3장. 프로세스

프로세스 관리

- 3장 – 프로세스(Process)
- 4장 – 쓰레드(Thread)
- 5장 – 프로세스 동기화(Synchronization)
- 6장 – CPU 스케쥴링

목표

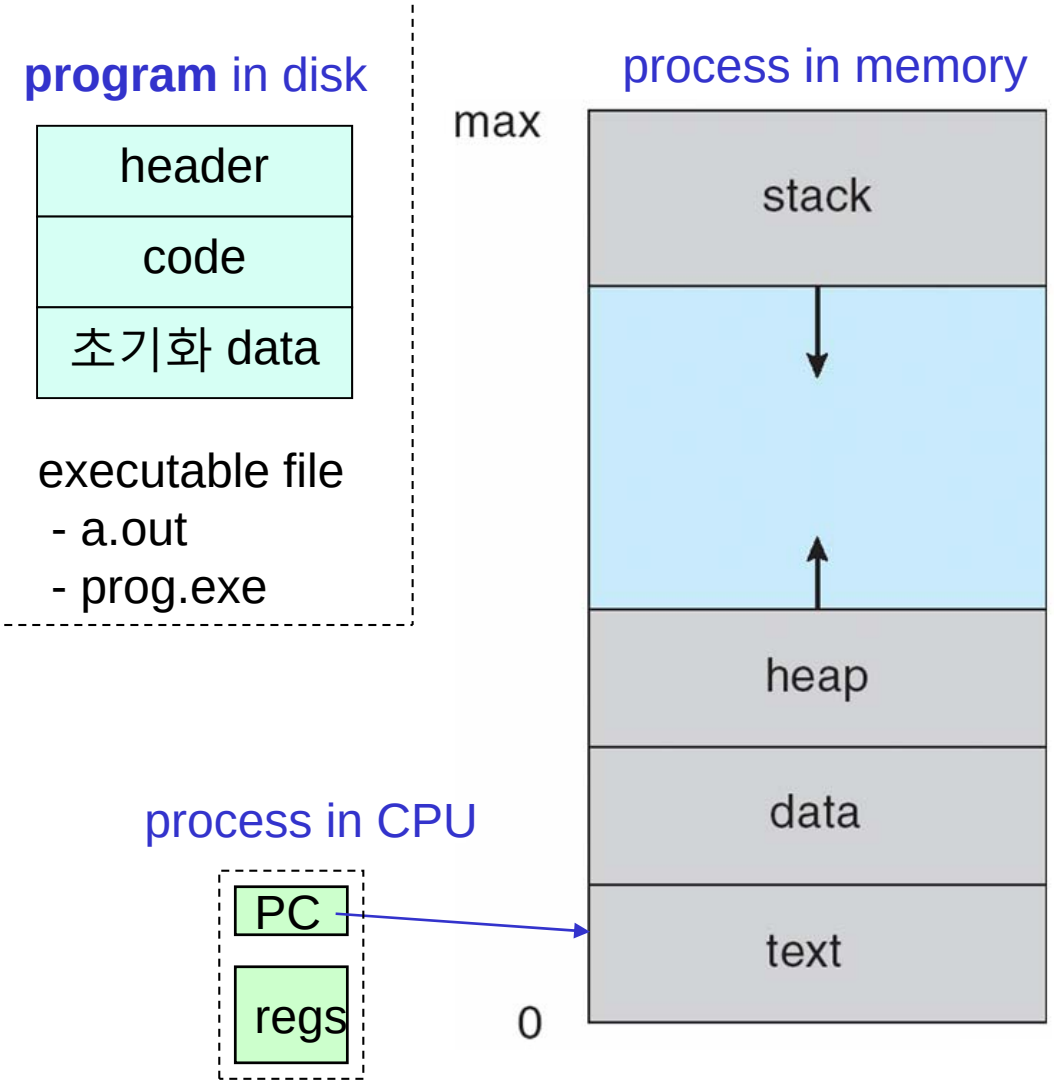
- 프로세스 개념
  - 프로세스는 실행 중인 프로그램으로 모든 계산의 기반이 됨
- 프로세스의 여러 특성
  - 스케쥴링, 생성 및 종료 등
- 프로세스간 통신
  - 공유 메모리(shared memory), 메시지 전달(message passing)
- 클라이언트-서버 시스템에서의 통신

3.1 프로세스 개념

- CPU activity를 부르는 명칭
  - Batch 시스템 → 작업(Job)
  - Time-shared 시스템 → User program, Task, Process
- Job, Task, Process 용어는 거의 같은 의미로 사용됨

- 프로세스 - 실행 상태에 있는 능동적 개체, 연관된 자원 사용
  - 프로그램 코드보다 더 많은 부분을 포함
    - **메모리** : code + data + stack + heap
    - **CPU** : program counter(PC) + registers
    - 프로세스 관리용 자료구조 : 프로세스 제어 블록(PCB)
    - …
- 프로그램 – 디스크에 파일로 저장된 수동적 개체
  - 프로그램 파일 = 코드(text section) + 초기화 데이터 + 헤더정보 등

프로세스와 프로그램



stack : 임시 데이터 저장
- 함수 매개변수
- return 주소
- 지역변수

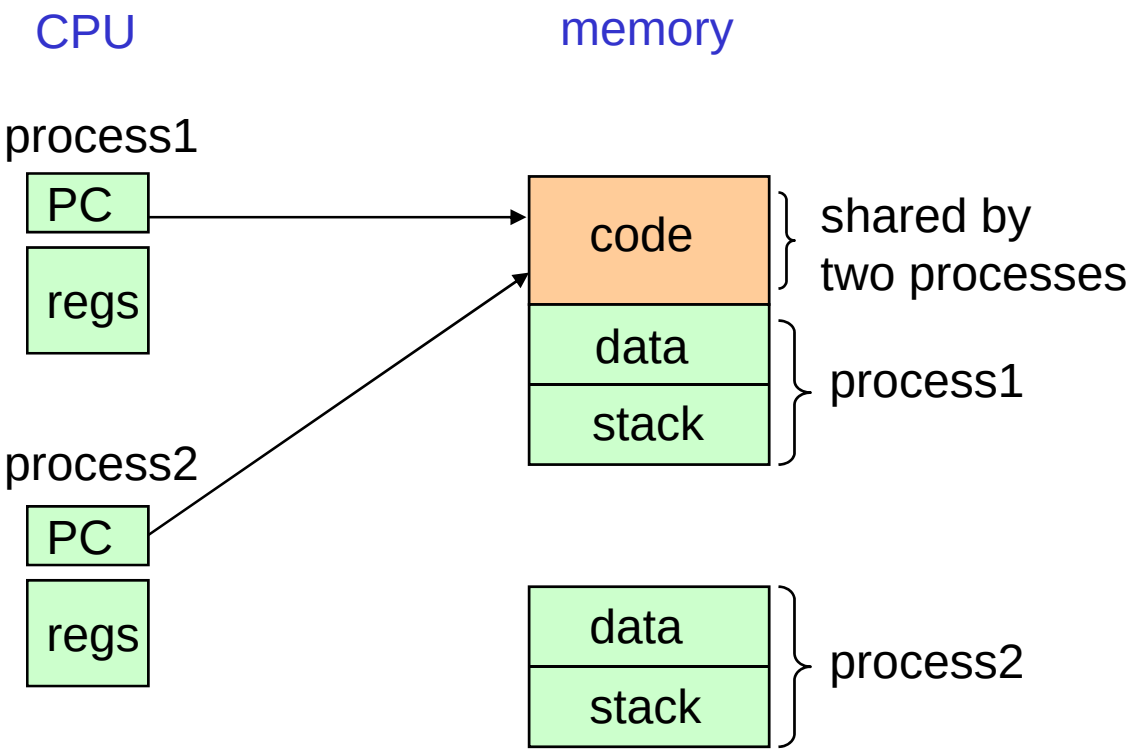
heap : 실행시간 동안에
동적 할당되는 메모리

data section : 전역변수
– 초기화 전역변수 (data)
– 비초기화 전역변수 (bss)

text section
– 프로그램 코드

프로세스와 프로그램

- 여러 개의 프로세스가 같은 프로그램 사용 가능
  - text section은 같음 – 같은 메모리 공유, 개별적인 실행 시퀀스
  - data/stack/heap section은 다름 – 프로세스마다 개별적인 메모리 할당

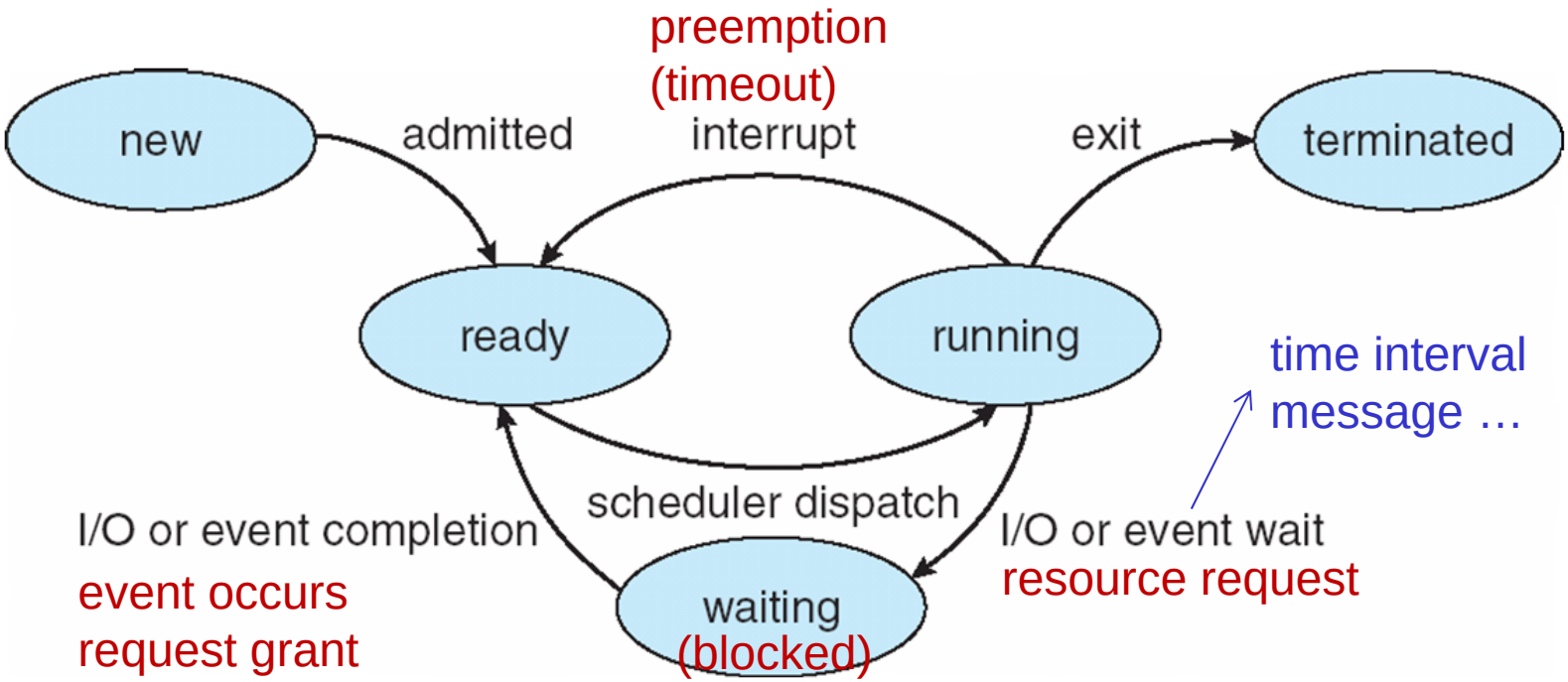


프로세스 상태

- 프로세스 상태
  - 프로세스의 현재 활동을 나타내며, 실행하는 동안 상태 변화가 발생함
- 프로세스 상태의 종류
  - **생성(new)** – 프로세스가 생성 중임
  - **실행(running)** – 명령어가 실행되고 있음
  - **대기(waiting)** – 어떤 사건(입출력 완료, 신호 수신 등) 발생을 기다림
  - **준비(ready)** – 프로세스가 실행할 준비가 됨 → CPU의 할당을 기다림
  - **종료(terminated)** – 프로세스의 실행이 종료됨 →종료 후 처리를 기다림

프로세스 상태 이름과 종류는 운영체제마다 차이가 있다.

프로세스 상태도

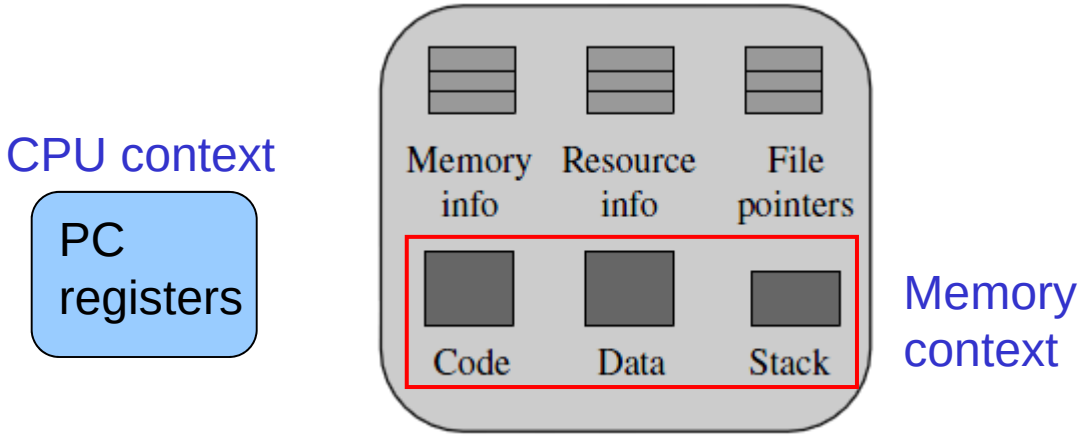


- 한 CPU에서는 어떤 순간에 한 프로세스만 실행될 수 있다.
- 많은 수의 프로세스가 ready 또는 waiting 상태일 수가 있다.
- 프로세스가 다른 프로세스에 의해서 종료되면
  - 프로세스는 ready 또는 waiting 상태에서 terminated 상태로 이동될 수 있다.

프로세스 제어 블록(PCB)

- 프로세스 제어 블록(Process Control Block)
  - 프로세스에 대한 정보를 포함하는 OS 커널의 자료 구조
  - 태스크 제어 블록(task control block) 이라고도 함

- 프로세스 context
  - 프로그램의 실행 상태 정보 – 실행시간 동안 변경 가능
    - CPU context, Memory context 등



PCB 정보

- PCB 정보
  - Process state
  - Program Counter(PC)
  - CPU registers
  - Memory-management 정보 (7장)

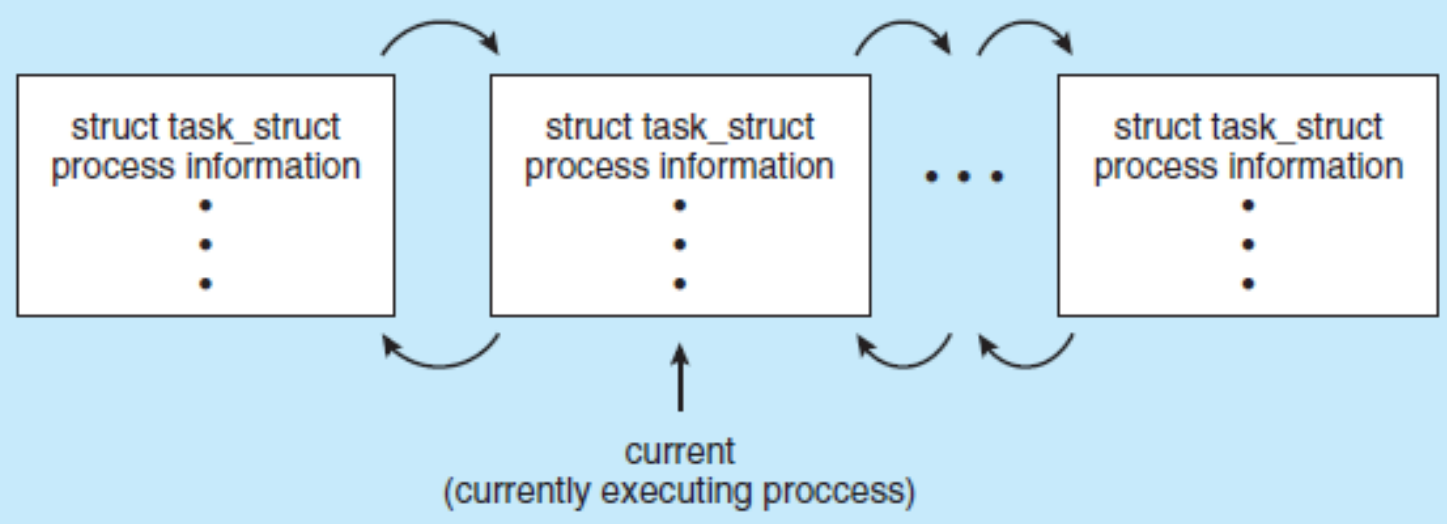
  CPU context
  - CPU scheduling 정보 (6장)
  - I/O status 정보
    - the list of allocated I/O devices
    - the list of I/O requests
    - the list of open files …
  - Accounting information
    - process number(ID), account number
    - CPU time, time limit, ...

(예) 리눅스에서의 Process 표현

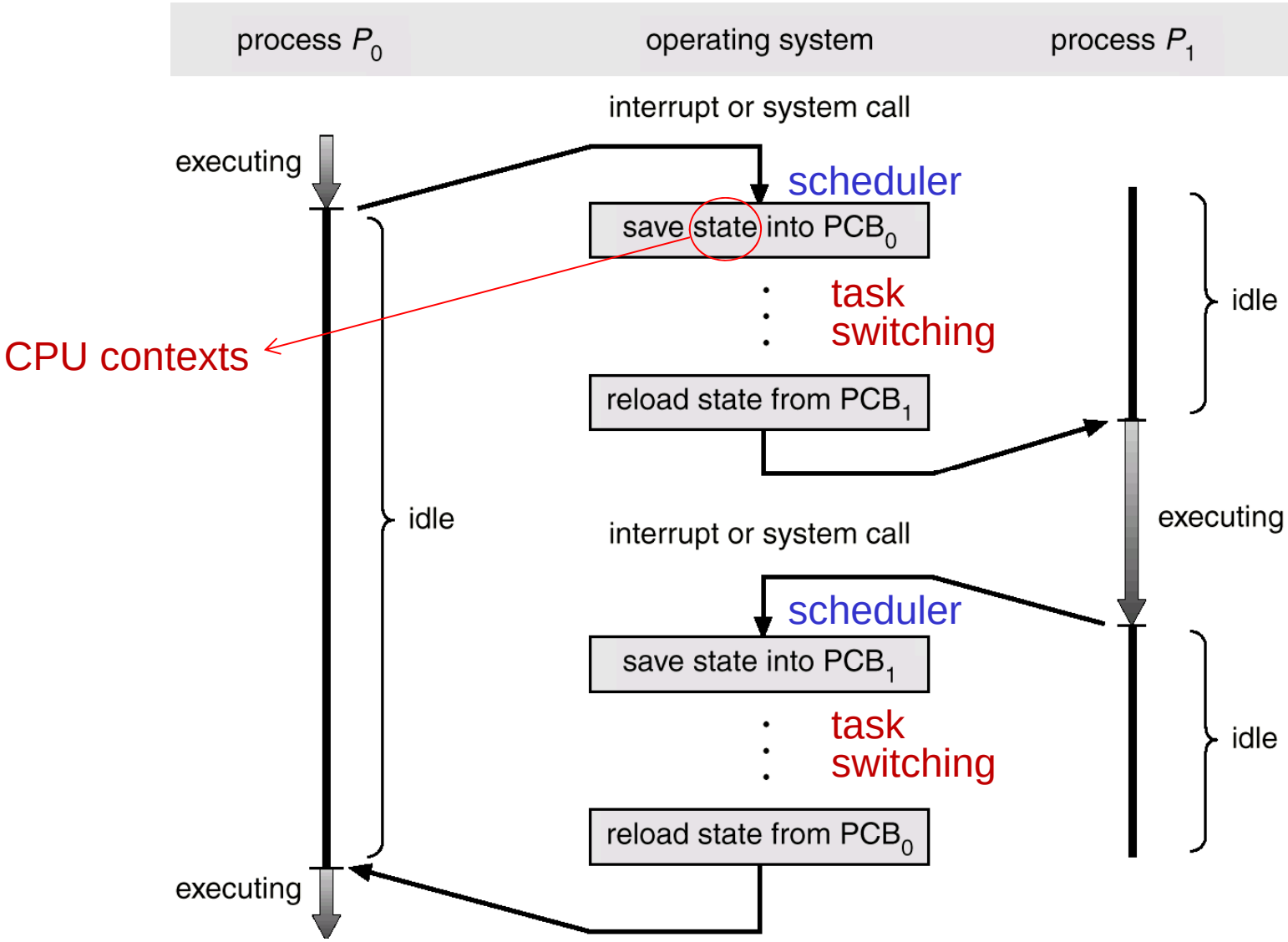
- struct task_struct

```
long state; /* state of the process */
struct sched_entity se; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
  ...
```

- 모든 active process는 task_struct 자료의 doubly link list로 관리됨



프로세스 간에 CPU switch

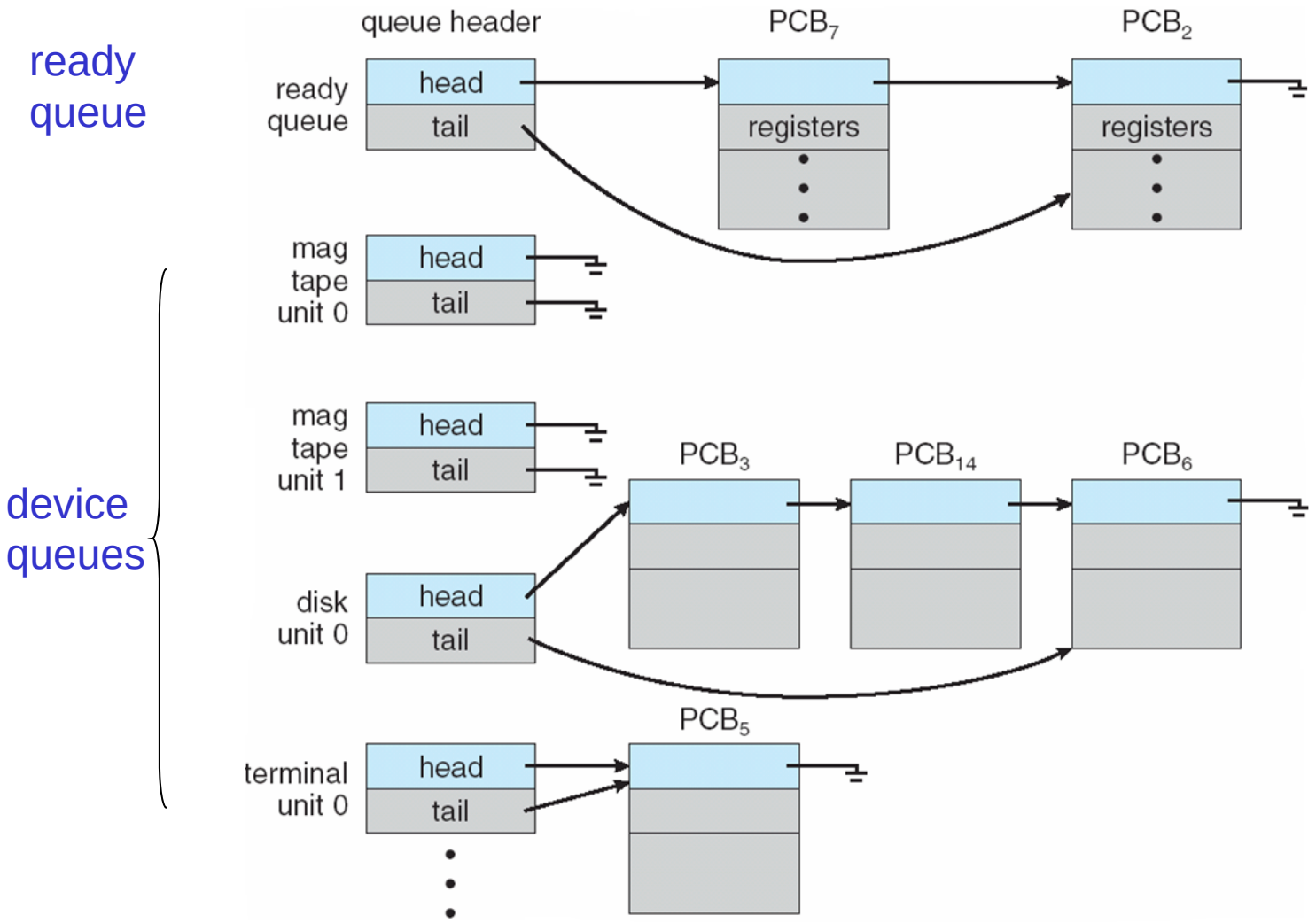


3.2 프로세스 스케쥴링

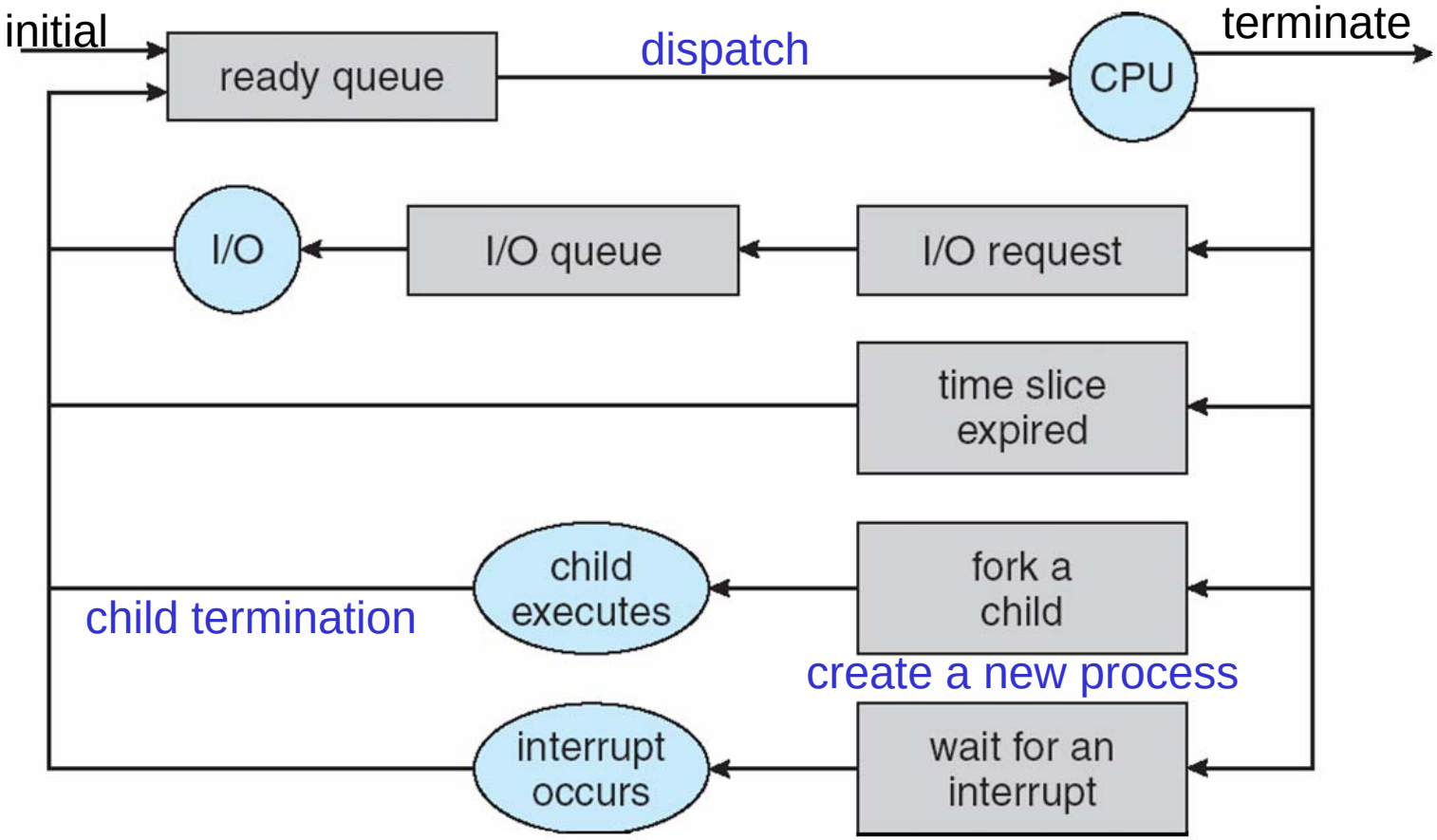
- 스케쥴링의 필요성
  - multiprogramming – CPU 이용률 극대화
  - time sharing – 사용자와의 상호 작용 → 빈번한 CPU 스위칭
- 프로세스 스케쥴링 Queue
  - Job queue: 모든 프로세스들의 집합
  - Ready queue: ready 상태의 프로세스들의 집합 (메모리에 위치)
  - Device queues: 특정 I/O 장치 사용을 대기하는 프로세스들의 집합

  → 일반적으로 linked list로 저장
- 프로세스들은 실행 동안 여러 queue들 사이에서 이동함
- 프로세스가 종료되면 queue에서 제거되며
  할당된 자원과 PCB도 반납(deallocate)함

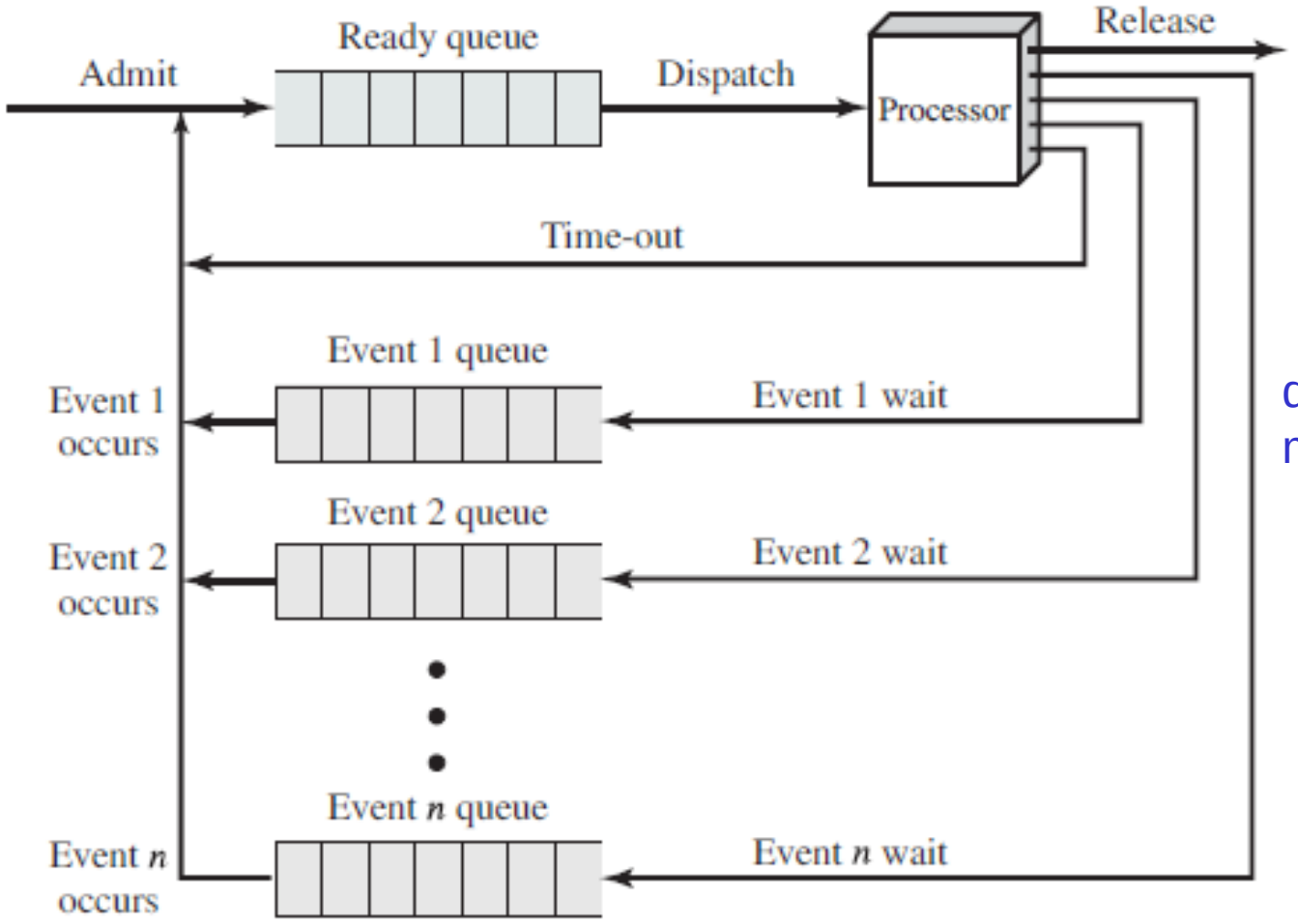
Ready Queue와 여러 I/O Device Queues



프로세스 스케쥴링에 대한 Queueing



프로세스의 Queue들 간의 이동



queue migration

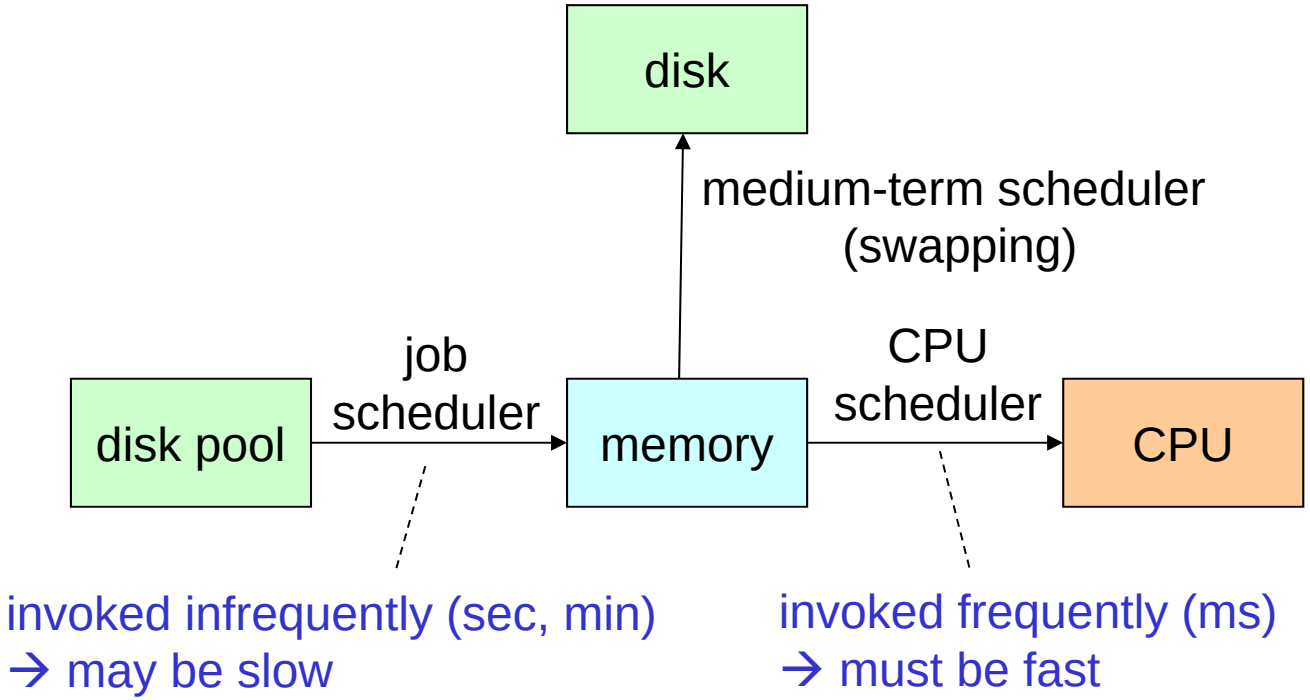
Waiting Queue

스케줄러

- 스케줄러
  - queue간에 이동할 프로세스를 선택하는 역할을 담당

- 장기(Long-term) 스케줄러 (job scheduler)
  - disk pool에서 ready queue로 이동시킬 프로세스를 선택
  - batch systems에서 사용
- 단기(Short-term) 스케줄러 (**CPU** scheduler) → 스케줄러
  - 다음에 실행시킬 프로세스를 선택하여 CPU를 할당함
- 중기(Medium-term) 스케줄러
  - time-sharing system에서 추가된 중간 레벨의 스케줄링
  - 메모리에서 disk로 이동시킬 프로세스를 선택 (swapping이라고 함)

3가지 스케줄러

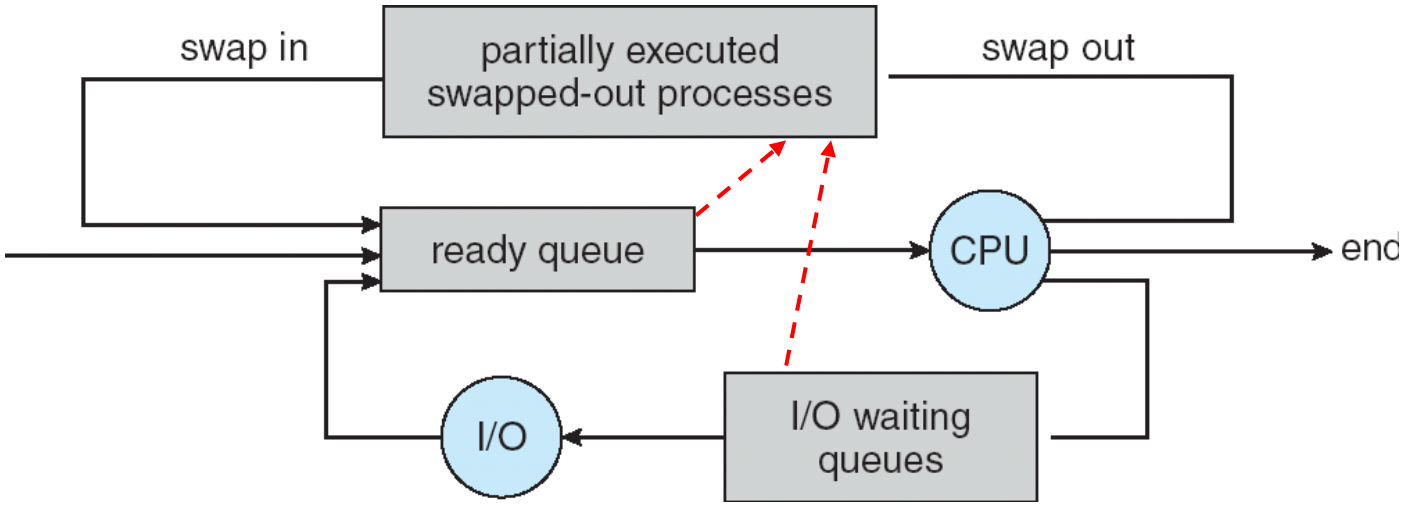

disk
medium-term scheduler
(swapping)
job
scheduler
CPU
scheduler
disk pool
memory
CPU
invoked infrequently (sec, min)
→ may be slow
invoked frequently (ms)
→ must be fast

Long-Term 스케줄러

- I/O bound process와 CPU bound process
  - I/O-bound process
    - 계산보다는 I/O 수행에 더 많은 시간을 소비함
    - 많은 수의 짧은 CPU bursts(연속적으로 CPU를 사용하는 시간)
  - CPU-bound process
    - 계산에 더 많은 시간을 소비함
    - 적은 수의 매우 긴 CPU bursts.
- 장기 스케줄러의 역할
  - multiprogramming의 정도(degree)를 제어함
  - I/O bound process와 CPU bound process들의 적절한 조합을 선택하는 것이 중요함
- 시분할(대화형) 시스템에서는 장기 스케줄러가 없거나 최소화
  - 물리적 제한(예:터미널 수)이나 인간의 스스로 조절하는 능력으로 대치됨

Medium-Term 스케줄러

- medium-term 스케줄러의 중요 아이디어
  - CPU에 대한 경쟁(contention)을 줄임 → multiprogramming 정도 감소
  - 새로 적재될 프로세스를 위한 메모리 공간 확보



Context Switch

- 프로세스의 CPU Context
  - PC, CPU registers, memory management 정보 등
  - Process switching 전에 PCB 등의 장소에 저장해야 함
- Context switch (= Process switching)
  - CPU가 다른 프로세스로 전환될 때에 운영체제는
    - old process의 현재 CPU 상태(context)를 저장하고
    - new process의 저장된 상태(context)를 CPU에 적재한다.
- 프로세스 전환 과정
  - 현재 process의 PCB에 있는 state를 “running”에서 다른 상태로 갱신하고 PCB를 ready queue에서 갱신 상태에 적절한 queue로 이동함
  - 새로 선택된 process의 PCB에 있는 state를 “running”으로 갱신
  - 현재 process의 CPU context를 자신의 PCB에 저장
  - 새 process의 PCB에서 CPU context를 CPU로 복원함

Context Switch Overhead

- Context-switch time은 overhead임
  - switching 동안 시스템은 유용한 작업을 하지 않음
  - 운영체제가 더 복잡할 수록 context switching 동안 더 많은 작업이 필요

    (예) memory management 사용은 추가적인 context data를 사용하고 이에 대한 코드가 추가적으로 필요 함
- Context-switch time은 하드웨어 지원에 크게 영향 받음
  - Sun UltraSPARC는 여러 개의 레지스터 집합을 제공
    - context switch 동안 현재 레지스터 집합에 대한 포인터만 변경
  - 많은 processor들이 모든 레지스터를 적재(load)하고 저장(store)하는 special instruction을 제공
- Process Switching은 간접적인 overhead도 가짐
  - 캐쉬 무효화(cache invalidation) 등.

모바일 시스템에서의 Multitasking

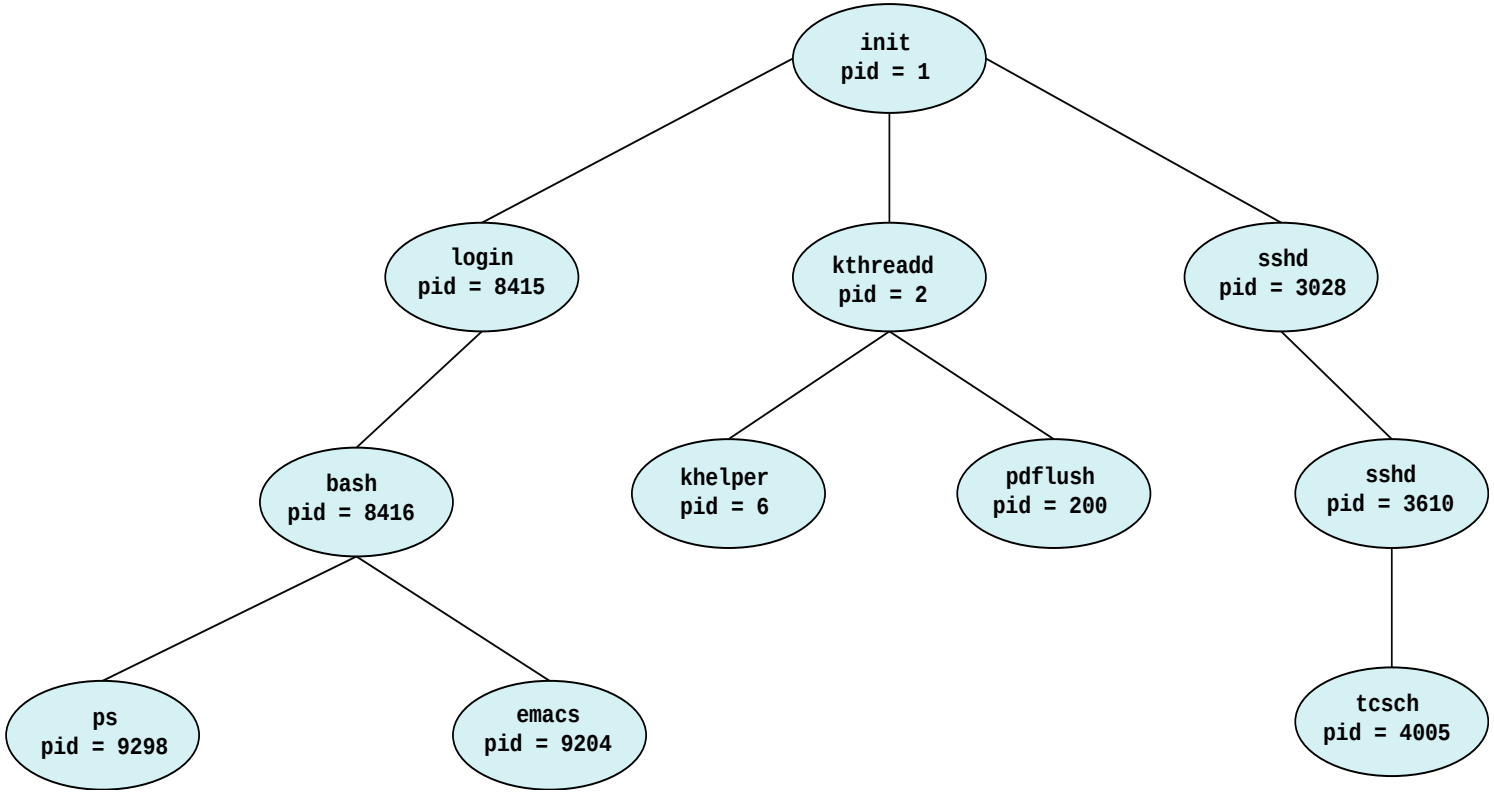
- 초기의 iOS는 사용자 application의 multitasking을 제공하지 않음
- iOS 4부터 제한된 형태의 multitasking 지원
  - 단일 foreground 프로세스 – 화면 사용
  - 다수의 background 프로세스 – 메모리에 있지만, 화면에 보이지 않음
    - 단일 작업을 수행하는 제한된 길이의 작업(예: 다운로드)
    - 사건 공지(notification) (예: 메일 메시지 수신)
    - 실행시간이 긴 작업 (예: 오디오 재생)
- Android는 background 응용에 제한이 없음
  - application은 background에서 service를 사용해야 함
  - service : background 프로세스 대신에 수행하는 분리된 응용 컴포넌트
  - service는 사용자 인터페이스를 갖고 있지 않고, 적은 메모리를 사용하여 모바일 환경에서의 multitasking에 효율적임

3.3 프로세스 연산

- 프로세스 생성(creation)
  - 프로세스는 create-process 시스템 호출을 사용하여 new 프로세스를 생성
    - 부모 프로세스 – 자식 프로세스
  - 생성된 프로세스들의 계속적인 프로세스 생성의 결과로 프로세스 트리(process tree)를 형성함 (다음 쪽 참조)
  - 프로세스들은 프로세스 식별자(process identifier, **pid**)로 구분

- 부모프로세스와 자식프로세스 간의 여러 가지 자원 공유 방법
  - 자식 프로세스가 OS에서 자원을 직접 얻음
    - 부모와 자식 프로세스간에 자원을 공유하지 않음
  - 자식 프로세스가 부모 프로세스 자원의 부분집합을 사용
    - 자식 프로세스는 부모 프로세스의 모든 자원 또는 일부 자원 공유
    - 부모 프로세스는 자식 프로세스들에게 자신의 자원을 분할 제공

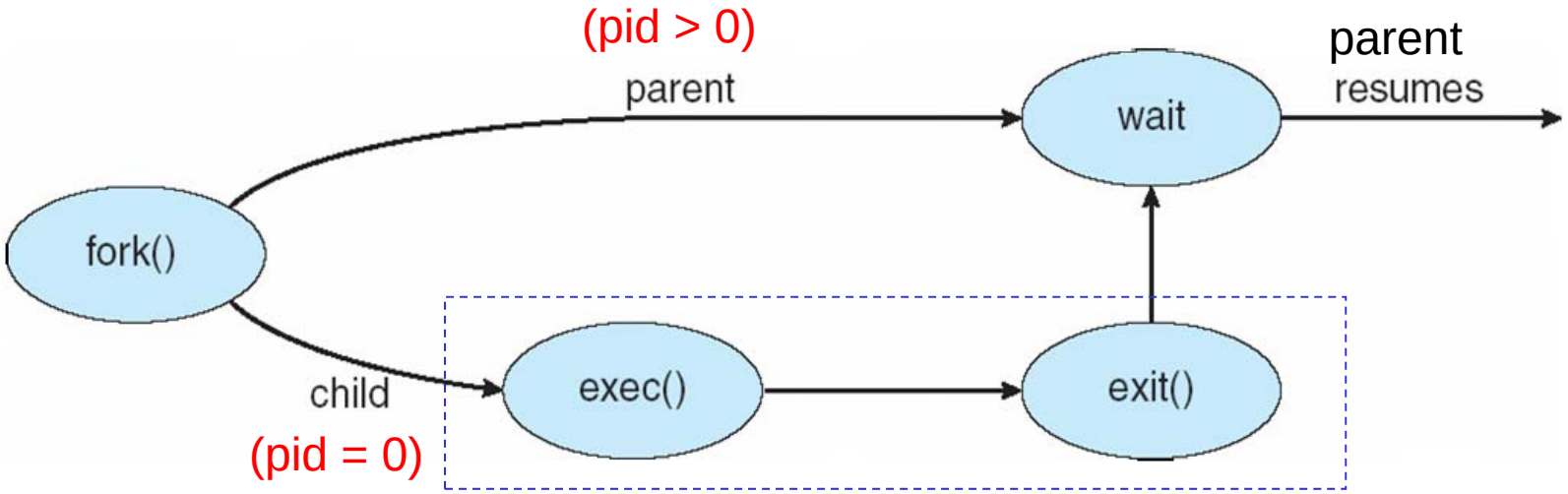
전형적 Linux 시스템의 프로세스 트리



프로세스 생성

- 실행(Execution)
  - 부모와 자식이 병행(concurrent) 수행, 또는
  - 부모가 모든 또는 일부 자식이 끝나기를 기다림(wait)
- 주소 공간(Address space)
  - 자식은 부모의 복사본(duplicate), 또는
  - 자식은 새로 적재되는 프로그램을 가짐
- Examples
  - UNIX: **fork** 와 **exec** system call 사용
    - **fork()** : new process 생성.
      부모 프로세스의 주소공간의 복제본으로 구성됨
    - **exec()** : 프로세스 메모리 공간을 새 프로그램으로 대치하여 새로운 프로그램을 실행
  - Win32 API:
    - CreateProcess() : 프로그램을 적재하여 자식프로세스를 생성하여 실행. 인수가 복잡함(10개 이상)

UNIX에서의 프로세스 생성

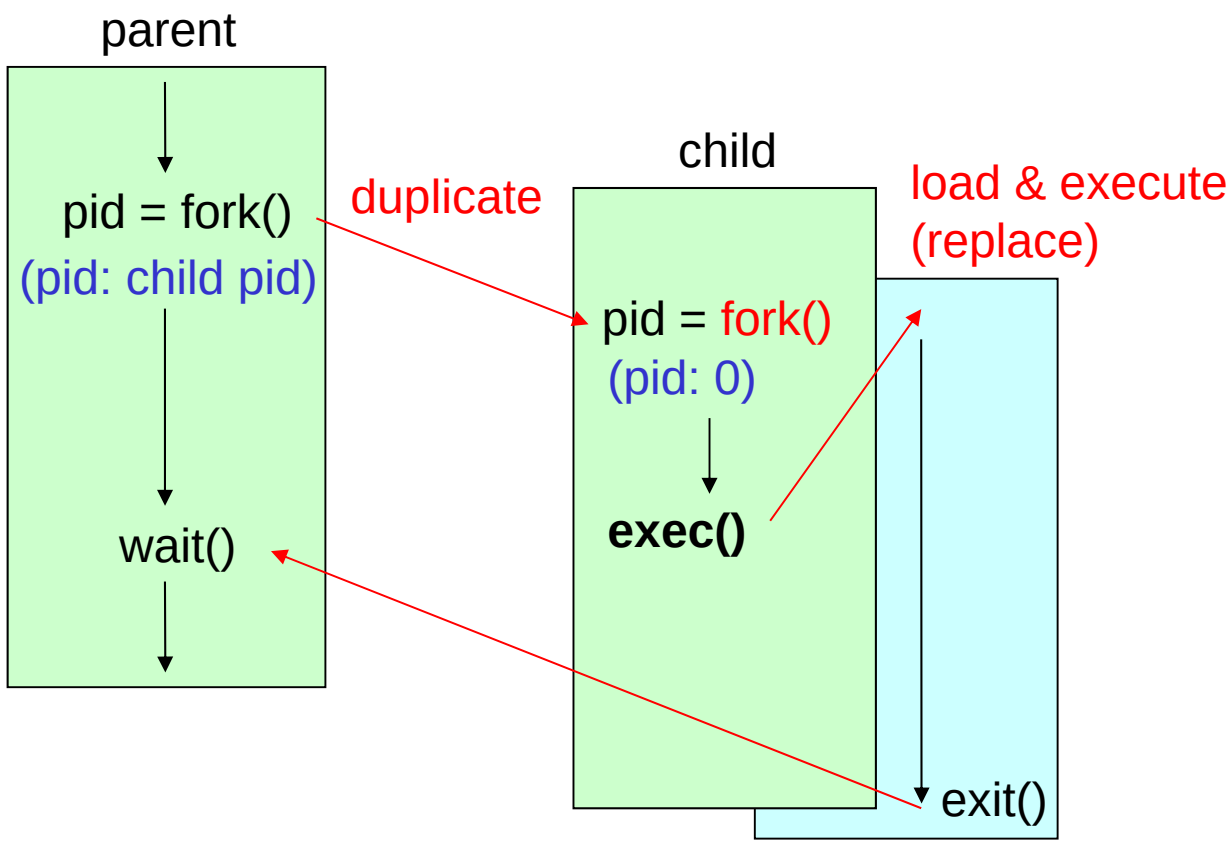


인수 형태에 따라서 여러 종류의
exec 계열 함수가 제공됨

프로세스를 생성하는 C 프로그램 (UNIX)

```
#include <stdio.h>
int main(void)
{
    int pid;
    pid = fork();              /* fork another process */
    if (pid < 0) {
        fprintf(stderr, "Fork failed");
        exit(-1);
    } else if (pid == 0) {     /* child process */
        execlp("/bin/ls", "ls", NULL);  /* execute a new program */
    } else {                   /* parent process */
        wait(NULL);                    /* wait for the child to complete */
        printf("Child complete");
        exit(0);
    }
}
```

UNIX에서의 fork/exec/wait 동작



Windows에서의 프로세스 생성

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

프로세스 종료 (정상적)

- 프로세스의 정상 종료
  - 프로세스가 마지막 문장의 실행이 끝나면 **exit()** 시스템 호출을 하여 운영체제에게 자신을 삭제를 요청함
  - 자식 프로세스는 부모에게 상태 값(대개 정수)을 반환할 수 있다.
    - 부모는 **wait()** 시스템 호출을 하여 이 값을 기다림
  - 운영체제는 프로세스가 사용한 자원을 반납(deallocate)함
- 좀비(Zombie) 프로세스
  - 종료되었지만 부모가 wait()를 아직 호출하지 않아서 남아있는 프로세스 (terminated/zombie 상태). 짧은 시간 동안 이 상태에 머무름

프로세스의 종료 (비정상적)

- 프로세스의 비정상 종료

  다음 경우에 부모가 **abort()** 시스템 호출을 하여 자식을 종료할 수 있음
  - child가 자신에게 할당된 자원을 초과하여 사용할 때
  - child에게 할당된 태스크(작업)가 더 이상 필요 없을 때
  - 부모가 종료하는 데, 운영체제가 자식이 계속하여 실행하는 것을 허용하지 않을 때 → 연쇄적 종료(cascading termination)

- 부모 프로세스 종료 후에 자식의 계속적인 실행을 허용하는 경우
  - 모든 프로세스는 종료 후 처리를 위하여 부모 프로세스가 있어야 함
  - 자식 프로세스는 고아(orphan) 프로세스가 되며,
    init 프로세스를 자식 프로세스의 새로운 부모 프로세스로 지정함

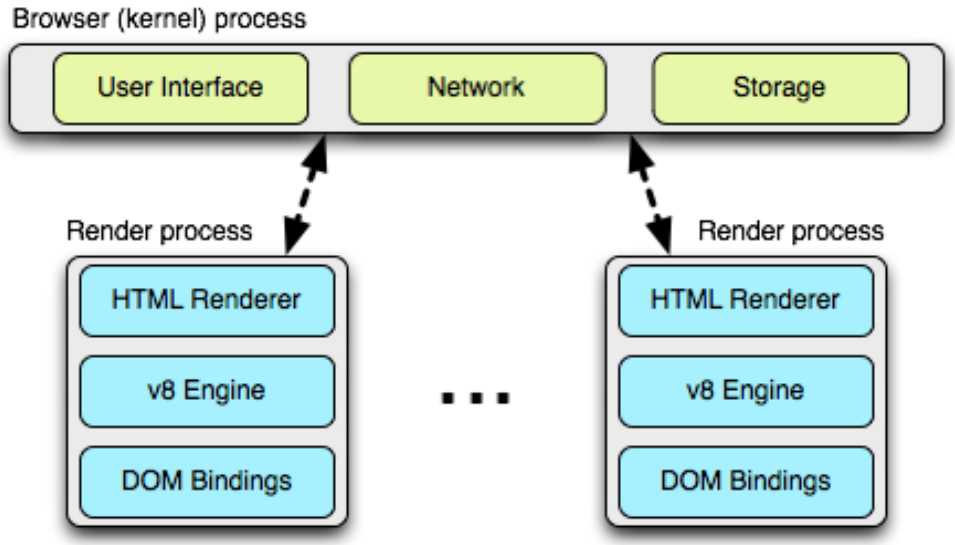
3.4 프로세스간 통신(Interprocess Comminication)

- 병행(concurrent) 프로세스의 종류
  - 독립(Independent) 프로세스:
    - 다른 프로세스의 실행에 영향을 주거나 받지 않음
  - 협력(Cooperating) 프로세스:
    - 다른 프로세스의 실행에 영향을 주거나 받음
- 프로세스 협력을 허용하는 이유
  - 정보 공유 (예) 공유 파일
  - 계산 가속화(Computation Speedup)
    - 작업을 서브 작업으로 나누어, 병렬 실행
  - 모듈성(Modularity) – 기능 모듈화
  - 편의성(Convenience)
    - 개별 사용자가 한 번에 여러 작업을 수행함
      (예) 편집, 프린트, 컴파일을 함께 수행 (병렬)

(예) 다중 프로세스 구조 – Chrome Browser

- 단일 프로세스 방식의 웹 브라우저
  - 한 웹사이트(JS, html5 포함)가 문제가 있으면 전체 브라우저에 영향
- Chrome Browser의 다중 프로세스 구조 – 3 프로세스 사용
  - 브라우저(browser)
  - 렌더러(renderer)
    - 웹사이트마다 별도의 렌더러 실행
  - 플러그인(plug-in)
- 특징과 장점
  - renderers는 sandbox에서 실행되며, 멀티코어에서 병렬 렌더링 지원
  - 웹사이트가 다른 웹사이트와 고립되어 동작

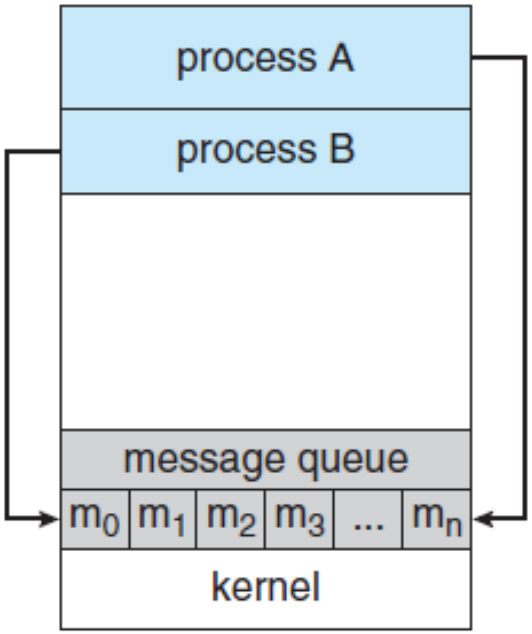
  →문제 발생시 다른 웹사이트의 렌더러에 영향을 주지 않음



sandbox :
디스크와 네트워크 I/O 접근이 제한됨
→ 보안 취약점(exploits)의 영향 최소화

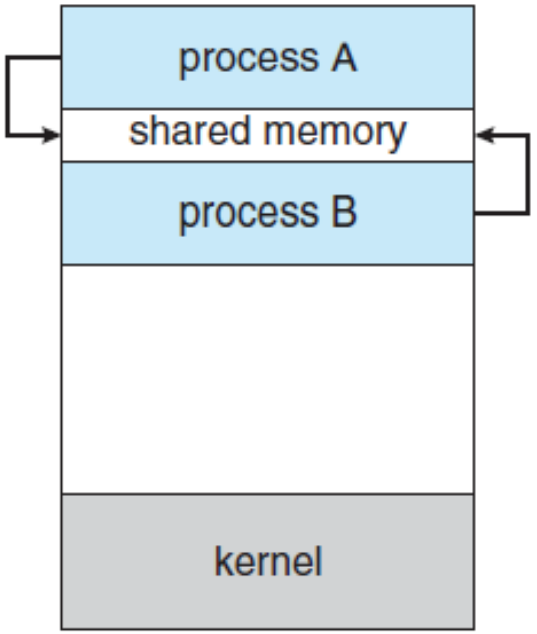
프로세스간 통신(IPC) 모델

Message Passing



(a)

Shared Memory



(b)

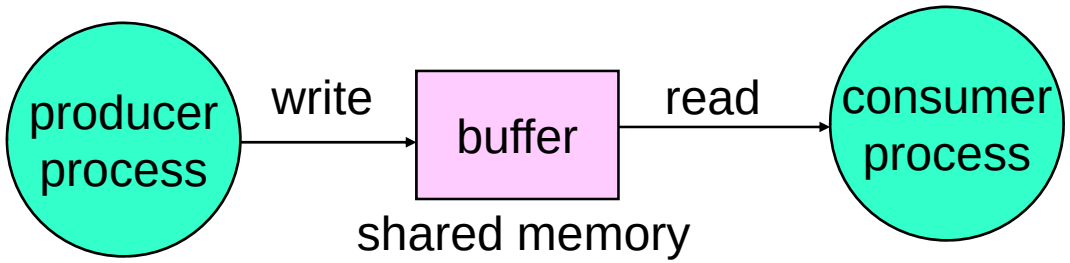
- 적은 양의 데이터 전달에 유용
- 충돌(conflict)이 없음
- 구현이 용이
- 코어 수가 증가할 때 더 나은 성능

- 최대 속도
- 편의성 : 공유메모리 영역 지정할 때에만 시스템 호출. 보통메모리처럼 접근
- 코어 수 증가 시 캐시일관성 문제로 성능저하

많은 운영체제가 두 IPC 모델을 모두 구현함

공유 메모리 시스템

- **Shared Memory Systems** – 공유 영역에 있는 버퍼 사용
  - 시스템 호출을 사용하여 공유 메모리 영역 지정
    - UNIX: shmget(), shmat() – 뒤에서 설명
  - 공유 메모리가 지정되면 보통의 메모리와 같이 접근됨
- 생산자-소비자 문제(Producer-Consumer)
  - 협력 프로세서들의 간단하고 전형적인 예
  - producer 프로세스는 정보를 생산하고, consumer 프로세스는 정보를 소비함
- 생산자-소비자 문제의 **공유 메모리** 구현 방법 – **두 유형의 버퍼**
  - **무한(unbounded) 버퍼** : 버퍼 크기의 제한이 없음
  - **유한(bounded) 버퍼** : 고정된 버퍼 크기
    - 버퍼가 full이면 producer는 기다려야 함



Producer-Consumer 프로그램 – 공유 메모리

- shared buffer

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

producer

```
item next_produced;

while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

consumer

```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

메시지 전달 시스템

- 메시지 전달(Message passing)
  - 같은 주소 공간을 공유하지 않고, 통신을 하고 동작을 동기화하는 기능을 제공함 (네트워크로 연결된 분산환경에 특히 유용)
  - 메시지 전달 IPC 함수 이용
- 메시지 전달 IPC 함수 – 최소 두 가지
  - **send**(*message*)
  - **receive**(*message*)
- 메시지 크기와 IPC 구현

	구현	**프로그래밍**
fixed size:	간단함	더 복잡함
variable size:	더 복잡함	더 간단함

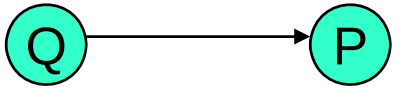
  - 한 번에 전송/수신할 수 있는 메시지 길이는 제한됨. 길이가 길면 제한된 길이로 분할하여 여러 번 전송하거나, 여러 번 수신하여 합해야 함
    - fixed size IPC : 프로그래머가 이 작업을 수행
    - variable size IPC : IPC 함수에서 이 작업을 수행

IPC의 통신 연결(Communication Link)

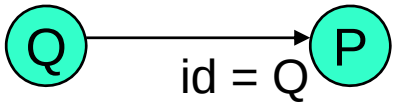
- communication link
  - 통신 프로세스 P와 Q가 서로 통신을 하려면 그들 간에 통신 연결이 설정되어야 함
- communication link의 구현
  - 물리적 구현 : 공유 메모리, 하드웨어 버스, 네트워크
  - **논리적 구현** : 운영체제의 관심 사항
- communication link와 send/receive의 논리적 구현 방법
  - direct / indirect communication → 명명(naming) 방법
    - 상대편을 참조하는(가리키는) 방법
  - synchronous / asynchronous communication
    - send와 receive의 동작 동기화 여부
  - 자동(automatic) / 명시적(explicit) buffering

Naming - Direct Communication

- Direct Communication – 서로의 이름을 명시 (대칭 주소 지정)
  - send (*P, message*) – process P에게 message 전송
  - receive(*Q, message*) –process Q로부터 message 수신



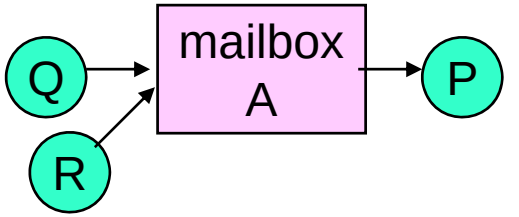
- 비대칭(asymmetric) 주소 지정
  - send(*P*, message)
  - receive(***id***, message) – 임의의 process로부터 message 수신
    id = sending process의 이름
- communication link의 특성
  - 링크가 자동적으로 설정
  - 링크는 두 프로세스들 사이에서만 연관됨
  - 두 프로세스들 사이에 정확히 1개의 링크만 존재

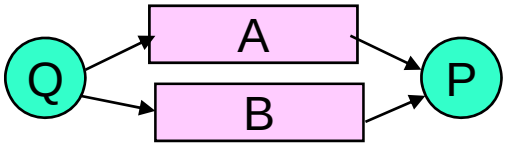


- 문제점
  - 프로세스의 이름을 변경하면, 모든 다른 프로세스 정의에 대한 검사가 필요함 → 해결책: 간접 통신

Naming - Indirect Communication

- Indirect communication – mailbox (또는 port)를 경유하여 message를 송신하거나 수신함
  - 각 메일박스는 고유의 ID를 가짐
  - 프로세스들은 메일박스를 공유할 때에, 메일박스를 통하여 통신
- 메일박스를 사용한 send/receive
  - send(*A*, message): mailbox A로 message 송신
  - receive(*A*, message): mailbox A에서 message 수신
- communication link의 특성
  - 프로세스들이 공유(common) 메일박스를 가질 때에만 통신 링크가 설정됨
  - 링크는 두 개 이상의 프로세스와 연관가능
  - 두 프로세스들 간에 다수의 링크를 공유 가능
- 메일박스 생성 및 소유
  - 운영체제는 메일박스 생성/송수신/삭제 기능
  - 메일박스는 생성한 프로세스의 소유임
  - 운영체제도 자체의 메일박스 소유 가능





동기화(Synchronization)

- **동기식(synchronous) 통신 – Blocking** 송수신

  동작이 상대편 동작에 영향을 받음

  - **Blocking send –** 송신 프로세스는 수신 프로세스나 메일박스가 메시지를 받을 때까지 block 되어 있음
  - **Blocking receive –** 수신 프로세스는 수신 메시지가 있을 때까지 block 되어 있음

- **비동기식(asynchronous) 통신 - Non-blocking** 송수신

  동작이 상대편 동작에 영향을 받지 않음

  - **Non-blocking send** – 송신 프로세스는 메시지를 보내고 바로 return. 작업을 계속 수행함 (수신 여부와 관계없이)
  - **Non-blocking receive** – 수신 프로세스는 유효한 메시지를 받거나 널(null)을 받고 바로 return. 작업을 계속 수행함

- 동기식 통신이 프로그래머가 사용하기가 더 쉽다.

버퍼링(Buffering)

- 메시지 큐
  - 프로세스간에 교환되는 메시지는 링크에 연관된 message queue(버퍼)에 저장되어 전송됨

- buffer queue의 구현 방법
  1. 유한용량(**Bounded capacity)** – 유한한 길이의 버퍼(길이 n)
     - queue가 full이면 sender는 기다려야 함(blocking).
  2. 무한용량(**Unbounded capacity)** – 무한한 길이의 버퍼(이상적)
     - Sender는 결코 기다리지 않음(nonblocking)
  3. 무용량(**Zero capacity)** – 링크에 버퍼가 없음.
     - Sender는 receiver가 준비되어 직접 수신할 때까지 기다려야 함.
       → 랑데부(*rendezvous*).

3.5 IPC 시스템 사례 - UNIX의 공유 메모리 함수

- Shared Memory IPC
  - 공유 메모리 세그먼트 생성, id 반환

```
segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

  - id가 지정하는 공유 메모리를 프로세스에 연결(attach)

```
shared_memory = (char *) shmat(id, NULL, 0);
```

  - 공유 메모리 세그먼트에 읽고, 쓰기 (배열 read/write와 동일)

```
sprintf(shared_memory, "Writing to shared memory");
```

  - 공유 메모리를 프로세스에서 제거(detach)

```
shmdt(shared_memory);
```

- POSIX shared memory IPC
  - shm_open() : 공유메모리 객체 생성
  - ftrunccate() : 객체의 크기 설정
  - mmap() : 공유메모리 객체를 프로세스 메모리 공간 맵핑
  - shm_unlink() : 공유메모리 객체 제거

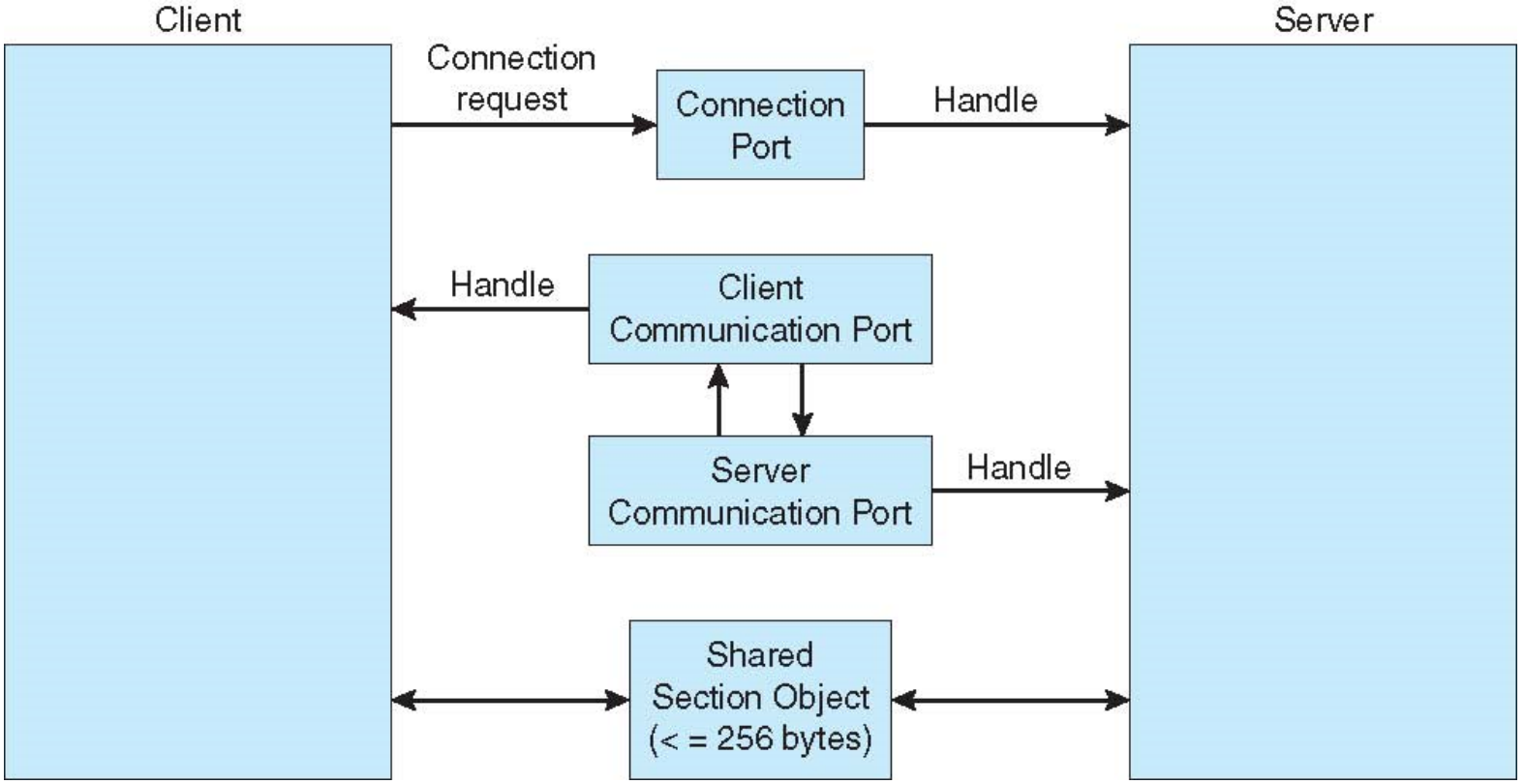
Mach의 message passing

- 메일박스 기반 메시지 전송 – 메일박스를 port라고 부름

- 메시지 전송 시스템 호출
  - msg_send() – 메일박스) 전송
  - msg_receive() – 메일박스 수신
  - msg_rpc() – remote procedure call
  - port_allocate() – 메일박스 생성

Windows의 IPC - Local Procedure Calls

- message passing – port(메일박스와 같음)를 경유
  - Advanced Local Procedure Call(ALPC) 이용



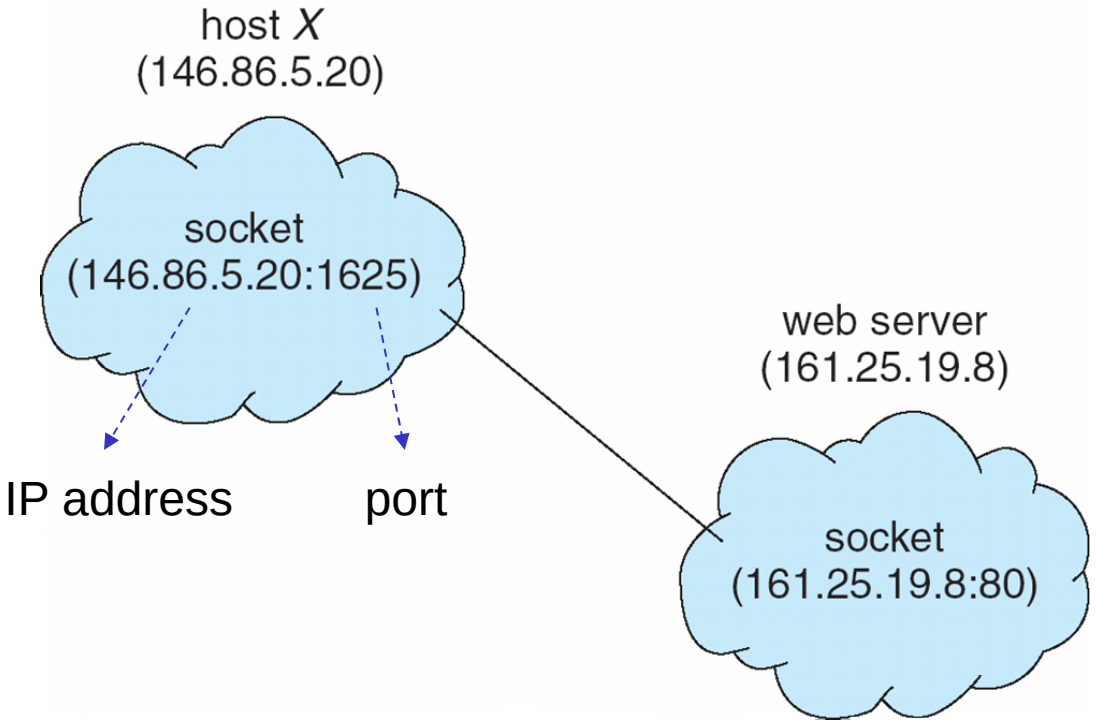
- 작은 메시지: port의 메시지 큐 이용
- 대용량 메시지: 공유 section object 이용
- 매우 큰 메시지: 서버가 클라이언트 주소공간 직접 접근 API 사용

3.6 클라이언트-서버 시스템에서의 통신

- 소켓(Sockets)
- 원격 프로시저 호출(Remote Procedure Calls)
- 파이프(pipes)

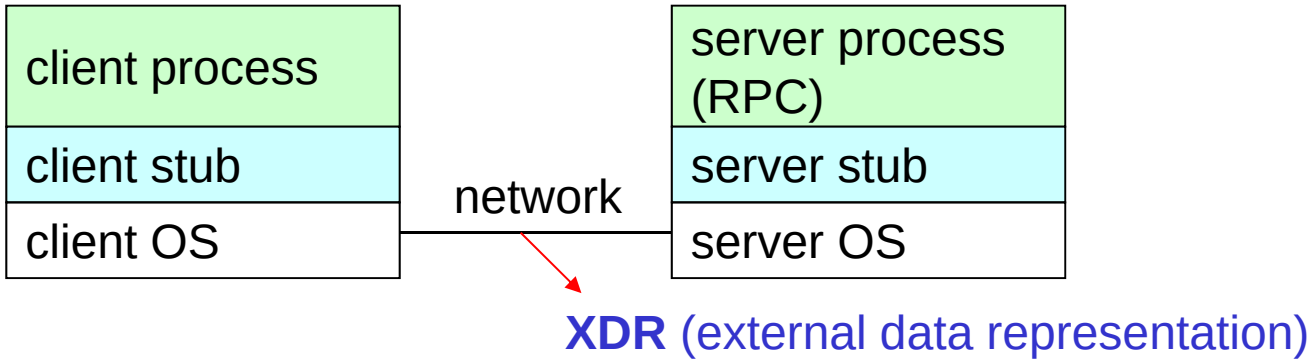
Sockets

- A socket
  - 응용 프로그램 간에 통신이 연결되는 종단점(endpoint)
  - IP 주소와 포트번호 에 의해서 식별됨
    - IP 주소는 시스템을 가리킴,
    - 포트 번호는 해당 시스템의 프로세스와 연결됨



Remote Procedure Calls

- Remote procedure call (RPC)
  - 네트워크에 연결되어 있는 시스템의 프로세스들 간에 procedure calls 을 추상화한 것으로 IPC 기반으로 구현됨
- Stubs
  - 서버의 프로시저에 대한 원격 호출을 대행해주는 프로시저(proxy)



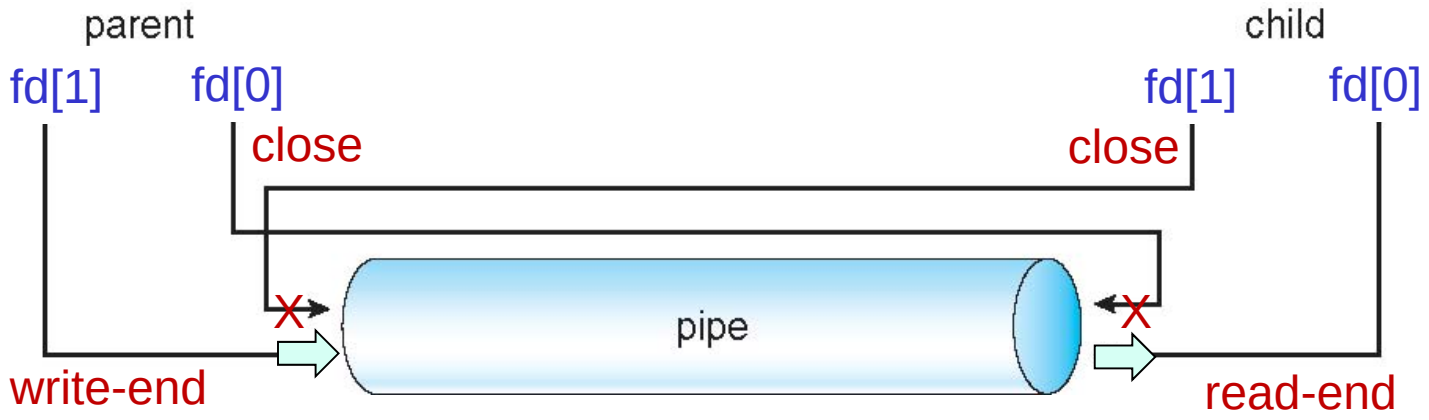
  - The client-side stub – 원격서버의 포트를 찾고, 매개변수를 중립적 표현 방식으로 정돈(marshal)
  - The server-side stub – 메시지 수신, 정돈된 메시지 해제, 서버에서 요청한 프로시저 호출하여 수행하여 결과 전송

Pipes

- 파이프(pipe)
  - 두 개의 프로세스가 서로 통신이 가능하도록 전달자 역할 수행
- 고려사항
  - 통신 방향: 단방향(unidirectional) 또는 양방향(bidirectional)
  - 양방향 통신의 경우: 반이중(half-duplex) 또는 전이중(full-duplex)
  - 통신하는 두 프로세스 간에 특별한 관계(예: 부모-자식) 필요 여부
  - 네트워크 통신 가능 여부
- 일반 파이프(ordinary pipe)
  - 생성한 프로세스만 접근 가능
  - 부모 프로세스가 파이프를 생성하고, 자식 프로세스를 생성하여 파이프를 사용하여 자식 프로세스와 통신
- 지명 파이프(named pipe)
  - 파이프 특성을 가지며, 보통의 파일처럼 존재
  - 지명 파이프를 사용하는 프로세스들은 부모-자식 관계가 필요하지 않음.

일반 Pipes (Windows의 anonymous pipe)

- 생산자-소비자 형태로 두 프로세스 간의 통신을 허용
  - Producer는 한쪽 끝(write-end)으로 쓰기를 함
  - Consumer는 다른쪽 끝(read-end)에서 읽기를 함
- 일반 pipe는 단방향 통신만 가능
- 통신하는 두 프로세스는 부모-자식 관계가 필요함
  - 부모가 생성한 pipe가 자식에게 복제되어 공유됨



fd[0] : read
fd[1] : write

UNIX : pipe() 시스템 호출
Windows : CreatePipe()

지명(Named) Pipes

- 일반 pipe보다 강력한 기능 제공
- 지명 pipe는 양방향 통신 가능
- 부모-자식 관계가 필요 없음
- 여러 프로세스들이 지명 pipe를 사용하여 통신 가능

- UNIX에서는 FIFO 특수 파일로 나타남
  - half-duplex 양방향 전송을 함
  - 동시에 양방향으로 전송하려면 두 개의 FIFO 파일이 필요
  - mkfifo()를 사용하여 FIFO 파일 생성
- Windows는 full-duplex 양방향 전송 가능
  - CreateNamedPipe() 사용하여 생성