Sistemas Operativos Avanzados

CC-571

César Lara Avila

Universidad Nacional de Ingeniería

(actualización: 2020-06-18)

Bienvenidos

Sesión 2

Virtualización

Temario de la sesión 2

- Llamadas al sistema *fork()*, *wait()*, *exec()*
- Control de procesos y usuarios
- El mecanismo de ejecución directa limitada
- Problema 1: Operaciones restringidas
- Problema 2: Cambio entre procesos
- Mecanismo de cambio de contexto
- Concurrencia

¿Qué API proporciona el OS a los programas de usuario?

- API = interfaz de programación de aplicaciones.

  ```
  = funciones disponibles para escribir programas de usuario.
  ```

- La API proporcionada por el OS es un conjunto de **llamadas al sistema.**
  - La llamada al sistema es una llamada de función al código del OS que se ejecuta en un nivel de privilegio más alto de la CPU.
  - Las operaciones sensibles (por ejemplo, acceso al hardware) solo se permiten en un nivel de privilegio más alto.
  - Algunas llamadas al sistema de "blocking" hacen que el proceso se bloquee y se reprogramen (por ejemplo, leer desde el disco).

¿ Debemos reescribir los programas para cada OS?

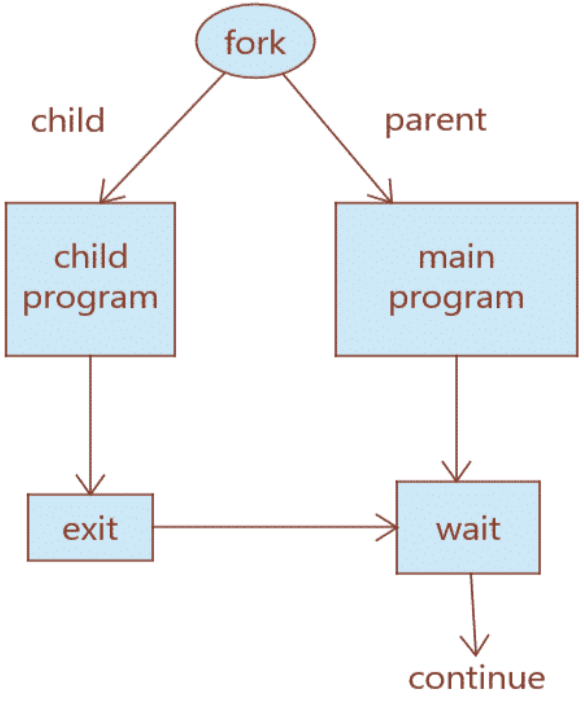
- **API POSIX**: un conjunto estándar de llamadas al sistema que un SO debe implementar.
  - Los programas escritos en la API POSIX pueden ejecutarse en cualquier OS compatible con POSIX.
  - La mayoría de los sistemas operativos modernos son compatibles con POSIX.
  - Asegura la portabilidad de un programa.
- Las librerías del lenguaje de un programa ocultan los detalles de invocar llamadas al sistema.
  - La función **printf** en la biblioteca C llama al sistema **write**, para escribir en la pantalla.
  - Los programas de usuario generalmente no necesitan preocuparse por invocar llamadas al sistema.

Llamadas al sistema *fork()*, *wait()*, *exec()*

- **fork()** crea un nuevo proceso hijo
  - Todos los procesos se crean bifurcando (forking) desde un proceso padre.
  - El proceso *init* es ancestro de todos los procesos.
- **exec()** hace que un proceso ejecute un ejecutable dado.
- **exit()**) finaliza un proceso.
- **wait()** hace que un proceso padre se bloquee hasta que el proceso hijo termine.
- Existen muchas variantes de las llamadas al sistema anteriores con diferentes argumentos.

¿Qué ocurre durante un *fork()*

- Se crea un nuevo proceso al hacer una copia de la imagen de memoria de los padres.
- El nuevo proceso se agrega a la lista de procesos del OS y se programa.
- La ejecución padre e hijo comienza justo después de la bifurcación (con diferentes valores de retorno).
- Los procesos padre y el hijo ejecutan y modifican los datos de la memoria de forma independiente.



- Flujo básico de *fork()*.

Ejercicios

- Analiza el **Ejemplo1.c**.
- ¿Por qué aparecen estos resultados?

```
(base) c-laraavila@Lara1:~/Desktop$ ./Ejemplo1
hola mundo (pid:20189)
Hola, soy un proceso padre de 20190 (pid:20189)
Hola a todos, soy un proceso hijo (pid:20190)
(base) c-laraavila@Lara1:~/Desktop$ ./Ejemplo1
hola mundo (pid:20191)
Hola, soy un proceso padre de 20192 (pid:20191)
Hola a todos, soy un proceso hijo (pid:20192)
```

Esperando a que mueran los proce hijos

- Escenarios de término de procesos.
  - Al llamar a *exit()* (se llama a *exit* automáticamente cuando se alcanza el final de *main*).
  - El OS finaliza un proceso de mal comportamiento.
- El proceso terminado existe como zombie.
- Cuando un padre llama a *wait()*, el proceso hijo zombie es "reaped".
- *wait()* bloquea al proceso padre hasta que el proceso hijo termine (existen formas sin bloqueo para invocar *wait*).
- ¿Qué pasa si el proceso padre termina antes que el proceso hijo? el proceso *init* adopta huérfanos y los "reaps".

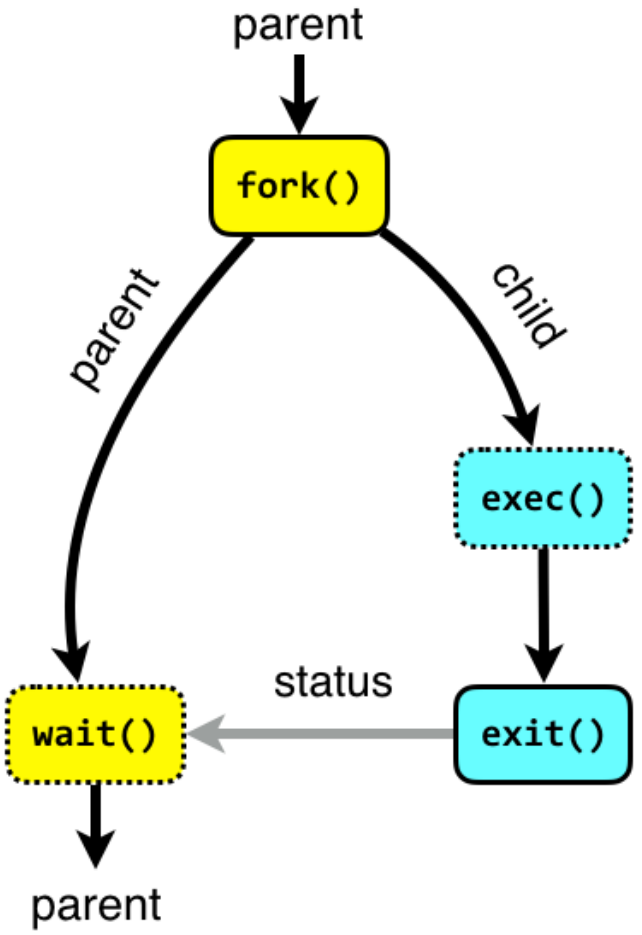
Ejercicios

- Analiza el **Ejemplo2.c**.
- ¿Por qué aparecen estos resultados?

```
(base) c-laraavila@Lara1:~/Desktop$ ./Ejemplo2
Hola CC-571 (pid:20790)
Hola, soy un proceso hijo(pid:20791)
Hola CC-571, soy un proceso padre de 20791 (wc:20791) (pid:20790
```

¿Qué pasa durante *exec()*?

- Después de *fork* los procesos padre e hijo ejecutan el mismo código.
  - No es muy útil!.
- Un proceso puede ejecutar *exec()* para cargar otro ejecutable en su imagen de memoria.
  - Entonces, un hijo puede ejecutar un programa diferente del padre.
- Variantes de *exec()*, por ejemplo, para pasar argumentos de línea de comandos a un nuevo ejecutable.



Ejercicios

- Analiza el **Ejemplo3.c**.
- ¿Qué significan estos resultados?

```
(base) c-laraavila@Lara1:~/Desktop$ ./Ejemplo3
Hola CC-571 (pid:22038)
Hola, soy un proceso hijo(pid:22039)
  31  127 1009 Ejemplo3.c
Hola soy un proceso padre de 22039 (wc:22039) (pid:22038)
```

¿Cómo funciona un shell?

- En un OS básico, el proceso *init* se crea después de la inicialización del hardware.
- El proceso *init* genera un shell como `bash` o `tcsh`.
- El shell lee el comando del usuario, bifurca a un proceso hijo, ejecuta el comando ejecutable, espera a que termine y lee el siguiente comando.
- Los comandos comunes como *ls* son todos ejecutables que simplemente son ejecutados por el shell:

```
(base) c-laraavila@Lara1:~/Desktop$ ps aux | wc -l
253
```

- El shell puede manipular al proceso hijo de maneras extrañas . Suponga que desea redirigir la salida de un comando a un archivo:

```
ls> foo.txt
```

- Entonces el shell genera un proceso hijo, vuelve a conectar su salida estándar a un archivo, y luego llama a *exec* en el proceso hijo.

Ejercicios

- Analiza el **Ejemplo4.c**.
- ¿Qué significan estos resultados?

```
(base) c-laraavila@Lara1:~/Desktop$ gcc -o Ejemplo4 Ejemplo4.c
(base) c-laraavila@Lara1:~/Desktop$ ./Ejemplo4
(base) c-laraavila@Lara1:~/Desktop$ cat Ejemplo4.output
35  129 1001 Ejemplo4.c
```

Ley de Lampson

- A veces, solo tienes que hacer lo correcto, y cuando lo haces, es mucho mejor que las alternativas. [1]
- Hay muchas formas de diseñar API para la creación de procesos.
- La combinación de *fork()* y *exec()* es simple e inmensamente poderosa. Aquí, los diseñadores de UNIX simplemente acertaron.

[1] . **Hints for Computer System Design.**

Control de procesos y usuarios

- ¿Qué procesos pueden ser controlados por una persona en particula}r se encapsula en la noción de usuario.
- El OS permite que múltiples usuarios ingresen al sistema y garantiza que los usuarios solo puedan controlar sus propios procesos.
- Un superusuario puede controlar todos los procesos (y de hecho hacer muchas otras cosas).

El mecanismo de ejecución directa limitada.

¿Cómo virtualizar eficientemente la CPU con control?

- El OS necesita compartir la CPU física por tiempo compartido.
- Problema
  - Rendimiento: ¿cómo podemos implementar la virtualización sin agregar una sobrecarga excesiva al sistema?
  - Control: ¿Cómo podemos ejecutar procesos de manera eficiente mientras mantenemos el control sobre la CPU?

Mecanismos de bajo nivel

- ¿Cómo ejecuta el OS un proceso?
- ¿Cómo maneja una llamada al sistema?
- ¿Cómo cambia el contexto de un proceso otro?

Ejecución de Procesos

- El OS asigna memoria y crea una imagen de memoria.
  - Código y datos
  - Stack y heap
- Apunta el contador del programa de la CPU a la instrucción actual.
  - Otros registros pueden almacenar operandos, valores de retorno, etc.
- Después de la configuración, el OS está fuera del camino y el proceso se ejecuta directamente en la CPU.

Una llamada de función

- Una llamada de función se traduce en una instrucción de salto (jump).
- Un nuevo stack frame se lleva a la pila y stack pointer (SP) se actualiza.
- Valor anterior de PC (valor de retorno) se lleva a la pila y la PC es actualizada.
- El stack frame contiene valor de retorno, argumentos de función, etc.

¿Cómo es diferente una llamada al sistema?

- El hardware de la CPU tiene múltiples niveles de privilegio.
  - Uno para ejecutar el código de usuario: modo de usuario.
  - Uno para ejecutar el código del OS como llamadas del sistema: modo kernel.
  - Algunas instrucciones se ejecutan solo en modo kernel.
- El kernel no confía en la pila de usuarios.
  - Utiliza una pila de kernel separada cuando está en modo kernel.
- El Kernel no confía en las direcciones proporcionadas por el usuario, salta al:
  - kernel que configura la tabla de descriptores de interrupción (IDT) en el arranque.
  - IDT que tiene las direcciones de funciones del kernel para ejecutar llamadas al sistema y otros eventos.

Llamada al sistema

- Permite que el kernel exponga cuidadosamente ciertas funciones clave al programa del usuario, como:
  - Acceder al sistema de archivos.
  - Crear y destruir procesos.
  - Comunicarse con otros procesos.
  - Asignar más memoria.

Mecanismo de llamada al sistema: la instrucción trap

- Cuando se debe realizar una llamada al sistema, se ejecuta una instrucción *trap* (generalmente oculta para el usuario por *libc*).
- Ejecución de la instrucción trap:
  - Mueve la CPU a un nivel de privilegio más alto.
  - Cambia a la pila del kernel.
  - Guarda contexto (PC antigua, registros) en la pila del kernel.
  - Busca la dirección en IDT y salta para atrapar la función del controlador en el código del OS.
- La instrucción trap se ejecuta en el hardware en los siguientes casos:
  - Llamada del sistema (el programa necesita servicio del OS).
  - Error del programa (el programa hace algo ilegal, por ejemplo, acceder a la memoria a la que no tiene acceso).
  - Interrupción (el dispositivo externo necesita atención del OS, por ejemplo, un paquete de red ha llegado a la tarjeta de red).
- En todos los casos, el mecanismo es: guardar el contexto en la pila del kernel y cambiar a la dirección del OS en IDT.

Regreso de trap (return from trap)

- Cuando el OS termina de manejar el **syscall** o interruption, llama a una instrucción especial **return-from-trap** que se encarga de:
  - Restaura el contexto de los registros de la CPU desde la pila del kernel.
  - Cambia el privilegio de CPU del modo kernel al modo usuario.
  - Restaura la PC y saltar al código de usuario después del trap.
- El proceso del usuario no sabe que se suspendió, reanuda la ejecución como siempre.
- Antes de volver al modo de usuario, el OS comprueba si debe cambiar a otro proceso.

Pregunta:

- ¿Debe volver siempre al mismo proceso de usuario desde el modo kernel?

Problema 1: Operaciones restringidas.

- ¿Qué sucede si un proceso desea realizar algún tipo de operación restringida como?:
  - Emitir una solicitud de E/S a un disco.
  - Obtener acceso a más recursos del sistema, como CPU o memoria.
- **Solución**: uso de transferencia de control protegida.
  - Modo de usuario: las aplicaciones no tienen acceso completo a los recursos de hardware.
  - Modo kernel: el OS tiene acceso a todos los recursos de la máquina,

¿Por qué cambiar entre procesos?

- A veces, cuando el OS está en modo kernel, no puede volver al mismo proceso que dejó, entonces:
  - El proceso ha salido o debe terminarse (por ejemplo, *segfault*).
  - El proceso ha realizado una llamada al sistema blocking.
- A veces, el OS no quiere volver al mismo proceso:
  - El proceso lleva demasiado tiempo.
  - Debe compartir el tiempo de CPU con otros procesos.

Problema 2: Cambio entre procesos.

- ¿Cómo puede el OS recuperar el control de la CPU para que pueda cambiar entre procesos?
  - Un enfoque cooperativo: **esperar las llamadas del sistema.**
  - Un enfoque no cooperativo: **el OS toma el control.**

Un enfoque cooperativo: esperar las llamadas del sistema

- Los procesos abandonan periódicamente la CPU haciendo llamadas al sistema.
  - El OS decide ejecutar alguna otra tarea.
  - La aplicación también transfiere el control al OS cuando hacen algo ilegal.
    - Dividir entre cero.
    - Intenta acceder a la memoria a la que no debería poder acceder.
  - Primeras versiones del OS Macintosh, el el sistema Xerox Alto.

```
Si proceso se atasca en un bucle infinito -> Reinicia la máquina.
```

Un enfoque no cooperativo: El OS toma el control

- **Una interrupción del temporizador**
  - Durante la secuencia de arranque, el OS inicia el temporizador.
  - El temporizador genera una interrupción cada tantos milisegundos.
  - Cuando se levanta la interrupción:
    - El proceso actualmente en ejecución se detiene.
    - Guarda el estado del programa.
    - Se ejecuta un controlador de interrupciones preconfigurado en el OS.
- Una interrupción del temporizador le da al OS la capacidad de ejecutarse nuevamente en una CPU.

El planificador del OS

- El planificador del OS tiene dos partes:
  - Una política para elegir qué proceso ejecutar.
  - Mecanismo para cambiar a ese proceso.
- Los planificadores no preventivos (cooperativos) son educados.
  - Cambian solo si el proceso está bloqueado o terminado.
- Los planificadores preventivos (no cooperativos) pueden cambiar incluso cuando el proceso está listo para continuar.
  - La CPU genera una interrupción periódica del temporizador.
  - Después de dar servicio a la interrupción, el OS comprueba si el proceso actual se ha ejecutado durante demasiado tiempo.

Guardar y restaurar contexto

El planificador toma una decisión:

- Ya sea para continuar ejecutando el proceso actual o para cambiar a uno diferente.
- Si se toma la decisión de cambiar, el OS ejecuta el **cambio de contexto**.

Mecanismo de cambio de contexto

Ejemplo

El proceso A ha pasado del modo de usuario al modo kernel, el OS decide que debe cambiar de A a B.

- Guardar el contexto (PC, registros, puntero de pila del kernel) de A en la pila de kernel.
- Cambia el SP a la pila del kernel de B.
- Restaura el contexto de la pila del kernel de B.
- Ahora, la CPU está ejecutando B en modo kernel, return-from-trap para cambiar al modo de usuario de B.

Pregunta

- ¿Quién ha guardado los registros en la pila del kernel de B?

A tener en cuenta sobre el cambio contexto

- Contextos (PC y otros registros de CPU) guardados en la pila del kernel en dos escenarios diferentes.
- Al pasar del modo de usuario al modo de kernel, el contexto de usuario (por ejemplo, en qué instrucción del código de usuario se detuvo) se guarda en la pila del kernel mediante la instrucción **trap**.
    - Restaurado por el return-from-trap.
- Durante un cambio de contexto, el código de cambio de contexto guarda el contexto del kernel (por ejemplo, donde se detuvo en el código del OS) en la pila del kernel de A.
    - Restaura el contexto del kernel del proceso B.

Algo más del cambio de contexto

- Código ensamblador de bajo nivel:
  - Guarda algunos valores de registro para el proceso actual en su pila de kernel.
    - Registros de uso general.
    - Puntero de la pila del kernel.
    - PC.
  - Restaura algunos para el proceso que pronto se ejecutará desde su pila de kernel.
  - Cambie a la pila del kernel para el proceso que pronto se ejecutará.

Concurrencia

- ¿Qué sucede si, durante la manipulación de interrupciones o *traps*, se produce otra interrupción?
- El SO maneja estas situaciones:
  - Deshabilita interrupciones (disable interrupts) durante el procesamiento de interrupciones.
  - Utiliza varios esquemas de bloqueo (locking) sofisticados para proteger el acceso concurrente a las estructuras de datos internas.

Fin!