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

IIC2333 — Sistemas Operativos y Redes — 1/2024

**Tarea 0**

**Fecha de Entrega: Lunes 8-Abril-2024 a las 21:00**
**Composición: Tarea en parejas**

# Objetivos

- Implementar el manejo del ciclo de vida de los procesos utilizando *syscalls*.
- Establecer un mecanismo de comunicación entre procesos mediante señales.
- Modelar la ejecución de un algoritmo de *scheduling* de procesos.

# Parte I: Procesos

## 1. Funcionalidad del programa `Runner`

Deberá crear un programa denominado `runner`, el cual deberá ser capaz de ejecutar múltiples programas de forma concurrente mediante la creación de procesos, y controlar que estos no superen un tiempo de ejecución determinado. Las tareas que debe cumplir su programa son:

1. Recibir los argumentos entregados a su programa por medio de la línea de comandos.
2. Leer un archivo, el cual contendrá instrucciones sobre la ejecución de un conjunto de programas.
3. Ejecutar el conjunto de programas externos **de forma paralela** mediante la creación de múltiples procesos. El número de procesos hijos que se encuentra en memoria no puede superar un valor $\langle amount \rangle$, por lo que, en caso de alcanzarse, se debe esperar a que uno de los procesos termine para continuar con la creación de nuevos procesos.
4. Capturar señales enviadas por el usuario y manejarlas.
5. Una vez se cumpla un tiempo $\langle max \rangle$ desde la ejecución de **`runner`**, este debe enviar una señal `SIGINT` a todos los procesos hijos en ejecución. En el caso que los procesos hijos no finalicen su ejecución pasados 10 segundos, se les debe enviar la señal `SIGTERM`.
6. Recolectar estadísticas de los programas externos ejecutados y entregarlas en un archivo en formato `.csv`.

## 2. Ejecución

El programa principal será ejecutado por líneas de comandos con la siguiente sintaxis:

```
./runner <input> <output> <amount> [<max>]
```

Donde:

- `<input>` es la ruta de un archivo de entrada, el cual posee la lista de todos los programas externos a ejecutar.

- `<output>` es la ruta de un archivo de texto con las estadísticas de la ejecución, en formato `.csv`, que debe ser escrito por su programa [1].
- `<amount>` corresponde a la cantidad máxima de programas externos que pueden estar corriendo de manera concurrente.
- `<max>` es un parámetro **opcional**, que es un entero que indica la cantidad máxima de segundos que puede demorarse en correr un programa antes de que sea terminado. Si no se entrega este parámetro, se debe considerar que los programas tienen tiempo ilimitado para ejecutar.

Por ejemplo, algunas ejecuciones podrían ser las siguientes:

```
./runner input.txt output.csv 5
./runner hola/input.txt output.csv 5 10
```

## 3. Archivo de entrada (*input*)

El archivo de entrada (*input*) será un archivo de texto, que empieza con una línea con un entero `N`, el cual indica la cantidad total de instrucciones. A continuación, hay `N` líneas de instrucciones donde cada una corresponde a una de las siguientes alternativas:

1. **Ejecutables externos =** Estas líneas contendrán las instrucciones de un programa a ejecutar, separados por un espacio:
   - Número de argumentos
   - Nombre del ejecutable (programa externo)
   - Argumentos para el ejecutable

   Por ejemplo, la siguiente es una línea válida que invoca a `/bin/python3` con 1 argumento:

   ```
   1 /bin/python3 hello.py
   ```

2. **Comando `wait_all` =** Esta línea describe el comando especial `wait_all`. Este comando debe bloquear[2] al programa principal hasta que los procesos en ejecución hayan terminado. La línea es de la siguiente forma:

   ```
   -1 wait_all timeout
   ```

   Donde `<timeout>` es el tiempo máximo de espera en segundos desde que el comando es procesado. Si un proceso no finaliza luego de transcurrir `<timeout>` segundos, se debe enviar la señal `SIGKILL` a los procesos correspondientes.

   En caso de que el tiempo `<max>` se cumpla antes de `<timeout>`, el programa debe seguir las instrucciones descritas en el punto 5 de la sección Funcionalidad del programa `Runner`.

El siguiente es un ejemplo de *input* válido:

```
5
2 ./servir_cliente 'choripan' 5
2 python3 play_cuecas.py 15
-1 wait_all 6
1 python3 comprar_pan.py
0 ./concretar_venta
```

[1] El output de su proceso **debe** ser el archivo ingresado, de lo contrario no se contará como correcto.
[2] Por lo que no debe procesar más input durante este periodo

Notar que:

- Puede existir un ejecutable con cantidad de argumentos igual a cero.
- La cantidad de argumentos indicada en cada línea siempre será correcta.

### 4. *Output*

El programa debe finalizar luego de terminar la ejecución de todas las instrucciones, o bien, en caso de recibir una interrupción del usuario. Al finalizar se debe escribir un archivo `.csv` con los resultados finales. Este *output* debe contener `N` líneas, una por cada programa externo indicado en el archivo de *input*, y donde cada línea debe seguir **estrictamente** el siguiente formato:

```
nombre_ejecutable,tiempo_ejecucion,status
```

Donde:

- `nombre_ejecutable` es el nombre del programa externo ejecutado.
- `tiempo_ejecucion` se refiere al tiempo de ejecución del programa externo. Si fue interrumpido, es el tiempo que alcanzó a ejecutar.
- `status` se refiere al estado del proceso, el cual depende de las señales emitidas. En caso de que el proceso termine exitosamente, considerar el *exit code*. Por ejemplo, si se emite una señal `SIGINT`, el estado sería 2.

## Interrupciones

Además de la ejecución de programas de forma concurrente, el programa principal debe ser capaz de **capturar** la señal `SIGTSTP`, la que se genera al oprimir Ctrl + Z . En el contexto de esta tarea, al recibir esta señal el programa principal **no debe** finalizar, sino que debe terminar el funcionamiento de los hijos actualmente instanciados, es decir, debe **interrumpir** todos los programas externos que estén siendo ejecutados concurrentemente en ese instante. En el caso que los procesos hijos no finalicen su ejecución pasados 10 segundos, deben ser **interrumpidos** con la señal `SIGKILL`. Posterior a esto, el programa principal debe escribir las estadísticas en el archivo de salida y finalizar su ejecución. Es importante notar que, naturalmente, esto incidirá tanto en las **estadísticas de tiempo** como en la cantidad de casos **anormales** en la ejecución.

# Parte II: *Scheduler*

## 1. *Schedulesly*

El scheduler que deberán implementar se trata de **una simulación** que se encargará de la administración de turnos en una única CPU de varios procesos agrupados según el **PID** del proceso, cuyo padre es el SO. Se considera el **PID** del **SO** como 0, así que los **PID** de los procesos parten en 1. Se usará una variación del esquema LIFO[3] dentro de cada grupo de procesos, y los grupos generales se ubicarán según el orden de creación del proceso.
Cada grupo de procesos es representado por una línea del input, y cada ciclo que está activo se le asignan ciertas **unidades de trabajo** (tiempo de CPU, *burst*) para que sus procesos corran. El orden en el que corren los procesos dentro de un grupo es el mismo que el del input, es decir, si una línea indica una acción que requiere de **unidades de trabajo**, esta acción debe cumplir todas sus unidades antes de pasar a la siguiente.

El orden de ejecución de cada grupo depende únicamente de cuándo llegaron al **SO**. Es el **SO** el que les asigna el **PID** a cada proceso y este lo asigna según su orden de llegada. Se entregarán varios grupos de procesos y cada uno contendrá la información mostrada en la sección 5.1 Archivo de entrada (*input*).

## 2. Proceso

Un proceso contiene la siguiente información:

- **PID**, el ID único de cada proceso, son asignados por el **SO** por orden creación del proceso (cuándo se lee el **CI**).
- **PPID**, el ID del proceso que creó al proceso actual. En caso de que no haya proceso padre, se dice que lo creó el **SO**, por lo que el valor del PPID usado en cuyo caso es 0.
- **GID**, el ID del grupo de procesos. Si es que el **PPID** del proceso es 0, se usa el **PID** del proceso. En caso contrario, se usa el **GID** del padre del proceso.
- **Estado**, que puede ser `READY`, `RUNNING`, `WAITING`, `FINISHED`.
- Además, podría tener un hijo que está siendo ejecutado.

## 3. Flujo

### 3.1. Trabajo de un proceso

Sea $q$ la cantidad de **unidades de trabajo** que posee actualmente un grupo. Un proceso del grupo ejecuta el mínimo entre $q$ y lo que le queda por trabajar. En caso de que el proceso se haya ejecutado una cantidad distinta de 0, se debe cambiar el estado de este a `RUNNING` y reportar el `RUN` del proceso al **SO**. Notar que si no se usó todo el valor de $q$ disponible, el grupo al que pertenece el proceso sigue usando las **unidades de trabajo** restantes.

### 3.2. Consideraciones

Se entenderá como que un grupo **puede trabajar** si es que necesita **unidades de trabajo** para seguir con el paso actual. En caso contrario, se considera que el grupo **no puede trabajar**.

**Algoritmo**:
Sea $q_i$ las **unidades de trabajo** que le asignó el **SO** a un grupo y $q$ la cantidad de **unidades de trabajo** que le quedan al grupo, asignando $q = q_i$, se hacen las siguientes operaciones en *loop* mientras $q > 0$ `OR NOT` **puede trabajar** el grupo.

1. Se revisa si es que el grupo no ha terminado con todos sus procesos. En caso de que todos sus procesos hayan terminado, termina el *loop*.

[3] Last in, first out

2. Si es que el grupo no **puede trabajar**:
    - Si el proceso terminó de ejecutarse, entonces se agrega este proceso a la lista de **procesos finalizados**.
    - Se avanza al siguiente paso.

    En otro caso, trabaja y el **SO** avanza el tiempo que se haya trabajado.
3. Finalmente se agrega al **output** la información de los cambios ocurridos en esta iteración del *loop*.

Una vez finalizado el loop, se actualiza el $q$ del grupo de la siguiente manera:

$$q_{i+i} := max\{q_i - qdelta, qmin\}$$

con $q_i$ el tiempo de trabajo **q** que tenía el grupo antes empezar el *loop*, y $qdelta, qmin$ parámetros presentes en el archivo de input

### 3.3. Flujo SO

La simulación del Scheduler hace el siguiente *loop* para simular al **SO**:

1. Revisa los grupos que tiene:
    - Si hay grupos que aún no empiezan la ejecución y que deberían haber empezado según su **TI**, el **SO** revisa cuáles puede agregar y los agrega al sistema según su **TI**[4]. Luego, en caso de que no haya grupos actualmente activos, el **SO** entra en estado `IDLE` y avanza hasta que llega el siguiente grupo.
2. El sistema itera por sus **grupos activos** (los que han sido atendidos por el **SO** y aún no han terminado) y va asignando las **unidades de trabajo** $q_i$ a cada grupo y haciéndolos trabajar en orden de llegada. En caso de que un grupo finalice, ese grupo sale del conjunto de **grupos activos** y se descarta.
3. Si ya han entrado y terminado todos los grupos, se da por terminada la simulación.

Cabe mencionar que el tiempo del **SO** inicia en $0$.

## 4. Ejecución de la Simulación

El simulador será ejecutado por línea de comandos con la siguiente instrucción:

```
./schedulesly <file> <output>
```

Y estos elementos representan los siguientes valores:

- `<file>` corresponderá a un nombre de archivo que deberá leer como *input*.
- `<output>` corresponderá a la ruta de un archivo `TXT` con las estadísticas de la simulación, que debe ser escrito por su proceso.[5]

## 5. Archivos

### 5.1. Archivo de entrada (*input*)

Los datos de la simulación se entregan como entrada en un archivo de texto, donde la primera línea indica la cantidad $K$ que corresponde líneas que vienen a continuación. La siguiente línea indica tres valores: *qstart* las **unidades de trabajo** que el **SO** le asigna inicialmente a cada grupo de procesos, *qdelta* el valor de cuánto bajan las **unidades de trabajo** del grupo cada iteración del **SO** por todos los grupos, y *qmin* que es la cantidad mínima de unidades de trabajo que se le pueden asignar a un grupo en cada iteración. Las siguientes $K - 1$ líneas siguen el siguiente formato:

[4] Como desempate usa en qué linea del input aparecieron, dando preferencia a las líneas menores
[5] El output de su proceso **debe** ser el archivo ingresado, de lo contrario no se contará como correcto.

```
TI CI NH CI_1 NH_1 ... CF_1 CE CI_2 NH_2 ... CF_2 CE CI_3 NH_3 ... CF_3 ... CF
```

El formato se lee de la siguiente manera (se agregan paréntesis para separar lo que no se trata del proceso inicial)

```
TI CI NH (CI_1 NH_1 ... CF_1) CE (CI_2 NH_2 ... CF_2) CE (CI_3 NH_3 ... CF_3) ... CF
```

Donde:

- `TI` es el tiempo en segundos de llegada del grupo a la cola. Considere que $\texttt{TI} \geq 0$. Notar que este valor sólo aparece una vez por línea.
- `CI` es la cantidad de tiempo en segundos que el proceso correrá antes de posiblemente crear a su hijo. Considere $\texttt{CI} \geq 1$.
- `NH` es la cantidad de procesos que creará el proceso.
- `CE` es la cantidad de tiempo en segundos que el proceso correrá entre creaciones de hijos. Notar que este valor aparecerá entre cada par de hijos y puede que no sea siempre el mismo. Considere $\texttt{CE} \geq 1$
- `CF` es la cantidad de tiempo en segundos que el proceso correrá después de crear todos sus hijos. Si el proceso tiene hijos, $\texttt{CF} \geq 1$, y si no tiene hijos el proceso no ejecuta después de la creación de hijos por lo que $\texttt{CF} = 0$

Puede utilizar los siguientes supuestos:

- Cada número es un entero no negativo y que no sobrepasa el valor máximo de un `int` de 32 bits.
- Habrá al menos un proceso descrito en el archivo.

El siguiente ejemplo ilustra cómo se vería un posible archivo de entrada:

```
3
30 10 12
3 5 2 4 0 0 4 20 0 0 8
7 10 1 10 1 10 1 10 0 0 3 3 3
```

**Ejemplo línea input grupo**

```
7 8 3 4 0 0 1 5 0 0 2 6 0 0 9
```

Explicación:

7 8 3 4 0 0 1 5 1 10 0 0 2 2 6 0 0 9

- **TI** $= 7$
- Para el primer proceso:
  - **CI** $= 8$
  - **NH** $= 3$
  - **CE** $= 1$ cuando se está entre el primer y segundo hijo
  - **CE** $= 2$ cuando se está entre el segundo y tercer hijo
  - **CF** $= 9$

- Para el primer hijo (segundo proceso)
  - **CI** $= 4$
  - **NH** $= 0$
  - **CF** $= 0$ (cuando un proceso no tenga hijos, este valor siempre será $0$ pero de todas formas se entrega en el input)
- Para el segundo hijo (tercer proceso)
  - **CI** $= 5$
  - **NH** $= 1$
  - **CF** $= 2$
- Para el hijo del segundo hijo (cuarto proceso)
  - **CI** $= 10$
  - **NH** $= 0$
  - **CF** $= 0$
- Para el tercer hijo (quinto proceso)
  - **CI** $= 6$
  - **NH** $= 0$
  - **CF** $= 0$

Notar que los **PID** asignados por el **SO** (si es que este es el único grupo en el input) serán $1$ para el padre, y luego $2$ para el primer hijo, $3$ para el segundo, $4$ para el hijo del segundo y $5$ para el tercero.

### 5.2. Archivo de salida (*Output*)

Dentro de cada loop se irán registrando los eventos reportados al **SO**, estos son:

- `IDLE`: Reporta el tiempo que permaneció en IDLE el **SO**.

```
IDLE <TIEMPO_IDLE>
IDLE 2
```

- `ENTER`: Reporta que entra un proceso por primera vez a la **SO**.

```
ENTER 1 0 1 TIME 7 LINE 1 ARG 1
ENTER <PID> <PPID> <GID> TIME <TIEMPO_ACTUAL_SO> LINE <NUM_LINEA> ARG <NUM_ARGUMENTOS>
```

- `RUN`: Reporta el tiempo que trabajó el proceso.

```
RUN 1 8
RUN <PID> <TIEMPO_TRABAJADO>
```

- `WAIT`: Reporta que el proceso entró a estado `WAITING`.

```
WAIT <PID>
WAIT 1
```

- `RESUME`: Reporta que un proceso vuelve a estado `READY`.

```
RESUME <PID>
RESUME 1
```

- `END`: Reporta que un proceso termina su ejecución.

  ```
  END <PID> TIME <TIEMPO_ACTUAL_SO>
  END 2 TIME 19
  ```

Al final de cada loop del **SO**, el **SO** reporta estadísticas generales de cada proceso. Estas son por orden de llegada de cada grupo al **SO**, y luego se reportan los procesos finalizados (para estos se usa **GID**$= 0$).[6]

1. Se reporta el texto REPORT START.

   ```
   REPORT START
   ```

2. Se reporta el texto TIME seguido del tiempo actual del **SO**.

   ```
   TIME <TIEMPO_ACTUAL_SO>
   TIME 20
   ```

3. Se reporta por cada grupo en orden:

   - Una línea indicando el texto GRUP, GID, número de procesos en el grupo.

     ```
     GROUP <GID> <NUM_PROGRAMAS_GRUPÖ>
     GROUP 1 2
     ```

   - Por cada proceso del grupo, según su orden de entrada al grupo, se reporta: texto PROGRAM, PID, PPID, GID, status del proceso, cantidad de CPU que ha usado.

     ```
     PROGRAM <PID> <PPID> <GID> <STATUS_PROCESO> <CPU_TOTAL_USADA>
     PROGRAM 1 0 1 WAITING 9
     ```

4. Se reporta el texto REPORT END.

   ```
   REPORT END
   ```

**Importante**: Podrá notar que, básicamente, se solicita un `TXT` donde en cada fila cada dato se presenta **con solo un único espacio por medio**.

# Formalidades

A cada alumno se le asignó (o asignará en un futuro muy cercano) un nombre de usuario y una contraseña para el servidor del curso[7]. Para entregar su tarea usted deberá crear una carpeta llamada `T0` en el directorio principal de su carpeta personal y subir su tarea a esa carpeta. En esta carpeta **solo debe incluir el código fuente** necesario para compilar su tarea y un `Makefile`. Se revisará el contenido de dicha carpeta el día Lunes 8-Abril-2024 a las 21:00.
Solo uno de los integrantes de cada grupo debe entregar en su carpeta. Las parejas los elegirán ustedes y en caso de que alguien no tenga compañero, puede intentar buscar por el foro del curso o el chat de telegram.
Antes de acabado el plazo de entrega de la tarea, se enviará un form donde podrán registrar las parejas junto con el nombre de usuario de la persona que realizará la entrega en el servidor.

- **NO debe subir su tarea a un repositorio público**. En caso contrario, tendrá como nota máxima 4.0.
- **NO debe incluir archivos binarios en su entrega**. En caso contrario, tendrá un descuento de 0.3 puntos en su nota final.
- Si inscribe de forma incorrecta su grupo, tendrá un descuento de 0.3 puntos.

[6] Dentro de cada grupo, se reporta según cuando empezaron a ejecutarse
[7] `iic2333.ing.puc.cl`

- Su tarea deberá compilar utilizando el comando `make` en la carpeta `T0`, y generar los ejecutables `runner` y `schedulesly` en esta misma. Si su programa **no tiene** un `Makefile` o el nombre de la carpeta no es el correcto, **no se corregirá** y tendrá un descuento de 0.5 puntos en la recorrección.
- Si usan el *plugin* de SSH para VSCode o uno similar, que instale software en el servidor, tendrán un descuento de 1 punto en su tarea.
- Si su programa **no compila** o **no funciona** (*segmentation fault*), obtendrán la nota mínima, pudiendo recorregir modificando líneas de código con un descuento de una décima por cada cuatro líneas modificada, con un máximo de 20 líneas a modificar.

El no respeto de las formalidades o un código extremadamente desordenado podría originar descuentos adicionales a criterio de el o la ayudante que corrija. Se recomienda modularizar, utilizar funciones y ocupar nombres de variables explicativos.

# Evaluación

Cada parte de esta tarea será evaluada con una nota entre 1 y 7. Así, la nota final será el promedio entre las 2 partes de la tarea. El puntaje de cada una de las partes seguirá la siguiente distribución:

- **6.0 pts.** Parte I: Procesos.
  - **0.8 pts.** Correcta implementacion de `wait_all`.
  - **0.5 pts.** Correcta implementacion de tiempo $\langle max \rangle$.
  - **1.5 pts.** Correcta implementación de múltiples procesos paralelos.
  - **1.5 pts.** Comunicación entre procesos por medio de señales, implementación del término por Ctrl Z.
  - **1.7 pts.** *Output* correcto.
  - **Descuento: Manejo de Memoria** Se descontará hasta **1.0 pts** si `valgrind` no reporta en su código 0 *leaks* y 0 errores de memoria en **todo caso de uso**[8].
- **6.0 pts.** Parte II: Scheduler.
  - **1.0 pt.** Tests con grupos que poseen un solo programa.
  - **2.0 pts.** Tests con grupos que poseen hasta 3 programas.
  - **3.0 pts.** Tests con grupos que poseen cantidad arbitraria de programas.
  - **Descuento: Manejo de Memoria** Se descontará hasta **1.0 pts** si `valgrind` no reporta en su código 0 *leaks* y 0 errores de memoria en **todo caso de uso**.

## Importante:

Con el objetivo de que ambas partes de la tarea sean realizadas y no se deje de lado ninguna, en caso de que alguna de las dos partes de la tarea tenga un puntaje menor a **2.5 pts.**, se asignará la nota de la tarea usando el puntaje menor ponderado por **0.8** y el puntaje mayor ponderado por **0.2**.

$$\text{Puntaje obtenido} = \text{mín}\{\text{Puntaje Parte I}, \text{Puntaje Parte II}\} \cdot 0.8 + \text{máx}\{\text{Puntaje Parte I}, \text{Puntaje Parte II}\} \cdot 0.2$$

La tarea deberá ser realizada en el lenguaje de programación **C**. Cualquier tarea escrita en otro lenguaje de programación no será revisada y será evaluada con nota 1. Además, su trabajo será revisado usando Moss, un software de detección de plagio y similaridad en el código, por lo que cualquier código que no sea de su autoría que utilicen en sus tareas debe ser adecuadamente referenciado. Esto excluye al código base entregado a los estudiantes.

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

## Preguntas

Cualquier duda preguntar a través del foro oficial.

## Política de Integridad Académica y Código de Honor de la UC

Los alumnos de la Escuela de Ingeniería de la Pontificia Universidad Católica de Chile deben mantener un comportamiento acorde al Código de Honor de la Universidad:

*"Como miembro de la comunidad de la Pontificia Universidad Católica de Chile me comprometo a respetar los principios y normativas que la rigen. Asimismo, prometo actuar con rectitud y honestidad en las relaciones con los demás integrantes de la comunidad y en la realización de todo trabajo, particularmente en aquellas actividades vinculadas a la docencia, el aprendizaje y la creación, difusión y transferencia del conocimiento. Además, velaré por la integridad de las personas y cuidaré los bienes de la Universidad."*