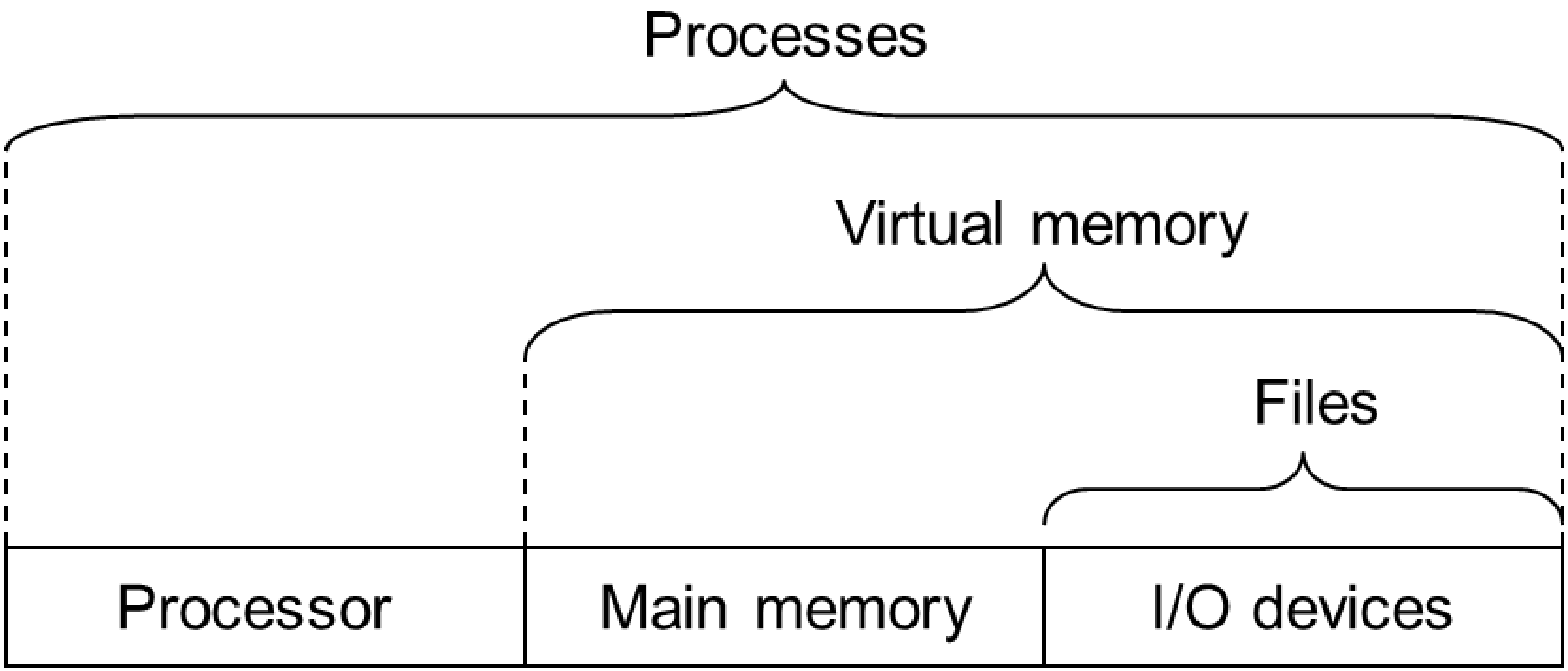
Running Programs on a System

Process Management

Process Concept

Process Management

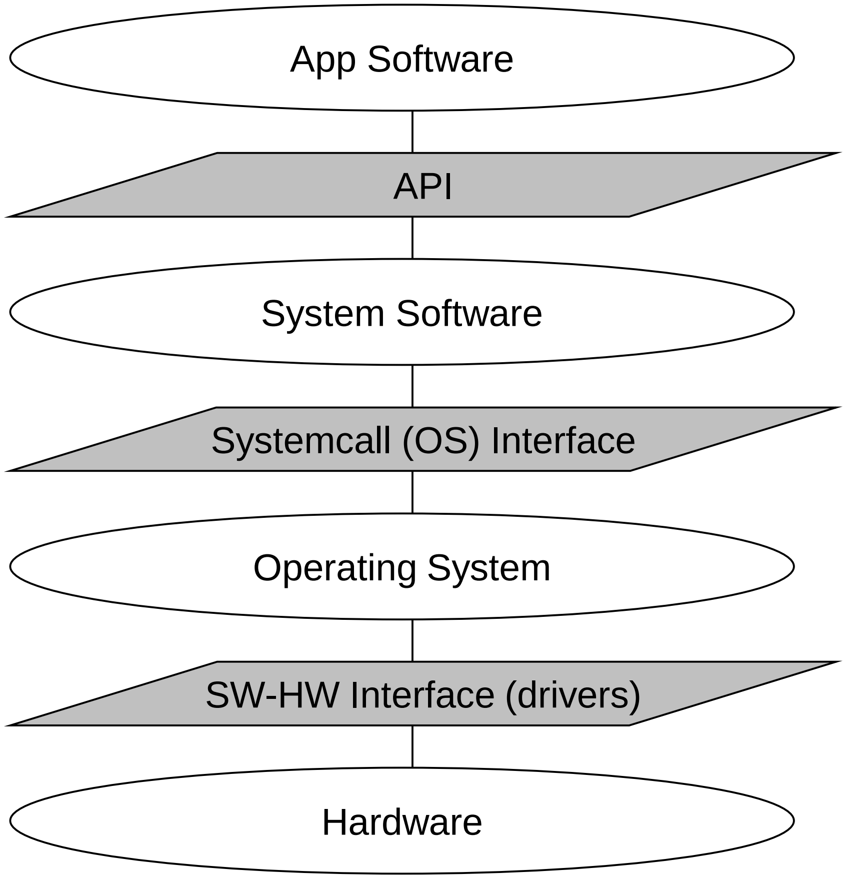
- The process is one of the fundamental abstractions in modern operating systems



CSE 컴퓨터공학부
Department of Computer Science & Engineering

Process Management

- Linux (system)calls for process management
  - `fork()`
  - `exec()` and variants
  - `wait()` and variants
  - `exit()`



Process Concept

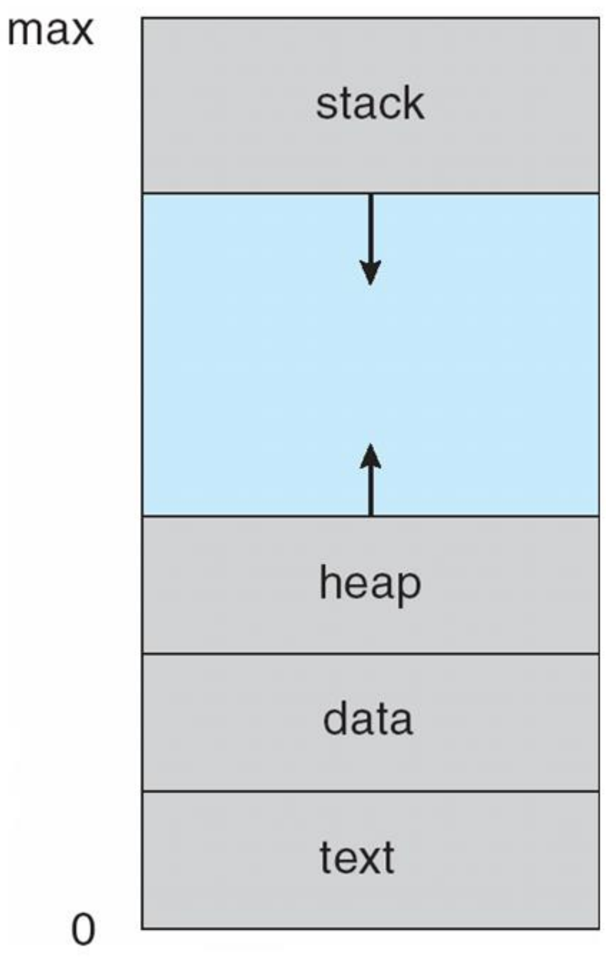
- Definition: A process is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
  - Private virtual address space
    - Each program seems to have exclusive use of main memory

- How are these Illusions maintained?
  - Process execution in sequence, interleaved, or run on separate cores
  - Address spaces managed by virtual memory system

Process Concept

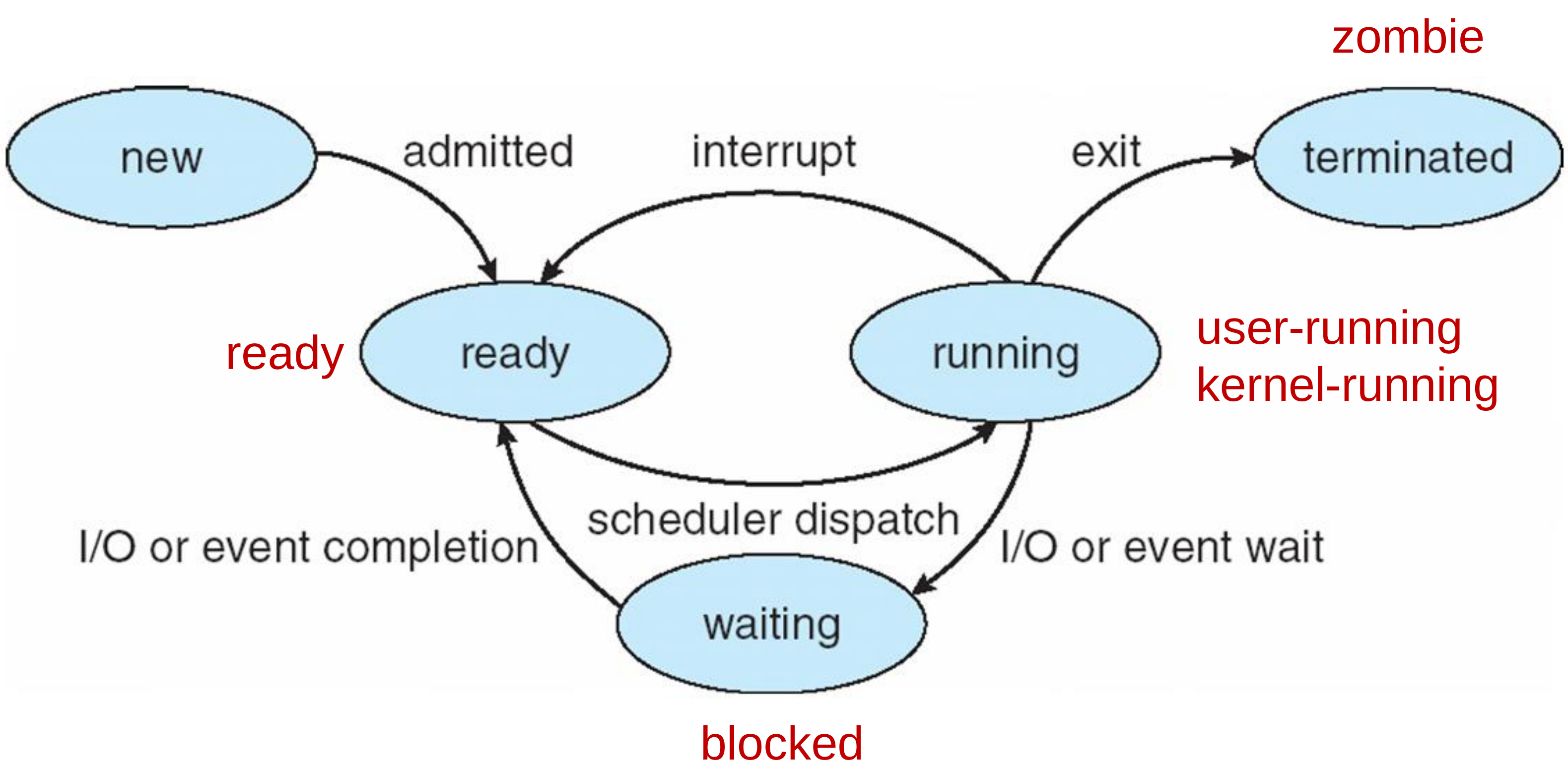
- A process includes:
  - program counter
  - stack
  - data section
  - open files



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

Diagram of Process State

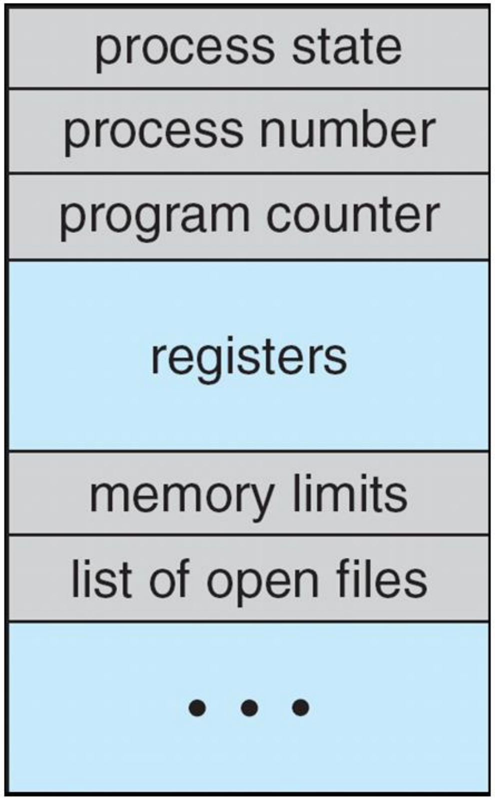


in red: Linux terminology

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Process Control Block (PCB)

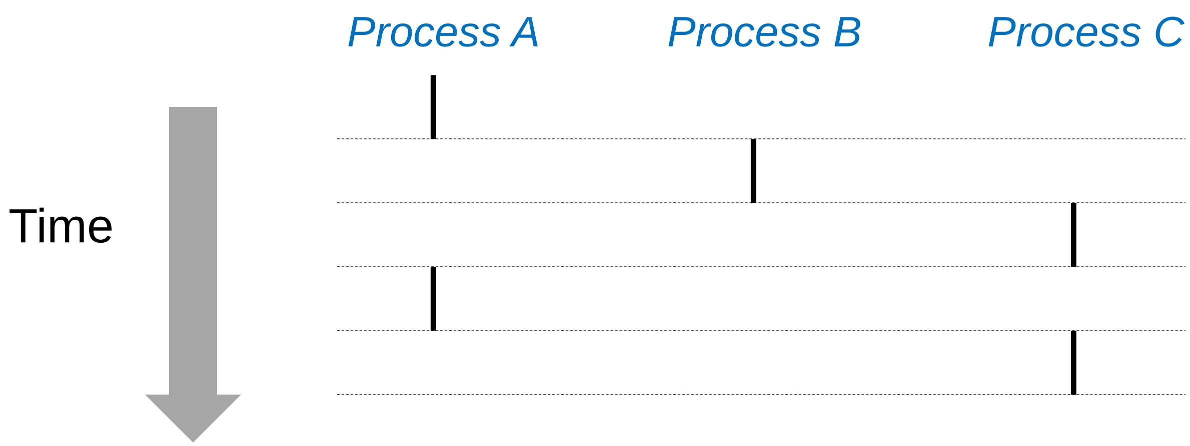
- Information associated with each process
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information



- In Linux: task_struct, thread_struct
  - include/linux/sched.h

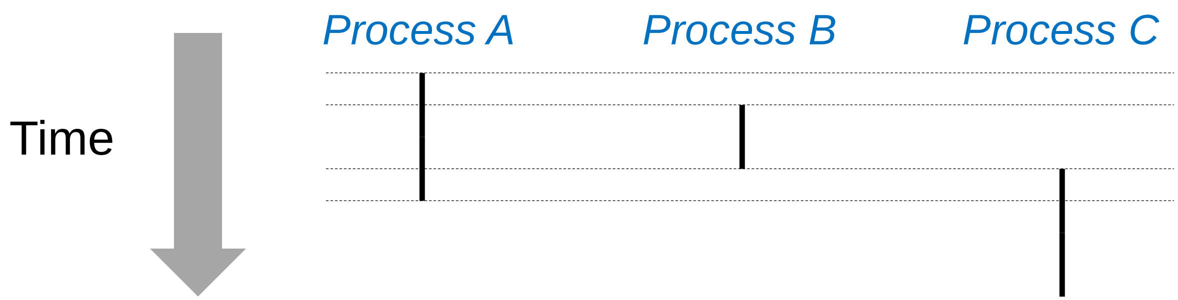
Concurrent Processes

- Two processes run concurrently (are concurrent) if their flows overlap in time
- Otherwise, they are sequential
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



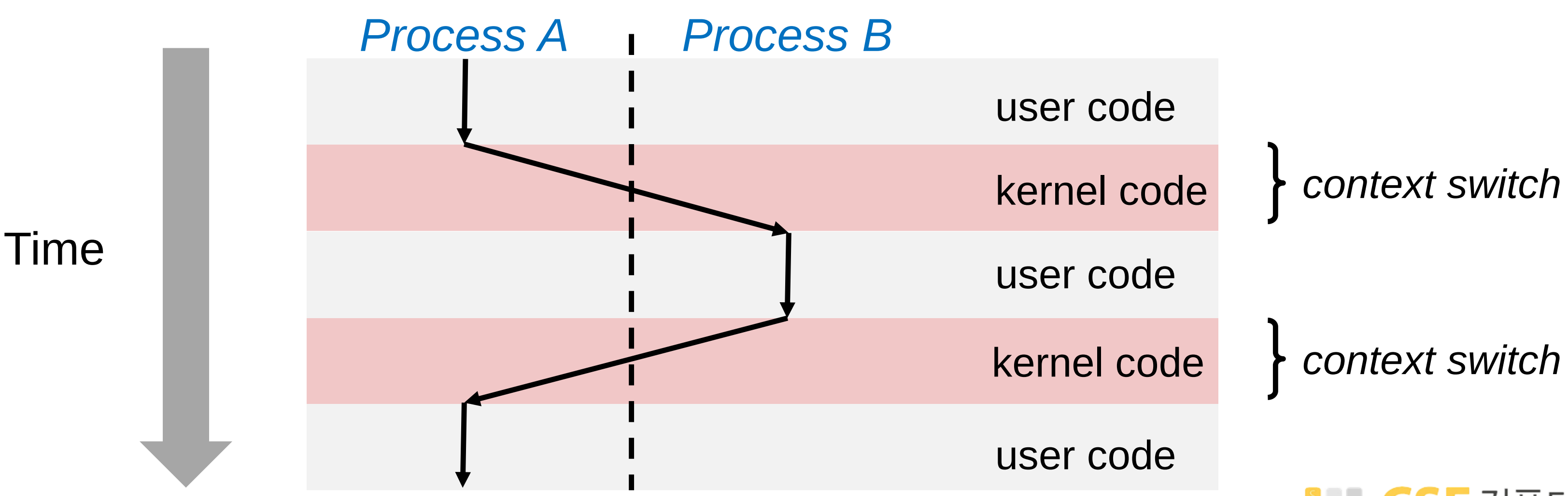
User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time
- However, we can think of concurrent processes are running in parallel with each other

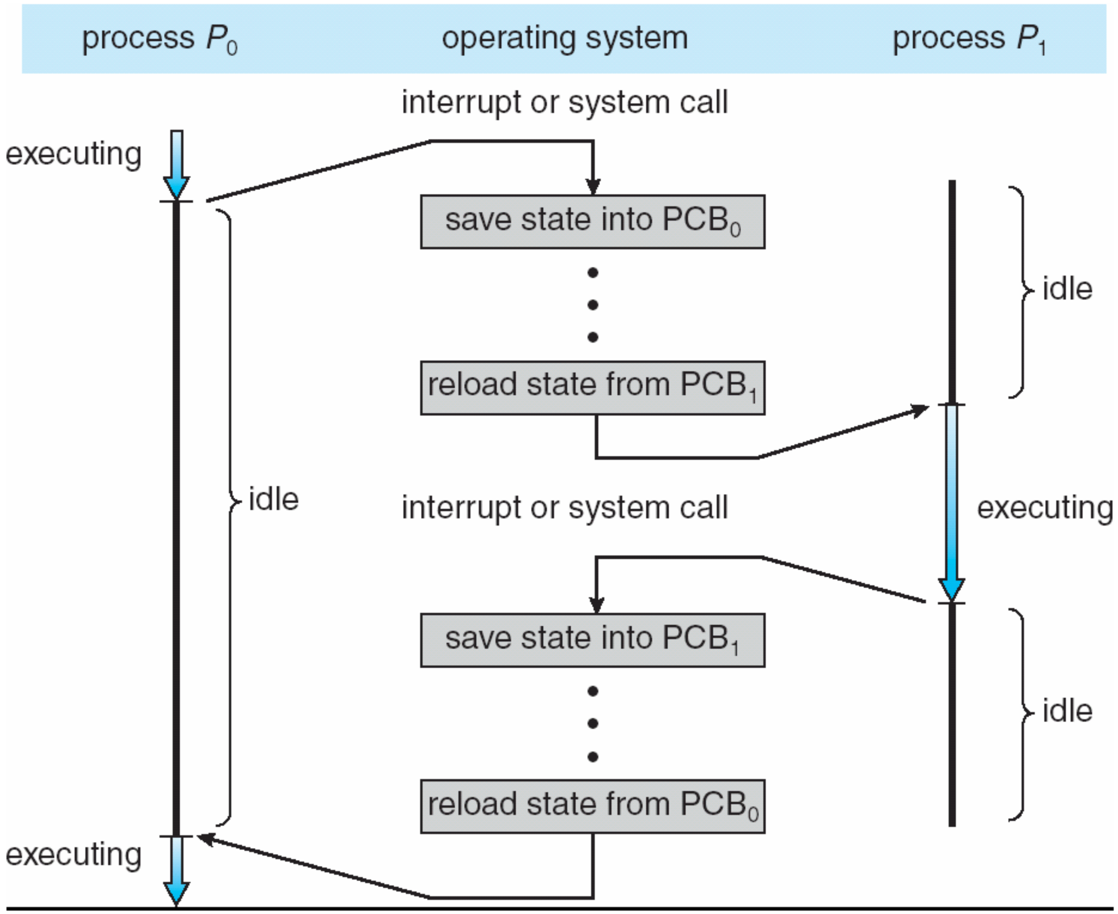


Context Switching

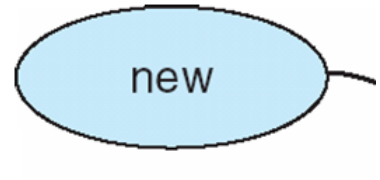
- Processes are managed by a shared chunk of OS code called the kernel
  - the kernel is not a separate process, but rather runs as part of some user process
- Control flow passes from one process to another via a context switch
  - save state of old process, load saved state for new process (in PCB)
  - Context-switch time is overhead; the system does no useful work while switching
  - Time dependent on hardware support



Context Switch Details



Process Creation



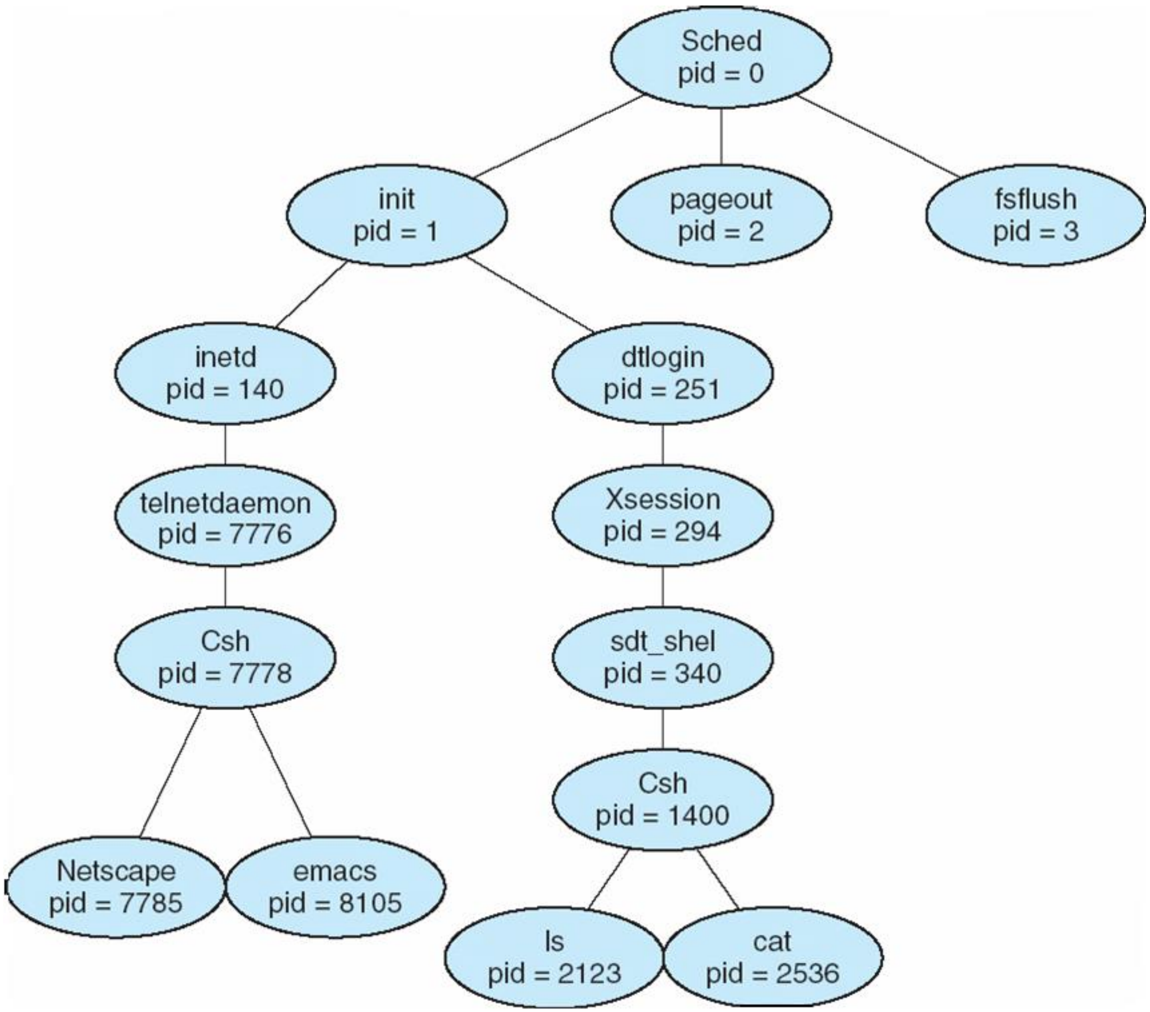
- A process is always created by a parent process
  - tree-like hierarchy of processes

- Process identified and managed via **a process identifier** (**PID**)

- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources

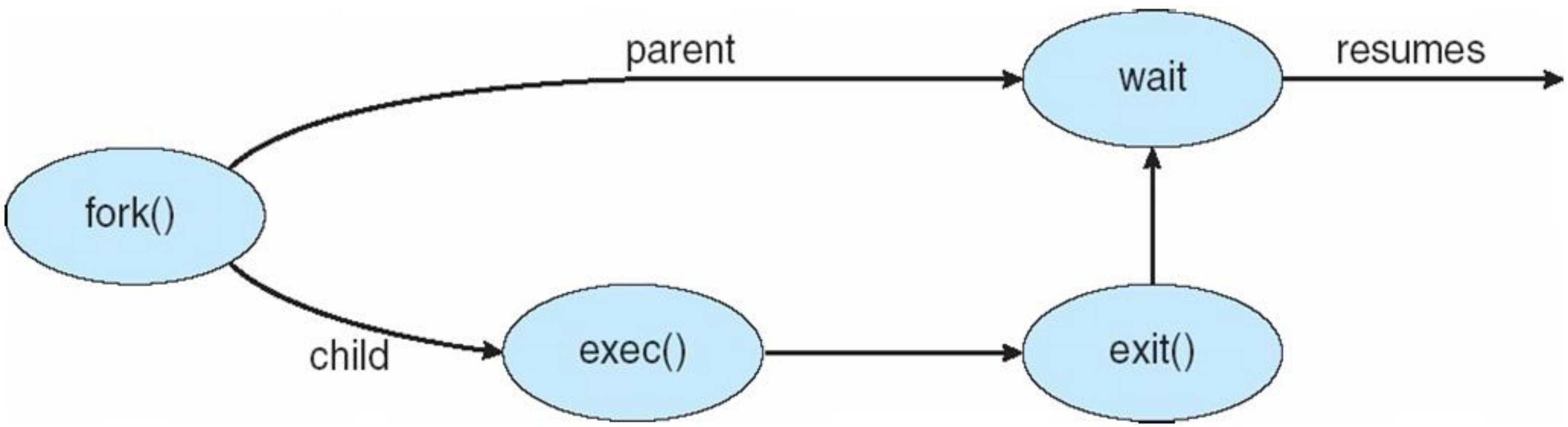
- Execution
  - Parent and children execute concurrently
  - Parent waits until child/children terminate

A tree of processes on a typical Solaris System



Process Creation (Cont)

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



C Program Forking Separate Process

```
int main()
{
        pid_t  pid;

        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
        }
}
```

Process Termination

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

- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - cascading termination

fork(): Creating New Processes

- **`int fork(void)`**
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child's pid to the parent process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

- `fork()` is interesting (and often confusing) because it is called *once* but returns *twice*

Understanding fork

Process n

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Child Process m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`hello from parent` *Which one is first?* `hello from child`

fork Example #1

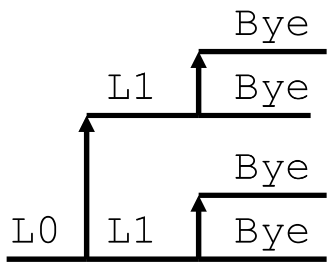
- Parent and child both run same code
  - Distinguish parent from child by return value from fork
- Start with same state, but each has private copy
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

fork Example #2

- Both parent and child can continue forking

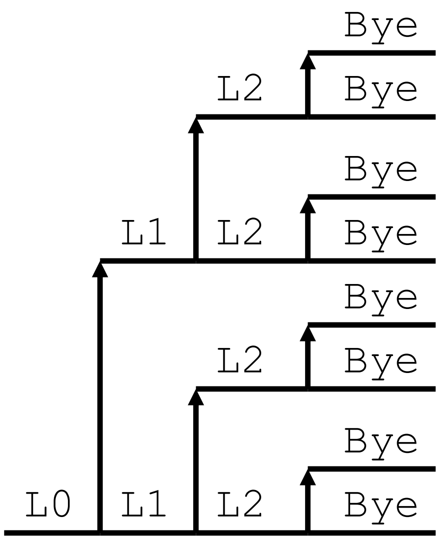
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



fork Example #3

- Both parent and child can continue forking

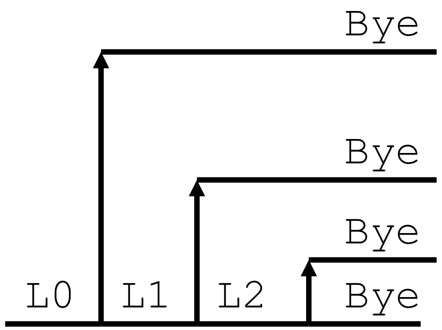
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



fork Example #4

- Both parent and child can continue forking

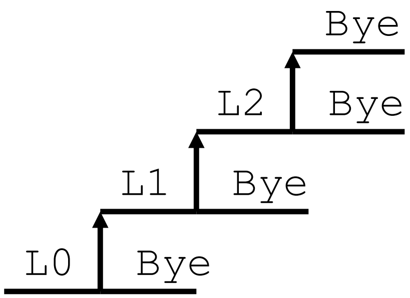
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



fork Example #5

- Both parent and child can continue forking

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
          printf("L2\n");
          fork();
       }
    }
    printf("Bye\n");
}
```



fork and Virtual Memory

- VM and memory mapping explain how fork provides private address space for each process.

- To create virtual address for new process
  - Create exact copies of current mm_struct, vm_area_struct, and page tables.
  - Flag each page in both processes as read-only
  - Flag each vm_area_struct in both processes as private COW

- On return, each process has exact copy of virtual memory

- Subsequent writes create new pages using COW mechanism.

exit(): Ending a process

- **`void exit(int status)`**
  - exits a process
    - Normally return with status 0
  - atexit() registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies

- Idea
  - When process terminates, still consumes system resources
    - Various tables maintained by OS
  - Called a “zombie”
    - Living corpse, half alive and half dead

- Reaping
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process

- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then child will be reaped by the init process
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Zombie Example

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- ps shows child process as "defunct"
- Killing parent allows child to be reaped by init

Nonterminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

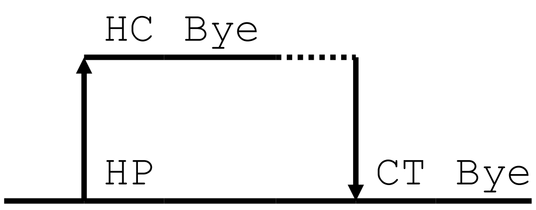
wait(): Synchronizing with Children

- **`int wait(int *child_status)`**
  - suspends current process until one of its children terminates
  - return value is the pid of the child process that terminated
  - if child_status != NULL, then the object it points to will be set to a status indicating why the child process terminated

wait: Synchronizing with Children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



wait() Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

waitpid(): Waiting for a Specific Process

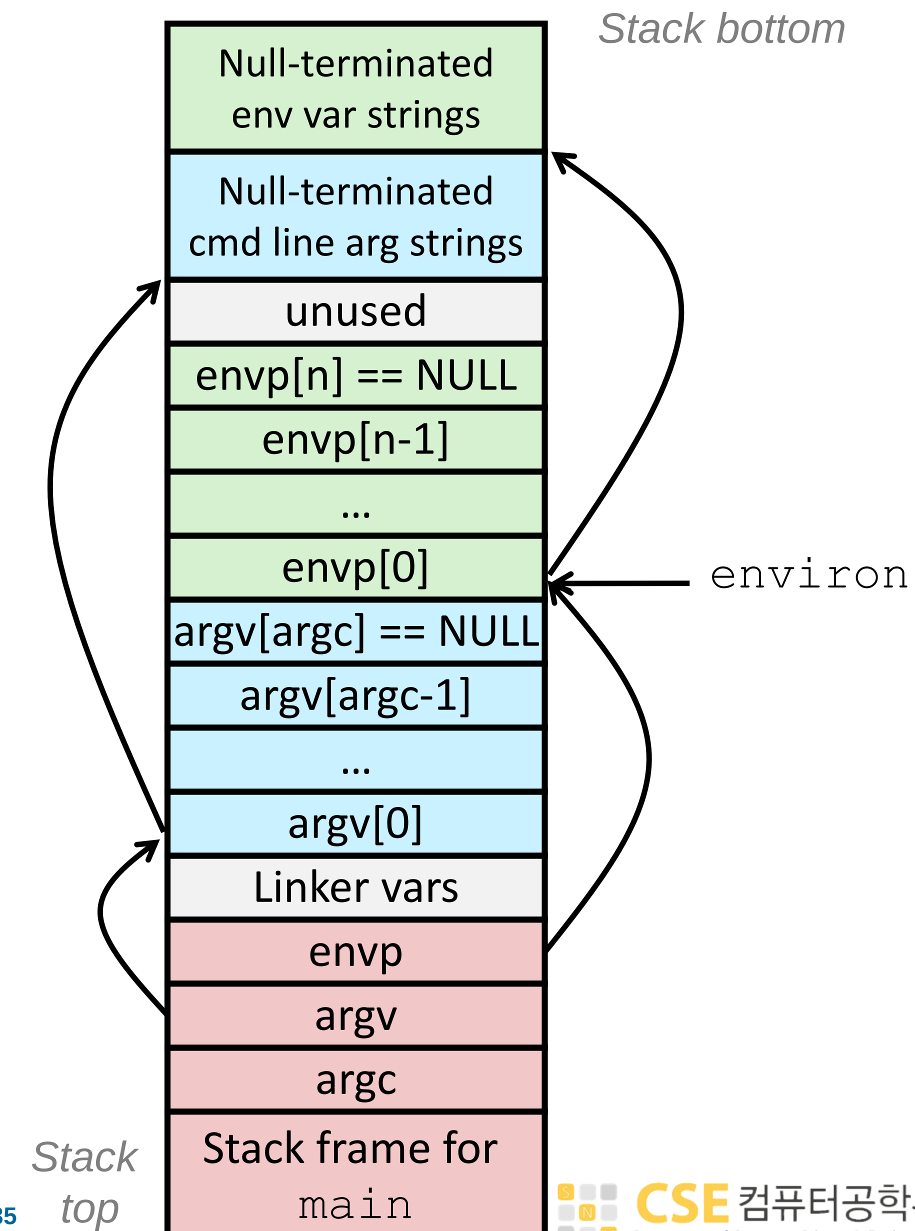
- **`waitpid(pid, &status, options)`**
  - suspends current process until specific process terminates
  - various options (see textbook)

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```


CSE 컴퓨터공학부
Department of Computer Science & Engineering

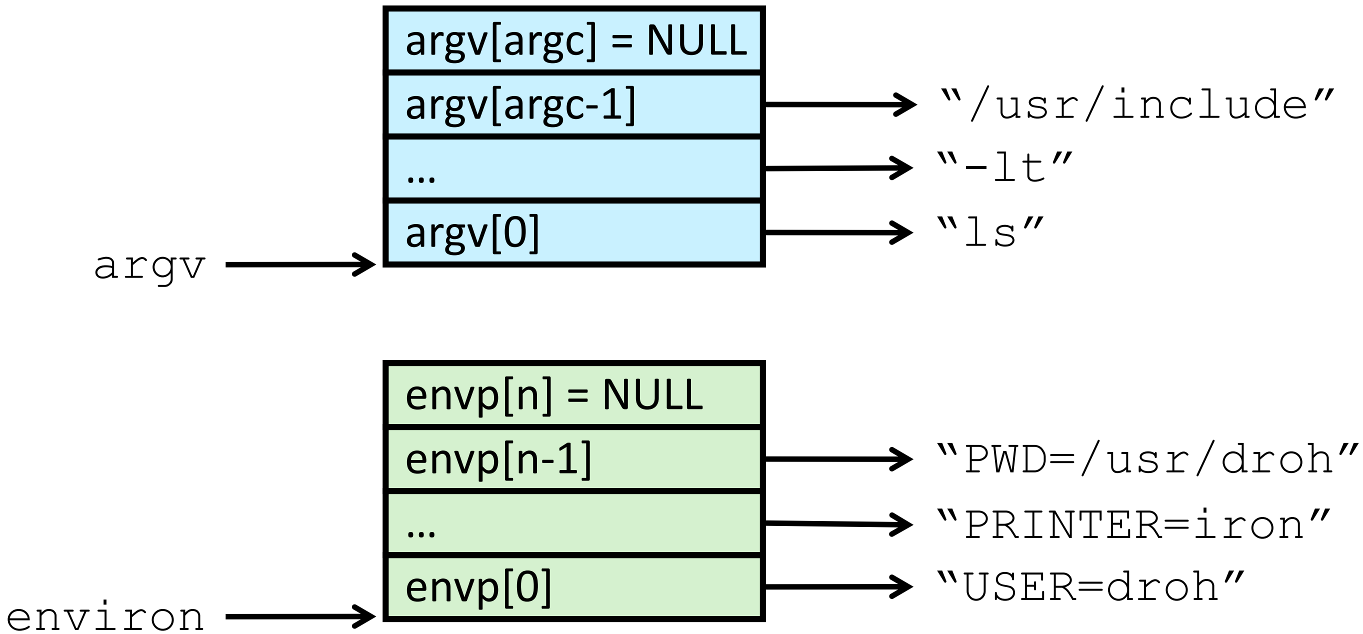
execve(): Loading and Running Programs

- ```
  int execve(
     char *filename,
     char *argv[],
     char *envp[]
  )
  ```
  - Loads and runs in current process:
    - executable `filename`
    - argument list `argv`
    - environment variable list `envp`
  - Does not return (unless error)
  - Overwrites code, data, and stack
    - keeps pid, open files and signal context
  - Environment variables:
    - "name=value" strings
    - getenv and putenv

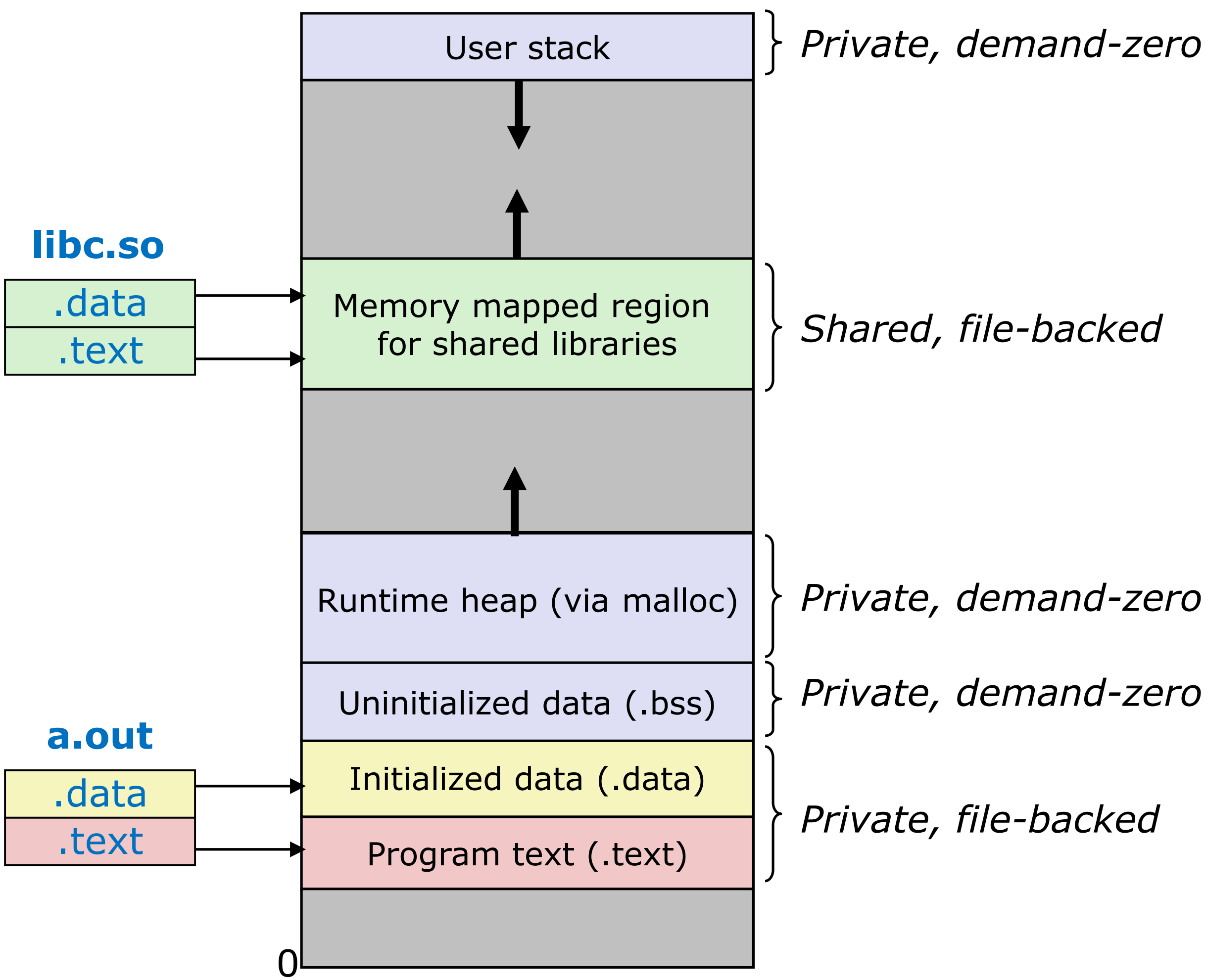

  ```

execve Example

```
if ((pid = Fork()) == 0) { /* Child runs user job */
    if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}
```



execve and Virtual Memory



- To load and run a new program in the current process using execve:
- Free vm_area_struct's and page tables for old areas
- Create vm_area_struct's and page tables for new areas
  - Programs and initialized data backed by object files.
  - .bss and stack backed by anonymous files .
- Set PC to entry point in .text
  - Linux will fault in code and data pages as needed

Signals

- higher-level form of exceptional control flow
- allows processes (and the kernel) to interrupt other processes

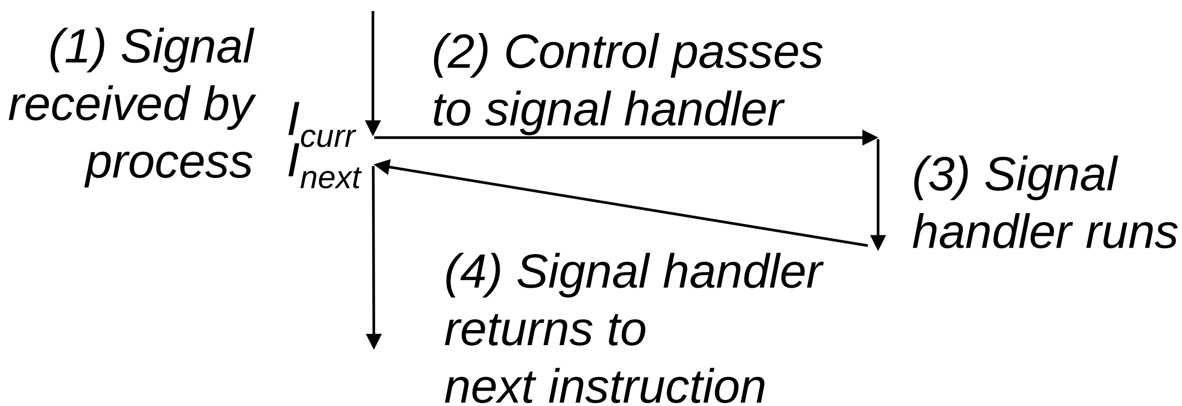
- different signals correspond to different events

#	Name	Default Action	Event
1	SIGHUP	terminate	terminal line hangup
2	SIGINT	terminate	interrupt from keyboard
3	SIGQUIT	terminate	quit from keyboard
4	SIGILL	terminate	illegal instruction
9	SIGKILL	terminate	kill program
11	SIGSEGV	terminate + coredump	segmentation fault (illegal memory ref)
14	SIGALRM	terminate	alarm() timer signal
15	SIGTERM	terminate	software termination signal
17	SIGCHLD	ignore	a child process has stopped or terminated
18	SIGCONT	ignore	continue process if stopped

→ see textbook fig. 8.25 for more details

Sending and Receiving Signals

- Signals are send by a program (or the kernel) and delivered by the kernel
  - /bin/kill: program to send arbitrary signals
  - keyboard: ctrl-c, ctrl-z
  - `int kill(pid_t, int sig)`
- The receiver can either ignore (block) the signal, terminate or catch the signal
- A signal that has been sent but not yet received is a *pending signal*



Signal Handling Issues

- Subtle issues when catching multiple signals
  - pending signals are blocked
  - pending signals are not queued
  - system calls may be interrupted

- Example: simple shell
  - parent process creates some children
  - children run independently and then terminate
  - parent must reap children to avoid leaving zombies in the system
  - parent must keep running while the children are running
    - use SIGCHLD handler to reap the children

Signal Handling Issues: Shell, Try 1

```c
#include "csapp.h"

void handler1(int sig) {
    pid_t pid;

    if ((pid = waitpid(-1, NULL, 0)) < 0)
            unix_error("waitpid error");
    printf("Handler reaped child %d\n", (int)pid);
    Sleep(2);
    return;
}

int main() {
    int i, n;
    char buf[MAXBUF];

    if (signal(SIGCHLD, handler1) == SIG_ERR)
            unix_error("signal error");
```

```c
    /* Parent creates children */
    for (i = 0; i < 3; i++) {
            if (Fork() == 0) {
                    printf("Hello from child %d\n",
                           (int)getpid());
                    Sleep(1);
                    exit(0);
            }
    }

    /* Parent waits for terminal input and then
       processes it */
    if ((n=read(STDIN_FILENO,buf,sizeof(buf))) < 0)
            unix_error("read");

    printf("Parent processing input\n");
    while (1)
            ;

    exit(0);
}
```

→ signals are blocked and not queued

Signal Handling Issues: Shell, Try 2

```
#include "csapp.h"

void handler2(int sig) {
    pid_t pid;

    while ((pid = waitpid(-1, NULL, 0)) > 0)
        printf("Handler reaped child %d\n",
        (int)pid);
    if (errno != ECHILD)
        unix_error("waitpid error");
    Sleep(2);
    return;
}

int main()
{
    int i, n;
    char buf[MAXBUF];

    if (signal(SIGCHLD, handler2) == SIG_ERR)
        unix_error("signal error");
```

```
    /* Parent creates children */
    for (i = 0; i < 3; i++) {
        if (Fork() == 0) {
            printf("Hello from child %d\n",
                (int)getpid());
            Sleep(1);
            exit(0);
        }
    }

    /* Parent waits for terminal input and then
       processes it */
    if ((n=read(STDIN_FILENO,buf,sizeof(buf))) < 0)
        unix_error("read");

    printf("Parent processing input\n");
    while (1)
        ;

    exit(0);
}
```

→ system calls can be interrupted

Signal Handling Issues: Shell, Try 3

```c
#include "csapp.h"

void handler2(int sig) {
    pid_t pid;

    while ((pid = waitpid(-1, NULL, 0)) > 0)
        printf("Handler reaped child %d\n",
        (int)pid);
    if (errno != ECHILD)
        unix_error("waitpid error");
    Sleep(2);
    return;
}


int main()
{
    int i, n;
    char buf[MAXBUF];

    if (signal(SIGCHLD, handler2) == SIG_ERR)
        unix_error("signal error");
```

```c
    /* Parent creates children */
    for (i = 0; i < 3; i++) {
        if (Fork() == 0) {
            printf("Hello from child %d\n",
                (int)getpid());
            Sleep(1);
            exit(0);
        }
    }

    /* Manually restart the read call if it is
       interrupted */
    while ((n = read(STDIN_FILENO, buf,
            sizeof(buf))) < 0)
        if (errno != EINTR)
            unix_error("read error");

    printf("Parent processing input\n");
    while (1)
        ;

    exit(0);
}
```

Signal Handling

- Portable signal handling
  - semantics differ from system to system
    - it's a big mess
  - portable Posix standard:

```
int sigaction(int signum, struct sigaction *act, struct sigaction *oldact);
```

- Read textbook chapter 8.5 carefully

Thread Concept

Threads vs Processes

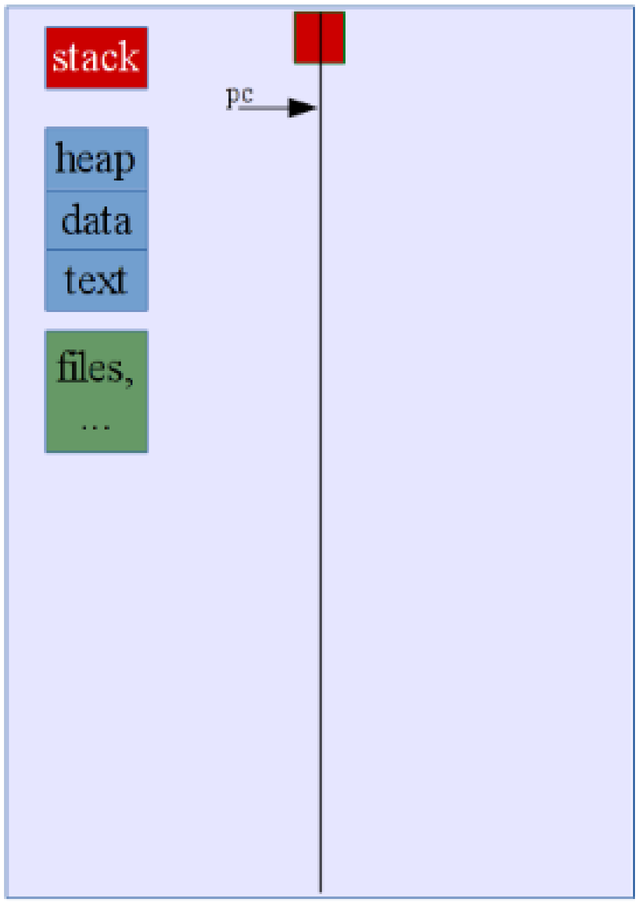
- A thread is an object of activity within a process
  - Sequential programs: one thread per process
  - Parallel programs: multiple threads per program

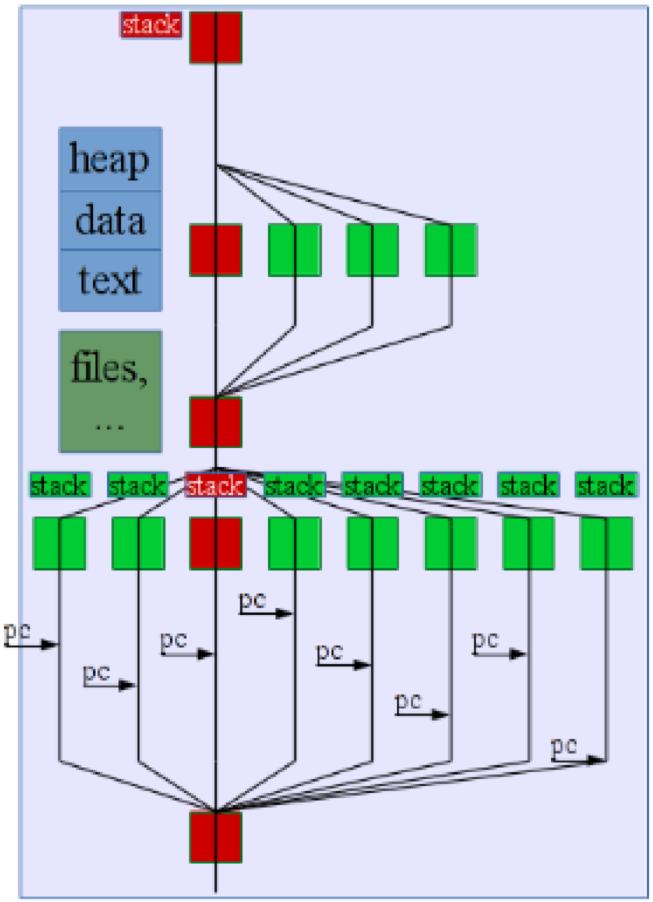
- Share the same address space and file descriptors
- Separate program counter, stack, registers

- The Linux kernel schedules “tasks”
  - task = unit of activity, process or thread
  - no distinction between (sequential) processes and a multi-threaded process

Threads vs Processes

- Process (single-threaded) vs. multi-threaded





Threads vs Processes

- Single-threaded vs multi-threaded

```
#include <stdio.h>

void count(int id, int N, unsigned W)
{
  int n,w,idle;

  for (n=0; n<N; n++) {
    // busy loop
    for (w=0; w<W; w++) {
      for (idle=0; idle<100000; idle++);
    }

    // print id & counter
    printf("[%05d] %'10d\n", id, n);
  }
}

int main(void)
{
  count(1, 100, 100);

  return 0;
}
```

same address space → race conditions?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void count(int id, int N, unsigned W)
{
 … // same as on the left
}

void *p_count(void *p)
{
  int id = *(int*)p;
  printf("[%05d] hi, this is thread %d.\n", id, id);

  count(id, 100, 100);

  return NULL;
}

int main(int argc, char *argv[])
{
  int i, n_threads = 0;
  pthread_t *t;

  if (argc > 1) n_threads = atoi(argv[1]);
  if (n_threads < 1) n_threads = 1;

  t = (pthread_t*)calloc(n_threads, sizeof(pthread_t));
  if (t == NULL) exit(1);

  printf("[main ] creating %d threads...\n", n_threads);
  for (i=0; i<n_threads; i++) {
    if (pthread_create(&t[i], NULL, p_count, &i) != 0) {
      printf("Failed to create thread %d\n.", i);
    }
  }

  printf("[main ] waiting for threads...\n");

  for (i=0; i<n_threads; i++) {
    if (pthread_join(t[i], NULL) != 0) {
      printf("Failed to join thread %d\n.", i);
    }
  }

  printf("[main ] done.\n");

  return 0;
}
```

Threading Libraries

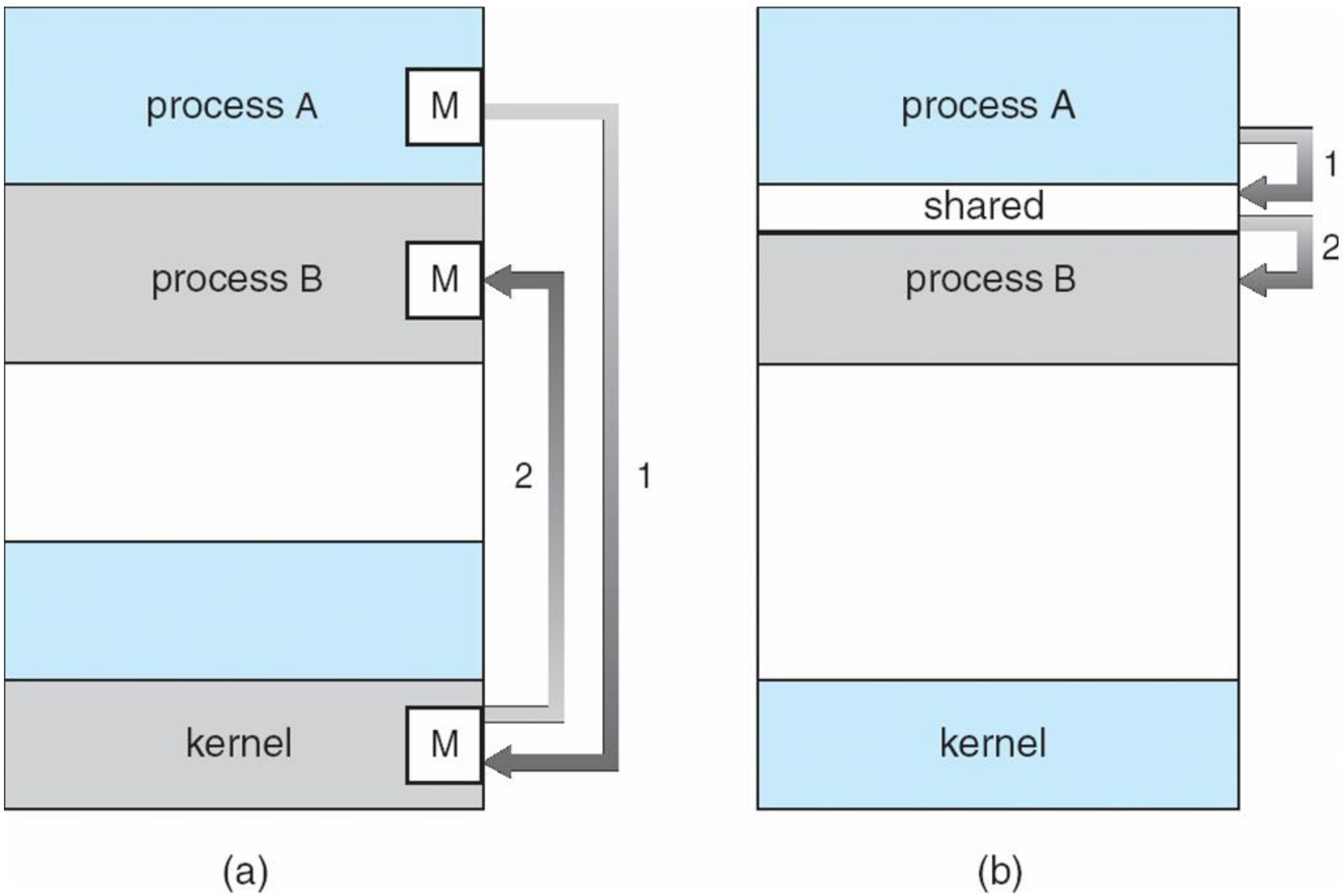
- pthreads
- OpenMP
- OpenCL
- MPI

Interprocess Communication

Interprocess Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models



Producer-Consumer Problem

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
  - 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;
```

- empty: in == out, full: ((in+1) % BUFFER_SIZE) == out
- 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;
}
```

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables

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

- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive

- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - send (P, message) – send a message to process P
  - receive(Q, message) – receive a message from process Q

- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox

- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - 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
  - 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

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    segment id = shmget(IPC PRIVATE, size, S IRUSR | S IWUSR);
  - Process wanting access to that shared memory must attach to it
    shared memory = (char *) shmat(id, NULL, 0);
  - Now the process could write to the shared memory
    sprintf(shared memory, "Writing to shared memory");
  - When done a process can detach the shared memory from its address space
    shmdt(shared memory);

CSE 컴퓨터공학부 Department of Computer Science & Engineering

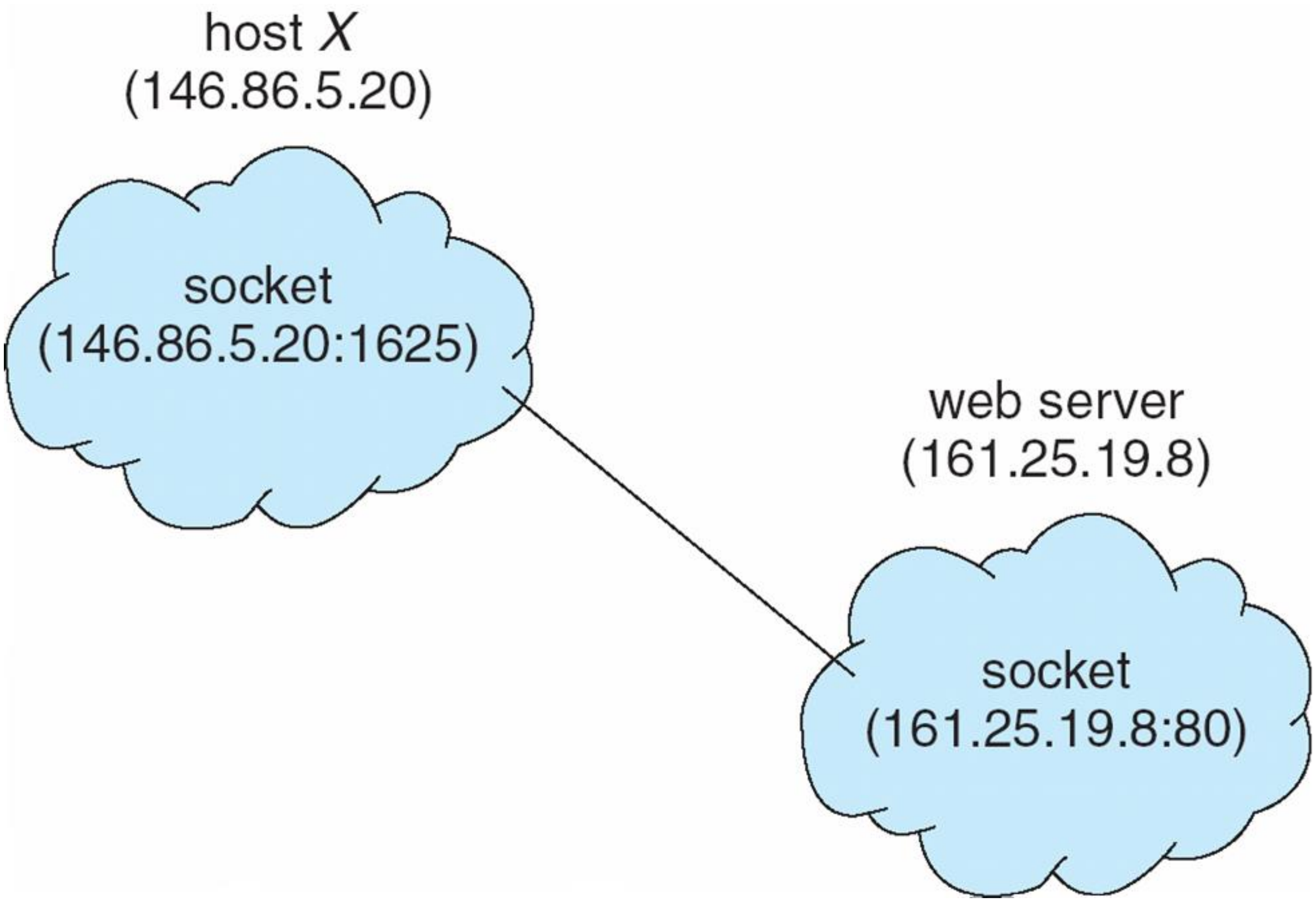
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
  (Remote Method Invocation (Java))
- Pipes

Sockets

- A socket is defined as an endpoint for communication
- Concatenation of IP address and port
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets

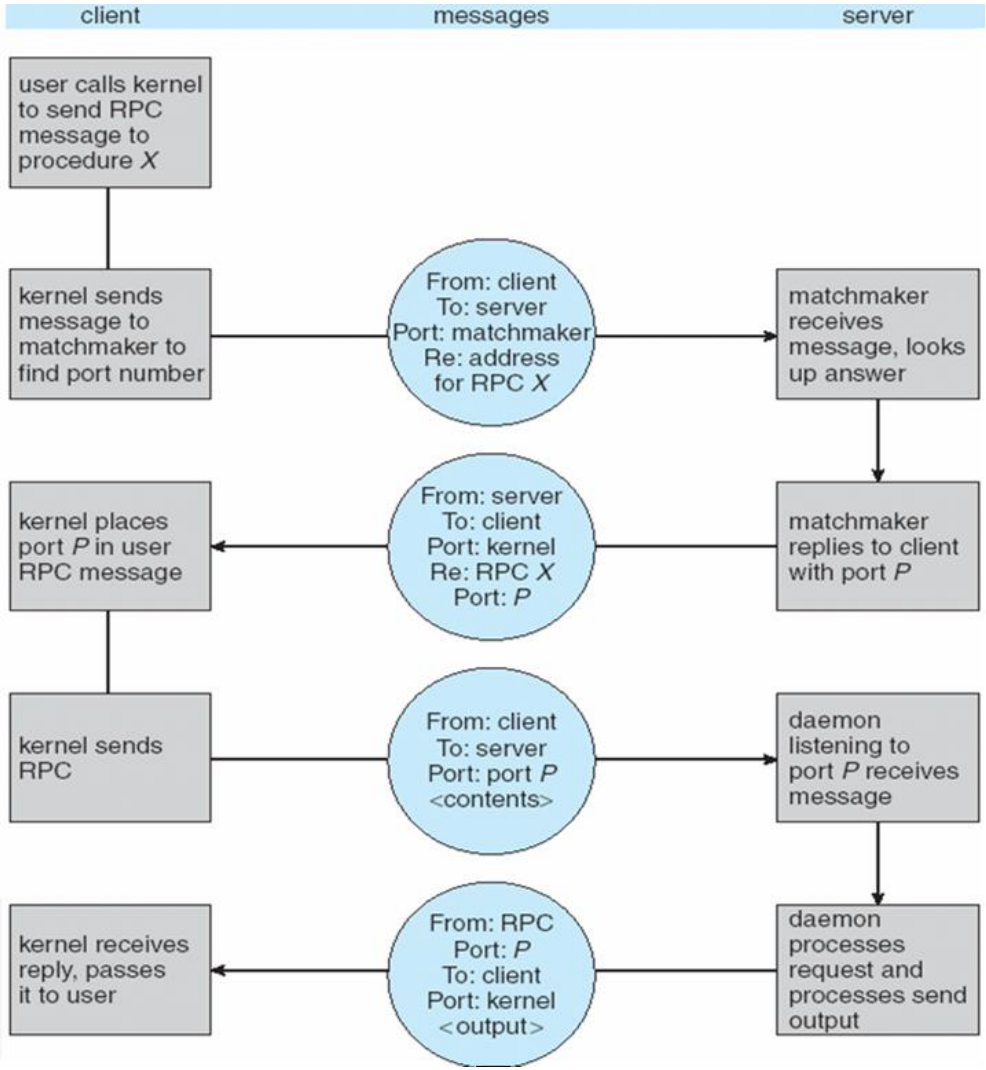
Socket Communication



Remote Procedure Calls

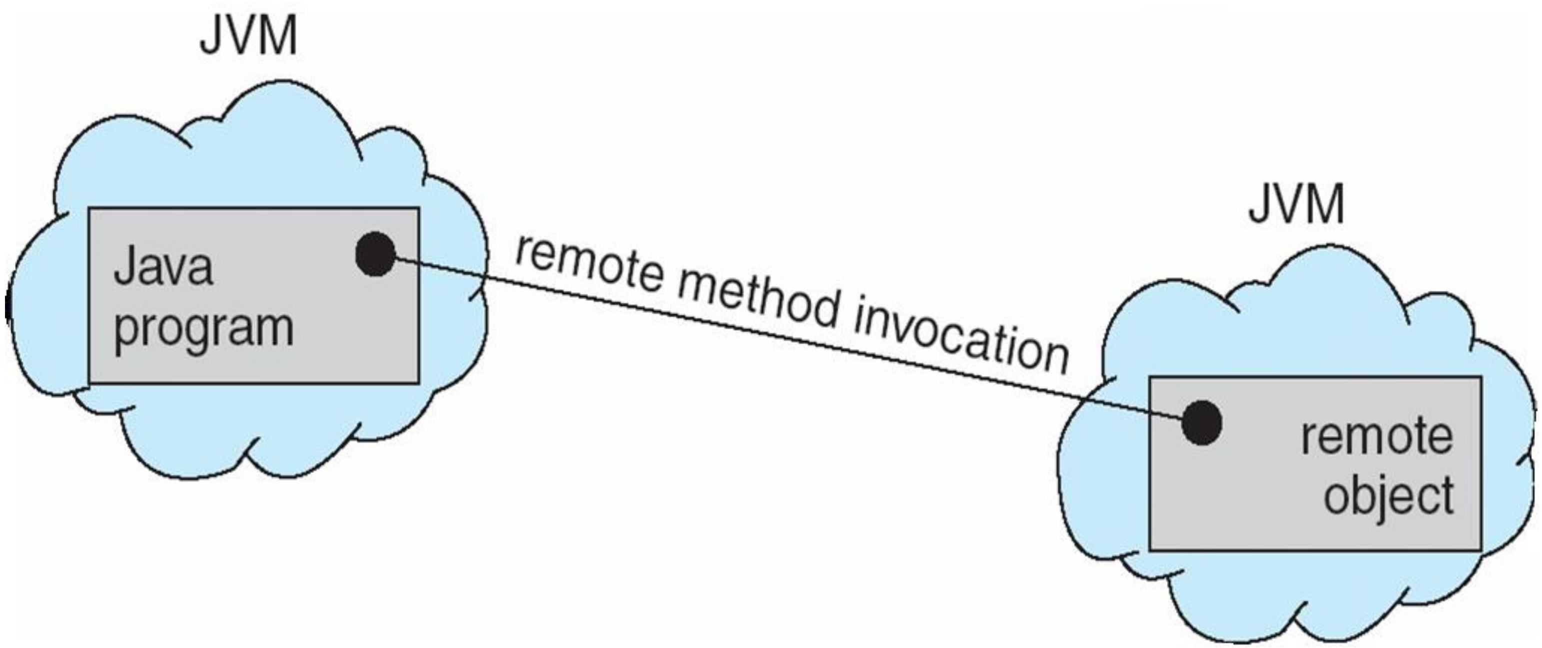
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- Stubs – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and marshalls the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server

Execution of RPC

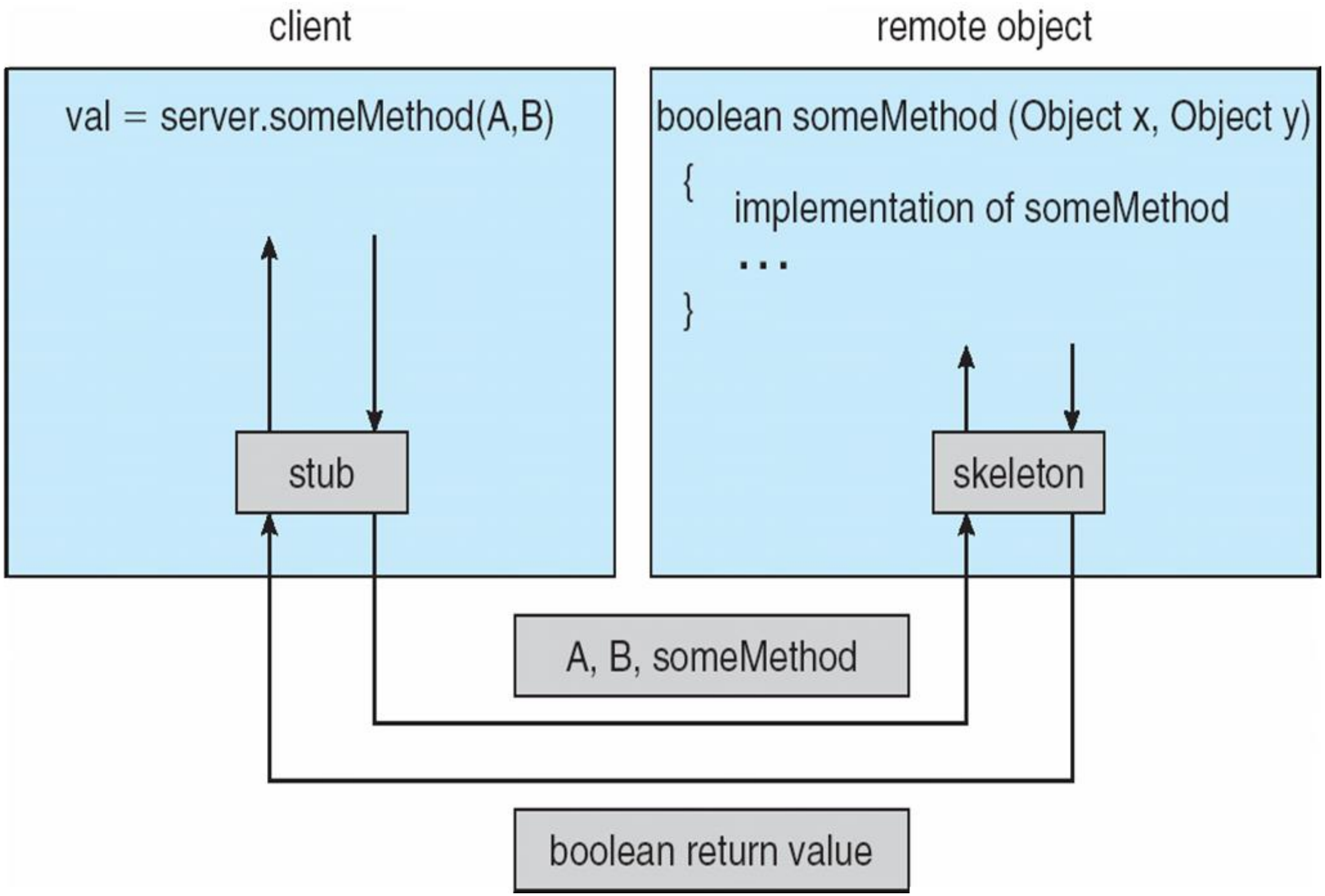


Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object



Marshalling Parameters



Pipes

- A pipe is a conduit allowing processes to communicate
- Unstructured contents ; structure imposed by communicating processes
- Issues to be considered
  - unidirectional/bidirectional?
  - for bidirectional pipes: half-duplex or full-duplex?
  - relationship between processes
  - local machine only or network transport possible?

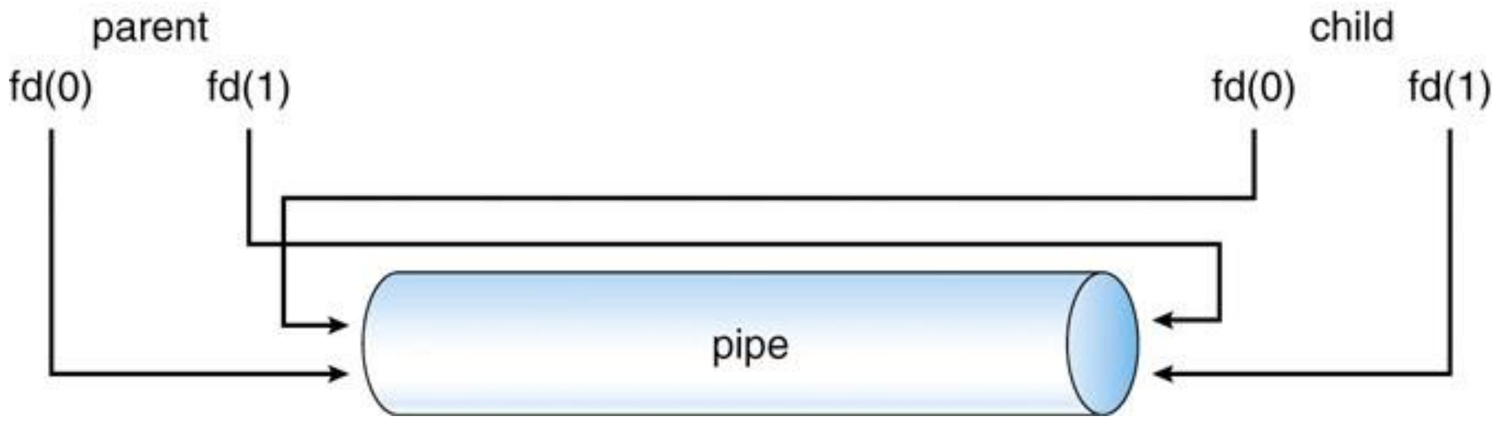
Ordinary Pipes

- In UNIX, pipe descriptors are file descriptors
  - use read()/write() to communicate, close() to close the pipe
  - pipe(int fd[]) opens a pipe (fd[0]: read, fd[1]: write)

- In Windows, ordinary pipes are called anonymous pipes
  - same concept as in UNIX (file handle, use standard read/write calls)

- ordinary pipes are
  - unidirectional
  - cannot be accessed from outside the process that creates it
    → requires a parent-child relationship for communicating processes
    → only local communication possible

UNIX Pipes



Named Pipes

- Named pipes
  - typically “live” in the file system
  - can be bidirectional, no parent-child relationship required
  - persistent
  - UNIX:
    - local machine only
    - half-duplex bidirectional
  - Windows:
    - local/remote communication
    - full-duplex bidirectional

Summary

- Processes
  - At any given time, system has multiple active processes
  - Only one can execute at a time on a single core, though
  - Each process appears to have total control of processor + private memory space

  - Spawning processes: fork()
    - One call, two returns
  - Process completion: exit()
    - One call, no return
  - Reaping and waiting for Processes: wait(), waitpid()
  - Loading and running Programs: execve() and variants
    - One call, no return (unless an error occurred)

Summary (cont.)

- Signals
  - slow & limited way to send events between related processes

- Interprocess communication
  - faster, less limitations
  - shared vs. message passing
  - many different ways to do it