UNIVERSIDAD NACIONAL
COSTA RICA

Facultad de Ciencias Exactas y Naturales

Asignatura:

Sistemas Operativos

PROYECTO 1: MEMORIA

Profesor:

Eddy Miguel Ramírez

Estudiantes:

José Isaac Zeledón Jiménez
Jonathan Estrada Vargas

I CICLO

2019

Índice

1. Descripción del documento y del problema

En este documento se describe como se desarrolló el proyecto 1 de Sistemas Operativos en el que se refozaron los conceptos concernientes a la memoria y a la asignación de esta.
En este proyecto el problema inicia con el generador de calendarización de procesos el cual debe de recibir por consola

- Número de procesos
- Pid inicial
- Memoria mínima
- Memoria máxima
- Duración mínima
- Duración máxima
- Hora llegada inicial
- lambda media de Poisson

De estos datos se debía de generar los procesos que posteriormente serían utilizados para la calendarización en "memoria", todo esto se lograría con un programa en C linux, el cual como salida en la consola daría:

- Número o identificador único
- Unidad de tiempo que inicia
- Duración en unidades de tiempo
- Un tamaño en bytes

Con estos datos y utilizando el operador ">"de la consola de linux se debía de generar un archivo que guardara todos los procesos generados, para la simulación, este archivo es el que leería el módulo encargado de la calendarización, que además de poseer los datos de todos los procesos, debe leer un archivo de configuración con el cual podrá comportarse de la manera deseada ya que esta parte del proyecto requería que la simulación pudiera hacer la calendarización del recurso de la memoria en diferentes enfoques, más que todo para poseer una visualización del comportamiento de los diferentes algoritmos y formas de asignar memoria a procesos.

En este documento se describirá la manera en que abordamos el problema, los problemas que se afrontaron durante la realización del mismo, los algoritmos utilizados para cada uno de los modos de ejecución de la simulación y las conclusiones, las cuales van en función de los aprendizajes adquiridos a la hora del desarrollo de la simulación y además de los comportamientos observados durante la pruebas de las misma simulación. Con esto se refuerza de una forma visual los conceptos relacionados a la asignación de la memoria y la calendarización de procesos en esta.

2. Especificacion de la solución

Para la primera parte del proyecto se debía de utilizar una distribución de probabilidad la cual iban a seguir los procesos a la hora de su creación, esta distribución de probabilidad es la de Poisson.
Para esto se utilizó el cálculo de esta distribución de la probabilidad de Poisson que sigue esta fórmula:

$$\frac{e^{-lambda} lambda^{k}}{k!}$$

Esta fórmula se utilizó para que los procesos en el generador, se crearan, pero con esta distribución se debía de utilizar un concepto llamado tabla acumulada debido a que para calcular las horas de creación de los procesos se necesitaban valores en el eje de las abcisas osea enteros positivos, que estuvieran relacionados a una probabilidad de esta distribución. Para esto se utilizo el seguiente codigo brindado por el profesor del curso, mas adenlante en problemas encontrados se describirá como se intentó resolver pero no se tuvo éxito. Tabla acumulada:

```
long double * tablaPoisson;

void creaTabla(int lambda){
    tablaPoisson = calloc(sizeof(long double), 3*lambda);
    tablaPoisson[0] = exp(-1*lambda);
    int i =0; int ifactorial = 1;
    while(i++ <3*lambda){
        ifactorial *=i;
        tablaPoisson[i] = tablaPoisson[i-1]+
                          tablaPoisson[0]*
                          pow(lambda,i)/ifactorial;
    }
    tablaPoisson[i-1] =1.0;
}
```

Con el código anterior se generaba la tabla, que para un lambda n generaba una tabla de probabilidades desde $0 * lambda$ hasta $3 * lambda$ que era el rango máximo que el profesor especifico en el documento del proyecto.
Luego se utilizaba un método el cual para un número random cualquiera, se devolvía un entero que correspondía a un valor en en las abcisas de la distribución.
Codigo que retorna un valor en la distribución de Poisson:

```
int valorPoisson(long double r){
    int respuesta =0;
    while(r>tablaPoisson[respuesta++]);
    return respuesta-1;
}
```

Con lo anterior se solucionaba la forma en la que los procesos variaban en sus horas de llegada a la cola de procesos

2.1. Descripción de los algoritmos utilizados

2.1.1. Lectura del Archivo de Procesos

Para este problema se necesitó la ayuda de un método que pudiera leer de un archivo .txt previamente creado por el generador de procesos. Antes de poder leer el archivo se creó una lista enlazada de procesos con las siguientes caracteristicas:

```
struct proceso
{
int id;
int duracion;
int horallegada;
int tama;
struct proceso *next;
}
```

Esto con el fin de poder almacenar cada uno de los procesos leídos durante el método. Para relizar dicha lectura, se utilizó el método "fscanf"de la libería "stdio.h". Este método lee las cádenas de caracteres que están antes de un tab o un salto de línea. Gracias a eso y ayudados por un arreglo de tamaño 4 se pudo leer cada uno de los atributos del proceso ya que era sabido que son 4 atributos. Cada vez que la variable cantidad sea 3, se sabe que ya leyó 3 atributos de un proceso. Entonces solo queda un atributo más por leer, se lee el atributo y se inserta en la lista enlazada un proceso con los 4 atributos antes leídos.

```
aux
while (!feof(file))
{
char aux[6];
fscanf(file,"%s",aux);
if(cant==3){
array[3]=atoi(aux);
push(head, array[0],array[1],array[2],array[3]);
cant=0;
}
else
array[cant++]=atoi(aux);
}
```

2.1.2. Lectura del Archivo de Cofiguración

La lectura de este archivo fue más compleja que la del archivo de procesos dado que el archivo tenía su propio sistema de comentado con el caratér '#'

. Esta lectura fue posible gracias a la función 'fgets()' de la librería "stdio". Esta función lee toda la cadena de caracteres que en la línea, y al encontrarse con un salto de línea, sigue con la siguiente. Leída una línea del archivo se analizaba la cadena de caracteres leída y se descartaba todo lo que estuviera después de un espacio, incluido el espacio. Luego de esto se inserta en un arreglo pasado por parámetro la cadena resultante.

```
  char s[MAX];
int i = 0;
while (!feof(file))
{
fgets(s, MAX, file);
char *aux = calloc(sizeof(char), 15);
for (int j = 0; j < MAX; j++){
if (s[j] == ' ')
break;
else
aux[j] = s[j];
}
arg[i++] = aux;
memset(&(s[0]), 0, MAX);
}
```

3. Descripción del manejo de la simulación

3.1. Modo Secuencial

3.1.1. FIFO

Este algoritmo es el más sencillo de todos ya que no se necesita iterar la lista de procesos ni la memoria. Simplemente el proceso que esté de primero entra a la memoria, siempre y cuando su hora de llegada lo permita. Para este método se utilizó la lista de procesos mencionada anteriormente y para la memoria se utilizó una lista enlazada con las siguientes características:

```
typedef struct bloque
{
int id;
int tama;
int duracion;
struct bloque *next;
}
```

Para la inserción en la lista enlazada de bloques de memoria, se realizó el método 'pushMem()'. Este método inserta el proceso en la lista, siempre y cuando haya espacio disponible. Cuando lo va a insertar, se inserta otro nodo con el resultado de la resta del tamaño del bloque disponible con el tamaño del bloque a insertar. Luego de esto se le resta el tamaño del bloque insertado al tamaño total de la memoria.

```
bloque *temp = (bloque*)malloc(sizeof(bloque));
temp->id = 0;
temp->tama = current->tama - temp->tama;
temp->duracion = 0;
current->id = id;
current->duracion=duracion;
current->tama = tama;
temp->next = current->next;
current->next = temp;
currentSize -= tama;
```

Para nuestro control, un bloque de memoria con un ID menor o igual a cero es un bloque de memoria desocupado. Y puesto que teníamos que estar actulizando los bloques mediante el tiempo transcurrido, hicimos el método 'limpiaId()' que básicamente lo que hace es poner en cero a todos los ID de los bloques que tengan una duración menor o igual a cero.

```
void limpiaId(bloque **head){
bloque * current = (*head);
while(current != NULL ){
if(current->id != 0 && current->duracion<=0 )
current->id=0;

current = current->next;
    }
}
```

Pero teníamos un problema, no teníamos forma de restar unidades de tiempo internamente en la memoria. Dado esto problema, la solución que aplicamos fue hacer una función que iterara los bloques de la memoria y restara una unidad de tiempo a cada uno de los bloques donde la duración fuera estrictamente mayor a cero.

```
void restaDuracion(bloque**head){
bloque* current= (*head);
while(current != NULL ){
if(current->duracion > 0)
current->duracion--;
current=current->next;
   }
}
```

El mayor problema que tuvimos a la hora de realizar los algoritmos de modo secuencial fue a la hora de que dos bloques que estuvieran desocupados se hicieran un solo bloque, sumando así sus respectivos tamaños. Para eso implementamos la función 'juntarBloques()'. Esta función fue la que más tiempo duramos en implementar, ya que había que tomar en cuenta todos los posibles casos en que pudiera estar acomodada la lista enlazada. Puesto que si habían dos bloques consecutivos pero uno de esos bloques no tenía el ID en cero (recordemos que cero en el ID es un bloque desocupado), no se podía realizar la operación. Además, había que tomar en cuenta si solo había un bloque y también si habían más de dos bloques consecutivos que estaban desocupados. Para saber si la simulación ya había terminda se realizó la función 'Terminado()' la cual retorna verdadero si todos los bloques de la lista enlazada tienen su ID menor o igual a cero(Recordemos la función 'limpiaId()' explicada anteriormente).

```
   bool Terminado(bloque**head){
bloque* current=(*head);
while(current != NULL){
if(current->duracion > 0 && current->id > 0)
return false;

current=current->next;
      }
return true;
 }
```

3.1.2. Best-fit

Para este algoritmo se utilizó la mayoría de métodos que se implementaron para FIFO a diferencía que cuando la memoria se llena y todavía quedan procesos sin entrar a la memoria se busca: En la lista de procesos al proceso con menor tamaño y además que su hora de llegada sea menor o igual a la unidad de tiempo actual. En la lista de bloques de memoria, se busca el bloque con menor tamaño que pueda equiparar el tamaño del proceso a insertar. Esto se hace con el objetivo de se produzca un bloque resultante con el menor tamaño posible.

```
En la lista de procesos
int retornaPosBF(p** head,int tiempo){

int pos=0;
int aux1=0;
if((*head)->horallegada <= tiempo)
aux1=(*head)->tama;
else
aux1 = -1;

p* current=(*head);
while(current != NULL){
if(aux1 == -1){
if(current->horallegada <= tiempo)
aux1 = current->tama;
}
else
{
if(current->tama < aux1 && current->horallegada <= tiempo)
aux1=current->tama;
}

current=current->next;
}
current = (*head);
while(current != NULL){
if(current->tama == aux1){
return pos;
}
current = current->next;
```

```
pos++;
}
return -1;
}
```

En la lista de bloques de memoria

```
 int retornaPosMBF(bloque** head, int tam){
int aux = 0;
if((*head)->id > 0)
aux=-1;
else
aux= (*head)->tama;

int pos=0;
bloque* current = (*head);

while(current != NULL) {
if(aux==-1){
if(current->id >0)
aux = current->tama;
}
else{
if(current->tama < aux && current->id > 0 )
aux=current->tama;
}
current = current->next;
}

current=(*head);
while(current != NULL){
if(current->tama == aux)
return pos;

current= current->next;
pos++;
}
return -1;

}
```

El manejo de las unidades de tiempo y la verificación de si la simulación ya estaba terminada se implementó exactamente igual que se hizo en FIFO.

3.1.3. Worst-fit

Este modo es muy parecido a Best Fit a diferencia que su objetivo es insertar en memoria los procesos con mayor tamaño. Y así el tamaño sobrante es más grande y puede ser aprovechado. Esto se realiza iterando la lista de procesos en busca del proceso con mayor tamaño(siempre y cuando su hora de llegada lo permita). En la lista de bloques de memoria se busca el bloque con el tamaño más grande que pueda equiparar el tamaño del proceso a insertar.

En la lista de procesos

```
int retornaPosWF(p** head,int tiempo){
int pos=0;
int aux1=0;
if((*head)->horallegada <= tiempo)
aux1=(*head)->tama;
else
aux1 = -1;
```

```
p* current=(*head);
while(current != NULL){
if(aux1 == -1){
if(current->horallegada <= tiempo)
aux1 = current->tama;
}
else
{
if(current->tama > aux1 && current->horallegada <= tiempo)
aux1=current->tama;
}

current=current->next;
}
current = (*head);
while(current != NULL){
if(current->tama == aux1){
return pos;
}
current = current->next;
pos++;
}
return -1;
}
```

En la lista de bloques de memoria

```
int retornaPosMWF(bloque** head, int tam){
int aux = 0;
if((*head)->id > 0)
aux=-1;
else
aux= (*head)->tama;

int pos=0;
bloque* current = (*head);

while(current != NULL) {
if(aux==-1){
if(current->id >0)
aux = current->tama;
}
else{
if(current->tama > aux && current->id > 0 )
aux=current->tama;
}
current = current->next;
}
current=(*head);
while(current != NULL){
if(current->tama == aux)
return pos;

current= current->next;
pos++;
}
return -1;
```

```
}
```

3.2. Modo Paginación

Este modo de manejo de la memoria es el que ejemplifica a la técnica de asignación de memoria actual el cual como se vio en clase es una combinación entre la memoria secuencial y la memoria dínamica(esta solo conceptual, nunca se implemento).
Para el manejo de la memoria en modo de paginación se decidió que la mejor manera de simular esta calendarización era utilizando una estructura de datos que se asemejara a como se distribuye la memoria en realidad, se utilizó una matriz en la que cada posicion de (matriz[i][j]) representaba una pagina de la memoria, asi que el problema era simular como la paginación tiene lugar actualmente. Para esto se utilizó además un struct página del cual se instanciaría en la matriz, que se utilizaría.

```
typedef struct pagina
{
    int id;
    int duracion;
    int tama;
    int sobrante;
} page;
```

En este struct como se puede apreciar que las páginas comparten ciertos atributos con los procesos que están en la otra estructura de datos auxiliar que se utilizo para el manejo de la calendarizción de los procesos en memoria.

```
typedef struct proceso
{
    int id;
    int duracion;
    int horallegada;
    int tama;
    struct proceso *next;
} p;
```

Una de la diferencias es más significativas de estos dos structs es que "pagina"posee un atributo de tipo entero llamado sobrante el cual guardará en él, el residuo de operar el tamaño del proceso con el tamaño de la página para que esto almacene este residuo y se mostrase de ser necesario ya que un proceso puede entrar y necesitar de sólo de una parte de la memoria total de la página.
El enfoque utilizado para la simulación de paginación fue el de calendarizar a los procesos conforme a su hora de llegada, así que se utilizó FIFO en la calendarización de procesos.
En paginación los procesos se ingresa en las páginas que esten vacías, asi que se puede ver como los procesos dentro de la memoria se distribuye en posiciones desordenadas de la matriz que representa la memoria.
En el siguiente segmento de código es lo que se utilizó en el proyecto para que manejar el ingreso de procesos a la memoria:

```
...
if (matriz[i][j] == NULL)
                    {
                        page *nueva = malloc(sizeof(page));
                        matriz[i][j] = nueva;
                        nueva->id = (*head)->id;
                        nueva->duracion = (*head)->duracion;
                        nueva->sobrante = 0;
                        (*head)->tama -= tampage;
                        if ((tampage - (*head)->tama) >= 0)
                        {
                            nueva->sobrante = tampage - (*head)->tama;
                            dequeue(head);
```

```
                        espera = *head;
                        break;
                    }
                }
...
```

Se puede apreciar el código anterior como es el proceso de asignar una página de memoria para un proceso, y que además se hace la comprobación de si ya el proceso está completamente en memoria para así liberar la cola de procesos, así en la salida de Latex se puede apreciar como se llenan las páginas de la memoria, y como en instantes de la simulación algunos procesos tardan más que otros en entrar en memoria.

Además de ingresar procesos en memoria también se les debe de retirar de la memoria los procesos que cumplan un tiempo en ella.

```
...
else
                {
                    matriz[i][j]->duracion -= 1;
                    if (matriz[i][j]->duracion == 0)
                    {
                        free(matriz[i][j]);
                        matriz[i][j] = NULL;
                    }
                }
...
```

La seccion de código anterior es la utilizada para liberar las "páginas"de la memoria de los procesos que ya cumplian con la condición de que su duración fuera igual a cero, en la parte de liberar las páginas se consideró al inicio uitlizar páginas "fantasma"lo que haría que al final se deba de hacer un método solo para liberar la memoria de la matriz ya que se utlizó malloc para asignar a cada página, ademas de que igualar a NULL nos es útil a la hora de avanzar en los instantes de la simulación.

3.3. Impresión en el archivo LaTeX

Para la impresión en el archivo laTeX se utilizaron tres funciónes:

- 'printBegin()'
- 'printTex()'
- 'printEnd()'

En 'printBegin' básicamente se imprime en el archivo .tex todo lo relacionado al encabezado principal. Este método se pone al principio de cada método que se implementó para los diferentes algoritmos de manejo de memoria.

```
  fputs("\\documentclass[10pt,a4paper]{article}\n\\usepackage[utf8]{inputenc}\n\\begin{document}\n\\beg

fprintf(tex,"\\section*{Configuracion}\n\\begin{description}\n\\item[Algoritmo:] %s \n", algoritmo);

fprintf(tex,"\\item[Tamaño total:] %d \n\\end{description}\n\\end{center}\n",tamaOri);
```

Recordemos que en C el backslash'(\)' es un caractér reservado. Dado esto, para poder imprimir un solo backslash se necesita escribir dos veces el bacslash.
En 'printTex()' es donde se imprime las listas. Para cada instante se imprime la memoria simulada que en nuestro caso sería la lista enlazada de bloques de memoria en forma de stack. La lista de procesos se imprime sí y solo sí hay procesos que su hora de llegada es mayor al instante y no han podido ingresar a la memoria (simulando una cola). Para eso, se itera la lista de bloques de memoria y se imprime. Luego de esta impresión se verifica si hay algún proceso 'encolado' y de ser así se imprime. Si la lista de procesos está vacía se imprime que la lista está vacía para ese instante.

Codigo en C

```
char * msg="\\begin{center}\n\nInstante: ";
fprintf(tex,"%s%d%s",msg,tiempo,"\n\n");
char* msg2="\\begin{tabular}{|c|c|}\n\\hline\n";
fprintf(tex,"%s",msg2);
bloque* currentM = (*mem);
p* currentL = (*head);

while(currentM != NULL){
fprintf(tex,"ID=%d & TAM=%d \\\\ \\hline\n",currentM->id,currentM->tama);
currentM = currentM->next;
}
```

Codigo LaTeX Resultante

```
\documentclass[10pt,a4paper]{article}
\usepackage[utf8]{inputenc}
\begin{document}
\begin{center}
\section*{Configuracion}
\begin{description}
\item[Algoritmo:] FIFO
\item[Tamaño total:] 800
\end{description}
\end{center}
\begin{center}
Instante: 0
Memoria Vacia\\
\end{center}

\begin{center}
Instante: 1
Memoria Vacia\\
\end{center}

  ...

Instante: 17

\begin{tabular}{|c|c|}
\hline
ID=10 & TAM=73 \\ \hline
ID=11 & TAM=122 \\ \hline
ID=-1 & TAM=605 \\ \hline
\end{tabular}
\\
\hfill \break
\hfill \break
\hfill \break
En cola
\\
\begin{tabular}{|c|c|}
\hline
ID=12 & TAM=38 \\ \hline
\end{tabular}
\end{center}
\pagebreak
```

```
\begin{center}
```

En 'printEnd()' solamente se imprime en el archivo la etiqueta de cierre de documento necesaria para que el archivo LaTeX compile correctamente.

```
void printEnd(FILE * tex,char* filename){
char* msg="\\end{document}\n";
fputs(msg,tex);
}
```

4. Problemas encontrados

El primer problema encontrado fue el no poder generar los procesos de acuerdo a la distribución de Poisson. No podíamos avanzar en el proyecto dado que sin los procesos generados no se podía simular nada.
Todo esto porque solo calculabamos la probabilidad para el lambda que ingresaba por argumentos de la consola, y no teniamos la noción de la tabla acumulada.

El siguiente código es lo que teníamos para la distribución de Poisson:

```
int kfactorial(int n){
    int fac=1;
    for(int i=n; i>0; i--){
    fac= fac*i;
    }
    return fac;
}
double calculapoisson(int lambda, int k){
    double power= pow(M_E,(-lambda));
    double x= pow(lambda, k);
    int factk= kfactorial(k);
    double result=(power*x)/factk;
    return result;
```

Gracias a que el profesor hizo un ejemplo de cómo implementar la distribución de Poisson, pudimos avanzar en ese aspecto.
Otro de los principales problemas fue la sincronización entre la unidad de tiempo, la hora de llegada y el tiempo de ejecución del procesador, ya que pensabamos que sumando la duración de todos los procesos podíamos calcular el tiempo total de los mismos en memoria, pero dejamos de lado las horas de llegada que hacian que el tiempo total variara con respecto al calculo previamente hecho.
Al principio no sabíamos como indicarle al calendarizador para que no hiciera todo el proceso de un solo, sino que "simulara.^{es}as unidades de tiempo. Eso lo resolvimos con un número contador, y los instantes se simularon con la ayuda de las estructuras de datos de apoyo con las cuales se definió que cada unidad de tiempo era un ciclo en el que se verificaba que las estructuras de datos auxiliares no estuvieran vacías. Resuelto esa parte del problema, teníamos que idear una forma en que conforme esa unidad de tiempo fuera aumentando en el método donde teníamos ese contador, fuera disminuyendo la duración de los procesos en memoria. Este problema lo resolvimos con la función restaDuracion().^{ex}plicada anteriormente.
Otro de los problemas que frecuentemenete tuvimos fueron los segmentation faults. Esto muchas veces ocurría puesto que estábamos utilizando lista enlazadas que en el fondo son punteros. Cuando se utilizaba un puntero qué no había sido asignado o estaba apuntando a NULL, ahí era donde se producía el segmentation fault. Un problema recurrente en la simulación de paginación fue que la simulación terminaba sin que la memoria se limpiara del todo, esto se solucionó cambiando el ciclo principal de instantes y se modificó con condiciones previas al ciclo para que terminara de forma exitosa.
Al principio en paginación se tenía el enfoque de que siempre la estructura de datos para simular la memoria o sea la matriz, sería una cuadrada, misma cantidad de filas y columnas, pero esto era un enfoque erróneo ya que dados distintos números en el archivo de configuración la matriz resultante de eso no sería una matriz cuadrada, así que rápidamente, se hizo un método que factoriza la cantidad de páginas, esto para encontrar los dos factores de este con la diferencia más pequeña entre los dos.

5. Conclusiones

- Se concluye que conforme a lo visto en las simulaciones que los instantes iniciales en todos los modos de sucuencial, son similares, pero que al momento de avanzar instantes, la simulación varia de forma de que se puede notar la características de cada uno de los algoritmos en secuencial.

- Con respecto a paginación y secuencial se puede ver como en paginación al tener la posibilidad de calendarizar en una forma aleatoria en las páginas puede en menor cantidad de instantes calendarizar la misma cantidad de procesos que todos los "modos"de trabajo de la memoria secuencial.

- La implementacion de secuencial es más complicada en cuestión de la simulación ya que la estructura de datos utilizada para simular la memoria es una lista enlazada a la cual se le deben de añadir operaciones para que esta maneje los espacios vacíos en ella.

- Se puede concluir que en el modo secuencial, Best fit y Worst fit consumen más tiempo de procesador que FIFO, dado que en los dos algoritmos anteriores se hace una búsqueda del bloque deseado (ya sea de mayor o de menor tamaño).
- También se puede observar que Best Fit y Worst Fit hacen mejor manejo del espacio de la memoria que FIFO. Sin embargo, se puede observar que Best Fit acomoda mejor el espacio que Worst Fit ya que el primer algoritmo busca el bloque con el espacio justo del proceso a insertar en memoria, mientras que Worst Fit busca el bloque más grande lo que hace que pueda desperdiciar espacio en memoria.

- Se concluye que el manejo de calendarización de los procesos es importante para manejar una mayor multiprogramación, esto importante a nivel de sistemas operativos.