III. 프로세스 (Process)

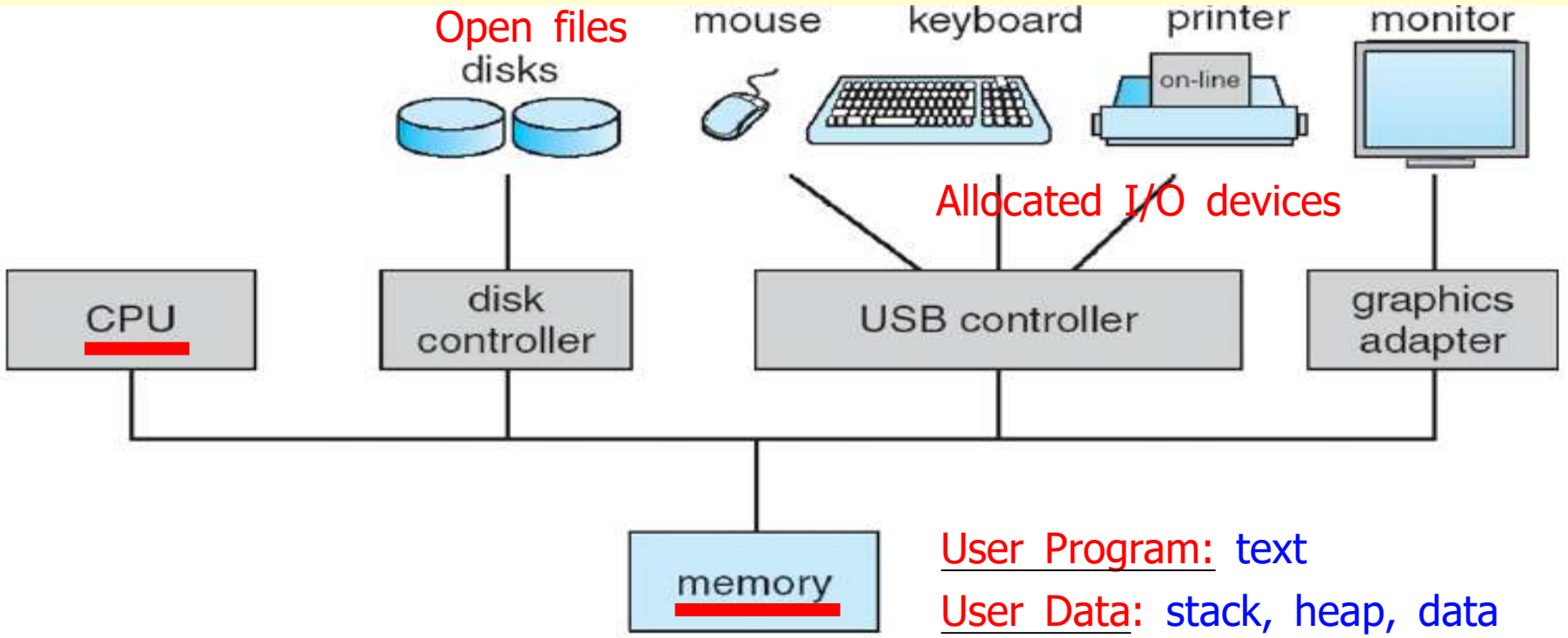
프로세스 관리 (Process Management)

Ⅲ. 프로세스 • 프로세스 • 프로세스 스케줄링 • 프로세스 생성과 종료 • 프로세스 간(間, Inter)의 통신 • 클라이언트-서버間 통신	Ⅳ. **스레드 (Thread)** Ⅴ. **프로세스 스케줄링** Ⅵ. **동기화 (Synchronization)** Ⅶ. **교착상태 (Deadlock)**

1. 프로세스 (Process)

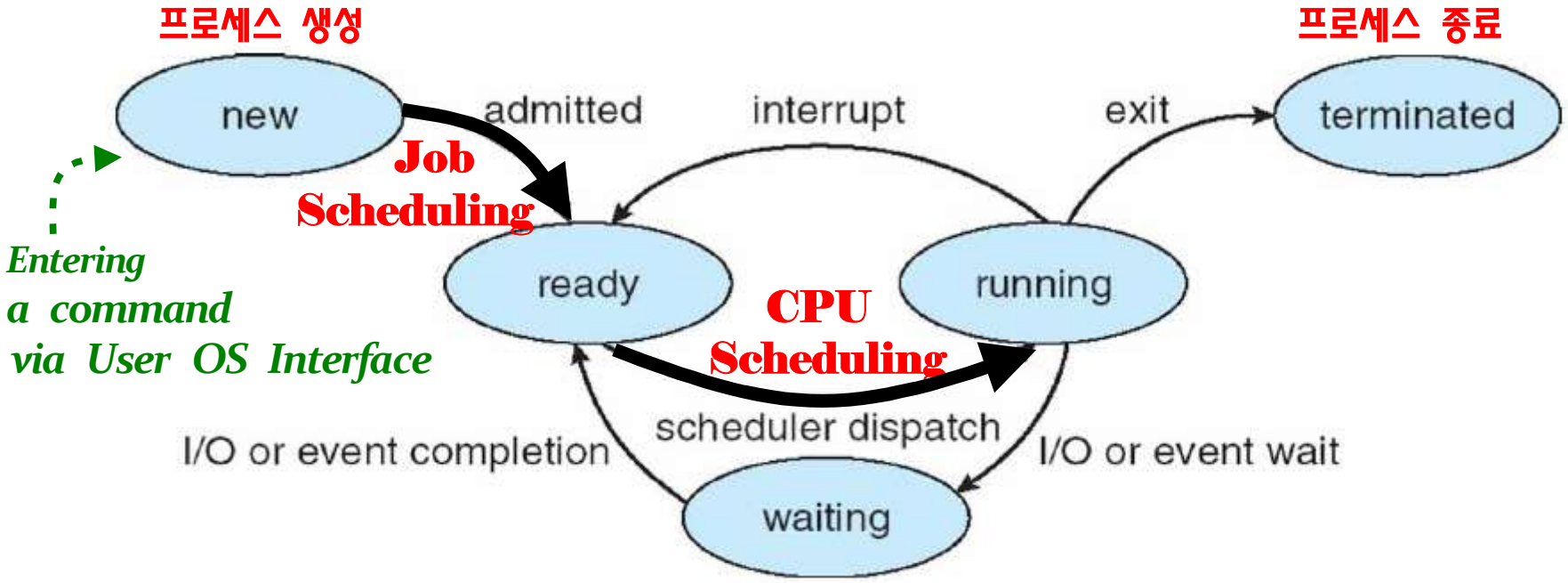
❑ Process

- (정의) **실행중인 프로그램** (*A program in an execution*)
- 운영체제는 프로세스에게 **CPU**를 포함한 **System Resource**를 할당 함.

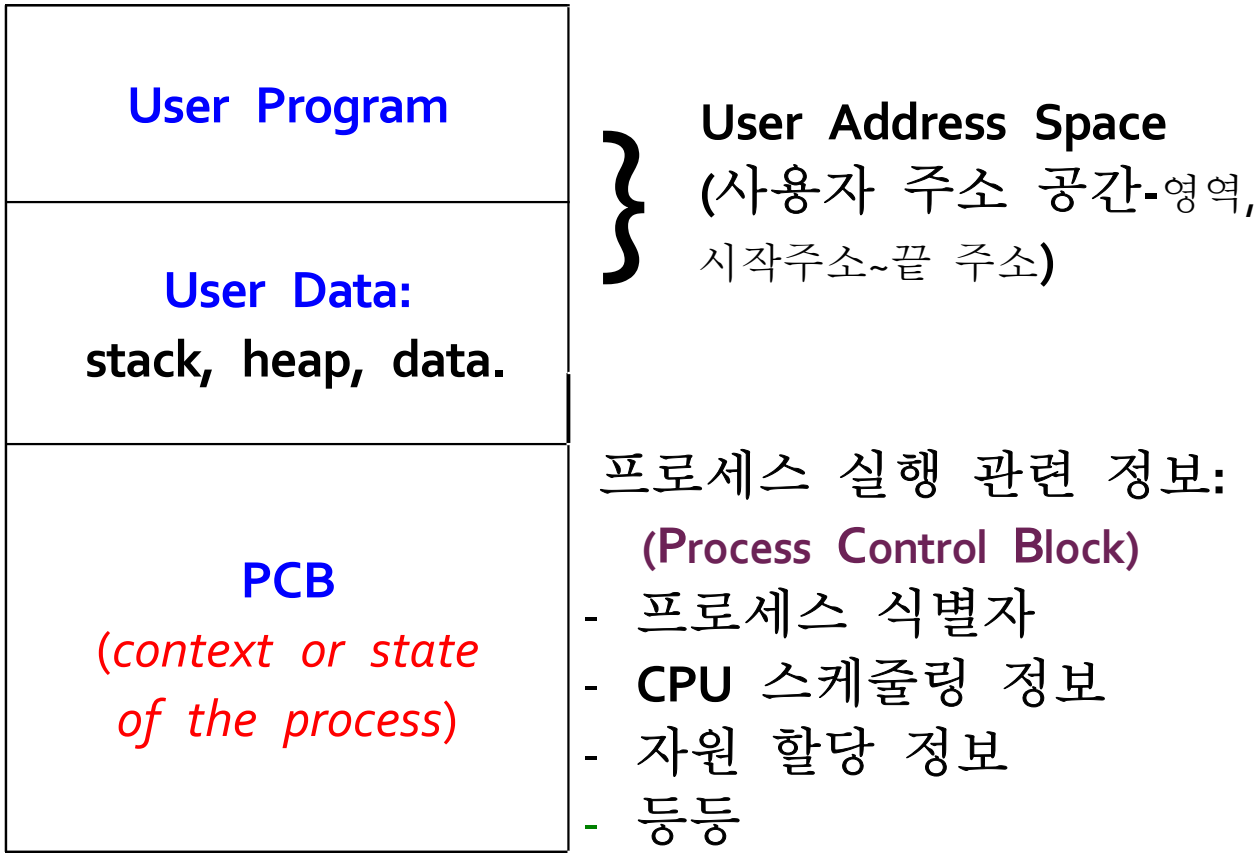


- 시분할 운영체제는 Process Scheduling을 수행함.

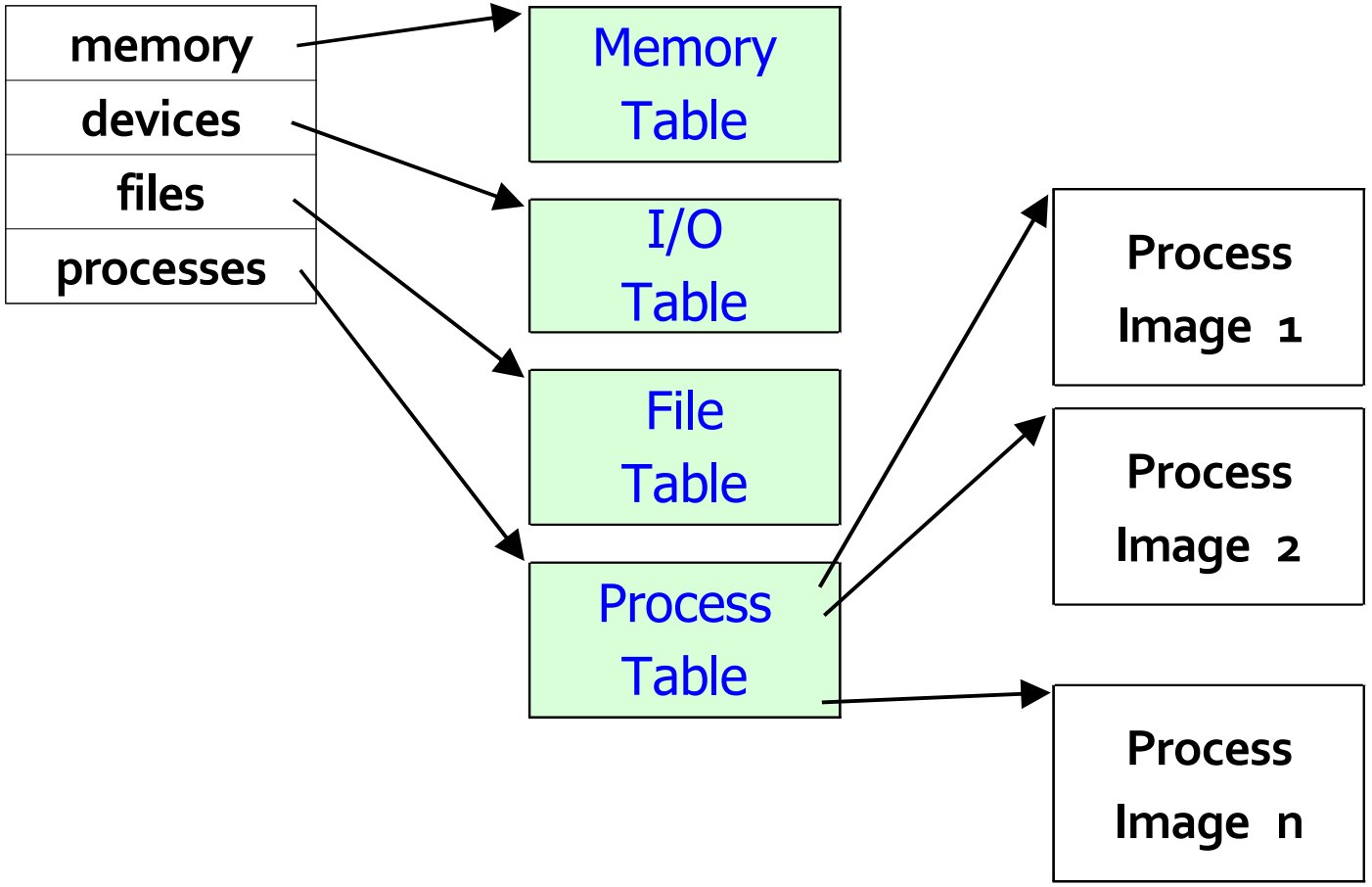
<Scheduling과 프로세스의 일생(States)>



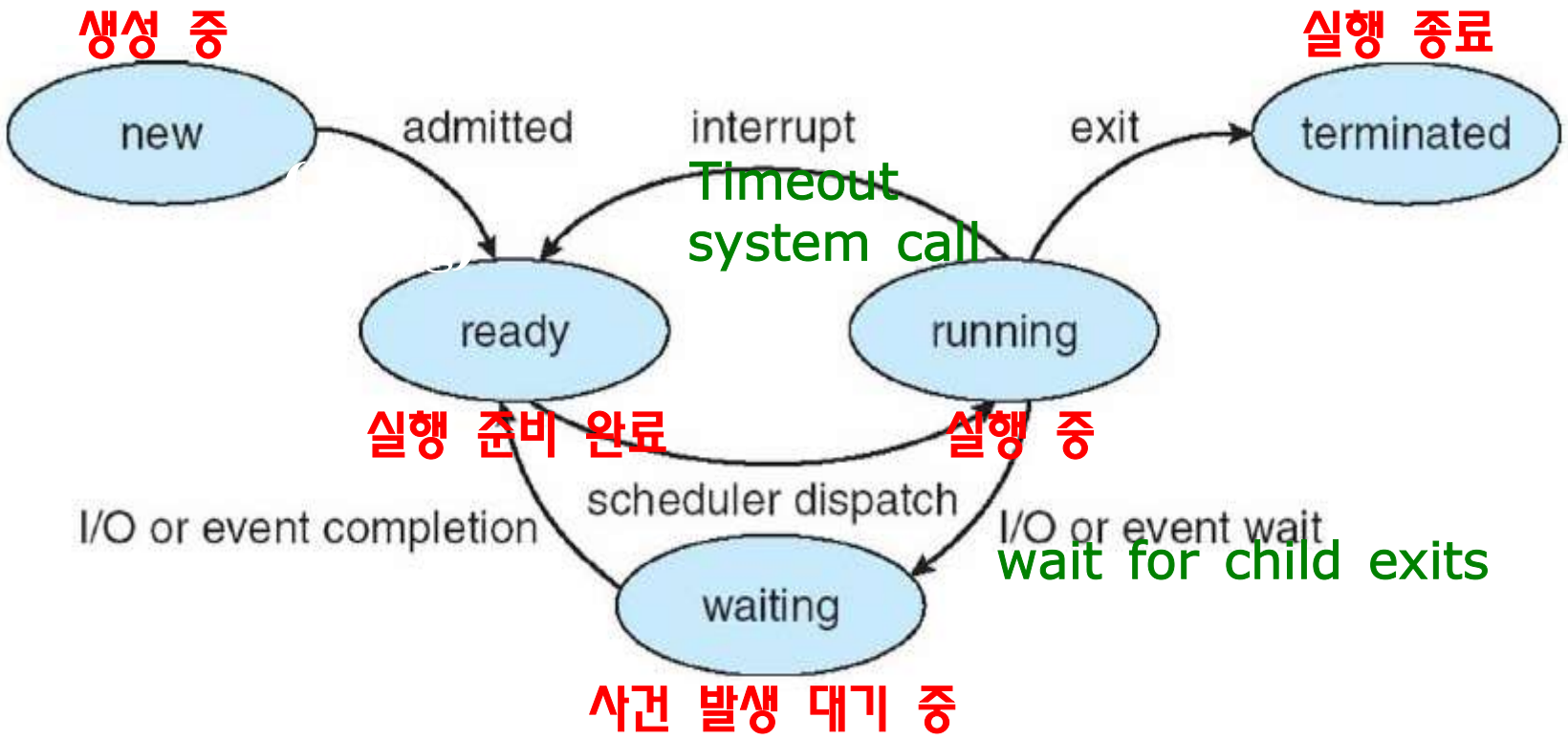
- **Process** image



- 운영체제 제어 테이블 (OS Control Tables)



❑ 프로세스 상태(Process State)와 상태 전이(State Transition)



※ **Ready**: 프로세스 식별자 배정, 메모리 할당, 프로그램 적재, PCB 초기화, 스케줄링 큐에 추가

❑ 프로세스 제어 블록 (Process Control Block, PCB)

- **PCB -** 특정 프로세스 관련 정보 저장 (Per-process):
  프로세스 식별, 프로세스 상태, 스케줄링, 자원 사용, 보안 등에 대한 정보

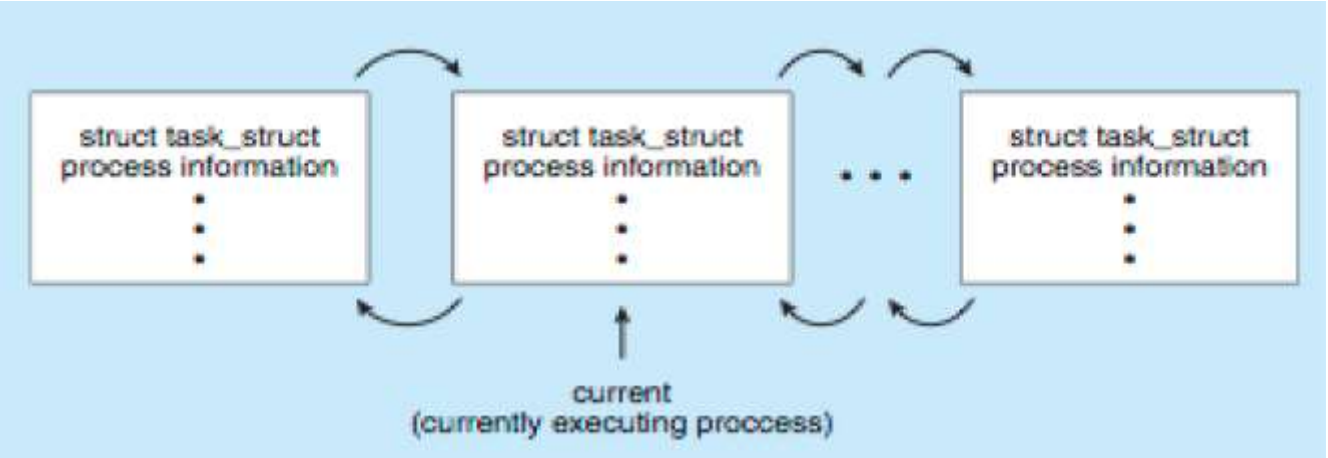
프로세스 식별자	account name, 부모 프로세스 식별자 (ppid)
프로세스 상태	new/ready/running/waiting/terminated 중 하나
process priority	
회계 accounting 정보	CPU 사용 시간, time limit, etc.
PC	} *CPU context* /* **Context Switching**
CPU registers	
메모리 관리 정보	base/limit register, page table, etc.
입출력 상태 정보	할당된 입출력 장치 리스트, open file 리스트
...	

(예) Linux PCB

task_struct:

pid_t pid;	프로세스 식별자
long state;	프로세스 상태
unsigned int time_slice;	스케줄링 정보
struct task_struct *parent;	부모 프로세스
struct list_head children;	자식 프로세스 리스트
struct files_struct *files;	open file 리스트
struct mm_struct *mm;	주소 공간

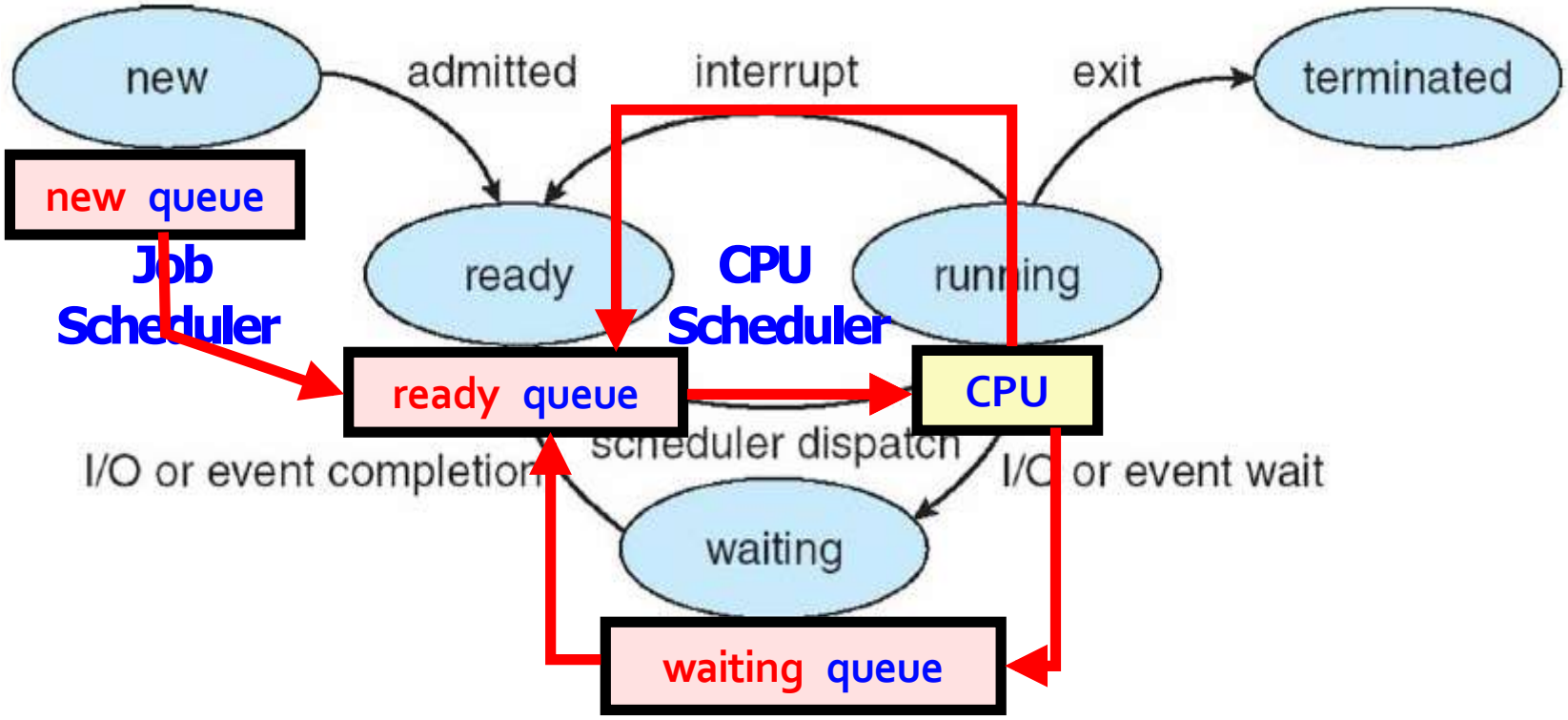
Active Process List



2. 프로세스 스케줄링

❑ Process Scheduling

- CPU 공유 (in **time-sharing**), CPU **utilization** 최대화 (in **multiprogramming**)



❑ Schedulers

- **Long-term** Scheduler (or **Job Scheduler**)
  - new queue의 프로세스(job) 중 Ready Queue에 들어 갈 프로세스 선정.
  - *degree of multiprogramming*을 제어함:
    프로세스 평균 생성율 ≈ 프로세스 평균 이탈율 ⇨ 안정성⬆
  - CPU-bound와 I/O-bound process를 적절히 혼합[1]하여 선정 ⇨ 성능⬆
  - 운영체제에 따라 job scheduler를 두지 **않는** 경우도 있음
    (예) UNIX, Microsoft Windows
    ⇨ 시스템 안정성(*Stability*)을 위해 terminal 수 제한(Number of Users, Number of Processes-per system, user)과 같은 물리적 제약을 가하거나 사용자들의 자기 조정 특성에 의존함

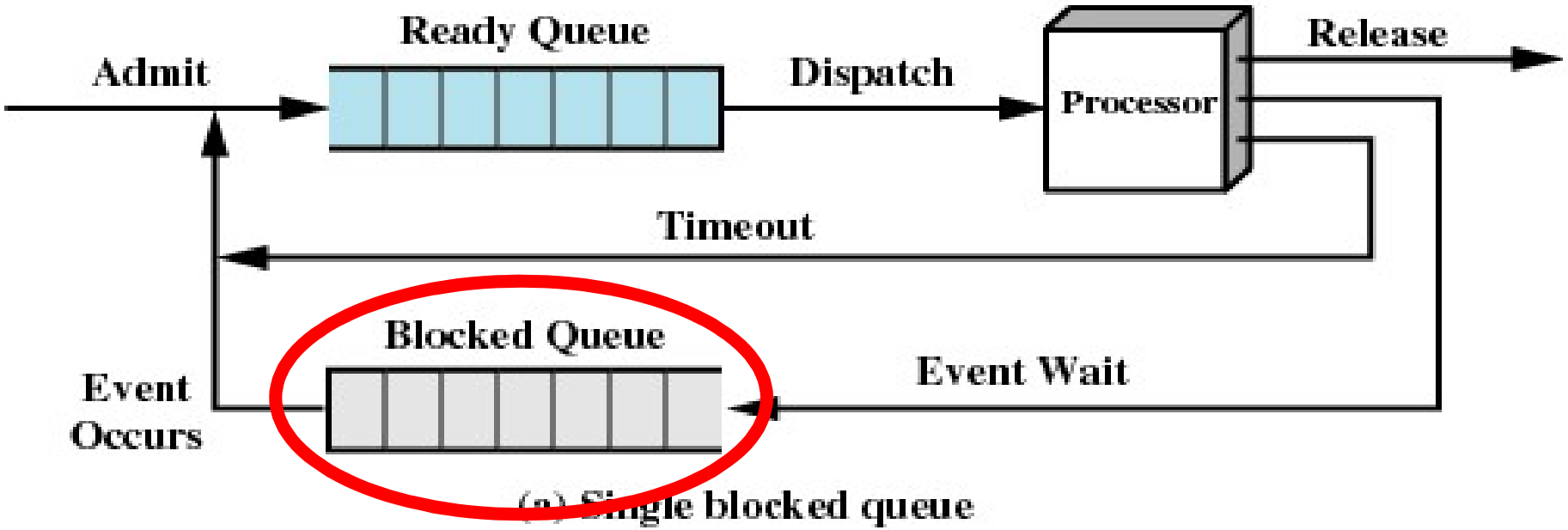
- **Short-term** Scheduler (or **CPU scheduler**)
  - ready queue의 프로세스 중 CPU를 할당 받을 프로세스 선정

1) process mix

❑ Waiting Queue

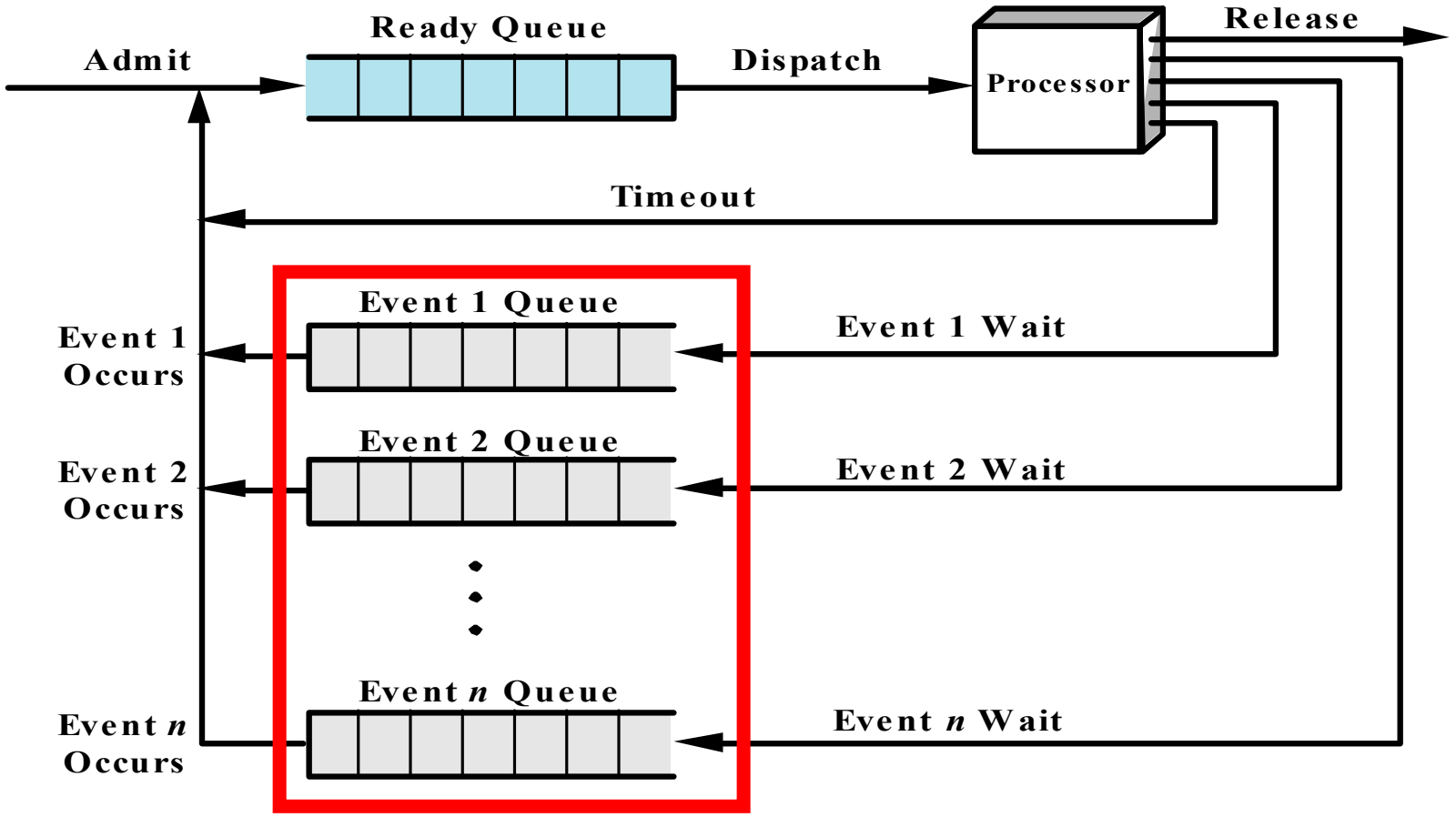
- 하나 또는 여러 개.

- **event** 유형에 관계없이 하나의 대기 큐(Single waiting queue)를 사용하는 경우



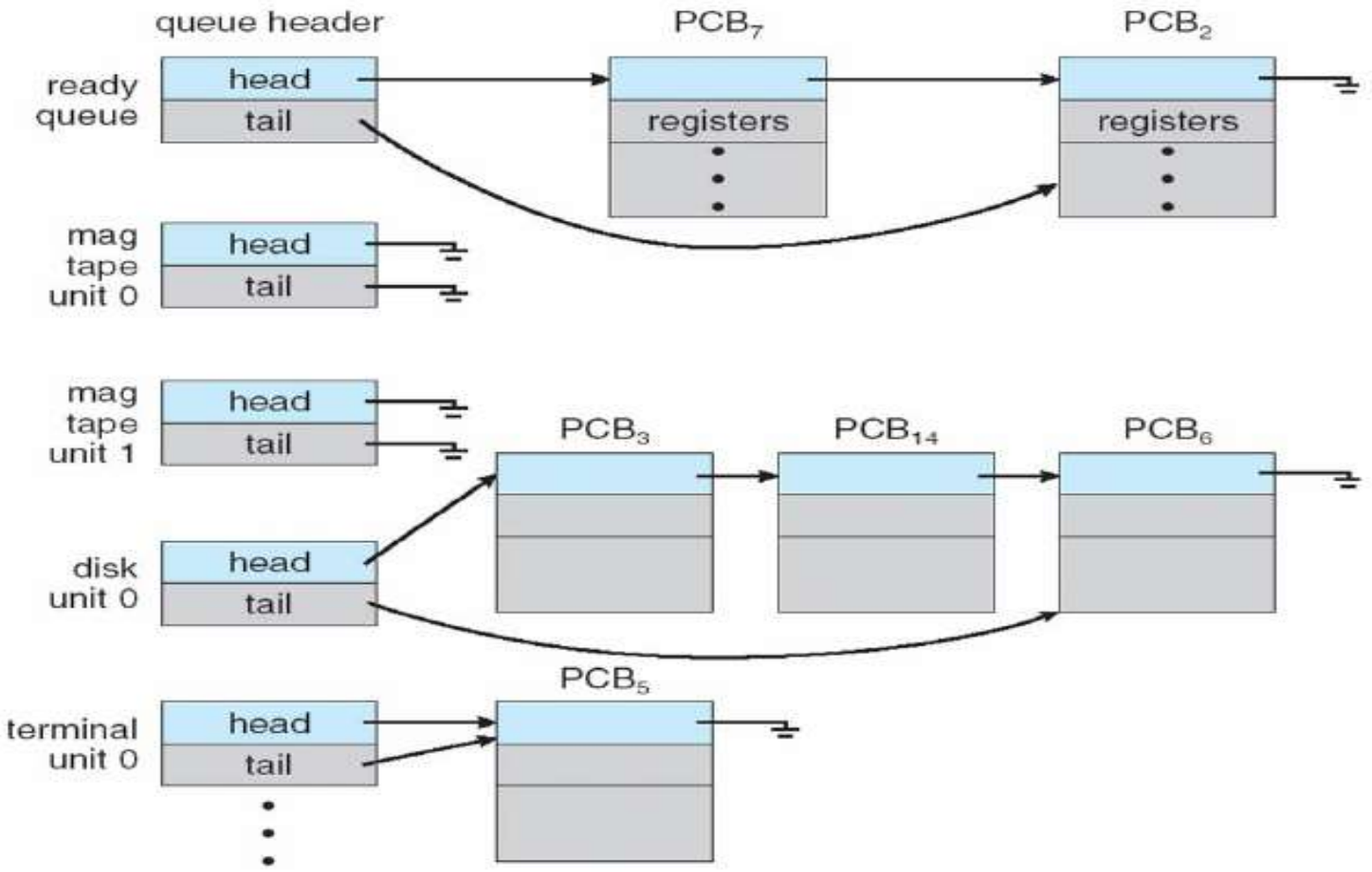
(a) Single blocked queue

- **event** 유형別로 여러 개의 독립적인 대기 큐를 사용하는 경우



(b) Multiple blocked queues

- **Ready queue와 여러 <u>I/O Device Queues</u>**



❑ Medium-term Scheduler

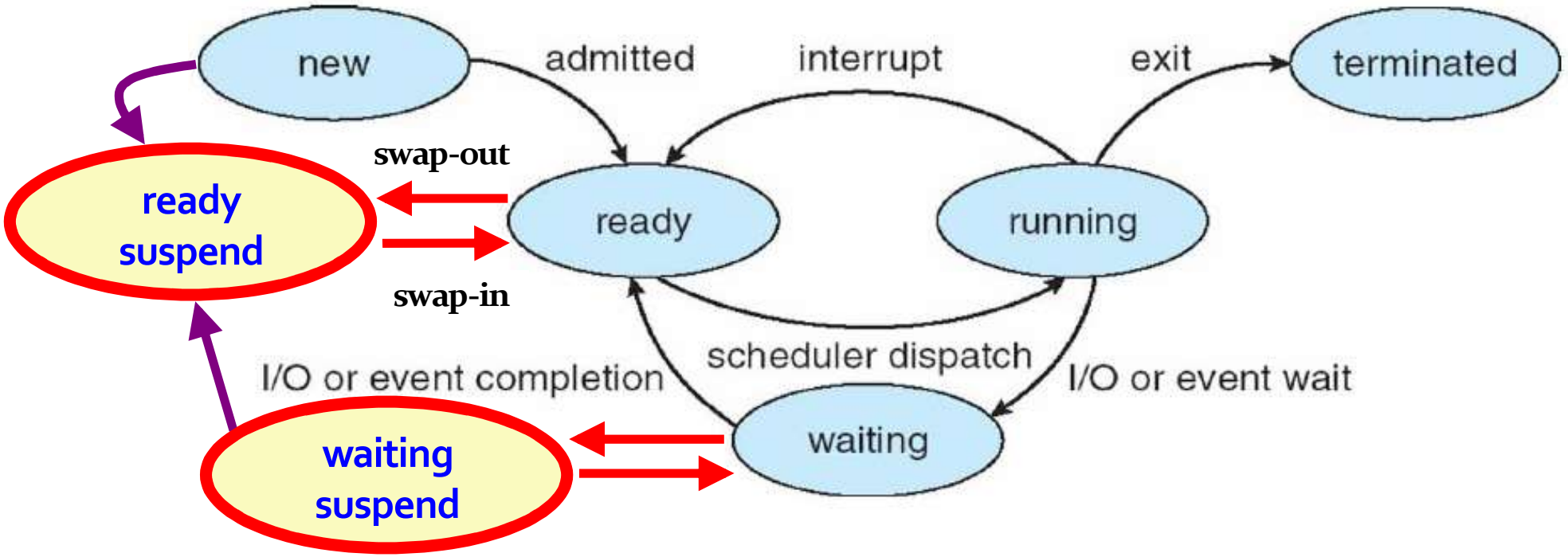
- *Swapping* 관련 스케줄링 작업 수행
  - victim process 선정; swap-out; swap-in.

    Swapping is a mechanism in which a process can be swapped/moved temporarily out of main Memory to a backing Store(Disk), and then brought back into memory for continued execution.

  - swap-out이 필요한 경우
    - process mix 개선, multiprogramming degree 완화
    - 가용한 메모리 부족
    - 주기적으로 실행되는 프로세스, 부모 프로세스의 요청, etc.

- 7-State process model
  - swapping이 허용되는 경우 추가의 2가지 상태가 요구됨:
    - ready 상태에서 swap-out ⇨ Ready suspend 상태.
    - waiting 상태에서 swap-out ⇨ Waiting suspend 상태.

- 추가의 상태 전이

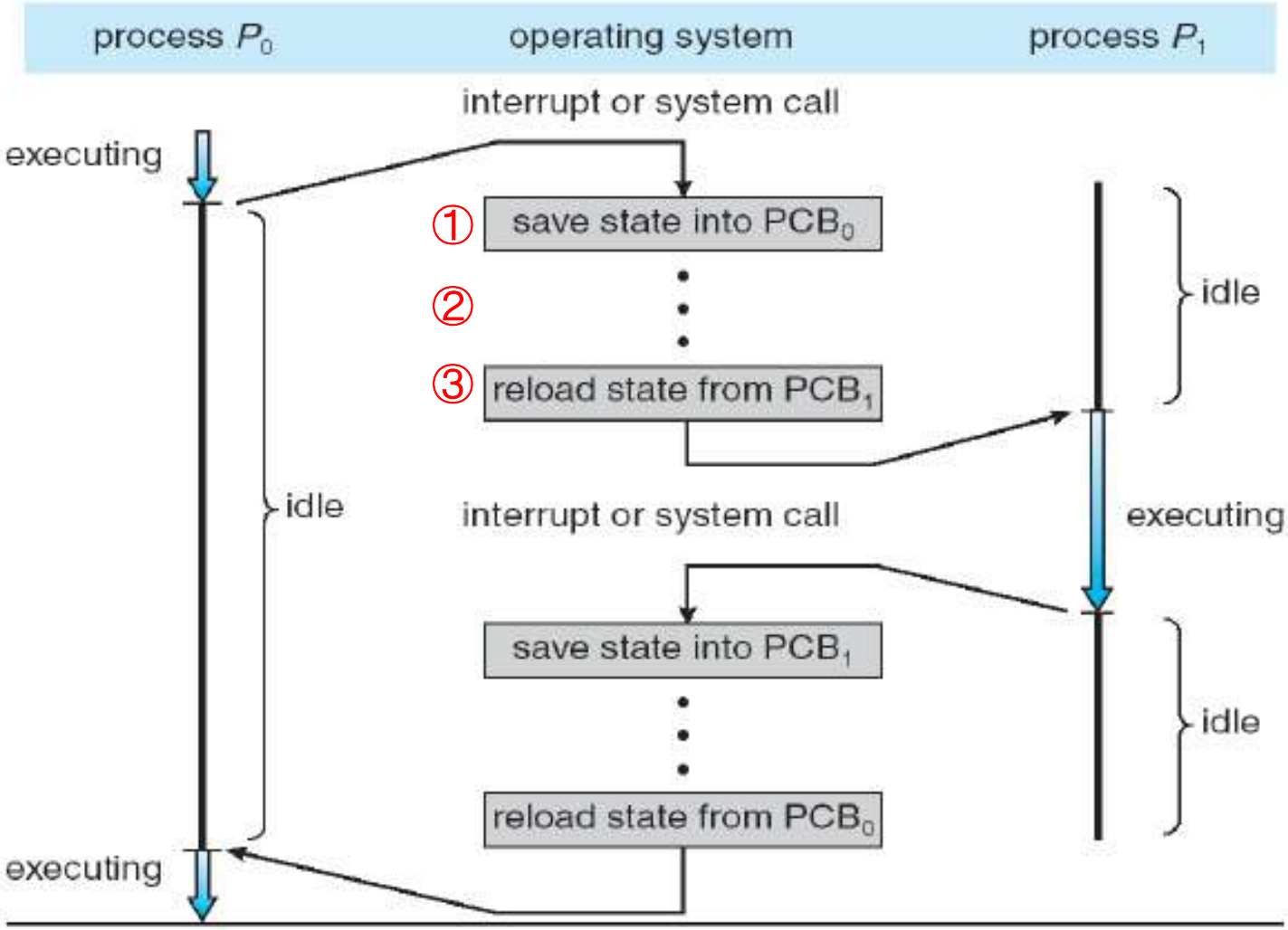


Swap-Out	Swap-In
waiting → waiting suspend: 보통 선호됨.	ready suspend → ready: • ready가 없는 경우. • ready 보다 우선순위↑.
ready → ready suspend: • ready가 아주 큰 경우. • 곧 깨어날 waiting이 ready 보다 우선순위↑.	waiting suspend → waiting: 곧 깨어날 waiting suspend가 ready suspend 보다 우선순위↑.

Swap-Out
running → ready suspend: 방금 깨어 난 waiting suspend에 의해 preemption(선점, 강점) 되는 경우.

❑ Context Switching

CPU dispatcher: Context switching + User mode로 전환 + Next Instruction으로 Jump.



Context Switching:

① **current process**의 context를 자신의 PCB에 보존함(*Save*).

(② CPU scheduling.)

③ **selected process**의 context를 환원함 (*Restore*).

(note) **Dispatch Latency**

3. 프로세스 생성과 종료

❑ Process Creation(생성)

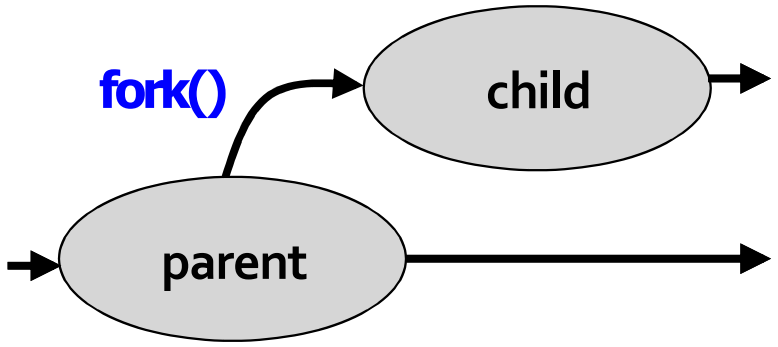
- 프로세스(Parent)는 새로운 프로세스(Child)를 생성할 수 있음.
  - OS는 프로세스 생성 서비스(*system call*)를 제공함. (예) UNIX fork().
  - *process tree* (프로세스 트리)

- 부모, 자식 프로세서 사이(間) 관계
  - 자원 공유: 부모 자원 전부 공유 vs. 일부 공유 vs. 공유 않음.
  - 실행: 동시 실행 vs. child 종료까지 대기(wait).
  - 주소 공간: 부모와 동일한 프로그램 실행 vs. 별도 프로그램 실행.

❑ Unix에서의 프로세스 생성: fork(), exec(), wait() 시스템 호출(system call)

- **fork()** : 자신을 복제하여 자식 프로세스를 생성함.

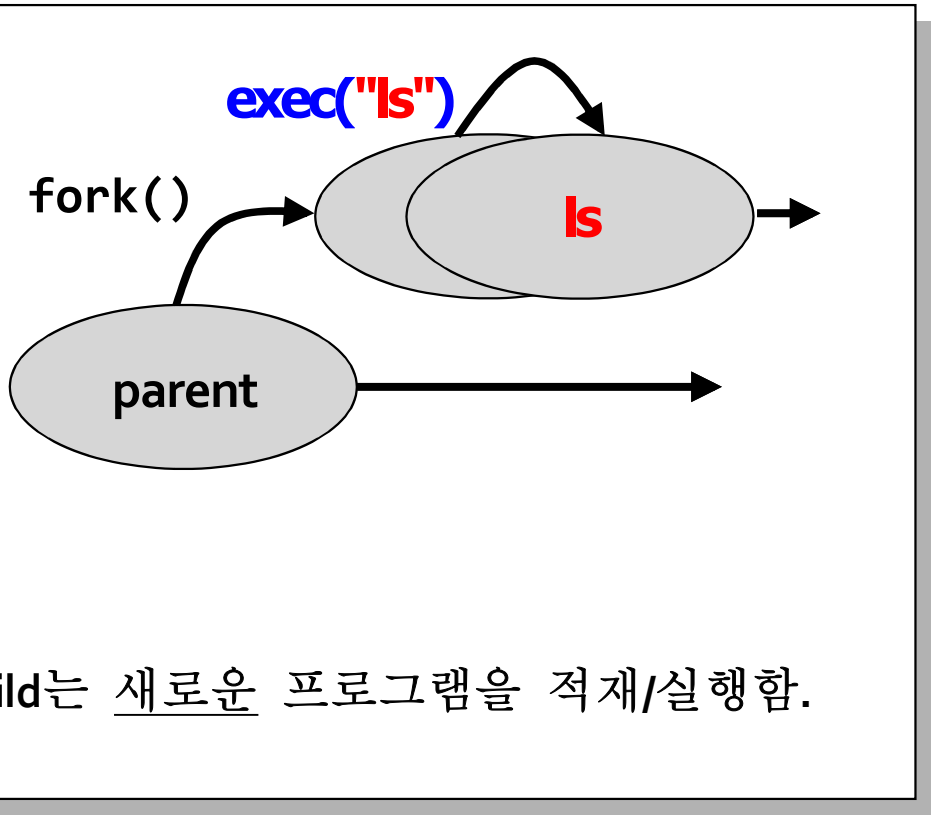
  fork 수행시 return(반환) 값: 부모 프로세스에게는 자식 프로세스의 pid(>0)를 반환(주고)하고, 자식 프로세스에게는 0(zero)을 반환(준다)함.



- child는 parent의 복제본:
  - *동일 프로그램* 실행.
  - *자원 공유*.
  - 동시 실행.

```
int main() {
    pid_t pid;
    ...
    pid = fork();   // return pid
    if (pid == 0) {
        어떤 프로세스가 실행?
    } else {
        어떤 프로세스가 실행?
    }
    어떤 프로세스가 실행?
}
```

- **exec(*new_program*)** : 새로운 프로그램을 적재하고 그것을 실행함.



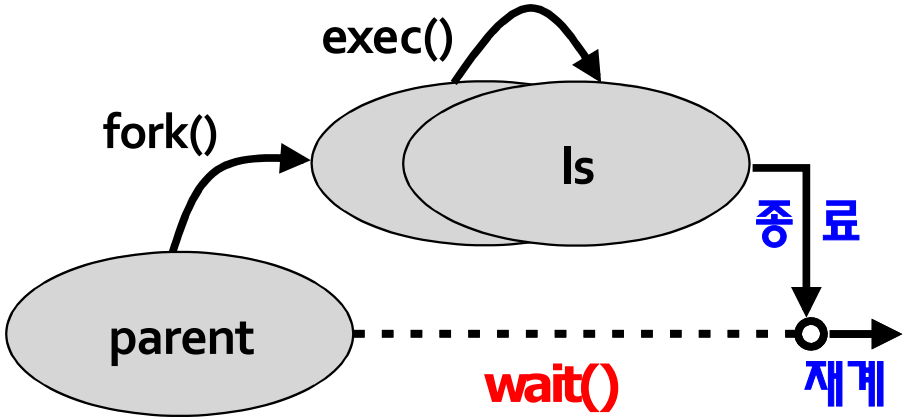
- child는 새로운 프로그램을 적재/실행함.

```
int main() {
    pid_t;
    ...
    pid = fork();
    if (pid == 0) {
        execlp("/bin/ls", ls, NULL);
    } else {
        부보 프로세스가 실행.
    }
    부모 프로세스가 실행.
}
```

exec 관련 system call – execl, execle, execlp, execv, execve, execvp

e: It is an array of pointers that points to environment variables and is passed explicitly to the newly loaded process.
l: l is for the command line arguments passed a list to the function
p: p is the path environment variable to find file passed as an argument to be loaded process.
v: v is for the command line arguments. These are passed as an array of pointers to the function.

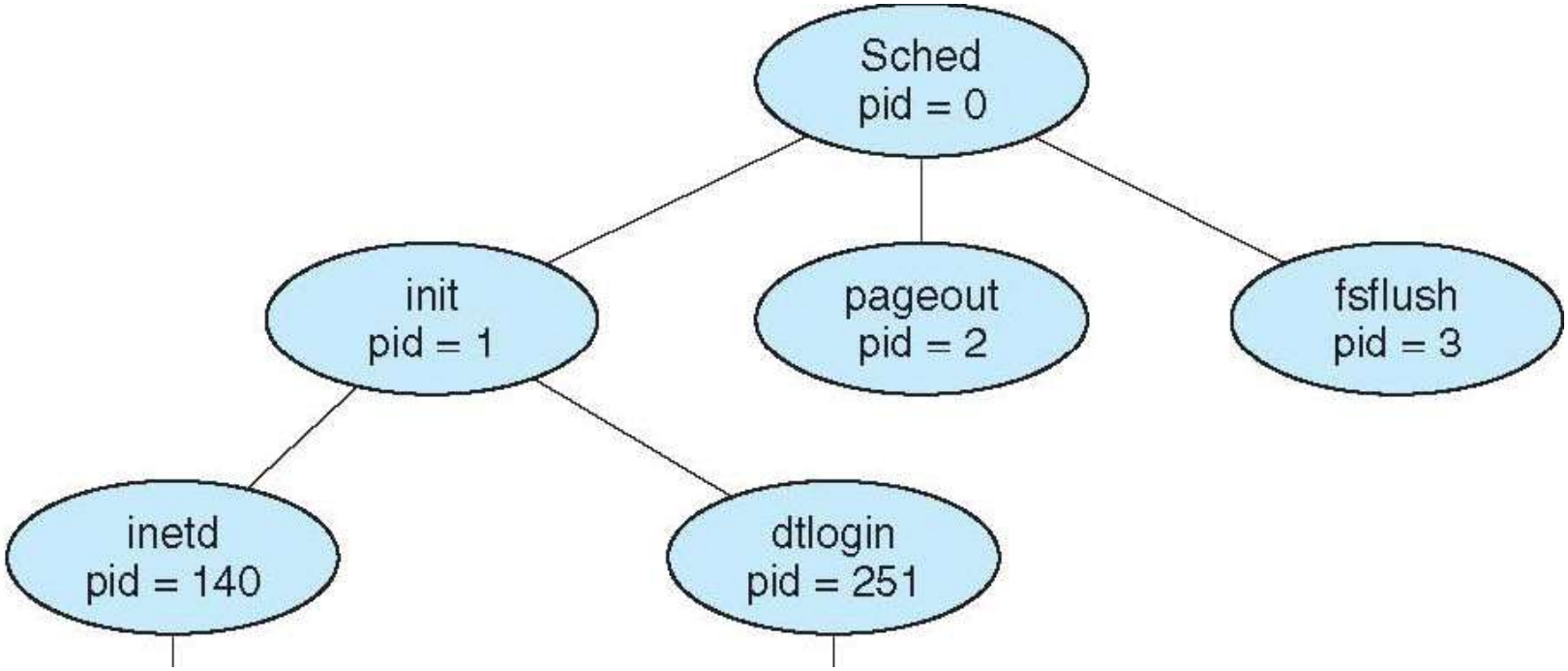
- **wait()** : parent는 child가 종료될 때까지 대기함.



- parent는 child가 종료될 때까지 기다림.
  child가 종료되면 parent는 실행을 재계함.

```
int main() {
    pid_t;
    ...
    pid = fork();
    if (pid == 0) {
        execlp("/bin/ls", ls, NULL);
    } else {
        wait(NULL);
    }
    부모 프로세스가 실행.
}
```

❑ Process Tree on Solaris



sched*d* 최상위 프로세스 (***d*** : **Daemon Process**)

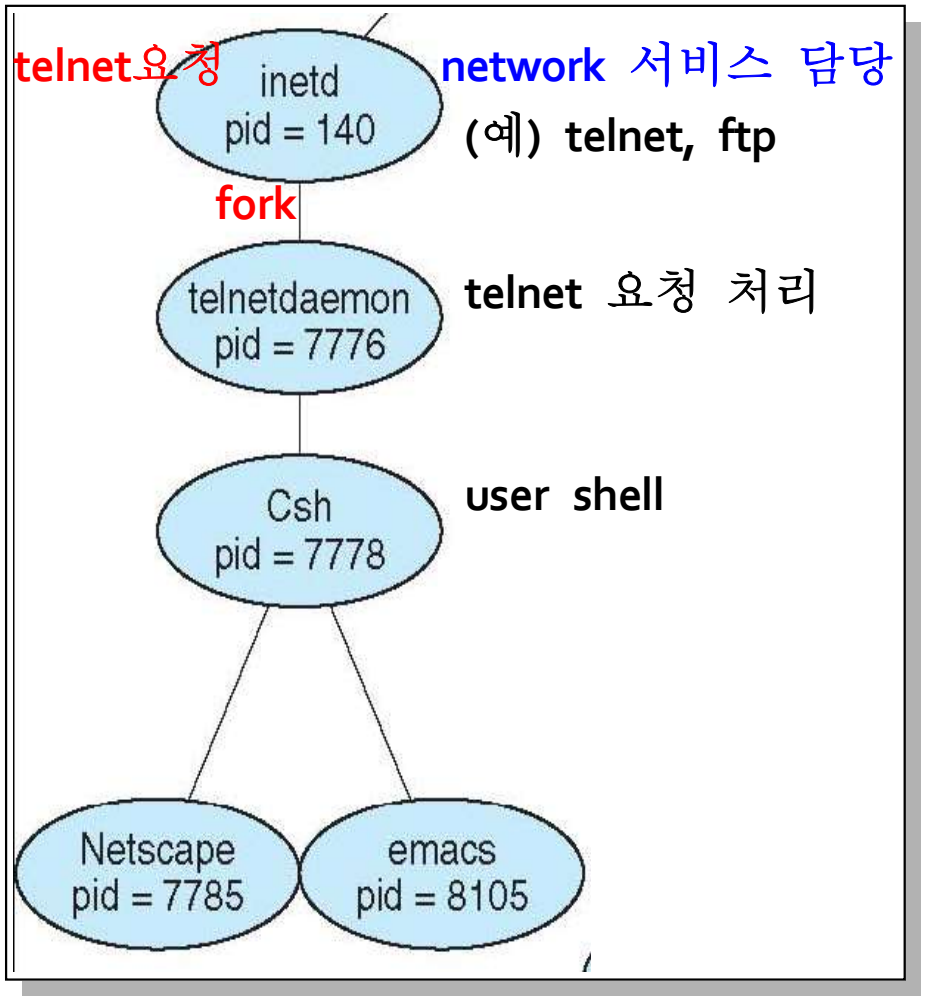
init 모든 **user process**의 **root**

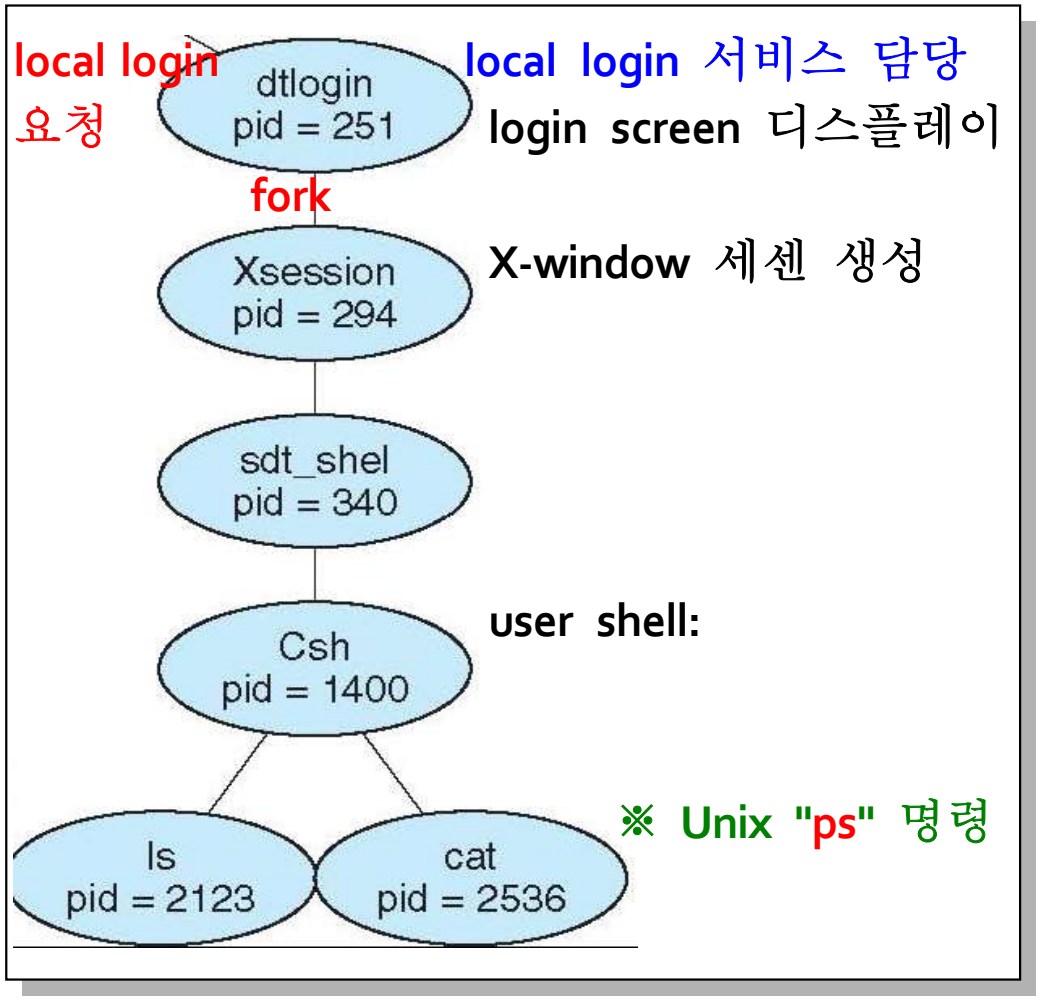
inet*d* 모든 네트워크 서비스를 처리함.

dtlogin 모든 **local login**을 처리함.

pageout 메모리 관리

fsflush 파일시스템 관리


telnet요청
inetd
pid = 140
network 서비스 담당
(예) telnet, ftp
fork
telnetdaemon
pid = 7776
telnet 요청 처리
Csh
pid = 7778
user shell
Netscape
pid = 7785
emacs
pid = 8105


local login
요청
dtlogin
pid = 251
local login 서비스 담당
login screen 디스플레이
fork
Xsession
pid = 294
X-window 세션 생성
sdt_shel
pid = 340
Csh
pid = 1400
user shell:
ls
pid = 2123
cat
pid = 2536
※ Unix "ps" 명령

❑ Process Termination (프로세스 종료)

- 프로세스 자신이 **exit()** system call을 호출하는 경우
  - OS는 프로세스가 할당 받은 모든 자원을 회수함 (resource return).
  - wait() 중인 parent에게 자신의 pid, 종료 상태(exit status)를 반환함.

```
pid_t pid; int status;
…
pid = wait(&status);
```

(note) *Zombie Process* 종료 시점부터 parent가 wait()를 호출할 때까지.

Zombie Process - 자식 프로세스가 부모 프로세스 보다 먼저 종료되고, 부모 프로세스가 자식 프로세스의 종료 상태를 회수하지 않았을 경우에, 자식 프로세스를 좀비 프로세스라고 함.

- parent가 child의 실행을 강제 종료시키는 경우 **(Abort)**
  - child가 할당된 자원을 초과하여 사용할 경우.
  - child에게 부여한 작업이 더 이상 필요 없는 경우.

- descendents를 가지는 parent가 종료되는 경우: (*Orphan Process* 처리 방법)

Orphan Process - 부모 프로세스가 자식 프로세스보다 먼저 종료되면, 자식 프로세스는 고아 프로세스가 됨.

(예) VMS 모든 자손 연쇄 종료 (*Cascade termination*)

(예) UNIX init 프로세스를 새로운 parent로 설정함.

4. 프로세스 間 통신

1. Cooperating Process와 Inter-Process Communication (IPC)

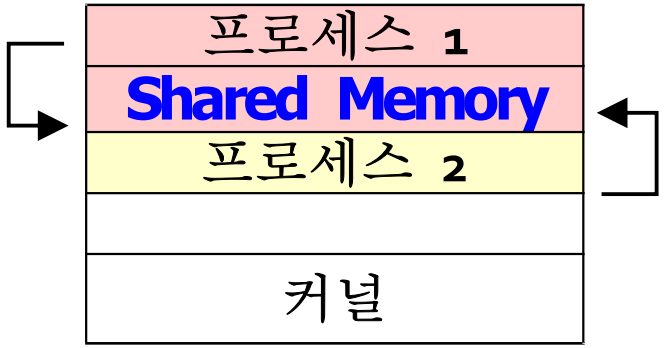
- Basically, processes are independent with each other.
  독립적 = 서로 영향을 주지도 받지도 않음.

- A process is *Cooperating*
  if it can affect or be affected by other processes executing in the system.
  (note) 정보 공유, 성능 제고, 모듈화, ...

- Cooperating processes require an *IPC Mechanism* that
  will allow them *to exchange data and information*.

- There two fundamental models of IPC:
  - 공유 메모리 모델 (Shared Memory Model)
  - 메시지 전송 모델 (Message Passing Model)

❑ IPC의 Shared Memory Model



① 한 프로세스가 자신의 주소공간에 Shared Memory 설정함. (시스템 호출 이용)
② 다른 프로세스는 공유메모리를 자신의 주소공간에 Attach.(시스템호출)
③ 읽고 씀 (note) 동기화 Synchronization

(예-POSIX) 생산자 프로세스 (note) 상세 코드 및 소비자 코드는 교재 참고.

```
// 공유메모리 생성 및 크기(bytes) 설정; shm_fd: 공유메모리 file descriptor.
int shm_fd = shm_open(공유메모리이름, O_CREAT|O_RDWR, 접근허가);
ftruncate(shm_fd, 4096);    //파일을 지정한 크기로 변경

// Treats shared memory as (memory-mapped) FILE; 공유 가능 설정.
void *p = mmap(..., MAP_SHARED, shm_fd, ...);

// 파일 쓰기 함수를 사용하여 공유메모리에 씀.
sprintf(p, "%s", "hello");        p += strlen("hello");
sprintf(p, "%s", "world");        p += strlen("world");
...
```

```
/* Producer process illustrating POSIX shared-memory API */
#include <stdio.h>   #include <stlib.h>   #include <string.h>
#include <fcntl.h>   #include <sys/shm.h>   #include <sys/stat.h>

int main() {
    const int SIZE 4096;      /* the size of shared memory */
    const char *name = "OS";   /* name of the shared memory */
    int shm fd;                /* shared memory file descriptor */
    void *ptr;                 /* pointer to shared memory */

    const char *message_0 = "Hello";  /* strings written to shared_mem*/
    const char *message_1 = "World!";

        /* create the shared memory */
    shm fd = shm open(name, O CREAT | O RDRW, 0666);
        /* configure the size of the shared memory */
    ftruncate(shm fd, SIZE);
        /* memory map the shared memory */
    ptr = mmap(0, SIZE, PROT WRITE, MAP SHARED, shm fd, 0);
    sprintf(ptr,"%s",message_0);  /* write to the shared memory */
    ptr += strlen(message_0);
```

```
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

메모리 매핑 관련 함수로

기능	함수원형
메모리 매핑	void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
메모리 매핑 해제	int munmap(void *addr, size_t len);
파일 크기 조정	int truncate(const char *path, off_t length); int ftruncate(int fildes, off_t length);
매핑된 메모리 동기화	int msync(void *addr, size_t len, int flags);

메모리 매핑

Memory Mapping은 파일을 프로세스의 메모리에 매핑하는 것. 즉, 프로세스에서 전달한 데이터를 저장할 파일에 직접 프로세스의 가상 주소 공간으로 매핑함. 따라서 파일에서 사용하는 read나 write 함수를 사용하지 않고도 프로그램 내부에서 정의한 변수를 사용해 파일에서 데이터를 읽거(read)나 쓸(write) 수 있다.

```
/* Consumer process illustrating POSIX shared-memory API */
#include <stdio.h>   #include <stlib.h>   #include <fcntl.h>
#include <sys/shm.h>    #include <sys/stat.h>

int main() {
    const int SIZE 4096;          /* the size of shared memory */
    const char *name = "OS";      /* name of the shared memory */
    int shm fd;                   /* shared memory file descriptor */
    void *ptr;                    /* pointer to shared memory */

        /* open the shared memory */
    shm fd = shm open(name, O RDONLY, 0666);
        /* memory map the shared memory */
    ptr = mmap(0, SIZE, PROT READ, MAP SHARED, shm fd, 0);

    printf("%s",(char *)ptr);   /* read from the shared memory */

    shm unlink(name);           /* remove the shared memory */
    return 0;
}
```

❑ IPC의 Message Passing Model

- 공유메모리를 사용하지 않음. 분산 환경에 적합.
- Two basic operations: send(메시지), receive(메시지)
- 통신을 위해 양자間 *communication link*가 필요함.

- 송신자, 수신자의 명명(Naming, 지정)

직접 통신	간접 통신
send(수신자, mesg) receive(송신자, mesg)	send(메일박스, mesg) receive(메일박스, mesg)
- sender, receiver 명시. - 일대일 통신. (전용 link)	- mailbox(or port)를 공유하는 프로세스間 통신. - 다대다 통신
Asymmetry 방식: send(수신자, mesg) receive(mesg)	system이 수신할 receiver 선택 방법: - 최대 두 프로세스 間 link 설정. - 한순간 오직 하나의 프로세스만 receiver() 실행 허용. - 임의의 receiver 선정. (note) 선정 알고리즘에 따름.

- 송수신 동기화 (synchronization)

송신	동기적	Sender is blocked until mesg is received by receiver or mailbox.
	비동기적	Sender sends mesg and resumes operation.

수신	동기적	Receiver blocks until a mesg is available.
	비동기적	Receiver retrieves either a valid mesg or a null.

  - 임의의 조합으로 통신 가능함.
  - 랑데뷰(랑데뷰, 만날 약속, 회의) ***Rendezvous*** = blocking send+blocking receiver.

- 버퍼링 (Buffering) - 통신(직접, 간접) 時 사용되는 임시 큐의 유형:

	큐의 길이	송신 프로세스의 blocking
Zero capacity	zero.	수신자가 수신할 때까지 (랑데뷰)
Bounded capacity	유한 길이.	큐가 full일 때
Unbounded capacity	무한 길이.	Never blocks.

2. Client-Server System에서의 통신

공유 메모리, 메시지 전송 외에도 socket, Remote Procedure Call (RPC), Pipe 등의 방식이 있음.

❑ Socket

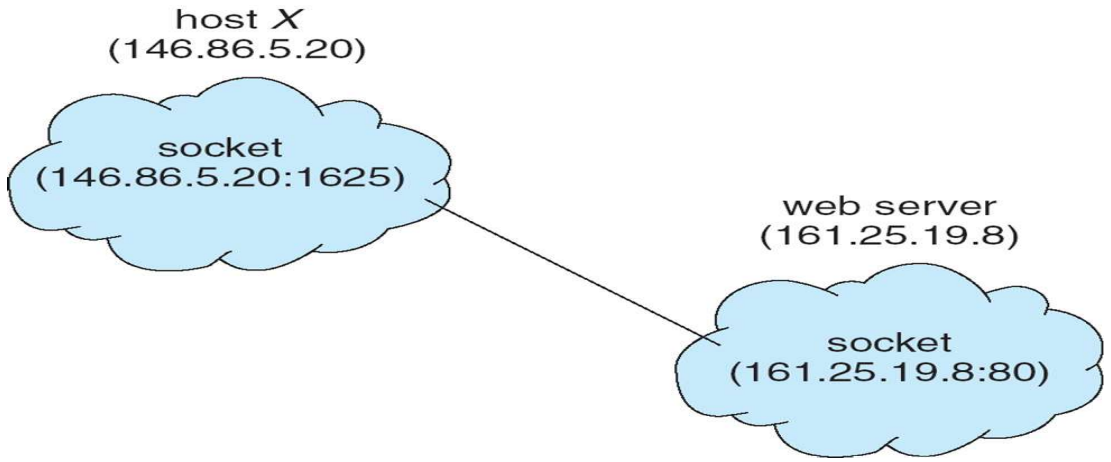
- 통신 종단점 (Communication Endpoint): [protocol://]IP주소:포트번호

  Port) 특정 process 또는 network service type을 식별하는 논리적 장치.

  telnet server listens to port 23; *ssh* server 22;

  FTP server 21; *http* server 80.

- 한 쌍의 socket이 communication link를 형성함.



(예) Java Socket

Date 서버 (포트 2000)

```
// port 2000 요청을 listening 할 서버 소켓.
ServerSocket ss = new ServerSocket(2000);

while (true) {
        // client의 연결 요청을 기다림.
        // 요청時, client와 통신할 소켓 생성.
        Socket socket = ss.accept();

        // 소켓의 OutputStream에 연결될
        // 출력스트림 생성; 현재 시각 write.
        PrinterWriter out = new PrinterWriter(
                socket.getOutputStream(), true);

        out.println(new Date().toString());

        sock.close();
}
```

Client - Date 서버로부터 현재 시각 획득

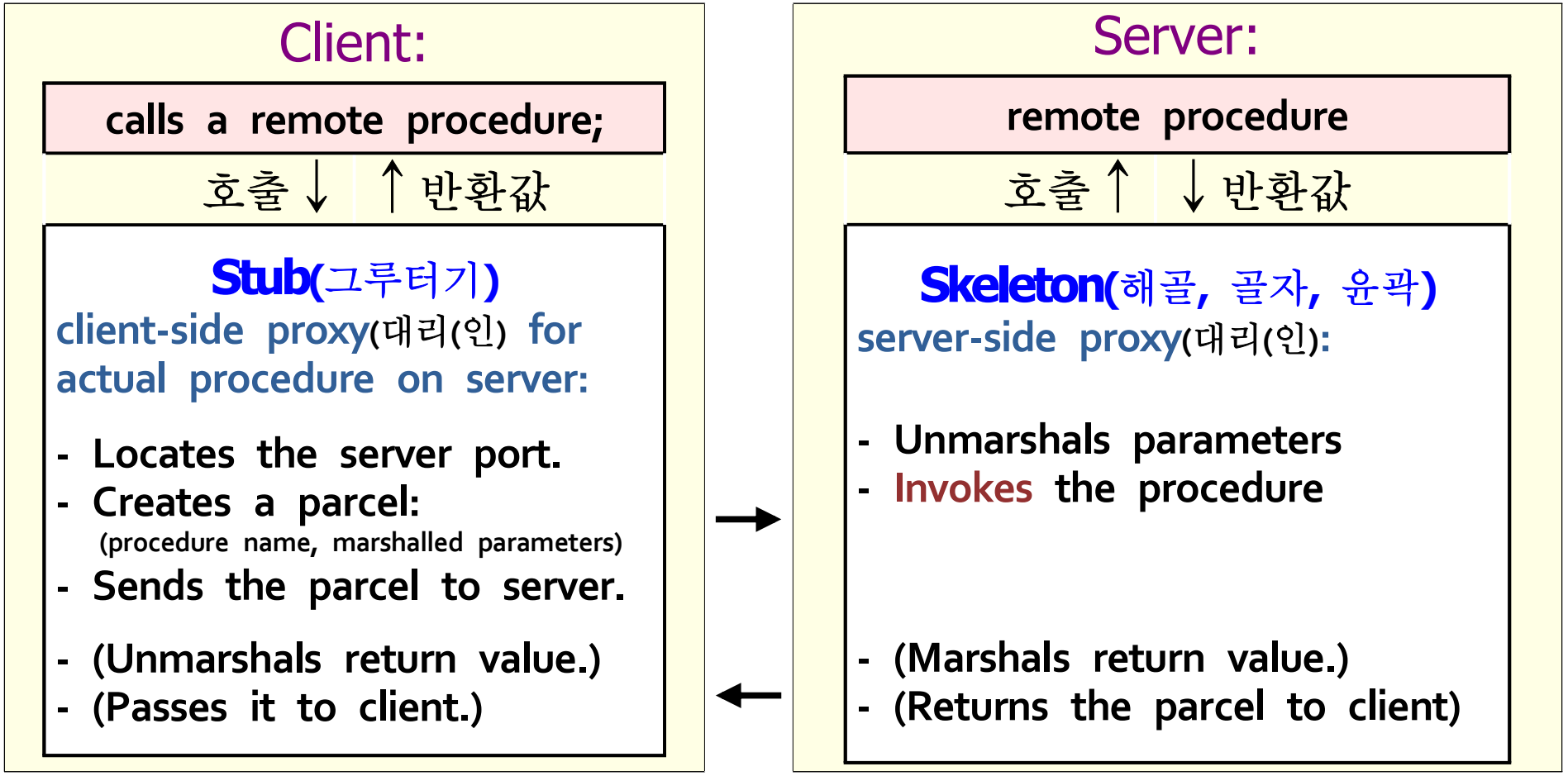
```
// Date 서버(port 2000)와 통신할
// 소켓을 생성하고 연결을 요청함.
Socket socket = new Socket(IP주소, 2000);

// 소켓의 InputStream에 연결될
// 입력스트림 생성; 읽기.
BufferedReader in = new BufferedReader(
        new InputStreamReader(
                socket.getInputStream()));

while ((line = in.readLine()) != null) {
        System.out.println(line);
}
```

```
import java.net.*;
import java.io.*;

public class DateServer {
    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(2000); // A server socket
            while (true) {
                // client socket that is dedicated to a client - comm. endpoint
                Socket sock = ss.accept(); // server socket listens at port 2000

                PrinterWriter pw =
                    new PrinterWriter(sock.getOutputStream(), true);
                pw.println(new java.util.Date().toString());
                sock.close();
            }
        } catch (IOException e) { System.err.println(e); }
    }
}
```

```
import java.net.*;
import java.io.*;

public class DateClient {
    public static void main(String[] args) {
        try {
            Socket socket = new Socket( "127.0.0.1", 2000 );

            BufferedReader in = new BufferedReader(
                new InputStreamReader(sock.getInputStream()));

            String line;
            while ((line = in.readLine()) != null) {
                System.out.println(line);
            }
            socket.close();
        } catch (IOException e) { System.err.println(e); }
    }
}
```

❑ Remote Procedure Call (RPC)

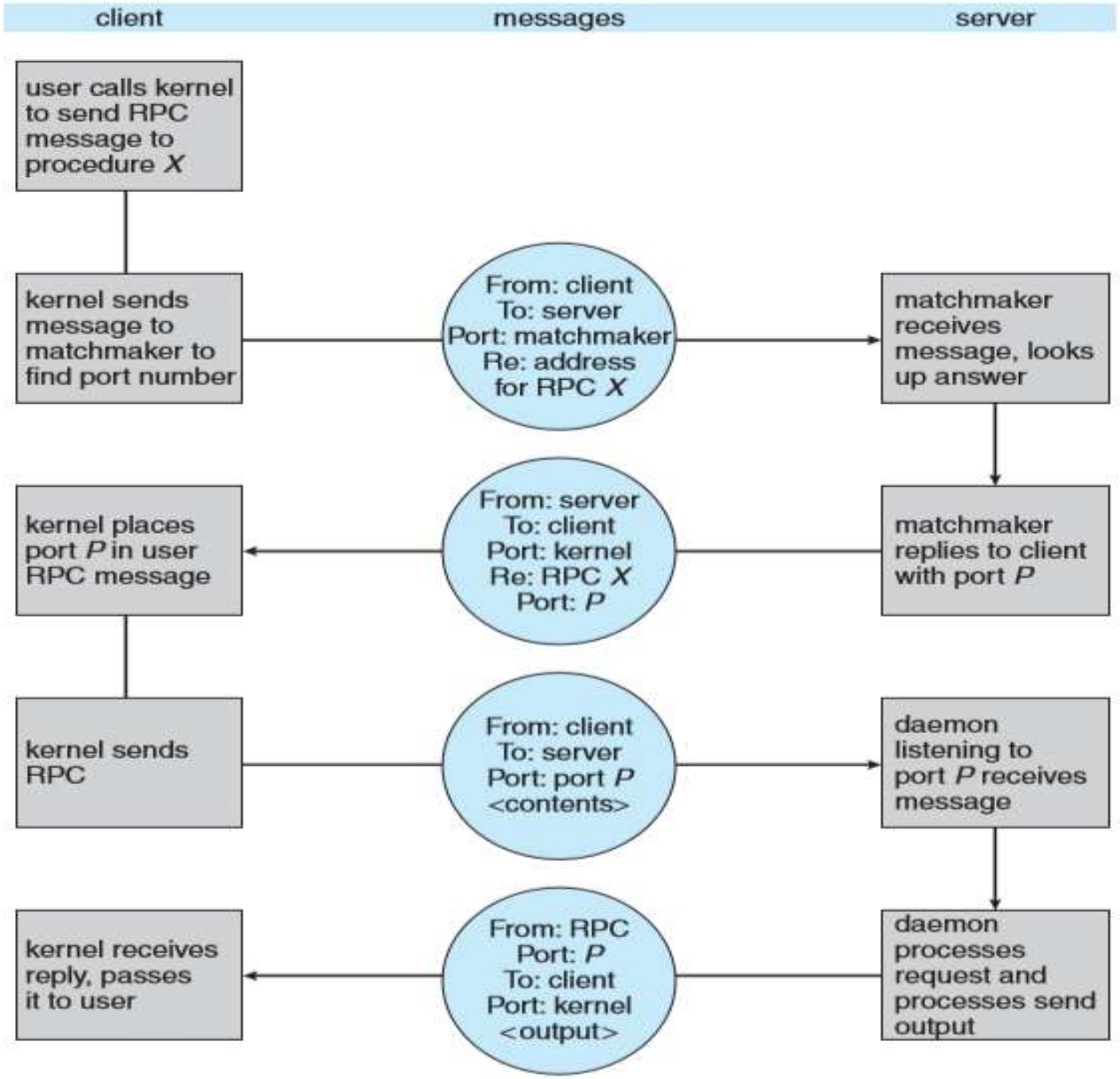
- Abstracts Procedure Calls between processes on networked systems.



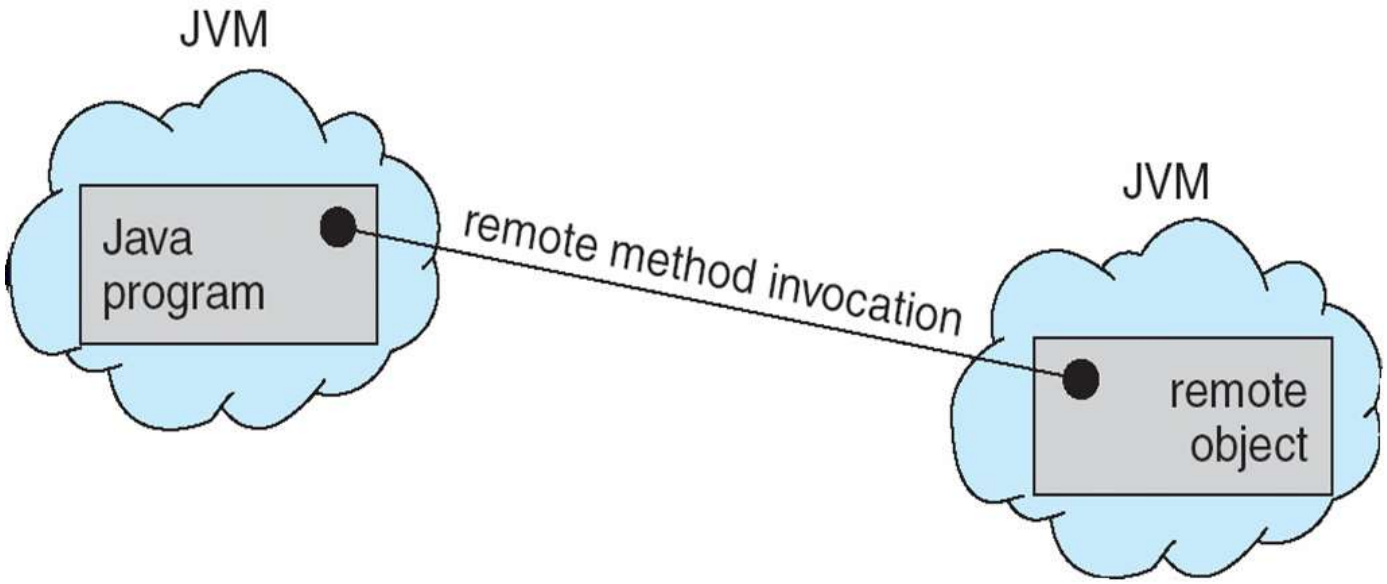
marshal = gather people or things together and arrange them for a particular purpose

unmarshal = decode from a marshalled state.

Execution of a remote procedure call (RPC)



(예) Java Remote Method Invocation (RMI)



- Must define a *remote interface* (note) server, client 모두 필요로 함.
- client와 remote object는 서로 다른 주소공간(address space)에 존재함.
- client-side proxy (called *stub*)와 server-side proxy (called *skeleton*)
  client와 remote object를 대신하여 실제로 통신을 수행함.

```
// Remote interface
public interface RemoteDate extends java.rmi.Remote {
    public Date getDate() throws RemoteException;
}
```

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class RMIServer extends UnicastRemoteObject implements RemoteDate {
    public RMIServer() throws RemoteException { }
    public Date getDate() throws RemoteException { return new Date(); }

    public static void main(String[] args) {
        try {
            RemoteDate ds = new RMIServer(); // remote object 생성.
            Naming.rebind("DateServer", ds);    // name server에 등록.
        } catch (Exception e) { }
    }
}
```

```
import java.rmi.*;

public class RMIClient {
   public static void main(String[] args) {
      try {
            String remoteServer = "rmi//" + "127.0.0.1" + "/DateServer";

            // Java RMI registry lookup to get a proxy for the remote object
            RemoteDate ds = (RemoteDate) Naming.lookup( remoteServer );
            System.out.println( ds.getDate() ); // remote object invocation
      } catch (Exception e) { }
   }
}
```

컴파일 및 실행:

1. 소스 코드 컴파일.
2. Java name server 실행: (UNIX) rmiregistry& (Windows) start rmiregistry
3. 서버 실행 後 클라이언트 실행.

❑ Pipe

- 명령에서 **Pipe** 사용

  (참고 - Unix) 두 프로그램(명령)의 *pipelining*

  앞 명령의 stdout이 pipe를 통해 뒤 명령의 stdin과 연결됨.

  (예) 주소파일에서 성이 "김"인 사람의 인원수 구하기:

  % cat 주소파일 | grep "^김" | wc -l

  - 주소파일: { <이름 주소> }
  - Regular expression 정규표현) 단어 패턴 기술 언어.

- **Pipe 란?**

  - conduit(도관, 전달자)처럼 행동하는 프로세스 間 통신 장치.

  - Design issues:

    - 통신

      단반향 unidirectional) 생산자-소비자 형태의 통신.

      생산자는 pipe의 write-end에 쓰기만 가능하고,

      소비자는 pipe의 read-end에서 읽기만 가능함.

      양방향 bidirectional) 서로 송수신이 가능함.

      반이중 half duplex) 동시 송신과 수신이 불가능함.

      전이중 full duplex) 동시 송신과 수신이 가능함.

    - 부모-자식 間 vs. 임의의 프로세스 間 통신 가능

    - local vs. remote 통신 가능

  - Ordinary pipe, Named pipe 두 종류가 있음.

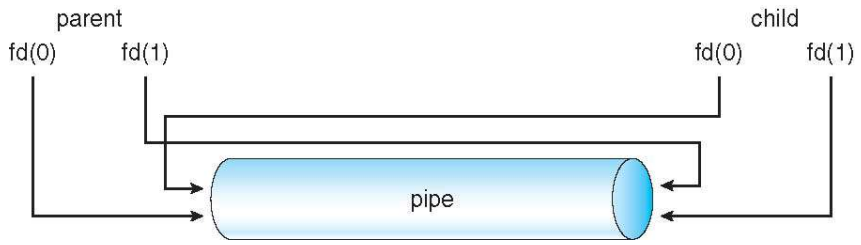
- **Ordinary Pipe**
  - only unidirectional
  - 오직 parent-child 간 통신만 가능 (note) parent가 생성하고 child는 상속함.
  - 파일처럼 취급함: (Unix) file read, write 시스템호출을 사용하여 통신함.

- **Named Pipe**
  - bidirectional
  - 임의의 프로세스들 間 통신 가능

Unix	Windows
• byte 단위 전송 • half duplex 통신만 가능 • local 내 통신만 가능	• byte 및 message 단위 전송 • full duplex 통신도 가능 • remote 間 통신도 가능
• FIFO라 부름 • 파일시스템에 정규 파일처럼 존재함 - persistent - 생성: mkfifo() 시스템호출 - 사용: open(), read(), write(), close() 시스템호출	

(예) Unix Ordinary pipe (교재 그림 3.25, 3.26 참고)

```
int fd[2]; pipe(fd); // pipe 생성.

pid_t pid = fork(); // child 생성.
```



0:READ_END, 1:WRITE_END.

```
if (pid > 0) { // parent
        close(fd[READ_END]);    // unused pipe end 닫음.
        write(fd[WRITE_END], 쓸 메시지, 길이); // pipe에 write.
        close(fd[WRITE_END]);
} else { // child
        close(fd[WRITE_END]); // unused pipe end 닫음.
        read(fd[READ_END], 읽은 메시지, 길이); // pipe에서 read.
        close(fd[READ_END]);
        printf("read %s", 읽은 메시지);
}
```