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

IIC2333 — Sistemas Operativos y Redes — 1/2025

Tarea 1 - Scheduling

Viernes 5 de Septiembre 2025

Fecha de Entrega: Viernes 26 de Septiembre a las 21:00 hrs.
Composición: Tarea en parejas

Objetivos

- Modelar un scheduler de procesos utilizando `structs`.
- Simular la ejecución de dicho scheduler sobre un grupo de procesos.

DCCambios

Tras haber resuelto el desafío de gestionar las misiones, la capitana se da cuenta de lo ineficiente que es el orden de ejecución de cada una de ellas. Gastaron demasiado tiempo y recursos para llevar a cabo una investigación rutinaria mientras la nave estaba a punto de estrellarse contra un grupo de asteroides. Por lo tanto, surge la necesidad de garantizar que todas las misiones sean atendidas de manera eficiente.
Para abordar este problema, el piloto sugiere una solución: *DCCambios*, un programa que distribuye los procesos a medida que llegan, asegurando que todos puedan ser atendidos y finalicen sus actividades. Para ello, se propone implementar un scheduler de tipo MLFQ (MultiLevel Feedback Queue) que, eficientemente, atiende de manera óptima y efectiva a los procesos.

Descripción

El *scheduling* consiste en elegir el siguiente proceso a ejecutar en la CPU. Dado que no siempre se tiene información completa sobre los procesos, existen diferentes algoritmos para intentar tomar la mejor decisión. Uno de ellos es el MLFQ, que deberán implementar.

Un MLFQ es un scheduler en que se utilizan varias colas con diferentes prioridades, a modo de reducir la probabilidad de inanición de un proceso. En esta ocasión ustedes deberán modelar un scheduler MLFQ con dos colas, una de alta prioridad y otra de baja prioridad.

Modelamiento Propuesto para Procesos

Debe construir un `struct Process`, que modelará un proceso. Cada proceso tiene al menos las siguientes variables asociadas:

1. Nombre.
2. PID.
3. Estado $\in$ {`RUNNING, READY, WAITING, FINISHED, DEAD`}.
4. Tiempo de ejecución por ráfaga (burst).

5. Número de ráfagas de ejecución en CPU.
6. Tiempo de espera para I/O entre ráfagas.
7. Deadline de ejecución (instante de tiempo absoluto en el que el proceso debe haber finalizado la totalidad de sus *bursts*).

Modelamiento Propuesto para Colas de Procesos

También debe existir un `struct Queue`, el cual modelará una cola de procesos. Esta estructura debe estar diseñada por ustedes y debe ser capaz de recibir un número arbitrario de procesos en estado READY de manera simultánea.

Cada cola tiene un *quantum* asociado:
En el caso de la cola *High*, el *quantum* se calcula según la fórmula $quantum = 2 * q$, y para la cola *Low* según $quantum = q$, donde q es un parámetro que recibe el programa.

Estados

El *scheduler* puede contener procesos en las colas. Los procesos en una cola pueden estar en dos estados posibles:

- `READY`, indica que el proceso está listo para ejecutar.
- `WAITING`, indica que el proceso está esperando y no puede ejecutar.

En caso de que un proceso no se encuentre dentro de una cola, sus estados pueden ser:

- `RUNNING`, indica que el proceso está ejecutando dentro de la CPU. Sólo puede haber un proceso en este estado a la vez.
- `FINISHED`, indica que el proceso ya ha terminado su ejecución.
- `DEAD`, indica que el proceso no pudo terminar su ejecución (bursts) en su deadline.

El scheduler debe asegurar que el estado de cada proceso cambie de manera consistente.

DCCambios

Su programa debe implementar el *scheduler* MFLQ de dos colas, *High* y *Low*, de acuerdo a las siguientes reglas:

- Los procesos de la cola *High* siempre tienen prioridad sobre la cola *Low*.
- Todo proceso que ingresa al *scheduler* por primera vez, ingresa a la cola *High* con estado `READY`.
- Dentro de una misma cola, los procesos se ordenan según el valor de *Prioridad*, la cual se calcula con la siguiente fórmula:

$$\textit{Prioridad} = \frac{1}{T_{until\_deadline}} + \textit{n° bursts restantes}$$

  - $T_{until\_deadline} = T_{deadline} - T_{actual}$.
  - *n°bursts restantes* = *n° bursts* − *n° bursts completados*.
  - En caso de empate, tiene mayor prioridad el proceso con menor *PID*.
- Si el proceso cede la CPU, este se mantiene en la misma cola en la que se encontraba, y el $quantum$ **no** se reinicia.

- Si un proceso consume todo su *quantum*, este pasa a la cola Low. Si ya se encontraba en esta cola, esta se mantiene.
- Los procesos en estado `WAITING` **no** son elegibles para ejecutar.
- Si para un proceso en la cola *Low* se cumple que, $2 * T_{deadline} < T_{actual} - T_{LCPU}$[1] , este proceso sube a la cola High.

Eventos

El scheduler tendrá la capacidad de recibir eventos. Los eventos serán utilizados para ingresar inmediatamente a CPU un proceso con cierto PID. Cuando exista un evento, se deben seguir las siguientes reglas:

- Si existe un proceso en CPU distinto al indicado por el evento, entonces debe ser interrumpido.
- Si el proceso en la CPU es el mismo que el indicado por el evento, entonces no debe ser interrumpido.
- El proceso indicado por el evento debe ingresar a la CPU o continuar su ejecución, sin importar si se encuentra en estado `WAITING`, `READY` o `RUNNING`.
- El proceso interrumpido pasa a la cola High con máxima prioridad, ignorando la fórmula de prioridad hasta que logre ingresar nuevamente a ejecutar.

Flujo del *scheduler*

Por cada *tick*, el *scheduler* debe realizar las siguientes tareas, en el orden indicado:

1. Actualizar los procesos que hayan terminado su tiempo de espera de I/O de `WAITING` a `READY`.
2. Actualizar los procesos en las colas que hayan cumplido su deadline pero no completaron sus bursts a estado DEAD.
3. Si hay un proceso en estado `RUNNING`, actualizar su estado según el siguiente orden:
   - 3.1) Alcanzó su deadline
   - 3.2) Termninó su CPU *burst*
   - 3.3) Su *quantum* se acabó
   - 3.4) Si ocurre un evento que involucra a un proceso distinto al que está en la CPU, entonces el proceso en ejecución debe abandonar la CPU
   - 3.5) Continua ejecutando con normalidad
4. Ingresar los procesos a las colas según corresponda:
   - 4.1) Si un proceso salió de la CPU, ingresarlo a la cola que corresponda.
   - 4.2) Para cada proceso, si el tiempo de inicio se cumple, ingresarlo a la cola *High*.
   - 4.3) Para cada proceso de la cola Low, revisar si se cumple la condición para subir a la cola High y cambiarlos de cola según corresponda.
5. Actualizar las prioridades de todos los procesos según la fórmula dada.
6. Ingresar proceso a la CPU siguiendo este orden de prioridad:
   - 6.1) Si se cumplió el tiempo de un evento, ingresar el proceso indicado.
   - 6.2) Primer proceso en estado READY de la cola High.
   - 6.3) Primer proceso en estado READY de la cola Low.

[1] Consideren T_{LCPU} como el tiempo *(tick)* en que el proceso salió por última vez de la CPU.

Consideraciones especiales

- Si el proceso termina su ráfaga de CPU al mismo momento que se acaba su *quantum*, se considera que el proceso cedio la CPU, por tanto no baja de cola.
- Si el proceso termina su ejecución al mismo momento que se acaba su *quantum*, este pasa a estado `FINISHED`.

Ejecución de la simulación

La simulación se debe ejecutar con el siguiente comando:

```
./DCCambios <input_file> <output_file>
```

Donde `input_file` es el nombre del archivo de *input*, `output_file` corresponde a la ruta a un archivo CSV con las estadísticas que debe guardar el programa, Una posible ejecución sería `./DCCambios processos.txt salida.csv`.

1. Archivo de entrada (*input*)

Este archivo de entrada contiene los datos de la simulación, donde la primera línea es el *q* para el *quantum* asociado a cada cola, la segunda línea contiene un entero K y la tercera línea contiene un entero N donde K indica la cantidad de procesos y N la cantidad de eventos.
Las siguientes K líneas tienen el siguiente formato:

```
NOMBRE_PROCESO PID T_INICIO T_CPU_BURST N_BURSTS IO_WAIT T_DEADLINE
```

Donde:

- `NOMBRE_PROCESO` es el nombre de dicho proceso.
- `PID` es el Process ID del proceso.
- `T_INICIO` es el tiempo en el que el proceso entra a la cola por primera vez.
- `T_CPU_BURST` es el tiempo que dura una ráfaga de ejecución en la CPU.
- `N_BURSTS` es la cantidad de ráfagas de ejecución en la CPU que tiene el proceso.
- `IO_WAIT` es el tiempo de espera entre ráfagas de ejecución.
- `T_DEADLINE` es el tiempo **absoluto** del deadline de ejecución.

Y luego de los procesos, las siguientes N líneas tienen el siguiente formato:

```
PID T_EVENTO
```

Donde:

- `PID` corresponde al *PID* de un proceso ya existente que interrumpe e ingresa forzosamente a la CPU.
- `T_EVENTO` es el tick en el cual ocurre este evento.

Finalmente:

- Puede suponer que no habrá tiempos negativos
- Puede suponer que para cada proceso `T_DEADLINE` > `T_INICIO`
- También pueden asumir que `T_DEADLINE` > 0 y `IO_WAIT` ≥ 0. Todos los tiempos se entregan sin dimensión y caben en un entero sin signo de 32 bits.
- El orden de los eventos no necesariamente están en orden de tiempo a ejecutar.
- No se entregarán archivos vacíos.

Archivo de salida (*output*)

En su archivo de salida, debe incluir una fila con las siguientes estadísticas para cada proceso:

- Nombre del proceso
- PID
- Estado del proceso
- Número de interrupciones
- El turnaround time
- El response time, considerando el tiempo que tardó en entrar por primera vez a la CPU.
- El waiting time, este es la suma de todos los intervalos en que estuvo en estado WAITING o READY para ejecutar.

El orden en el que se imprimen los procesos es por su id, en orden ascendente. Por lo tando, el archivo de salida se veria como sigue:

```
nombre_proceso_a,pid_a,estado_a,interrupciones_a,turnaround_a,response_a,waiting_a
nombre_proceso_b,pid_b,estado_b,interrupciones_b,turnaround_b,response_b,waiting_b
nombre_proceso_c,pid_c,estado_c,interrupciones_c,turnaround_c,response_c,waiting_c
```

Es importante que **respeten el formato del output.**

Ejemplo Input y Output

A continuación, se les presenta un input y output válidos.
Input:

```
7
4
1
PROCESS_0 0 0 1 2 1 26
PROCESS_1 1 0 1 2 1 26
PROCESS_2 2 0 2 2 1 24
PROCESS_3 3 0 2 2 1 24
3 15
```

Output:

```
PROCESS_0,0,FINISHED,0,14,7,11
PROCESS_1,1,FINISHED,0,12,9,9
PROCESS_2,2,FINISHED,0,20,1,15
PROCESS_3,3,FINISHED,0,17,4,12
```

Formalidades

A cada alumno se le asignó un nombre de usuario y una contraseña para el servidor del curso[2]. Para entregar su tarea usted deberá crear una carpeta llamada exactamente `T1`[3] 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 es necesario para compilar su tarea y un `Makefile`. Se revisará el contenido de dicha carpeta el día Viernes 26 de Septiembre a las 21:00 hrs..

- La tarea debe ser realizada solamente en parejas.
- La tarea deberá ser realizada en el lenguaje de programación **C**. Cualquier tarea escrita en otro lenguaje de programación no será revisada.
- **NO debe incluir archivos binarios**[4]. En caso contrario, tendrá un descuento de 0.2 décimas en su nota final.
- **NO debe incluir un repositorio de git**[5]. En caso contrario, tendrá un descuento de 0.2 décimas en su nota final.
- **NO debe usar VSCode para entrar al servidor**[6]. En caso contrario, tendrá un descuento de 0,2 puntos en su nota final.
- Si inscribe de forma incorrecta su grupo o no lo inscribe, tendrá un descuento de 0.3 décimas
- Su tarea debe compilarse utilizando el comando `make`, y generar un ejecutable llamado `DCCambios` misma carpeta. Si su programa no tiene un `Makefile`, tendrá un descuento de 0.5 décimas en su nota final y corre riesgo que su tarea no sea corregida.
- En caso de no cumplir con el formato de entrega, tendrá un descuento de 0.3 décimas en la nota final.
- Si ésta 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, los cuales quedarán a discreción del ayudante corrector. 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.

Importante: En caso de no cumplir con el formato de entrega, tendrá un descuento de 0.3 décimas en la nota final.

Evaluación

- **1.5 pts.** Pasa correctamente los Tests *Easy*.
- **1.5 pts.** Pasa correctamente los Tests *Medium*.
- **2.5 pts.** Pasa correctamente los Tests *Hard*.
- **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[7].

---

2 `iic2333.ing.puc.cl`
3 Se debe respetar el uso de mayúsculas, sino, la tarea no sera recolectada.
4 Los archivos que resulten del proceso de compilar, como lo son los ejecutables y los object files
5 Si es que lo hace, puede eliminar la carpeta oculta `.git` antes de la fecha de entrega.
6 Si es que lo hace, puede eliminar la carpeta oculta `.vscode-server` antes de la fecha de entrega.
7 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 2 días hábiles de atraso. La fórmula a seguir es la siguiente:

$$N_{T_1}^{\text{Atraso}} = \text{mín}(N_{T_1}, 7{,}0 - 0{,}75 \cdot d)$$

Siendo d la cantidad de días de atraso. 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. El uso de días de atraso no implica días extras para alguna tarea futura, por lo que deben usarse bajo su propio riesgo.

Preguntas

Cualquier duda preguntar a través del foro oficial.