Chapter 3: Processes and Inter Process Communications

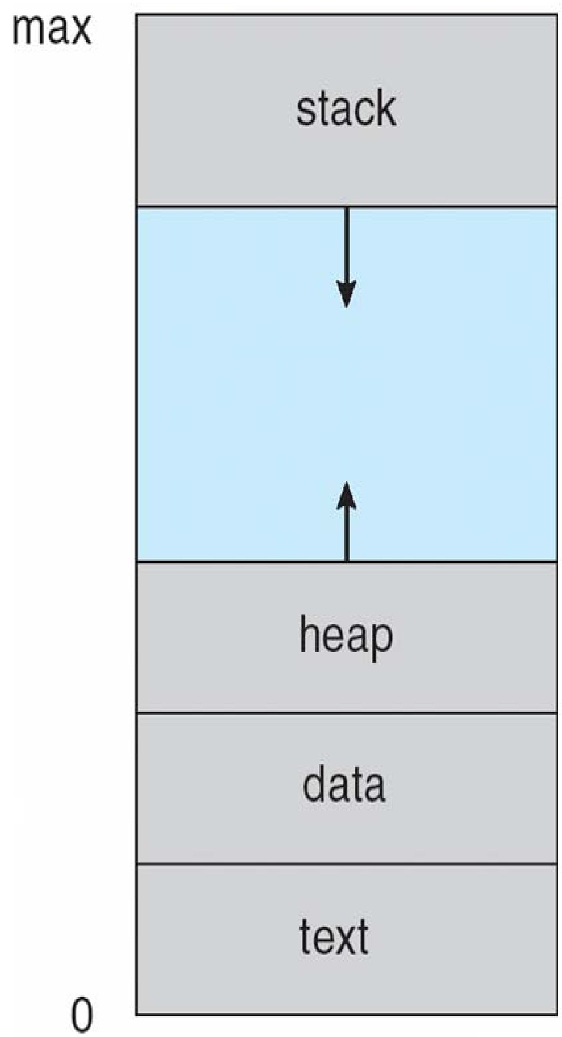
Processes

- Objectives
  - To introduce the notion of a process – a program in execution, which forms the basis of all computation
  - To describe the various features of processes, including scheduling, creation and termination, and communication
- Topics
  - Process Concept
  - Process Scheduling
  - Operations on Processes
  - Interprocess Communication
  - Examples of IPC Systems
  - Communication in Client-Server Systems

Process Concept

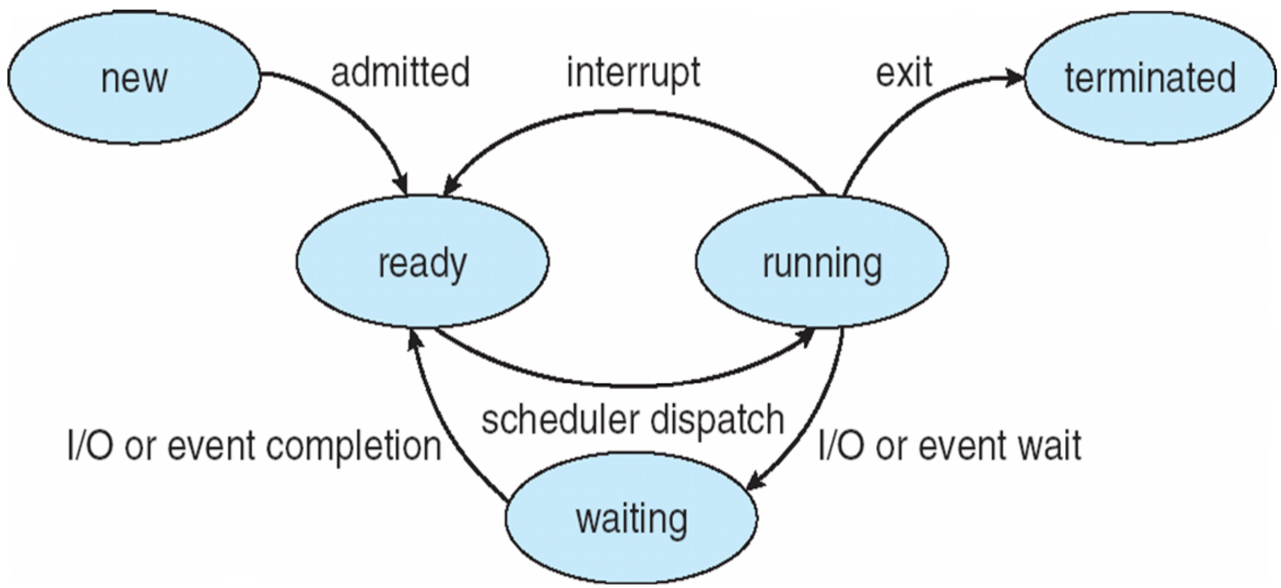
- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms job and process almost interchangeably
- Process – a program in execution; process execution must progress in a sequential fashion
- A process includes:
  - Program counter
  - Stack (and heap)
  - Data section

Process in Memory



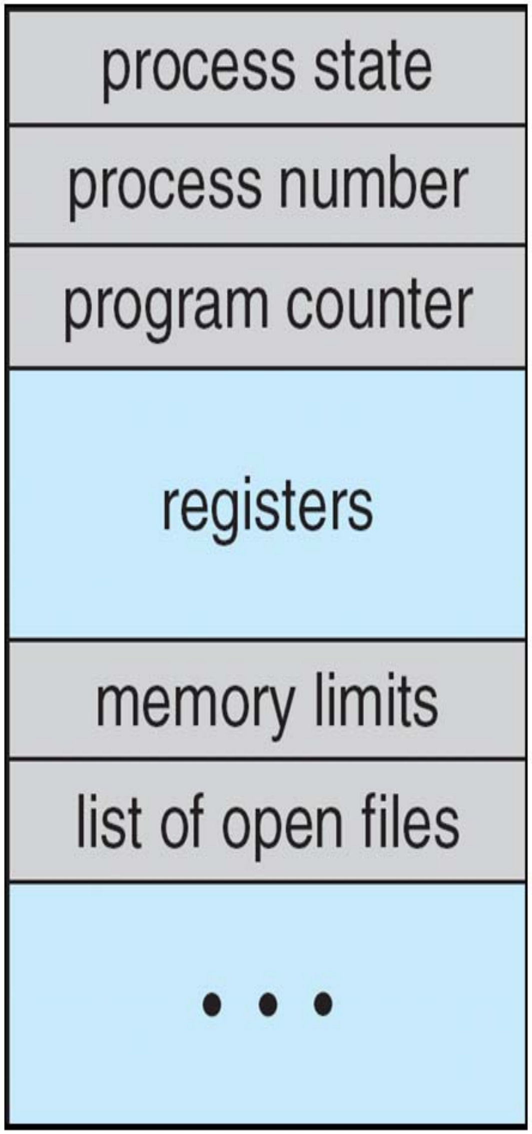
Process States

- As a process executes, it changes state (Linux)
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur
  - ready: The process is waiting to be assigned to a processor
  - terminated: The process has finished execution



Process Control Block (PCB)

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

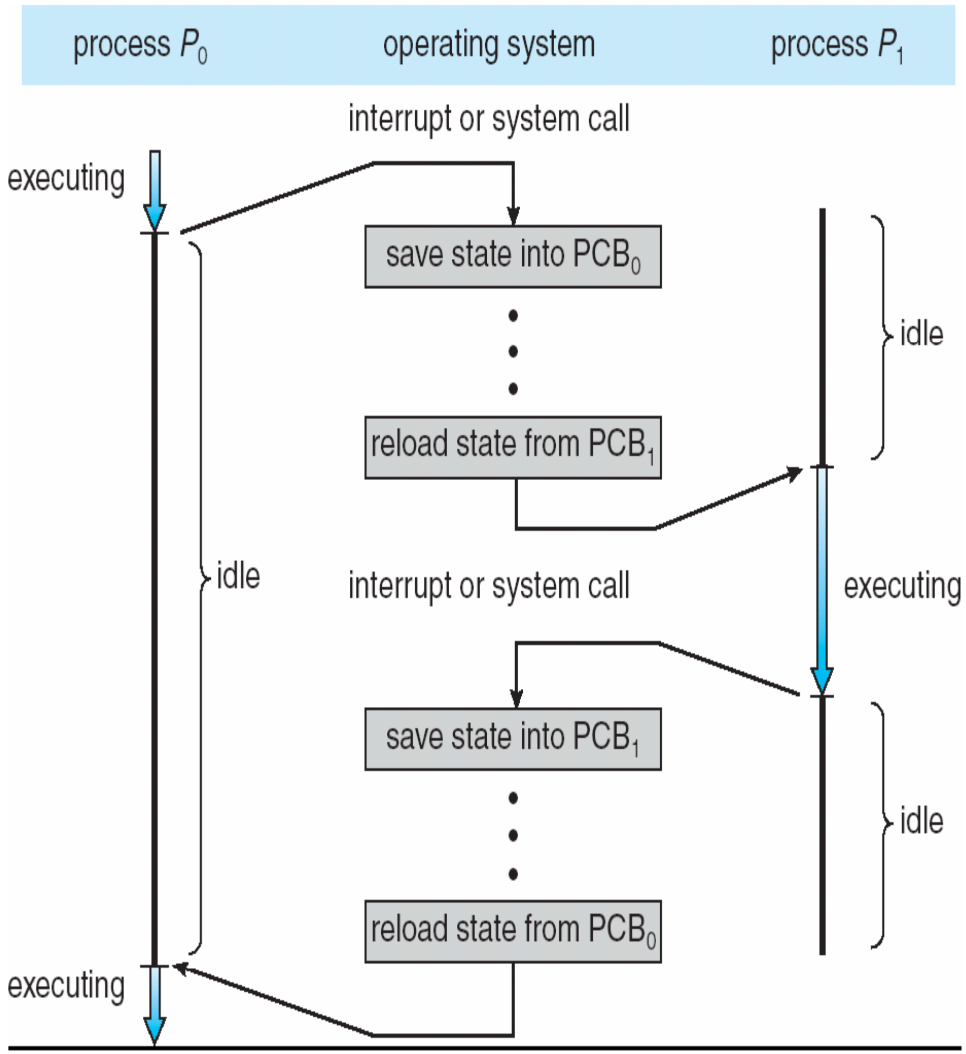


Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch

- Context of a process represented in the PCB
- Context-switch time is overhead
  - The system does no useful work while switching
- Time is dependent on hardware support

CPU Switch From Process to Process

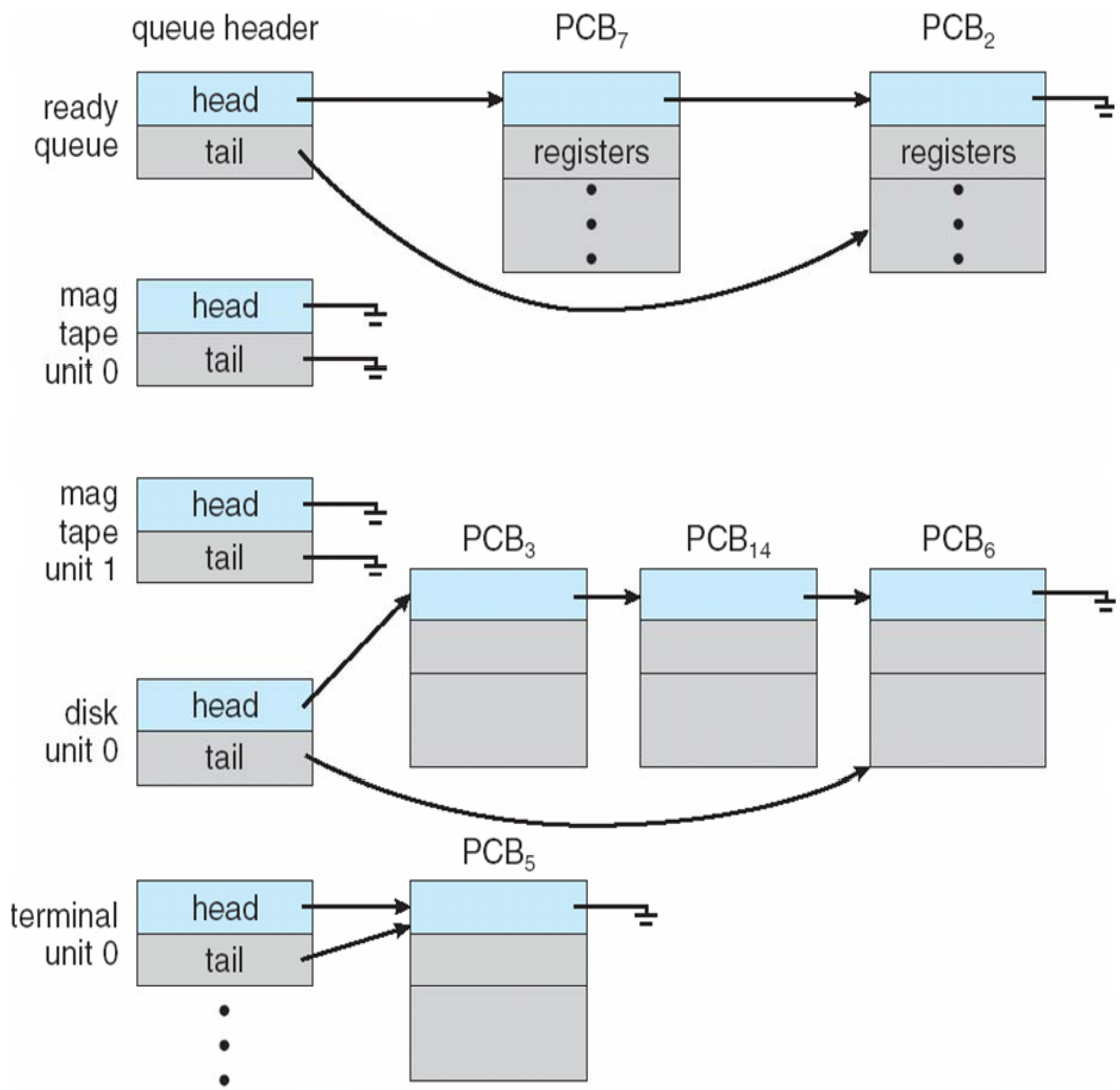


Process Scheduling Queues

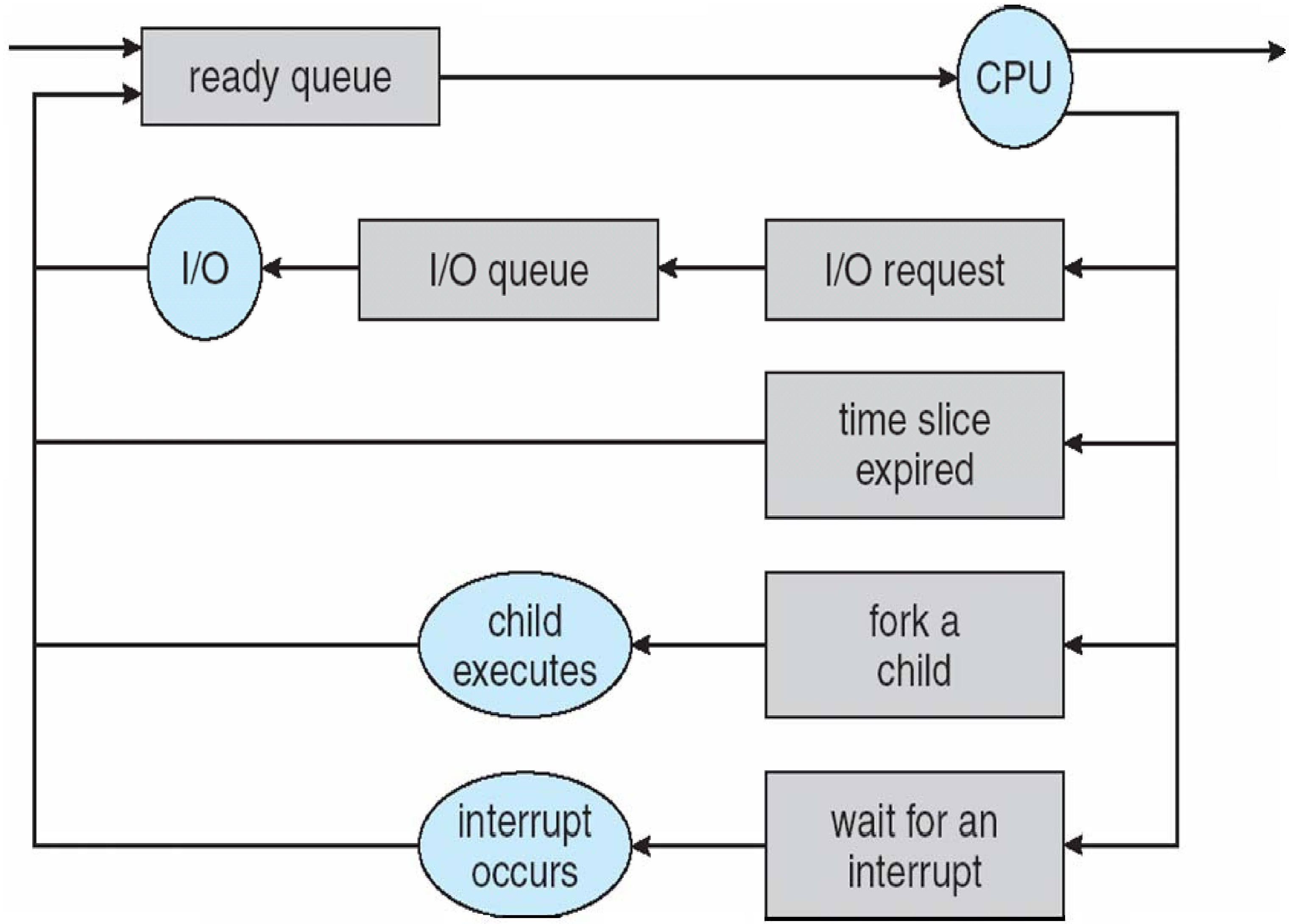
- Job queue
  – Set of all processes in the system
- Ready queue
  – Set of all processes residing in main memory, ready and waiting to execute
- Device queues
  – Set of processes waiting for an I/O device

- Processes migrate among the various queues

Ready Queue and Various I/O Device Queues

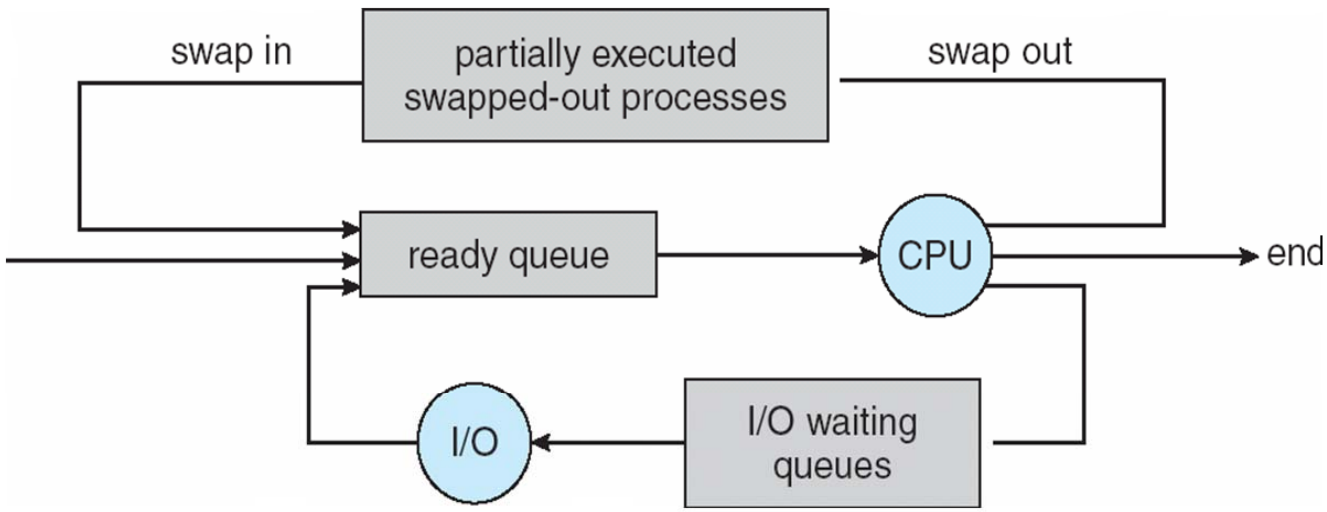


Representation of Process Scheduling



Schedulers (1)

- Long-term scheduler (or job scheduler)
  - Selects which processes should be brought into the ready queue
- Short-term scheduler (or CPU scheduler)
  - Selects which process should be executed next and allocates CPU
- Addition of medium-term scheduling
  - Swapping process in/out



Schedulers (2)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
  - May be absent, jobs simply added to ready queue

- Processes can be described as either
  - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process – spends more time doing computations; few very long CPU bursts

Process Creation (1)

- Parent process creates children processes, creates other processes, which results in forming a tree of processes

- Generally, all processes are 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

Process Creation (2)

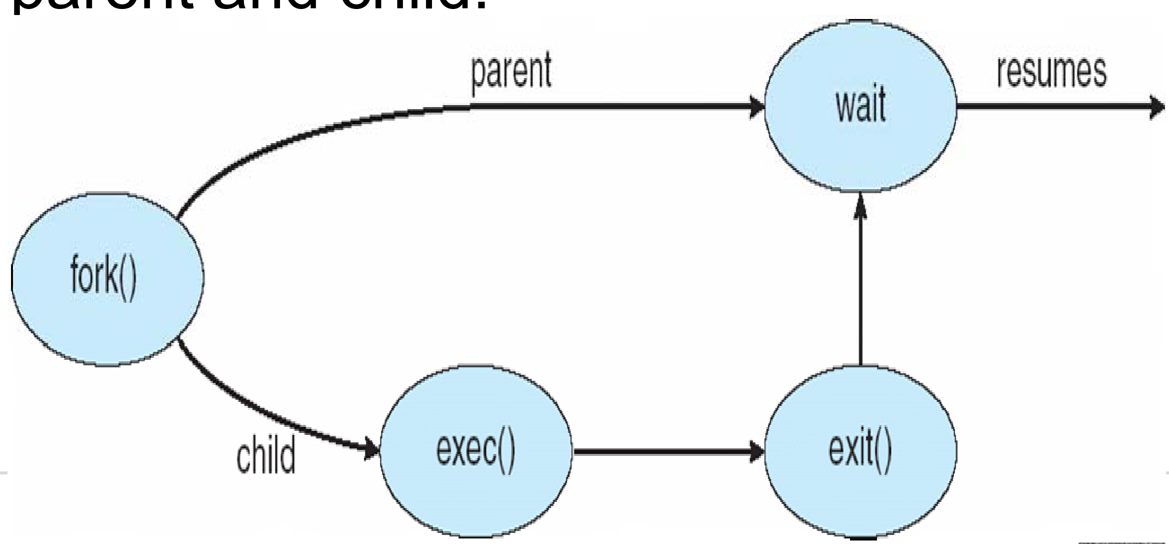
- Execution strategy options
  - Parent and children execute concurrently
  - Parent waits until children terminate

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

Explicit Process Creation

- UNIX examples
  - fork system call creates new process
- procid = fork() replicates calling process
  - exec system call used after a fork to replace the process' memory space with a new program
- Parent and child identical except for the value of procid
  - Use procid to diverge parent and child:

```
if (procid == 0)
  do_child_processing
else
  do_parent_processing
```



C Program Forking Separate Process

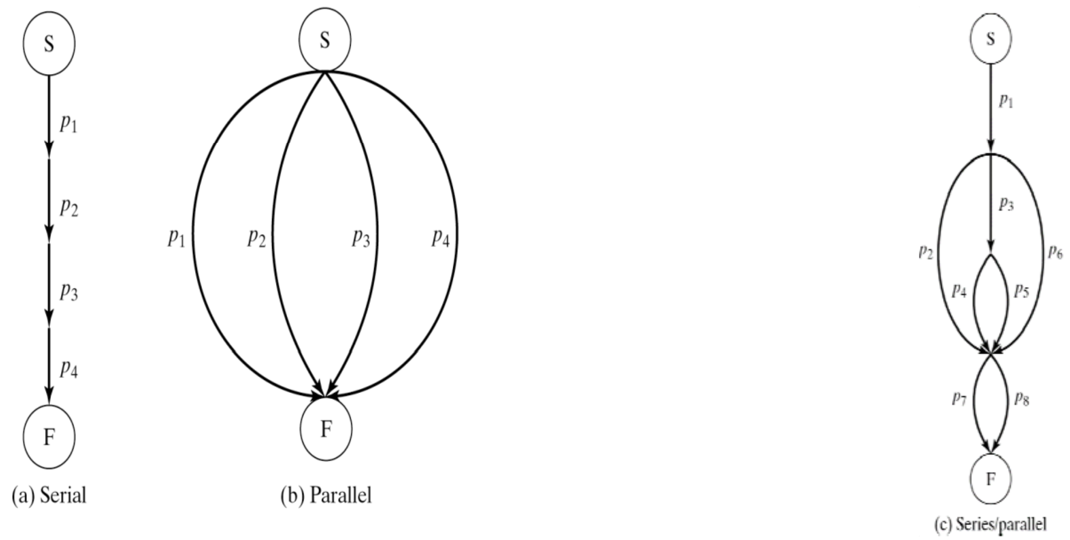
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
		pid_t pid;

		pid = fork(); /* fork a child process */

		if (pid < 0) { /* error occurred */
				printf("Fork Failed\n");
				exit(-1);
		}
		else if (pid == 0) { /* child process */
				printf("I am the child %d\n",pid);
				execlp("/bin/ls","ls",NULL);
		}
		else { /* parent process */
				printf("I am the parent %d\n",pid); /* parent will wait for the child to complete */
				wait(NULL);

				printf("Child Complete\n");
				exit(0);
		}
}
```

Process Termination

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

- Parent may terminate execution of children processes
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - Some operating system do not allow child to continue if its parent terminates
    - All children terminated - cascading termination
    - Linux: reparenting to pid 1 (init)

Process Ordering and Precedence



- Process flow graph depicts process order, parallelism, and precedence
  - Serial execution is expressed as: S(p1, p2, …)
  - Parallel execution is expressed as: P(p1, p2, …)
- Figure (c) represents the following: S(p1, P(p2, S(p3, P(p4, p5)), p6), P(p7, p8))

Data Parallelism

- Same code is applied to different data
  - *E.g.*, SIMD vector instructions

SIMD: Single Instruction Multiple Data

- The forall statement
  - Syntax: forall (parameters) statements
  - Semantics (meaning):
    - Parameters specify set of data items
    - Statements are executed for each item parallel

Data Parallelism

- Example matrix multiplication: A = B × C

```
forall ( i:1..n, j:1..m )
  A[i][j] = 0;
  for ( k=1; k<=r; ++k )
    A[i][j] = A[i][j] +
              B[i][k]*C[k][j];
```

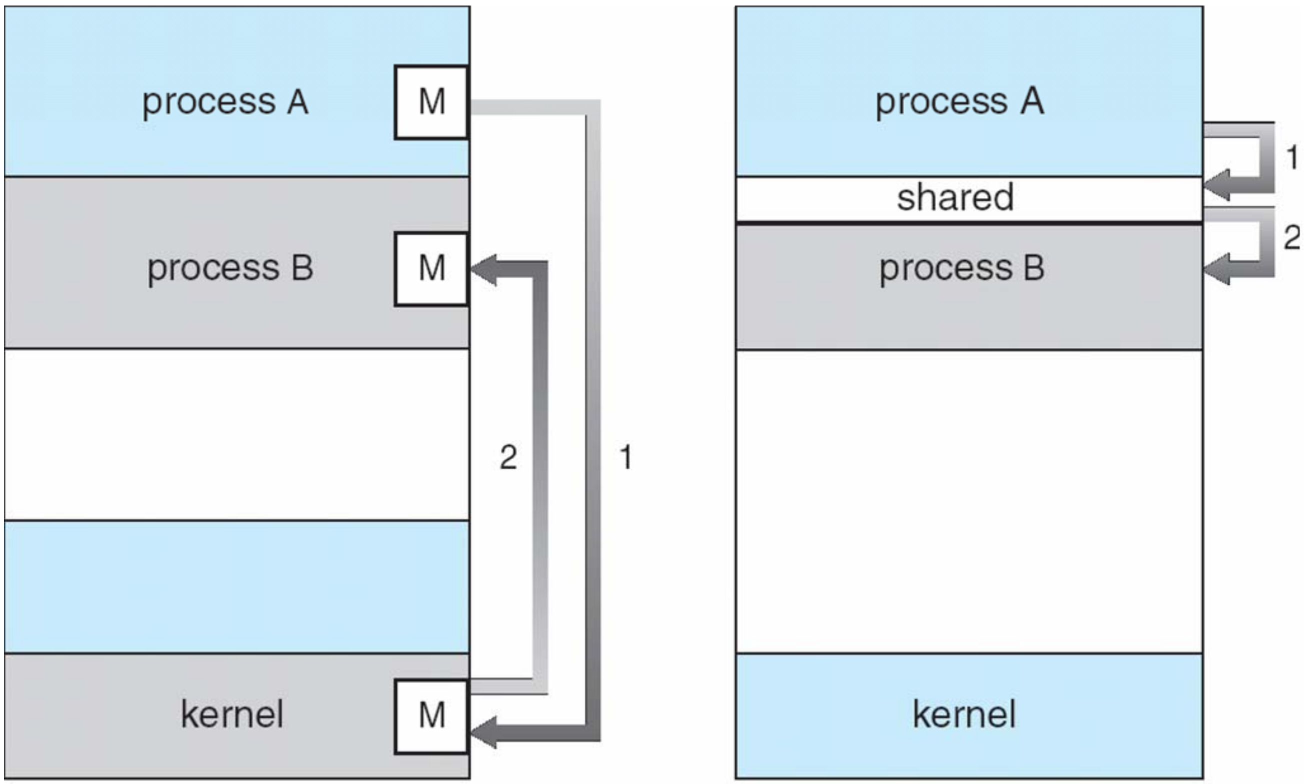
  - Each inner product is computed sequentially
  - All column-row products are computed in parallel

Interprocess Communication (IPC)

- Processes in a system: independent or cooperating
  - Independent processes cannot affect or be affected by the execution of another process
  - Cooperating processes can affect or be affected by the execution of another process
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need Interprocess Communication (IPC) mechanisms

Communications Models

- Two models of IPC
  - Message passing Shared memory



(a) (b)

Producer-Consumer Problem

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
  - Unbounded-buffer places no practical limit on the size of the buffer
  - Bounded-buffer assumes that there is a fixed buffer size
- Buffer data structure as a circular array of items
  - Using shared memory for IPC

```
#define BUFFER_SIZE 10
class item {
   . . .
};

item
buffer[BUFFER_SIZE];
// points to empty slot
int in = 0;
// next available item
int out = 0;
```

Bounded-Buffer – Producer

```
while (true) {
  // Produce an item
  while (((in + 1) % BUFFER SIZE count)  ==
out);
  // do nothing -- no free buffers
  buffer[in] = item;
  in = (in + 1) % BUFFER SIZE;
}
```

%: remainder after division (modulo division)

Bounded Buffer – Consumer

```
while (true) {
  while (in == out);
  // do nothing -- nothing to consume

  // remove an item from the buffer
  item = buffer[out];
  out = (out + 1) % BUFFER SIZE;
  // do something with the item
}
```

Message Passing (1)

- Mechanism for processes to communicate and to synchronize their actions

- Message system – processes communicate with each other without resorting to shared variables

- IPC facility provides two operations (setup – msgget()):
  - send(message) – msgsnd ()
  - receive(message) – msgrcv()

Message Passing (2)

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

- Implementation of communication link
  - Physical (*e.g.*, shared memory, hardware bus, network)
  - Logical (*e.g.*, logical properties)

- Logical properties of a link
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

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 the 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
  - receive(id, message) is possible too – receive message from any process (symmetric / asymmetric addressing)

Indirect Communication (1)

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

Indirect Communication (2)

- Mailbox sharing
  - P1, P2, and P3 share mailbox A
  - P1, sends; P2 and P3 receive
  - Who gets the message?

- Solution
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver
    - Sender is notified who the receiver was

Synchronization

- Message passing may be either blocking or non-blocking (IPC_NOWAIT)
- Blocking is considered synchronous
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send has the sender send the message and continues
  - Non-blocking receive has the receiver receive a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of n messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Examples of IPC Systems – POSIX

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

Examples of IPC Systems – Mach

- Mach communication is message-based
  - Even system calls are messages
  - Each task gets two mailboxes at creation
    - Kernel and Notify
  - Only three system calls needed for message transfer
    - `msg_send()`, `msg_receive()`, `msg_rpc()`
  - Mailboxes needed for communication, created via
    - `port_allocate()`

Shared Memory + Fork

```
#include <unistd.h>
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
```

Shared memory preparation

```
    int pid;
    int segment_id; /* the identifier for the
shared memory segment */
    char *shared_memory; /* a pointer to the
shared memory segment */
    const int segment_size = 4096; /* the size
(in bytes) of the shared memory segment */
```

```
    /** allocate  a shared memory segment */
    segment_id = shmget(IPC_PRIVATE,
segment_size, S_IRUSR | S_IWUSR);

    /** attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id,
NULL, 0);
    //printf("shared memory segment %d attached
at address %p\n", segment_id, shared_memory);
```

Shared memory allocation & attachment

Forking

```
    pid = fork(); /* fork another process */
```

```
    if (pid == 0) { /* child process */
        /** write a message to the shared memory
segment   */
        printf("%s\n", shared_memory);
        sprintf(shared_memory, "Thomas was here");
    } else { /* parent process */
        wait(NULL);

        /** now print out the string from shared
memory */
        printf("*Parent Got: %s*\n",
shared_memory);
```

Writing

```
        /** now detach the shared memory segment
*/
        if (shmdt(shared_memory) == -1) {
            fprintf(stderr, "Unable to detach\n");
        }
        /** now remove the shared memory segment
*/
       shmctl(segment_id, IPC_RMID, NULL);

       return 0;
    }
}
```

Communications in Client-Server Systems

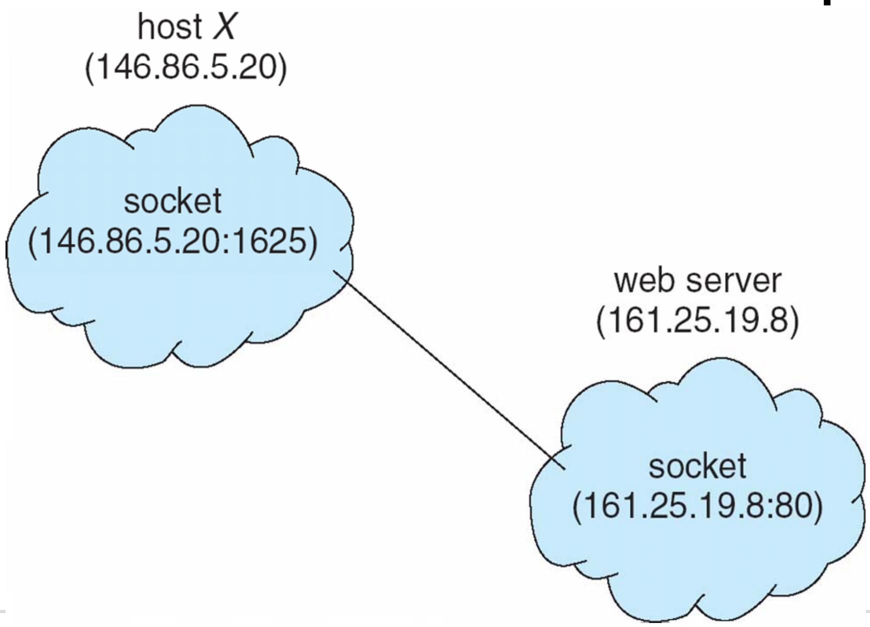
- Sockets

- Remote Procedure Calls
  - Remote Method Invocation (Java)
  - gRPC
    - open source remote procedure call (RPC)
    - Uses HTTP/2, Protocol Buffers
  - Example NFS

- Pipes

Sockets

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

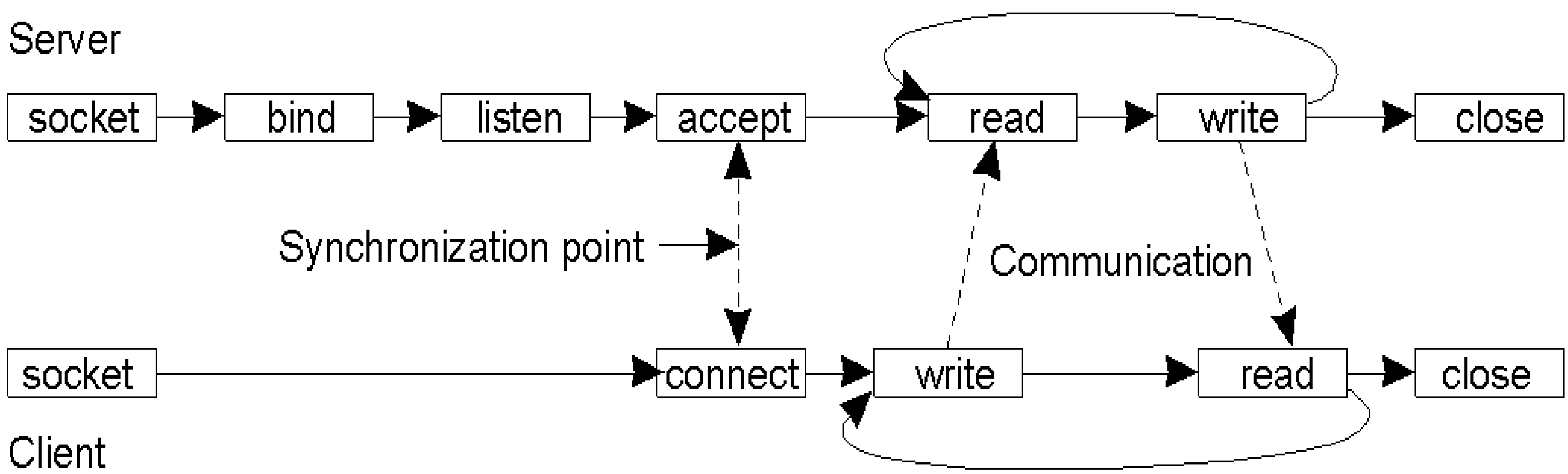


System Calls for TCP/IP Sockets

System call	*Who calls it*	*Meaning*
Socket	Server / Client	Create a new communication endpoint
Bind	Server	Attach a local address and port to a socket
Listen	Server	Define how many clients can be queued
Accept	Server	Block until a connection request arrives
Connect	Client	Actively attempt to establish a connection
Write	Server / Client	Send some data over the connection
Read	Server / Client	Receive some data over the connection
Close	Server / Client	Release the connection

TCP/IP Communication

- Connection-oriented



Java Example

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

public class DateServer
{
    public static void main(String[] args)  {
        try {
            ServerSocket sock = new ServerSocket(10117);

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

                PrintWriter pout = new
PrintWriter(client.getOutputStream(), true);
                // write the Date to the socket
                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);
        }
    }
}
```

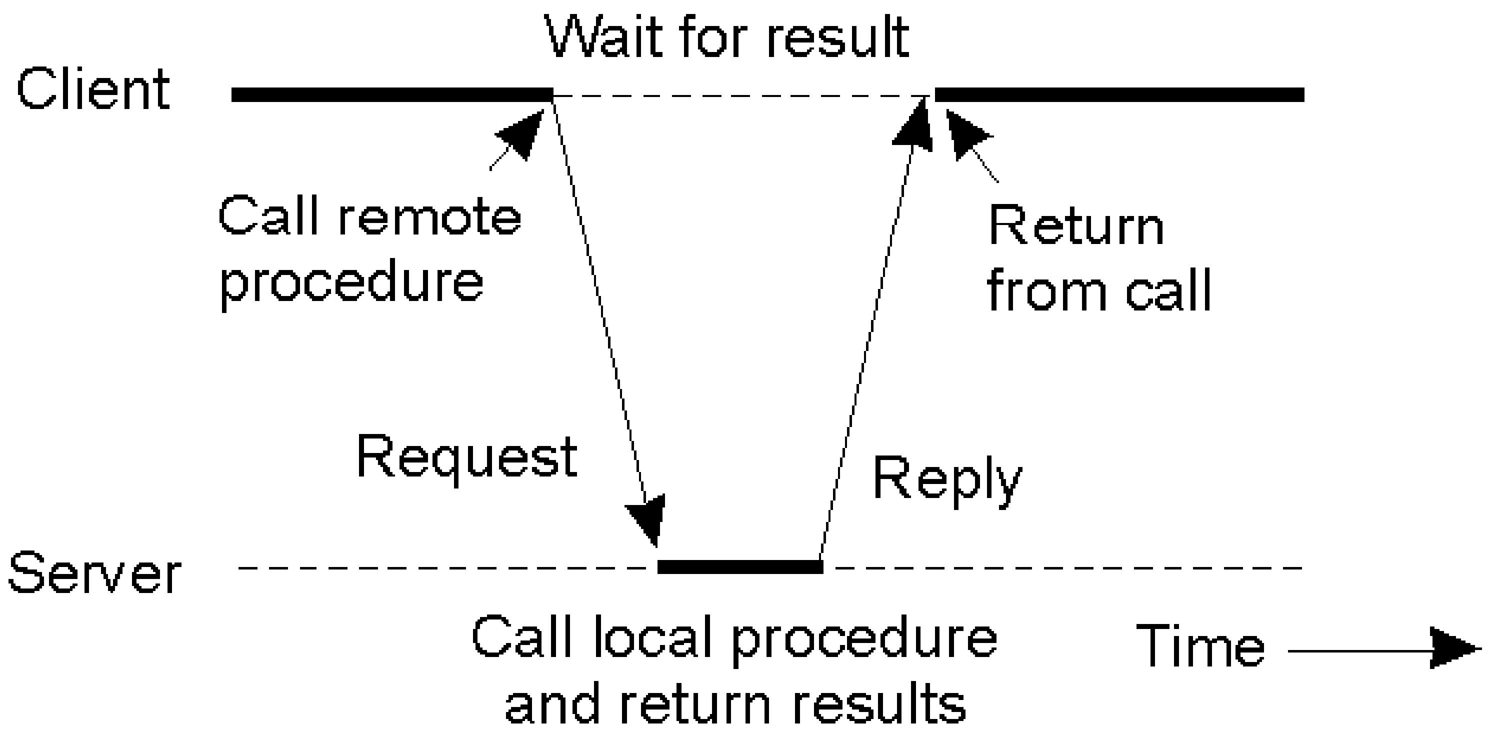
```
import java.net.*;
import java.io.*;

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",10117);
            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);
        }
    }
}
```

Remote Procedure Call (RPC) (1)

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

Remote Procedure Call (RPC) (2)

- RPC between client and server program
- Request-reply communication



TMS, Fig 2.3

Pipes in Practice

- Pipe between two processes

  `ps ax | less` or `ps ax | grep init` or

  `ps ax | grep init | awk {'print $1'}`

- Named pipes

  ```
  mkfifo tompipe
  ls -l > tompipe
  ls -l > tompipe
  ```

- Pipes and network with netcat

  `cat web.sh | nc -l 8080`

  `nc bocek.ch 8080 > web.sh` (unencrypted!!)