Preguntas 90 de 100 puntos

Son 20 preguntas en total.

✓ La syscall “exec” reemplaza todo el *address space* del proceso actual (datos + código binario) pero preserva la configuración de los “file descriptors”: *5/5

- (●) Verdadero ✓
- ( ) Falso

✓ Un comando ejecutado en “background”: * 5/5

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

✗ ¿Cómo se logra la redirección de un flujo estándar en un archivo? * 0/5

- ○ 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

Respuesta correcta

- ◉ Se "apunta" el flujo estándar al archivo deseado

✓ Cuando se llama "waitpid(0, ...)" el comportamiento es: * 5/5

- ○ Esperar por cualquier proceso hijo
- ◉ Espera por todos los procesos hijos cuyo PGID sea el mismo que el del proceso que ejecuta la syscall ✓
- ○ Es un argumento inválido y la syscall falla
- ○ Esperar por el proceso hijo cuyo PID es el cero

✓ ¿Cuál es el mecanismo para setear las variables de entorno temporales? * 5/5

- (●) 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

✓ ¿Cómo se produce la expansión de variables? * 5/5

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

✗ Los valores de las variables “mágicas”: * 0/5

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

Respuesta correcta

- (●) Se obtienen en runtime de acuerdo al estado de la shell

✓ La expansión de una variable que no existe, por ejemplo "echo hola $NO_EXISTE", resulta en que a "exec" le llegue: *5/5

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

✓ La configuración de los handlers de señales: * 5/5

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

✓ Las características de un “file descriptor” son: * 5/5

- (●) Es una referencia al archivo subyacente (independientemente de la naturaleza de ese archivo). ✓
- ( ) Cuando se cierra, se elimina directamente el archivo relacionado.
- ( ) Es el archivo abierto “per se”.
- ( ) No se puede duplicar

✓ Sobre el comando “pwd”: * 5/5

- ○ Existe solamente como built-in
- ◉ Se puede implementar tanto como binario ejecutable como built-in ✓
- ○ Existe solamente como binario ejecutable
- ○ No es un comando válido de la shell

✓ Para un comando de tipo “pipe”: * 5/5

- ○ La shell no espera por ninguno y devuelve el prompt inmediatamente
- ◉ 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 ejecución de los comandos en “pipe”: * 5/5

- ◉ 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

✓ Sobre el comando “cd”: * 5/5

- ○ Se implementa con la syscall "cd" (mismo nombre)
- ◉ 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.
- ○ Puede implementarse perfectamente como binario ejecutable.

✓ La función exit() a diferencia de _exit(): * 5/5

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

✓ 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: * 5/5

- ○ Verdadero
- ◉ Falso ✓

✓ ¿Qué ocurre cuando una señal interrumpe la ejecución de una syscall? * 5/5

- ○ La syscall se reanuda automáticamente cuando termina la ejecución del handler de la señal
- ○ No es un comportamiento que esté definido
- ◉ La syscall se reanuda únicamente cuando se configuró el handler apropiadamente ✓
- ○ La syscall nunca se reanuda y falla con el error EINTR

✓ Todo proceso siempre comienza con tres “file descriptors” abiertos: * 5/5

- Entrada estándar
- Salida estándar
- Un pipe para comunicarse con el padre

- ◉ Falso ✓
- ○ Verdadero

✓ Es necesario colocar a los procesos en segundo plano en un mismo grupo: *5/5

- ○ Para que al hacer "exec" no se genere un error con los flujos de redirección estándar
- ◉ 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 creo un nuevo proceso con “fork”: * 5/5

- [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