운영체제 (OS, Operating System)

1. 역할

1) 시스템 자원(System Resource) 관리
- 시스템 자원 = 컴퓨터 하드웨어(CPU, 메모리, 입출력장치, 저장장치)
 ㄴ CPU: 각 프로그램이 CPU를 얼마나 사용할지 결정
 ㄴ Memory: 각 프로그램이 어느 주소에 저장되며, 메모리 공간을 얼마나 확보할지 결정
 ㄴ HDD/SSD: 어디에, 어떻게, 어떤 구조로 저장하고 불러올지 결정 (파일명/파일 구조 등)
 ㄴ 키보드/마우스: 입력에 따라 동작 수행이나 커서 움직임 등을 표시
- 운영체제는 컴퓨터 하드웨어를 적절히 조작하고 결정을 내린다.

* OS 미설치 노트북은 어떻게 동작하는가?
 - 검은 화면에 OS가 설치된 저장매체를 요구하는 문구만을 띄우고 동작을 멈춘다.
 - 최신 메인보드는 BIOS나 UEFI 등을 통해 제조사가 몇 가지 기능을 넣어두기도 한다.

2) 사용자와 컴퓨터 간의 커뮤니케이션 지원
- 입력기기를 사용해서 컴퓨터에게 명령을 내리고, 출력기기로 결과를 확인할 수 있다.
- OS 없이는 사용자의 명령을 컴퓨터에 전달하거나, 컴퓨터의 작업 내용을 확인할 방법이 없다.

3) 응용 프로그램 제어
- 프로그램 = 소프트웨어 = 운영체제 + 응용 프로그램
- 운영체제는 하드웨어와 응용 프로그램을 관리한다.
- 응용 프로그램을 실행시키고 권한을 관리한다.
 ㄴ 잘못된 코드로 프로그램이 다운되거나 무한루프를 도는 현상 방지
 ㄴ 외부 파일을 수정 또는 삭제할 권한 관리
- 응용 프로그램이 요청하는 시스템 자원을 효율적으로 분배
 ㄴ 하드웨어 자원을 특정 프로그램이 과도하게 사용하는 등의 행위 방지
- 응용 프로그램을 사용하는 사용자도 관리한다. (로그인 등)

* 폰 노이만 구조
- 모든 프로그램은 메모리에 올라가고, 작업을 하나씩 CPU에 넘겨서 실행한다. 운영체제도 하나의 프로그램이므로, 저장장치에 저장되어 있다가 부팅 시 메모리에 올라간다.

2. 기능

1) 사용자 인터페이스 제공
- 셸(Shell): 사용자가 운영체제 기능과 서비스를 조작할 수 있도록 인터페이스를 제공하는 프로그램.

ㄴ 셸도 응용 프로그램의 하나에 속한다.
ㄴ 터미널 환경(CLI)과 GUI 환경의 두 종류로 분류된다.
ㄴ 리눅스의 bash, bsh, csh, 윈도우의 cmd, PowerShell(CLI), explorer(GUI) 등이 있다.

2) 응용 프로그램을 위한 인터페이스 제공
- 시스템 콜(System Call): 운영체제의 각 기능을 응용 프로그램에서 사용할 수 있도록 명령 또는 함수를 제공
ㄴ API도 내부적으로 시스템 콜을 호출하는 형태로 만들어져 있다.
ㄴ 예) POSIZ API, Win API 등
- API: 응용 프로그램과 상호작용할 수 있는 형식.
ㄴ 주로 함수 형태로 사용하며 라이브러리로 제공된다.
ㄴ 각 언어별 운영체제 기능을 호출하는 인터페이스 함수로 구성된다.
- 시스템 콜을 직접 호출하는 경우도 있으나, 보통은 시스템 콜을 사용해 만든 언어별 라이브러리(API)를 사용한다. 즉 API를 통해 시스템 콜을 호출한다.

3) CPU Protection Rings
- 응용 프로그램뿐 아니라 CPU도 권한 모드를 가지고 있다. CPU의 권한은 Ring 0 ~ Ring 3의 4단계로 분류된다. 보통은 Ring 0과 Ring 3만 사용된다.
ㄴ 사용자 모드(user mode, Ring 3)와 커널 모드(kernel mode, Ring 0)로 구분된다.
ㄴ '커널(kernel)'은 OS가 사용하며, 사용자 모드는 응용 프로그램이 사용한다.
- 커널 모드의 실행은 반드시 OS가 제공하는 시스템 콜을 거쳐야 한다.
ㄴ 시스템 콜을 호출하면 해당 명령은 커널 영역으로 들어가 하드웨어에 전달된다.
ㄴ 함부로 응용 프로그램이 컴퓨터 시스템을 해치지 못 하게 하기 위함
- 운영체제 기능이 필요한 시스템 콜 호출 시, 커널 모드로 전환하여 해당 작업을 수행한다.
ㄴ ex) 입출력 라이브러리를 통해 파일을 여는 open() 함수를 사용하면, 시스템 콜이 호출되어 커널 모드로 전환, 커널 함수(sys_open() 등)를 호출하게 된다. 이를 통해 파일을 여는 로우 레벨 연산이 수행되고 나면, 다시 사용자 모드로 전환되어 다음 동작을 계속 수행한다.

* 사전적 의미로 '커널(kernel)'은 견과류나 씨앗의 알맹이, '셸(shell)'은 껍질을 의미한다.
* 운영체제를 만든다면 운영체제 핵심 부분(커널)을 먼저 개발하고, 시스템 콜(주로 C언어), API, Shell 프로그램, 응용 프로그램 순으로 개발하게 된다.

<table>
<tr><td colspan="2">사용자(User)</td></tr>
<tr><td>응용 프로그램
(Application)</td><td>쉘(Shell)</td></tr>
<tr><td colspan="2">API / Library</td></tr>
<tr><td colspan="2">시스템 콜(System Call)</td></tr>
<tr><td colspan="2">운영체제(OS, kernel)</td></tr>
<tr><td colspan="2">하드웨어
(CPU, 메모리, 저장장치, 네트워크 등)</td></tr>
</table>

4) 프로세스 스케쥴링
- 여러 프로그램의 실행 순서를 관리한다.
- 배치 처리 시스템(Batch Processing): 응용 프로그램의 명령들을 큐(Queue) 방식으로 나열하여 순차적으로 처리하도록 만든다. 자동으로 다음 응용 프로그램이 이어서 실행된다.
 ㄴ 실행 시간이 긴 작업이 실행중이라면 나머지 작업은 오래 기다려야 함
 ㄴ 두 개 이상의 작업을 동시에 처리할 수 없다. (동시 실행/다중 사용자 등)
- 시분할 시스템: 여러 작업을 짧게 분할하여 실행해서, 응답 시간을 최소화한다.
 ㄴ 작업 1 - 작업 2 - 작업 3 - 작업 1 - 작업 2 - ... 와 같이 각 작업을 조금씩 처리
 ㄴ 다중 사용자 지원을 위해 응답 시간을 최소화하기 위해 도입되었다.
- 멀티 태스킹: 단일 CPU에서 여러 응용 프로그램이 동시에 실행되는 것처럼 보이게 하는 시스템. 작동 방식은 기본적으로 시분할 시스템과 같아서 유사한 의미로 혼용되는 용어이다.
 ㄴ 하나의 프로그램을 여러 CPU에 병렬로 실행하는 '멀티 프로세싱' 과는 구분된다.
- 멀티 프로그래밍: 시간 대비 CPU 활용도를 높여서, 최대한 CPU를 많이 활용하게 하는 시스템.
 ㄴ 응용 프로그래밍은 실행 중에 CPU 외에 다른 작업을 필요로 하는 경우가 많다.
 ㄴ ex) 파일 읽기 수행 시, 저장장치 속도가 CPU보다 훨씬 느리므로 파일 읽기가 끝날 때까지 CPU는 놀게 된다.

* 대략적인 액세스 시간 비교

CPU	레지스터(Register)	1 cycle
	캐시 메모리(level 1)	2-4 cycles
	캐시 메모리(level 2)	10 cycles
	캐시 메모리(level 3)	40 cycles
메모리	메인 메모리	200 cycles
저장장치	SSD	10-100 us
	HDD	10 ms

* 용어 정리
 - 여러 응용 프로그램을 CPU에서 효율적으로 처리하도록 한다는 점에서 목적이 같다.
 - 시분할 시스템: 다중 사용자 지원, 컴퓨터 응답시간 최소화
 - 멀티 태스킹: 한 CPU에서 여러 프로그램이 동시 실행되는 것처럼 보이게 함
 - 멀티 프로세싱: 여러 CPU에서 한 프로그램을 병렬로 실행하여 실행속도를 높임
 - 멀티 프로그래밍: 단위 시간당 CPU를 최대한 많이 사용하게 하여 CPU 사용률을 극대화하는 기법.

3. 스케쥴러 (Scheduler)

1) 프로세스와 스케쥴러
- 프로세스: 메모리에 올려져서 실행 중인 프로그램. 작업, task, job 이라고도 불린다.

ㄴ 하나의 응용 프로그램이 여러 개의 프로세스로 이루어질 수 있다.
- 스케쥴러: 프로세스 실행을 관리하는 프로그램
ㄴ 각 스케쥴러는 목표에 맞게 프로세스 실행 순서를 관리하는 알고리즘을 가진다.
ㄴ ex) 시분할 시스템은 응답 시간 최소화를 목표로 한다.

2) 기본적인 스케쥴링 알고리즘
(1) FIFO 스케쥴러
- 먼저 요청된 작업 순으로 처리(배치 처리 시스템)
- FCFS(First Come First Served) 라고도 부른다.
(2) 최단 작업 우선(Shortest Job First, SJF) 스케쥴러
- 적게 걸리는 작업부터 처리
- 프로세스 실행 시점에서 대기중인 프로세스들의 소요시간을 확인한다.
- 소요시간을 미리 알아야 한다는 단점이 있다.
(3) 우선순위 기반(Priority-Based) 스케쥴러
- 정적 우선순위: 프로세스마다 우선순위를 미리 지정
- 동적 우선순위: 스케쥴러가 상황에 따라 우선순위를 변경
(4) Round Robin 스케쥴러
- 시분할 시스템을 위한 기본 알고리즘
- 실행 순서는 FIFO 스케쥴러와 같으나, 각 프로세스는 일정 시간만 실행한다.
- 일정 시간 경과 후 작업이 모두 처리되지 않았다면, 큐에 다시 넣는다.

* RTOS와 GPOS
- Real-Time OS: 응용 프로그램의 실시간 성능 보장을 목표로 한다.
ㄴ 프로그램의 정확한 시작-완료 시간 보장. 시간에 민감한 환경(공장 등)에서 사용
ㄴ Hardware RTOS, Software RTOS 등도 있다.
- General Purpose OS: 프로세스 실행시간에 민감하지 않고 일반적인 목적으로 사용.
ㄴ 윈도우나 리눅스 등이 여기에 속한다.

3) 프로세스 상태
- 멀티 프로그래밍: CPU 활용도를 극대화하는 스케쥴링 알고리즘
ㄴ 프로세스들은 CPU의 작업을 요하는 상태와 그렇지 않은 상태가 있다. CPU에는 가능한 한 작업이 필요한 프로세스만을 올려두어 최대한 CPU 활용도를 높여야 한다.
ㄴ 따라서 프로세스의 상태를 알아야 효율적인 스케쥴링이 가능하다.
- 프로세스 상태는 생성(new)과 종료(exit) 사이에 [ready - running - block] 상태를 오가게 된다.
ㄴ running state: 현재 CPU에서 실행중인 상태. 실행 후 ready 또는 block으로 이동.
ㄴ ready state: CPU에서 실행이 가능한 상태(실행 대기 상태). 스케쥴러가 선택해서 CPU에 올리는 대상.
ㄴ block state: CPU에서 실행 불가, 특정 이벤트 발생 대기 상태. 이벤트 발생(파일 읽기 완료 등)시 ready 상태로 이동. blocking/blocked/wait/waiting 등의 용어로 혼용된다.
ㄴ ready 프로세스들 중 running 이후에 block 상태가 얼마나 지속될지 실행 전에는 알 수

없으므로, 이를 잘 고려하여 우선순위를 정해야 한다.
- 프로세스 상태 기반 알고리즘: 각 state별로 Queue를 만들어 관리하는 방식
 ㄴ Ready Queue에서 작업을 하나 꺼내어 Running으로 넘긴다.
 ㄴ running 이후 작업은 Block Queue 또는 Ready Queue로 이동한다.
 ㄴ block 상태가 끝난 작업은 Ready Queue로 이동한다.

4) 선점형과 비선점형
- 선점형(Preemptive) 스케쥴러: 하나의 프로세스가 다른 프로세스 대신 프로세서(CPU)를 차지할 수 있다. 즉 스케쥴러가 CPU에서 현재 실행중인 작업을 종료시킬 수 있다.
 ㄴ 최근 OS의 스케쥴러들은 대부분 선점형 형태를 띠고 있다.
 ㄴ 시분할 시스템은 기본적으로 선점형 스케쥴러여야 한다.
- 비선점형(Non-Preemptive) 스케쥴러: 하나의 프로세스가 끝나거나 자발적으로 blocking 상태에 들어가기 전까지, 다른 프로세스는 CPU를 사용할 수 없다.
 ㄴ 선점형 스케쥴러는 만들기 어렵기 때문에, 예전엔 대부분 비선점형을 사용했다.
 ㄴ FIFO, SJF, Priority-Based 등은 비선점형 스케쥴러의 알고리즘에 가깝다.

* 일반적으로 최신 스케쥴러들은 여러 알고리즘을 조합하여 만들어진다.
 - ex) SJF 기반으로 실행 순서 선택 + 시분할(선점형) + 프로세스 상태는 Queue로 + ...

4. 인터럽트 (Interrupt)

1) 역할
- 인터럽트: CPU가 다른 장치와 커뮤니케이션하는 수단. 특정 이벤트를 발생시켜 CPU가 실행 중인 프로그램을 중단하고 다른 작업을 실행하도록 하는 기술.
 ㄴ 선점형 스케쥴러의 경우, 프로세스 running 중에 스케쥴러가 이를 중단시키고 다른 프로세스로 교체하기 위해 인터럽트를 거는 동작이 필요하다.
 ㄴ IO 장치에 접근(파일 읽기 등)하는 동안 프로세스는 block(wait) 상태가 되므로, 해당 처리가 끝났을 때 다시 CPU에 알려서 작업을 계속하게 해야 한다.
 ㄴ 예외 상황 발생 시 CPU가 작업을 계속하지 않고 예외 처리를 할 수 있도록(예외 상황에서의 동작 실행 또는 프로세스 중단 등) 알려줘야 한다.
- IDT (Interrupt Descriptor Table): 인터럽트의 고유 번호. 인터럽트 발생 시 IDT를 확인하여 종류에 맞게 처리한다.
 ㄴ ex) 리눅스의 IDT는 0~31은 예외상황, 32~47은 하드웨어, 128은 시스템 콜을 의미한다.

2) 주요 인터럽트
- 내부 인터럽트: 잘못된 명령 또는 잘못된 데이터 사용 시 발생. 컴파일 타임에 감지할 수 없는 에러가 발생할 경우, 실행 시간에 운영체제가 인터럽트를 보내서 실행을 중단시키고 에러 메시지를 띄우게 해 준다.
 ㄴ Divide-by-Zero, 허용되지 않은 공간 접근, Overflow/Underflow, 시스템 콜 등

- 외부 인터럽트: 주로 하드웨어에서 발생되는 이벤트
 ㄴ 타이머 인터럽트: 컴퓨터 내에 타이머 인터럽트 발생 장치가 별도로 존재해서, 일정 시간마다 운영체제에 알려준다.
 ㄴ 전원 이상 또는 하드웨어에 문제가 있을 경우
 ㄴ IO 이벤트: 저장장치/프린터/키보드 등의 동작을 기다려야 할 경우, 해당 동작이 끝나면 이벤트가 발생하여 CPU에 알려주게 된다.

* 내부/외부 인터럽트는 소프트웨어/하드웨어 인터럽트라고도 한다.

2-1) 시스템 콜 인터럽트
- 내부 인터럽트의 하나. 리눅스 기준 인터럽트 번호 0x80 (==128)
- 시스템 콜은 각 명령마다 번호를 가진다. eax 레지스터에 시스템 콜 번호를, ebx에는 인자의 값을 넣고, 소프트웨어 인터럽트 번호를 전달하여 명령을 내린다.
- 시스템 콜을 호출 시 커널 모드로 전환, 인터럽트 번호가 시스템 콜임을 확인하고 시스템 콜 번호(eax)를 찾아서 해당 함수를 실행시킨다. 이후 다시 사용자 모드로 전환하고 다음 코드를 실행한다.

5. 프로세스 (Process)

1) 구조
- 프로세스는 크게 스택, 힙, 데이터, 코드 영역으로 구분된다.
 ㄴ 스택(stack): 함수의 인자나 멤버 변수, 반환된 결과값 주소 등 임시 데이터가 저장된다.
 ㄴ 힙(heap): 동적으로 생성되는 값이 저장된다.
 ㄴ 데이터(data): 전역 변수, 정적 변수로 선언된 데이터와 초기화값이 저장된다.
 ㄴ 코드(code): 기계어로 변환된 코드가 들어있다. 텍스트(text) 영역이라고도 부른다.

* 데이터 영역은 다시 BSS와 DATA로 나뉜다. BSS는 초기화되지 않은 전역 변수가 들어간다.

2) 컨텍스트 (Context)
- CPU에서 현재 작업중인 상황을 나타내는 정보들. 주요한 요소로는 레지스터의 스택 포인터(SP)와 프로그램 카운터(PC)가 있다.
 ㄴ 스택 포인터(Stack Pointer): 스택 영역(stask frame)의 최상단을 가리킨다.
 ㄴ 프로그램 카운터(Program Counter): 다음에 실행할 명령어 주소를 가리킨다.
- 실행 중인 프로세스를 도중에 변경할 경우, 운영체제는 SP와 PC를 포함한 현재 프로세스의 컨텍스트 정보를 PCB에 저장하고 새로운 프로세스를 실행시킨다.
 ㄴ 프로세스 제어 블록(PCB, Process Control Block): 운영체제가 프로세스에 대한 정보를 기록해두는 메모리 공간. 프로세스의 ID, 레지스터 값(PC, SP 등), 상태, 메모리 크기, 스케쥴링 정보 등을 저장한다.
 ㄴ 다시 이전 프로세스를 실행시키면 PCB의 정보를 다시 꺼내어 레지스터에 올린다.

3) 컨텍스트 스위칭 (Context Switching)
- CPU에 실행할 프로세스를 교체하는 기술
(1) 현재 프로세스 정보를 해당 프로세스의 PCB에 업데이트해서 메인 메모리에 저장
(2) 다음 프로세스의 PCB를 메모리에서 로드해서 CPU에 넣고 실행
- 실제로 매우 짧은 시간(ms) 단위로 컨텍스트 스위칭이 일어나므로, 스위칭에 따른 오버헤드를 최소화할 필요가 있다. 가령 리눅스의 경우 이 부분만 어셈블리어로 작성되기도 했다.

* (2)와 같이 ready 상태의 프로세스를 CPU에 넣어 running 상태로 만드는 것을 '디스패치(Dispatch)' 라고 한다.

4) 프로세스 간 커뮤니케이션
- 원칙적으로 프로세스는 다른 프로세스의 공간에 접근할 수 없다.
ㄴ 리눅스 기준, 모든 프로세스는 4GB의 가상 메모리 공간을 가지며 그 중 1GB는 커널 공간에 할당되어 있다. 각 프로세스의 커널 공간은 실제 물리 메모리에서 동일한 공간으로 공유하지만, 나머지 3GB의 사용자 공간은 완전히 분리되어 있다.
- 성능을 높이기 위해 하나의 프로그램을 여러 프로세스로 만들어 동시 실행하는 등의 상황에서, 프로세스간 상태 확인 및 데이터 송수신이 필요하게 된다.
- fork(): 시스템 콜의 하나로, 프로세스를 복사해서 새로운 프로세스로 만든다. 원본 프로세스를 부모 프로세스, 새로 만들어진 프로세스를 자식 프로세스라고 한다.
ㄴ ex) 1~10000 더하기 작업을 fork()로 10개의 프로세스를 만들어 1000 단위씩 더하게 해서 시간을 1/10으로 단축시킬 수 있다. 하지만 더한 값을 모두 합하는 작업이 필요하다.
ㄴ ex) 웹서버는 클라이언트 요청이 올 때마다 HTML 파일을 제공한다. 요청이 올 때나다 fork()로 새 프로세스를 만들면 CPU 병렬 처리가 가능할 때 빠른 대응이 가능하다

5) IPC 기법 (InterProcess Communication)
- 프로세스 사이에 통신을 위한 여러 가지 기법.
- 대부분의 IPC 기법은 여러 프로세스 간 공유되는 커널 공간을 활용한다.
(1) 파일(file): 파일을 만들어 저장하면 어떤 프로세스든 접근이 가능하다. 원시적인 방법.
    - 파일 특성상 읽고 쓰는 시간이 오래 걸려 실시간 데이터 전달이 어렵다.
(2) 파이프(pipe): 공유되는 커널 공간을 통한 부모 -> 자식 프로세스의 단방향 통신
    - pipe()에 배열을 넘겨 커널 공간에서 관리되는 파이프를 만든다.
    - fork()로 자식 프로세스를 만들어, pipe()에 넘긴 배열 주소를 읽게 한다.
    - 부모 프로세스는 pipe()에 넘긴 배열 주소에 데이터를 기록한다.
(3) 메시지 큐: key값을 이용해 공유되는 큐에 접근하여 데이터를 주고받을 수 있다.
    - 메시지 큐의 아이디를 넣고, 데이터를 보내거나 받는다.
    - 파이프와 달리 어느 프로세스간에도 데이터 송수신이 가능하다.
(4) 공유 메모리: 커널 영역에 메모리 공간을 할당하여 변수처럼 쓰는 방식
    - 단방향이나 FIFO와 같은 제약 없이 변수처럼 쓸 수 있다.
    - 공유 메모리 key만 있으면 여러 프로세스가 접근 가능하다.
(5) 시그널(signal): 커널 또는 프로세스에서 다른 프로세스에 이벤트를 전달하는 방식

- 유닉스에서 30년 이상 사용된 전통 기법
- 시그널 핸들러에서 이벤트를 받아 해당 시그널 처리 실행
- 프로세스 킬, 인터럽트, 실행 등 시그널과 처리할 내용은 미리 정의되어 있다.
  ㄴ 특정 시그널은 프로세스에서 원하는 동작을 할당하거나 재정의할 수 있다.

(6) 소켓(socket): 네트워크 통신을 위한 기술을 한 컴퓨터 내의 IPC 기법으로 활용한 것

6) 실행파일과 프로세스

- 실행파일: 저장매체에 저장되어 있으며, 실행 시 복사되어 메모리에 올라간다.
 ㄴ '코드 이미지' 또는 '바이너리', '응용 프로그램', '어플리케이션' 등으로도 부른다.
- 프로세스: 실행파일이 복사되어 메모리에 올라간 것. 프로세서에 의해 실행 가능하며 프로세스 상태 정보(PCB)를 포함한다.
 ㄴ 현재 필요한 코드나 데이터만 물리 메모리에 올라가서 실행된다.
 ㄴ 스케쥴링 단위가 되며, 운영체제의 스케쥴러를 통해 제어된다.
 ㄴ 가상 및 물리 메모리 정보, 사용중인 시스템 리소스 등 프로세스 상태 정보는 실행 파일엔 존재하지 않는다.

6. 스레드 (Thread)

1) 개념

- 개별적인 스택 공간, SP, PC를 가지는 하나의 작업 단위.
 ㄴ 레지스터와 스택만 별개이며 코드, 데이터, 힙 영역은 공유한다.
- 하나의 프로세스에 여러 스레드 생성 및 동시에 실행 가능
- 프로세스 안에 있으므로, 같은 프로세스 내의 데이터에 모두 접근 가능
 ㄴ Light Weight Process 라고도 한다.
- 최근 운영체제는 여러 프로세스의 동시 실행(멀티태스킹) + 프로세스 내에 여러 스레드 실행(멀티프로세싱)을 모두 지원한다.

2) 장단점

- 장점

(1) 사용자에 대한 응답성 향상
: 한 스레드가 무거운 연산을 수행하는 동안 다른 스레드는 사용자와 상호작용할 수 있다.
(2) 자원 공유 효율
: IPC 기법과 같이 자원 공유를 위한 번거로운 작업이 불필요하여 오버헤드 감소
(3) 작업 분리로 인한 CPU 활용도 향상
: 각 스레드가 실행할 함수를 분리하여 멀티 프로세싱 활용이 가능하며, 멀티 프로그래밍에 유리하다.
 ㄴ 작성하기에 따라 코드가 간결해지는 효과도 있다.

- 단점

(1) 예외 상황에 취약
: 한 스레드가 문제가 생기면 전체 프로세스가 영향을 받는다.

ㄴ 멀티 프로세스의 경우 프로세스 하나에 문제가 생겨도 다른 프로세스는 영향 X
(2) 컨텍스트 스위칭 비용 발생
: 스레드를 많이 생성하면 컨택스트 스위칭이 잦아 성능이 저하될 수 있다.
ㄴ ex) 리눅스는 스레드를 프로세스처럼 다루며 모든 스레드를 스케쥴링한다.

* POSIX Thread (PThread)
- 스레드 관련 표준 API. 코드상에서 스레드를 사용할 수 있게 해 주는 인터페이스.
ㄴ POSIX: 유닉스의 API 규격. 시스템 콜도 여기에 정의되어 있다.
ㄴ 각 언어의 API도 내부적으로 PThread를 사용하는 경우가 많다.

3) 스레드 동기화 문제
- 멀티스레드 환경에서 한 스레드가 작업 중에 컨텍스트 스위칭이 발생하여 예기치 않은 결과가 나오는 문제
ㄴ 하나의 작업은 몇 단계의 연산으로 나뉘어 있는데, 일부 작업은 여러 연산이 도중에 멈추지 않고 한 번에 실행되어야 하는 경우가 있다.
- 임계 구역(Critical Section): 여러 스레드 또는 프로세스가 동시에 접근할 수 없는 영역
ㄴ 상호 배제(Mutual Exclution): 임의의 시점에서 한 프로세스만 접근하도록 제어하는 것

4) 세마포어 (Semaphore)
- 스레드 동기화 문제 해결을 위해 임계 구역을 제어하기 위한 도구
- P(검사), V(증가), S(세마포어 값)로 구성된다.
ㄴ P: 임계구역 접근 시 S값이 1 이상인지 확인하고 1 감소시킨다.
ㄴ V: 임계구역을 나올 때 S값을 1 더한다.
ㄴ S: 이 값이 0 이하면 임계 구역 진입이 불가하다.
ㄴ 즉 임계 구역은 최초 S값의 수만큼 동시 접근이 가능하게 된다.
- S값이 0일 경우 해당 프로세스는 대기해야 한다.
ㄴ 바쁜 대기: 초기 방식으로, 반복문을 돌며 무한 대기한다.
ㄴ 대기 큐: 작업을 큐에 넣어두고 S값이 증가하면 wakeup() 등의 함수로 재실행한다.

* POSIX 세마포어에 세마포어 관련 함수가 정의되어 있다.

5) 교착 상태와 기아 상태
- 임계 구역을 제어하는 과정에서 특정 프로세스나 스레드가 진행되지 않는 상태
- 교착 상태(Deadlock): 둘 이상의 프로세스가 서로가 끝나기를 대기하는 무한루프 상태
ㄴ 설계 시 순환구조 방지, 대기 시 자원 점유 해제 등으로 해결
- 기아 상태(Starvation): 특정 프로세스 우선순위가 낮아 계속 자원을 할당받지 못하는 상태
ㄴ 오래 기다린 프로세스 우선순위를 높이거나, FIFO처럼 순서대로 처리하는 등 우선순위 없이 처리하는 방식 도입으로 해결

* 참고: 교착상태 발생 조건
- 상호 배제: 해당 작업이 임계 구역에 해당할 것

- 점유 대기: 특정 자원을 점유한 채 다른 자원을 기다림
- 비선점: 다른 프로세스의 자원을 뺏을 수 없음
- 순환 대기: 각 프로세스는 상대방이 기다리는 자원을 가지고 있음
- 교착 상태를 방지하려면 이 조건들 중 하나 이상을 방지하는 동작이 필요하다.

7. 가상 메모리

1) 개념
- 메모리가 실제(물리) 메모리보다 많아 보이게 하는 기술
  ㄴ 실제 사용하는 메모리는 할당된 것보다 작다는 점에 착안하여 고안된 기술
- 여러 프로세스를 동시 실행하는 환경에서 메모리 크기가 부족하기 때문에 사용한다.
  ㄴ ex) 리눅스는 프로세스당 4GB 필요
  ㄴ 폰 노이만 구조에서, 코드는 반드시 메모리에 위치해야 한다.
- 프로세스는 가상 주소를 사용하고, 실제 메모리 주소에는 일부만 올려놓는다.
- MMU(Memory Management Unit): CPU가 다루는 가상 메모리 주소를 물리 주소로 빠르게 변환시켜주는 하드웨어 칩. 빠른 동작을 위해 하드웨어 장치를 사용한다.

* 프로세스당 4GB를 사용하는 이유는, 32bit 시스템에서 2^32바이트 = 4GB면 모든 주소를 표현할 수 있기 때문

2) 페이징 시스템
- 페이징(paging): 가상 메모리를 동일한 크기의 '페이지'로 나누어, 페이지 번호를 기반으로 가상/물리 메모리의 주소를 매핑하여 사용한다. 하드웨어 지원이 필요하다.
  ㄴ 인텔 x86(32bit) 시스템은 4KB, 2MB, 1GB를 지원한다. 그 중 리눅스는 4KB를 사용한다.
  ㄴ 페이지는 크기가 고정적이며, '페이지 프레임(page frame)'으로도 불린다.
- 페이지 테이블(Page Table): 페이지별로 가상-물리 메모리 주소의 매핑 정보를 기록한 테이블. 프로세스의 PCB에는 페이지 테이블 구조체(테이블 시작점)의 주소가 들어 있다.
  ㄴ 프로세스 구동 시 페이지 테이블 시작점의 주소는 CR3 레지스터에 올라간다.
  ㄴ [페이지 번호 - 가상 메모리 주소 - 물리 메모리 주소 - 유효성 비트] 로 구성된다.
  ㄴ 유효성 비트는 실제 물리 메모리에 해당 정보가 들어 있는지의 여부를 나타내는 비트
- 가상 주소는 페이지 번호와 오프셋으로 구성된다.
- 페이징 시스템은 아래와 같이 동작한다
  (1) CPU에서 가상 주소를 보내며 데이터를 요청
  (2) MMU가 가상 주소를 가지고 PCB(물리 메모리에 있음)의 페이지 테이블 접근
  (3) 가상 주소의 페이지 번호를 찾아서 매칭된 물리 메모리 페이지 확인
  (4) 물리 메모리 페이지 주소 + 오프셋으로 원하는 물리 메모리 주소 확인
  (5) 해당 주소의 데이터 반환

* 32bit 시스템에서 리눅스 기준 한 페이지는 4KB, 하위 12bit는 오프셋, 상위 20bit는 페이징 번호로 구성된다.

2-1) 다중 단계 페이징 시스템
- 페이지 정보를 단계를 나누어 생성한다. 페이지들의 집합을 페이지 디렉토리로 만들어, 사용하지 않는 페이지 테이블은 디렉토리 단계에서 무시할 수 있다.
 ㄴ 프로세스에 할당된 4GB 전체를 페이징하는 오버헤드를 줄이기 위해 사용
- 가상 주소의 페이지 번호 부분은 다시 [디렉토리 + 페이지 번호] 로 나뉘게 된다.

2-2) 요구 페이징(Demand Paging)
- 프로세스 데이터를 필요한 시점에만 물리 메모리에 적재하는 기법
- 탐색한 페이지가 유효하지 않다면(물리 메모리에 없다면) '페이지 폴트(page fault)' 인터럽트가 발생하고, 운영체제가 해당 페이지의 데이터를 메모리에 올린다.
- 페이지 폴트가 자주 일어나면 시간이 오래 걸리므로, 향후 실행될 코드/데이터를 미리 예측하여 올려두면 좋다. 이를 위한 알고리즘들도 존재한다.
 ㄴ 스레싱: 페이지 폴트가 자주 발생하여 페이지 교체 시간이 오래 걸리는 상황

* 물리 메모리에 올라가지 않은 데이터는 저장매체(HDD/SSD)에 위치한다.

3) TLB (Translation Look-a-side Buffer)
- 캐시의 일종으로, 최근 탐색한 페이지 테이블 정보를 저장해 놓는 공간
- CPU가 가상 주소를 보내면 MMU가 메모리(PCB)의 페이지 테이블을 탐색, 실제 메모리 주소를 알아내어 다시 해당 주소를 탐색한다. 즉 메모리를 2번 탐색한다.
- TLB를 사용하면 MMU는 TLB에 해당 주소 정보가 있는지 먼저 탐색한다. TLB에 정보가 있으면 그 주소를 가지고 메모리에서 데이터를 가져온다. 즉 메모리를 1번만 탐색해도 된다.

4) 공유 메모리
- 서로 다른 프로세스가 동일한 물리 주소를 가리킬 수 있다.
 ㄴ fork()로 자식 프로세스를 만들 때 사용되어, 공간 절약, 메모리 할당 시간 절약이 가능
- 각 프로세스에 할당된 가상 메모리는 다르지만, 두 프로세스의 페이지 테이블이 동일한 물리 메모리 주소를 가리키면 동일한 메모리 공간을 사용할 수 있다.
- 단, 데이터를 수정할 경우 다른 프로세스의 데이터가 수정되면 안 되므로, 해당 부분의 데이터에 한하여 복사되어 각 프로세스에 다르게 할당된다.

5) 페이지 교체 알고리즘 (page replacement algorithm)
- 페이지 교체 정책: 특정 페이지를 메모리에 올리려는데, 메모리가 꽉 차 있을 경우, 현재 메모리에 있는 페이지를 다시 저장매체로 옮긴다.
- 페이지 교체 알고리즘: 물리 메모리에서 어떤 페이지를 저장 매체로 옮길 것인가?
 (1) FIFO: 먼저 메모리에 올라간 페이지를 내린다.
 (2) OPT (OPTimal Replacement Algorithm)
  - 앞으로 가장 오래 사용하지 않을 페이지를 내린다.
  - 이상적인 방식이지만 사실상 구현 불가, 이를 목적으로 여러 알고리즘이 파생됨
 (3) LRU (Least Recently Used)

- 가장 오래 전에 사용된 페이지를 내린다.
- OPT 교체 알고리즘을 과거 기록을 기반으로 예측하여 시도한 것

(4) LFU (Least Frequently Used)
- 가장 적게 사용된 페이지를 내린다.

(5) NUR (Not Used Recently)
- LRU와 마찬가지로 최근 사용하지 않은 페이지부터 내린다.
- 각 페이지마다 참조, 수정 비트를 두어 해당 페이지를 읽거나 썼는지 확인한다.
- 참조+수정됨 < 참조됨 < 수정됨 < 둘다X 순으로 우선순위를 매겨 내린다.

5) 세그멘테이션 (Segmentation)
- 가상 메모리를 서로 크기가 다른 논리적 단위인 '세그먼트'로 분할한다.
 ㄴ Code Segment, Data Segment, Stack Segment, Extra Segment 등 세그먼트의 역할에 따라 논리적으로 나뉜다. 페이징 기법이 같은 크기의 블록으로 분할하는 점과 구분된다.
 ㄴ 대부분은 페이징 기법이 사용되니 참고용으로만 알아둘 것
 ㄴ 이 기법 또한 하드웨어 지원이 필요
- 세그먼트 가상주소는 [세그먼트 번호 + 오프셋] 으로 구성된다.
- 페이지 기법은 내부 단편화, 세그멘테이션은 외부 단편화 발생 가능성이 있다.
 ㄴ 외부 단편화: 한 세그먼트의 크기가 클 때, 물리 메모리가 원하는 크기의 메모리를 제공해주지 못할 경우
 ㄴ 내부 단편화: 페이지 블록만큼 데이터가 딱 맞게 채워지지 않을 때 공간이 낭비된다. 예를 들어 1KB를 저장하려 해도 4KB 페이지가 할당되어야 한다.

8. 파일 시스템

1) 개념
- 운영체제가 저장매체에 파일을 쓰기 위한 자료구조 또는 알고리즘
- 0과 1의 데이터를 비트 단위로 주소를 붙여 관리하면 오버헤드가 너무 크므로, 블록 단위(보통 1~4KB)로 고유 번호를 부여해서 관리하는 것에서 시작되었다.
 ㄴ 블록 번호는 사용자가 관리하기 어려우니 '파일'이라는 추상적 객체로 묶은 것. 각 파일은 블록 단위로 관리된다.
- 연속적인 공간에 저장하면 관리가 쉬우나, 외부 단편화 문제나 파일 사이즈 변경 등의 문제가 있다. 따라서 불연속 공간에 파일 저장 기능이 필요해졌다.
 ㄴ 블록 체인: 블록을 링크드 리스트로 연결하는 기술
  - 첫 번째 블록 주소로 파일을 열며, 끝 블록까지 순서대로 탐색해야 한다.
 ㄴ 인덱스 블록: 각 블록의 위치 정보를 별도의 자료구조로 기록해두는 기술
  - 원하는 바로 블록을 찾아갈 수 있다.

2) 가상 파일 시스템 (Virtual File System)
- 모든 디바이스를 파일 시스템 인터페이스로 관리하는 기능
 ㄴ 모든 자원에 대한 추상화 인터페이스로서 파일 시스템 인터페이스를 활용

- 파일 읽기/쓰기 시스템 콜 호출 시, 각 기기 및 파일 시스템에 맞는 처리를 OS에서 구현해 놓았으므로, 파일이 어떻게 저장되든 동일한 함수로 파일 시스템을 사용할 수 있다.
 ㄴ 윈도우: FAT, FAT32, NTFS (최근엔 NTFS 많이 사용)
  - 블록 위치를 FAT라는 자료 구조에 기록한다.
 ㄴ 리눅스: ext2, ext3, ext4
  - 인덱스 블록 기법인 inode 방식 사용
  - inode는 유닉스 계열에서 가장 핵심이 되는 파일 시스템
- 가상 파일 시스템은 네트워크나 주변기기도 파일 시스템처럼 시스템 콜을 사용하여 다룰 수 있도록, 각 기기별로 내부 동작을 운영체제에 구현해 놓은 것을 의미한다.
 ㄴ 마우스, 키보드 등도 파일을 읽고 쓰는 것처럼 이루어진다.

* 특수 파일
 - 블록 디바이스(HDD/CD/DVD): 블록/섹터 등 정해진 단위로 데이터 전송
  ㄴ IO 송수신 속도가 높다.
 - 캐릭터 디바이스(키보드/마우스): byte 단위로 데이터 전송
  ㄴ IO 송수신 정도가 낮다.
 - 내부적인 처리를 위한 구분일 뿐 외부적으로 사용하는 시스템 콜 인터페이즈는 동일

3) 아이노드(inode) 방식 파일 시스템
- 가상 메모리의 페이징 시스템처럼, 운영체제의 파일 시스템은 inode 방식이 대부분이다.
- 파일 시스템 기본 구조는 수퍼 블록, 아이노드 블록, 데이터 블록으로 구성된다.
 (1) 수퍼 블록: 파일 시스템 전체에 대한 정보, 파티션 정보
  - 사용중인 용량, 디렉토리들, 1KB 파일 개수 등등
 (2) 아이노드 블록: 각 파일마다 가진 상세 정보
  - 파일이 생길 때, 각 파일 이름에 매칭되는 inode 번호가 생성된다.
  - inode 번호에 매칭된 inode 블록에는 해당 파일에 관한 메타데이터가 저장된다.
   ㄴ 파일 권한, 소유자, 크기, 저장 위치, 생성시간, direct block, indirect 등
  - 아이노드 블록 내부의 direct block에 데이터 블록의 주소가 들어 있다.
 (3) 데이터 블록: 실제 데이터
- 파일 호출 시 [해당 파일명의 inode 번호 -> inode 블록 탐색 -> inode 블록에서 데이터 블록 주소 탐색 -> 데이터 블록 읽어서 반환] 순으로 데이터를 불러온다.
 ㄴ 파일명은 파일 주소의 디렉토리 엔트리를 순차적으로 탐색하여, 파일이 위치하는 디렉토리에서 파일명을 하나씩 탐색한다.
- (Single) Indirect: 아이노드 블록의 구성 요소로, 4KB 공간에 데이터 블록을 가리키는 주소만을 저장하고 있다. 주소 하나가 4byte면 1024개의 주소로 4KB * 1024 = 4MB의 데이터를 가리키고 있는 것.
 ㄴ direct block은 12개뿐이므로 저장용량이 일정 수준 이상이면 이 방식을 사용
 ㄴ Double Indirect는 4KB 공간에 Single Indirect들을 가리킨다. 즉 4MB * 1024 = 4GB 용량을 나타낼 수 있다. Triple Indirect도 같은 방식으로 Double Indirect를 가리킨다.

9. 부팅과 가상 머신

1) 부팅 (Boot)
- 컴퓨터를 켜서 동작시키는 절차
(1) ROM에 저장된 펌웨어인 'BIOS 코드'를 로딩, BIOS 프로그램을 메모리에 올린다.
    - 컴퓨터가 꺼져도 데이터가 유지되는 메모리인 ROM 칩에 ROM-BIOS가 들어 있다.
(2) BIOS는 하드웨어를 초기화하고 저장장치의 특정 부분(MBR)에서 부트 로더를 읽어온다.
    - POST(Power On Self Test)라는 주변 하드웨어를 체크한다.
    - MBR(Master Boot Record): 하드 디스크 맨 앞부분의 시스템 기동 영역
    - 부트스트랩이 MBR에서 부팅정보를 읽어 램에 올린다.
(3) 부트 로더는 저장매체의 부트섹터 파티션을 찾아 부트 코드를 로드한다.
(4) 부트 코드는 부트섹터 안의 커널 이미지(운영체제 실행파일)의 주소를 메모리에 불러와서, CPU의 프로그램 카운터를 커널의 첫 실행 위치로 가져다 놓는다.
    - 이 때 운영체제 이미지(윈도우 로고)도 화면에 출력된다.
(6) 이후 CPU에 의해 운영체제 프로그램이 실행된다.
    - 부팅 직후 최초 프로세스(init)가 실행되고, 이후 프로그램은 fork()로 실행된다.
    - 쉘 프로그램도 fork를 통해 실행된다.

* 실행파일을 '코드 이미지' 또는 '바이너리'로도 부른다.

2) 가상 머신 (Virtual Machine)
- 하나의 하드웨어에 다수의 운영체제를 설치, 개별 컴퓨터처럼 동작하도록 하는 프로그램
- 하드웨어는 하나지만 그 위에 가상 기계를 여러 대인 것처럼 구현, 각 가상 기계마다 커널을 실행하여 개별 동작하게 한다.
- 하이퍼바이저 또는 VMM(Virtual Machine Monitor) 소프트웨어에 의해 구현된다.
(1) Virtual Machine Type 1 (native 또는 bare metal)
  - 전가상화(Full Virtualization). 가상 머신마다 OS가 개별적으로 구동된다.
  - 소프트웨어를 하드웨어 바로 위에 설치해서 직접 구동하는 방식. (Xen, KVM 등)
  - 하이퍼바이저는 OS의 명령을 하드웨어에 전달하는 통역사 역할.
  - 하이퍼바이저가 마치 하드웨어인 것처럼 동작, OS는 자신이 가상 머신인지 모른다.
(2) Virtual Machine Type 2
  - 반가상화. Host OS 위에 소프트웨어를 설치한다. (VMWare, Parallels Desktop 등)
  - 호스트 운영체제를 거쳐야 하므로 속도가 Type1 보다는 느리다.
  - 하이퍼바이저는 각 OS에 분배되는 리소스를 관리한다.
  - 각 OS는 자신이 가상 머신임을 인지하고, 각 명령에 하이퍼바이저 명령을 추가하여 하드웨어와 직접 통신한다. (통역 불필요)
- 최근에는 하드웨어 성능 개선으로 전가상화 기술을 선호한다.

3) 가상 머신 프로그램
(1) VMWare: 대중적인 가상머신 프로그램 (Type2)
(2) KVM: 아마존 클라우드 서비스(AWS) 등에서 사용 (Type1)

- 리눅스 커널을 사용하며, ioctl 시스템 콜로 하드웨어를 직접 제어한다.
- 해당 시스템 콜을 가상 CPU를 만들며, CPU가 지원해야 사용할 수 있다.
- Intel-VT 등 가상화 기능을 가진 CPU는 VMX root/non-root 모드 존재
- 게스트 커널(protection ring 0) 위에서 응용 프로그램(ring 3) 사용
  ㄴ 게스트 커널이 하드웨어 자원 요청 시 KVM 모듈에서 처리

(3) Docker: 운영체제 레벨에서 별도로 분리된 실행환경을 제공
- 리눅스의 chroot 명령어를 사용하여, 처음 설치했을 때의 실행환경을 만든다.
- 하드웨어 가상화가 아닌 커널 추상화에 해당, 가상머신엔 별도의 OS가 불필요
- 리눅스 외 운영체제에 설치할 때는 사실상 리눅스+Docker 환경을 설치하는 것
- 경량 이미지로 실행환경을 통째로 백업, 실행 가능
  ㄴ 실행환경을 설치하는 것이므로 환경 설정 + 프로그램이 한번에 배포된다.
  ㄴ 가상머신마다 환경설정을 할 필요가 없다
  ㄴ 자동 환경설정 및 업데이트 -> Jenkins 등과 연계되어 사용된다.

(4) Java Virtual Machine
- 파일을 어느 운영체제에서나 실행 가능하도록 가상 환경을 만드는 것
- 가상 머신과는 다르며, 응용프로그램 레벨에서의 가상화

*** 메모**

- 운영체제: 커널(운영체제) + 시스템 프로그램(쉘) + 응용 프로그램
  ㄴ 리눅스 쉘은 Bourne-Again Shell (bash)를 디폴트로 사용
  ㄴ CPU: 스케쥴러
  ㄴ 메모리: 가상 메모리, 페이징 시스템
  ㄴ 저장장치: 파일 시스템, 블록
  ㄴ IO장치: 가상 파일 시스템 - 캐릭터 디바이스
  ㄴ 네트워크 (추후 다룰 내용)
- 시스템 프로그램: 핵심은 쉘(shell). 내부적으로 해당 운영체제의 시스템 콜 호출.
  ㄴ Android OS: 리눅스 커널 + 쉘 + C library/Java 가상머신 + 안드로이드 플랫폼(프레임워크) + 해당 프레임워크를 이용해 만든 안드로이드 응용 프로그램들
  ㄴ 안드로이드는 사실상 리눅스 OS에 안드로이드 플랫폼을 얹은 것