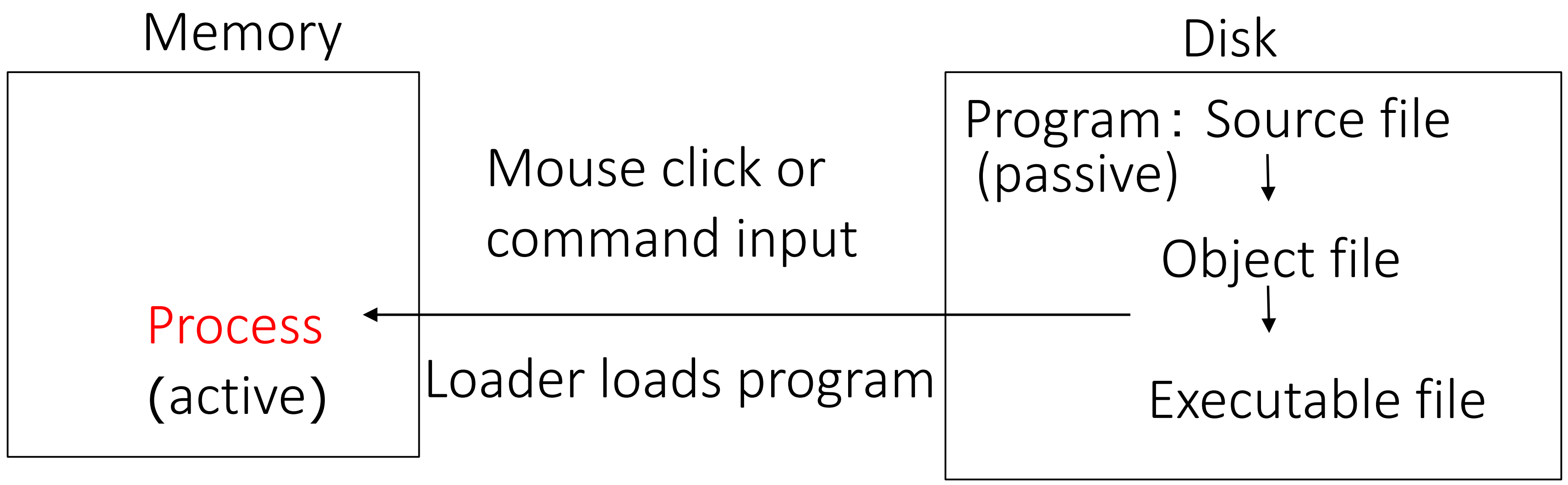
Chapter 3: Processes

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Inter-process Communication (IPC)
- Communication in Client-Server Systems

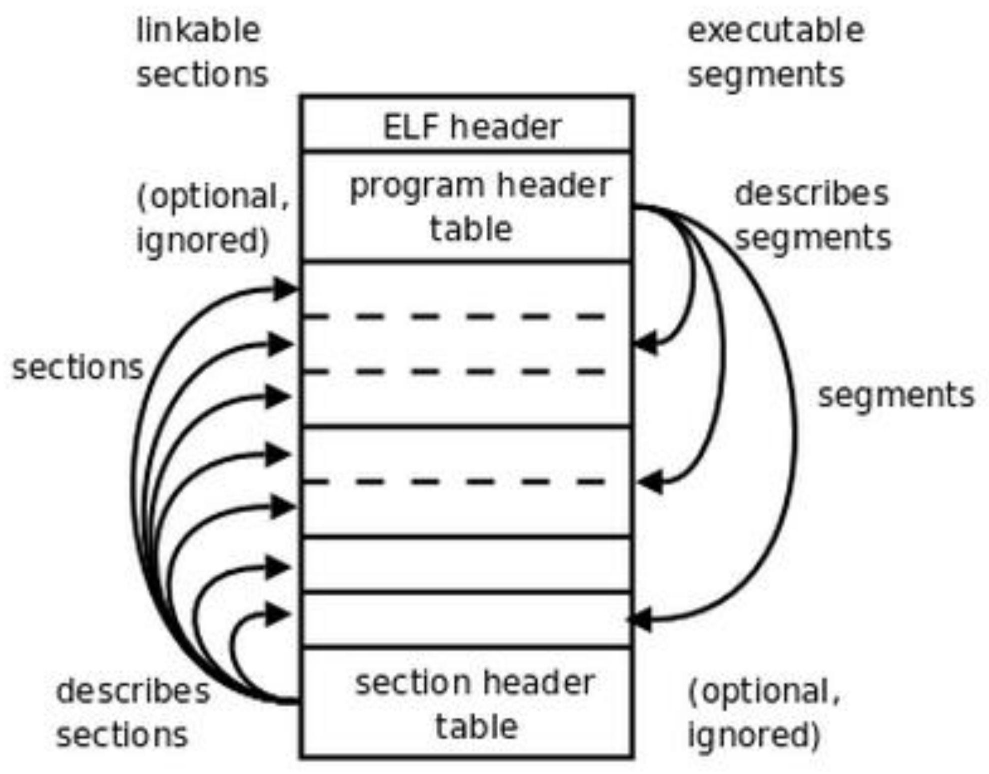
Process Concept

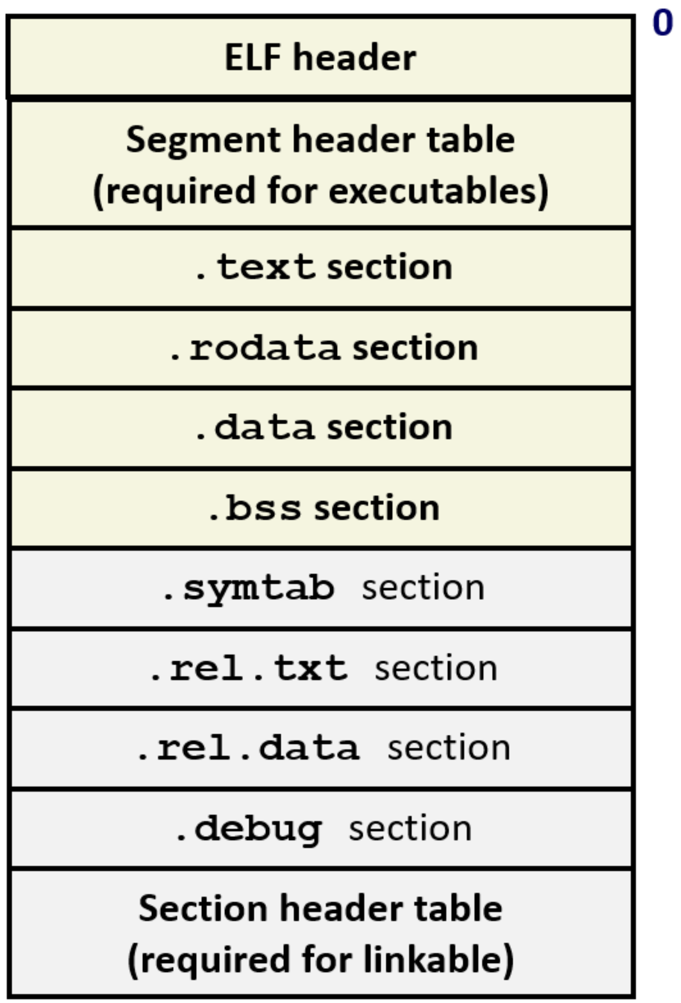
- Process
  - a program in execution in memory
  - execution must progress in sequential fashion
- There can be several processes for one program
  - Consider multiple users executing the same program



ELF Object File Format

ELF (Executable and Linkable Format , Generic name: ELF binaries)





Process Concept

- A process includes multiple parts
  - the program code (also called text section)
  - stack containing temporary data
    - E.g., function parameters, return addresses, local variables
  - data section containing global variables and static variables
  - heap containing memory dynamically allocated during run time
  - program counter, processor registers

(include all current data of the program inside the CPU)

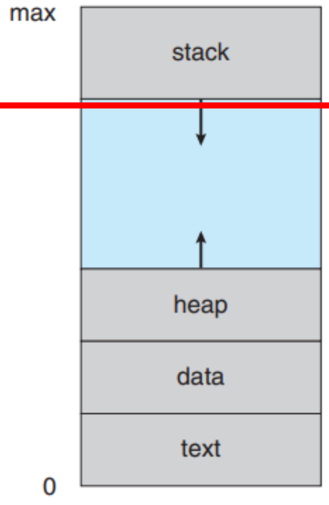
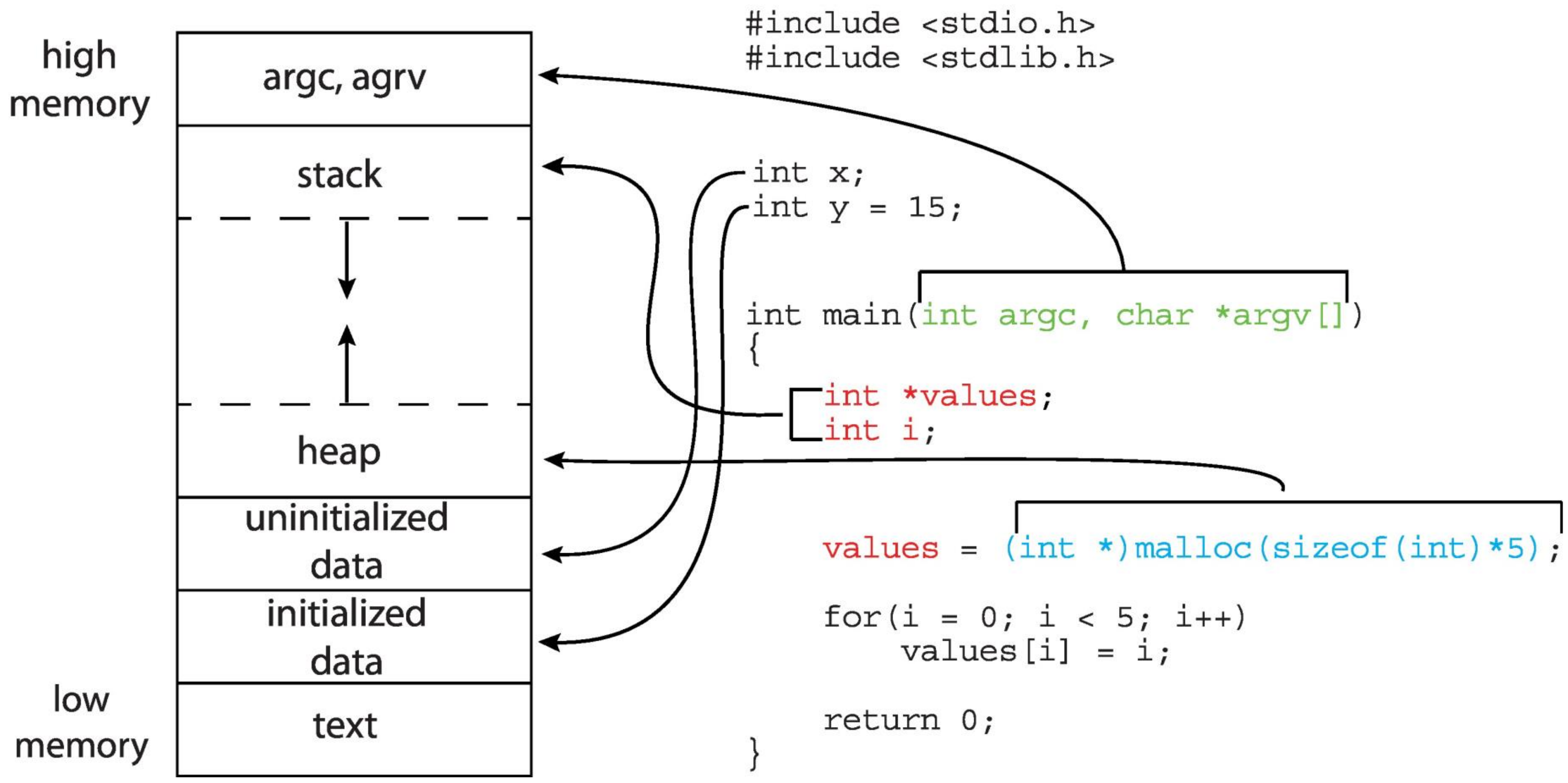


Figure 10.2 Virtual address space of a process in memory.

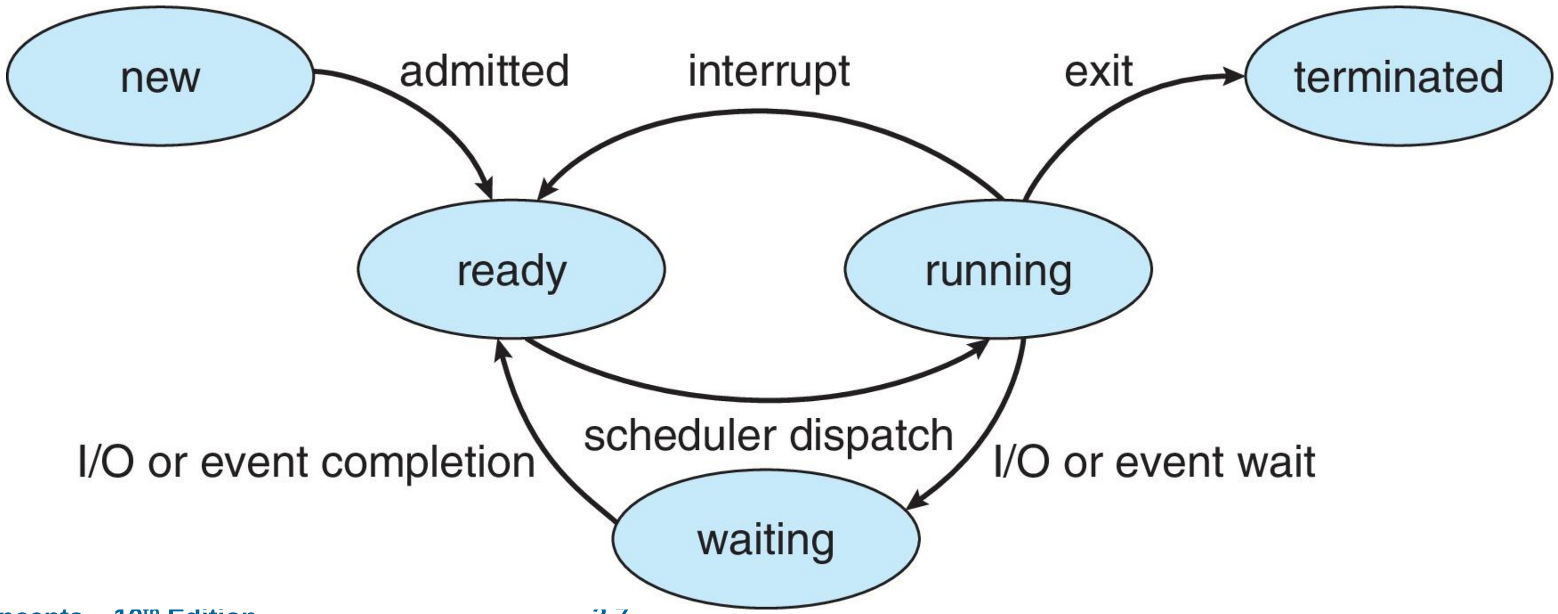
Memory (Logical) Layout of A C Program



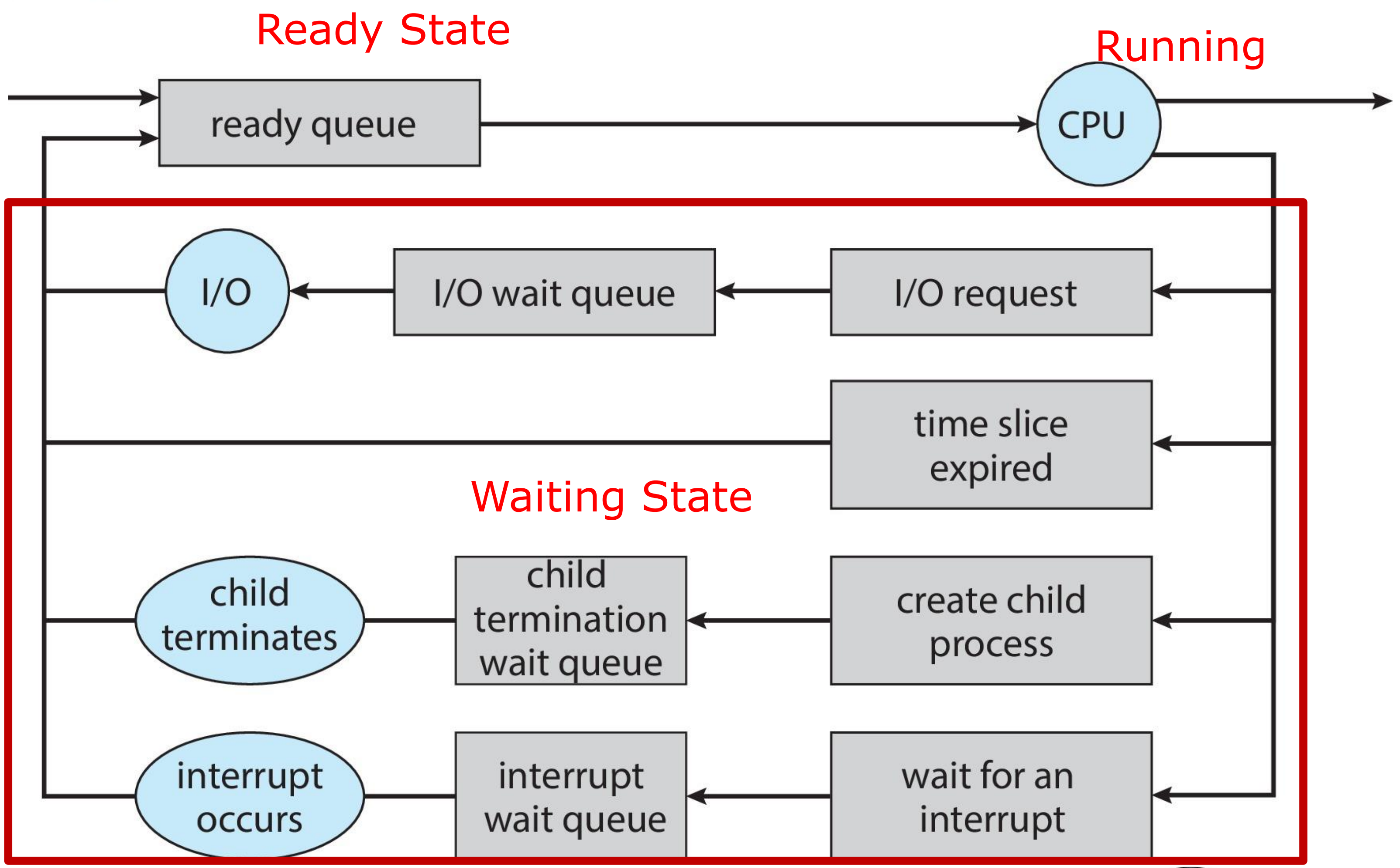
```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

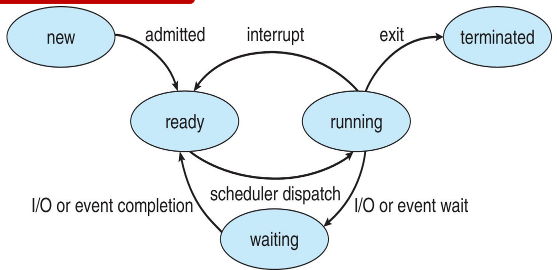
Process State

- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed by CPU
  - **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



Representation of Process Scheduling





Process Control Block (PCB)

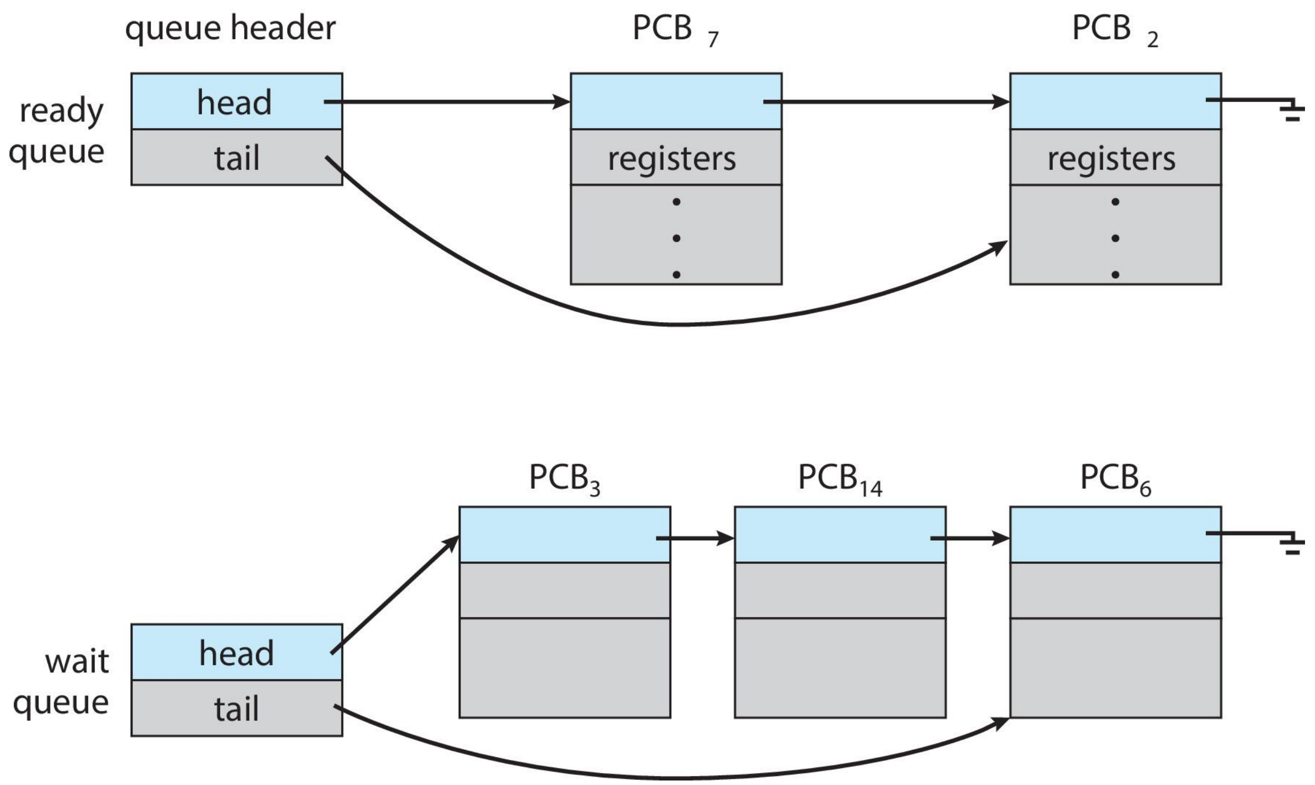
- PCB: Information associated with each process, can be stored in a struct type (also called task control block)
  - **Process state** – running, waiting, etc.
  - **Program counter** – location of instruction to next execute
  - **CPU registers** – contents of all process-centric registers
  - CPU **scheduling information-** priorities, scheduling queue pointers
  - **Memory-management information** – memory allocated to the process
  - **Accounting information** – CPU used, clock time elapsed since start, time limits
  - **I/O status information** – I/O devices allocated to process, list of open files

Process Control Block (PCB)

- A PCB is a kernel data structure,
  - stored in the memory reserved for the kernel;
  - invisible to the process itself;
  - changed only by the kernel.
- Each process has a corresponding unique PCB in the kernel.
  - When a new process is created, the kernel creates a new PCB for it.
  - When a process dies, the kernel deletes the process's PCB.

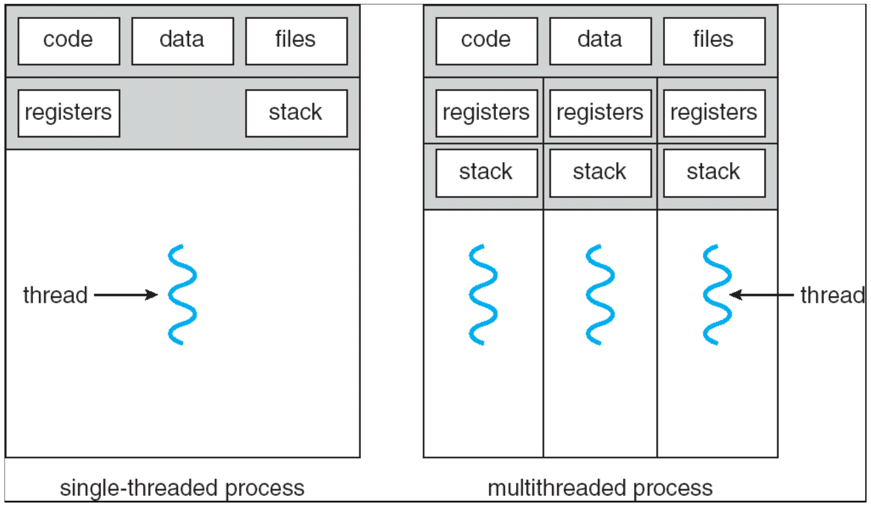
Process Control Block (PCB)

- All the PCBs together is how the kernel keeps track of which processes exist in memory, where they are in memory, what they are currently doing (executing, waiting, ...), etc.



Threads

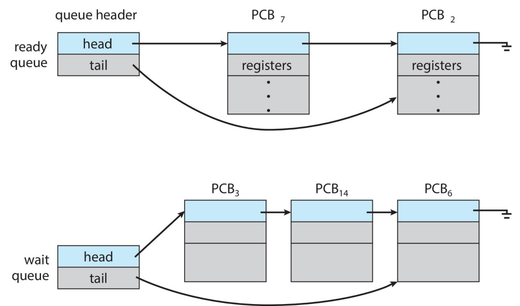
- If process has a single thread [1] of execution
  - One program counter
- If a process has multiple threads of execution
  - Kernel will keep the control information for each thread
  - Multiple program counters
- Explored in detail in Chapter 4



[1] *A thread is a single sequence stream within a process*. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*.

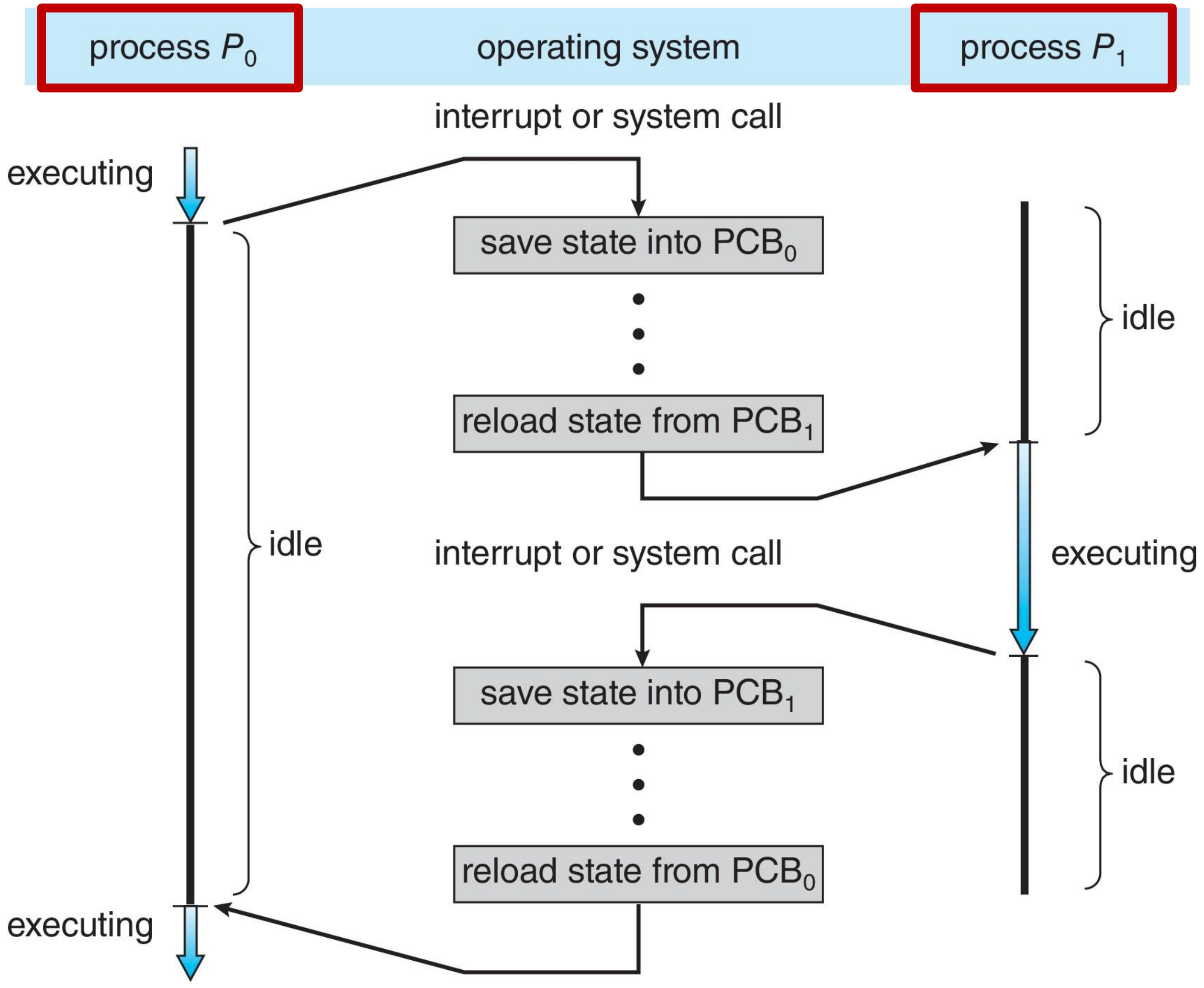
Process Scheduling

- Several processes want to use one CPU (or CPU core)
- Process scheduler（调度）(algorithm inside the kernel, software) selects among available processes(i.e. in ready state) for next execution on CPU core
  - Maintains scheduling queues of processes
    - Ready queue – set of all processes residing in main memory, ready and waiting to execute
    - Wait queues – set of processes waiting for an event (i.e. I/O)
- Processes migrate among the various queues
- Scheduling purpose: maximize CPU use, quickly switch processes



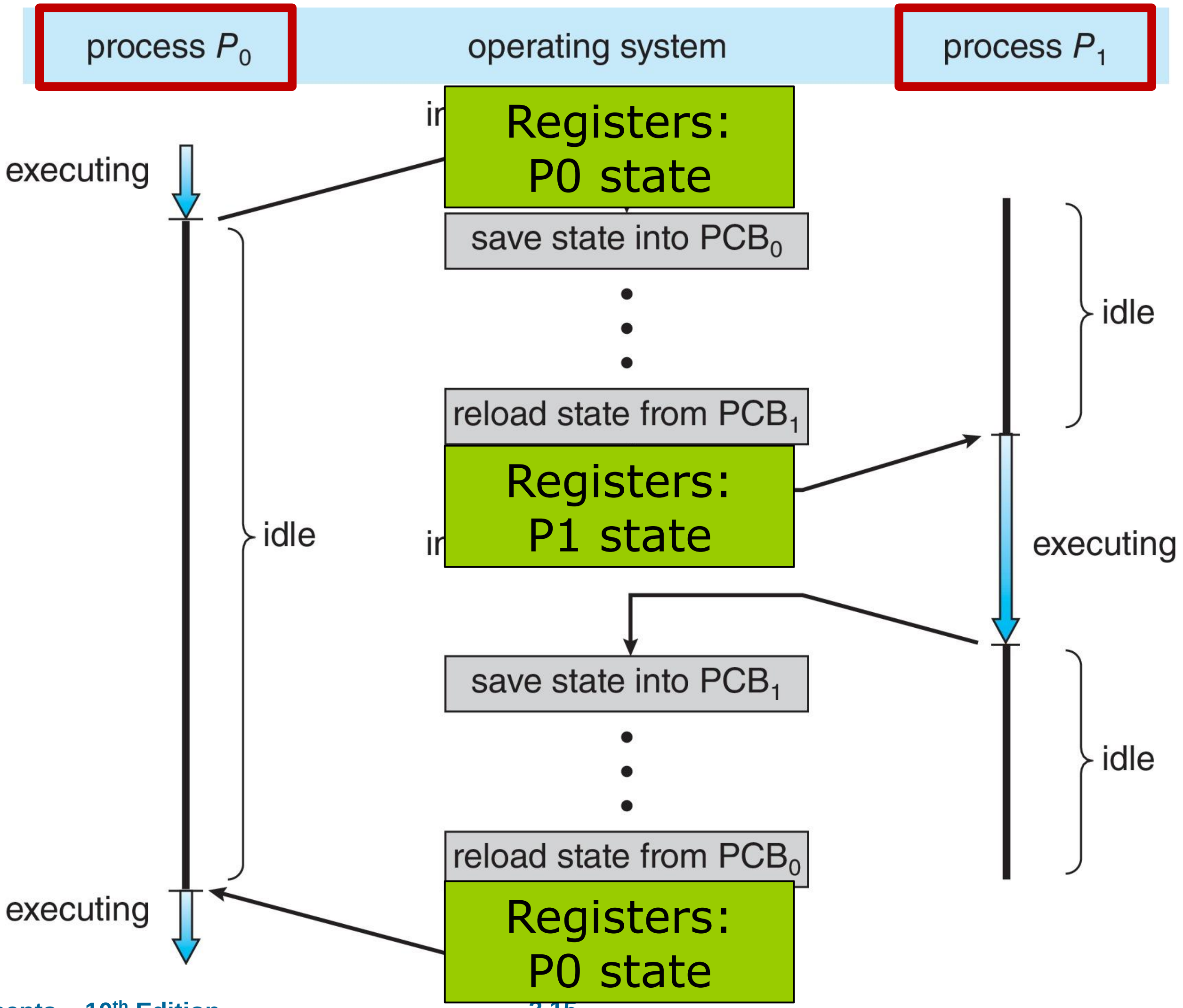
CPU Switch From Process to Process

A context switch occurs when the CPU switches from one process to another.



CPU Switch From Process to Process

A context switch occurs when the CPU switches from one process to another.



Context Switch

- When CPU switches to another process, the system must
  - save the state of the old process (the one is running on CPU) and
  - load the saved state (CPU registers, program counter in PCB) for the new process (the one will run on CPU) via a context switch (i.e., switch PCB)
- Context-switch time is
  - overhead, the system does no useful work while switching
  - dependent on the complexity of OS. The more complex the OS and the PCB,   the longer time the context switch
  - also dependent on hardware support
    - some hardware provides multiple sets of registers per CPU
      - multiple contexts loaded at once, i,e, no need to switch context when load another process to run

Examples: Multitasking in Mobile Systems

- iOS
  - At the beginning, supported only single task, but now multi-tasking
  - Due to screen (UI) real estate limitation, iOS provides
    - Single foreground process- controlled via user interface (process with active window focus / input)
    - Multiple background processes– in memory, running, but not on the display, and with limits
      - Types limits include single, short task, receiving notification of events, specific long-running tasks like audio playback

Examples: Multitasking in Mobile Systems

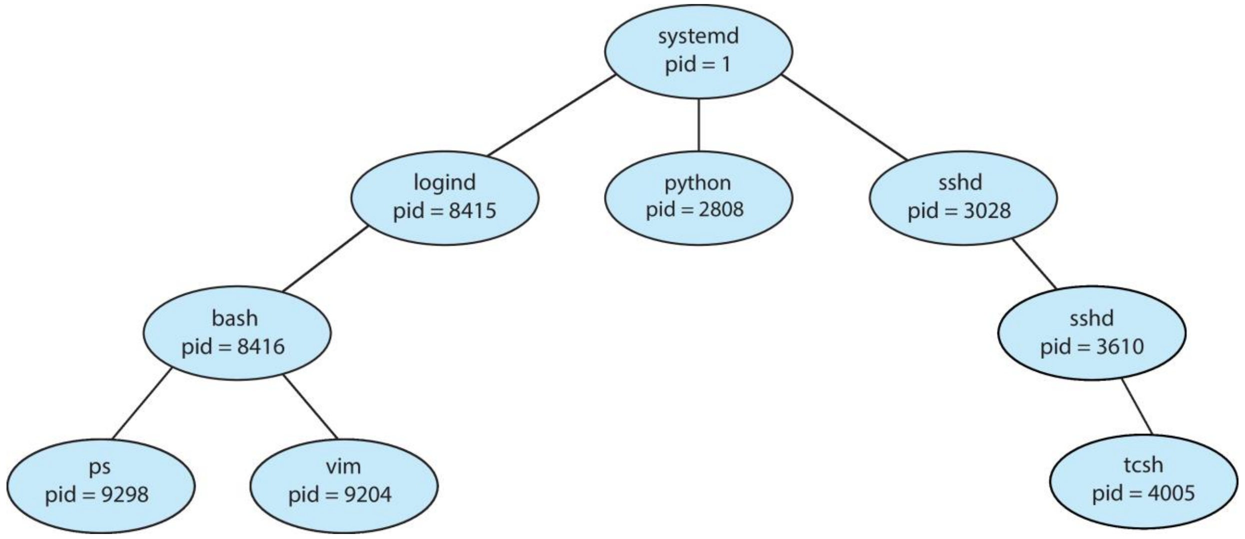
- Android
  - Always supports multi-tasking
  - runs foreground and background, with fewer limits
  - Background process uses a service to
    - perform tasks
    - keep running even if background process is suspended
    - Service
      - is a separate application component that runs on behalf of the background process
      - has no user interface, small memory use
  - No constraints on background applications types

Operations on Processes

- OS must provide **mechanisms** for:
  - process creation
  - process termination

Process Creation

- Parent process
  - create children processes,
  - then, in turn create other processes,
  - finally forming a tree of processes
- Generally, process is identified and managed via a process identifier (pid)



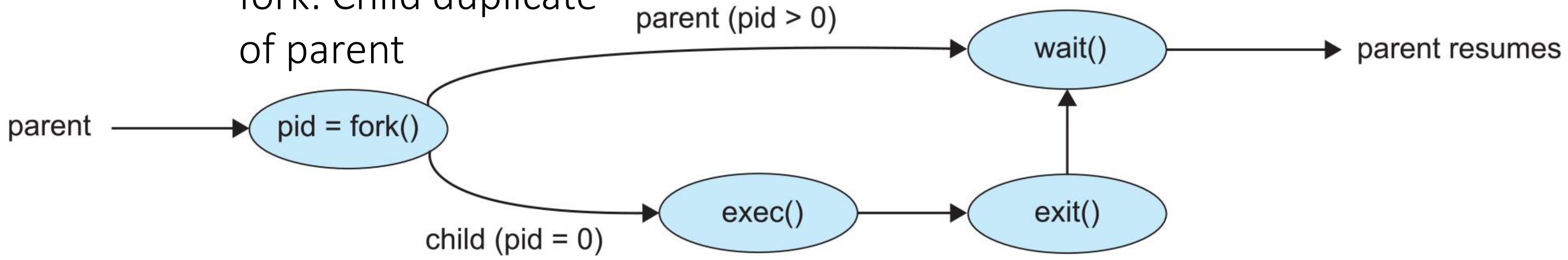
Process Creation

- Parent and children
  - Resource sharing options
    1. Parent and children share all resources
    2. Children share subset of parent’s resources
    3. Parent and child share no resources
  - Execution options
    1. Parent and children execute concurrently
    2. Parent waits until children terminate

Process Creation (Cont.)

- UNIX/Linux examples
  - `fork()` system call creates new process, create a duplicate of the parent
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program
  - Parent process calls `wait()` for the child to terminate

fork: Child duplicate of parent



exec: Child has a program loaded into it

C Program Forking Separate Process

int execlp(const char *file, const char* arg, ..., NULL);
int execvp(const char *file, char* const argv[]);

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
int main(void) {
    pid_t pid;  // PID of child. pid_t is the same as int
    int status; // Exit status of child.
    printf("Parent: calling fork\n");
    pid = fork(); // New child process starts executing code from here.
    if(pid < 0) { // No child process created.
        printf("Parent: fork failed\n");
        return 1; // Parent process dies.
    }
    else if(pid == 0) {
        // Code only executed by the new child process.
        printf("Child: now running the same program as parent, doing exec\n");
        execlp("xcalc", "xcalc", NULL);
        // If exec succeeded, the new child now starts executing the code of
        // the main function of the xcalc program.  The new child normally
        // never executes the code below, unless exec failed.
        printf("Child: exec failed, die\n");
        return 1; // Child process dies after failed exec.
    } else {
        // Code only executed by the parent process.
        printf("Parent: now sleeping and waiting for child %d to end\n", pid);
        wait(&status);
        printf("Parent: finished waiting for child, child is dead\n");
        printf("Parent: child exit status is %d\n", status);
        return 0; // Parent process dies.
    }
}
```

More detailed explanation for this program refers to slides 24-28

Process Creation Procedure

Example:

- Process 1 contains the code of Program 1.
- The code of Program 1 inside Process 1 calls the fork system call:
  - the CPU switches to kernel mode and executes the kernel code for the fork system call;
  - that kernel code creates a new Process 2, which is a new child process of Process 1 (the parent);
  - that kernel code also copies the whole content of Process 1's memory into the memory of Process 2;
    - Process 2 is now an exact copy in memory of Process 1, so Process 2 contains the code of Program 1 too!

Process Creation Procedure(Cont.)

- When the fork system call is over, the CPU returns to user mode and continues executing the code of Program 1 in both Process 1 and Process 2 (doing context switches between the two processes).
  - When the CPU executes Program 1 again inside Process 1, the fork system call returns with the PID of Process 2 (the child of Process 1) as result.
  - When the CPU executes Program 1 inside Process 2, the fork system call returns with the PID 0 as result.
  - The result of the fork system call is the only way that Process 1 and Process 2 have to know whether they are the parent process or the child process after the fork!
- Note that the fork system call is called only once by Process 1, but returns twice, **once in Process 1 and once in Process 2**, because the new Process 2 is an exact copy of Process 1.
  - After the fork system call is over, Process 1 and Process 2 are executing the exact same code of Program 1 at the exact same place in that code.
  - Again: the only difference between the two processes is the value returned as result by the fork system call.

Process Creation Procedure(Cont.)

- After the fork system call returns, the code of Program 1 tests the result returned by fork:
  - if the current process is the parent process (Process 1) then the code of Program 1 calls the wait system call, which moves Process 1 to the wait queue, to wait for the end of the child process;
  - if the current process is the child process (Process 2) then the code of Program 1 calls the exec system call, which completely replaces the code of Program 1 inside Process 2 (and its corresponding data section and heap and stack; everything) with the code of a new Program 2.
    - The CPU then starts executing Program 2 inside Process 2 from the beginning of the main function of the code of Program 2.
- Note that the exec system call is called once by Process 2, but never returns because the code of Program 1 inside Process 2 (the code that does the exec system call) is replaced with the code of Program 2 when the exec system call is done.
  - The only exception to this is if the exec system call fails (usually because Program 2 does not exist). In that case the exec system call returns and Process 2 continues executing Program 1.

Process Creation Procedure(Cont.)

- After some time, Program 2 in Process 2 ends. There are two ways for that to happen.
  1. Program 2 directly calls the exit system call. For example: **`exit(23);`**
  2. The main function of Program 2 returns. For example: **`return 23;`**
     - In that case, the exit system call still happens, but instead of being done by the code of Program 2 itself, it is done by the special loader code inside Process 2 which started the main function!
     - The value 23 returned by the main function as result is then automatically given as argument to the exit system call done by the **special loader code**.
- Either way, directly or indirectly, Process 2 ends up calling the exit system call with the value 23 (as an example) as argument.
  - This value given as argument to the exit system call is called **the "exit status" of Process 2.**

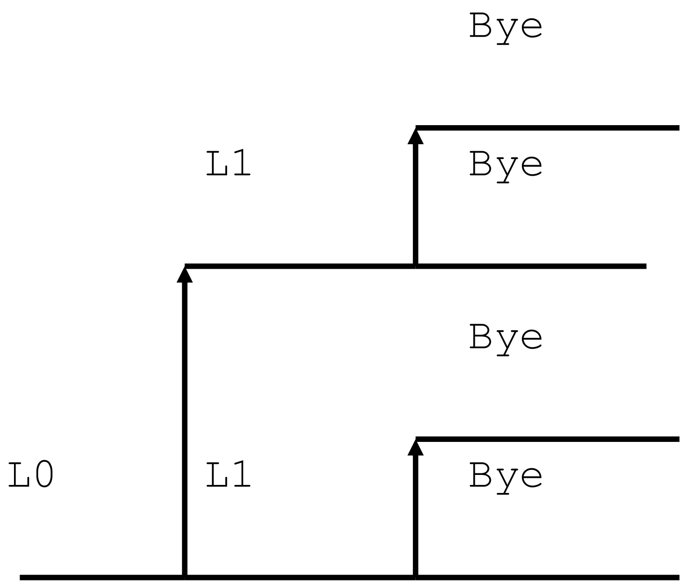
Process Creation Procedure (Cont.)

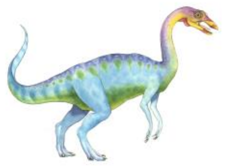
- When Process 2 calls the exit system call:
  1. the kernel destroys Process 2 and deletes its PCB in the kernel's memory;
  2. the kernel wakes up Process 1, which was waiting for its child process to end;
  3. the exit status 23 which was given to the kernel by Process 2 (the child process) as argument to the exit system call is then given by the kernel to Process 1 (the parent process) through the pointer given as argument to the wait system call done by Process 1.
- This allows the parent process to:
  1. **detect when the child process ends**;
  2. **learn why the child process ended**.
     1. An exit status of 0 means the child process terminated normally.
        - This is why in C the main function usually ends with: **`return 0;`**
     2. An exit status different from 0 means the child terminated with an error and the exit code received by the parent process from the child process then gives some information to the parent about why the child died.

Fork Example 1

- Both parent and child can continue forking

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main() {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
    return 0;
}
```

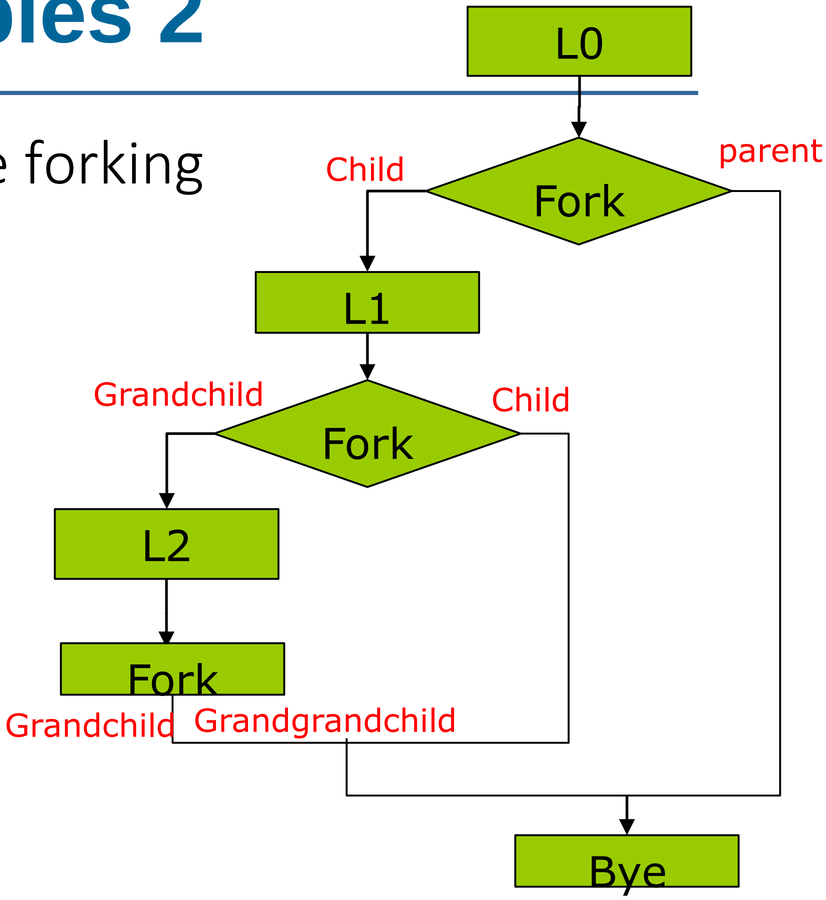


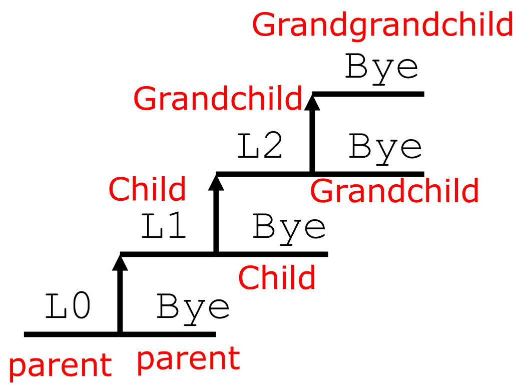


Fork Examples 2

- Both parent and child can continue forking

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main (){
    printf("L0\n");
    if (fork() == 0) {
     printf("L1\n");
     if (fork() == 0) {
         printf("L2\n");
         fork();
      }
    }
    printf("Bye\n");
    return 0;
}
```





Fork Bomb

- Unix:

```
while(1)
   fork();
```

  - 1 parent -> 2 Children -> 4 grandchildren -> 8 ……
  - Result:
    - an exponential number of processes are created,
    - CPU: busy with creation of process, context switches
    - Memory is full
    - Consequence: computer slows down , un-usable
  - Therefore modern Unix systems limit the number of processes that a user is allowed to create.

Process Termination

- Child process asks OS to terminate itself
  - using the `exit()` system call.
    - Returns status data from child to parent (via `wait()`)
    - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes (use `abort()` system call) for some reasons:
  1. Child has exceeded allocated resources
  2. Task assigned to child is no longer required
  3. Parent terminated (some OSes do not allow children to be alive)

Process Termination

- Cascading (倾泻式的) termination
  - Some operating systems do not allow child to exist if its parent has terminated.
  - When parent is terminated, all children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- `wait()` system call
  - The parent process may wait for termination of a child process
  - The call returns status information and the *pid* of the terminated process

Zombie and Orphan

- A zombie (僵尸) process is
  - living corpse, half alive and half dead
  - terminated, but still consumes system resources
    - still has an entry in the process table
      - where the entry is still needed to allow the parent process to read its child's exit status.
      - once the exit status is read by parent via the wait system call, the zombie's entry is removed from the process table ("reaped“).
- An orphan（孤儿）process is
  - child process that is still running
  - but parent process has finished or terminated.

Zombie

- Reaping (回收)
  - Performed by parent on terminated child
  - Parent is given exit status information (by OS)
  - Kernel discards process
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then child will be reaped by init or system process
    - So, only explicitly reaping is needed when parent is a long-running processes. e.g., shells and servers

Zombie and Orphan Examples

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(){
    pid_t pid = fork();
    if (pid == 0) {
      /* Child */
      printf("Terminating Child, PID = %d\n", getpid());
      exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
          ; /* Infinite loop */
    }
    return 0;
}
```

Zoombie or Orphan?

Zombie and Orphan Examples

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(){
 pid_t pid = fork();
 if (pid == 0) {
     /* Child */
    printf("Running Child, PID = %d\n", getpid());
     while (1)
       ; /* Infinite loop */
 } else {
     printf("Terminating Parent, PID = %d\n", getpid());
     exit(0);
 }
 return 0;
}
```

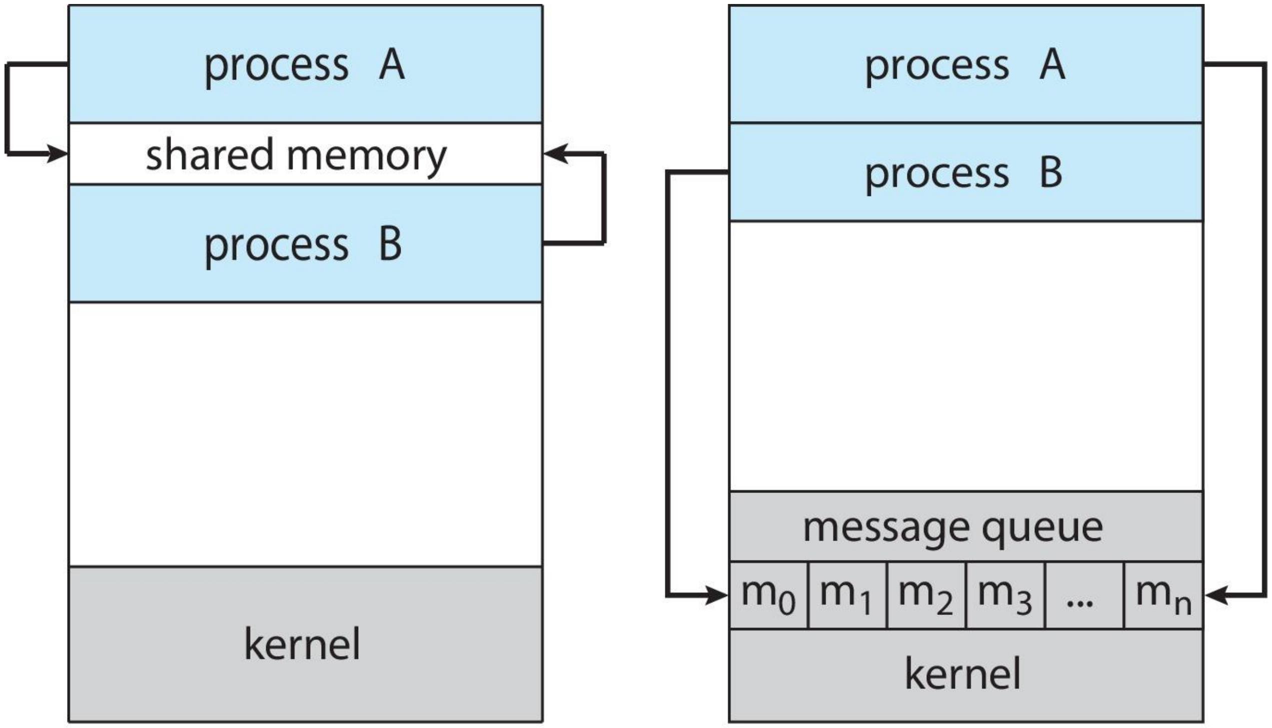
Zoombie or Orphan?

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
  - Independent process cannot affect or be affected by another process
  - Cooperating process can affect or be affected by other processes, because they share data
- Advantages/Reasons of process cooperation
  1. Information sharing
  2. Computation speed-up
  3. Modularity
  4. Convenience
- Disadvantages
  1. Added complexity
  2. Deadlocks (死锁) possible
  3. Starvation (饥饿) possible

Communications Models

- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - Shared memory //user processes control
  - Message passing //kernel control



(a) Shared memory (b) Message passing

Interprocess Communication – Shared Memory

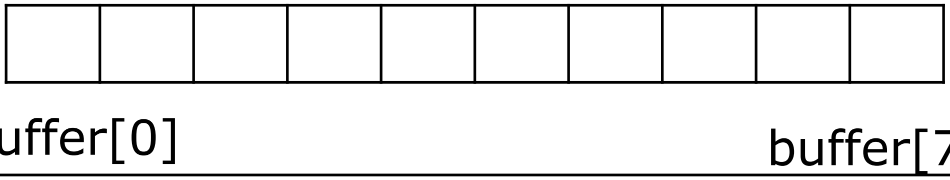
- Processes communicate through a shared memory
- User processes control
- Major issues
  - Synchronization (同步)
  - Discussed in Chapters 6 & 7.

Producer-Consumer Problem

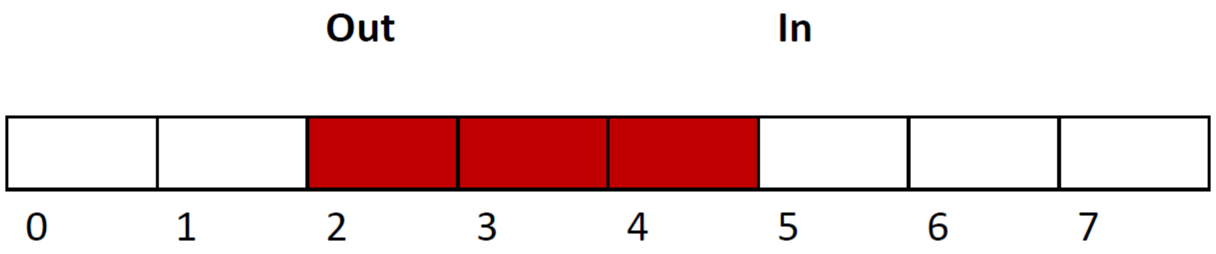
- Producer-Consumer Problem: Paradigm (范例) for cooperating processes
  - Producer process produces information that is consumed by a consumer process
    - unbounded-buffer: no practical limit on the size of the buffer
      - Producer never has to wait because there is always extra space available for new information;
      - Only consumer might have to wait if no information is available to read
    - bounded-buffer: buffer size is fixed
      - Producer might have to wait if there is no space available to store new information;
      - Consumer might have to wait if no information is available to read

Bounded-Buffer – Shared-Memory Solution

- Shared data

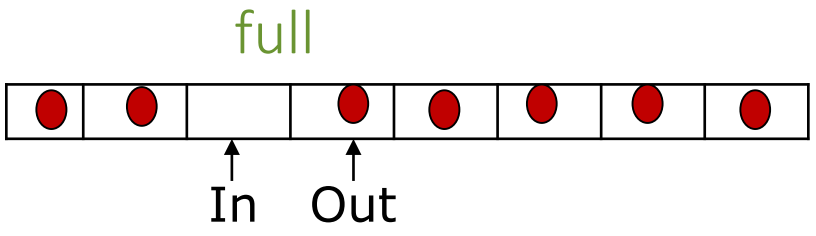


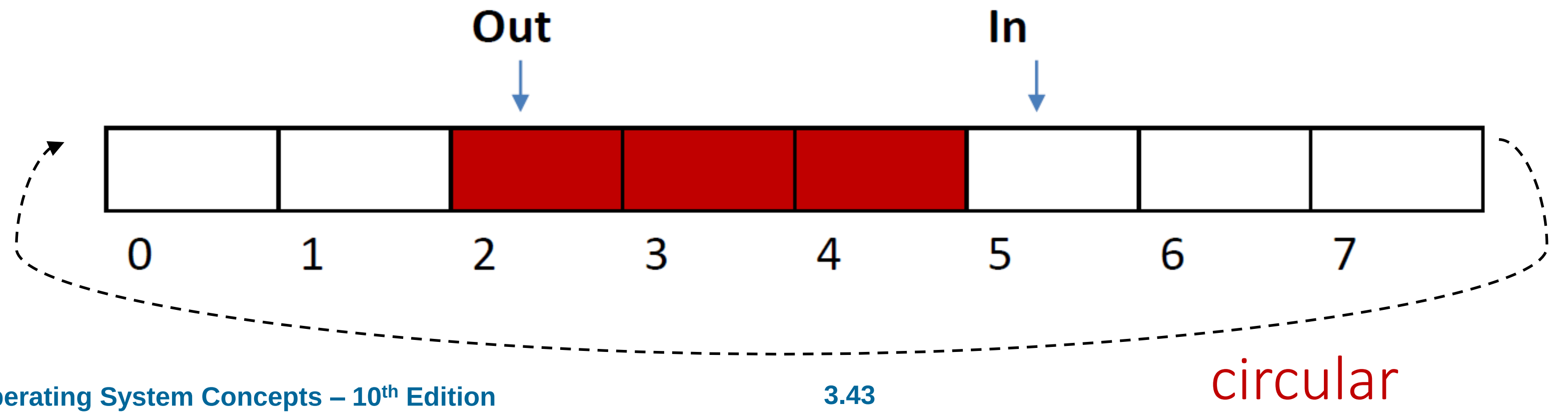
```
#define BUFFER_SIZE 8
typedef struct {
        . . .
} item; //Global variable

item buffer[BUFFER_SIZE];
int in = 0; //place that an element can be put in
int out = 0; //place that an element can be consumed
```

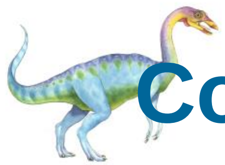


Producer Process – Shared Memory

```
item next_produced;

while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out) //full
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
```

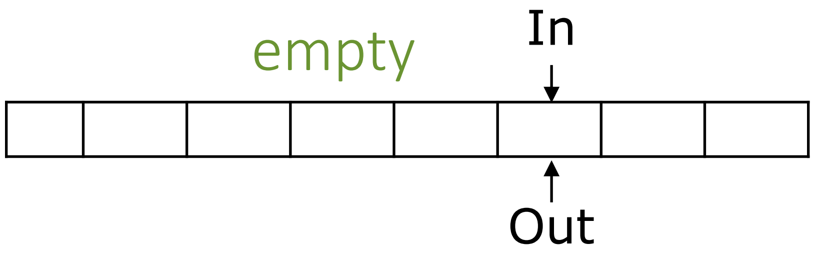


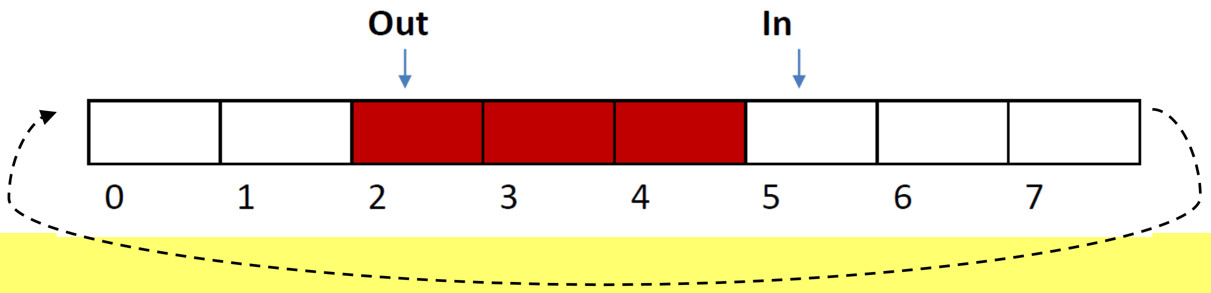




Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
        while (in == out) //empty
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```





Think after class:

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

Interprocess Communication – Message Passing

- Message passing
  - Kernel control
  - two operations (*message* size is either fixed or variable**可变的**):
    - send(*message*)
    - receive(*message*)
- A communication link between processes must be created before communication. Implementation issues to be considered:
  - 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?

Message Passing (Cont.)

- Implementation of communication link
  - Physical level:
    - Shared memory
    - Hardware bus
    - Network
  - Logical level:
    - Direct (process to process) or indirect (mail box)
    - Synchronous (blocking) or asynchronous (non-blocking)
    - Automatic or explicit buffering

Message Passing : Direct Communication

- Processes must name each other explicitly.
- Send() and receive() primitives (原语) are defined as
  - `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

Message Passing：Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports(端口))
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- The send() and receive() primitives are defined as
  - send(*A, message*) – send a message to mailbox A
  - receive(*A, message*) – receive a message from mailbox A

Message Passing: Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy 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

Message Passing：Synchronization(同步)

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send -- the sender is blocked until the message is received by the receiving process or by the mailbox
  - Blocking receive -- the receiver is blocked until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send -- the sender sends the message and continues without waiting for the message to be received
  - Non-blocking receive -- the receiver receives:
    - A valid message, or
    - Null message

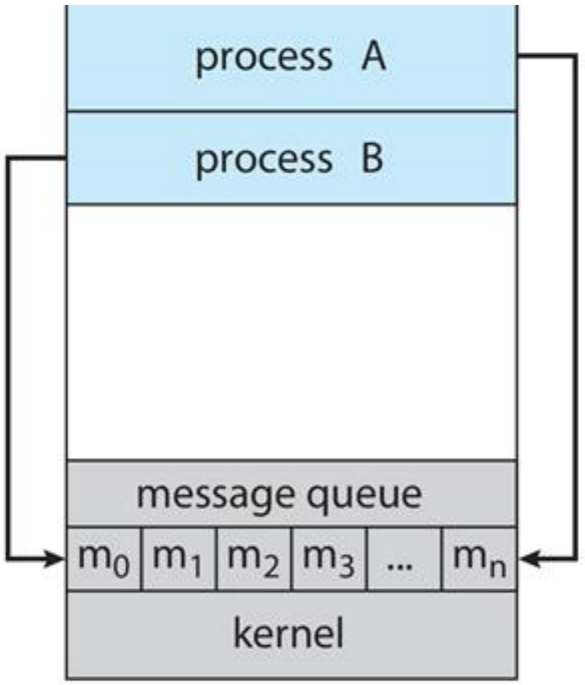
Message Passing：Synchronization(同步)

- Different combinations possible
  - If both send and receive are blocking, this case is called rendezvous（会合）

Message Passing: Buffering

- Queue of messages attached to the link, in **kernel memory**
- Implemented in one of three ways

1. Zero capacity – no messages are queued on a link.
   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

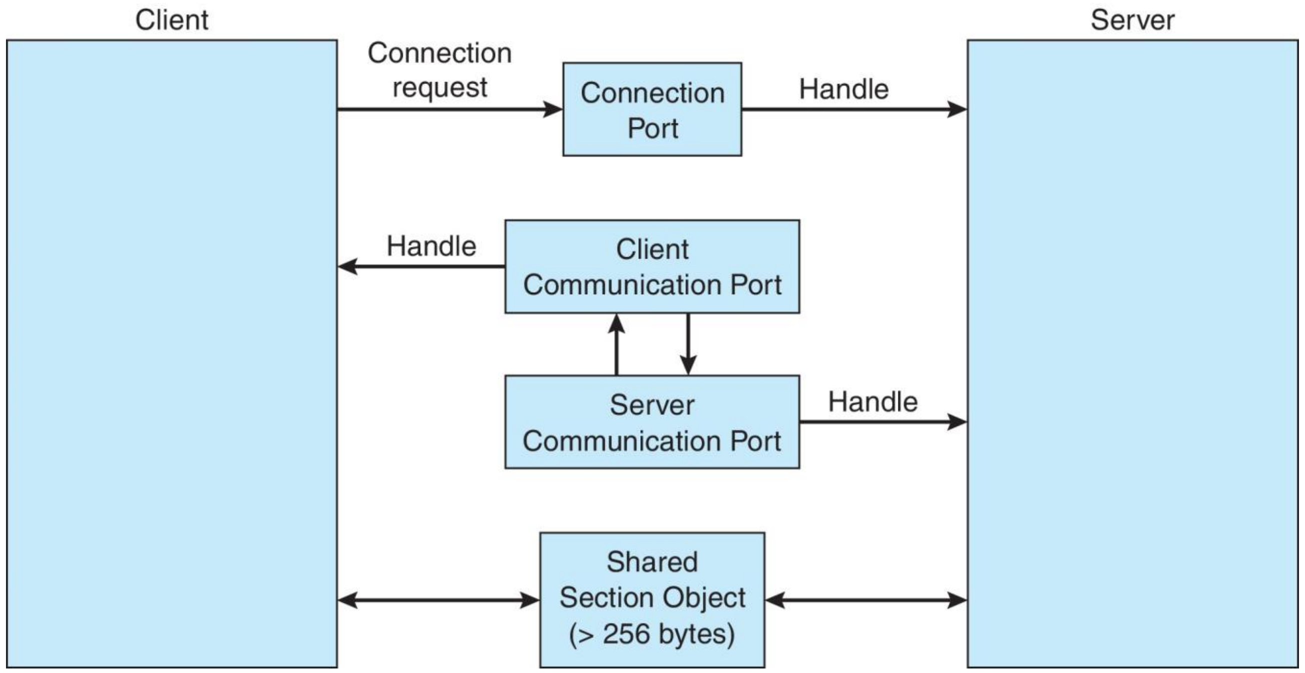


(b)

Examples of IPC Systems – Windows

- Message-passing centric via advanced local procedure call (ALPC) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as following steps:
    1. The client opens a handle (句柄,标示资源) to the subsystem's connection port object.
    2. The client sends a connection request.
    3. The server creates two private communication ports and returns the handle to one of them to the client.
    4. The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

ALPC in Windows


Client
Server
Connection request
Connection Port
Handle
Handle
Client Communication Port
Server Communication Port
Handle
Shared Section Object (> 256 bytes)

1. For small messages (up to 256 bytes), the port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
2. Larger messages must be passed through a section object, which is a region of shared memory associated with the channel.
3. When the amount of data is too large to fit into a section object, an API is available that allows server processes to read and write directly into the address space of a client.

The ALPC facility in Windows is not part of the Windows API and hence is not visible to the application programmer.

Pipes

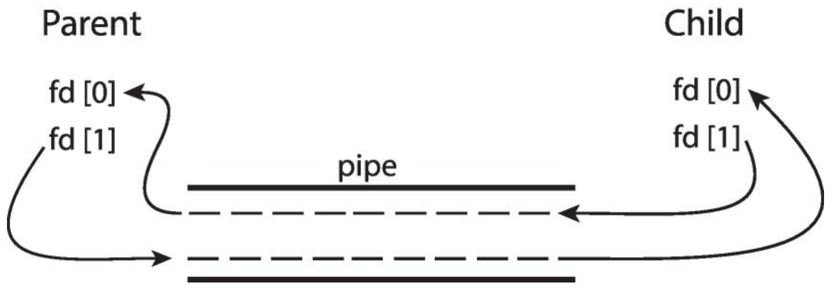
- Acts as a conduit (管道) allowing two processes to communicate on the same computer
- Issues to consider in create of such communication:
  1. Is communication unidirectional or bidirectional?
  2. In two-way communication, is it half or full-duplex(data can travel in both directions at the same time)?
  3. Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  4. Can the pipes be used over a network?

Pipes

- Ordinary (anonymous（匿名）) pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

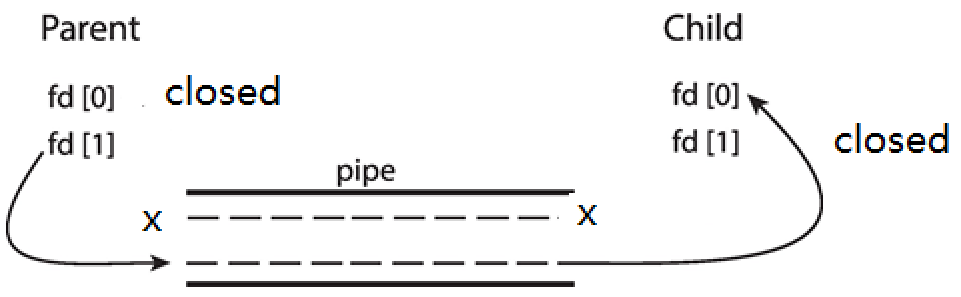
Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore unidirectional (单向)
  - create two separate pipes if bidirectional communication is necessary
- Require parent-child relationship between communicating processes



Ordinary (Anonymous) Pipes

Ordinary pipe in UNIX pipe.c



```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define BUFFER_SIZE 80
#define READ_END 0
#define WRITE_END 1

int main(void)
{
        char write_msg[BUFFER_SIZE] = "Greetings
        from parent process";
        char read_msg[BUFFER_SIZE];
        int fd[2];
        pid_t pid;
        int status;
        /* create the pipe */
        if (pipe(fd) == -1) {
                fprintf(stderr,"Pipe failed");
                return 1;
        }
        /* fork a child process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        if (pid > 0) { /* parent process */
                /* close the unused end of the pipe */
                close(fd[READ_END]); //optional ? important
                /* 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]);
                pid = wait(&status);
                printf("Parent: child(%d) exit(%d)\n",pid,
                                status);
        }
        else { /* child process */
                /* close the unused end of the pipe */
                close(fd[WRITE_END]); //optional ? important
                /* read from the pipe */
                read(fd[READ_END], read_msg, BUFFER_SIZE);
                printf("Child Read: %s",read_msg);
                /* close the read end of the pipe */
                close(fd[READ_END]);
        }
    return 0;
}
```

```
gcc –o pipe pipe.c
```

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several (>=2) processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

Named Pipes Example1

writer.c

```
gcc –o writer writer.c
./writer &
```

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    int fd;
    char *myfifo = "myfifo1";
    char msg[]= "Hi,this is named pipe";

    /* create the FIFO (named pipe) */
    mkfifo(myfifo, 0666);

    /* write msg to the FIFO */
    fd = open(myfifo, O_WRONLY);
    write(fd, msg, sizeof(msg));

    close(fd);

    /* remove the FIFO */
    unlink(myfifo);
    return 0;
}
```

reader.c

```
gcc –o reader reader.c
./reader
```

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>

#define MAX_BUF 1024
int main() {
    int fd;
    char *myfifo = "myfifo1";
    char buf[MAX_BUF];

    // open, read, & display the msg from the FIFO
    fd = open(myfifo, O_RDONLY);
    read(fd, buf, MAX_BUF);
    printf("Received: %s\n", buf);

    close(fd);

    return 0;
}
```

Named Pipes Example2

fifo_server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE       "MYFIFO2"

int main(void) {
        FILE *fp;
        char readbuf[80];

        /* Create the FIFO if it does not exist */

        umask(0);  // set file mode creation mask
        mknod(FIFO_FILE, S_IFIFO | 0666, 0);
                   //create a FIFO-special file
        while (1) {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
                fclose(fp);
                if (strncmp(readbuf, "exit", 4) == 0)
                    break;
        }

        return(0);
}
```

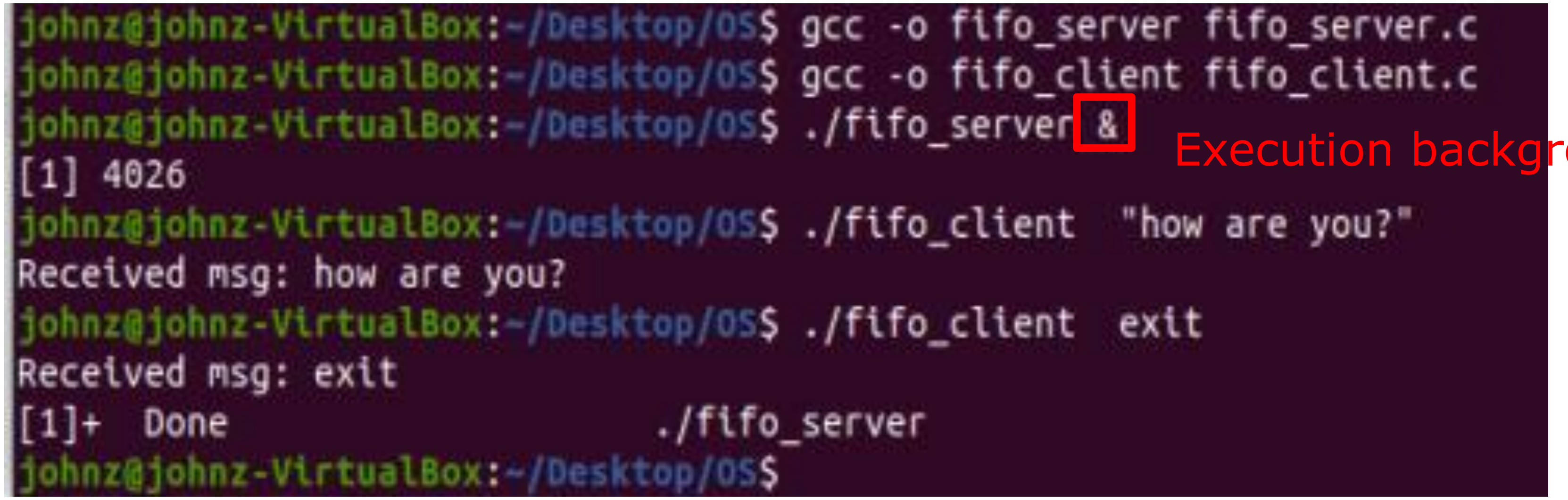
fifo_client.c

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

#define FIFO_FILE       "MYFIFO2"

int main(int argc, char *argv[]) {
        FILE *fp;

        if (argc != 2) {
                printf("USAGE: fifo_client [string]\n");
                exit(1);
        }

        if ((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

        fputs(argv[1], fp);

        fclose(fp);
        return(0);
}
```

int mknod(const char ***_pathname_, mode_t** _mode_**, dev_t** _dev_**);**

The system call **mknod**() creates a filesystem node (file, device special file or named pipe) named _pathname_, with attributes specified by _mode_ and _dev_.

Named Pipes



```
johnz@johnz-VirtualBox:~/Desktop/OS$ gcc -o fifo_server fifo_server.c
johnz@johnz-VirtualBox:~/Desktop/OS$ gcc -o fifo_client fifo_client.c
johnz@johnz-VirtualBox:~/Desktop/OS$ ./fifo_server &
[1] 4026
johnz@johnz-VirtualBox:~/Desktop/OS$ ./fifo_client  "how are you?"
Received msg: how are you?
johnz@johnz-VirtualBox:~/Desktop/OS$ ./fifo_client  exit
Received msg: exit
[1]+  Done                    ./fifo_server
johnz@johnz-VirtualBox:~/Desktop/OS$
```

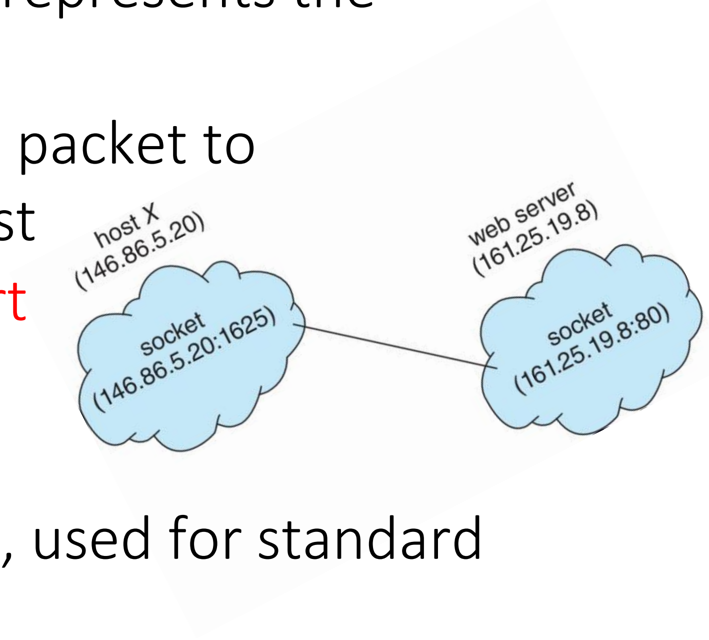
Execution background

Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls (RPC)

Sockets

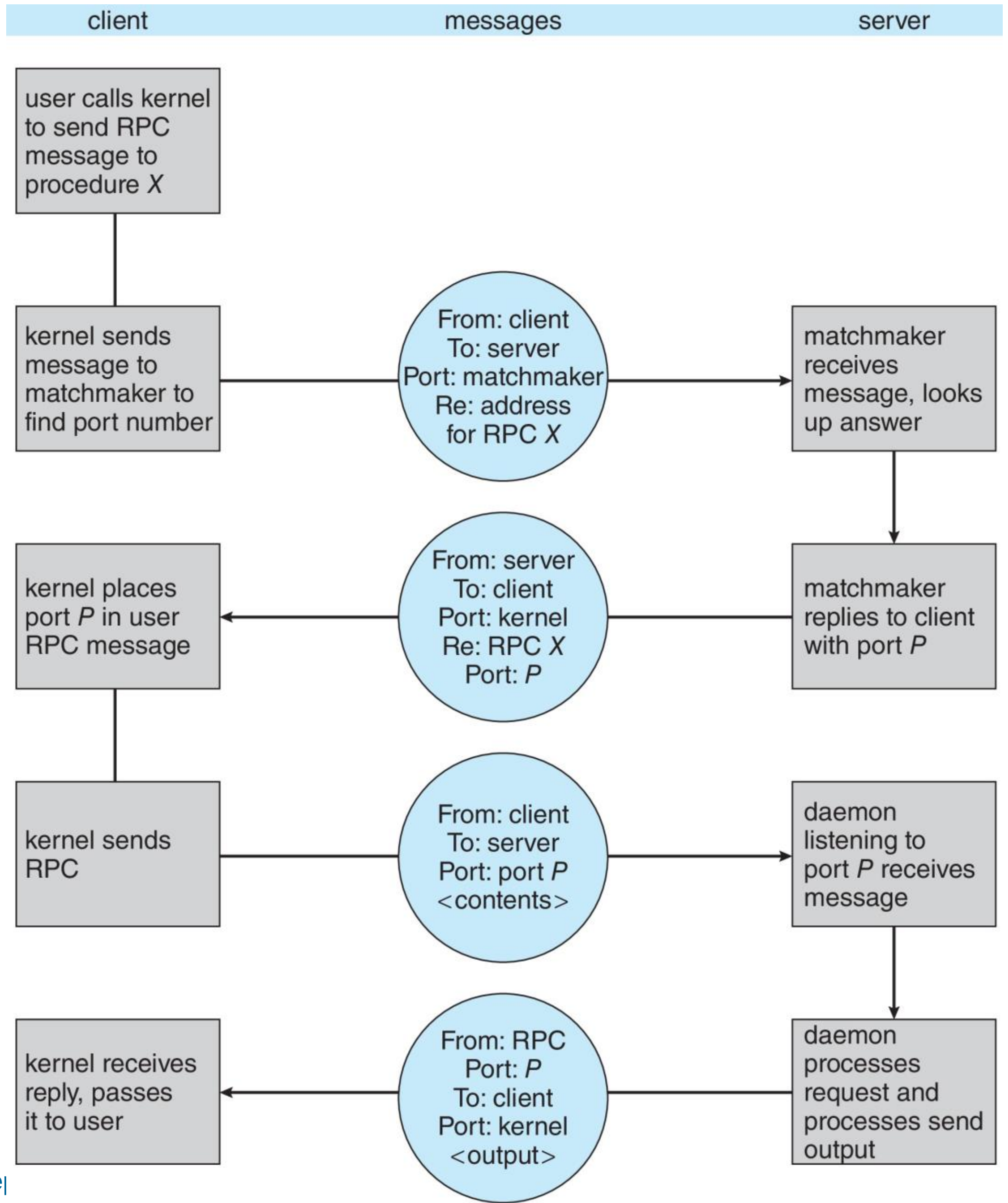
- A socket (套接字)
  - endpoint for communication
  - a data structure inside the kernel that represents the local end of a network connection
  - a number included at start of message packet to differentiate network services on a host
    - Concatenation of IP address and port
      - E.g., 161.25.19.8:1625
        - port 1625 on host 161.25.19.8
    - All ports below 1024 are *well known*, used for standard services
    - Special IP address 127.0.0.1 (loopback) to refer to system on which process is running
- Communication happens between a pair of sockets, one on the local host and one on the remote host



Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Looks like a normal function call but done through the network
  - Uses ports for service differentiation
- OS typically provides a matchmaker service to connect client and server

Execution of RPC


client
messages
server
user calls kernel to send RPC message to procedure *X*
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC *X*
matchmaker receives message, looks up answer
kernel places port *P* in user RPC message
From: server To: client Port: kernel Re: RPC *X* Port: *P*
matchmaker replies to client with port *P*
kernel sends RPC
From: client To: server Port: port *P* <contents>
daemon listening to port *P* receives message
kernel receives reply, passes it to user
From: RPC Port: *P* To: client Port: kernel <output>
daemon processes request and processes send output

Remote Procedure Calls: Steps

- RPC communication steps
  1. The client-side **stub** (server proxy)
     - 1) locates the server, 2) **marshals** the parameters, 3) sends parameters to server-side stub in a network message
  2. The server-side **stub** (client proxy)
     - 1) receives this message, 2)unpacks the marshalled parameters, 3) performs the procedure call on the server, marshals the result of the call, and 4) sends it back to the client-side stub in another message
  3. The client-side **stub**
     - 1) receives this second message, 2) unpacks the marshalled result, and 3) gives it back to the client that did the RPC

Remote Procedure Calls: Stub

- Stubs
  - manages the network connection between client and server
  - extra code on the client side and server side
  - Typically, a separate stub exists for each separate remote procedure
  - On Windows, stub code compile from specification written in Microsoft Interface Definition Language (MIDL)

Remote Procedure Calls: Marshal

- Use External Data Representation (XDR) format to account for different CPU architectures
- Data representation can be different in different CPU
  - E.g., data: 0xAABBEEFF address: 0x1000, 0x1001, 0x1002, 0x1003

Big-endian (**大端法**) e.g., ARM

AA	BB	EE	FF
0x1000	0x1001	0x1002	0x1003

Little-endian (**小端法**, intel), e.g., Intel

AA	BB	EE	FF
0x1003	0x1002	0x1001	0x1000

- On the client side, parameter marshalling involves converting the machine-dependent data into XDR before they are sent to server.
- On the server side, the XDR data are un-marshalled and converted to the machine-dependent representation for the server.

End of Chapter 3

Appendices

- The appendix parts are for students who are interested in knowing more about the programming related to communications introduced in this lecture.

C Program Forking Separate Process: Use CVP

int execlp(const char *file, const char* arg, ..., NULL);
int execvp(const char *file, char* const argv[]);

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
int main(void) {
    pid_t pid;  // PID of child. pid_t is the same as int
    int status; // Exit status of child.
    char *argv[] = {"xcalc", NULL};
    printf("Parent: calling fork\n");
    pid = fork(); // New child process starts executing code from here.
    if(pid < 0) { // No child process created.
        printf("Parent: fork failed\n");
        return 1; // Parent process dies.
    }
    else if(pid == 0) {
        // Code only executed by the new child process.
        printf("Child: now running the same program as parent, doing exec\n");
        execvp(argv[0], argv);
        // If exec succeeded, the new child now starts executing the code of
        // the main function of the xcalc program.  The new child normally
        // never executes the code below, unless exec failed.
        printf("Child: exec failed, die\n");
        return 1; // Child process dies after failed exec.
    } else {
        // Code only executed by the parent process.
        printf("Parent: now sleeping and waiting for child %d to end\n", pid);
        wait(&status);
        printf("Parent: finished waiting for child, child is dead\n");
        printf("Parent: child exit status is %d\n", status);
        return 0; // Parent process dies.
    }
}
```

Creating a Separate Process: Windows API

```
#include <windows.h>
#include <stdio.h>
int main( VOID ) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    DWORD status;
    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    // Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line).
        "C:\\WINDOWS\\system32\\mspaint.exe", // Command line.
        NULL,             // Process handle not inheritable.
        NULL,             // Thread handle not inheritable.
        FALSE,            // Set handle inheritance to FALSE.
        0,                // No creation flags.
        NULL,             // Use parent's environment block.
        NULL,             // Use parent's starting directory.
        &si,              // Pointer to STARTUPINFO structure.
        &pi ) )            // Pointer to PROCESS_INFORMATION structure.
    {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return -1;
    }
    // Wait until child process exits.
    status = WaitForSingleObject( pi.hProcess, INFINITE );
    printf("Child exit status is %d\n", status);
    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
    return 0;
}
```

Examples of IPC Systems - POSIX

- POSIX Shared Memory -- Producer
  - Steps:
    1. Producer process first creates or opens an existing shared memory segment

       `shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`
    2. Set the size of the file object

       `ftruncate(shm_fd, 4096);`
    3. map the shared memory segment in the address space of the process

       ```
       ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
                       shm_fd, 0);
       ```
    4. Then the process could write to the shared memory

       `sprintf(ptr, "%s", "Writing to shared memory");`

mmap: 将一个文件或者其它对象映射进内存, chapter 14 more detail

```
void* mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int ftruncate(int fd, off_t  length);
```

Examples of IPC Systems - POSIX

- POSIX Shared Memory – Consumer
  - Steps:
    1. consumer process opens shared memory object name

       shm_fd = shm_open (name, O_RDONLY, 0666);

       - Returns file descriptor (int) which identifies the file
    2. map the shared memory object in the address space of the process

       ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
    3. Now the process can read from the shared memory object

       printf("%s", (char *)ptr);
    4. remove the shared memory object once finished

       shm_unlink(name);

```
int shm_open( const char *name, int oflag, mode_t mode);
void* mmap(void* start,size_t length,int prot,int flags,int fd,off_t offset);
int shm_unlink( const char *name);
```

IPC POSIX Producer

shm_producer.c
compiling command:
gcc –o producer shm_producer.c -lrt

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
int main() {
       const int SIZE = 4096;          //the size of shared memory object
       const char *name = "OS";        //name of the shared memory object
       const char *message0= "Studying Operating Systems ";//string written to shared memory
       const char *message1= "Is Fun!\n";

       int shm_fd;          //shared memory file descriptor
       void *ptr;           //pointer to shared memory object

       /* create the shared memory segment/object */
       shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

       ftruncate(shm_fd, SIZE); /* configure the size of the shared memory segment */

       /* now map the shared memory segment in the address space of the process */
       ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
       if (ptr == MAP_FAILED) {
              printf("Map failed\n");
              return -1;
       }
       /* Now write to the shared memory region.
        * Note we must increment the value of ptr after each write. */
       sprintf(ptr,"%s",message0);
       ptr += strlen(message0);
       sprintf(ptr,"%s",message1);
       ptr += strlen(message1);
       return 0;
}
```

IPC POSIX Consumer

shm_consumer.c
compiling command:
gcc –o consumer shm_consumer.c -lrt

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
int main() {
       const char *name = "OS";          //name of the shared memory object
       const int SIZE = 4096;            //size of shared memory object
       int shm_fd;                       //shared memory file descriptor
       void *ptr;                        //pointer to shared memory object

       /* open the shared memory segment */
       shm_fd = shm_open(name, O_RDONLY, 0666);
       if (shm_fd == -1) {
              printf("shared memory failed\n");
              exit(-1);
       }

       /* now map the shared memory segment in the address space of the process */
       ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
       if (ptr == MAP_FAILED) {
              printf("Map failed\n");
              exit(-1);
       }

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

       /* remove the shared memory segment */
       if (shm_unlink(name) == -1) {
              printf("Error removing %s\n", name);
              exit(-1);
       }
       return 0;
}
```

Ordinary Pipes

Ordinary pipe in Windows win_pipe_parent.c

```
//Windows anonymous pipe -- parent process
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#define BUFFER_SIZE 25
int main(VOID) {
		HANDLE ReadHandle, WriteHandle;
		STARTUPINFO si;
		PROCESS_INFORMATION pi;
		char message[BUFFER_SIZE] = "Greetings";
		DWORD written;

		/* set up security attributes allowing pipes to be inherited */
		SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};

		/* allocate memory */
		ZeroMemory(&pi, sizeof(pi));

		/* create the pipe */
		if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
				fprintf(stderr, "Create Pipe Failed");
				return 1;
		}

		/* establish the START_INFO structure for the child process */
		GetStartupInfo(&si);
		si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

		/* redirect standard input to the read end of the pipe */
		si.hStdInput = ReadHandle;
		si.dwFlags = STARTF_USESTDHANDLES;
```

Ordinary Pipes

Ordinary pipe in Windows cont.   win_pipe_parent.c

```
        /* don’t allow the child to inherit the write end of pipe */
        SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

        /* create the child process */
        CreateProcess(NULL, "child.exe", NULL, NULL,
        TRUE, /* inherit handles */
        0, NULL, NULL, &si, &pi);

        /* close the unused end of the pipe */
        CloseHandle(ReadHandle);

        /* the parent writes to the pipe */
        if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
                fprintf(stderr, "Error writing to pipe.");

        /* close the write end of the pipe */
        CloseHandle(WriteHandle);

        /* wait for the child to exit */
        WaitForSingleObject(pi.hProcess, INFINITE);
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
        //Sleep(10000);
        return 0;
}
```

Ordinary Pipes

Ordinary pipe in Windows cont.   win_pipe_child.c

```
//Windows anonymous pipe -- child process
//executable filename: child.exe

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

#define BUFFER_SIZE 25

int main(VOID) {
		HANDLE ReadHandle;
		CHAR buffer[BUFFER_SIZE];
		DWORD read;

		/* get the read handle of the pipe */
		ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

		/* the child reads from the pipe */
		if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
			printf("child read message[%s]",buffer);
		else
			fprintf(stderr, "Error reading from pipe");
		return 0;
}
```

Sockets in Java

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

public class DateServer {
    public static void main(String[] args)  {
            try {
                ServerSocket sock = new ServerSocket(6013); //6013 is the port number

                // now listen for connections
                while (true) {
                      Socket client = sock.accept();
                      // we have a connection
                      PrintWriter pout = new PrintWriter(client.getOutputStream(), true);

                      //write a few messages to the client
                      pout.println("Hello Client! ");
                      pout.println(client.getInetAddress().toString());
                      pout.println(client.getPort());
                      // write the Date to the client
                      pout.println(new java.util.Date().toString());

                      // close the socket and resume listening for more connections
                      client.close();
                }
            }
            catch (IOException ioe) {
                System.err.println(ioe);
            }
    }
}
```

Sockets in Java

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

public class DateClient
{
    public static void main(String[] args)  {
                try {
                                // this could be changed to an IP name or address other than the localhost
                                Socket sock = new Socket("127.0.0.1",6013);
                                InputStream in = sock.getInputStream();

                                BufferedReader bin = new BufferedReader(new InputStreamReader(in));

                                String line;
                                while( (line = bin.readLine()) != null)
                                                System.out.println(line);

                                sock.close();
                }
                catch (IOException ioe) {
                                System.err.println(ioe);
                }
    }
}
```