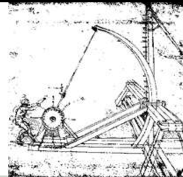
Pogramación Paralela y Distribuida

Licenciatura en
Ciencias de la Computación

UNCuyo – Facultad de Ingeniería

Early Adopters Awarded
Fall 2011 (NSF/IEEE)



Pogramación Paralela y Distribuida

CONCURRENCIA:
Herramientas y Conceptos

Anexo

Contenido

- Introducción
- Threads
- Procesos
- Comunicación y Sincronización
  - Semáforos
  - Variables Condicionales
  - Pipes
  - Sockets
- Resumen

Introducción

- Antes definimos…
  - *Algoritmo secuencial*
    - Secuencia de un solo flujo de sentencias para resolver un problema mediante la utilización de un unicomputador.
  - *Algoritmo paralelo*
    - Secuencia que indica cómo resolver el problema mediante la utilización de un sistema paralelo
- Y también vimos la importancia de la **concurrencia**
  - Habilidad de un algoritmo para ejecutar múltiples operaciones en un momento dado
- La concurrencia ¿sólo puede explotarse en los sistemas paralelos?

¡NO!

Introducción

- Hacia el final del curso veremos las técnicas de programación comúnmente aplicables a sistemas paralelos
- En este anexo revisaremos algunas técnicas de concurrencia ya conocidas aplicables tanto a sistemas uniprocesador como a sistemas paralelos que en combinación con las específicas representan una fuerte herramienta de programación.
- Nos familiarizaremos con los siguientes conceptos:
  - Concurrencia y threads
  - Procesos
  - Mecanismos de comunicación
  - Sincronización
- Trabajaremos en un entorno POSIX (UNIX, LINUX, etc.)

Contenido

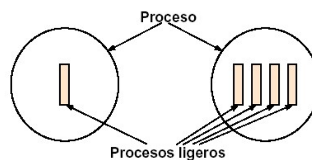
- Introducción
- Threads
- Procesos
- Comunicación y Sincronización
  - Semáforos
  - Variables Condicionales
  - Pipes
  - Sockets
- Resumen

Threads

- **Thread**
  - Es un mecanismo que habilita la concurrencia
    - Permite a un programa hacer más de una cosa por vez
    - Es un hilo o flujo de ejecución dentro del programa

  - Es un programa en ejecución que comparte la imagen de memoria y otras informaciones con otros threads.
  - Conceptualmente, un thread es parte de un proceso
    - Constituye una unidad de granularidad más fina
    - Posee un *thread ID*
    - Comparte los atributos globales del proceso que lo contiene
      - Para ejecutar diferentes threads no es necesario cambiar el contexto de la computadora (lo cual conllevaría tiempo)
    - Por ello se denominan "procesos livianos"

Threads



COMPARTE

- Espacio de memoria
- Variables globales
- Archivos abiertos
- Procesos hijos
- Temporizadores
- Señales y semáforos

NO COMPARTE

- Contador de programa
- Pila
- Registros
- Estado (ejec., listo, bloq.)

Threads

- Funciones para trabajar con threads
  - pthread_create
    - Crea un nuevo thread
  - pthread_join
    - Espera a que los threads creados terminen de ejecutarse
  - …

Threads

```
int pthread_create( pthread_t  *new_thread_id, pthread_attr_t
*attr, void *(*start_func)(void *), void *arg);
```

Añade un nuevo "thread" al proceso que la llama

- **Debe incluirse:** pthread.h
- **`new_thread_id`:** Identificador del *thread* (unsigned long int).
- **`attr`**: características de comportamiento del *thread,* como tamaño de la pila, prioridad, política de planificación, etc (dependen de la librería). Si se pasa NULL el nuevo *thread* tendrá los atributos por defecto.
- **`start_func`**: rutina que contiene el código del *thread* (su main).
- **`arg`**: argumento que puede pasarse a start_func.
- **Retorno**: 0 si pudo crearse el nuevo thread, != 0 si error.

Threads

```
int pthread_exit(void *status);
```

Permite a un thread terminar su ejecución indicando el estado de terminación del mismo

- **`status:`** Si hay algún otro *thread* esperando a que este acabe entonces dicho *thread* recibirá el estado de finalización que se indique en esta variable (si no este valor se pierde).

Threads

```
int pthread_join( pthread_t  target_thread, void **status);
```

Suspende la ejecución del "thread" que utiliza la llamada hasta que el "thread" con identificador `target_thread` acaba. Devuelve el estado.

- **`target_thread:`** Identificador del *thread* a esperar, debe pertenecer al mismo proceso y no haber sido desasociado ni convertido en *daemon*.
- **`status:`** valor retornado por el *thread* esperado en un return o en la llamada *`pthread_exit`*.
- **Retorno**: 0 si tiene éxito, != 0 si error.

```
pthread_t pthread_self();
```

Retorna el identificador del "thread" que la llama

Threads

- Sincronización y Regiones Críticas
  - Cuando se programa con threads surgen regiones críticas dado que:
    - Comparten toda la información global del proceso
    - Se desconoce el orden y el intervalo de tiempo que se ejecutará cada thread
  - Cada vez que se ejecuta el programa, la secuencia de ejecución puede ser diferente según lo antedicho
  - Para asegurar un comportamiento coherente en cada ejecución es necesario introducir primitivas de sincronización
    - Lo cual dificulta la programación con threads

Threads

- Conceptos
  - Recurso crítico
    - Es cualquier recurso compartido por más de un programa.
  - Región crítica
    - Fragmento de código en el que se accede y/o modifican ciertos recursos críticos (recursos compartidos). Una sección crítica no puede ser ejecutada por más de un thread o proceso a la vez

Threads

Ejemplo 1

- Recurso Crítico: cruce de ferrocarril
- Región Crítica(tren): avanzar en el cruce con la calle.
- Región Crítica(coche): cruzar las vías
- Solución: barreras o en este caso un semáforo

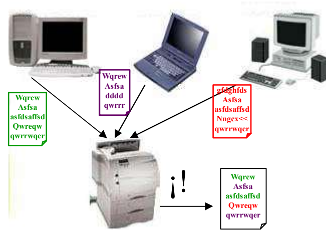
Threads

Ejemplo 2

- Recurso Crítico: cruce de calles
- Región Crítica(vehículo): cruzar la calle

Threads

Ejemplo 3



- Recurso Crítico: impresora
- Región Crítica(pc): imprimir el documento

Threads

- En estos ejemplos y en muchas otras situaciones debe asegurarse que la región crítica sea tratada por un solo flujo a la vez
- Para ello se requiere de exclusión mutua
  - Más adelante veremos semáforos y variables condicionales que permiten implementar exclusión mutua

LICPaD (UTN-FRM) Dr. Germán Bianchini – Dra. Paola Caymes Scutari

Contenido

- Introducción
- Threads
- Procesos
- Comunicación y Sincronización
  - Semáforos
  - Variables Condicionales
  - Pipes
  - Sockets
- Resumen

Procesos

- **Proceso**
  - Es un programa en ejecución
  - En el sistema, cada proceso tiene asignado un espacio de memoria en el que se incluyen sus variables, la administración de su E/S, memoria extra, el programa que debe ejecutar, etc.
  - En general, cada ventana que está abierta en un momento determinado representa un proceso diferente (editor de textos, correo electrónico, calculadora, navegador de internet, reproductor de música, programa de aplicación, etc.)

Procesos

- Cuando se desea que parte de un programa se ejecute en un proceso independiente, el programa original o ***proceso padre*** debe:
  - Crear el nuevo proceso o ***proceso hijo*** ***(fork())***
    - Para el cual debe definir su funcionalidad (***exec()?***)
  - Continuar ejecutando su propia funcionalidad
  - Esperar a que el proceso hijo termine de ejecutarse antes de terminar su propia ejecución ***(wait())***

Procesos

- fork()
  - Función que crea un proceso hijo idéntico al proceso padre
  - Diferencias entre padre e hijo
    - Identificador de proceso (PID)
    - Porción de código que ejecuta, dado que…

```
pid_hijo = fork ();
if (pid_hijo != 0)
    // código del proceso padre
else
    // código del proceso hijo
```

fork() es una función que devuelve dos resultados:
- al *proceso padre* le devuelve el **PID** del proceso **hijo**
- al *proceso hijo* le devuelve **0**

Procesos

- fork()
  - Deben definirse **separadamente** las funcionalidades del padre y del hijo para que cada nuevo proceso ejecute el código correspondiente

```
pid_hijo = fork ();
if (pid_hijo != 0)
    // código del proceso padre
else
    // código del proceso hijo
```

fork() es una función que devuelve dos resultados:
• al *proceso padre* le devuelve el **PID** del proceso **hijo**
• al *proceso hijo* le devuelve **0**

Procesos

- **exec()**
  - Familia de funciones que reemplaza el código actual del proceso por el de otro programa
    - *execv(), execl(), execlp(), execvp(), etc.*
  - Normalmente se utiliza una función exec en el código del proceso hijo, para separar su funcionalidad en otro programa
    - Así en el código del proceso padre se reduce la porción de código definida para el proceso hijo
      - Especialmente útil cuando el programa es extenso
      - Ofrece flexibilidad para definir el código del hijo

Procesos

- **wait()**
  - Función que se utiliza en el código del proceso padre para esperar la terminación del proceso hijo
    - Permite obtener información de terminación de los procesos hijos
    - Favorece la correcta terminación del programa en conjunto
    - El proceso padre puede realizar cálculos dependientes de la finalización de los procesos hijos

Threads VS Procesos

- ¿Proceso o Thread?
  - Cuando se ejecuta un programa el SO crea un proceso dentro del cual crea un thread que ejecuta el programa secuencialmente
  - Dicho thread puede crear threads adicionales
  - Todos los threads ejecutan el mismo programa en el mismo proceso
    - Cada thread puede ejecutar una parte diferente del programa

Threads VS Procesos

- ¿Proceso o Thread?
  - Un proceso hijo (creado con *fork()*) es una copia idéntica de su proceso padre
    - Se diferencian en el PID y en lo que devuelve la llamada *fork()* (que se utiliza para definir la porción de código que cada uno continuará ejecutando)
  - Ambos procesos (padre e hijo) pueden modificar sus propios atributos (variables, programa, etc) sin que ello afecte al otro proceso, ya que son independientes

Threads VS Procesos

- ¿Proceso o Thread?
  - Un thread (creado con *create()*) comparte los atributos globales del thread que lo creó dentro del proceso
  - Si un thread modifica un atributo global (por ejemplo una variable) todos los threads verán dicho valor modificado
  - Para ejecutar diferentes threads no es necesario cambiar de contexto, todos son parte del mismo proceso

Threads VS Procesos

- ¿Cómo decidir si usar procesos o threads?

  Depende de:
  - la granularidad de las tareas que pueden realizarse simultáneamente
    - Si cada subtarea de un programa es suficientemente compleja, es conveniente utilizar un proceso para cada tarea, ya que asegura un tiempo de CPU dedicado exclusivamente a la tarea
  - el grado de acoplamiento
    - Si se comparten muchos datos, puede hacerse uso de memoria compartida que evita envío y recepción de mensajes, para ello se utilizan threads
  - A la vez, y siempre dependiendo de las características del problema, pueden combinarse ambas posibilidades

Contenido

- Introducción
- Threads
- Procesos
- Comunicación y Sincronización
  - Semáforos
  - Variables Condicionales
  - Pipes
  - Sockets
- Resumen

Semáforos

- **Semáforos**
  - Edsgar Dijkstra abstrajo el concepto de primitivas de exclusión mutua en el concepto de **semáforo**
  - Los semáforos proveen una solución para el problema de la exclusión mutua
  - Un semáforo es una variable protegida cuyo valor puede obtenerse y modificarse sólo a través de las operaciones *P()* y *V()*, y una operación de inicialización denominada *semaphore_initialize()*

Semáforos

- P y V derivan de 2 palabras alemanas:
  - Passieren *("pasar") y* Proberen *("testear")*
  - Vrijmagen ("*liberar*") Verhogen ("*incrementar*")
- Los semáforos binarios sólo pueden asumir los valores 0 o 1.
- Los semáforos n-arios pueden asumir n valores no negativos.

Semáforos

- La operación P(S) sobre el semáforo S se define de la siguiente manera:

```
si (S > 0 ) entonces
        S := S - 1
sino
        Wait por S
```

- La operación V(S) sobre el semáforo S se define como:

```
Si (uno o más procesos esperan (Wait) por S) entonces
        permitir el paso a un proceso
else
        S := S + 1
```


Semáforos

Se asume que:

- P() y V() son operaciones atómicas.
- Hay una cola FIFO para los procesos que esperan por S por medio de P()
- La exclusión mutua de S está forzada y asegurada por P() y V().
- P() y V() aseguran la no inanición.

Semáforos

- Interpretaciones de semáforo
  - El valor de un semáforo representa la cantidad de recursos disponibles

  - Si el valor del semáforo es negativo, el número de procesos bloqueados viene dado por |S|
  - Si se desea permitir a más de un proceso estar en la sección crítica, entonces el valor inicial del semáforo deberá ser dicho número

Mutex

- **Mutex** (Mutual Exclusion – Exclusión Mutua)
  - Un mutex es un mecanismo de sincronización indicado para threads
  - Es un semáforo binario con dos operaciones atómicas:
  - *lock(m)* Intenta bloquear el mutex, si el mutex ya está bloqueado el proceso se suspende (*P*)
  - *unlock(m)* Desbloquea el mutex, si existen procesos bloqueados en el mutex se desbloquea a uno (*V*)

Mutex

```
int pthread_mutex_init(pthread_mutex_t  *mp, pthread_mutexattr_t *attr);
```

Inicializa el mutex `mp` con los atributos indicados en `attr`. En caso de que se quiera inicializar con los atributos por defecto basta con utilizar la macro `PTHREAD_MUTEX_INITIALIZER`.

- **`mp:`** nombre del mutex inicializado.
- **`attr:`** características de comportamiento del mutex (dependen de la librería), si se pasa **NULL** el nuevo mutex tendrá los atributos por defecto.
- **Retorno**: 0 si pudo crearse el nuevo mutex, != 0 en caso de error.

```
int pthread_mutex_destroy( pthread_mutex_t  *mp);
```


Mutex

```
int pthread_mutex_lock( pthread_mutex_t  *m);
```

Esta llamada cierra –bloquea- el mutex `m`. Si el mutex ya está cerrado –bloqueado-, el thread que realiza la llamada se bloquea hasta que el mutex está disponible.

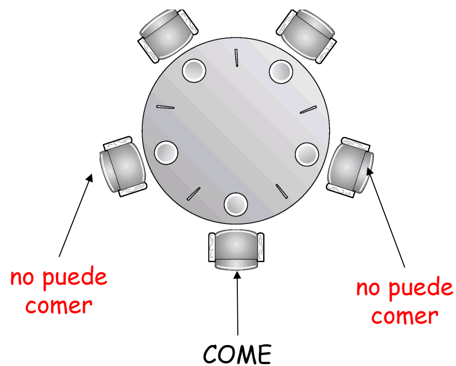
```
int pthread_mutex_unlock( pthread_mutex_t  *m);
```

Esta llamada libera el mutex `m`. Si existen threads bloqueados en el mutex, la política de “scheduling” determinará cual accederá el mutex.

Retorno: 0 si todo va bien, != 0 si error.

Ejemplo Mutex

- **Los Filósofos comensales**
  - 5 filósofos se juntan a cenar comida china en una mesa circular
  - Hay sólo 5 palitos en la mesa intercalados entre los platos
  - Cada filósofo necesita un par de palitos para comer (el izquierdo y el derecho)
  - La participación de cada filósofo consiste en ciclos alternativos de comer y pensar



Ejemplo Mutex

```
#include <stdio.h>
#include <pthread.h>
pthread_t tid[5];
pthread_mutex_t palito[5];
main()
{
  int i;
  for (i=0;i<5;i++) {
       pthread_mutex_init(&palito[i],NULL);
   }
  for (i=0;i<5;i++)
    pthread_create(&tid[i],NULL,(void *) filosofo,(void *)i);
  for (i=0;i<5;i++)
    pthread_join(tid[i], NULL);
}
```

Ejemplo Mutex

```
void filosofo(void *arg)
{ int i = (int) arg;
  while(1) {
    PENSAR();     //wait(1)
    //tomar_tenedor_izq;
    pthread_mutex_lock(&palito[i]);
    //tomar_tenedor_der();
    pthread_mutex_lock(&palito[(i+1)%5]);
    COMER();      //wait(1)
    pthread_mutex_unlock(&palito[i]);
    pthread_mutex_unlock(&palito[(i+1)%5]);
  }
}
```


Ejemplo Mutex

- Este ejemplo puede presentar algunos problemas:
  - Todos los filósofos levantan el palito de su derecha a la vez
  - Cada filósofo tiene un palito y espera eternamente a que se libere el par

Cond

- **Cond** (**Cond**itional Variable – Variable Condicional)
  - Es una variable de sincronización que se utiliza asociada a un mutex
  - Permite bloquear un thread hasta que ocurra algún suceso.
    - Por ello se denominan “*condicionales*”
    - Es necesario utilizar una o más variables compartidas utilizadas como predicados lógicos
    - Los predicados lógicos:
      - Son consultados por cada thread para decidir si debe bloquearse o no
      - Son modificados por los threads cuando cambie la condición representada por el predicado

Cond

- **Cond** (**Cond**itional Variable – Variable **Cond**icional)
  - Tienen dos operaciones atómicas para esperar y señalizar
    - *wait(c, m)*, bloquea al thread que ejecuta la llamada y lo expulsa del mutex m dentro del cual se ejecuta y al que está asociado la variable condicional c, permitiendo que algún otro proceso adquiera el mutex.
    - *signal(c)*, desbloquea a un proceso suspendido en la variable condicional c. El proceso que se despierta debe competir nuevamente por el mutex.

Cond

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);

    pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
```

Inicializa la variable condicional `cond` con los atributos `attr`. Si `attr` es NULL, se utilizan los atributos por defecto, igual que en la segunda opción de inicialización

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Destruye la variable condicional `cond`. Es recomendable destruir una variable condicional sólo si no existen threads bloqueados por ella.

Cond

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

El thread que hace la llamada se bloquea esperando que otro thread cambie la variable condicional (el mutex asociado queda automáticamente liberado).

```
int pthread_cond_signal( pthread_cond_t  *cond);
```

Desbloquea a uno de los threads bloqueados en la variable condicional.

```
int pthread_cond_broadcast( pthread_cond_t  *cond);
```

Desbloquea todos los threads bloqueados en la variable condicional pero luego por una política se decide cuál continúa.

Retorno: 0 si todo va bien, != 0 si error.

Ver ejemplo: http://www.llnl.gov/computing/tutorials/pthreads/#ConditionVariables

Mutex y Cond

- Permiten implementar sincronización entre threads
- Un **mutex** (o varios) se utiliza para proteger o delimitar una región crítica
- Una **variable condicional** se utiliza para bloquear la ejecución de un thread cuando una vez dentro de la región crítica no puede continuar su ejecución en caso de que no se cumpla alguna condición; asimismo, libera el mutex para que otros threads puedan acceder a la región crítica
- Normalmente, se utiliza una **variable compartida** asociada a cada variable condicional, como mecanismo de decisión para bloquear o continuar la ejecución del thread

Contenido

- Introducción
- Threads
- Procesos
- Comunicación y Sincronización
  - Semáforos
  - Variables Condicionales
  - Pipes
  - Sockets
- Resumen

Pipes

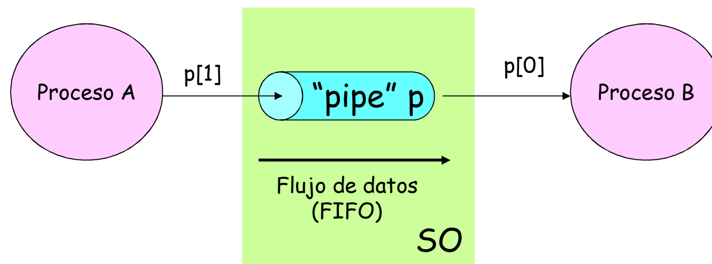
- **Pipes o Tuberías**
  - Mecanismo de comunicación y sincronización
  - Pseudoarchivo mantenido por el SO
  - Cada proceso ve al pipe como un conducto con dos extremos:
    - Un extremo se utiliza para escribir o insertar datos
    - El otro extremo se utiliza para leer o extraer datos
  - Las operaciones de lecto/escritura son análogas a las utilizadas para manejo de archivos

Pipes

- Flujo de datos:
  - Unidireccional
    - Los datos fluyen en un solo sentido.
    - Un proceso será el emisor y otro el receptor.
    - Si se precisa comunicación bidireccional será necesario utilizar dos pipes
  - FIFO
    - Los datos se extraen del pipe en el mismo orden en que se insertaron

Pipes

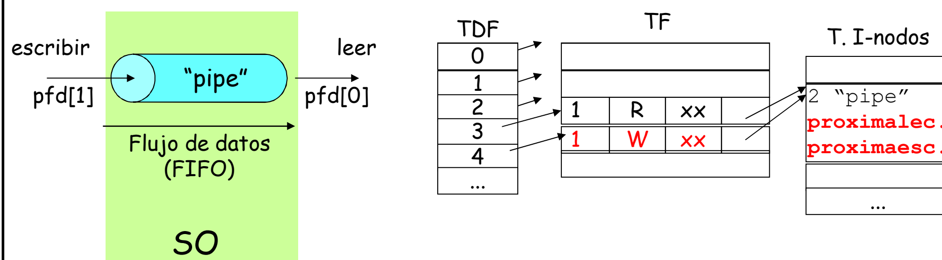
- Comunicación mediante un pipe:



Pipes

`int pipe(int pfd[2])` → Creación de un mecanismo de comunicación entre procesos llamado "pipe".

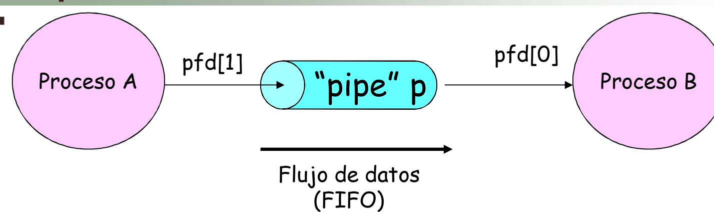
Retorna:
- -1 si error
- 0 si correcto. En `pfd[]` devuelve dos descriptores de acceso al canal de comunicación, uno para escribir (`pfd[1]`) y uno para leer (`pfd[0]`).



Pipes

- Un pipe es un mecanismo de comunicación con almacenamiento
  - El tamaño varía de acuerdo al SO
    - Suele ser de 4 Kbytes.
- Sobre un pipe puede haber múltiples procesos lectores y/o escritores
- Por ello, los pipes se implementan como regiones de memoria compartida entre los procesos que los utilizan
- Antes definimos a los pipes como mecanismos de **comunicación** y **sincronización**
  - A continuación veremos por qué…

Pipes



read(pfd[0],..,..)

Una lectura de "pipe" vacío, hará que el proceso se bloquee hasta que existan datos para leer en el "pipe". Una lectura de un "pipe" el cual no tiene ningún descriptor de escritura abierto, devuelve 0 (EOF).

write(pfd[1],..,..)

Una escritura en un "pipe" lleno, hará que el proceso se bloquee hasta que exista espacio en el "pipe".

Sockets

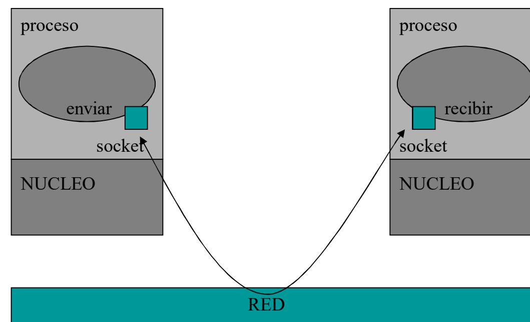
- Socket
  - Dispositivo de comunicación bidireccional que puede usarse para comunicarse con otros procesos locales o remotos
  - Mecanismo comúnmente usado por diversas aplicaciones de Internet como Telnet, FTP, talk, rlogin, etc.

Sockets

- Un socket es un punto final de comunicación (dirección IP y puerto)
- Abstracción que:
  - Ofrece interfaz de acceso a los servicios de red en el nivel de transporte
    - Protocolo TCP
    - Protocolo UDP
  - Representa un extremo de una comunicación bidireccional con una dirección asociada

Sockets

- Utilizando la interfaz que plantean los sockets, dos procesos que deseen comunicarse deben crear un socket cada uno de ellos. Cada socket se encuentra asociado a una dirección y permite enviar, o recibir, datos a través de él



Sockets

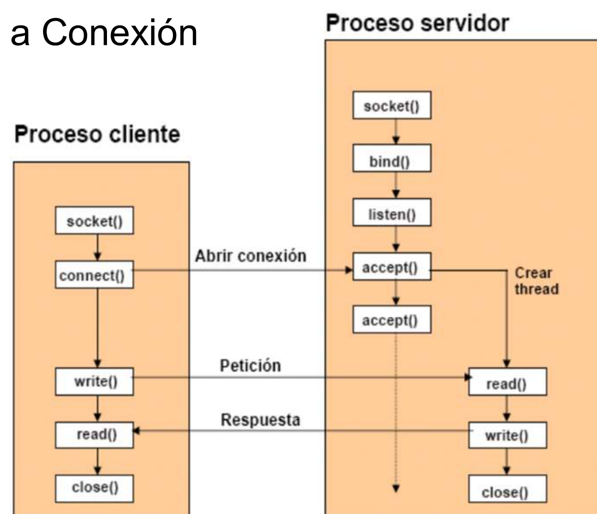
- Al crear un socket debe especificarse:
  - **Estilo de comunicación**: cómo se tratan los datos a enviar y cuál es la cantidad de conexiones que podrá mantener
    - Orientado a conexión
      - Garantiza el envío ordenado de los paquetes
      - El receptor resolicita los paquetes perdidos
      - Ejemplo: llamado telefónico
    - NO orientado a conexión o datagrama
      - Pueden desordenarse los paquetes
      - Ejemplo: correo postal
  - **Espacio de direcciones**: cómo se escribe la dirección
    - Espacio de direcciones de Internet: dirección IP + puerto
    - Espacio de direcciones local: nombre de archivo
  - **Protocolo**: cómo se transmiten los datos
    - TCP/IP
    - UDP

Sockets

- Llamadas a sistema para socket
  - socket
    - Crea un socket
  - close
    - Destruye un socket
  - connect
    - Crea una conexión entre dos sockets
  - bind
    - Asocia un socket con una dirección
  - listen
    - Configura un socket para aceptar conexiones
  - accept
    - Acepta una conexión y crea un nuevo socket para la conexión

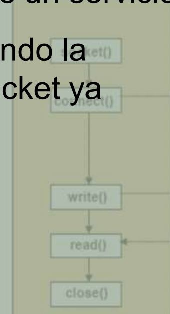
Sockets

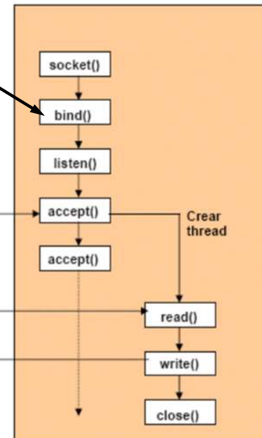
- Orientado a Conexión



Sockets

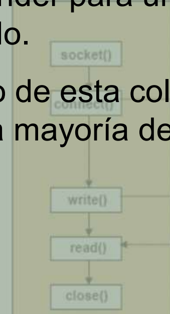
- ¿Para que sirve un "bind"?
  - Para que otros procesos puedan conectarse con un proceso que ofrece un servicio.
  - Se realiza exportando la dirección de un socket ya creado.

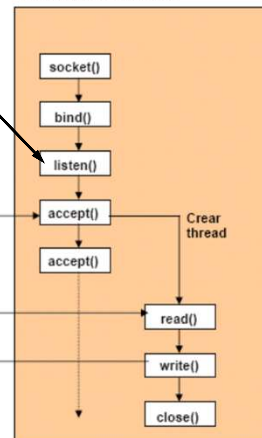




Sockets

- ¿Para que sirve un "listen"?
  - Permite definir el tamaño de la cola de peticiones de conexión pendientes de atender para un socket determinado.
  - El tamaño máximo de esta cola suele ser 5 para la mayoría de los sistemas

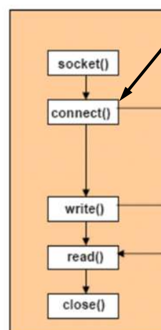




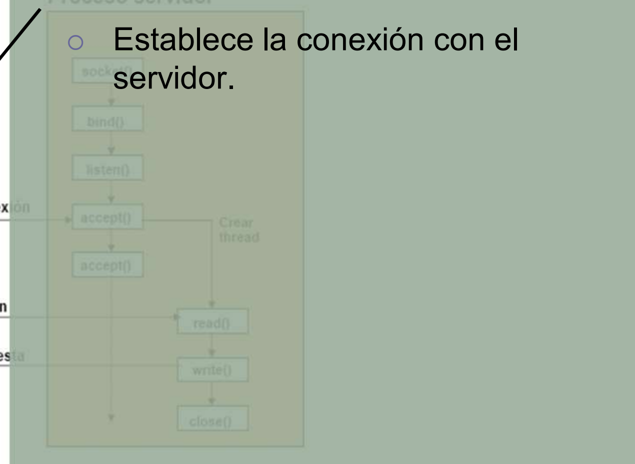
Sockets

- **¿Para que sirve un "accept"?**
  - Bloquea el proceso que llama hasta que un cliente solicita una conexión.
  - Retorna un nuevo socket (mismas propiedades socket servidor) que deberemos utilizar para comunicarnos con el cliente junto con su dirección (la del cliente).

Sockets



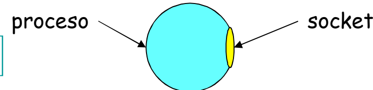
- **¿Para que sirve un "connect"?**
  - Establece la conexión con el servidor.



Sockets

```
int socket( int domain, int type,int protocol);
int close (int s);
```

Devuelve un descriptor de socket

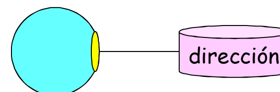


- **Debe incluirse:** `sys/types.h y sys/sockets.h` (en este orden)
- **`domain:`** Indica la familia de protocolos en la que va a realizarse la comunicación `(AF_UNIX, AF_INET)`
- **`type:`** Indica si va a utilizar un protocolo fiable orientado a conexión `(SOCK_STREAM)` o no `(SOCK_DGRAM)`.
- **`protocol:`** Lo normal es que para una familia y tipo dado exista sólo un protocolo definido, y por tanto este parámetro será 0.
- **`s:`** Descriptor del *socket* que se desea cerrar.
- **Retorno**: La función retorna -1 si hay error (usar *perror* para saber cual) o un *socket descriptor* si no.

Sockets

```
int bind( int s, const struct sockaddr *name,int namelen);
int unlink(const char *path);
```

Asocia un socket a una dirección



- **Deben Incluirse**: `sys/types.h y sys/socket.h`
- **`s:`** Descriptor del *socket* al que quiere asociarse una dirección.
- **`name:`** Dirección a la que se asocia el *socket*. Si pertenece a la familia `INET` será una dirección IP más un puerto. Si pertenece a la familia `UNIX` será el nombre de un archivo.

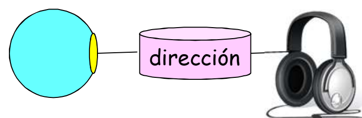
```
Ej: struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    bind( s, (struct sockaddr *)&serv_addr, sizeof( sa ));
```

- **`namelen:`** tamaño en bytes de la estructura `name`.
- **Retorno**: -1 si error (usar *perror*), 0 si no.

Sockets

```
int listen(int socket, int backlog);
```

Habilita al socket para recibir conexiones

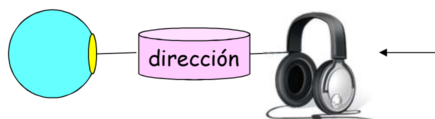


- **Deben Incluirse**: `sys/types.h y sys/socket.h`
- **`s:`** Descriptor del *socket* que se habilita para recibir conexiones.
- **`backlog:`** Cantidad de conexiones que el socket puede atender a la vez. pu
- **`namelen:`** tamaño en bytes de la estructura *`name`*.
- **Retorno**: -1 si error (usar *perror*), 0 si no.

Sockets

```
int connect(int s, const struct sockaddr *addr, socklen_t
addr_len);
```

Solicita conexión al socket



- **Deben Incluirse**: `sys/types.h y sys/socket.h`
- **`s:`** Descriptor del *socket* al que se solicita conexión.
- **`addr:`** Apunta a una estructura sockaddr que contiene la dirección a la cual se desea realizar la conexión.
- **`addr_len:`** tamaño en bytes de la estructura *`addr`*.
- **Retorno**: -1 si error (usar *perror*), 0 si no.

Sockets

```
int accept (int s, struct sockaddr *addr, socklen_t
*addr_len);
```

Permite atender/aceptar la primera conexión de la cola de conexiones pendientes.
Crea un nuevo socket "réplica" del original y le asigna un nuevo descriptor para la atención exclusiva de esa conexión.

- **Deben Incluirse**: `sys/types.h` y
- `s:` Descriptor del *socket* al que se s
- `addr:` Apunta a una estructura socka
desea realizar la conexión.
- `addr_len:` tamaño en bytes de la estr
- **Retorno**: -1 si error (usar *perror*), 0 si no

El programador es responsable de proveer el programa de las estructuras y funciones necesarias para la atención de múltiples peticiones de conexión.

Sockets

```
ssize_t recv(int s, void *buf, size_t len, int flags);
```

Recibir información

IDEM read() + flag

- **Deben incluirse**: `sys/types.h` y sy
- `s:` Descriptor del *socket* creado por el p ...be.
- `buff:` Estructura en la que se almacenar ...ensaje recibido (el parámetro real no tiene por que ser obligatoriamente de tipo char *).
- `len:` Tamaño de `buff` en bytes.
- `flags:` Especifica el tipo de mensaje recibido. En general, en lo que a nosotros respecta utilizaremos 0.
- **Retorna**: -1 si error (usar *perror*), número de bytes recibidos si no.

Sockets

```
ssize_t send(int s, const void *msg, size_t len, int
flags);
```

Enviar información

IDEM write() + flag

- **Deben Incluirse**: `sys/types.h y sy`
- **`s:`** Descriptor del socket por el que se e ... saje.
- **`msg:`** Puntero al mensaje (no tiene por que ... necesariamente de tipo `char *`)
- **`len:`** longitud (en bytes) del mensaje.
- **`flags:`** Especifica el tipo de mensaje recibido. En general, en lo que a nosotros respecta utilizaremos 0.
- **Retorno**: -1 si error (usar *perror*), número de bytes enviados si no.

Pipes y Sockets

- Son dos mecanismos diferentes de comunicación y sincronización de procesos
- Los pipes sólo pueden usarse dentro de una máquina, mientras que los sockets permiten comunicación intermáquinas

Contenido

- Introducción
- Threads
- Procesos
- Comunicación y Sincronización
  - Semáforos
  - Variables Condicionales
  - Pipes
  - Sockets
- Resumen

Resumen

- Herramientas para Concurrencia
  - Threads
    - Mutex y Cond
  - Procesos
    - Pipes y Sockets

Tarea para la casa…

- **Repasar los conceptos presentados**
- **Ampliar la información presentada en clases sobre herramientas para programar concurrentemente**
- **Resolver la Práctica**

LICPaD (UTN-FRM) Dr. Germán Bianchini - Dra. Paola Caymes Scutari

Bibliografía

- **Carretero Pérez J., De Miguel Anasagasti P., García Caballeira F., Pérez Costoya F.**, *"Sistemas Operativos-una visión aplicada". McGraw-Hill, 2001.*
- **Mitchell Mark, Oldham Jeffrey, Samuel Alex.** "*Advanced Linux Programming*", New Riders Publishing. 2001 (también disponible on-line en http://www.advancedlinuxprogramming.com/) (Capítulos 3, 4 y 5)
- **Silberschatz A., Galvin P.**, *"Operating Systems Concepts".* Addison-Wesley, 1998.

LICPaD (UTN-FRM) Dr. Germán Bianchini - Dra. Paola Caymes Scutari