Grupo : 2202
Asignatura : Sistemas operativos
Alumnos : Lucía Asencio y Rodrigo De Pool

Introducción:

En el siguiente documento responderemos a las preguntas propuestas en la práctica. Además, aclararemos posibles dudas que puedan surgir del código entregado.

Nota:

El código de todos los ejercicios se encuentra en el directorio “codigo”. Para generar todos los ejecutables basta con ejecutar el comando make dentro de este directorio. Los ejecutables se generarán en el mismo.

Documentación:

La documentación se encuentra dentro de la carpeta doc. El archivo refman.pdf la tiene en el formato indicado y dentro de la carpeta html se puede abrir el archivo index.html que tiene la documentación en formato html.

Ejercicio 3:

Este ejercicio consta de dos apartados, donde podemos contrastar el funcionamiento de los hilos con la creacion de nuevos procesos (forks). Estos son dos ejemplos de la salida de ambas ejecuciones:

El programa con threads tarda 0.599094
El programa con forks tarda 0.613713

El programa con threads tarda 0.619387
El programa con forks tarda 0.625419

Aunque se nos pedia compararla para el calculo de 10000 primos, tomamos datos de diferentes potencias de 10 (entre 0 y 6, en orden creciente) y estos fueron los resultados:

El programa con threads tarda 0.1474
El programa con threads tarda 0.1431
El programa con threads tarda 0.3868
El programa con threads tarda 0.34576
El programa con threads tarda 0.611612
El programa con threads tarda 13.505416
El programa con threads tarda 353.117578

El programa con forks tarda 0.14669
El programa con forks tarda 0.14943
El programa con forks tarda 0.13954
El programa con forks tarda 0.58362
El programa con forks tarda 0.628060
El programa con forks tarda 13.452353
El programa con forks tarda 346.45748

En la primera ejecución se percibe que el tiempo de creacion y ejecucion de procesos es ligeramente menor que de hilos, y esto concuerda con lo estudiado en las clases de teoría.

Grupo : 2202
Asignatura : Sistemas operativos
Alumnos : Lucía Asencio y Rodrigo De Pool

Por otro lado, con lo segundos datos se ve que, aunque siempre oscilan en los mismos valores, no siempre se cumple que la ejecución con threads sea más rápida que con nuevos procesos: por ejemplo, para 10^6 primos el programa con forks tardo 6 segundos menos que el de threads (de todos modos, 6 segundos son pocos de los 6 minutos de ejecución). Investigando en internet, encontramos en el planificador de Linux un atributo llamado “thread group id”, equivalente al “group id” estudiado en teoría, pero en este caso para los hilos de un mismo proceso. Esta puede ser la causa de que, al crear nuestro proceso 100 hilos diferentes, la ejecución se vea ralentizada y, la ventaja de tiempo ganada en el hecho de que no haga falta crear un nuevo proceso, se pierda por la existencia de estos grupos de hilos.

Por último, estudiamos también la ejecución de los hilos uno a uno y de los procesos uno a uno, y llegamos a unos resultados que, aunque no hemos sabido explicar, nos resultaron interesantes. Los procesos siempre tardaban un número equivalente de ticks en su creacion, calculo y destruccion. Sin embargo, el tiempo de creacion, calculo y destruccion de hilos oscilaba muchismo: en general, tardaba varias centenas o mil ticks, pero cada 5/10 hilos, el tiempo bajaba a decenas de ticks. Mostramos una porcion de los resultados.

Ejecución con threads, tiempo medido en ticks:

“Thread 56 116
Thread 57 1002
Thread 58 83
Thread 59 112
Thread 60 717
Thread 61 1611
Thread 62 118
Thread 63 317
Thread 64 1464
Thread 65 82”

Ejecución con forks, tiempo medido en ticks:

“Child 5 2801
Child 6 2774
Child 7 2688
Child 8 2699
Child 9 2503
Child 10 2411
Child 11 2500
Child 12 2427
Child 13 2620
Child 14 2490
Child 15 2559”

Grupo : 2202
Asignatura : Sistemas operativos
Alumnos : Lucía Asencio y Rodrigo De Pool

Ejercicio 4:

En el apartado a) hemos realizado el ejercicio propuesto. Para ello creamos una estructura info, que inicializamos con la información introducida por el usuario, la cual luego pasamos como argumento a los hilos que ejecutan la función de multiplicación de escalar por matriz.

En el apartado b) optamos por agregar a nuestra estructura "Info" dos nuevos punteros a entero. Cada hilo tiene su propia estructura info pero para establecer comunicación hicimos que ambas estructuras apuntaran a los mismos enteros (mismas posiciones de memoria). Cada hilo almacenará en uno de los enteros la fila por la que se va ejecutando, a su vez podrá acceder a la estructura y por ello al puntero donde se almacena la fila por la que se va ejecutando el otro hilo y, de este modo, imprimirlo por pantalla.

Este ejercicio nos permite ver como dos hilos pueden acceder a las mismas posiciones de memoria compartiendo variables, sin necesidad de crear tuberías.

Ejercicio 6:

En este ejercicio podemos ver como las señales enviadas a través de la función 'kill' afectan a otros procesos. En el ejercicio propuesto vemos como la función kill devuelve que la señal ha sido correctamente enviada a pesar de que el proceso al que se envía la señal, ha probablemente finalizado. Variando un poco los tiempos de 'sleep' observamos que el retorno de la función kill es independiente de que esta señal llegue, o no, finalmente al proceso, sino que nos confirma, únicamente, que la señal será procesada y enviada por el sistema operativo.

Ejercicio 8:

Nota 1: Nos hemos encontrado con un problema. No entendíamos bien si el enunciado pedía que fuera el manejador de una señal el que enviara la señal al siguiente proceso, pero llegamos a la conclusión de que, si era esto lo que se pedía, no podíamos hacerlo sin variables globales (no podíamos saber, fuera del main donde se realizaba el fork, el pid del hijo generado al que hbaría que enviar la señal). Y como todos los profesores nos han dicho en estos dos años que son una mala práctica de programación, hemos decidido que los manejadores hicieran sólo el "printf"/sleep correspondiente, y la ejecución del signal se realizara en el main, para evitar el uso de variables globales.

Nota 2: Justo antes de la primera señal que se manda (del ultimo hijo al proceso raiz) hay un sleep de 5 segundos. En el enunciado podía entenderse que este sleep se repetía para cada vez que la señal SIGUSR1 daba una vuelta y volvía al padre, o sólo la primera vez. Nosotros hemos implementado la primera opción: el sleep de 5 segundos se hace justo antes de la primera ejecucion, y en todas las demas señales SIGUSR1 el sleep es de 2 segundos.

En este ejercicio, el proceso padre genera tantos hijos como se le indica como primer argumento de entrada, de manera secuencial. Cuando el ultimo hijo se crea, este envia al root una señal SIGUSR1 tratada según el enunciado, que se pasa de padres a hijos V veces (segundo argumento de entrada). Cada proceso espera (pause()) V veces a esta señal. Tras V vueltas, el proceso padre envia a su hijo, y este a su hijo, etc SIGTERM, señal que tambie es manejada y esperada por cada hijo. El ultimo hijo la envia al padre, que muere y acaba con la ejecucion del proceso.

Ejercicio 10:

Para este ejercicio realizamos una primera version , ejercicio10sinSleep.c, en la cual implementamos lo solicitado sin tener en cuenta ninguna sincronización y eliminando el sleep del

Grupo : 2202
Asignatura : Sistemas operativos
Alumnos : Lucía Asencio y Rodrigo De Pool

proceso B. Resultó que el resultado del programa era tremendamente inconsistente, examinaremos ahora algunos de los resultados obtenidos:

En uno de los casos el programa no escribía nada en el fichero. Esto ocurre cuando se ejecuta primero todo el proceso padre, este proceso intenta leer de un fichero vacío a lo cual simplemente se devuelve el carácter de fin de fichero. Como está en bucle, lee entonces 50 veces el final de fichero y finaliza la ejecución, sin que ningún proceso hijo se haya podido ejecutar. Finalmente elimina el proceso hijo y finaliza la ejecución.

Otro caso distinto fue en el que el programa se ejecutaba de forma aparentemente normal. Se escribían las palabras en el fichero y el padre los iba leyendo y abriendo nuevos procesos cuando fuere necesario. Imprimiendo las palabras leídas, se observaba que, en efecto, se leían 50 palabras.

Por último, tuvimos varios casos intermedios en los que, por ejemplo, funcionaban las primera 5 o las últimas 5 lecturas del fichero. Esto puede ocurrir combinando las dos situaciones explicadas anteriormente.

Luego realizamos una segunda version, ejercicio10conSleep.c, en la que no tuvimos en cuenta la sincronización pero sí utilizamos el sleep. En este caso el comportamiento fue más complejo. A pesar de no estar sincronizados el funcionamiento del código parece correcto, lo que ocurre es que como el proceso B tenía que esperar 5 segundos, al proceso A le da tiempo de escribir en el fichero hasta que aleatoriamente escribe “FIN”, en cuyo caso simplemente termina su ejecución. Luego el proceso B empieza a leer, pero para entonces ya no existe ningun problema puesto que A ha finalizado. La causa de que los problemas de sincronización no sean evidentes es la lentitud del proceso B, sin embargo, podría ocurrir un imporbable caso en el que el resultado del programa sea inesperado.

Por ello añadimos una tercera versión en la que proponemos un método para sincronizar los dos procesos. Creamos un manejador de señales para SIGUSR1 en la cual unicamente imprimimos la string “Inicia la siguiente tanda de lectura:”, y agregamos al main un flag que inicializamos a cero. Entonces justo antes de que el padre intente leer fichero hacemos un if en el que vemos si el flag esta a cero, en caso de que lo este lo ponemos a 1 y llamamos a la funcion pause. Luego se ejecutará el hijo y solo cuando termine su escritura mandará al padre la señal SIGUSR1 permitiéndole a este leer el fichero, leerá sucesivamente hasta que encuentre FIN (no volverá a llamar a pause gracias al flag) y una vez llega a fin antes de inicializar el nuevo proceso hijo que escriba en el fichero colocamos el flag a cero. Entonces se repetirá la misma sincronización. Podemos ver en la ejecución que este código nos da una salida correcta.