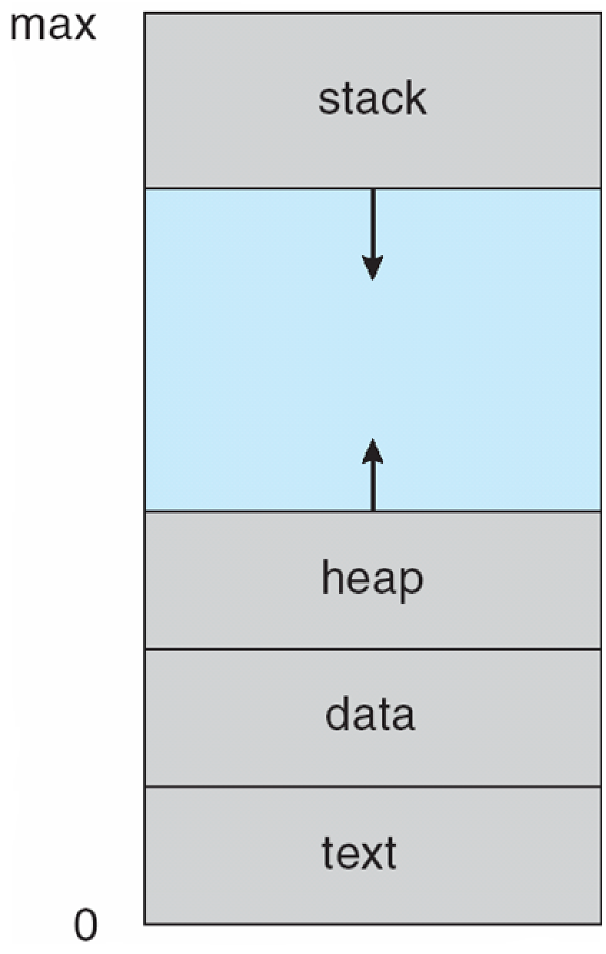
Chapter 3: Processes

- What is a process ?
- What is process scheduling ?
- What are the common operations on processes ?
- How to conduct process-level communication ?
- How to conduct client-server communication ?

Process Concept

- Process
  - is a program in execution
  - is an instance of a computer program being sequentially executed
  - process execution must progress in sequential fashion
  - process is also called a *job*
- Program Vs. process
  - program is a *passive* entity; process is an *active* entity
  - program only contains text; process is associated with code, data, PC, heap, stack, registers, and other information
  - program becomes a process when an executable file is loaded into memory
  - same program executed multiple times will correspond to different process each time

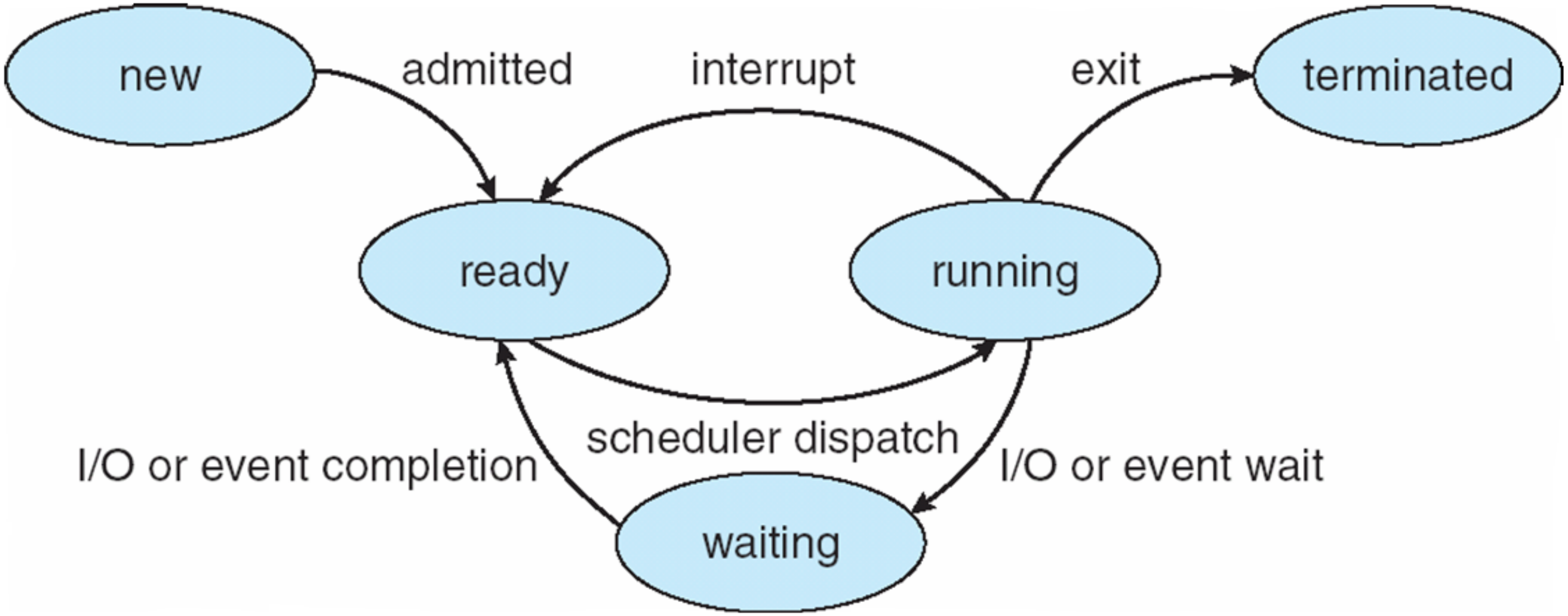
Process in Memory



Process State

- During execution, the process may be in one of the following *states*
  - new – process is being created
  - running – instructions are being executed
  - waiting – waiting for some event to occur
  - ready – waiting to be assigned a processor
  - terminated – process has finished execution
- Each processor can only run one process at a instant.

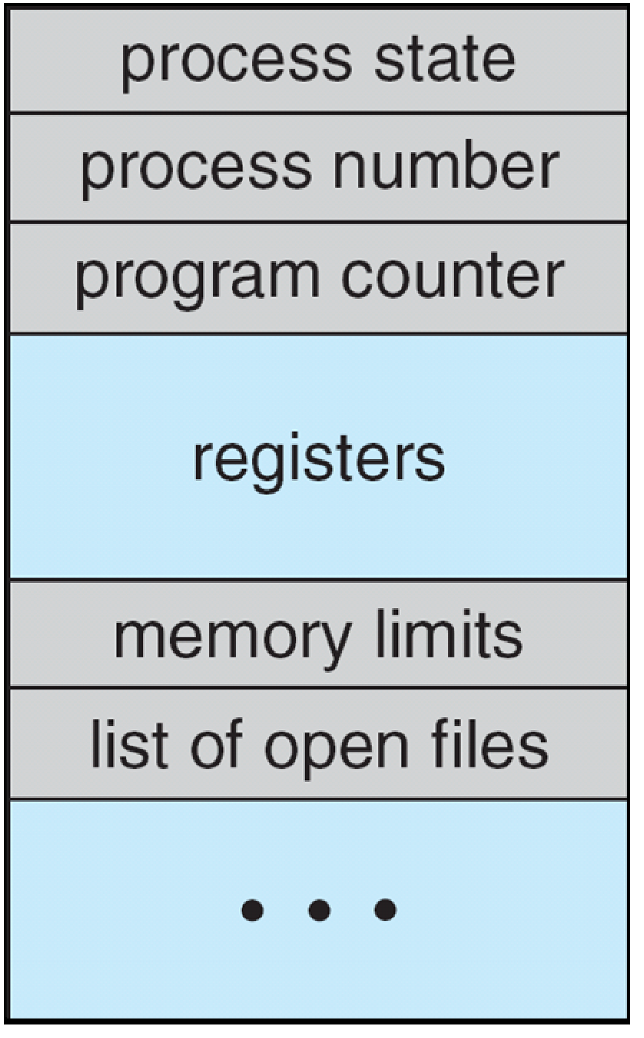
Diagram of Process State



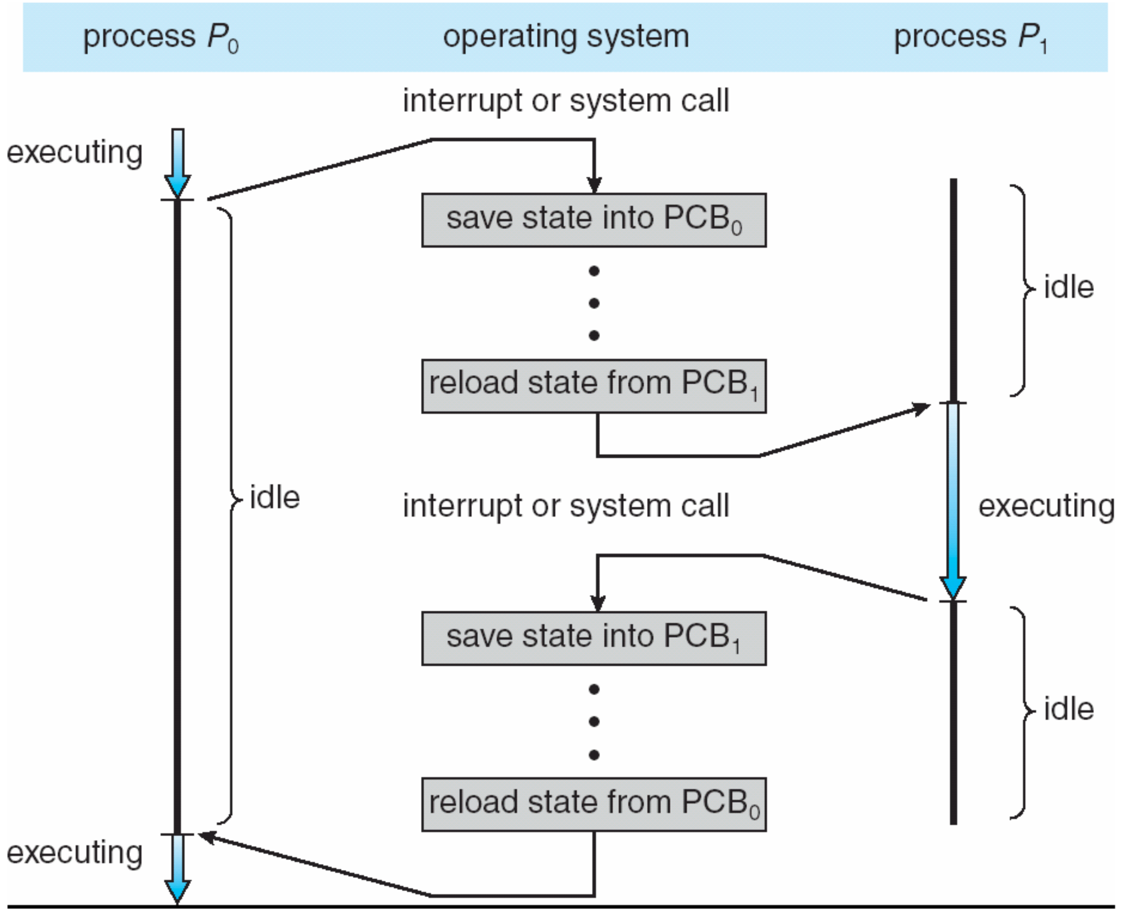
Process Control Block (PCB)

- PCB is representation of a process in an operating system.
  - maintains process-specific information
  - necessary for scheduling
- Information associated with each process
  - process state
  - program counter
  - CPU registers
  - CPU scheduling information
  - memory-management information
  - accounting information
  - I/O status information

Process Control Block (PCB) (2)



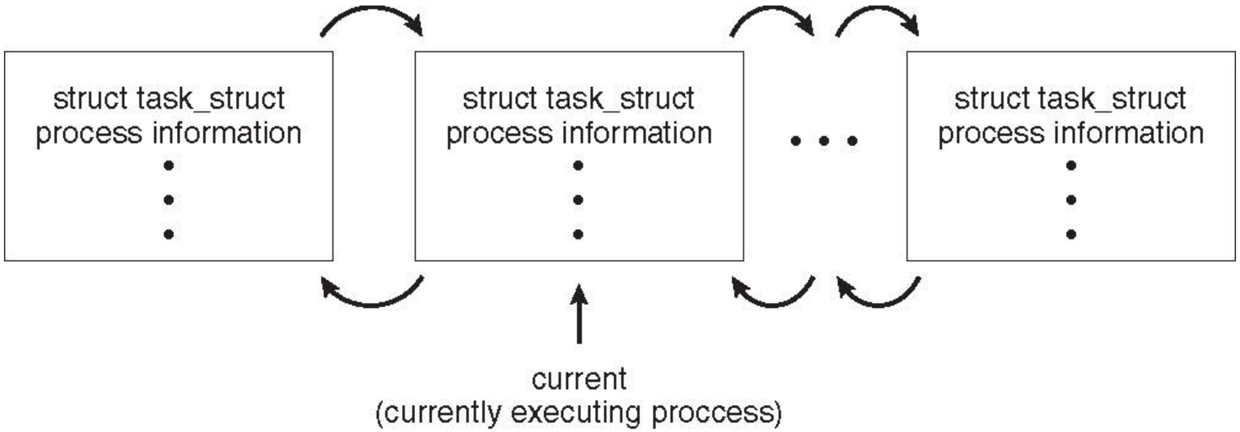
CPU Switch From Process to Process



Process Representation in Linux

Represented by the C structure `task_struct`

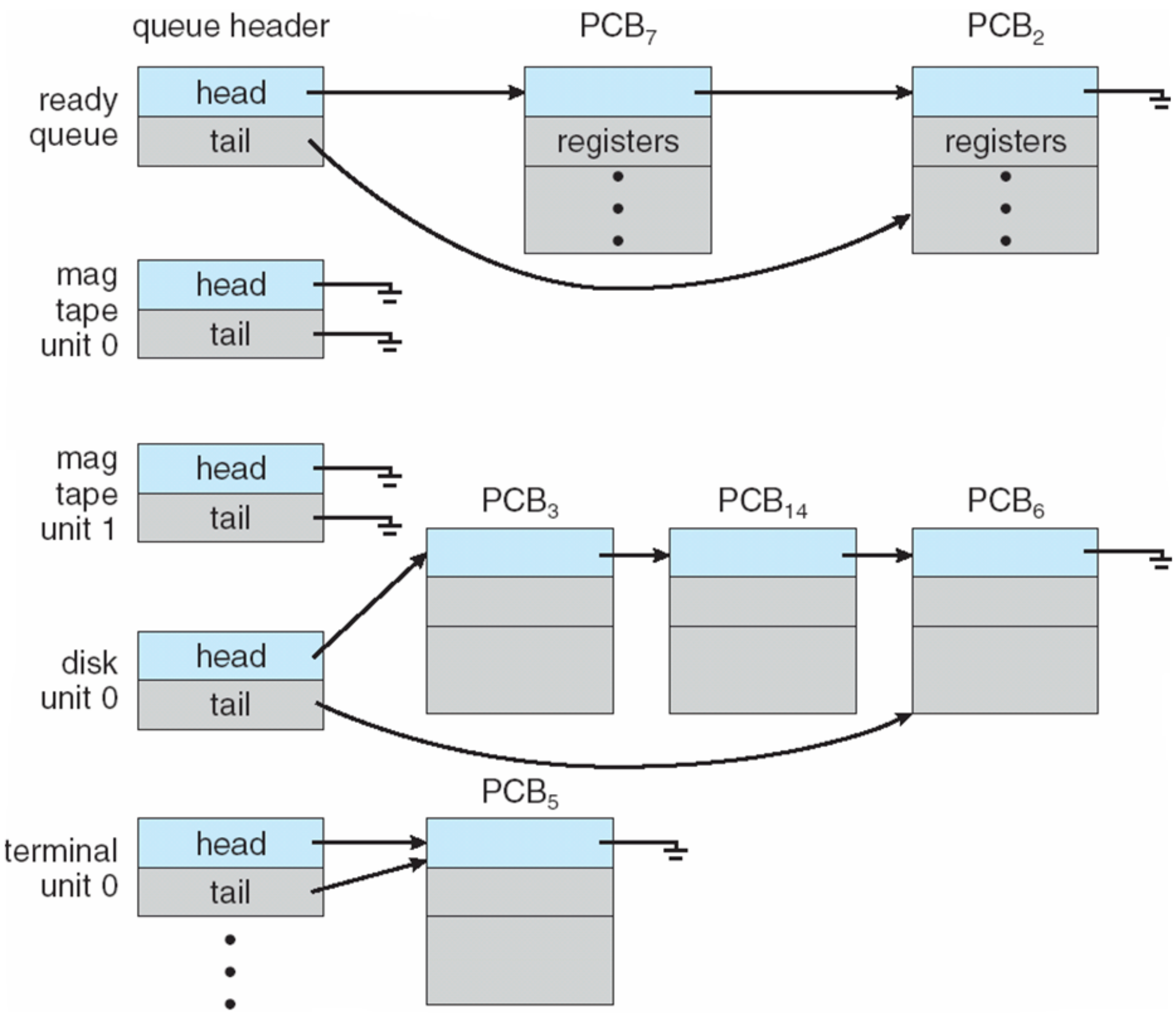
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



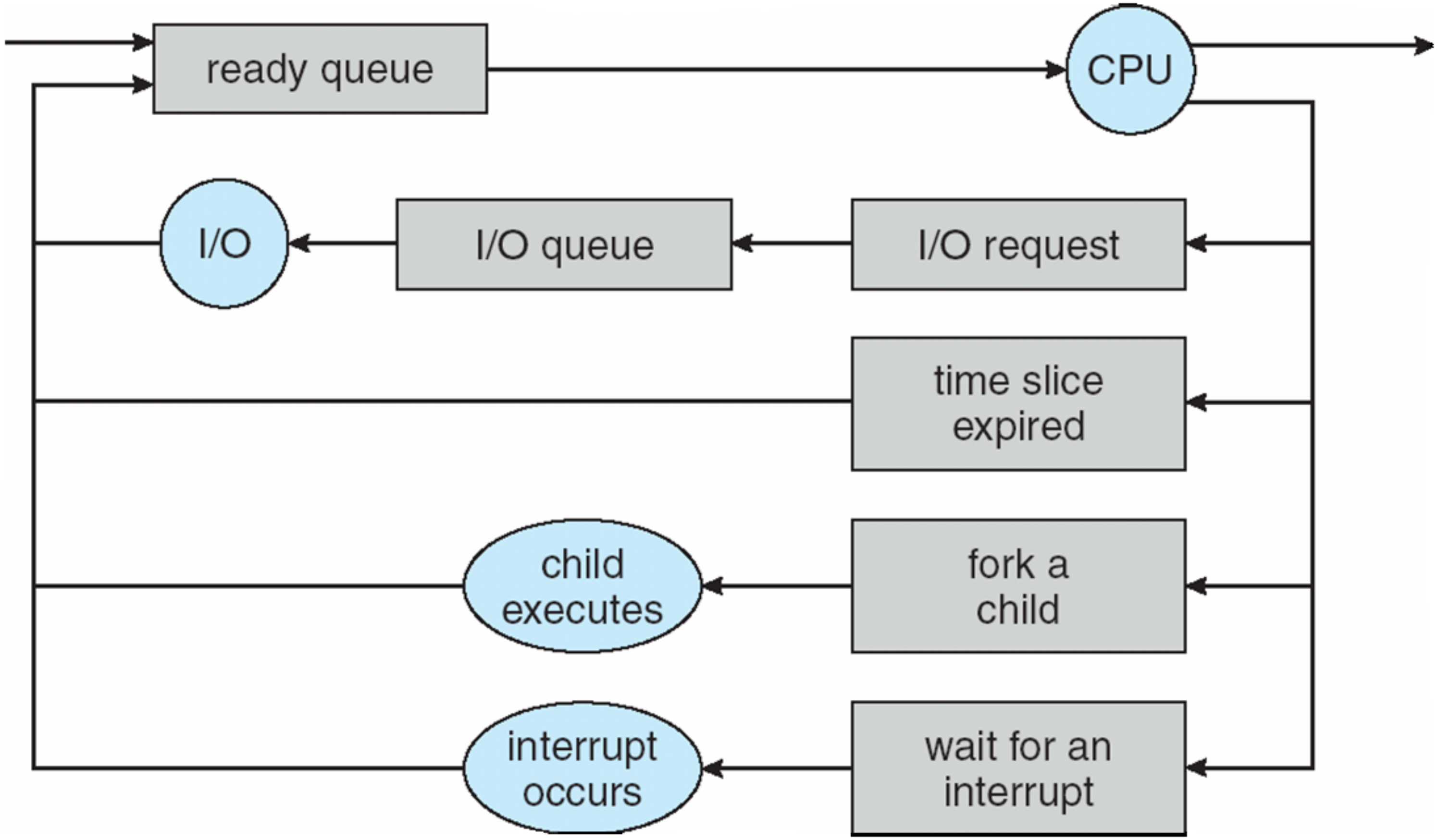
Process Scheduling

- Process scheduling selects the process to run on a CPU
  - maximizes CPU utilization in a multiprogramming OS
  - provides illusion of each process owning the system in a time-shared OS
- Terminology used in OS schedulers
  - **job queue** – set of all processes in the system
  - **ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Schedulers

- Systems with a possibility of huge deluge of job requests may use multiple schedulers.
- **Long-term scheduler** (or job scheduler)
  - selects processes to be brought into the ready queue
  - controls the *degree of multiprogramming*
  - controls the mix of active CPU-bound and I/O-bound processes
  - invoked infrequently
  - can afford more time to make selection decision
- **Short-term scheduler** (or CPU scheduler)
  - selects the process to be executed next and allocates CPU
  - invoked frequently
  - necessary to limit scheduling overhead

Context Switch

- A *context switch* is the process of storing and restoring the state (context) of the CPU such that multiple processes can share a single CPU resource
  - for time-shared or multiprogramming environments
  - context of a process represented in the PCB
  - context switch involves a state *save* of the current process, and a state *restore* of the process being resumed next
  - switch from *user* to *kernel* mode or vice-versa is a mode switch
- Context-switch time is overhead
  - the system does no useful work while switching
  - overhead depends on hardware support
    - Sun UltraSPARC provides multiple banks of registers
    - Intel x86 processors also provide some support

Process Creation

- Any process can create other processes during its execution
  - operating systems have a *primordial* process
  - creating process called parent process
  - new process called child process
  - processes 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 options
  - parent and children execute concurrently
  - parent waits until children terminate

Process Creation (Cont)

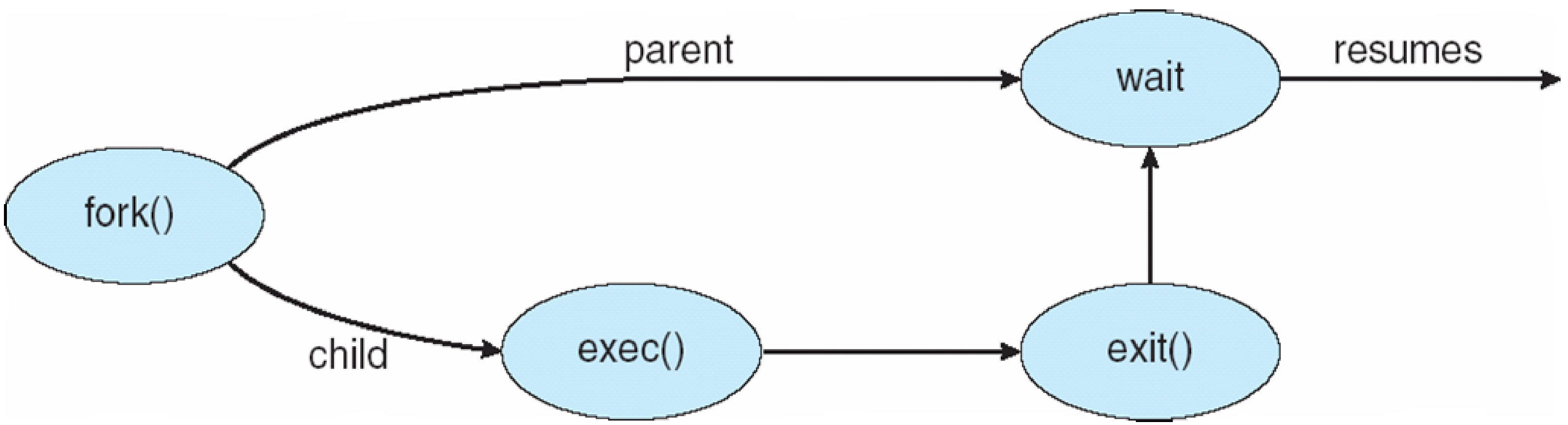
- Address space options
  - child duplicate of parent
  - child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

Process Creation Example on Unix

```
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 Creation

- Parent waiting for child process to finish



Process Termination

- Process terminates after executing last statement
  - can explicitly invoke the **exit** system call to terminate
  - OS implicitly calls exit
  - child can pass return status 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**

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

Interprocess Communication

- Communication within the same system.
- Processes may need to *co-operate* for several reasons
  - information sharing
  - computation speedup
  - modularity
  - convenience
- Cooperating process can affect or be affected by other processes
  - typically, by sharing data
- Cooperating processes need **interprocess communication** (**IPC**)

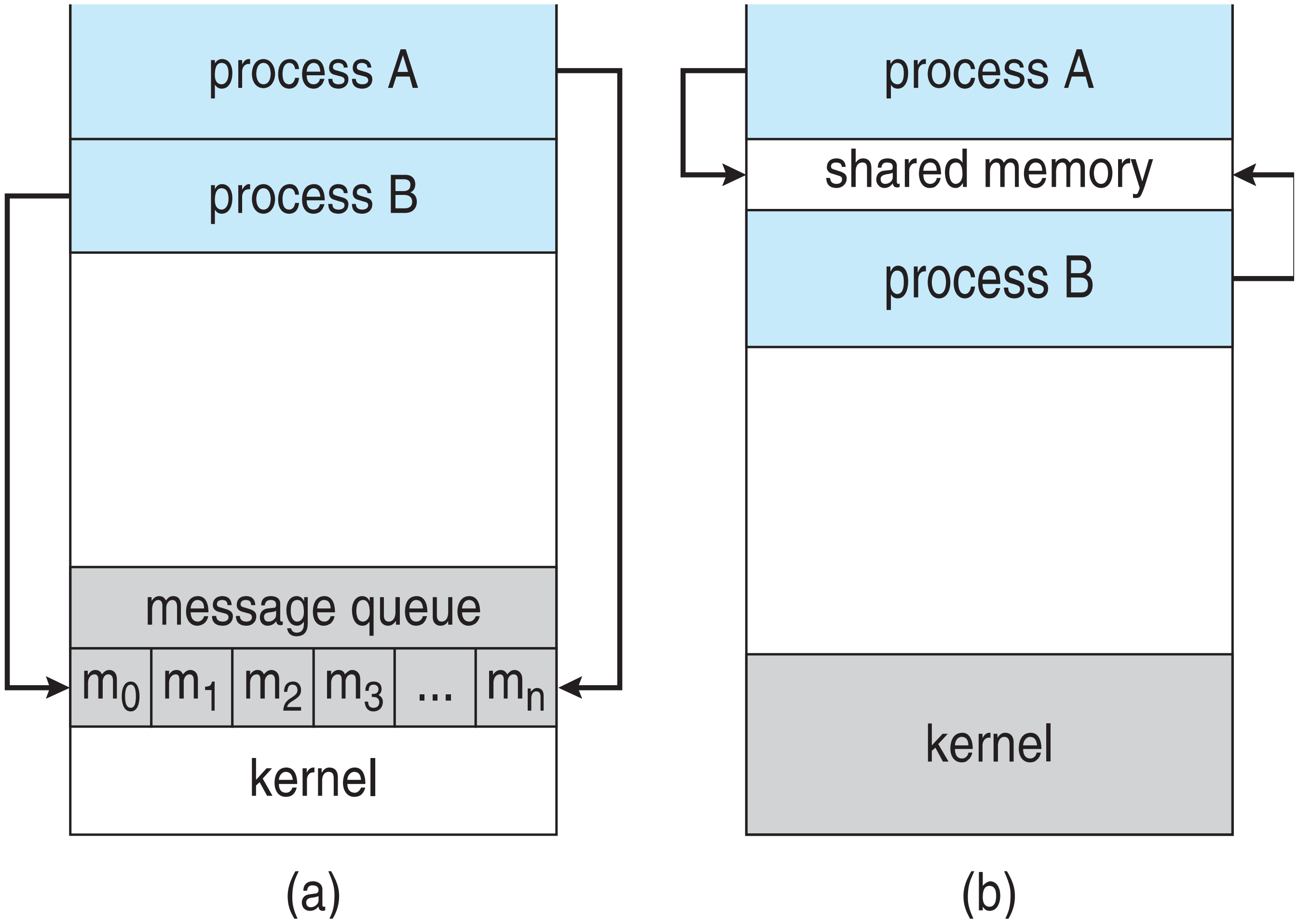
Producer-Consumer Problem

- Common paradigm for co-operating processes
  - *producer* process produces information
  - *consumer* process consumes the produced information
- Processes need synchronization
  - *consumer* cannot use information before it is produced by the *producer*
- Abstraction models
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Models of IPC

- Shared memory
  - share a region of memory between co-operating processes
  - read or write to the shared memory region
  - fast communication
  - convenient communication
- Message passing
  - exchange messages (*send* and *receive*)
  - typically, messages do not overwrite each other
    - no need for conflict resolution
  - typically, used for sending smaller amounts of data
  - slower communication
  - easy to implement (even for inter-computer communication)

Models of IPC (2)



message passing

shared memory

Message Passing

- Another mechanism for interprocess communication
  - can be employed for client-server communication
- Message passing facility provides at least two operations:
  - **send** (*message*) and **receive** (*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation issues
  - how are links established?
  - can a link be associated with more than two processes?
  - how many links between every pair of communicating processes?
  - what is the capacity of a link?
  - fixed or variable sized message ?
  - is the link unidirectional or bi-directional?

Message Passing – Naming

- 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
  - disadvantage
    - process identifiers are hard-coded

Message Passing – Naming (2)

- Indirect communication
  - messages are directed and received from mailboxes (also referred to as ports)
    - **send** (A, message) – send a message to mailbox A
    - **receive** (A, message) – receive a message from mailbox A
  - each mailbox has a unique id
  - processes can communicate only if they share a mailbox
  - properties of communication link
    - link may be associated with many processes
    - each pair of processes may share several communication links
    - link may be unidirectional or bi-directional
    - multiple receivers may need synchronization
  - mailbox can be held in the process address space or in the kernel

Message Passing (3)

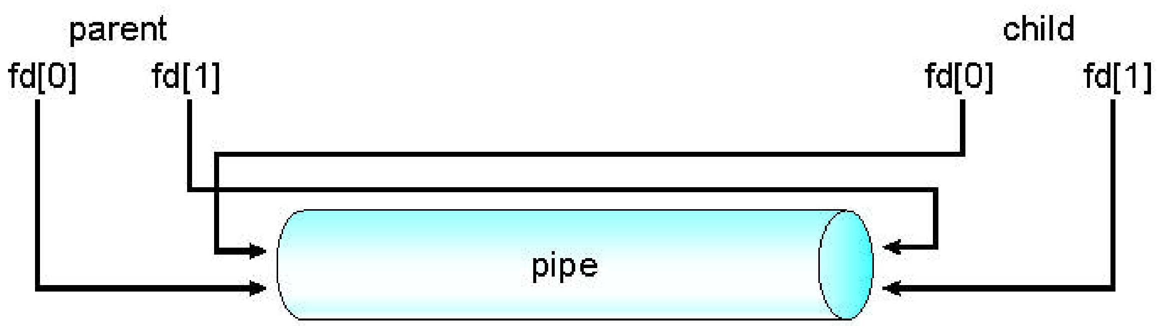
- Synchronization
  - message passing may be either blocking (synchronous) or non-blocking (asynchronous)
  - **blocking send** has the sender block until the message is received
  - **blocking receive** has the receiver block until a message is available
  - **non-blocking** send has the sender send the message and continue
  - **non-blocking** receive has the receiver receive a valid message or null
- Buffering – queue of messages attached to the link
  - zero capacity – 0 messages
    - Sender must wait for receiver
  - bounded capacity – finite length of *n* messages
    - Sender must wait if link full
  - unbounded capacity – infinite length
    - Sender never waits

Interprocess Communication in Unix

- Provides multiple modes of IPC
  - pipes
  - FIFOs (names pipes)
  - message queues
  - shared memory
  - sockets

Pipes

- Most basic form of IPC on all Unix systems
  - also provides a useful command-line interface
- Conduit for two processes to communicate
  - ordinary pipes require parent-child relationship between communicating processes



Pipes

- Issues to be addressed
  - is communication unidirectional or bidirectional ?
    - Unix pipes only allow unidirectional communication
  - should communication processes be related ?
    - *anonymous* pipes can only be constructed between parent-child
  - can pipes communicate over a network
    - processes must be controlled by the same OS
- Pipes exist only until the processes exist
  - pre-mature process exit may cause data loss
- Data can only be collected in FIFO order

Simple Example Using Pipes

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>

main()
{
  char *s, buf[1024];
  int fds[2];
  s  = "EECS 678 Spring 2009\n";

  /* open a pipe. fd[0] is opened for reading,
     and fd[1] for writing.*/
  pipe(fds);

  /* write to the write-end of the pipe */
  write(fds[1], s, strlen(s));

  /* This can be read from the other end of the pipe */
  read(fds[0], buf, strlen(s));

  printf("fds[0]=%d, fds[1]=%d\n", fds[0], fds[1]);
  write(1, buf, strlen(s));
}
```

IPC Example Using Pipes

```
main()
{
  char *s, buf[1024];
  int fds[2];
  s  = "EECS 678 Spring 2009. Pipe program 2\n";

  /* create a pipe */
  pipe(fds);

  /* create a new process using fork */
  if (fork() == 0) {

    /* child process. All file descriptors, including
       pipe are inherited, and copied.*/
    write(fds[1], s, strlen(s));
    exit(0);
  }

  /* parent process */
  read(fds[0], buf, strlen(s));
  write(1, buf, strlen(s));
}
```

Pipes Used for Process Synchronization

```
main()
{
  char *s, buf[1024];
  int fds[2];
  s  = "EECS 678 Spring 2009. Pipe program 3\n";

  /* create a pipe */
  pipe(fds);

  if (fork() == 0) {

    /* child process. */
    printf("Child line 1\n");
    read(fds[0], s, strlen(s));
    printf("Child line 2\n");
  } else {

    /* parent process */
    printf("Parent line 1\n");
     write(fds[1], buf, strlen(s));
    printf("Parent line 2\n");
  }
}
```

Pipes Used in Unix Shells

- Pipes commonly used in most Unix shells
  - output of one command is input to the next command
  - example: `/bin/ps -ef | /bin/more`
- How does the shell realize this command?
  - create a process to run `ps -ef`
  - create a process to run `more`
  - create a pipe from `ps -ef` to `more`
  - the standard output of the process to run `ps -ef` is redirected to a pipe streaming to the process to run `more`
  - the standard input of the process to run `more` is redirected to be the pipe from the process running `ps -ef`

FIFO (Named Pipes)

- Pipe with a name !
- More powerful than *anonymous* pipes
  - no parent-sibling relationship required
  - allow bidirectional communication
  - FIFOs exists even after creating process is terminated
- Characteristics of FIFOs
  - appear as typical files
  - only allow half-duplex communication
  - communicating process must reside on the same machine

Producer Consumer Example with FIFO

- Producer Code:

```
main()
{
  char str[MAX_LENGTH];
  int num, fd;

  mkfifo(FIFO_NAME, 0666); // create FIFO file

  printf("waiting for readers...");
  fd = open(FIFO_NAME, O_WRONLY); // open FIFO for writing
  printf("got a reader !\n");

  printf("Enter text to write in the FIFO file: ");
  fgets(str, MAX_LENGTH, stdin);
  while(!(feof(stdin))){
    if ((num = write(fd, str, strlen(str))) == -1)
      perror("write");
    else
      printf("producer: wrote %d bytes\n", num);
    fgets(str, MAX_LENGTH, stdin);
  }
}
```

Producer Consumer Example with FIFO (2)

- Consumer code:

```
main()
{
  char str[MAX_LENGTH];
  int num, fd;

  mkfifo(FIFO_NAME, 0666); // make fifo, if not already present

  printf("waiting for writers...");
  fd = open(FIFO_NAME, O_RDONLY); // open fifo for reading
  printf("got a writer !\n");

  do{
    if((num = read(fd, str, MAX_LENGTH)) == -1)
      perror("read");
    else{
      str[num] = '\0';
      printf("consumer: read %d bytes\n", num);
      printf("%s", str);
    }
  }while(num > 0);
}
```

Message Passing in Unix

- Linux uses indirect communication or mailboxes.
- Queues can be associated with multiple processes
  - synchronization may be required
- Communicating processes can use any number of queues
  - each queue is identified by a unique identifier
- Capacity of the link is system initialized
  - can be over-ridden by the user
- Messages are of a fixed size
  - specified by the buffer length
- Each communicating process can send and receive from the same queue.

Message Queue Example

```
int main()
{
  /* identifier for the message queue */
  int queue_id;
  /* send and receive message buffers */
  struct msg_buf send_buf, recv_buf;

  /* create a message queue */
  queue_id = msgget(0, S_IRUSR|S_IWUSR|IPC_CREAT);

  /* send a message to the queue */
  send_buf.mtype = 1;
  strcpy(send_buf.buffer, "EECS 678 Class");
  msgsnd(queue_id, (struct msg_buf *)&send_buf, sizeof(send_buf));

  /* get the message from the queue */
  msgrcv(queue_id, (struct msg_buf *)&recv_buf, sizeof(recv_buf), 0, 0);
  printf("%s\n", recv_buf.buffer);

  /* delete the message queue, and deallocate resources */
  msgctl(queue_id, IPC_RMID, NULL);

  return 0;
}
```

```
struct msg_buf{
   long mtype;
   char buffer[1000];
}
```

Message Queues Example (2)

- Message passing in Linux is done via message queues.
- msgget – create a new message queue
  - return existing queue identifier if it exists
- msgsnd – send a message to the queue
  - each message should be in a buffer like,

```
struct msg_buf {
     long mtype;
     char mtext[1]; }
```

  - nonblocking, unless no space in the queue
- msgrcv – receive message from the queue
  - `mtype` can be used to get specific messages
- msgctl – perform control operations specified by *cmd*
  - second argument, we use it to terminate queue

Memory Sharing in Unix

- Multiple processes share single chunk of memory.
- Implementation principles
  - uniquely naming the shared segment
    - system-wide or anonymous name
  - specifying access permissions
    - read, write, execute
  - dealing with race conditions
    - atomic, synchronized access
- Most *thread*-level communication is via shared memory.

Shared Memory Example

```
int main()
{
  int segment_id;
  char *shared_memory;
  const int size = 4096;

  /* allocate and attach a shared memory segment */
  segment_id = shmget(IPC_PRIVATE, size, S_IRUSR|S_IWUSR);
  shared_memory = (char *) shmat(segment_id, NULL, 0);

  /* write and print a message to the shared memory segment */
  sprintf(shared_memory, "EECS 678 Spring 2009 Class");
  printf("%s\n", shared_memory);

  /* detach and remove the shared memory segment */
  shmdt(shared_memory);
  shmctl(segment_id, IPC_RMID, NULL);

  return 0;
}
```

Shared Memory Example (2)

- shmget – create shared memory segment
  - IPC_PRIVATE specifies creation of new memory segment of size rounded to the system page size
  - access permissions as for normal file access
  - returns identifier of shared memory segment
- shmat – attach shared memory segment
  - must for every process wanting access to the region
  - segment identified by segment_id
  - system chooses a suitable attach address
- shmctl – performs the control operation specified by *cmd*
  - command is IPC_RMID to remove shared segment
- see program shared_memory2.c
- Read man pages!

Unix Domain Sockets

- Sockets
  - can be defined as an end-point for communications
  - two-way communication pipe
  - can be used in a variety of domains, including *Internet*
- Unix Domain Sockets
  - communication between processes on the same Unix system
  - special file in the file system
- Mostly used for client-server programming
  - client sending requests for information, processing
  - server waiting for user requests
  - server performing the requested activity and sending updates to client
- Socket communication modes
  - connection-based, TCP
  - connection-less, UDP

Unix Domain Sockets – System Calls

- socket ( ) - create the Unix socket
  - `int socket(int domain, int type, int protocol);`
  - domain is `AF_UNIX`
- bind ( ) - assign a name to a socket
  - `int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);`
  - `my_addr` is `addrlen` bytes long
- listen ( ) - listen to incoming client requests
  - `int listen(int sockfd, int backlog);`
  - `backlog` specifies the queue limit for incoming connctions

Unix Domain Sockets – System Calls (2)

- accept ( ) - create a new connected socket
  - `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
  - only for connection-based protocols
- recv ( ) - receive messages from socket
  - `ssize_t recv(int s, void *buf, size_t len, int flags);`
  - message placed in `buf`
- close ( ) - close the socket connection

Socket Example – Echo Server

- see socket_server.c
- see socket_client.c

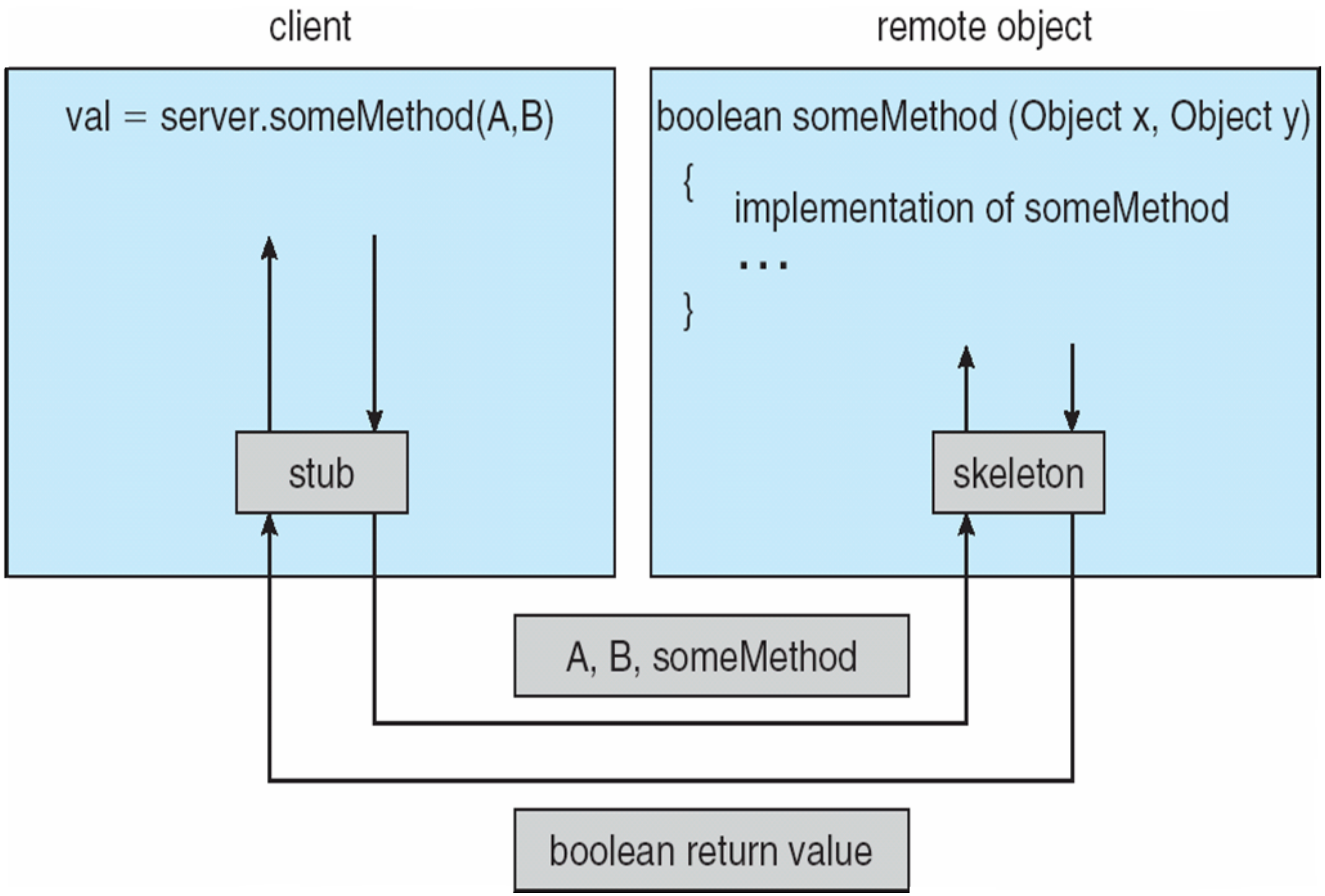
Communications in Client-Server Systems

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

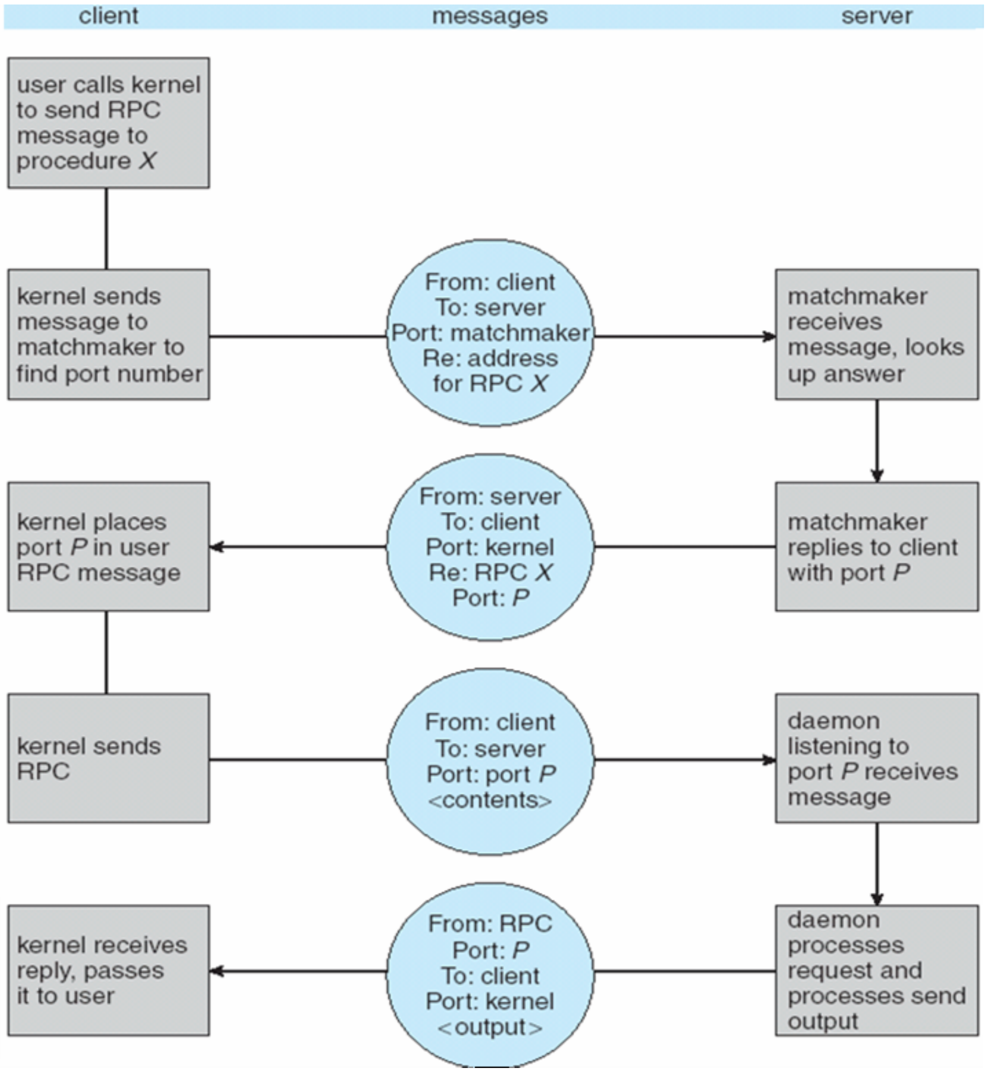
Remote Procedure Calls

- Remote procedure call (RPC) abstracts subroutine calls between processes on networked systems
  - subroutine executes in another address space
  - uses message passing communication model
  - messages are well-structured
  - RPC daemon on the server handles the remote calls
- Client-side *stub*
  - proxy for the actual procedure on the server
  - responsible for locating correct port on the server
  - responsible for *marshalling* the procedure parameters
- Server-side stub
  - receives the message; unpacks the marshalled parameters
  - performs the procedure on the server, returns result

Marshalling Parameters



Execution of RPC



Remote Method Invocation

- Remote Method Invocation (RMI)
  - Java mechanism (API) to perform RPCs
  - Java remote method protocol (JRMP) only allows calls from one JVM to another JVM
  - CORBA is used to support communication with non-JVM code
  - em