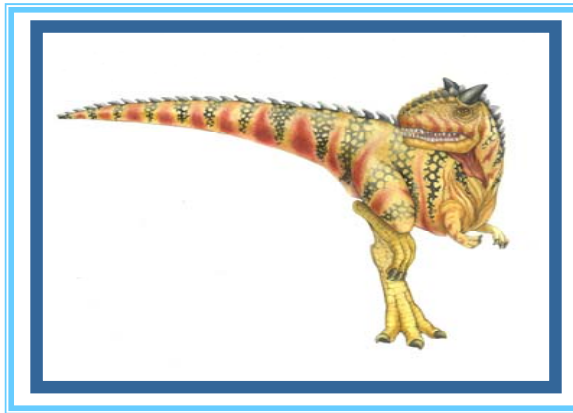


Chapter 3: Processes





Process Concept

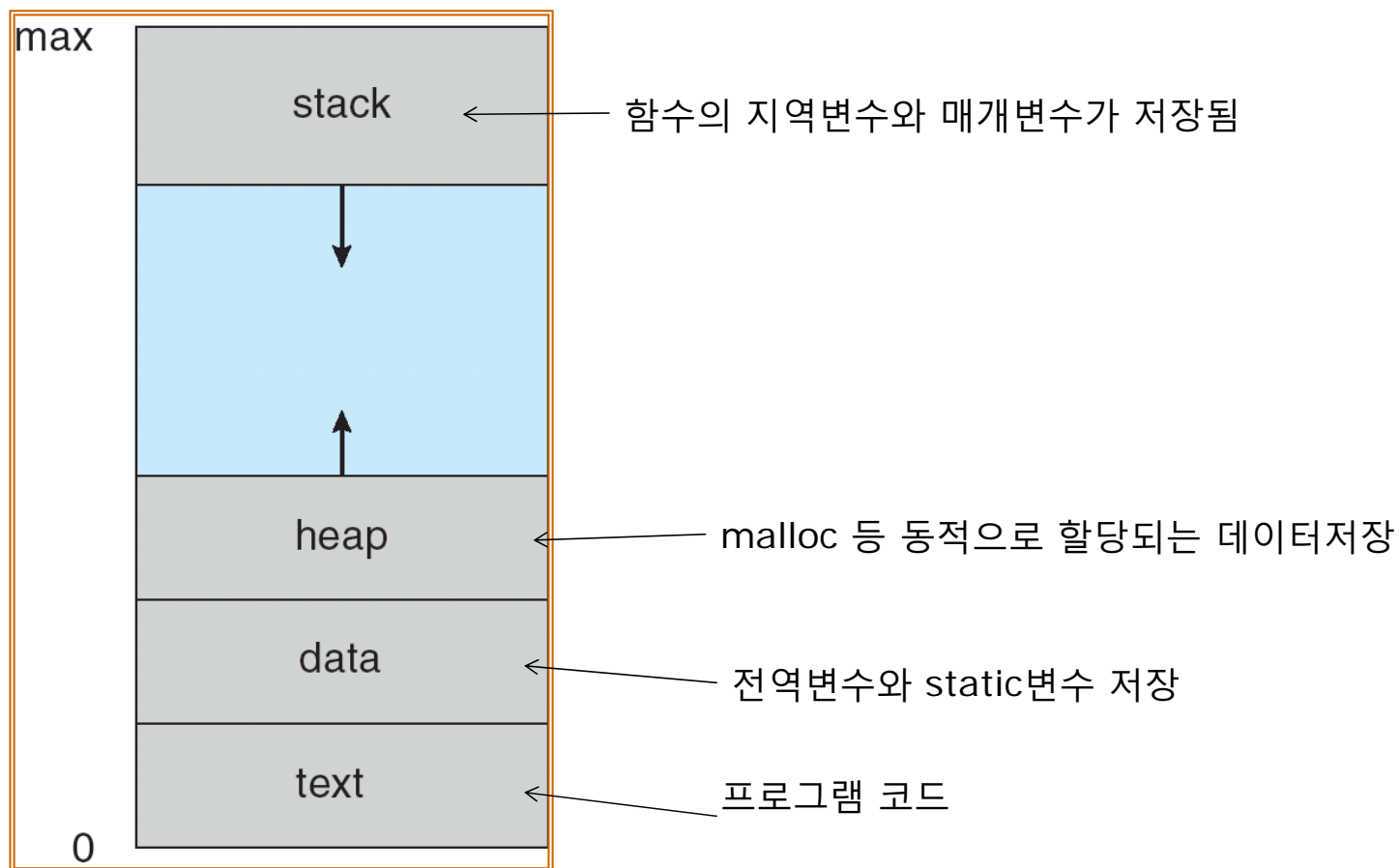
- **Process** : 수행중인 프로그램, 현대 시분할 시스템에서 작업의 단위
- **An operating system executes a variety of programs:**
 - Batch system – jobs
 - Time-shared systems – user programs or tasks
- **Process != Program**
 - **Program**은 디스크에 저장된 파일의 내용과 같은 **passive entity**인 반면
 - **Process**는 실행할 명령어를 저장하는 **program counter**와 연관된 자원의 집합을 가진 **active entity**
- **A process includes:**
 - **program counter**
 - **stack**
 - **data section**

Process vs. Thread?





Process Concept - Process in Memory



PCB : Current activity including **program counter**, processor registers





Process Concept - Process in Memory

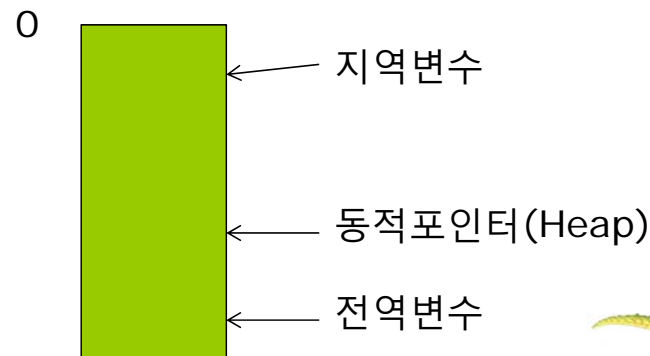
■ Heap 과 Stack

- **Heap : run-time**시에 크기가 결정되는 요소들의 저장공간
 - ▶ **c**의 **malloc()** 함수나 **C++**의 **new** 연산
- **Stack** : 컴파일시에 크기가 결정되어있는 요소들이 저장되는 공간
 - ▶ 함수의 매개변수 지역변수

```
#include <stdio.h>
int A, B;
main()
{
    int a = 0;
    int b = 0;
    int *p1 = NULL;
    int *p2 = NULL;
    p1 = (int*)malloc(sizeof(A));
    p2 = (int*)malloc(sizeof(A));
    printf("전역 변수의 주소값 출력\n");
    printf("%d\n", &A);
    printf("%d\n", &B);
    printf("동적할당된 포인터의 주소값 출력\n");
    printf("%d\n", p1);
    printf("%d\n", p2);
    printf("지역 변수의 주소값 출력\n");
    printf("%d\n", &a);
    printf("%d\n", &b);
    free(p1);
    free(p2);
}
```

[출처] [\[C/C++\]Heap 과 Stack 영역](#) | 작성자 [우기우기](#)

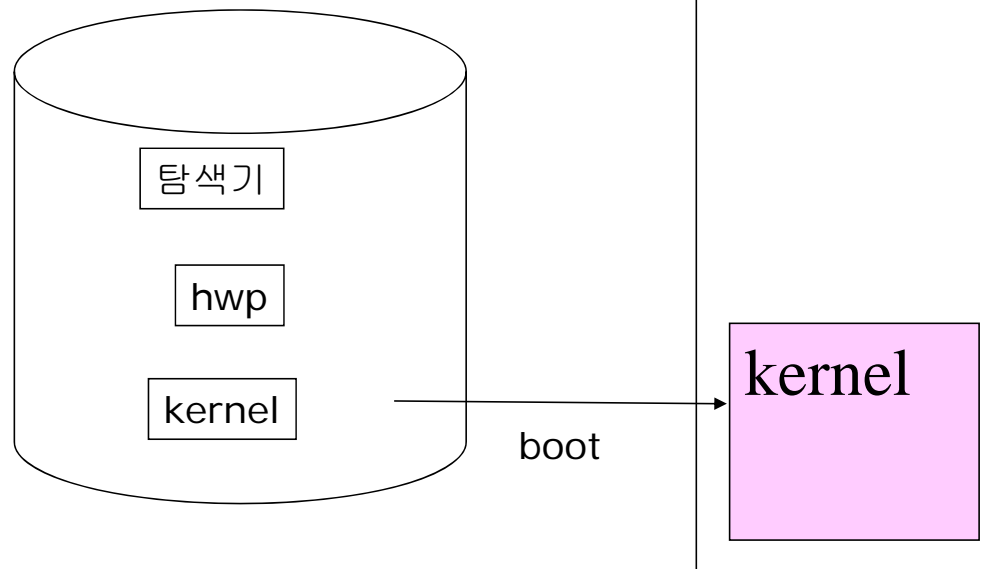
```
전역 변수의 주소값 출력
4359392
4359396
동적할당된 포인터의 주소값 출력
3681776
3681824
지역 변수의 주소값 출력
1244884
1244872
Press any key to continue_
```





Process Concept – Process의 구동(1)

Multi-user System -- multiple shells



or CLI

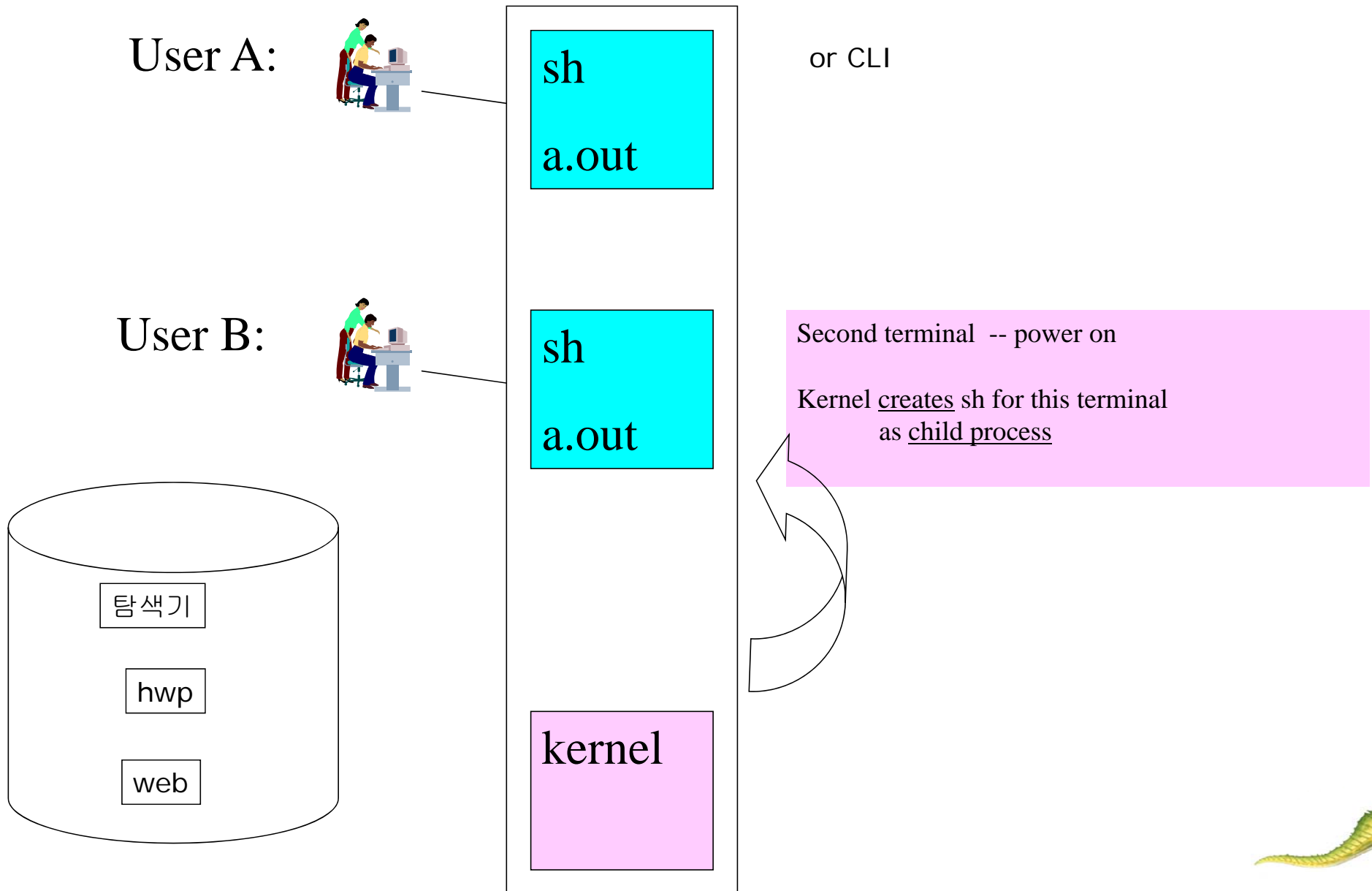
Second terminal -- power on

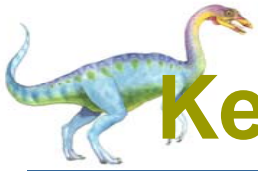
Kernel creates sh for this terminal



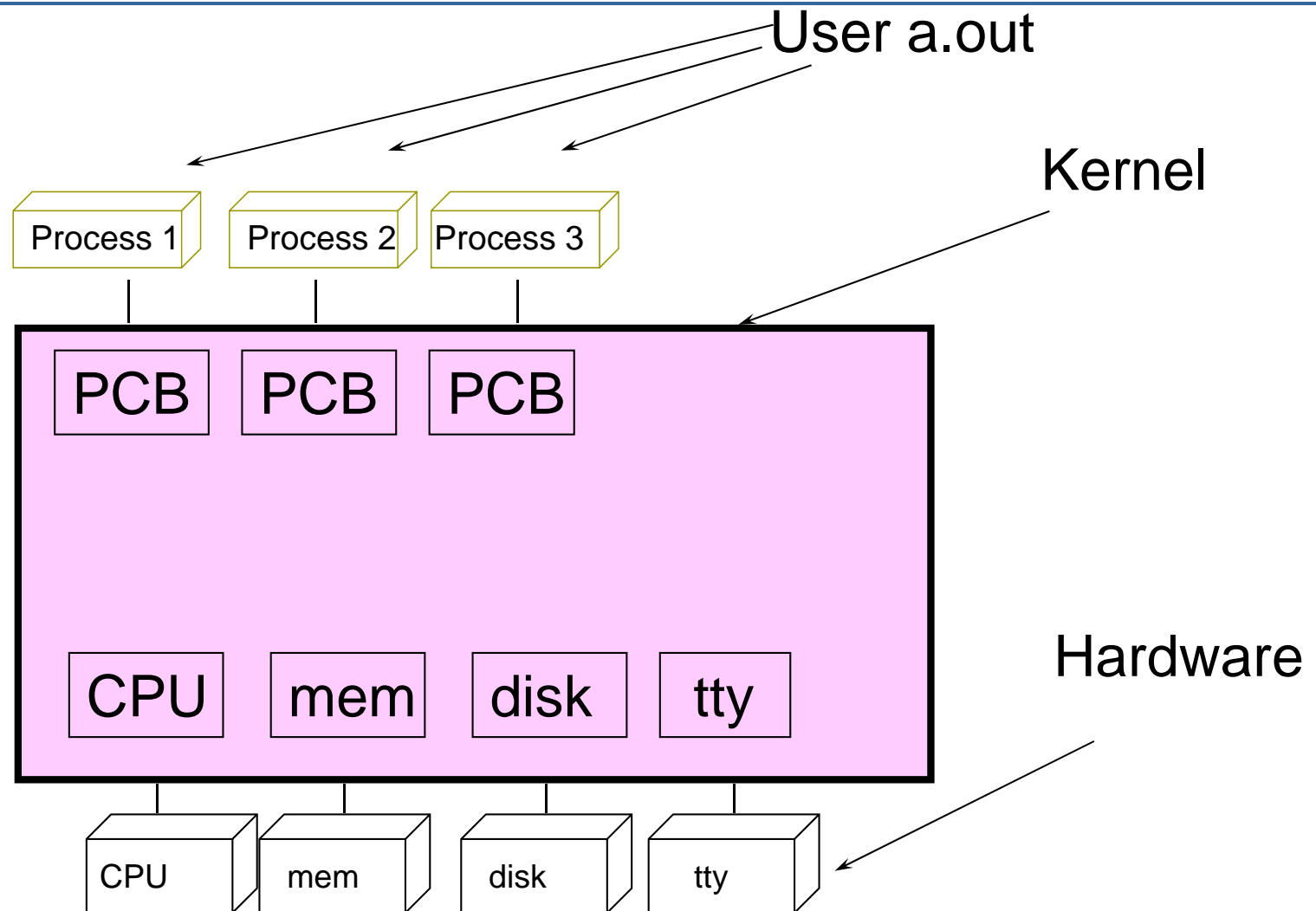


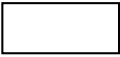
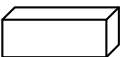
Process Concept – Process의 구동(2)





Kernel 내부에서의 Process(3)

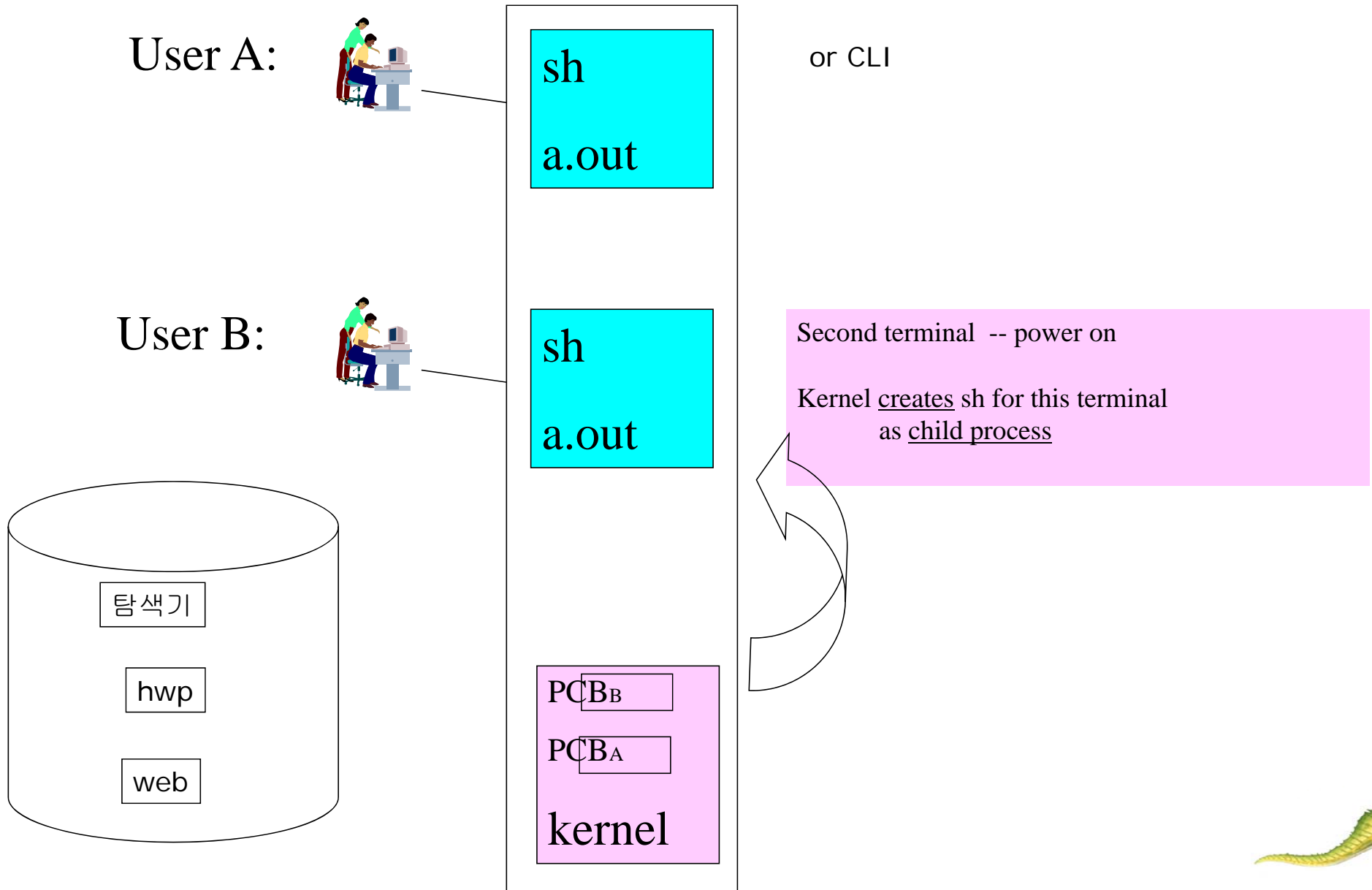


-  : Table (Data Structure)
-  : Object (hardware or software)





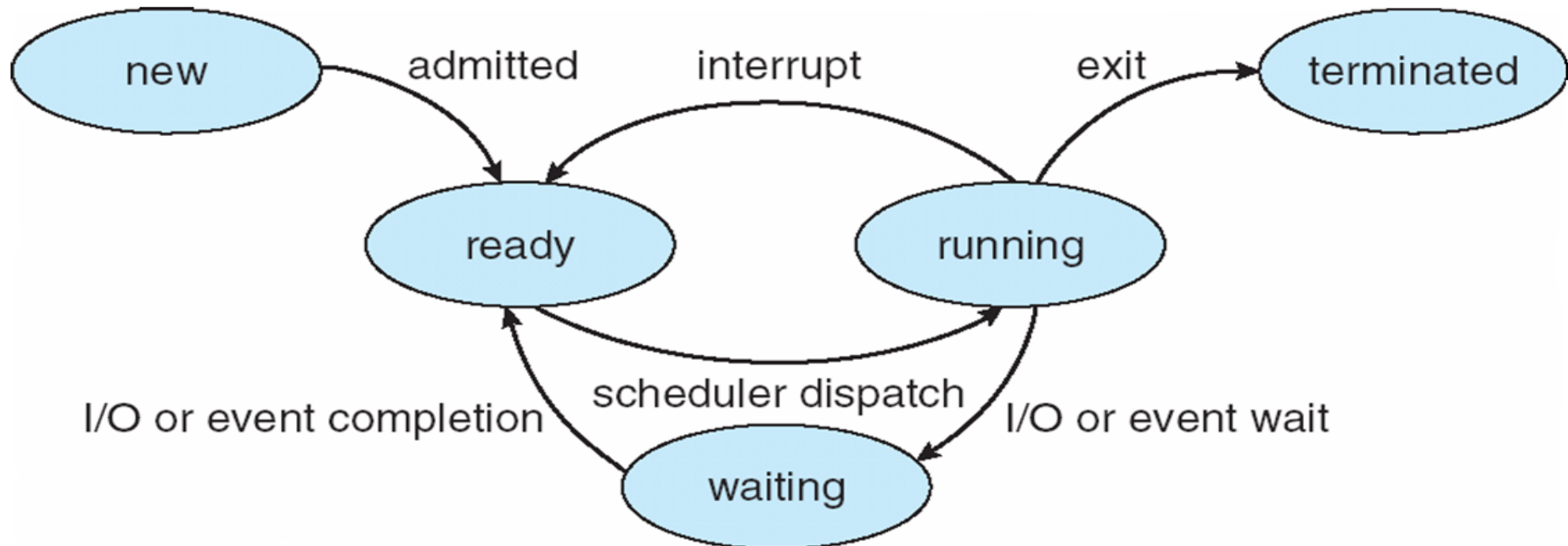
Process Concept – Process의 구동(4)





Process State

- As a process executes, it changes *state*
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished execution



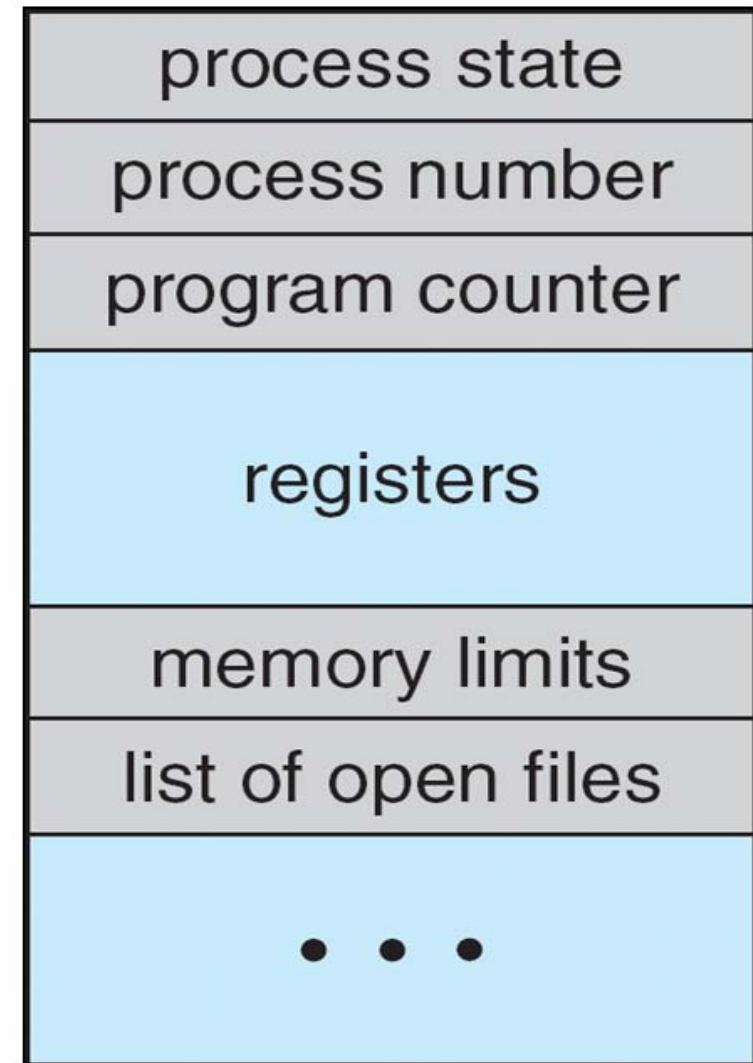


Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter(PC) register
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

PC : instruction pointer, instruction address register





Process Concept – Linux에서의 프로세스 표현

- Linux Process : task_struct
 - <http://www.ibm.com/developerworks/kr/library/l-linux-process-management/index.html>

```
struct task_struct {  
  
    volatile long state;  
    void *stack;  
    unsigned int flags;  
  
    int prio, static_prio;  
  
    struct list_head tasks;  
  
    struct mm_struct *mm, *active_mm;  
  
    pid_t pid;  
    pid_t tgid;  
  
    struct task_struct *real_parent;  
  
    char comm[TASK_COMM_LEN];  
  
    struct thread_struct thread;  
  
    struct files_struct *files;  
  
    ...  
  
};
```

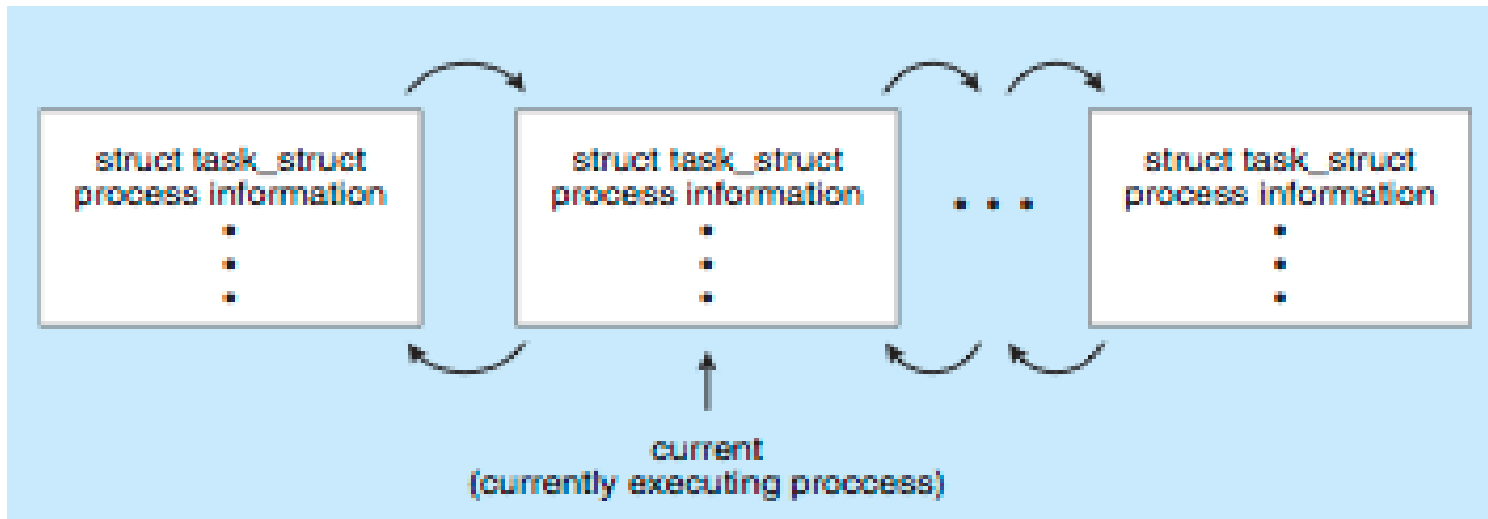
volatile?





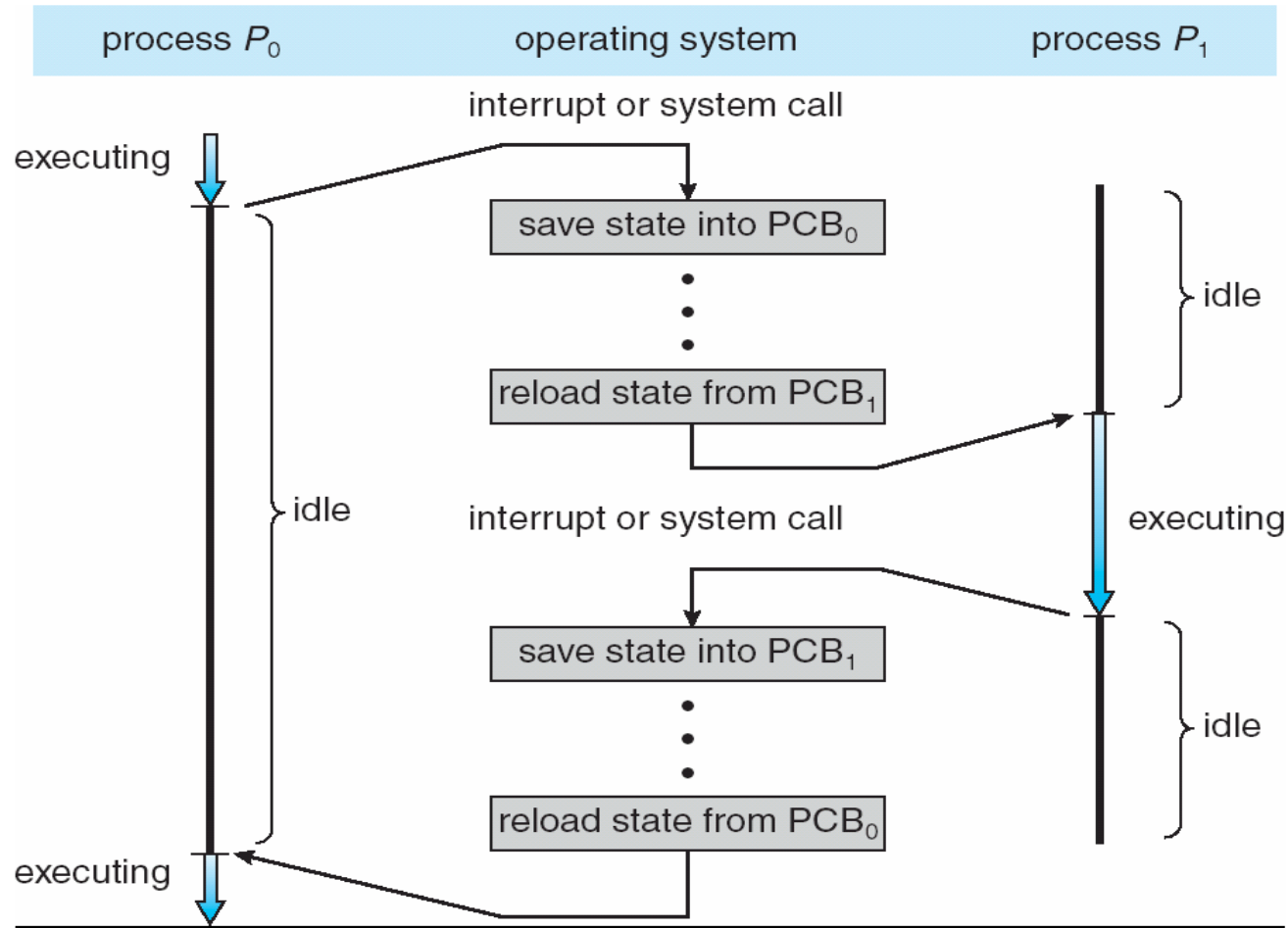
Process Representation in Linux

- **Represented by the C structure** `task_struct`
`pid_t pid; /* process identifier */`
`long state; /* state of the process */`
`unsigned int time_slice /* 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 */`





CPU Switch From Process to Process





Process Scheduling – Process Scheduling Queues

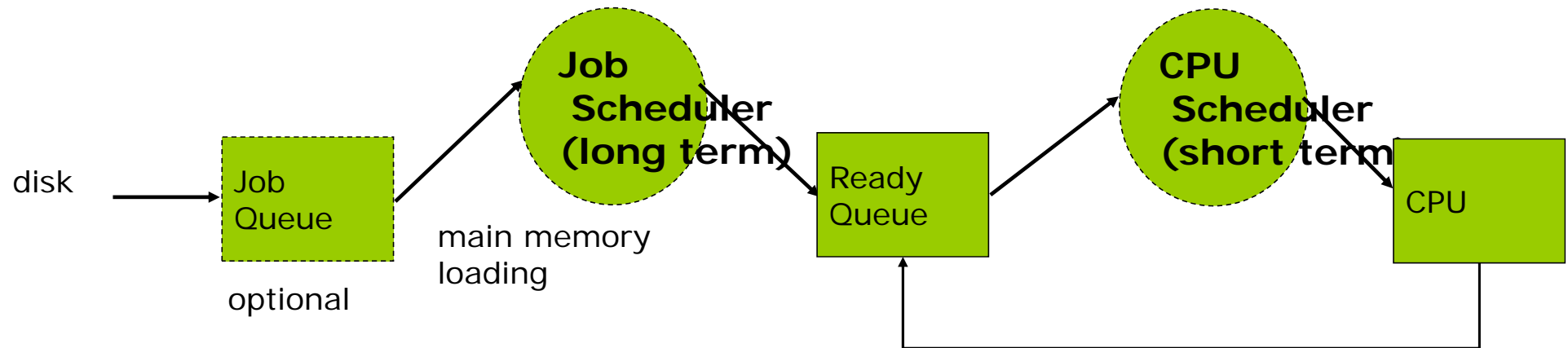
- **Job queue** – set of all processes in the system.
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute.
- **Device queues** – set of processes waiting for an I/O device.
- **Process migration** between the various queues.





Process Scheduling – Schedulers

- Long-term scheduler (or **job scheduler**) – selects which processes should be brought into the ready queue.
- Short-term scheduler (or **CPU scheduler**) – selects which process should be executed next and allocates CPU.



Unix는 Long term Scheduler가 없음 => 물리적인 제한에 의존함





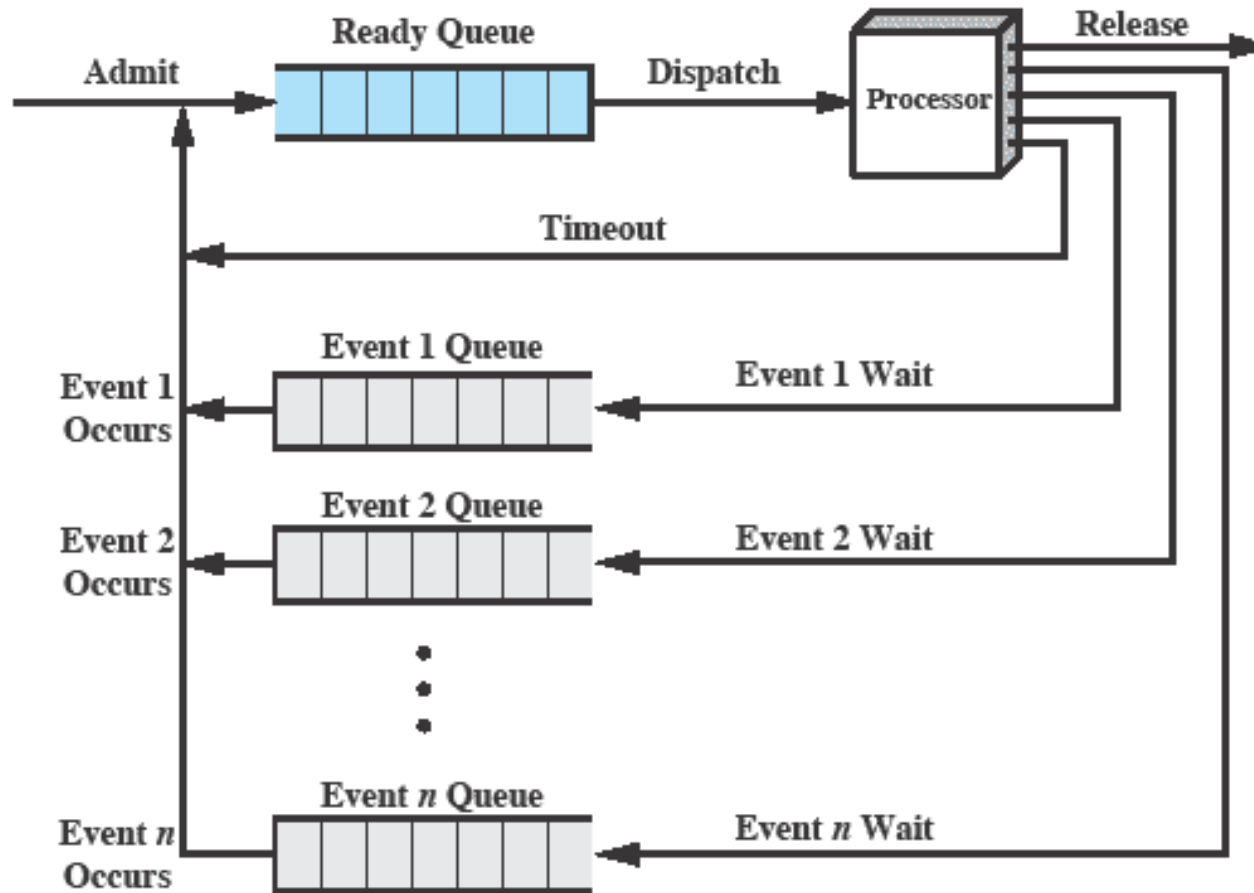
Process Scheduling –Schedulers

- **Long-term scheduler (or job scheduler)** – selects which processes should be brought into the ready queue.
 - ➔ Giving Memory
 - ➔ very infrequently (seconds, minutes)
 - ⇒ (may be slow).
- **Short-term scheduler (or CPU scheduler)** – selects which process should be executed next and allocates CPU.
 - ➔ Giving CPU
 - very frequently (milliseconds) ⇒ (must be fast).
- **Mid-term scheduler – Swapping**의 고려





Process Scheduling

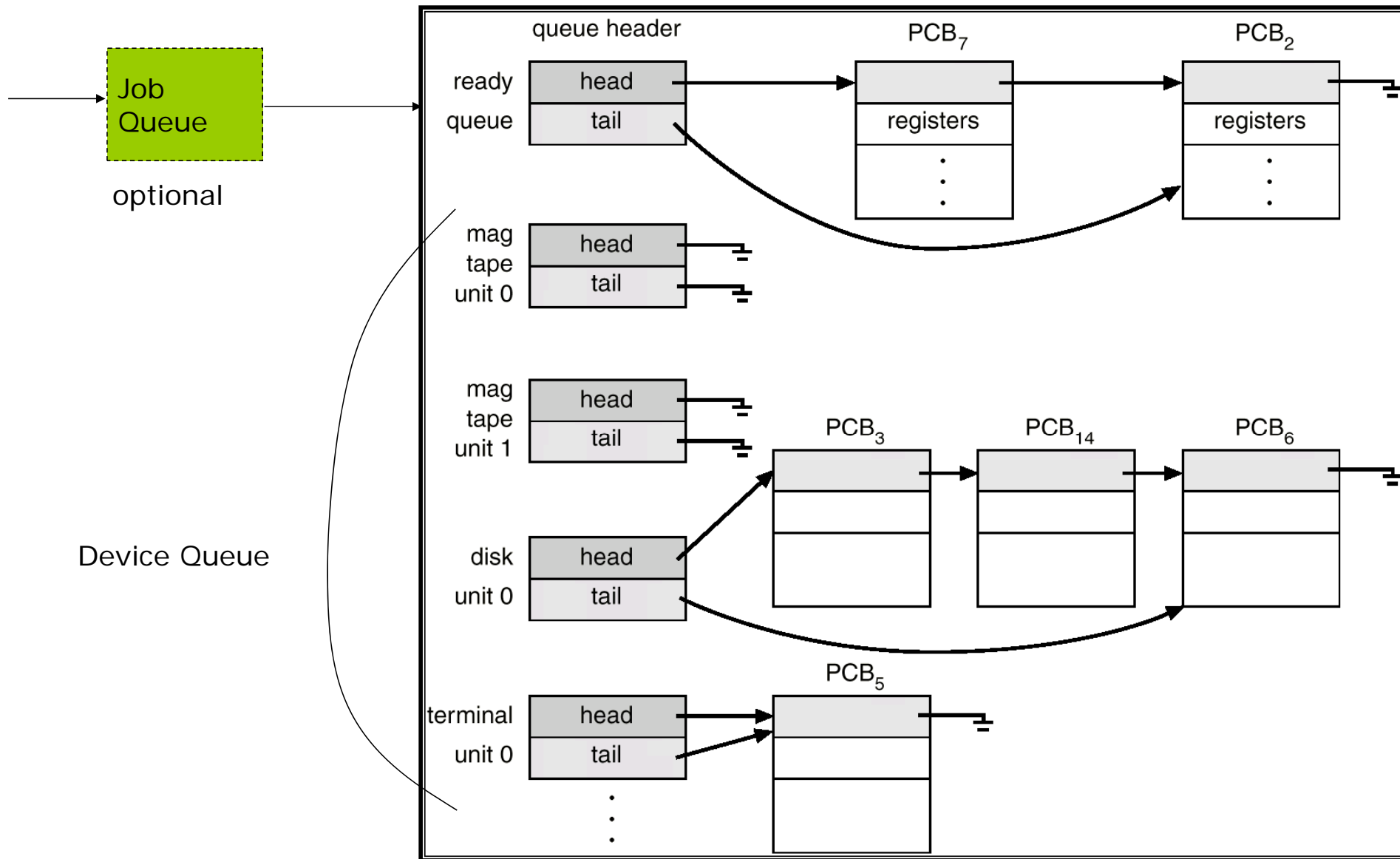


(b) Multiple blocked queues



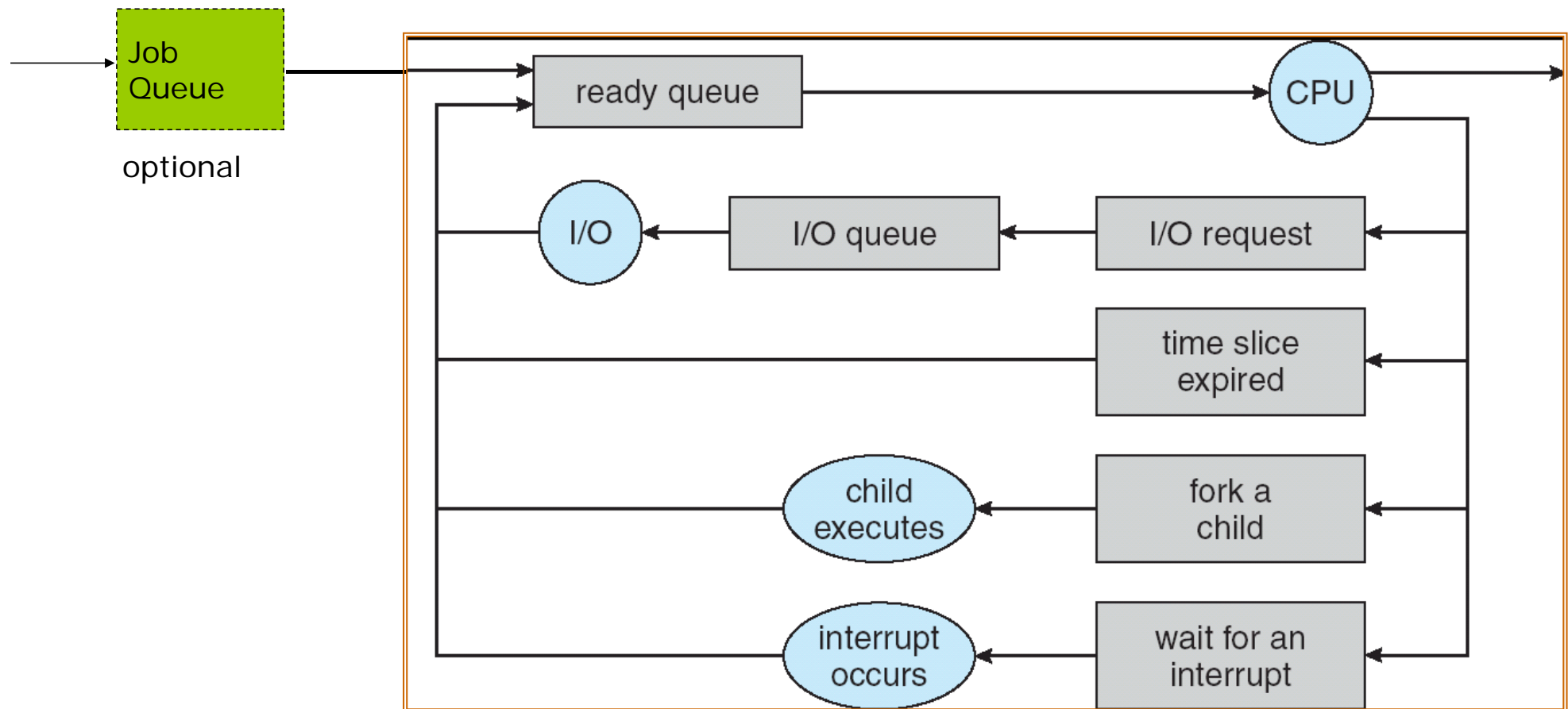


Process Scheduling – Ready Queue And Various I/O Device Queues



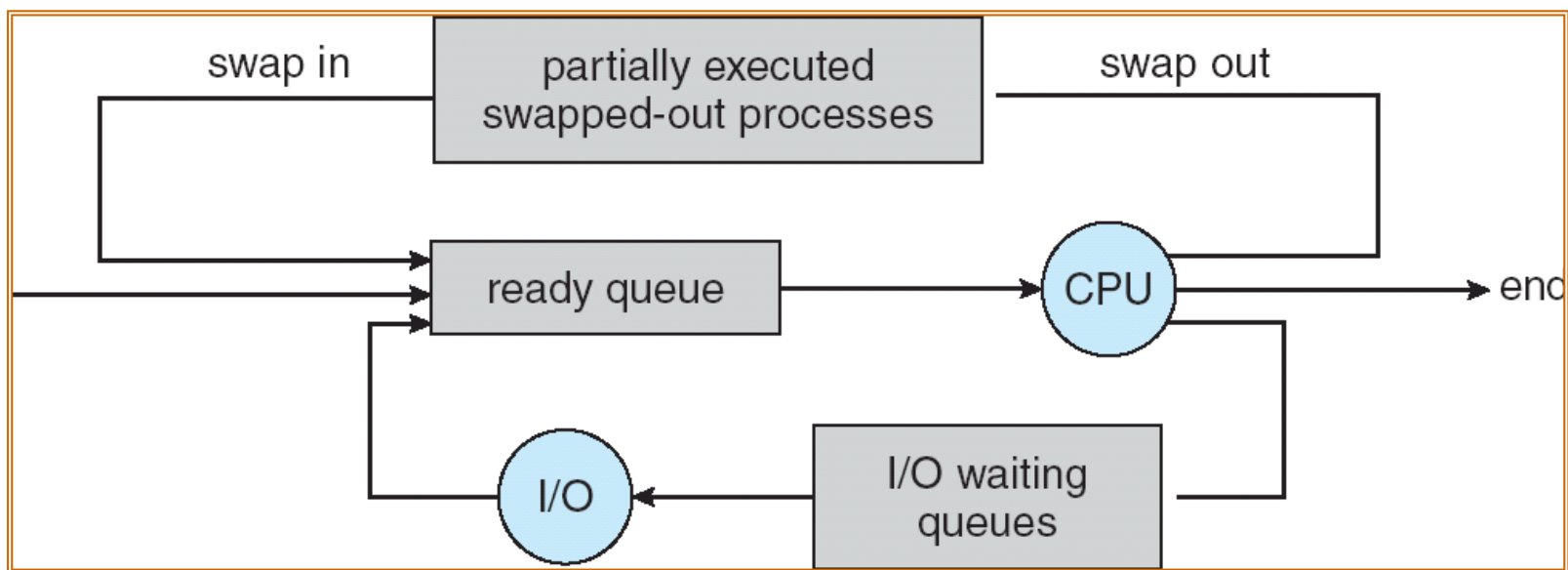


Process Scheduling – Representation of Process Scheduling





Process Scheduling – Addition of Medium Term Scheduling



Medium Term Scheduler : 메모리에서 CPU를 위해 적극적으로 경쟁하는 프로세스들을 일시적으로 제거하여 multiprogramming의 정도를 완화하도록 함 => Swapping





Process Scheduling –Schedulers (Cont.)

- Processes can be described as either:
 - ***I/O-bound process*** – spends more time doing I/O than computations, many short CPU bursts.
 - ***CPU-bound process*** – spends more time doing computations; few very long CPU bursts.





Context Switch

- 컨텍스트 스위치(**Context Switch**) :
 - 하나의 프로세스가 **CPU**를 사용 중인 상태에서 다른 프로세스가 **CPU**를 사용하도록 하기 위해, 이전의 프로세스의 상태(문맥)를 보관하고 새로운 프로세스의 상태를 적재하는 작업
 - **Context** of a process represented in the PCB
- **Context Switching Overhead** : 문맥을 교환하는 동안에는 다른 작업을 수행할 수 없기 때문에, 문맥 교환 시간은 일종의 **오버헤드**
 - 복잡한 **OS**와 **PCB** -> longer the context switch
 - **RISC**가 **CISC**보다 **register** 가 많으므로 상대적으로 **overhead**가 큼

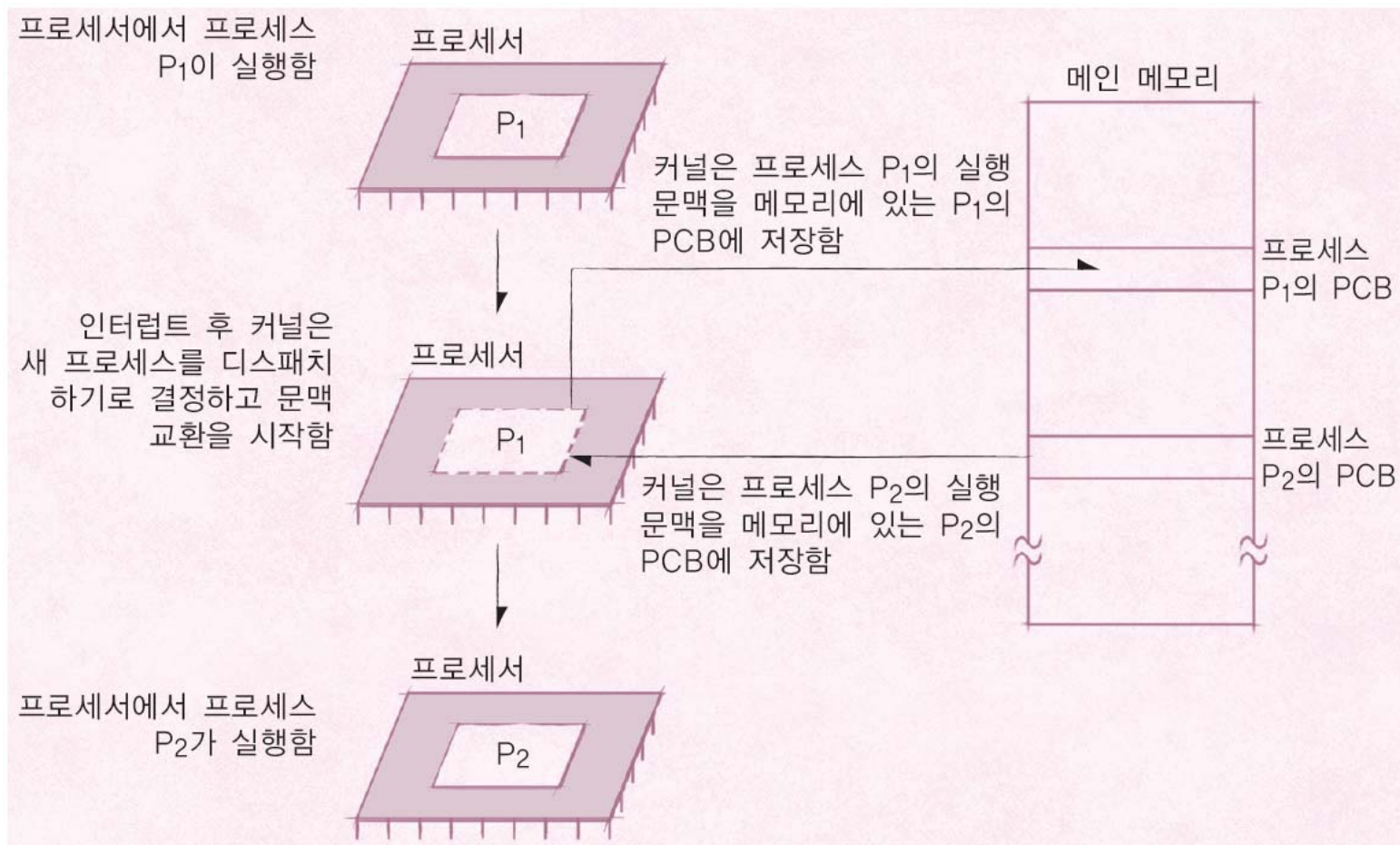
효율적인 Context Switch

=> Sun Ultra SPARC의 경우 여러 개의 레지스터 집합을 제공하여 메모리 및 CPU의 overhead를 감소시킴





Context Switch



[그림 3-6] 문맥 교환

참조 : 한빛미디어 운영체제론

3.23





Context Switch

■ Context Switch가 발생하는 경우

- 실행 중인 프로세스 인터럽트
 - ▶ 입출력 인터럽트 : 프로세스 준비 상태로 바꿈(실행 프로세스 결정)
 - ▶ 클럭 인터럽트 : 할당 시간 조사
프로세스를 준비 상태로 변환하고 다른 프로세스를 디스패치
- 운영체제에서 인터럽트 발생
 - ▶ 실행 프로세스와는 별도로 외부에서 발생하는 여러 종류의 이벤트 (예 입출력 동작의 종료)에 의해 발생
- 트랩(trap) 발생
 - ▶ 부적절한 파일 접근
 - ▶ 실행 프로세스에 의해 발생하는 오류나 예외 상황 때문에 발생
- 하드웨어 인터럽트 처리

인터럽트 처리 루틴으로 제어 이동

⇒ 인터럽트 형태에 따라 관련된 운영체제 루틴으로 분기





Process Creation

- 일반적으로 프로세스는 프로세스 식별자 pid에 의해 식별되고 관리됨
- Resource sharing
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution
 - Parent and children execute concurrently
 - Parent waits until children terminate





Process Creation

■ ps -eal in linux

process id

parent process id

thread : light-weight process

```
max@linux:~$ ps -elfcL
F S UID      PID     PPID     LWP  NLWP  CLS  PRI  ADDR  SZ  WCHAN    STIME  TTY      TIME  CMD
4 S root        1      0        1      1  TS   23   -   147  schedu Oct12 ?   00:00:04  init [5]
1 S root        2      1        2      1  TS    5   -    0  ksofti Oct12 ?   00:00:00  [ksoftirqd/0]
1 S root        3      1        3      1  TS   34   -    0  worker Oct12 ?   00:00:00  [events/0]
1 S root        4      3        4      1  TS   34   -    0  worker Oct12 ?   00:00:00  [khelper]
1 S root        5      3        5      1  TS   26   -    0  worker Oct12 ?   00:00:00  [kacpid]
1 S root       36      3       36      1  TS   34   -    0  worker Oct12 ?   00:00:05  [kblockd/0]
1 S root       56      3       56      1  TS   24   -    0  pdflus Oct12 ?   00:00:02  [pdflush]
1 S root       58      3       58      1  TS   32   -    0  worker Oct12 ?   00:00:00  [aio/0]
1 S root       57      1       57      1  TS   24   -    0  kswapd Oct12 ?   00:00:08  [kswapd0]
1 S root     1401      1    1401      1  TS   21   -    0  serio_ Oct12 ?   00:00:00  [kseriod]
1 S root     1740      3    1740      1  TS   34   -    0  worker Oct12 ?   00:00:00  [reiserfs/0]
1 S root     2989      1    2989      1  TS   24   -    0  hub_th Oct12 ?   00:00:00  [khubd]
4 S root     3508      1    3508      1  TS   23   -   358  msgrcv Oct12 ?   00:00:00  [hwscand]
<!-- output omitted -->
5 S root     5104      1    5104      6  TS   23 - 10630  schedu Oct12 ?   00:00:00  /usr/sbin/nscd
1 S root     5104      1    5105      6  TS   23 - 10630  schedu Oct12 ?   00:00:00  /usr/sbin/nscd
1 S root     5104      1    5106      6  TS   16 - 10630  schedu Oct12 ?   00:00:00  /usr/sbin/nscd
1 S root     5104      1    5107      6  TS   15 - 10630  schedu Oct12 ?   00:00:00  /usr/sbin/nscd
1 S root     5104      1    5110      6  TS   14 - 10630  schedu Oct12 ?   00:00:00  /usr/sbin/nscd
1 S root     5104      1    5111      6  TS   14 - 10630  schedu Oct12 ?   00:00:00  /usr/sbin/nscd
<!-- output omitted -->
0 R max      6479     6417    6479      1  TS   19   -   595   -      09:17 pts/4    00:00:00  ps -elfcL
max@linux:~$
```

name service
cache daemon





Process Creation (Cont.)

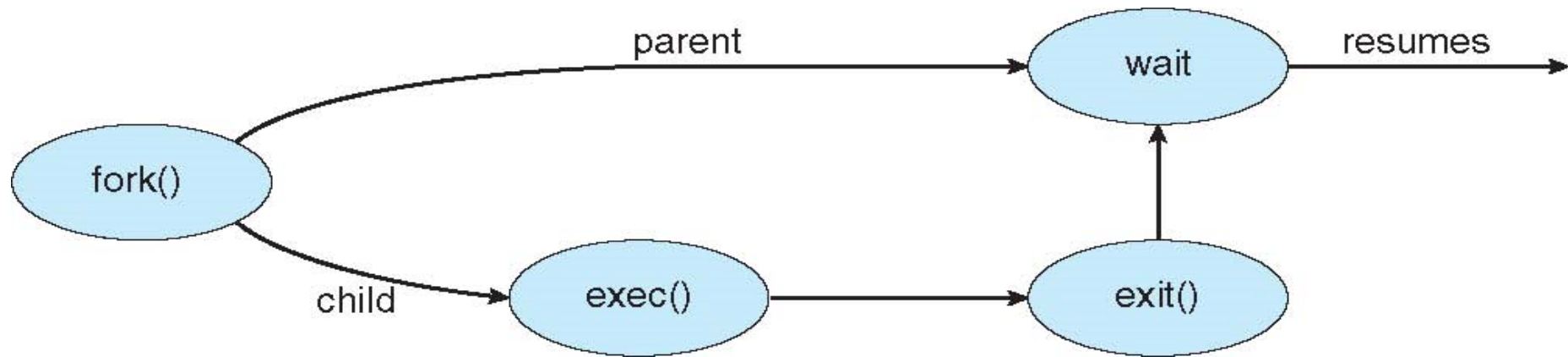
- Address space
 - Child duplicate of parent
 - Child has a program loaded into it

- UNIX examples
 - **fork** system call creates new process
 - **exec** system call used after a fork to replace the process' memory space with a new program





Process Creation





C Program Forking Separate Process

C Program Forking Separate Process

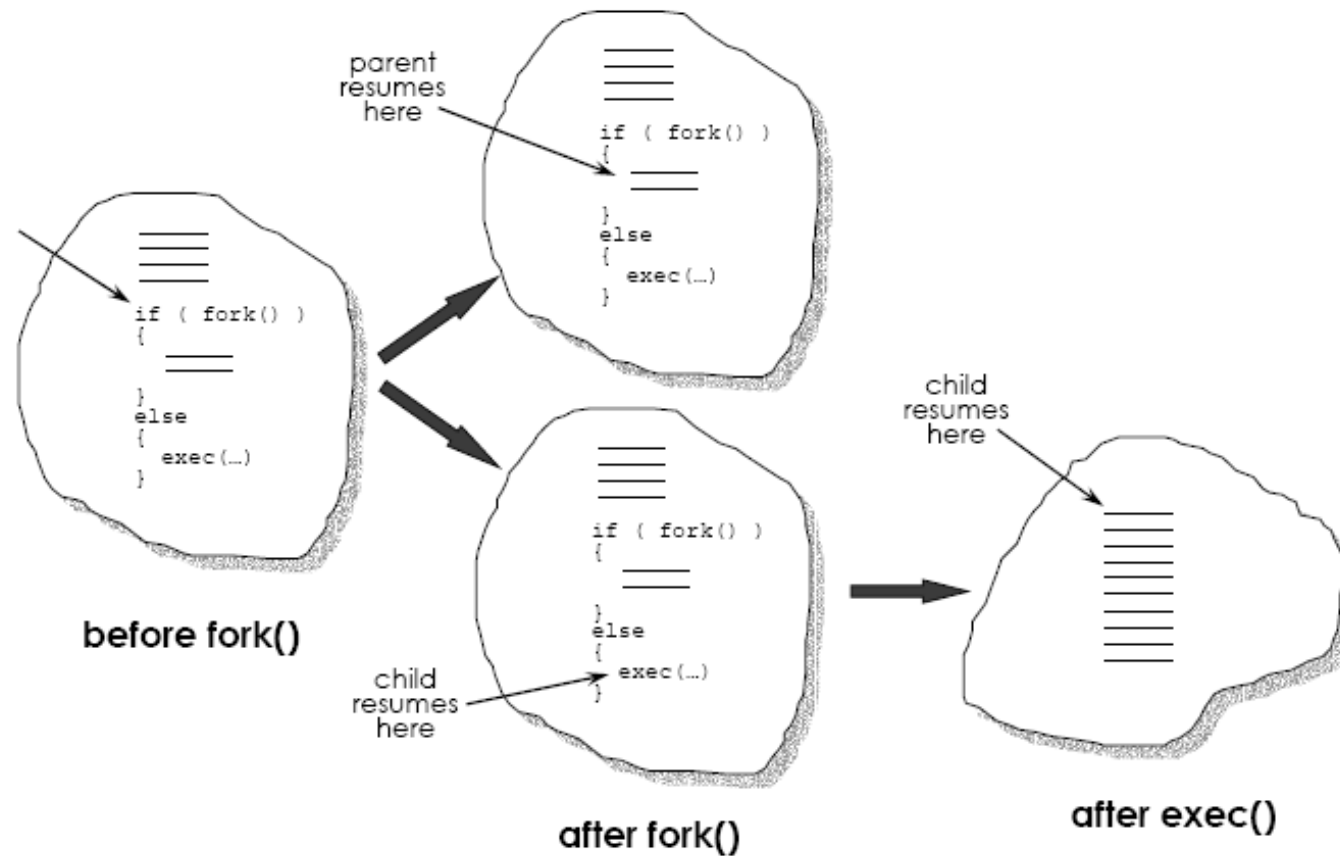
```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/lis", "lis", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```





Operation on Process – Process Creation

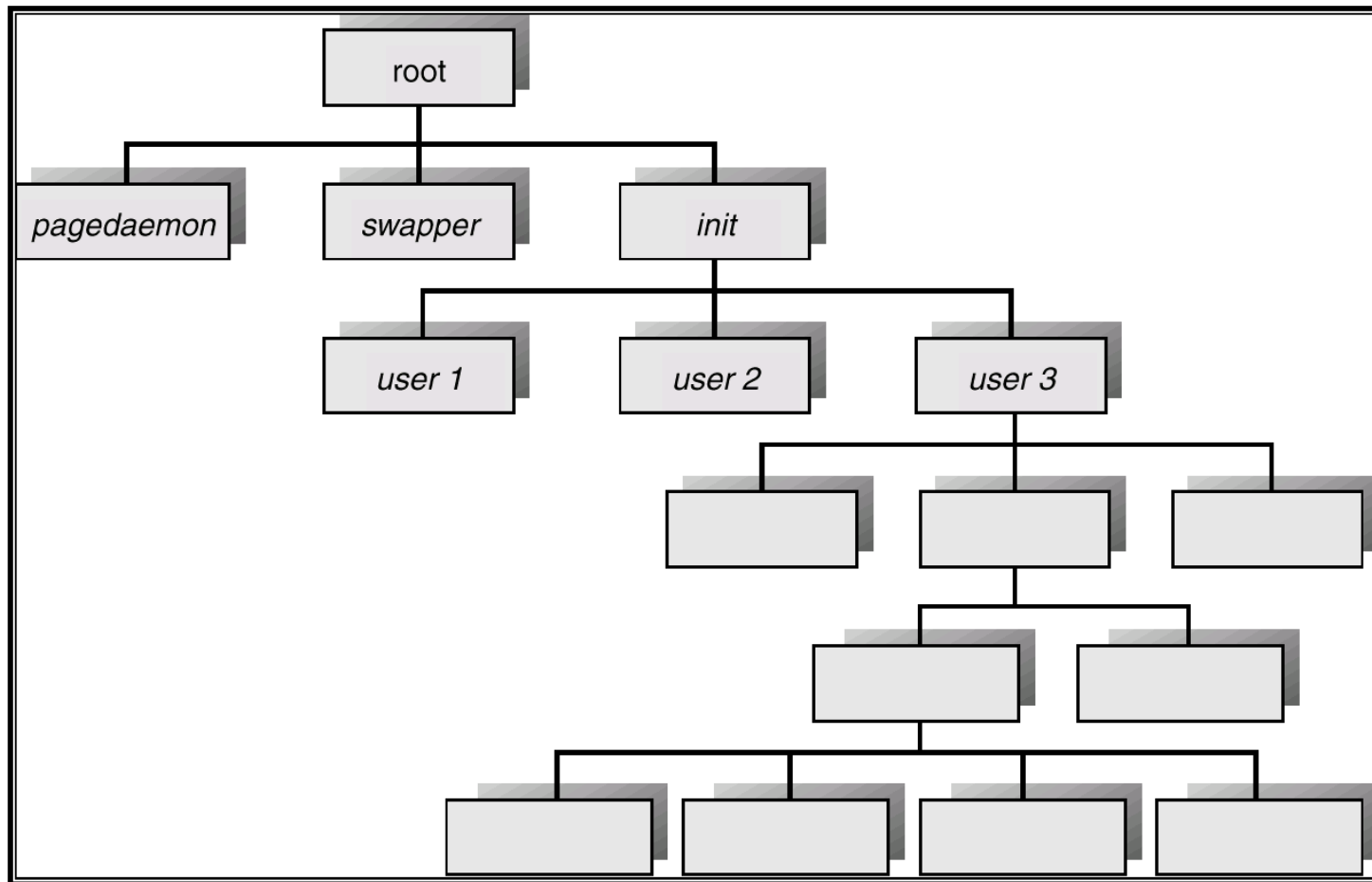
- C Program Forking Separate Process
 - Fork – Before and After





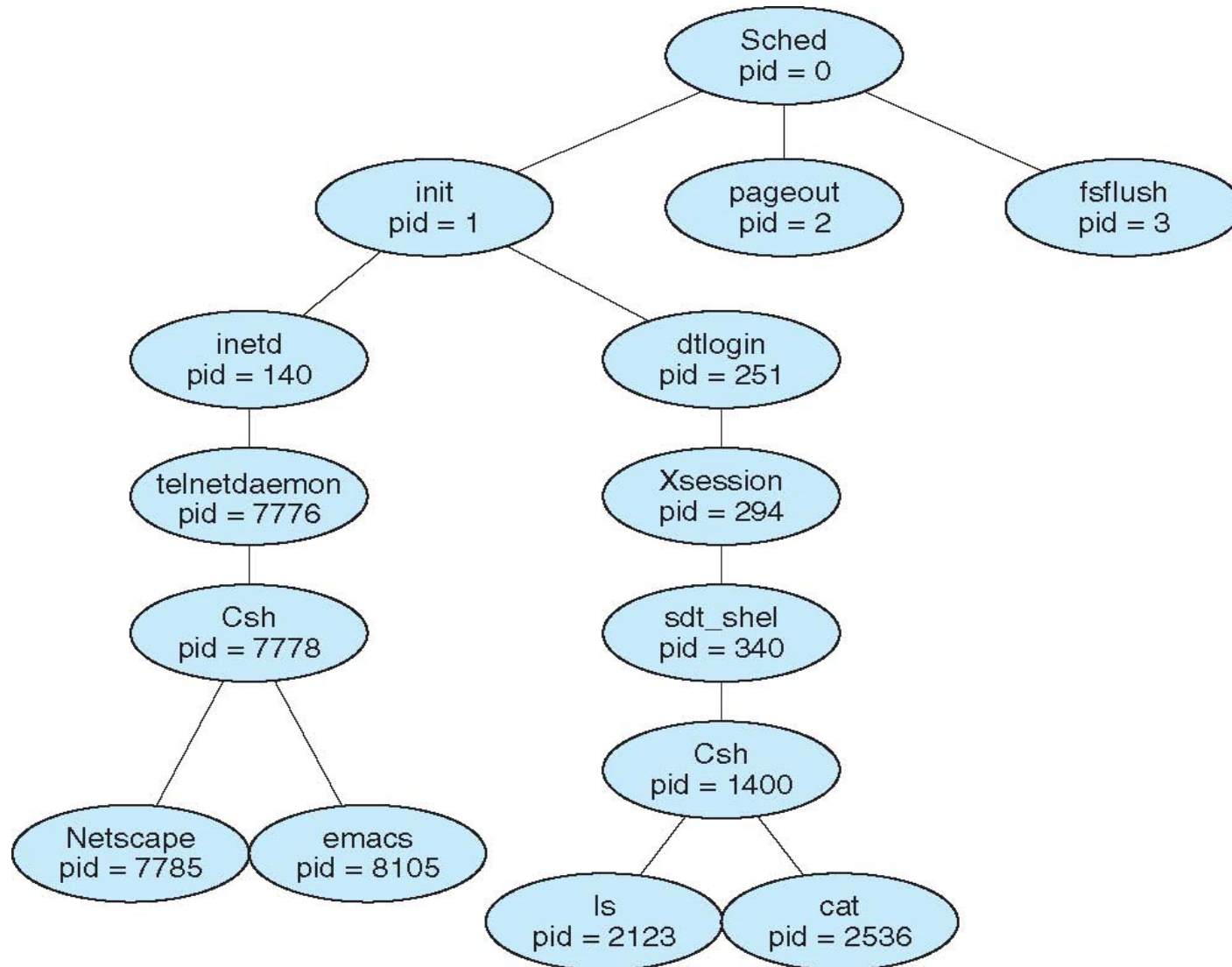
Operation on Process – Process Creation

Processes Tree on a UNIX System





A Tree of Processes on Solaris





Operation on Process – Termination

- Process executes last statement and asks the operating system to decide it (exit).
 - Output data from child to parent (via wait).
 - Process' resources are deallocated by operating system.

- Parent가 children processes을 종료할 수 있음
 - 자식이 자신에게 할당된 자원을 초과하여 사용할 경우
 - 자식에게 할당된 **task**가 더 이상 필요없을 때
 - Parent가 종료되면서 **child**가 계속되는 것을 허용하지 않을 경우 (**Cascading termination.**)





Operation on Process – Suspend

■ Waiting State (Suspend, Resume)

- 프로세스 일시 정지
 - 아직 소멸되지는 않고 프로세서를 차지하려고 하는 경쟁 대열에서 무기한으로 배제
 - 악성 코드 실행과 같은 보안 위협 요인을 추적하거나 디버깅할 때 유용
 - 일시 정지는 해당 프로세스 혹은 다른 프로세스에 의해 발생
 - 일시 정지 블록 상태 프로세스는 다른 프로세스에 의해 재 시작 가능
 - 일시 정지 상태
 - 일시 정지 준비(**suspendedready**)
 - 일시 정지 블록(**suspendedblocked**)





Interrupt(인터럽트)

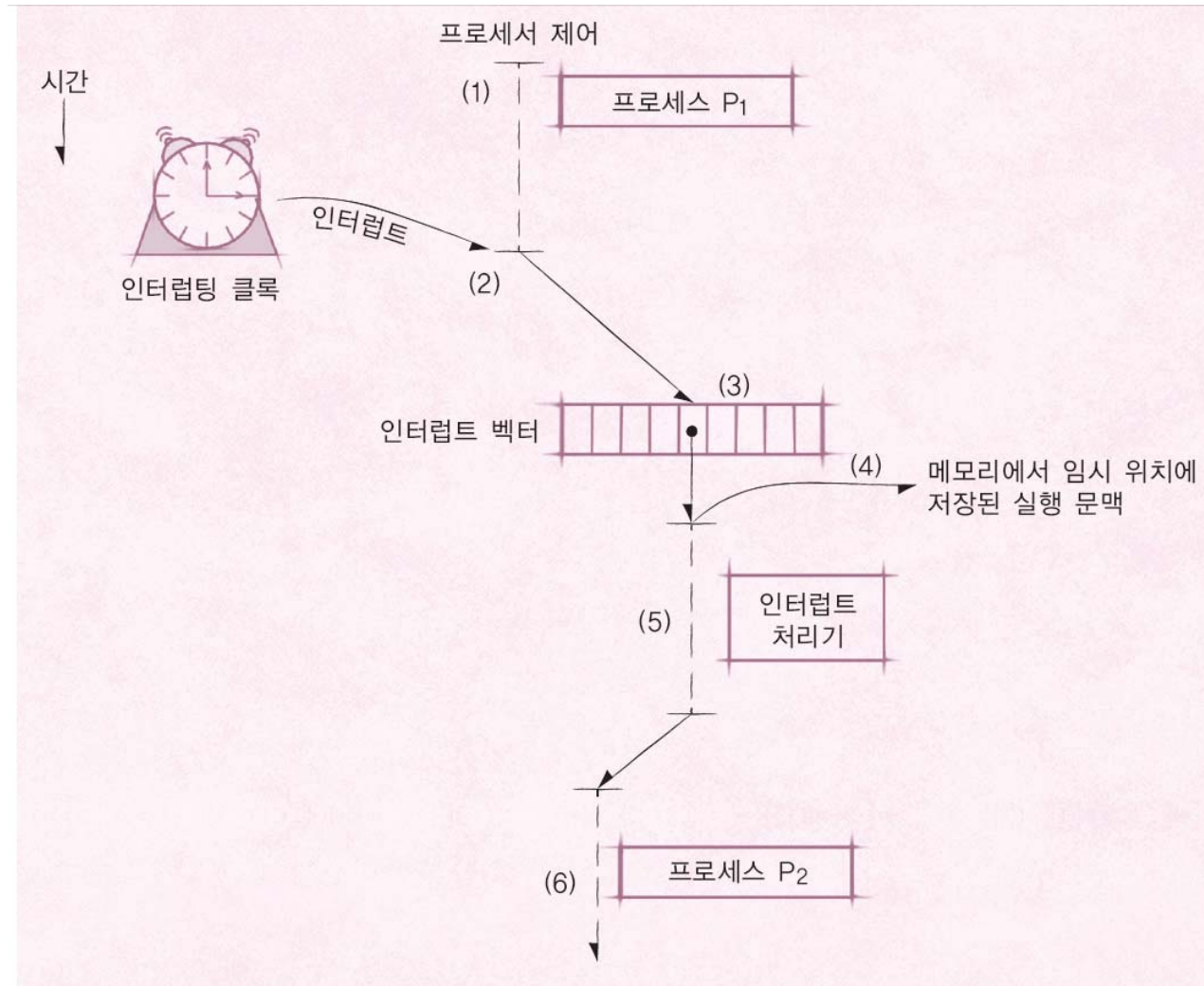
- 인터럽트는 소프트웨어가 하드웨어로부터 오는 신호에 반응할 수 있게 함
 - 프로세서는 프로세스의 명령어를 실행한 결과로 인터럽트를 발생
 - ▶ 트랩(trap)
 - ▶ 프로세스의 작동과 동기(synchronous)
 - 예를 들어, 0을 나누거나 보호되는 메모리 위치를 참조하려고 하는 경우
 - 프로세서의 현재 명령어와 관련 없는 이벤트에 의해서도 인터럽트 발생
 - ▶ 프로세스 작동과 비동기(asynchronous)
 - ▶ 예를 들어, 사용자가 키보드를 누르거나 마우스를 움직이는 경우
 - 적은 오버헤드
- 폴링(polling)
 - 인터럽트의 대안
 - 프로세서가 각 장치의 상태를 반복적으로 확인
 - 컴퓨터 시스템의 복잡도가 증가할수록 오버헤드 증가

참조 : 한빛미디어 운영체제론





Interrupt(인터럽트)



[그림 3-7] 인터럽트 처리





Interrupt(인터럽트)

- 인터럽트 클래스
 - 인터럽트 집합은 시스템 아키텍처에 따라 상이
 - **IA-32** 명세에서 프로세서가 받을 수 있는 신호
 - ✓ 인터럽트(interrupts)
 - » 이벤트 발생
 - » 외부 장치의 상태 변경
 - ✓ 예외(exceptions)
 - » 하드웨어나 소프트웨어 명령어의 실행 결과 오류
 - » 폴트(fault), 트랩(trap), 중단(abort)





Interrupt(인터럽트)

■ 인터럽트 클래스

[표 3-1] IA-32 아키텍처에서 인식하는 공통 인터럽트 유형

인터럽트 유형	각 유형의 인터럽트 상세 내용
입출력	입출력 하드웨어에서 발생하는 인터럽트다. 입출력 인터럽트는 프로세서에 채널이나 장치의 상태가 변한 사실을 알려준다. 예를 들면, 입출력 인터럽트는 입출력 작업이 완료될 때 발생한다.
타이머	시스템은 주기적으로 인터럽트를 발생시키는 장치를 가질 수도 있다. 이러한 인터럽트는 시간을 점검하거나 성능 모니터링 임무 등에 사용할 수 있다. 운영체제는 타이머를 통해 프로세스들의 쿼텀이 다 되었는지 확인할 수 있다.
프로세서 간 인터럽트	이 유형의 인터럽트는 멀티프로세서 시스템에서 한 프로세서가 다른 프로세서에 메시지를 보낼 수 있게 해준다.





Interrupt(인터럽트)

■ 인터럽트 클래스

[표 3-2] 인텔 IA-32의 예외 분류

예외 분류	상세 설명
폴트	폴트는 프로그램의 기계어 명령어가 실행될 때 발생하는 넓은 범위의 문제 때문에 발생한다. 이런 예외의 예로는 0으로 나누려 하거나 처리 중인 데이터의 포맷이 잘못되었거나 유효하지 않은 코드를 실행하거나 실제 메모리의 한계를 넘어서는 메모리 위치 참조, 사용자 프로세스가 특권 명령어를 실행하거나, 보호되는 자원에 접근하려고 하는 일 등을 포함한다.
트랩	트랩은 오버플로우(레지스터에 저장된 값이 레지스터의 용량을 초과하는 것)와 같은 예외 때문에 발생하거나 프로그램 제어권이 코드의 중단점에 도달할 때 발생한다.
중단	중단은 프로세서가 프로세스가 극복해낼 수 없는 오류를 탐지할 때 발생한다. 예를 들어, 예외 처리 루틴 자체가 예외를 발생시킬 때 프로세서가 두 가지 오류를 순차적으로 처리할 수 없을 때 발생한다. 이를 이중 폴트 예외라고 하며, 이 경우, 예외를 발생시킨 프로세스가 종료된다.





프로세스간 통신(Interprocess Communication:IPC)

- 독립적 프로세스 : **Independent** process cannot affect or be affected by the execution of another process.
- 협력적 프로세스 : **Cooperating** process can affect or be affected by the execution of another process
- 프로세스간 협력을 제공하는 이유
 - 정보공유
 - 계산 가속화
 - 모듈성
 - 편의성





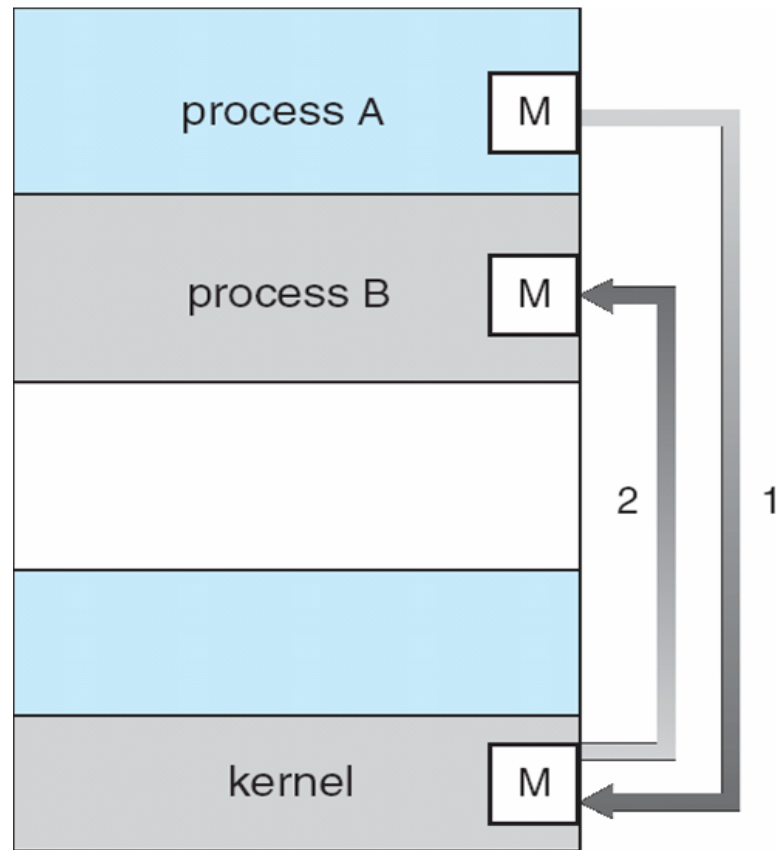
IPC

- IPC를 위한 기본적인 기법
 - 공유 메모리(**shared memory**)
 - ▶ 생산자와 소비자간의 공유하는 메모리를 사용하여 연동
 - 동일한 주소공간을 공유
 - 메시지 전달(**message passing**)
 - ▶ 동일한 주소공간을 공유하지 않고 통신하며 동기화

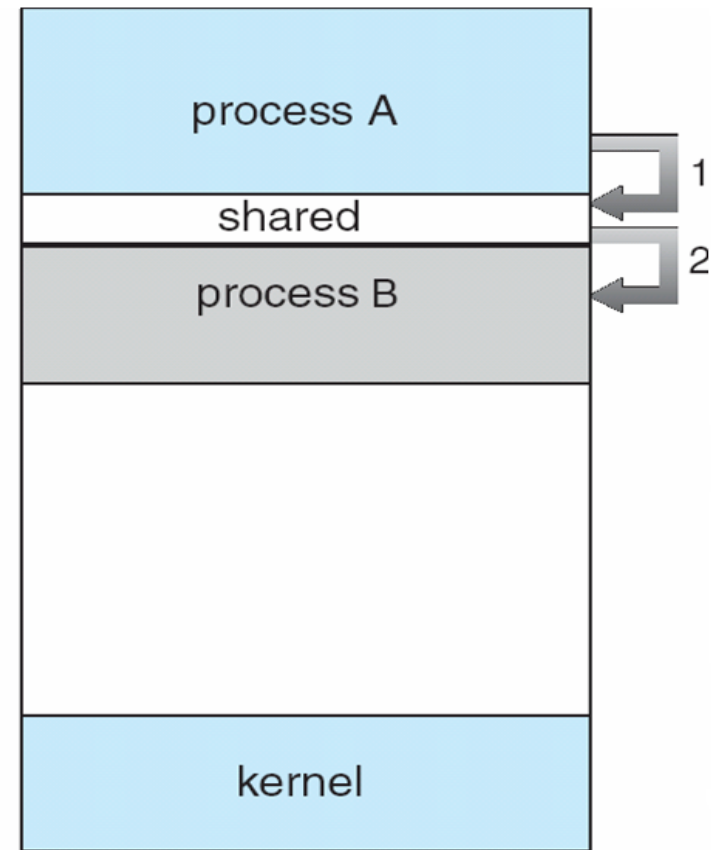




Communications Models



(a)



(b)

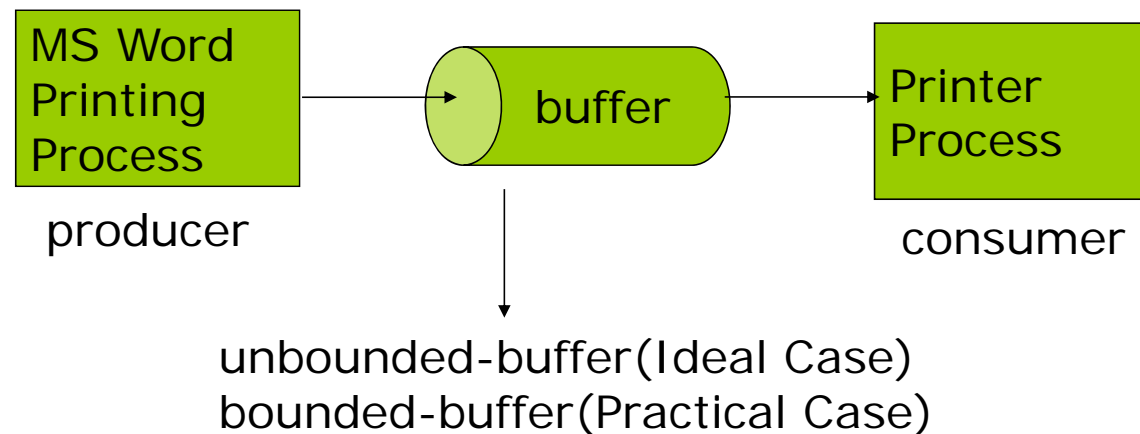




IPC-공유메모리 시스템

■ Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
- 두가지 유형의 **buffer**
 - ▶ *unbounded-buffer* places no practical limit on the size of the buffer.
 - ▶ *bounded-buffer* assumes that there is a fixed buffer size.





Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10  
typedef struct {  
    . . .  
} item;  
  
item buffer[BUFFER_SIZE];  
int in = 0;  
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements





Bounded-Buffer – Producer

```
while (true) {  
    /* Produce an item */  
    while (((in = (in + 1) % BUFFER SIZE count) == out)  
        ; /* do nothing -- no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER SIZE;  
}
```





Bounded Buffer – Consumer

```
while (true) {  
    while (in == out)  
        ; // do nothing -- nothing to consume  
  
    // remove an item from the buffer  
    item = buffer[out];  
    out = (out + 1) % BUFFER SIZE;  
    return item;  
}
```





Message Passing System

- 직접 또는 간접 통신
- 대칭 또는 비 대칭 통신
- 자동 또는 명시적 버퍼링
- 복사에 의한 전송 또는 참조에 의한 전송
- 고정 길이 또는 가변길이 메세지





Interprocess Communication – Message Passing

- IPC facility provides two operations:
 - **send(*message*)** – message size fixed or variable
 - **receive(*message*)**





Direct/Indirect Communication

- **Direct Communication(직접통신)**
 - **send(*P, message*)** – send a message to process P
 - **receive(*Q, message*)** – receive a message from process Q

- **Indirect Communication(간접통신)**
 - 예 : mailbox
 - **send(*A, message*)** – send a message to mailbox A
 - **receive(*A, message*)** – receive a message from mailbox A





Blocking/Non-blocking

- Message passing may be either blocking or non-blocking
- **Blocking** is considered synchronous(동기 통신)
 - Blocking send has the sender block until the message is received
 - Blocking receive has the receiver block until a message is available
- **Non-blocking** is considered asynchronous(비동기 통신)
 - Non-blocking send has the sender send the message and continue
 - Non-blocking receive has the receiver receive a valid message or null





Buffering

- Queue of messages attached to the link; implemented in one of three ways
 1. **Zero** capacity – 0 messages
Sender must wait for receiver (rendezvous)
 2. **Bounded** capacity – finite length of n messages
Sender must wait if link full
 3. **Unbounded** capacity – infinite length
Sender never waits





Pipes(파이프)

■ Pipe

- 운영체제에서 보호하는 메모리 영역으로, 버퍼 역할을 해 프로세스 둘 이상이 데이터를 교환

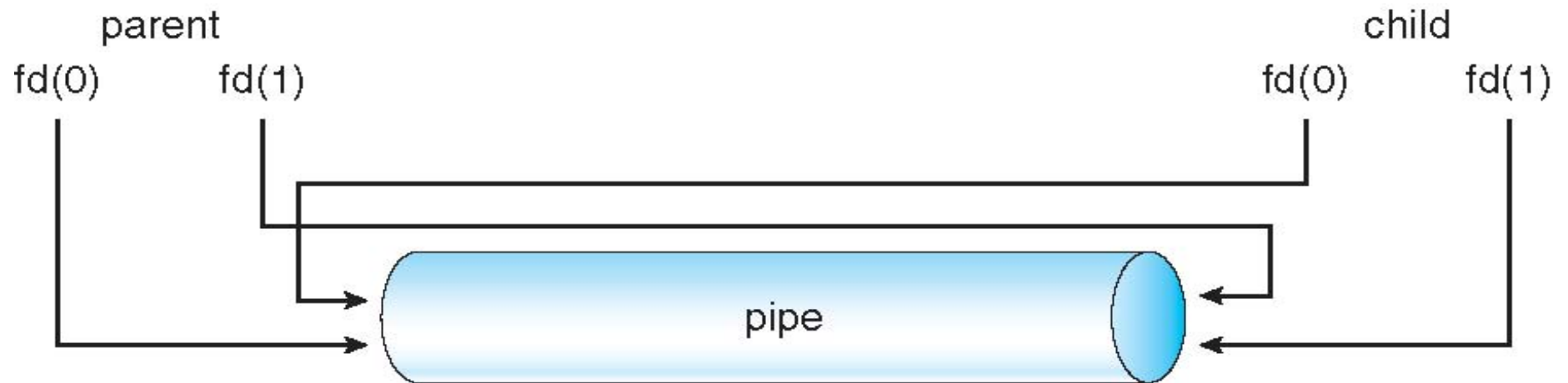
■ Named Pipes

- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems





Ordinary Pipes



End of Chapter 3

