.UBAfiuba
FACULTAD DE INGENIERÍA

Pedro Gallino <pgallino@fi.uba.ar>

**FISOP - Parcialito TP1**
1 mensaje

**Formularios de Google** <forms-receipts-noreply@google.com>
Para: pgallino@fi.uba.ar

6 de octubre de 2023, 19:25

Gracias por rellenar FISOP - Parcialito TP1

Esto es lo que se recibió.

# FISOP - Parcialito TP1

Parcialito sobre el TP1 de la materia Sistemas Operativos (FIUBA)

Se ha registrado tu correo (**pgallino@fi.uba.ar**) al enviar este formulario.

Antes de arrancar, dejanos tus datos.

Ingresá tu padrón: *

107587

Y tu nombre completo (apellido y nombre) *

Gallino Pedro

Preguntas

Son 20 preguntas en total.

¿Cuál es el mecanismo para setear las variables de entorno temporales?

*

- ○ En el proceso ejecutor del comando, se hace un setenv de cada variable (antes de hacer "exec")
- ○ Antes de crear el proceso ejecutor del comando, se hace un setenv de cada variable.
- ○ En el proceso ejecutor del comando, se pasan esos únicos valores como tercer argumento (eargv) a la syscall “exec”.
- ◉ Ninguna de las anteriores

Las características de un “file descriptor” son:

*

- ◉ Es una referencia al archivo subyacente (independientemente de la naturaleza de ese archivo).

- Es el archivo abierto “per se”.
- Cuando se cierra, se elimina directamente el archivo relacionado.
- No se puede duplicar

¿Cómo se logra la redirección de un flujo estándar en un archivo?

*

- (●) Se "apunta" el flujo estándar al archivo deseado
- ( ) Se envía un argumento extra como parte de la syscall “exec”
- ( ) Se envía un argumento extra como parte de la syscall “open” al abrir el archivo
- ( ) Ninguna de las anteriores

Todo proceso siempre comienza con tres “file descriptors” abiertos:

- Entrada estándar

- Salida estándar

- Un pipe para comunicarse con el padre

*

- ( ) Verdadero
- (●) Falso

La syscall “exec” reemplaza todo el *address space* del proceso actual (datos + código binario) pero preserva la configuración de los “file descriptors”:

*

- (●) Verdadero
- ( ) Falso

Cuando creo un nuevo proceso con “fork”:

*

- [x] El código binario del proceso nuevo es el mismo que el del padre
- [x] Los “file descriptors” son un duplicado de los que tenía el padre (referencian a los mismos archivos).
- [ ] La ejecución arranca desde el comienzo del programa.
- [ ] Las variables de entorno del proceso nuevo se resetean (no comparte ninguna con el padre)
- [ ] Todas las anteriores

Cuando la shell realiza la redirección de la salida estándar (*stdout*) en un archivo, los datos se envían tanto a la pantalla como al archivo:

*

- ( ) Verdadero

- [x] Falso

¿Cómo se produce la expansión de variables?

*

- [ ] La syscall “exec” reemplaza toda ocurrencia del patrón “$VARIABLE” por el valor de la misma.
- [x] La shell reemplaza toda ocurrencia del patrón “$VARIABLE” por el valor de la misma, antes de llamar a “fork”.
- [ ] En el proceso ejecutor, antes de hacer “exec”, se reemplaza toda ocurrencia del patrón “$VARIABLE” por el valor de la misma
- [ ] El binario que se termina ejecutando las reemplaza como parte de su código

La expansión de una variable que no existe, por ejemplo "echo hola $NO_EXISTE", resulta en que a "exec" le llegue:

*

- [x] exec("echo", ["echo", "hola", ""])
- [ ] exec("echo", ["echo", "hola"])
- [ ] exec("echo", ["echo", "hola", " "])
- [ ] exec("echo", ["echo", "hola", "\n"])

Los valores de las variables “mágicas”:

*

- ( ) Se cargan en la inicialización de la shell, para luego ser consumidas
- (•) Se obtienen en runtime de acuerdo al estado de la shell
- ( ) Se obtienen de variables de entorno especiales que dispone el kernel
- ( ) La syscall “exec” es capaz de obtener esos valores y expandirlos.

La función exit() a diferencia de _exit():

*

- ( ) Libera la memoria y “file descriptors” alocados por el proceso para que, al terminar, el sistema operativo no pierda memoria de manera permanente.
- ( ) Es meramente un wrapper de la syscall exit.
- (•) Realiza algunas tareas de mantenimiento relacionadas con estructuras creadas por la libc (biblioteca estándar de C) antes de llamar a la syscall exit.
- ( ) No existe ninguna diferencia y son aliases una de la otra por motivos de compatibilidad con versiones anteriores de la libc.

Sobre el comando “cd”:

*

- [ ] Puede implementarse perfectamente como binario ejecutable.
- [x] Debe ser un built-in de la shell para que cumpla su cometido.
- [ ] Debe ser un built-in de la shell por motivos de performance.
- [ ] Se implementa con la syscall "cd" (mismo nombre)

Sobre el comando “pwd”:

*

- [x] Se puede implementar tanto como binario ejecutable como built-in
- [ ] Existe solamente como binario ejecutable
- [ ] Existe solamente como built-in
- [ ] No es un comando válido de la shell

La ejecución de los comandos en “pipe”:

*

- [x] Ocurren en simultáneo: es decir, el comando de la izquierda escribe mientras el comando de la derecha ya está leyendo.
- [ ] Ocurren en secuencia: es decir, el comando de la derecha tiene que esperar a que termine el de la izquierda para poder ser ejecutado.
- [ ] Ocurre en orden inverso: es decir, el comando de la derecha se ejecuta antes que el izquierdo pueda iniciar.

- [ ] Ninguna de las anteriores

Para un comando de tipo “pipe”:

*

- [x] La shell espera a que terminen ambos procesos para devolver el prompt
- [ ] La shell solamente espera a que termine el comando de más a la izquierda
- [ ] La shell solamente espera a que termine el comando de más a la derecha
- [ ] La shell no espera por ninguno y devuelve el prompt inmediatamente

Un comando ejecutado en “background”:

*

- [ ] Es un proceso al cual nunca se le hace “wait”
- [x] Se lo “monitorea” para que cuando finalice no quede zombie
- [ ] No puede tener redirección de su flujo estándar
- [ ] Todas las anteriores

¿Qué ocurre cuando una señal interrumpe la ejecución de una syscall? *

- [ ] La syscall se reanuda automáticamente cuando termina la ejecución del handler de la señal

- [x] La syscall se reanuda únicamente cuando se configuró el handler apropiadamente
- [ ] La syscall nunca se reanuda y falla con el error EINTR
- [ ] No es un comportamiento que esté definido

La configuración de los handlers de señales: *

- [x] Se preservan a través de un "fork(2)"
- [ ] Se preservan a través de un "exec(2)"
- [ ] No se preservan a través de un "fork(2)"
- [x] No se preservan a través de un "exec(2)"

Es necesario colocar a los procesos en segundo plano en un mismo grupo: *

- [ ] Para que al hacer "exec" no se genere un error con los flujos de redirección estándar
- [x] Para que el "wait" del handler de SIGCHILD sea efectivo y libere los recursos del proceso
- [ ] Para que efectivamente el proceso pueda correr en segundo plano
- [ ] Ninguna de las anteriores

Cuando se llama "waitpid(0, ...)" el comportamiento es: *

- [x] Espera por todos los procesos hijos cuyo PGID sea el mismo que el del proceso que ejecuta la syscall
- [ ] Esperar por cualquier proceso hijo

- Esperar por el proceso hijo cuyo PID es el cero
- Es un argumento inválido y la syscall falla