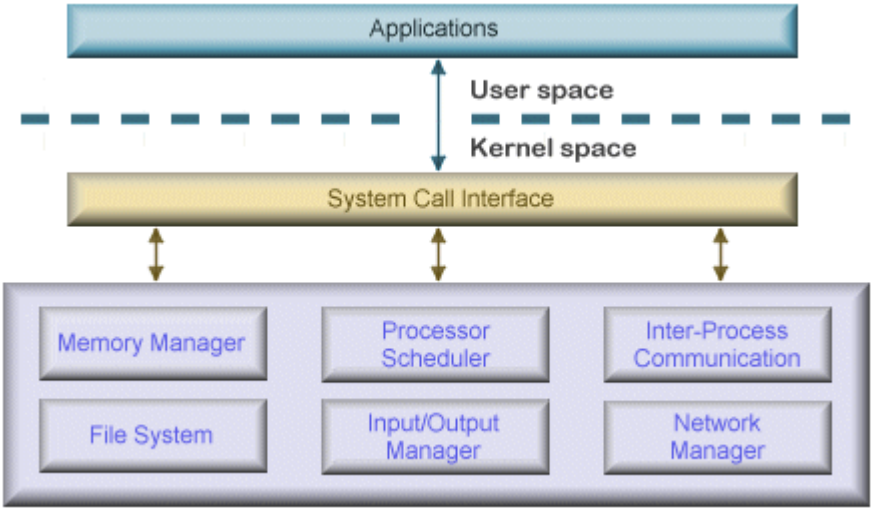
1. Operation System (OS, process & thread)

https://suhyunsim.github.io/2023-03-14/운영체제-면접질문

---

1. OS 기본

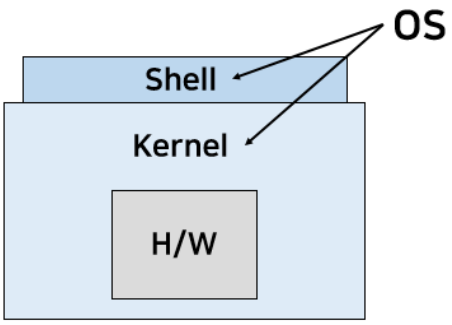
1. OS란 무엇인가요?



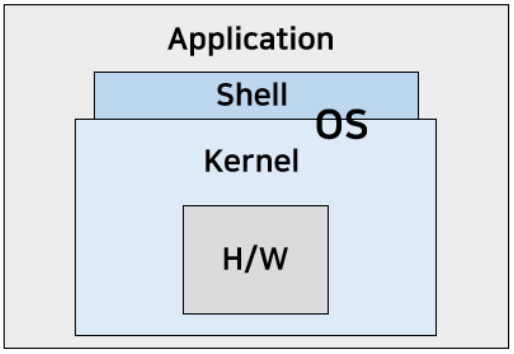
⇒ 하드웨어로 구성된 컴퓨터를 사용하기위한 최소한 환경을 제공하는 소프트웨어, 컴퓨터 위에서 실행한 모든 **프로그램**들은 운영체제가 **관리 및 제어**한다.

⇒ CPU, 메모리, 디스크 IO 장치 등의 하드웨어와 응용 소프트웨어 간의 가교를 담당, 즉 **하드웨어**를 **제어**함으로써 컴퓨터 내부 동작을 관리하는 소프트웨어 인터페이스 (HW ↔ OS ↔ SW)

⇒ 복잡한 하드웨어 관리를 OS가 담당함으로써, 사용자는 이에 신경쓰지 않고 컴퓨터 사용!



※ 통신 과정 : User ↔ System Program (Shell) ↔ Kernel ↔ Hardware



⇒ 애플리케이션은 운영체제 위에서 수행하며, 하드웨어 자원을 직접적으로 이용하지 않고 운영체제가 제공하는 자원만을 사용할 수 있다.

⇒ 애플리케이션에 비해, utility라 부르는 시스템 프로그램은 유저가 컴퓨터를 이용 및 관리에 필요한 보조 기능을 제공하는 프로그램이다.

⇒ 유틸리티 프로그램에는 쉘, 디버거, 컴파일러, 디스크 관리 프로그램 등이 해당된다.

- 프로세스, 메모리, 하드디스크 등 하드웨어 자원이 존재하고, 이를 효율적으로 사용해야 한다.
- 자원 관리를 위해 프로세스 관리, 메모리 관리, 디스크 관리, 네트워크, 보안 등 기능이 나눠져 있다.
- 애플리케이션들의 요청에 따라 각 기능들이 수행하여 적절히 자원을 분배한다.
- 에러 및 각종 오류 상황을 감지하고 이를 상위 계층에게 보고하는 역할도 수행한다.

1. Kernel

⇒ “운영체제 핵심”, 컴퓨터 자원(CPU, File, Network, IO 장치)을 관리하는 프로세스 집합, 때문에 User와 상호작용을 하지 않는다.

⇒ 메모리에 상주하는 운영체제의 일부분으로 운영체제의 항상 필요한 핵심부분을 의미!

⇒ 컴퓨터의 물리적 자원(하드웨어)과, 이를 추상화한 자원을 관리하는 역할을 수행한다.

⇒ 커널의 추상화 : 하나뿐인 하드웨어를 여러 개인것 처럼 사용할 수 있는 기술

⇒ 이를 활용, 하나의 컴퓨터에 등록된 여러 명의 사용자들을 위한 여러 개의 추상화된 객체로 관리

⇒ 커널이 이러한 물리적 자원을 추상화하여 관리하는 이유?

- 사용자가 물리적 하드웨어에 접근하고 사용할 수 있도록 매개하기 위함
- 하드웨어 ↔ 커널 ↔ 사용자, 하드웨어와 연결될 수 있는 핵심 가교 역할!

```
물리적 자원 이름 -> 추상화한 자원 용어CPU -> 태스크(Task)
메모리(memory) -> 페이지(page), 세그먼트(segment)
디스크(disk) -> 파일(file)
네트워크(network) -> 소켓(socket)
```

※ Kernel의 5가지 관리자

1. 프로세스 관리

   ⇒ 프로세스 생성 및 실행, 제어, 관리

2. Task 관리자

   ⇒ 물리적 자원인 CPU를 추상자원인 Task로 제공

3. 메모리 관리자

   ⇒ 물리적 자원인 메모리를 추상자원인 페이지 & 세그먼트로 제공 (메모리 할당 및 회수)

4. 파일 시스템 관리자

   ⇒ 물리적 자원인 디스크를 추상자원인 파일로 제공 (파일 저장 및 파일시스템 운영)

5. 네트워크 관리자

   ⇒ 물리적 자원인 네트워크 장치를 추상자원인 소켓으로 제공

6. 디바이스 드라이버 관리자 (입출력 관리자)

   ⇒ 각종 외부 IO (프린터, GPU 등)에 대한 접근을 제공

7. 정보 보안 관리자

    ⇒ 사용자 인증 및 실행권한 관리

※ 모노리틱 커널 : 프로세스 및 메모리 관리, 파일 시스템 등의 커널 주요 기능이, 커널 모드에서 실행되도록 구현된 구조 (운영체제 실행속도 ↑, 코드 난잡성 및 수정 불편,)

※ 마이크로 커널 : 커널 기능이 외부로 옮겨져 사용자 프로그램 형태로 수행, 커널 자체는 핵심 기본적인 기능만 남아서 크기가 작아짐 (OS 크기 ↓, 확장성 ↑, 개발지향적, IPC & 모드 체인지 → 속도 ↓)

2. 시스템 프로그램 (Shell)

⇒ 사용자가 컴퓨터에 전달하는 명령을 해석하는 프로그램 집합, 커널과 사용자간 가교 역할

⇒ User가 명령을 Shell을 통해 커널에게 전달하는 주체!

3. 인터페이스

⇒ 사용자에게 명령을 전달하고 결과를 사용자에게 알려주는 화면단

⇒ GUI : 윈도우 바탕화면, 아이콘 & IO 조작 = 사용자 편의성

⇒ CLI : 터미널 연결, 키보드 활용 인터페이스

4. 시스템 콜

⇒ OS가 컴퓨터 하드웨어를 다루기 위해 입출력, 메모리할당, 프로세스의 생성 등을 수행하는 코드의 집합이다.

⇒ 사용자나 프로그램이 직접적으로 컴퓨터 자원에 접근하는 것을 막고 커널을 보호하기 위해 사용

⇒ 위와 같은 이유로, 운영 체제는 시스템 콜 함수를 통해서만 커널에 접근 할 수 있다.

    (프로그램 ↔ 운영체제 인터페이스, 프로그램이 특정 서비스를 Sys call로 호출하는 방식!)

⇒ User Mode(CPU 명령어 사용 제한)과 Kernel Mode(CPU 명령어 활용, 하드웨어 제어) 사이는 System call을 통해 전환된다.

5. 인터럽트 & 트랩 & 시그널

⇒ 인터럽트 및 트랩 : 컴퓨터 시스템에서 이벤트 발생 시 커널에게 직접적으로 알리는 방법

⇒ 시그널 : 이벤트가 발생했을 때 프로세스에게 알리기 위해 사용하는 flag, 일반 프로세스도 이러한 이벤트를 받을 수 있게 하는 방법

⇒ 모든 신호는 커널을 통해 전달된다. 또한 프로세스는 시그널에 따라 4가지 동작을 처리한다.

- 수신한 프로세스가 죽는다.
- 수신한 프로세스가 잠시 멈춘다.
- 시그널을 단순히 받고, 추가적인 처리는 하지 않고 넘어간다.
- 사용자 정의 함수를 실행시키게 한다.

6. 드라이버

⇒ 프린터, 키보드 등의 IO 장치와 OS간의 통신을 가능케 하는 SW이다.

⇒ 컴퓨터 시스템과 이질적인 구조를 가진 하드웨어 (프린터)의 경우, 드라이버를 통해 컴퓨터 System과 소통을 할 수 있다.

7. 프로세서

⇒ 프로세서로의 입력 & 출력 bus, 인스트럭션 실행하는 ALU 연산 부분, 연산에 이용되는 정보가 잠시 저장되는 레지스터들, 그리고 이를 제어하는 control unit으로 구성된다.

⇒ 레지스터는 3가지 그룹으로 나뉜다. (인텔 CPU)

1. 범용 레지스터 (eax, ebx, ecx, edx) : 프로세스 수행 중 일반 연산에 주로 사용
2. 특수목적 레지스터 : 다음 수행할 인스트럭션 주소 가리키거나, 현재 스택 사용 위치 등
3. 세그먼트를 가리키는 레지스터 : 프로세스 스위치 시, 교체될 PCB 안에 저장됨

---

2. 부팅, 컴퓨터 시스템, 사용자 및 커널 모드

1. 부팅

⇒ 컴퓨터 전원을 넣은 후, 사람이 컴퓨터를 사용할 수 있도록 준비하는 과정

⇒ 운영체제는 메모리에도 있고 보조기억장치에도 존재한다. 보조기억장치에 운영체제의 이미지로 존재하며, 하드디스크의 파일 형태로써 저장되어 있다. (다른 프로그램도 이와 동일하다.)

⇒ OS SW가 하드 디스크와 같은 보조 기억장치에 저장되며, 여기에 저장된 OS를 읽어 메인 메모리로 옮기고 실행시키는 작업을 의미한다.

2. 프로그래밍 방식 (1개의 프로세서에 대해)

1. **유니프로그래밍**

   ⇒ 한번에 프로그램 하나만을 실행, 하나가 완전히 끝날 때 까지 계속 기다리는 방식

2. **멀티프로그래밍 (컨커런트 프로그래밍)**

   ⇒ 프로세서가 특정 프로그램 입출력을 기다리는 동안, 다른 프로그램을 실행하는 방식

   ⇒ 상대적으로 오래 실행해야 하는 프로그램들을 여러 개 모아 차례로 실행하는 데 적합

   ⇒ 실시간 상호작용에 필요한 CPU 활용도가 떨어질 때, 블록 큐에 들어가 있는 그러한 프로그램을 꺼내 실행하는데, 이러한 실행 방식을 배치 프로세싱이라 한다.

3. **타임 쉐어링**

   ⇒ 멀티프로그래밍 + 타임 슬라이스 or 타임 퀀텀 방식

   ⇒ 타임 슬라이스는 일정 시간만큼 각 프로그램을 번갈아 실행시키는 방법 (기본 0.1 sec)

   ⇒ 초당 거의 10개 프로세스들이 실행, 모든 프로그램들이 마치 각각 CPU를 점유하게 있는 듯한 착시효과를 제공!

   ⇒ 마치 대화형으로 작업하는 프로그램에 적합, 반응시간이 짧음! 하지만 잦은 컨텍스트 스위칭으로 인해 운영체제의 부담이 가중된다.

3. 유저 모드 vs 커널 모드

1. 유저 모드

   ⇒ 유저 환경 상에서 실제 상호작용함으로써 동작되는 모드, 우리가 평소 프로그램 조작 시 활용

2. 커널 모드

   ⇒ 모든 컴퓨팅 자원에 접근 가능, 컴퓨터의 모든 동작을 가능케 하며, 유저 접근은 제한됨.

3. 프로세스

1. 프로세스

⇒ 컴퓨터 환경 상에서 실행 중인 프로그램, 다음에 실행할 명령어를 지정하는 프로그램 카운터와 관련 자원 집합을 가진, 메모리에 올려져 특정 기능을 수행하는 객체

⇒ 프로그램은 명령어 리스트 등을 가진 파일 그 자체를 의미한다.

⇒ 실행파일 → 메모리에 적재 → 실행 → Program → Process

※ 프로세스 생성법

⇒ 포그라운드 모드 : 즉시 프로세스 실행, 대화형 형식으로 소통하는 인터렉티브 모드

⇒ 백그라운드 모드 : “&” + a.exe, 나중에 프로세스 실행, 실행시간을 의도적으로 늦춘다.

(단, 미룬 프로세스들을 CPU가 여유가 있을 때 한꺼번에 처리하므로, 배치 모드라고도 한다.)

⇒ 데몬 프로세스 : 덜 중요하거나 주기적인 작업은 커널이 서비스 프로세스를 만들어 작업 위임

⇒ 기존 프로세스가 fork를 통해, 다른 프로세스를 생성한다. (위 사례도 이를 통해 생성한 것)

⇒ **프로세스 컨텍스트** : 프로세스가 실행되는데 필요한 컴퓨터 내 정보 및 구성요소들의 집합

(프로세스를 구성하는 코드와 데이터, 스택 등)

1. 유저 컨텍스트 : 프로그램 작성자에 의해 결정되는, 프로그램 코드, 전역변수, 사용자 코드 내의 함수, 지역변수 및 유저 스택 등
2. 시스템 컨텍스트 : 운영체제가 사용자 프로세스 실행시키기 위해 내부적으로 생성한 것, kernel code 함수 호출 시 필요한 임시 정보를 저장하는 커널 스택 및 프로세스 정보 저장하는 PCB

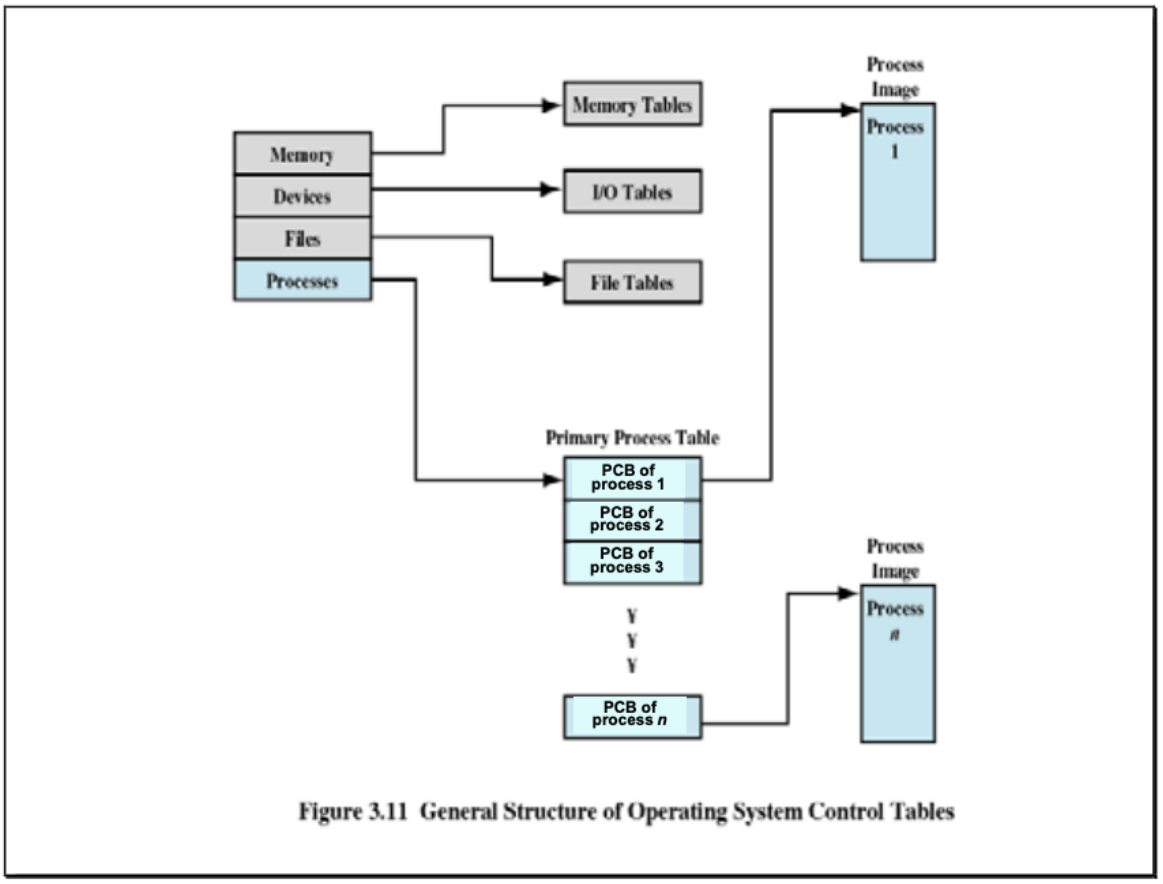


Figure 3.11 General Structure of Operating System Control Tables

PCB : 프로세스 정보를 담는 자료구조, Pid & 프로그램 카운터, (부모 프로세스) 파생된 기타 정보

⇒ 특정 프로세스가 생성되면, 기존 PCB 링크드 리스트에 해당 PCB를 삽입한다. (컨텍스트를 연결 하는게 아니라, 각 프로세스들의 PCB 연결리스트!)

⇒ PCB는 커널 메모리 안에 있으며, 운영체제가 생성한다. 운영체제는 컨텍스트 대신, PCB를 활용!

※ 컨텍스트 스위치

⇒ 실행 중인 프로세스에서, 다른 프로세스를 실행하기 위해 프로세스 실행환경을 변경하는 것

- 현재 프로세스의 상태를 PCB에 저장 (중단되는 당시 CPU 레지스터 값 등)
- 그 직전 타임 슬라이스에서 CPU 등의 자원을 얼마나 사용했는지 업데이트
- 중단되는 프로세스 상태를 준비 또는 대기 상태로 PCB 상태값을 바꿈으로써 전환
- 알맞은 상태 큐로 해당 프로세스 PCB를 옮긴 후, 스케줄러에 따라 다음 프로세스 선택

< 스위치가 발생하는 조건 >

1. 실행하던 프로세스가 끝까지 다 실행하여 종료
2. 실행 프로세스가 입출력을 요청하는 것처럼, 대기하는 상황으로 들어갈 때
3. 타임쉐어링 시스템에서, 타임 슬라이스가 경과할 때
4. 인터럽트 발생하여, 이를 처리하고 난 후, 현재 프로세스보다 우선순위가 더 높은 프로세스가 wait 상태에서 깨어나 실행되어야 할 때

※ 시스템 콜과 프로세스 흐름도

1. Fork() : 자식 프로세스 생성, 부모의 PCB 및 CPU 레지스터 상태값을 복사하여 가져온다.
2. Block()
3. Sleep()
4. Wait() :
5. Clone() : 새로운 쓰레드를 생성하는 시스템 콜, 이후 P의 코드 & 데이터를 공유시킨다.
6. Exit() : 종료 시스템 콜
    a. 메모리 부족 / 정보보호 / 산술에러 ⇒ 강제 호출 & 또는 사용자 의도적 호출
    b. (부모 요청 등) 선택 프로세스에 따라, 커널이 PCB 제외 프로세스 할당된 자원 회수
    c. 부모에게 death-of-child 시그널 → 종료할 프로세스의 PCB 상태값 종료 상태로 변경
    d. 부모가 이 시그널을 받으면, wait() 시스템 콜을 호출, 커널은 PCB를 지운다.

⇒ 프로세스 실행 도중, fork() 시스템 콜이 호출된다면 자식 프로세스가 부모 프로세스의 컨텍스트와 동일한 컨텍스트를 가지고 생성된다. (단, PID 등으로 구분, idle 프로세스 pid = 0 로 미리 선점 중)

⇒ 그래서 부모 프로세스가 sleep 등으로 대기하고, 자식이 처리될 때, 부모가 진행한 프로그램 cnt의 바로 다음 문장부터 실행한다. (Fork() 된 시점의 부모의 실행 상태 역시 가져오기 때문에)

※ 자식이 Fork() 되었을 때, 부모와 자식 프로세스 중 누가 먼저 끝날까요?

⇒ 모른다. 스케줄러가 결정하는 종료 시점은 정확히 누가 먼저 끝날지 예측할 수 없다.

※ 멀티 쓰레드 환경에서는, 부모와 자식 중 누가 먼저 끝날까요?

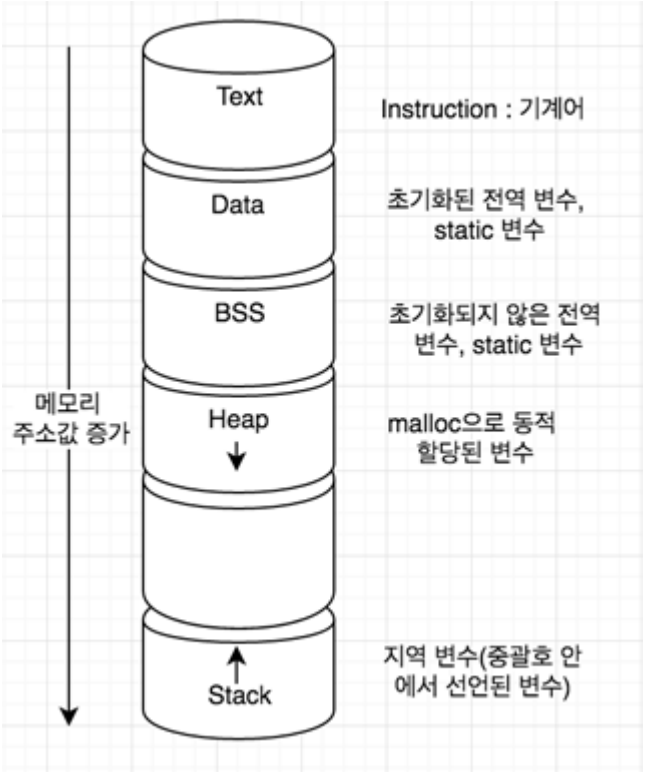
⇒ 멀티 쓰레드 환경에서는, 부모가 "메인 컨텍스트" 역할을 하기 때문에 모든 자식 쓰레드가 끝난 뒤에 부모 쓰레드까지 종료된다. 즉, 부모가 가장 마지막에 끝난다.

⇒ 이러한 특징 때문에, 부모가 도중에 죽는다면, 파생된 모든 쓰레드 역시 모두 터지게 된다.

※ 종료하는 프로세스가 자식 프로세스보다 먼저 끝나면 어떻게 되는가요?

⇒ 부모가 자식 프로세스를 성공적으로 처리할 wait() 호출할 수 없어서, PCB가 영원히 존재!

⇒ init 이라는 프로세스로 자식 프로세스를 따로 보냄으로 관리하도록 한다.



⇒ 프로세스는 각각 독립된 메모리 영역 (Stack, Heap, Data, Code)를 가진다.

- 스택 = (함수 인자 값, 지역 변수, 호출 당시 프로세서 레지스터 값) 저장
- 스택에는 커널 스택과 사용자 스택 2가지로 구분, 저장되는 정보 및 용도는 동일함

⇒ 프로세스 당 최소 1개의 쓰레드를 가지고 있다.

⇒ 각 프로세스는 별도의 주소 공간을 가지며, 다른 프로세스의 영역에 침범할 수 없다. 이 주소공간은 4GB의 가상 주소공간으로 불린다.

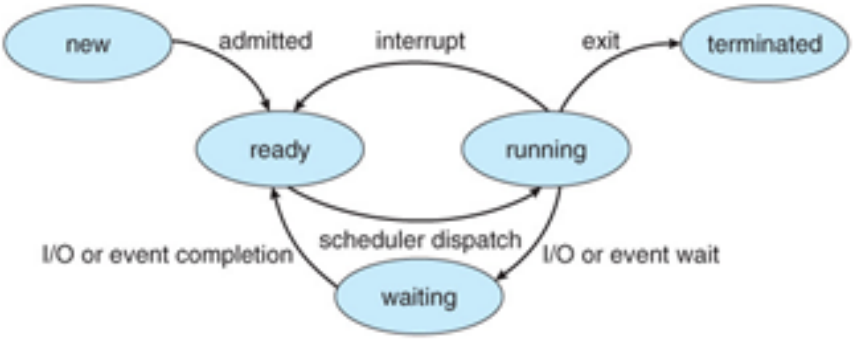
- 0G ~ 3G = 사용자 프로그램, 나머지 3G ~ 4G = 커널이 들어있는 시스템 컨텍스트
- 0G ~ 3G = 유저 모드에서 실행, 3G ~ 4G = 시스템 콜 함수 호출 시, 커널 모드로 변경해 실행
- 커널 모드로 모드 체인지 시, 사용자 프로세스는 커널로 변신해 OS 역할을 수행!

⇒ (데이터, 스택) = 보조 기억장치, (코드) = 별도로 디스크 할당 X, (커널) = 메인 메모리에서 매핑

- 가상주소공간에서 각 영역의 시작~끝 주소값을 PCB에 저장, 마치 디스크에 존재하도록 함
- 실제로 디스크에 장소를 차지하고 있는 건 코드, 데이터, 스택 / 커널은 메인 메모리에 상주
- 각 프로세스의 3~4G 영역에, 실제 존재하는 커널 1개를 매핑해서 사용하는 것!

⇒ 접근하기 위해선 IPC(Inter-Process Communication)를 활용한다.

❋ Process State



- **new**: 프로세스가 생성 중이다
- **ready**: 프로세스가 프로세서에 할당되기를 기다리고 있는 상태. 프로세스는 ready 큐에서 실행을 준비하고 있다.
- **running**: ready 큐에 있는 프로세스가 프로세서에 할당받아 실행되는 상태

- **waiting**: 프로세스에 입출력 처리가 필요하면 현재 실행 중인 프로세스가 중단되고, 입출력 처리가 완료될 때까지 대기하고 있는 상태
- **terminated**: 프로세스의 실행이 끝나고 프로세스 할당이 해지된 상태
- **suspended** : 실행 중 메모리 부족 등의 이유로, 프로세스가 잠시 보조기억장치에 옮겨진 상태
  - swap out : 메인메모리 → 보조기억장치, (block → suspended block state)
  - swap in : 보조기억장치 → 메인메모리, (ex) suspended ready state → ready state)

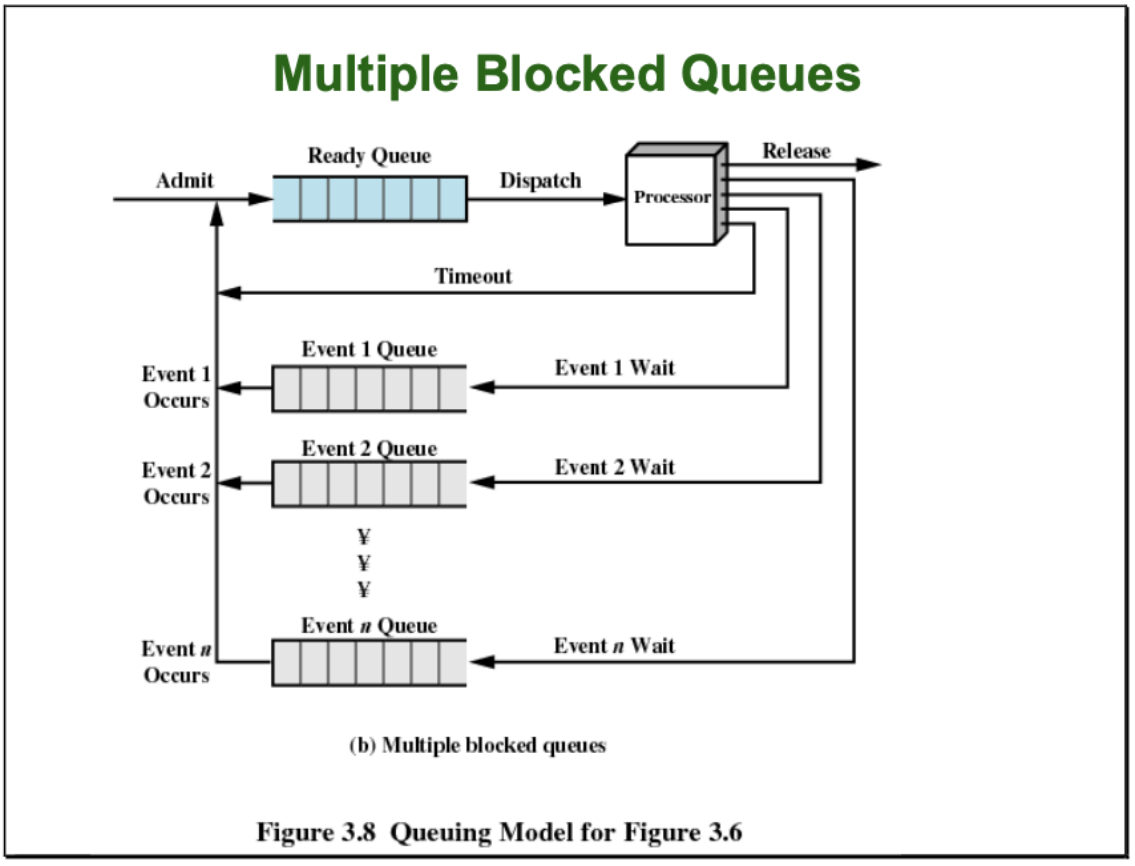


Figure 3.8 Queuing Model for Figure 3.6

⇒ 이벤트에 따라 우선순위 등이 구분되며, 각각의 응답 대기 큐가 존재한다.

※ IPC (Inter-Process Communication)

⇒ 프로세스 간 통신, 커널 영역에서 IPC를 사용하여 프로세스들 간에 통신을 제공한다.

1. **메시지 패싱**

⇒ 프로세스가 **커널을 통해** 메시지가 저장되는 메시지 큐를 만들고, 메시지를 받을 프로세스를 fork() 하여 만든 후, 그 프로세스가 부모가 만든 메시지 큐의 식별자 정보를 받아 공유하게 된다.

⇒ 부모가 자식에게 보내고자 하는 메시지를 만들어, 시스템 콜을 통해 커널에게 보내면, 커널은 메시지 큐 식별자를 통해 큐에 넣는다. (유저 모드 → 커널 모드, send() sys call)

⇒ 자식이 실행되면, 메시지를 확인 후, 받았다는 수신 시스템 콜을 보낸다. 그리고 커널로 바뀌어 이를 처리한 후 다시 유저 모드로 바꾼다. (recevice() sys call)

⇒ 공유 메모리보단 사용하기 쉬우나, 빈번한 모드체인지로 인한 시간적 비효율성이 있다.

2. **쉐어드 메모리**

⇒ 두 프로세스가 같은 메모리 영역을 공유하기 위해, 커널에 시스템 콜을 통해 요청을 한다.

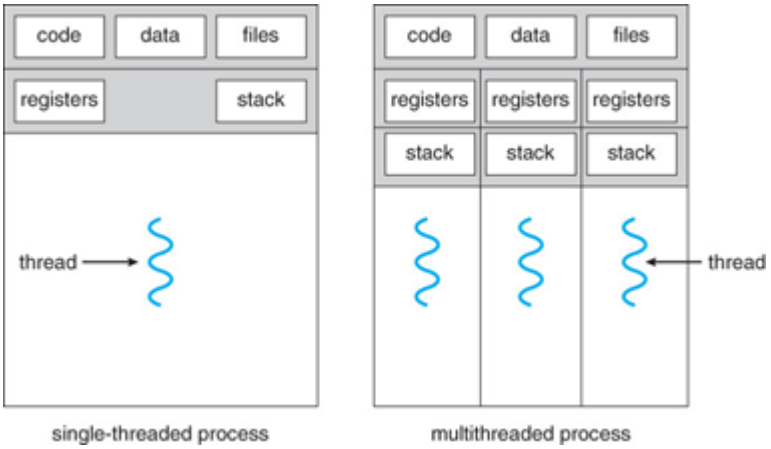
⇒ 쉐어드 메모리로 가는 것은 각 프로세스의 가상주소공간의 유저 영역에 할당된다.

⇒ 단순히 유저 메모리 영역에 읽고 씀으로써, 메시지 패싱보다 더 빠른 처리가 가능하다.

⇒ 단, 프로세스 동기화 해결방법이 선행화 되어야만 한다.

---

4. 쓰레드



⇒ 프로세스의 실행 단위, 프로세스 내에서 실제로 작업을 수행하는 주체 또는 실행 흐름 단위

⇒ 실행 단위는 인스트럭션의 묶음이며, 프로세스가 작업들을 수행하는 과정의 집합이다.

⇒ 프로세스가 실행된다는 의미는, 프로세스 내부의 쓰레드가 구동된다는 의미이다.

⇒ 쓰레드 : 스택 + CPU 레지스터 값들을 저장하기 위한 메모리 공간

1. **커널 레벨 쓰레드** : 쓰레드 생성 및 관리를 커널이 하는 경우
    - 유저 레벨에 비교해, 동시 수행성이 보장된다.
2. **유저 레벨 쓰레드** : 커널 밖 유저 모드에서 돌아가는 쓰레드 라이브러리가 하는 경우
    - 생성, 관리 실행 속도가 유저 레벨이 더 빠르다. (커널 모드 체인지 X)
    - 하지만, 커널 관점에선 그냥 프로세스 하나에 속한 단일 쓰레드로 간주한다.
    - 유저 레벨 쓰레드 중 하나가 입력 요청하면, 커널은 해당 프로세스를 대기상태로 보낸다.
    - 그럼 다른 모든 쓰레드도 대기상태로 처리, 결국 여러 쓰레드 만드는 의미가 없다.
    - 운영체제에 연연하지 않고 독립적인 실행환경을 제공한다.
    - 스케줄링을 유저가 제어, 커널 방식을 따르지 안흔 자유로움이 장점!

⇒ 2개 이상의 쓰레드를 가진 프로세스를 멀티 쓰레드 프로세스라고 한다.

⇒ 멀티쓰레드 : 커널이 프로세스를 쓰레드가 실행될 수 있는 환경을 제공하는 틀로 간주한다.

⇒ 하나의 프로세스 안 쓰레드들은 각자의 스택 영역만을 할당받고, 나머지 영역은 공유한다.

⇒ (프로세스용 코드, 데이터, 파일 공유) + (스택, 레지스터 저장 메모리는 각자) = 자원 절약!

장점 : **프로세스보다 생성할 때 오버헤드도 적고 공유 자원에 대한 오버헤드도 적다**.

⇒ PCB을 공유함으로써, 자원 관리 효율 및 Context Switching에 들어가는 비용이 줄어든다.

⇒ IPC로 인한 통신 비용을 쓰레드를 활용함으로써 줄일 수 있다. (데이터 영역만 활용)

⇒ 각 프로세스마다 생성되는 PCB 비용보다, TCB 비용이 비교적 싸다. (스택만 따로 분리!)

※ 프로세스와 스레드의 차이?

프로세스 : 실행 중인 프로그램, 다른 프로세스와 상관없이 독립적으로 메모리 공간을 할당

쓰레드 : 일종의 경량화된 프로세스로 프로세스 안의 제어 흐름, 프로세스의 수행 과정

※ 각 기능을 함수로 호출하기 보다, 왜 쓰레드에 전담기능을 독립시켜 사용하는지?

⇒ 입력이 올 때 까지 프로세스는 대기상태로 바뀌면서 cpu를 뺏긴다.

⇒ 함수로 구현한 경우, 이때까지 다양한 전담기능 함수를 실행할 수 없다.

2. 멀티 프로세싱 vs 멀티 쓰레드

1. 멀티 프로세싱 (병렬 프로그래밍, 다른 프로세스가 동시 실행)

   ⇒ 프로세서가 여러 개, 프로그램들을 분담해 처리하며, 각 프로세서는 멀티프로그래밍으로 동작

   ⇒ 각 프로세서들이 각자 실행할 명령어를 가지고 오기 때문에, 같은 메모리를 점유하거나 특정 자원 접근 충돌 문제를 해결하기 위해 “프로세스 동기화” 기술 활용!

2. 멀티 쓰레드

   ⇒ 하나의 프로세스에 여러 쓰레드를 할당하여, 스케줄러가 스케줄링을 통해 처리하는 방법

   ⇒ 다중 프로세스에 비해 메모리 자원소모가 적으며, IPC 필요없기 때문에 통신속도도 빠름!

   ⇒ 스레드 Context Switching은 프로세스 Context Switching보다 빠름

   ⇒ 단, 힙 영역에서는 자원 경쟁 상태 → 동기화 요구 & 락 과도하게 사용 시 성능 저하

   ⇒ 하나의 쓰레드에서 에러가 발생 시, 모든 쓰레드가 멈출 우려가 있다.

※ 즉, 멀티 프로세싱을 통해, 멀티 프로그래밍 및 쓰레드 작업을 훨씬 더 빠르게 처리할 수 있다!