

Unix: Simple Sockets Threads

Simple Sockets

- Para evitar posibles complicaciones con el uso de sockets, se puede utilizar la biblioteca jsocket.h:

```
int j_socket();
int j_bind(int, int);
int j_accept(int);
int j_connect(int, char *, int);
```

- Revisar jsocket.c

Simple Sockets

- Resumen de Operaciones:

- Crear un socket:

```
s = j_socket();
```

- Asignar un port a un socket (incluye el listen):

```
status=j_bind(s, port); /* OK: 0, Error: -1*/
```

- Esperar una conexión (o aceptar conexión pendiente):

```
t = j_accept(s); /* -1 si error */
```

Simple Sockets

- Iniciar conexión:

```
status = j_connect(s, host, port); /* OK: 0,
                                     error: -1 */
```

- Ejemplo de servidor: jservidor.c
- Ejemplo de cliente: jcliente.c
- Para compilar, primero se compila la biblioteca:

```
gcc -c jsocket.c
```

- Obtenemos el archivo jsocket.o, el cual se utiliza para compilar los programas

Simple Sockets

- Para compilar los programas cliente y servidor:

```
gcc -Wall -pedantic -o jservidor jservidor.c
jssocket.o
```

- Para compilar en las estaciones se deben agregar las bibliotecas: -lnsl -lsocket

Threads

- Múltiples ramas de ejecución en un mismo programa
- Linux, tal como otros sistemas operativos, son capaces de ejecutar múltiples procesos simultáneamente
- Todos los procesos tienen al menos un thread de ejecución
- Diferencia entre fork y threads:
 - Cuando llamamos a fork, una nueva copia del proceso es creada, con sus propias variables y PID
 - En fork el nuevo proceso se administra independientemente

Threads

- Diferencia entre fork y threads:
 - Cuando se crea un nuevo thread, este tendrá su propio stack de variables locales, pero compartirá las variables globales, file descriptor, manejadores de señales
 - Generalmente el crear un nuevo thread es menos costoso que crear un nuevo proceso
- Algunas veces es útil hacer un programa que parezca hacer 2 cosas a la vez:
 - Contador de palabras en tiempo real, mientras se mantiene la edición del texto.
 - Un thread se encarga de lo que el usuario escribe, generando la edición
 - Otro thread se encarga de contar las palabras

Threads

- Ejemplo de thread:
 - Un tercer thread (o el primero), se podría encargar de leer una variable compartida en la cual está el contador de palabras, para informarlo al usuario
- Los threads también tienen desventajas:
 - Podrían no estar disponibles
 - Es muy probable en una aplicación multithread el tener errores con variables compartidas no intencionalmente
 - El “debug” de un programa multithread es mucho más difícil que en un programa single thread

Threads

- Un programa podría dividir una operación matemática grande en 2 partes:
 - La ejecución se realizaría en 2 threads distintos
 - No necesariamente será mas rápida la ejecución en una máquina con 1 procesador
- Existe un grupo de funciones que podemos utilizar para crear programas con threads, incluidas en <pthread.h>
- Para crear un thread:

```
int pthread_create(pthread_t * thread,
                  pthread_attr_t * attr, void *
                  (*start_routine)(void *), void * arg);
```

Threads

- El primer argumento, un puntero a "pthread_t", se utilizará para identificar al thread después de su creación
- El segundo argumento asigna ciertas características al thread (generalmente es NULL)
- Los dos últimos argumentos indican al thread que función es la que debe utilizar y con que argumentos
- Con la llamada:

```
void * (*start_routine)(void *)
```

- Estamos diciendo que la función puede recibir y retornar cualquier cosa

Threads

- Si se utiliza fork, la ejecución continúa en el mismo lugar en donde se llamó, con un código distinto de retorno
 - Utilizando "pthread_create" estamos diciendo explícitamente la función en donde el thread debe comenzar su ejecución
- Retornará 0 en caso de éxito, o un número de error en caso contrario
- Cuando un thread termina, se debe llamar:

```
void pthread_exit(void *retval);
```

Threads

- Para esperar un thread se utiliza:

```
int pthread_join(pthread_t th,
                 void **thread_return);
```

- Se suspende la ejecución (similar a wait) hasta que el thread identificado por "th" termine
 - Un thread terminará si llama a pthread_exit o es anulado desde otro lado
- El valor de "thread_return" será el argumento dado a "pthread_exit"

Threads

- Ejemplo: thread.c
- En la compilación se debe agregar: -lpthread
- En el ejemplo:
 - La idea es modificar la variable global "message"
 - El thread que se lanzará ejecutará la función llamada "thread_function"
 - Después de llamar exitosamente a "pthread_create" se tendrán 2 threads en ejecución
 - El thread original (main), continuará la ejecución del código que sigue a "pthread_create"
 - El nuevo thread comenzará su ejecución en la función dada como parámetro

Threads

- El nuevo thread:
 - Comienza ejecutando "thread_function", imprime sus argumentos, "duerme" por unos segundos, actualiza una variable global y retorna un string al thread principal
 - El nuevo thread escribe el **mismo** arreglo que el thread principal accede, esto no es posible de hacer con fork
- Ejecución simultánea:
 - Hasta el momento no tenemos más herramientas de sincronización que una simple variable compartida
 - Ejemplo thread2.c

Threads

- En el ejemplo:
 - Cada thread indica al otro cuando debe imprimir o "dormir" cambiando el valor de la variable "run_now"
 - Cuando un thread ejecuta, debe esperar a que el otro thread modifique la variable para volver a ejecutar
 - Este ejemplo demuestra que la ejecución pasa entre dos threads automáticamente, compartiendo variables

Sincronización

- Existe un grupo de funciones diseñadas para controlar la ejecución de threads y el acceso a secciones críticas de código
 - Semáforos: actúan permitiendo y restringiendo el paso a cierta parte de código
 - Mutex: actúa como un dispositivo de exclusión mutua para una sección de código
- Mutex y Semáforos son similares (uno se podría implementar en términos del otro)
- Dependiendo de la semántica del problema uno podría resultar más adecuado que el otro

Sincronización

- Por ejemplo:
 - Controlar el acceso a memoria compartida, en donde sólo 1 thread puede accederla a la vez, es más adecuado utilizar mutex
 - Controlar el acceso a un grupo de objetos idénticos como un todo, como entregar una línea telefónica a un thread de 5 líneas disponibles, conviene tener un semáforo como contador
- Normalmente, un semáforo binario es usado para proteger una pieza de código que puede ser utilizada sólo por 1 thread, indicando si está disponible o no

Sincronización

- Si se desea que un número limitado de threads ejecute cierta parte de código, se usa un semáforo contador
- Las funciones que proveen el uso de semáforos comienzan con “sem_”, definidas en <semaphore.h>
- Para crear un semáforo:

```
int sem_init(sem_t *sem, int pshared,
             unsigned int value);
```

- Inicializa un objeto semáforo apuntado por sem

Semáforos

- El segundo parámetro indica si el semáforo será compartido con otros procesos o si será local:
 - 0 indica semáforo local, otros valores el semáforo será compartido
- El tercer parámetro indica el valor con el que se inicia el contador del semáforo
- Por el momento utilizaremos semáforos que no son compartidos entre procesos
- Actualmente los threads en Linux no soportan semáforos compartidos,

Semáforos

- Funciones que permiten controlar semáforos:

```
int sem_wait(sem_t * sem);
int sem_post(sem_t * sem);
```

- Ambas reciben como parámetro un objeto inicializado con “sem_init”
- “sem_post” incrementa atómicamente el valor del semáforo en 1:
 - Si 2 threads intentan incrementar el semáforo en 1, no se interfieren, por lo que el semáforo será incrementado correctamente en 2 (no así con archivos)

Semáforos

- “sem_wait” disminuye atómicamente en 1 el valor del semáforo:
 - Siempre espera a que el semáforo tenga un valor distinto de cero al principio
 - Si el semáforo está en cero, esperará a que otro thread incremente el valor del semáforo a un valor distinto de cero
 - Si dos threads están esperando hacer “sem_wait” al mismo semáforo que está en cero, y un tercero incrementa el valor, sólo 1 se desbloquea
- También existe “sem_trywait” es similar a sem_wait, pero no se bloquea, retornando un error en caso de que el contador sea cero

Semáforos

- Para eliminar un semáforo y liberar los recursos que está utilizando se utiliza:

```
int sem_destroy(sem_t * sem);
```

- Si se intenta finalizar un semáforo para el cual existe un “wait” se obtendrá un error
- Ejemplo: thread3.c
- En el ejemplo la idea es tener un thread que se encarga de leer la entrada y otro se encarga de contar los caracteres de entrada

Semáforos

- El thread principal se encarga de leer la entrada, asignar la variable compartida e incrementar el semáforo
- El nuevo thread esperará su turno en el semáforo y contará los caracteres de la variable compartida
- Los threads comparten la variable “work_area”
- Ejemplo modificado: thread3a.c
- Se incrementa el semáforo e inmediatamente se modifica la variable compartida
- La variable compartida recibe 2 set de palabras muy rápido: una desde el teclado y otra automáticamente