Desarrollo de Aplicaciones en Red

José Rafael Rojano Cáceres
http://www.uv.mx/rrojano

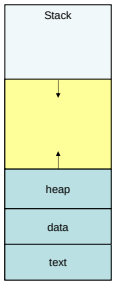
Process management

- A process can be thought as a program in execution.
- Process are the unit of work on modern time-sharing system.
- As a part of the execution the process could need resources. These resources are allocated for it.
- Usually a process has a thread of execution but it can hold more
- The OS is responsible of:
  - *Process management*
  - *Process scheduling*
  - *Process communication*

Concepts

- A program containing a collection of instructions is not a process.
- A process is an alive entity that is running.
- A program is a passive entity
- A process is an active entity
- A program becomes a process when it's loaded into memory

Process in memory



- *Text, collection of instructions. Responsible for the program counter*
- *Data, collection of global variables*
- *Stack, temporary data like parameters*
- *Heap, dynamic memory during process run time*

Process state

- A process can be in the next states:
  - *New: The process is being created*
  - *Running: instructions are executing*
  - *Waiting: process is waiting signaling*
  - *Ready: process is waiting for processor*
  - *Terminated: process finished execution*

Process state diagram

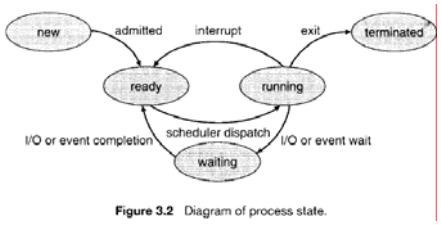


Figure 3.2 Diagram of process state.

Process scheduling

7

Concepts

- **Multiprogramming objective is to have some process running at all times, to maximize CPU use.**
- **Time sharing objective is to switch CPU among process.**
- **Process scheduler achieve the task of select the available process**

8

Scheduling queues

- ***All process in the system are put into a job queue.***
- ***All process ready or waiting belongs to the ready queue. This queue point to the PCB of the process.***
- ***A new process is queued, once it is dispatched several event occurs:***
  - **Process can issue I/O request**
  - **Can create a subprocess and wait for its finishing**
  - **Process can be removed**

9

Scheduler vision

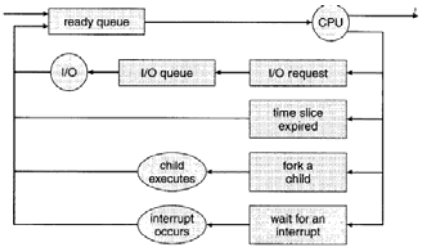


Figure 3.7 Queueing-diagram representation of process scheduling.

10

Scheduler typos

- **Short term scheduler**
  - ***Schedule process to the CPU***
  - ***Fast decision***
- **Long term scheduler**
  - ***Schedule jobs***
  - ***Take from pool to load into memory***
  - ***Executed less frequency***
- **Medium term scheduler**
  - ***Removes process from memory for later reentering to cpu***
  - ***Swapping process***

11

Process creation (1)

- **When a process create a new process:**
  1. ***The parent continues to execute with its children***
  2. ***The parent wait until all children finish***
- **Respect to the address space:**
  1. ***Child is a duplicate from parent (same data a text areas)***
  2. ***Child is a new program loaded into it.***

12

Process creation (2)

- ***Fork creates same address space***
- ***Exec replace memory space, old process is replaced for new one***

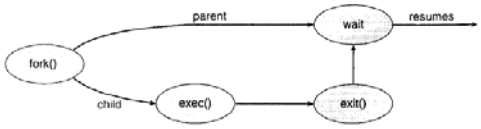


Figure 3.11 Process creation.

Proccess creation API win32

```
#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 line
    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);
}
```

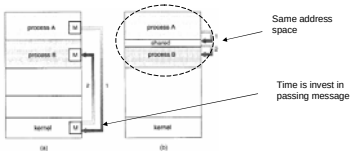

Interprocess communication

How the process communicates

- **Each process running can be either independent or cooperative**
- **Some reasons for cooperation are:**
  - ***To share information***
  - ***Computation speedup***
  - ***Modularity***
  - ***Convenience***
- **To achieve cooperation process require IPC (InterProcess Communication)**

IPC models

- **Shared memory**
  - ***It's established a common memory area***
- **Message passing**
  - ***Message are sent by process***



(a) Message passing. (b) Shared memory.

Shared memory

- ***Should be established a common address space***
- ***The process is responsible for accessing and writing into the same space***

Message passing

- *MP provides a mechanism to communicate a to synchronize without use the same address space*
- *To do this some considerations are necessary:*
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering
- *On each case a link must be establish first*

19

Direct communication (1)

- *The process that want to communicate must have a way to refer each other.*
- *Direct communication*
  - **The name of receiver or sender must be explicitly named**
  - **For example:**
    - Send (p, message) P=receiver
    - Receive (q, message) Q=sender

20

Direct communication (2)

- *The link established under this scheme has the next properties:*
  - **A link between each process is established, need to know each process**
  - **A link is associated with exactly two process**
  - **Between each process exist one link**
- *This process is symmetric because each process must name the other to communicate*

21

Direct communication (3)

- *In an asymmetric communication the sender knows the receiver, the receiver is not required to know the sender*
- *Here the primitives would be like:*
  - **Send (p, message)**
  - **Receive (id, message)** **id=variable process**
- *The disadvantage in any scheme is the dependency and limited modularity*
- *This technique is hard-coding*

22

Indirect communication (1)

- *With this scheme messages are passed and received from mailbox or ports.*
- *A mailbox can be viewed as a object where message can be left.*
- *Each mailbox has a unique id.*
- *The primitives would be something like*
  - **Send (A, message)**
  - **Receive (A, message)** **A=common mailbox**

23

Indirect communication (2)

- *With this scheme messages are passed and received from mailbox or ports.*
- *A mailbox can be viewed as a object where message can be left.*
- *Each mailbox has a unique id.*
- *The primitives would be something like*
  - **Send (A, message)**
  - **Receive (A, message)** **A=common mailbox**

24

Indirect communication (3)

- ***The link established under this scheme has the next properties:***
  - **Communication possible if the two process share a mailbox**
  - **A link can be associated with more than two process**
  - **Between each process exist more than one link**

25

Synchronous or asynchronous communication (1)

- ***MP can be blocking (sync) or nonblocking (async)***
- ***As the communication takes place using the primitives send and receive, does exist different options to implement this:***
  - **Blocking send**
  - **nonBlocking send**
  - **Blocking receive**
  - **nonBlocking receive**

26

Synchronous or asynchronous communication (2)

- ***Blocking send:***
  - **The sending process is blocked until the message is received**
- ***nonBlocking send:***
  - **The sending process sends the message and finish**
- ***Blocking receive:***
  - **The receiver block upto a message is received**
- ***nonBlocking receive:***
  - **The receiver retrives either a message or null**

27

Automatic or explicit buffering (1)

- ***Whether communication is direct or indirect a temporary queue is necessary***
- ***This queue can be implemented as:***
  - **Zero capacity**
  - **Bounded capacity**
  - **Unbounded capacity**

28

Automatic or explicit buffering (2)

- ***Zero capacity***
  - **The queue does not have capacity, in this case the sender must block until the message is delivery**
- ***Bounded capacity***
  - **The queue has finite N capacity. The sender can continue working, but if queue is full blocking is necessary**
- ***Unbounded capacity***
  - **Potentially the queue is infinite, any quantity of message can be wait in it.**

29

Examples

30

Posix: shared memory

- ***In Posix system it's possible to use API***
  1. **Create an shared memory area →shmget**
  2. **Attach the segment to the address space of a new process → shmat**
  3. **Update the shared memory area**
  4. **Release the area from address space**


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

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

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

    /* attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

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

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

    /* now detach the shared memory segment */
    shmdt(shared_memory);

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

    return 0;
}
```

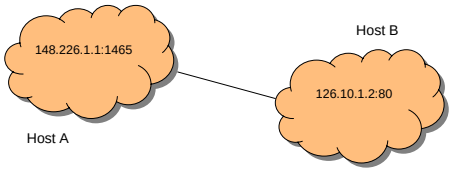
Indicates:
1. A new shared memory is created
2. The size
3. Type of access

Indicates:
1. Id for access
2. Area of memory for the segment, null specify OS decision
3. Type of access, 0=W&R

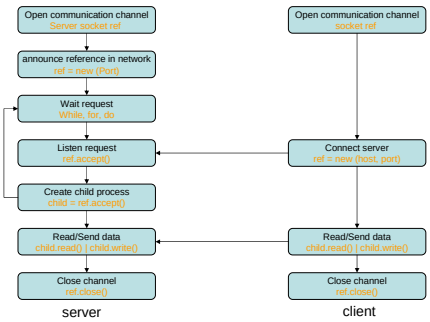
- ***Besides of a process can share communication using shared memory or message passing other three scheme can be used in the model client server***
  - **Sockets**
  - **RPC**
  - **RMI**

Sockets

- ***A socket is defined as an endpoint for communication***
- ***Two process communicating use two sockets***



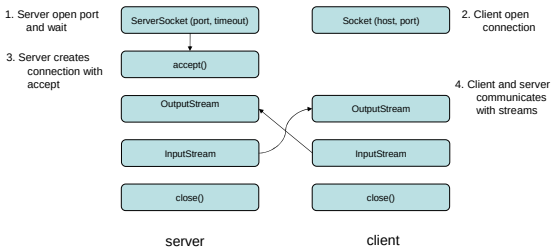
General Scheme for communication



Deployment sockets

- **The elements for working with are in java.net.Socket**
- **Through this class establish a independent environment from platform instead of using native code from platform.**
- **Java.net.ServerSocket implements a socket to listen from the server**

Simple communication scheme for java



Useful class for communication

- ***Socket*: basic object to communicate via TCP. Communication take place through streams.**
- ***ServerSocket*: Listen in server for request connections, Socket object must be used for establish the connection.**
- ***DatagramSocket*: implements UDP sockets.**
- ***MuticastSocket*: mutilcast for machines.**
- ***NetworkServer & NetworkClient*: class used for create methods or variables in a server or client TCP/IP.**
- ***SocketImpl*: let us implement our own Socket class, useful to write firewall or bar reader.**

Opening sockets

- ***For a client:***

```
Socket myClient;
try {
    myClient = new Socket( "host",
    "port");
} catch (IOException e) {
    System.out.println(e);
}
```

- ***For a server:***

```
Socket myServer;
try {
    myServer = new ServerSocket (
    "port");
} catch (IOException e) {
    System.out.println(e);
}
```

Socket Client

```
Socket socketServer = null;
try {
    socketServer = myServer.accept();
} catch (IOException e) {
    System.out.println(e);
}
```


Creating Input streams

- ***For a client:***

```
DataInputStream Input_data;
try {
    Input_data = new DataInputStream(
    myClient.getInputStream() );
} catch( IOException e ) {
    System.out.println( e );
}
```

DataInputStream: let us read lines y primitives java data.

Function for this are available:

- *read(), readChar(), readInt(), readDouble() y readLine().*

- ***For a server:***

```
DataInputStream Input_data;
try {
    Intput_data =
new DataInputStream(
    socketServer.getInputStream() );
} catch( IOException e ) {
    System.out.println( e );
}
```


Creating Output streams

- ***For a client:***

```
PrintStream Output_data;
try {
    Output_data = new PrintStream(
    myClient.getOutputStream() );
} catch( IOException e ) {
    System.out.println( e );
}

DataOutputStream Output_data;
try {
    Output_data = new
    DataOutputStream(myClient..getOut
    putStream() );
} catch( IOException e ) {
    System.out.println( e );
}
```

- ***For a server:***

```
PrintStream Output_data;
try {
Output_data = new PrintStream(
    socketServer.getOutputStream() );
} catch( IOException e ) {
System.out.println( e );
}
```


Closing sockets

- ***For a client:***

```
try {
    Input_data.close();
    Output_data.close();
    myClient.close();
} catch( IOException e ) {
    System.out.println( e );
}
```

- ***For a server:***

```
try {
    Input_data.close();
    Output_data.close();
    socketServer.close();
    myServer.close();
} catch( IOException e ) {
    System.out.println( e );
}
```

Cliente.java

```
import java.io.*;
import java.net.*;
class Cliente
{
        static final String HOST = "localhost";
        static final int PUERTO = 5000;
        public Cliente()
        {
                try
                {
                        Socket skCliente = new Socket(HOST, PUERTO);
                        InputStream aux = skCliente.getInputStream();
                        DataInputStream flujo = new DataInputStream(aux);
                        System.out.println(flujo.readUTF());
                        skCliente.close();
                }
                catch (Exception e)
                {
                        System.out.println(e.getMessage());
                }
        }
        public static void main(String[] arg)
        {
                new Cliente();
        }
}
```

43

Servidor.java

```
import java.io.*;
import java.net.*;
class Servidor
{
        static final int PUERTO = 5000;
        public Servidor() {
try {
ServerSocket skServidor = new ServerSocket(PUERTO);
System.out.println("Escucho el puerto " + PUERTO );
for ( int numCli = 0; numCli < 3; numCli++ ) {
Socket skCliente = skServidor.accept(); // Crea objeto
System.out.println("Sirvo al cliente " + numCli);
OutputStream aux = skCliente.getOutputStream();
DataOutputStream flujo= new DataOutputStream( aux );
flujo.writeUTF( "Hola cliente " + numCli );
skCliente.close();
}
System.out.println("Demasiados clientes por hoy");
} catch( Exception e ) {
System.out.println( e.getMessage() );
}
}
        public static void main(String[] arg)
        {
                new Servidor();
        }
}
```

44

RPC

- ***The Remote Procedure Call is a mechanism to abstract the procedure call mechanism.***
- ***It's similar to IPC, but it's in client-server environment***
- ***All the message are addressed to RPC daemon***

45

RMI

- ***The Remote Method Invocation allows to invoke a method on a remote object.***
- ***Object are considered as remote if they are running on different JVM***
- ***The different between RMI and RPC are:***
  - **RPC support procedural programming, only procedure and functions can be called**
  - **RMI is based in object, so method can be invoked**
  - **In RPC the parameters are based in structures**
  - **In RMI can be passed objects as parameters**
- ***Through RMI is possible to develop distributed application across a network***

46

Reference

- **Silberschatz et Al, Operating Systems concepts 7th. Wiley.**

47