```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/*
 *
 */

/*----------------------------ESTRUCTURAS-------------------------------------*/

typedef struct {
   char* nombre;
   int llegada;
   int copiaLlegada;
   int vacio;
   int tiempoEjec;
   int ciclosEj;
   int tPermanencia;
   int tRespuesta;
   int tEspera;
} Proceso;

typedef struct {
   int ciclosTotal;
   int ciclosCPU;
   char** liTiempo;
} CPU;
```

```
enum {
   FIFO, SJF
};


/*----------------------------------FUNCIONES-------------------------------*/

//Impresion de informacion
char* imprimirInfoCPU(CPU cpu);
char* imprimirProceso(Proceso proceso);
char* imprimirLista(Proceso *proceso, int numProcesos);

//Funciones para colas
int ordenarProceso(Proceso *proceso, int numProcesos, int tipoCola);
int esVacio(Proceso p1, Proceso p2);
int colaFIFO(Proceso p1, Proceso p2);
int colaSJF(Proceso p1, Proceso p2);
int calcCiclos(Proceso *proceso, int numProcesos);

//Inicialización de CPU, procesos y listas
void iniciarCPU(CPU *cpu, int ciclos);
void iniciarProceso(Proceso *proceso);
void iniciarListaProceso(Proceso *proceso, int numProcesos);

//Metodos principales
char* shortestJobFirst(Proceso *espera, int numProcesos, int tiempoCPU);
char* roundRobin(Proceso *espera, int numProcesos, int tiempoCPU, int cuanto);
void escribirFichero(char* nombre, char* info);

/*--------------------------------------------------------------------------*/
```

```
int main(int argc, char** argv) {
    int tiempoCPU, procesos, cuanto, i, j;
    char caracter[3];
    char fichero[255];
    Proceso *lista;
    FILE *ficheroBase;
    char op;

    /*-------------------------COMPROBACION DE ERRORES-------------------------*/

    /* 1: Número correcto de argumentos*/
    if (argc != 2) {
        printf("falta el parametro de llamada.\n");
        printf("Sintaxis Correcta: Planificador_SO.exe \"C:\\ficheroEntrada.txt\"\n");
        return (EXIT_FAILURE);
    }

    /* 2: Error al abrir el archivo*/
    ficheroBase = fopen(argv[1], "r");
    if (ficheroBase == NULL) {
        fprintf(stderr, "fopen() failed in file %s at line # %d\n", __FILE__, __LINE__);
        printf("ERROR: el fichero pasado como parametro no existe.\n");
        return (EXIT_FAILURE);
    }

    /* 3: Datos erróneos en la primera linea del archivo*/
    i = fscanf(ficheroBase, "%c%c%d%c", &caracter[0], &caracter[1], &procesos, &caracter[2]);
    if (i != 4 || caracter[0] != 'p' || caracter[1] != '=' || caracter[2] != '\n') {
        printf("El fichero de entrada no contiene en la primera linea: p='numero'\n");
        return (EXIT_FAILURE);
```

```
}

/* 4: Datos erróneos en la segunda linea del archivo*/
lista = (Proceso*) malloc(procesos * sizeof (Proceso));
iniciarListaProceso(lista, procesos);
i = fscanf(ficheroBase, "%c%c%d%c", &caracter[0], &caracter[1], &cuanto, &caracter[2]);
if (i != 4 || caracter[0] != 'q' || caracter[1] != '=' || caracter[2] != '\n') {
    printf("El fichero de entrada no contiene en la segunda linea: q='numero'\n");
   return (EXIT_FAILURE);
}
i = 0;

/* 5: Comprueba si todos los procesos tienen todos los datos*/
while (!feof(ficheroBase) && i < procesos) {
   lista[i].nombre = (char*) malloc(255 * sizeof (char));
   if (fscanf(ficheroBase, "%s %d %d%*c", &lista[i].nombre[0],
         &lista[i].llegada, &lista[i].tiempoEjec) != 3) {
      printf("Error en el fichero de entrada.\n");
      printf("Formato: <Nombre> <TLlegada> <TEjecucion>.\n");
      printf("Ej.: X 4 3\n");
      return (EXIT_FAILURE);
   }
   lista[i].vacio = 0;
   i++;
}

/* 6: Comprueba si hay datos para todos los procesos*/
fclose(ficheroBase);
if (i != procesos) {
   printf("No hay datos para todos los procesos.");
   return (EXIT_FAILURE);
```

```
}
/*-------------------------------------------------------------------------*/

//Se realiza una copia de las llegadas por si fueran necesarias
for (i = 0; i < procesos; i++) {
   lista[i].copiaLlegada = lista[i].llegada;
}

// A continuacion ordenamos las lista de procesos segun el orden de llamada
// y calculamos el tiempo de ejecucion
i = ordenarProceso(lista, procesos, FIFO);
tiempoCPU = calcCiclos(lista, procesos);

// Ejecutamos los dos algoritmos y guardamos los resultados
printf("////////////////////PLANIFICADOR DE PROCESOS EN C////////////////////\n\n");
printf("Elaborado por: \n");
printf("   Paloma Sanchez de la Torre\n");
printf("   Cristina Olmedilla Lopez\n");
printf("   Jose Maria Cuevas Izaguirre\n\n");
printf("////////////////////////////////////////////////////////////////////\n\n");

while(op != 'E'){
        printf("Introduzca el algoritmo que desea simular:\n");
        printf("S: Algoritmo SJF.\n");
        printf("R: Algoritmo Round Robin.\n");
        printf("E: Salir del programa.\n\n");
        scanf(" %c",&op);
        switch (op) {
         case 'S': //EJECUTA FUNCION SJF
            sprintf(fichero, "%s_SJF", argv[1]);
            escribirFichero(fichero, shortestJobFirst(lista, procesos, tiempoCPU));
```

```
                printf("SJF Ejecutado y resultados guardados.\n\n");
                break;
            case 'R': //EJECUTA RR
                sprintf(fichero, "%s_RR", argv[1]);
                escribirFichero(fichero, roundRobin(lista, procesos, tiempoCPU, cuanto));
                printf("RR ejecutado y resultados guardados.\n\n");
                break;

            case 'E':
                free(lista);
                return (EXIT_SUCCESS);
                break;
        }
         }
         return 0;



}

/*Devuelve un string con la información de la CPU pasada por parámetro*/
char* imprimirInfoCPU(CPU cpu) {
   int i;
   float uso;
   char* solucion = (char*) malloc(255 * sizeof (char));
   sprintf(solucion, "\n-");
   for (i = 0; i < cpu.ciclosTotal; i++) {
      sprintf(solucion, "%s %s -", solucion, cpu.liTiempo[i]);
   }
   if (cpu.ciclosTotal == 0) {
      uso = 0;
```

```
    } else {
        uso = (float) cpu.ciclosCPU / (float) cpu.ciclosTotal;
    }
    sprintf(solucion, "%s\nUso de la CPU en tanto por ciento: %f%%", solucion, uso * 100);
    return solucion;
}

/*Devuelve un string con la información del proceso pasado por parámetro*/
char* imprimirProceso(Proceso proceso) {
    char* solucion = (char*) malloc(255 * sizeof (char));
    sprintf(solucion, "\nNombre de proceso %s", proceso.nombre);
    sprintf(solucion, "%s\nAcaba en el ciclo %d\nTiempo Permanencia = %d", solucion,
proceso.llegada, proceso.tPermanencia);
    sprintf(solucion, "%s\nTiempo de Respuesta = %d\nTiempo de Espera = %d\n", solucion,
proceso.tRespuesta, proceso.tEspera);
    return solucion;
}

/*Devuelve un string con la información de los procesos pasados por parámetro*/
char* imprimirLista(Proceso *proceso, int numProceso) {
    char* solucion = (char*) malloc(255 * sizeof (char));
    int i;
    solucion = imprimirProceso(proceso[0]);
    for (i = 1; i < numProceso; i++) {
        sprintf(solucion, "%s%s", solucion, imprimirProceso(proceso[i]));
    }
    return solucion;
}

/*Ordena los procesos y devuelve el que está en primer lugar*/
int ordenarProceso(Proceso *proceso, int numProcesos, int tipoCola) {
    int i, j, posicion = numProcesos;
```

```
Proceso auxiliar;
if (tipoCola == 0) {
   for (i = 0; i < numProcesos - 1; i++) {
      for (j = 0; j < numProcesos - i - 1; j++) {
         if (colaFIFO(proceso[j], proceso[j + 1]) == 1) {
            auxiliar = proceso[j];
            proceso[j] = proceso[j + 1];
            proceso[j + 1] = auxiliar;
         }
      }
   }
} else if (tipoCola == 1) {
   for (i = 0; i < numProcesos - 1; i++) {
      for (j = 0; j < numProcesos - i - 1; j++) {
         if (colaSJF(proceso[j], proceso[j + 1]) == 1) {
            auxiliar = proceso[j];
            proceso[j] = proceso[j + 1];
            proceso[j + 1] = auxiliar;
         }
      }
   }
}
for (i = 0; i < (numProcesos - 1); i++) {
   for (j = 0; j < numProcesos - 1 - i; j++) {
      if (esVacio(proceso[j], proceso[j + 1]) == 1) {
         auxiliar = proceso[j];
         proceso[j] = proceso[j + 1];
         proceso[j + 1] = auxiliar;
      }
   }
}
```

```
    i = 0;
    while (i < numProcesos && posicion == numProcesos) {
        if (proceso[i].vacio == 1) {
            posicion = i;
        }
        i++;
    }
    return posicion;
}

/*Comprueba si p2 está vacio*/
int esVacio(Proceso p1, Proceso p2) {
    if (p1.vacio == p2.vacio)
        return 0;
    else if (p1.vacio < p2.vacio)
        return -1;
    else
        return 1;
}

/*Ordena segun el algoritmo FIFO*/
int colaFIFO(Proceso p1, Proceso p2) {
    if (p1.llegada < p2.llegada) {
        return -1;
    } else if (p1.llegada > p2.llegada) {
        return 1;
    } else {
        return strcmp(p1.nombre, p2.nombre);
    }
}
```

```
/*Ordena los procesos empezando por el más corto*/
int colaSJF(Proceso p1, Proceso p2) {
   if (p1.tiempoEjec < p2.tiempoEjec)
      return -1;
   else if (p1.tiempoEjec > p2.tiempoEjec)
      return 1;
   else
      return colaFIFO(p1, p2);
}

/*Calcula el numero total de ciclos que la CPU debe tener para finalizar*/
int calcCiclos(Proceso *proceso, int numProcesos) {
   int i, total = 0;
   for (i = 0; i < numProcesos; i++) {
      if (proceso[i].llegada > total) {
         total = proceso[i].llegada;
      }
      total += proceso[i].tiempoEjec;
   }
   return total;
}

/*Se inicializa una estructura CPU con valores iniciales*/
void iniciarCPU(CPU *cpu, int ciclos) {
   int i;
   cpu->ciclosTotal = 0;
   cpu->ciclosCPU = 0;
   cpu->liTiempo = (char**) malloc(ciclos * sizeof (char*));
   for (i = 0; i < ciclos; ++i) {
      cpu->liTiempo[i] = (char*) malloc(4 * sizeof (char));
      cpu->liTiempo[i] = "nop";
```

```
    }
}

/*Se inicializa una estructura Proceso con valores iniciales*/
void iniciarProceso(Proceso *proceso) {
    proceso->nombre = (char*) malloc(4 * sizeof (char));
    proceso->nombre = "nop";
    proceso->llegada = 0;
    proceso->copiaLlegada = 0;
    proceso->vacio = 1;
    proceso->tiempoEjec = 0;
    proceso->ciclosEj = 0;
    proceso->tPermanencia = 0;
    proceso->tRespuesta = 0;
    proceso->tEspera = 0;
}

/*Se inicializa una lista de procesos con valores iniciales*/
void iniciarListaProceso(Proceso *proceso, int numProceso) {
    int i;
    for (i = 0; i < numProceso; i++) {
        iniciarProceso(&proceso[i]);
    }
}

/*Algoritmo SJF, donde los procesos más cortos se ejecutan antes*/
char* shortestJobFirst(Proceso *espera, int numProcesos, int tiempoCPU) {
    int i, j, k;
    int x = 0;
    char *solucion;
    Proceso listos[numProcesos];
```

```
Proceso terminados[numProcesos];
Proceso enEjecucion;
CPU cpu;

//Inicialización de estructuras
iniciarProceso(&enEjecucion);
iniciarListaProceso(listos, numProcesos);
iniciarListaProceso(terminados, numProcesos);
iniciarCPU(&cpu, tiempoCPU);

for (i = 0; i < tiempoCPU; i++) {
   //Se ordenan procesos por orden de llegada
   ordenarProceso(espera, numProcesos, FIFO);

   //Solo aquellos que no esten vacios y que hayan llegado en el ciclo
   //pasan a ordenarse (SJF) en lista de procesos listos
   for (j = 0; j < numProcesos; j++) {
      if (espera[j].vacio == 0) {
         if (espera[j].llegada == i) {
            k = ordenarProceso(listos, numProcesos, SJF);
            listos[k] = espera[j];
         }
      }
   }
   k = ordenarProceso(listos, numProcesos, SJF);

   //Mientras no haya ningun proceso en ejecución y existe un proceso listo,
   //los valores de respuesta y se marca como vacio en la cola de listos
   if (enEjecucion.vacio == 1) {
      if (listos[0].vacio == 0) {
         enEjecucion = listos[0];
```

```
            enEjecucion.tRespuesta = i - enEjecucion.llegada;
            listos[0].vacio = 1;
        }
    }

    //Se realiza la ejecución del proceso, sumando 1 a los ciclos de la CPU,
    //colocando el nombre de proceso en la linea de tiempo y sumando 1 a
    //los ciclos ejecutados
    if (enEjecucion.vacio == 0) {
        cpu.ciclosCPU++;
        cpu.liTiempo[i] = enEjecucion.nombre;
        enEjecucion.ciclosEj++;

        //Si el proceso hubiera terminado, se modifican los valores de permanencia,
        //espera y llegada, y se introduce al proceso en la lista de terminados.
        //Finalmente se marca al procesador como libre y se añade un ciclo
        //al numero total
        if (enEjecucion.ciclosEj == enEjecucion.tiempoEjec) {
            enEjecucion.tPermanencia = i + 1 - enEjecucion.copiaLlegada;
            enEjecucion.tEspera = enEjecucion.tPermanencia - enEjecucion.tiempoEjec;
            enEjecucion.llegada = i + 1;
            terminados[x] = enEjecucion;
            x++;
            enEjecucion.vacio = 1;
        }
    }
    cpu.ciclosTotal++; // Se suma 1 al numero total de ciclos de la simulación
}
i = strlen(imprimirInfoCPU(cpu));
i += strlen(imprimirLista(terminados, numProcesos));
solucion = (char*) malloc((37 + i) * sizeof (char));
```

```
   sprintf(solucion, "Datos de la CPU\n\nLinea de tiempo:%s\n\nPROCESOS%s\n",
imprimirInfoCPU(cpu), imprimirLista(terminados, numProcesos));
   return solucion;
}

/*Algoritmo RR, donde se siguen un cuanto y una cola FIFO para ejecutar los procesos*/
char* roundRobin(Proceso *espera, int numProcesos, int tiempoCPU, int cuanto) {
   int i, j, k;
   int x = 0;
   int cuantoActual = 0;
   char *solucion;
   Proceso listos[numProcesos];
   Proceso terminados[numProcesos];
   Proceso enEjecucion;
   CPU cpu;

   //Inicialización de estructuras
   iniciarProceso(&enEjecucion);
   iniciarListaProceso(listos, numProcesos);
   iniciarListaProceso(terminados, numProcesos);
   iniciarCPU(&cpu, tiempoCPU);


   for (i = 0; i < tiempoCPU; i++) {
      for (j = 0; j < numProcesos; j++) {
         //Solo aquellos que no esten vacios y que hayan llegado en el ciclo
         //pasan a ordenarse (FIFO) en lista de procesos listos
         if (espera[j].vacio == 0) {
            if (espera[j].llegada == i) {
               k = ordenarProceso(listos, numProcesos, FIFO);
               listos[k] = espera[j];
```

```
        }
    }
}
k = ordenarProceso(listos, numProcesos, FIFO);

//Mientras no haya ningun proceso en ejecución y existe un proceso listo,
//si éste útimo es ejecutado por primera vez, los valores de respuesta
//y se marca como vacio en la cola de listos
if (enEjecucion.vacio == 1) {
    if (listos[0].vacio == 0) {
        enEjecucion = listos[0];
        if (enEjecucion.copiaLlegada == enEjecucion.llegada) {
            enEjecucion.tRespuesta = i - enEjecucion.llegada;
        }
        listos[0].vacio = 1;
    }
}

//Se realiza la ejecución del proceso, sumando 1 a los ciclos de la CPU,
//colocando el nombre de proceso en la linea de tiempo y restando 1 a
//los ciclos necesarios para terminar
if (enEjecucion.vacio == 0) {
    cpu.ciclosCPU++;
    cpu.liTiempo[i] = enEjecucion.nombre;
    enEjecucion.ciclosEj++;

    //Si el proceso hubiera terminado, se modifican los valores de permanencia,
    //espera y llegada
    if (enEjecucion.ciclosEj == enEjecucion.tiempoEjec) {
        enEjecucion.tPermanencia = i + 1 - enEjecucion.copiaLlegada;
        enEjecucion.tEspera = enEjecucion.tPermanencia - enEjecucion.tiempoEjec;
```

```
            enEjecucion.llegada = i + 1;
            terminados[x] = enEjecucion;
            x++;
            enEjecucion.vacio = 1;
            cuantoActual = 0;
          } else {

            //Si el proceso no hubiera terminado, la llegada se modificaria con
            //el ciclo siguiente, se reordenaria la cola de listos para obtener
            //el primer hueco. Ademas se tendrá que marcar a la CPU como libre
            cuantoActual++;
            if (cuantoActual >= cuanto) {
              enEjecucion.llegada = i + 1;
              k = ordenarProceso(listos, numProcesos, FIFO);
              listos[k] = enEjecucion;
              enEjecucion.vacio = 1;
              cuantoActual = 0;
            }
          }
        }
        cpu.ciclosTotal++;
    }
    i = strlen(imprimirInfoCPU(cpu));
    i += strlen(imprimirLista(terminados, numProcesos));
    solucion = (char*) malloc((37 + i) * sizeof (char));
    sprintf(solucion, "Datos de la CPU\n\nLinea de tiempo:%s\n\nPROCESOS%s\n",
imprimirInfoCPU(cpu), imprimirLista(terminados, numProcesos));
    return solucion;
}

/*Se encarga de la escritura de un contenido utilizando los valores pasads por parámetro*/
```

```
void escribirFichero(char* nombre, char* info) {
    FILE *resultado = fopen(nombre, "w");
    if (resultado != NULL) {
        fputs(info, resultado);
        fclose(resultado);
    } else {
        printf("Error al crear el archivo %s", nombre);
    }
}
```