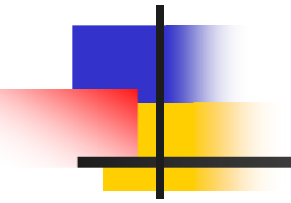
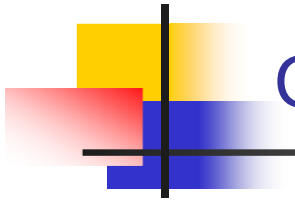


Operating Systems
(Processes)

Chapter 3

These lecture materials are modified from the lecture notes written by A. Silberschatz, P. Galvin and G. Gagne.

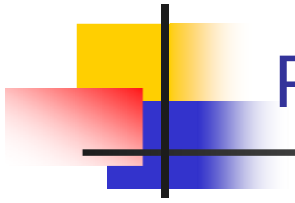
August, 2022



Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Communication in Client–Server Systems

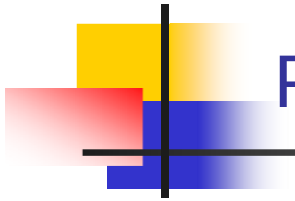
Process Concept



Program / Process

- 프로그램은 프로세스가 아님!

- 프로그램:
  ‘디스크에 저장된 명령 목록(실행 파일이라고 불림)을 포함하는 파일’과 같은 수동 개체(passive entity)
- 프로세스:
  ‘실행할 다음 명령 및 관련 리소스 집합을 지정하는 프로그램 카운터를 가지는’ 활성 개체(active entity)



Program / Process

- 프로그램은 실행 파일이 메모리에 로드될 때 프로세스가 됨
  - 실행 파일을 메모리에 로드하려면 실행 파일을 실행하면 됨
- 프로세스의 범위 : 메모리 구조 + Register set

프로세스의 Register set

- Register set :
  CPU가 특정 프로그램을 실행 중이라면

  프로세스가 Running 상태면

  CPU 내부 레지스터들은 프로그램 실행을 위한 데이터들로 채워짐

- 그러므로 레지스터의 상태도 프로세스의 일부로 포함시켜야 함

프로세스의 메모리 구조

- 프로세스의 메모리 구조는
  일반적으로 여러 섹션으로 나뉨
  - Text section— 실행 코드
  - Data section— 전역 변수
  - Heap section— 프로그램 실행 중에
    동적으로 할당되는 메모리
  - Stack section— 함수 호출 시
    임시 데이터 저장
    (예: function parameters,
    return addresses, and local variables)

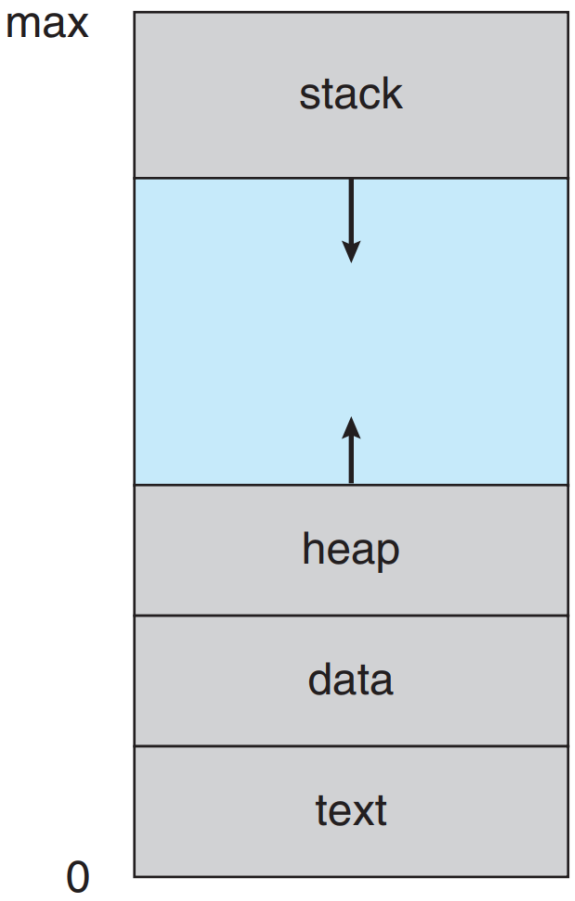


Figure 3.1 Layout of a process in memory.

프로세스의 메모리 구조

- 텍스트 및 데이터 섹션의 크기는 프로그램 실행 시간 동안 변경되지 않기 때문에 고정되어 있음
- 스택 및 힙 섹션은 프로그램 실행 중에 동적으로 축소 및 확장될 수 있음

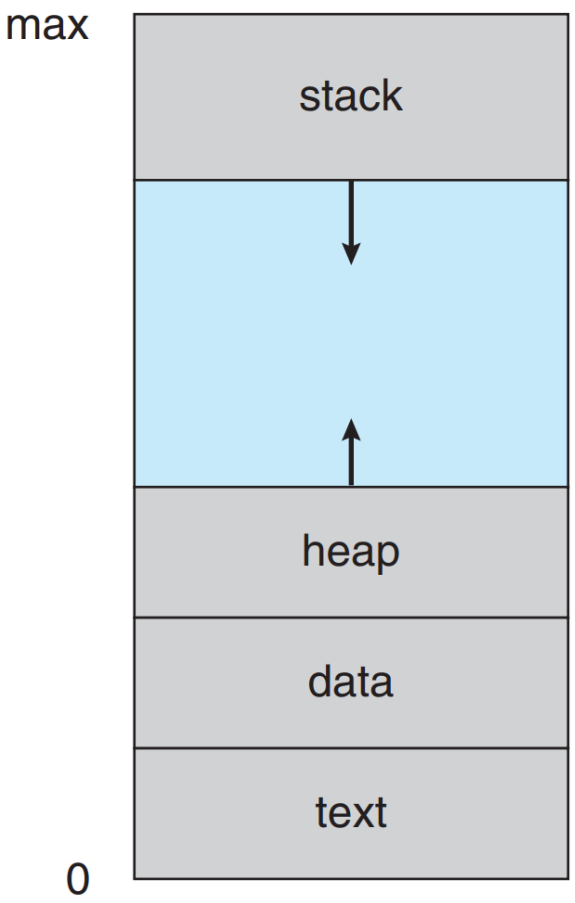


Figure 3.1 Layout of a process in memory.

프로세스의 메모리 구조

- 함수가 호출될 때마다 activation record가 스택에 push 됨
  - 함수에서 제어가 반환되면 activation record가 스택에서 빠짐.
  - activation record엔 function parameters, return addresses, local variable가 포함
- 힙은 메모리가 동적으로 할당됨에 따라 커짐
  - 메모리가 시스템에 반환되면 축소됨
- OS는 스택과 힙 섹션이 커지더라도 겹치지 않도록 해야함.

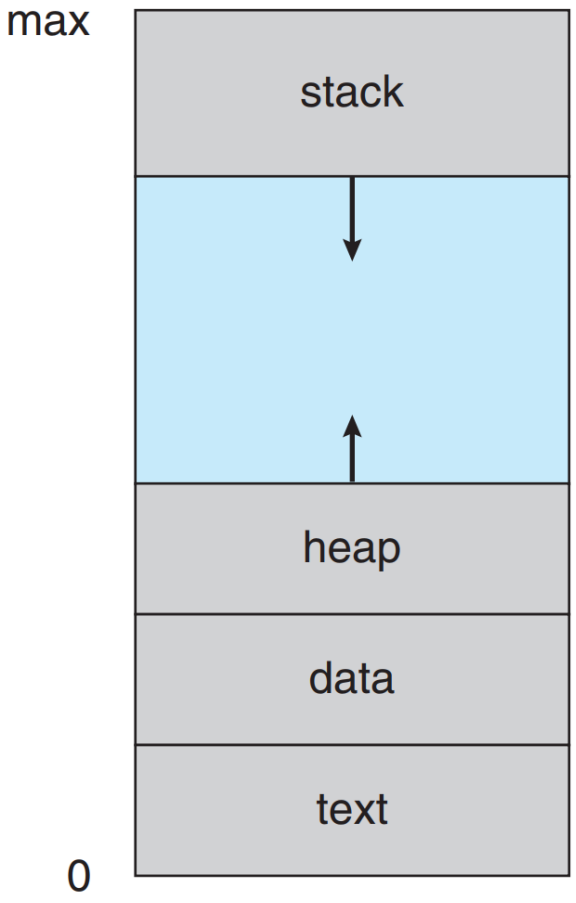
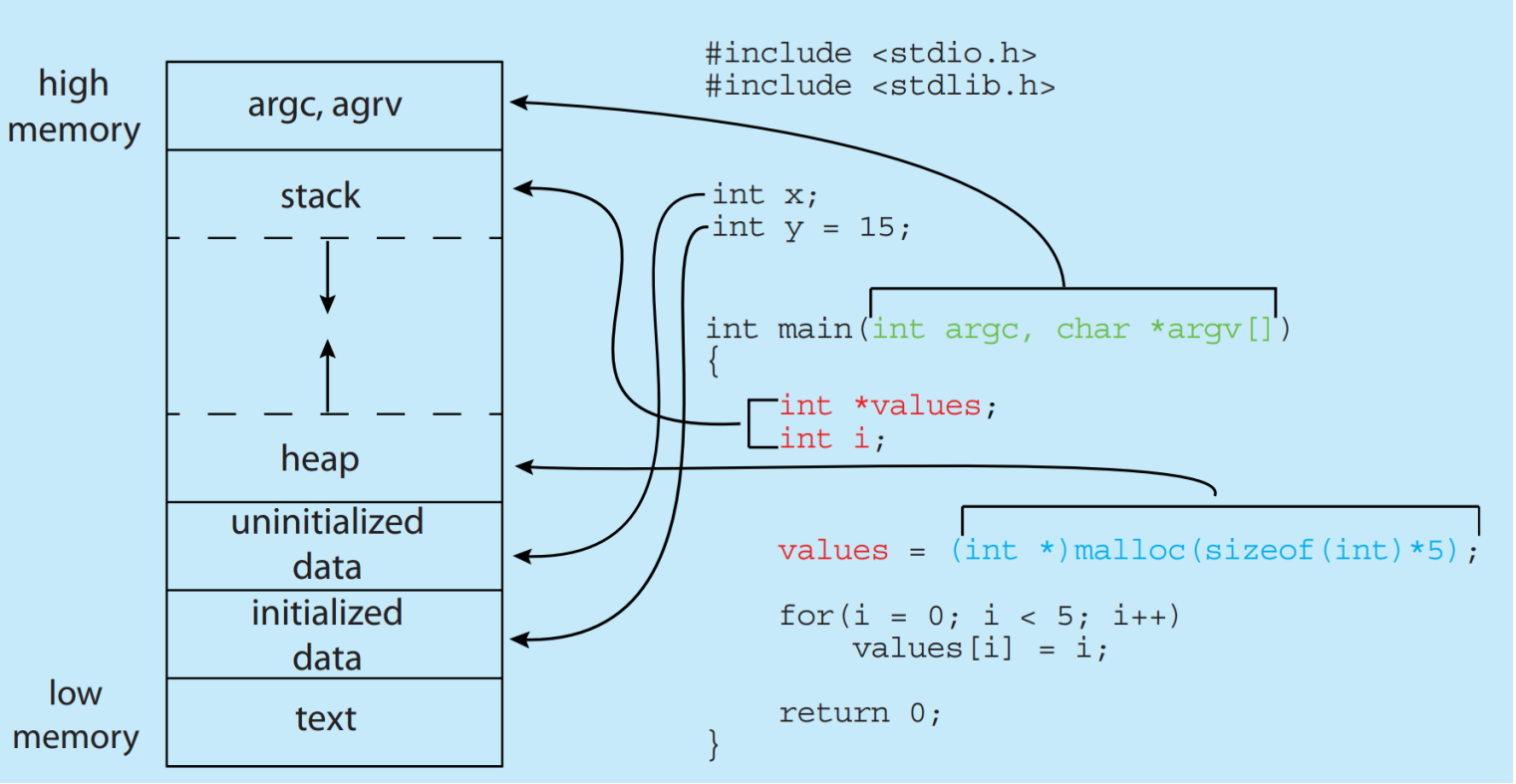
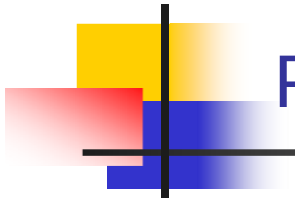


Figure 3.1 Layout of a process in memory.

프로세스의 메모리 구조



- high memory는 사용자 공간 -> 응용 프로그램을 위한 공간
- low memory는 커널 메모리 -> OS를 위한 공간



Process State

- 모든 코어에서는 한 번에 하나의 프로세스만 실행할 수 있음.
  - 그러나 많은 프로세스가 준비되어 대기 중일 수도 있음.
  - 프로세스가 실행되면 state가 변경됨
- 프로세스의 state는 해당 프로세스의 현재 activity 에 의해 정의됨.
- OS에 따라 state의 이름과 state의 개수가 다를 수 있음

Process State

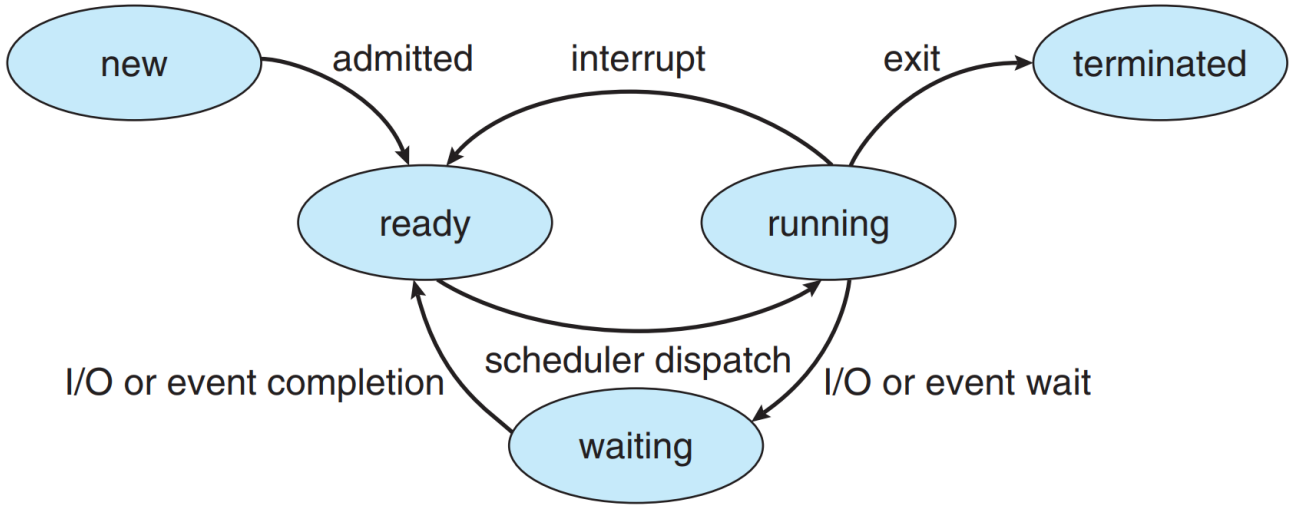


Figure 3.2 Diagram of process state.

- New : 프로세스가 생성되는 중
- Running : 명령어가 실행되고 있음
- Waiting : 프로세스가 어떤 이벤트가 발생하기를 기다리는 중
  (예: I/O 완료 또는 signal 수신).
- Ready: 프로세스가 프로세서에 할당되기를 기다리는 중
- Terminated: 프로세스가 실행을 완료하고 종료됨

Process Control Block (PCB)

- OS에서 특정 프로세스의 정보를 담고있는 블록.
  Task Control Block이라고도 함.

- PCB는 일부 accounting data와 함께 프로세스를 시작하거나 다시 시작하는 데 필요한 모든 데이터를 저장함.

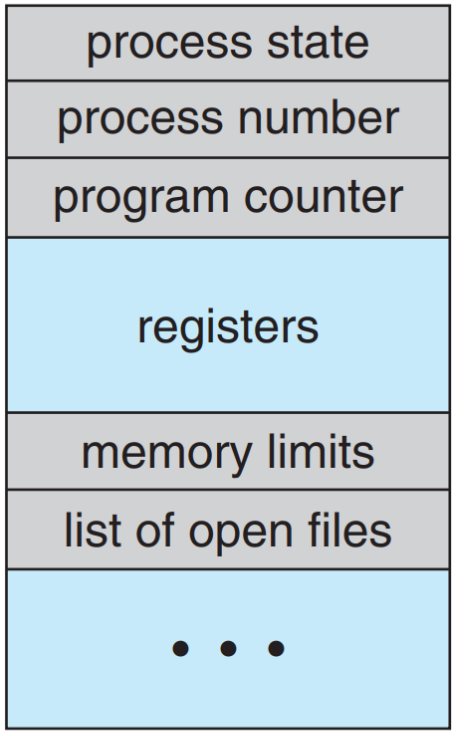


Figure 3.3 Process control block (PCB).

Process Control Block (PCB)

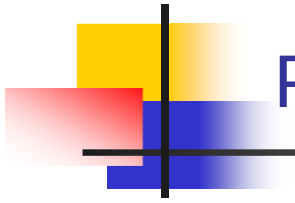
- **Process state:** 프로세스의 상태(앞에서 설명)
- **Process number:** PID. 프로세스 식별자
- **Program counter :** CPU 내부에 있는 레지스터 중 하나로서, 다음에 실행될 명령어의 주소를 가짐.
- **CPU 레지스터:** OS에 따라 사용처가 다르지만, 보통 인터럽트 이후 프로세스가 기존에 하던 일을 올바르게 계속될 수 있도록 하는 stack pointer로 사용됨
- **메모리 관리 정보:** base, limit 레지스터, 페이지 테이블 또는 세그먼트 테이블의 값
- **I/O status 정보:** 프로세스에 할당된 I/O 장치 목록, 열린 파일 목록 등
- **Accounting 정보:** 사용된 CPU 및 실시간 시간, 시간 제한, task 또는 프로세스의 개수 등
- **CPU 스케줄링 정보:** 프로세스 우선순위, 스케줄링 큐 포인터 등 스케줄링 파라미터 정보

Process Scheduling

Multi-programming / Multitasking

- **멀티 프로그래밍**: CPU 활용도를 최대화하기 위해
  하나의 프로세서가 하나의 프로세스를 수행하는 동안
  다른 프로세스에 접근할 수 있도록 하는 방법

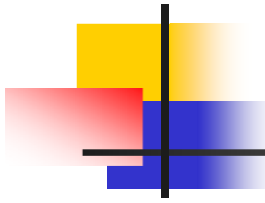
- **멀티 태스킹**: 프로세스 사이에 CPU 코어를 자주 전환하여
  사용자가 실행 중인 각 프로그램과
  상호작용할 수 있도록 하는 것



Process Scheduling

- 프로세스 스케줄러는 코어에서 사용 가능한 프로세스를 선택함.
- 각 코어는 한 번에 하나의 프로세스를 실행할 수 있음.

- **프로세스 스케줄링 :**
  어떤 프로세스를 어떤 코어에 할당할 것인가 결정하는 것



Process Scheduling - Scheduling Queues

- Ready queue : CPU 할당을 대기중인 프로세스 중
  실행될 준비가 된 프로세스를 위한 큐.
- Wait queue : CPU 코어에서 실행되는 프로세스가
  특정 이벤트가 발생할 때까지 기다릴 때 들어가는 큐.
  (I/O 요청 등)
- Job queue: 메모리 할당을 대기중인 프로세스를 위한 큐.
- Device queue: I/O 장치 할당을 대기중인 프로세스를 위한 큐.

Process Scheduling

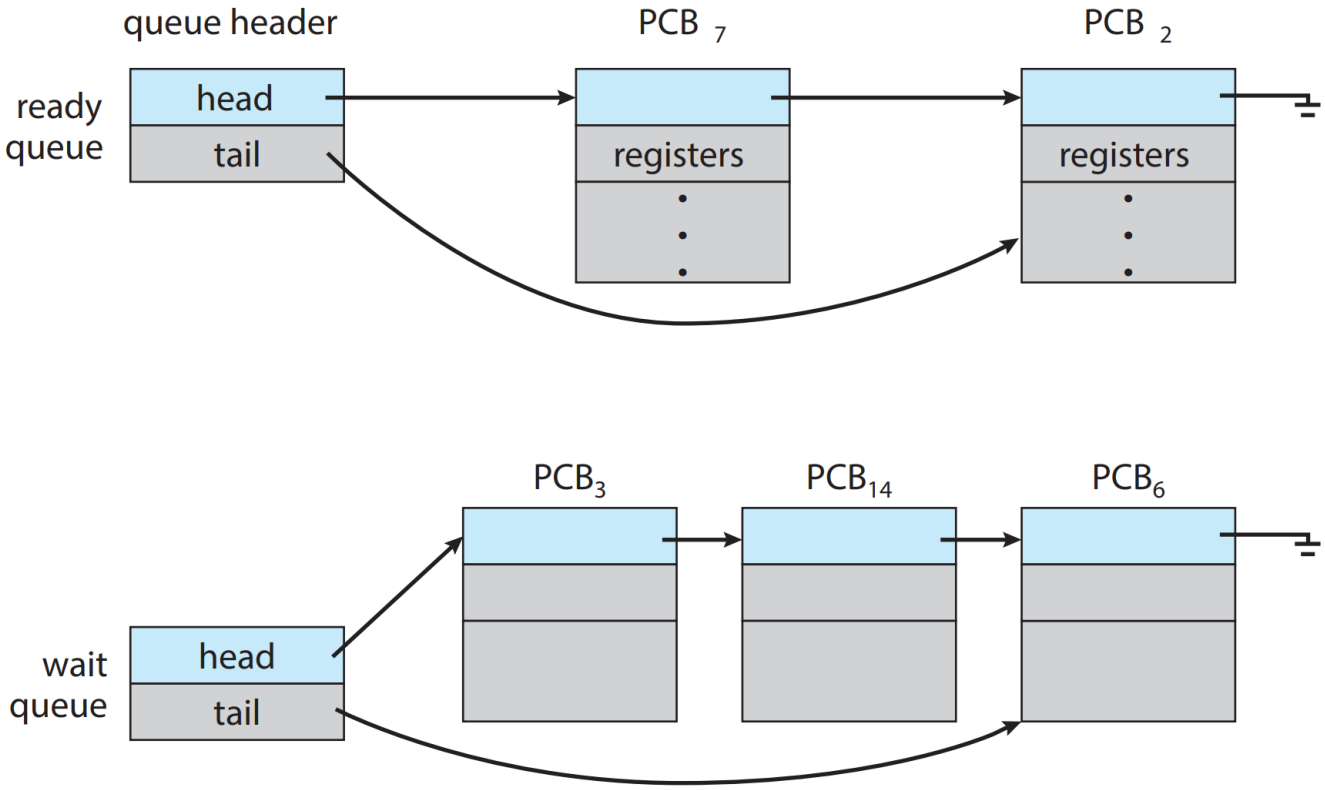


Figure 3.4 The ready queue and wait queues.

- 각 queue header에는 목록의 첫 번째 PCB에 대한 포인터가 포함
- 각 PCB에는 준비 대기열의 다음 PCB를 가리키는 포인터 필드가 포함

Process Scheduling

- (1) 새 프로세스는 처음에 ready queue에 들어감.
  실행을 위해 선택되거나 디스패치될 때까지 대기함.

- (2) 프로세스에 CPU 코어가 할당되고 실행됨.

- (3) 프로세스에 CPU 코어가 할당되고 실행되면
  아래 이벤트 중 하나가 발생할 수 있음.
  - 프로세스는 I/O 요청을 한다음, I/O wait queue에 배치될 수 있음
  - 프로세스는 새 자식 프로세스를 만든 다음, 자식의 종료를 기다리는 동안 wait queue에 배치될 수 있음.
  - 프로세스는 인터럽트 또는 제한 시간 만료로 인해 코어에서 강제로 제거되고 wait queue에 다시 배치될 수 있음.

Process Scheduling

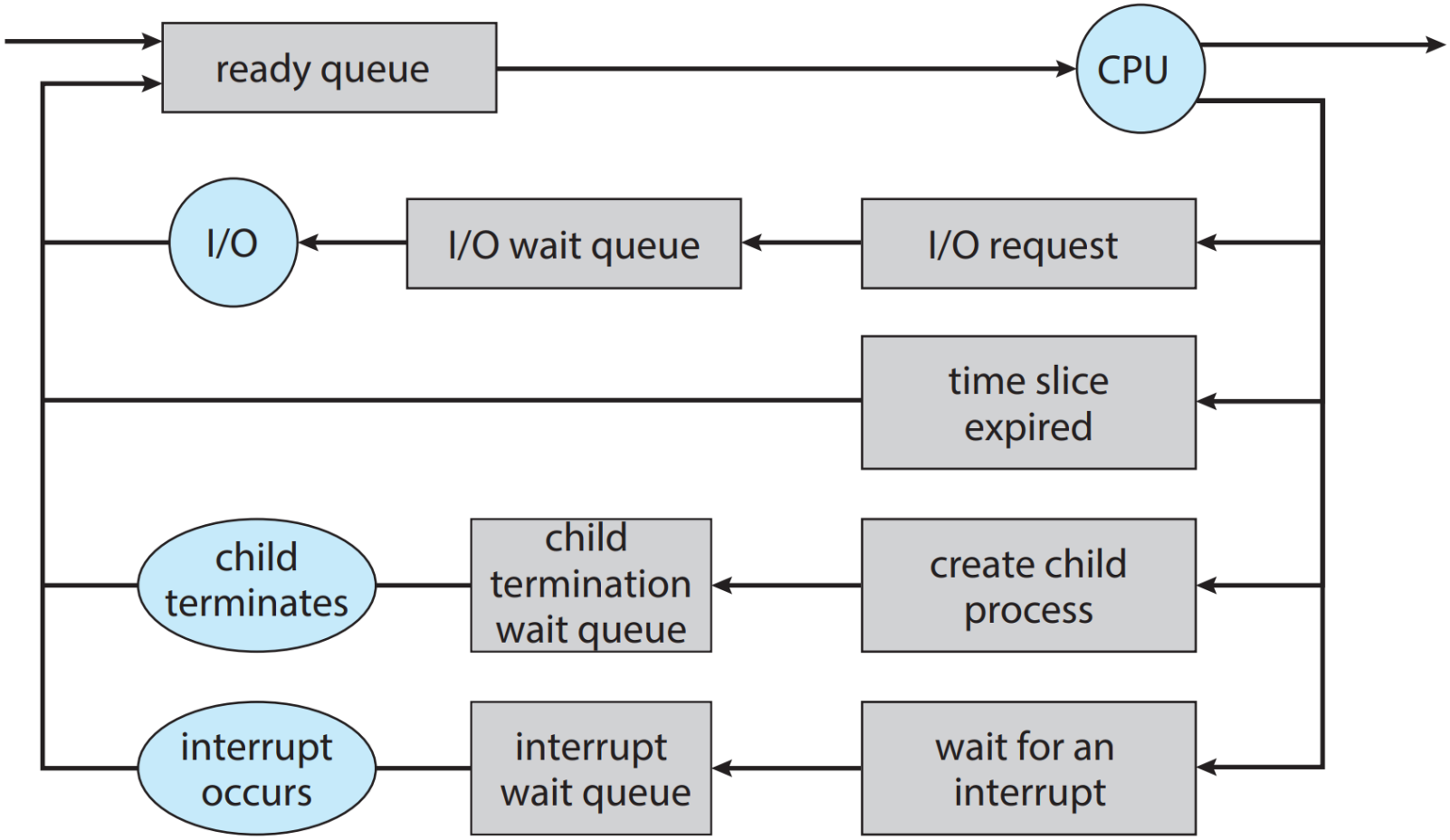
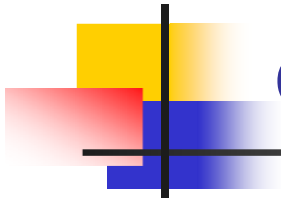


Figure 3.5 Queueing-diagram representation of process scheduling.

Process Scheduling

- (4) 프로세스는 종료될 때까지 이 사이클을 반복함.

- (5) 프로세스가 종료될 때 모든 queue에서 제거되고,
  PCB 및 리소스 할당이 해제됨



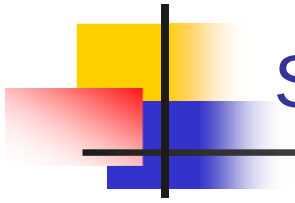
CPU Scheduler

- 프로세스는 수명 동안
  ready queue와 다양한 wait queue들 사이를 이동함

- CPU 스케줄러는 wait queue 에 있는 프로세스 중에서
  프로세스를 하나 선택하고, 그 프로세스에 코어를 할당함

CPU Scheduler 관점에서의 Process 구분

- I/O-bound process:
  - 계산보다 I/O를 수행하는 데 더 많은 시간을 소비함.
  - 많고 짧은 CPU bursts를 가짐. 즉 I/O 요청을 기다리기 전에 몇 밀리초 동안만 CPU에서 실행됨
- CPU-bound process:
  - 계산에 더 많은 시간을 할애함.
  - 매우 긴 CPU bursts가 거의 없음. 즉 더 긴 기간 동안 CPU 코어를 필요로 함.
  - 그러나 스케줄러는 장기간 동안 코어를 한 프로세스에 부여하려고 하지 않을 것이므로, 이에 대한 핸들링이 필요함.
- 스케줄러는 CPU-bound process보다 I/O-bound process에 더 높은 우선 순위를 부여하는 경향이 있음.



Swapping

- 메모리나 CPU에 대해 경합하는 프로세스의 수가 증가할 수록 오버헤드가 늘어나 성능이 하락할 수 있음.
- 이 중 몇몇 프로세스를 제거하여(swapped out) multi-programming의 정도를 줄이는 것.
- 나중에 프로세스를 메모리에 다시 도입할 수 있으며 중단된 위치에서 계속 실행할 수 있음.

Context Switch

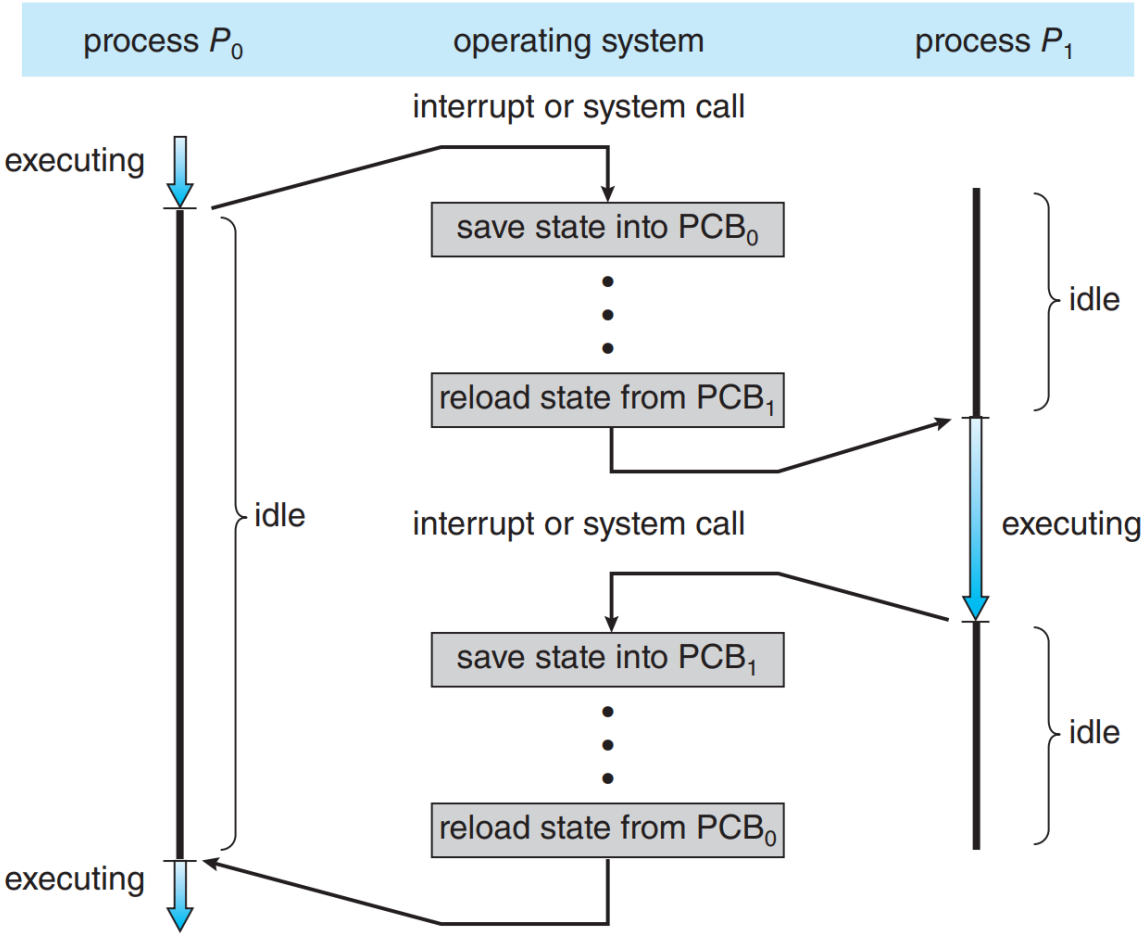
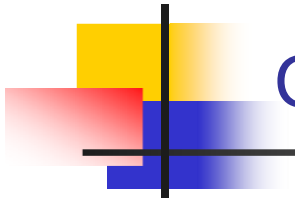


Figure 3.6 Diagram showing context switch from process to process.

- 인터럽트가 발생하여 코어를 다른 프로세스로 전환할 때,
현재 프로세스의 상태를 저장하고,
다른 프로세스의 상태 복원을 수행하는 것

- Context Switch 발생시, PCB를 통해 (1) 이전 프로세스의 Context 를 저장하고, (2) 실행이 예약된 새 프로세스의 저장된 Context를 로드함

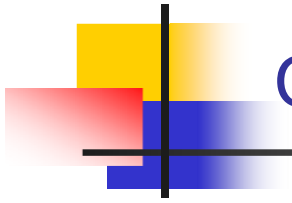


Context Switch

- Context Switch에 걸리는 시간은,
  전환하는 동안 시스템이 유용한 다른 작업을 수행하지 않기 때문에 순수 오버헤드.
  - 이 시간은 하드웨어에 크게 의존함.

- OS가 복잡할수록 Context Switch 중에 수행해야 하는 작업의 양이 늘어남.

Operations on Processes



Operations on Processes

- 대부분의 시스템에서 프로세스는 동시에 실행될 수 있으며 동적으로 생성 및 삭제될 수 있음.

- 따라서 OS는 프로세스 생성 및 종료를 위한 메커니즘을 제공해야 함.

Process Creation

- 실행 과정에서 프로세스는 새 프로세스들을 생성할 수 있음.
- 생성 프로세스를 부모 프로세스라고 하고,
  새 프로세스를 해당 프로세스의 자식 프로세스라고 함.
  - 이를 통해 프로세스 트리를 형성할 수 있음.

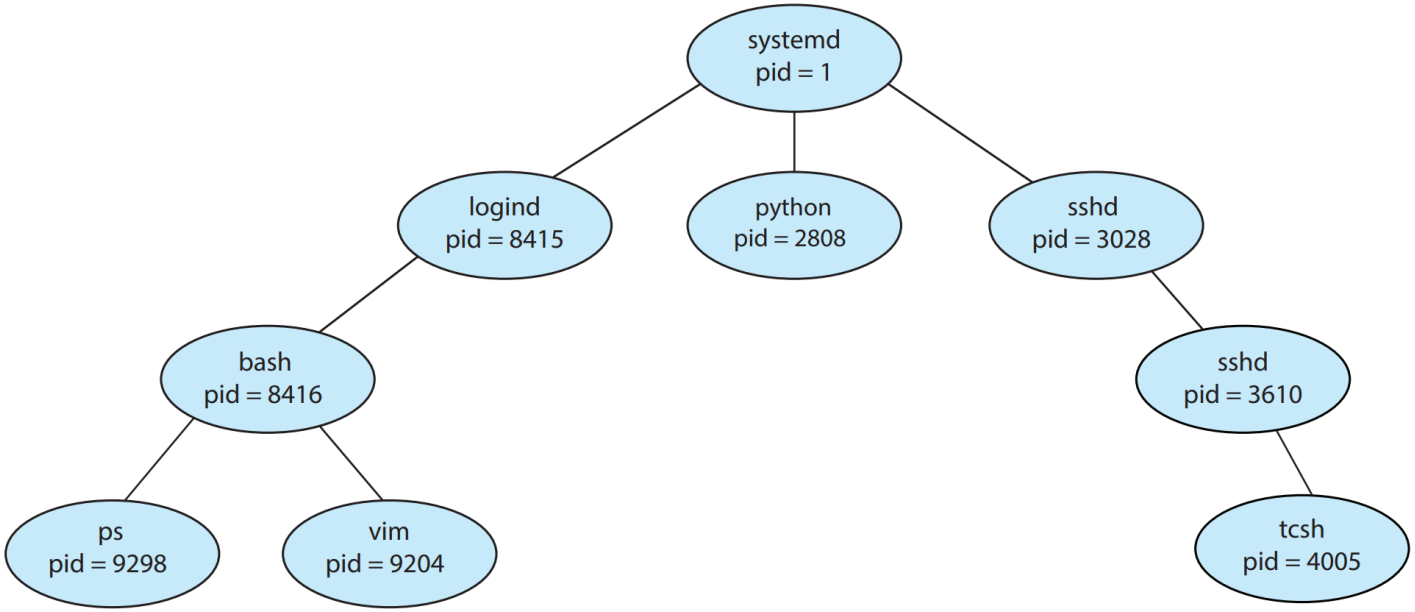


Figure 3.7 A tree of processes on a typical Linux system.

Process Creation

- OS는 고유한 Process ID(pid)에 따라 프로세스를 식별함.
- **pid:** 시스템의 각 프로세스마다 유니크하게 주어지는 값.
  각 프로세스에 액세스하기 위한 인덱스로 사용됨.

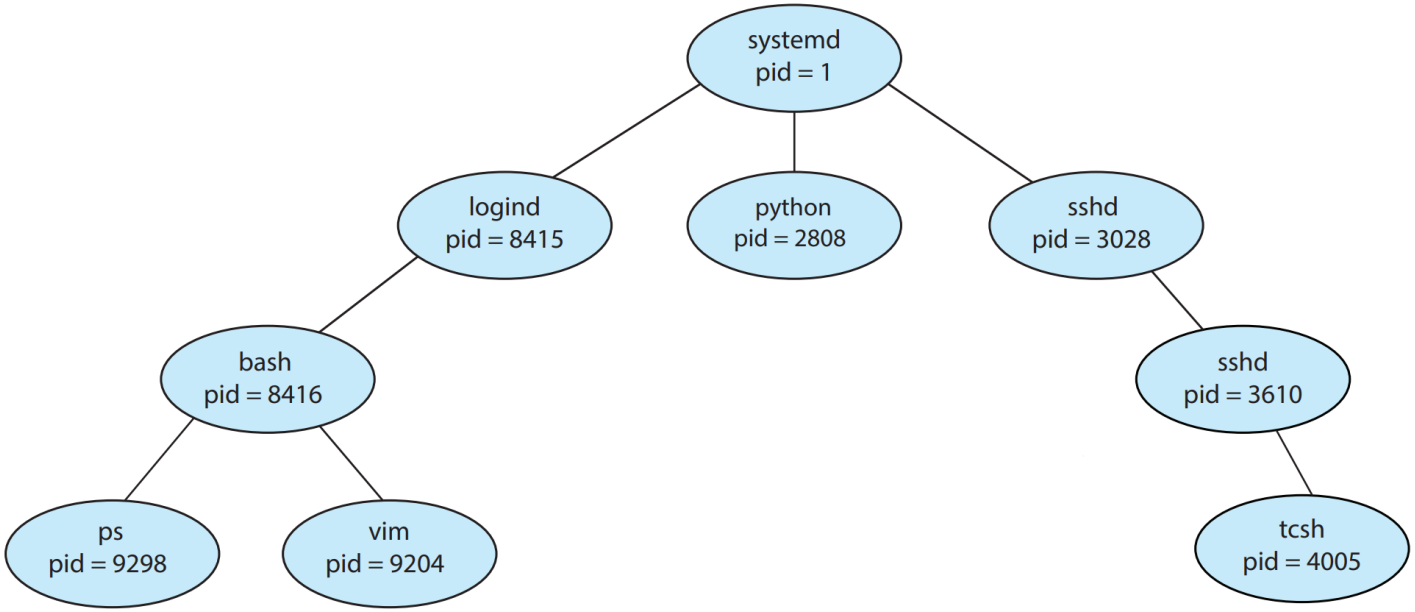
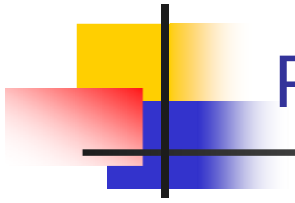


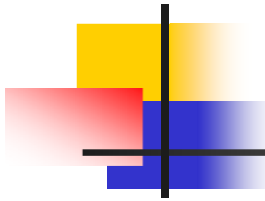
Figure 3.7 A tree of processes on a typical Linux system.



Process Creation

- 프로세스가 자식 프로세스를 만들 때,
  해당 자식 프로세스는 작업을 수행하기 위한
  특정 리소스(CPU 시간, 메모리, 파일, I/O 장치)가 필요함.

- 자식 프로세스는 운영 체제에서 직접 리소스를 얻을 수 있지만,
  보통은 사용 가능한 리소스가 부모 프로세스의 리소스로 제한됨
  - 프로세스가 너무 많은 자식 프로세스를 만들어 시스템에 과부하가 걸리는 것을 방지하기 위함.



Process Creation

- 프로세스가 새 프로세스를 생성할 때 (fork 시스템 콜 사용)
  두 가지 실행 가능성이 있음.
  - 1. 부모는 자식과 동시에 계속 실행됨.
  - 2. 부모는 자식의 일부 또는 전체가 종료될 때까지 기다림.
    (wait 시스템 콜 사용)

- 또한 새 프로세스에는 두 가지 주소 공간 가능성이 있음.
  - 1. 자식 프로세스는 부모 프로세스의 복제본
    (부모와 동일한 프로그램 및 데이터를 가짐)
  - 2. 자식 프로세스에 새 프로그램이 로드되어 있음
    (exec 시스템 콜 사용)

Process Creation

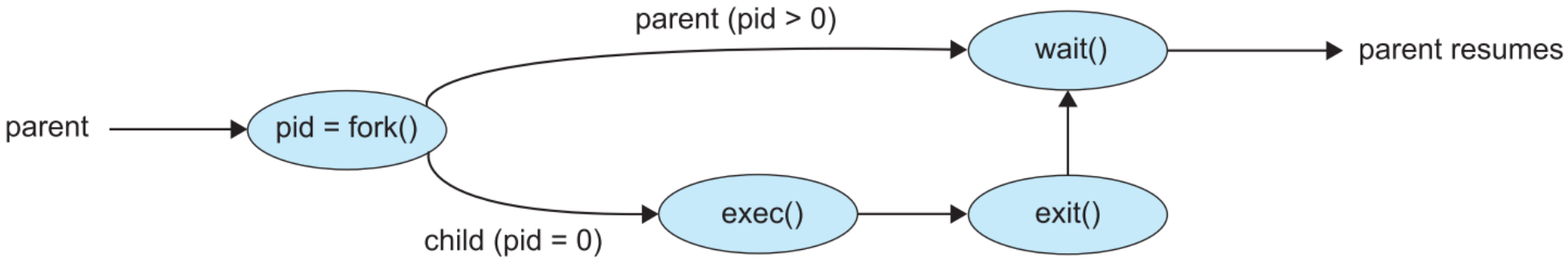
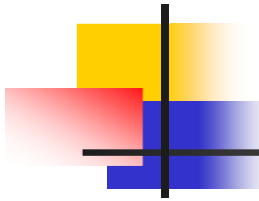
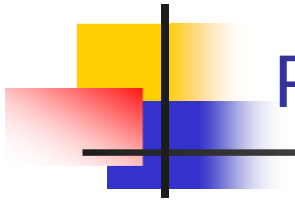


Figure 3.9 Process creation using the `fork()` system call.



Process Creation

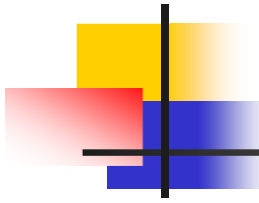
- fork() 시스템 콜에 의해 새 프로세스가 생성됨.
  - 새 프로세스는 원래 프로세스 복사본
  - 한 가지 차이점 : 자식의 pid는 0임
  - 두 프로세스(부모 및 자식) 모두 fork() 이후의 명령어에서 계속 실행됨.
- 자식 프로세스는 exec() 시스템 콜을 사용하여, 프로세스의 메모리 공간을 다른 프로세스의 내용으로 교체하고 실행 시작
- 부모 프로세스는 wait() 시스템 콜을 사용하여 자식 프로세스가 종료할 때 까지 기다림.
  - wait() 시스템 콜은 종료된 자식의 pid를 반환함.
  - 자식중 어떤 자식이 종료되었는지 식별하기 위함



Process Termination

- 프로세스는 최종 명령문 실행을 완료하고 운영 체제에 exit() 시스템 콜을 사용하여 삭제하도록 요청하면 종료됨.

- 할당된 메모리, 열린 파일, I/O 버퍼를 포함한 프로세스의 모든 리소스는 운영 체제에서 할당을 취소하고 회수됨.

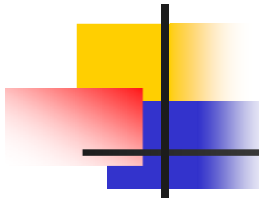


좀비 프로세스(zombie process)

- process table에서 자식 프로세스의 entry은 부모 프로세스가 wait() 을 호출할 때까지 남아있음

- 종료가 되었지만, 부모 프로세스가 아직 wait() 호출을 하지 않은 프로세스를 좀비 프로세스라고 함

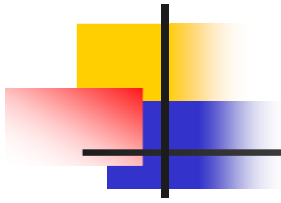
- OS는 한정된 숫자의 pid를 가지고 있음.
- 좀비 프로세스가 pid의 한계까지 차지하면, pid를 할당해야 할 프로세스에 더 이상 pid를 할당할 수 없게 되어 다른 프로세스 실행을 방해하게 됨.



고아 프로세스(orphan process)

- 부모가 wait()를 호출하지 않고 종료되면, 자식 프로세스는 orphan process가 됨.
- 리소스가 낭비되어 성능 저하의 원인이 됨.
- OS는 보통 부모보다 더 상위의 프로세스를 고아 프로세스에 새 부모로 할당함.
  - init 프로세스: UNIX 시스템에서 프로세스 계층의 루트 역할
- init 프로세스는 주기적으로 wait() 시스템 콜을 호출하여 고아 프로세스를 종료하고 리소스를 반환함.

Interprocess Communication



Interprocess Communication

- **independent process:** 시스템에서 실행 중인 다른 프로세스와 데이터를 공유하지 않는 프로세스
- **cooperating process:** 다른 프로세스와 데이터를 공유하는 모든 프로세스
- 시스템에서 실행 중인 다른 프로세스에 영향을 미치거나 영향을 받을 수 있음

Interprocess Communication

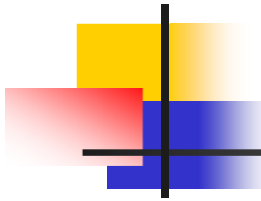
- 프로세스 협력을 허용하는 환경을 제공하는 이유
  - 1) 정보 공유: 여러 애플리케이션이 동일한 정보가 필요한 경우를 위해 정보에 동시 액세스를 허용
    - (예: 복사 및 붙여넣기)
  - 2) 계산 속도 향상: 각 task를 subtasks로 나누어 병렬 실행하여 속도를 향상. 멀티프로세서 환경에서만 가능
  - 3)모듈화: 시스템 기능을 별도의 프로세스 또는 스레드로 분할하여 모듈 방식으로 시스템을 구성할 수 있음
    - 각 모듈을 독립적으로 관리하는 것이 가능
    - 프로세스에 실시간으로 모듈을 붙여 작동시킬 수 있음
    - 이런 점들을 통해 효과적인 시스템 유지가 가능

IPC(interprocess communication)

- Cooperating process에는 데이터 교환, 즉 서로 데이터를 주고 받을 수 있는 IPC(interprocess communication) 메커니즘이 필요

- IPC에는 (1) shared memory와 (2) message passing이라는 두 가지 기본 모델이 있음.

- 대부분의 OS가 두 가지를 모두 지원.



Producer-Consumer Problem

- Cooperating process 중
  정보를 생산하는 프로세스를 생산자(Producer)
  정보를 소비하는 프로세스를 소비자(Consumer)라고 부름

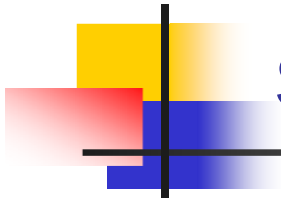
- Producer-Consumer Problem는
  두 프로세스가 동시에 동작할 때 일어나는 이슈.
- 보통 정보가 생산되는 속도가 소비하는 속도보다 빠르기 때문에 동기화 문제가 발생함.
  - ex) 서버를 Producer로, 클라이언트를 Consumer라고 할 때, 웹 서버는 리소스를 요청하는 클라이언트 웹 브라우저가 읽는 웹 콘텐츠를 생성하고 제공함. 이 때, 클라이언트가 최신 업데이트 된 콘텐츠를 소비하도록 할 필요가 있음.

Producer-Consumer Problem

- 해결법: Shared memory
- Producer-Consumer Problem을 해결하기 위해,
  메모리에 생산된 데이터를 담아두는 버퍼(Buffer)를 만듦.

- **유한 버퍼(Bounded buffer):**
  Consumer는 버퍼가 비어 있으면 기다려야 하고, Producer는 버퍼가 가득 차면 기다려야 함.

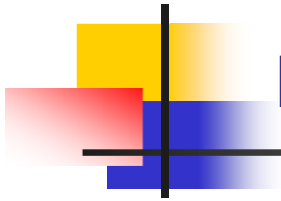
- **무한 버퍼(Unbounded buffer):**
  버퍼의 시작과 끝을 이어붙여 크기가 무한한 버퍼



Shared memory

- Cooperating process가 공유하는 메모리 영역이 설정됨.
- 그런 다음 프로세스는 공유 영역에서 데이터를 읽고 쓰는 방식으로 정보를 교환

- 충돌을 막기 위해 메모리에 동시 접근하는 것을 따로 방지해야 함.
  - 개발자의 입장에서는 구현에 난이도가 있음.



Message passing

- Cooperating process 간에 교환되는 메시지를 통해 통신

- 1. 프로세스들 사이에 통신 링크를 설정
- 2. 보내기/받기를 통해 메시지 교환

- 메시지 패싱은 Context Switch가 발생하기 때문에 속도가 느림.
- 커널이 기본적인 기능을 제공
  - 공유 메모리 방식에 비해 개발자의 입장에서는 구현이 쉬움.

Shared memory / Message passing

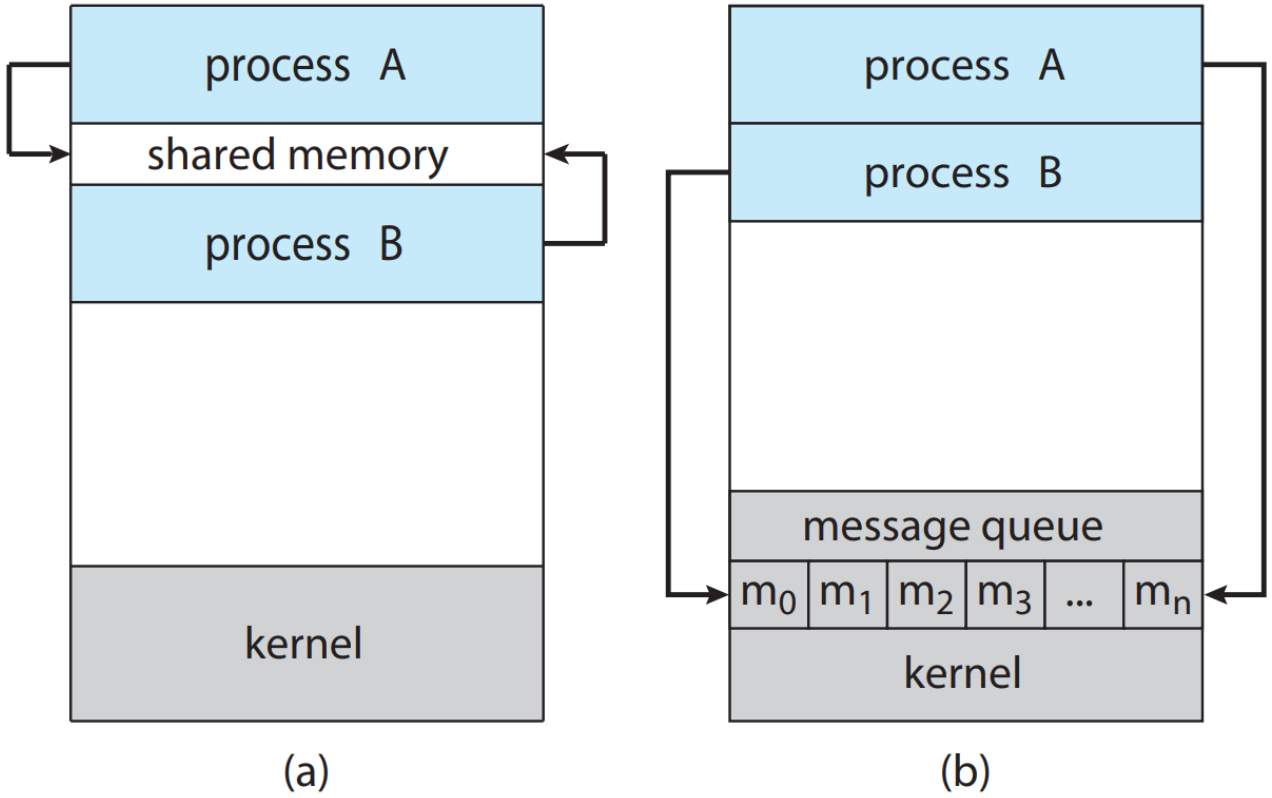
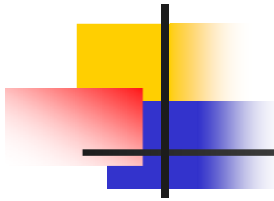


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.



Message passing

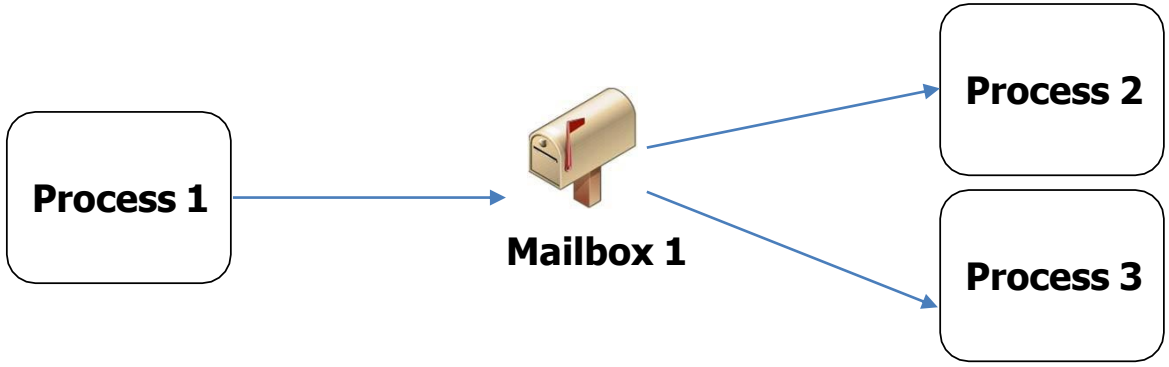
- Message passing의 링크 설정에는 Direct, Indirect 방식이 있음.

- **Direct communication**
  - 프로세스는 서로의 이름을 명시적으로 지정해야 함.
  - 쌍 사이에는 하나의 양방향 링크가 있음
  - 단점: 이름을 미리 알기가 어려움

- **Indirect communication**
  - Mail box(port라고도 함)를 통해 링크가 생성.
  - 링크는 많은 프로세스와 연관될 수 있음.
  - 프로세스는 여러 통신 링크를 공유할 수 있음.
  - 링크는 단방향일 수도 양방향일 수도 있음.

Message passing

- Mailbox를 프로세스 1이 프로세스 2, 3에게 메시지를 보낼 때 사용할 경우



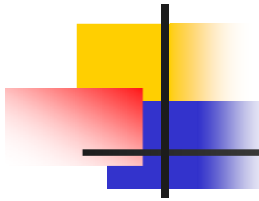
- Problem: P1이 메시지를 보냈을 때, P2, P3 중 누가 받을 것인가?
- Solution
  - 1.링크가 최대 두 개의 프로세스만 연결되도록 함
  - 2.Broadcasting
  - 3.하나의 프로세스만 메시지 수신 허용
  - 4.송신 프로세스가 수신자를 선택하도록 허용

Mailbox Synchronization

- Message passing의 동기화를 위해
  blocking 방식과 non-blocking 방식이 사용됨
  - blocking 방식: Synchronous 동기식
  - non-blocking 방식 : Asynchronous 비동기식

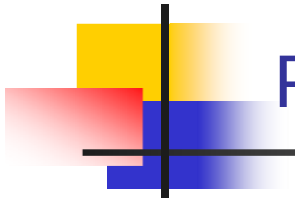
- **Blocking send:** 수신자가 메시지를 받을 때까지 송신자는 block됨
- **Blocking receive:** 메시지를 수신할 때까지 수신자는 block됨

- **Non-blocking send:** 송신자가 메시지를 보내고 작업을 계속함
- **None-blocking receive:** 수신자가 유효한 메시지나 Null 메시지를 받음



Pipe (파이프)

- 두 프로세스가 통신할 수 있도록 하는 연결관 역할을 함.
- 파이프는 초기 UNIX 시스템에서 최초의 IPC 메커니즘 중 하나.
- 단방향 통신만 가능함. 양방향 통신이 필요한 경우 각 파이프가 다른 방향으로 데이터를 보내는 두 개의 파이프를 사용해야 함.
- ordinary pipes는 부모자식 관계에만 사용 가능하지만,
  named pipes는 그렇지 않은 프로세스 끼리도 통신 가능.



Pipe (파이프)

- 두 프로세스가 통신할 수 있도록 하는 연결관 역할을 함.
- 파이프는 초기 UNIX 시스템에서 최초의 IPC 메커니즘 중 하나.
- ordinary pipes는 부모-자식 관계에만 통신할 수 있지만, named pipes는 그렇지 않은 프로세스 간에도 통신이 가능.

Pipe (파이프)

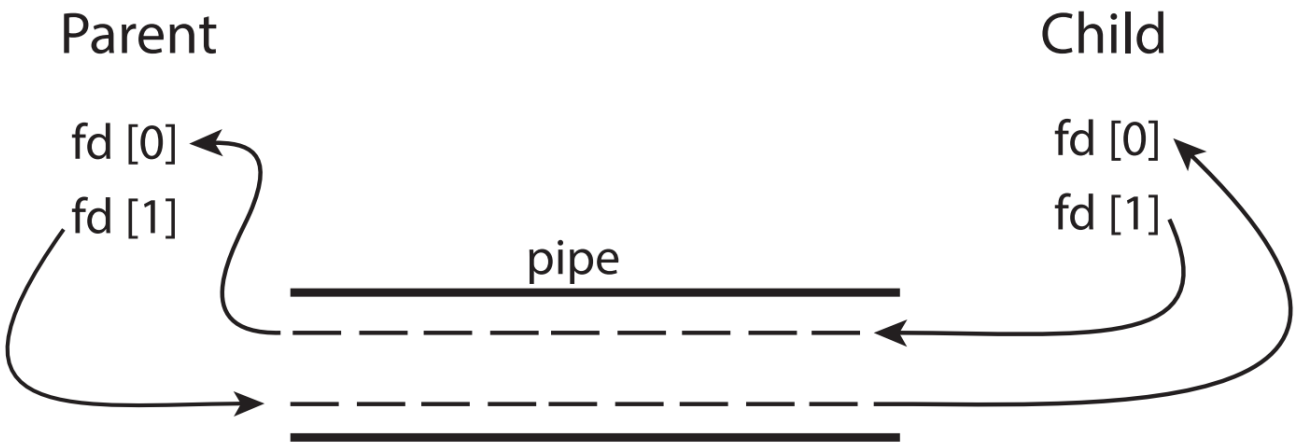


Figure 3.20 File descriptors for an ordinary pipe.

- 단방향 통신만 가능함.
- 양방향 통신이 필요한 경우 각 파이프가 다른 방향으로 데이터를 보내는 두 개의 파이프를 사용해야 함.

Communication in Client–Server Systems

Communication in Client–Server Systems

- 클라이언트와 서버의 통신을 위해
  일반적으로 아래의 2가지 방법이 사용됨
  - 1) Sockets
  - 2) RPC(Remote Procedure Calls)

Sockets

- 소켓은 서버와 클라이언트가 통신하는 방식.
- IP주소와 포트 정보가 있으면 클라이언트는 네트워크를 통해 서버 프로세스에 접근할 수 있음.

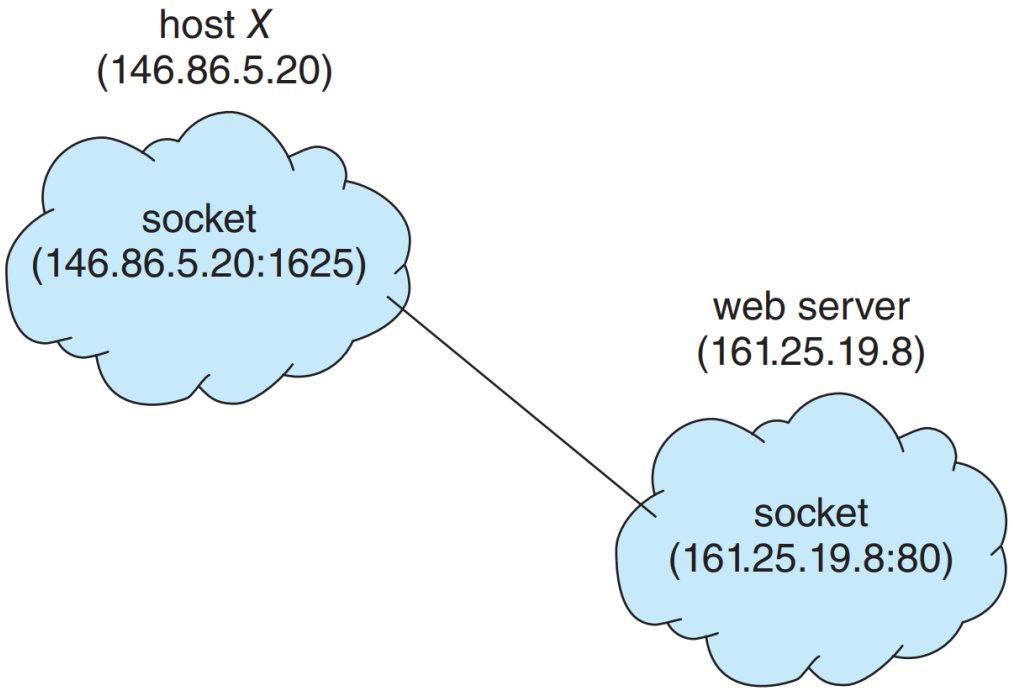


Figure 3.26 Communication using sockets.

RPC(Remote Procedure Calls)

- 별도의 원격 제어를 위한 코딩 없이 별도의 컴퓨터에 있는 다른 프로세스에서 function (procedure)를 호출할 수 있는 방식
  - 프로세스(Process)는 원래 같은 컴퓨터 내의 function (procedure)만 호출하여 실행 가능함.
  - RPC는 별도의 컴퓨터의 프로세스의 function (procedure)를 실행할 수 있게 해줌.

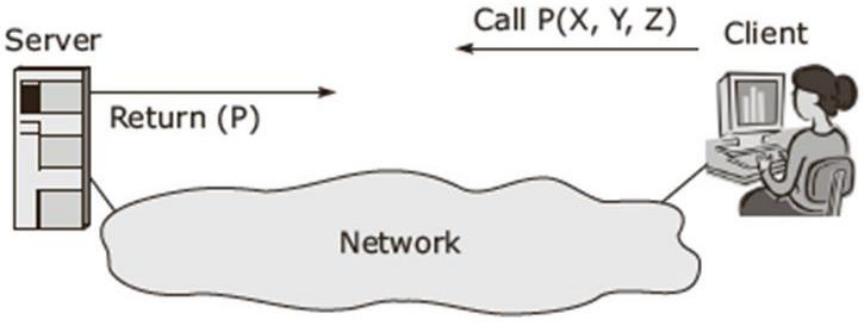


Figure 4-3 Basic RPC model

RPC(Remote Procedure Calls)

- 클라이언트-서버 간의 커뮤니케이션에 필요한 상세정보를 최대한 감출 수 있음
- 클라이언트는 일반 메소드를 호출하는 것처럼 간단하게 원격 컴퓨터의 function (procedure)를 호출할 수 있음

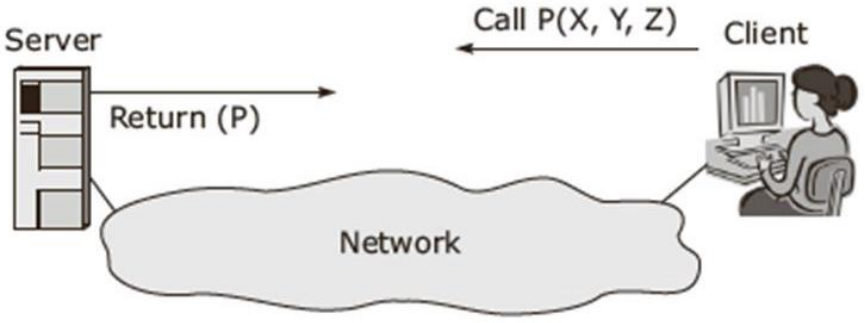


Figure 4-3 Basic RPC model

Chapter 3
Finish