

OS 의 최우선 과제 :

Convenience for users(Virtualization, Abstraction),
Efficiency & Fairness on resource management

OS 의 역사

(처음엔) 손수 코딩한 바이너리 프로그램을 진공관 스위치로 실행하기

- Batch Processing (한 번에 한 가지 태스크만 실행)
- Multi-programming Systems (I/O-CPU Overlap, 그러나 여전히 off-line batch processing 방식)
- Time Sharing Systems (on-line processing, interactive CRT Terminal, Round-Robin Scheduling)
- Personal Computers (microprocessor = 1 chip CPU)
- Today's Servers (Modern OS features : GUI, Multimedia, WWW, Network)

Multi-Processor Systems (SMP)

- 1 shared common memory
- memory bottleneck ++ per CPU local cache
- S/W is easy but H/W is difficult
(이유 : 소프트웨어 입장에서는 메모리가 하나이고 CPU 도 한 번에 하나씩 사용하므로 Uni-Processor 와 마찬가지로)

Multi-Processor Systems (NUMA: Non-Uniform Memory Access)

- one memory for each CPU
- 각 CPU 는 서로의 Memory 에 접근할 수 있다. 이 과정에서 시스템이 더욱 느려질 수도

Distributed Systems

- network-based collaboration of multiple computers
- each node has its own local memory? (NUMA 와 다르게 다른 컴퓨터의 memory 에 접근 불가능)
- 목적 : Resource Sharing, Parallel Computing, Reliability
- H/W is easy but S/W is difficult
- 예시 : NFS (분산 파일 시스템), Cloud Systems

Cloud Computing

- Distributed Servers + Hypervisor + Virtual Machines

Virtualization Technology

- emulation S/W of a single computer
(Simulation: 소프트웨어의 동작을 흉내내는 것)
(Emulation: 하드웨어의 동작을 흉내내는 것)
- Hypervisor : VM, guest OS 관리, VM 의 H/W 작동을 native H/W 장치와 연결
- type : Full-Virtualization(전 가상화), Para-Virtualization(반 가상화)

Docker (2013, Docker Inc. Open Source Container Project)

Para-Virtualization : Share the host & kernel

- Container???

VM 을 실행하면 성능이 좋지 않고 하이퍼바이저의 매핑이 반드시 필요하며 대용량의 이미지(.iso file)가 필요하다. 이런 한계를 넘어서기 위해서 light-weight virtualization technology 로 세상에 등장했다.

Linux Kernel 의 cgroups(CPU, Memory, I/O, Networks 등 자원 활용을 제한하는 기법), namespace(각 태스크 별 isolation 이 가능) 을 이용한다.

Embedded Systems

- a special purposed S/W system on a special purposed H/W
- 예시: 소비자 가전제품, 통신기기, 스마트폰, 군사용 무기, 자동차, 항공기, 의료 기기
- 지금은 IoT 라는 용어가 널리 사용됨

Realtime Systems

- : 태스크를 처리완료하기까지 시간 제한이 존재하는 경우
- : 너무 늦을 경우 심각한 문제가 발생하는 경우

Computing Environment

- Traditional (Main Server ↔ Terminals)
- Client-Server computing
- Peer-to-peer(P2P) computer
- Web-based computing

“The future is already here. It’s already distributed widely.” - 교수님의 말씀

Smart ICT Trends

원래는 manufacturing capability 가 중요했지만 현재는 사용자 관점의 S/W capability 가 중시된다.
legacy products,
new service, (context), IoT, big data, cloud, data analytics, machine learning, insight
=> convergence
(e.g. fin-tech)

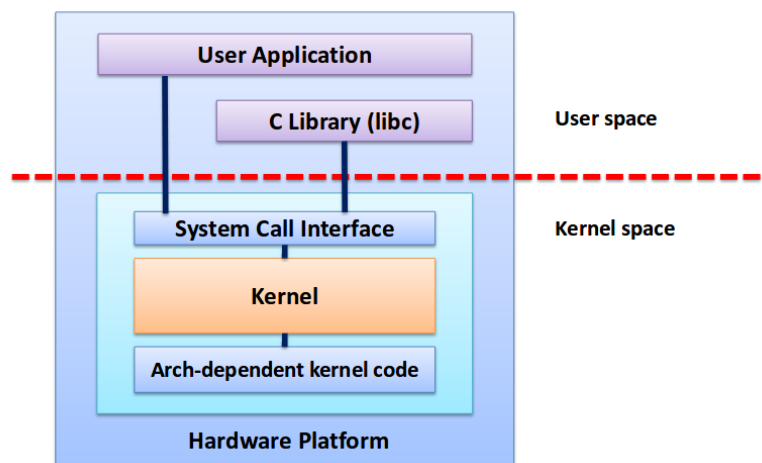
IoT

RFID/USN/M2M 등은 근거리망, 센서 중심으로 이루어져서 단순히 정보를 수집하고 모니터링 하기만 하는 단순한 시스템이었다. 서비스도 통시적인 관점에서만 이루어졌다.
그러나 사물인터넷 시대에는 인터넷(장거리 통신망), 각종 다양한 기능과 자율 판단 기능, 모니터링을 기반으로 한 자율제어, 수백 억 개의 사물을 다루는 시스템이 등장했다. 서비스도 리얼타임이 가능해졌다.

Big Data

수많은 센서가 발생시키고 수집해오는 스트리밍 데이터를 분석해서 아주 짧은 시간에 의사결정을 가능하게 만들.
각종 비즈니스 프로세스와 다채널 실시간 마케팅과 같이 시간에 민감한 프로세스에서 중요하게 활용됨

Blockchain/Bitcoin



Kernel : 항상 메모리에 상주하는 운영체제의 핵심적인 부분

(왜 메모리에 있음? 자주 사용하니까. 결국 커널도 하나의 프로세스임)

- Operating System : Kernel + System Process + Libraries + Utility Services

- kernel엔 뭐가 있을까?

1. system call functions

2. interrupt handlers = device drivers

- what is kernel? A resource manager, A provider of Virtual Service (encapsulation of H/W)

1) 프로세스, 스레드 관리 2) 프로세스 스케줄링 3) 메모리 관리 4) 디바이스 드라이버 5) IPC(Inter Process Communication) 6) 네트워크 7) 파일시스템

Process

- program in execution: 프로그램은 보통 디스크에 저장되어있다. 그걸 메모리로 끌어올리면 프로세스

- PID(프로세스 아이디)로 식별 가능

- state 가 있다 (running or stopped)

- context 가 있다 (current register contents, memory contents 등등)

- 하드웨어 보호 기능:

1) execution of privileged instructions by user codes 는 실제로 only executable inside the kernel (I/O instructions, Special instructions such as halt)

2) division between memory areas of the kernel and processes

Dual Modes “kernel-mode vs user-mode”

User Mode:

만약 privileged instructions 를 실행하려고 하면 exception/fault (abort)

만약 memory invasion 발생하면 segment violation

Kernel Mode:

in a system call / in an interrupt handler

PSW(Processor Status Word) 레지스터의 running mode flag 를 바꾸면서 표시한다

모든 privileged instructions 수행 가능

모든 memory protection 의 제한을 넘어설 수 있음

시스템 내에서 여러 개의 프로세스가 병행적으로 (concurrently) 동작할 수 있는 이유는?

CPU/IO Overlap 으로 극적인 성능 향상 가능

Time-Slice 방식으로 cpu sharing 이 일어남

Preemption 도 있음

cpu 를 점유하는 주체가 변하는 경우 : on time-slice burst, on task complete, with preemptino

Monolithic Kernel

하나의 커널 속에 커널의 모든 기능이 구현되어 있음 (similar to a single huge object)

커널 속에는 여러개의 layers 가 있다.

machine-independent vs machine-dependent layers

file-system independent vs file-system dependent layers

Micro-kernel

핵심적인 서비스만 구현된 커널 (프로세스/스레드 관리, message-based-IPC, Clock IRQ)

나머지 부가적인 기능은 또다른 (서버) 프로세스로 실행한다.

장점 : Flexibility

I/O System

디바이스도 파일처럼 다뤄짐 (대신에 특별한 파일) - 그러므로 open, close, read, write 로 작동함
커널 속에 있는 I/O API 는 IO Request Block 을 만들고 그걸 device drive queue 에 들여보냄

Device Controller (H/W 구현)

Controller

communicates with device driver (S/W) of a kernel.

control the physical device to perform actual I/O

- command(instruction) register : 요청으로 들어온 I/O 명령

- status register : BUSY, DONE (done 으로 set 되면 interrupt 가 발생하고 커널은 I/O 가 완료되었다는 것을 알게 된다)

- data buffer register : I/O 의 결과로 얻은 데이터

Interface < Controller < I/O Processor (성능 비교)

I/O Processor < Controller < Interface (속도 빠르기 비교)

Device Driver (S/W in Kernel)

I/O 컨트롤러로 인해서 interrupt 가 발생하면 그 즉시 커널은 IRQ Handler 코드를 실행한다.

1) character-oriented device

2) block-oriented device – DMA 사용

커널이 interrupt 를 포착하는 방법

1. Polling (= busy-waiting)

cpu 가 기다리는 동안에 loop 안에서 I/O 가 완료되었는지 아닌지 확인하고
완료되었을 때에만 break;

=> cpu cycle 을 심각하게 낭비

2. Hardware Interrupt (=interrupt-driven I/O)

과정

::: cpu 는 I/O 커맨드를 인터페이스로 보냄

→ cpu 는 I/O 가 이루어질 동안에 다른 작업을 실행 (CPU/IO Overlap, 성능 향상)

→ 디바이스 내에서 I/O 작업이 완료되면 I/O controller 는 cpu 에 그 사실을 알리는 인터럽트를 발생시킴

→ cpu 는 현재 작업을 일시정지하고 IRQ Handler(in kernel)을 실행하기 시작함 (결과값 처리 및 다음 I/O 작업을 위한 초기화)

→ 인터럽트 핸들링 완료후, 이전 위치 또는 커널의 프로세스 스케줄러로 되돌아감

커널 내부에서 interrupt handling 은 어떻게 이루어질까?

과정 (커널모드에서 이루어짐)

::: current state of the cpu(이를 context 라고 한다)를 안전한 위치에 저장

→ 어떤 종류의 인터럽트인지 확인 (보통 interrupt vector 를 사용)

→ 해당 인터럽트를 처리하는 곳으로 이동 (ISR: interrupt service routine)

→ 처리가 끝나면 다시 프로세스 스케줄러 / 유저모드로

I/O 의 종류 (H/W 수준)

- Isolated I/O : special cpu instructions for I/O (reading, writing the interface registers (status, buffer) are used

- memory mapped I/O : device controller(or interface) registers are mapped into the designated memory locations. reading and writing data, info, command from/to the special locations of memory.

I/O 시스템의 종류 (어떤 단위로 이루어지나)

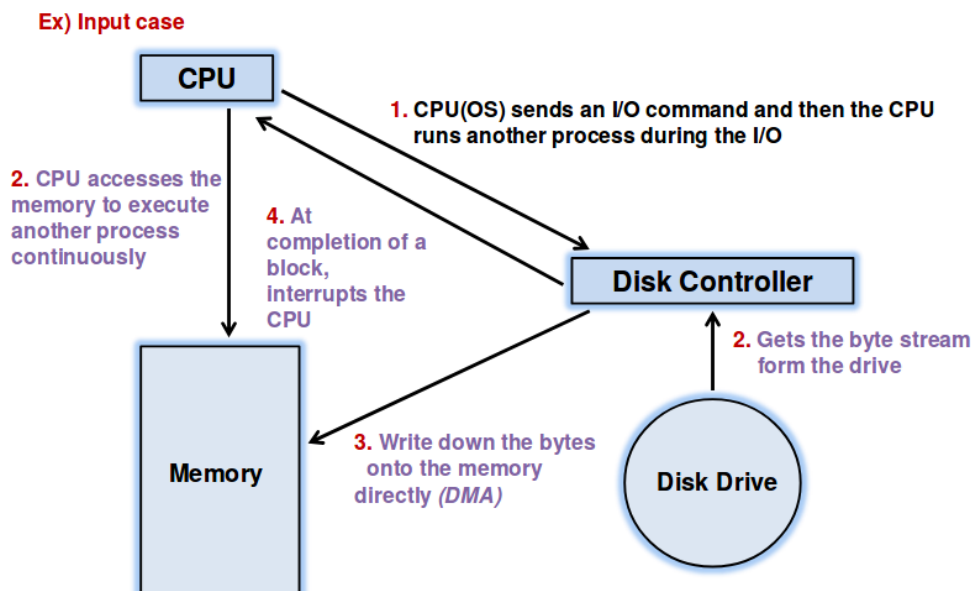
- character oriented I/O : 한 글자마다 interrupt 가 발생한다
- block oriented I/O : 물리적 블록을 단위로 해서 인터럽트가 발생한다. (DMA, cache 등의 기법을 활용)
- Block-Oriented IO 의 경우

DMA(Direct Memory Access)를 활용하는 경우가 많다. 원래는 데이터를 I/O Device 로 전달하는 과정까지도 cpu 가 담당해야 했다. 그러나 I/O controller 가 직접 메모리에 접근해서 필요한 데이터를 가져오므로써 그 시간동안에 cpu 는 다른 태스크를 처리할 수 있다.

DMA 를 통한 data retriever 가 완료되면 cpu 인터럽트 발생 (I/O 완료를 알리고, 커널로부터 다음 I/O 명령을 가져오기 위해서)

Cycle Stealing

Block-Oriented IO 방식일 때 DMA 를 사용하는데, 분명 이 방식은 cpu 의 할일을 줄여주는 효과가 있다. 그러나 DMA 는 cpu 가 관장하는 부분이 아니다보니 cpu 가 태스크를 처리하는 과정에서 필요로 하는 메모리 영역과 겹칠 수도 있다. 그럴 경우에는 항상 IO controller 가 해당 메모리에 대한 우선권을 갖는다. cpu 의 접근 요청이나 속도가 훨씬 빠르기 때문에 더 오랜 시간이 걸리는 IO controller 의 태스크를 먼저 완료하도록 한다.



I/O 시스템의 변화 과정: Busy Waiting → Interrupt → DMA + Interrupt

Multi-priority Interrupt

우선순위

1. 기계 결함
2. 전원 나감
3. 클락 인터럽트 : periodic interrupt (1/1000 초 또는 1/100 초 단위로) 인터럽트 중에서는 가장 중요한 인터럽트
4. 디스크, 네트워크 I/O : DMA 사용
5. Cha-oriented I/O

인터럽트 핸들링

과정

::: 모든 인터럽트를 중단 (이 인터럽트가 다른 인터럽트에 의해서 중단되는 것을 막기 위해 시작하자마자 이 작업)
→ 현재 context(register contents)를 저장한다. 인터럽트 핸들링이 종료된 뒤에 이 위치로 돌아와야 하므로

- 높은 우선순위 인터럽트를 허용한다 (interrupt masking)
- device interface 에 ACk 를 날린다. (해당 인터럽트가 cpu 에 의해 성공적으로 캐치 되었음을 알린다)
- input / output 작업을 실시
- 다음 IO 를 위해서 초기화
- 모든 인터럽트를 허용하도록 만듦
- 다시 중단한 위치로 돌아간다.

Memory Hierarchy

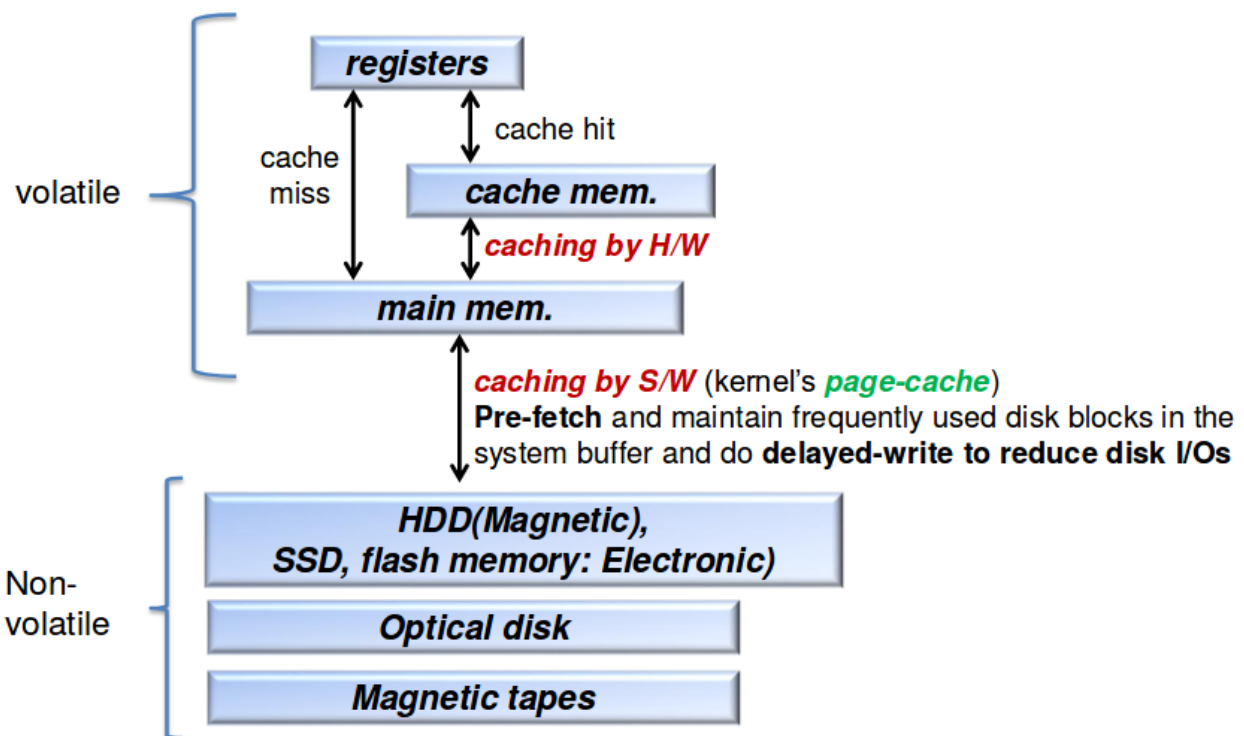
cpu – register – memory – disk/flash memory

빠르고, 작고, 값비싼 메모리를 cpu 가까운 곳에 위치시킨다.

빠른 메모리는 다음 레벨의 보다 느린 메모리에 대한 캐시로 사용한다.

Cache 는 왜 할까?

- ㄱ) 거대한 용량을 다루기 위해서 ㄴ) 빠른 액세스를 위해서



레지스터, 메인메모리 + 캐시는 volatile 메모리이다.

디스크는 non-volatile 메모리이다.

용량이 가장 크며 속도가 가장 느린 디스크에 대한 캐시는 메인메모리(RAM)이다.

ㄴ 이것이 caching by S/W 이다. (kernel's page-cache) pre-fetch 와 delayed-write 를 통해서 디스크 IO 횟수를 감소시킨다.

메인메모리에 대한 캐시는 캐시메모리이다.

- ㄴ 이것은 caching by H/W 이다.

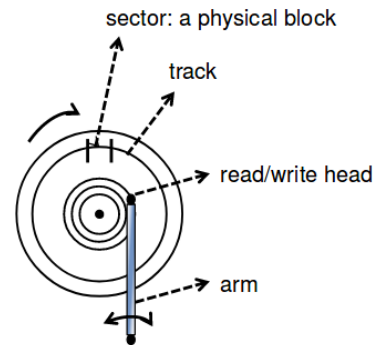
cpu 는 register 에서 값을 탐색하고 없으면 cache memory 에서 값을 탐색한다.

cache-hit / cache-miss 에 따라서 다음 동작은 달라진다.

Hard Disk Access

▪ Generally, a moving-head magnetic disk (too slower than the CPU)

- **Track/sector**
- **Cylinder** (the group of tracks with the same position in a multiple disk array)
- Motion control and data transfer are done by the disk controller
- Access time
 1. **Seek time** : time taken to position the head on the wanted track, $O(10ms)$, **much slower.**
 2. **Rotational latency**: wanted sector, $O(1ms)$
 3. **Read/write time** = transfer time
- **Reducing of disk access time**
 - **S/W caches** (kernel *page cache*,..)
 - **Disk scheduling** to minimize the avg. *seek time* (covered later !)



하드디스크는 헤드를 움직여서 정보를 가져오는 자기식 저장장치.

트랙(플래터 위의 동심원), 섹터(트랙 위의 한 지점, 데이터가 저장되는 장소), 실린더(disk assembly 에서 여러 플래터 중 동일한 위치의 트랙 모음)

Disk Controller - 헤드를 움직이고 데이터를 읽고 쓰는 작업을 관리

Access Time

1. Seek Time($\pi\pi\pi\pi\pi$) 2. rotational latency (τ) 3. read/write time ($o - o$)

액세스 타임을 줄이기 위해서 “S/W 캐싱, 디스크 스케줄링 등이 등장”

Process in the Kernel

모든 프로세스는 하나의 상태에 존재한다.

1. running : cpu 에 의해서 실행되고 있는 경우 (kernel-mode 또는 user-mode)
2. ready : 스케줄링을 기다리고 있는 상태. 중단된 상태라고 봐도 된다. (cpu 사용 X)
3. blocked : 이벤트를 기다리는 중. 이것도 중단된 상태 (cpu 사용 X)

- 세 번째인 blocked state 는 scheduling queue 가 아니라 sleeping queue 에 들어가 있는 상태이다.

- 예시 :

I/O 요청을 날린 뒤 I/O 가 완료되기를 기다리고 있는 프로세스

mutex 가 다른 프로세스로부터 unlock 되기를 기다리고 있는 프로세스

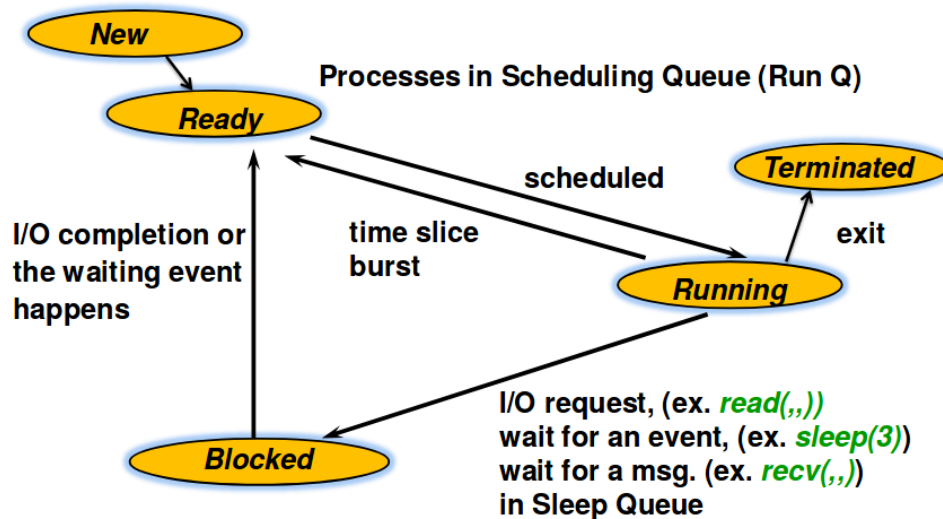
IPC 메세지 수신을 기다리고 있는 상태

공유 자원(e.g. memory page, printer, etc)을 점유하는 걸 기다리고 있는 상태

* 블록 되는 순간??? cpu 점유를 다른 프로세스에게 넘긴다. 그리고 즉시 I/O 요청을 보낸다

I/O 가 끝날 때까지 다시 스케줄링 되지 않는다.

Process Blocking Mechanism enables CPU/IO Overlap!



새로운 프로세스 생성

스케줄링 큐(Run Q)에 들어감 (1 Ready)

스케줄 되어서 cpu 를 점유하면 (2 Running) - 태스크 완료시 exit

타임슬라이드 완료 시 다시 스케줄링 큐로 들어감 (1 Ready)

실행중에 종료 또는 타임슬라이스 완료 외의 다른 이유로 중단되면 (3 Blocked)

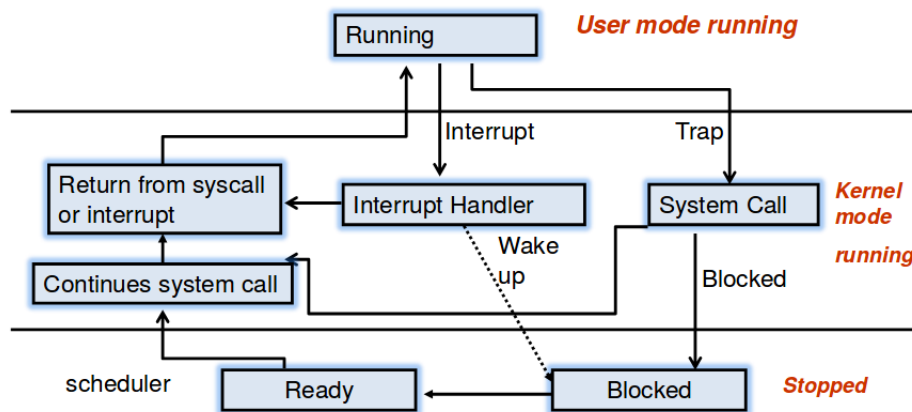
read() 처럼 I/O 가 발생하거나

sleep() 처럼 이벤트(시그널)를 기다리거나

recv() 처럼 메시지를 기다리거나

이 모든건 웨이팅 큐(Sleep Q) 안에서

IO 작업이 완료되었거나 기다리던 이벤트가 발생하면 해당 프로세스는 다시 스케줄링 큐로 들어감 (1 Ready)



"Wake up" means making a **blocked** process to be **ready**

[kernel-mode vs user-mode 차원에서 살펴보자면]

처음에 프로세스는 user-mode 에서 (2 running) 상태

system call 호출로 kernel-mode 로 진입

커널모드에서 해야할 처리를 마친 뒤

그 프로세스는 (3 Blocked) 상태로 바뀜

...

나중에 IO 완료했다는 인터럽트 발생하면 wake-up

스케줄링 큐로 들어가면서 (1 Ready)

다시 cpu 를 점유하게 되면
system call 에서 하던 일을 끝나치고
리턴
다시 user-mode 로 돌아감
중단되었던 라인에서 다시 시작 (2 Running)

Linux Process States

#define TASK_RUNNING	0 // running or ready
#define TASK_INTERRUPTIBLE	1 // blocked
#define TASK_UNINTERRUPTIBLE	2 // blocked
#define __TASK_STOPPED	4 // stopped or debugging
#define EXIT_ZOMBIE	16 // zombie state (exit but not yet parent wait)

TASK_INTERRUPTIBLE

blocked for slow IO (e.g. getchar()) IO 를 하다가도 signal 발생 시 중단됨

TASK_UNINTERRUPTIBLE

blocked for fast IO (e.g. read() on disk)

I/O 에 걸리는 시간이 짧게 지정되어 있다.

해당 상태에 놓인 프로세스는 어떤 시그널에 의해서도 깨어날 수 없다.

모든 시그널 핸들링은 I/O 완료된 이후로 이뤄진다.

(c.f. TASK_KILLABLE: -9 SIGKILL 시그널은 받아들일 수 있으므로 임의 중단종료 가능)

메모리 공간

Text : 명령어 코드 (RD only)

Data/bss : 전역 변수

Heap : 동적 메모리 할당

Stack : 지역 변수, 함수 호출 프레임

PCB (Process Control Block)

프로세스에 대한 메타데이터

process-id, user-id, binary program that executed this program,

*scheduling priority,

*state (read, running, blocked...),

*the event waiting for,

resource allocated, working directory, memory management info (memory address space size,

virtual memory management information such as page tables... -later)

*machine context

*pending signal

etc.

Asynchronous Concurrent Programming

시스템에는 하나의 cpu 밖에 없지만 스케줄링 큐에 있는 모든 프로세스는 running – stop 반복적으로 오가면서 cpu 에 의해서 실행된다. 사용자 입장에서 모든 프로세스가 동시에 실행되는 듯한 느낌을 받는다.

물론 multi-core 환경에서는 각 cpu 마다 하나의 프로세스를 처리해서 실제로 parallel programming 이 될 수도 있다.

Asynchronous? 프로세스 상태 변화는 cpu 와 커널 내부 사정에 달린 일이므로 사용자는 순서를 알 수 없다.

cpu 가 프로세스에 대한 처리를 중단 stop 하는 순간 process context 를 모두 저장한 뒤에 다른 프로세스 처리하기 위해서 넘어가야 한다. 그래서 방금 중단되었던 프로세스를 다시 실행하기 시작할 수 있다.

text, data, heap, stack 정보 (user-level context)

++ 이것들은 process state 에 영향을 받지 않으므로 Memory 에 저장된다. 특별히 신경쓰지 않아도 됨

System-level context

++ cpu register contents : 다른 프로세스를 실행하려면 레지스터 내용이 바뀌어야 한다. 재시작하기 전에 복원

Context Switch

어떤 이유에서든 프로세스가 중단되면 (blocked on I/O request, ready on time-slice burst), cpu 에 대한 점유권은 다른 프로세스로 넘어간다.

A 프로세스에서 B 프로세스로 'jump' 전에 A 프로세스의 context 는 전부 저장되어야 한다.

그리고 이제부터 시작될 B 프로세스가 중단되기 이전 context 를 다시 실행할 수 있는 상태로 불러와야 한다.

이런식으로 스케줄링이 반복되다보면 언젠가는 A 프로세스를 다시 실행할 차례가 온다.

그러면 다시 A 프로세스의 context(system-level context)를 전부 다시 불러와서 실행할 준비를 해야한다.

PC(Program Counter), SP(Stack Pointer), etc.

이 모든 과정은 항상 **Kernel 안에서만** 발생하는 것

So Process are always stopped in a kernel and resumes in a kernel.

- 정의 : 하나의 프로세스에서 다른 프로세스로 cpu 점유권을 넘어가는 것

- 오버헤드 :

saving and loading registers and memory maps

flushing and reloading the memory cache

updating various tables and lists, etc.

스케줄링 큐 (run_queue in Linux)

A Linked List (or a Tree) of PCBs of runnable (running, ready)

일반적으로 multi-level priority queues

Kernel 속의 "current" 가 현재 running PCB 를 가리키고 있다

Sleep Queue

A Linked List of blocked PCB per event

event 1 (disk I/O) : PCB – PCB – PCB

event 2 (KB I/O) : PCB – PCB – PCB

event 3 (network) : PCB – PCB – PCB

커널속으로 trap 되는 경우는 언제일까?

1. system call 호출시 (privileged instructions 는 kernel-mode 에서만 실행 가능) trap 명령어

2. H/W interrupts (IRQ Handler) 발생 시

3. Fault or Exception (Fault Handler)

divide by zero, segmentation fault => 인터럽트 핸들링과 동일한 방식으로 처리됨

Page Cache (S/W Cache)

1. Pre-fetch : 한 바이트만 요청할지라도 해당 디스크 블록 전체를 디스크->메모리로 가져온다.

2. 가져온 블록은 메모리에 최대한 오래도록 저장

3. 다음 IO 발생시, 우선 해당 디스크 메모리를 포함하는 디스크 블록이 메모리에 올려져 있는지 확인

4. 있을 경우 : HIT, 디스크 I/O 없이 해당 메모리 블록을 가지고 처리 (write 의 경우도 데이터를 디스크까지 보내지 않고 메모리에서만 실행, 이를 delayed-write / dirty write 라고 함)

5. 일정 시간이 지난 후 또는 일정 조건에 도달하면 dirty 로 표시된 캐시 블록을 전부 sync 한다. (write from memory to disk) 이때가 진짜로 기록되는 순간.

6. 없을 경우 : MISS, 1 을 실행

Disk Interrupt Handling

I/O 가 완료되었을 때 발생하는 인터럽트

1. 해당 IO 를 요청한 프로세스를 Wake-Up

process state : Blocked => Ready

PCB : Sleep Q => Run Q

2. 다음 IO 요청을 위한 초기화

과정 (프로세스 X 가 디스크 I/O 를 요청했다고 가정한다)

::: 모든 인터럽트 disable

→ 프로세스 X 의 context 를 모두 저장

→ interrupt mask (보다 높은 우선순위 인터럽트만 허용하는 상태)

→ 디스크 I/O 스케줄링 큐에서 첫번째 블록을 빼낸다

→ 처리...처리...처리...

→ 해당 I/O 요청한 PCB 를 sleep Q 에서 꺼냄

→ 해당 PCB 의 프로세스 상태를 Ready 로 바꿈

→ 스케줄링 큐에 PCB 를 집어넣음

→ 디스크 IO 스케줄링 큐에 커맨드가 남아있으면 initialization for the next I/O

→ 인터럽트 서비스 루틴을 종료

Clock Interrupt Handler

- Clock tick : 클럭 인터럽트는 시스템이 구동되는 동안에 1/1000sec, 1/100sec 마다 주기적으로 발생

- S/W 인터럽트 중에서는 가장 높은 우선순위

- 시스템 타임과 관련이 있다 (jiffy++: 부팅한 순간부터 매 순간)

- sleep() / alarm() 시스템 호출도 다 이것과 관련이 있다

- 주기적인 시스템의 임무를 수행, 예를 들면

Timeout Functions (periodic calls of kernel functions), 프로세스의 타임 슬라이스가 완료되었음 알려줌

Wakeup (blocked → ready) : 오래 걸리는 작업은 인터럽트 핸들러 안에서 하지 않고 커널 프로세스에 의해서 수행된다. 왜냐하면 “인터럽트” 라는 것은 (그 안에서 context switch 가 일어날 수 없으므로) 빠르게 완료되어야 하기 때문이다. system processes 또한 user processes 처럼 스케줄되어야 실행할 수 있다는 것은 마찬가지이다. 대신에 priorities are higher.

Clock Interrupt Handler

과정

::: 시스템 타임을 업데이트 (jiffy++)

→ 커널 타임아웃 함수를 호출 (매 n 클럭틱마다 호출)

→ sleep() 처럼 타임큐에 잠들어 있는 프로세스를 Wake-Up

→ 우선순위를 재조정 (왜??? 타임 슬라이스의 완료 시 PCB 가 큐에 enqueue, dequeue 되고 이에 따라서 starvation, shared-lock 등을 고려하여 priority 를 재조정하게 된다)

→ current_PCB.time_slice_left -= 1;

if (time_slice_left == 0) current_PCB.need_resched = true;

```

→ RETURN from interrupt : 시스템 호출
do_softIRQ( ); // 부드러운 인터럽트 핸들링?
If (current_PCB.need_resched)
{
    current_PCB.state = Ready;
    choose the highest priority from the scheduling queue;
    do context_switch(...);
}

```

System call functions	Interrupt handler
<pre> sys_call_routine() { : IO request, lock-waiting, etc. Voluntary Context Switch; (blocked) : : : ret_from_syscall: If rescheduling is necessary, do Involuntary Context Switch; } </pre>	<pre> irq_routine() { : interrupt handling no context switch during in an interrupt handling !! (because it is a jump) : : : ret_from_syscall: (= ret_from_interrupt) If rescheduling is necessary, (ex. time slice burst) do Involuntary Context Switch; } </pre>

context switch 가 일어나는 경우 3 가지

=> 시스템 호출 중간에 I/O request 발생하는 경우 (lock-waiting) (이걸 voluntary CS)

=> 시스템 호출이 완료되었을 때 (involuntary CS)

=> 인터럽트 핸들러 함수가 완료되었을 때 (involuntary CS)

** 인터럽트 핸들러 중간에는 cs 가 일어나지 않는다.

그러므로 IRQ Handler 안에서는 mutex 나 sleep 등이 존재할 수 없고 신속하게 태스크 완료해야 한다

Concurrent Programming 의 예시

networked (or distributed) applications : 클라이언트-서버, 클라우드 컴퓨팅

real-time applications (input: sensors / output : actuators) *actuator : 기계를 동작시키는 동력원

parallel applications : 3D graphics, multi-core demanding tasks

CPU/IO Overlap : 프로그램 수준에서

asynchronous event handling : 여러개의 input sources 를 다룸. Input sources 의 순서가 임의 발생

multi-threaded server : worker model, peer model, pipeline model (웹서버)

Process Hierarchy

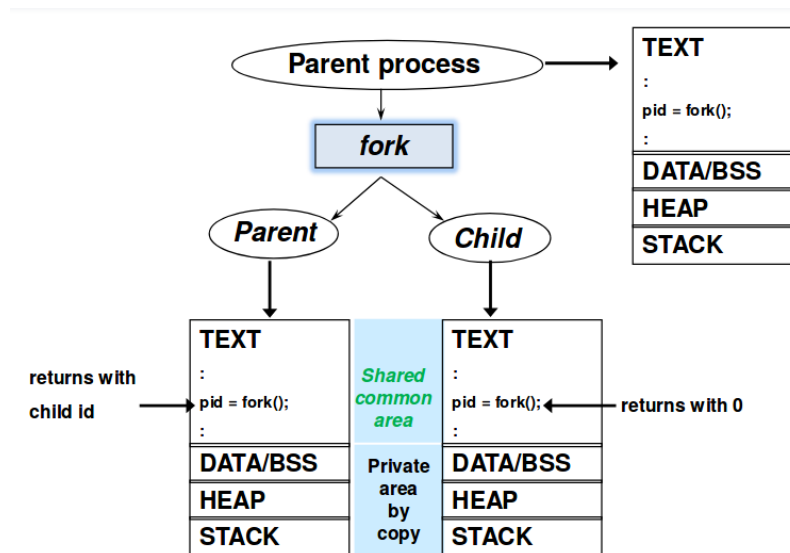
- 프로세스는 다른 프로세스를 생성할 수 있으며 parent-child relationship 을 갖는다

- Linux calls the hierarchy a “process group”

- Background Processes : 사용자와 interaction 없음 (Daemons 라고 부름)

fork() system call { pid_t fork(void); } : 프로세스 생성

fork() 를 통해서 만들어진 프로세스 사이의 관계



parent 쪽에서 fork()의 return value 는 child process id

child 쪽에서 fork()의 return value 는 0

parent-child 사이에는 TEXT 메모리 영역만 공유되고 나머지는 전부 각각 다름

fork()하기 직전까지의 리소스는 양쪽 모두에 할당됨 (open files, r/w offsets)

이후로는 data, stack 이 복사되어서 전혀 다른 프로세스로 생성됨

wait() system call { pid_t wait(int* wstatus) } : 프로세스 join 할 때까지 calling process 대기
프로세스 내에서 fork() 개수만큼 wait() 도 호출해야 함

Process Termination

: Normal Exit, Error Exit (voluntary)

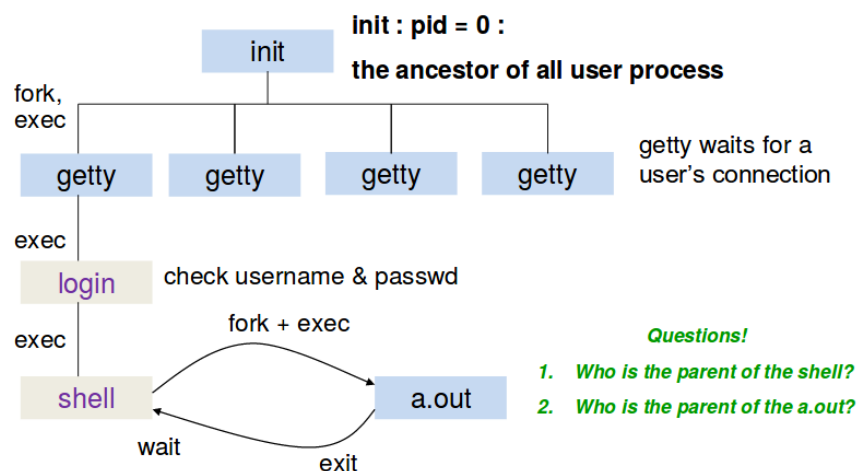
: Fatal Error (involuntary) : exceed allocated resources, segmentation fault, protection fault

: Killed by another process (involuntary) : receiving a signal

exec() system call family

- argument 로 함수에 전달한 프로그램을 해당 프로세스에 로드한 뒤에 프로그램을 main()부터 실행하기 시작하는 것. 새로운 프로세스를 생성하는 건 아니다

- program body 는 바뀌지만 프로세스에 대한 다른 모든 정보는 그대로 유지된다.



윈도우에서의 CreateProcess() 명령어는 fork() + exec() 를 합쳐놓은 것과 같음

IPC (Inter-Process Communication)

방법 : sharing files, sharing file pointer of an opened file, message queue, **Semaphore**, signals, pipe (pipe = circular queue between process)

Across machine :

network sockets, RPCs(remote procedure calls), java RMI (remote method invocation)

pipe

inter-process communication 에 사용됨

하나의 파일을 두 개의 offset 으로 관리 (r / w pointers)

empty pipe 에 대한 read 시도는 블록된다

블록되었던 read 을 재개하는 방법은??? write occurs /or/ the pipe closed

Synchronization

서로 다른 프로세스는 서로의 작업 진행 정도에 영향을 받을 수 있다.

Determinacy

Mutual Exclusion

Synchronization

Starvation

Deadlock

Determinacy

물론 concurrent-process 를 실행하면 매번의 실행때마다 순서는 뒤죽박죽 서로 다르게 실행될 수 있다. 그러나 최종적으로 볼 때 동일한 input 에 대해서는 항상 동일한 output 을 얻을 수 있어야 한다.

Precedence Graph 를 그려서 설계한다.

두 개 이상의 프로세스 사이에 공유하는 변수가 있으면 실행 순서에 따라서 최종 결과가 달라질 수 있다. 이런 방식으로 만들어지는 수 많은 프로세스 사이의 관계를 determinacy 를 잃어버린 것

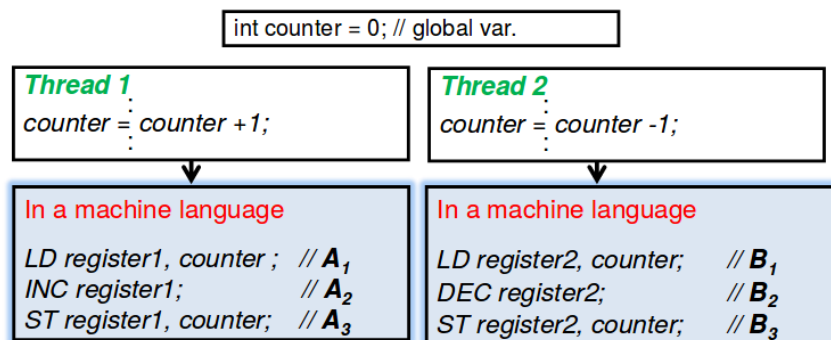
(cyclic precedence graph 이 등장할 경우를 배제하면 determinacy 를 보존할 수 있다)

conceptual language 를 사용해서도 분석 가능 : begin/end & parbegin/paraend +++ fork & join & goto

Critical Section

프로그램은 실행 중에 다른 프로세스/스레드와 전역 리소스를 공유하는 경우가 발생한다. 두 프로세스의 태스크에 따라서 어떤 순서로 해당 리소스에 접근하는가에 따라서 결과가 다르게 나타날 수 있다.

Race Condition



- scenario 1: **A₁ A₂ A₃ B₁ B₂ B₃** → counter will be 0
- scenario 2: **A₁ A₂ B₁ B₂ B₃ A₃** → counter will be 1
- scenario 3: **B₁ B₂ A₁ A₂ A₃ B₃** → counter will be -1
- **No one knows which scenario will be taken because of the concurrency !**
- Solution: **During execution of A₁ A₂ A₃, the execution of B₁ B₂ B₃ must be delayed until A part finishes or vice versa.**

dirty read, dirty write 문제가 발생할 수 있다.

The Classic Example (Bank Account)

A, B 두 사람이 하나의 계좌에서 동시에 100 만원을 인출하는 경우를 생각해보자. 두 사람은 서로 멀리 떨어진 두 대의 ATM 기를 사용한다고 가정.

함수: 인출하기 (계좌, 금액)

```
{
  잔액 = 잔액읽어들이기(계좌);
  잔액 -= 금액;
  잔액기록하기(잔액);
  반환 : 잔액;
}
```

위의 과정 중에서 어느 위치에서 context switch 가 발생할지는 전혀 예측할 수 없다.

Thread 1	Thread 2	Balance
Read Balance : \$1000		\$1000
	Read Balance: \$1000	\$1000
	Withdraw \$200	\$800
Withdraw \$200		\$800
Update Balance: \$(1000-200)		\$800
	Update Balance: \$(1000-200)	\$800

Shared resource 에 접근하면서 어떤 동기화 매커니즘도 사용하지 않았기 때문에 문제가 발생했다.

Race Condition 이란

=> the situation where several processes access and manipulate shared data concurrently, and the resource is non-deterministic and depends on timing.

1> Thread 간에

local variables 는 공유되지 않는다. (데이터가 스택에 기록되는데, 각각의 스레드는 ‘고유한 스택’ 을 갖는다)
global variables 는 공유된다. dynamic objects 는 공유된다. 모든 스레드

2> Process 간에

shared-memory object, files 가 공유된다.

Mutual Exclusion

프로세스가 critical section 에 진입할 때 배타적인 실행권을 얻기 위한 프로토콜이다.

하나의 프로세스가 critical section 에 관련된 부분을 실행하는 중에 context switch 가 발생하더라도 다른 프로세스는 critical section 에 진입할 수 없다.

Enter a critical section

다른 프로세스가 critical section 에 접근중인지 확인한다.

Yes 일 경우, 그 프로세스가 critical section 에서 해야할 일을 완료할 때까지 wait 해야 한다

Exit a critical section

다른 프로세스에게 critical section 에 접근할 수 있다고 알려준다.

Implementation (구현할 때 고려해야할 점) (S/W 로도 H/W 로도 양쪽 모두 구현 가능하다)

1. lock-unlock 기능이 있어야 한다

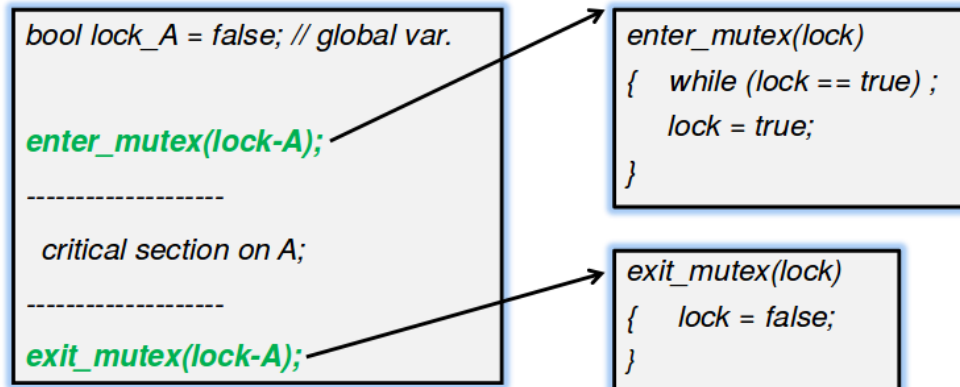
2. (tie-breaking rule) 여러 개의 프로세스가 기다리고 있을 때 어떤 프로세스가 먼저 진입하는지 결정하는 알고리즘이 있어야 한다.

3. Bounded Waiting (indefinite postponement) : 위에서 말한 tie-breaking rule 은 최대한 fair 해야 한다.
어떤 프로세스도 새로운 이벤트에 의해서 깨워질 때까지 무작정 길게 대기하는 일이 있어서는 안 된다.
4. Performance : 오버헤드가 너무 커도 안 된다.

Mutex 매커니즘을 구현하는 여러 가지 방법

- Lock / Unlock
- Semaphore
- Monitor : java “synchronized”
- Messages : 동기화를 기반으로 하는 단순한 데이터 전송 모델, distributed system 에서 적용가능

(((Wrong Version 1)))

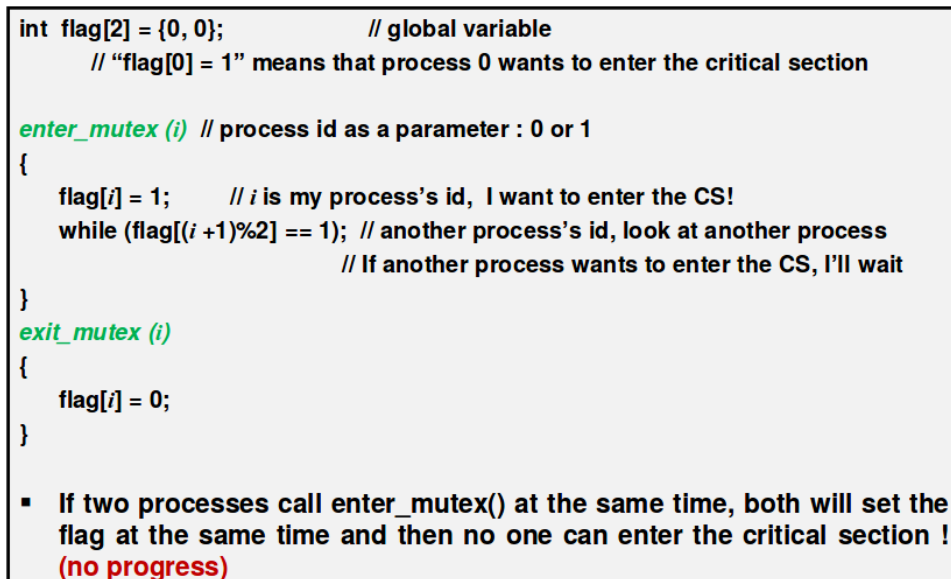


- If two processes call `enter_mutex()` at the same time, the two processes test the lock value at the same time. In this case, both processes get the false values and then both processes enter their critical sections together! **(no mutual exclusion)**
- **`enter_mutex()` function itself is critical !**

위와 같은 방식은 두 개 이상의 프로세스가 동시에 `enter_mutex()` 를 호출해서 동시에 lock-value 를 테스트할 경우 둘 이상의 프로세스가 동시에 critical section 에 접근할 수도 있다.

`enter_mutex()` 함수 자체가 다시 critical 하다

(((Wrong Version 2)))



설명하자면 flag 를 두 개 설정하고 둘 중에 하나의 flag 에 대해서만 진입을 허용하는 방법이다.

enter_mutex(int i) 함수 안에서

flag[i] = 1 즉, 프로세스 자신의 flag 를 1 로 바꾼 뒤(critical section 진입하겠다는 표시)

다른 flag 가 1 이면 이미 다른 프로세스가 사용중이므로 =0 될 때까지 wait

아니면 곧바로 진입.

exit_mutex(int I) 함수에서

flag[i] = 0 즉, 프로세스 자신의 flag 를 0 으로 바꾼 뒤(critical section 에 다른 프로세스가 진입해도 된다는 표시) 나온다.

이 방식의 문제점은 이번에는 둘 이상의 프로세스가 동시에 enter_mutex() 에 진입해서 각자의 flag 를 동시에 =1 로 만들면 어떤 프로세스도 critical section 에 진입하지 못하고 무한정 기다리게 된다는 문제점이 있다.

(((Wrong Version 3)))

```
int turn = 0;           // global var.
                        // "turn = i" means that it's process i's turn to enter the CS

enter_mutex (i)         // process id as a parameter : 0 or 1
{
    while (turn != i);   // i is my process's id, test if it is my turn
}

exit_mutex (i)
{
    turn = (i + 1)%2;    // set the turn for another process.
}

▪ In this case, process 0 and 1 enter their critical sections one by one.
▪ Even if a process wants to enter its critical section more frequently, it's not possible. (no bounded waiting)
```

turn 이라는 변수를 두고 0 으로 초기화한다.

enter_mutex(int I) 함수 안에서

turn 이 i 값인지 확인해서 자신의 차례일 때만 critical section 에 진입하기

exit_mutex(int I) 함수 안에서

turn 을 상대방의 차례로 돌려놓기

위 방식의 문제점은 다른 사람들이 전부 다 쓸 때까지 기다리고 난 뒤에만 내 차례가 돌아오기 때문에 No Bounded Waiting (= definite postponement) 이 되어버린다.

(((올바른 방식의 구현 : Peterson's Algorithm)))

```
int flag[2] = {0,0}; // global var. to declare that a process wants to enter the CS
int turn = 0;        // global var. for tie-breaking

enter_mutex (i)
{
    flag[i] = 1;      // i is process id
    turn = (i + 1)%2;
    while (flag[(i + 1)%2] == 1 && turn == (i + 1)%2) ; // tie-breaking rule
}

exit_mutex (i)
{
    flag[i] = 0;
}

▪ Variable "turn" is only used for tie-breaking! (to choose one from several waiting processes)
```

flag 와 turn 을 동시에 사용하기

→ enter_mutex () 에서

자신의 flag 를 1 로 만들고 나서 turn 을 상대방에게 진입을 허용한다.

진입을 종료하기 전에 tie-breaking rule 을 확인한다.

이것은 대체 무엇일까???

조건 1) 다른 프로세스 flag 가 1 이면 내가 진입할 수 없는 것이므로 wait 한다.

조건 2) 다른 프로세스 flag 가 0 이더라도 나 말고 바로 다른 프로세스 차례이면 wait 한다.

(why??? 상대 프로세스 쪽에서 자신 프로세스의 차례라고 설정해줄 것이므로 그때서야 내가 critical section 에 진입할 수 있게 된다)

→ exit_mutex () 에서는 자신의 flag 만 0 으로 만들고 나서 종료

이 알고리즘에서 turn 은 오직 tie-breaking 을 위해서만 사용된다.

(((올바른 방식의 구현 : Lamport's bakery algorithm)))

critical section 에 진입을 시도하는 모든 프로세스에게 번호표를 부여한다. 번호표는 항상 오름차순으로 부여된다.

진입을 시도하는 프로세스가 가진 번호표의 번호가 동일할 경우(concurrent programming 상황에서는 동시에 번호표를 받아갈 수도 있다) process-id 가 작은 것부터 차례대로 진입한다.

그러나 H/W Solution 이 훨씬 더 튼튼하다.

instruction : Test-and-Set !!! (spinlock, SMP machine 에서 많이 사용)

위에서 말한 turn 변수를 읽어와서 변화시키는 것을 atomic machine instruction 으로 바꿨다. 구현은

```
while ( test-and-set( &lock ) )
```

```
{ ; }
```

```
/*
```

```
* critical section
```

```
*/
```

```
lock = false;
```

와 같이 된다. 만약 test-and-set 을 S/W 로 구현했다면 [Wrong Answer 1] 처럼 문제가 발생하지만 하드웨어 명령어는 중간에 context-switch 가 일어나지 않는다.

→ 여기다가 bounded waiting 속성을 추가할 수도 있다. (슬라이드 p.30)

Instruction : Swap !!! (test-and-set 의 변종)

```
key = true;
do
    swap ( &lock, &key )
while ( key );
/*
 * critical section
 */
lock = false;
```

SPINLOCK 의 문제점

- Horribly Wasteful of CPU cycle
- 이를 구현하는 것은 지나치게 단순한 해결책을 제시하는 것
- 그래도 SMP 아키텍처에서는 사용하긴 함

instruction : interrupt_disable() & interrupt_enable() 활용하기

critical section 에 해당하는 코드를 실행하고 있을 때는 어떤 인터럽트도 거부한다. 당연히 context switch 는 발생하지 않는다.

이것은 privileged instructions 이므로 user-code 에서 사용할 수 없다.

커널 속의 critical section 을 보호하기 위해서 사용된다.

예시:

어떤 프로세스에 대한 fork() 가 발생해서 child 프로세스를 scheduling queue 에 집어넣고 있는데 time_slice_left == 0 이라서 clock interrupt 가 발생했다고 가정해보자. 그러면 시스템 전체에 문제가 생긴다. 만약 SMP 시스템에서라면 spinlock + interrupt_disable() 을 함께 사용해서 각각의 CPU 가 제각각 lock – unlock 되도록 한다. If one CPU is in critical section, all other CPUs wait in a busy-waiting loop(such as spinlock) before entering its kernel's critical section.

Semaphore

위에서 언급한 모든 S/W 알고리즘과 Test-and-Set H/W 명령어는 전부 busy-waiting 방식이다. time-slice wasted

1960 년대에 Dijkstra 라는 천재적인 컴퓨터 엔지니어가 고안해낸 방안

- block / wakeup 방식으로 작동한다. 당연히 blocked 된 프로세스는 cpu 를 점유하지 않는다.
- critical section 에 진입한 프로세스가 exit 하는 순간에 waiting process 를 깨워서 ready 상태로 만든다.
- waiting queue 가 존재한다.

```

Semaphore::wait( )
// “P” operation
{
    value--;
    if (value < 0)
        block the calling process;
        add it to the wait queue of this semaphore;
}

```

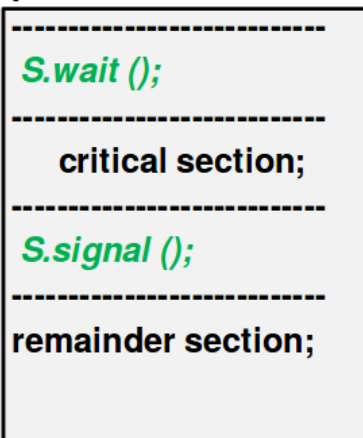
```

Semaphore::signal( )
// “V” operation
{
    value++;
    if (value <= 0)
        Remove the first process from the wait queue;
        Add it to the scheduling queue;
}

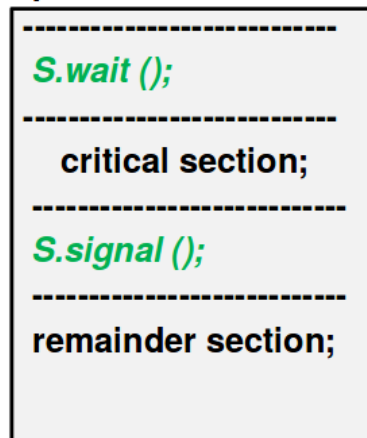
```

Semaphore S(1); // global var. for mutex, init with 1

process/thread A



process/thread B



P operation 과 V operation 은 서로 Semaphore::value 변수를 공유하고 있다. critical section problem again. 그래서 semaphore 는 커널 속에 구현되어 있어서 사용자에게 제공되는 것이지 이를 직접적으로 user-level code 에서 구현해서 사용할 수는 없다. (interrupt_disable() 사용)

Semaphore Printer(3);

~

Printer.wait();

~ use a printer ~

Printer.signal();

~

Synchronization with Semaphore

Semaphore Sync(0); // global var. with the initial value of 0
// One way synchronization

```

P0:
:
Sync.signal ();
:

P1:
:
Sync.wait ();
:
    
```

case 1) P₁ 이 먼저 .wait () 위치에 도달하면 P₀ 가 .signal () 에 도달할 때까지 P₁ 은 blocked.

P₀ 이 호출한 .wait () 에 의해서 Sync.value < 0 이 되었다.

P₁ 이 .signal () 호출해야 다시 Sync.value <= 0 이 되어서 P₁ 실행이 계속된다.

Case 2) P₀ 이 먼저 .signal () 위치에 도달하면 P₀ 는 중단없이 계속해서 진행한다.

P₁ 이 .wait () 위치에 도달해도 Sync.value > 0 이므로 별다른 처리없이 실행이 계속된다.

pthread_cond_wait (), pthread_cond_signal () 을 사용해서 위와 똑같은 프로그램을 구성한다고 가정하면 P₀ 이 pthread_signal () 위치에 먼저 도달했을 때 P₁ 에서 pthread_wait () 가 호출된 적이 없어서 pending signal 이 없다면 P₀ 이 P₁ 에게 보낸 시그널은 무시되어 버린다.

그러나 Semaphore 를 이용해서 구현하면 순서에 상관없이 동기화가 제대로 이루어진다.

Producer – Consumer Relationship (= Bounded Buffer Problem)

producer : 끊임없이 데이터를 만들어서 내보낸다.

consumer : 끊임없이 데이터를 끌어와서 소모한다.

만약 producer, consumer 를 concurrently 실행한다면 속도를 대폭 향상할 수 있다.

그러나 P 와 C 사이에 속도차이가 발생할 수 있으므로 동기화를 위해서 buffer 를 만들어야 한다.

이 때, 버퍼는 producer 와 consumer 사이에 공유되는 자원이므로 Mutual Exclusion 으로 관리해야 한다.

더 깊이 생각해보자면

- 1) 버퍼가 가득 찼을 때 producer 가 기다리게 하는 기능 (언제까지? consumer 가 데이터를 하나라도 가져갈 때까지)
- 2) 버퍼가 비었을 때 consumer 가 기다리게 하는 기능 (until when? producer 가 데이터를 하나라도 만들어낼 때까지)

Producer


```

while (1) {
    ...
    produce an item in pdata;
    ...
    while (count == n) ;
    // wait for an empty buffer
    buffer[rear] = pdata;
    rear = (rear + 1) % n ;
    counter++;
}
    
```

Consumer

```

while (1) {
    while (counter == 0) ;
    // wait for an item
    cdata = buffer[front] ;
    front = (front + 1) % n ;
    count--;
    ...
    process the item in cdata;
    ...
}
    
```

 Concurrent processing is possible!

 **Critical sections**

```

// global variables
Semaphore mutex(1), num_buffer(n), num_data(0);
int rear = front = -1; count = 0;

while (1) // producer
{
    produce an item;
    mutex.wait(); // unnecessary
    num_buffer.wait (); // for sync
    buffer[rear] = pdata;
    rear = (rear + 1) % n ;
    num_data.signal(); // for sync
    mutex.signal(); // unnecessary
}

while(1) // consumer
{
    mutex.wait(); // unnecessary
    num_data.wait(); // for sync
    cdata = buffer[front] ;
    front = (front + 1) % n ;
    num_buffer.signal (); // for sync
    mutex.signal(); // unnecessary
    process an item;
}

```

1. “**Count**” variable is unnecessary because we use a counting semaphores !
Counting semaphore **num_data** itself has the number of data.
2. So **mutex.wait()** and **mutex.signal()** is unnecessary !

Threads

Processes (heavy weight) vs Threads (light weight) : 프로세스, 스레드 전부 다 스케줄링 대상
하나의 프로세스 안에 여러 개의 스레드. 스레드는 함수 단위로 분할.

All threads of a process 는 서로 text, heap, data 메모리를 공유한다.

All threads of a process 는 서로 stack 을 공유는 하지만 각자 고유의 stack pointer 를 갖는다.

새로운 스레드가 생성되고 나면 process 는 여러 스레드가 실행되는 컨테이너처럼 기능.

스레드 사이에는 자원을 공유하기 때문에

하나의 스레드에서 변화시킨 결과는 다른 스레드에도 동일하게 반영된다.

IEEE POSIX thread (-lpthreads) 컴파일 옵션 (win32 threads 도 있다)

pthread_create

(pthread_t* thread, const pthread_attr_t* attr, void* (*start_routine), void* arg) → int

→ 한 프로세스 내에서 만들어진 스레드 사이에는 siblings 관계

→ attr (detachable, joinable)

→ start_routine : 스레드가 실행해야 하는 함수의 주소값

→ return value : 스레드 ID

pthread_exit (void* status) → void

→ 스레드가 종료되는 방법

::: 함수 실행 완료

::: 해당 스레드가 pthread_exit() 을 호출 (이때 자원 관리는 프로그래머가 손수 해줘야 함)

::: 다른 스레드가 pthread_cancel() 을 호출

::: 전체 프로세스를 종료하는 exec() 또는 exit() 함수를 호출

pthread_self(void) → pthread_t

→ 스레드 ID 를 알고 싶을 때 사용

* Join : 스레드 사이의 동기화 방법 중 하나

pthread_join (pthread_t thread_id, void **status) → int

→ 호출 시 thread_id 를 가진 스레드가 종료할 때까지 이 함수를 호출한 스레드가 blocked 상태이다.

→ pthread_exit() 의 수행결과와 void* 타입으로 선언된 변수에 기록한다.

→ 한번 detach 한 스레드를 다시 join 할 수는 없다.

pthread_mutex_init

(pthread_mutex_t* mutex, const pthread_mutexattr_t* attr) → int

pthread_mutex_destroy

(pthread_mutex_t* mutex) → int

Mutex 변수를 선언하는 방법 2 가지

정적 선언 : pthread_mutex_t X = THREAD_MUTEX_INITIALIZER;

런타임 동적 선언 : pthread_mutex_init (&X, NULL);

mutex 는 초기화하자마자 unlock 상태 유지

pthread_mutex_lock(pthread_mutex_t* mutex) → int

pthread_mutex_trylock (pthread_mutex_t* mutex) → int

pthread_mutex_unlock (pthread_mutex_t* mutex) → int

→ 만약 해당 mutex 가 이미 unlock 상태이면 ERROR

→ 만약 mutex 가 이 함수를 호출한 스레드가 아닌 다른 스레드에 의해서 점유되었으면 ERROR

Condition Variable

→ 스레드 사이의 동기화를 위해서 사용

어떤 조건 하에서 blocked 되고

일정한 조건에 따라 wakeup 된다

일반적으로 이 mutex 프로토콜에는 condition variable 을 사용한다.

호출 함수들

pthread_cond_init (pthread_cond_t* cond, pthread_condattr_t* attr) → int

정적 선언 : pthread_cond_t X = PTHREAD_COND_INITIALIZER

동적 선언 : pthread_cond_init (&X, &attributes)

pthread_cond_destroy(pthread_cond_t* cond) → int

pthread_cond_wait (pthread_cond_t * cond, pthread_mutex_t* mutex) → int

→ condition variable 에 signal 이 도달할 때까지 wait (= blocked) 상태가 된다.

→ 블록되는 순간에 mutex 를 unlock 한다. 다른 스레드에 시그널을 보내서 알린다.

→ On Resume, mutex 를 다시 lock 한다.

pthread_cond_signal (pthread_cond_t* cond) → int

→ 시그널은 쌓이지 않는다. (반면에 semaphore's signal 은 쌓인다)

pthread_cond_broadcast (pthread_cond_t* cond) → int

```

void producer (void)
{
    int i;
    for (i=0; i < 1000; i++) {
        pthread_mutex_lock(&mutex);
        if (count == 100) // buffer full !
            pthread_cond_wait (&buffer_has_space,
                               &mutex);
        in++; in %= 100;
        buffer[in] = i;
        count++;
        pthread_cond_signal(&buffer_has_data);

        pthread_mutex_unlock(&mutex);
    }
}

```

```

void consumer (void)
{
    int i,data;
    for (i=0; i < 1000; i++) {
        pthread_mutex_lock(&mutex);
        if (count == 0) // buffer empty !
            pthread_cond_wait(&buffer_has_data
                               &mutex);
        out++; out %= 100;
        data = buffer[out];
        count-;
        pthread_cond_signal(&buffer_has_space);

        pthread_mutex_unlock(&mutex);
        printf("data = %d\n",data);
    }
}

```

* pthread_cond_wait() 호출은 항상 mutex_lock() 상태에서 이루어져야 한다.

Manager/Worker Model... 하나의 매니저 스레드가 여러 워커 스레드에 일감을 분담하는 형식

Pipeline Model... suboprations 로 이루어짐. 속도 향상

Peer Model... 하나의 스레드가 다른 스레드를 모두 만드는건 맞지만 메인 스레드가 스레드 생성을 완료하고 나면 전체 스레드가 작업에 동참함

POSIX Semaphore --- #include <semaphore.h>

sem_init (sem_t* sem, int pshared, unsigned int value) → int

→ 세마포어 객체를 초기화

→ value 로 세마포어 카운트

→ pshared == 0 이면 multi-thread programming, pthread > 0 이면 multi-processing

sem_wait(sem_t* sem) → int

sem_trywait(sem_t* sem) → int

→ “P” operation

sem_post(sem_t* sem) → int

→ “V” operation

sem_getvalue(sem_t* sem, int* sval) → int

→ sval 에다가 세마포어 변수값을 저장

sem_destory(sem_t* sem) → int

→ 세마포어 객체가 점유하고 있는 리소스를 전부 반환하면서 그 객체를 제거

→ 해당 세마포어 객체에 대해서 어떤 스레드도 wait 상태가 아니어야 한다.

→ 사실 리눅스에서 세마포어 객체에 특별한 리소스를 할당할 수 없으므로 blocked 된 스레드가 있지만 확인한다.

Shared Memory between Processes

원래 프로세스들 사이에는 메모리를 공유하지 않는다. 설령 parent-child 관계라고 할지라도 text 메모리만 공유하기 때문에 모든 프로세스가 접근할 수 있는 shared memory obj 를 생성하고(일종의 파일) 여기에 데이터를 기록하므로서 IPC 를 할 수 있다.

이 과정은 kernel 에 의하지 않고, user-space copy 이므로 속도 굉장히 빠름

반대로 [pipe, message queue] 에서는

읽기 연산 시에는 user to kernel memory copy

쓰기 연산 시에는 kernel to user memory copy 가 일어나기 때문에 속도가 다소 늦다.

대신에 shared memory 에는 Mutex 철저히 설정해줘야 한다. POSIX Semaphore 를 주로 사용.

POSIX Shared Memory

/tmpfs 파일 시스템에 기록 (정확한 path 로 말하자면 /dev/shm)

컴파일 옵션 (-lrt)

shm_open()

→ 만약 기존에 공유메모리가 있으면 그것을 open

→ 없으면 새로운 공유메모리 객체를 만들거 이를 open

→ return value 는 파일디스크립터

ftruncate() : 공유메모리 크기 지정

mmap() : 공유메모리 객체를 이를 다루려는 프로세스의 로컬 메모리 공간에 매핑한다.

munmap() : 매핑을 취소한다.

close() : 파일디스크립터를 close 한다

shm_unlink() : remove SHM object by name(type: const char*)

fstat() : 공유메모리 객체에 대한 정보 반환

공유메모리 객체에 대한 접근은 POSIX unnamed semaphore 를 사용하는 것이 편리하다.

세마포어 객체를 shared memory region 에 위치시켜서 프로세스 간에 공유하면서 동기화할 수 있고 세마포어에 반드시 이름을 지정하지 않아도 되기 때문이다.

그 외에 Synchronization Tools

Unix/Linux

- signal
- pipe
- file lock/unlock

Pthread Library

- POSIX Semaphore
- mutex_lock
- condition variable

Hoares's Monitor

- a multi-thread safe object (java's synchronized object)

Concurrent-Readers / Exclusive-Writers Problem

공유 리소스에 대해서 Read 만 하는 프로세스와 Write 만 하는 프로세스가 있다고 할 때, 공유 자원에 하나의 프로세스 밖에 접근하지 못한다고 하면 성능이 좋지 않다.

Write 할 때에 배타적으로 접근해야 하는건 당연하지만

Read 할 때에 wrtie 와 겹치지만 않고 다른 Read 와 동시에 공유 데이터에 접근해도 아무 상관이 없다. 특히 DB에서는 80%의 쿼리가 읽기 동작이므로 이를 통해서 성능을 상당히 향상시킬 수 있다.

그러므로 reader 와 writer 에 대해 서로 다른 lock 을 걸어야 한다.

1) Writer's Lock (= Exclusive Lock)

- mutex for writers
- same as the original mutex
- when writing, any other writier & reader cannot access shared resource

2) Reader's Lock (= Shared Lock)

- a special mutex for readers
- when reading, another reader can access the resource, but no writer can access it.

```
#include <fcntl.h>
```

```
fcntl ( int file_descriptor, int command, struct file_lock* lock ) → int
```

command :

F_GETLK : 현재 lock 상태를 반환

F_SETLK : lock 을 시도하고 이미 locked 상태이면 error

F_SETLKW : lock 을 시도하고 이미 locked 상태이면 wait...blocked... lock 을 걸 수 있는 상태가 되면 프로세스는 깨어난다.

```
Struct flock {
```

```
    short l_type;
```

```
        F_RDLCK : reader's lock
```

```
        F_WRLCK : writer's lock
```

```
        F_UNLCK : unlock
```

```
}
```

Starvation

불공정하고 불확정적으로 리소스를 무한정 기다려야 하는 상황을 의미한다.

위에서 말한 reader - writer 문제를 가지로 말하자면 reader 가 계속해서 진입하다보니 아주 이른 시기에 진입을 시도하려고 기다리기 시작한 writer 가 결코 shared resource 를 획득하지 못하는 문제점이 발생한다.

가장 간단한 해결책은 Aging 을 도입하는 것이다.

일정 수준 이상으로 대기 시간이 길어진 스레드에게 공유 자원에 대한 우선순위를 높게 설정한다.

만약 writer 가 대기중이라면 더 이상 reader 가 진입하지 못한다. 현재 공유자원을 사용중인 마지막 reader 가 unlock 하는 순간 writer 가 lock 을 건다.

POSIX Read-Write Locks

```
pthread_rwlock_rdlock, pthread_rwlock_rdunlock
```

→ 공유 자원에 write-lock 이 걸려있지 않고 blocked writer 가 하나도 없을 때만 호출하는 스레드가 read-lock 을 획득할 수 있다.

```
pthread_rwlock_wrlock, pthread_rwlock_wrunlock
```

→ 공유 자원에 read-lock 이든 write-lock 이든 아무것도 걸려있지 않을 때만 호출하는 스레드가 write-lock 을 획득할 수 있다.

Hoare's Monitor

모니터(Monitor) 라는 개념은 객체 지향적인 synchronization tool 이다

모든 mutex 매커니즘이 하나의 객체에 포함되어 있는 것이다. (multi-thread safe class)

여러 개의 스레드와 프로세스에 의해서 접근되더라도 객체의 안정성이 유지된다는 것을 보장하는 것이 모니터.

e.g. Java's Synchronized Object

하나의 모니터 안에서 최대 하나의 프로세스 / 스레드가 실행될 수 있다.
 추가적인 프로세스/스레드는 waiting queue 에 집어넣는다.

Deadlock

하나 또는 둘 이상의 프로세스가 결코 일어날 수 없는 이벤트를 영원히 기다리고 있는 concurrent program 의 상태를 의미한다.

더 이상 프로그램이 진전되지 않는다.

P_1	P_2
<code>mutex_lock(A); // a₁</code>	<code>mutex_lock(B); // b₁</code>
<code>mutex_lock(B); // a₂</code>	<code>mutex_lock(A); // b₂</code>
:	:
<code>mutex_unlock(B);</code>	<code>mutex_unlock(A);</code>
<code>mutex_unlock(A);</code>	<code>mutex_unlock(B);</code>

mutex 활용에 의한 데드락 발생 가능성은
 프로그래머가 책임지고 방지해야 한다.

P_1	P_2
<code>file_lock(A); // a₁</code>	<code>file_lock (B); // b₁</code>
<code>file_lock(B); // a₂</code>	<code>file_lock (A); // b₂</code>
:	:
<code>file_unlock(B);</code>	<code>file_unlock(A);</code>
<code>file_unlock(A);</code>	<code>file_unlock(B);</code>

file 은 커널의 자원이다.

이런 상황의 발생을 막기 위해서 kernel 속에 데드락 방지 기능이 존재해야 한다.

대표적인 예시 : Dining Philosopher's Problem

반대로 LiveLock 이라는 것도 발생할 수 있다. 반복문 안에서 lock-unlock 을 끝없이 반복하는 것이다.

데드락이 발생할 4 가지 필요조건

1. Mutual Exclusion

모든 리소스는 다른 어떤 프로세스에 대해서도 배타적으로 점유되어야 한다.

2. Hold and Wait

어떤 프로세스가 하나의 리소스를 점유한 상태에서 다른 리소스에 대한 요청을 기다리며 blocked 된 상태

3. Non-Preemption

하나의 프로세스에 할당되어서 lock 자원은 그것이 unlock 전까지는 다른 프로세스에 재할당될 수 없다.

4. Circular Wait

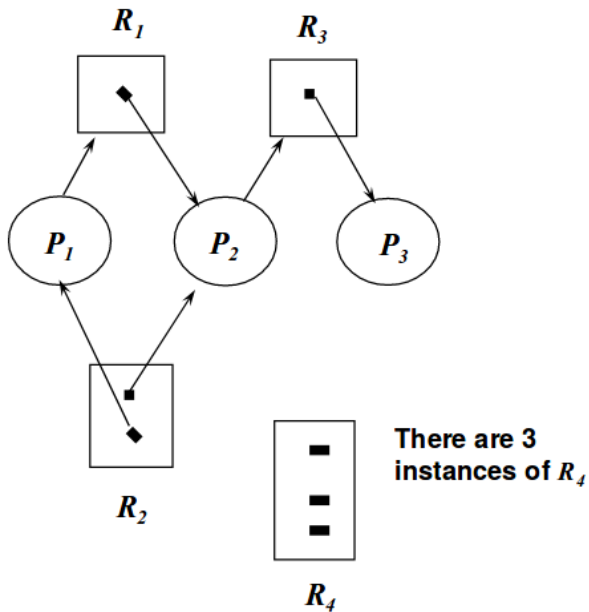
A Set of Blocked Process { P₀, P₁, P₂, P₃ }

P₀ < P₁ < P₂ < P₃ < P₀

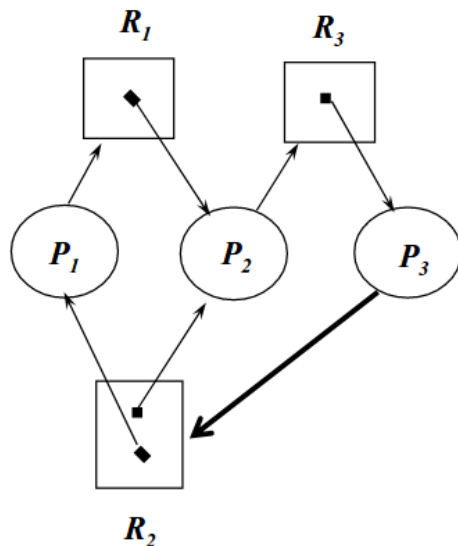
위의 4 가지 조건은 필요조건이지 충분조건이 아니다. 위의 4 가지 조건을 모두 갖추고도 데드락이 발생하지 않을 수 있다. 그러나 데드락이 발생하려면 위의 4 가지 조건이 반드시 갖추고 있어야 한다.

그러므로 데드락을 방지하려면 위 중에서 최소한 하나라도 제거하면 된다.

Resource Allocated Graph

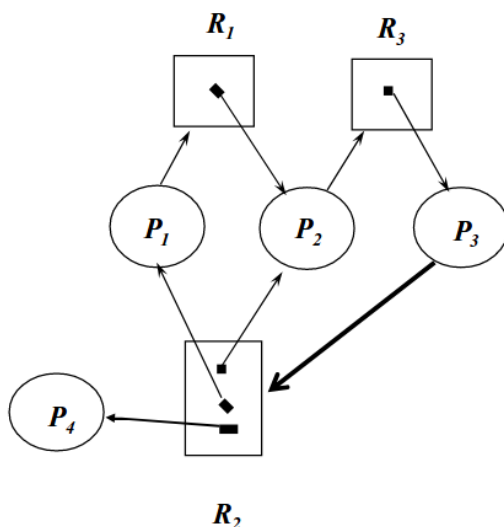


No **circular wait** exists.
So **currently**, there's no deadlock yet.



A **circular wait** exists.
And no way to finish P_1 , P_2 and P_3 .

There's a deadlock !



A **circular wait** exists.
But there is a possible scenario to finish P_1 , P_2 and P_3 .

1. P_4 finishes, release R_2 ;
2. R_2 is allocated to P_3 ;
3. P_3 finishes and releases R_2 , R_3 ;
4. P_2 finishes, releases R_3 , R_1 ;
5. P_1 finishes;

Not a **deadlock** currently!

데드락을 다루는 방법에는 크게 세 가지 접근법이 있다.

- Prevention : 데드락의 4 가지 필요조건 중에서 최소한 한 가지를 시스템에서 발생하지 않도록 만드는 것이다.
즉, 이것은 프로그래머의 역량이자 역할
- Avoidance : 만약 resource-request 가 시스템을 불안정하게 만든다면(데드락 걸릴 수 있는 경우) 그 태스크를 거부한다.
- Detection & Recovery :
커널은 데드락 방지, 회피를 위한 어떠한 노력도 하지 않고 무조건 모든 태스크를 실행한다.
간헐적으로 resource allocation graph 를 살펴며 데드락이 존재하는지 확인한다.
데드락이 발생하고 나면 데드락과 관련된 모든 프로세스를 강제종료 시킨다.

Deadlock Prevention

- “Mutual Exclusion” 조건을 제거하는 것은 불가능하다. 각 프로세스가 반드시 배타적으로 접근해야 하는 리소스가 있다. (e.g. 프린터, 파일 write – lock)
- “Non-preemption” 조건을 제거하는 것을 불가능하다. I/O 는 타임슬라이스에 상관없이 처음부터 끝까지 실행되어야 하기 때문이다.

* “Hold and Wait” 조건 제거하기

하나의 프로세스는 하나의 리소스만을 요청하도록 만든다. 그러므로 모든 리소스는 All or Nothing 방식으로 할당된다.

문제점 : 리소스를 실제로 사용하지 않는 순간에도 프로세스가 점유하고 있으므로 인해서 다른 프로세스에 starvation 이 발생하거나 시스템 전체의 성능이 저하된다.

* “Circular Wait” 조건 제거하기

동일한 타입의 리소스의 여러 오브젝트에 대해서 일정한 id number 를 할당한다. 일정한 정렬 순서에 따라서만 추가적인 리소스 할당이 가능하도록 만든다.

Resource Pool : { A=1, B=2, C=3 }

Lock(A) __ Lock(B) : OK

Lock(C) __ Lock(A) : reject or avoided

문제점 : hold & wait 조건을 제거하는 것보다는 성능이 좋아지지만 device utilization 이 낮아진다.

(이유는??? 순서를 고정시켰기 때문에)

일반적으로 Kernel 은 데드락을 방지하기 위해서 어떠한 노력도 기울이지 않는다.

그러므로 항상 프로그램 작성자가 데드락에 주의하며 구현해야만 한다.

Deadlock Avoidance 기법 - Dijkstra’s Banker’s Algorithm

프로세스의 최대 요청량, 시스템 내 자원 가용량을 전부 알고 있을 때에만 사용이 가능한 방안

항상 safe state 를 유지한다.

Safe State : 자원을 계속해서 할당하고 반환하면서 모든 프로세스를 완료할 수 있는 방안이 존재하는 실행 시퀀스.

Unsafe state :

	<i>max requests</i>	<i>current alloc.</i>	<i>left: 3</i>	safe state
p_0	10	5		
p_1	4	2		
p_2	9	2		

- p_2 requested one more printers. (accept or reject?)

	max requests	current alloc.	left: 2
p_0	10	5	
p_1	4	2	
p_2	9	3	

- p_2 got 1 more printer, then left = 1 .
- No process can finish if processes request their maximums.
- So reject the request of p_2
- But unsafe state does not always cause a deadlock.
- A deadlock only can occur when processes requests their maximums.

일반적으로 Banker's Algorithm 은 오버헤드가 너무 크기 때문에 실현하기 어렵다.
논문 페이퍼를 위한 알고리즘.

Deadlock Detection & Recovery

먼저 Detection

- single instanced resource type : resource allocation graph 에서 cycle 이 있는지만 확인하면 됨
- multiple instanced resource : allocation vector / request vector 를 만들어서 safety algorithm 실행
- 예시 :

```
m : int // Processes 개수
n : int // Resources 개수
available [ 1...m ] // 어떤 타입의 resource 를 사용할 수 있는지
request [ 1...n, 1...m ] // i 번 프로세스가 리소스 j 를 요청하고 있음
allocation [ 1...n, 1...m ] // i 번 프로세스가 리소스 j 를 점유하고 있음
finish [ 1...n ] // i 번 프로세스에 할당될 수 있으면 true 로 바꿈
```

우선 allocation 벡터에서 i 번 프로세스 행이 전부 0 이면 finish[i] = true

데드락 체크 :

finish[i] == false && request[i][] <= work 인 것을 찾는다.

만약 어떤 프로세스도 조건에 맞지 않으면 "Deadlock"

계속해서 모든 allocation 에 대해서 work = work + allocation[i][] 하면서 데드락 체크

해당 i 번 프로세스에 대해서 finish[i] == true 가 되면

i+1 번 프로세스에 대해서 데드락 체크하기 시작 (go to 데드락 체크)

중간에 하나라도 finish[i] == false 로 끝나는 프로세스가 있으면 "Deadlock"

다음으로 Recovery

두 가지 방안이 있는데, deadlock 이 사라질 때까지 데드락과 관련이 있는 프로세스를 하나씩 강제종료 해보거나 이미 할당된 리소스의 우선점유권을 다른곳으로 넘기는 것이다.

그러나 이런 과정을 실제로 구현한다고 가정하면 너무나 많은 오버헤드가 요구되므로 잘 시행하지 않음.

Signal Processing (Asynchronous Event Processing)

Signal : user-process 는 인터럽트와 같은 비동기적인 이벤트를 받거나 보내서 처리하기 위한 매커니즘으로 signal & signal-handlers 를 사용한다.

보통 시그널은 kernel → user 또는 user → user 로 송수신

- User's View

::: user-mode 실행중 → signal 발생 → user program 실행을 잠시 중지 → signal-Handler 코드 실행 시작 (still user-mode running) → 시그널처리 완료 후 다시 user program 로 돌아가기 (S/W IRQ Handling 과 구조적으로 비슷)

- Kernel's View

::: 어떤 이벤트 발생 (e.g. segment fault, a special interrupt)

→ 관련된 프로세스의 PCB 에 어떤 시그널인지 표시(mark the signal)

→ user-mode 로 되돌아가는 바로 그 순간에, 바로 직전에 signal handler 가 동작해서 그 시그널에 맞는 동작을 수행한다

→ user-program 을 이어서 실행한다.

(interrupt-handling 은 어떠한 딜레이도 없지만 signal handling 과정에서는 약간의 딜레이가 발생할 수 있다)

시그널이 도달하면 PCB 속의 pending signal table 에 표시한다. (mask set 방식으로)

도달했지만 아직 처리되지 않은 시그널은 pending signal 이라고 한다.

signal-handler 는 해당 프로세스의 User-Mode 로 돌아오기 직전에 실행되는 것이므로 kernel-mode 에서 실행하는 태스크는 전부 다 실행한다.

시그널 처리 방식은 사용자에게 의해서 얼마든지 재조정될 수 있다. (signal bocking)

3 Ways of Signal Handling

1> 커널이 제공하는 원래대로 처리 (SIG_DFL)

2> 무시하기 (SIG_IGN)

3> 사용자가 정의하는 시그널 핸들러 사용하기

예를 들어 ^C (= ctrl + C) 는 보통 프로세스를 종료하는 명령어이지만 프로그래머가 다르게 동작하도록 바꿔놓을 수 있다.

시그널 이름	발생하는 이유	value	Default Handler
SIGABRT	program abort	6	Core-dump & Exit
SIGALARM	timer alarm	14	Exit
SIGCHILD	death of a child	18	Ignore
SIGCONT	continue a stopped process	25	Restart / Ignore
SIGILL	Not an instruction (all illegal jump to a data section)	4	Core-dump & Exit
SIGINT	^C	2	Exit
SIGKILL	kill request (kill -9 pid)	9	Exit
SIGPIPE	write attempt to pipe with no reader (broken pipe)	13	Exit
SIGQUIT	^Z	3	Core-dump & Exit
SIGSEGV	Illegal memory access (pointer, access kernel's or other process's area, write on read-only area)	11	Core-dump & Exit

SIGSTOP	Stop (e.g. debugger)	23	Stop
SIGTERM	graceful kill request	15	Exit
SIGUSR1	user defined signal 1	16	Eixt
SIGUSR2	User defined signal 2	17	Exit

#include <signal.h> - 이 안에 typedef void (*sighandler_t)(int); 로 선언되어 있음

signal (int signum, sighandler_t handler) → sighandler

사용자 정의 시그널 핸들러를 세팅하는 함수

- signum : 몇 번 시그널 번호를 사용할지

- sighandler : 사용자가 정의하는 함수 이름을 여기에 넣는다

- return value : 성공하면 old-signal-handler 의 주소값, 실패하면 SIG_ERR

* 주의할 점 : OS 에 따라서 다르지만 한번 시그널처리를 끝내고 나면 새롭게 정의된 시그널 핸들러가 사라지고 SIG_DFL 가 되기도 한다. 어떤 OS 는 설정된 사용자 정의 시그널 핸들러를 계속 유지하기도 함.

kill (pid_t pid, int sig_num) → int

- pid : 시그널을 받게되는 process-id

(if pid > 0) { 해당 pid 프로세스에게 보내기 }

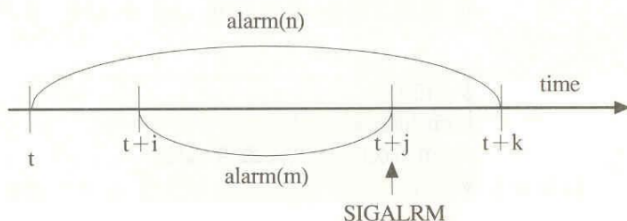
(if pid == 0) { 모든 child 프로세스에게 보내기 }

(if pid == -1) { broadcasting }

- sig_num : 어떤 시그널을 보내는지

- return value : 성공하면 0, 실패하면 -1

alarm() 함수의 문제점



alarm() 함수를 여러개 사용해서 제각각 SIGALRM 을 날릴 수가 없다.

sleep() 함수의 호출 이후에 깨어날 때도 SIGALRM 시그널을 사용하기 때문에 시스템을 제어하기가 어렵다.

sleep 함수는 parameter 로 초단위 숫자를 지정

시스템 호출 과정에서 일어나는 시그널 핸들링 절차 (Signal Handling during System Call)

type 1))) 커널 속에서 일어나는 system call 절차가 모두 끝난 뒤에 signal handling 이 시작.

- I/O 가 모두 완료된 뒤에

- TASK_UNINTERRUPTIBLE 상태 (즉, I/O 가 완료된다는 것이 확실히 보장됨)

type 2))) system call 을 중단하고 signal handling 을 한 뒤에 resum or error-return

- 프로세스는 blocked 에서 Ready 로 깨어난다(wake up)

- 시그널 처리가 끝난 뒤에

Linux default : I/O system call 함수를 다시 호출

UNIX : I/O system call 함수에서 error-return

두 가지 경우 모두 siginterrupt (signal_number, TRUE/FALSE) 함수를 통해서 동작하게 되는데,

true 이면 system call 에서 ERROR-Return 하면서 종료하고

false 이면 system call restart 한다.

Multi-Thread 환경에서 모든 스레드는 프로세스의 signal handler 를 그대로 물려받는다.

시그널은 ‘프로세스’에 대해서 도달하는 것이고, 그 프로세스에 여러 개의 스레드가 구동중이라면 any one thread of the sibling group 으로 가서 처리된다. (Not a Fixed One)

그러므로 확실하게 하나의 스레드에서 시그널이 처리되도록 하려면 다른 모든 스레드에서 signal block 하면 된다.

만약 모든 thread 에서 해당 signal block 되어 있으면 signal will be queued.

POSIX Timer (Advanced)

앞에서 언급한 SIGALRM 의 한계를 뛰어넘기 위해서 (SIGRTMIN ~ SIGRTMAX) 범위 중에서 하나의 시그널 번호에 하나의 시그널 핸들러를 대응시킨다.

Signal handler 로는 function or thread 둘 중에 하나를 선택하면 된다.

Overrun???

예를 들어 signal 이 주기적으로 발생하는 시스템이 있다고 가정하자. 그 시스템에서 1 번째 시그널을 처리하는 데 예정보다 길어지는 바람에 2 번째 시그널이 도달할 때까지도 1 번째 시그널 처리가 완료되지 못하면 2 번째 시그널을 무시되어 버린다. 이를 overrun 이라고 하면 POSIX Timer 는 이를 인식하고 사용자가 확인할 수 있는 함수 (timer_getoverrun(...)) 를 내장한다.

그러나 Portability 는 낮으므로 cross-compile 환경에서 사용하기에는 적절하지 않음.

Linux File System

File System

>> User's Viewpoint : directories and files (hierarchical file system), Logical I/O

>> Kernel's Viewpoint : data and metadata (flat file system),

여러가지 파일 시스템 종류

1. Disk (FAT, ext2/3/4, xfs, btrfs, ntfs, etc.)
2. Flash (f2fs)
3. Network (nfs, ...)
4. Memory (ramfs, ...)
5. Pseudo
6. Stackable
7. Object Store
8. FUSE (Filesystem in userspace) : 개발자가 직접 파일 시스템을 구현하고 이를 user space 에서만 사용하는 방법이다. 실제 속도는 느리지만 구현하는 것은 커널보다는 훨씬 쉽다.
9. isofs (CD-ROM)
- 10.

The Virtual File System (VFS)

- user code 는 굳이 file-system-aware 할 필요는 없다. 동일한 기능을 하는 코드를 중복해서 삽입 X
- a kernel layer for multiple applied layers
- user-mode 에서 하나의 기능에 대응되는 system call 함수를 호출하면 커널 안에서 적절한 파일시스템에 맞는 specified function 을 호출하는 방식으로 작동한다.
- 모든 파일 시스템에 공통으로 들어가는 부분도 커널이 처리한다. (e.g. S/W caching, readahead)
- filesystem-dependent functions (C 에서 object-oriented programming 스타일로 코드를 짜면 이런 모습이 된다.)

```
mount -t xfs /dev/sdb1 /home
```

mount : 파일시스템을 마운트 하기

-t : 파일시스템 타입

/dev/sdb1 : 마운트하려는 대상 스토리지

/home : 마운되는 위치 지정하기

→ 마운트 상태인 드라이브끼리는 cp 연산도 적용 가능