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

IIC2333 — Sistemas Operativos y Redes — 2/2021

Tarea 1

Fecha de Entrega: Lunes 6 de Septiembre, 23:59
Ayudantía: Grabada
Composición: Tarea en parejas

Objetivos

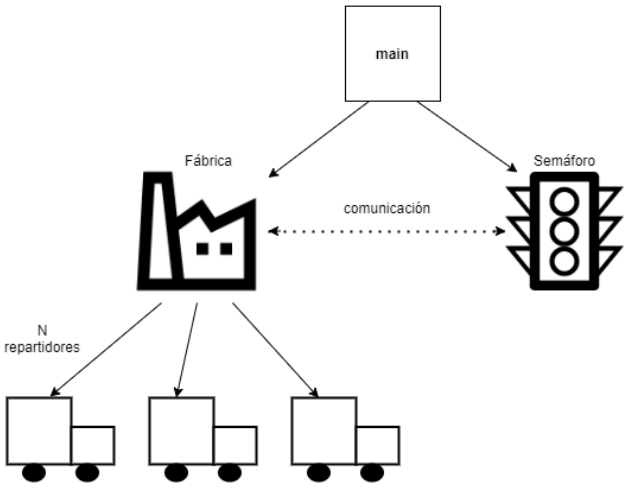
- Utilizar *syscalls* para construir un programa que administre un conjunto de procesos.
- Comunicar múltiples procesos por medio del uso de señales.

DCCUBER

La empresa `DCCUBER` es reconocida por sus cajas en forma de cubo. Dada su reciente popularidad ha tenido que arrendar una bodega para mantener mayor stock. Para esto debe contar con repartidores que lleven las cajas de la fábrica a la bodega. Lamentablemente los autos repartidores están tan llenos de cajas que el conductor no puede mirar al frente. Por esto, la fábrica les pide que implementen una solución a través comunicación de procesos para avisar a los conductores el estado de los semáforos y así sepan cuando frenar o seguir avanzando.

Esta tarea consiste en simular el viaje de un `Repartidor` entre la `Fábrica` y la bodega. Los `Semáforos` en el camino frenan temporalmente el avance del `Repartidor`. La simulación se construirá mediante procesos del sistema operativo Linux que deberán ser creados y detenidos mediante *syscalls* de manejo de procesos.

Un proceso `Fábrica` crea a los procesos `Repartidor`, habrá tres procesos `Semáforo` independientes. El proceso `semáforo` **NO puede comunicarse con los procesos `Repartidor`**.



Procesos

Estructura General

En `DCCUBER` hay cuatro tipos de procesos:

- Proceso **principal** (*main*)
- Proceso **fábrica**
- Procesos **repartidor**
- Procesos **semáforo**

Todos estos procesos pueden ser identificados por su PID. El flujo que siguen estos procesos es el siguiente:

1. El proceso **principal** crea al proceso **fábrica** y a los tres procesos **semáforo**.
2. El proceso **fábrica** crea procesos **repartidor** periódicamente.
3. Los procesos **repartidores** finalizan al llegar a la bodega.
4. El proceso **fábrica** finaliza al haber finalizado todos los procesos **repartidor** necesarios.
5. El proceso **principal** indicará a los procesos **semáforo** que deben finalizar una vez haya finalizado el proceso **fábrica**.

Para compilar su programa se les entregará un Makefile **que deberán usar**. El resultado de la compilación de su programa deben ser tres ejecutables, `dccuber` (para los procesos principal y fábrica), `semaforo` (para el proceso semáforo) y `repartidor` (para el proceso repartidor).

Proceso Principal

Su implementación deberá estar en la carpeta `src/dccuber`.

Este es el proceso principal, es decir, el que es iniciado cuando ejecutamos `./dccuber`. Su rol es inicializar el proceso `fabrica` y los procesos `semaforo`, e indicarle a los procesos `semaforo` que deben finalizar, y encargarse de esperar al proceso `fábrica` y a los procesos `semaforo`.

Proceso Semáforo

Su implementación deberá estar en la carpeta `src/semaforo`.

El ejecutable debe recibir como mínimo los siguientes argumentos por consola:

- `id`: Un identificador único para el semáforo. Es de libre elección qué elemento se usará como identificador.
- `delay`: El intervalo en segundos que le toma al semáforo cambiar de color.
- `parent`: El PID del proceso fábrica.

Al cambiar de color el proceso semáforo debe enviarle a la fábrica una señal con su identificador único para indicarle que ha cambiado de color. Todos los semáforos parten en color **verde**. **Solo los procesos principal y semáforo pueden conocer el *delay***.

Este proceso **debe** ser creado mediante `fork` y `exec` desde el proceso `principal`.

Proceso Fábrica

Su implementación deberá estar en la carpeta `src/dccuber`.

Este proceso debe crearse antes que los semáforos y los repartidores. Se encargará de crear procesos `repartidor`

según un intervalo de tiempo[1] hasta llegar a un número de `N` repartidores. También debe avisar a los repartidores del cambio de color de los semáforos mediante señales.

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

Proceso Repartidor

Su implementación deberá estar en la carpeta `src/repartidor`.

Una vez iniciado, el repartidor avanza a una velocidad de una unidad por segundo hacia la bodega. La fábrica le informará el estado de los semáforos, y en caso de encontrarse con uno en rojo no podrá avanzar y deberá intentarlo de nuevo en el siguiente segundo.

Este proceso `debe` ser creado mediante un `fork` y `exec` desde el proceso `fábrica`.

Estos procesos se encargan de ejecutar un programa y llevar cuenta de algunas estadísticas. Se debe mantener:

1. Cantidad de turnos desde el inicio hasta que cruza[2] el semáforo 1.
2. Cantidad de turnos desde el inicio hasta que cruza el semáforo 2.
3. Cantidad de turnos desde el inicio hasta que cruza el semáforo 3.
4. Cantidad de turnos de llegada la bodega.

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

Comunicación entre Procesos

Señales

Dentro del sistema toda comunicación entre los procesos deberá ser mediante el uso de señales. Se diferenciará esta comunicación en dos casos posibles:

- Señales de información
- Señales de término

Las señales de información serán todas las señales para traspasar información entre procesos. En este caso serían las señales usadas para informar el estado de los semáforos.

Las señales de término serán todas las señales usadas para indicar que el proceso debiese terminar. En este caso serían usadas para que los procesos puedan escribir sus archivos antes de finalizar de forma abrupta.

Para ambos casos deberán enviar las señales con `kill/sigqueue` y definir una función *handler* que las procese con `signal/sigaction`. En el caso de las señales de información se deberá usar `sigqueue` y `sigaction` mientras que en el caso de las señales de término se deberá usar `kill` y `signal`.

---

1 Se recomienda usar **alarm** para esto

2 El repartidor cruza un semáforo cuando la posición del repartidor es igual a la posición de este.

Señales de Término

Como señales de término se usarán `SIGINT`[3] y `SIGABRT` y se deberá usar `signal` y `kill`.

Los casos de uso de `SIGABRT`:

- Si un proceso repartidor la recibe, debe escribir inmediatamente su archivo y finalizar.
- Si el proceso fábrica la recibe, debe propagar la señal a todos los repartidores[4] y finalizar.
- Si un proceso semáforo la recibe, debe escribir inmediatamente su archivo y finalizar.
- El proceso principal le enviará esta señal a los procesos semáforo en cuanto haya detectado el término del proceso fábrica.

Los casos de uso de `SIGINT`:

- Si el proceso principal lo recibe, debe manda un `SIGABRT` al proceso fábrica y a los procesos semáforo y espera a que esos procesos terminen antes de finalizar.
- Todos los demás procesos deben ignorar esta señal y continuar funcionando normalmente.

Señales de Información

Si bien el caso más común de uso de señales es simplemente enviar una señal, existe la opción de enviarla junto a información adicional. Para esto se deben usar las funciones `sigaction` y `sigqueue` que nos permiten una forma más avanzada de comunicación de señales entre procesos.

Sin embargo, el uso de estas funciones es un poco más avanzado, por lo que tendrán como código base de dos funciones que les facilitará usar este caso de comunicación con señales:

- `connect_sigaction`: Esta función conecta un *handler*[5] a una señal. Recibe la señal a interceptar y la función a utilizar. Esta función usa `sigaction` para configurar el *handler*.
- `send_signal_with_int`: Esta función envía a un proceso la señal `SIGUSR1`[6]. Recibe el pid del proceso y el número a enviar. Esta función usa `sigqueue` para enviar la señal.

Funcionalidad del programa

Su programa deberá leer un archivo de entrada que contiene las descripciones de las distancias, y de los procesos *semáforo* y *fábrica* los que deberá ejecutar siguiendo las reglas descritas en la sección anterior.

Ejecución

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

```
./dccuber <input>
```

Donde `<input>` es la ruta de un archivo de entrada, el cual posee las instrucciones del programa a ejecutar.

Un ejemplo de ejecución podría ser el siguiente:

```
./dccuber input.txt
```

---

[3] Cuando haces `CTRL+C` en una consola el sistema operativo manda un `SIGINT` a todo el grupo de procesos asociado al programa actualmente siendo ejecutado en la consola

[4] Debe asegurarse que estos hayan terminado antes de seguir

[5] La función handler debe seguir la siguiente estructura de tipos: `void *handler(int, siginfo_t *, void *)`, siendo posible cambiarle el nombre `handler` al nombre de la función que hayas creado.

[6] Las señales `SIGUSR1` y `SIGUSR2` son pensadas para ser "de uso general". La gracia de esto es que no tienen un significado semántico asociado (por ej, `SIGINT` significa que queremos interrumpir un programa). Esto las hace idóneas para este caso de uso.

Esto significa que en el archivo `input.txt` se encuentra la descripción de nuestros procesos.

Archivo de Entrada (*input*)

El archivo de entrada será un archivo de texto plano, en el cual la primera línea es la línea de distancias, y la segunda línea corresponde a la fábrica y los tres semáforos respectivamente:

- Línea de distancias:

  `distancia_semaforo1,distancia_semaforo2,distancia_semaforo3,distancia_bodega`

  Los valores corresponden a las distancias desde la fábrica a cada lugar. Puedes suponer que es un camino recto y único, por lo que, por ejemplo, `distancia_semaforo2` - `distancia_semaforo1` resulta en la distancia entre ambos semáforos.

- Línea de procesos:

  `tiempo_de_creacion,envios_necesarios,tiempo_1,tiempo_2,tiempo_3`

  Donde los cada uno de los valores signficia:
  1. `tiempo_de_creacion`: Es la cantidad de tiempo en segundos que le toma a la `fabrica` en sacar un nuevo `repartidor` en dirección a la bodega.
  2. `envios_necesarios`: Es la cantidad de envios que se necesitan terminar para que finalice el proceso, que es igual a la cantidad total de `repartidores` a crear.
  3. `tiempo_i`: Es la cantidad de tiempo en segundos que le toma al `semaforo` número `i` en cambiar entre verde y rojo.

Un ejemplo de archivo de entrada es el siguiente:

```
2,3,5,9
1,8,4,6,7
```

Podemos ver que la distancia desde la fabrica al *semáforo 3* es de 5, mientras que a la *bodega* es de 9, por lo que se desprende que la distancia del *semáforo 3* a la *bodega* es de 4.

Por otra parte, tenemos que el primer semaforo cambia su estado cada 4 segundos, por lo que del segundo 0 al 3 está *verde*, y cambia a rojo en el segundo 4. Tenemos además que la *fábrica* envía los vehículos a la bodega en los segundos $1, 2, 3, ...$ La creación se realiza hasta cuando 8 repartidores hayan sido creados.

Archivo de Salida (*output*)

El *output* a evaluar de esta tarea son los archivos generados. Todo proceso a excepción del proceso `principal` deben escribir un archivo al terminar su ejecución.

Archivo *semáforo*

El nombre del archivo generado por los procesos semáforo debe ser `semaforo_<id>.txt` donde `<id>` es el identificador único del semáforo. El archivo debe seguir el siguiente formato:

```
CANTIDAD_DE_CAMBIOS
```

Donde `CANTIDAD_DE_CAMBIOS` es la cantidad de veces que el semáforo cambió de color. Un archivo posible es uno que se llamaría **semaforo_2.txt** y que posea el siguiente contenido:

```
8
```

Lo que significaría que el semáforo de `id` 2 cambió 8 veces de color.

Archivo repartidor

El nombre del archivo generado por los procesos repartidor debe ser `repartidor_<indice>.txt` donde `<indice>` es el numero del repartidor creado (un valor que va de `0` a `envios_necesarios-1`), debe seguir el siguiente formato:

```
TIEMPO_SEMAFORO1,TIEMPO_SEMAFORO2,TIEMPO_SEMAFORO3,TIEMPO_BODEGA
```

Donde cada `TIEMPO` es la cantidad de turnos de un segundo que se tarda en llegar a cada lugar desde el inicio de la ejecución del repartidor.

En caso de recibir un `SIGABRT`, se llenan los valores de los tiempos que no fueron obtenidos con `-1`.

Por ejemplo, si el repartidor se tardó 2 turnos en cruzar el cruze del primer semáforo (lo que necesita que el semáforo haya estado en verde), un archivo posible es:

```
2,3,5,8
```

Notar que el `3` indica que al *repartidor* le tomó 3 turnos (que dura un segundo cada uno) en cruzar el cruce del segundo semáforo desde la fábrica.

Aspectos a tener en consideración

- El programa de la *fábrica* puede finalizar cuando hay repartidores en el camino
- Solo habrá un proceso *fábrica*.
- Puedes asumir que no habrán semáforos en la posición 0
- El repartidor en su tiempo 0 no avanza.
- Debe utilizar la **API POSIX** (`fork`, `exec`, `wait`, ...).
- Puedes averiguar de lo que hace **signal**, **alarm**, **sleep**
- Si `Semáforo 1` está en distancia 5 y el conductor está en distancia 4. Para avanzar a 5, el conductor debe preguntar si el primer semáforo está en verde, y solo si es que lo está, avanza a 5, en caso contrario, sigue en 4.[7]

Formalidades

A cada alumno se le asignó un nombre de usuario y una contraseña para el servidor del curso[8]. 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 Lunes 6 de Septiembre, 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 *discussion* en el foro del curso buscando uno.

[7] Puedes pensarlo como que los semaforos están al inicio del cuadrado, que se indica con el número de distancia

[8] `iic2333.ing.puc.cl`

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 en 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 `dccuber` (junto a los ejecutables `semaforo` y `repartidor`) en esta misma. Si su programa **no usa** el `Makefile` entregado, 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 *fábrica*.
- **0.5 pts.** Correcta implementación de los procesos *semáforo*.
- **0.5 pts.** Correcta implementación de los procesos *repartidor*.
- **1.5 pts.** Comunicación entre procesos:
  - **1.0 pts.** Correcta comunicación mediante señales.
  - **0.5 pts.** No dejar procesos *huerfanos* ni *zombies*.
- **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**[9].

Preguntas

Cualquier duda preguntar a través del foro oficial.

[9] Es decir, debe reportar 0 *leaks* y 0 errores para todo *test*, sin importar si este termina normalmente o por medio de una interrupción.