PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2333 — Sistemas Operativos y Redes — 1/2021

Tarea 1

Fecha de Entrega: Miércoles 28 de Abril, 23:59
Composición: Tarea en parejas

Objetivos

- Utilizar *syscalls* para construir un programa que administre el ciclo de vida de un conjunto de procesos.
- Comunicar múltiples procesos por medio del uso de señales.

CR-TREE

El profesor Cristian, cansado de que el servidor se quede sin recursos por la ineficiencia en su forma de manejar procesos, les pide a los alumnos de Sistemas Operativos™, que en tan solo tres clases se volvieron expertos en procesos, que prueben en la práctica una nueva forma de generar procesos que ha ideado. Este mecanismo se llama `CR-TREE` y deben construir un programa que permita simular su funcionamiento.

Procesos

En `CR-TREE` hay dos tipos de procesos, los procesos *worker* y los procesos *manager*. Éstos últimos se separan en los procesos *manager root* y los procesos *manager* no *root*. Tanto los procesos *worker*, *manager* y *manager root* tienen un identificador asociado.

Proceso *Worker*

El identificador asociados a estos procesos es el caracter "`W`". Estos procesos se encargan de ejecutar un programa y llevar cuenta de algunas estadísticas de este. En específico las estadísticas que llevan son:

1. El nombre del programa a ejecutar.
2. Los argumentos por consola del programa a ejecutar.
3. Cuanto tiempo se demoró en ejecutar el programa.
4. El código de retorno del programa ejecutado.
5. Si es que el proceso *worker* fue interrumpido.

Una vez terminada la recopilación de estadísticas, estas deben ser guardados en un archivo. Es decir, cada proceso *worker* tendrá como *output* un archivo.

Además, los procesos *worker* deben interceptar las señales `SIGINT` y `SIGABORT`. En el caso de `SIGINT`, esta señal debe ser ignorada y el programa debe seguir funcionando con normalidad. En el caso de `SIGABRT` esta señal se considera como interrupción y el proceso debe escribir su archivo antes de terminar.

Finalmente, un proceso *worker* **no debe dejar procesos huérfanos ni *zombies***, es decir, tiene que terminar solo después de que el programa que está ejecutando haya terminado y debe hacer `wait` a este.

Procesos *Manager*

Estos procesos tienen como identificador el caracter "`R`" en caso de ser *root* y "`M`" si es que son *managers* no *root*. Ambos tipos de proceso se encargan de **iniciar y monitorear** una serie de procesos hijos, los cuales pueden ser *managers* o *workers*. Además todo *manager* recibe como *input* un tiempo límite denominado `timeout`.

Al iniciar, el *manager* debe crear un nuevo proceso por cada uno de sus hijos asignados, y esto debe **ocurrir inmediatamente**, es decir, el *manager* inicia a sus procesos hijos uno después del otro sin esperar a que cada uno termine antes de iniciar el siguiente.

Una vez iniciados los procesos hijos, el proceso *manager* espera a que pase cualquiera de los siguientes casos:

1. **Termine la ejecución de todos sus hijos**:

   Cuando esto pase el proceso debe tomar todos los archivos generados por sus hijos y unirlos en su propio archivo de output. Es decir, que al igual que los *workers*, el output de los procesos *manager* es un archivo.

2. **Se alcance el tiempo de `timeout`**:

   Si esto pasa el proceso debe mandar una señal a todos sus hijos. En específico debe mandar la señal `SIGABRT`, luego una vez terminados los hijos une sus archivos en el suyo. Cabe recalcar que si todos los hijos terminan antes del `timeout` el *manager* debe hacer la unión de archivos y terminar **inmediatamente**. Por ejemplo, si el `timeout` es de dos años y los hijos se demoran 5 segundos en terminar el *manager* debiese terminar en aproximadamente 5 segundos, no dos años.

3. **Reciba una señal**:

   Si el proceso recibe un `SIGABRT`, envía `SIGABRT` a sus hijos y une sus archivos. En cambio si recibe un `SIGINT` puede pasar una de dos cosas:

   3.1) El proceso es `root`, en cuyo caso propaga `SIGABRT` a sus hijos.

   3.2) El proceso no es `root`, en cuyo caso ignora la señal y continua con su funcionamiento normal.

Un proceso *manager* **no debe dejar procesos huérfanos ni *zombies***, es decir, tiene que terminar solo después de que todos sus hijos hayan terminado y debe hacer `wait` a todos.

Funcionalidad del programa

Su programa deberá leer un archivo de entrada que contiene las descripciones de múltiples procesos *manager* y *worker* y deberá ejecutarlos siguiendo las reglas descritas en la sección anterior.

Ejecución

El programa principal será ejecutado por línea de comandos con las siguiente sintaxis:

```
./crtree <input> <process>
```

Donde:

- `<input>` es la ruta de un archivo de entrada, el cual posee la lista de los procesos a ejecutar.
- `<process>` es índice del proceso desde donde debe empezar la ejecución.

Un ejemplo de ejecución podría ser el siguiente:

```
./crtree input.txt 0
```

Esto significa que en el archivo `input.txt` se encuentra la descripción de nuestros procesos y queremos iniciar ejecutando el primero (índice 0).

Archivo de Entrada (*input*)

El archivo de entrada será un archivo de texto plano, en el cual la primera línea es un número que nos dice cuantos procesos están descritos en este. Luego, cada una de las siguientes líneas representará un proceso.

- Línea de proceso *manager*:

  `identificador,timeout,n,linea_1,...,linea_n`

  El significado de cada uno de los valores es:

  1. `identificador`: Es el identificador del proceso `manager`, puede ser "`R`" o "`M`".
  2. `timeout`: Es el tiempo límite de ejecución de los hijos en segundos. Es un número entero.
  3. `n`: Es la cantidad de procesos hijos que tiene el *manager*.
  4. `linea_i`: Es un entero que representa uno de los procesos hijos del *manager*. En específico es el índice del proceso en el archivo de entrada **sin considerar la primera línea**[1].

- Línea de proceso *worker*:

  `identificador,ejecutable,n,arg_1,...,arg_n`

  El significado de cada uno de los valores es:

  1. `identificador`: Es el identificador del proceso `worker`, su valor es "`W`".
  2. `ejecutable`: Es el ejecutable que el `worker` debe ejecutar.
  3. `n`: Es la cantidad de argumentos por consola que recibe el *worker*.
  4. `arg_i`: Es uno de los argumentos por consola que recibe el *worker*.

Un ejemplo de archivo de entrada es el siguiente:

```
5
W,./hello_world,0
M,15,2,0,2
W,./sum,2,2,5
R,100,2,1,4
W,./avg,9,1,2,3,4,5,6,7,8,9
```

Podemos ver que los procesos de índice 0, 2 y 4 son *workers*, mientras que el proceso 1 es un *manager* y el proceso 3 es un *manager root*. Los procesos *workers* ejecutan los siguientes programas:

- El proceso 0 ejecuta el programa `./hello_world`.
- El proceso 2 ejecuta el programa `./sum 2 5`.
- El proceso 4 ejecuta el programa `./avg 1 2 3 4 5 6 7 8 9`.

También podemos ver que el proceso 3 tiene un `timeout` de 100 segundos y sus hijos son los procesos 1 y 4, mientras que el proceso 1 tiene un `timeout` de 15 segundos y sus hijos son los procesos 0 y 2.

[1] Por ejemplo, `linea_i=2` representaría la cuarta línea del archivo.

Archivo de Salida (*output*)

El *output* a evaluar de esta tarea son los archivos generados. Todo proceso al terminar debe escribir un archivo cuyo nombre sea `<indice>.txt` donde `<indice>` es el índice del proceso en el archivo de entrada **ignorando la primera línea**.

Archivo *Worker*

El archivo generado por los *workers* debe seguir el siguiente formato:

```
PROGRAM,ARG_1,...,ARG_N,TIME,RETURN_CODE,INTERRUPTED
```

Donde `PROGRAM` es el nombre del ejecutable, `ARG_i` son los argumentos por consola del ejecutable, `TIME` es el tiempo en segundos que duró la ejecución, `RETURN_CODE` es el código con de retorno del ejecutable y `INTERRUPTED` índica si el programa fue interrumpido.

Por ejemplo, si el *worker* debe ejecutar el programa `./sum 1 2` y este se demoró 2 segundos, tuvo código de retorno 0 y no fue interrumpido, entonces el archivo sería:

```
./sum,1,2,2,0,0
```

Donde el `0` indica que no ocurrió ninguna interrupción. En caso de haber ocurrido una interrupción, se debe ingresar un 1. Cabe notar que para el proceso *worker* recibir un `SIGINT` no cuenta como interrupción, ya que debe ignorarlo.

Archivo *Manager*

El archivo generado por los *managers* es la unión, o mejor dicho la concatenanación de los archivos generados por todos sus hijos. Es decir, los archivos creados se verían de la forma:

```
PROGRAM_1,ARG_1_1,...,ARG_N_1,TIME_1,RETURN_CODE_1,INTERRUPTED_1
...
PROGRAM_i,ARG_1_i,...,ARG_N_i,TIME_i,RETURN_CODE_i,INTERRUPTED_i
...
PROGRAM_N,ARG_1_N,...,ARG_N_N,TIME_N,RETURN_CODE_1,INTERRUPTED_N
```

Siendo N la cantidad de familiares[2] que tiene el *manager*.

Aspectos a tener en consideración

- El programa puede partir en cualquier linea del archivo, a excepción de la primera.
- Solo habrá un proceso *manager root*.
- El proceso *manager root* estará conectado a todos los procesos.
- No habrán dos procesos *manager* que compartan un proceso hijo.
- Debe utilizar la **API POSIX** (`fork`, `exec`, `wait`, ...).

[2] Es decir, sus hijos, nietos, bisnietos, tataranietos ...

Formalidades

A cada alumno se le asignó un nombre de usuario y una contraseña para el servidor del curso[3]. Para entregar su tarea usted deberá crear una carpeta llamada `T1` en el directorio principal de su carpeta personal y subir su tarea a esa carpeta. En su carpeta `T1` **solo debe incluir el código fuente** necesario para compilar su tarea y un `Makefile`. Se revisará el contenido de dicha carpeta el día Miércoles 28 de Abril, 23:59.

Solo uno de los integrantes de cada grupo debe entregar en su carpeta. Los grupos los elegirán ustedes y en caso de que alguien no tenga compañero, puede crear una *issue* en el foro del curso buscando uno.

Antes de acabado el plazo de entrega de la tarea, se enviará un form donde podrán registrar los grupos junto con el nombre de usuario de la persona que realizará la entrega vía el servidor.

- **NO debe incluir archivos binarios**. En caso contrario, tendrá un descuento de 0.5 puntos en su nota final.
- Su tarea deberá compilar utilizando el comando `make` en la carpeta `T1`, y generar un ejecutable llamado `edf` en esta misma. Si su programa **no tiene** un `Makefile`, tendrá un descuento de 0.5 puntos en su nota final.
- Es muy importante que su tarea corra dentro del servidor del curso. Si esta **no compila** o **no funciona** (*segmentation fault*), obtendrán la nota mínima, teniendo como base 0.5 puntos menos en el caso de que soliciten recorrección.

El no respeto de las formalidades o un código extremadamente desordenado podría originar descuentos adicionales. Se recomienda modularizar, utilizar funciones y ocupar nombres de variables explicativos. En el caso de no entregar en la carpeta especificada, la tarea **no** se corregirá.

Evaluación

- **0.5 pts.** Lectura de `stdin`. Paso de argumentos. Construcción de `argc` y `argv`.
- **1.0 pts.** Correcta implementación de proceso *manager*.
- **1.0 pts.** Correcta implementación de proceso *worker*.
- **1.5 pts.** Comunicación entre procesos por medio de señales[4].
- **1.0 pts.** *Output* correcto.
- **1.0 pts.** Manejo de memoria. Se obtiene este puntaje si `valgrind` reporta en su código 0 *leaks* y 0 errores de memoria en **todo caso de uso**[5].

Preguntas

Cualquier duda preguntar a través del foro oficial.

[3] `iic2333.ing.puc.cl`
[4] En este punto se considera que los procesos *manager* no dejen procesos **huérfanos ni *zombies***
[5] Es decir, debe reportar 0 *leaks* y 0 errores para todo *test*, sin importar si este termina normalmente o por medio de una interrupción.