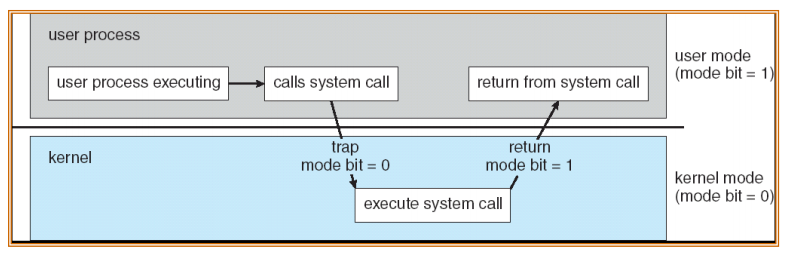
1. OS
2. OS: 시스템 하드웨어를 관리할 뿐 아니라 응용 소프트웨어를 실행하기 위하여 하드웨어 추상화 플랫폼과 공통 시스템 서비스를 제공하는 시스템 소프트웨어.

하지만 뚜렷하게 내려진 정의는 없음.

1. 역할
2. Hardware management
3. Access to I/O devices
4. Access to files
5. Accounting: 어떤 프로세스, 어떤 user가 어떤 종류의 resource를 얼마나 차지하는지 추적하여 제공(Windows에서의 작업관리자)
6. Error detection: 에러의 타입에 따라 취해야할 행동을 수행
7. Program execution
8. Scheduling
9. Error reporting
10. 동작

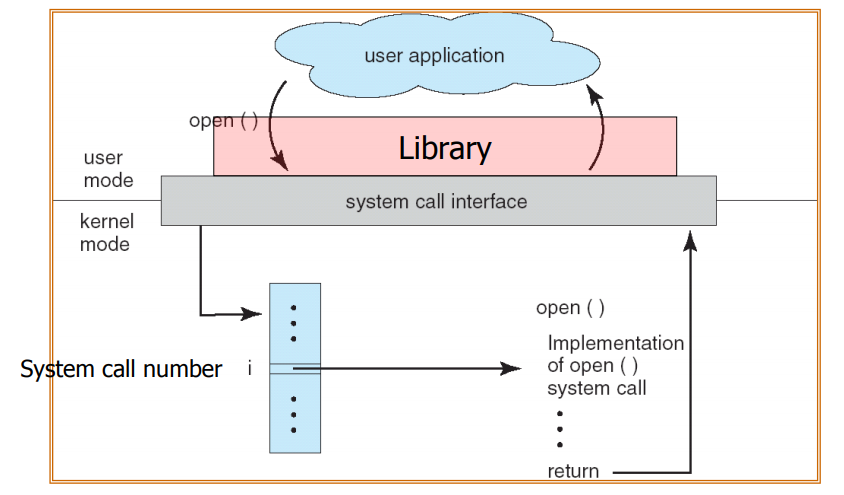
두 개의 독립된 동작 모드를 가지고 있음. User mode, Kernel mode

최초 부팅할 때 kernel mode로 시작하여 OS가 올라오고 나면 user mode가 시작된다. trap이나 interrupt가 발생할 때 마다 kernel mode로 전환한다.



1. Kernel mode 진입 방법
2. Hardware interrupt
3. Trap (software interrupt): exception, system call
4. System call

kernel영역의 기능을 API로 user mode가 사용 가능하게 기능을 제공.



1. System call 호출 시 파라미터 전달 방법

parameter가 몇 개 안될 때는 register로 전달, 많으면 parameter가 저장된 memory block의 주소 전달

1. OS Structure 종류
2. Simple structure

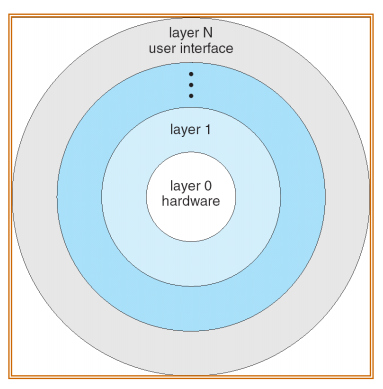
module이나 layer로 나누어 지지 않는 structure. 이중모드도 없고, HW protection도 없음.

Ex) MS-DOS

1. Layered approach structure

Layer(level)로 나눌 수 있음. 예를 들어 user interface는 N layer, hardware는 0 layer. 각 layer는 아래 layer에 대한 operation들을 제공받음

EX) 옛날의 UNIX



1. Microkernel structure

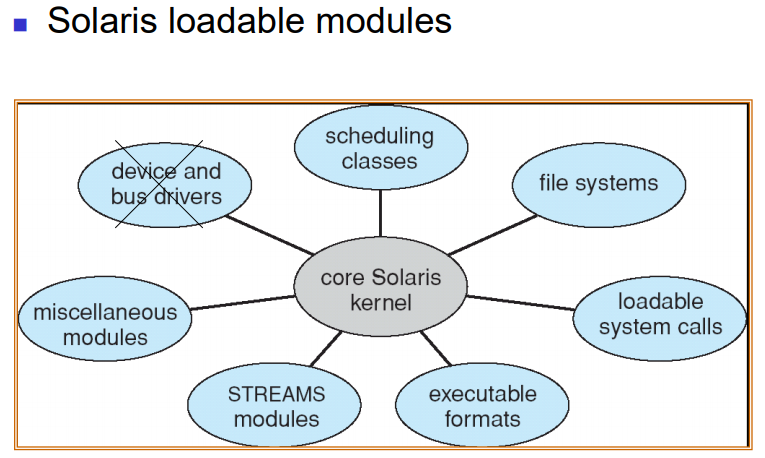
kernel 모드에서 실행할 수 있는 것들을 최대한 많이 user mode로 옮긴 것.

Ex) Mach OS, QnX, real-time OS 등. 미사일 컴퓨터 같은 컴퓨터에서 사용

1. Module structure

각 core component가 별도로 존재하게 하여 kernel에서 각각 load할 수 있게 함.

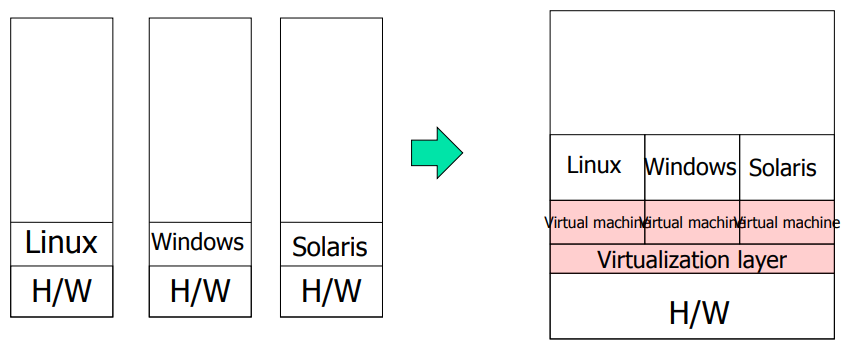
지금의 LINUX = layered approach + Modules



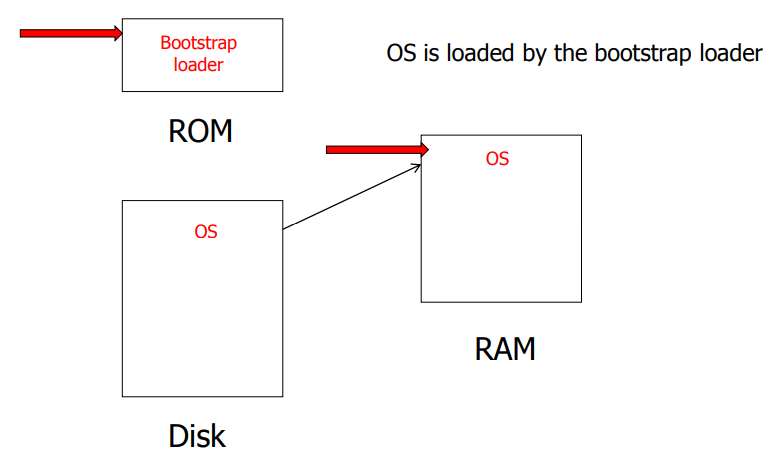
1. Virtual machine

각 프로세스들이 사실은 같은 컴퓨터에서 돌아가지만 다른 환경의 컴퓨터(OS)에 있는 것처럼 느끼게(동작하게) 함. 구현이 어려움. 병렬적으로 test하기에 좋은 concept.

1. Virtualization layer: 다중 virtual machine들이 하나의 physical machine에서 동작 가능하도록 multiprogramming함

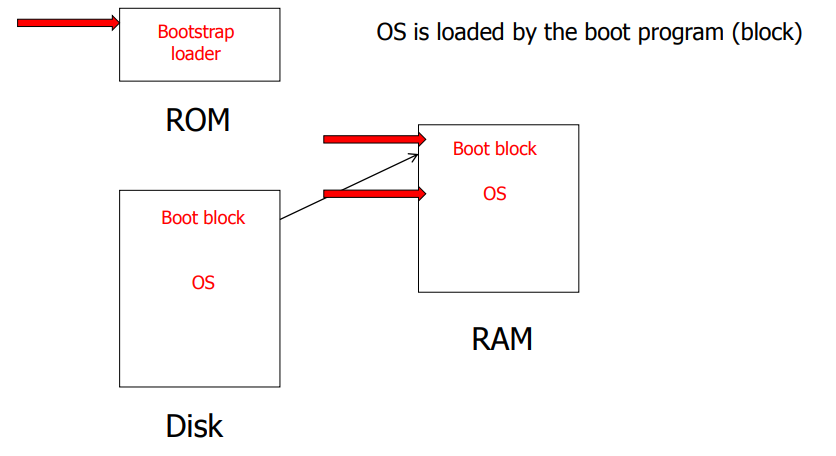


1. Boot
2. Booting: kernel을 loading하여 컴퓨터를 시작하는 procedure
3. Bootstrap loader: kernel(ROM)에 위치한 작은 코드 조각. 메모리로 불려와 시작됨
4. Boot 방법
5. Single-step approach

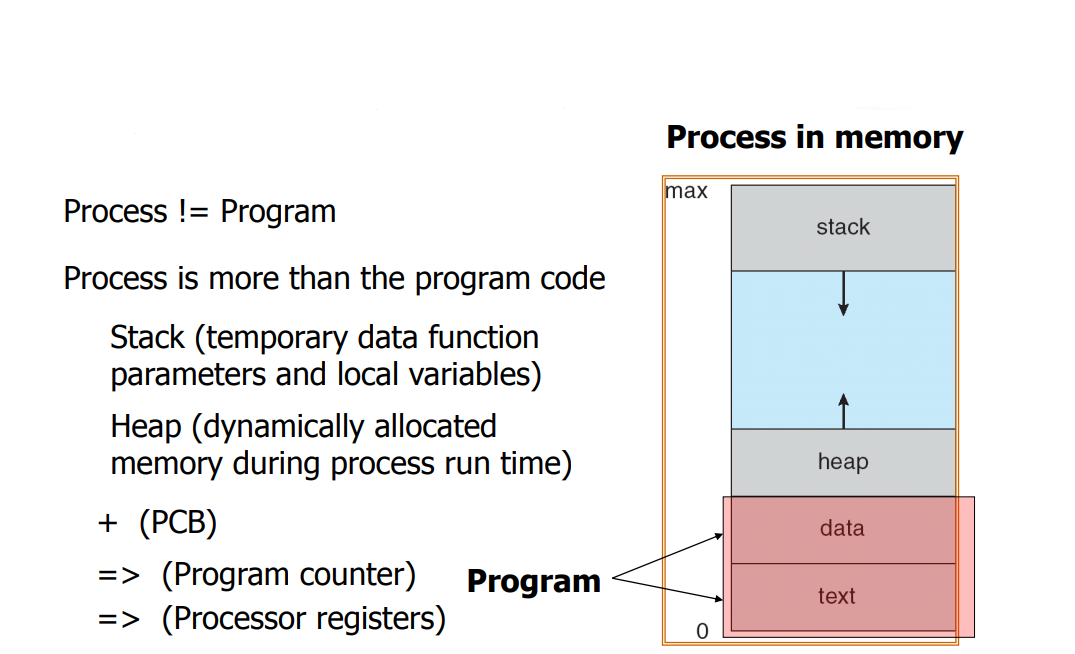


1. Two-step approach

좀 더 boot program이 복잡할 때 bootstrap loader가 boot block을 disk에서 가져와 boot program을 실행할 수 있게 함



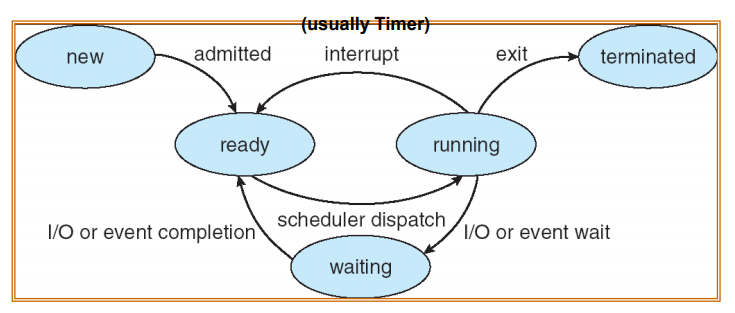
1. Process
2. 정의: 실행중인 program. Program일 때는 disk에 코드로 저장되어 있다가 실행 시작하면 memory에 올림.



Process들은 부모 process가 자식 process를 만들기 때문에 tree구조를 가짐.

자식 프로세스는 태어날 때 부모 프로세스를 복사하며 태어나는데 부모의 resource 중 일부(code, file 등)을 공유함. 부모 프로세스는 보통 자식 프로세스가 종료될 때 수거를 하기 위해 기다림.

1. State
2. New: 새로 created된 상태
3. Running: 동작 중인 상태
4. Waiting: event발생을 기다리는 상태
5. Ready: 프로세서에 지정되기를 기다리는 상태(동작할 준비 완료)
6. Terminated: 종료된 상태
7. Process의 종료
8. 일반적인 종료: exit으로 무사히 종료되어 OS가 delete한 경우
9. Orphan process: 원래 부모 프로세스를 자식 프로세스의 종료를 기다리고 수거해가는데 자식보다 부모가 먼저 종료된 자식 프로세스
10. Zombie process: Orphan process처럼 부모가 죽어서 부모의 상위 process가 자신을 수거해 줄 때까지 종료하지 못하고 기다려야하는 프로세스



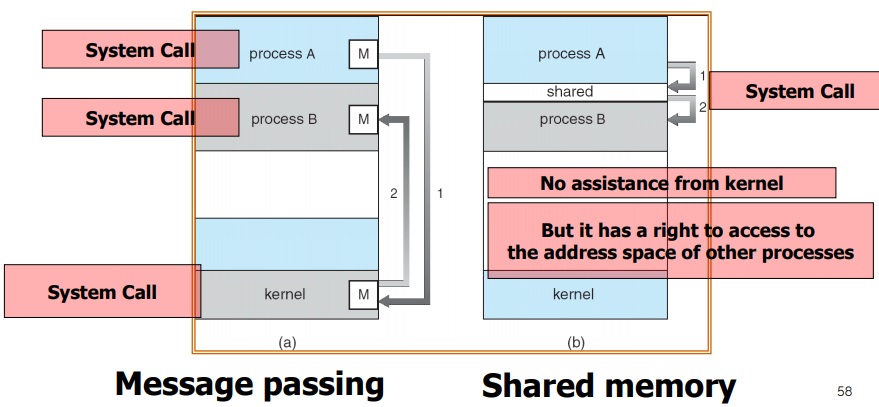
1. PCB (Process Control Block): process에 대한 정보들이 담겨 있는 block

Process state, Program counter: 다음 수행할 부분, CPU registers, CPU scheduling information: process 우선순위 정보, Memory-management information, Accounting information, I/O status information: 열린 파일들

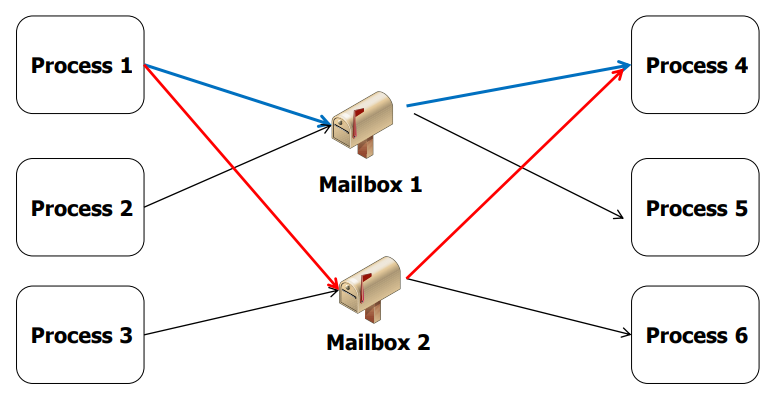
1. IPC

Independent process는 다른 프로세스의 실행에 영향을 주거나 받을 수 없지만 cooperating process는 다른 프로세스의 실행에 영향을 주거나 받을 수 있음.

Process cooperation으로 information sharing, computation 속도 상승 등의 효율성을 늘리기 위해 아래의 방법으로 process들 간의 communication을 함.



1. Message passing: Send와 receive로 메시지를 주고받음. 작은 양의 message에 대하여 유용하고 프로그래밍이 쉬움
2. 관련 이슈 – Establishing link
3. Direct communication: Send(대상, message), receive(대상, message) 이런 식으로 direct하게 메시지 전달. 메시지를 보내는 프로세스와 받는 프로세스 사이에 하나의 링크가 자동적으로 설립되어 양방향적으로 운영됨
4. Indirect communication: Send(mailbox이름, message), receive(mailbox이름, message) 이런 식으로 Mailbox에 메시지를 보내고 받는 식으로 운영. 프로세스에서 mailbox사이에 링크가 설립되어 양방향 혹은 단방향으로 운영됨.



여러 프로세스가 같은 메일박스(링크)를 사용할 때의 문제점 발생

* Mailbox에 연결된 프로세스를 최대 2개까지 제한/broadcasting으로 대상을 알림/한 process만 메시지를 받을 수 있게 함./send하는 process가 받는 process를 선택하게 함. 이런 방법들로 해결

1. Shared memory: 한 프로세스의 address space에서 공유할 memory구역을 만들어서 공유. 공간을 만들 때만 system call을 호출하기 때문에 빠름, 하지만 메모리 영역의 protection같은 이슈들이 있어서 신경 써야함
2. Context switch: CPU를 사용하는 process를 변경할 때 save로 동작하던 old process의 state를 저장하고 load로 new process의 state를 받음. Context-switch time은 큰 overhead임. 따라서 효율적으로 해야함
3. Dispatch: 고르는 작업을 scheduling이라고 한다면 고른 작업을 CPU에 올리는 작업을 dispatch라고 함. Dispatcher는 dispatch를 하는 module을 말함.
4. Burst
5. I/O-bound process: CPU 사용보다 I/O접근에 더 많은 시간이 소모됨.
6. CPU-bound process: I/O접근보다 CPU사용에 더 많은 시간이 소모됨.

CPU가 수행되는 구간을 CPU burst, I/O때문에 block되는 시간을 I/O burst라고 함.

CPU burst크기가 크면 CPU intensive한 프로세스, 반대로 작으면 I/O intensive한 것.

1. Scheduling
2. queue 종류
3. Job queue: ready queue에 보낼 프로세스를 결정.
4. Ready queue: ready, waiting 상태에서 실행되기를 기다리는 프로세스들
5. Device queue: I/O device를 기다리는 프로세스들
6. scheduler 종류
7. Long-term scheduler: job scheduler. Second, minute단위로 갱신
8. Short-term scheduler: CPU scheduler. millisecond단위로 갱신
9. Medium-term scheduler: multiple programming을 줄이기 위한 process swapping할 때 사용(자주 사용하지 않음)
10. CPU scheduler의 decision을 내리는 때
11. Non-preemptive or cooperative
12. Process가 block될 때(running -> waiting)
13. Process가 종료될 때(running -> terminate)
14. Preemptive: 수행 중인 프로세스가 block되지 않아도 바뀔 수 있음.
15. Event 종료(block된 process가 ready될 때)
16. Timer interrupt
17. Shared data에 접근할 때
18. CPU scheduling 기준
19. CPU utilization: 전제 system에 대한 CPU 이용률
20. Throughput: 전체 system에 대한 일정 시간동안 해결한 task의 수
21. Turnaround time: process 하나에 대한 task 요청 후부터 끝날 때까지의 시간
22. Waiting time: process 하나에 대한 task 요청 후부터 ready queue에 있는 시간.
23. Response time: process 하나에 대한 task 요청 후부터 첫 실행까지 걸리는 시간.
24. CPU scheduling algorithm
25. FCFS (First-Come, First-Served): 먼저 온 process 먼저 처리
26. SJF: 실행시간이 가장 짧은 process 먼저 처리.

* 실행시간을 미리 알 수 없음. 비현실적.

Exponential averaging(공식)로 실행시간을 예측.

Preemptive/Non-preemptive 버전이 있음.

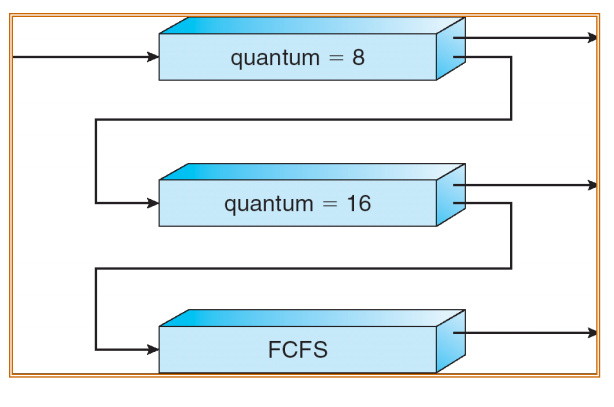
1. Priority scheduling: 우선순위가 높은 process 먼저 처리
2. 이슈 - Static/Dynamic priority: 우선순위를 고정/변동
3. 이슈 - Starvation(명시되지 않은 blocking): 우선순위가 낮은 process가 수행되지 못해서 생기는 현상 – aging(오래된 process의 우선순위를 높임)같은 방법으로 보완 가능.
4. Round-robin scheduling: time quantum(time slice)만큼 돌아가면서 수행. Process가 끝나지 않아도 time quantum만큼 지나면 block되기 때문에 preemptive scheduling임.

Time quantum의 크기에 따라 context-switch하는 횟수와 turnaround time이 결정됨.

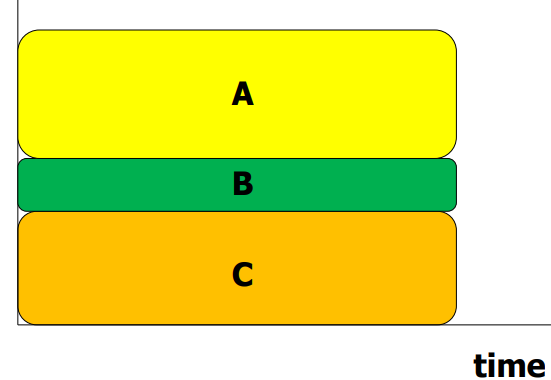
1. Multi-level queue: 앞의 scheduling algorithm들을 queue형식으로 만들어 여러 개의 queue를 같이 사용하는 방법.

Ex) foreground는 RR, background는 FCFS

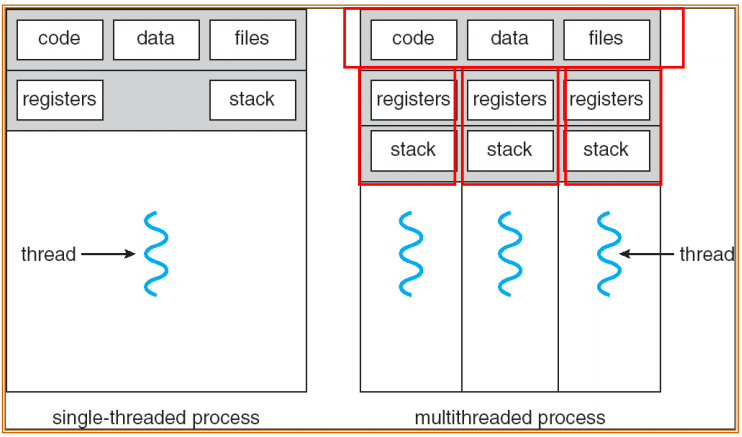
1. Multi-level Feedback queue: multi-level queue처럼 여러 종류, 여러 개의 queue를 사용하면서 어떤 queue의 결과가 다음 queue에 영향을 미치는 방법.



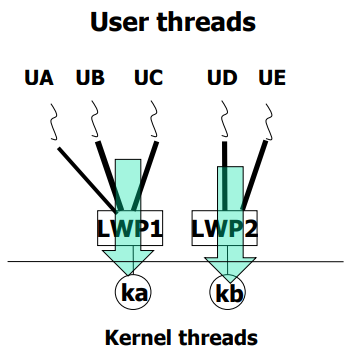
1. CFS (Completely Fair Sharing): 모든 시간에 대해서 각각의 process가 process의 가중치 비율만큼의 processor(CPU)의 시간을 사용할 수 있게 하려는 방법.



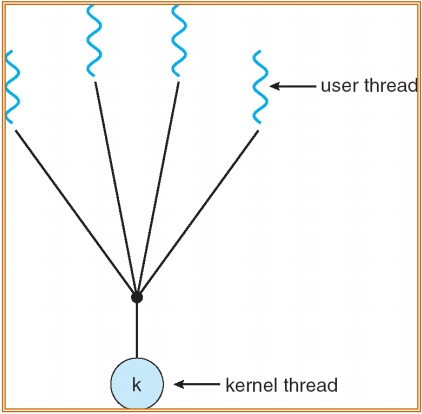
1. Scheduling algorithm evaluation
2. Deterministic modeling: waiting time등의 특정 기준을 대한 알고리즘마다의 성능 평가.
3. Queueing models: CPU와 I/O burst 분포에 대한 결과로 평가.
4. Simulation: 모의 실험을 통해 평가.
5. Multi-processor scheduling: 다중 CPU(core)가 사용 가능할 때 CPU할당 scheduling.
6. Symmetric Multiprocessing (SMP): 각각의 CPU(core)가 각각의 scheduling 알고리즘을 가지는 방법
7. Asymmetric Multiprocessing: 한 CPU(core)만 system data에 접근 가능하여 그 한 CPU가 분배해주는 방법. 이 CPU(processor)를 Master server라고 함.
8. 분배 이슈 – process affinity: CPU마다 cash나 레지스터 값들 등으로 인해 특정 process가 수행하기 더 좋은 환경이 있음.
9. 분배 이슈 – load balancing
10. Push migration: 특정 task를 다른 CPU(processor)에게 넘기는 것.
11. Pull migration: 바쁜 CPU(processor)의 task를 가져오는 것.
12. Multi-Thread



1. Thread
2. Kernel thread: Kernel에 의해 생성되고 제거되는 thread. OS에 의해 관리됨.
3. User thread: Thread library에 있는 함수 호출로 인해 관리되는 thread. Kernel thread의 지원을 받지만 kernel에 포함되지 않음.
4. Lightweight process (LWP): kernel thread와 user thread를 mapping 시켜주는 역할을 하는 thread. 연결된 kernel thread가 block되면 LWP도 block되고 그 위의 user thread 또한 block됨.

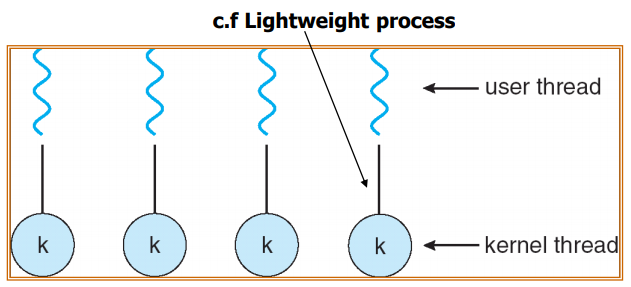


1. Multi-thread 장점
2. Responsiveness: 여러 개의 thread 중에 몇 개가 block되어있어도 남은 thread들이 응답할 수 있기 때문에 응답성이 좋다.
3. Resource sharing: Code, data등의 resource를 공유할 수 있다.
4. Economy: Process를 새로 create하는 것보다 메모리나 resource의 할당이 비용이 적게 든다.
5. Utilization of MP (multi-processor) Architecture: thread별로 다른 CPU에 할당하여 병렬적으로 활용할 수 있다.
6. Thread mapping model
7. Many-to-one: 모든 user thread가 하나의 kernel thread에 mapping되는 model. Thread의 관리가 user mode에서이루어 지기 때문에 빠르고 overhead가 작다는 장점이 있음. 하지만 kernel thread가 block되면 모든 user thread가 block 됨.

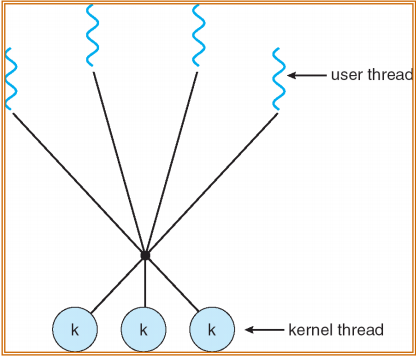


1. One-to-one: Linux와 windows에서 사용중인 모델.

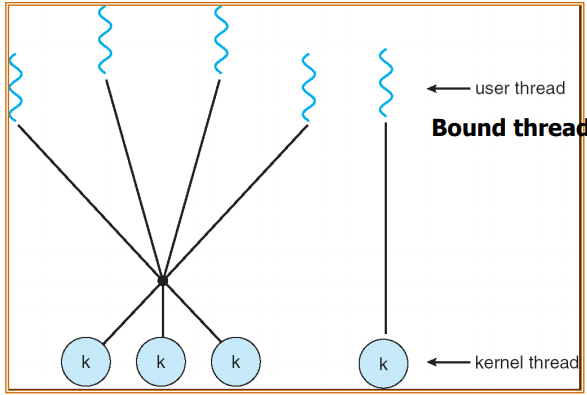
모든 user thread와 kernel thread가 일대일로 매칭되어 있음. 연결된 kernel thread가 block되었을 때만 user thread가 block되는 점이 장점. 이 특징으로 multiple thread를 병렬적으로 돌릴 수 있음. 하지만 user thread를 만들 때마다 kernel thread를 만드는 것은 비용(resource)이 많이 듦.



1. Many-to-Many: One-to-one 모델과 many-to-many 모델을 섞은 모델. User thread를 원하는 만큼 많이 만들 수 있고 kernel thread도 병렬적으로 돌릴 수 있다는 장점이 있음. 하지만 당연히 LWP에서 thread할당 이슈가 있음.



1. Two-level: Many-to-many 모델과 ono-to-one 모델을 함께 사용하는 모델. 중요한 thread에게 전용 kernel thread를 줄 수 있다는 점이 장점.

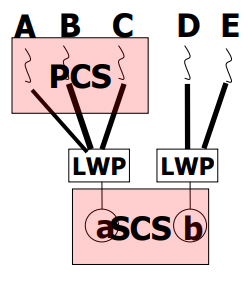


1. Thread scheduling
2. Local Scheduling: LWP에서 어떤 user thread를 동작 시킬지 scheduling

* Process-contention scope

1. Global Scheduling: 어떤 kernel thread를 동작 시킬지에 대한 scheduling.

* System-contention scope



1. Signal
2. Synchronous signal: interrupt에 의한 signal
3. Asynchronous signal: 동작 중인 프로세스의 외부 event로 인한 signal
4. Critical-Section Problem
5. Background
6. Critical-section: 공유 자원과 같이 여러 process가 한번에 접근하면 안되는 부분.
7. Race condition: 두 개 이상의 thread들이 공유된 자원에 접근하려 할 때 동기화 메커니즘이 없어 하나의 자원을 서로 사용하려 경쟁(race)하는 상황. 실행 순서에 따라 결과가 바뀌게 되어 있음.
8. Solution
9. Mutex (Mutual Exclusion): 한 process가 이미 공유 자원에 접근 중일 때 다른 process는 접근하지 못하도록 막는 방법.
10. Peterson’s Solution: critical section에 접근 중인 process가 있을 때는 특정 변수가 특정 값을 갖게 하여 critical section에 들어가기 전에 기다리게 한다.

Waiting이 너무 길어질 수 있으며 critical section이나 process가 많아지면 구현이 어렵고 중간에 끊길 수 있어 Atomic하지 않다는 단점이 있다.

1. Semaphore: 신호기, 깃발을 의미하며 각 process에 제어신호를 전달하여 순서대로 작업할 수 있도록 하는 기법. Wait(S), signal(S) 2개의 연산으로 처리. Semaphore를 lock의 수로 하여 여러 개의 lock에서도 처리 가능.

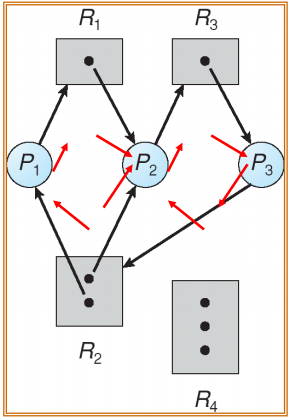
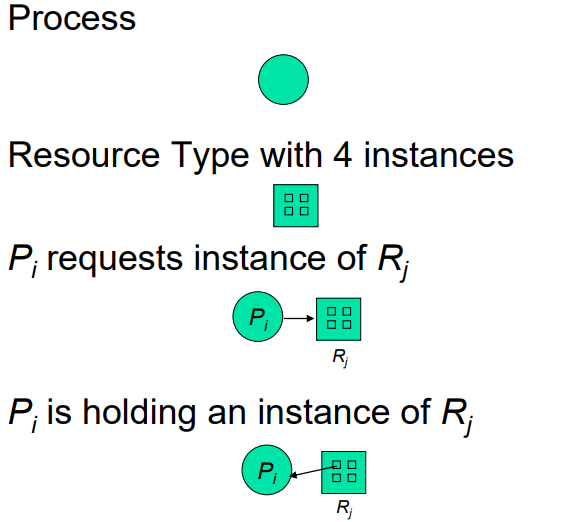
이때 lock을 얻을 수 있을 때까지 반복문으로 lock을 확인하는 것을 spin lock이라고 하며 이렇게 바쁘게 기다리는 것(CPU를 양보하지 않는 것)을 busy waiting이라고 한다.

Spin lock으로 context switch를 줄여 효율적일 수 있으나 processor(CPU나 core)의 수가 적다면 유용하지 않다. 이를 보완한 block, wakeup함수를 사용하는 경우도 있다.

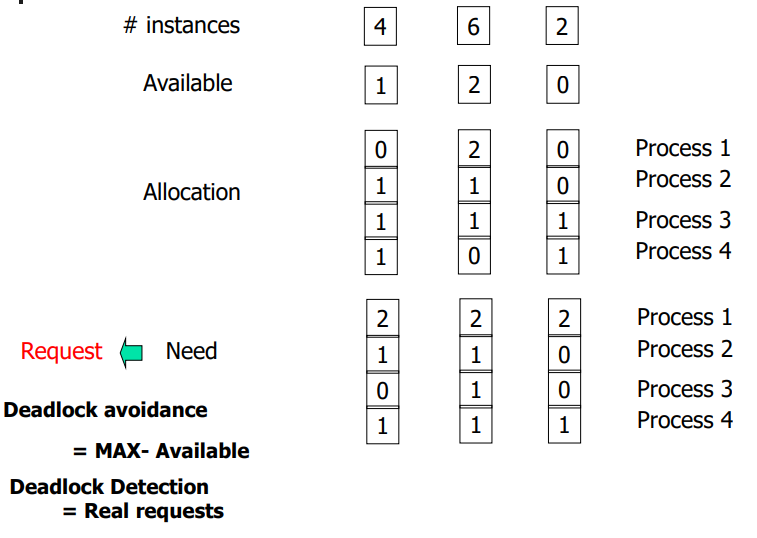
1. Monitor: 필요한 공유 자원을 process에게 할당하는데 필요한 데이터와 데이터를 처리하는 processor로 구성. 정보를 은폐하는 방식으로 다른 외부 프로시저가 접근하지 못하도록 한다. 한번에 하나의 processor만 접근 가능(wait, signal함수 사용).
2. Deadlock
3. 정의: 프로세스가 자원을 얻지 못해 다음 처리를 하지 못하는 상태
4. 발생 조건: 아래의 4가지가 모두 성립할 때 발생
5. Mutual exclusion (상호 배제): 자원은 한 번에 한 프로세스만 사용 가능해야 한다.
6. Hold and Wait: 최소한 하나의 자원을 점유하고 있으면서 다른 process에 할당되어 사용하고 있는 자원을 추가로 점유하기 위해 대기하는 process가 있어야 한다.
7. No preemption: 다른 process에 할당된 자원은 사용이 끝날 때까지 강제로 뺏을 수 없다.
8. Circular wait: process의 집합 {P1, P2, …, Pn-1}에서 P1은 P2가 점유한 자원을 대기하고 P2는 P3가 점유한 자원을 대기하는 등 P(i % n)은 P(i+1%n)가 점유한 자원을 요구해야 한다.
9. Deadlock detection
10. Resource Allocation Graph: single instance

아래 그림처럼 그래프를 그려 cycle이 없다면 deadlock이 없는 것.

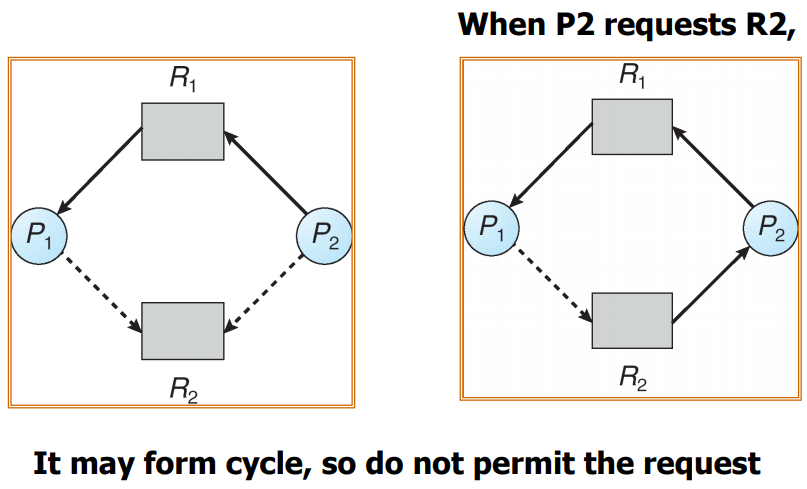
만약 cycle이 있다면 모든 process가 1개씩의 자원을 점유하고 있다면 deadlock이 있는 것, 아니라면 deadlock의 가능성이 있는 것.



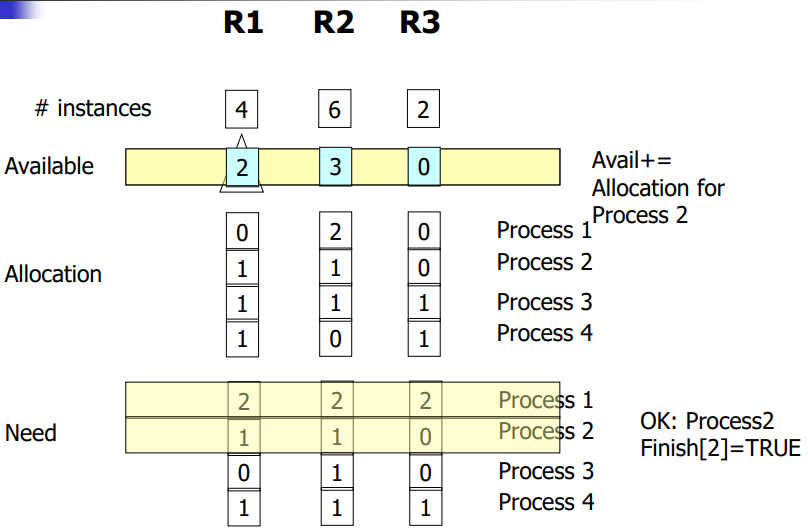
1. Corresponding wait-for graph: resource allocation graph를 resource그림을 상략하여 간단화 시킨 것.
2. Banker’s algorithm과 비슷한 방법. Need 대신 request로 현재 필요한 자원 수 사용. Safe sequence가 없으면 deadlock. (아래 그림은 기억 상기용)



1. Deadlock avoidance: deadlock 상태가 발생하면 피하는 방법
2. Resource allocation graph를 그려 사이클이 생길 것 같으면 자원의 점유를 허락하지 않는다.

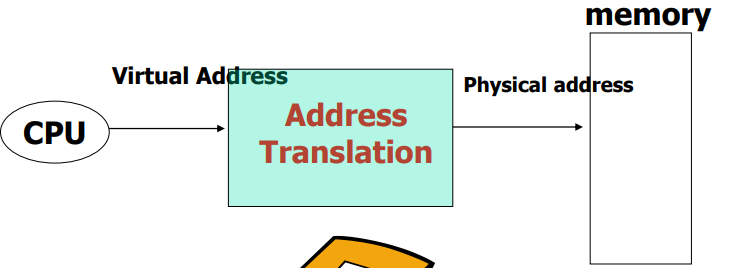


1. Banker’s Algorithm: 자원할당 요청이 들어오면 자원이 안전한지 여부에 따라 허락, 거부. Available: 공유가능한 자원 수, Allocation: process마다 점유하고 있는 자원 수, Need: process마다 더 필요한 자원 수. (아래 그림은 기억 상기용)

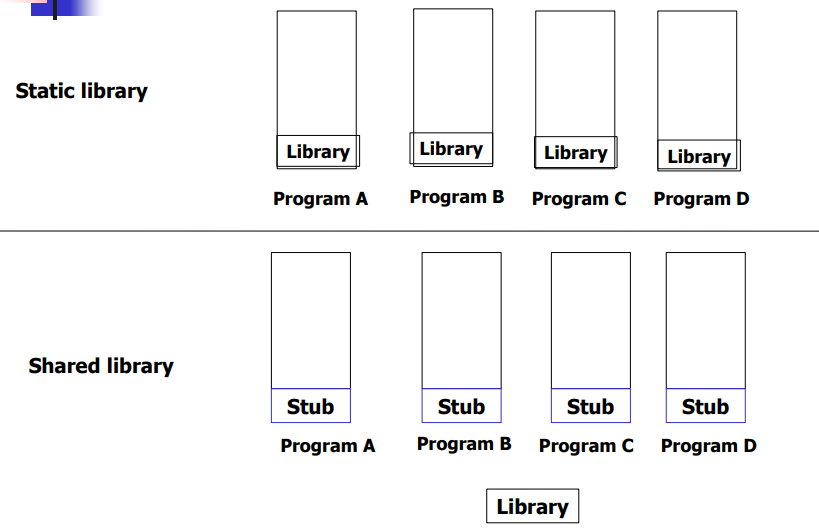


1. Deadlock recovery
2. Process 종료: deadlock 상태의 process를 모두 중지하고 deadlock이 풀릴 때까지 하나씩 종료한다.
3. Checkpoint & rollback: victim을 선택하여 safe state(deadlock이 발생하지 않는 상태)로 되돌린다.
4. Deadlock prevention: dead lock 발생 조건을 달성하지 못하게 하는 것. (자원낭비 심함)
5. 상호 배제(mutual exclusion) 부정: 여러 개의 process가 공유 자원을 사용가능 하게 한다.
6. Hold and wait 부정: process가 실행되기 전에 필요한 모든 자원을 할당한다.
7. No preemptive 부정: 자원을 점유하고 있는 process가 다른 자원을 요구할 때 점유하고 있는 자원을 반납하고, 요구한 자원을 사용하기 위해 기다리게 한다.
8. Circular wait 부정: 자원에 고유한 번호를 할당하고, 번호 순서대로 자원을 요구하도록 한다.
9. Memory management
10. Background
11. Address: Virtual(logical) address space > Physical address space

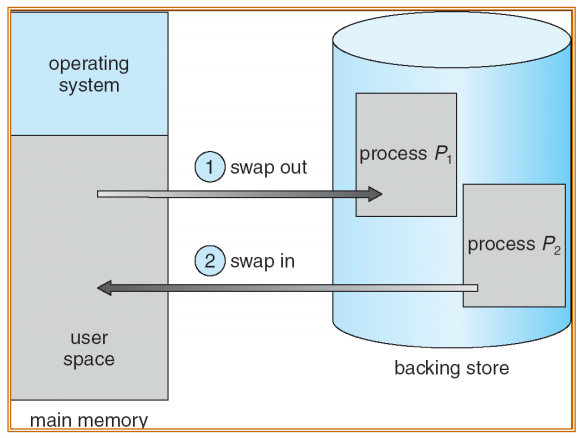
* Address translation 필요



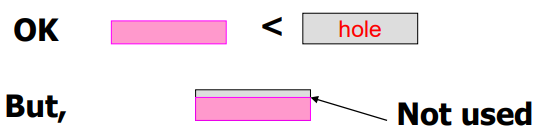
1. MMU (Memory Management Unit): address translation(mapping)에 도움을 주는 hardware
2. Shared library: 실행 시(runtime) library 메모리 주소 간단하게 불러올 수 있기 때문에 효율적임. Stub: library 위치(주소)



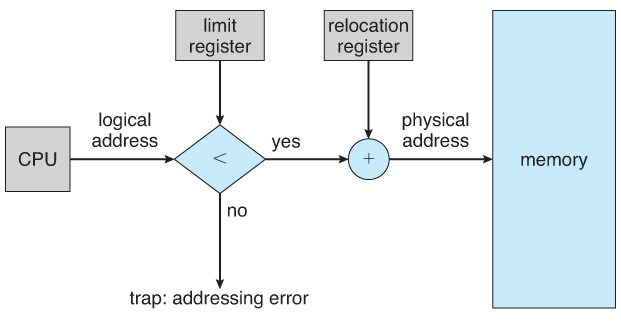
1. Swapping: 메모리에 올리는 process를 바꾸는 작업. Overhead임. 밀려난 process가 다시 메모리에 올려질 수 있을 때 있는 곳이 backing store



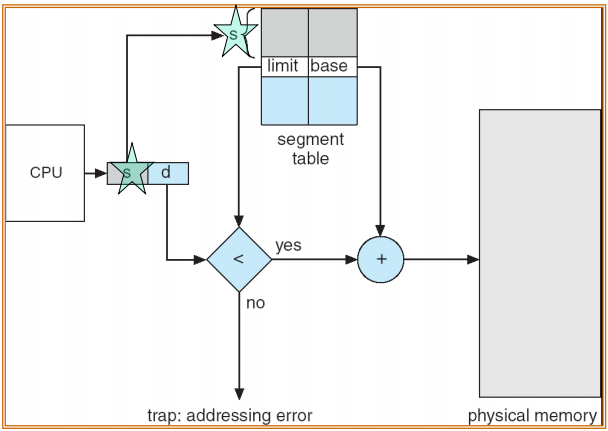
1. Hole: memory를 할당했다가 process가 나가 남은 빈 공간.
2. First-fit: 가장 첫번째 hole이 충분히 크다면 할당하는 방법.
3. Best-fit: 요구하는 크기 이상의 hole 중 가장 작은 hole을 할당하는 방법.
4. Worst-fit: 가장 큰 hole을 할당하는 방법.
5. Fragmentation(단편화)
6. External Fragmentation: 메모리 전체 hole의 크기 합은 process보다 크지만 process 크기보다 작은 hole들이 연속되지 않고 떨어져 있는 현상.
7. Internal Fragmentation: process 크기보다 hole의 크기가 커서 process가 올라갔지만 남은 빈공간이 생기는 현상.



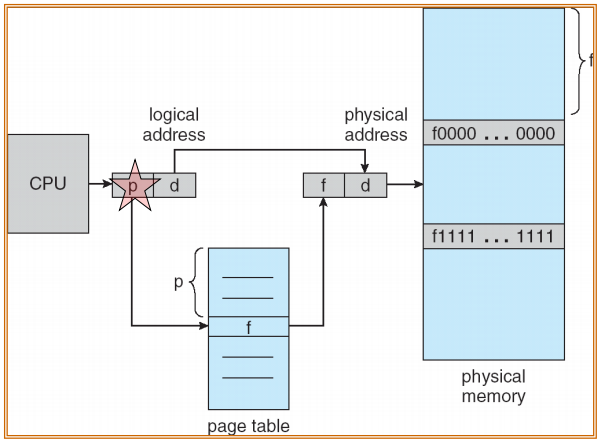
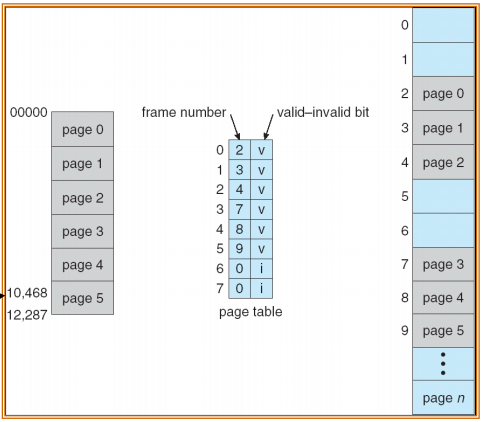
1. Memory management - Physical address와 Virtual address mapping 방법
2. Contiguous Allocation: process를 연속적으로 저장하는 기법. Process의 시작주소만 알면 됨.
3. MMU: Relocation register, Limit register
4. Protection: Relocation register ~ Relocation register+ Limit register가 process의 범위.
5. 이슈: Hole process가 들어갈 hole을 선택하기 위해 지속적으로 hole에 대한 정보를 유지해야 함.



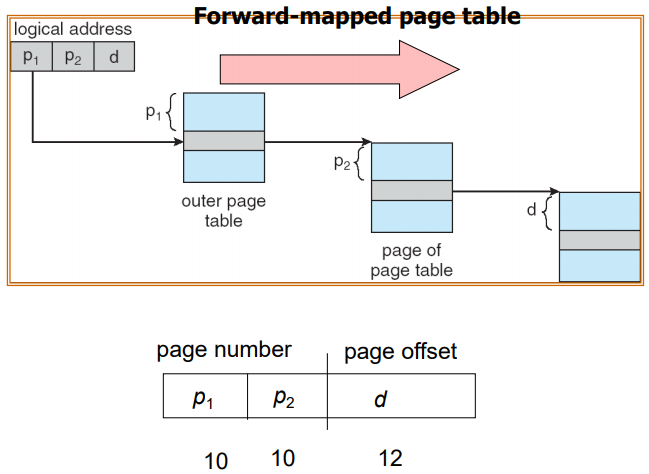
1. Segmentation: process를 segment 단위로 나눠 저장. Segment-number(s)로 base(시작주소) register값을 찾아 offset(d)을 더해 physical 주소를 찾을 수 있음.
2. MMU: Segment Table Base Register(STBR), Segment Table Length Register(STLR)
3. Protection: segment table의 validation bit로 illegal segment인지 판별, segment-number(s)<STLR이어야 legal.
4. 이슈: segment의 크기에 따라 segment table의 크기가 결정됨.



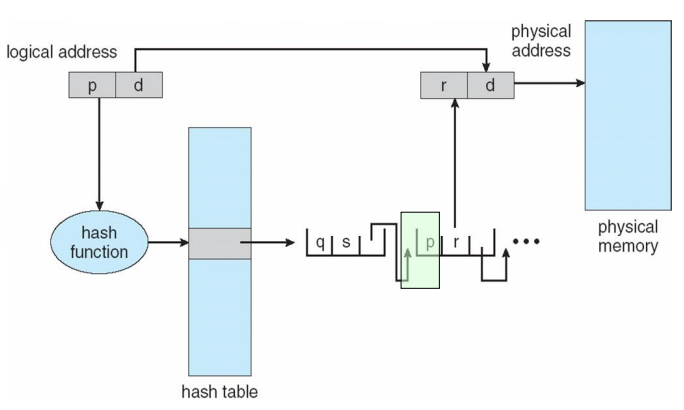
1. Paging: logical memory를 page크기로, physical memory를 frame크기로 잘라 mapping시키는 기법. Page number(p)로 page table에서 frame번호를 알아낸 뒤 offset을 뒤에 붙여 physical 주소를 찾는다. Page table(PCB)는 process마다 저장 되어있다.
2. MMU: Page-table base register(PTBR), page-table length register(PRLR), TLB(translation look-aside buffer): cache를 이용하여 접근했던 frame의 number를 빠르게 얻을 수 있게 함.
3. Protection: frame에 valid-invalid bit를 부여하여 확인한다.
4. 이슈: 정해진 page, frame크기로 인한 internal fragmentation 발생 가능. 모든 메모리 공간이 주어진 크기로 잘려 존재하기 때문에 External fragmentation은 발생하지 않는다. page크기에 따라 PCB의 크기 또한 결정되므로 잘 정해야 한다. 보통 4KB에서 8KB로 사용한다.



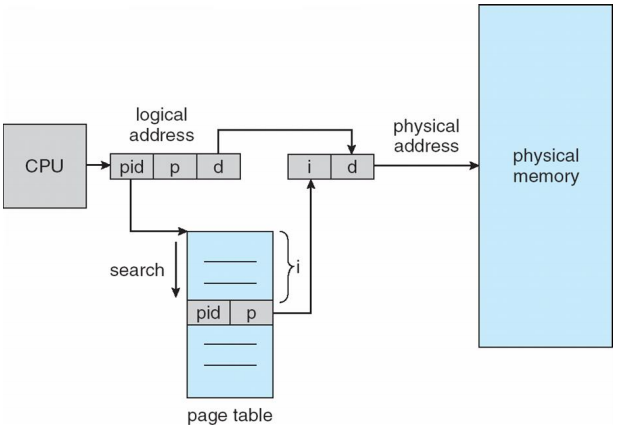
1. Page Table structure
2. Hierarchical Paging: 큰 주소공간에 대하여 계층적으로 주소를 찾을 수 있게 하는 방법. Ex) Two-level page table



1. Hashed Page Table: hash함수를 통해 mapping시키는 방법.



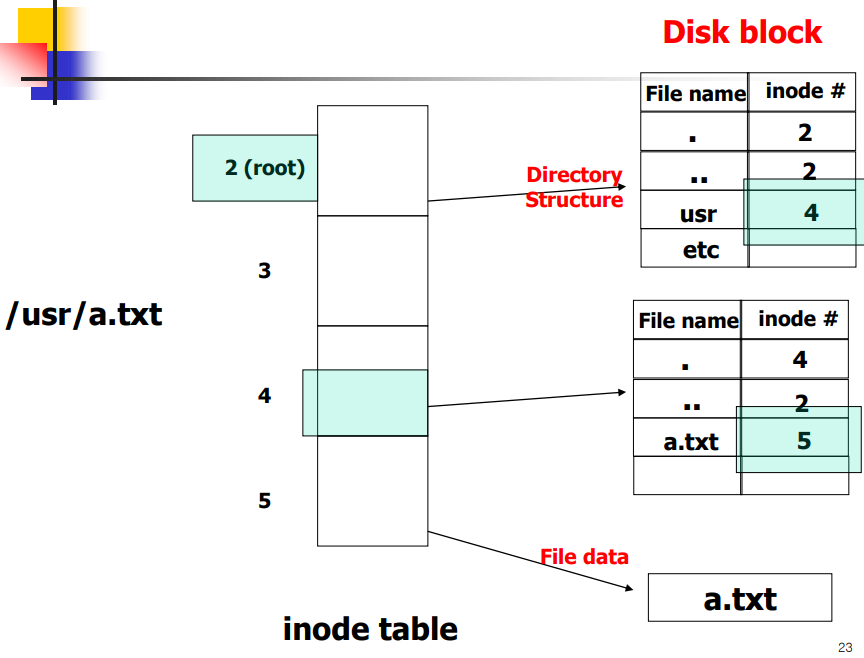
1. Inverted Page Table: page table의 크기가 너무 클 때 process당 1개씩 가지고 있게 하지 않고 system전체의 table을 하나 만들어 사용하는 방법. (PID를 얻어야 함.)



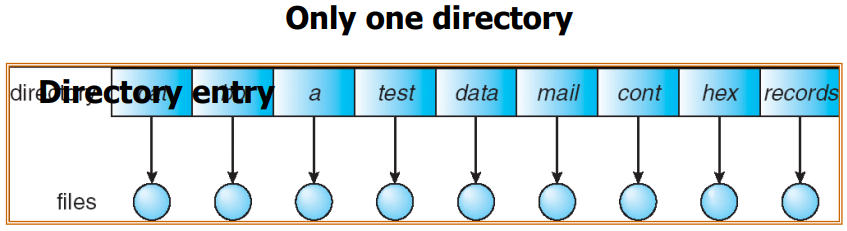
1. Memory management - Virtual memory: 프로세스 전체를 메모리에 올리지 않아도 실행 가능하도록 하는 기법
2. Demand paging: 필요한 process만 메모리에 올리는 방법.

Swaper가 전체 process에 대해 메모리에 올릴지 말지 결정한다면 pager(lazy swapper)는 process내의 개별 page들을 관리한다. Pager는 사용되지 않을 page를 메모리로 가져오지 않음으로써 시간, 메모리의 낭비를 줄인다.

1. valid bit: 어느 page가 디스크에만 있는지, 메모리에 올라와 있는지 구별하기 위해 valid-invalid bit를 사용한다.
2. page fault trap: Process가 메모리에 없는 page에 접근하려 할 때 page fault trap을 발생시켜 OS에 trap을 걸어 알린다.
3. Pure paging: 모든 page가 메모리에 올라와 더 이상 page fault trap이 발생하지 않는 상태
4. 성능 검사: Demand paging의 성능을 알아보기 위해서는 page fault rate, EAT(Effective Access Time)를 이용하여 체크한다.
5. Copy-on-Write: 부모 process가 자식 process를 생성할 때 원래는 자식에게 새 page를 할당하지만 COW에서는 변경이 이뤄져야 하는 경우에 한해서만 데이터를 복사하고 변경이 이뤄지지 않는 부분은 공유함으로써 메모리 낭비를 줄인다.
6. Page replacement: page-fault-rate를 낮추는 것을 목적으로 page를 교체한다.
7. FIFO: 먼저 메모리에 올라온 page를 내보낸다.
8. OPT (Optimal replacement): 미래를 보고 앞으로 가장 사용 안 될 page를 내보낸다. -> 사실상 불가능.
9. LRU (Least Recently Used): 최근에 가장 오랫동안 사용하지 않은 page를 내보낸다.
10. LFU (Least Frequently Used): 사용빈도가 가장 적은 page를 내보낸다.
11. MFU (Most Frequently Used): 사용빈도가 가장 많은 page를 내보낸다.
12. Trashing: 너무 많은 process가 동작하느라 process가 page swapping하느라 더 바쁘게 되는 현상. -> swapping이나 physical 메모리를 늘려야함.
13. Working Set: 어느 시점에 특정 process가 접근하는 page들의 집합
14. Working set model: 특정 page가 Working set안에 속하지 않으면 내보내는 방식으로 trashing을 예방.
15. Allocating Kernel Memory
16. Buddy System: 2의 n승으로 메모리를 분할(splitting)하여 할당. 자유로워진 메모리(버퍼)를 합치고 다시 분할하는 과정을 반복하며 할당.
17. Slab Allocator: 자료(리소스)의 타입에 따라 할당 방법을 다르게 함. Slab 단위로 메모리를 관리하면서 internal fragmentation을 해결하고 초기화 속도를 향상시킴. Slab은 물리적으로 연속된 메모리로 완전히 할당/일부 할당/비어 있는 상태 3가지로 나눈다. 메모리 할당 요청을 받으면 slab allocator는 미리 할당해둔 slab 속의 메모리를 준다. 메모리를 반환할 때도 slab allocator에게 반환하여 재활용하기 때문에 비용을 줄일 수 있다.
18. File system
19. File system: file operation과 directory를 관리하고 file 이름과 offset으로 disk block 주소를 알아낼 수 있게 함.
20. FCB (file control block): inode라고도 부름. file의 metadata. file들에 대한 정보들이 담긴 block. 크기, 권한, file data block 등.
21. inode 찾는 방법: inode table에서 찾으면 됨.

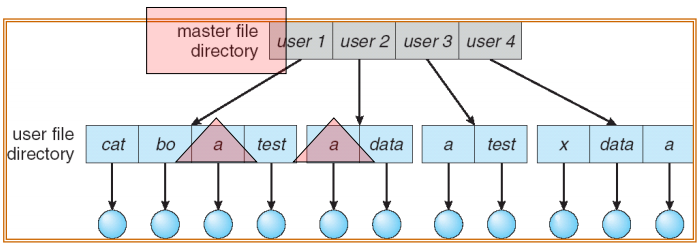


1. File 연산: create, write, read, reposition within file, delete, truncate
2. Partition: file system을 나누는 최소 단위. 같은 partition안에서는 같은 file system 사용. 같은 disk라도 다른 partition이면 다른 file system을 사용할 수 있음.
3. Directory structure
4. Single-level directory



같은 이름의 directory를 생성할 수 없기 때문에 name 이슈가 있음.

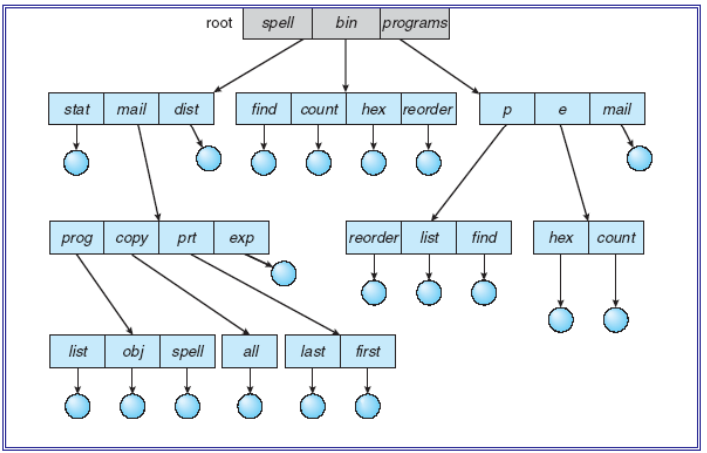
1. Two-level directory



User에 따라 directory를 나눈 구조. 파일 공유가 불가능.

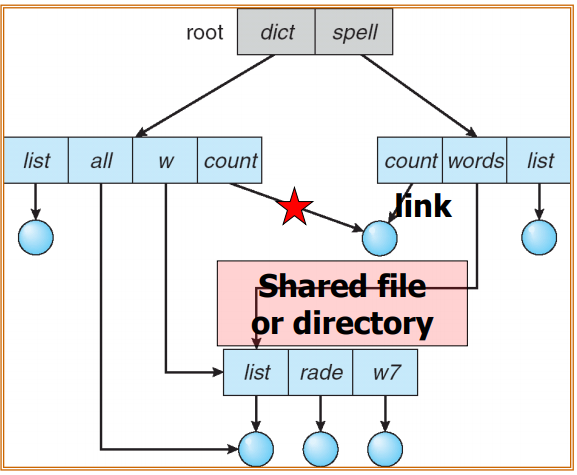
1. Tree-structured directory

Directory가 tree구조를 가짐. 경로에는 절대 경로와 상대 경로가 있음.



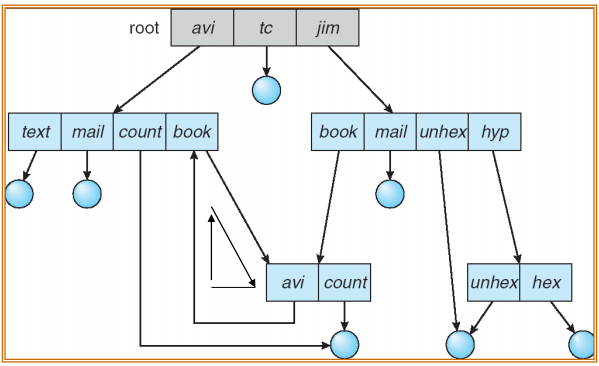
1. Acyclic graph directory (비순환그래프)

Tree structured 에서는 파일 또는 directory의 공유를 허용하지 않지만 Acyclic graph에서는 허용한다. 아래의 shared file에서 파일이 삭제되고 포인터만 남게 되는 dangling pointer 문제가 방생할 수 있다.

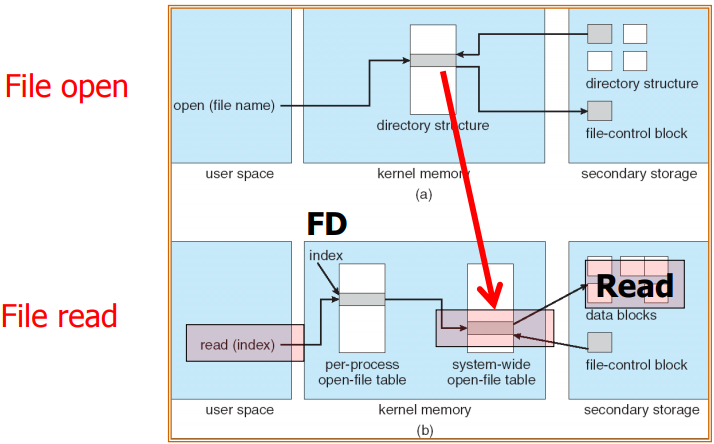


1. General Graph Directory

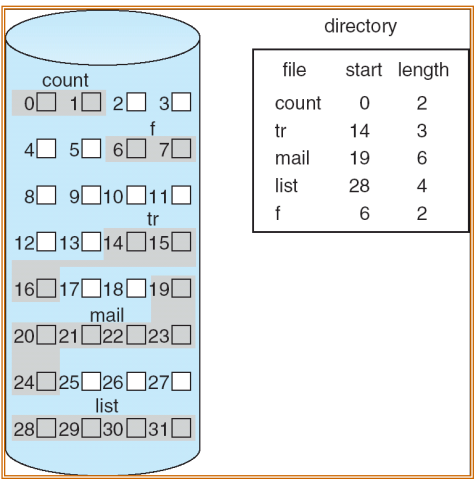
비순환그래프에서의 dangling pointer문제를 해결한 구조. 해결 방법은 공유하는 파일에 cycle을 만들어 원본 파일을 알 수 있게 한다. 사이클이 허용되기 때문에 search방법이 더 어려워진다.



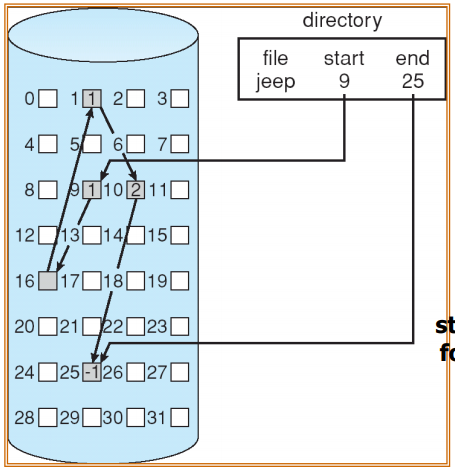
1. Link
2. Hard link: i-node를 공유하여 파일 공유. 파일이 바뀌면 공유하는 파일들 모두 바뀌는 것. 원본 파일이 삭제되어도 파일의 link count만 감소시키면 되므로 링크에는 문제가 없다. 이름만 다른 같은 파일인 것.
3. Soft link (symbolic link): 바로가기와 같은 것. i-node도 다르고 원본 파일이 삭제되면 같이 삭제된다.
4. File을 위한 table들
5. System-wide open-file table: 열려 있는 file의 FCB를 복사하여 가지고 있는 table
6. Per-process open-file table: system wide open file table의 주소와 함께 file의 offset과 권한 등을 나타내는 table



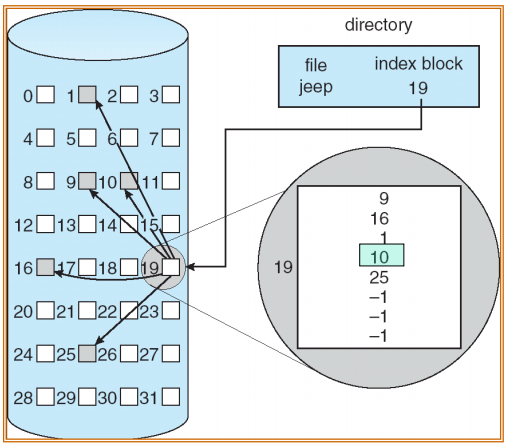
1. VFS (Virtual File System): 파일 시스템 인터페이스를 유저 space에서의 지원 혹은 다양한 파일 시스템을 지원하기 위해서 구현된 kernel의 서브 시스템. Vnode라는 VFS에서의 inode가 존재하고 File system과 user space 사이에 존재. inode, file, superblock(마운트 된 파일 시스템의 정보를 저장하는 block), dentry(디렉토리 항복과 이에 대응하는 파일의 연결에 대한 정보) 4가지 object를 제공.
2. Directory 구현
3. Linear list: 선형 리스트로 파일 이름과 data block을 가리키는 포인터들을 저장. 당연히 선형 리스트이기 때문에 search시 느리다.
4. Hash table: hash 함수를 사용하여 파일 이름으로부터 data block을 가리키는 포인터를 얻을 수 있게 하는 방법. Hash 특성상 Collision(충돌)에 관한 이슈가 있다.
5. Allocation method
6. Contiguous allocation: 각 파일이 디스크 내에서 반드시 연속적인 공간을 차지하도록 할당하는 방법. External fragmentation이 발생할 수 있고 파일을 위해 얼만큼의 공간을 할당할지 정해야 함.



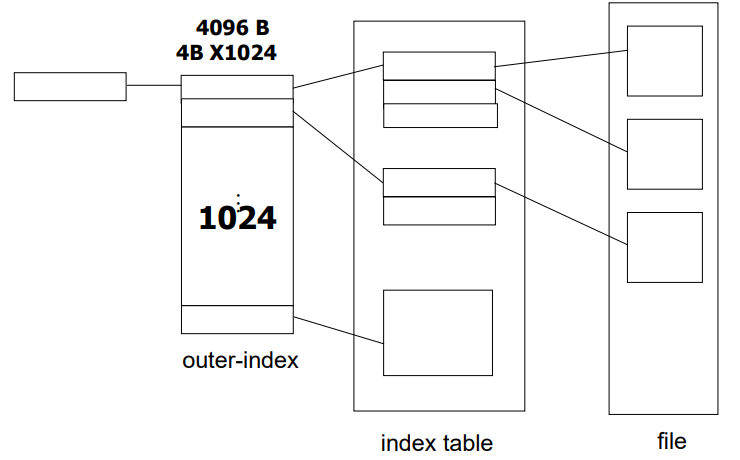
1. Linked allocation: 디렉토리에는 시작블록과 마지막 블록만 가리키고 블록과 블록들이 서로 가리키는 포인터가 있어 처음 블록부터 마지막 블록까지 이어진다. i번째 블록을 찾으려면 i번 포인터를 거쳐야 하기 때문에 비효율적일 수 있다.



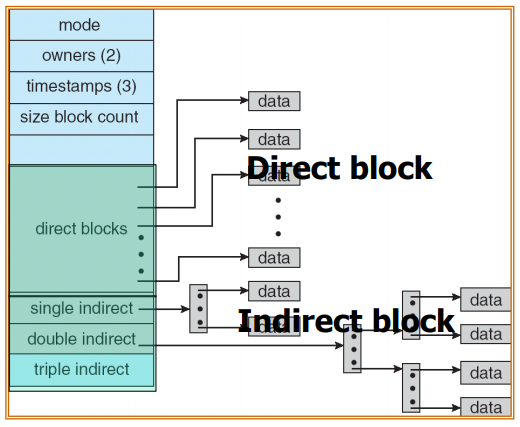
1. Indexed allocation: 포인터들을 한 곳에 모아 놓는 방법. 블록 수가 많아지면 포인터가 모여 있는 index block의 크기가 커지기 때문에 잘 결정해야 함.



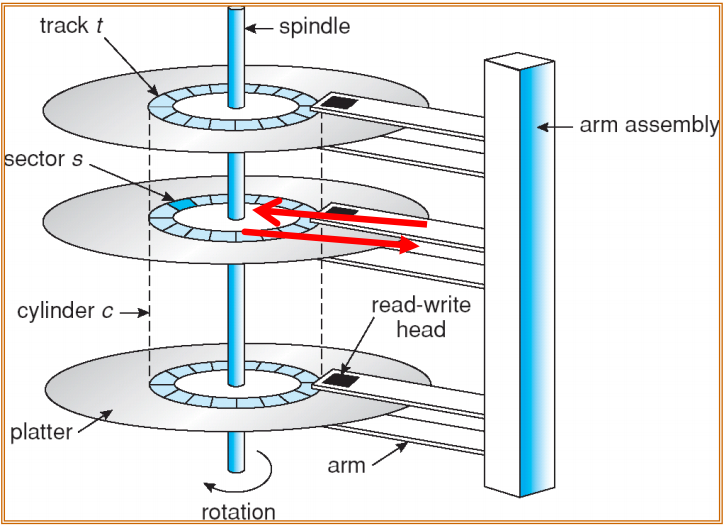
1. Linked scheme (연결 기법): index block의 크기가 커지면 여러 개의 index block들을 연결한다.
2. Multilevel Index: 여러 계층의 index block이 가리키게 하는 방식.



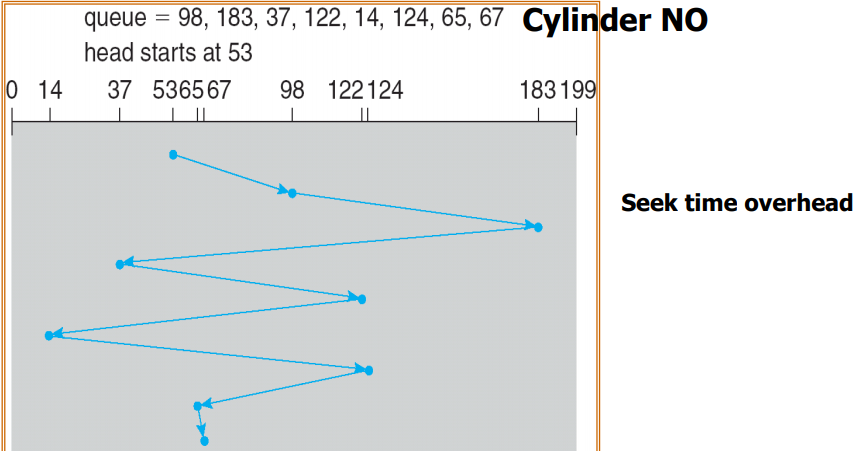
1. Combined scheme: (ㄱ), (ㄴ)을 결함 시킨 방법.



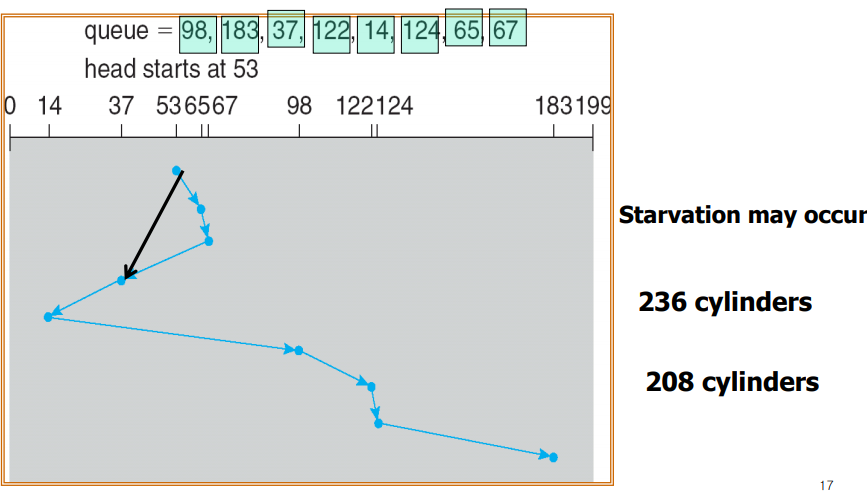
1. Storage Structure
2. Background
3. Transfer rate: drive에서 computer 사이의 데이터 전송률(속도)
4. Seek time: disk arm에서 목표 cylinder로 이동하는데 걸리는 시간
5. Rotational latency: 목표 sector가 회전하여 disk head 아래로 가는데 걸리는 시간
6. Positioning time (random-access time): seek time + rotational latency
7. Head crash: disk head가 disk surface에 닿는 문제.
8. Access time: seek time + rotational delay(latency) + transfer time
9. Disk Structure



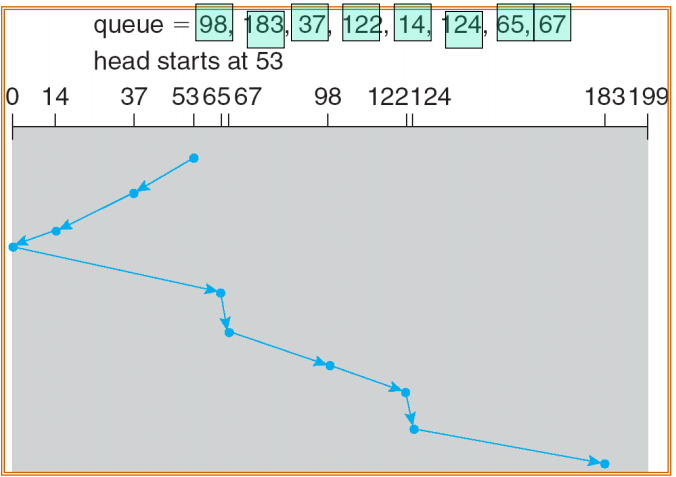
1. Disk Scheduling
2. FCFS: 빨리 오는 것부터.



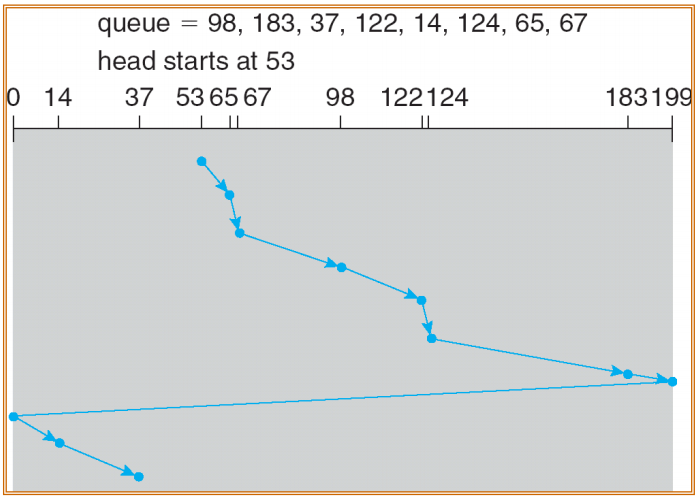
1. SSTF: seek time(거리)이 짧은 것부터. 짧은 것들만 돌다가 starvation 발생 가능.



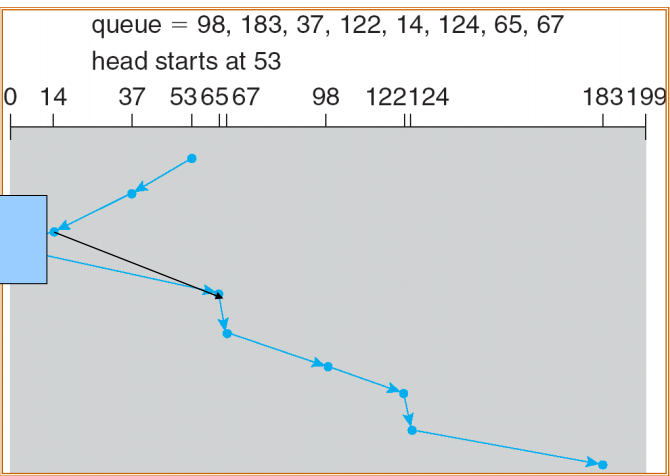
1. SCAN (elevator algorithm): head의 범위 끝[0, 199]을 오가면서.



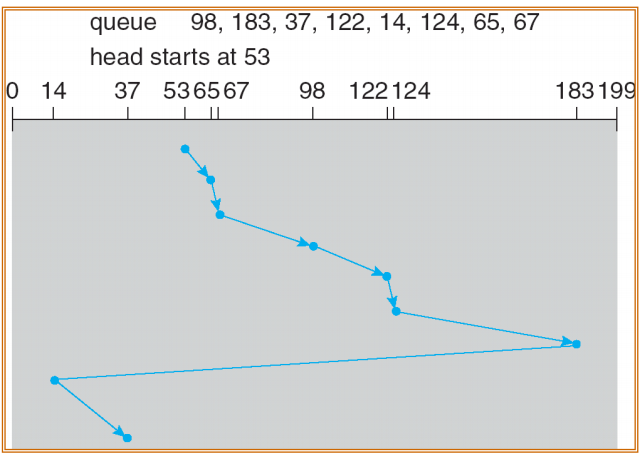
1. C-SCAN (one direction): head 범위의 처음에서 끝으로 이동하면서. 끝에 오면 다시 처음으로.



1. Look algorithm: queue에서 최솟값 쪽으로 갔다가 최대값 쪽으로 반복



1. C-Look: queue에서의 queue에서의 최댓값 쪽으로. 최댓값 후에는 최솟값으로 간 후 다시 증가하는 방향(one direction)



1. Disk Management
2. Disk formatting: low-level혹은 physical formatting이라고도 함. Disk를 sector들로 나누는 작업.
3. RAID Structure: 여러 개의 하드디스크를 하나의 하드디스크처럼 사용하는 방식. 비용을 절감시키고 신뢰성과 성능을 향상시킬 수 있음. RAID를 구성하는 디스크의 개수가 같아도, 구성 방식(RAID level)에 따라 신뢰성을 높이거나 성능을 높일 수 있다.