Diseño de sistemas operativos. Febrero de 2001. Ejercicio 1

Enunciado

Se plantean una serie de cuestiones sobre la implementación de procesos en UNIX.

a) ¿Qué valor inicial toma el contador de programa en un `FORK`, en un `EXEC` y en un `pthread_create`? ¿Y el puntero de pila? ¿Cuál es el contenido inicial de la nueva pila para estas tres llamadas? ¿Qué valores iniciales tomarán los registros generales?

Se distinguen las siguientes operaciones internas del sistema operativo:

- Reservar y liberar una entrada de la tabla de procesos
- Rellenar/actualizar el BCP (se debe explicar en qué consiste la actualización)
- Insertar y eliminar un proceso de una cola de procesos
- Cambiar de contexto
- Planificar
- Leer e interpretar un ejecutable
- Crear pila (se debe especificar cuál es su contenido inicial)
- Crear una región de memoria (privada o compartida, asociada a un ejecutable o sin soporte)
- Compartir y duplicar una región
- Eliminar una región de memoria

Se pide especificar, basándose en las anteriores operaciones, cómo se llevan a cabo las siguientes llamadas:

b) `FORK`

c) `EXEC`

d) `EXIT` y `WAIT` (tenga en cuenta la sincronización asociada a la terminación de procesos en UNIX).

e) En Linux existe una llamada denominada `CLONE` que generaliza el `FORK` convencional, permitiendo especificar que el padre y el hijo compartan realmente parte de su información, en lugar de tratarse de una copia. Puesto que se trata de un servicio destinado principalmente al soporte de *threads*, analice qué tipo de información debería ser compartida entre ambos y cuál no. ¿Cómo debería rediseñarse el BCP para implementar esta funcionalidad extendida de Linux?

Solución

a) Se analiza, en primer lugar, qué valor toma el contador de programa en las llamadas especificadas:

- `FORK`. Dado que esta llamada crea un nuevo proceso que es un duplicado del proceso que la invoca, el valor inicial del contador de programa para este nuevo proceso es el mismo que el del proceso padre.
- `EXEC`. Puesto que esta llamada inicia la ejecución de un nuevo programa en el contexto del mismo proceso que la invocó, el contador de programa debe tomar un valor que corresponda con la primera instrucción del nuevo programa. Este valor estará contenido en la cabecera del ejecutable que contiene el programa.
- `pthread_create`. El valor inicial del contador de programa para el nuevo *thread* se corresponderá con la dirección, dentro del mapa de memoria actual, de la primera instrucción que ejecutará. Este valor se recibe como parámetro de la llamada y, normalmente, se corresponde con la dirección de comienzo de una función contenida

en el programa actual.

A continuación, se analiza qué valor tomará el puntero de pila y cuál será el contenido inicial de la nueva pila para cada una de las tres llamadas:

- `FORK`. El puntero de pila inicial del hijo será una copia del que tiene el padre. El contenido de la pila será una copia de la del padre, ya sea una copia real o basada en *copy-on-write*. Nótese que, aunque el valor del puntero de pila sea el mismo, se corresponde con dos espacios lógicos de memoria independientes.
- `EXEC`. Esta llamada implica eliminar la pila actual del proceso y crear una nueva que contenga el entorno del proceso y los argumentos del programa. El puntero de pila apuntará justo encima de la información apilada.
- `pthread_create`. Cada *thread* debe tener asociada su propia pila que le permita almacenar los registros de activación de las llamadas a función que realice. Por tanto, en esta llamada se debe crear una pila para el nuevo *thread* que contendrá inicialmente el argumento que se le quiere pasar a dicho *thread*. Nótese que, además, se incluirá otra información como, por ejemplo, la necesaria para que, cuando termine la función asociada al *thread*, se llame implícitamente a `pthread_exit`, en el caso de que el *thread* no lo haya hecho de forma explícita. El puntero de pila del nuevo *thread* quedará apuntando a esa información apilada. Hay que hacer notar que POSIX permite que se especifique en la propia llamada `pthread_create` una pila para el nuevo *thread*, que el programa habrá creado previamente antes de invocar a `pthread_create`. Si éste es el caso, la llamada `pthread_create` no crea una nueva pila, sino que usa la que ha recibido como parámetro, incluyendo en ella la información anteriormente comentada y haciendo que el puntero de pila apunte a dicha información.

Por último, se analiza qué ocurre con los valores almacenados en los registros de propósito general del procesador.

- `FORK`. Dado que el espacio de memoria del hijo es un duplicado del que posee el padre en el momento de invocar a esta función, los valores de los registros serán una copia de los del proceso padre. Nótese que los registros forman parte del mapa de memoria del proceso.
- `EXEC`. Puesto que se va a empezar la ejecución de un nuevo programa, no es necesario cargar ningún valor específico en los registros generales, ya que, normalmente, el programa no asumirá que existe ningún valor específico almacenado en los mismos. La propia ejecución del nuevo programa causará que los registros generales tomen los valores pertinentes.
- `pthread_create`. Dado que el nuevo *thread* comparte el espacio de memoria con el resto de *threads* del proceso, los valores de los registros deben ser una copia de los del *thread* padre.

b) Las operaciones básicas asociadas al `FORK` serían las siguientes:

- Reservar una entrada de la tabla de procesos.
- Rellenar el BCP reservado copiando los valores del padre (esta copia incluye información como los registros salvados, los descriptores de ficheros o el tratamiento de las señales). Sin embargo, algunos campos del BCP deben tomar valores específicos para el hijo, tales como su identificador o la información relacionada con la contabilidad sobre el uso de recursos de cada proceso. Se pondrá al proceso en estado de listo para ejecutar.
- Con respecto a la gestión de memoria, por cada región de memoria del mapa del proceso padre:
  - Si es de carácter compartido, se comparte entre el padre y el hijo.
  - Si es de carácter privado, se duplica en el hijo, ya sea de manera inmediata o usando *copy-on-write*.
- Como resultado de las operaciones de memoria anteriormente comentadas, se ha creado implícitamente una nueva pila cuyo contenido es un duplicado de la pila del padre.
- Insertar el nuevo proceso al final de la cola de listos

c) Las operaciones básicas asociadas al `EXEC` serían las siguientes:

- Leer e interpretar la cabecera del ejecutable para obtener los datos de cada región. Esta operación de lectura conlleva el bloqueo del proceso mientras que se lee del disco la cabecera del ejecutable, a no ser que el bloque correspondiente estuviera almacenado en la caché del sistema de ficheros. Este bloqueo implicaría a su vez las siguientes operaciones:
  - Poner al proceso en estado bloqueado.
  - Eliminar el BCP de la cola de listos e insertarlo en la cola de procesos bloqueados esperando la

finalización de una operación sobre el disco.
    - Planificar y cambiar de contexto.
    - Cuando ocurra la interrupción que indica que ha terminado la operación del disco, el proceso será movido de la cola de bloqueados a la de listos y, cuando sea posteriormente elegido por el planificador, continuará procesando esta llamada `EXEC`.
- Eliminar las regiones del mapa actual, salvando previamente la información que corresponde con los argumentos y el entorno que recibirá el nuevo programa.
- Crear las regiones especificadas en el ejecutable de acuerdo con sus características específicas:
    - Código: Permiso de lectura y ejecución, de carácter compartido y asociado al ejecutable.
    - Datos con valor inicial: Permiso de lectura y escritura, de carácter privado y asociado al ejecutable.
    - Datos sin valor inicial: Permiso de lectura y escritura, de carácter privado y sin soporte.
- Crear pila como una región con permiso de lectura y escritura, de carácter privado y sin soporte, cuyo contenido inicial serán los argumentos y el entorno pasados al nuevo programa.
- Actualizar el BCP del proceso especificando, entre otras cosas, la nueva información sobre el mapa de memoria y los nuevos valores del contador de programa (primera instrucción del nuevo programa)y del puntero de pila (apuntando a la nueva pila).

d) Las operaciones básicas asociadas al `EXIT` serían las siguientes:

- Eliminar las regiones del mapa actual y liberar otros recursos usados por el proceso (cerrar ficheros abiertos, liberar semáforos usados, etc.).
- Actualizar el BCP del proceso para reflejar esas operaciones y poner al proceso en estado *Zombie*. Asimismo, habría que actualizar el BCP de los procesos hijos para especificar que pasan a ser hijos directos del proceso init.
- Sincronización con el proceso padre:
    - Si el proceso padre está bloqueado en `WAIT`
        - Eliminar de la cola de espera al proceso padre e insertarlo en la de procesos listos para ejecutar.
- Eliminar al proceso de la cola de procesos listos para ejecutar.
- Planificar y cambiar de contexto.

Las operaciones básicas asociadas al `WAIT` serían las siguientes:

- Si no hay ningun proceso hijo en estado *Zombie*
    - Poner al proceso en estado bloqueado.
    - Eliminar el BCP de la cola de listos e insertarlo en una cola de espera.
    - Planificar y cambiar de contexto.
- Recoger información dejada por el proceso hijo ya terminado en su BCP (estado de terminación, contabilidad de su uso de recursos, etc.).
- Liberar la entrada de la tabla de procesos usada por el proceso hijo.

e) Puesto que se trata de un servicio cuyo principal uso es el soporte de *threads*, se debería compartir la información del proceso que normalmente comparten los *threads*:

- Información sobre ficheros: los descriptores de ficheros abiertos, la máscara de creación de ficheros (`umask`), el directorio actual y el directorio raíz.
- El tratamiento de las señales.
- El mapa de memoria.

Es importante resaltar que, para conseguir una semántica adecuada para la implementación de *threads*, es necesario que realmente se comparta esta información, no siendo suficiente el uso de un duplicado de la misma. Así, si, por ejemplo, un proceso abre un fichero, un proceso hijo creado mediante `CLONE` deberá poder acceder al fichero abierto por el padre usando el mismo descriptor. Nótese que este comportamiento no se da con un proceso hijo creado con `FORK`. Por lo que se refiere el mapa de memoria, éste debería ser realmente compartido entre el padre y el hijo en el caso del `CLONE` (no se usaría, por tanto, *copy-on-write*), de manera que, si un proceso proyecta un fichero, éste

estuviera también accesible al proceso hijo en el mapa de memoria compartido.

Por lo que se refiere a qué información no debe compartirse, sería aquélla que es específica de cada proceso, o sea, la que en un sistema con *threads* fuera propia de cada *thread*. Se podría resaltar la siguiente:

- La copia de los registros.
- El estado del proceso.
- Temporizadores.

Por último, con respecto al BCP, sería necesario rediseñarlo de manera que sea válido tanto para procesos creados con `FORK` como mediante `CLONE`. Para ello, se puede hacer que los campos que corresponden con información a compartir (`CLONE`) o a duplicar (`FORK`) no almacenen el valor en sí, sino una referencia (puntero) al mismo. Así, si se trata de una llamada `CLONE`, sólo habrá que hacer que el puntero correspondiente del BCP referencie al mismo objeto que el del padre (o sea, copiar el puntero). Si se trata de un `FORK`, habrá que reservar una nueva zona de memoria para almacenar el campo, hacer un duplicado del campo correspondiente del padre y hacer que el puntero en el BCP del hijo apunte a la nueva zona reservada.

A continuación, usaremos como ejemplo la máscara de creación de ficheros (`umask`). Supongamos que en el BCP original está definida de la siguiente manera:

```
struct BCP {
        ......
        int umask;
        ......
}
```

En el BCP rediseñado, la definición de este campo usaría un puntero:

```
struct BCP_nuevo {
        ......
        int *umask;
        ......
}
```

En la llamada `CLONE` el tratamiento de este campo sería el siguiente:

```
clone() {
        ......
        hijo->umask=padre->umask;
        ......
}
```

Mientras que en la llamada `FORK`, se realizaría de la siguiente forma:

```
fork() {
        ......
        hijo->umask=malloc(sizeof(int));
        *hijo->umask=*padre->umask;
        ......
}
```