인데 사용자가 자신의 프로그램을 실행할 때 nice커맨드로 실행하면 그 프로그램의 priority 가 낮아짐. 상대방을 배려하는 것. 실제로는 사용이 되지는 않는것같다.

동적(dynamic). priority를 프로세스 수행하는 중간에 동적으로 주는것. 예를 하나 들어보면, I/O bound process인 경우에서비스를 더 좋게한다. priority를 더 높게준다 .이런 방법 이걸 어떻게 구현하냐. priority를 .수행 중인 프로세스 priority를 1/f로 준다. priority가 높을수록 우선순위가 높다고 치자. f가 뭐냐면 프로세스가 가장 최근에(마지막으로) 사용한 quantum의 fraction. quantum이 10ms이면 프로세스 a는 가장 최근에 1ms씀. 프로세스 b는 가장 최근에 quantum을 5ms 쓰고 i,o 요청.그러면 누가 더 i/o bound라고 볼수 있냐면 1ms 만 쓰고 요청한 a가 좀더 i/o bound process이다. 그럼얘한테 priority를 높게 주겠다는 것. 전체적으로 시스템한테 좋게 된다는 철학.

프로세스 a는 10ms quantum인데 1ms가장 최근에 씀. 그다음 i/o 요청.f가 1/10 이다. 그럼 1/f가 priority이니까 1/1/10=10이된다. 프로세스 b는 5ms 썼다. 따라서 f값이 5/10. 1/f는 2가 된다. 프로세슨 a는 priority가 10이고 프로세스 b는 priority가 2이다. 프로세스 a가 우선순위가 높다. 프로세스 a가 i/o bound process일 가능성이 높다(1ms 밖에 안썼으니까.) i/o bound process일 가능성이 높은 a가 cpu-bound process일 가능성이 높을 프로세스 b보다우선순위가 높게 할당받는다 이 공식대로 동적으로 priority를 주면,



그러면은 시스템 전반적으로 효율이 좋아진다 -> i/o bound process에게 priority 높게 줘서 좋은 서비스 제공하느냐. i/o-bound process는 목적이 cpu 오래쓰는게 아니라 i/o 하는데 목적. cpu burst 짧아서 조금밖에 안씀. 얘네들은 빨리 시스템에서 종료시켜서 내보내는게좋다.어차피 cpu쓰는애들이 아니라서. 빨리 cpu가 필요로 할때 우선순위 높여서 해주고 주로 i/o 쓰니까 cpu한테 폐 안끼침. 필요로 할때마다 쓰게 해서 오랬동안 시스템에 남아있게 하지 말고 빨리 쓰고 나가게함. 시스템에서 빨리 종료할수 있게 배려. i/o -bound가 나가면 주로 cpu-bound가 쓰면된다. i/o bound는 빨리빨리 cpu 쓰고 나가니까 좋다. 남아있는 cpu-bound process들은 이제 i/o -bound process가 메모리차지 안하고 있으니까 좀더 시스템이 여유로운 환경에서 cpu-bound process들이 cpu 를 나눠가질 수 있다. (참고로만 알고있기)

priority scheduling 에 있어 priority class를 사용하는 방법 . priority class를 사용하는 방법은 그림과 같다. priority class 가 4개가 있다. priority 4 class 가 가장 높은 class.그래서 클래스 간에는 priority scheduling을한다. 그다음에 각 class 속에서는 round robin scheduling을 한다 결과적으로는 higher priority .더 높은 priority class의 runable process 즉 ready 상태에 있는 프로세스들이 있는 한 그 프로세스 들을 1 quantum 씩 수행을 한다 round robin 방식으로 한다. 그게 point. 그림과 같은 경우에는 priority 4부터 수행. priority class 간에는 priority scheduling을 하니까. priority 4부터 수행을 하는데 그 안에 runnable process가 3개가 메달려있다 . 걔내들을 round robin으로 수행. priority 4의 queue 가 텅 비게 된다. 클래스에 아무 프로세스가 존재하지 않으면 priority class 3에 있는 프로세스들을 round robin 방식으로 1quantum 씩 수행한다.그러다가 priority 3 queue 도 비면 priority 2 class로 이동해서 수행. 맨 마지막에 이방식대로 하다 보면 lower priority에 있는 프로세스 들은 굶어죽을수도 있지 않겠느냐는 얘기. multiple queue 방식은 priority class 쓰는 방식의 응용.
class간에는 priority scheduling을 한다. priority 1부터 4까지. 4가우선순위 제일 높다. 각 클래스 안에서는 round robin schedule한다. 자기보다 높은 higher priority class의 ready 상태의 프로세스가 있는 한 그 프로세스들을 quantum씩. quantum이 10ms이면 10ms씩 round robin으로 돌아가면서 수행. priority process에 a라는 프로세스가 맨앞에 있으면 이제 걔를 quantum을 주고 수행을한다. 10ms 주고 수행 시켰는데 quantum이 지났는데도 a가 여전히 cpu 차지하고 수행중이면 round robin이니까 줄의 끝에다가 가져다 붙인다.

multiple queues 는 priority class사용하는 것을 "' 조금 더 응용했다. 예로 ctss 와 xds 940 소개 ctss에 주목
CTSS(1962) 이 시스템은 스케줄링 알고리즘을 만들 때 두 가지 점에 주목했다. 첫번째 cpu-bound process에게 긴 quantum을 주는게 유리하다. quantum의 길이가 길면 cpu-bound process에게 당연히 효율적. i/o 사용 안하고 cpu 주로 사용하는데 얘한테 quantum을 짧게 해서조금씩 주면 조금 수행하고 context switch 일어나서 종료될 때까지 context switch가 많이 일어나서 cpu 효율이 떨어진다. context switch에 시간이 너무 많이 소요된다. 결과적으로 cpu는 낭비된다. cpu bound process에게 긴 quantum을 주는게 더 효율적이다. 또 하나 주목한 점은 , 그렇지만 large quantum을 모든 프로세스에게 주는것은 response time을 떨어뜨린다. response time이 나빠지는 것을 의미한다. cpu를 차지하기 위해 ready 프로세스들이 줄서있는데 앞에있는 애들이 quantum을 길게 쓰면 뒤쪽에 줄서있는 에는 response가 늦다. cpu를 늦게 차지하게 된다. poor response time이 된다. 이 두가지 점에 주목해서 CTSS 는 디자인되었다.
우선, priority class를 set up 한다. (출발점) 변형을 어떻게 시켰냐 하면 highest class(그림에서는 priority 4 쪽 class) 에 있는 프로세스들은 quantum이 10ms라 치면 one quantum .10ms가짐 그 밑에 next highest class는. 2 quanta를 받는다. (quanta 는 복수형 / 20ms) . 그 다음 클래스의 프로세스들은 4 quanta(40ms)를 받는다 . 2배씩 늘어난다. 프로세스가 자기에게 할당된 quantum을 써버리면 . 그때까지도 cpu를 사용하고 있으면 한 class 강등된다.
cpu-bound process가 100quanta를 필요로 한다. 처음에 맨 위 priority class에 할당. 처음에 1 quanta 사용. 그다음에 계속 수행중이니까 한 class강등. 아래 클래스로 내려가서 2 quanta 쓰고 수행중이면 강등 . 아래 클래스에서 줄서있다가 자기 차례가 되면 cpu를 차지해서 4 quanta 시간만큼 사용. 4quanta 지났는데도 cpu 사용중이면 한 클래스 강등. 8->강등 후 16 사용 -> 강등 후 32 사용. 다 더하면 63 quanta를 사용 37을 더 써야함. 강등되어서 64 quanta 짜리 class로 내려왔는데 64를 할당받겠지만 결과적으로 37 써서 100 채우고 종료. 그러면 context switch가 처음에 initial loading 하는것까지 하면 7번 이 일어난다. 순수한 round robin 알고리즘 쓰면 무조건 1quantum밖에 안줘서 1quantum 쓰고 process switch 일어나고 ->context switch가 100번 일어난다. 따라서 1번,cpu bound process는 긴 quantum을 줘야한다 에 어느정도 부합한다. 그 다음에 프로세스가 priority queue를 계속 강등되면서 밑으로 내려간다. 그러면서 점점 덜 수행된다. 수행될 기회가 줄어든다/ priority가 낮은데로 가있으니까. 높은 애들이 주로 수행될테니까. 그래서 결과적으로는 짧고 interactive한 프로세스들에게 cpu를 양보하는 셈이 된다. 높은 priority 애들은 짧은 interactive process들일 것이다. priority가 높아서 자주 수행되고, 현재 주인공 process는 계속 강등되면서 수행될 기회가 자꾸 줄어준다. 2번 포인트가 모든 프로세스에게 다 긴 quantum을 주면 response time이 나쁠 것이다. 이시스템은 짧은 interactive process . 요런 response time을 빨리 되길 원하는 process일수록 위쪽 큐에 있기 때문에 걔네들 response time이 좋다. 주인공 process처럼 i/o 안하는 cpu-bound process는 아래쪽으로 강등 ,response time에 별로 구애받지 않는애들. 괜찮다. 2번 포인트에도 신경을 쓴 알고리즘. 그러면 short interact process 는 왜 위쪽 priority queue에 남아있을까. 그것은 바로 queue가 있는데 위가 높은 priority. 아래가 낮은 priority. 프로세스들이 줄 서있음. 맨 위에서 수행하고 있다가 처음에 quantum을 1을 씀. 1을 쓰다가 강등. 나중에 자기 차례가 오면 2 쓴다. short interactive process는 i/o bound process 즉 키보드 input 받아들이고 그런애들. quantum 이 되기도 전에 i/o요청을한다. 얘네들은 i/o요청을 하게되면 blocked 상태로 빠져나간다. 큐에 있는 애들은 ready 상태에 있는 애들. i/o요청을 하는 놈은 아직 quantum이 되기도 전에 i/o 요청해서 blocked 상태로 간다. i/o 가 완료되면 ready로 간다. ready로 갈때 원래 떠났던 priority 쪽으로 다시 간다. 줄의 끝에 선다. 나중에 i/o 가 완료 후 ready 상태로 돌아올 때 같은 큐로 돌아온다. 강등되지 않고 . 그게 중요한 포인트. 따라서 short,interactive process같은 경우는 자꾸 quantum 채우기도 전에 키보드 입출력 기다리는 등 요구. blocked로 빠지고 같은 priority에 들어옴. 높은 쪽 priority에 남아있는다 계속. response time이 좋다. CPSS 는 그런것들을 잘 해결. 반면에 priority class 처음 알고리즘은 i/o 요청을 해서 blocked 상태로 갔다. i/o가 왼료되서 돌아오면 같은 priority 로 돌아온다. i/o 요청해서 blocked였다가 다시 돌아올때는 original priority class 알고리즘도 같은 queue로 돌아온다. 그러나 original priority class에서는 quantum이 다 되었을 때 같은 큐에선다. CTSS 는 quantum이 일단 다르다. original priority class 알고리즘은 quantum이 다 되면 그냥 뒤로 가서 붙고(같은 큐의) CTSS에서는 강등. quantum이 다 되었는데 cpu 사용하고 있으면 강등된다. CTSS 굉장히 중요. 프로세스 a가 있다고 치자. 프로세스 a가 수행하려면 10 quantum이 필요. 수행을 priority 맨 위에서 부터 수행 . 1quantum 받는다.(i/o요청은 안함). 계속 쓰고 있으니까 다음 클래스로 내려감. 다음 클래스에서 2quantum 받는다 2quantum 중에 처음 1 quantum 쓰고 i/o 요청하는 상황.

2quantum 중에 1quantum ' 쓰고 i/o 요청.blocked 로 빠짐. 빠졌다가 i/o 완료되면 큐로 돌아오는데 같은 큐로 돌아옴. 두번째 큐에서 빠졌으니 두번째 큐의 끝에 붙음. 줄 서 있다가 자기 차례가 되면 다시 2quantum 가지고 사용. 2quantum이 지나는데도 i.o요청 안하면 밑으로 강등 . 4 quantum을 할당받음. 그러한 메커니즘으로 돌아간다. CTSS 정확히 이해하고 체크해두기.
XDS 940 간단하게만 설명. priority class 사용
4개의 priority class사용 . 맨 위의 priority는 terminal priority. terminal input을 기다리고 있는 프로세스가 깨어났을 때 terminal priority class로 간다. (사용자가 키보드 입력을 했을때가 깨어날 때임). 그다음에 두번째 priority process는 i/o class. disk block을 기다리고 잇는 프로세스가 ready가 될 때. disk block을 다 읽었을 때 ready. 걔네들은 i/o class로 들어감. 그다음에 세번째 priority class는 short quantum. process가 처음에 quantum이 다 되었는데도 아직 running하고 있으면 shor quantum이라는 3번째 class에 들어감. 프로세스가 연속적으로 많이 quantum을 다 써버리면 중간에 맨 마지막 class로 강등 long quantum. 이렇게 한다는 정도만 알고있기.

Shortest Process Next. batch system의 SJF의 interactive system 버전. minimal everage response time 제공. 이런 정도로 알면 된다. 과거에 프로세스의 behavior에 기반해서 다음 실행시간을 추정하는것. 각 프로세스의 다음 실행시간을 추정한 후에, 가장 실행 추정시간이 짧은 놈을 선택해서 수행한다. 그게 바로 shortest process next. 결과적으로는 ,system에 여러개의 로그인한 terminal이 있다고 치자. 각 terminal에서 이제 사용자가 command를 치고 execute하고 command 입력하고 execute 하고 이런것을 반복. 그러면 이때 각 terminal 마다 command가 수행이 될텐데 (계속 반복적으로) 시스템 입장에서는 여러 개 terminal 각각에 수행을 앞두고 있는 프로세스 .걔네들의 실행시간을 추정하는 것. 그다음에 제일짧은 추정시간 가진 프로세스 먼저 수행. 그런데 각 terminal 의 command의 실행시간을 어떻게 추정? terminal에서 전에 수행되었던 process의 command. 실행시간을 바탕으로 해서 추정. 그게 무슨 얘기냐면. Aging이란 알고리즘을 주로 사용하는데 Aging을 이해하면 된다. 이것은 현재 터미널에서 수행되는 terminal의 실행시간을 어떻게 추정하느냐. estimate를 어떻게 하느냐 aT0+(1-a)T1으로. T0는 수행할 커맨드의 실행 추정시간. 그다음에 T1은 그래놓고 이 커맨드를 수행했는데 실제적으로 실행시간. T0와 T1 가지고 다음 실행시간을 추정하는데 공식이 저것이다. 이게 뭔지를 좀더 보면 추정치 estimate 실행시간 runtime이 있다(필기) 처음에 command의 실행시간 추정치가 T0였다. 실제로 실행했더니 T1이 나옴. 그 다음 command의 실행시간 추정치는 aging 알고리즘에 의해서 aT0+(1-a)T1이 된다. 그리고 실행을 했더니 T2가 나왔다. 그 다음 커맨드의 실행시간 추정치는 공식 계속 적용해서 왼쪽에다 a 곱하고 오른쪽에 (1-a)를 곱한다. 전 추정치에 a를 곱하고 전 실행시간에 (1-a)를 곱해서 더하면된다. 따라서 a를 1/2로하면 1/2T0+1/2T1. 전추정치에 1/2를 곱하면 1/4T0+1/4T1+1/2T2가 된다(그림) 수행해보니 T3 가 나옴. 그 다음 추정치는. 왼쪽에 1/2 곱하고 오른쪽에 (1-1/2)=1/2를 곱해서 더하면된다. 1/8T0+1/8T1+1/4T2+1/2T3. ppt 맨 아래는 추정치를 늘여놓은것. a가 1/2이면 T0와 T1이 반반씩 영향을 미친다. 전추정치와 실행시간이 반반씩 영향을 미친다. 1/2보다 작으면 전 추정치는 비교적 적은 영향. 실행시간이 더 많은 영향. 반대로 a가 1/2 보다 크면 전추정치가 더 큰 영향을 미치고 실행시간이 더 작은영향을 미치는 concept. 실제 값을 가지고 계산하면 더 빠름(변수 쓰지않고) 계산하는 방법 굉장히 중요하니까 꼭 이해하고 명심해야한다.

guaranteed scheduling
cpu의 몫을 보장. n개의 프로세스가 돌고있다면 , 각 프로세스는 cpu의 1/n씩 할당받는게 공평. 그것을 보장하는 것. 어떻게 하냐면 , 각 프로세스가 생성된 이후에 cpu를 실제로 얼마나 썼는가를 keep track한다. 어떤 프로세스가 있는데 생성된 이후에 cpu를 10ms썼다. 다른애들이 cpu차지하고 5ms썼다. 또 다른애들이 쓰다가 1ms쓰고 I/O요청. 이런식으로 하면 cpu 쓴 시간들이 있다. 다 keep track해서 현재시점에서 그때까지 cpu 쓴 시간 다 더하면 실제로 생성된 이후에 cpu 사용한 시간. 그것을 말하는것. actual CPU time consumed 분모에는 주어진 CPU time. 즉 공평하게 현재 프로세스가 생긴 이후로 10초가 흘렀는데 5개의 프로세스가 돌고있다 . 그러면은 이 프로세스 하나에 대해서는 다섯개의 프로세스가 10초동안 돌고있었으니까. 각 프로세스는 2초씩 가지는게 공평함. 그게 바로 분모의 CPU time entitled. 나눠서 나오는 ratio. ratio가 0.5다 라고 하면 자기에게 주어진 cpu time의 반밖에 못씀. 1에 가까우면 거의 다 자기에게 주어진 cpu time을 뽑아 쓴 것.0에 가까울수록 잘 못쓴것. ratio를 계산 한 후 가장 작은 ratio 를 우선적으로 수행한다. 그 ratio 가 가장 가까운 경쟁자보다 높아질때 까지. concept 정도 알아두기.

Lottery Scheduling. 복권 스케줄링 정도로 번역. 프로세스에게 복권을 주는것. 다양한 시스템 resource 에 대한 복권. cpu time(예를들면) scheduling decision이 내려져야 할때 그때 복권추첨을 한다. random하게 뽑는다. 복권들은 당첨될 확률이 다 같다.(random이라서) 뽑아진 복권은 . 소유하고 있는 프로세스에게 resource를 준다. 결과적으로는 ticket에 fratcion f를 가지고있는 프로세스. 전체티켓의 f분 가지고 있는 프로세스는 전체 resource 예를들어 cpu에 fraction f. f분을(분모) 갖게 된다. 예를들어 전체 100개의 티켓. 어떤 프로세스가 10장 가지고있으면 결과적으로 cpu의 10/100을 소유하게 될수밖에없다. random하게 추첨하니까. highly responsive하다. system에 갑자기 프로세스가 수행이 되기 시작해도 생성되었을 때 복권을 주는만큼 바로 효과가 있다. 100장중 50장을 줬다 하면 50/100 확률로 cpu를 바로 사용할 수 있다. cooperating process. 협력하는 프로세스 관계는 서로 티켓을 교환할수도있다. 상대방을 도울수도 있다. video server의 예를 들었는데 video server가 몇개의 child process 가지고있다. child process는 client를 상대한다. 그러면 이제 child process 1 , 2,3이있는데 child process1은 프리미엄 고객들. 이런고객들은 해상도 높고 frame rate 높은. 스트리밍서비스. child process1은 cpu를 많이 차지해야 한다. 복권 많이준다. child process 2는 일반 멤버. child process 3은 저렴한 멤버쉽. 해상도도 더 낮고. 아래로 갈수록 process는 cpu를 적게 쓰면 된다. 복권을 적게 줘서 cpu를 적게 차지하게 control 할 수 있다. 프로세스가 자원을 차지하는 것을 control 할 수 있다는 얘기.

fair-share scheduling. 지금까지 스케줄링 알고리즘 여러 개. process의 owner을 고려하지는 않음. 누가 이 프로세스를 돌리고있는지 그것을 고려하지는 않았다. 예를들어서 round-robin scheduilng을 얘기할때도 user 1이 9개의 프로세스를 돌리고있고 user 2가 1개의 프로세스를 돌리고 있었을 때 round-robin scheduuling을 적용하면 user 2는 cpu의 10프로밖에 못쓴다.user 1은 90프로를 쓴다. 따라서 사용자 간에 공평하지 않다. fair-share 공평한 몫. 사용자에게 공평한 몫을 주자. 누가 프로세스를 소유하고 있는 지를 고려. user1은 4개의 프로세스. user 2가 한개의 프로세스 user 1이 a,b,c,d 돌리고 user 2가 e를 돌리는데 user1은 50 퍼 user2도 50퍼 사용자간에 공평한 몫을 주려면 두명의 사용자가 시스템 안에 있으니까 50대 50으로 줘야한다 cpu. round robin으로 하더라도 ABCDEABCDE 하면 안된다. fair—share 스케줄링 적용하면. AEBECEDEAEBE 이런식으로 할수밖에 없다. 공평한 몫을 50프로씩 두사람한테 제공해야하니까. 누가 사용자냐. 사용자까지 고려해서 사용자에게 공평한 몫 주는 스케줄링. 이정도로만 알고있기.
real -time system scheduling.
real-time system은 deadline까지 external event(외부 이벤트에 대해서) must react appropriately 적절하게 행동을 해야하는 시스템. 얘를 들면 compact disk player(cd player) 돌고있는 cd로부터 데이터가 계속 들어온다. 데이터 가지고 음악으로 바꿔줘야하는데 제시간에 바꿔주지 못하면 소리 깨진다. real time system. patient monitoring int a hospital intensive-care unit. 중환자실에서 환자 모니터링 시스템. 적절하게 대처. autopilot in an aircraft 비행기의 오토파일럿 시스템. , robot control in a factory. 자동차 생산, 컨베이어 벨트 조립 자동차 지나갈때 그앞의 로봇은 자신의 작업을 시간 내에 끝내야 한다. 그렇지 않으면 컨베이어 벨트 따라서 조립되고있는 자동차 지나가버린다. real-time system들이 있다. 올바른 답을 너무 늦게 내게 되면은 그것은 전혀 답을 못낸 것 만큼 나쁘다. hard-realtime 과 soft real time이 있다.
hard real time은 아주 절대적인 deadline. 그것을 만족해야함. soft real time은 종종 deadline을 놓치는건 바람직하진 않아도 용납은 된다. 아까와 같은 cd player은 deadline 지나면 소리, 영상 깨질 수 는 있어도 큰 사고는 아니라 soft real time에 속한다. autopilot이나 robot control은 hard real time이다.
real-time behavior real-time system의 behavior을 보면 보통은 real-time system으로 돌아가는 프로그램은 프로그램을 여러개의 프로세스로 쪼개는데 쪼개긴 프로세스는 behavior 가 predictable하다. 예측가능하다. 정도로 알고있기.
deadline안의 external event에 대해 reaction을 취해줘야한다. 이때 그 external event. event는 periodic, aperiodic. 주기적, 비주기적. periodic event는 regular interval 마다 일어난다. 정규 시간 간격마다. 그런데 aperiodic은 비정규적. 비정규적으로 이벤트가 발생할 때 . real-time system이 schedulable 하려면 어떤 것을 만족해야하는지가 나와있다. 일단 먼저, m개의 periodic event가 있다고 가정. 여기에 주기적인 이벤트만 등장. m개의 periodic event가 있을 때 event i는 주기를 Pi 고. event i를 처리하는데 걸리는 시간은 Ci second로 가정. cpu가 Ci second 걸려서 처리할 수 있다. 그러면, 부하가 처리될 수 있다. -> 오직 각 프로세스에 의해서 사용되는 cpu의 fraction(부분) 의 합(sum)이 1 이하일 때 load 가 처리될 수 있다.

이것을 식으로 표현하면 i는 1부터 m까지 시그마 Pi분의 Ci 의 합이 1 이하이다. 이러면은 load가 handle 될 수 있다. 이때 이 real time system은 schedulable하다. 이 event 들을 계속 처리할 수가 있다. example 굉장히 중요하다. real-time system이 schedulable 한지 안한지 알아볼 수 있는 그 식. 굉장히 중요하다. 지금 어떤 real-time system 이 있는데 periodic event가 3개 등장한다. 하나는 주기가 100ms. 처리하는데 cpu time이 50ms. 또하나의 이벤트는 주기가 200ms, cpu time은 30 ms. 마지막은 500(100) 4번째 이벤트를 추가하고싶다. 4번째 이벤트는 period가 1sec. 1000ms이다. 이 네번째 이벤트를 추가해도 여전히 real-time system이 load를 핸들할 수 있는. 즉 , schedulable 하려면 cpu time은 (4번째 이벤트의) 몇 이하여야 하는지. 굉장히중요//// C1/P1+C2/P2+C3/P3+C4/P4 <= 1이하이다. 이렇게 식을 세워야한다.
50/100 + 30/200+ 100/500+x/1000 이게 <=1 이하면 이 real time system은 load 가 handle 될 수 있다. 풀어보면 양쪽의 1000을 곱해서 500+150+200+x <=1000 x <=150(ms)이하가 된다. 150ms 인 경우에 한해서 4번째 event가 추가될 수 있다. 여전히 realtime system이 schedulable. x가 150이하라면. 꼭 명심하기.
real-time scheduling algorithms은 static, dynamic으로 나눌 수 있다. static 한 것은 시스템이 시작하기 전에 scheduling decision을 미리 다 내려놓는 것. dynamic 은 runtime. 수행하면서 scheduling이 decision을 내림. 이정도로 알아두기.

policy versus mechanism.
scheduling policy와 scheduling mechanism을 분리하는 얘기. 그런데, 이게 무슨 얘기냐. 책에 나온예제. 예를들어서, dbms process를 얘를들어봄. DBMS process는 children process를 가질수 있는데 , 각 child 프로세스는 서로 다른 사용자의 DB request를 처리하는데 사용될 수 도 있고, 특정한 기능을 수행하는데 동원될 수 도있다. query 파싱한다던가. 이때 DBMS system의 main process는 자기의 child process 중에 누가 중요하고 누가 time critical 하고 누가 덜 중요한지 이런것들을 잘 알고있다. 그래서 scheduling 하는데 이 main process가 child process scheduling 하는데 좀 빨리 스케줄링 해야하고 쟤는 좀 나중에해도 되는데 이런것들을 잘 알고있는데 지금까지 우리가봤던 스케줄링 알고리즘은 그것을 반영할 방법이 없다.그래서 main process가 child process 1번을 먼저 수행해줘이런것들을 시스템에 요구할 수 있는 방법 설명된 것이 없다. 지금까지 알고리즘들은 그런것들 반영 ㄴ. policy와 mechanism이 그냥 다 합쳐져 있기 때문에 그렇다. 그런데, 이러한 DBMS 프로세스의 얘를 반영하려면 policy와 mechanism을 분류하는 것을 생각해 볼 수 있다. 이게 무슨얘기인가 하면 scheduling 알고리즘을 파라미터 화 시킨다. 그리고, 그 파라미터들을 user process가 채워넣는다 . 이렇게 하면 scheduling mechanism/policy 분리 된다. 구체적인 예.
kernel이 일단 scheduling mechanism을 제공한다. 즉, kernel이 예를들어서 priority scheduling을 제공한다.그것은 이제 scheduling mechanism. 그러면서 동시에 system call을 제공. user process가 system call을 call함으로써 자신의 자식 프로세스의 priority를 세팅 할 수 있게 해줌. 그러면 parent process 는 컨트롤 할 수 있다. 세부적으로 자신의 child process들이 scheduling되는것을 컨트롤 할 수 있다. parent process 자체가 priority scheduling 자체를 하지는 않음. 스케줄링 메카니즘은 kernel에 들어있다. 그렇지만 policy는 user process가 들어간다. 이 main process. 자신의 어느 child process가 우선순위가 7이어야 하고 , 그거 보다 낮은 3이되어야하고 등등.. policy는 user 프로세스가 세팅을 할 수 있다.system call의 파라미터를 통해서. 기본적인 scheduling mechanism은 kernel에 들어있다. 예를 들어서 priority scheduling. 이 스케줄링 메커니즘인데 kernel에 들어있다. 이게 바로 policy와 mechanism을 분리하는 내용.

지금까지는 프로세스 모델을 배움. 프로세스 모델- resource grouping , thread(of excusion).
resource grouping. 자원을 그루핑.즉, 프로세스마다 address space (그 프로세스의 텍스트,데이타,스택 세그먼트 가 된다. 코어 이미지)자원이 있고 open file(해당 프로세스가 오픈해놓은 파일)자원, child process, accounting information 이런것들이 있다.

프로세스는 Thread,, of excution의 개념이 있다, (thread- 실,줄,끈. 여기서는 이제 흐름. 수행의 실. 줄 .끈 .흐름 이런것)
프로세스는 이. 수행이된다. 그러면 수행의 흐름이 있다 ->thread of excution. thread는 추상적인 개념. 프로세스 모델에서는 thread of excusion수행의 흐름이 프로세스 당 하나만 존재하는 것. 그래서 그것에 관련된 자원이 프로그램 카운터, 레지스터, stack 이런것들이 있다. 프로그램 카운터는 현재 수행이 어디까지 이루어졌는지. 현재 수행중인 instruction의 주소.를 가진다. 레지스터는 상태들을 가진다. stack.은 지금 프로세스의 thread 가 어느 함수를 call했는데 함수가 return 하지 않았으면 함수의 return address라던가 local varible 가지고있다. stack에. 그래서, thread of excution에 이런 자원들이 관련이 되어있다.process 당 thread of excusion이 하나였다. 그게 process model.

thread model의 컨셉
같은 프로세스환경에(즉,하나의 프로세스 환경에 여러개의 thread ->multiple thread of executions)를 허용하는것이 thread model 이다. ==multithreading. 한 프로세스 안에 여러개의 thread 가 존재 하는것 허용하는 게 thread model,multithreading.
thread는 . 이때 cpu에서의 수행을 위해서 스케줄되는 단위. process model에서는 process 가. 그 단위었는데, thread model에서는 thread가 그 단위.lightweight process 라고도 한다. thread of execution에 관련된 그런. 정보는 program counter, 레지스터, 스택 이런정도. process 에 비해서 관련정보가 적게 관리. lightweight process 라고도 말한다.
그림을 보면, 왼쪽의 그림이 그냥 프로세스 모델.
프로세스 1,2,3. 그다음에 각 프로세스 안에는 thread 가 하나씩 있다.그게 프로세스 모델. 3개의 프로세스가 관련이 없는 보통 경우 .그런 경우가 많다. 그런데 이제 b 같은 경우에는 thread 모델. multithreading. 하나의 프로세스. 동그라미가 프로세스를 나타냄. 하나의 프로세스 안에 multiple thread. 이 경우에는 3개의 thread 가 있다. 그래서 thread들은 프로세스의 address space를 share 하는 것. 그래서 이 세개의 thread는 같은 job의 부분들을 담당하고 있는 것. 굉장히 긴밀하게 서로 협조하고 있다. b가 멀티 threading. thread 모델을 보여주고 있다. thread 간에는 protection이 필요하지 않다. 왜냐하면 process 안에 있는 여러개의 multiple thread들은 cooperate 하게 프로그래머가 코딩을 해놨으니까. 그다음에 per process item과 per thread item.굉장히 중요하니까 반드시 명심.
per process item은 프로세스 안에 모든 thread 가 share하는 것. per thread item은 각 thread마다 private하게 있는 것. 그래서 per process item의 경우에는 address space 라던가 global variable open file child process pending alarms 등등. per thread item에는 program counter , register, stack , state 들이 있다. 그래서, 지금 표가 중요하다. 여러개의 thread가 자원을 공유한다. 그래서 그들이 어떤 임무를 수행하기 위해서 긴밀하게 같이 일할 수 있다.

전통적인 process 처럼 , running blocked ready 상태를 가질수가 있다. 각 thread는 자신만의 stack을 가진다. stack은 call 했지만 return 하지 않은 그런 procedure 에 대한 정보를 갖는다. procedure 당 하나의 frame의 정보. ->procedure의 local variable이라던가 , 즉 parameter. 선언된 automatic variable들. return address. function으로부터 return 하는 것들을 stack에 갖게 된다. 각 thread는 일반적으로 서로 다른 procedure를 자연스럽게 call 할 수 있다. 그래서 서로 다른 execution history를 가지게 된다. 자신만의 stack 이 있어야한다. 동그라미 ==프로세스 같은 프로세스 안에 thread가 3개. multithreading은. 각 thread 마다 stack이 있다.

thread 관련 library procedure. thread _create은 멀티쓰레딩 환경에서 프로세스가 보통 single thread로 시작.그러다가 그 thread가 새로운 thread를 create. thread_create를 call 함으로써. thread_create할때 argument. thread_create call할때는 새로운 thread를 생성하는 것. 그러면 새로운 thread가 생성되어 수행할 프로시저의 이름을 아규먼트로 준다. 그다음에 두번쨰, thread_exit. thread가 자신의 일을 마치고 종료할 댸 call. thread_wait. 다른 thread가 종료할때 까지 이것을 call한 thread를 block. a thread가 b thread에 대해 thread_wait을 call하면 b thread가 종료될때 까지 a thread는 blocked 상태에 가있다. thread_yield 는 자발적으로 cpu를 양보. 다른 thread가 수행될 수 있도록, -> mutex에서 등장. mutex 는 semaphore의 simplify된 버전. 두가지 state 중 하나. unlocked(0) or locked (1 or 아무거나) mutex_lock, critical_region, mutex_unlock 순서로 call. example 어셈블리어로 ->18page
mutex_lock: TSL REGISTER,MUTEX 이니까 mutex 값이 레지스터로 들어가고 mutex 1로세팅. mutex가 0이었다면 레지스터에 0들어가고 JZE jump if zero 레지스터와 0과의 차이가 0이면, ok로 점프 후 return. mutex가 unlock 즉 0이면 바로 return하게 된다. critical region으로 들어간다. mutex가 locked였으면2 레지스터 값에 1이들어가니까 차이가 0이 아니라서 ok로 못가고 CALL thread_yield. cpu를 다른 thread에게 양보한다. 현재 이것을 call한 thread -> ready로 밀려난다.os 스케줄러가 ready에 있는애 뽑아서 running으로 둔다. 다른 thread에게 cpu 양보. thread는 ready상태에서 맨뒤에 있지만 기다리다 보면 cpu차지하게 될 수 있다. running으로 돌아오면, CALL로부터 return하게 된다. JMP mutex_lock으로 돌아가서. 다시 반복한다. lock이 되면 thread_yield 하면서 반복. unlock이 될때까지 기다림. unlock이 되면 jze 로 return 해서 critical region으로 들어간다.
많은 어플리케이션의 경우, 애플리케이션 안에 multiple activity가 일어난다. 동시에. 워드프로세서 어플리케이션 같은 경우에 이것을 수행시키면 그안에 여러가지 activity가 돌아간다.어떤 activity ? 사용자와 interaction. 사용자의 입력 바탕으로 reformatting. 주기적으로 디스크에 버퍼 내용을 save 하는 activity. activity 하나를 하나의 thread로. 프로그래밍 하면 프로그래밍 하기가 명료해진다. 쉽고 편해진다. 그다음에 thread가 프로세스보다 create, destroy 하는게 쉽다. 프로세스는 create하려면 여러가지 자원 setup해줘야한다. 프로세스의 자원들 (resource grouping == address space, … 다 자원) set up 해줘야하기때문에 굉장히 시간이걸린다. 그것에비해 thread 굉장히 만들기 간단. program counter,register,stack 필요.per process item이 훨씬 복잡.무겁다. process create하기 시간 많이 소요 thread 가볍다. performance gain. 성능 이득. 프로세스가 여러개의 thread로 되어있을 때 한 thread가 i/o하느라고 blocked된 동안에 다른 thread가 cpu사용해서 계산한다던가. 성능상의 향상이 있다. 마지막으로, multiple cpu환경에서는 thread하나를 그 cpu하나에 할당. 하드웨어적으로 여러개의 cpu환경이니까 각 cpu마다 thread하나씩 돌아간다. 진정한 병렬처리. 처리가빨라진다.

thread 사용의 예 word processor,web server
word processor의 경우 3개의 thread로 만들 수 있다.interactive thread, reformatting thread, disk-backup thread. 동그라미가 word processor 프로세스. 왼쪽 이 interactive thread 가운데가 reformatting thread 오른쪽이 disk-backup thread . interactive thread는 사용자가 키보드를 써서 명령을 내리는것. 사용자와 interaction하면서 명령어 받아들이는. scroll 등. reformatting thread는 결국에는 사용자가 줄을 ,문장을 첨가한다더나 삭제한다거나 이런걸했을때 해당 페이지의 내용 부분이 바뀌게된다. 다음페이지와 현재페이지의 경계가 바뀜. 그런것들을 reformat해준다. 버퍼에있는 내용에 page-break.line-break이런것을 넣어준다. 사용자의 입력을 바탕으로.disk backup thread. 주기적으로 버퍼에있는 내용을 disk로 save해주는 thread. 이것을 single-thread환경이랑 비교해보는게 나옴.만약에 3개의 thread로 만들지 않고 하나의thread로 만들면 .. 800페이지 문서 그중에 1페이지에서 하나의 문장 삭제. 그러면 사용자는 사용자는 그것을 삭제를한다음에 1번페이지 문장 삭제한후 1번페이지가 잘 되어있는지 확인.

그 후에 사용자가 페이지600 의 내용을 수정하고싶음. 사용자가 명령을 내림 600페이지로가라.wordprocessor는 사용자가. 1페이지를 수정해서, reformatting을 다 해야함. 그래서 reformatting 다 한다음에야 600페이지 보여줄 수 있다.600페이지까지 reformatting. delay생김. 사용자입장은 기다려야함. single-thread환경에서는 다 해야해서 그럴수있다..multiple thread환경에서는 reformatting thread존재. 사용자가 1페이지 문장 삭제. 그러면 interactive thread는 그 내용 reformatting thread에게 알려줌. 그래서 reformatting thread에게 페이지1문장이 삭제되엇고 그것에 기반해서 책 전체 reformatting하라고 interactive thread가 reformatting thread에게 말해줌.reformatting thread는 background에서 reformatting.그러는 동안에 사용자는 1페이지 삭제했으니 확인.확인하는 동안 시간이 흘러, 600페이지로 가라고명령내리면 reformmatting thread가 일을 다 한 후일수 있어서 600페이지. 바로 보여줄 수 있다. 두번째는 web server. ->multithreaded.
동그라미가 web server process. 왼쪽에dispatcher thread. 오른쪽아래 worker thread. 네트워크를 타고 client로부터 web page 요청이 들어오면 그 예를들어 main page 보여달라 등등. 요청을 dispatcher thread가 받아서 페이지요청 하나하나를 worker thread에게 할당해준다. 일감을 던져준다. worker thread는 특정 페이지 요청을 받아다가 그것을 읽어다가 돌려준다. worker thread입장에서는 들어온 요청이 web page cache에 content 가 있으면, 거기서 바로 읽어서 돌려주면된다. 만약에없으면 disk 에서 읽어서 돌려준다. 그림아래부분은 코드.a->dispatcher thread b->worker thread.
a는 while(true)루프를돈다. get_next_request(&buf)->네트워크 통해 들어오는 요청 받는다. 특정페이지(url) 그 것을buf로받는다. handoff_Work(&buf) 페이지 url 같은거. worker thread에게 넘겨준다. worker thread 코드 b. wait_for_work. 웹페이지 요청을 dispatch로부터 기다린다. 넘겨주면 받아다가 ,버퍼의 특정페이지 url같은게 넘어옴. look_for_page_In_cache cache에먼저 페이지가 있는지 체크. 아규먼트 buf,page buf->특정 페이지 url. page->그것을 cache에서 읽으면 두번째 argument 에 content가 들어감.cache에 없으면 null이 들어가거나 그런다.if(page_not_in_cache(&page)page내용이 비어있으면 disk로부터 읽어라.cache 또는 disk로부터 읽혀진 page content 가 page 라는 variable 에 들어간다. 그래서 return_page(&page); 페이지를 돌려준다.
thread를 implementing하는 방법. 크게 두가지 ->user space에서, kernel 에서 implement. 앞을 user level thread. 뒤를 kernel level thread. user level thread는 thread 패키지가 user space에 있다.kernel은 thhread의 존재를 모름. kernel은 프로세스만 알고있다(관ㄹㅣ)thread는 user space에서 관리된다ㅏ. process동그라미 아래 직사각형 있고 그 위에 thread가 있다. thread가 직사각형 위에서 수행. 직사각형 ->runtime system. runtime system상에서 thread 수행.runtime system은 thread를 관리하는 프로시저들의 collection(모음) 어떤 프로시저가있느냐 ->thread_creat,wait,exit,yield같은거. 각 프로세스가 자신만의 thread table 가짐. user-level thread 특징 중 하나.runtime system이 thread table을 관리. kernel은 프로세스 table 관리. user-level thread 구현의 이점
1. thread를 지원하지 않는 os 에서도 구현을 할 수 있다. user space에서 thread package 구현.
2. thread switching이 kernel-level에 비해 빠르다. kernel -level은 switching할때 kernel로 들어가야함. 시간이 많이걸림. user-level thread는 커널로 들어가지 않아 user level 에서 switching이 일어나서 더 빠르다. thread가 다른 thread 기다리면서 block 이 되어있는 상태. thread a 가 thread b가 일 끝마치기를 기다리면서 block되있는것 wait같은거 call. thread_Waik를 call하게 되면 runtime system이 처리. runtime system은 thread_wait call한 a blocked 로놓고 ready 상태에있는 애들중에 하나를 running으로. (cpu를 차지하게된다) 이 과정에 있어 kernel로 들어가지않는다. switching. runtime system이 user space에서 처리.
3. 각 프로세스는 자신만의 customized된 스케줄링 알고리즘을 가질수있다.
4. 스케일이 더 잘된다.
kernel -level thread는 thread table을 kernel에 가지고있어야함. 처음에 kernel에 thread table setup. 얼마나 잡아야하나? ->넉넉하게 잡음. 얼마나 잡아야 넉넉한지도 애매. 처음에 setup해둔 thread table이 충분치 않으면 문제 가능성. user-level인경우는 user space에 있어서 프로세스 하나 생성될때마다 user space의 프로세스를 위한 thread table만들기만 하면된다. 따로 뭘 잡아둘필요 없다. scale 이 더 잘된다.

user-level thread. disadvantage
1.blocing system call을 implement하는게 까다롭다. user-level thread implementation에서는 어떤 프로세스의 한 thread가 blocking system call 해버리면 프로세스에 속한 다른 thread들도 stop 시켜버림.왜냐하면 kernel입장에서는 thread 존재 몰라서. 그래서 그림에서 9페이지 그림의 thread 중에 그중의 하나가 blocking system call 한다. 그러면 kernel은 thread 존재를 모르니까 그냥 blocking system call한thread 가 속해있는 프로세스 자체를 block 상태로 보낸다. 다른 thread 마저 blocked 상태로 끌려간다. 같ㅇ튼 프로세스에 속한 다른 애들까지. stop. 해결방법 1. nonblocking call로 system call 뜯어고친다. read system call 뜯어고친다. 키보드에 대한 read인데 타이핑이 없을때 ,버퍼에 없을때 키보드에 대한 read콜하면 그냥 0 리턴. block이 안된다. 문제는 매력적이지 않다. 2. blocking s ystem call 이 block 일으킬지아닐지 미리 체크. block 안할경우에만 call을한다. unix의 select system call: select system call 후에 하는것(ex.read system calll) 들이 block 할지 안할지를 알려줌.safe 인지아닌지.그런 기능이있다. read library procedure수정. 바로 read system call 수행하게하지말고 일단 select systsem call 먼저수행. 앞으로 readsystem call 할때 block안하는게 확보->read system call. block 이 된다는 결과가 오면 read system call 안하고 다른 thread 에게 cpu 양보하고 ready. running상태로 올때 다시 select call해서 체크.
이런식으로 read system call 주위에 checking 하는코드. wrapper. or jacket 참고로만 아세요

구현하는 방법에 대해서 얘기 user level thread즉 user space에서 thread 구현하는 방법에 대해서 설명. 그림에서 보는바와 같이 thread package가 user space 쪽에 있다. user level space 관리하는것이 runtime system. 프로세스마다 thread table이 존재. user level thread에서는 .runtime system이 해당 프로세스의 thread들을 관리. kernel은 thread 존제 자체 모름. process의 존재 자체만 알고 process table만 관리

user level thread의 advantage이점 중요하다. disadvantage도 중요 . 명시. kernel -level thread의 특징도 중요, 명심.. 특징,장단점(user,kernel-level thread)
대표적인게 os가 thread support 안해도 구현할 수 있고 thread switching이 kernel-level thread보다 빠르다. kerner로 안들어가니까 빠르다. 각각의 프로세스가 자신만의 customized된 스케줄링 알고리즘을 가지는것을 허용한다. 스케일이 더 크다. user-level thread의 단점(disadvantage) blocking system을 구현하는게 사실상 어렵다. 왜냐하면, 어떤 process의 thread가 blocking system call 해버리면 (ex.read) 그 thread가 속한 process가 kernel에 의해 block. kernel은 process의 존재만 아니까. 그렇게되면 thread와 같이 해당 프로세스에 속해있던 다른 thread들도 blocked상태로 넘어간다. kernel이 해당 프로세스 자체를 통채로 blocked상태로 보내버리니까. 이것때문에 문제가 된다고 지난시간에 문제. nonblocking system call로 고친다던가, wrapper를 씌운다던가. 이런식으로 지저분한 해결책을 생각할 수 밖에 없다. blocking system call하게 되면 문제가 발생. 프로세스 전체가 block. 또하나의 문제점은, giving up the CPU. thread가 한번 어떤 프로세스 속에 cpu를 할당받아서 수행되기 시작하면 그 프로세스의 다른 thread들은 cpu를 차지할 기회가 사실 없다. 그 수행되고 있는 thread가 자발적으로 cpu를 give up 하지 않는이상. 즉, thread가 수행되고 있는데 thread마다 quantum이 있다고 쳐가지고 너 지금 이 thread quantum 다 썼으니까 다른 thread에게 넘겨줄게를 하려면 runtime system이 clock interrupt를 받아들일 수 있어야하는데, thread level에서 runtime system이 clock interrupt를 받아들이도록 구현하는게 상당히 어렵다. 결과적으로 process안에서 process의 quantum을 쪼개서 thread마다 조그만 quantum을 줘서 퀀텀이 다 차면 clock interrupt를 통해 그 프로세스의 다른 thread에게 cpu 주고 이런것이 구현하기가 힘들다. 결과적으로는 , 프로세스의 한 thread가 수행중이면 얘는 줄창 얘가 속한 프로세스의 quantum이 다할때까지 그냥 cpu를 쓸 수가있다. 단 얘가 cpu를 양보하게 되면 다른 thread가 수행할 기회가 생긴다. give up 하면. give up 하지 않는이상 다른 thread에게 기회를 주기가 상당히 복잡하다.힘들다. 사실상 user-level thread에서는 thread간의 switching이 일어나는 것은 보통 thread yield 같은것 call하게 되면 runtime system이 해당 thread yield를 call한 thread를 blocked 상태로 보내고 ready 상태에 있는 thread 중에 하나를 running 상태로 보낸다. 이런식으로 thread switching.(thread yield 같은거 call해서) call한 thread는 ready 상태로 간다. blocked상태가 아니고.. ready 상태에 있는 애들중의 하나가 running상태로 간다. thread yield를 call한 thread는 ready상태로 강등된다. thread wait를 call. 다른 thread를 기다릴 때. runtime process가 처리를 해줄 수 있다. thread wait를 call한 애를 blocked로 보내고 ready상태에 있는 thread를 running 상태로. thread switching이 user-level thread에서는 이런식으로 일어난다. thread yield한다던가 wait를 한다던가.한마디로 thread level에 quantum을 줘서 clock interrrupt를 통해서 thread를 돌아가면서 수행하는게 사실상 힘들다. 단점. 한놈의 thread가 양보를 안하면 그 프로세스의 퀀텀을 다 쓸수도 있다.

누리겠다.uslevel thread. thread package가 kernel에 있다. kernel이 thread를 관리. 실제로 thread table이 kernel에 있다. 프로세스가 지금 user space에 두개가 있다 동그라미 두개. 이 안에 thread들이 들어있다. 이 thread들에 대한 정보가 kernel이 관리하는 thread table속에 있다. kernel level thread에서는 어떤 thread하나가 blocking system call을 하면, kernel로 들어간다.(disk로부터 파일을 읽는다거나 그럴때) kernel에 들어가면 kernel이 해당 thread를 blocked상태로 보내면 된다. kernel은 같은 프로세스에 속한 다른 thread를 running상태로 놓고 수행하면 된다. user level thread와 같이 어떤 프로세스의 thread가 blocking system call하면 이 프로세스의 전체가 blocked상태로 감으로써 다른 thread가 수행할 기회를 놓치는게 아니다. kernel이 같은 프로세스의 다른 thread를 수행할 수 있다 해당 thread만 block시킨다. kernel 자체가 thread의 존재를 알고있다(관리) process뿐만 아니라 thread의 존재도 kernel이 알고있으니까. user level thread와 같이 힘들게 할 필요 없다.blocking system call 하도록 내버려둬도 된다. 한 프로세스에 한 thread 중 하나가 blocking system call하더라도 그 같은 프로세스에 속한 다른 thread들은 blocked되지 않는다. 이게 좋은 점

user-level thread와 kernel-level thread의 특징 명심 (파란글씨 특징의 핵심)

여기서부터는 참고로만 아세요
Hybrid implementations
kernel에 이미 thread의 존재가 있다. kernel에 있는 thread 하나에 대해서 user space의 프로세스 속에는 여러개의 user thread가 대응되어 있다. user -level thread들을 multiplex 시켰다. kernel level thread가. 이정도로만 알고 넘어가기

scheduler activations
user-level thread와 kernel-level thread의 장점을 좀 섞은 방법. user-level thread가 이런 장점. 예를들어서 한 thread가 thread wait를 call해서 다른 thread를 기다리면서 얘는 blocked로 간다. runtime system이 thread wait를 call한 애를 blocked 상태로 보내고, 그다음에 process에 속한 다른 thread들 중에 ready상태에 있는 다른 thread들 중 하나를 running상태로 보내는데 이걸 runtime system이 해준다. kernel로 안들어가서 속도가 빠르다. 그래서 이 이점을 그대로 누리겠다. kernel로 들어갈 필요 없다. 근데, blocking system call을 하면.. user-level thread의 개념으로 처리하면 프로세스 전체가 blocked된다. 그러면 다른 thread들도 다 block. 이때는 kernel의 도움을 받는다. thread하나가 read system call을 했다. 일단 kernel로 들어간다 trap일어나면서. 이 kernel이 blocking system call을 한 thread의 정체 (thread number라던가 어떤 i.o를 요청했다.) 그런걸 요청했다는 내용을.. 어떤 thread가 blocking system call을 했고 어떤 event를 기다리는지. 이런 내용을 kernel이 runtime system에게 정보를 전해준다 runtime 시스템의 procedure을 call함으로써.kernel이. kernel이 user space에 있는 runtime system에 procedure을 call해서 이러한 정보를 전해준다. thread number라던가 어떤 이벤트인지. 이게 up call이라고 한다. 그런다음에 runtime system이 정보를 받았다. 이 thread를 blocked 상태로 보낸다(user level에 있는 runtime system이) 자신이 관리하는 ready상태의 thread 중 하나 running으로 . 시간이 흘러 I/O 완료. 그러면, 이거에 대해서 다시 kernel은 runtime system에게 알려준다. thread가 기다리던 event가 완료. 디스크에서 file 다 읽었어 라고 알려준다. runtime system이 kernel로부터 그 얘기를 듣고 blocked상태로 보냈던 애를 ready로 보낸다. 이때도 kernel이 up call을 한다. runtime system에게 이벤트가 발생했다고 알려줌, up call의 이름이 왜 up call이나면 일반적으로 software architecture는 위에 application이 있고 그밑에 library가 있고 그다음에 그 밑에 OS kerneld l 있다. 위에서 아래로 일반적으로 call이 일어남. 지금은 kernel이 위쪽으로 call한것이라서 up call이라고 한다. 그래서 이게 scheduler activation 방법이다. kernel이 user space에 있는 runtime system call하면. up call 이것은 별로 좋지 못하다. 일반적으로는.

pop-up threads
일반적으로 서버 프로세스 같은 것들이 incoming message 처리하는 것은 보통 서버 프로세스 안에 thread가 여러개가 있어서 thread가 receive system call 해서 보통 incoming message가 없을때는 receive system call을 하면 blocked 된다. incoming message가 들어오게 되면 그 thread가 이제 결과적으로 running 상태로 가게되고 메세지를 처리해야함. blocked 되있던 thread가 message가 들어옴으로써 running상태로 가게 되면 자기가 save 되었던 state를 다시 loading을 해야한다. 레지스터 값들이라던가. 그래서 이런것들이 시간이 좀 걸린다 그것을 단축하기 위해 등장한것이 pop-up thread. 이것은 기존에 thread가 존재하고 있으면서 blocked되있다가 나중에 message 들어왔을 때 상태 loading하는게 아니고 message가 들어왔을때 그 때 thread를 생성. create. 그렇게 되면 상태 loading할것도 없다. 시간이 좀더 적게 걸린다. 그런 의미에서 사용한다. 그정도로만

single threaded code를 multithreaded로 바꿀 때 생각해야 할 issue
예를 들어가지고 thread가 두개가 있는데 thread 1이 같은 프로세스에 속해있다고 했을 때 thread 1이 access system call을 했다. 어떤 file에 대해서 내가 access 할 function이 있는지. 그것을 check하는 system call. 그런데. system call 하게 되면 에러가 발생하면 global variable errno라는 global variable에 error code를 넣어준다. access system call을 했을때도 errno라는 global variable에 err value가 세팅. multithread 환경. thread 1이 access system call했으면 그것을 처리하면서 errno가 setting되었을텐데 그런데 이제 thread 1이 조금 수행되다가 switching이 일어나서 thread2가 수행. 얘가 수행되면서 open system call하면 수행하면서 에러가 발생하면 errno에 값이 씌여진다 .아까 thread1이 access하면서 그것을 처리하면서 errno에 세팅한 값이 overwrite된다. thread 1으로 switching 했을 때 errno를 thread1이 체크해보면 이미 자기가 원했던 값은 사라져있다. global variable 들이 multithreaded환경에서는 제대로 사용될수없을수가 있다. thread 수준의 global variable이 필요. 그런것들을 하기 위해서는 thread마다 global variable 저장하는 부분을 만들자. 프로세스 image 속에.

thread scheduling.
앞에가 user-level thread의 scheduling. 뒤에가 kernel-level thread의 scheduling.
user-level thread의 scheduling 먼저. user-level에서는 kernel이 thread의 존재를 모른다. kernel이 프로세스 스케줄링만 한다. 프로세스a를 예를들어서 kernel이 선택을 해서 cpu를 준다 running으로 둔다. 그런데 이제 예를 들어서 kernel 이 프로세스 a를 선택해서 running상태로 놨다. 그럼 이안에서 runtime system은 thread 1. process a의 thread 1이니까 A1이라고 부른다. thread A1을 runtime system이 수행할 수 있다 . 그런데 thread간에는 clock interrupt이용이 힘들기 때문에 한번 thread가 runtime system에 의해 running 상태가 되면 얘는 그냥 계속 프로세스의 quantum을 잡아먹게 된다. 프로세스의 주어진 퀀텀안에서 원하는 만큼수행. 50ms이다. 이러면 runtime system에 의해서 선택된 thread A1은 running상태가 되면 자기가 마음만 먹으면 프로세스의 퀀텀 50ms 다 독점할 수 있다. 그래서, 이런 경우를 생각을 해본다. process A가 running상태가 되었고, runtime system에 의해 thread A1이 선택,수행시작. 얘가 5ms 수행이 되고, 그다음에 예를 들어서 yield를 한다. 즉 thread yield를 call한다. 이제 runtime system이 A2를 running 상태에 놓는다. yield를 call한 thread A1은 ready상태로 끌어낸다. A2가 running상태에 대해수행. A2도 5ms 사용한다고 가정. 5ms 사용하고 thread yield를 call. 그러면은 runtime system은 A2를 running에서 끌어내려서 ready로 보내고 ready 중 하나에 있는 예를들어 A3 를 running으로 놓을 수 있다. A3도 5ms… 이런식으로 돌아가면서 수행. A1,A2,A3,A1,A2,A3,A1 10번하면 프로세스 b가 kernel에 의해 수행. 이 안의 thread들이 runtime system에 의해 실행. possilbe not possible이 A1,B1 .. 프로세스 a의 퀀텀이 끝날때 까지는 이 안에서만 스케줄링. 프로세스 간에 건너 뛰면서는 안된다. switching할때 퍼포먼스가 좋다. user-level thread라 kernel 안들어가고 runtime system에서 바로 switch. 빠르다. high performance.

mon.is -level thread 에서 scheduling. kernel이 therad의 존재를 알고있다. user-level에서는 kernel이 thread 존재 모르고 process만 scheduling. kernel 이 thread 존재 알고 관리. pick 해서 수행 시킨다. process a의 thread1 A1을 kernel이 선택해서 수행을 한다. 했을 때 A1,A2,A3,A1,A2,A3 도 가능하고 A1,B1,A2,B2 처럼 프로세스 왔다갔다 하면서도 가능하다. kernel이 thread의 존재를 알고있기 때문. kernel은 thread -level의 quantum을 줄 수도 있다. 예를 들어서 thread 1의 thread-level 의 작은 quantum을 주고 걔가 thread-level의 quantum이 다 될때까지 수행 해주면 cpu를 뺏어가지고 다른 thread에게 cpu를 넘겨준다던가. 그것도 가능하다. kernel 이 thread의 존재를 알고있기 때문에 가능 . user-level thread에서는 runtime system이 clock interrupt를 이용해서 kernel level quantum을 thread level quantum을 처리하려고 했는데 그것은 어렵고 (user-level에서) kernel 수준에서는 용이하다. kernel은 thread level의 quantum을 주고 scheduling 하는것이 가능하다. kernel level thread같은 경우에는 I/O를 call해도. 예를 들어 A1이라는 thread가 blocking system call 즉 i/o를 call해도 전체 프로세스 a가 block되지는 않는다. 전체 프로세스 가 block되는건 아니다 kernel-level thread의 장점. 이정도 까지만 알고 있으면 된다.

참고로만 아세요 23 페이지 Monitor 자바 코드 example
producer-consumer을 java로 구현. monitor와 같은 기능 서포트 하는게 java에 있다.

main 메소드를 보면 이게 처음 수행되는 것. p.start c.start 라고 있는데 producer thread, consumer thread 시작. 각각.p는 new producer(); producer는 thread라는 class로부터 derived된것. thread구현할 때는 파생시켜서 하는 경우가 꽤 많다. static class producer extends Thread. 그 방법 사용 run이라는 메소드가 구현. c.start 의 c는 consumer로부터 만들어진 객체. consumer class도 thread로부터 파생. 이 안에 run이 구현. 각각 producer와 consumer thread는 run을 구현해놓은 상태. p.start() c.start() 하게 되면 producer thread의 run, consumer thread의 run이 실행. p.start에서 run이 수행되니까 while루프가 돈다.item 하나 produce 하고 그다음에 mon.insert(item) 만들어진 item을 mon.insert의 argument로 준다 consumer의 run에서는 돌면서 mon.remove() . 아이템을 받고 그것을 소비 consume_item(itme);
mon은, our_monitor 로부터 instanciation 된 객체. our_monitor class는 뒤에 나와있다.

중요한것은 메소드인데, insert와 remove 메소드 insert는 공유 버퍼에 아이템 하나 넣는거. remove 메소드는 버퍼에서 한 아이템 빼는 것. 카운트 감소. 결국은 이부분을 (25page0monitor 처럼 만드는것. 어떻게 만드느냐. insert 앞부분에 synchronized 라는 수식어가 붙어있다 이렇게 되면 한마디로 our_monitor로부터 instanciate된 object가 있는데 여기서는 mon.. 이 오브젝트에 대해서 insert를 call. insert가 synchronized이기 때문에 call 하게 되면 mon에 lock 이 걸린다. 그래서 다른 thread가 이쪽으로 못들어온다. 한 thread가 synchronized 메소드 하나 call해서 그 안에 수행하고 있으면 다른 thread가 synchronized 메소드 call 하면 그 메소드 속으로 들어갈 수 없다. 대기. monitor에는 한놈만 들어올 수 있는데 그것을 가능하게 한다 자바에서 mutual exclusion 제공 .또하나가 뭔가 하면 insert와 remove는동기화가 되어야한다. 예를 들어 버퍼가 비어있는데 remove가 call하면 count 가 0이라 go to sleep으로 간다. go_to_sleep 이라는 메소드는 wait를 call하게 된다. 자바의 wait는 어떤 thread가 call하면 걔는 이 object에 대해 block이 된다. 나중에 다른 object가 lock이 풀리기 때문에 들어올 수 있다. insert 속으로 들어와서 수행 가능. insert 메소드에서 count 1까지 수행 하고 notify() 자바에서 notify를 call하면 그 object에 대해서 blocked 된에(wait를 call해서) 깨워준다. 따라서 consumer sleep 한 애가 깨어난다. 이런식으로 해서 동기화를 한다. 이정도로만. 참고로만 알고 있자.
메모리 관리 . 프로그래머들은 크고, 속도가 빠르고, 메모리에 데이터를 썼을 때 없어지지 않는.살아있지 않는 메모리 원한다. 메모리의 hierarchy계층 구조 위쪽 cache. 일반적으로 cpu속에 들어있고 용량은 굉장히 작다 굉장히 빠르다. 비싸다 . 그 아래계층에 main memory ram이 있다. 속도는 cache보다는 느리다. 용량은 cache보다 많고 가격은 cache보다 좀 덜 비싸다. 맨 아래쪽 계층. disk storage가 있다,, 보통은 이제 하드 디스크를 많이 썼었다 요즘은 ssd. disk storage는 굉장히 용량이 크다. 단가 싸고 속도가 느리다. memory manager첫줄정도만 이해하고 있기. 메모리 hierarchy를 관리하는 operating system의 일부분이다. 일단은 넘어감. 기본 메모리 관리. 기본적인 메모리 관리에 대해 책에 나와있다. 이 메모리 관리의 가장 단순하고 기본적인 형태. 한번에 한 프로그램씩 수행한다. 그것을 위에 monoprogramming이라고 한다. 가장 단순한 메모리 관리의 형태. 한번에 한프로그램만 수행을 한다. 그래서 파란글씨로 써져있지만 os가 메모리에 있고 한개의 user process가 있다. 한번에 한 process만 실행을 한다. 세가지 타입이 있다고 볼 수 있다.

a가 첫번째 style인데 이게 이제 main frame computer는 예전 mini computer에서 많이 쓰이는 방식. ram이 있으면 ram의 아래쪽 주소에 os가 존재하고 그 위에서부터는 user program이 존재하는것. b방식은 예전에 palmtop computer,embedded system에서 많이 쓰이는 방식으로 메모리 아래주소에 user program이 있고 os는 위쪽 주소에 보통 있는데 ram이 아니라 위쪽 주소에 os가 있는 부분을 rom(Read only memory) 읽기만 할수있는 . 그부분에 os 장착. personal computer 는 c의 방법으로 메모리 관리. 메모리의 낮은 주소에 os가 있고 (ram) 그 윗부분에 user program이 있고 더 높은 주소에는 rom이 있어서, 그 rom에 device driver. 장치 관리자 부분이 있다. software 부분. 이것도 사실 os의 일부분. i/o device 장치들을 관리하는 software 부분, 운영체제 일부. 3가지 정도 메모리 구성하는 방법이 있었다.

멀티프로그래밍 방식으로 메모리를 구성하는 것. 여러개의 프로세스들. multiple process가 메모리에 존재하면서 걔네들을 돌아가면서 수행하는 것을 multiprogramming. 같이 수행하는 것. 그래서 보통은 여러개의 프로세스들을 동시에 돌리는 multiprogramming은 예를 들어 한프로세스가 i/o 요청하고 대기하고 있을 때 다른 프로세스가 cpu를 차지해서 쓰는 이런 식으로 보통 하는경우가 많다. 한 프로세스가 quantum을 다 쓰면 그다음에 걔를 running상태에서 내리고 다른 ready 중의 하나를 돌리는 그런 방식. 그래서 결국은 여러개의 프로세스들을 한꺼번에 돌리는것 . 그게 multiprogramming. 그래서 multiprogramming 방법으로 구성하는 것에 대해 나옴. 그림 a . a,b 가 다 multiprogramming with fixed partitions다. 메모리를 partition으로 나누는 것. 여러가지 크기의 partition으로 미리 나눠둔다. 그래서 system이 처음에 시작할때 나눠져 있는것 . 그림 a 먼저 보면 job이 시스템에 도착하면 job의 크기에 맞는 가장 작은 파티션에 input queue에다가 넣는다. partition 1,2,3,4가 다 크기가 조금씩 다르다. 예를들어서 job이 시스템에 도착했는데 크기가 90k면, 90k를 수용할 수 있는 파티션 중 제일 작은 파티션에 줄을 세운다. 그게 파티션 1 정도가 될 수 있다. separate input queues for each partition 각 파티션마다 input queue 를 따로따로 두는 것이다. job들을 줄을 세운다 . 아쉬운 점은 예를 들어 , 어떤 job이 94k짜리. 그것을 수용할 수 있는 가장 작은 파티션에 놔야하니까 파티션1에 놔야하는데 파티션3은 사용중이 아니다. 대기하고 있는 job이 없다. 새롭게 도착한 job은 3번 파티션에 배정되었더라면 당장 수행을 할 수 있을텐데 1으로 배당이 되었으니까 줄서서 기다려야 하는 아쉬움이 있다. 파티션마다 인풋 큐를 따로 두는것의 아쉬움. 해결해보려고 나온 게 input queue를 하나만 두는것. b 방식이다. 그래서 job이 도착하면 input queue가 하나밖에 없어서 그 꼬리에 붙인다. job들은 단일 input queue에 붙이고 파티션이 비었다. 예를 들어서 파티션 3이 비었다면 , input queue의 앞에서부터 조사를 시작해서 파티션 3에 들어갈 수 있는 사이즈면 무조건 그 파티션에 job을 할당해 준다. 그래서 이 줄에 먼저 도착한 job이 아무래도 유리할 수가 있다. 앞에서부터 검사를 하니까. 일찍 도착한 job이 94k라도 300k 짜리 파티션이 비었으니까 300k짜리 파티션 할당받을 수 있다. multiprogramming with fixed partitions.

relocation and protection
multiprogramming 방식에서는 두가지 해결해야하는 필수적인 문제 가 있다. 하나가 relocation 또하나가 protection 문제. 예를들어서 program을 작성해서 complie해서 link. 그게 결과적으로는 executable file이 된다. 그런데 executable file에서 첫번째 instruction. 주소 100에 대한 procedure에 대한 call. executable file의 제일 앞부분이 주소 0. procedure을 call하는데 주소 100에 있는 procedure를 call한다는 것은 binary file의 시작 부분에서 100바이트 떨어진 부분의 프로시저를 call한다는 것. 그러면 예를 들어서 executable(binary) file이 파티션1에 load가 되었다고 가정. 100k에서 시작. 지금 명령문이 주소 100에 있는 procedure call한다고 했는데 그게 그대로 수행이 되면 operating system 쪽으로 call이 일어난다 주소 100이 100k보다 적으니까 . 실제로는 프로그램이 100k에 loading. 그 executable file 그 안에서 주소 100으로 procedure call이 일어나면 실제로는 100k+100 부분으로 call이 일어나야 함. 그 프로그램이 파티션2에 loading -> 주소 100으로의 call은 200k+100으로의 call이 되어야함. 이게 바로 relocation problem. 변수의 위치. 주소. 그런것들. 바이너리 파일의 주소를 직접 쓸 수가 없다. 그 부분을 어떻게 해결하느냐. relocation 문제를 해결하는 첫번째 방법. 프로그램이 메모리에 로딩 될 때 명령어를 수정. executable file에서는 100번지에 있는 프로시저를 call하는데 그 명령을 파티션 1에 loading할 때는 call 100k+100번째 이렇게 수정을 해서 loading을 한다. os의 loading하는 파트가 그렇게 한다. 그렇게 하기 위해서는 linker가 프로그램의 word중에 어떤 부분이 relocate해야 하는지 명시를 해둬야 함. loading할 때. 주소를 바꿀 수 있다. 변수의 주소, 프로시저의 주소 를 로딩할 때 바꾼다. 두번째 방법. base register와 limit register사용. 이거는 뭔가 하면 loading 할 때 instruction을 바꾸지 않는다. executable의 명령어가 call 100번지 이렇게 되어있으면 그걸 그대로 loading. 그런데 명령어를 수행 할때마다 call100번지 하면 100번지 에다가 base register의 주소를 더해서 수행한다. base register의 값에 뭐가 있냐면 loading된 파티션의 시작주소가 있다.

예를 들어서 executable file이 파티션 1 에 로딩. base register 값이 100k로 세팅. 명령어를 수행할 때 calll 100번지 . base register 값 + 100 번지. 올바른 위치로 call이 된다. 이게 두번째 방법.
protection issue
multiprogramming은 메모리 안에 여러개의 프로그램이 들어간다. 여러개의 프로세스를 한꺼번에 돌린다. 한프로세스가 다른 프로세스의 파티션을 침범하게 하면 안된다. protection issue. 그러면 이것을 어떻게 해결하는가. 앞의 두가지 각각의 방법에 그 떄 사용하는 protection 방법을 설명하면, relocation 해결 첫번째 방법이 executable file을 메모리에 loading할 때 instruction의 주소 부분을 다 바꾼다. 그 방법을 썼을 때 그당시에 쓰는 protection 메커니즘이 ibm 에서 쓰던 , (ibm360) 메모리를 block 단위로 나눈다 한 block이 2kb 되는 . 각 block에다가 protection code 할당. 4bit짜리. psw (program status word) cpu의 psw에는 4bit key가 포함. 명령어를 수행할 때 360 하드웨어가 지금 수행하는 프로세스가 자신의 psw상에서의 key값과 access하는 메모리의 block의 프로텍션 코드값과 일치하지 않으면 trap 발생. 프로그램은 자신이 수행할 때 psw의 4bit의 key값과 같은 protection code (4bit) 가 assign된 block만 access할 수 있다. os는 이런 방식으로 user process가 다른 프로세스의 파티션을 access하는 것을 막았다. 그다음에 relocation 해결하는 두번째 방법은 base 와 limit register를 쓰는 것. 그래서, 이방법은 relocation을 base register에다가 현재 수행되고 있는 프로그램의 파티션의 시작주소를 넣어준다. 그다음에 수행할 때마다 거기에 등장하는 주소. 거기에다가 base register 값을 더해서 수행.이때 protection은 limit 레지스터 사용. limit레지스터는 무슨 값을 가지고 있느냐. 예를들어서, 현재 프로그램이 파티션 1에 그림 a 에서 로딩되었다. base레지스터 값은 파티션1의 시작주소인 100k limit 레지스터는 파티션 1의 사이즈를 가진다. size니까 200-100=100k. limit레지스터가 파티션의 사이즈 값을 가진다. protection을 어떻게 해결하느냐. call 100 이 있으면 그 명령어에 등장하는 프로시저의 주소를 100이라고 치면 그것과 limit 레지스터 값 비교 limit은 100k call 100이면 100은 100k보다 작다. protection violation이 안일어남. call 105k는 100k보다 크므로 protection violation 이 일어난다. 이렇게 해서 자신의 partition이 아닌 다른 partition에 access하는 것을 막는다. relocation하는 두가지 방법이 장단점이 있을 수 있다. instruction을 메모리에 loading할 때 등장하는 주소들을 바꿔주는것. loading할때 시간이 많이 걸린다. loading time이 느려진다. 수행할 때는 바로바로 수행. 반면 base와 limit register사용하는 방법은 loading은 그대로 executable file을 instruction 수정 안하니까 loading은 시간 별로 안걸림. 반면에 수행할 때 instruction에 등장하는 변수 주소나 procedure 주소를 base register와 더해야 해서 instruction 수행할 때 속도가 느리다. 이것이 단점.
base와 limit 레지스터는 명령어에서 메모리 참조할 때마다 base register 값을 더해준다. 속도가 느리다

memory management
batch system. 메모리를 fixed partition으로 나눔. 단순하고 효과적. 그래서 job이 시스템에 도착하면 queue에서 줄을 서 있다가 자기 차례가 되면 이제 partition을 차지하게 된다. partition을 한번 차지하면 쭉 수행을 하게 된다 끝날때까지. 그래서 굉장히 simple한 방식이다.메모리에 다섯개의 job이있는데 메모리에 다섯개의 job이 다 들어가있다. 그러면 문제가 없는 것, 시스템에 존재하는 5개의 잡을 메모리의 파티션이 다 수용할 수 있으면 다 넣고 돌리면 된다. 그게 batch system 때는 그렇게 보통 했다. time sharing system이 등장하고 personal computer 가 등장하면서 상황이 바뀌었다. active한 process들을 모두 다 main memory에 넣지 못하는 상황 . 남는 프로세스들은 이제 disk로 임시적으로 보관 .그래서 자기 차례가 되었을 때 disk로 밀려나있던 애가 메모리로 들어오고 그런다. 메모리 관리의 두가지 방법
Swapping
각 프로세스를 통째로 들여온다. 수행을 하다가 disk로 옮긴다. 나중에 disk에서 읽어 오기도 하고. process를 통째로 디스크로 내보냈다가 필요할 때 메모리로 불러들여와서 수행.
Virtual Memory
프로그램들을 수행. 프로그램들이 메모리에 프로그램 하나하나가 통째로 들어와있지 않아도 수행이 가능. virtual memory 방식을 쓰면 여러개의 프로그램이 메모리에 들어와있는데 각 프로그램의 일부분이 메모리에 들어와있다고 생각. 자세한 것은 나중에 얘기.

걸린다.meng .
a~g 시간의 흐름. a에는 A라는 프로세스가 들어와있다. 그림 b에 B. 그림 c에 C라는 프로세스. 그다음에 D라는 프로세스가 수행을 할 필요가 있어서. D는 들어올 자리가 없다. D를 들여오기 위해 A를 내보냈다. 그다음에 그림 e에 D프로세스가 들어옴. A프로세스 수행하기 위해 자리 만들기 위해 B가 나간게 그림 f . 그림 g 에서 A프로세스가 들어옴.
파티션은 variable partition이라고 볼 수가 있다. 아까 fixed partition 그림에 나온건 계속 파티션이 고정. 크기가. 근데 이거는 그림 c에서는 A의 크기만큼 (예를들어 100k) . d,e,를 거치면서 e는 50k의 파티션을 쓰고있다. 그리고 이제 또 g로가면 A는 다른 위치에 100k 파티션을 잡고 있다. variable partition. fixed partition과 다른 개념. 고정이 되어있지 않음. multiprogramming with variable partitions 라는 소제목이 있다. swapping을 하다보면 프로세스가 나가고 들어오고 하다보고 하다보면 빈공간이 생긴다(hole) 빗금친 공간. multiple holes가 생길 수 있다. hole들을 하나로 합치면 쓸모가 있게 된다. 이것을 memory compaction (이 메커니즘) 프로세스를 한쪽으로 다 밀어내야함. 프로세스 image를 움직여야 하니까 시간이 걸린다. 그래서, 보통 data segement. 그림 a를 보면 프로세스의 image중에 data segment가 늘어날 때 어떻게 대응하느냐 이런 얘기를 하고있다. a같은 경우에는 메모리 상에 프로세스 a의 image와 b의 image가 있다. 그런데 이제 각 프로세스의 data segement가 커지면 여유 공간이 필요. 각각 프로세스 A와 B에 잡아놈. a가 가장 단순한 방법, b가 정교한 방법.각 프로그램의 segment를 text data stack 이렇게 나누고 text 가 code segment(a,b의 제일 아래부분) data segement와 stack segment는 서로를 향해서 자라나게. 자랄 때 여유가 있게. 스택 쌓이면 쌓일수록 아래로 자라나고, data는 쌓이면 쌓일수록 위로 자라난다. data segment가 자라나는 부분은 malloc,calloc할때가 예이며 heap이라고 한다.

메모리가 dynamic하게 assign 되면 os가 관리를 해야한다.일반적으로 메모리 관리하는 방법. 메모리 사용을 keep track하는 방법에는 크게 두가지가 있다. bit map이랑 list사용
10페이지 위에있는 그림이 메모리를 펼쳐둔 것. 왼쪽이메모리의 낮은 주소 오른쪽으로 갈수록 높은 주소 . 가장 메모리 낮은 주소에 프로세스 a할당. 빗금친것은 free. 할당이 안된부분. b,c 할당. free, d,e 할당 ,free. 이런식.메모리 관리 방법. 두가지 하나는 bit map, list. bit map을 사용하는 방법이 그림 b. list 사용하는 방식은 그림 c. list를 사용해서 메모리 관리하는 방법. 그림b에 나와있는 bit map 사용해서 메모리 관리하는 방법부터 . 현재 메모리 상태는 위에.
메모리가 allocation unit으로 나뉘어져 있다. 빗금들이 있다.나눠진 마크가 allocation unit나타낸다. A는 allocation unit 5개. 그 다음에는 free한 부분 hole. allocation unit이 3개 해당. B는 allocation unit 6개 차지. 이것을 bitmap으로 어떻게 하느냐 가장 낮은 주소에 있는 allocation unit의 한bit가 mapping. 그다음 allocation unit은 두번째 bit. 그 다음은 세번째 bit. 그래서 해당 allocation unit에 프로세스가 할당되어 있으면 그 해당 bit는 1로 세팅. 해당 allocation unit이 비어있으면. bit map 의 비트는 0으로 한다. 이러한 메모리 상황이라면 11111(A)000(hole)11111111(B랑C) bit map을 쓰면 allocation unit의 크거나 작을 경우에 다음과 같은 특징이 있다. allocation unit이 작으면 작을수록, bitmap이 더 커질수밖에 없다. allocation unit이 작아지면, allocation unit 작게 만들면 개수가 많아진다 똑같은 메모리에 해당하는 allocation unit의 개수가 많아진다. unit 하나당 bit 하나가 할당된 bit map은 bit들이 많아져 bit map 자체가 커진다. bitmap 자체 메모리 더 많이 잡아먹는다. 크게 만들면(allocation unit)을 A같은 경우에 5개를 할당하면된다. allocation unit의 크기를 넓혀서 할당되는 allocation unit의 개수를 줄일 수는 있다. 그런데 그렇게 되면 a의 마지막 부분에 할당된 allocation unit은 빈공간이 좀 크다. A에 3개의 allocation unit만 할당하면 된다고 했을때, 마지막 3번째 allocation unit에는 a의 끝부분이 차지하고 남은 부분. 그 부분이 wasted된다. 메모리가 낭비될 수 있다. 사용되지 않는 부분이 커질 수 있다. 새로운 프로세스를 메모리를 할당해줘야함. 새로운 프로세스 크기가 allocation unit 7개가. 필요. 그러면 bitmap 방식을 썼다고 가정했을 때 A에게 어떻게 메모리를 할당해주냐. bitmap table에 처음부터 쭉 봐야함. 그래서, 새로 할당되는 프로세스의 크기가 a.u 7개가 필요하니까 bitamp의 처음부터 쭉 보면서 0이 최소한 7개가 연속되있는 부분을 찾는다. 그런 hole 찾으면 새로운 프로세스에게 할당. 그중 7개의 allocation. 메모리 사용을 keep track 하는 방법 두번째.
리스트를 사용. 이방법은 그림 c에 나와있다. 메모리 상태는 맨 위에 나와있는 것 그대로 사용. 이것을 어떻게 list로 관리하는가. linked list로 관리. 노드들이 쭉 연결이 되어있다. 첫번째 노드는 주소의 가장 낮은부분. 두번째 노드는 주소의 그다음 부분 이런거에 다 mapping. 첫번째 노드는 제일 낮은 주소의 프로세스 a를 나타낸다. 잘 들여다 보면 처음에 p는 process . 0은 프로세스의 시작주소. 그다음에 5 는 프로세스의 크기. allocation unit 5개가 할당되어있다. h는 hole . 5는 시작주소, 크기는 3 이다.빗금친 부분은 free한 공간 즉 hole이다.

list는 앞의 노드부터 뒤의 노드로 갈수록 주소가 점점 증가.sorting이 되어 있다. 그다음에 프로세스 X가 있는데 종료하는 상황이라고 가정. 프로세스 X가 A,B프로세스에 둘러쌓여있는 경우. A다음이 X그다음이hole. hole 그다음 X,B 순서. 그다음에 앞뒤로 hole인경우. 4가지 경우에 대해 x가 terminate(종료하는상황) a같은 경우 . list로 관리하면 , 종료하기 전에는 A에 해당하는 노드가 있고 걔가 가리키는 다음 노드가 X를 나타내고 그다음 X를 나타내는 노드는B를 나타내는 노드를 가리킨다. X가 없어지면 hole이된다. 프로세스 노드에서 hole 노드로 바뀐다. 그렇게 list update해줘야한다. 프로세스 노드 프로세스 노드 홀 노드인 그림 b는 X가 terminate되면 X가 terminate되면 hole이 되니까 두개를 merge해서 hole노드로 바꿔준다. C는 반대경우, D는 hole -procees-hole이기 때문에 하나의 hole 노드로 linked list를 update 해준다.

프로세스가 생성되는 상황이라고 가정, 생성되는 프로세스에게 빈공간,hole을 할당해줘야함. 몇가지 방법이 있다. 첫번째 알고리즘이 first fit
linked list앞에서부터 쭉 scan. 그러면서 새로 생기는 create되는 프로세를 수용할 수 있을만한 크기의 hole을 찾는다. 찾으면 거기서 그놈에다가 프로세스를 assign해준다. list를 scan하면서 제일 먼저 생성되는 프로세스를 수용할 수 있는 hole을 return해주는것. best fit은 말그대로 list 전체를 다 뒤진다. 생성되는 프로세스를 수용할 수 있는 hole중에 제일 작은 놈을 선택한다. 거기에 프로세스를 할당. 제일 잘 맞는 hole을 찾는게 best fit이다. worst fit. best fit 했을 경우에 제일 잘맞는 놈을프로세스에 할당해주다보니 남는 홀이 있는데 남는 홀은 쓸모가 별로 없다. 그런것들이 자꾸 생기니까 처음부터 사전에 예방하려면 가장 큰 홀을 할당해준다. 남는 홀도 커서 거기에 또 다른 프로세스를 할당할 수 있지 않을까. ->좋은 아이디어는 아니다

virtual memory. 그래서 옛날에 프로그래머들이 프로그램을 짜다보니까 자신들이 짠 프로그램이 커서 available 한 메모리에 들어갈 수 없게 된것. solution을 생각. 생각해 낸게 overlay 라는 방식.. 프로그래머가 프로그램을 조각을 내는 것 split. 조각 하나를 overlay라고 하는데 overlay들로 조각을 내고 그다음에 먼저 first overlay를 수행. 그게 끝나면 next overlay를 call. 이러면은 second overlay가 디스크로부터 메모리로 swapping 된다. 프로그래머에게는 굉장히 귀찮은 작업. 생각해낸 방법이 바로 virtual memory라는 방법. 그냥 컴퓨터가 알아서 관리를 해준다. 결과적으로는 operating system이 virtual memory를 사용하게 되면 프로그램의 일부를 main memory에 관리하게.나머지는 disk 실제 프로그램의 사이즈가 physical 메모리보다 커도 수행이 가능한 방식. os가 다 관리를 해준다. virtual memory system. virtual memory 방식은 multiprogramming 환경에도 적용이 될 수 있다. 즉, 프로그램이 여러개가 메모리에 loading이 될 수있다. 전체가 loading이 될 필요 없이. 피지컬 메모리가 있는데 프로그램 a,b,c,d 이렇게 프로그램이 loading된다고 봤을때 일부만 loading이 될 수가 있다. 프로그램a의일부,b의일부c의 일부 이렇게 각각이 loading이 된 상태에서 수행될 수 있다. multiprogramming환경에서 적용이 될 수 있다. 그래서 virtual memory system은 많은 virtual memory system이 paging이라는 기법을 쓰고있다. paging이라는 virtual memory 기법을 시작한다. 많이 사용되는 방법.

Paging
virtaul address의 개념 설명. 가상 주소의 개념 . program generated address이다. 프로그램이 MOV REG,1000 명령어. 주소 1000에 잇는 값을 레지스터에 넣는 그런 명령어. 이게 가상 주소가 되는 것. 가상주소는 결국 가상 주소 공간. virtual address space를 구성. 15페이지 그림. virtual memory system이 supprot되지 않는 그런 컴퓨터에서는 주소 1000이 바로 메모리 쪽으로 전달. 바로 메모리에 주소 1000을 access. virtual memory system인 경우에는 이 virtual address인 1000이 cpu로부터 메모리 바로 가지 않고 memory management unit에 전달. mnu 로 전달. mnu가 그것을 받아들인 1000을 physical address로 바꿔준다. bus에다가 출력을 한다. physical address가 메모리 access할때 사용. 16페이지 그림이 나와있다.

16page. program generate address. 즉 virtual address가 16bit address.즉 0부터 64k까지 virtual address가 access하는 공간 .virtual address space를 구성. 64k. 환경을 보여준다. 16bit address로 access. 이 computer는 physical 메모리는 32k. virtual address space의 반밖에 안된다. 그래서, 이러한 system에서는 프로그램이 64k 프로그램 돌릴수는 있는데 통째로 메모리에 loading할 수는 없다. 그런 상황. 일부만 loading 되면서 수행. 물론 프로그램의 core image 전체는 완벽한 copy 는 disk에 당연히 있다. 64k까지 가능. disk에 있는 코어 이미지는 필요할때마다 메모리 쪽으로 loading이 되는 상황. 이 virtual memory system에서 paging기법 쓰는 경우에는. virtual address space가 page로 .virtual page로 구성. 줄여서 page라고도 한다. physical memory 쪽은 page frame으로 구성되어 있다.page와 page frame은 같은 크기를 같는다 그림에서는 4k이다. 이경우에 virtual address space는 64k. virtual page 하나는 4k. 16개의 virtual page로 구성.그다음에 physical memory는 32k. page frame은 4k. 8개의 page frame이 존재. disk 쪽에는 complete core image가 있다. disk로부터 읽어오고, 또는 physical memory에 있는 page frame을 disk로 내보내고 이런것은 page frame 크기 단위로 이루어진다.unit이 transfer할 때 page unit으로 transfer. 책에 예제가 나와있다.

MOV R,0 0이 program generated address. virtual address이다. 이게 이제 mmu쪽으로 주어진다. 그러면은 mmu는 이 주소 0이 virtual address space 상에서 어디 부분에 해당하는지 0k-4k 사이. 가장 첫번째 page. page 0.. 에 해당된다는 것을 안다. 그리고 mapping에 의하면 page 2에 mapping.화살표를 따라가보면 page frame 2에 mapping (zero부터 시작한다고 가정) mmu가 그것을 안다. mapping information은 mmu가 가지고있다. 눈으로 보기쉽게 화살표로 확인. mmu가 이제 virtual address 지도 봤을 때 virtual page 0번 속에 있다는걸 알고 physical memory page 2에 매핑된다는 걸 안다.이 주소 0을 physical adress에 이 주소 0은 virtual page 0 의 가장 첫번째 byte. virtual page 0 가 physical memory 속에 page frame 2에 mappimng되니까 page frame 2의 첫번째 byte 가 될것이다. 그래서 결국은 mmu는 0이라는 virtual address를 physical address인 8192 즉 physical page frame 2의 가장 첫번째 바이트가 8192. 8192로 바꿔준다. 8192가 bus를 통해 memory 쪽으로 전달 memory 쪽에서는 8192 주소 전달. 8192를 메모리가 access. 이런 상황이 책에 예제로 나오는게 있는데

MOV R,32780 virtual page 8이니까. virtual page 8의 12번째 byte. 32780이. 명령어에서 32780이 12번째 byte이다. 그런데 mapping 정보를 보면 X로 되어있다. 아까와 다름. mapping이 아직 안되있음. physical memory에 해당 page가 아직 안들어와있다. physical memory에 존재하지 않는다. mapping 정보가 그냥 X로 기재. 그럼 mmu는 unmap mapping이 안되어있구나. cpu로 하여금 os로 trap을 하게 만든다. 이 trap을 page fault라고 부른다. 그러면 os는 어떤 작업을 하느냐. 잘 사용되지 않는 page frame을 선택. physical memory상의 page frame 중에 잘 사용하지 않는 놈 선택. 그 page frame 내용 디스크에 쓴다. 그다음에 지금 참조한 page 내용을 디스크로부터 읽어서 막 비어진 page framme에다가 넣는다. 즉 mapping 정보를 약간 바꾸고 trap 된 instruction을 다시 수행. 예를 책에서 들고있다. os 가 physical memory로부터 page frame 1을 선택. page frame 1 이면 4k-8k 얘를 내쫓을 page frame으로 선택. 그러면 그다음에, disk로부터 virtual page 8의 내용을 loading.이자리에다가. loading을 하고 mapping 정보를 약간 바꾼다. virtual page 1의 entry를 unmap으로 바꾼다. virtual page 1은 .. 지금 physical page frame 1을 추출대상 선택. 거기에다가 virtual page 8의 내용 disk로부터 읽어들임. 그다음에 얘를 가리키고 있던 원래 virtual page는 virtual page 1이다 (4k-8k) 얘는 더이상 mapping이 안되있는것. X로 바꿔준다. physical memory의 page frame 1에는 이제 virtaul page 8에 해당하는 내용이 들어와있다. mapping 정보를 X로 바꾼다. 또하나 mapping 정보 바꿀게 virtual page 8의 엔트리. 더이상 X가 아니고 1로 바꿔준다. 그래서 이제는 page frame 1에 mapping. trap을 발생시켰던 instruction인 mov r,32780을 다시 실행. 제대로 실행이 될것이다.

virtual page access하려고 mmu가 봤는데 그 페이지가 mapping이 안되어있다. 즉, physical memory의 page frame이. virtual page의 content를 담고있는 page frame이 존재하지 않는다. 그런걸 발견 ,cpu로 하여금 os로 trap하게 만든다. page fault 이다 그게. virtual page 6번을 access하는 상황이 되었다고 가정. virtual page 6번은 X라고 되어있다. 대응하는 physical memory 속에 page frame이 없다는 것. 이게 발견되면 trap이 발생되는데 그걸 바로 page fault. os속으로 들어가게 되고 os가 잘 사용하지 않는 page frame 선택,, 그것의 내용물 disk로 쓴다(쫓아낸다)지금 막 참조한 virtual page 6에 해당하는 내용을 disk에서 읽어옴. 그래서 쫓아낸 자리. 에다가 넣는다. 그다음에 mapping을 수정한다. trap을 발생시킨 명령어를 다시 수행하면 정상적으로 수행이 된다.

17page. 좀더 자세하게 mmu속에서 어떤 일이 일어나는지 실제로 메커니즘을 자세히 본 그림.mmu가 어떻게 해서 virtual address를 physical address로 바꿔주는지.
64k virtual address space가 있고 32kphysical memory가 있다. virtual address space에는 virtual page가 16개. physical memory에는 page frame이 8개. 이때 virtual page크기와 page frame 의 크기는 같아야한다. 실제로 일어나는 메커니즘 들여다보면. 17page. MMU
아래쪽으로 virtual address가 cpu에서 들어온다. 위쪽으로 만들어진 physical address가 출력되어 나감. 15page 그림에 cpu package안에 mmu가 있다. cpu로부터 virtual address가 들어오면 mmu가 physical address로 바꿔서 bus로 출력하면 memory로 전달이 된다. 다시 17page. 8196이라는 virtual address가 mmu속으로 들어왔다고 가정. 8196을 2진수로 바꿈. virtual address spaces는 64k 즉 16bit으로 addressing. 2의16이 64k. 16bit가 아래 그림이다. 가상 주소. 최종적인 phsycial address는 15bit 왜냐하면 32k physical memory address이기 때문이다. 15bit이 최종적으로 출력이 된다. 8196으로 매핑을 해야하는데 상위 4비트와 하위 12비트로 쪼개서 하위 12비트는 그대로 physical address로 copy하면 된다. 상위 4비트는 virtual page number에 해당하게 된다. 이걸 가지고 page table의 index 이경우에는 0010 이니까 10진수 2이다 위로 가면 2 에 대응되는 physical memory 속에 page frame number가 들어있다. 110 십진수로 6 이게 위쪽으로 들어간다. 상위 physical address 3bit으로 붙는다. 최종적인 physical address가 된다. mapping할 때 110을 위쪽으로 보내기 전에오른쪽에 present.absent bit을 먼저 봐야함.그게 1일 때 하면 된다. virtual page에 대응되는 content가 physical memory에 있다는 얘기. 1이있으면. 0이라는건 virtual page의 content가 physical memory에 없다는것. 0이면 -> page fault가 발생한다. 1이면 그걸 그대로 가진다. page table entry는 virtual page 개수만큼 16개가 있다. 010,001,110 이런게 10진수 2,1,6 에 해당. 16page가 high level에서 그린 그림 17page가 low level에서 그린 그림. virtual address의 상위4bit을 index로 쓴다

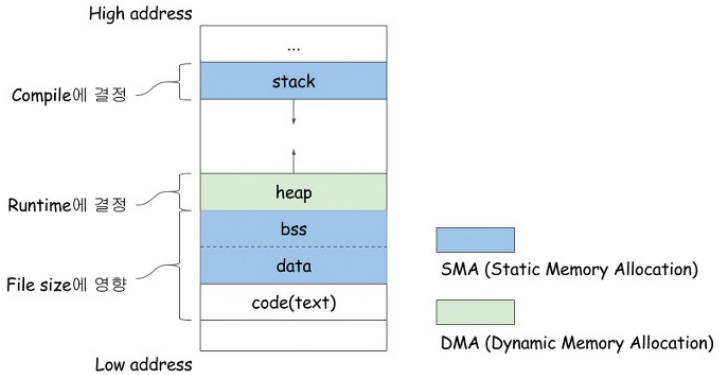
page table은 두가지 major issue를 가진다.
하나는 page table이 굉장히 커질 수가있다.
가상 주소 공간이 32bit-address space라면 2의 32승 바이트. 페이지 하나당 4kb가정. 4kb이면 2의 12승 바이트. 2의32승을 2의 12승으로 나누면 2의 20승 개의 page가 존재한다.. page table은 entry 개수가 page 개수만큼 존재해야하니까 entry가 2의20승entry 개. 32bit address space는 2의 32승 바이트 즉 4 기가바이트. 한 페이지가 4kb 2의 12승바이트. 2의 20승 virtual page 2의 20승 개 존재=약 100만개 page table이 굉장히 커진다.
64bit일때는 훨씬 더많이 필요 page table이 매우커진다.
또하나는 mapping이 빨라야한다.
virtual to physical mapping이 명령어에서 변수 주소나 프로시저의 주소라던가 그런것을 사용을 하게 될 때 결과적으로 메모리를 access. mapping이 필요하게 된다. 그러면 메모리 reference마다 virtual to physical mapping이 일어나야 하니까 page table 이용한 mapping이 빨라야한다.
두가지 디자인
빠른 하드웨어 register의 모임으로 이루어진 single page table로 둔다. 그게 바로 17page의 방법. 이게 mmu속에 page table이 덩그렇게 들어있다. 얘네들이 굉장히 빠른 hardware register의 모임이다. 이방법을 쓰면 mapping을 할때 mmu속에 page table을 access하기만 하면 되서 memory를 access할 필요는 없다. 15page. mmu속 page table access하는것이니까 memory를 mapping하는건 아니다. 이게장점. 단점은 모든 context switch. process switch 마다 page table을 full로 loading해야하는데 비싸다 시간이오래걸린다. context switch가 일어나면 이 프로세스 page table 내용에서 running상태의 page table 내용으로 바꿔야한다. loading을 다 해야함. 그게 expensive시간이 걸린다. 아쉬운점이다.
page table을 mmu속의 빠른 하드웨어 모임이 아닌 main memory에 둔다. 각 프로세스마다 page table이 다르기 때문에 각 프로세스의 page table이 main memory에 있다. 그래서 결국은 context swtich가 일어날 때 register 하나를 써서 register가 해당 running상태로 되는 process 의 page table의 주소를 가지게 한다(시작주소) context switch할때 loading 안해도 된다. memory 속에 존재하는 프로세스의 page table의 시작 주소를 single register가 가리키게 한다. 그것만 하면 된다. context stirch할때 fast memory map changer ㅏ 가능

각 명령어 수행 할 때 memory refernece가 발생. 즉, 변수나 이런 프로시저를 access할 때 결과적으로는 mapping이 필요한데, virtual to physical mapping이 필요한데 page table이 이 두번째 design에서는 main memory였다. memory 참조를 해야한다. 시간이 걸린다. 첫번째에서는 그런일이 없다. 직접 mmu속에 page table이 있었다.

Multilevel Page Table
100만개의 page table entry갖는것은 굉장히 부담스럽다. multilevel page table. page table이 여러 레벨에 걸쳐서 있음 2개의 level.
32bit address space 의 가상주소. offset은 12bit. 20비트를 다 virtual page number로 사용해서 한개의 큰 page table의 index로 쓰는게 아니고 이걸 다시 두개로 쪼갠다. 앞의 10bit는 first level 즉 top-level page table의 index로 쓰고. 그 해당 entry 찾아간다음에 거기서 다시 가리키는게 second-level page table이 된다. second-level page table에서 다시 두번째 field를 index로 사용. page frame number 가 있다. 한단계를 더 거치는것 2level. 이경우에는 process image가 있는데 가상주소 공간이 32bit 이니까 4기가바이트 address space를 자기의 image로 가질 수 있다. 보통 그런데 프로세스가 text,data,stack segement로 이루어져있다. text가 4메가 data가 4메가 stack이 4메가라고 가정. 그것을 two level로 표현하면 , Top-level page table의 맨 밑이 text . 그 위에가 data. 맨 위에가 stack. 맨 아래 text가 second-level page table의 1023을 가리키면 second level에서 1024개가 있는데 . 하나에 4kb 이다. 1024 곱하기 2의12승. 2의 22승 4 메가바이트를 담당. 텍스트가 4 메가바이트 segement이고 data가 4메가바이트 segement고 stack이 4메가바이트 segment라면 이렇게 entry가 1024개 짜리 4개만 있으면 끝. 4096개의 entry. one-level로 하면 백만개 (2의20승개) 차이가 엄청나게 크다. multilevel로 하면 모든 page table 을 entry들을 memory에 다 가지고있는걸 피할 수 있다. 참고로 잘못된것이 두번째(data)부분이 오른쪽 하얀색 을 가리키도록 수정. 각각이 4kb(page frame number) 4kb * 2의 10승. 4mb를 담당한다.하나하나가 4kb를 가리킨다고 볼 수 있다. 빗금쳐져있는 빈 공간. 원래 보통 process image가 data segment가 위쪽으로 자라나고 stack segement가 위쪽으로 자라난다. 딱 그 구조이다. process core image 완성. 결과적으로는 메모리에 들어가는 page table entry가 약 4000개로 줄어든다. page table entry가 간단하게 나온게 17page. 가장 핵심정보는 page frame number. present/absent bit. 그것을 좀더 자세하게 그려준게 21page. page table entry. page frame number 가 가장 중요. present/absent bit. protection bit = 어떤 종류의 access가 허용 . 0:read/write 1:Read only 그다음에 modified bit(dirty bit) : page가 modified 되었는지 나타내는 bit. 1번이라도 modify -> 1로 setting, 한번도 modify 아니면 clean page. 한번이라도 modify->dirty page. 명심하세요. referenced : page가 referenced(참조되었을 때 . 읽거나 쓰거나 할 때) bit 1로 setting. caching ~ 몰라도된다 마지막줄.

TLB or associative memory
대부분의 프로그램이 적은 개수의 페이지를 많이 여러번 참조한다.이것에 창안해서 나온것이 TLB 작은 하드웨어 장치. virtual address를 physical address로 mapping하기 위한. page table을 통하지 않고. 보통 mmu 속에 들어있다. 작은 개수의 entry 들로 이루어져 있다.23page를 보면 핵심정보는 virtual page number와 그것에 매핑되는 page frame number. 그것이 핵심. 먼저 mmu에 virtual address가 주어짐. 하드웨어가 virtual page number가 TLB 에 있는지 체크. 참고로 TLB는 129를 찾는다 하면 순차적으로 찾지 않고 병렬적으로 검색(associative memory라) 순간적으로 . 찾는다. 한번만에 찾는다. 그러면. match가 발견. protection bit에 위배되지 않으면. page frame이 TLB로 부터 직접적으로 취해진다. page frame 가져다 쓴다 ->mapping . page table로 갈필요 없고 TLB level에서 해결. 그런데 match가 발견되지 않으면, miss. 정상적인 page talbe look up. page table을 보는수밖에 없다. TLB entry에서 하나를 쫓아낸다. 거기에다가 지금 막 look up 한 page table의 entry를 쫓아낸 자리에 들여온다. 그 다음번에는 TLB 체크했을때 그게 있다. 그리고 참고로, entry가 TLB에서 쫓겨날 때 modified bit은 memory 속에 page table entry에 copy가 된다. memory 속에 page table entry에 고대로 copy가 된다. 그래서. 이렇게 하기때문에 23page 그림을 참고로 보면 protection bit로 추측. executable 한개 19,20,21 이다. 얘네들이 아마 program image 중에 execute 가능한 .즉 text segment 즉 코드 부분으로 추측. RW 860 861은 아마 stack segment에 해당하는 page로 보고 129 130 140을 data segment로 보는 듯 하다. TLB를 사용해서 작은 개수의 virtual page 참조하는 대부분의 프로그램. TLB에서 매치가 발생하는 바람에 실제 page table까지 참조 안해도 되니까 굉장히 mapping이 빨라질 수 있다.굉장히 빠르다.

multilevel page table 보충설명 : 초기에 2의20개의 entry를 가진 page table을 만들어야하는 one level과 달리 two level은 처음 실행에 필요한 main 함수를 위한 스택 부분, 처음 실행 코드가 적혀잇는 text부분 이렇게 두 부분에 대한 second page table을 생성해주고, 나머지는 사용해야할때 유동적으로 생성해주면 된다. page table의 크기를 줄일 수 있다. 처음 10비트 + first page 포인터로 얻는게 second table 포인터. second talbe 포인터 + 그후 10비트로 가서 얻는게 page frame number. page frame number랑 offset합치면 mmu에서의 주소. second table은 필요할 때 만들어서 사용 ,onelevel 사이즈 줄일 수 있다. 메모리 구조 중간에 빈 공간. 그 부분에 대한 page table 만들어 줄 필요 없으니까 second page table은 안만들어준다.



$1kB = 2^{10} byte$ $1MB = 2^{10} kB$.

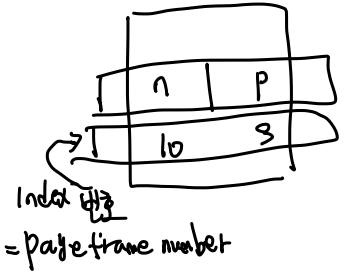
$2^{10} \times 4kB = 4MB$ $4MB = 2^{10} \times 4kB$.

기존의 page table은 page table이 굉장히 커질 수 있다. (걱정되는 점) memory를 많이 잡아먹을 수 있다. 그것을 해결하기 위한게 multilevel page table.
top level과 second level. top level 기본적으로 필요. second level. two level page table일 경우에. second level에 3개정도 page table 갖춤으로써 프로세스 시작할 때 page table 의 entry 수가 적었다. 그렇게 해결을 하려고 함. 또 다른 방법이 page table이 너무 크니까.
Inverted page table
32bit address space 예를 들었다. page 하나 4KB. 2의32승 바이트를 4KB로 나누면 2의20승 virtual page가 들어갈 수 있다. page table의 entry 개수도 그만큼 있어야 함.2의 20승 약 백만개. 64bit address space는 page table이 더 커진다.. 극복하고자 나온게 multilevel page talbe. first level second level해서 일단은 필요한 것 부터 page table을 할당. 프로세스 이미지가 text data stack. 즉 tet는 code부분이고 data segment. stack segment 3개로 이루어져 있는데 그게 전체 가상주소 공간에서 맨 아래 부분이 text. 그 위에가 data. 맨 위에가 stack. TOp level page table의 밑 두개와 맨 위가 담당. 가장 밑이 text 담당하기 위해서 second level table 가리킨다. 1024개가 있어서 page frame number들이 있다. text segment.하나하나가 4KB짜리 page 하나에 관한 page frame number을 가지고있음 1024=2의10승. 하나하나가 page table entry. 요기에 page frame number가 들어있다. 한 entry당 4KB page frame의 number. 2의 10승 곱하기 4KB =4MB 의 text segment에 대한 page table이라고 할 수 있다. 빗금친건 아직 사용하지 않는것. 두번째가 data segment. 하나하나가 page frame 하나의 정보를 각각 가지고 있다. 4kb짜리 결과적으론 4MB 담당. 맨위는 stack segement의 4MB 담당. 결과적으로는 page table entry개수를 4096개로 일단은 처음에 줄인 것. 프로세스 이미지가 전체 32bit address. space. 즉 4GB address space

에서 맨 하위 4MB 가 text. 바로 위 4MB 가 data segment 맨위가 stack segment. data segement와 stack segment는 서로를 향해서 dynamic 하게 자랄 수 있다. 원 테이블 single level은 백만개 정도.

Inverted Page Tables
에 나온 그림은 18page 의 64-bit address space 그림이라고 보면 된다. 2의 64승 나누기 2의 12승(page 하나가 4KB). 을 해야지 virtual page 개수가 나옴. 그거만큼 page table의 entry수가 있다 . 2의 52승 개의 page table entry가 필요하다. 그게 바로 24page 에 나온 그림. 맨 왼쪽에 있는 그림이 64bit address space에서 사용되는 page table 이다. entry개수가 2의 52승개. virtual page number가 들어옴 그것을 인덱스로 사용해서 access하면 거기에 page table entry가 있고 그 안에 가장 중요한 정보인 page frame number가 있다. single page table 사용할때 그림이 왼쪽. entry 개수가 2의 52개. 한 entry가 4byte 라고 하면 2의 52 곱하기 4byte의 entry 공간이 필요하다. 너무 메모리를 많이 차지한다. inverted page table은 page table 크기를 상당히 줄여주는 효과가 있다. 64-bit address space 인 system. physical 메모리가 256MB라고 가정. 2의28승 byte. physical memory가 256MB인데 page frame이 몇개 들어가냐. page frame의 크기는 virtual page 크기와 같으므로 (4KB) . 2의 28/ 2의 12승 = 2의16승 개의 page frame이 존재. 2의16 개의 4KB page frame. Inverted page table을 사용하면 Inverted page table의 entry개수가 2의16승개가 된다. 즉 page frame 개수와 같다. Inverted page table에서는 entry하나가 page frame당 entry 하나가 존재. 기존 page table은 virtual page 하나당 entry 하나. inverted page table은 page frame 하나당 entry 하나 존재. 칸 하나는 page frame(4KB짜리) . 밑에다가 inverted page table을 그려본다면 entry개수가 page frame과 같으니까 여기도 2의16승개의 entry가 존재. 기존의 page table에는 page table 메커니즘이라면 2의 52승 개의 entry. inverted page table에서는 2의16개만 entry가 존재. 어떻게 생긴거냐. 생긴 모양

page frame 당 한 entry 존재 ————Inverted page table .. 명심하세요. physical memory 256MB. page frame이 4KB. inverted page table의 entry 개수는 phycial memory의 page frame 수와 같으므로 2의16승 개이다 ,중요하다 . 기존의 page table에서는 entry개수가 64bit address space. 2의64byte 나누기 virtual page 하나의 크기가 4KB 이니까 2의12승 byte=2의 52승 개의 entry가 필요. Inverted page table은 2의 28승 나누기 2의 12승 =2의 16승개의 entry가 존재해야한다. 중요하다 계산하는 법 엄청 중요. Inverted page table을 그림. entry개수는 phycal memory의 page frame수와 같다. 2의16승 개. virtual page 3을 access하는 상황을 생각해보면. 기존의 page table에서는 virtual page 3을 index로 써서, 아래서부터 0,1,2,3 3번째 mapping. 여기 access하면 여기에 mapping 된 physical memory 속의 page frame number가 있다. 그러면 mapping이 된 것. page frame number를 찾았다.
그런데, 그러면 Inverted page table은 virtual page number를 가지고 어떻게 page frame number를 찾는가. 어떻게 찾느냐. virtual page 3이 주어졌으면 , 그걸 가지고 Inverted Page Table의 Index로 쓰는게 아니고, Inverted Page Table의 entry를 하나씩 다 뒤져본다. scan하면서. 그러면 entry들은 어떻게 생겼나 하면 entry를 하나 들여다보면 여기에 virtual page number가 이 안에 들어가있다. virtual page number를 p라고 하면 , 들어있다. 프로세스 number인 n도 들어있다.



virtual page 3이 주어지면 Inverted page table속에. 줄여서 IPT. entry 하나하나를 쭉 스캔하면서 그 안에 P field가 바로 page number이니까 현재 찾으려는게 3. 3을 가진 entry 를 찾는다. 정확하게 process number도 search. 프로세스 번호가 10이고 access하는 virtual page가 3이면 10과 3의 쌍. 으로 Inverted page table 을 쭉 하나하나 보면서 찾는다. 찾다보면 match 발생. 엔트리가 프로세스 번호 10 virtual page number 3이었다. 매치 발생하는게 있다고 치면 그것의 index 번호. (맨밑에서부터 얼마만큼 올라왔는가. index 번호가 바로 page frame number이다. 이게 바로 page frame number. Index가 예를들어서 17이다. 17이 page frame number. mapping이 끝난 것.

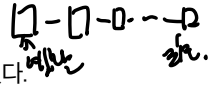
기존의 page table사용, virtaul page 3가지고 시작하면 page table의 index로 3 사용.밑에가 0 1 2 3 이니까 밑에서 4번째 . 찾아가면 page frame number가 있다. 예를 들어 17. page frame number 찾음. Inverted page table은 이렇게 한다. 그린 그림처럼 IPT를 scan한다. virtual page number와 프로세스 넘버까지 봐야한다 .Inverted page table이 entry하나가 결과적으로는 physical memory의 page frame하나에 mapping.대응. 실제로 physical memory에는 page frame하나하나를 보면 그 multiprogramming의 경우에 여러 프로세스의 page frame이 있을 수 잇다. physical memory 전체에 딱 한 프로세스에만 page frame만 있는게 아니다. 그렇기 때문에 physical memory의 page frame 그거 하나하나에 대응하는 IPT의 entry들은 process number까지 entry 정보로 가지고 있어야함. 이경우에는 virtual page number가 주어졌을 때 virtual page number가 3이다. process 번호가 10이라고 하면 10과 3을 가지고 IPT를 scan하면서 search, 매칭이 발생하면 그때의 index number를 본다. 이경우에는 index가 17. page frame number가 17이다. mapping이 끝난것. 거꾸로 매핑을 한다고 한다. 기존의 page table은 virtual page number을 page의 index로 써서 거기에 content를 보면 page frame number가 튀어나오는데 IPT는 거꾸로 virtual page number를 가지고 page table의 content를 search해서 matching.그 테이블의 index 값이 page frame number. Inverted Page Table 메커니즘.

참고로 , Inverted Page Table은 속의 내용을 다 봐야하니까 시간이 많이걸린다. 이것을 빨리 하기 위해서는 책에 제시된 아이디어가 TLB . 방법을 쓰는것이다. 좀더 빠른 속도를 갖고있는 TLB를 이용한다. Hash table쓰는방법은 참고로만. virtual page 3이 page frame 17에 저장되있던 상태. 그래서 기존의 page table로 mapping한 것과 그 IPT로 mapping하는 방법 두가지.
기존에는 IPT의 entry가 어떻게 만들어지는거죠> 즉 처음에 virtual page3의 content가 메모리로 올라올 때 page frame 17에 적재(loading이 되면 그때 single page table . index 3번에 (왼쪽 큰 직사각형) 요기 entry가 setup. 거기에 17을 메모리 frame number가 들어감. inverted page table은 어떻게 entry 세팅?
마찬가지로 virtual page 3이 처음 메모리에 로딩이 될 때(page frame 17에 로딩이 될 때) 결국 page frame 17에 로딩되니까 IPT의 17. index 17에 해당하는 entry에다가 setting을 한다. 그 entry에다가 virtual page number 3. process number 10.이렇게 index number 17인데다가 처음에 entry setting을 해놓는다. 그래야만 나중에 virtual page 3 search 할 때. page table 쫙 IPT content를 scan, 그러다가 matching이 발생할 때 index를 보면 17이 되어있을 것이다 . 그러면 실제로 17에 page frame 17에 virtual page 3의 content가 들어있는것이다. 그러니까 mapping이 IPT로도 가능하다.

참고로 해쉬 테이블은 아주 짧게 말하면 search하는데 시간이 많이 걸림 TLB 사용가능. TLB에도 올라오지 않은것은 TLB miss가 발생하고 그다음에는 hash table의 도움을 받자. 기존의 Inverted table 방식 쓰지 말고 hash table이라는 것을 둬서 entry를 조절할 수 있는데 지금처럼 2의16승개로 조절, 뭘로 indexing이 되냐 하면 virtual page에 대해서 hash 값으로. hash 값이라는게 hash 함수를 돌렷을때 값. 처음에 virtual page 3이다. 3이 있으면 걔를 hash function에 입력. 결과값이 나온다. 그러면 예를 들어 function의 값이 8이다. 8엔트리가 중간이라고 하면 여기다가 virtual page 3의 entry를 뒤에다가 붙인다. virtual page number 3 이 뒤에 들어간다. page frame 값을 17. virtual page가 3이면 메모리에 로딩할 때 virtual page 값 3을 hash funtion에 입력하고 그 나온값이 8이면 hash table index 8에다가 노드를 하나 붙인다. 노드의 앞이 virtual page. 뒤에가 page frame field. 앞이 3 뒤에가 17. 한번 넣어두면 나중에 page fault. mapping을 하기 위해 virtual page 3이 주어짐 .그것을 가지고 page frame 17을 찾아야함. 그것을 어떻게 찾느냐. 다시 virtual page 3 가지고 hash function에 입력. 그럼 어떻게 되는가? 8이 나온다. 인덱스 8을 찾아간다. 8에 이미 virtual page 3. 그것에 해당하는 page frame 17 정보가 달려있다. 그래서 page frame number 17을 알아서 access. 참고로만. 같은 virtual page number에 대해 같은 hash 값이 나올 수 있다. 그래서 이제 같은 hash 값이 나오면 linked list로 연결. 실제 virtual page에 해당하는 page frame 따라가면서 찾는다. 참고로만 알고있자 이해안되도 걱정 ㄴ

Page Replacement Algorithm.
Page fault가 발생하면 os로 trap이 일어난다. os는 해당 page를 디스크로부터 메모리 속으로 불러들여와야함. 그러기 위해서는 먼저 메모리에 있는 페이지 하나를 내보내야한다 메모리가 꽉차있다면. 어느 페이지를 내보낼것인가 하는 문제. 새로 들어오는 page를 disk에서 불러오기 위한 공간을 만들어야한다. 어느 페이지를 제거할것인가. 그것을 결정하는 알고리즘이 Page 교체 알고리즘. 내쫓는 페이지가 memory 로 처음 올라온 뒤에 modified 되었으면 내쫓을 때 그 내용을 disk에 써야함. save. 내쫓기는 page가 modify가 안되었다면 , save 할 필요는 없다. 그냥 새롭게 들어오는 page가 위로 overwrite. 마지막에 나오는 얘기는 내쫓을 때 자주 사용되는 page는 선택하지 않는 게 좋겠다. 아마도 곳 사용될 필요가 있을 것 같으니까. 여러개 알고리즘을 소개

optimal page replacement algorithm 실제로 쓰이는 알고리즘은 아님. optimal 말그대로 최적화된 알고리즘. 기준점으로 사용될 수는 있다.
어떤놈을 내쫓냐. 미래에 가장 먼 시점에 필요되는 page를 내쫓는다. 메모리에 page frame들이 있는데 그중에 미래에 가장 먼 시점에 필요로 되는 page를 내쫓는다. optimal 하지만 unrealizable. 한번 수행을 해봄 process를. process가 명령어들을 수행하면서 virtual page들을 access. 그것을 다 기록으로 남김. 수행을 끝내고 log가 만들어짐. 두번째 수행할 때 그 log를 이용해서 내쫓는다. 두번째 어떤 instruction 수행하는데 page 3을 access. 근데 page fault발생 .memory에 있는 놈 중에 내쫓는다. memory에 page frame이 꽉 차있다고 가정. page frame이 10개 들어가는 메모리라고 가정. log를 보고 가장 먼 미래에 memory에 존제하는 page 중에 가장 먼 미래에 사용될 page 내쫓는다 log 보고. 두번째 수행할 때 optimal하게 page 교체 . 실제 시스템에서는 page fault 발생하면 바로 쫓아낼 것 결정해야하는데 이건 그렇게 할 수 없다. 두번째 돌릴 때 알 수있다. 다른 알고리즘. 여러가지 알고리즘의 성능 비교할 때 사용.



하나하나가 다 중요함 여기서부터는 실제로 사용 Not Recently Used Page Replacement Algorithm 최근에 참조되지 않은 page 교체하겠다.
NRU . 최근에 사용되지 않은 page를 내쫓겠다 교체하겠다. 일단 각 page는 .. 21page를 보면 R bit 과 M bit이 있다 referenced bit, modified bit.. R bit은 페이지가 referenced 될 때 (read.write)될 때. R bit가 1로 세팅 M bit은 페이지가 modified(Written)될 때 1로 세팅. 주기적으로 R bit를 clear. clock이 발생할때마다 R bit을 clear한다. 최근에 참조되지 않은 page를 최근에 참조된 page랑 구별하기 위해서. R bit를 주기적으로 clock이 발생할 때마다 clear한다.0으로 세팅. page fault가 발생하면, page를 4개로 분류한다. 가장 위의. 클래스중 page중 하나를 랜덤하게 내쫓는다. 맨 위의 클래스가 텅 비어있으면 그다음 클래스의 페이지중 하나를 내쫓는다 . 맨위 클래스가 쫓겨날 확률 높다. page fault가 발생. page를 4개로 클래스로 나눈다 책에서는 0,1,2,3 으로 class 이름을 붙임. NRU 는 가장 첫번째 클래스부터 내쫓는다 R bit 0 M bit 0. 가장 먼저 쫓겨날 운명. 그다음은 참조되지 않았고, modified. 그다음에 referenced, not modified. R bit은 1 M bit은 0 read는 되고 write는 안된것. 가장 마지막 class가 R bit 1 M bit 1. 이 알고리즘에서 포인트. 최근에 사용되지 않은 놈을 내쫓겠다.가장 최근에 사용된 것은 맨 아래 class라는게 이 알고리즘 컨셉. R bit과 M bit으로 판단. 2,3 번은 순서가 애매하다.. 사실상 M 이 1이되면 R도 1이된다. 그런데 어떻게 이게 가능하냐. 아까 얘기했지만 이 알고리즘은 주기적으로 R bit을 0으로 리셋. 그래서 가능. 자주 사용되는 깨끗한 페이지는 (modified 되지 않은 페이지) 그것보다는 최소한 한 clock tick 동안에 참조되지 않은 modified page 즉 dirty page를 내쫓는게 낫다. 가장 최신의 clock tick이 딱 떨어진 다음에 이 3번 class는 reference가 한번 되었다. 2번은 not referenced. modified인데 옛날에 modify 되었을 때 r 이 1이었을 것. not reference라는 건 가장 최신의 clock tick 이후에는 0. 참조가 안되어있다. 밑에 있는건 가장 최신 clock tick 이후에도 참조가 되었다. 그렇기 때문에 NRU에는 3을 더 살려준다. 적당한 퍼포먼스를 낸다. clock tick- clock interrupt가 걸릴때마다 R bit clear .
FIFO. FIFO page Replacement Algorith,. First in First out. 먼저들어온 놈을 먼저 내보낸다.page 중에메모리에 가장 먼저 온 놈,, 가장 옛날에 loading된 놈을 page fault 발생했을 때 가장 먼저 내쫓는다. linked list로 page관리하면 된다. 메모리에 올라온 순서대로. list 에 맨 앞쪽에는 가장 옛날에 올라온 page. linked list 의 맨 끝에는 가장 최근에 올라온 page가 있다.page fault가 발생하면 list 의 맨 앞에 있는 page를 교체. 메모리에 올라온지 가장 오래된. FIFO. 그게 바로 FIFO 알고리즘. 그런데 이 알고리즘 에서 단점에 대해서 마지막에 나와있다. 메모리에 올라온지 가장 오래된 애가 굉장히 자주 사용되는 page일 수도 있다. 현재에도 자주 사용되는 page라고 했을 때. 효율적이지 못하게 된다.

Second Chance algorithm
FIFO 알고리즘 조금 더 개선.일단은 FIFO 처럼 list로 관리. 메모리에 로딩된 순서로, 위에있는게 loading time. 0,3,7,8… 값이 증가할수록 최근으로 오고있는것. A같은 경우에는 loading time이 0 B는 3. H가 가장 최근에 loading. 현재시간이 20인데 page fault가 발생. 그러면 먼저 맨 앞에있는 page를 본다. page A. FIFO였으면 걔를 내쫓음. Second chance는 page A의 R bit를 체크한다. 그래서. R bit가 0이다. FIFO 처럼 추출. dirty 페이지면 disk에 쓰고, clean 페이지면 disk에 쓰지 않는다 dirty 페이지는 메모리에 loading 된 이후에 modified 되었다는 얘기. 그리고 clean page면 modefied가 안된 page. R bit가 0이면 내쫓는다 목표 달성. R bitdl 1인 경우, Second Chance. 내쫓지 않고 기회를 한번 더 준다. A를 리스트의 맨 뒤로 보낸다. 그림 b처럼 리스트의 맨 뒤로 보낸다. loading time은 20. page fault가 20에 발생. 그러면서 R bit는 clear. page A의 R bit는 이제 더이상 1이 아니고 0으로 바뀐다. 그다음에 search를 계속 한다. B를 가지고 똑같은 작업을 한다. B의 Rbit이 0이면 알고리즘 목적 달성하고 끝. B가 R bit이 1이다 그럼 얘도 뒤로 보낸다. 현재시간20을 loading. B의 R bit을 0으로 clear. 결과적으로는 R bit가 0인놈 찾으면서 추출하면서 끝난다. 모든 게 다 Rbit이 1이면 ? R bit이 1이라는 건 가장 최근 clock tick 이후로 참조가 되었다는 의미 굉장히 최신에 참조되었다. 그래서, 그러면 이제 모든 게 뒤로 붙으면서 다시 순서가 그림 a와 같은 순서. R bit이 다 0이 되었으므로 A 를 추출한다. 이 알고리즘의 단점은 효율적이지 않다. 노드를 맨 앞에서 지우고 뒤에 붙이고.. list update 하는 것이 시간을 잡아먹는다. 비효율적이다. 그 단점 극복하고자

Clock 알고리즘 Clock Page Replacement Algorithm
Clock 처럼 circular list로 관리한다. 이제 화살표가 C를 가리키고 있는데 가장 오래된 페이지이다.그래서, 그다음에 이제 D가 그다음 오래된것. 이 순서에 따라 시계방향으로 가면서 점점 더 최근으로 간다. 이렇게 관리를 하는데, page fault가 발생. 화살표가 가리키는 페이지가 가장 오래된 페이지 메모리에 로딩된지. 걔의 R bit을 본다. Second chance alg.와 같다. R 이 0이면 추출한다 Evict the page. 목표를 달성 R 이 1이다. R 비트를 clear 하고 화살표를 하나 앞으로 진행. C 에서 R bit이 1이면 C를 바로 추출하지 않고 R bit을 0으로 clear 하는 대신에 화살표 하나 진행시켜서 D 가리키게 한다. Second chance와 차이는 linear-circular list 차이. Second 에 비해 list 관리가 용이. 화살표(포인터)만 바꾸면 된다. 앞의 Second와 같은 데 implementation 부분만 다르다.

least recently used (LRU) 가장 최근에 사용되지 않은. 가장 옛날에 사용되었다.
가장 예전에사용된 페이지를 추출하겠다. 어떤 개념에서 출발 ? 최근에 사용된 페이지는 곧 다시 사용될 것이다. 가장 예전에 사용된 페이지 내쫓겠다. 구현하는 가장 간단한 방법 메모리에 있는. page에 linked list 를 관리하는 것. 가장 최근에 사용된 page를 맨앞에. 가장 예전에 사용된 page 맨 뒤에 오게. 그런식으로 계속 update. memory reference가 발생할때마다 해당 페이지 노드를 linked list 의 맨 앞으로 가져온다. 항상 맨 앞에는 가장 최근에 참조한 페이지가 오게. 그러다가 page가 fault. 리스트의 가장 맨 뒤에있는 애를 내쫓는다. 가장 예전에 사용된 페이지일 테니까. software적으로 단순하게 생각하면. 하드웨어의 도움을 받아서 하는 방법. 두가지 방법. HW 1번 방법. 뒤에 나와있는게 HW 2번 방법. HW 는 hardware.
Hw 도움받는 첫번째 방법
각 table entry의 counter field를 둔다. page table이 있으면 entry들이 있을텐데 field를 하나 둔다. counter field. 여기에는 counter 값 저장. 각 page의. system에는 counter c라는 counter register 가 있다. 이런 상태에서 명령어 하나 수행할때마다 유일한 counter register 값이 1 씩 증가. memory reference 가 일어날 때마다 현재의 counter 값이 지금 참조한 page의 . page table entry가 있을텐데.. 그 놈의 C field에 저장이된다. 예를 들어서 , 이 page table entry 라고 치면. C 필드 부분에 현재 counter 값이 저장이 된다. 메모리 참조가 일어날 때마다 이 counter register의 값이 해당 page. 참조한 page의 page table entry 속에 counter field 에 저장이 된다. 그러다가 언젠가 page fault가 발생. OS 가 page table의 counter 값을 다 체크. 다 scan해서 가장 최소값을 갖는 애가 가장 least recently used. 가장 예전에 사용된 페이지 . 걔를 내쫓는다. counter field 값이 가장 작은게 가장 예전. counter field 값은 counter 값에서 copy. counter 값은 instruction 수행할 때마다 증가.

LRU 하드웨어 도움 받아서 하는 두번째 방법. 이것은 이제 다음과 같다. m 개의 page frame 이경우에는 4개의 page frame을 예로 들었다. 이때 m*n bit의 matrix를 관리한다. 4개의 page frame이 있는 system. 4*4 bit를 갖는 matrix. 하드웨어적으로 관리. 그 방법을 설명. page가 이순서대로 참조. 01,2,3,2,1,0,3,2,3
처음에 page 0을 참조했을 때는 4*4 행렬에서 0번 행을 1로 다 세팅. 0번 0을 0으로 다 reset. 처음에는 matrix가 0으로 초기화. 그다음에 page 참조가 1이다. 1번행을 다 1로 세팅. 1번 열을 0으로 clear. 2. 2번 행을 1로 세팅 2번 열을 0으로 clear. 이런식으로. 마지막으로 3. 3번 행 1로 세팅. 3번 열을 0으로 clear. 중간에 page fault가 발생할 수 있다. 마지막 그림 j를 하고 났는데 page fault가 발생했다고 가정 . 그러면 , 뭔가 하면 행들의 값을 본다. 그 값이 제일 적은 놈이 least recently used page이다. 왼쪽을 보면 페이지 0부터 3까지이다. 순서대로 0100,0000,1100,1110 이다. 두번째가 제일 작다. 페이지 1을 결과적으로는 추출하겠다.
LRU를 software로 simulation하는법. 가장 정확히 하는 방법은 아까 7페이지에 must keep~ 이부분. 가장 최근에 사용된걸 앞에, 가장 적게 사용된것 뒤에. 매번의 메모리 참조마다 업데이트 리스트. 이것은 시간이 많이 걸려서 부하가 많이 걸린다. memory reference마다 list를 update. 근사치를 제공하는 software적인 방법

NFU 부터 얘기를 하면, Not Frequently Used (NFU 써서 LRU 흉내 첫번째)
각 페이지 마다 software counter 부여. 처음에는 0으로 세팅. 그래서 page가 여러 개 있으면 그거마다 entry가 있다. 그러면 software counter 값이 들어있다. 처음에는 0. clock interrupt 가 발생할때마다 operating system은 메모리에 있는 모든 page scan. page에 page table entry가 있는데 그것의 R bit을 본다. 0이나 1이다. 그것을 해당 page의 software counter. 예를 들어 페이지 3이라고 가정. page 3의 counter 값을 그 페이지 3의 R bit 값과 더한다. 그래서, 그렇게 하는 것. 메모리에 있는 모든 page scan. 하면서 그 페이지의 page table 상의 entry의 R bit 값을 software counter . 각 page의 software counter 값에다가 더한다. counter 값은. keep track. 각 페이지가 몇번 참조되었는지를 keep track하는것이라고 볼 수 있다. 그래서 page fault가 발생하면 , counter 값을 보면서 가장 작은 page를 교체한다. 결과적으로 counter값은 페이지의 참조 횟수를 나타낸다. 참조 횟수가 가장 참조를 덜 한 페이지를 내쫓겠다. 사실 LRU 원래 취지는 그게 아님. 가장 옛날에 참조된 페이지 내쫓겠다는 건데. software 로 simulation 하는 방법인 LRU는 Not Frequently USed. 즉 참조 횟수가 적은. 자주 참조되지 않은 그런 page를 내쫓겠다. 철학적으로 LRU와 같지 않다. 근사치 를 제공하는것. simulate라고 표현. Never forgets anything이 단점이다. 어떤 프로그램이 시작해서 수행될 때 , 시작 초반에는 어떤 페이지를 많이 참조. 여러번. 걔는 NFU 알고리즘이라면 . 결과적으로는 걔는 초반에 굉장히 많이 참조-> software counter 값이 굉장히 커져있다. 초반에 굉장히 많이 키워둠. 그런데 그러다가 중반 후반으로 가다가 page fault 발생. 중반 후반으로 가면서 전반에 많이 사용되긴 했지만 중반에는 사용 안됨. 중후반에 page fault가 발생하면 software counter값이 적은 애가 쫓겨나는데 방금 말한 얘는 초반에 counter 값을 올려놔서 쫓겨나지 않는다. 그런데 결과적으로 얘는 중후반에는 사용되지 않는 page라고 가정. 초반에 잔뜩 사용.. 그러면 중후반에 page fault 가 발생하면 얘를 내쫓아야함. 중후반에 와서는 사용되지 않고 있었으니까. 그런데 NFU 알고리즘은 그냥 몇번 프로세스가 사용되었는지 횟수만 counting. 사용횟수를 굉장히 높게 축적. 살아남는다.

두번째 Aging. 방법 software적으로 LRU 흉내
page가 지금. 6개가 있다고 가정. page 0부터 5까지 있는데 R bit를 둔 것. 시간이 언젠가 하면 clock tick이 발생한 후에 R bit 봤을 때 이렇단 것. 그림(a). 그러면 Aging 알고리즘은 첫번째 clock tick이 발생했을 때 각 page의 R bit를 모아둠. 그럼 이걸 보고 각 page 마다 counter가 있다. counter의 비트를 오른쪽으로 한비트 shift하고 왼쪽에다가 R bit 값을 공급을한다. 101011 이 순서대로 들어간다. 왼쪽에다가 (그림 보고 이해) 두번 째 clock tick 발생했을 때도 그때 각 page 6개의 R bit를 왼쪽에 넣는다. 넣기전에 오른쪽으로 하나씩 shift 를 해줘야함. 그럼 page 0입장에서 보면 11000000. 이런식으로 clock tick 발생할때마다 각 page 의 주어진 counter 값을 오른쪽으로 한 bit shift하고 왼쪽에 그 page의 R bit 값을 집어넣는다. leftmost bit. 그러다가 page fault가 발생. 각 page의 counter 값을 보고 가장 작은 counter 값을 가지는 놈을 추출. 마지막 그림 e를 보면 shift한다음에 값을 넣었을 때 그림 e인데 page fault가 발생. 이 알고리즘에 의하면 counter 값을 비교. 가장 작은 값을 찾는다. 이중에 제일 작은애는 3번. 이놈을 추출.Aging방법. LRU 100프로 구현한것이 아닌 흉내낸것. 이정도만 알고있으면 된다.

Working Set 알고리즘
Demand Paging이라는 컨셉. Demand Paging은 page가 demand 가 발생할 때 이런 요구가 발생할 때 loading. not in advance 미리 로딩이 되지 않고. process가 처음에 시작이 되면 메모리에 page가 하나도 로딩이 안된 상태로 시작. 그래서 cpu가 첫번째 instruction fetch 가져오려고 할것. 그러면 page fault가 발생. 아직 instruction을 담고 있는 page조차도 메모리에 안올라와있을 때니까. page fault 발생. 첫번째 instruction 담고 있는 page를 메모리로 들여온다. 그렇게 수행이 되면 시간이 어느정도 지나면 프로세스는 필요로 하는 대부분의 페이지를 이제 메모리에 가지게 된다. 안정이 된다. 상대적으로 page fault 횟수가 적어진다. 그런 상황이다.

Locality of reference 참조의 지역성. 중요한 개념중의 하나 .프로세스는 상대적으로 작은 fraction의 page들을 참조를 한다. 그 수행의 어느 단계에 있어서도 page의 일부. 그 페이지들을 참조 한다. 모든 페이지들을 훑어가면서 하는게 아니라. 어느 순간에 있어서 during any phase of execution 어느 단계를 보더라도 일부분의 페이지들을 참조하는. locality of reference 참조의 지역성.

Working set 굉장히 중요한 개념
프로세스가 현재 사용하고 있는 페이지의 집합이다.

Thrashing.
명령어 두세개 수행 할때마다 page fault를 발생 시키는 프로그램을 thrashing 하고 있다고 말함. 이건 언제 발생할 수 있느냐 available한 메모리가 너무 작아서 working set 전체를 다 hold를 못할 때 ..

Working set model(Prepaging)
많은 paging system들이 각 프로세스의 working set을 keep track. 그래서 해당 프로세스를 running상태로 두기 바로 직전에 memory에 해당 프로세스의 working set을 미리 loading을 미리 loading을 쫙 해둠 . 그다음에 running을 한다. 단 working set이라는것은 시간에 따라 변할 수 있다 , 그것을 명심

Working set을 수학적으로 표현
시간 t에 있어서 k개의 가장 최근의 메모리 참조. k개의 가장 최근의 메모리 참조에 의해서 사용된 page의 집합. w(k,t) 이게 working set. x축이 k이다. k가 1,2,3,4, 이런식으로 간다. 예를 들어서 most recent memory reference의 개수가 k. 1은 most recent memory reference를 1개로 잡은 것. k가 1. 가장 최근에 메모리 참조한것. 2는 가장 최근에 참조한 메모리 그 전에 참조. 가장 최근 두개의 메모리 참조.. 그때 각각에 있어서 사용된 page 집합. 그게 w(k,t) 값이다3은 가장 최근의 메모리 참조 3개 . 그때 그거에 의해서 사용된 page 의 집합이 w(k,t) 이런 식으로 증가를 할수밖에 없다 뒤에게 앞의 것을 포함. page의 집합 working set 시간 t에 있어서 .. 그래프는 수렴할 수 밖에 없다. virtual address space 상에서의 page 개수는 finite 유한하다.

무엇보다 제일 중요한 건 Working set이라는 개념. 프로세스가 현재 사용하고 있는 page의 집합. 굉장히 중요함.
Working Set 알고리즘. 기본적인 전략. page fault가 발생하면 working set 에 있지않은 페이지 찾아서 쫓아낸다. current virtual time 현재 가상시간 영어로 외우는것이 중요해요. 프로세스가 실제로 시작된 이후에 실제로 사용한 cpu시간. current virtual time. 매 clock tick 마다 R bit를 clear. 모든 page fualt 에서 page table 을. scan 하면서 쫓아낼 적정한 page 찾는다, 좀더 자세하게 그림 13page.
어떤 프로세스가 수행중이다가 page fault 가 발생. 프로세스의 current virtual time이 표시 2204. 시간이 흐르면서 값이 증가.page fault 발생했으니까 이 알고리즘은 page table 을 스캔한다.그러면 entry 들이 등장. page table 의 한 entry는 두칸. 첫번째 칸은 사용하고있는 게 거의 아니라 표시 안했고 밑의 칸에 중요한 filed들. 하나는 R bit. referenced bit . 나머지 하나는 TIme of last use 마지막으로 가장 최근에 사용한 시간. 그래서 R bit 이 1인것과 0인 entry,. 1인것은 가장 최근의 clock tick 이후로 참조됨.0이면 가장 최근의 clock tick 이후로 참조가 안됨. process가 page fault 발생시킴.page를 스캔하면서 R bit를 본다.R이 1이면 해당 page의 Time of last use를 current virtual time 값으로 overwrite. 그리고 R==0이고 age>타우 이면 이것을 Working set 에 있지않은놈으로 보는것. 이 페이지를 쫓아냄. Working set이라는게 프로세스가 사용하고 있는 page 의 집합. 그런데 여기서 그런 개념은 보이지 않는다. Working set 자체를 implementation하기는 복잡.시간적 개념으로 근사화. working set에 근사하는 개념으로어떤 특정 시간. 현재로부터 거꾸로 가서 과거의 어느 시간까지. 그만큼의 interval를 타우로 본다,age=current virtual time-그 페이지의Time of last use. 즉 그페이지가마지막으로 사용된 이후로부터 흐른 시간. age가 타우라는 인터벌보다 크면 좀 오래된것, workingset에 있지않다고보고 쫓아낸다. 단, scan 은 계속.끝까지한다.R==0인데 age가 타우보다 작거나같은경우 working set에 들어가있다. 시간적 개념으로보면.age<=타우이면 working set에 있다고 보고 아직 쫓아내지 말고 smallest time of last use 기억.다시말하면 greatest age기억. 예를들어서 타우보다 age큰놈 못찾았으면 age가 타우보다 작거나같은 놈 중에 time of last use 가 가장작은, 즉 age가 제일 큰 놈을 쫓아낸다. 다끝까지 훑었는데 R==0이 하나도없다. 그럼, random 하게 하나를뽑는다. (dirty한 page말고 clean한. page.를 내쫓아라 dirty는 modified .page. 메모리에 로딩된다음에 수정된적이 이씨는 페이지. clean 한 페이지는 not modified. 로딩된다음 수정된적 없음.dirty page 를 내좇을때는 그 내용을 또 disk에 써야함. 시간이 걸리고 번거로움.
그다음 17페이지 Segmentation으로 건너뜀
요즘 다루는 virtual memory의 paging기법. virtual address space를 하나 생각. 한개의 virtual address space 관리 0부터 최대 주소까지 상당수 상황에는 두개이상의 virtual address space 가 존재하면 굉장히 유리. 이때 하나하나의 virtual address space. 독립된. ->segment. segment하나가 virtual address space.독립된 address space하나. segment의 크기는 0부터 Maximum. segment마다 segment 의 크기는 달라질 수 있다. segmentation 의 필요성 얘기하면서 하나의 예제를 들었음.
compiler 프로그램이 기존에 virtual address space하나인 paging기법에서 사용되었을 때는 컴파일러 수행할 때 필요한 symbol table constant table parse tree source text call stack 이런것들이 virtual address space가 하나밖에 없으니까 존재를 하다가 혹시라도 symbol table이 많이 커지면 상대방 쪽 침범.부딪힐 수 있다 불편. 하나의 virtual address space는 불편. segmentation기법 설명
각각의 독립된 segment에 있다. 각각 독립된 address space. 자라나도 부딪히지 않는다. 19page. segmentation과paging의 개념을 비교. 프로그래머가 해당 테크닉이 사용되고 있는 것을 알 필요가 있느냐. paging은 no. segmentation은 yes. 프로그래머가 code는 이 setment. 저쪽 segment에는 data. 이런 컨셉을 프로그래머가 알고 있는것. 얼마나 많은 선형 주소공간이 필요하냐. paging은 하나. virtual address space 하나. segmentatioon은 그런게 여러개있을 수 있다. 전체 주소 공간이 physical memory의 크기를 초과할 수 있느냐. paging은 yes. segmentation도 yes. 프로시저와 데이터가 구별되어서 따로 protect될 수 있는가 paging은 no segmentation은 yes. segmentation마다 따로따로 넣어넣고 segmentation마다 protecgtion 따로 줄 수 있다. code가 들어있는 segment에는 executable protection. data쪽. segment에는 read/write가능하게 이런식으로. table이 크기가 커졌다 작아졌다 하는것을 수용하는가 paging에서는 no. segmentation은 yes. 사용자간의 프로시저의 sharing이 잘 support? paging은 no segmentation은 yes. 공유하려는 프로시저를 특정 segment에 넣어두고 사용자들의 프로세스들이 특정 세그먼트를 access하게 하면 된다. 왜 이테크닉이 발명되었느냐. paging의 경우에는 더 많은 physical memory를 살 필요 없이 큰 선형 주소 공간을 얻기 위해서 . segmentation은 program과 data. 즉 code와 data 가 논리적으로 독립된 주소 공간에 들어가게 하고 sharing과 protection을 도와주기 위해서.

filed.sementation
메모리 있는 그냥 그대로를 segmentation 쪼개서 쓰는것.a같은 경우에는 segment 5개 메모리 차지. 그림 b를 보면 segment 7이 들어오기 위해 segment 0이 빠짐.빈공간 hole이 생김. segment 5가 들어오기 위해 4 가 나감 그림 c. 3K짜리 hole. segment 6이 들어오기 위해 3이 나감. 4K hole 그림 d. 이런식으로hole들이 생기는 것을 checkerboarding. external fragmentation이라고 한다. 외부 파편.checkerboarding이 못쓰는 작은 공간이니까 이것을 합치기 위해서 segment를 다 한쪽으로 모은다. hole 들을 크게 만드는. 사용할 수 있게 hole들을 크게->compaction.

Segmentation with Paging:Pentium
Pentium은 seg,paging 둘다 제공 할 수 있다. 하나만 사용 가능. 둘다 동시에 사용할수도 있다. 대략적인 그림. 21page logical address를 physical address로 바꿔야 함. logical address는 selector 와 offset으로 나뉘어져있다. 그래서 selector가 descriptor table의 index로 사용 이 안의 한 entry가 segment descriptor. 아주중요하다 명심하기 segment deescriptor에는 segment의 시작 주소같은 게 있다. 그것을 offset과 합하면 linear address가 나온다. linear address가지고 paging 기능이 꺼져있을 때는 이것 자체가 physical address가 된다. paging 기능이 켜져있으면 linear address가 virtual address로 취급. 다시 virtual to physical mapping이 일어난다. 그러면 , pentium의 경우에는 2 level paging. 앞의 field가 directory인데 . top level directory .first level. (First level) page table의 index로 사용. first level page table을 여기서 page directory라고 부르는데, page directory base register가 page directory의 시작 주소를 가리키고 directory filed가 index를 나타내기 때문에 찾아가면 directory entry가 있다. directory entry가 second level page table의 시작 주소를 가리킨다. 그러면, 두번째 필드 page가 second level page table의 index.. entry가 page frame number를 가지고 있다. 어떤 page frame인지 특정 할 수 있다. virtual address의 offset부분을 보고 해당 page frame의 offset만큼 읽으면 원하는 data의 code가 거기있다. 그래서 page table entry의 page frame number와 offset을 합성하면 physical address가 된다. 특정 page frame의 해당 부분을 찾아갈 수 있다. 자세한 것은 이제 설명

시작은 logical address의 selector 부분. 이부분이 descriptor table의 시작부분을 가리킨다. descriptor table의 index. selector를 보면 22page. 13bit이 Index로 사용되는 부분. descriptor table이 두가지 type이 있다. 하나는 GDT global descriptor table. 또하나는 LDT local descriptor table GDT 는 시스템에걸쳐 하나가있다. LDT 는 프로세스 마다 하나씩 존재한다. selector가 처음에 index값이 어느 descriptor table의 index인지 알려면 그 필드를 봐야한다. 0이면 GDT 1이면 LDT 해당 gdt나 ldt로 가서 인덱스를 사용해서 segment descriptor를 찾는다. segment descriptor가 그림 23page. segment descriptor는 굉장히 중요한 컨셉,
몇가지 type이 있다. 그러나 기본은 (핵심은) base field. 해당 segment 의 시작 주소를 가짐. 이경우에는 base field들이 쪼개져 있는데. 하나의 setment descriptor는 8byte으로 되어있다. 그것을 두개로 쪼개서 얹어놓은것. 처음에 4바이트 그다음이 위에. 가장 핵심은 segment의 시작 주소인 base filed base는 토탈 32bit. 4기가 공간을 나타낼 수 있다.segment descriptor에는 몇가지 종류가 있다. code segment descriptor와 data segment descriptor가 있고 stack segment descriptor, 기타 등등. code segment descriptor는 결과적으론느 code segment의 시작 주소를 갖는 정보를 담고있는 segment descriptor. data segment descriptor는 data segment의 시작 주소나 기타 등등을 갖고있는 segment descriptor 뭐 이런것. 내용을 좀더 들여다보면 Base이게 32bit. segment의 시작 주소. limit 필드는 segment의 크기.를 나타낸다. base가 32bit이면 4기가 공간인데 limit이 토탈 20bit. 1메가. limit field가 너무 작다. 이걸 어떻게 할 수 있냐. 왼쪽위 G. granularity filed. 이게 0이면 limit field 값이 단위가 byte. 이게 1이면 limit 필드 값이 page. 펜티엄에서 1page 값이 4KB. 2의12승 byte. G를 1로쓰면 limit 값의 단위가 4KB 이므로 리미트가 2의 20승 곱하기 2의 12승=2의 32승을 나타낼 수 있다. segment 값을 G값을 0으로 놓고 쓸 수도 있다. 22page의 selector. 24page에 다시한번 정리가 되어있다. logical address의 selector와 offset. selector가 descriptor table의 한 entry를 가리킨다. 거기에 segment descriptor가 있다. 여러가지 중에 핵심은 BAse address.. limit (segment의 크기)그리고 other filed.selector 가 descriptor table의 index로 쓰이니까 해당 entry찾아갈 수 있고 base address를 offset과 합치면 linear address가 나온다. 그게 아까 말한 중간 부분. paging이 enable =virtual address paging이 disable 이면 그거 자체가 physical address. selector 값이 결과적으로는 code segment data segment stack segment를 결국은 segment descriptor 통해서 가리킨다. 프로그램이 동작할 때 pentium의 segment register들에다가 selector 값을 loading을 한다. code segment register에는 code segment register를 index할 수 있는

가리킨다.deeer 값을 setting. data ,stack 도 마찬가지. linear address 25page. 아까 paging이 enable. virtual address. 그 얘기가 25pag'e에 한번 더. linear address의 앞의 10bit을 first level page table인 page directory라는 놈에 index로 써서 간다. 걔가 second level page table의 시작주소를 가지고, 거기서 다시 second field의 page field를 가지고 index로 써서 가면 page frame number가 있다. 그러면 해당 page frame 특정 가능. offset만큼 가면 원하는 data나 code access가능. word가 아닌 page frame 상당수의 operating system이 pentium을 사용할 때 code,data,stack segment를 별도로 사용하지 않고 하나의 segment로 통합시켜서 사용하는 경우 많다. 그렇게 되면 segment 하나. paging은 enable. 하나의 normal 한 paging 한개 주소공간을 쓴다. segment를 여러개 쓰지 않고 하나로 합쳐서 사용을 한다 그안에서 paging기법을 쓴다. virtual address space 공간을 그냥 하나만 쓰는 경우가 많다.

pentium은 4개의 protection level 제공0부터 3까지. 0이 제일 권한이 높은 level. 3이 권한이 작은 level. page 26. pentium에서 수행중일 때 프로그램은. 특정 protection level에서 수행.지금 수행중인 프로그램이 어떤 protection level에서 수행중인지 어떻게 알 수 있을까. 그것은 바로 cpu의 psw 레지스터의 bit 중에 두개의 필드가 있는데 program status word. 여기에 field가 여러개가 있는데 protection level 나타내는 두비트짜리. 걔를 보면 된다. 수행중인 프로그램이 어느 protection level에서 수행중인지 알 수 있다 .그리고 ,pentitum에서는 메모리를 segmentation과 paging을 다 support segmentation에 있어서 각 segment가 protection level을 가지게 된다. 이렇게 해서 어떤 메커니즘으로 protection level을 갖게 되나 하면은. 21page. pentium에서 segmentation with paging을 어떻게 support하느냐.맨 위에 logical address. selector값은 어디에 들어있냐 하면 segment register ㅇ 들어있다. code segment register,data segment register,stack segment register 이런것들이 있다. code segment register인 경우에는 code segment에 해당하는 selector 값을 가진다. selector 값이 descriptor table의 index로 사용. sesgment descriptor를 가리킨다.descriptor table 속에 있는 entry가 segment descriptor 굉장히 중요, segment descriptor는 base 필드 0-31 까지 있다. limit가 크기를 가리킨다. segment descriptor 속에 base field 거기에 segment의 시작 주소가 있는데 offset 더해서 linear address 생성. paging disable 시켰을 때는 linear address가 physical address paging이 enable linear address가 virtual address. 로 인식 first level, second level page table 사용. 거쳐서 결과적으로 page frame을 찾아간다. offset을 이용해서 page frame 내에서 offset만큼 이동하면 data 나 procedure 찾을 수 있다 segment descriptor 중요. code segment에 대한 정보 담고잇는데 code segment descriptor. data segment descriptor도 있고. stack segment descriptor도 있다 내부 구조는 거의 같다 base field가 segment의 시작주소 가지고있다 limit =segment의 크기. DPL. 이 바로 segment descriptor가 가리키고 있는 segment의 privilege level. segment의 privilege level. 0부터 3까지 어떤 프로그램이 수행중일 때 psw 레지스터 .. segment descriptor 속에 DPL field. 가 해당 segment의 protection level을 가리킨다. 수행되고 있는 프로그램의 psw에 있는 protection level값과 program이 access할려는 segment를 가리키는 segment descriptor의 DPL 필드에 있는 protection level이 같으면 아무 문제가 없다. 접근이 허용 즉, 자기의 protection level과 같은 level의 segment를 access하는 것이 허용. 그런데 ,자기의 protection level과 다른 level의 segment를 access할 경우 이때는 data segment access하는 경우와 code segment access하는 두가지 경우로 나뉜다. data segment를 access하는 경우는
지금 수행중인 프로그램이 자기보다 protection level이 큰 segment를 access하는 건 허용. 즉, protection level 1에서 수행 중 . protection level 3인 data segment를 access하는 것은 허용. 권한이 더 작은 것에 접근하는 것. 그런데자기의 level보다 작은. 즉 권한이 더 강한 data segment access하는 것은 허락되지 않는다. level 작을수록 권한이 강하다. code segment access하는 경우. 프로시저를 call할때 자기의 protection level보다 protection level값이 (권한이 약한것) 높은 segment에 프로시저를 call. 반대로 자기의 protection level보다 프로시저의 protection level이 작은 (권한이 더 센) segment 의 procedure call . 양쪽 다 된다. 양쪽 다 허용 단 조건. 특별히 제어된 방법으로 call.
CALL 다음에 오는게 프로시저의 주소를 직접 쓸수 없다. 반드시 selector값 써야함 selector는 CALL GATE를 가리키게 된다. 21page selector가 결국은 descriptor table속에 segment descriptor를 가리킴. 같은 메커니즘. 다음에 오는 selector 값이 CALL GATE (descriptor의 일종) 을 가리키게 된다. CALL GATE를 들어가보면 CALL gate도 23page의 구조. 여기에 나와있는 것은 code segment descriptor의 그림이지만 CALL GATE의 그림도 거의 비슷. 그안에 base 필드가 있다 base 필드에 procedure의 시작 주소가 있다. 그래서 이런 식으로 CALL GATE를 통해 프로시저 CALL. 이렇게만 CALL 할수 있다. 그래서 . 결국은 descriptor에 Type 필드가 각 descriptor가 code/data/stack segment descriptor/CALL GATE인지 이런것을 나타낸다.

27page. level 0 가 권한이 제일 강하다. kernel이 0 . level 1,2,3. level3:user programs. 여러분들이 프로그램 작성, 일반적으로 level 3에서 수행된다. os의 가장 critical 한 main memory 관리. 입출력 kernel.은 level 0에서 수행. 이제 file로 넘어간다.

Files: 장기 정보 저장. 일반적으로 메모리를 이요해 정보를 address space에 일시적 저장 가능. 그러나 그것은 process가 종료되면 그 process의 address space에 있던 데이터는 더이상 존재하지 않게 된다. access할 수 없게 된다. long -term information storage의 필요성이 있다. 요건은 다음 세가지
1. 많은 양의 데이터 저장할 수 있어야한다 2. 그것을사용하는 프로세스가 종료되더라도 계속 존재해야한다. 3. 여러개의 프로세스가 그 정보를 동시에 access할 수 있어야한다. 이러한 요구조건에 맞게 만들어진 것 File.

File Naming. 이름을 두개의 파트로 나눈다. prog . - 점 앞에, 점 뒤에. 점 뒷부분은 확장자. extension. 다양한 extension이 있다. gif와 jpg는 그림이나 사진 압축해서 저장. gif는 손실이 없게 jpg는 일반적으로 손실이 있다. gif는 손실이 없는대신 압축률이 좋지않을수도. jpg손실이 있는대신 압축률이높다. application이 스스로 이들을 관리하는경우가 있다. compiler같은 경우. c compiler가 . c compiler는 application window 는 확장자 존재 알고 관리. 그래서 확장자를 일반적으로 프로세스가 확장자를 등록. 그래서 어떤 프로그램이 그 확장자를 소유하는지 결정이되게된다. dos확장자는 word 라는 application에 assign, 그다음부터는 msword file sample.dos 파일 더블클릭하면 등록되있는 프로그램 즉 msword 그게 자동적으로 수행 그다음에 structure. 파일은 몇가지 방법으로 구조화시킬 수 있다. 대표적인 File STructure는 a,b,c 세가지
a 는 byte sequence b는 record sequence c는 tree. a byte sequence. byte의 연속. 그림 a 와 같다. windows가 a와 같은 경우. +unix. 그것에 반해서 b는 record sequence.이다 record sequence는 고정된 길이의 record의 sequence. 의 연속. 그림 b. c가 tree 구조이다. 이것은 record들이 tree 형태로 구성되있다. 그리고 중요한 것은 각 record에 Key field 가 있다. Ant 라는 key 값을 가진 레코드. 얘네들이 트리 구조를 가진다. 이러한 트리 구조를 가지고 있는 system의 경우에는 어느 때 유리하냐 특정 key를 가진 record를 찾을 때 유리하다.hen이라는 key값을 가진. 레코드 찾고 싶다. 이 tree 구조에서는 위에서부터 시작. H는 F와 P 사이 에 있다. 중앙 . 화살표로 가고 Goat 가 나옴 Goat 와 Lion 사이. Hen. 이 나온다 이런식으로 찾을 수가 있다. tree 구조file structure. 이런 걸 기존에 unix os와 windows os와는 다르다. tree구조는 실제로 상업적 data 처리. 많이 사용
File Types
대부분의 운영체제들은 여러가지 File Type을 support. 리눅스나 윈도우 같은 경우에는 regular file. directory 이러한 file type을 support한다. 그 이외에 추가적으로 character special file. block special file 제공 (유닉스) regular file은 여러분들이 알고있는 그냥 보통 file. directory 는 system file의 일종 . file system의 구조에 관한 정보를 담고있다. 그리고 character special file은 serial I/O devicef를 modeling. 예를들면 network. block special file은 disk를 modeling. regular file을 보면
regular file은 ASCII file과 binary file로 나뉜다 . 아스키 파일은 텍스트의 줄들로 이루어져 있다. 줄의 끝을 나타내는 문자가 있다. /r /n 이 될수도 있고 운영체제마다 다르다.이러한 아스키 파일은 보통 문자정보를 담고 그것을 보여주고 출력할 때 많이 사용. 그다음에 binary file. binary file에는 ascii 파일이 아닌 파일. page5에 그림이 있는데 이게 뭔가 하면 unix 운영체제 경우에 binary file에 두가지 종류가 있다 하나가 executable file. 또하나가 archive. 그게 그림의 a와 b. a executable file을 먼저 들여다보자.
executable file은 a와 같은 구조로 되어있다. Header text data … entry point는 text(code)의 시작 명령어 주소. magic number라고 하는게 바로 이 file이 executable file임을 나타내는 것. 이 file이 executable 이라는것을 identify, 그게 magic number. 이 magic number가 있으므로 os는 수행할 파일이 executable file인지 아닌지 알수잇다. executable file이 아닌걸 실수로 수행하는 일이 없다. 그다음에, unix에서 binary file의 또한가지가 archive. 그림 b. library procedure의 collection. 이다 library라는것이다. 그림 b를 들여다보면 header와 object module로 되어있다. header를 들여다보면 module name. (프로시저 이름) 생성된 date,owner 등등이 있다. 다음으로 넘어감.

File Access하는 방법. 두가지 설명
초기 os 는 sequential access를 support. 기록 (저장 매체)가 magnetic tape일 때 사용. 그래서 작동 메커니즘은 모든 byte가 record를 처음부터 순차적으로 읽는다. 특정 byte나 record로 점프 불가. rewind 가능 하드디스크가 등장하면서 생겨난 file access.
Random access.
byte나 record를 어떠한 순서로도 읽을 수 있다 순차적일 필요가 없다. database system에서는 필수적 random access를 support하는 system에서 일단 file read system calld을 하면 read한 만큼 file marker 혹은 position이 움직이게 된다. 그래서 next read할때 그 위치부터 read. 또는 특정 위치부터 read하고 싶다고하면 file marker를 옮겨야 한다. 그게 바로 seek라는 call을 통해서 가능. seek로 file marker옮겨놓고 file read를 call. 거기서부터 읽을 수 있다.

file attributes 모든 파일은 이름이 있고 data를 가지고있다.모든 os는 이것외에도 추가적인 information. 그 file이 생성된 날짜 시간. 파일의 크기 이러한 정보를 보통 file의 속성 이라고 한다.그림의 table에 전형적인 file의 attribute들이 있다.처음 4개 attribute는 protection 보안. 소유주. 만든사람 등등 .read only냐 아니냐. 등등 그다음 3개는 record 구조일때 (key position. key length 등등). creation time부터는 시간정보. 마지막 두개는 size 정보들. File에 관련된 operation . system call.
9page 소스코드. File을 copy하는 프로그램. 프로그램 이름을 copyfile이라고 침. copyfile abc xyz 로 실행 이 프로그램이 하는일은 abc파일 읽어서 그 내용을 새로운 파일을 만들어서 copy 새로운 파일이름은 xyz 10page에 open system call. source file을 open. read only mode로 open. creat 읽은 파일의 내용을 write할 새로운 파일 생성 output mode로. open 과 creat는 return value가 있다. file discripter?로부터 읽는다. 나중에 자세히 설명 while loop를 돌게된다. read system call. in_fd. 아까 read하려고 open한 파일 abc라는 file. while 루프 read. input source file 읽어서 버퍼에 넣는다 그것을 write system call을 해서 output file에다가 buffer내용을 쓴다. read system call을 할때 return 되는게 읽은 byte 수. read system call 할때 return 되는게 읽은 바이트 수. 가 rd_count d | 다. 그게 wirte할때 세번째 argument로 가게된다.실제 쓰여진 byte 수는 wt_count이다. 쭉 다 쓰고 문제 없으면 file close. 문제가 없으면 exit 0

Memory-Mapped File은 참고로만.간단히 말하면 프로세스에 메모리 상에 address space에 file mapping. 그래서 그address space 상의 data를 쓰고,읽게 되면 나중에 실질적으로는 file에 data가 쓰여지고 읽혀진다. 그런 컨셉이 있다
Directory.
초기의 디렉토리 버전은 single level root directory가 있고 밑에 사용자들의 file 이 있다. 네모는 directory. 동그라미는 file 동그라미 속 알파벳은 owner . single level이니까 Root directory 밑에 여러 사용자의 file이 주르륵 있다. 아무래도 불편하다. 너무 단순하기 때문에 문제. 그런데 이것은 작은 임베디드시스템. 내장형 시스템에서는 매력적인 directory 구조.
two level . 루트 디렉토리 밑에 각 사용자의 directory. 그다음에 file들이 있다. 각 사용자의 file.여전히 불편하다. 각 사용자도 다시 자신의 file들을 구분할 필요가 있다. 그래서 나온게 Hierarchical Directory Systems.
루트 밑에 디렉토리가 있고 그밑에 sub directory.. 와 file. tree구조. 익숙한 directory system 15page 디렉토리 구조 그림으로 설명 . 필기 보기
file system이 directory 구조로 되어있으면 특정 file을 specify하는 방법이 필요하다. 두가지 방법이 보통 있다.
absoulte path name /relative path name . dict라는 파일을 absoulte path name 으로 명시하려면 root부터 시작. 절대 경로는 root부터 시작. /user/lib/dict 이렇게 된다
relative path name을 알려면 먼저 working directory 또는 current directory 이 개념을 알아야한다. cd 로 특정 디렉토리로 이동하면 그 디렉토디가 working =current directory 이다. 그래서 사용자는 어떤 특정 디렉토리의 current directory가 된다. 어떤 file을 relative path로 지정한다는 것은 사용자의 working directory로부터 시작해서 지정. 예를들어 사용자의 working directory가 lib.(사용자가 lib 디렉토리에있다) relative path로 dict 파일 명시하려면 lib부터 시작하니까 그냥 dict가 된다.현재 lib에 있으니까 dot 과 dot dot을 알아야한다 점 하나 는 working =current directory . dot dot은 parent directory 를 의미. 따라서 현재 사용자가 lib에 있으면 dict를 명시할 때 working directory 부터 명시하는거니까 ./dict으로 할 수도 있따.

중요해요만 작성. (chapter 6 file 16page부터) 34page
i-node 가 2byte. 2byte는 16bit이다. 이 file system에 존재할 수 있는 최대 file의 개수 유추. 이 file shystem에서 허용하는 최대 file의 개수. i node 하나가 file, dir 하나에 match. 그러니까 그 file system. 그 partition에 존재하는 최대 file 개수는 inode number가 나타낼수있는 최대 수를 생각하면된다 결과적으로 16bit 이기 때문에 2진수 16개 2의 16승 개의 i-nodes를 나타낼 수 있다. i node 하나당 file 하나 match. 최대 허용 file 개수는 2의 16승 개가 될 것이다.

35page 계산 .커버할 수 있는 최대 file 크기. 굉장히 중요해요 계산하는 것 그림과 함께 익히기 (inode 구조에서)
36page. 굉장히 중요한 그림 /usr/ast/mbox 의 과정 꼭 명심하기
37page i-node 하나당 하나의 file or directory 가 대응

각 RAID 의 컨셉. 성능 .reliablitiy. redundancy. 계산은참고로만
cylinder skew 중요한 부분 25page !!