Memoria PL2 EEDD

Alberto José Castillo Toma - 09126517W

Roberto Seco Volkava - 09854422A

Repositorio de GitHub: Repositorio PL2

TAD's creados, y sus métodos

TAD Pila

Archivos: *Pila.h* y *NodoPila.h*

- **Métodos principales:**
  - `void push(Proceso p);`
    Añade un proceso a la cima de la pila.
  - `Proceso pop();`
    Elimina y devuelve el proceso que está en la cima de la pila.
  - `Proceso top();`
    Devuelve el proceso que está en la cima de la pila sin eliminarlo.
- **Métodos de consulta:**
  - `bool isEmpty();`
    Indica si la pila está vacía.
  - `int length();`
    Retorna el número de elementos en la pila.
  - `bool isOrder();`
    Comprueba si los elementos de la pila están ordenados.
- **Métodos de manipulación:**
  - `void reverse();`
    Invierte el orden de los elementos de la pila.
  - `void popLast();`
    Elimina el último proceso de la pila.
  - `void concat(Pila sec);`
    Une la pila actual con otra pila dada.
  - `void sortTTL();`
    Ordena la pila según el atributo `ttl` de los procesos.
  - `void clear();`
    Vacía completamente la pila.
- **Métodos auxiliares:**
  - `int getAllTTL();`
    Suma y devuelve el total de los valores `ttl` de los procesos en la pila.
  - `void toString();`
    Convierte la pila en una representación de texto para mostrar la cima.
  - `void showAll();`
    Muestra todos los elementos de la pila como texto.

TAD Cola

Archivos: *Cola.h* y *NodoCola.h*

- **Métodos principales:**
  - `void push(Proceso p);`
    Añade un proceso al final de la cola.
  - `Proceso pop();`
    Elimina y devuelve el primer proceso de la cola.
- **Métodos de consulta:**
  - `Proceso first();`
    Devuelve el primer proceso de la cola sin eliminarlo.
  - `Proceso last();`
    Devuelve el último proceso de la cola sin eliminarlo.
  - `bool isEmpty();`
    Indica si la cola está vacía.
  - `int getLength();`
    Devuelve el número de elementos en la cola.
- **Métodos de manipulación:**
  - `void reverse();`
    Invierte el orden de los elementos de la cola.
  - `void sort();`
    Ordena la cola en función de algún criterio definido.
- **Métodos auxiliares:**
  - `void showQueue();`
    Convierte la cola en una representación de texto para mostrar todos sus elementos.

TAD ABB (Árbol Binario de Búsqueda)

Archivos: *Arbol.h* y *NodoArbol.h*

- **Métodos principales:**
  - `void append(Proceso p, parbol a);`
    Añade un proceso al árbol según el criterio de orden definido.
  - `parbol izq();`
    Devuelve el subárbol izquierdo del nodo actual.

    - `parbol dch();`
      Devuelve el subárbol derecho del nodo actual.
    - `parbol getRoot();`
      Devuelve la raíz del árbol.
- **Métodos de consulta:**
    - `bool isEmpty();`
      Indica si el árbol está vacío.
    - `int getSize();`
      Devuelve el tamaño (altura) del árbol.
    - `int getMin();`
      Devuelve el proceso con la mínima prioridad en el árbol.
    - `int getMax();`
      Devuelve el proceso con la máxima prioridad en el árbol.
    - `bool existsNode(parbol a, int p);`
      Comprueba si existe un nodo con la prioridad p en el árbol.
- **Métodos de manipulación:**
    - `ListaProc getProcsByPriority(parbol a, int p);`
      Devuelve una lista con todos los procesos que tienen la prioridad p.
    - `int getNumProcsByPriority(parbol a, int p);`
      Devuelve el número de procesos con la prioridad p.
    - `void addProccessToList(parbol a, Proceso p);`
      Añade un proceso a una lista a partir de los elementos del árbol.
- **Métodos de representación:**
    - `void toString(parbol a);`
      Convierte el árbol en una representación textual.
    - `void verInorden(parbol a);`
      Muestra los nodos del árbol en orden.
- **Estadísticas:**
    - `void showPriorities(parbol a);`
      Muestra las prioridades de los nodos del árbol.
    - `double getAverageTimeByPriority(parbol a, int priority);`
      Calcula el tiempo promedio de ejecución de procesos con una prioridad específica.
    - `void printAvgExecutionTimeAllPriorities(parbol a);`
      Calcula y muestra el tiempo promedio de ejecución para todas las prioridades.

TAD ListaCore

Estructura para manejar núcleos del sistema.

- **Métodos principales:**
  - `void append(Core c);`
    Añade un núcleo al final de la lista.
  - `void appendFront(Core c);`
    Añade un núcleo al principio de la lista.
  - `void appendIndex(Core c, int idx);`
    Añade un núcleo en la posición indicada.
  - `void popLast();`
    Elimina el último núcleo de la lista.
  - `void popFront();`
    Elimina el primer núcleo de la lista.
  - `void popIndex(int idx);`
    Elimina un núcleo en la posición indicada.
- **Métodos de consulta y representación:**
  - `Core getFirst();`
    Devuelve el primer núcleo de la lista.
  - `Core getLast();`
    Devuelve el último núcleo de la lista.
  - `Core getIndex(int idx);`
    Devuelve el núcleo en la posición indicada.
  - `void toString();`
    Muestra todos los núcleos como texto.

TAD ListaProc

Estructura similar a `ListaCore`, pero para manejar procesos.

Clases creadas para manejar los TAD's

Clase Proceso

Define las características y atributos de un proceso.
Atributos:
`PID`, `PPID`, `startTime`, `TTL`, `priority`, `core`, `execTime`.

- **Métodos:**
  - `void decrementLifeTime(int quantity);`
    Reduce el tiempo de vida del proceso en `quantity`.
  - `int getPriority();`
    Devuelve la prioridad del proceso.
  - `void toString();`
    Devuelve una representación textual del proceso.
  - `double getExecuteTime();`
    Retorna el tiempo de ejecución del proceso.
  - `void setExecuteTime(int tiempo);`
    Establece el tiempo de ejecución del proceso.

Clase Core

Representa un núcleo del sistema.

- **Método principal:**
  - `void toString();`
    Muestra una representación textual del núcleo.

Clase Scheduler

Planificador dedicado a la gestión de procesos y núcleos en la primera parte de la práctica.

Métodos:

- `void addProcessToQueue(int sys_clk);`
  Agrega un proceso a la cola en función del reloj del sistema.
- `void addProcessToCore(int time);`
  Asigna un proceso a un núcleo disponible en el momento indicado.

- `void freeCore(int core, int time);`
  Libera el núcleo especificado tras completar un proceso.
- `void addProcess(Proceso p);`
  Agrega un nuevo proceso al sistema.
- `void showProcesos();`
  Muestra todos los procesos almacenados en el planificador.
- `void showQueue();`
  Muestra el contenido actual de la cola de procesos.
- `void printCores();`
  Imprime la lista de núcleos y su estado actual.
- `void check(int time);`
  Verifica el estado de los procesos y núcleos en un instante dado.
- `void sortP();`
  Ordena los procesos según su prioridad u otro criterio.
- `void init(int clk);`
  Inicializa el planificador con el valor inicial del reloj del sistema.
- `bool allProcessesCompleted();`
  Indica si todos los procesos han sido completados.
- `void clearProcesses();`
  Elimina todos los procesos gestionados.
- `Pila getProcesos();`
  Devuelve la pila de procesos gestionados.
- `vector<int> getTiempos();`
  Devuelve los tiempos asociados a los procesos.
- `void setNumeroProcesos(int n);`
  Establece el número total de procesos a gestionar.
- `int getNumeroProcesos();`
  Devuelve el número total de procesos configurados.

Clase ELScheduler

Planificador extendido para la gestión avanzada de procesos, núcleos y estructuras de datos en las siguientes prácticas.

Métodos:

- `void init(int clk);`
  Inicializa el planificador con el valor del reloj del sistema.

- `void addProcessToStack(Proceso p);`
  Añade un proceso a la pila de procesos.
- `void addProcessToQueue(int time);`
  Inserta un proceso en la cola en función del tiempo especificado.
- `void addProcessToCore(int time);`
  Asigna un proceso a un núcleo disponible.
- `void addCore(int time);`
  Añade un nuevo núcleo al sistema en un instante determinado.
- `void popCore(int idx);`
  Elimina el núcleo especificado por índice.
- `void freeCore(int core, int time);`
  Libera el núcleo indicado tras completar el proceso asignado.
- `void check(int time);`
  Comprueba el estado de los procesos y núcleos en un momento dado.
- `void addProcessToABB(Proceso p);`
  Inserta un proceso en el árbol binario de búsqueda (ABB).
- `void sortStack();`
  Ordena la pila de procesos según el TTL u otro criterio.
- `void toString();`
  Genera una representación textual del estado actual del planificador.
- `bool allProcessesCompleted();`
  Indica si todos los procesos han finalizado.
- `int getTotalCores();`
  Devuelve el número total de núcleos disponibles.
- `ListaCores getCores();`
  Devuelve la lista de núcleos gestionados.
- `void printLeastOccupiedCores();`
  Muestra los núcleos con menor carga de trabajo.
- `void printMostOccupiedCores();`
  Muestra los núcleos con mayor carga de trabajo.
- `void printTree();`
  Imprime una representación del árbol binario de búsqueda (ABB).
- `void printPriorityList(int p);`
  Imprime la lista de procesos con la prioridad especificada.
- `int getMinLoad();`
  Devuelve la carga mínima entre los núcleos.
- `int getMaxLoad();`
  Devuelve la carga máxima entre los núcleos.

- `void showProcesos();`
  Muestra los procesos gestionados por el planificador.
- `void showCores();`
  Muestra el estado actual de los núcleos.
- `vector<int> getTiempos();`
  Devuelve los tiempos de ejecución registrados.
- `void showPriorities();`
  Muestra todas las prioridades existentes en el sistema.
- `void printAvgExecutionTimeByPriority(int priority);`
  Imprime el tiempo promedio de ejecución para procesos con la prioridad indicada.
- `void printAvgExecutionTimeAllPriorities();`
  Imprime el tiempo promedio de ejecución para todas las prioridades.

Además, hemos creado dos bibliotecas llamadas SYSTEM33.h y SYSWOW.h, para no tener que incluir 1000 #includes dentro de cada main.

Problemas Encontrados Durante el Desarrollo del Proyecto

Inicialmente, planteamos la idea de que los núcleos del sistema fueran procesos en ejecución. Sin embargo, al intentar escalar esta solución para implementar la **Práctica Laboratorio 2 (PL2)**, nos dimos cuenta de que no era viable. Por ello, optamos por una nueva estructura basada en dos clases principales: **ListaCore** y **Core**.

La mayoría de las dificultades que enfrentamos surgieron en el manejo de punteros y en problemas relacionados con recursividad infinita, que provocaban errores de tipo **SIGSEGV** (segfault). Al no contar con información explícita sobre estos errores, recurrimos a herramientas como **`cout`** para imprimir valores intermedios y al uso del depurador **gdb**. Este enfoque resultó particularmente útil para depurar la implementación de la función **`append()`** en los árboles.

En cuanto al diseño de las clases, para gestionar las dependencias entre aquellas que denominamos "privadas" (es decir, no accesibles directamente por el usuario), decidimos emplear la directiva **`friend class`**, con el objetivo de simplificar la interacción interna entre ellas. Sin embargo, para garantizar la encapsulación y evitar que el usuario manipule directamente las estructuras internas como la pila, implementamos una interfaz en el **Scheduler**. Esta interfaz consiste en métodos accesibles para el usuario que, a su vez, interactúan con las funciones internas de las estructuras. Este enfoque mejora la seguridad y reduce riesgos de manipulación no deseada.

Descripción del Comportamiento del Programa

El programa se inicia mostrando un menú interactivo que presenta las diferentes opciones disponibles para el usuario.

A partir de la selección de una opción (mediante un número), un **switch** decide cuál de las funciones será ejecutada. Estas funciones definidas en el archivo **`main.cpp`** invocan métodos del **Scheduler**, lo que refuerza la seguridad al evitar accesos directos a las estructuras internas.

El **Scheduler** incluye una variable global denominada **`SYS_CLK`**, que simula el reloj del dispositivo. Esto permite realizar cálculos precisos sobre los tiempos de ejecución de los procesos. Una vez completada la ejecución de todos los procesos en cola, el programa no finaliza automáticamente; en su lugar, el menú principal se vuelve a mostrar. Esto brinda al usuario la posibilidad de ejecutar funcionalidades adicionales, como obtener información detallada sobre cómo se llevaron a cabo las ejecuciones previas.

Este diseño asegura un flujo continuo de interacción y refuerza la usabilidad del sistema, manteniendo al mismo tiempo la seguridad de las estructuras internas.