Comunicación entre procesos - Informática II

Clase 2

¡Hola! ¿Cómo están? ¿Pudieron explorar y probar los ejemplos que les subimos? En clase anterior pusimos de manifiesto diferentes mecanismos de comunicación y sincronización entre procesos bajo una lógica **productor – consumidor**.

Continuamos entonces esta nueva clase bajo la modalidad en línea de Informática II y comenzaremos a trabajar sobre dos mecanismos, uno de procesamiento y otro de control o sincronización.

¿Qué veremos?

- ***Hilos (threads)*****:** estrategia dedicada a la división y resolución de tareas. Si bien posee algunos aspectos comunes con los procesos hijo, veremos en particular sus diferencias.

- ***Mutex*****:** es un mecanismo de control o *sincronización* que utilizaremos para manejar hilos o threads.

Para recordar y no perder de vista

La invocación de la función **fork** crea un proceso **hijo** que se diferencia de su **padre** solamente por su *PID* (identificador del proceso) y por su *PPID* (identificador del proceso padre del proceso actual).

Luego de la creación del proceso hijo, *ambos procesos son ejecutados concurrentemente*, es decir, tenemos dos procesos ejecutando el código del programa a partir de una bifurcación generada luego de la invocación de **fork**.

La diferencia entre los dos se encuentra en lo que devuelve la llamada al **fork**, en cada uno de esos procesos *este valor es diferente*, para el proceso padre fork devuelve el PID del proceso hijo y para el proceso creado fork devuelve 0.

En el modelo tradicional de Unix cuando un proceso necesitaba realizar una tarea o actividad por fuera de la lógica principal, generaba un proceso hijo, este proceso realizaba la tarea y le avisaba a su padre cuando finalizaba. En algunos casos intercambiando información tal como vieron en informática 1 y continuamos viendo en informática 2. La mayoría de los proyectos o ejercicios que hicieron con socket funcionaba de esta manera.

Si bien este paradigma funcionó por muchos años, la invocación de **fork**, resulta costosa a nivel computacional.

¿Por qué sucede esto?

- La memoria se copia del padre al hijo
- Todos los descriptores también se copian
- Si bien el hijo recibe una copia de la información, cuando debe devolverle o enviarle datos a su padre, se requiere de mecanismos específicos como los que hemos visto.
- Si bien en la actualidad, se han mejorado algunos aspectos como ser que solo se duplica realmente la memoria cuando el hijo necesita acceder a la misma, todo esto resulta una gestión de mucha cantidad de memoria.

¿Cómo lo solucionamos?

La utilización de **hilos** o **theads** nos ayudará a resolver estos problemas. Los **threads** muchas veces son llamados **procesos livianos** (*lightweight process*). Se denominan así porque su creación es de 10 a 100 veces más rápida que la de un proceso realizado con **fork**.

Todos los **threads** dentro de un mismo proceso padre comparten la misma memoria global. Esto hace que el intercambio de información entre **threads** sea más sencillo. Sin embargo, esta simplificación trae aparejado el *problema de la sincronización.*

¿Cómo empezamos a trabajar?

A continuación veremos algunas funciones que nos permitirán: *crear, controlar y eliminar un* **thread**.

Crear un thread

int **pthread_create** (pthread_t *tid, const pthread_attr_t *attr, void* (*func) (void*), void *arg);

- El primer parámetro tid es pasado por referencia y generará el identificador del thread.
- Cada thread tiene diferentes atributos attr , asociados a su prioridad, el tamaño de su stack inicial entre otros. Si no queremos modificar los valores por defecto podemos utilizar como argumento de esta variable NULL.

- Cuando un **thread** se crea, debemos especificar una función para su ejecución, la cual llamaremos **función inicial** (*thread start function*). En este caso, recibe un puntero a función, por lo cual al invocarla lo completaremos con el nombre de la función a ejecutar.

Control de un thread

De forma de empezar a trabajar sobre el control de un **thread**, podemos hacer uso de la función **pthread_join**. En forma análoga a los procesos, esta función cumple con el mismo rol que lo hacía ***waitpid*** cuando queríamos esperar a la finalización de un proceso hijo.

La función posee el siguiente prototipo:

int **pthread_join** (pthread_t tid, void **status);

- El primer argumento tid refiere al **thread** que debemos esperar que finalice para continuar con nuestra ejecución.
- El segundo argumento status nos permite obtener el valor devuelto por el **thread** que estábamos esperando que finalice.

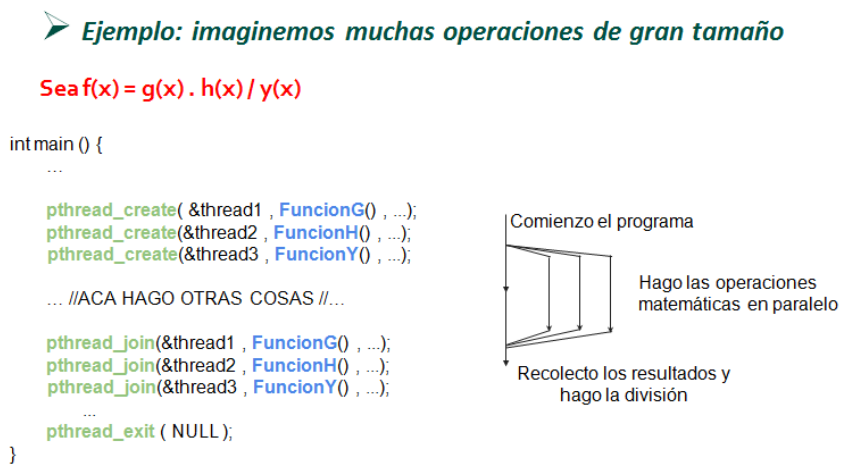
Finalización de un thread

La forma de finalizar un **thread** es a través de la invocación de la función **pthread_exit**, cuyo prototipo es el siguiente:

void **pthread_exit** (void *status);

En este caso al invocarla, finalizará el **thread** desde donde se encuentre pudiendo devolver valores a través de su argumento.

Veamos un esquema que nos orientará en lo abordado hasta aquí



Veamos un ejemplo sencillo:

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

void* mostrarX(void *);

int main ()
{
       pthread_t thread_id; //crea la variable thread_id de tipo pthread_t

       //imprime el id del proceso padre por la salida estandar.
       fprintf (stdout, "El pid del thread principal
       es %d\n", (int) getpid ());

       /* realiza la llamada para crear el thread */

       pthread_create(&thread_id,NULL,&mostrarX,NULL);

       while(1)
          fputc('o',stdout);

       return 0;
}

void* mostrarX(void *sinUso)
{
  while(1)
    fputc('x',stdout);

  return NULL;

}
```

A lo largo de estas dos clases hemos charlado sobre los mecanismos de comunicación, poniendo especial énfasis en los mecanismos de sincronización. Tal como vimos en el video de la segunda parte de la clase pasada, los semáforos fueron la respuesta para poder compartir, acceder y modificar un segmento de memoria compartido.

En el caso de **threads**, existe otro mecanismo y se lo conoce como **mutex**. Su nombre proviene de **exclusión mutua** (*mutual exclusion*). Un **mutex** es utilizado para proteger una región crítica, de forma de garantizar que pueda acceder a la misma de un solo **thread** a la vez.

```
       Bloqueo_el_acceso() ;

              Trabajo_sobre_región_crítica() ;

       Desbloqueo_el_acceso() ;
```

¿Cómo trabajo con Mutex?

Para trabajar con **mutex**, haremos uso de 3 funciones:

- int **pthread_mutex_lock** (pthread_mutex_t *mptr);
- int **pthread_mutex_trylock** (pthread_mutex_t *mptr);
- int **pthread_mutex_unlock** (pthread_mutex_t *mptr);

Nota 1: todas estas funciones devuelven el valor 0 en caso de éxito

Si intentamos bloquear un **mutex** , que ya se encuentra bloqueado por otro **thread**, la función **pthread_mutex_lock** se bloqueará hasta que el recurso se libere o desbloquee. En forma complementaria, contamos con la función **pthread_mutex_trylock**, la cual es una función no bloqueante que nos permite garantizar el funcionamiento y nos informa si el **mutex** está bloqueado a través de un flag cuyo valor es *EBUSY*.

Final de esta clase

Bueno, este es el final de la lectura de la clase los y las invitamos a ver los videos publicados en el aula virtual con relación a los temas abordados.

Dentro del aula virtual encontrarán ejemplos en el siguiente link.

Si surgen dudas o consultas, pueden escribirnos en el foro correspondiente.

Nos leemos!