System Programming

Xabier Elkorobarrutia

MGEP

Outline

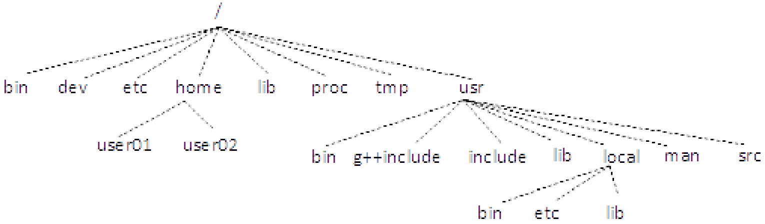
1. File System
2. Libraries
3. Introduction to Make
4. Multi-Processing
5. Processes
6. Semaphores
7. Monitors
8. Sockets
9. POSIX Threads
10. Bibliography

File System

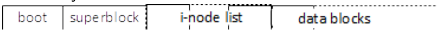
1. File System
    - Structure
    - i-Nodes
    - Run-Time Structures
2. Libraries
3. Introduction to Make
4. Multi-Processing
5. Processes
6. Semaphores
7. Monitors
8. Sockets
9. POSIX Threads
10. Bibliography

Structure

- A file system is an *abstraction mechanism* for using a physical device in a logical way without knowledge of its hardware particularities.
- In UNIX-like system file systems are characterized by:
  - They have a hierarchical structure.
  - Uses files as abstraction for data collections.
  - Protects data.
  - Use peripherals and devices as they were files.



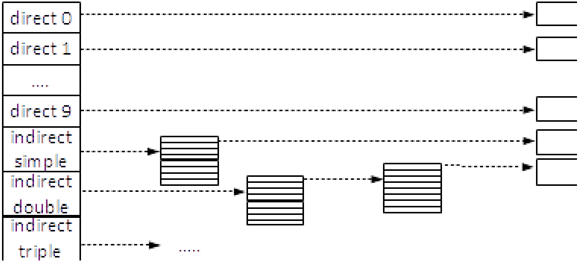
- A file System is composed of a sequence of *"logical block"* of fixed size which is a multiple of 512 bytes.
- Those logical block are sequentially numbered so that we can abstract from specific storage device hardware.
- Structure of a UNIX V file system:



  - **boot**: this very first block can contain boot code.
  - **superblock**: describes the file system itself: its size, maximum number of files, free space, . . .
  - **i-node list**: each file has a description of itself. This is an i-node.

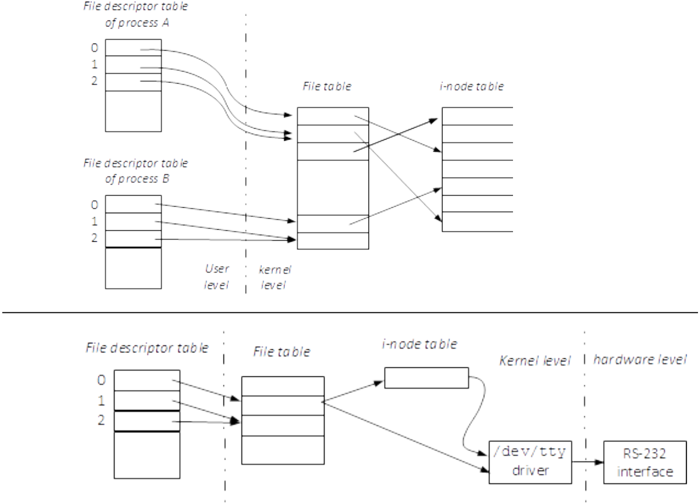
i-Nodes I

- The fields that an i-node has and describes a file:
  - identifier of its owner
  - type: normal file, directory, device file, . . .
  - number of links
  - size
  - reference to data blocks:



Run-Time Structures

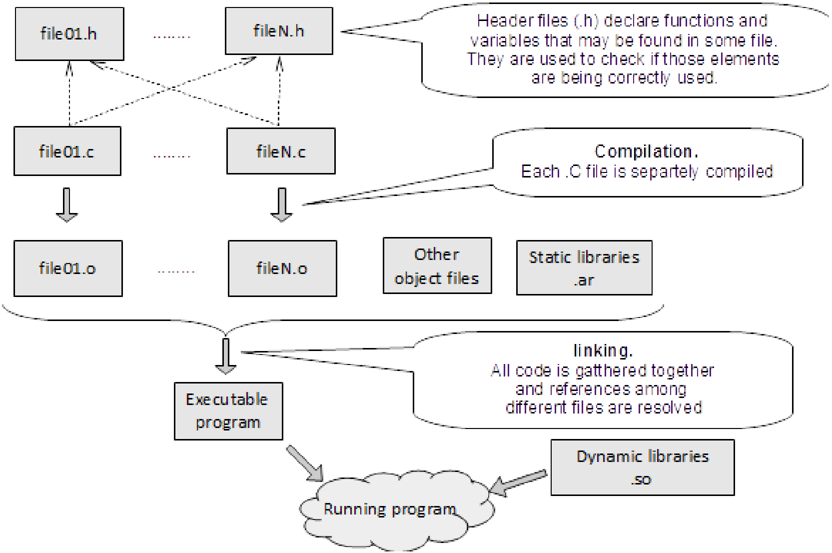
- The kernel of Linux maintains at run-time some structures related to the file system.



Libraries

1. File System
2. Libraries
    - Overview of Program Development in C
    - Libraries
    - Static Libraries
    - Shared Libraries
    - Dynamically Loaded Libraries
    - Miscellaneous
3. Introduction to Make
4. Multi-Processing
5. Processes
6. Semaphores
7. Monitors
8. Sockets
9. POSIX Threads
10. Bibliography

Overview of Program Development in C



Libraries

- A library is a chunk of code and data that is linked to an (incomplete) program. Depending of the linking time next classification can be made:
  - **Static libraries**: a collection of object files that are linked to other object code in order to produce an executable program.
  - **Shared libraries**: libraries that are linked to a program before this is going to be launched.
  - **Dynamic link libraries**: libraries that are linked to a program upon a explicit request and in a programmatic way.

Static Libraries

- They are a collection of object files.
  - They offer a mechanism for not showing source code.
  - They can be $1 - 5\,\%$ faster that other types of libraries.
- A sample example for creating and using a static library.
  1. Create the object files:

     ```
     gcc -c file01.c file02.c
     ```

  2. Add object files to an archive (library).

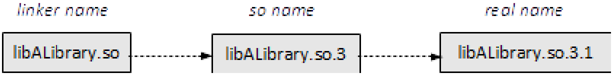
     ```
     ar rcs libaLibrary.a file01.o file02.o
     ```

  3. When creating a program specify the library to be linked with.

     ```
     gcc  sample.c -L. -laLibrary -o executableName
     ```

Shared Libraries I

- Shared libraries are libraries that are loaded by programs when they start.
  - Various programs can share the same library.
  - It is possible to install new version of the library while preserving the old ones if needed.
  - Override specific libraries or functions when executing a particular program.
- On Linux Systems shared libraries have 3 names:



  - The ***real name***. The file that really contains the shared library.
  - The ***soname***. A link to the real name.
  - A ***linker name***. A link to . . . or other version. It is used by the linker and can be useful for debugging purposes.
- For placing libraries in the file system, GNU recomends `/usr/lib` `/usr/local/lib` (although it offers ways for overriding this placement)
- By means of `ldconfig` command
  - Sonames for existing libraries in `/usr/lib` or other predefined places are created
  - `/etc/ld.so.cache` file is updated so that library loading becomes faster.
  - It doesn't create linker names
- On Linux systems program loading is done by `/lib/ld-linux.so.<version>` and
  - In addition of starting up a program, it finds and loads all other needed libraries.
  - The list of directories to be searched is stored in the file `/etc/ld.so.conf` .
  - In order to override some functions in a library, it can be done by means of `/etc/ld.so.preload` file.

Shared Libraries II

- The previous process can be controlled and modified by some environment variables (useful for debugging purposes):
  - If `LD_LIBRARY_PATH` is defined, it takes precedence over the standard search path.
  - By means of `LD_PRELOAD` some functions of a library can be overridden.
- Shared library creation.
  - When compiling `-fPIC` option must be enabled (position independent code)
  - The general format for linking and creating the shared library is:

```
gcc -shared -Wl,-soname,yourSoname -o libraryName \
     fileList libraryList
```

  - Example that creates a shared library from two source files:

```
gcc -fPIC -c file01.c
gcc -fPIC -c file02.c
gcc -shared -fPIC -Wl,-soname,libALibrary.so.1 \
    -o libaLibrary.so.1.0 file01.o file02.o
```

Dynamically Loaded Libraries I

- Dynamically loaded libraries are not loaded at startup of the program but at any moment during its runtime it decides to.
  - It is a mechanism for implementing plugins and loading them only when needed.
  - Even for implementin a JIT (Just in Time Compiler)
  - ...
- In creating such libraries there are not differences with shared libraries. The differences are in the programs that use it; they must load them; find a function; or unload them.
- There is an API for dynamically loading a library. Source programs must include `<dlfcn.h>` and must be linked with `libdl` library. Such API is:
  - For loading a library:

```
void * dlopen(const char *filename, int flag);
```

  - For unloading a library

```
int dlclose(void *handle);
```

  - For looking up a symbol in the library (i.e. a function)

```
void * dlsym(void *handle, char *symbol);
```

- Let's show an example that dinamically loads a library called *"lib.dll.so.1"*.

Dynamically Loaded Libraries II

```
#include "functions01.h"
#include <stdio.h>
#include <string.h>
#include <dlfcn.h>

typedef int (*PGCD) (int,  int);

int main (int argc, char* argv [])
{
    char str[32];
    int n1, n2;
    void *handle;
    PGCD  pGcd;

    // ask to the user 2 integer numbers and read to n1 and n2

    handle = dlopen ("libdll.so.1", RTLD_NOW);
    if (handle == NULL)
    {
        perror(dlerror());
        return 1;
    }

    pGcd = (PGCD) dlsym (handle, "gcd");
    if (pGcd == NULL && dlerror () != NULL)
    {
       ....
    }
    printf("gcd(%u,%u)=%u\n", n1, n2, pGcd(n1,n2));
    dlclose (handle);
    return 0;
}
```

Miscellaneous

- The `nm` command list the exported symbols of a library.
- Libraries can have a constructor and a destructor that are called when loading an unloading the library respectively.

```
void __attribute__ ((constructor)) my_init(void);
void __attribute__ ((destructor)) my_fini(void);
```

- For looking up the dynamic symbols needed by a program

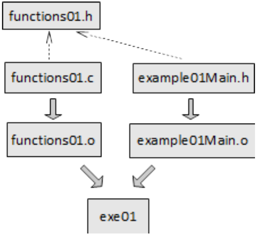
```
objdump -T exe
```

Introduction to Make

1. File System
2. Libraries
3. Introduction to Make
    - Make, What For?
    - Rules
    - Variables
    - More on Rules
    - Functions for Transforming Text
4. Multi-Processing
5. Processes
6. Semaphores
7. Monitors
8. Sockets
9. POSIX Threads
10. Bibliography

Make, What For?

- The `make` utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.
- Relations among source, intermediate and final files are defined through a *"makefile"*.




```
all: exe01

exe01: example01Main.o functions01.o
    gcc example01Main.o functions01.o -o exe01

functions01.o: functions01.c functions01.h
    gcc -c -Wall functions01.c

example01Main.o: example01Main.c functions01.h
    gcc -Wall -c example01Main.c

clean:
    rm -f functions01.o example01Main.o

cleanAll: clean
    rm -f exe01
```

- The `make` utility runs in two phases:
  - During the first one it creates a dependency tree
  - In the second one renews the targets that need to be reconstructed.

Rules

- The main elements of a makefile are the rules. They define dependencies among different files and specify the command to be issued if some of them must be reconstructed.

```
target: prerequisites
<tab> recipe
<tab> more recipes if needed
```

- A target may appear in many rules as far as only one of them specifies hows to construct it.

```
example01Main.o: example01Main.c
       cc -c example01Main.c

example01Main.o: functions01.h
```

- Targets are constructed if they are older than some of their prerequisites:
  - Not existent target ⇒ obsolete target
  - If some prerequisite do not exist ⇒ obsolete target
- A target may not be a real file. The associated rule will not work if a file with the same name exist. It can be marked as a *"false"* target.

```
clean:
    rm *.o
```

```
.PHONY: clean

clean:
    rm *.o
```

Variables

- A variable can represent a string of text

```
x=word01 word02
y:=$(x) word03
# y will contain "word01␣word02␣word03"
```

  - Some variables (the ones in which := is used) are inemdiately expanded during the first phase
  - Others (the ones asigned by =) are expanded when needed: deferred expansion.
- Some *Automatic variables*:
  - `$@` : the target of a rule
  - `$%` :
  - `$<` : the first prerequisite.
  - `$?` : the list of more recent prerequisites than the target.
  - `$^` : All prerequisites without duplicates.
  - `$+` : as `$^` but maintaining the duplicates.
  - ...

More on Rules

- ***Static pattern rules*** are rules which specify multiple targets and construct the prerequisite names for each target based on the target name.

```
$(objects): %.o: %.c
$(CC) $(CFLAGS) $< -o $@
```

  - for each target on `objects` every item ending whith `.o` depends on the same item replacing `.o` by `.c`
  - The % symbol, called the ***stem***, represent any part of a word that matches the pattern.
- A ***suffix rule*** is an old fashioned way of defining rules.

```
.o.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@
```

  - They are maintained for compatibility reason; instead, pattern rules are recommended.
  - They can be defined by a rule:

```
%.o: %.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@
```

- `make` has some predefined rules, called ***implicit rules*** and variables. The previous rule is one of them.

Functions for Transforming Text

- In addition to variable substitution, make offers more sophisticated functions for of processing text. Their syntax is:

```
\$(function comma-separated-arguments)
```

- Some examples:
  - In next example `a.o b.o` is obtained

```
$(patsubst %.c,%.o, a.c b.c )
```

  - `$(filter pattern, text)` : returns the words that match the pattern
  - `$(filter-out pattern, text)` : returns the words that do not match the pattern
  - `$(dir text)` : returns the directory part of each file name
  - `$(notdir text)` : extracts the directory part from each file name
  - `$(addsuffix suffix,text)` : adds the specified suffix to each file name
  - `$(addpreffix preffix,text)` : adds the specified prefix to each file name
  - ...

Multi-Processing

1 File System

2 Libraries

3 Introduction to Make

4 Multi-Processing

- Multi-Processing
- Multi-Threading
- Differences among Multi-Threading/Multi-Processing
- Usefulness of Multiprocessing
- Race Conditions and Lack of Synchronization
- Issues to guarantee/solve in Tasks' Interactions

5 Processes

6 Semaphores

7 Monitors

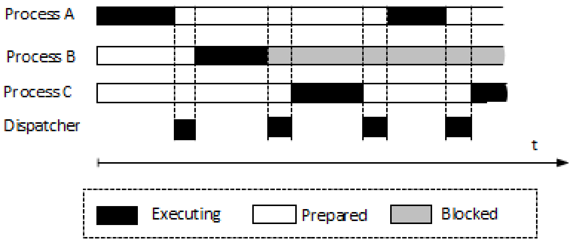
8 Sockets

9 POSIX Threads

10 Bibliography

Multi-Processing

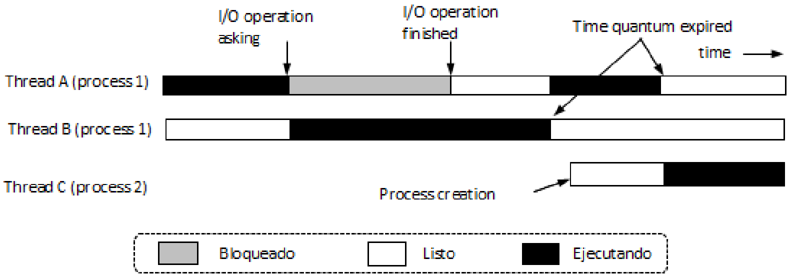
- **Process**: Program in execution.
- Multiprocessing: ability to execute various processes *"virtually or actually in parallel"*.
- The operating system is responsible for dispatching/preempting precesses according to the policies it applies.



- Form the OS point of view, a process, in addition to be a unit of expedition, also is a ***unit of resource assignment***.
    - If a resource is being used by a process, no other process can use it.
    - The OS protects every process not allowing to *"cross their limits"*.
    - The OS must act as a mediator for processes to be able to coordinate/collaborate.

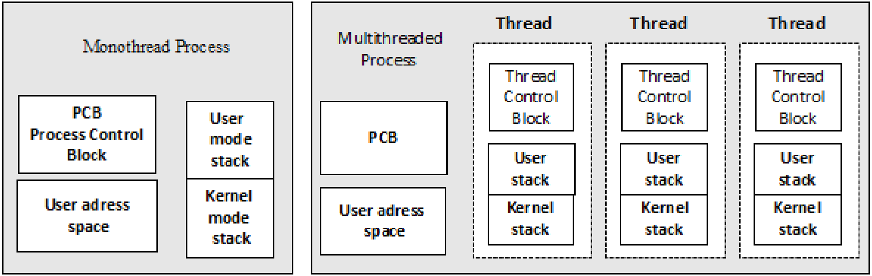
Multi-Threading

- Lets distinguish the concepts of ***unit of resource assignment*** and ***unit of expedition***. The latter will be called ***Thread*** or ***Ligthway process***. .
- **Multi-Threading**: Ability to have various threads in a single process.



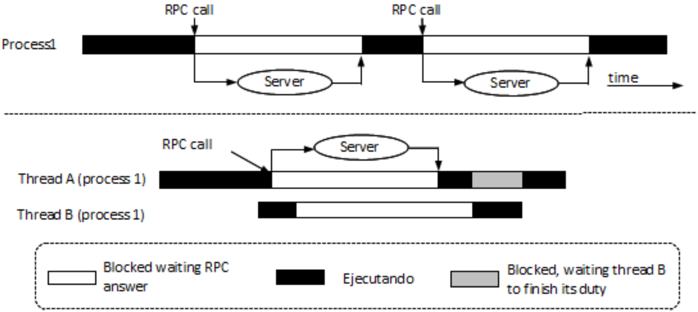
Differences among Multi-Threading/Multi-Processing

- Advantages of multi-threading with respect to multi-processing:
  - Needs less time in creation/destruction.
  - It is faster to swap among threads than among processes.
  - The communication among threads does not need the mediation of the OS.
- Disadvantages of multi-threading with respect to multi-processing:
  - The OS can not help in protecting undesired interaction among threads.
- What does involve the management of threads and processes:



Usefulness of Multiprocessing

- Text editor with two threads where:
  - The first one attends the user
  - The second makes backups.
- Spread sheet with two threads where:
  - The first one attends the user
  - The second one makes the necessary calculi on the cells affected by changes made by the user,
- Parallelize functionality in order to waste as less as possible time when waiting I/O operations.



- *There are some computational problems that are easier to solve with the support for concurrent programming.*

Race Conditions and Lack of Synchronization

- **Example 1**. Let `x` and `y` be global variables and `readChar()` an atomic operation. Which different outputs can produce next code if both tasks execute concurrently?

```
  //task01
funTask()
{
  x=readChar();
  y=x;
  writeChar(y)
}
```

```
  //task02
funTask()
{
  x=readChar();
  y=x;
  writeChar(y)
}
```

- **Example 2**. Which are the lower and upper values that variable `cnt` can have when writing its content?

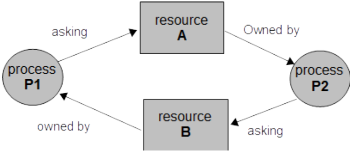
```
int cnt=0;

void total()
{
  int i ;
  for (i =0; i<50 ; i++) cnt++;
}
void main ( )
{
  parbegin( total , total) ;
  write(cnt) ;
}
```

Issues to guarantee/solve in Tasks' Interactions

- First of all, it can help to characterize the interaction among tasks:
  - Tasks that does not know each other but compete in acquiring a resource.
  - Interdependent task: e.g., Tasks that read/write in the same memory area.
  - Task that collaborate among them: Sending messages, . . .
- **Mutual Exclusion**. Need to guarantee the individual access to a resource.
- **Deadlocks**:

```
//task01
allocate R1
allocate R2
.....
free R2
free R1
```

```
//task02
allocate R2
allocate R1
.....
free R1
free R2
```



- **Starvation**: A task can be indefinitely waiting in a resource due other most priority task are using it or due to a bad management.
- In a set of collaborating tasks, one of them can be awaiting a message from another one that never will send it.

Processes

1 File System

2 Libraries

3 Introduction to Make

4 Multi-Processing

5 Processes
- Processes
- Process Creation
- Process Termination
- Signals

6 Semaphores

7 Monitors

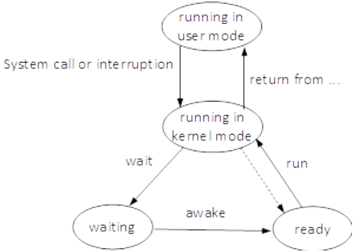
8 Sockets

9 POSIX Threads

10 Bibliography

Processes

- A Process is a running instance of a program.
- Next figure illustrates the states a process can be in during its lifetime.



- In Linux all processes are arranged in a hierarchical structure establishing a ***parent-child*** relationship among them.
  - All processes are created by some other process; except the very first called `init` .
  - The creator of another process becomes the *"father"* of this second one,

```
pid_t getpid()   //gets de identifier of the running process
pid_t getppid()   //gets de identifier of the patent of the running process
```

Process Creation I

- The exec functions family replaces the program of a process by another program.
  - The running process is stopped when executing such functions and continues at the beginning of the new program.
  - There are many variant of exec functions: execvp , execv execve ...
  - Here is an example that dependig of its command line arguments *"mutates"* to ls or ps program.

```
int main(int argc,char* argv[])
{
  int opcion;
  char *linCom1[]={"ls",NULL};
  char *linCom2[]={"ps",NULL};

  if(argc!=2)
  {
    printf("calling format: process02 (1|2)\n");
    exit(-1);
  }
  sscanf(argv[1],"%d",&opcion);
  switch(opcion)
  {
    case 1:execvp("ls",linCom1);
           break;
    case 2:execvp("ps",linCom2);
           break;
    default:printf("not valid option\n");
  }
  return 0;
}
```

Process Creation II

- By means of `fork()` system call, a process is duplicated. The created process continues where its father was.
  - Its heap, stack, variables are a copy of its father's.
  - Thus, how can be known which process we are in? `fork()` returns the `pid` of its child to the father and 0 to the child.
  - Here is an example that creates 3 children one of which transform to `ps`

```
int main(int argc,char* argv[])
{
  char *strPsCommand[]={"ps",NULL};

  if(fork()==0)
  {
    sleep(1);
    printf("1rst child endind\n");
  }
  else if(fork()==0)
  {
    sleep(1);
    printf("2nd child ending\n");
  }
  else if(fork()==0)
  {
    execvp("ps",strPsCommand);
  }
  else
  {
    sleep(5);
    wait(NULL);  wait(NULL);  wait(NULL);
    printf("main process ending\n");
  }
  return 0;
}
```

Process Termination

- A process finishes when exiting its `main` function or executing the `exit` function.
- Each process returns an exit code to its father (the return value of `main` or the parameter of the `exit` function).
- A father process must wait until the termination of all its children. If not those will become ***zombie*** processes: they are not running but still are in the system process table.
- `wait()` function serves this purpose.

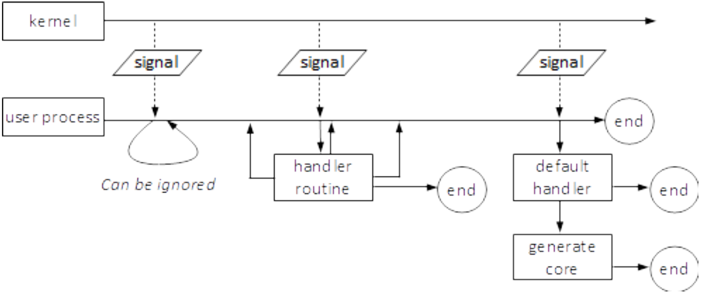
```
pid_t wait(int *pStatus)
```

  - it gets blocked until some child finishes or has finished.
  - returns the `pid` of a finished process.
  - In its parameter return the exit code

```
int WIFEXITED(int status) // returns TRUE if the process finished normally
int WEXITSTATUS(int status)   // returns the exit code
```

Signals I

- A signal is an asynchronous message that can be sent to a process.
- When a process receives a signal it, interrupts its execution and jumps to a ***signal handler***.



- Some signals:

name	when sent or intent	default action
SIGINT	when user presses Ctrl C	finish the process
SIGKILL	killing a process	generate core and finish. Can't be ignored
SIGFPE	floating point error	generate core and finish
SIGALARM	some time has expired	generate core and finish
SIGUSR1	reserved for user usage	finish
SIGUSR2	reserved for user usage	finish
SIGTERM	indication that it should finish	finish
...	...	...

Signals II

- The signal function allows to specify a handler routine for a specific signal.

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

  - handler can be the address of the routine handler
  - If SIG_DFL the default routine is installed.
  - If SIG_IGN the signal is ignored.
  - Once a signal is received its default handler is installed.
- Ways of provoking the sending of a signal to a process:
  - From the shell through kill command.
  - From a process using kill function.

```
int kill(pid_t pid, int sig);
```

  - A process can send a signal to itself

```
int raise(int sig);
```

- pause() functions blocks a process until it receives a signal.

Signals III

- Sample program

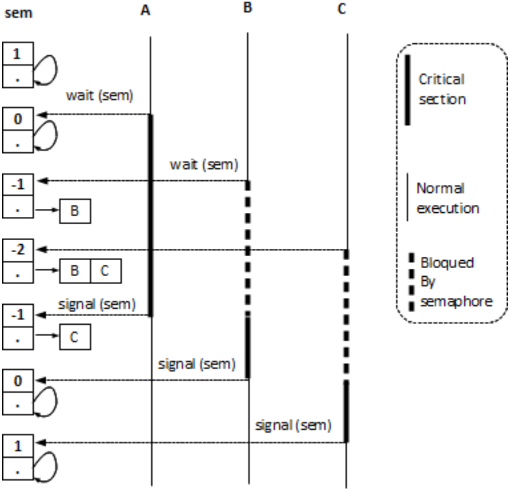
```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sigHandler();
int nSignals=0;

int main(int argc,char* argv[])
{
 signal(SIGINT,sigHandler);
 while(nSignals<5)
 {
   pause();  //not too perfect
   printf("You␣have␣\"Ctrl-C\"␣%d␣times\n",nSignals);
 }
 return 0;
}

void sigHandler()
{
 signal(SIGINT,SIG_IGN);
 nSignals+=1;
 signal(SIGINT,sigHandler);
}
```

Semaphores

1. File System
2. Libraries
3. Introduction to Make
4. Multi-Processing
5. Processes
6. Semaphores
    - Semaphore vConcept
    - Critical Sections
    - Producers/Consumers' Problem
7. Monitors
8. Sockets
9. POSIX Threads
10. Bibliography

Semaphore vConcept

- A signal exchange mechanism by which tasks can collaborate.
  - It has an initial amount of *"permissions"*, *"tokens"*, ...
  - `wait()`. This operation allows task to take a token or be blocked until one is available.
  - `signal()`. This operation is used for returning tokens.
  - Both operations must be atomic.



Critical Sections

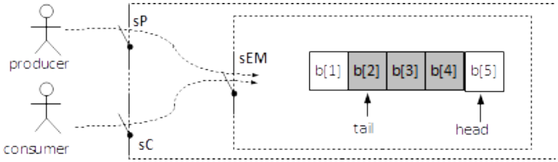
- Example: Concurrent access to a *critical resource* using semaphores:

```
//globals
Semaphore s(1);
```

```
//task 1
funTask1()
{
  while(...)
  {
    .....
    wait(s);
    //critical section
    signal(s);
  }
}
```

```
//task2
funTask2()
{
  while(...)
  {
    .....
    wait(s);
    //critical section
    signal(s);
}
```

Producers/Consumers' Problem



//globals
 	
Semaphore sMe(1);
Semaphore sC(0);
Semaphore sP(5);
Buffer buf;
	void producer()
	{
	  Element item;
	
	  while (true)
	  {
	    item=produce();
	    wait(sP);
	    wait(sMe);
	    put(buf, item);
	    signal(sMe);
	    signal(sC);
	  }
	}
		void consumer()
		{
		  Element element;
		
		  while(true)
		  {
		    wait(sC);
		    wait(sMe);
		    get(buf, &element);
		    signal(sMe);
		    signal(sP);
		    consume(element);
		  }
		}

Monitors

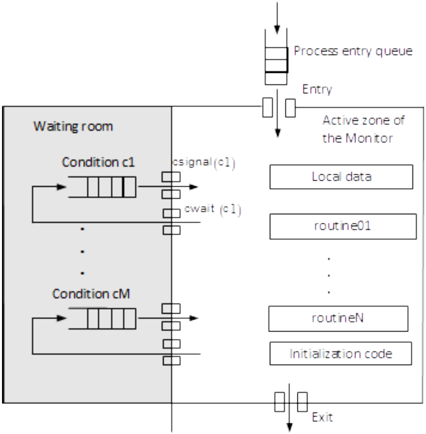
1. File System
2. Libraries
3. Introduction to Make
4. Multi-Processing
5. Processes
6. Semaphores
7. Monitors
   - Background and Motivation
   - Concept
   - Producer/Consumers' Problem
   - POSIX Mutex and Conditions
   - Producers/Consumers, without a Monitor?
8. Sockets
9. POSIX Threads
10. Bibliography

Background and Motivation

- Monitors had been defined and redefined many times: Per Brinch Hansen (1973), Tony Hoare (1994), Butler Lampson and David Redell (1980)
- We will use last one.
- They aim to help to structure the code better.
- A Monitor is a programming-time construct: it can be supported by the language itself, be a design idiom or . . .

Concept

- A monitor is a module that encompasses a set of data and functions.
- It only can be used *exclusively*: only one thread can be active within the monitor at a time.
- **Condicitions**: They constitute a synchronization mechanism that monitors offer by which a thread can wait until some *"condition"* meets. This event must be signalled by another thread.
  - `cwait(aCond)` : A thread is blocked in a condition. Another thread can enter in the monitor.
  - `cnotify(aCond)` : One thread can use it to awake the first thread awaiting in such condition.
  - `cbroadcast(aCond)` : All awaiting threads are awakened.



Producer/Consumers' Problem I

```
Monitor FIFO
{
  int head;
  int tail;
  int numElems;
  float buffer[MAX_BUFFER_SIZE];
  condition condNotFull;
  condition condNotEmpty;

  void initFifo() { head=tail=numElems=0;}

  int put(float elem)
  {
    while(numElems == MAX_BUFFER_SIZE) cwait(condNoTFull);
    buffer[head] = elem;
    head = (head+1) % MAX_BUFFER;
    numElems++;
    if(numElems == 1) cnotify(condNotEmpty);
  }
  int get( float * pElem)
  {
    while(numElems == 0) cwait(condNotEmpty);
    *pElem = buffer[tail];
    tail = (tail+1) % MAX_BUFFER;
    numElems--;
    if(numElems == (MAX_BUFFER_SIZE -1)) cbroadcast(condNotFull);
  }
}
```

Producer/Consumers' Problem II

```
FIFO fifo;

main
{
  fifo.initFifo();
  parbegin(producer,consumer);
}
```

```
void producer()
{
  float element;

  while(...)
  {
    element=produce();
    fifo.put(element);
    ......
  }
}

void consumer()
{
  float element;

  while(...)
  {
    fifo.get(&element);
    consume(element);
    ......
  }
}
```

POSIX Mutex and Conditions

- Another synchronization mechanism offered by POSIX is constituted by ***Mutexes*** and ***Conditions***.
  - Functionally, a `mutex` is analogous to a binary semaphore.
  - ***Conditions*** are a signalling mechanism by a thread can be blocked awaiting a signal another thread sends.
- They can be used without the notion of monitor, but
  - They constitute a base mechanism to implement monitors.
  - Implicitly, we would be using monitors; or if not, the code would not be as *"clean"* as it could be from a software-engineering point of view.

Producers/Consumers, without a Monitor?

```
  //globales

Mutex mtx;
Condition cNotFull, cNotEmpty;
Buffer buf;
```

```
void producer()
{
  Element element;

  while (true)
  {
    element=produce();
    lock(mtx);
    while(isFull(buf)
      cwait(cNotFull,mtx);
    put(buf, element);
    csignal(cNotEmpty);
    unlock(mtx);
  }
}
```

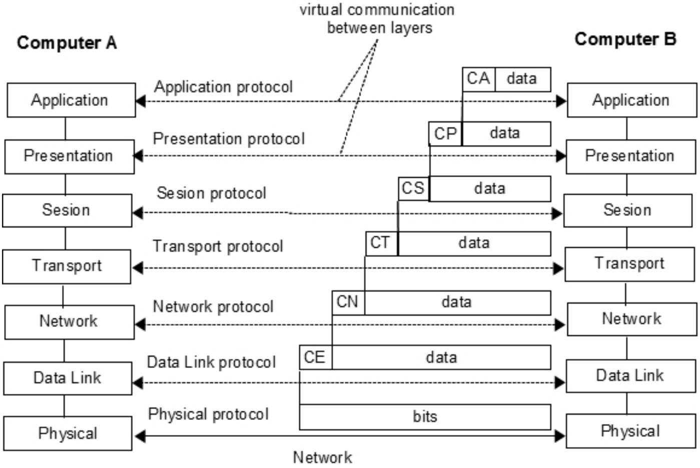
```
void consumer()
{
 Element element;

 while(true)
  {
    lock(mtx);
    while(isEmpty(buffer))
      cwait(cNotEmpty,mtx);
    element=get(buf);
    csignal(cNotFull);
    unlock(mtx);
    consum(element);
  }
}
```

Sockets

1. File System
2. Libraries
3. Introduction to Make
4. Multi-Processing
5. Processes
6. Semaphores
7. Monitors
8. Sockets
9. POSIX Threads
10. Bibliography

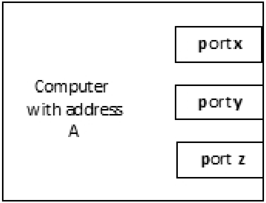
- OSI Reference Model
- Transport Layer
- Connection-Oriented Sockets
- Message-Oriented Sockets
- Practices

OSI Reference Model



Transport Layer

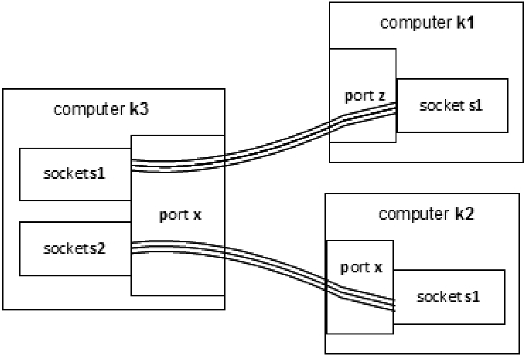
- TCP/IP is the most used transport protocol.
- **Socket's interface**. Is an API by which applications can use the transport layer. Originally developed at Berkeley.
- ***IP addresses*** are used to identify machines at network layer.
- One abstraction that sockets offers is the ***port***. They enable to address various applications in a single IP.



- TCP/IP sockets allows two kind of communications:
  - **Connection-oriented**. Similar to a phone
  - **Message-oriented**. Similar to a post service.

Connection-Oriented Sockets

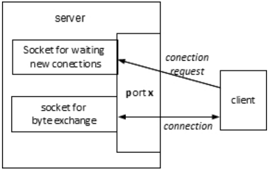
- *The endpoint af a bidirectional communication channel between two machines.*

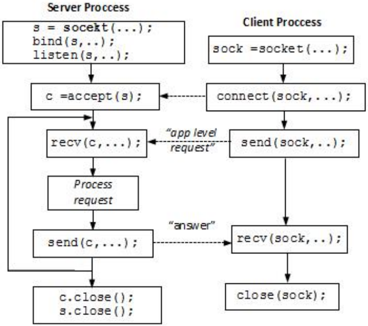


- TCP is the most used protocol.
- Bytes transmitted at one end (a socket) of the communication channel are received in the same other in the other end.
- A socket that is being used for communication:
  - Sends and receives information from a given IP and port of the machine it is working.
  - Knows the IP and port of its counterpart sockets.

Connection Establishment

- In the establishment of a connection of two processes, those play an asymmetric rol; the process that makes the call is known as the *client* and the one that receives the call, the *server*.
- The server must use two sockets: one for accepting connections and a second one for communicating.



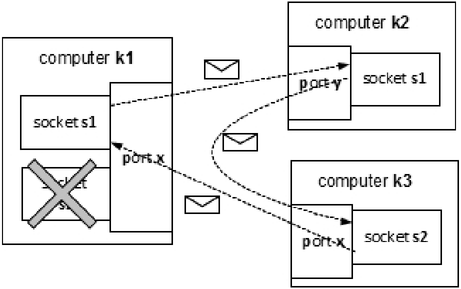


Examples of Connection-Oriented Sockets

1. `EcoCliente.java` and `EcoServ.java` . First one sends $'\backslash n'$ terminated character strings to the second and displays the response. The second one, after receiving such a string, converts it to upper case and sends back this string.
2. `EcoServ2.java` : an improvements of `EcoServ.java` that allow to serve various clients (not concurrently).
3. `EcoServ3.java` : an improvements of `EcoServ2.java` that allows to serve various clients concurrently.
4. A calculus server:
   - A server that calculates GCD and factorial. It is composed of `ServCalculo.java` and `GestorProtocolo.java` classes.
   - A client applications that uses the above mentioned service.It is composed of `ClientCalculo.java` and `ProxyCalculo.java` classes.

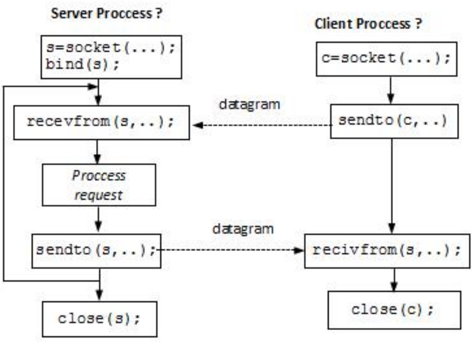
Message-Oriented Sockets

- The communication model consist on the exchange of messages called ***datagrams***.
- A datagram is a bounded sequence of bytes.
- UDP is the most used protocol.
- In this case a sockets is an element by which an application can send and receive datagrams.



Appication Models

- From a communication point of views, communicating processes do not play asymmetric roles.
- Anyway, from the business point of view, lot of times a process act as a server an another one as a client.



- Message-Oriented socket examples:
  - `EcoServMens.java` : application that after receiving a string that contains Datagram, respond with another Datagram that contains the same string converted to upper case
  - `EcoClientMens.java` : Application that asks to the user for some string, uses the above mentioned service to renders the result on the screen.

Practices

1. Improve the Calculi Server shown in classroom following next steps:
   - The server should be able to attend many clients concurrently.
   - The server controls the number of connected clients so that never are more than 10 clients being attended.
   - The server must finish when the user presses `Return` key meeting next conditions:
     - For then on, it will not accept new connections
     - It will wait for the connection already established to be closed.
2. Implement a Calculi server using UDP datagrams as communication mean. Also a client to test it.
   - Calculus request will be sent by means of an UDP datagram.
   - When calculating, sleep this algorithm to simulate a large duration calculus (by means of `sleep`)
   - Many request can be attended simultaneously but never more than 10.
   - The server must finish when the user presses `Return` key:
     - For then on, it will not accept new requests.
     - It will wait for requests already being served to be attended.

POSIX Threads

1 File System
2 Libraries
3 Introduction to Make
4 Multi-Processing
5 Processes
6 Semaphores
7 Monitors
8 Sockets
9 POSIX Threads
- Creating and Joining Threads
- POSIX Semaphores
- POSIX API for Mutex and Conditions

10 Bibliography

Creating and Joining Threads I

- ***POSIX threads*** or ***pthreads***. Is an API for the creation and manipulation of threads (POSIX.1c).
- Natively available on UNIX like platforms. And on Windows through a library on top of windows API. Other platforms . . .

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

void* threadMainFunction1(void*);
void* threadMainFunction2(void*);

int main()
{
  pthread_t idThread1, idThread2, idThread3;

  printf("creating threads ......\n");
  pthread_create(&idThread1, NULL, threadMainFunction1, (void*) 1);
  pthread_create(&idThread2, NULL, threadMainFunction1, (void*)2);
  pthread_create(&idThread3, NULL, threadMainFunction2, (void*)"kuku");
  pthread_join (idThread1, NULL);
  pthread_join (idThread2, NULL);
  pthread_join (idThread3, NULL);
  printf("Threads finished. Main going to finish \n");
  return 0;
}
```

Creating and Joining Threads II

```
void *threadMainFunction1 (void *arg)
{
  int i, j,  n = (int) arg;

  for (i=0;i<10;i++)
  {
    sleep(1);
    for(j=0;j<n;j++) printf("\t\t");
    printf ("␣.......␣%d␣...........\n", n);
  }
  return NULL;
}

void *threadMainFunction2 (void *arg)
{
  int i;
  char* pStr= (char*)arg;

  for(i=0;i<5;i++)
  {
    sleep(2);
    printf("%s\n", pStr);
  }
  return NULL;
}
```

POSIX Semaphores

- The type and prototype definitions of POSIX semaphores are located in `semaphore.h`.
- Semaphore definition:

```
sem_t aSemaforo;
```

- POSIX function for semaphore manipulation:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
int sem_getvalue(sem_t *sem, int *valp);
```

POSIX API for Mutex and Conditions

- Declared in: <pthread.h>.
- Definition (creation) of a mutex and a condition :

```
    pthread_mutex_t unMutex;
    pthread_condition_t unMutex;
```

- Functions for mutex manipulation:

```
int pthread_mutex_init(pthread_mutex_t * mutex, pthread_mutexattr_t * attr );
int pthread_mutex_destroy(pthread_mutex_t * mutex );
int pthread_mutex_lock(pthread_mutex_t * mutex );
int pthread_mutex_trylock(pthread_mutex_t * mutex );
int pthread_mutex_unlock(pthread_mutex_t * mutex );
```

- Functions for Conditions manipulation (note that a mutex is needed always):

```
int pthread_cond_init(pthread_cond_t * cond , pthread_condattr_t * attr );
int pthread_cond_destroy(pthread_cond_t * cond );
int pthread_cond_signal(pthread_cond_t * cond );
int pthread_cond_wait(pthread_cond_t * cond ,pthread_mutex_t * mutex );
int pthread_cond_timedwait(pthread_cond_t * cond ,pthread_mutex_t * mutex , struct timespec * timeout);
int pthread_cond_broadcast(pthread_cond_t * cond );
```

Bibliography

Gnu make manual.
http://www.gnu.org/software/make/manual/, 2013.

Rovert Love.
Linux System Programming.
O'Reilly Media Inc, 2nd edition, 2013.

Jon Masters and Richard Blum.
Professional Linux Programming.
Willey Publishing Inc, 2007.

Mark Mitchell, Jeffrey Oldham, and Alex Samuel.
Advanced Linux Programminge.
New Riders, 1st edition, Jun 2001.

Francisco M. Márquez.
UNIX Progrmación Avanzada.
Ra-Ma, 3rd edition, 2004.

David A. Wheeler.
Program library howto.
http://tldp.org/HOWTO/Program-Library-HOWTO/, 2010.