Implementación de un Planificador Round Robin Ponderado en C

Estructuras de Datos.

Dpto. Lenguajes y Ciencias de la Computación. Universidad de Málaga

---

Objetivo: En este ejercicio, implementarás una estructura de datos en C para representar un Planificador Round Robin Ponderado para tareas. La estructura será una lista circular enlazada simple, y mantendrás un puntero al último elemento en la cola. Este ejercicio te ayudará a entender listas enlazadas, estructuras de datos circulares y algoritmos de planificación, que son conceptos fundamentales en la informática.

Lista Circular Enlazada Simple Una **lista circular enlazada simple** es un tipo de lista enlazada donde cada nodo apunta al siguiente nodo en la secuencia, y el último nodo apunta de nuevo al primer nodo, formando un círculo. Aquí tienes un desglose de sus componentes y características:

1. **Estructura del Nodo**: Cada nodo contiene datos y un puntero al siguiente nodo.
2. **Naturaleza Circular**: El puntero del último nodo apunta al primer nodo, creando una estructura circular.
3. **Recorrido**: Puedes recorrer la lista comenzando desde cualquier nodo y eventualmente regresar al nodo donde comenzaste.

Ejemplo: Imagina una lista con tres nodos que contienen los valores 1, 2 y 3:

- El Nodo 1 apunta al Nodo 2.
- El Nodo 2 apunta al Nodo 3.
- El Nodo 3 apunta de nuevo al Nodo 1.

Funcionamiento del Planificador Round Robin Ponderado

El **Planificador Round Robin Ponderado (WRR)** es un algoritmo de planificación de tiempo compartido utilizado para ejecutar una serie de tareas donde a cada tarea se le asigna un quantum de tiempo diferente basado en su peso. Así es como funciona:

1. **Inicialización**: Cada tarea se añade a una cola (implementada con una lista circular enlazada simple) con un peso asociado (quantum de tiempo).
2. **Ejecución**:
   - El planificador selecciona la primera tarea en la cola y le permite ejecutarse durante su quantum de tiempo asignado.
   - Después de que el quantum de tiempo expira, la tarea se mueve al final de la cola.
   - La siguiente tarea en la cola es seleccionada y ejecutada durante su quantum de tiempo.
3. **Repetir**: Este proceso continúa, asegurando que cada tarea reciba tiempo de CPU proporcional a su peso.

Significado de Quanta

En el contexto de la planificación, un **quantum** (plural: quanta) es la cantidad fija de tiempo que se permite a una tarea ejecutarse antes de que el planificador pase a la siguiente tarea. En un planificador WRR, cada tarea puede tener un quantum diferente basado en su peso.

Ejemplo Simple

Consideremos un ejemplo simple con tres tareas: T1, T2 y T3, con pesos 2, 1 y 3, respectivamente.

1. **Cola Inicial**: [T1 (2), T2 (1), T3 (3)]
2. **Ejecución**:
   - T1 se ejecuta durante 2 unidades, luego se mueve al final: [T2 (1), T3 (3), T1 (2)]
   - T2 se ejecuta durante 1 unidad, luego se mueve al final: [T3 (3), T1 (2), T2 (1)]
   - T3 se ejecuta durante 3 unidades, luego se mueve al final: [T1 (2), T2 (1), T3 (3)]
3. **Continuar**: Este proceso continúa, con cada tarea recibiendo tiempo de CPU basado en su peso.

Ejercicio: Implementación de un Planificador Round Robin Ponderado en C

Escenario: Imagina que eres un ingeniero de software en una empresa tecnológica. Tu equipo está desarrollando un nuevo sistema operativo, y te han encargado implementar el planificador de tareas. El planificador necesita gestionar múltiples tareas y asegurar que cada tarea reciba tiempo de CPU proporcional a su importancia. Para lograr esto, utilizarás un algoritmo de planificación Round Robin Ponderado.

Estructura de Datos: Implementarás el planificador utilizando una lista circular enlazada simple. Se mantendrá un puntero al último elemento en la cola para facilitar la inserción y eliminación de tareas.

Estructura de la Tarea: Cada tarea debe tener los siguientes atributos:

- **ID**: Un identificador único sin signo para la tarea.
- **Nombre**: El nombre de la tarea.
- **Peso**: El quantum de tiempo asignado a la tarea.

Funciones del Planificador:

1. **Inicializar Planificador**: Crear un planificador vacío.
2. **Añadir Tarea (Enqueue)**: Añadir una nueva tarea al final de la cola de tareas.
3. **Obtener Tarea (First)**: Devolver, sin eliminar, la primera tarea en la cola.
4. **Eliminar Tarea (Dequeue)**: Eliminar la primera tarea en la cola del planificador.
5. **Tamaño del Planificador**: Devolver el número de tareas en el planificador.
6. **Limpiar Planificador**: Eliminar todas las tareas del planificador.
7. **Mostrar Tareas**: Imprimir todas las tareas en el planificador.

Implementación: La estructura de la tarea puede ser implementada en un módulo separado. El encabezado del módulo (Task.h) sería el siguiente:

```
#ifndef TASK_H // Inclusión condicional
#define TASK_H // Evita múltiples inclusiones

#include <stddef.h>

#define MAX_NAME_LEN 20 // Longitud máxima del nombre de la tarea

struct Task {
  unsigned int id;              // Identificador único de la tarea
  char name[1 + MAX_NAME_LEN];  // Cadena con el nombre de la tarea
  int quantum;                  // Quantum de tiempo asignado a la tarea
};

struct Task* Task_new(unsigned int id, const char name[], int quantum);
void Task_free(struct Task** p_p_task);
struct Task* Task_copyOf(const struct Task* p_task);
void Task_print(const struct Task* p_task);
#endif
```

Tendrás que implementar las diferentes funciones en este módulo. El propósito de cada una de ellas es el siguiente:

- La función `Task_new` crea una nueva tarea con el ID, nombre y quantum dados. Asigna memoria para la estructura de la tarea en el heap e inicializa la tarea con los valores proporcionados. Devuelve un puntero a la nueva tarea. Esta función debe verificar que el nombre no sea más largo que `MAX_NAME_LEN` caracteres, ya que de lo contrario desbordaría el tamaño del array que contiene el nombre.
- La función `Task_free` toma un puntero a un puntero a una tarea previamente asignada en el heap y libera su memoria. También establece el puntero a la tarea en NULL para evitar punteros colgantes.
- La función `Task_copyOf` crea una copia de la tarea proporcionada. Asigna memoria para una nueva estructura de tarea y la inicializa con los mismos valores que la tarea proporcionada. La función devuelve un puntero a la nueva tarea.

- La función `Task_print` imprime el ID, nombre y quantum de la tarea proporcionada en la salida estándar.

Como se explicó anteriormente, la cola de tareas se implementará utilizando una lista circular enlazada simple. Cada nodo en la lista será una estructura que incluye una tarea y un puntero al siguiente nodo. El último nodo apuntará de nuevo al primer nodo, creando una estructura circular. Un planificador será representado por un puntero al **último** nodo en la lista circular.

El archivo de encabezado (Scheduler.h) que representa el planificador sería el siguiente:

```
#ifndef SCHEDULER_H
#define SCHEDULER_H

#include "Task.h"

#include <stddef.h>

struct Node {
  struct Task task;    // Tarea a ejecutar
  struct Node* p_next; // Puntero al siguiente nodo en la lista circular
};

// Inicializar un planificador vacío
struct Node* Scheduler_new();
// Devolver el número de tareas en el planificador
size_t Scheduler_size(const struct Node* p_last);
// Eliminar todas las tareas del planificador dejándolo vacío
void Scheduler_clear(struct Node** p_p_last);
// Devolver una copia asignada en el heap de la primera tarea en el planificador
struct Task* Scheduler_first(const struct Node* p_last);
// Añadir una copia de la tarea al final de la cola
void Scheduler_enqueue(struct Node** p_p_last, const struct Task* p_task);
// Eliminar la primera tarea del planificador
void Scheduler_dequeue(struct Node** p_p_last);
// Imprimir todas las tareas en el planificador
void Scheduler_print(const struct Node* p_last);
#endif
```

Un planificador vacío está representado por un puntero `NULL` a un nodo. Un planificador no vacío está representado por un puntero al último nodo en la lista circular.

La siguiente figura ilustra la estructura de un Planificador que incluye tres nodos, siendo el primero el que contiene una tarea cuyo ID es 1:

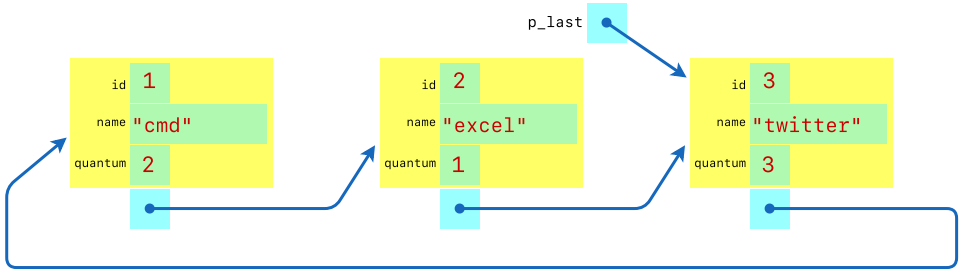

Figure 1: Queue of Tasks as a Circular Linked List

Las cajas en cian representan punteros a nodos y las de amarillo representan tareas. Como se puede ver, cada nodo contiene una tarea y un puntero al siguiente nodo. El último nodo apunta de nuevo al primer nodo, creando una estructura circular. El puntero `p_last` siempre apunta al último nodo en el planificador y es el que se pasa como argumento en todas las funciones del planificador. Cada tarea tiene un ID, una cadena con el nombre de la tarea y un quantum (quantum de tiempo asignado a la tarea).

Tendrás que implementar las diferentes funciones en este módulo. Nota que `Scheduler_enqueue` debe asignar un nuevo nodo en el heap, almacenar una copia de la tarea recibida en el nodo y enlazar el nuevo nodo al final de la lista. El puntero `p_last` apuntado por el parámetro `p_p_last` debe actualizarse para apuntar al nodo recién añadido, ya que este es ahora el último nodo en la lista circular. De manera similar, `Scheduler_first` debe asignar memoria en el heap para una copia de la primera tarea en la cola y devolver un puntero a ella. Las funciones que eliminan nodos de la estructura (`Scheduler_dequeue` y `Scheduler_clear`) deben liberar la memoria correspondiente al nodo que se está eliminando y actualizar el puntero `p_last` apuntado por `p_p_last` si es necesario.

Consejos:

- Asegúrate de manejar casos extremos, como eliminar el último nodo o añadir un nodo a un planificador vacío.
- Prueba tus funciones a fondo para asegurarte de que funcionan como se espera.
- Verifica posibles punteros `NULL` y fugas de memoria en tu implementación.
- Escribe un programa principal simple para probar las funciones de tu planificador.