03-intro

Ejemplos de programas para apreciar características del sistema operativo

Estos ejemplos están basados en las prácticas realizadas en *preview*

La idea fundamental es mostrar prácticamente mediante pocos programas las tres características básicas e importantes a comprender de los sistemas operativos, a saber

- **Virtualización**
  - **Procesos**
  - **Memoria**
- **Concurrencia**
- **Persistencia**

Estos conceptos se compararán con los que presentan los sistemas operativos Unix, hoy en día, descendientes y equivalentes (típicamente, pero no solamente, Linux y Mac OS X); los programas realizados corren exactamente iguales en ambos sistemas.

A continuación, se describen los cinco programas y sus resultados; se recomienda entender el archivo *makefile* que se usa para compilar cada uno de los programas mediante el comando *make*.

- **11-cpu.c:**
- *-Código:*
- – Este programa necesita de un argumento en la linea de comando que puede ser cualquier string.
- – Sencillamente, lee la identificación de proceso y luego realiza un lazo de repetición de 10, mostrando en cada iteración el número de proceso y el string que se entregó en el argumento, haciendo una demora de 1 segundo.

- – El medio para hacer la demora es llamando a la función *spin_time* que se encuentra en el archivo *common.c* y cuyo prototipo está en el archivo de inclusión *common.h*.

- *Ejecución*:

- – Si se ejecuta desde el *shell* mediante: **$ ./11-cpu A**, lo que se obtendrá será la repetición de 10 líneas que contienen algo como lo siguiente: **(5010) -> A**, donde obviamente *5010* es el numero de identificación de proceso. (**Nota**: 5010 es circunstancial y a modo de ejemplo)

- – Donde se obtiene una conclusión importante es si se lanzan varias instancias de este mismo programa con el *script* **mult_cpu**

- – Este *script* contiene cuatro invocaciones sucesivas de el mismo programa pero dada una con un argumento distinto (en este caso los strings A, B, C y D sucesivamente) y todas estas invocaciones enviadas en *background*.

- *Conclusiones:-*

- – El resultado es que cada una de las invocaciones han generado sendos procesos, donde se muestra para cada uno el número de proceso y el argumento que recibió *verificando claramente que, si bien el programa original es el mismo, existe un proceso para cada invocación* que es totalmente independiente de los demás.

- – La imagen que se tiene, entonces, es que el sistema operativo ha creado la ilusión como que existiesen cuatro CPU totalmente independientes que estuviesen totalmente involucradas en la ejecución respectiva de cada uno de los procesos; a esto se denomina *virtualización de CPU o de proceso*.

- **12-mem:**

- *Código:*

- – En este caso, se trata de un simple programa que cuando se ejecuta, determina cual es su identificación de proceso y presenta la dirección de memoria donde se encuentra el comienzo de la función *main* y las direcciones de las dos variables *c* y *pid*

- – Luego entra en un loop sobre la variable *c* desde 1 hasta 10, incrementado dicha variable en cada iteración; en cada iteración muestra el valor de la identificación de proceso y el valor de la variable *c*, haciendo una pausa mediante la función *spin_time* de 2 segundos

- – Finalmente, al completar la iteración, muestra la finalización del proceso.

- *Ejecución;*

- – La ejecución directa mediante **$ ./12mem** no arroja nada inesperado.

- – Interesante, como en el caso anterior, es despachar varios procesos mediante el *script* **mult_mem** que tal cual se encuentra, permitirá despachar 4 de ellos en *backgroud*.

- – Para el caso de Linux y para poder sacar conclusiones válidas, previamente vamos a ejecutar el *script* denominado **linux_no_aslr** (Para el caso de *Mac OS X* ésto no es necesario).

- – Si ejecutamos luego el *script* **mult_mem** veremos que las memorias mostradas de cada uno de los procesos tienen *exactamente la misma dirección* y que, evidentemente, cada proceso muestra que el incremento de su variable c es independiente de los otros procesos.

- – Solamente en el caso de Linux, para volver las cosas a la normalidad después de haber hecho la experiencia, debe darse a nivel de *shell* el comando **exit**.

- – También, en el caso de Linux, si ahora se repite la experiencia, se va a encontrar que las direcciones de memoria no coinciden pero el incremento de la variable *c* se sigue manteniendo independiente de los otros procesos.

- – Para comprender porqué este comportamiento, se aconseja visitar el siguiente link: ASLR

- – En el caso de mac OS X, para desconectar ASLR, se debe recompilar el programa cambiando en el *makefile* el contenido de la línea **8** que sea igual al de la línea **5** y volver a compilar y ejecutar.

- *Conclusiones:*

- – Teniendo en cuenta el comportamiento del programa sin ASLR (que es el original de los sistemas operativos de este tipo) se llega a la conclusión que la virtualización no es solamente de CPU sino también de memoria, con lo cual las direcciones de memoria donde aparentemente se ejecuta el programa y las direcciones de sus variables *son las mismas* independientemente del proceso en cuestión.

- – Esto significa que la virtualización de CPU y memoria llevan a la ilusión que cada proceso tiene una CPU propia y que además tiene un mapa de memoria propio.

- **13-race1.c**:
- – Si bien la virtualización genera que nos da una bastracción para cada proceso de una CPU propia y de una memoria cuyo acceso está vedado a agentes externos, este blindaje se opone muchas veces a que dos o más procesos puedan colaborar a un mismo fin.
- – Para ello, se ha preparado un programa para tratar de demostrar los problemas que pueden acaecer en el proceso de colaboración.
- *Código:*
- – El objetivo de este ejercicio es generar dos procesos que tratan de enviar texto a la salida *standard*.
- – Básicamente, mediante *fork* se crea un proceso *hijo* y, posteriormente, tanto el padre como el hijo envían a la salida *standard* un string terminado por *new line* que es diferente para cada uno de ellos.
- – Teniendo en cuenta si existe o no el argumento de llamada, se va a tomar una decisión respecto de la salida *standard* antes de crear el proceso hijo.
- – La salida *standard* del sistema operativo está configurada normalmente de manera que, lo que se envíe a ella, se hace a través de un *buffer* intermedio (invisible para el usuario) y sólo se envía realmente a la salida *standard* cuando el almacenamiento en el *buffer* se completa o cuando arriba un *new line*.
- – Este comportamiento es el que se mantiene en el caso que no haya argumentos a la llamada a la ejecución del programa, y la variable *in_race* valdrá cero.
- – En el caso que haya un argumento, cualesquiera que fuese él, la variable *in_race* será distinta de cero y entonces, se ejecutará la función de biblioteca *standard setvbuf*, la cual solicita que *stdout* trabaje sin *buffer*, es decir que se envíe caracter por caracter a la salida *standard*.
- *Ejecución:*
- – La primera ejecución la realizamos sin argumentos, con lo cual veremos el texto **No race** cuando ejecutemos y se verá en la pantalla claramente que los dos textos salen correctamente.
- – Considerando que las condiciones de carrera (*race* en inglés) pueden producirse azarosamente, se intentará repetir el proceso *n veces* de manera de estar seguros.

- – Para ello, recurrimos al *script* **mult_race** pasándole como argumento la cantidad de veces como en, por ejemplo **$ ./mult_race 10**: el resultado es que no falla y todo se desarrolla normalmente.
- – Ahora, ejecutaremos el programa con un argumento cualquiera, como en **$ ./13-race X**: veremos efectivamente que los caracteres están mezclados entre los dos textos y cada vez que lo ejecutemos, la mezcla es distinta.
- *Conclusiones:*
- – El sistema operativo provee una abstracción de manejo de la salida *standard* que permite competir y/o colaborar dos o más procesos sobre un recurso, permitiendo la *concurrencia* o sincronización.
- – El hecho de cambiar el comportamiento del sistema operativo sobre el recurso trajo el problema de colisión o carrera.

- **14-race2.c:**
- – Este ejemplo es otro donde se muestran los problemas de falta de concurrencia o sincronización
- – Se aprovecha para mostrar el comportamiento de *threads* o hilos de ejecución dentro de un proceso.
- *Código:*
- – La idea del programa es tener dos hilos de ejecución que colaboren a incrementar la misma variable haciendo **N** incrementos cada uno sobre la misma variable e imprimiendo al terminar el valor final de la variable.
- – Si todo es correcto en el funcionamiento, la variable tendría que mostrar una cuenta que sea el doble de la cantidad de incrementos solicitados.
- – La cantidad de incrementos se entregan a través del argumento de la invocación como, por ejemplo **$ ./14-race2 1000**
- *Ejecución:*
- – Para producir el efecto a mostrar (el problema de concurrencia), depende mucho de las características del *hardware* y de sistema operativo, pero típicamente con un valor de 1000 es muy probable que se comporte correctamente y de como resultado 2000.

- – Se aconseja, por lo tanto, pasar a valores como 10000 o más aún hasta eu el valor final sea menos del doble del argumento pasado
- *Conclusiones:*
- – En el caso de los *threads*, lo único que se virtualiza es el procesador y su contexto inmediato, por lo cual a veces se lo indica también como *una especie de proceso ligero*
- – Por ello, las variables automáticas de las funciones y los argumentos de las mismas son propias de cada *thread* (ya que las mismas se alojan en el área de *stack*) pero las variables externas y estáticas son compartidas entre los *threads* y el proceso que los crea.
- – Con estos datos, se puede sacar la conclusión que la instrucción en la línea 24 constituye una zona *crítica* del código y que debería recurrirse a un mecanismo de sincronización de *threads*.

15-io.c:

- – El último mecanismo a mostrar es el de **persistencia** a través de un simple ejemplo.
- – En efecto, los resultados de la ejecución de los programas deben resguardarse de su pérdida y el mecanismo es el de colocarlo en un medio externo que lo preserve, por lo cual la persistencia siempre está asociada a un mecanismo de *entrada/salida* o *I/O*.
- – Los sistemas operativos tratan de crear una abstracción alrededor de entrada-salida y, en el caso de Unix, es el concepto generalizado de archivo o también llamado *stream* o flujo de datos.
- – *Código:*
- – El ejemplo es muy sencillo: ya de hecho hemos visto un ejemplo de manejo de salida en el caso de *14-race1* donde se intentaba enviar mediante dos procesos a la salida *standard* que, si bien se trata de la salida sobre una terminal, dicha salida se maneja por el mismo mecanismo que los archivos (en ese caso, el manejo de alto nivel de archivos a través de la biblioteca *standard*).
- – En este caso, se trata solamente de crear un archivo (en este caso de almacenamiento y de ahí al característica de persistencia), usando *llamadas de bajo nivel* o sea *system calls*.
- – Se trata de tres típicas llamadas, cuales son *open, write y close* además de la llamada *fsync*

- – La llamada a *open* retorna un entero que, de tener éxito, debe ser positivo; esta llamada a través del nombre *externo* del archivo, establece una vinculación entre dicho nombre y los datos del archivo, que después de esta llamada, quedan en una estructura perteneciente a un arreglo de archivos abiertos cuyo índice es el valor retornado por la llamada y que recibe el nombre de *file descriptor*.

- – Esta llamada recibe tres argumentos: la ruta del archivo a abrir, las condiciones de uso del archivo (en este caso dado por el OR bit a bit que indica que el archivo se abre para escritura solamente, que debe ser creado si no existe y el tercer bit indica que si existe debe ser previamente borrado, y por último los permisos de creación para el usuario, que indica que tiene permisos de lectura y escritura.

- – Inmediatamente, se le escribe el contenido que se encuentra en el *buffer* y que es el string "Hello World" si al programa se lo invocó sin argumentos o el string del argumento en caso contrario.

- – Por condiciones de eficiencia de transferencia, no siempre se escribe en el dispositivo; los dispositivos tienen une especie de *cache* tanto para escritura como para lectura con lo cual no basta con cerrar el archivo para asegurarse que se escriba en el, por lo cual previamente a cerrarlo, se realiza la llamada a *fsync* con el objetivo de, justamente, lograr que se escriba en el dispositivo.

- – Posteriormente, se llama a *close*, lo cual hace que se rompa la vinculación del proceso con las caracteristicas del archivo que se pusieron en memoria en la apertura y qye estaban apuntadas por fd, liberando la zona de la tabla para el uso en el caso de otra llamada a *open*.

- *Conclusiones:*

- – Se ha tratado de mostrar con un sencillo ejemplo el tema de manejo de *persistencia* en Unix que involucra no sólo al tema en si de almacenamiento, sino también al manejo generalizado de *entrada-salida*.

- – Por ello, se verá que todo el manejo de *entrada-salida* termina siendo realizado por las cuatro operaciones fundamentales *open, read, write, close*, aunque en las llamadas de más alto nivel a veces estén ocultas por otras llamadas equivalentes y que terminan derivando en éstas.