Soongsil Univ.
http://design.ssu.ac.kr

Digital System Design Lab.

Ch. 03 Processes

Chanho Lee
Soongsil University

1

Objectives

- Components of a process : representation and scheduling
- Process creation and termination and using the system calls
- Interprocess communication(IPC) using shared memory and message passing.
  - ►Pipes and POSIX shared memory
  - ►client–server communication using sockets and remote procedure calls (RPC)
- Design kernel modules in Linux

2

Soongsil Univ.
http://design.ssu.ac.kr

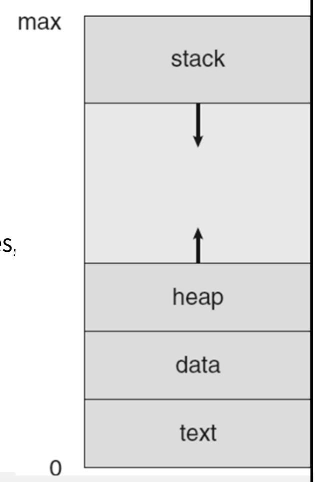
Digital System Design Lab.

4주차 process concept

3

3.1 Process Concept

- Processes
  - ▶Jobs (batch system) → tasks or user programs (time-shared system)
  - ▶Internal activities of OS
  - ▶Program(executable file, passive) in execution: active
  - ▶The status of the current activity of a process
    - program counter + processor's registers
  - ▶memory layout of a process
    - Text—the executable code
    - Data—global variables
    - Heap—dynamically allocated during run time
    - Stack—temporary data storage (such as function parameters, return addresses, and local variables → activation record)
  - ▶an execution environment for other code
    - Ex. JAVA: an executable Java program within the Java virtual machine (JVM)



Soongsil Univ.
Digital System Design Lab.

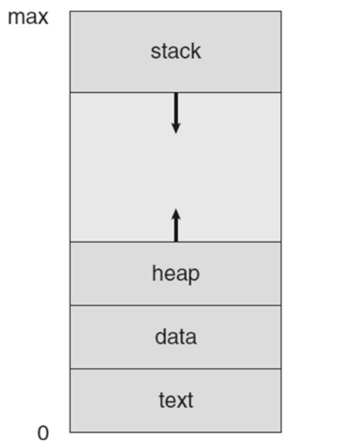
Operating Systems

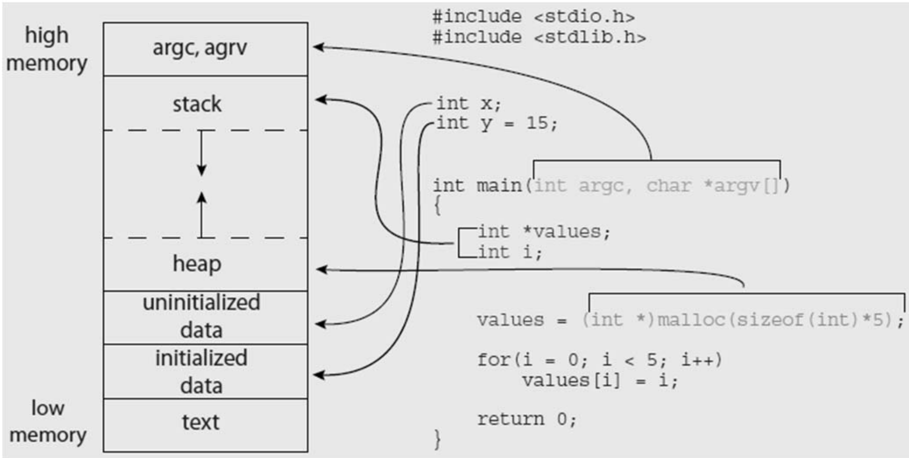
 4

4

3.1 Process Concept

- Memory layout of a C program



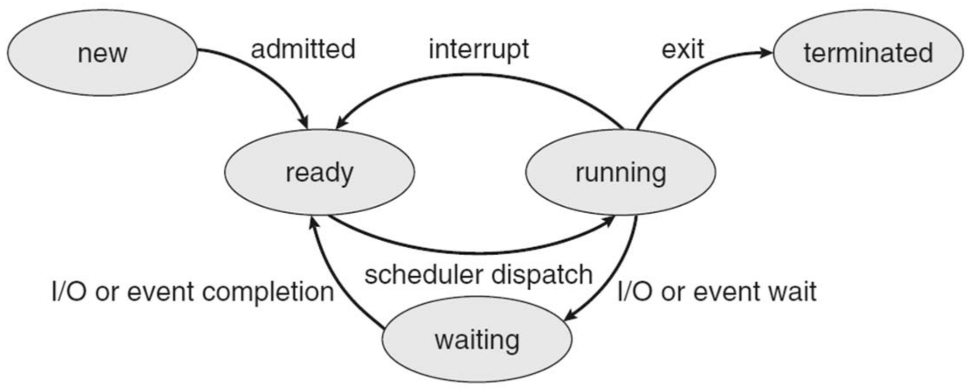



```
silicon@Si-VM:~/Documents/os$ size memory
   text    data     bss     dec     hex filename
   1603     604      12    2219     8ab memory
```

bss: block started by symbol

3.1 Process Concept

- Process State
  - ►defined in part by the current activity of that process



3.1 Process Concept

- Process Control Block (PCB)
  - ►task control block (TCB)
  - ►contains many pieces of information associated with a specific process
  - ►repository for all the data needed to start, or restart, a process, along with accounting data

PCB	
process state	
process number	
program counter	address of the next instruction
registers	**CPU-scheduling information**: process priority, pointers to scheduling queues, and any other scheduling parameters. (ch.5)
memory limits	**Memory-management information**: value of the base and limit registers and the page tables, or the segment tables, and so on. (ch.9).
list of open files	**Accounting information**: the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
• • •	**I/O status information**: list of I/O devices allocated to the process, a list of open files, and so on.

3.1 Process Concept

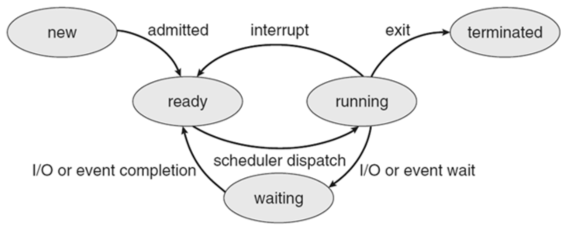
- Threads (ch.4)
  - ►Single thread of execution: a process to perform only one task at a time
    - ex. Word-processor: typing and then spell checking
  - ►Multiple threads of execution: a process to perform multiple tasks at a time
    - beneficial on multicore system
    - ex. Word-processor: typing and spell checking simultaneously (2 threads)
  - ►PCB includes thread information for multiple threads

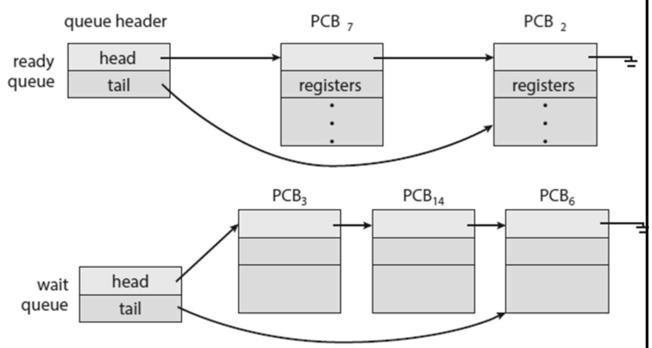
3.2 Process Scheduling

- Process scheduler
  - ►selects an available process
  - ►Multiprogramming: # of process > # of cores
    - Each CPU core can run one process at a time: waiting (ready) processes
    - Degree of multiprogramming: the number of processes currently in memory
  - ►CPU-bound or I/O bound process

The objective of multiprogramming: to maximize CPU utilization.
The objective of time sharing: frequent switching (multitasking)

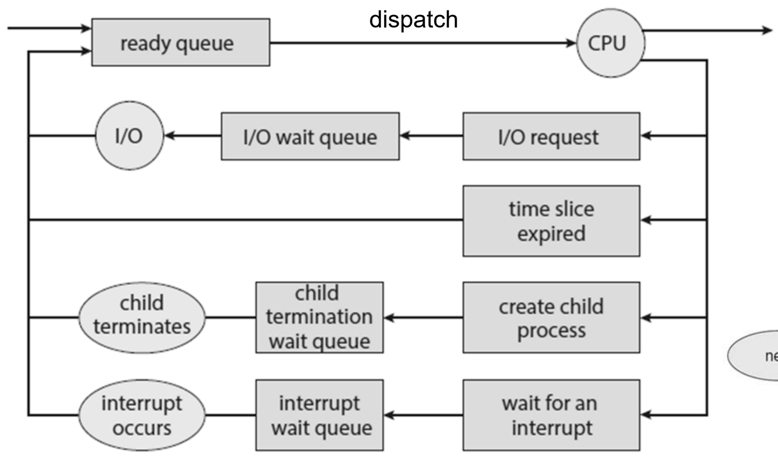
- Scheduling Queues
  - ►Ready queue: processes ready to run
  - ►Wait queue: processes waiting for events

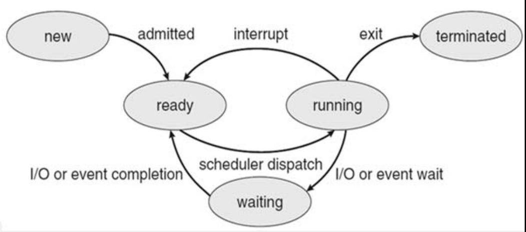




3.2 Process Scheduling

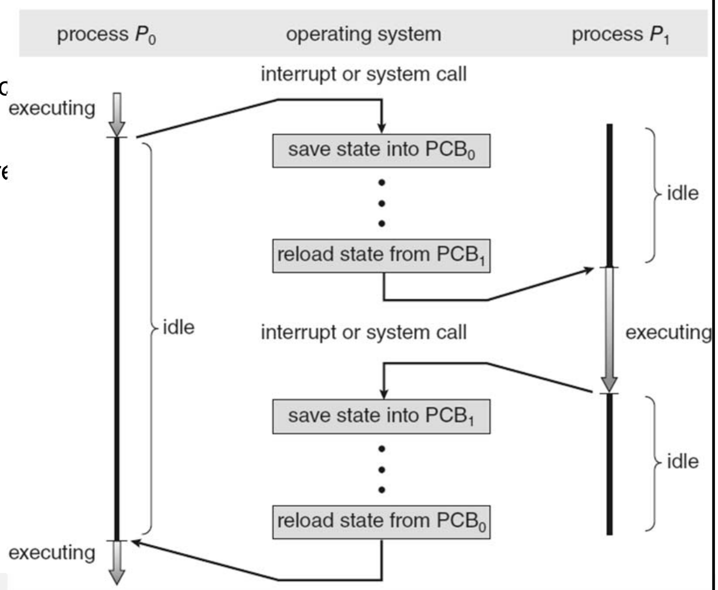
- Queueing diagram





3.2 Process Scheduling

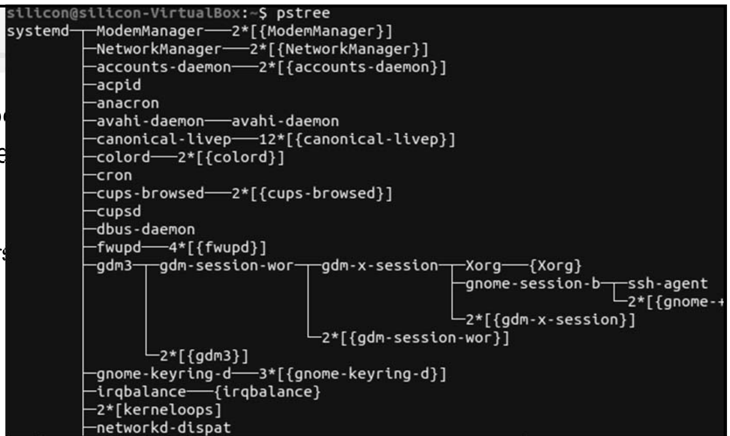
- **CPU Scheduling**
  - ►CPU scheduler: selects a process in the
    - I/o-bound and CPU-bound process
  - ►Swapping: remove a process from memc
- **Context Switch**
  - ►Context: includes the value of the CPU re information in PCB
  - ►State save and restore √
  - ►Context-switch time is pure overhead
    - Typically, a several μs



3.3 Operations on Processes

- **Process Creation**
  - ►A process may create several new pro
  - ►Unique process identifier (or pid): inde
  - ►Parent-child
  - ►Child: copy of PCB from parent

the firs
when



ps –el
ps -l
pstree

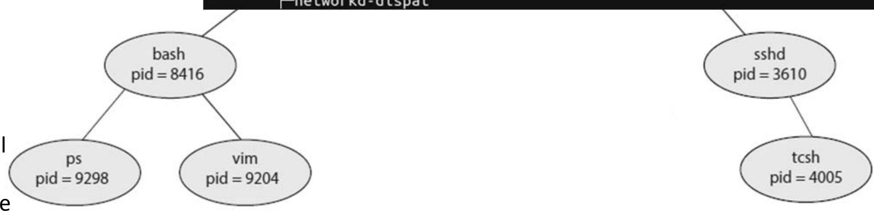


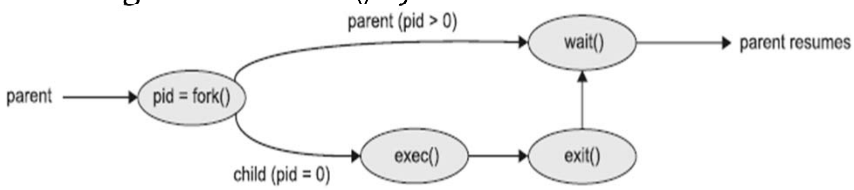
Figure 3.7 A tree of processes on a typical Linux system.

3.3 Operations on Processes

- Process Creation
  - ►Resources of child
    - From OS
    - From parent: constrained to a subset
  - ►Execution of parent
    - Concurrent with child
    - Waiting until the termination of child
  - ►Address-space possibilities for child
    - duplicate of the parent
    - new program loaded

3.3 Operations on Processes

- Creating a separate process using the UNIX fork() system call




```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
Pid_t pid;
  /* fork a child process */
  pid = fork(); //pid(p)>0: pid of child process; pid(c): 0
  if (pid < 0) { /* error occurred. Child starts here */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
  else if (pid == 0) { /* child process */
    execlp("/bin/ls","ls",NULL); // child ≠ parent now
  }
  else { /* parent process */
  /* parent will wait for the child to complete */
    wait(NULL); // transit to wait state
    printf("Child Complete");
  }
return 0;
}
```

3.3 Operations on Processes

- Process Termination
  - exit() system call in child
    - resources—including physical and virtual memory, open files, and I/O buffers—are deallocated
    - return a status value to waiting parent process (via wait())
      - Ex: exit status =1

```
/* exit with status 1 */
exit(1); //explicitly or implicitly
```

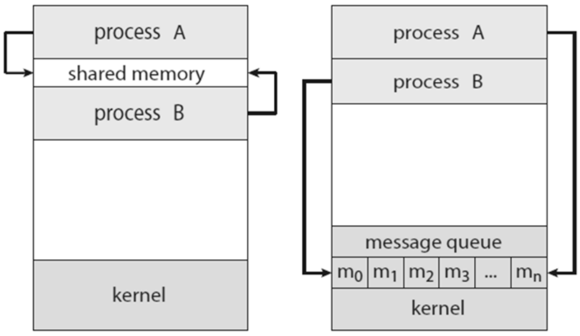
      - Ex.

```
pid_t pid;
int status;
pid = wait(&status); //parent gets exit status and child's pid
```

    - Zombie: child terminated before wait() is called → usually temporary
    - Orphan: parent terminated before calling wait() → init(systemd) is assigned as a new parent in UNIX/Linux
  - Termination by another process
    - By parent or OS
      - Excessive usage of resources by child
      - Task of child is no longer required
      - Parent is exiting or terminated abnormally (cascading termination)

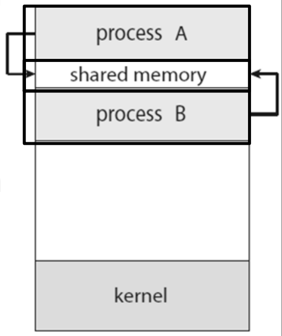
3.4 Interprocess Communication

- Sharing data or not: independent or cooperating
  - Information sharing: concurrent access to information.
  - Computation speedup: breaking into subtasks, executing in parallel with multiple cores.
  - Modularity: dividing system functions into separate processes or threads.
- Interprocess communication (IPC)
  - shared memory: faster than message passing
    - system calls are required only to establish shared memory regions
  - message passing
    - Typically implemented using system calls
    - useful for exchanging smaller amounts of data
    - easier to implement in a distributed system



3.5 IPC in Shared-Memory Systems

- Shared-memory
  - ►requires communicating processes to establish
  - ►Remove protection for shared memory
  - ►Read and write without OS's control
- Example of cooperating processes: producer–consumer problem
  - ►Concurrent operation of producer and consumer
  - ►a buffer of items in shared memory
  - ►filled by the producer and emptied by the consumer: synchronized
  - ►unbounded buffer/bounded buffer



3.5 IPC in Shared-Memory Systems

- Producer-consumer with bounded buffer

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} item;
item buffer[BUFFER_SIZE]; //circular array with two logical pointers: in, out
int in = 0; // empty when in=out
int out = 0; // full when ((in + 1) % BUFFER SIZE) == out
```

```
item next_produced;
while (true) {
/* produce an item in next_produced */
  while (((in + 1) % BUFFER_SIZE) == out)//full
  ; /* do nothing */
  buffer[in] = next_produced; //write
  in = (in + 1) % BUFFER_SIZE;
} //max BUFFER_SIZE-1 items in BUFFER: why?
```

```
item next_consumed;
while (true) {
  while (in == out) //empty
  ; /* do nothing */
  next_consumed = buffer[out]; //read
  out = (out + 1) % BUFFER_SIZE;
/* consume the item in next_consumed */
} //
```

3.6 IPC in Message-Passing Systems

- Message passing
  - ▶to communicate and to synchronize their actions without sharing the same address space.
  - ▶Operations: `send(message)` and `receive(message)`
  - ▶fixed or variable in size
  - ▶communication link:
    - ▪ implemented physically in a variety of ways
    - ▪ Logical implementation.
      - – Direct or indirect communication
      - – Synchronous or asynchronous communication
      - – Automatic or explicit buffering

3.6 IPC in Message-Passing Systems

- Naming
  - ▶Direct (symmetry)
    - ▪ send(P, message)—Send a message to process P.
    - ▪ receive(Q, message)—Receive a message from process Q.
    - ▪ A link is established automatically using identity
    - ▪ A link is associated with exactly two processes
    - ▪ Between each pair of processes, there exists exactly one link.
    - ▪ asymmetry
      - – send(P, message)—Send a message to process P.
      - – receive(id, message)—Receive a message from any process
    - ▪ limited modularity: changing the identifier requires examining all other definitions

3.6 IPC in Message-Passing Systems

- ►Indirect
  - messages are sent to and received from mailboxes, or ports
  - Mailbox: an object with a unique identification containing messages: placed and removed.
  - send(A, message)—Send a message to mailbox A. (A: integer for POSIX)
  - receive(A, message)—Receive a message from mailbox A.
  - A link is established with a shared mailbox
  - A link may be associated with more than two processes
  - Between each pair of processes, a number of different links with corresponding mailboxes are allowed
  - Mailboxes may be owned by a process or OS

3.6 IPC in Message-Passing Systems

- Synchronization
  - ►Blocking send: sending is blocked until the message is received or by the mailbox.
  - ►Nonblocking send: sends the message and resumes operation.
  - ►Blocking receive: receiver blocks until a message is available.
  - ►Nonblocking receive: receiver retrieves either a valid message or a null.
  - ►combinations of send() and receive() are possible
    - Both send() and receive() are blocking, we have a rendezvous: trivial solution to the producer–consumer problem

```
message next produced;
while (true) {
  /* produce an item in next produced */
  send(next produced);
}
```

```
message next consumed;
while (true) {
  receive(next consumed);
  /* consume the item in next consumed */
}
```


3.6 IPC in Message-Passing Systems

- Buffering: temporary queue implementation for IPC
  - ►Zero capacity:
    - maximum length of zero:
    - no waiting messages.
    - blocking only.
  - ►Bounded capacity.
    - finite length n: max n messages.
    - the sender is blocked if the link is full
  - ►Unbounded capacity:
    - potentially infinite length:
    - the sender is never blocked

3.7 Examples of IPC Systems

- POSIX Shared Memory
  - ►using memory-mapped files
    - `fd = shm_open(name, O_CREAT | O_RDWR, 0666); //Create a shared-memory object`

Integer file descriptor | Created if it does not yet exist | for reading and writing | file-access permissions

  - `ftruncate(fd, 4096);` //to configure the size of the object in bytes
  - `mmap()` function establishes a memory-mapped file containing the shared-memory object

3.7 Examples of IPC Systems

```
int main() //producer
{
  const int SIZE = 4096; //the size (in bytes) of shared memory object
  const char *name = "OS"; // name of the shared memory object
  /* strings written to shared memory */
  const char *message_0 = "Hello";
  const char *message_1 = "World!";
  int fd; //shared memory file descriptor
  char *ptr; //pointer to shared memory object
  fd = shm_open(name,O_CREAT | O_RDWR,0666); //create the shared memory object
  ftruncate(fd, SIZE); //configure the size of the shared memory object
  /* memory map the shared memory object */
  ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0); //read and write

  /* write to the shared memory object */
  sprintf(ptr,"%s",message_0); //write formatted string to the shared-memory object
  ptr += strlen(message_0);
  sprintf(ptr,"%s",message_1);
  ptr += strlen(message_1);
  return 0;
}
```

start

offset

changes to the shared-memory object will be visible

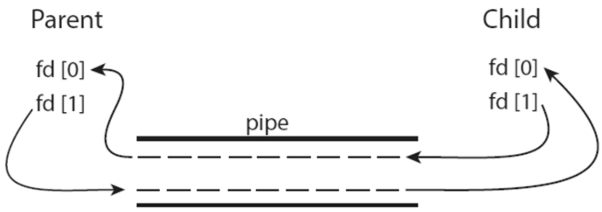
```
gcc -o producer producer.c -lrt
```


3.7 Examples of IPC Systems

```
int main() //consumer
{
  const int SIZE = 4096; //the size (in bytes) of shared memory object
  const char *name = "OS"; // name of the shared memory object
  /* strings written to shared memory */
  int fd; //shared memory file descriptor
  char *ptr; //pointer to shared memory object
  fd = shm_open(name, O_RDONLY, 0666); //open the shared memory object
  /* memory map the shared memory object */
  ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  /* read from the shared memory object */
  printf("%s",(char *)ptr);
  shm_unlink(name); //remove the shared memory object
  return 0;
}
```

3.7 Examples of IPC Systems

- Pipes
  - ▶ simple ways for IPC
  - ▶ Implementation issues
    - Bidirectional or unidirectional
    - half duplex or full duplex
    - Relationship (such as parent–child) required?
    - communicate over a network or not?
  - ▶ Ordinary Pipes
    - Unidirectional
    - cannot be accessed from outside; typically, IPC between parent-child
      - child inherits the pipe
    - Two pipes for two-way communication
      - `pipe(int fd[])`: fd[0]: read end, and fd[1]: write end
      - special type of file: `read()` and `write()` system calls
    - exist only while the processes are communicating with one another



3.7 Examples of IPC Systems

```
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1
int main(void)
{
  char write_msg[BUFFER_SIZE] = "Greetings";
  char read_msg[BUFFER_SIZE];
  int fd[2];
  pid_t pid;
  if (pipe(fd) == -1) {  // create the pipe
    fprintf(stderr,"Pipe failed");
    return 1;
  }
  pid = fork();  // fork a child process
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
```

```
  if (pid > 0) { /* parent process */
    close(fd[READ_END]); // close the unused end of the pipe
    /* write to the pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
    /* close the write end of the pipe */
    close(fd[WRITE_END]);
  }
  else { /* child process */
    /* close the unused end of the pipe */
    close(fd[WRITE_END]);
    /* read from the pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("read %s",read_msg);
    /* close the read end of the pipe */
    close(fd[READ_END]);
  }
  return 0;
}
```

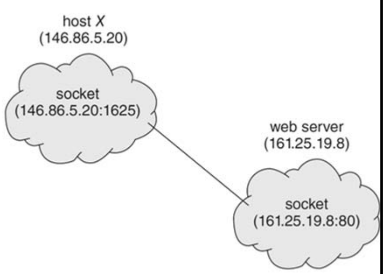
3.7 Examples of IPC Systems

- ►Named Pipes
  - Bidirectional (half-duplex), and no parent-child relationship is required
  - several processes can use it for communication
  - continue to exist after communicating processes have finished
  - referred to as FIFOs in UNIX systems
  - UNIX/Linus
    - `mkfifo()` system call
    - manipulated with the ordinary `open()`, `read()`, `write()`, and `close()` system calls
    - byte-oriented data may be transmitted

PIPES IN PRACTICE
ls | less
: results of list directory → input of less (listing a screen at a time)

3.8 Communication in Client–Server Systems

- Shared memory and message passing +
- Sockets
  - ►Socket:
    - endpoint for communication
    - identified by an IP address concatenated with a port number : 146.86.5.20:1625
    - client–server architecture
  - ►A pair of processes communicating over a network employs a pair of sockets
    - Server waits for incoming client requests by listening to a specified port.
    - Servers implementing specific services listen to well-known ports (<1024)
      - SSH:22; FTP: 21; HTTP: 80
    - client process is assigned a port by its host: arbitrary number > 1024
    - All connections must be unique: different port



3.8 Communication in Client–Server Systems

- Remote Procedure Calls (RPC)
  - ▶One of the most common forms of remote service
  - ▶well structured messages: no longer just packets of data
  - ▶contains an identifier specifying the function and parameters: execution and output sent back
  - ▶One IP address with multiple ports : multiple services
  - ▶Useful in implementing a distributed file system

Summary

- Process: a program in execution, the status of the current activity
- Layout of a process in memory: (1) text, (2) data, (3) heap, and (4) stack
- Process states: (1) ready, (2) running, (3) waiting, and (4) terminated
- PCB: kernel data structure
- Process scheduler: to select a process to run on a CPU.
- Context switch: switches from running one process to running another
- The fork(): to create processes
- IPC: shared memory/exchanging messages/pipe
- Client–server communication
  - ▶Sockets: data packet
  - ▶RPC: abstracting the concept of function (procedure) calls on a separate computer

Exercises, problems and projects

- Exercises
  - ▶3.1, 3.2, 3.5,
  - ▶3.10, 3.11, 3.12, 3.13, 3.16
- Problems
  - ▶3.4
- Projects

3.10 Explain the role of the init (or systemd) process on UNIX and Linux systems in regard to process termination.

3.11 Including the initial parent process, how many processes are created by the program shown below.

```
#include <stdio.h>
#include <unistd.h>
int main()
{
  int i;
  for (i = 0; i < 4; i++)
  fork();
  return 0;
}
```

3.12 Explain the circumstances under which the line of code marked printf("LINE J") will be reached.

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


Exercises, problems and projects

3.13 Using the program in Figure 3.34, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
 pid_t pid, pid1; /* fork a child process */
 pid = fork();
 if (pid < 0) { /* error occurred */
  fprintf(stderr, "Fork Failed");
  return 1;
  }
 else if (pid == 0) { /* child process */
  pid1 = getpid();
  printf("child: pid = %d",pid); /* A */
  printf("child: pid1 = %d",pid1); /* B */
 }
 else { /* parent process */
 pid1 = getpid();
 printf("parent: pid = %d",pid); /* C */
 printf("parent: pid1 = %d",pid1); /* D */
 wait(NULL);
 }
 return 0;
}
```

3.16 Using the program shown in Figure 3.35, explain what the output will be at lines X and Y.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#define SIZE 5
int nums[SIZE] = {0,1,2,3,4};
int main()
{
 int i;
 pid_t pid;
 pid = fork();
 if (pid == 0) {
 for (i = 0; i < SIZE; i++) {
  nums[i] *= -i;
  printf("CHILD: %d ",nums[i]); /* LINE X */
  }
 }
 else if (pid > 0) {
 wait(NULL);
 for (i = 0; i < SIZE; i++)
  printf("PARENT: %d ",nums[i]); /* LINE Y */
 }
return 0;
}
```