SISTEMAS OPERATIVOS

Ingeniería Informática

Práctica 3

Noel Martínez Pomares

GESTIÓN DE MEMORIA

```cpp
int main(int argc, char *argv[]){
    unsigned int cantMemoria = 2000;
    string entrada = "",salida = "";
    char seg, algoritmoUsado;

    bool correcto = checkArgumentos(argc, argv, entrada, salida);

    if(correcto == false){
        cout<<"Que algoritmo desea usar --> Mejor Hueco(m/M) o Siguiente Hueco(s/S): ";
        cin>>algoritmoUsado;

        if(algoritmoUsado == 's' || algoritmoUsado == 'S'){
            cout<<"Quieres seguir con la operacion del siguiente hueco (s/S): ";
            cin>>seg;

            if(seg == 's' || seg == 'S'){
                cout<<"Has seleccionado el algoritmo del siguiente hueco"<<endl;
                algoritmoSiguiente(entrada,  salida, cantMemoria);
            }
        }

        if(algoritmoUsado == 'm' || algoritmoUsado == 'M'){
            cout<<"Quieres seguir con la operacion del mejor hueco (s/S): ";
            cin>>seg;

            if(seg == 's' || seg == 'S'){
                cout<<"Has seleccionado el algoritmo del mejor hueco"<<endl;
                algoritmoMejor(entrada, salida, cantMemoria);
            }
        }
    }

    return 0;
}
```

En el main del programa declaramos, la memoria total del procesador y comprobamos que los ficheros introducidos son correctos, después hacemos elegir al usuario qué tipo de algoritmo elegir, y una vez elegido se pasará al algoritmo y lo enseñará por pantalla.

```cpp
#include<iostream>
#include <vector>
#include <stdlib.h>
#include <string>
#include <string.h>
#include <fstream>
#include <sstream>

using namespace std;

const string HUECO = "HUECO";
const int NEGATIVO = -1;

struct Proceso{
    string nombre;
    unsigned int tiempo;
    unsigned int llegada;
    unsigned int colaCant;
    unsigned int memNecesaria;
};

struct Procesador{
    unsigned int cantMemoria;
    vector <Proceso> enEjecucion;
    vector <Proceso> cola;
};

void lectura(vector<Proceso> &procesos, string entrada){
    ifstream ficheroEntrada;
    ficheroEntrada.open(entrada.c_str(), ios::in);

    if(ficheroEntrada.is_open()){
        string linea="";

        while(getline(ficheroEntrada, linea)){
            Proceso proceso;
            stringstream cinLine(linea);

            cinLine >> proceso.nombre >> proceso.llegada >> proceso.memNecesaria >> proceso.tiempo;
            proceso.colaCant = proceso.tiempo;
            procesos.push_back(proceso);
        }

        ficheroEntrada.close();
    }
}

void inicializar(Proceso &hueco, int memoria){
    hueco.nombre = HUECO;
    hueco.llegada = 0;
    hueco.colaCant = 0;
    hueco.tiempo = 0;
    hueco.memNecesaria = memoria;
}

void crearHueco(Procesador &procesador, int pos, int memoriaRestante){
    Proceso hueco;

    inicializar(hueco,memoriaRestante);
    procesador.enEjecucion.push_back(hueco);

    for(int i = procesador.enEjecucion.size() - 1 ; i > pos + 1; i--){
        swap(procesador.enEjecucion[i], procesador.enEjecucion[i-1]);
    }
}
```

Primero explicaré la función de los submétodos que se usarán en los algoritmos. Lo primero es declarar dos structs para los procesos. El método lectura abrirá el fichero pasado por parámetro e irá leyendo línea por línea e introduciendo los datos en un nuevo proceso.

El método inicializar, inicializa un nuevo proceso como si fuera un hueco y no un proceso.

El método crearHueco inicializa un hueco, después se irán colocando todos los procesos hacia la derecha y harán espacio al hueco restante.

```cpp
void reordenar(vector<Proceso> &procesos){
    for(int i=1; i < procesos.size(); i++){
        for(int j=0; j < procesos.size()-1; j++){
            if(procesos[j].llegada > procesos[j+1].llegada){
                swap(procesos[j], (procesos[j+1]));
            }
        }
    }
}

void eliminarProcesos(vector<Proceso> &enEjecucion){
    for(int k = enEjecucion.size() - 1; k >= 0 ; k--){
        if(enEjecucion[k].nombre.compare(HUECO) != 0){
            enEjecucion[k].colaCant--;
        }

        if(enEjecucion[k].colaCant == 0){
            enEjecucion[k].nombre = HUECO;
        }
    }
}

void juntarHuecos(vector<Proceso> &enEjecucion){
    for(int i = enEjecucion.size() -1; i >= 0; i--){
        if(i < enEjecucion.size()  - 1 &&  enEjecucion[i].nombre.compare(HUECO) == 0 && enEjecucion[i+1].nombre.compare(HUECO) == 0){
            enEjecucion[i].memNecesaria = enEjecucion[i].memNecesaria + enEjecucion[i+1].memNecesaria;
            int posicion = i+1;
            enEjecucion.erase(enEjecucion.begin() + posicion);
        }
    }
}

void intrCola(Procesador &procesador, bool &borrado){
    for(int i=0; i < procesador.enEjecucion.size() && borrado == false; i++){
        if(procesador.enEjecucion[i].nombre.compare(HUECO) == 0 && (procesador.cola[0].memNecesaria <= procesador.enEjecucion[i].memNecesaria)){
            int memoriaRestante = procesador.enEjecucion[i].memNecesaria - procesador.cola[0].memNecesaria;

            if(memoriaRestante == 0){
                swap(procesador.enEjecucion[i], procesador.cola[0]);
            }
            else{
                crearHueco(procesador, i, memoriaRestante);
                swap(procesador.enEjecucion[i], procesador.cola[0]);
                juntarHuecos(procesador.enEjecucion);
            }

            procesador.cola.erase(procesador.cola.begin());
            borrado = true;
        }
    }
}
```

El método reordenar, usa bucles y la opción swap (intercambia dos posiciones en un vector), para ordenar de mayor a menor los procesos.

El método eliminar procesos, recorre los procesos para saber si han finalizado, si es así se convertirán en huecos.

El método juntarHuecos recorre los vectores para ver los huecos que hay y los va juntando para crear uno más grande.

El método intrCola, mira los procesos de la cola y comprueba si caben en algún hueco del procesador.

```cpp
void intrPorcesador(vector<Proceso> &procesos, Procesador &procesador){
    bool posHueco = false;

    for(int i=0; i < procesador.enEjecucion.size(); i++){
        if(procesador.enEjecucion[i].nombre.compare(HUECO) == 0 && procesos[0].memNecesaria <= procesador.enEjecucion[i].memNecesaria){
            posHueco = true;
            int memoriaRestante = procesador.enEjecucion[i].memNecesaria - procesos[0].memNecesaria;

            if(memoriaRestante == 0){
                swap(procesador.enEjecucion[i], procesos[0]);
            }
            else{
                crearHueco(procesador, i, memoriaRestante);
                swap(procesador.enEjecucion[i], procesos[0]);
                juntarHuecos(procesador.enEjecucion);
            }
        }
    }

    if(posHueco == false){
        procesador.cola.push_back(procesos[0]);
    }

    procesos.erase(procesos.begin());
}

bool controlTiempo(vector<Proceso> cola, vector<Proceso> enEjecucion, int cantMemoria){
    bool seguir = false;

    if(cola.empty()){
        for(int i = 0; i < enEjecucion.size() && seguir == false; i++){
            if(enEjecucion[i].colaCant != 0){
                seguir = true;
            }
        }

        if(enEjecucion[0].nombre.compare(HUECO) == 0 && enEjecucion[0].memNecesaria == cantMemoria){
            seguir = false;
        }
    }
    else{
        seguir = true;
    }

    return seguir;
}
```

El método intrProcesador mira los procesos que hay en el procesador y comprueba si hay algún hueco donde quepan.

El método controlTiempo comprueba que no haya procesos en la cola que estén sin introducir, y después comprueba que sigan en funcionamiento.

```
void algoritmoSiguiente(string entrada, string salida, unsigned int cantMemoria){
    vector<Proceso> procesos;
    Procesador procesador;
    Proceso hueco;
    bool seguir = true;

    lectura(procesos,entrada);
    reordenar(procesos);
    inicializar(hueco,cantMemoria);

    procesador.cantMemoria = cantMemoria;
    procesador.enEjecucion.push_back(hueco);

    ofstream ficheroSalida;
    ficheroSalida.open(salida.c_str(), ios::out);

    if(ficheroSalida.is_open()){
        for(int instante = 1; seguir == true || procesos.empty()==false; instante++){
            cout << instante<<": ";
            ficheroSalida << instante<<": ";
            bool borrado;

            do{
                borrado = false;

                if(procesador.cola.empty() == false && (procesador.cola[0].memNecesaria > procesador.cantMemoria)){
                    procesador.cola.erase(procesador.cola.begin());
                    borrado = true;
                }

                if(procesador.cola.empty() == false){
                    intrCola(procesador, borrado);
                }
            }while(borrado == true);

            while(procesos.empty() == false && procesos[0].llegada == instante){
                intrPorcesador(procesos, procesador);
            }

            int posicionDeMemoria = 0;

            for(int i = 0; i < procesador.enEjecucion.size(); i++){
                cout << "[" << posicionDeMemoria << " " << procesador.enEjecucion[i].nombre << " " << procesador.enEjecucion[i].memNecesaria << "] ";
                ficheroSalida << "[" << posicionDeMemoria << " " << procesador.enEjecucion[i].nombre << " " << procesador.enEjecucion[i].memNecesaria << "] ";
                posicionDeMemoria = posicionDeMemoria + procesador.enEjecucion[i].memNecesaria;
            }
            cout << endl;
            ficheroSalida << endl;

            seguir = controlTiempo(procesador.cola, procesador.enEjecucion, cantMemoria);
            eliminarProcesos(procesador.enEjecucion);
            juntarHuecos(procesador.enEjecucion);
        }

        ficheroSalida.close();
    }
}
```

El algoritmo del siguiente hueco consiste en buscar un hueco libre a partir de la posición del último procesos asignado.

Primero se inicializa un procesador, hacemos la lectura del fichero de entrada, ordenamos los procesos que hayan sido introducidos por el fichero, y después se inicializa un proceso hueco.

Después se abre el fichero salida y se ponen los procesos en cola, después se ubican en memoria del procesador y se añaden al fichero de salida y se enseñan por pantalla.

Por último se comprueba que no haya procesos en cola o por colocar, se eliminan los procesos terminados y se juntan los huecos para juntar memoria.

```
void algoritmoMejor(string entrada, string salida, unsigned int cantMemoria){
    vector<Proceso> procesos;
    Procesador procesador;
    Proceso hueco;
    bool seguir = true;

    lectura(procesos,entrada);
    reordenar(procesos);
    inicializar(hueco,cantMemoria);

    procesador.cantMemoria = cantMemoria;
    procesador.enEjecucion.push_back(hueco);

    ofstream ficheroSalida;
    ficheroSalida.open(salida.c_str(), ios::out);

    if(ficheroSalida.is_open()){
        for(int i=1; seguir == true || procesos.empty() == false; i++){
            cout << i <<": ";
            ficheroSalida << i <<": ";
            bool borrado;

            do{
                borrado = false;

                if(procesador.cola.empty() == false && (procesador.cola[0].memNecesaria > procesador.cantMemoria)){
                    procesador.cola.erase(procesador.cola.begin());
                    borrado = true;
                }

                if(procesador.cola.empty() == false){
                    intrCola(procesador, borrado);
                }
            }while(borrado == true);

            while(procesos.empty() == false && procesos[0].llegada == i){
                int diff = NEGATIVO;
                int posicionProceso = NEGATIVO;

                for(int i=0; i < procesador.enEjecucion.size(); i++){
                    if(procesador.enEjecucion[i].nombre.compare(HUECO) == 0 && procesos[0].memNecesaria <= procesador.enEjecucion[i].memNecesaria){
                        if(diff > procesador.enEjecucion[i].memNecesaria - procesos[0].memNecesaria || diff == NEGATIVO){
                            posicionProceso = i;
                            diff = procesador.enEjecucion[i].memNecesaria - procesos[0].memNecesaria;
                        }
                    }
                }

                if(posicionProceso == NEGATIVO){
                    procesador.cola.push_back(procesos[0]);
                }
                else{
                    if(diff == 0){
                        swap(procesador.enEjecucion[posicionProceso], procesos[0]);
                    }
                    else{
                        crearHueco(procesador, posicionProceso, diff);
                        swap(procesador.enEjecucion[posicionProceso], procesos[0]);
                        juntarHuecos(procesador.enEjecucion);
                    }
                }

                procesos.erase(procesos.begin());
            }

            int posicionDeMemoria = 0;

            for(int i = 0; i < procesador.enEjecucion.size(); i++){
                cout << "[" << posicionDeMemoria << " " << procesador.enEjecucion[i].nombre << " " << procesador.enEjecucion[i].memNecesaria << "] ";
                ficheroSalida << "[" << posicionDeMemoria << " " << procesador.enEjecucion[i].nombre << " " << procesador.enEjecucion[i].memNecesaria << "] ";
                posicionDeMemoria = posicionDeMemoria + procesador.enEjecucion[i].memNecesaria;
            }
            cout << endl;
            ficheroSalida << endl;

            seguir = controlTiempo(procesador.cola, procesador.enEjecucion, cantMemoria);
            eliminarProcesos(procesador.enEjecucion);
            juntarHuecos(procesador.enEjecucion);
        }

        ficheroSalida.close();
    }
}

int checkArgumentos(int argc, char *argv[], string &entrada, string &salida){
    bool errores = false;

    for(int i = 1; i < argc && errores == false; i++){
        switch(i){
            case 1:
                entrada = argv[i];
            break;

            case 2:
                salida = argv[i];
            break;
        }
    }
    return errores;
}
```

El algoritmo del mejor hueco consiste en buscar el mejor hueco y más optimo para introducir los procesos

Primero se inicializa un procesador, hacemos la lectura del fichero de entrada, ordenamos los procesos que hayan sido introducidos por el fichero, y después se inicializa un proceso hueco.

Después se abre el fichero salida y se ponen los procesos en cola, se comprueba que los procesos tengan un hueco perfecto, si no es así se añaden a la cola y se crea un hueco en la memoria para el siguiente.

Luego se añaden al fichero de salida y se enseñan por pantalla.

Por último se comprueba que no haya procesos en cola o por colocar, se eliminan los procesos terminados y se juntan los huecos para juntar memoria.