Mecanismos de comunicación y sincronización entre procesos
Colas de mensajes

Departamento de Ingeniería de Sistemas y Automática. EII.

Universidad de Valladolid

Colas de mensajes

- Una **cola de mensajes** es una estructura de datos gestionada por el *kernel*, en la cual van a poder escribir y leer los procesos que se ejecuten en el sistema. Los mecanismos de sincronización para que no se produzca colisión son responsabilidad del *kernel*.
- A una cola de mensajes puede acceder cualquier proceso que tenga acceso al fichero y tenga los permisos necesarios, pudiendo leer y escribir en ella.
- Las colas de mensajes ofrecen un mecanismo de **comunicación y sincronización**. El nombre de la cola permite compartir ésta para que múltiples procesos puedan enviar y recibir datos de forma bloqueante o no bloqueante.

.

Colas de Mensajes POSIX

- Mecanismo de comunicación y sincronización
- Asigna a las colas nombres de ficheros
- Sólo pueden utilizar colas de mensajes procesos que comparten un mismo sistema de ficheros
- Tamaño del mensaje variable
- Flujo de datos bidireccional

Llamadas POSIX para colas de mensajes

- Creación y apertura de una cola de mensajes. **mq_open**
- Cierre y borrado de una cola de mensajes. **mq_close** **mq_unlink**
- Escritura en colas de mensajes. **mq_send**
- Lectura de colas de mensajes. **mq_receive**

Llamadas POSIX para colas de mensajes

- Las llamadas anteriores tienen una interfaz compleja.
- Emplearemos una clase llamada **colamsg** que hemos definido en **colamsg.hpp** empleando las funciones POSIX y que nos facilitará mucho la programación

```
#include < colamsg.hpp>
```

Creación de una cola.

- Creación de una cola de mensajes con nombre llamando al constructor colamsg.
- Ejemplo:

  colamsg cola*("micola", CREAT, WRONLY, numBytes***);**
- Crea cola con nombre "*micola*" para escribir solo y *numBytes* es el tamaño del mensaje
- El nombre de la cola sigue la convención del nombrado de archivos.
- Para el tercer parámetro:
  - RDONLY Crea una cola para lectura
  - WRONLY Crea una cola para escritura
  - RDWR Crea una cola para lectura y escritura

Abrir una cola.

- Apertura de una cola de mensajes ya creada al constructor colamsg.
- Ejemplo**:**

colamsg cola*("micola", OPEN, RDONLY***);**

- Abre la cola con nombre "*micola*" para escribir solo (*RDONLY*). La cola ha tenido que ser **previamente creada** por otro proceso.
- El nombre de la cola sigue la convención del nombrado de archivos.
- Para el tercer parámetro:
  - RDONLY Crea una cola para lectura
  - WRONLY Crea una cola para escritura
  - RDWR Crea una cola para lectura y escritura

Enviar y recibir mensajes

- Una vez habilitada la cola, los procesos fundamentalmente se dedicarán a **enviar y a recibir mensajes** por medio de ella.
- Los procesos que hayan habilitado una cola de mensajes pueden enviar y recibir mensajes indistintamente (siempre que los permisos lo permitan), no viéndose limitados a realizar una sola función.

Enviar mensajes

- Para enviar mensaje a cola:

cola.send(*punt,numBytes*);

Donde *punt* es un puntero a memoria donde se encuentra el mensaje de *numBytes* que deseamos enviar.

Recibir mensajes

- Para recibir mensaje de cola:

cola.receive(*punt,numBytes*);

Donde *punt* es un puntero a memoria donde se almacenará el mensaje de *numBytes* que viene por la cola.

Cerrar y borrar una cola de mensajes

- Cerrar la cola de mensajes cola:

  cola.close();

- Borrar una cola de mensajes con *unlink*(). Previamente cerraremos la cola de mensajes:

  cola.unlink();

Ejemplo. Paradigma Cliente-Servidor

- A continuación veremos un ejemplo que Ilustra el funcionamiento de las colas de mensajes. No precisa de la sincronización con semáforos pues la sincronización está implícita en la colas.
- Se trata de realizar la comunicación de dos procesos. El proceso **servidor** que recibe por una cola un mensaje con una cadena de caracteres del cliente, lo transforma a mayúsculas y lo devuelve de nuevo al cliente en mayúsculas por otra cola.

Ejemplo. Paradigma Cliente-Servidor

- El proceso **cliente** solicita al usuario una cadena de caracteres que envía al servidor por una cola, quedando a continuación a la espera de la cadena en mayúsculas en la otra cola.
- Al introducir * en el cliente finalizan ambos procesos cerrando y destruyendo el servidor los mecanismos IPC.
- IMPORTANTE: El **servidor** debe ejecutarse **primero** pues es el que crea las colas de mensajes que luego abrirá el cliente.

Ejemplo Cliente-Servidor. Servidor

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string>
#include "colamsg.hpp"
#define TAMCADENA 15
using namespace std;

int main ()
{
    string strMin;
    char cadMin[TAMCADENA], cadMay[TAMCADENA];
    //Creamos la cola de las mayusc. para escribir solo. TAMCADENA es el tamaño del mensaje
    colamsg colaMay("colaMayusculas", CREAT, WRONLY, TAMCADENA*sizeof(char));
    //Creamos la cola de las minusc. para leer solo. TAMCADENA es el tamaño del mensaje
    colamsg colaMin("colaMinusculas", CREAT, RDONLY, TAMCADENA*sizeof(char));
    do
    {
        colaMin.receive(cadMin, TAMCADENA*sizeof(char));
        cout << "Recibida cadena:  " << cadMin << endl;
        for(int i=0; i<TAMCADENA; i++) cadMay[i]=toupper(cadMin[i]);//transforma a mayusculas
        colaMay.send(cadMay,  TAMCADENA*sizeof(char));
    }
    while(cadMin[0]!='*'); // si el usuario introduce * como primer caracter el programa acaba
    colaMin.close();
    colaMay.close();
    colaMin.unlink();
    colaMay.unlink();
}
```

Ejemplo Cliente-Servidor. Cliente

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string>
#include "colamsg.hpp"
#define TAMCADENA 15
using namespace std;

int main ()
{
    string strMin;
    char cadMin[TAMCADENA], cadMay[TAMCADENA];

    colamsg colaMay("colaMayusculas", 0 , RDONLY); //Abrimos la cola de las mayusc. para leer solo
    colamsg colaMin("colaMinusculas", 0 , WRONLY); //Abrimos la cola de las minusc. para escribir solo
    do
    {
        cout << "Intro cadena (menos de " << TAMCADENA << "caract):" ;
        getline (cin,strMin);
        strcpy(cadMin,strMin.c_str());
        colaMin.send(cadMin,  TAMCADENA*sizeof(char));
        colaMay.receive(cadMay, TAMCADENA*sizeof(char));
        cout << "Recibido de servidor: " << cadMay << endl;
    }
    while(cadMin[0]!='*'); // si el usuario introduce * como primer caracter el programa acaba
    colaMin.close();
    colaMay.close();
}
```

Compilación

- Al utilizar la API de POSIX es necesario enlazar la librería **librt:**

```
g++ servidor.cpp -o servidor -lrt
g++ cliente.cpp -o cliente -lrt
```

Mecanismos de comunicación y sincronización entre procesos Memoria Compartida

Departamento de Ingeniería de Sistemas y Automática. EII.

Universidad de Valladolid

Memoria Compartida.

- Mediante este mecanismo varios procesos pueden **compartir parte de su espacio de direcciones,** de tal manera que la comunicación se realiza mediante simples accesos a memoria.
- Este mecanismo **sólo sirve para comunicación entre procesos, no para sincronización** de procesos. Por eso normalmente **precisa de semáforos para la sincronización**.
- A diferencia de las colas de mensajes el kernel no interviene en la transferencia de información entre los procesos. Los procesos hacen accesible una zona de memoria para leer y/o escribir entre ellos. Es por esto un **mecanismo de comunicación muy rápido**.

Memoria Compartida en POSIX.

- Las llamadas al sistema que ofrece POSIX para trabajar con memoria compartida son:

  - **shm_open()** llamada para crear una nueva región de memoria virtual compartida o bien para empezar a trabajar con una ya existente. Devuelve un descriptor de fichero que utilizaremos en las llamadas posteriores.
  - **ftruncate()** permite establecer el tamaño de la memoria compartida.
  - **mmap() y munmap()** permiten vincular y desvincular resp. regiones al / del espacio de direcciones del proceso. Una vez vinculada la región de memoria compartida al espacio virtual de un proceso, éste puede acceder al mismo utilizando las instrucciones normales de acceso a memoria, con lo que no son necesarias más llamadas al sistema para trabajar con la memoria compartida.
  - **close()** cierra el descriptor de fichero devuelto por **shm_open()**
  - **shm_unlink()** destruye la memo compartida

Memoria Compartida en POSIX.

- No obstante las llamadas anteriores tienen una interfaz compleja.
- Emplearemos una clase llamada **memocomp** que hemos definido en memocomp.hpp empleando las funciones POSIX y que nos facilitará mucho la programación

```
#include <memocomp.hpp>
```

Crear Memoria Compartida

Para crear una región de memoria compartida se utiliza el constructor de la clase **memocomp** introduciendo como segundo parámetro CREAT. Ejemplo:

memocomp memo ("*mimemo***", *CREAT*, *RDWR, numBytes*);**

Crea segmento memoria compartida con tamaño *numBytes* con nombre *'mimemo*' para lectura y escritura (*RDWR*).

Si queremos crear el segmento solo para lectura: *RDONLY*

Abrir Memoria Compartida

Para abrir una región de memoria compartida se utiliza el constructor de la clase **memocomp** introduciendo como segundo parámetro OPEN. Ejemplo:

```
memocomp memo("mimemo", OPEN, RDONLY, numBytes);
```

Abre un segmento de memoria compartida con tamaño *numBytes* con nombre *'mimemo*' para lectura solo (*RDONLY*). La memoria que vamos a abrir ha debido ser creada previamente por otro proceso.

Vinculación de la memoria al espacio de direcciones

- La función miembro `getPointer()` permite vincular la memoria compartida al espacio de direcciones del proceso. Ejemplo:

  punt= memo.getPointer();

- Una vez vinculada la región de memoria compartida al espacio virtual de un proceso, éste puede acceder normalmente mediante punteros, con lo que no son necesarias más llamadas al sistema para trabajar con la memoria compartida.

Cierre y destrucción de memo compartida

- Para cerrar la memoria compartida:

```
memo.cerrar(); //Cierra memoria compartida memo
```

- Para destruir la memoria compartida:

```
memo.unlink(); //Destruye memoria compartida memo
```

Ejemplo. Paradigma Cliente-Servidor

- A continuación veremos un ejemplo que Ilustra el funcionamiento de la **memoria compartida**. Incluye sincronización con semáforos.
- Se trata de realizar la sincronización de dos procesos. El proceso **servidor** que recibe una letra del cliente y se la devuelve en mayúsculas.

Ejemplo. Paradigma Cliente-Servidor

- El proceso **cliente** solicita al usuario un carácter y lo envía al servidor, quedando a continuación a la espera del carácter en mayúsculas.
- Al introducir * en el cliente finalizan ambos procesos cerrando y destruyendo el servidor los mecanismos IPC.
- IMPORTANTE: El **servidor** debe ejecutarse **primero** pues es el que crea los semáforos y memo compartida

Ejemplo. Servidor

```
#include <stdlib.h>
#include <iostream>
#include "semaforo.hpp"
#include "memocomp.hpp"

int main(void)
{
    char *letra;    // puntero a la letra en memo compartida

    //Creamos dos semaforos para la sincronizacion llamados sem1 y sem2 inicializados a 1 y 0 resp.
    semaforo s1("sem1", 0); // Creamos s1 inicializado a 0
    semaforo s2("sem2", 0); // Creamos s2 inicializado a 0
    //creamos una zona de memo compartida de lect escrit y tamaño 1 byte (sizeof char)
    memocomp memo("memo", CREAT, RDWR, sizeof(char));
    letra = (char *)memo.getPointer(); //vinculamos esta memo a un puntero
    while(*letra != '*')    //mientras no llegue un *
    {
        s1.down();//espera a que cliente escriba minusc en memo compartida
        *letra=toupper(*letra);
        cout << "servidor: " << *letra << endl;
        s2.up(); //da paso a cliente para que lea la mayuscula
    }
    s1.close();     //cierra semaforo s1 porque ya no lo usará este proceso
    s2.close();     //cierra semaforo s2 porque ya no lo usará este proceso
    memo.cerrar();  //cierra memo porque ya no lo usará este proceso
    //Eliminamos ipcs
    memo.unlink();  //destruye la memoria compartida que habíamos solicitado
    s1.unlink();    //destruye semaforo s1 porque ya no lo usará ningún proceso
    s2.unlink();    //destruye semaforo s2 porque ya no lo usará ningún proceso
}
```

Ejemplo. Cliente

```
#include <iostream>
#include "semaforo.hpp"
#include "memocomp.hpp"

int main(void)
{
    char *letra;    // puntero a la letra en memo compartida
    //Abrimos dos semaforos para la sincronizacion llamados sem1 y sem2 inicializados a 1 y 0 resp.por el servidor
    semaforo s1("sem1"); // abrimos s1. Tiene que tener el mismo nombre "sem1" que en servidor. Si no no conecta con el
    semaforo s2("sem2"); // abrimos s2. Tiene que tener el mismo nombre "sem1" que en servidor. Si no no conecta con el
    //Abriremos la zona de memo compartida de lect escrit y tamaño 1 byte (sizeof char) creada por servidor
    //Tiene que tener el mismo nombre "memo" que en servidor. Si no, no conecta con la zona de memo compartida creada
    memocomp memo("memo", OPEN, RDWR, sizeof(char));
    letra = (char *)memo.getPointer(); //vinculamos esta memo a un puntero
    while(*letra != '*')// Introducir * para finalizar
    {
        cout << "Intro caracter: ";
        cin >> (*letra);
        s1.up();    //da paso al servidor. Ya está disponible la letra en memo
        s2.down();  //espera a que el servidor escriba la letra en mayuscula
        cout << "En mayusculas --> " << *letra << endl;
    }
    s1.close();     //cierra semaforo s1 porque ya no lo usará este proceso
    s2.close();     //cierra semaforo s2 porque ya no lo usará este proceso
    memo.cerrar();  //cierra memo shm_fd porque ya no lo usará este proceso

}
```

Compilar con g++ con opción -lrt

- g++ servidor.cpp -o servidor -lrt
- g++ cliente.cpp -o cliente -lrt

Mecanismos de comunicación y sincronización entre procesos Semáforos

Departamento de Ingeniería de Sistemas y Automática. EII.

Universidad de Valladolid

Semáforos POSIX

- Pertenecen al estándar POSIX.1b y son relativamente recientes: estándar en 1993 (Real Time).
- Existen otros IPC en UNIX como SystemV.
- Mecanismo más sencillo que System V.
- Gran compatibilidad en UNIX.
- Posible uso en otros SSOO
- POSIX define dos tipos

Semáforos POSIX

POSIX define dos tipos:

- **Semáforos sin nombre**: Permiten sincronizar procesos emparentados que heredan el semáforo a través de la llamada *fork* y para hilos que se ejecutan dentro de un mismo proceso.
- **Semáforos con nombre**: El semáforo lleva asociado un nombre que sigue la convención de nombrado que se utiliza para ficheros. Este tipo se semáforos sirve para sincronizar procesos no emparentados.

Semáforos POSIX

- Los semáforos POSIX son semáforos contadores que tienen valores no negativos y deben ser inicializados antes de ser usados.
- La única diferencia entre semáforos con nombre y sin nombre es la forma de crear y destruir el semáforo.
- Todas las llamadas de creación y manejo de semáforos devuelven -1 en caso de error y 0 (normalmente) en otro caso
- El fichero de cabecera necesario para utilizar estos servicios es semaphore.h:

```
#include <semaphore.h>
```

Semáforos POSIX

POSIX ofrece una serie de llamadas al sistema par:

- Crear y abrir semáforos ***sem_open***
- Operaciones sobre semáforos ***sem_wait*** y ***sem_post***
- Operación de lectura de semáforo ***sem_getvalue***
- Cerrar semáforo ***sem_close***
- Destruir semáforo ***sem_unlink***

Semáforos POSIX

- No obstante las funciones anteriores tienen una interfaz compleja.
- Emplearemos una clase llamada semáforo que hemos definido en semaforo.hpp empleando las funciones POSIX y que nos facilitará mucho la programación

```
#include <semaforo.hpp>
```

Creación de un semáforo con nombre

- Para CREAR semáforo:

  semaforo S("misemaforo**",** valor**);**

- Crea un semáforo *S* con nombre *'misemaforo*‘ y lo inicializa a *valor.*
- Un semáforo con nombre posee un nombre, un dueño y derechos de acceso similares a los de un archivo.
- El nombre de un semáforo es una cadena de caracteres que sigue la convención de nombrado de un archivo.
- El constructor *semaforo* establece una conexión entre un semáforo con nombre y una variable de tipo semáforo.

Apertura de un semáforo con nombre

- Para ABRIR semáforo:

  semaforo S("misemaforo**");**

- Abre semáforo S con nombre 'misemaforo‘.

- El semáforo debe haber sido creado antes con el mismo nombre por otro proceso.

Operación bajar semáforo

- Para BAJAR el semáforo S:

S.down();

- Decrementa el contador del semáforo
- Si el valor previo del semáforo es 0, *down()* efectúa un bloqueo del proceso llamante hasta que el semáforo sea incrementado

Operación subir semáforo

- Para SUBIR semáforo:

S.up();

- Incrementa el contador del semáforo

Leer el valor de un semáforo

int **S.get** ();

- *get*: obtiene en *val* el valor del contador del semáforo

Cierre de un semáforo

- **Cerrar** un semáforo significa romper la asociación que se había creado entre un proceso y un semáforo al crearle/abrile.
- Se utiliza por parte de un proceso para indicar que ya ha terminado de utilizar el semáforo. Al cerrar el semáforo se elimina cualquier recurso del sistema utilizado por este proceso para el uso de este semáforo.
- Para cerrar el semáforo S:

```
S.close();
```

Borrado de un semáforo

- El **borrado** de un semáforo significa la eliminación de este semáforo del sistema. La destrucción del semáforo se pospone hasta que todos los procesos que lo estén utilizando lo hayan cerrado con la función *close()*.
- Para borrar el semáforo S:

```
S.unlink();
```

Ejemplo. Imprime de números por dos procesos

- A continuación veremos un ejemplo que Ilustra el funcionamiento de la sincronización con semáforos.
- Se trata de realizar la sincronización de dos procesos: uno que imprime números impares y otro que imprime los pares para que la salida sea ordenada.
- IMPORTANTE: Impares debe ejecutarse primero pues es el que crea los semáforos

Ejemplo. Programa imprime impares

```
//Imprime IMPARES
#include <iostream>
#include <stdlib.h>
#include <sys/wait.h>
#include "semaforo.hpp"
using namespace std;

int main(void)
{
    //Crea dos objetos semaf llamados sem1 y sem2 inicializados a 1 y 0 resp.
    semaforo s1("sem1",1); // Creamos sem inicializado a 1
    semaforo s2("sem2",0); // Creamos sem inicializado a 0

    for(int i=1; i<=20; i+=2)
    {
        s1.down(); // espera a que pares suba el semaf
        cout << "Proceso Impar: " << i << endl;
        sleep(1); // espera 1 seg. para que se aprecie la correcta sincroniz.
        s2.up();  // sube semaf para pares
    }

    // libera semáforos
    s1.close();
    s2.close();
    // destruye semaforos
    s1.unlink();
    s2.unlink();
}
```

Ejemplo. Programa imprime pares

```
//Imprime PARES
#include <iostream>
#include <stdlib.h>
#include <sys/wait.h>
#include "semaforo.hpp"
using namespace std;

int main(void)
{
    //Abre dos objetos semaf llamados sem1 y sem2
    //s1 y s2 ya estarán creados e inicializados por el proceso que imprime impares
    semaforo s1("sem1"); // abrimos sem 1
    semaforo s2("sem2"); // abrimos sem 2

    for(int i=2; i<=20; i+=2)
    {
        s2.down();  // espera que pares incremente el semáforo
        cout << "Proceso Pares: " << i << endl;
        sleep(1);   // espera 1 seg. para que se aprecie la correcta sincroniz.
        s1.up();    // sube semaf para impares
    }
    // libera semáforos
    s1.close();
    s2.close();
    //Los destruye el proceso impares que los ha creado
}
```

Compilación de código con semáforos POSIX

Compilar añadiendo **–lpthread**
Ejemplo:

```
g++  miprog.cpp  –o miprog –lpthread
```