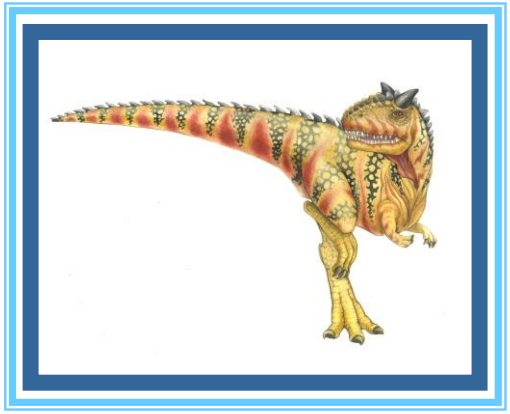
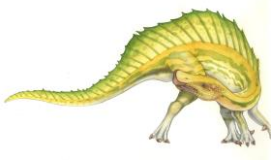
Chapter 3: Processes



Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems



Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

Controlul proceselor

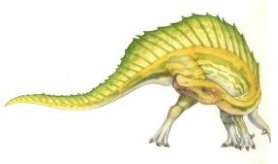
- proces = abstractia rularii unui program (instr + date)
  - in fapt, este programul + starea executiei sale la un moment dat (reflectata de registrele CPU si valorile variabilelor din program)
- multiprogramarea/multitasking-ul (concurenta virtuala/logica sau fizica, pt multiprocesoare) impun nevoia unui model de proces
  - fiecare proces e tratat ca si cand ar avea propriul CPU (abstractia de masina virtuala/extinsa)
- managerul de procese
  - componenta kernel care mentine informatii despre procesele din sistem
  - in principal, se bazeaza pe tabela de procese (contine PCB-uri) si cozi de procese (gestionate de CPU)
- PCB-ul contine: PID, prioritatea procesului, starea, valorile registrelor CPU, utilizarea memoriei, lista de fisiere deschise de proces, etc

Process Concept

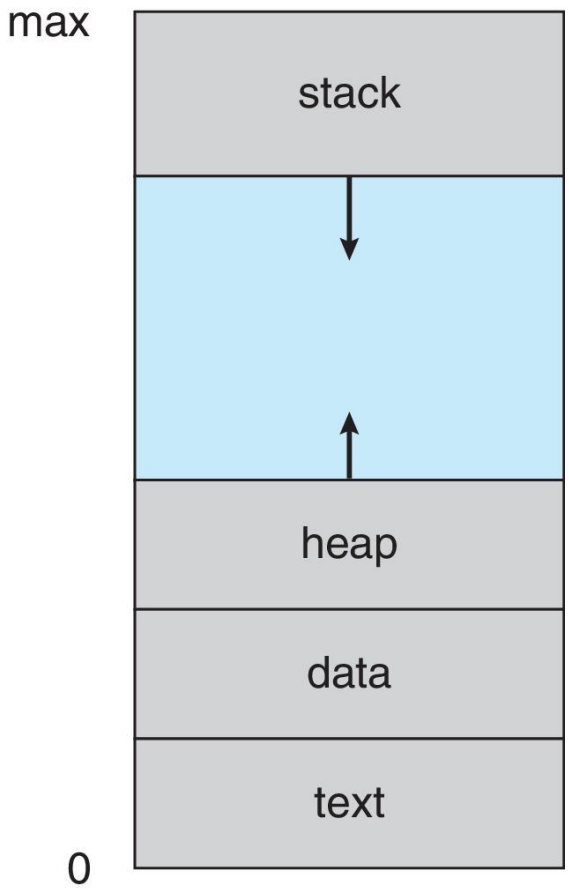
- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

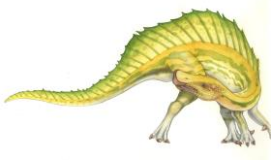
Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program



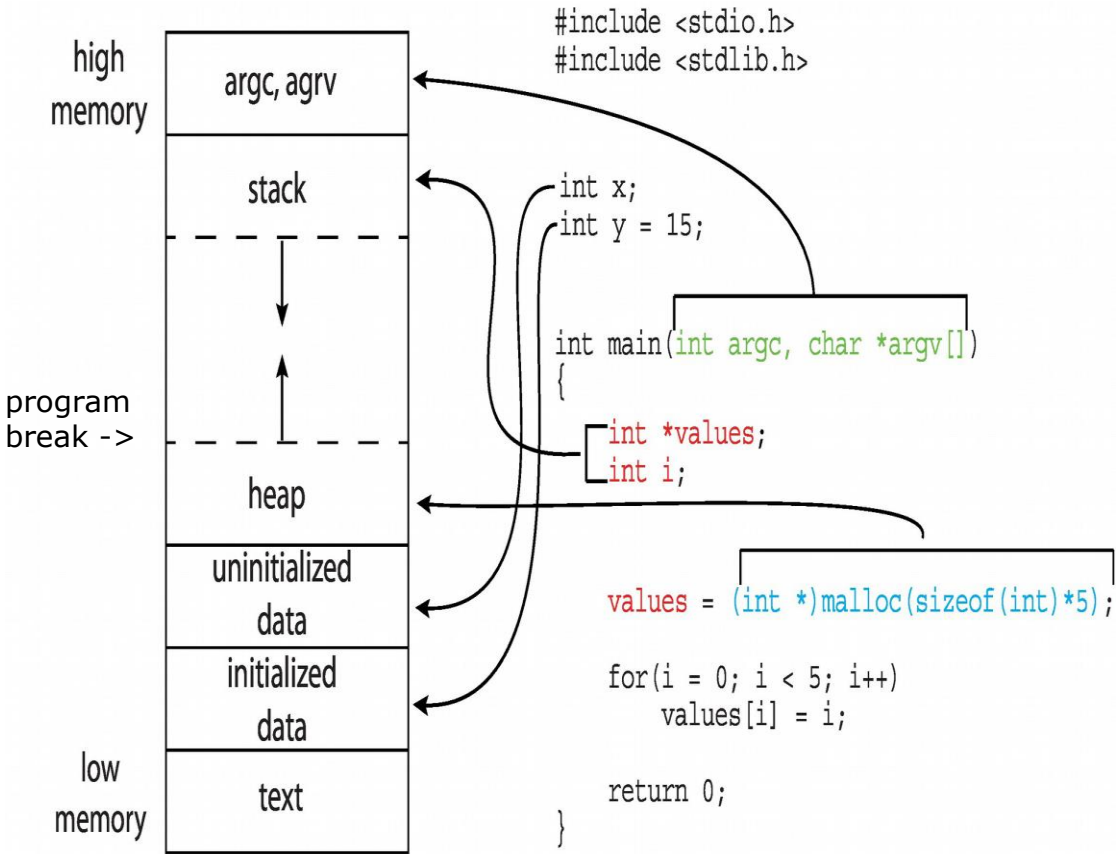
Process in Memory





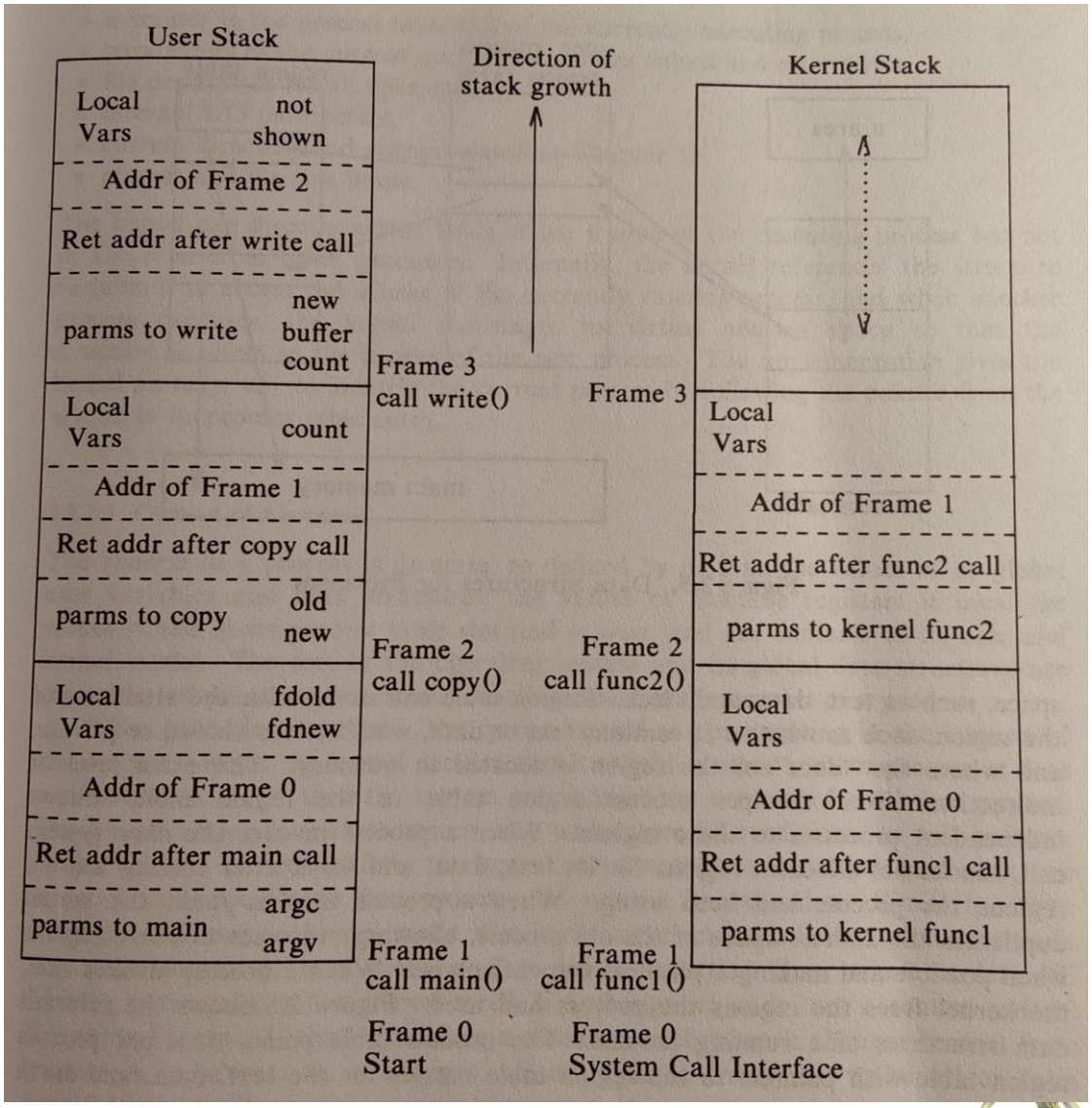
Imaginea unui program C in memorie

- codul si datele initializate sunt citite de *exec* din executabil (fisierul program de pe disc)
- datele neinitializate (bss) sunt initializate cu zero de *exec*
- heap-ul este gestionat ajutorul *program break*-ul (indica sfarsitul segmentului de date)
- valoarea break-ului este modificata prin apelurile de sistem *brk/sbrk*
  - cresterea valorii break-ului ⇔ alocare memorie dinamica
  - scaderea valorii break-ului ⇔ dealocare memorie dinamica
  - *sbrk(0)* intoarce valoarea curenta a program break-ului
  - in practica se evita folosirea *brk/sbrk* si se folosesc *malloc/free*
- stiva program (call stack) este gestionata cu ajutorul inregistrarilor de activare

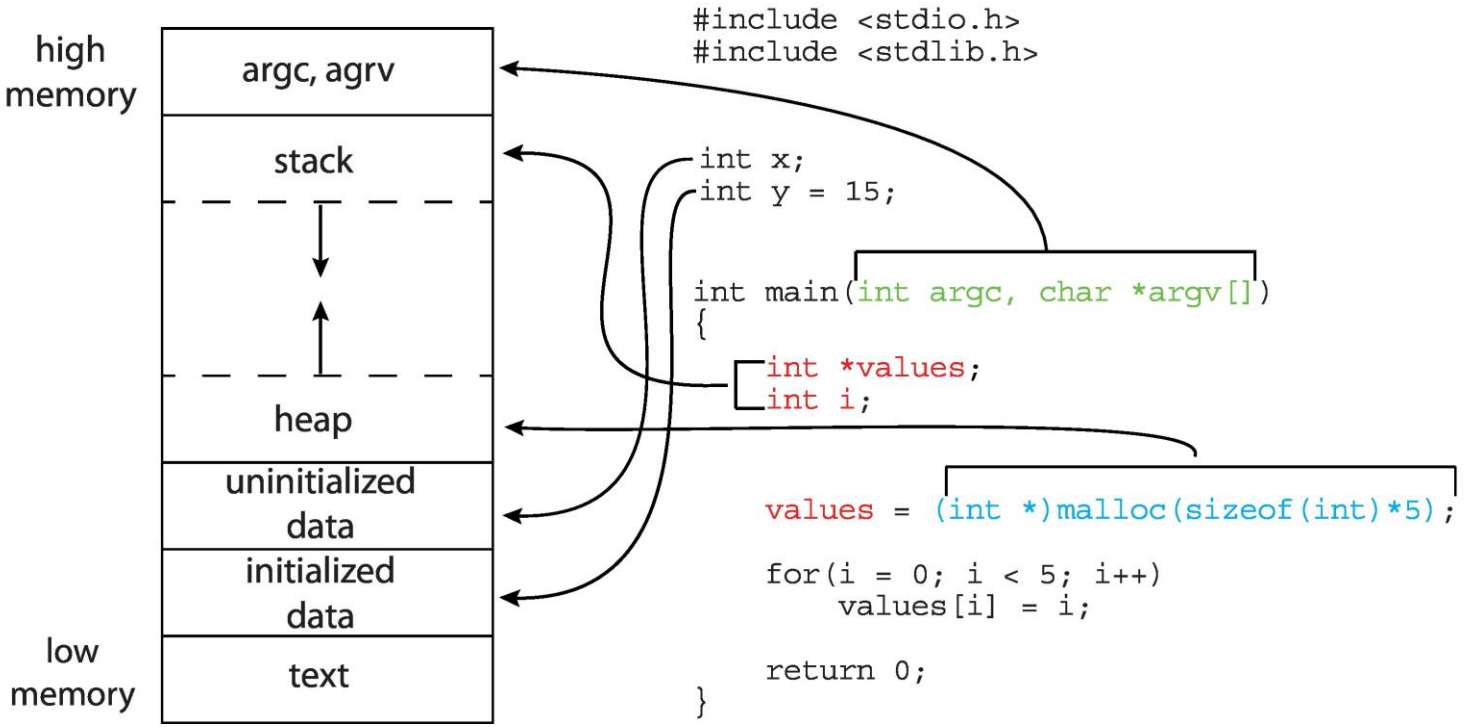


Call stack

```
char buf[4096];

int main(int argc, char* argv[]) {
    int fdold, fdnew;

    if(argc != 3) return 0;

    if((fdold=open(argv[1],
            O_RDONLY)) < 0)
        exit(1);
   if((fdnew=creat(argv[2],0666))
            < 0)
        exit(1);

   copy(fdold, fdnew);
   exit(0);
}

void copy(int old, int new) {
   int count;

   while((count =read(old, buf,
            4096)) > 0)
        write(new, buf, count);
}
```



Memory Layout of a C Program



Process State

- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

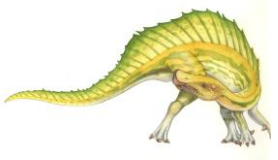
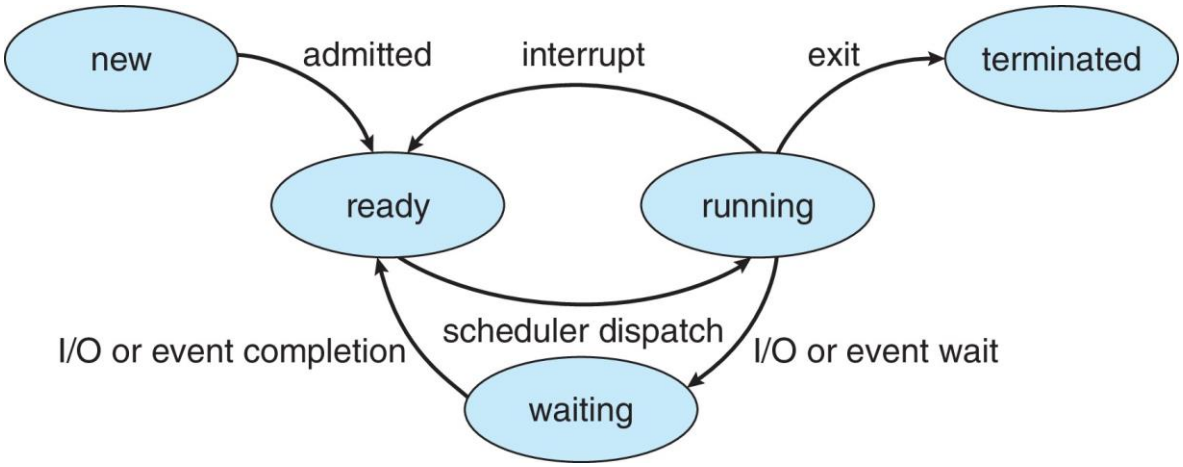
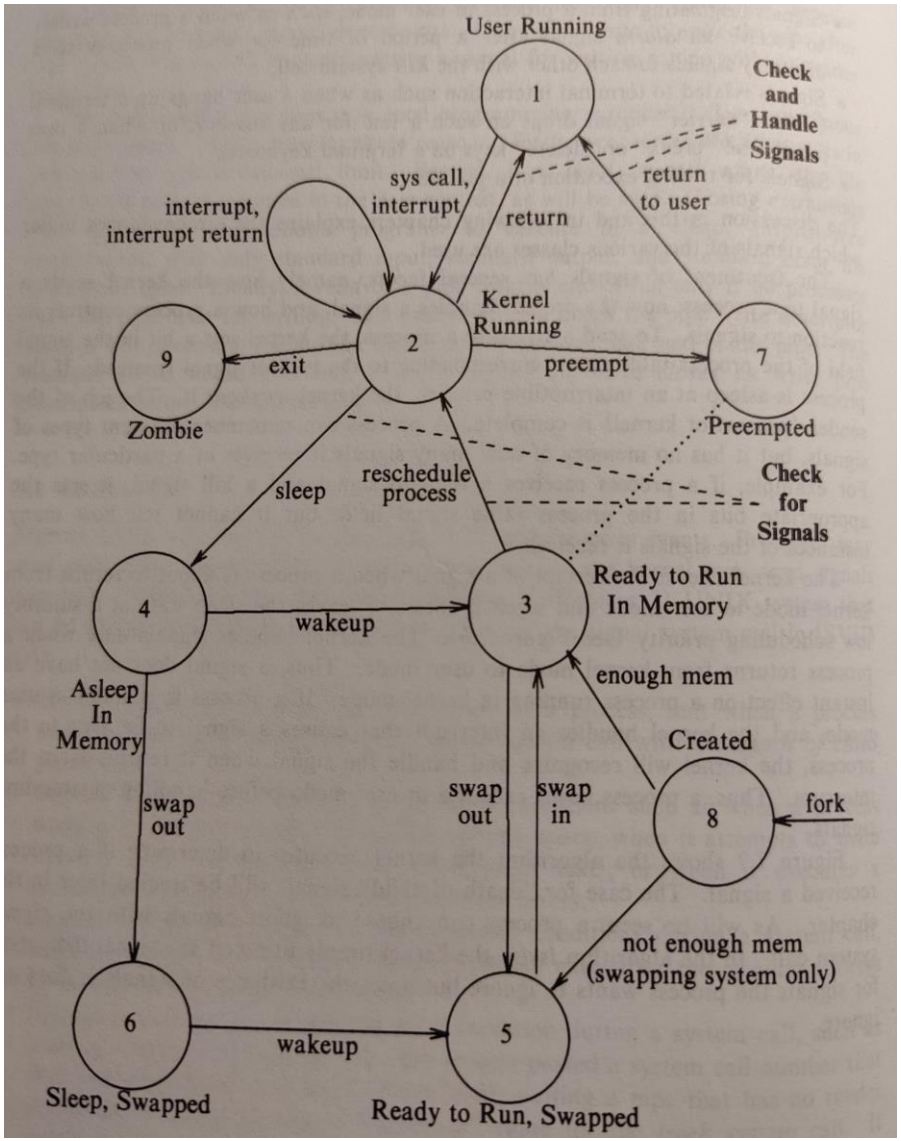


Diagram of Process State



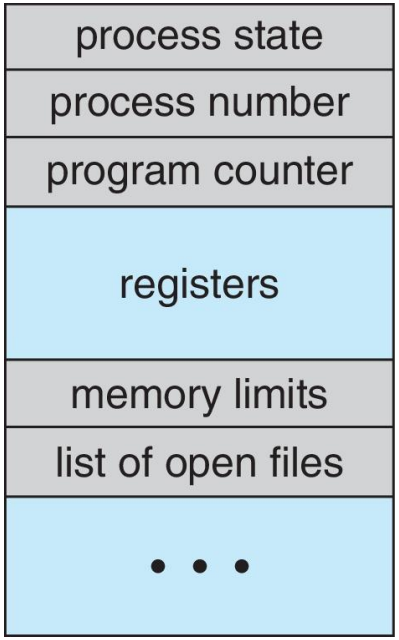
Process State Diagram – Unix example

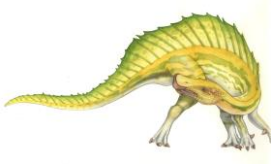


Process Control Block (PCB)

Information associated with each process(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





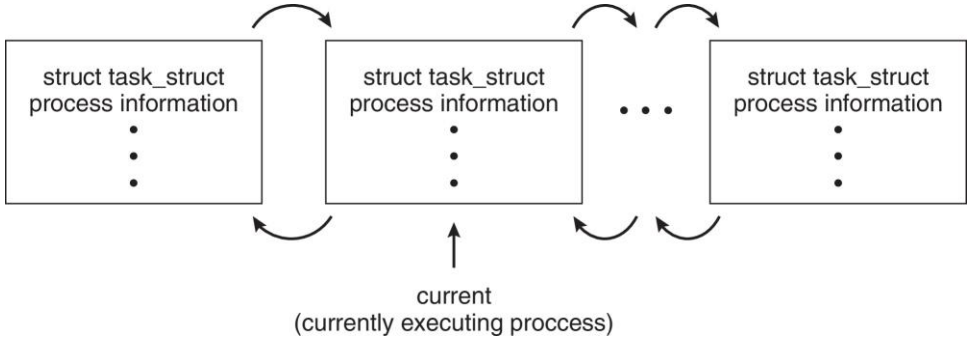
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4

Process Representation in Linux

Represented by the C structure `task_struct`

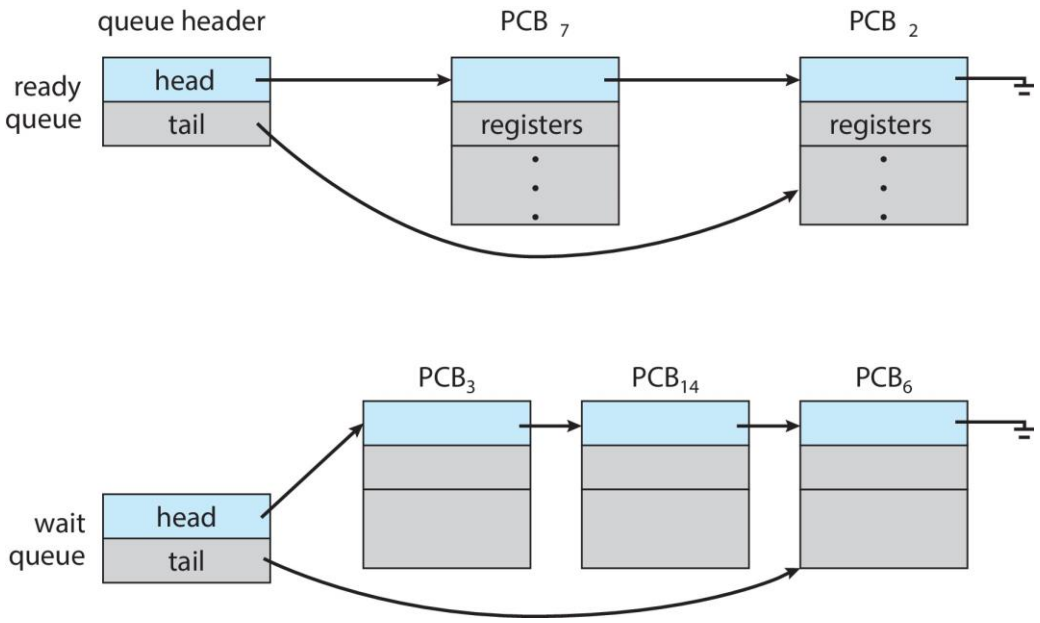
```
pid t_pid;                    /* process identifier */
long state;                   /* state of the process */
unsigned int time_slice       /* scheduling information */
struct task_struct *parent;/* this process's parent */
struct list_head children; /* this process's children */
struct files_struct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



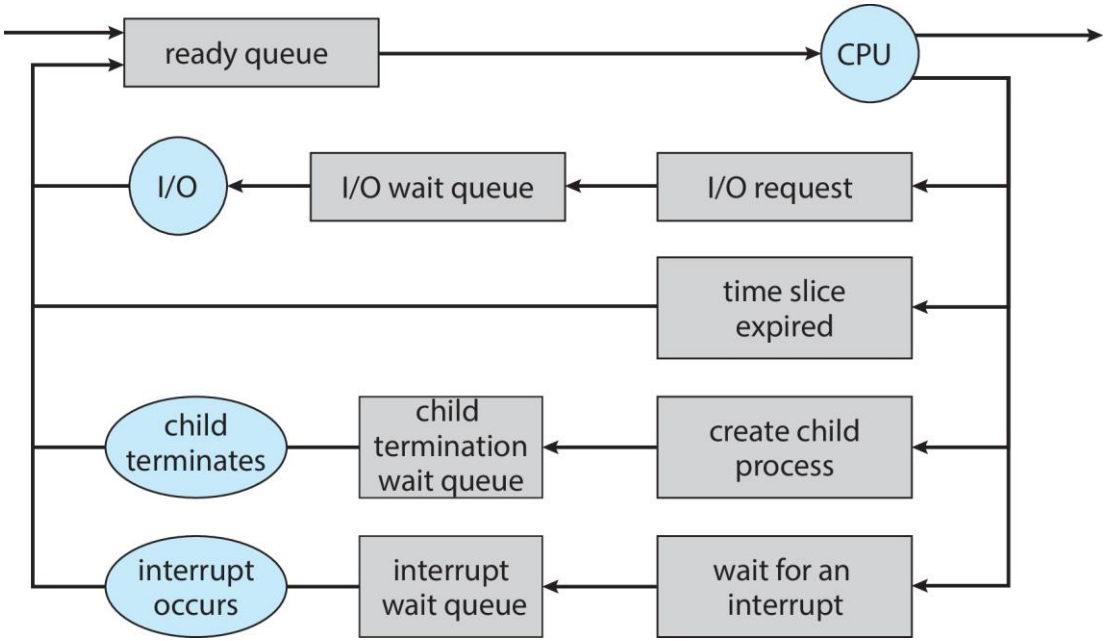
Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto 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

Ready and Wait Queues

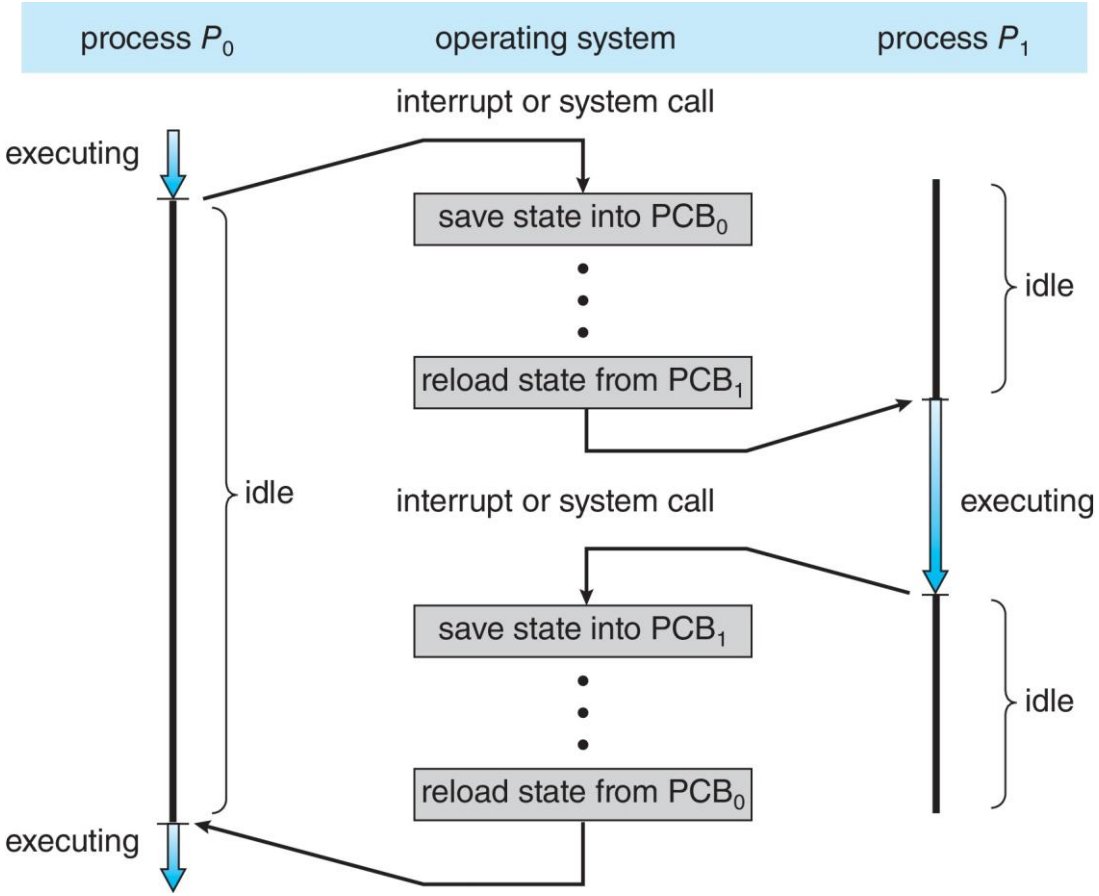


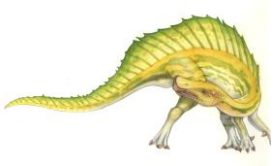
Representation of Process Scheduling



CPU Switch From Process to Process

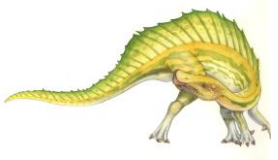
A **context switch** occurs when the CPU switches from one process to another.





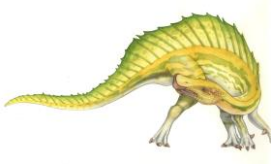
Context Switch

- When CPU switches to another process, the system must **save the state** of the old process 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 pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once



Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

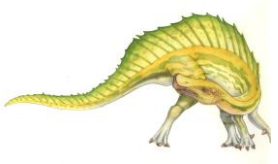


Gestiunea proceselor

- necesita mecanisme de creare si terminare a proceselor (ca si de asteptare a terminarii unui alt proces)
- crearea proceselor
  - model arborescent: un proces parinte creeaza procese copil
  - la momentul crearii, procesul copil este o copie exacta a parintelui (cod + date)
  - implementarea duplicarii tine de optiuni si ratiuni de performanta: procesele pot partaja toate resursele, o parte dintre ele sau nu partajeaza resurse deloc
  - din momentul aparitiei procesului copil nu se mai poate conta in nici un fel pe ordinea executiei parintelui si a copilului: ambele procese se executa concurent sub controlul planificatorului de procese din kernel
  - exista posibilitatea coordonarii explicite intre parinte si copil (procesul parinte “asteapta” terminarea copilului, fie sincron, fie asincron)

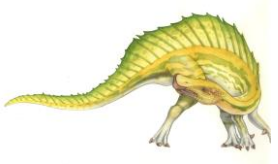
Operations on Processes

- System must provide mechanisms for:
  - Process creation
  - Process termination



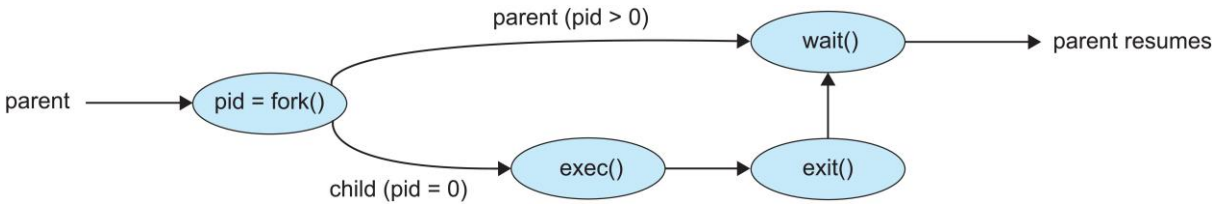
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate



Process Creation (Cont.)

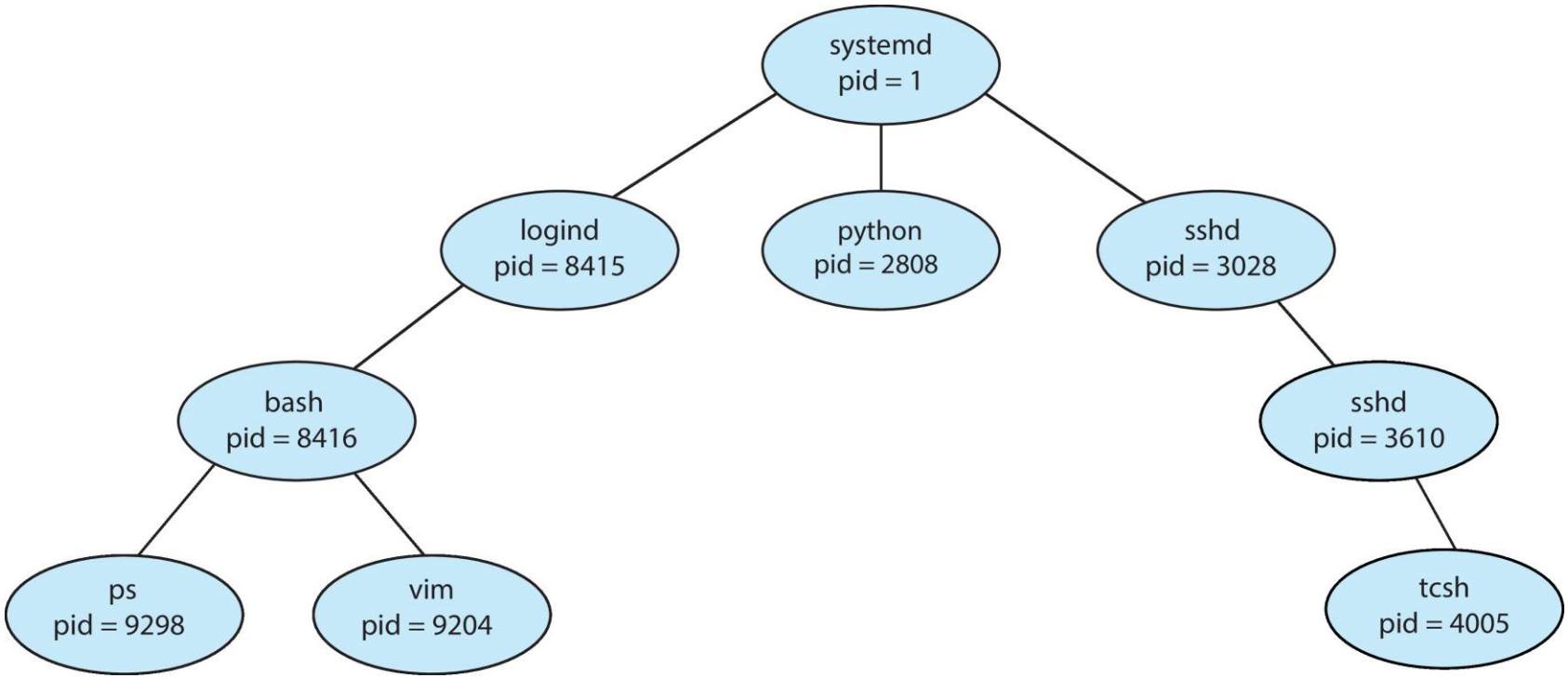
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
  - Parent process calls **`wait()`** waiting for the child to terminate



Executia fork in kernel

- kernelul verifica resursele disponibile
- aloca un PID unic si un PCB in tabela de procese
- verifica informatii de *quota* (de pilda daca utilizatorul nu are prea multe procese deschise)
- marcheaza starea procesului copil ca fiind *creat*
- copiaza PCB-ul parintelui in PCB-ul copilului (in particular, program counter-ul va fi util copilului pt reluarea executiei)
- incrementeaza referintele interne ale kernelului la directorul curent, directorul radacina si fisierele deschise
- copiaza (duplica) contextul de executie al parintelui (cod/text, date, stiva user si kernel)
- modifica in contextul procesului copil informatia necesara pt. ca acesta sa se poata recunoaste pe sine la reintoarcerea din apelul sistem
- apoi, parintele marcheaza starea copilului ca fiind *gata de rulare* si iese din kernel cu cod de retur egal cu noul PID alocat copilului
- copilul se intoarce si el din *fork* cu cod de retur zero

A Tree of Processes in Linux



C Program Forking Separate Process

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

int main()
{
pid_t pid;

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

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 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");
    }

    return 0;
}
```

Creating a Separate Process via Windows API

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Terminarea proceselor

- normal, prin terminarea programului (“iesirea” din functia *main* in C) sau anormal, prin primirea unor semnale (exceptii software)
- dpdv tehnic, prin apel de sistem *exit*
  - intoarce starea procesului care termina in procesul parinte
  - dealoca resursele procesului
  - tratarea specifica a evenimentului se poate face prin handlere asociate terminarii procesului cu ajutorul *atexit* (de ex: tratarea specifica a dealocarii memoriei, a inchiderii fisierelor deschise, etc)
- procesul parinte asteapta terminarea copilului fie sincron, fie asincron cu ajutorul *wait*
  - asincron: cand copilul se termina inainte ca parintele sa cheme *wait* => procesul copil intra in starea de “zombie” (nu mai are PCB, kernelul mentine doar informatie minimala,i.e. codul de iesire, PID, informatie “contabila” despre procesul copil pentru a le putea furniza parintelui can apeleaza *wait*)
- parintele nu apeleaza *wait* si termina inaintea copilului, procesul copil devine orfan
  - in Unix, devine copilul procesului *init*, primul proces pornit de sistem (PID = 1)
  - *init* va executa un *wait* asincron cand primeste SIGCHLD de la kernel la terminarea copilului

Process Termination

- Process executes last statement and then asks the operating system to delete it 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 using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting, and the operating systems does not allow a child to continue if its parent terminates

Process Termination

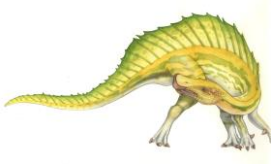
- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc., are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait()`**, process is an **orphan**

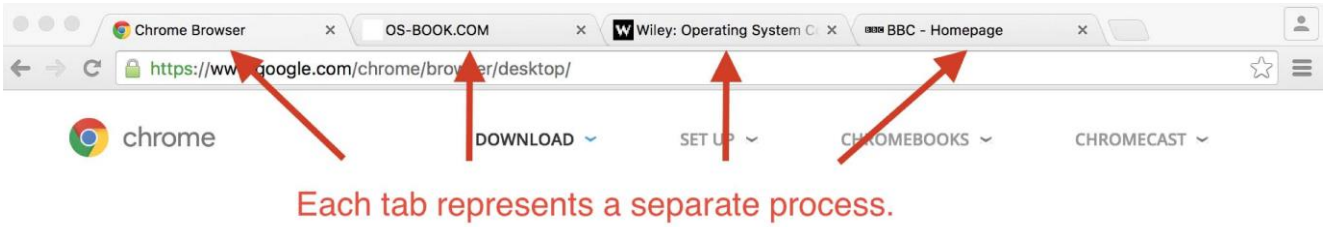
Android Process Importance Hierarchy

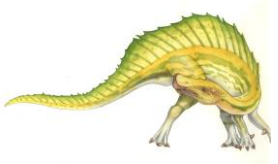
- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most** to **least** important:
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process
- Android will begin terminating processes that are least important.



Multiprocess Architecture – Chrome Browser

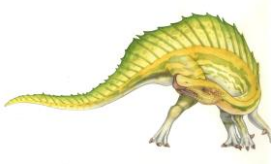
- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders 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

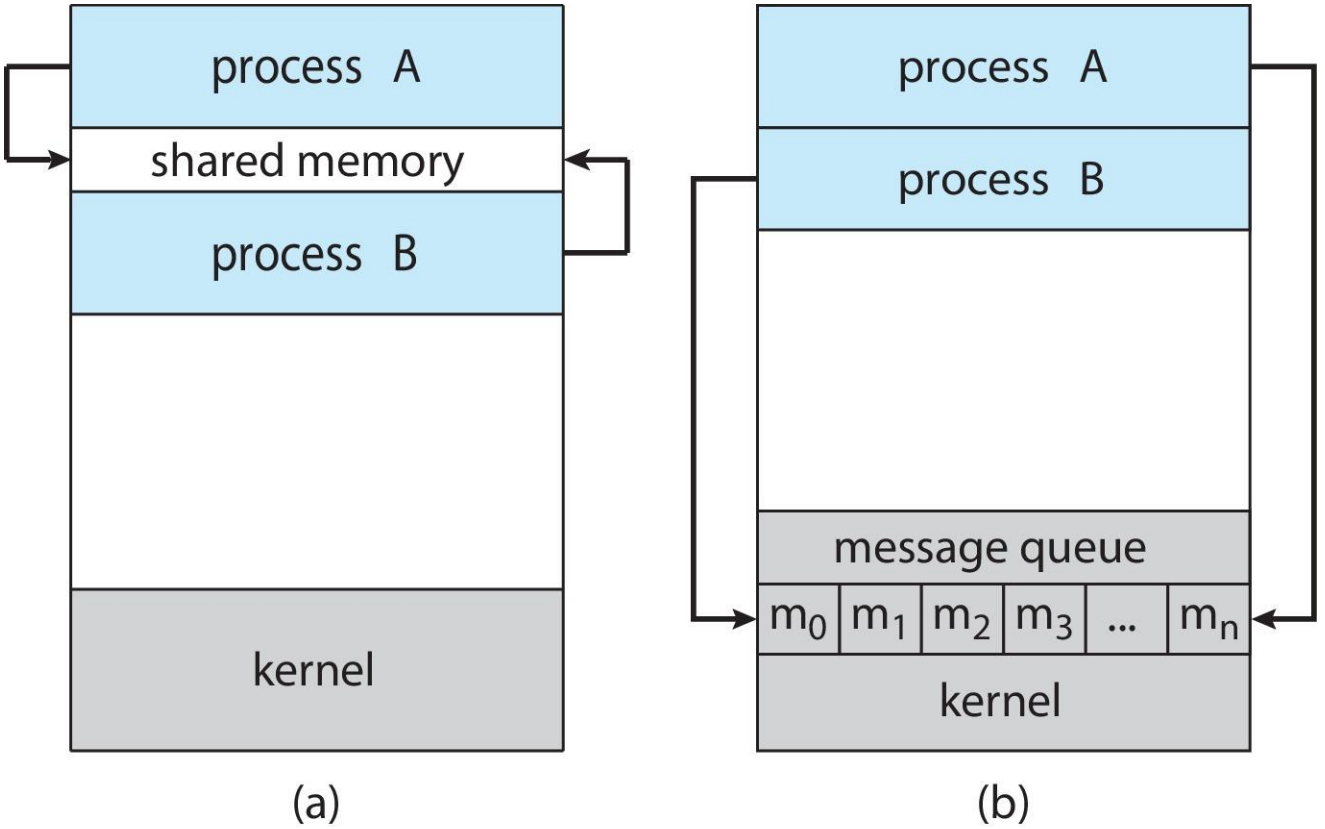
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes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

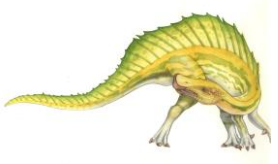


Communications Models

(a) Shared memory.

(b) Message passing.





Producer-Consumer Problem

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume

IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use **`BUFFER_SIZE-1`** elements

Producer Process – Shared Memory

```
item next_produced;

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

Consumer Process – Shared Memory

```
item next_consumed;

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

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

What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **`counter`** that keeps track of the number of full buffers.
- Initially, **`counter`** is set to 0.
- The integer **`counter`** is incremented by the producer after it produces a new buffer.
- The integer **`counter`** is and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next consumed */
}
```

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Race Condition (Cont.)

- Question – why was there no race condition in the first solution (where at most N – 1) buffers can be filled?
- More in Chapter 6.

IPC – Message Passing

- Processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *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 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?

Implementation of Communication Link

- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

Direct Communication

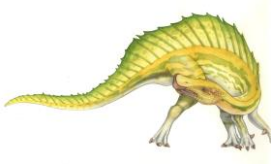
- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication (Cont.)

- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete a mailbox
- Primitives are defined as:
  - **`send`**(*A, message*) – send a message to mailbox A
  - **`receive`**(*A, message*) – receive a message from mailbox A



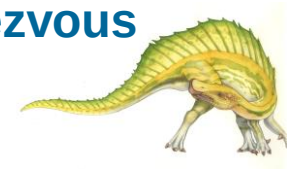
Indirect Communication (Cont.)

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - 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

- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **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 continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**



Producer-Consumer: Message Passing

- Producer

```
message next_produced;
while (true) {
  /* produce an item in next_produced */

  send(next_produced);
}
```

- Consumer

```
message next_consumed;
while (true) {
  receive(next_consumed)

  /* consume the item in next_consumed */
}
```

Buffering

- Queue of messages attached to the link.
- 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

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    `shm_fd = shm_open(name, O CREAT | O RDWR, 0666);`
  - Also used to open an existing segment
  - Set the size of the object

    `ftruncate(shm_fd, 4096);`
  - Use **`mmap()`** to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by **`mmap()`**.

IPC POSIX Producer

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

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

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

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

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

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

IPC POSIX Consumer

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

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

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

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two ports at creation - Kernel and Notify
  - Messages are sent and received using the **`mach_msg()`** function
  - Ports needed for communication, created via

    `mach_port_allocate()`
  - Send and receive are flexible; for example four options if mailbox full:
    - Wait indefinitely
    - Wait at most n milliseconds
    - Return immediately
    - Temporarily cache a message

Mach Messages

```
#include<mach/mach.h>

struct message {
        mach_msg_header_t header;
        int data;
};

mach port t client;
mach port t server;
```

Mach Message Passing - Client

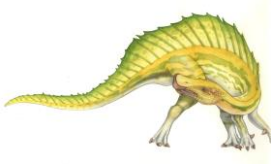
```
/* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
   MACH_SEND_MSG, // sending a message
   sizeof(message), // size of message sent
   0, // maximum size of received message - unnecessary
   MACH_PORT_NULL, // name of receive port - unnecessary
   MACH_MSG_TIMEOUT_NONE, // no time outs
   MACH_PORT_NULL // no notify port
);
```

Mach Message Passing - Server

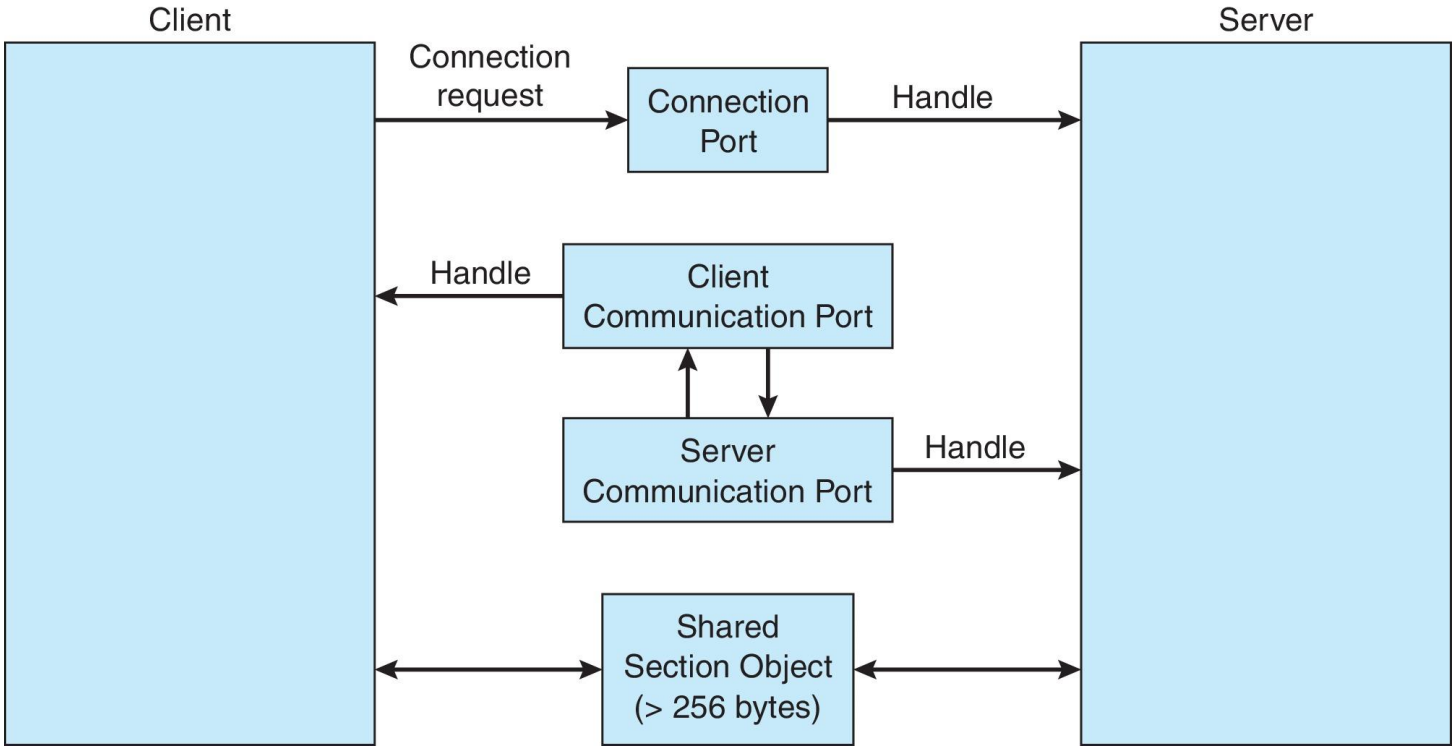
```
/* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
   MACH_RCV_MSG, // sending a message
   0, // size of message sent
   sizeof(message), // maximum size of received message
   server, // name of receive port
   MACH_MSG_TIMEOUT_NONE, // no time outs
   MACH_PORT_NULL // no notify port
);
```

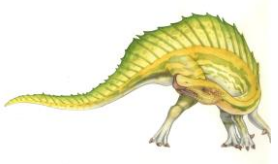
Examples of IPC Systems – Windows

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



Local Procedure Calls in Windows



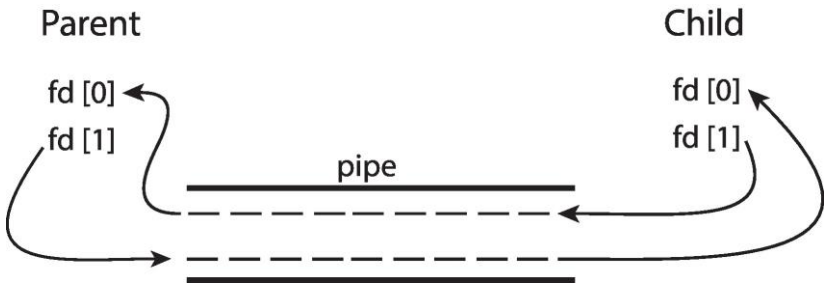


Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary 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
- Require parent-child relationship between communicating processes



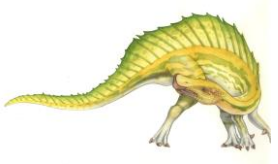
- Windows calls these **anonymous pipes**

Named Pipes

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

Communications in Client-Server Systems

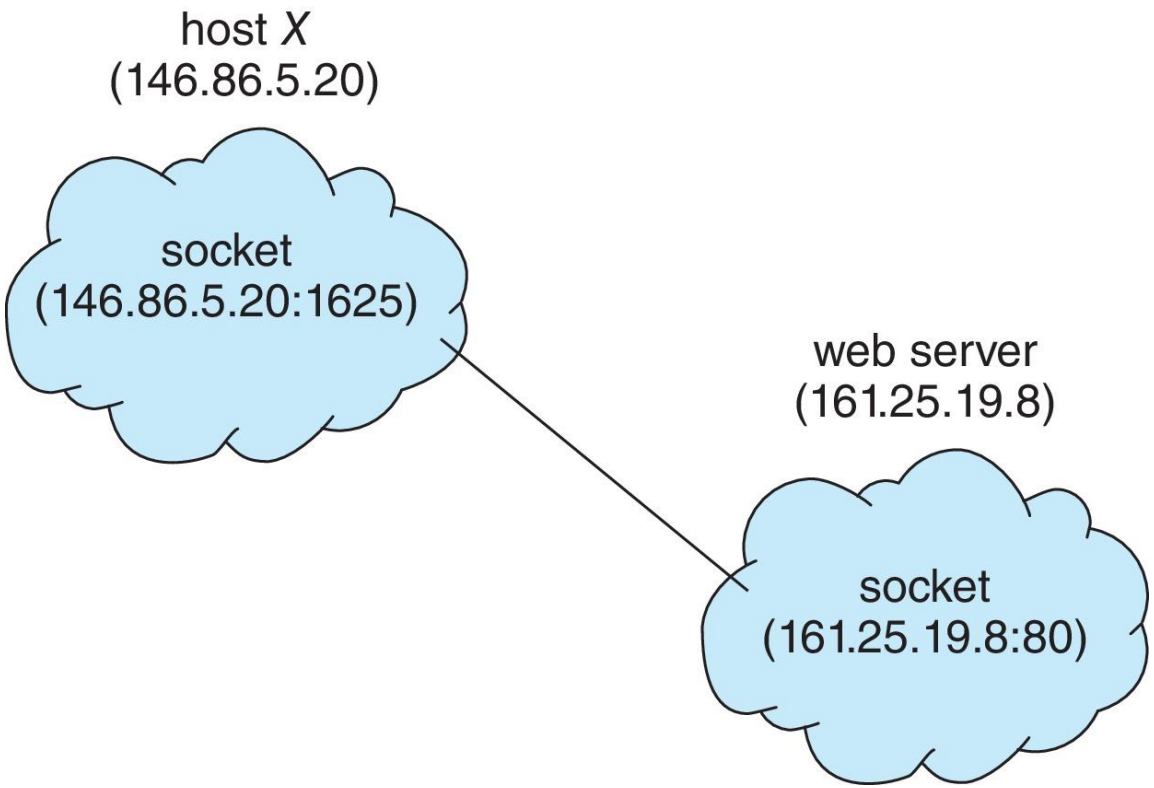
- Sockets
- Remote Procedure Calls



Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- 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

Socket Communication



Sockets in Java

- Three types of sockets
  - **Connection-oriented** (**TCP**)
  - **Connectionless** (**UDP**)
  - **`MulticastSocket`** class– data can be sent to multiple recipients
- Consider this “Date” server in Java:

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

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

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            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 connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Sockets in Java

The equivalent Date client

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

public class DateClient
{
   public static void main(String[] args) {
      try {
         /* make connection to server socket */
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         /* read the date from the socket */
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         /* close the socket connection*/
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Dezavantaje paradigma message passing

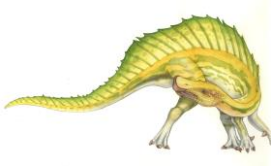
- modelul de programare uzual pt sisteme centralizate nu functioneaza pt sisteme distribuite (nu exista reprezentarea resurselor distribuite
  - ideal ar fi ca modelul uzual de programare sa functioneze si pt sisteme distribuite
  - pt sisteme distribuite trebuie modificate programele obisnuite prin luarea in consideratie a comunicatiei (problema de I/O)
- apar dificultati de programare
  - un mesaj e doar un sir de biti, programele opereaza cu tipuri de date complexe
  - e nevoie de modalitati de pastrare a tipurilor de date la transmisia datelor peste retea
  - in plus, arhitecturile de calcul conectate in retea pot fi eterogene, cu reprezentare diferita a tipurilor de date

    ex: intregi reprezentati in format little (Intel) respective big (Sparc) endian

Remote Procedure Calls (RPC)

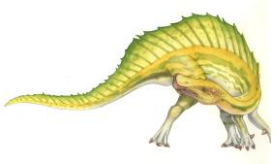
- Birell & Nelson, 1984
- incearca sa transceada limitelor masinii locale apeland proceduri aflate pe masini la distanta
- avantaj: model de programare centralizat, nu exista diferente intre apelurile locale si cele distante
- natura distribuita a sistemului impiedica totusi o identificare a celor doua tipuri de apeluri
  - procedura apelanta si cea apelata nu se afla in acelasi spatiu de adresa

  => pasarea parametrilor si intoarcerea rezultatului pot fi complicate
  - defectarea (“crash”) oricareia dintre cele doua masini fizice implicate creeaza probleme suplimentare, deopotriva tehnice si de semantica a apelului
- totusi modelul e popular pe arhitecturi NORMA
  - Java RMI, Web services



Functionarea RPC

- programul client se linkediteaza cu asa-numite rutine “stub” care vor fi invocate in locul procedurii propriu-zise aflate pe masina de la distanta
- invocarea stub-ului client determina urmatorul sir de actiuni:
  - (1) stub-ul converteste argumentele procedurii intr-un format independent de masina si le impacheteaza intr-un mesaj (“marshalling”)
  - (2) mesajul este livrat protocolului de transport din kernel (eg, TCP/UDP)
  - (3) protocolul de transport determina transmisia mesajului la distanta catre o rutina stub din server care despacheteaza argumentele procedurii (si identificatorul ei) si le converteste la reprezentarea masinii locale (“unmarshalling”)
  - (4) apoi, stub-ul server apeleaza procedura propriu-zisa

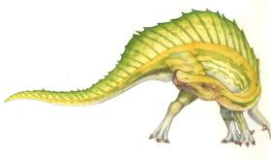


Functionarea RPC (cont.)

- programul client se linkediteaza cu asa-numite rutine “stub” care vor fi invocate in locul procedurii propriu-zise aflate pe masina de la distanta
- invocarea stub-ului client determina urmatorul sir de actiuni:

….

  - (5) rezultatele procedurii sunt supuse procedurii de marshalling intr-un mesaj de raspuns
  - (6) mesajul de raspuns e trimis inapoi clientului
  - (7) stub-ul client despacheteaza mesajul de raspuns si converteste datele la formatul local
  - (8) stub-ul client intoarce valoarea rezultatului RPC procedurii apelante



Transmiterea parametrilor in RPC

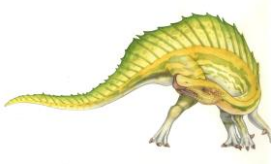
- call by value
  - se paseaza valoarea argumentului pe stiva
  - argumentul se poate modifica local in procedura, dar modificarile se pierd la return
  - in principiu, potrivit pt RPC
- call by reference
  - doar un pointer (adresa) catre date e pasat pe stiva
  - modificarile datelor facute prin indirectare supravietuiesc apelului (se vad la nivelul apelantului)
  - nu are sens pt RPC unde clientul si serverul nu partajeaza spatiile de adresa
- copy/restore
  - varianta asimilabila call by value
  - call by value dar la retur se copiaza inapoi valoarea argumentului suprascriindu-se valoarea apelantului
  - folosit in RPC

Reprezentarea datelor

- pana si tipurile de baza pot avea reprezentari diferite pe masini diferite (v. intregi)
- pt a functiona in medii eterogene parametrii se convertesc la o reprezentare standard folosita de stub-uri
  - ex: XDR (eXternal Data Representation), ASN.1 (Abstract Syntax Notation)
- in modelul OSI sunt transformari de nivel de prezentare a datelor (nivelul 6, imediat inainte de nivelul de aplicatie)
  - tot aici se pot opera transformari ale datelor de tip criptare (inclusiv in RPC)
- situatii clasice
  - intregi reprezentati little/big endian
  - string-uri codificate ASCII/EBCDIC/Unicode
- Obs: la intregi transformarea e simpla (se inverseaza bytes), dar la string-uri problema e mai complicata => nevoia de format standard
- optimizare: intre masini cu arhitecturi compatibile, convertirea e inutila, se recurge la notificarea tipului de format in mesaj

Protocolul de transport

- initial era prevazut un protocol dedicat
- UDP
  - fara overhead de setare a conexiunii
  - clientul trebuie sa trateze retransmisia pachetelor, erorile de transmisie, secventierea pachetelor
  - totusi, e protocolul preferat pe LAN unde pierderea de pachete si erorile sunt rare
  - oricum, mai putin performant decat un protocol dedicat (care n-ar face asamblarea/dezasamblarea fragmentelor, diverse checksum-uri, etc)
- TCP – potrivit pt WAN, unde tratarea automata a retransmisiei, erorilor si a secventierii mesajelor asigurata de TCP e un avantaj
- dimensiunea pachetului de retea folosita
  - overhead-ul RPC e fix, indiferent de marimea datelor trimise
  - pt performanta amortizata buna, se prefera transmisia mai multor date intr-un singur mesaj (aici conteaza si mediul de transmisie, v. Ethernet standard vs jumbo frames)

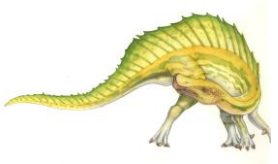


Semantica RPC

- diferita de semantica apelurilor locale
  - (1) se pot pierde mesaje
  - (2) se pot duplica mesaje
  - (3) serverul si/sau clientul se pot defecta/opri
- (1) & (2) se rezolva prin retransmisie si folosire nr de secventa
- operatii idempotente: efectul executiei multiple este acelasi cu cel al unei singure executii
- ex. operatii idempotente: setarea unui bit, citirea primului bloc de date dintr-un fisier
- ex. operatii care nu sunt idempotente: incrementarea unui intreg, adaugarea datelor la sfarsitul unui fisier (operatia de append)

Semantica RPC (cont.)

- *exactly once*
  - procedura de la distanta se executa o singura data
  - greu de asigurat
  - mesaje duplicate, rezolvate prin retransmisie si nr de secventa
  - server crash: rezolvare dificila, serverul poate crapa inainte de servirea cererii, dupa servirea ei dar inainte de transmisia unui raspuns sau dupa transmisia unui raspuns care se pierde
- *at most once*
  - procedura remote fie nu se executa deloc fie cel mult o data
  - in absenta unui raspuns de la server nu e clar daca operatia s-a executat sau nu
- *at least once*
  - procedura remote se executa cel putin o data (posibil de mai multe ori)
  - semantica potrivita pentru operatii idempotente
  - clientul retransmite cererea pana cand primeste ACK



Defectarea clientului, solutii

- *exterminare*
  - fiecare apel RPC se inregistreaza pe disc intr-un log
  - dupa reboot, daca vin raspunsuri de la server se ignora (“calculul e exterminat”)
- *reincarnare*
  - foloseste conceptul de epoca
  - la reboot clientul anunta prin broadcast o noua epoca tuturor serverelor, care in consecinta termina calculele aferente tuturor epocilor trecute
  - daca apar totusi raspunsuri din epoci trecute, clientul le ignora
- *reincarnare delicata*
  - varianta in care serverele notificate la inceputul unei noi epoci termina calculul doar daca nu pot identifica proprietarul ei

Defectarea clientului, solutii (cont.)

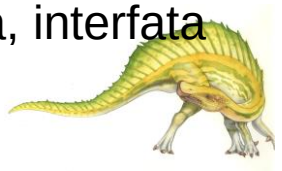
- *expirare*
  - fiecare RPC trebuie sa termine intr-un interval fix de timp T
  - dezavantaj: daca nu se poate termina in timp util, trebuie ceruta o noua perioada
  - Avantaj: daca dupa crash clientul asteapta o perioada T, atunci stie sigur ca nici un calcul nu mai e active
- Obs: nicio solutie nu e satisfacatoare (ex: dc. procedura remote care se termina fortat detine locks, resursele respective protejate de locks raman blocate definitive)

Binding

- procesul prin care clientul determina adresa serverului (host si port)
- (1) director de servicii global
  - serverele isi inregistreaza adresele si tipul de serviciu pe care il ofera in director
  - clientii cauta in director adresa unui server care ofera serviciul dorit
- (2) clientul stie adresa serverului
  - serverul ruleaza propriul director de servicii in care programele server isi inregistreaza serviciile
  - ex: SUN portmapper
- clientii folosesc porturi publice (“well-kown ports”) pentru a adresa cereri directorului de servicii

Suport RPC la nivelul limbajului

- RPC e o constructie de nivelul limbajelor de programare
- compilatorul trebuie informat ca procedura se va executa la distanta si ca apelurile se fac prin stub
- avand in vedere diferenta de semantica fata de apelurile obisnuite, se prefera ca apelurile RPC sa fie marcate explicit la nivel de limbaj prin keywords de tipul “remoteproc”
- solutia alternativa: Interface Definition Language
  - in loc sa se extinda compilatorul cu suport RPC, se defineste punctul de interactiune intre client si server ca fiind o *interfata*
  - interfata e definite intr-un limbaj special de definire a interfetei IDL
  - specificarea IDL a unei interfete se compileaza cu un compilator special care genereaza stub-uri client si server (ex: Sun rpcgen)
  - diferite compilatoare IDL pot fi folosite pt a genera strub-uri pt limbaje si conventii de apelare diferite
  - alternativ, exista limbaje care suporta nativ conceptul de interfata, interfata RPC nefiind decat un caz particular (ex: Java RMI)



Exemplu de implementare RPC

- Sun RPC are propriul IDL si un compilator specific (*rpcgen*) numit si “generator de stub-uri”
- fiecare masina server ruleaza un director de servicii numit *portmapper*
  - mapeaza identificatorul programului server pe un port de comunicatie
  - programul server e identificat prin nr de program si versiune
- exemplu de specificare a interfetei pt un serviciu simplu de fisiere
  - Interfata suporta operatia SFS_READ (operatia cu nr 1) care primeste un argument de tip *readargs* si intoarce un rezultat de tip *readres*
  - programul server are numarul 123321 si versiunea 9876
- interfata e definita in fisierul sfs.x iar stub-urile se genereaza cu comanda “rpcgen sfs.x”:
  - sfs_client.c – stub client
  - sfs_server.c – stub server
  - sfs_xdr.c – rutinele de marshalling/unmarshalling
  - sfs.h – interfata in limbaj C

Interfata IDL server de fisiere

```
 1 const SFS_MAXDATA = 4096;
 2
 3 struct readres {
 4   int count;
 5   opaque data<SFS_MAXDATA>;
 6 };
 7
 8 struct readargs {
 9   string file<SFS_MAXDATA>;
10   unsigned offset;
11   unsigned count;
12 };
13
14 program SFS_PROGRAM {
15   version SFS_VERSION {
16     readres SFS_READ(readargs) = 1;
17   } = 9876;
18 } = 123321;
```

Figure 2: Simple File Server interface definition file `sfs.x`.

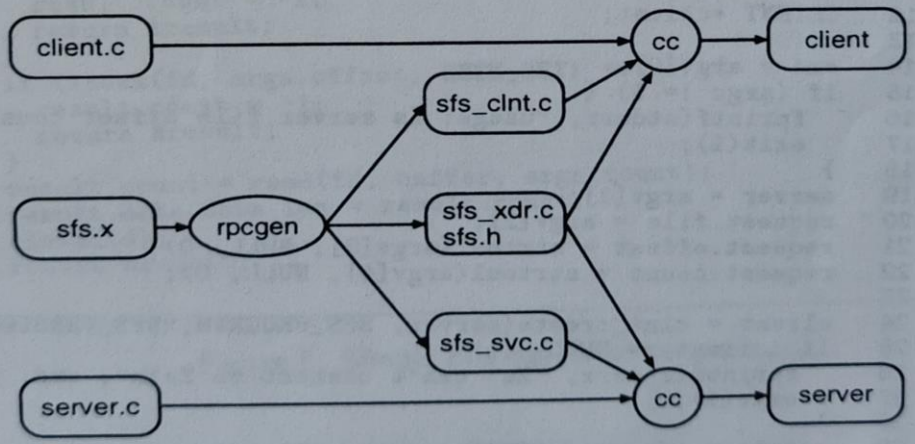


Figure 3: `rpcgen` operation.

Codul RPC client

- sfs_client.c
  - construieste o cerere
  - creaza un handle folosit pt. a executa RPC-uri pe server
  - *clnt_create* contacteaza portmapper-ul de pe server, cere portul corespuzatori perechii (SFS_PROGRAM,SFS_VERSION) si creeaza un socket UDP pt comunicare cu serverul
  - apelul RPC = apelul *sfs_read_9786*
    - foloseste o structura pt ca initial rpcgen-ul nu accepta decat proceduri cu un singur argument
    - versiunile moderne accepta mai multi parametri

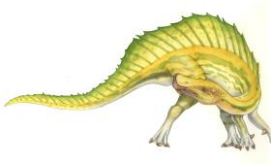
Cod RPC client

```
 1 #include <rpc.h>
 2 #include "sfs.h"
 3
 4 char *cmd;
 5
 6 int
 7 main(int argc, char *argv[])
 8 {
 9   struct readargs request;
10   struct readres *result;
11   char *server;
12   CLIENT *client;
13
14   cmd = argv[0];
15   if (argc != 5) {
16     fprintf(stderr, "usage: %s server file offset count\n", cmd);
17     exit(1);
18   }
19   server = argv[1];
20   request.file = argv[2];
21   request.offset = strtoul(argv[3], NULL, 0);
22   request.count = strtoul(argv[4], NULL, 0);
23
24   client = clnt_create(server, SFS_PROGRAM, SFS_VERSION, "udp");
25   if (client == NULL) {
26     fprintf(stderr, "%s: can't connect to %s\n", cmd, server);
27     exit(1);
28   }
29   result = sfs_read_9876(request, client);
30   if (result == NULL) {
31     fprintf(stderr, "%s: read failed\n", cmd);
32     exit(1);
33   }
34   write(1, result->data.data_val, result->data.data_len);
35   exit(0);
36 }
```

Figure 4: Simple File Server client code.

Codul RPC server

- sfs_server.c
  - apeleaza procedura *sfs_read_9786_svc*
  - aceasta deschide fisierul solicitat
  - muta file pointerul la offsetul solicitat
  - citeste nr de octeti solicitat
  - returneaza rezultatul



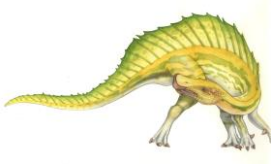
Cod RPC server

```
 1 #include <stdio.h>
 2 #include <fcntl.h>
 3 #include <rpc.h>
 4 #include "sfs.h"
 5
 6 readres *
 7 sfs_read_9876_svc(readargs args, struct svc_req *request)
 8 {
 9   static readres result;
10   static char buffer[SFS_MAXDATA];
11   int count;
12   int fd;
13
14   result.count = 0;
15   result.data.data_len = 0;
16   result.data.data_val = buffer;
17
18   if ((fd = open(args.file, O_RDONLY)) == -1) {
19     result.count = -1;
20     return &result;
21   }
22   if (lseek(fd, args.offset, SEEK_SET) == -1) {
23     result.count = -1;
24     return &result;
25   }
26   result.count = read(fd, buffer, args.count);
27   result.data.data_len = result.count;
28   close(fd);
29   return &result;
30 }
```

Figure 5: Simple File Server server code.

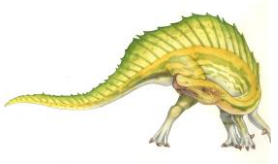
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **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
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

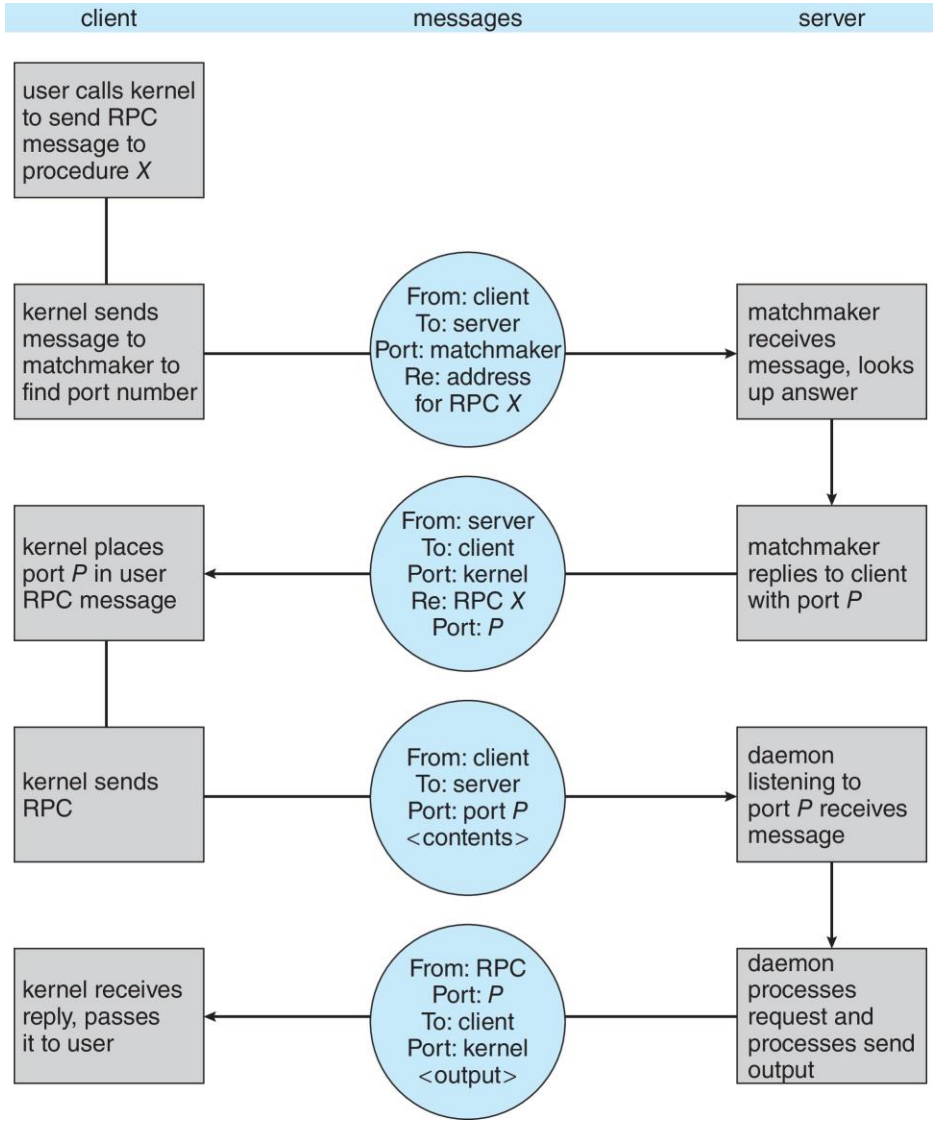


Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server



Execution of RPC



End of Chapter 3