Processi

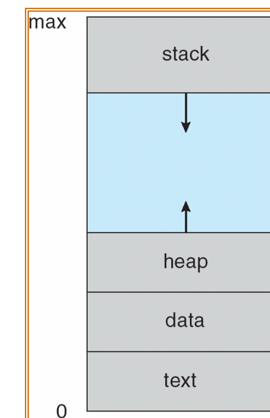
Processi

- Concetto di processo
- Scheduling dei processi
- Operazioni sui processi
- Processi cooperanti
- Comunicazione tra processi
- Comunicazione in sistemi Client-Server

Concetto di processo

- Un SO esegue una varietà di programmi:
  - Sistemi batch – job
  - Sistemi time-sharing – programmi utenti o task
- Il libro di testo usa i termini *job* e *processo* in modo intercambiabile
- Processo – un *programma in esecuzione*; l' esecuzione di un processo avviene in *modo sequenziale*
- Un processo include:
  - program counter
  - contenuto registri CPU
  - sezione testo
  - sezione dati
  - heap
  - stack

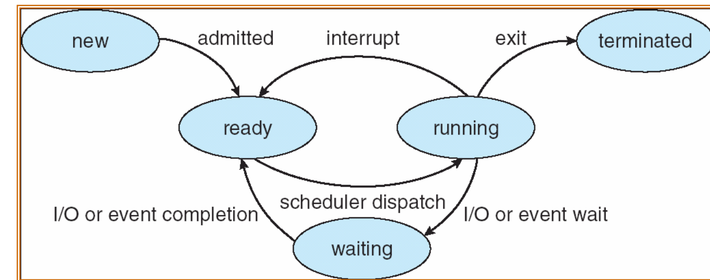
Immagine in memoria di un processo



Stati di un processo

- Durante l'esecuzione un processo cambia stato
  - **new**: Il processo è stato creato
  - **running**: Le sue istruzioni vengono eseguite
  - **waiting**: Il processo è in attesa di qualche evento
  - **ready**: Il processo è in attesa di essere assegnato ad un processore
  - **terminated**: Il processo ha terminato l'esecuzione

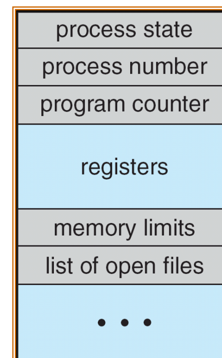
Diagramma di Stato dei Processi



Process Control Block (PCB)

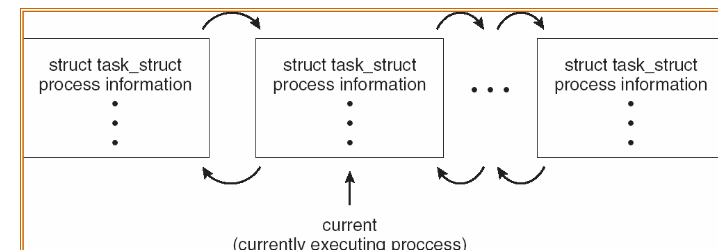
Contiene informazioni per la gestione del processo

- Stato del processo
- Program counter
- Registri della CPU
- Informazioni per lo scheduling CPU
- Informazioni per la gestione della Memoria
- Informazioni di contabilizzazione
- Informazioni sullo stato dell' I/O

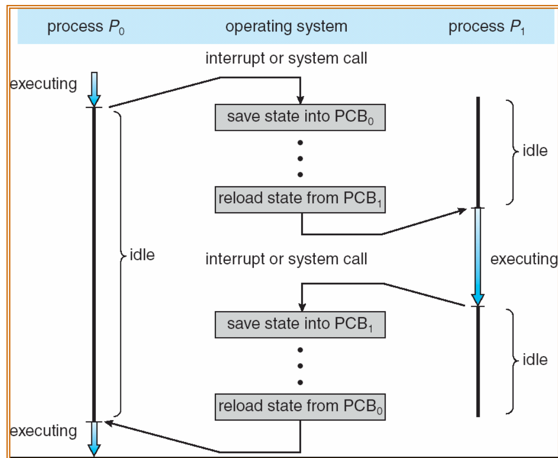


Rappresentazione di un processo in Linux

```
typedef task_struct {

        pid_t pid;                  /* process identifier
        long state;                 /* state
        unsigned int time_slice;    /* scheduling info
        struct files_struct *files; /* open files
        struct mm_struct *mm;       /* address space
        …
        }
```



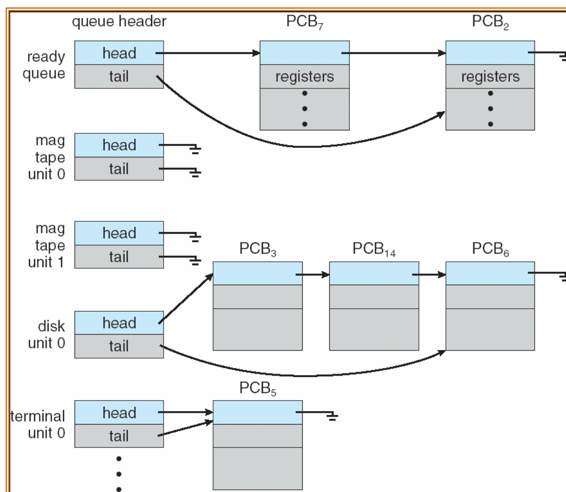
La CPU commuta da processo a processo



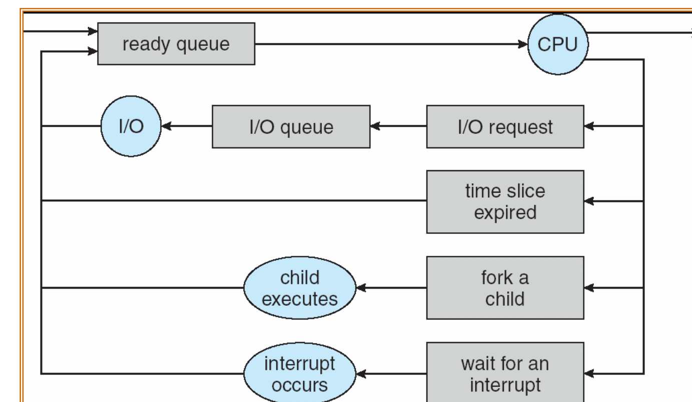
Code di scheduling per i processi

- **Job queue** – insieme di tutti i processi nel sistema
- **Ready queue** – insieme di tutti i processi in memoria centrale pronti per l'esecuzione
- **Code dei dispositivi** – insieme dei processi in attesa di qualche dispositivo di I/O
- I processi, durante la loro vita, *migrano tra varie code*

Ready Queue e varie code di I/O



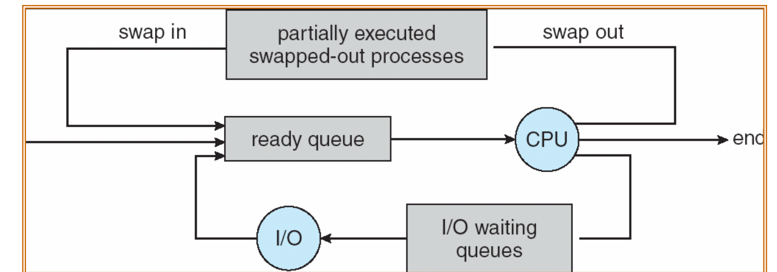
Ciclo di vita di un processo



Schedulatori

- **Schedulatore a lungo termine** (o job scheduler) – seleziona i processi che devono essere caricati in memoria centrale (ready queue)
- **Schedulatore a breve termine** (o CPU scheduler) – seleziona il prossimo processo che la CPU dovrebbe eseguire

Schedulatore a Medio Termine



Schedulatori

- Lo schedulatore a breve termine viene invocato molto *frequentemente* (millisecondi) ⇒ deve essere veloce
- Lo schedulatore a lungo termine viene invocato *raramente* (secondi, minuti) ⇒ può essere più lento
- Lo schedulatore a lungo termine controlla il *grado di multiprogrammazione*
- I processi possono essere descritti come:
  - **Processi I/O-bound** – consumano più tempo facendo I/O che computazione, contengono molti e brevi CPU burst
  - **Processi CPU-bound** – consumano più tempo facendo computazione; contengono pochi e lunghi CPU burst

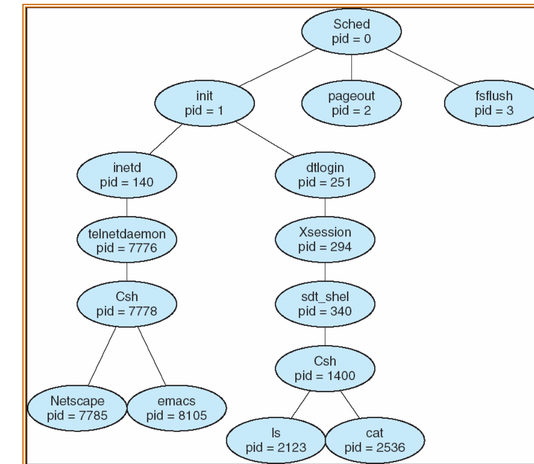
Cambio di contesto (Context Switch)

- Quando la CPU viene allocata ad un altro processo, il SO deve salvare lo stato del processo in esecuzione e caricare lo stato del nuovo processo
- Il tempo per un context switch è tempo sprecato (overhead); il sistema non fa nulla di utile mentre commuta
- Il tempo dipende dal supporto hardware
- Il context switch viene realizzato dal **dispatcher**

Creazione di processi

- Il sistema operativo fornisce meccanismi per creare e terminare processi
- Un processo padre crea processi figli che, a loro volta, creano altri processi, formando un albero di processi
- Ogni processo ha un identificatore (process indentifier) PID

Un albero di processi di Solaris



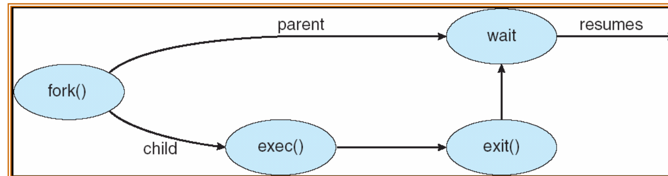
Creazione di processi: implementazione

- Condivisione di risorse
  - Padre e figli condividono tutte le risorse
  - I figli condividono un sottoinsieme delle risorse del padre
  - Padre e figlio non condividono niente
- Esecuzione
  - Padre e figli vengono eseguiti in concorrenza
  - Padre aspetta fino alla terminazione dei figli

Creazione di processi: implementazione

- Spazio di indirizzamento
  - Lo spazio del figlio è un duplicato di quello del padre
  - Il figlio carica in esso un nuovo programma
- Esempi UNIX
  - La chiamata **fork** crea un nuovo processo
  - La chiamata **exec** viene usata dopo una **fork** per sostituire l'immagine in memoria del processo con un nuovo programma
  - La chiamate **wait** permette al padre di aspettare che il figlio termini

Creazione di un processo



Programma C che crea un processo figlio – Unix

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

int main()
{
pid_t pid;

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

        if (pid < 0) { /* error occurred */
                fprintf(stderr, "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 */
                /* parent will wait for the child to complete */
                printf("I am the parent %d\n",pid);
                wait(NULL);

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

Programma C che crea un processo figlio – Windows

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

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

   ZeroMemory( &si, sizeof(si) );
   si.cb = sizeof(si);
   ZeroMemory( &pi, sizeof(pi) );

   // Start the child process.
   if( !CreateProcess( NULL,   // No module name (use command line).
      "C:\\WINDOWS\\system32\\mspaint.exe", // Command line.
      NULL,         // Process handle not inheritable.
      NULL,         // Thread handle not inheritable.
      FALSE,        // Set handle inheritance to FALSE.
      0,         // No creation flags.
      NULL,         // Use parent's environment block.
      NULL,         // Use parent's starting directory.
      &si,         // Pointer to STARTUPINFO structure.
      &pi )         // Pointer to PROCESS_INFORMATION structure.
   )
   {
      printf( "CreateProcess failed (%d).\n", GetLastError() );
      return -1;
   }

   // Wait until child process exits.
   WaitForSingleObject( pi.hProcess, INFINITE );

   // Close process and thread handles.
   CloseHandle( pi.hProcess );
   CloseHandle( pi.hThread );
}
```

Terminazione di Processi

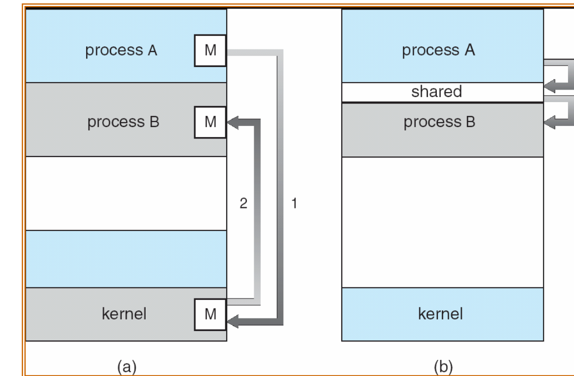
- Un processo esegue l'ultima istruzione e chiede al SO di eliminarlo (**exit**)
  - Trasmette un valore di output al padre (via **wait**)
  - Le risorse del processo sono recuperate dal SO
- Il processo padre può terminare l'esecuzione dei processi figli (**abort**)
  - Un figlio ha ecceduto nell'uso delle risorse
  - Il compito assegnato al figlio non è più richiesto
  - Se il padre ha terminato
    - Alcuni SO non permettono ai processi figli di continuare
    - Tutti i figli vengono terminati – terminazione a *cascata*

Processi cooperanti

- **Processi independenti** non possono influenzare o essere influenzati dall'esecuzione di altri processi
- **Processi cooperanti** possono essere influenzati dall'esecuzione di altri processi
- Vantaggi della cooperazione tra processi
  - Condivisione di informazione
  - Accelerazione delle computazioni
  - Modularità
  - Convenienza

Interprocess Communication (IPC)

Meccanismi per la comunicazione e la sincronizzazione tra processi



Problema Produttore-Consumatore

- Nel paradigma dei processi cooperanti, un processo *produttore* produce informazioni che vengono consumate da un processo *consumatore*
  - *Buffer-illimitato* non pone limiti alla taglia del buffer
  - *Buffer-limitato* assume che la taglia del buffer sia fissata

Buffer-limitato – Soluzione con memoria condivisa

- Dati condivisi

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

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

- La solutione è corretta, ma può usare soltanto BUFFER_SIZE-1 elementi

Produttore

```
item nextProduced;

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

Consumatore

```
item nextConsumed

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

        // remove an item from the buffer
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER SIZE;
       // consume the item in nextConsumed
    }
```

Scambio di messaggi

- Sistema di messaggi – I processi comunicano tra loro senza ricorrere a variabili condivise
- IPC fornisce due operazioni:
  - **send**(*message*) – message taglia fissa o variabile
  - **receive**(*message*)
- Se *P* e *Q* desiderano comunicare, hanno bisogno di:
  - Stabilire un *canale di comunicazione* tra loro
  - Scambiare messaggi via send/receive
- L' implementazione di un canale di comunicazione
  - fisica (e.g., memoria, hardware bus)
  - logica (e.g., proprietà logiche)

Questioni implementative

- Come sono stabilite le connessioni?
- Si può associare un canale a più di due processi?
- Quanti canali ci possono essere tra ogni coppia di processi comunicanti?
- Qual è la capacità di un canale?
- La taglia dei messaggi che un canale può trasmettere è fissa o variabile?
- Il canale è unidirezionale o bidirezionale?

Comunicazione diretta

- I processi debbono usare le proprie identità esplicitamente:
  - **send** (*P, message*) – manda un messaggio al processo P
  - **receive**(*Q, message*) – riceve un messaggio dal processo Q
- Proprietà di un canale di comunicazione
  - I canali sono stabiliti automaticamente
  - Un canale viene associato esattamente tra una coppia di processi comunicanti
  - Tra ciascuna coppie esiste esattamente un canale
  - Il canale può essere unidirezionale ma solitamente è bidirezionale

Comunicazione indiretta

- I messaggi sono inviati e ricevuti attraverso **mailbox** (anche chiamate porte)
  - Ciascuna mailbox ha un unico id
  - I processi possono comunicare solo se condividono una mailbox
- Proprietà di un canale di comunicazione
  - Un canale viene stabilito solo se i processi condividono una mailbox comune
  - Un canale può essere associato a molti processi
  - Ogni coppia di processi può condividere molti canali di comunicazione
  - I canali possono essere unidirezionali o bidirezionali

Communicazione indiretta

- Primitive definite come segue:

  send(*A, message*) – invia un messaggio alla mailbox *A*

  receive(*A, message*) – ricevi un messaggio dalla mailbox *A*

- Operazioni
  - crea una nuova mailbox
  - invia e ricevi messaggi attraverso la mailbox
  - distruggi la mailbox

Communicazione indiretta

- Problemi nella condivisione di una mailbox
  - P_1, P_2, e P_3 condividono la mailbox *A*
  - P_1, invia; P_2 e P_3 ricevono
  - Chi ottiene il messaggio?
- Soluzioni
  - Permettere che un canale possa essere associato al più a due processi
  - Permettere ad un processo alla volta soltanto di eseguire una operazione di ricezione
  - Permettere al sistema di selezionare arbitrariamente il ricevente. Il sender riceve una notifica circa l'identità del ricevente.

Sincronizzazione

- Lo scambio di messaggi può essere bloccante o non bloccante
- **Bloccante** è considerato **sincrono**
  - **Send bloccante** blocca il sender fino alla ricezione del messaggio
  - **Receive bloccante** blocca il receiver fino a quando il messaggio non diventa disponibile
- **Non-bloccante** è considerato **asincrono**
  - **Send non-bloccante:** il sender invia il messaggio e continua
  - **Receive non-bloccante: il** receiver ottiene o un messaggio valido oppure null

Buffering

- Coda di messaggi associata al canale di comunicazione; implementata in tre possibili modi

1. Capacità zero– 0 messaggi in coda
   Il sender deve attendere la ricezione (rendezvous)

2. Capacità limitata – lunghezza finita di *n* messaggi
   Il sender deve attendere se il buffer è pieno

3. Capacità illimitata – lunghezza infinita
   Il sender non aspetta mai

Memoria condivisa – API Posix

- Un processo deve prima creare un segmento di memoria condivisa

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR)
```

- I processi che vogliono usare il segmento lo debbono includere nel proprio spazio di indirizzamento

```
shared_memory = (char *) shmat(id,NULL,0);
```

- Una volta incluso, si può accedere al segmento

```
sprintf(“shared_memory, “writing to shared memory”);
```

Memoria condivisa – API Posix

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

int main()
{
        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);

        sprintf(shared_memory, "Hi there!"); /** write a message to the shared memory segment   */

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

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

        shmctl(segment_id, IPC_RMID, NULL); /** now remove the shared memory segment */

        return 0;
}
```

Scambio di messaggi – Mach

Mach è un esempio di SO basato su scambio di messaggi

I messaggi vengono inviati e ricevuti attraverso mailbox (**porte**)

- **msg_send()**
- **msg_receive()**
- **msg_rpc()**

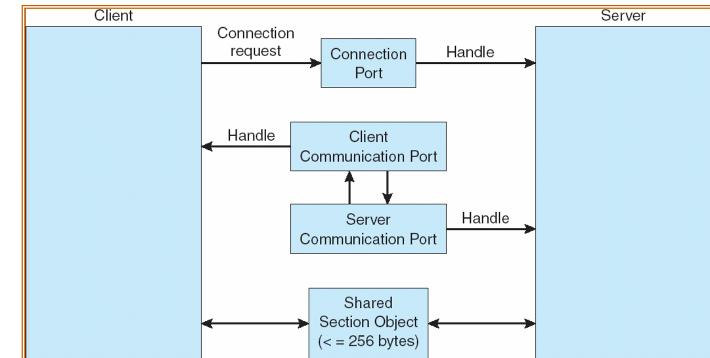
La system call **port_allocate()** crea una nuova porta e alloca spazio per la coda di messaggi (8 al più). I messaggi hanno un header di lunghezza fissa e una parte variabile

Un solo task può ricevere ad una porta – ma il diritto può essere ceduto

Un task può creare un **set di mailbox**

port_status()

Local Procedure Call – Windows XP



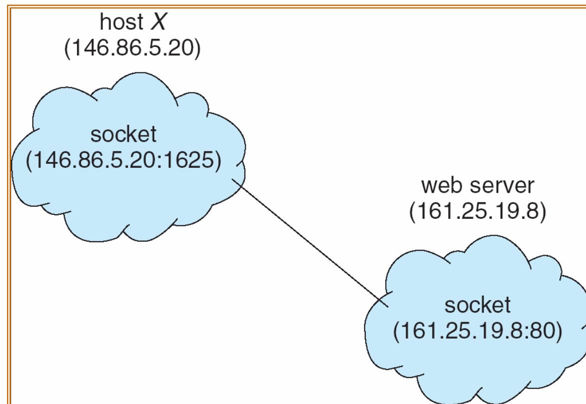
Comunicazioni Client-Server

- Socket
- Remote Procedure Call
- Remote Method Invocation (Java)

Socket

- Un socket è un'estremità di un canale di comunicazione
- Concatenazione di un indirizzo IP e di una porta
- Il socket **161.25.19.8:1625** fa riferimento alla porta **1625** sull'host **161.25.19.8**
- Un canale di comunicazione consiste in una coppia di socket

Comunicazione attraverso socket



Comunicazione attraverso Socket

```
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();
				   // 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);
			  }
		}
}
```

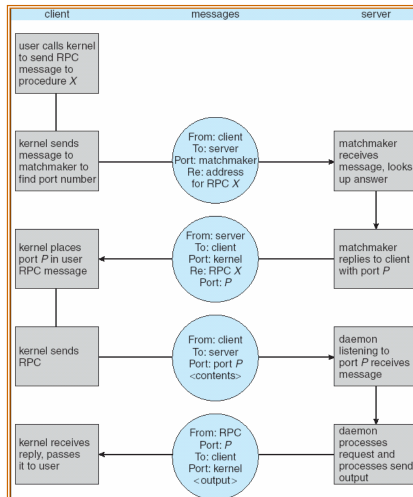
Comunicazione attraverso Socket

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

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

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

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

Remote Procedure Call

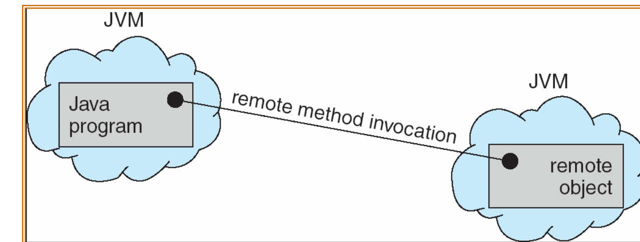
- Remote procedure call (RPC) astraggono le normali chiamate di procedura per uso tra sistemi connessi in rete.
- **Stub** – codice lato client che gestisce la chiamata della procedura sul server.
- Lo stub lato client localizza il server ed effettua il **marshall** dei parametri, i.e., rappresentazione dei dati in XDR (external data representation).
- Lo stub lato server riceve il messaggio, spacchetta i parametri impacchettati, ed esegue la chiamata sul server.

Esecuzione di una RPC



Remote Method Invocation

- Remote Method Invocation (RMI) è un meccanismo Java simile a RPC.
- RMI permette ad un programma Java su una macchina di invocare un metodo su un oggetto remoto in un'altra macchina.



Remote Method Invocation

• Uno **stub** lato client per l'oggetto remoto crea un **parcel** che consiste del nome del metodo e del marshall dei parametri

• Uno **skeleton** lato server riceve il parcel, traduce i parametri, invoca il metodo richiesto, costruisce un nuovo parcel con il risultato dell'invocazione del metodo e lo invia al client.

```
boolean val =server.someMethod(A,B);
```

• Passaggio dei parametri (copia e riferimento)

• Oggetti locali sono passati per copia (**object serialization**)

Remote Method Invocation