운영체제

▶ OS : Application 관점에서는 실행 환경을 제공해주고, System 관점에서는 컴퓨터 시스템의 리소스를 관리, Implementation 관점에서는 높은 동시성과 event-driven SW

1세대 : 진공관(50년대 초) 하나의 accumulator 2세대 : 트랜지스터(50년대 말) accumulator + register

3세대 : IC. Multiprogramming. Time sharing. Family design 4세대 : CPU (60년대 말). MSI, Memory

5세대 : VLSI

▶ Monolithic : 커다란 하나의 것으로 되어 있음. 성능에서 장점이 있으나 maintain과 upgrade에 문제

▶ Layered : modular로 maintain이 원할

▶ Microkernel : 각 module은 microkernel을 통해 communication. Thread scheduling, Message passing, Virtual memory, Device driver code 만을 포함. 유연하고 신뢰성있고 유지보수가 편하고 distributed system을 지원. 단점은 performance

Linux는 Monolithic kernel + layered kernel module, Mac OS X는 microkernel, Windows XP는 Microkernel-like

▶ Interrupt : HW가 만듬. Asynchronous

▶ Exception : SW가 만듬. Synchronous, interrupt handling과 동일

- Traps : 고의적임. System call, breakpoint, Floating point exception 등. 다음 명령어로 돌아감.
- Faults : 고의적 아니나 회복 가능. Page fault, protection fault(read-only인데 쓰거나 privileged page에 접근)
- Aborts : 고의적 아니고 회복 불가능. Parity error, machine check, 프로그램 종료, 재부팅, 수리 등

▶ System Call : OS가 제공하는 programming interface service.

▶ Timer : 이상한 프로세스가 CPU를 계속 쓰게 된다면 OS가 CPU를 쓰지 못하는 일이 발생하기에 timer interrupt를 두어 control을 OS로 돌리는 역할

▶ OS는 kernel mode와 user mode로 나누어 보호.

▶ Process : 실행중인 프로그램으로서 자신의 작업을 처리하기 위하여 CPU time, memory, file, I/O 장치와 같은 자원들을 필요로 한다. 대부분의 시스템에서 프로세스는 작업의 단위이며 이런 시스템은 프로세스들의 집합체로 구성된다. 운영체제 프로세스들은 시스템 코드를 실행하고, 사용자 프로세스들은 사용자 코드를 실행한다. 두 프로세스들이 동일한 프로그램에 관련되어 있어도, 이들은 두 개의 독립된 실행 순서를 가진다.

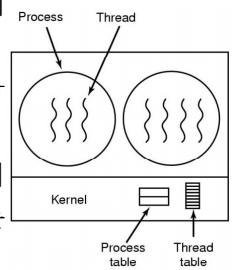
▶ Context switch : CPU가 실행하는 process를 바꿈. State queue. PCB

▶ fork : 새로운 프로세스를 만드는 UNIX system call. exec : 현재 프로세스를 중지하고 프로그램을 해당 프로세스에 올림. CreateProcess : Windows에서 프로그램 로딩시 사용. 부모 프로세스를 복사하지 않음.

▶ Thread : Process는 address space, 파일 디스크립터나 계정 정보 등의 OS 리소스, HW execution state 등의 많은 정보를 들고 있다. 따라서 새로운 프로세스를 만드는 것은 그만큼 비용이 높다. 또한 프로세스끼리의 통신은 OS를 거쳐야 하기에 이 역시 비용이 높다. 따라서 비슷한 작업을 할 경우 concurrency를 높이기 위해 프로세스를 새로 만드는 것 보다는 그보다 가벼운 thread를 사용하자. 같은 process 안의 thread끼리 data를 공유하는 것은 쉽다. 왜냐하면 address space를 공유하기 때문이다. 따라서 thread는 schedling의 기본 단위가 된다.

▶ Multithreading의 장점 : 프로세스를 만드는 것보다 싸다. Throughput 향상. 반응속도 향상(Server/Client), 리소스 공유, multiprocessor 환경에 맞출 수 있음.

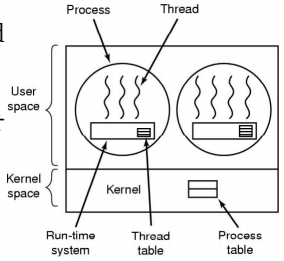
▶ Thread 종료에는 해당 thread를 즉시 종료하는 Asynchronous cancellation과 종료 지점에 도착하여 종료하는 Deferred cancellation이 있다.



▶ Signal을 관리하는 측면에서 모든 thread가 이를 보고 있으면 비효율적이기에 어떤 thread가 신호를 처리할지 지정하거나 아니면 이미 결정되어 있거나 해야 한다.

▶ Kernel level thread : 모든 thread 관련 operation을 kernel에서 한다. OS가 모든 thread의 scheduling을 관리한다. 하지만 이는 여전히 비싼 작업이고(syscall) kernel state가 이 모든 thread를 관리해야 한다.

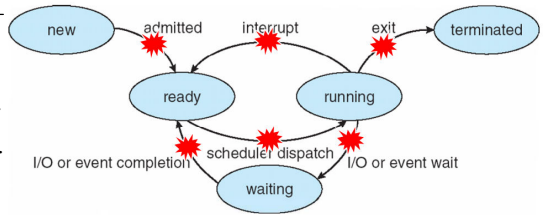
▶ User level thread : 좀 더 작고 빠른 thread를 위해 library 단계에서 user-level thread 지원. 하지만 OS에서는 user level thread가 보이지 않아 OS가 좋은 결정을 할 수 없게 한다.



▶ Non-preemptive scheduling : 모두가 협력해야 함. yield와 같은 것을 써서 자신이 쓰는 CPU를 놓아줘야 함.

▶ preemptive scheduling : Asynchronously CPU control을 얻음. Timer 등을 사용.

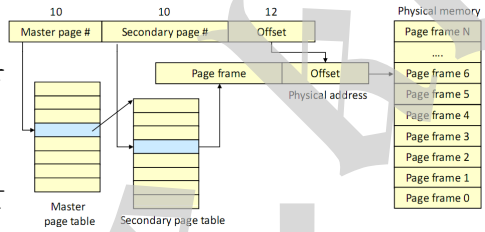
▶ Threading model로 User thread : Kernel thread로 N:1, 1:1(많이 쓰임), M:N이 있음.
▶ Synchronization : 리소스를 공유하다보면 쓰고 읽는데 correctness가 떨어지는 일이 발생. 이를 Synchronization을 이용해서 control.
▶ Race condition : 여러 프로세스들이 공유된 데이터에 동시에 접근하고 관리하는 상황. 결과는 비결정적이며 시간에 따라 달라짐.
▶ Critical section(C.S) : 공유되는 메모리나 파일, 리소스에 접근하는 프로그램 코드의 부분. mutual exclusion을 사용해야 함. 시간 상 오직 한 thread만이 critical section을 사용할 수 있음.
요구사항 : - Mutual exclusion : 최대 하나의 thread만이 작동.
- Progress : C.S 밖에 thread T는 C.S에 들어가려는 thread S를 막을 수 없다.
- Bounded waiting(no starvation) : C.S를 기다리는 T는 결국 C.S에 들어간다.
- Performance : 속도도 중요하죠.
종류 : Locks, Semaphores, Monitors, Messages
▶ Lock : 획득하고 풀어주는 일 수행. spin 혹은 block. lock 자체도 C.S. => SW만으로 처리(Dekker, Peterson 알고리즘), HW Atomic operation(test-and-set, swap), interrupt 끄고 켜기(kernel에만 유효, 멀티프로세서에서 안됨)
▶ Semaphore : Wait(down), Signal(Up). spining이 아님. Binary이거나 Counting이 가능. 여전히 global 변수 사용.
▶ Bounded buffer problem : full이냐 empty이냐를 semaphore를 이용하여 해결.
▶ Reader-Writers problem : reader는 여러 명 읽어도 되나 writer는 한 번에 한 명만 가능. reading이 끝나고 나서 writing이 가능. reading 시 첫 reader이면 rw를 wait, 마지막 reader이면 rw를 signal. wirter는 rw를 wait한 후 다 하고 나서 rw를 signal. reader는 rw를 wait하고 signal을 할 때 각각 또 다른 mutex로 C.S를 만듬.
▶ Dining Philosopher problem : 생각하고 배고프고 젓가락 들고 먹고 내리는 작업 반복. pickup시 mutex로 C.S를 만들고 자기 자신을 test한다. test는 자신이 배고프고 왼쪽과 오른쪽이 먹는 상황이 아니면 자신의 signal을 날린다. 그러고 나서 자신을 wait하는데 test에서 통과하였다면 wait는 그냥 지나가지만 아니라면 여기서 기다리게 된다. 먹고 난 후 putdown을 하는데 mutex로 C.S를 만든 후 양옆을 test한다. 그럼 양옆 철학자가 조건이 만족되면 wait하던 것은 먹을 수 있게 된다.
▶ Monitor : programming language가 만듬. Mutual exclusion과 condition variable 가능.
▶ Condition variable : - wait : monitor lock을 release, - signal : 최대 하나의 process를 wake up. semaphore와 다른 점은 history가 없다. - broadcast : 모든 기다리는 process를 깨운다.
▶ Hoare monitor : signal이 즉시 기다리는 thread로 바뀌게 전달된다.
▶ Mesa monitor : condition을 계속 check하여 thread 실행
▶ Starvation : 한 프로세스가 자신이 필요한 리소스를 다른 프로세스가 계속 잡고 있는 것.
=> 높은 priority를 가진 것이 계속 실행되어 낮은 priority를 가진 것이 실행되지 않음. 해결책 : Aging(waiting time이 높으면 priority도 높임)
▶ CPU scheduling : 다음 실행될 프로세스를 결정하는 것. 자주 일어나며 빠르게 처리되어야 함.



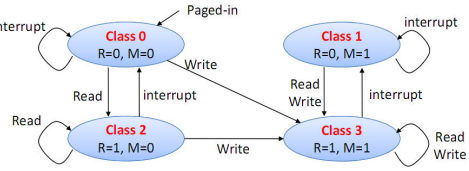
▶ FCFS/FIFO : non-preemptive, no starvation(동등하게 처리되기에).
▶ Shortest Job First(SJF) : non-preemptive. 가장 짧은 것부터 처리. starvation 발생할 수 있음.
▶ Shortest Remaining Time First(SRTF) : preemptive version of SJF. 남아있는 것중 가장 짧은 것을 선택.
▶ Round Robin(RR) : Circular FIFO Queue. time slice별로 각각 프로세스에 할당. preemptive.
▶ Priority scheduling : priority를 주어 높은 priority를 가진 프로세스를 실행하게 한다.
▶ Priority inversion problem : 낮은 priority를 가진 것이 리소스를 잡고 있어 높은 것이 일을 못함. 이를 해결하기 위해 Priority donation이 필요. -> pintos project 1

▶ Fixed partition : 파티션이 고정된 크기. 쉽게 만들 수는 있으나 Internal fragmentation과 partition size 문제 발생.
▶ Variable partition : base register와 limit register 사용하여 가변적으로 할당. internal fragmentation은 없으나 external fragmentation 있음.

▶ Internal fragmentation : 할당된 것보다 적은 양을 사용할 때 그 남는 양이 단편화
▶ External fragmentation : 대기 중인 작업에 비해 비어있는 공간이 너무 작아 쓰지 못하여 단편화
▶ Virtual Memory : 메모리 관리에 virtual address를 사용. address translation이 필요.
- 장점 : physical memory와 logical memory 구분. user는 메모리 전부를 이용한다고 가정. 파일과 메모리 구분. 각 프로세스 간 protection 제공.
- 단점 : performance.
▶ Segmentation : code, data, stack 등의 영역으로 나눠져 있는 것.
▶ Paging : 같은 크기의 page(frame)으로 나눠서 관리. valid bit, invalid bit 등으로 protection 보장. Page Table을 이용하여 physical address와 virtual address를 조정
- 장점 : physical memory 할당 쉬움. external fragmentation 없음. page-out이 쉬움. protect, 공유도 쉬움
- 단점 : Internal fragmentation 존재. memory reference overhead, page table 저장에 용량 차지
▶ Demand Paging : load lazy. 요청이 들어오면 HDD에서 RAM으로 load. OS가 이것을 하며 이 때 page replacement algorithm 적용할 수 있음. 처음 프로세스가 시작하면 필요한 것만 올린다. 그 후 사용하다가 필요한 곳이 발견되면 이를 하드에서 가져온다. pintos - project 2



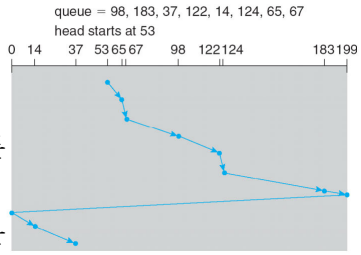
▶ Page Table : level을 두 개로 하여 Page table 크기를 줄인다. 혹은 Hashed Page Table, Inverted Page Table
▶ TLB : Fully associative cache. locality가 중요. MMU
▶ Page replacement : 가득차면 page를 HDD에 넣어야 한다. 어떤 것을 할 것이냐? 최대한 page fault가 덜 나게 하는 것이 목표다.
▶ Belady's Algorithm : 오랫동안 사용하지 않을 것을 victim으로 하자. 문제 : 미래 예측은 어떻게 하지?
▶ FIFO : 들어온 순서대로 내보내자. 예전에 사용한 것이니 아마 다시 사용하지 않을 가능성이 높기는 하지만, 반드시 그런 것만은 아니다.
▶ Belady's anomaly : Frame이 증가하였음에도 Page Fault가 증가하기도 한다.
▶ Least Recently Used(LRU) : locality. 최근에 사용하지 않은 것을 내보내자. 하지만 HW단의 implementation이 힘들어 순수한 것은 어려움.
▶ Approximating LRU : PTE에 reference(R) bit를 둔다. 접근하면 1, 적당히 0으로 떨어뜨리면 0인 것을 선택.
▶ Second Chance(LRU clock) : clock을 돌면서 R=0이면 내보내고 1이면 0으로 셋팅.
▶ Not Recently Used(NRU) : R bit와 함께 M(modify) bit도 두어 수정된 것까지 확인한다. 0 > 1 > 2 > 3



▶ Least Frequently Used(LFU) : 사용빈도가 낮은 것을 선택. 최근 것을 못 본다는 단점. 이 해결법으로 Aging 기법 있음.
▶ Victim을 선정할 때 process 안인 local하게 볼 것이냐 전체(global)하게 볼 것이냐 다르다.
▶ Thrashing : OS가 disk와 메인 메모리에서의 swap을 하는데 시간을 많이 보냄.
▶ Working Set Model : Working Set(window와 비슷한 개념)을 만들어 그 안에 있는 page만을 최근에 referenced
▶ Page Fault Frequency : PFF가 높으면 메모리를 더 꼽고 낮으면 메모리 제거해라(응?)
▶ Shared Memory : 프로세스끼리 메모리를 공유하는 것. 각 프로세스의 virtual address를 동일하게 하느냐 다르게 하느냐에 따라 다름. 다르게 하면 유연하나 포인터 변수를 공유할 경우 곤란. 같게 하면 반대 현상
▶ Copy on Write : 프로세스를 새로 만들 때 부모 프로세스로부터 모든 페이지를 복사하지 않고 처음에는 read only로 부모 프로세스의 프레임을 가져오다가 write가 발생하면 프레임을 할당하여 복사한다.
▶ Memory Mapped File : 파일을 메모리 페이지에 매핑하는 것. 파일 접근이 메모리와 같고 복사를 덜 하여도 되고 여러 프로세스가 이를 공유할 수 있어 좋지만, pipe, socket 등에는 사용하지 못한다.

▶ I/O buffering : 하지 않거나, user space에 하거나, kernel 후 user에 하거나 kernel에서 두 개를 한 후 그 중 하나를 user가 가져오는 방식
▶ fopen : C library. library에 버퍼가 있어 system call 감소. 왜냐하면 file read시 fread를 할 때 들어남. 따라서 큰 데이터 파일이라면 open, read(syscall), 자주 접근하는 데이터라면 fread가 좋다.
▶ Spooling : 프린터가 바쁜 상태더라도 task를 안에 넣을 수 있다.

▶ Disk 탐색 알고리즘



- Fisrt Come First Served (FCFS) : 들어오는 job 순서대로 읽는다.
- Shortest Seek Time First (SSTF) : seek time(arm movement)를 줄인다.
- SCAN : Elevator algorithm. 한쪽 방향으로 읽은 후 끝에 도달하면 반대 방향으로 튼다. uniform하게 기다리지 않는다.
- C-SCAN : Circular SCAN. 한쪽 방향으로만 읽는다. uniform하게 기다리는 시간을 가진다.
- LOOK / C-LOOK : SCAN/C-SCAN과 비슷하나 arm을 끝이 아닌 job들 중 가장자리에 있는 곳까지 간다.

▶ File : secondary stroage에 기록된 것에 관련된 정보를 지닌 collection의 이름. File contents와 attributes (metadaa), name을 가진다. metadata에는 크기, owner, 시각 등이 기록.

▶ File system : mapping problem을 가진다. Performance와 reliability가 목표.

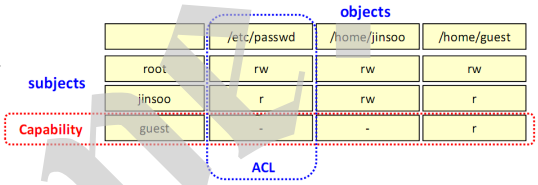
▶ Directory : list of (filename, file attribute)

▶ Pathname Translation : root directory를 찾은 후 하위 directory를 찾아간다. 마지막에 파일을 찾아간다. 이때 permission을 확인한다.

▶ Mounting : Windows : drive letter, UNIX : mount point(directory)

▶ File Locking : - flock : 모든 파일을 locking. - fcntl : 파일에 부분을 locking.

▶ Protection



- Access control lists(ACLs) : 각 object마다 각 사용자의 권한 리스트를 가지고 있음.
- Capabilities : 각 사용자마다 각 object의 권한 리스트를 가지고 있음. -> key와 같아 다른 사용자에게 donation이 편함.

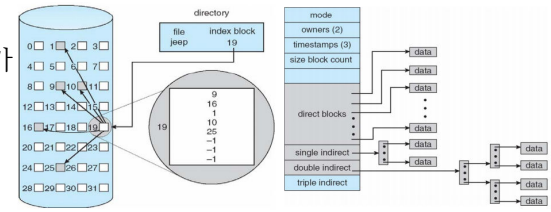
▶ Virtual File System : superblock(file system 정보), inode(특정 파일의 정보), file, dentry(directory entry)

▶ Directory implementation : Table, Liner list, Hash table

▶ metadata는 directory entry에 있거나 i-node처럼 떨어져 있거나 둘을 섞은 것이 있다.

▶ Disk에 Allocation하는 방법

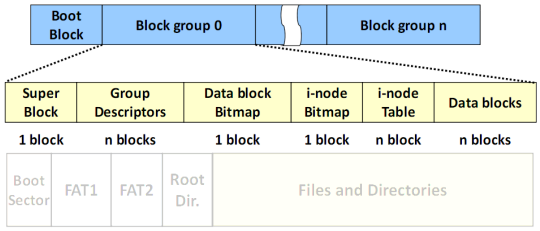
- Contiguous allocation : 연속적으로 할당. 찾는데 쉽고 directory 역시 쉽다. 하지만 First fit, best fit 등을 해야하고 external fragmentation. CD-ROM에 사용
- Modified Contiguous allocation : Contiguous allocation에서 자리가 모자라면 extent라는 다른 것을 붙인다. 여전히 directory에는 쉽지만, internal external fragmentation 문제. VxFS
- Linked allocation : 각 프레임별 링크. external fragmentation은 없고, directory도 간단하고 파일을 계속 쓸 수 있다. 하지만 파일에 순차적으로 접근하고 공간 낭비(pointer) 그리고 pointer를 잃어버리면 fragile 발생. FAT에서 사용. cache가 되어있어 빠름.
- Indexed allocation : i-node와 같은 index를 가짐. direct access 가능. 하지만 indexing 하는데 공간 낭비



▶ Free space 관리

- Bitmap or bit-vector : 각 블록이 1 bit를 가짐.
- Linked list : free block끼리 연결. FAT
- Grouping

▶ Ext2 File System : Block group으로 나뉜다.



▶ FAT File System : File Allocation Table이 있음.