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

IIC2333 — Sistemas Operativos y Redes — 2/2020

**Tarea 2**

Miércoles 9-Septiembre-2020

**Fecha de Entrega: Miércoles 16-Septiembre-2020, 14:00**
**Ayudantía: Viernes 11-Septiembre-2019, 08:30**
**Composición: Tarea en parejas**

# Objetivos

- Modelar la ejecución de un algoritmo de *scheduling* de procesos.
- Analizar las diferencias de rendimiento de un mismo algoritmo de *scheduling* para distinta cantidad de CPU.

## Deadline Scheduler

Después de haber logrado el desafío que fue para ti la creación de múltiples procesos, el gran Cruz te felicita por un trabajo bien hecho. Tanto el gran Cruz como los ayudantes de Sistemas Operativos ahora son capaces de correr sus propios programas usando tu interfaz, permitiéndoles un mejor uso de los recursos del computador en una sola consola.

Sin embargo, un día te llega una terrible noticia. Los ayudantes necesitan poder ahora elegir que procesos van a correr en sus CPU, ya que intentaron correr muchos procesos distintos, los cuales poseen un *deadline*, es decir que estos procesos necesitan haber terminado en un momento específico. Por esto necesitan que crees tu propio ***scheduler***, el cual se encargará de decidir cuando le entregaremos recursos a un proceso determinado.

## Descripción

El objetivo de esta tarea es **implementar** el algoritmo *Earliest Deadline First (EDF)*. Este algoritmo se encarga de asignarle la CPU al proceso cuyo **deadline** sea el más cercano, es decir el proceso que tenga que terminar antes que todo el resto.

# Modelamiento de los procesos

Debe existir un `struct Process`, que modelará un proceso. Un proceso tiene un `PID`, un nombre de hasta 32 caracteres, una prioridad y un estado, que puede ser `RUNNING`, `READY`, `WAITING` o `FINISHED`.

La simulación recibirá un archivo de entrada que describirá los procesos[1] a ejecutar y el comportamiento de cada proceso en cuanto a uso de CPU. Una vez terminado el tiempo de ejecución de un proceso, éste terminará tal como si hubiese ejecutado `exit()`, y pasará a estado `FINISHED`. Durante su ciclo de vida el proceso alterna entre un tiempo $A_i$ en que ejecuta código de usuario (CPU *burst*, ráfaga de ejecución), y un tiempo $B_i$ en que permanece bloqueado (I/O *burst*, tiempo de espera).

[1] La cantidad de procesos es variable, sin embargo será menor siempre a `INT_MAX`

## Modelamiento de la cola de procesos

Debe construir un `struct Queue` para modelar una cola de procesos. Esta estructura deberá ser construida por usted y deberá contener, como mínimo, el **arreglo** de estructuras `Process` de la simulación. La cola debe estar preparada para recibir **cualquier** cantidad de procesos que puedan estar en estado `READY` al mismo tiempo.

Si bien se modela una cola de procesos, **no es necesario** que creen exactamente una estructura de tipo cola. Por ejemplo, no es necesario que la compongan a partir de una lista enlazada con estructuras de clase "nodo", ni que posea métodos como *enqueue* o *dequeue*, basta con que se defina la `struct Queue` y que contenga, como mínimo, el **arreglo** de todos los procesos a simular.

## *Scheduler*

El *scheduler* decide qué procesos de la cola, en estado `READY`, entrarán en estado `RUNNING` de acuerdo al deadline de cada proceso. El *scheduler* debe sacar al proceso actual de la CPU, reemplazarlo por el siguiente y determinar la acción correspondiente para el proceso saliente, las que pueden ser:

- Pasar a estado `FINISHED` al finalizar su última ráfaga de ejecución.
- Pasar a estado `WAITING` al bloquearse, *i.e.* al terminar la ráfaga de ejecución actual y tener un tiempo de espera posterior.
- Pasar a estado `READY` en la cola al ser interrumpido, lo que sucede cuando llega a la cola un proceso con un deadline mas urgente.

El *scheduler* debe asegurar que el estado de cada proceso cambie de manera consistente. Por otra parte, su ejecución deberá ser eficiente en términos de tiempo[2].

## Múltiples CPU

En los sistemas operativos modernos, el *scheduler* debe manejar mas de sólo un procesador a la hora de hacer decisiones sobre qué procesos deben poder acceder a estos múltiples recursos. En su tarea deberá manejar la simulación para una cantidad arbitraria de CPUs, es decir, para efectos de su tarea, deberá poder soportar que más de un sólo proceso se encuentre en estado `RUNNING` a la vez.

Al igual que cuando se trabaja con un procesador, el *scheduler* debe ser capaz de interrumpir a los procesos que se encuentran ejecutando en cualquier CPU para entregarle estos recursos a un proceso que los necesite más. Para el caso en que se deba sacar un proceso para poner otro, se espera que su *scheduler* sea **óptimo** e interrumpa únicamente al proceso con el *deadline* mas lejano.

En caso que más de un proceso entre a su estado READY al mismo tiempo, siempre deberán priorizar a los procesos con *deadlines* mas cercanos.

[2]Para efectos de esta evaluación, se considerarán eficientes simulaciones que tarden, a lo más, un minuto en ejecutar para cada archivo de entrada. Para ver el tiempo de ejecución de un programa en C puede usar `time`.

# Ejecución de la Simulación

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

```
./edf <file> <output> [<n>]
```

Donde:

- `<file>` corresponderá a un nombre de archivo que deberá leer como *input*.
- `<output>` corresponderá a la ruta de un archivo `CSV` con las estadísticas de la simulación, que debe ser escrito por su programa.[3]
- `<n>` argumento opcional, el cual denota la cantidad de CPUs de su programa. En caso de no ser ingresado, será por defecto 1.

Por ejemplo, algunas ejecuciones podrían ser las siguientes:

```
./edf input.txt output.csv
./edf procesos.txt salida.csv 3
```

## 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 $K$ de procesos a simular y las siguientes $K$ líneas siguen el siguiente formato:

```
NOMBRE_PROCESO PID TIEMPO_INICIO DEADLINE N A_1 B_1 A_2 B_2 ... A_N-1 B_N-1 A_N
```

Donde:

- $N$ es la cantidad de ráfagas de CPU, con $N \geq 1$.
- La secuencia $A_1\ B_1\ \ldots\ A_{N-1}\ B_{N-1}\ A_N$ describe el comportamiento de cada proceso. Cada $A_i$ es la cantidad de unidades de tiempo que dura un ráfaga de CPU (*CPU burst*), es decir, la cantidad de tiempo que el proceso solicitará estar en estado `RUNNING`. Cada $B_i$ es la cantidad de unidades de tiempo que dura una *I/O burst*, es decir, la cantidad de tiempo que el proceso se mantendrá en estado `WAITING`.
- `NOMBRE_PROCESO` debe ser respetado para la impresión de estadísticas.
- `PID` es el Process ID del proceso.
- `TIEMPO_INICIO` es el tiempo de llegada a la cola. Considere que `TIEMPO_INICIO` $\geq 0$.
- `DEADLINE` es el **deadline** del proceso, es decir el tiempo máximo que tiene el proceso para terminar todas sus ráfagas desde su llegada al sistema.
- Los tiempos son entregados sin unidades, por lo que pueden ser trabajados como enteros adimensionales.

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.
- El nombre del proceso no contendrá espacios, ni será más largo que 31 caracteres.
- Habrá al menos un proceso descrito en el archivo.

[3] El output de su programa **debe** ser el archivo ingresado, de lo contrario no se contará como correcto.

- Todas las secuencias comienzan y terminan con un *burst* $A_i$. Es decir, un proceso nunca estará en estado `WAITING` al ingresar a la cola por primera vez ni antes de terminar su ejecución.

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

```
3
PROCESS1_RAUL 21 2 120 5 10 4 30 3 40 2 50 1 10
PROCESS2_RICHI 13 1 60 3 14 3 6 3 12
PROCESS3_ZEZIMA 5 3 200 4 14 3 6 3 12 3 18
```

**Importante**: Es posible que en algún momento de la simulación no haya procesos en estado `RUNNING` o `READY` o no hayan suficientes para llenar todas las CPUs. Los sistemas operativos manejan esto creando un proceso especial de nombre `idle` que no hace nada hasta que llega alguien a la cola `READY`. Pueden considerarlo en su simulación, pero no debe influir en los valores de la estadística, es decir, no se debe incluir en el archivo de salida.

## Orden de ejecución

Podrá notar que el orden exacto en el que realice los cambios no es fundamental, no obstante, **debe tener cuidado** con que su programa realice más de una modificación dentro de una misma iteración. Por ejemplo, un proceso que termina la ejecución de una de sus ráfagas $A_i$ y pasa a estado `WAITING`, **no puede aumentar en una unidad su tiempo de espera dentro de la misma iteración**. Es importante que tenga esto en consideración para todos los eventos que puedan ocurrir.

## Salida (*Output*)

Una vez que el programa haya terminado la simulación, su programa deberá escribir un archivo con los siguientes datos por proceso:

- El nombre del proceso.
- El número de veces que el proceso fue elegido para usar alguna CPU.
- El número de veces que fue interrumpido. Este equivale al número de veces que el *scheduler* sacó al proceso de la CPU.
- El *turnaround time*.
- El *response time*.
- El *waiting time*. Este equivale a la suma del tiempo en el que el proceso está en estado `READY` y `WAITING`.
- Si el proceso terminó o no antes de su deadline, donde `1` implica que sí logró terminar y `0` si no.[4]

Es importante que siga **rigurosamente** el siguiente formato:

```
nombre_proceso_i,turnos_CPU_i,interrupciones_i,turnaround_time_i,response_time_i,waiting_time_i,deadline
nombre_proceso_j,turnos_CPU_j,interrupciones_j,turnaround_time_j,response_time_j,waiting_time_j,deadline
...
```

Podrá notar que, básicamente, se solicita un `CSV` donde las columnas son los datos pedidos por proceso, **sin espacios de por medio**.

[4] Si un proceso termina exactamente al mismo tiempo que el deadline, este **sí** cuenta como cumplido.

### Casos especiales

Dada la naturaleza de este problema, existirán algunos casos límite que podrían generar resultados distintos según la implementación. Para evitar este problema, **deberán** considerar que:

- Si al momento de escoger un proceso existen dos con el mismo deadline, entonces será seleccionado el que posea un **menor** `PID`
- Un proceso **puede** llegar en tiempo $t = 0$. Su programa no se puede caer si llega un archivo *input* con ese valor.

## Formalidades

A cada alumno se le asignó un nombre de usuario y una contraseña para el servidor del curso[5]. Para entregar su tarea usted deberá crear una carpeta llamada `T2` en el directorio principal de su carpeta personal y subir su tarea a esa carpeta. En su carpeta `T2` **solo debe incluir el código fuente** necesario para compilar su tarea y un `Makefile`. Se revisará el contenido de dicha carpeta el día Miércoles 16-Septiembre-2020, 14:00.
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 `T2`, 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

El puntaje de su programa seguirá la siguiente distribución de puntaje:

- **5.5 pts.** Simulación correcta.
  - **0.5 pts.** Estructura y representación de procesos.
  - **1.0 pts.** Estructura y manejo de colas.
  - **1.5 pts.** *Earliest Deadline First* con 1 CPU [6].
  - **2.5 pts.** *Earliest Deadline First* con múltiples CPU[7].
- **0.5 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**[8].

[5] `iic2333.ing.puc.cl`
[6] Este puntaje será evaluado según la cantidad de *tests* a utilizar.
[7] Ver nota al pie de página 3.
[8] Es decir, debe reportar 0 *leaks* y 0 errores para todo *test*.

### Política de atraso

Se puede hacer entrega de la tarea con un máximo de 4 días de atraso. La fórmula a seguir es la siguiente:

$$N_{T_2}^{\text{Atraso}} = \text{mín}(N_{T_2}, 7{,}0 - 0{,}75 \cdot d - D)$$

Siendo $d$ la cantidad de días de atraso y $D$ los descuentos obtenidos. Notar que esto equivale a un descuento *soft*, es decir, cambia la nota máxima alcanzable y no se realiza un descuento directo sobre la nota obtenida.

## Preguntas

Cualquier duda preguntar a través del foro oficial.