



Instituto Tecnológico de Costa Rica (ITCR)

Escuela de Computación

Curso: Principios de Sistemas Operativos

Código del curso: IC-6600

Grupo 2

Profesor(a): Erika Marín Schumann

Estudiantes:

Marco Antonio Reveiz Rojas (2019053583)

David Solano Fuentes (2018167435)

David Castro Holguin (2018105813)

Índice

Introducción	3
Estrategias de Solución	3
Implementación de Hilos	3
Algoritmos de Planificación de Procesos	3
FIFO	5
Round Robin	5
HPF	7
SJF	8
Implementación de Sockets	9
Servidor	9
Cliente	10
Análisis de Resultados	12
Lecciones aprendidas	13
Casos de pruebas	14
Caso 1	14
Datos del Archivo	14
En el caso de FIFO:	14
Caso 2	15
En el caso de SJF:	15
Datos del Archivo	16
Caso 3	17
En el caso de HPF:	17
Datos del Archivo	17
Caso 4	19
En el caso de Round Robin:	19
Datos del Archivo	19
Comparativa entre los hilos de Java y Pthreads	20
Manual de usuario	21
Servidor	21
Cliente	22
Cliente Manual	24
Cliente Automatico	24
Servidor con Cliente	25
Bitácora de trabajo	26
Bibliografía	28

Introducción

La Unidad Central de Procesamiento (CPU) es la unidad encargada de que todo funcione correctamente, estableciendo las conexiones y realizando todos los cálculos precisos para que funcione. Esto quiere decir, que su principal función es leer, interpretar y procesar las instrucciones primero del sistema operativo. En el CPU existe una sección que se encarga de seleccionar a qué proceso se asigna el recurso procesador y durante cuánto tiempo, llamada Planificador de Procesos. El propósito de este documento, es describir la implementación de un simulador de un planificador de procesos en el CPU basado en el lenguaje de programación C, implementando hilos y sockets.

Estrategias de Solución

En esta sección, se explican las maneras en que se implementaron las soluciones a hilos, los algoritmos de planificación y la implementación de los sockets.

Implementación de Hilos

En las especificaciones del proyecto, se establece que se debe de utilizar la librería pthreads que permite trabajar con distintos hilos en ejecución al mismo instante.

Algoritmos de Planificación de Procesos

El almacenamiento de los procesos, se hace por medio de arrays, simulando una cola. Estos arrays se componen de una estructura que simula un PCB, en la cual se guardan los siguientes datos: el burst, el turnaround time, el waiting time, el pid, y la prioridad. Para esta estructura se diseñó el siguiente código:

```
typedef struct PCB {  
    int PID;  
    int burst;  
    int priority;  
    int tat;  
    int wt;  
} PCB;
```

También se implementaron algunas funciones auxiliares que facilitan el cálculo de los algoritmos. Como la obtención de la cantidad de procesos que tiene el array por medio de las siguiente función:

```
// Auxiliar para obtener la cantidad de procesos en el Queue  
int getQueueSize(PCB queue[100]) {  
    int i = 0;  
    int counter = 0;  
    for (i = 0 ; i < 100 ; i++) {  
        if (queue[i].PID == 0 & queue[i].burst == 0 & queue[i].priority==0){  
            break;  
        }  
        else  
            counter++;  
    }  
    return counter;  
}
```

I. FIFO

Para la implementación del algoritmo de planificación de procesos llamado First in - First Out (FIFO). Se determina la cantidad de procesos que hay en la cola, recibida como parámetro, y simplemente se calcula el waiting time y turn around time por medio de un ciclo for. El código que se implementó para el desarrollo de este algoritmo es:

```
void fifo(PCB readyQueue[100]) {
    // Cantidad de procesos
    int n = getQueueSize(readyQueue);
    int i, j;
    // Waiting Time
    int wt[100];
    // Turn Around Time
    int tat[100];

    printf("process\t\t burst time\t waiting time\t turn around time\n");
    for( i = 0 ; i < n ; i++)
    {
        wt[i] = 0;
        tat[i] = 0;

        for (j=0 ; j < i ; j++)
        {
            wt[i] = wt[i]+readyQueue[j].burst;
        }
        tat[i] = wt[i]+readyQueue[i].burst;
        // Iguala wt y tat en el PCB
        readyQueue[i].wt = wt[i];
        readyQueue[i].tat = tat[i];
        printf("%d\t\t\t%d\t\t\t%d\t\t\t%d\n", readyQueue[i].PID, readyQueue[i].burst, readyQueue[i].wt, readyQueue[i].tat);
    }
}
```

II. Round Robin

Para la implementación del algoritmo de planificación de procesos llamado Round Robin. Se determina la cantidad de procesos que hay en la cola, recibida como parámetro, junto con un número especificado por el usuario. Se realiza un ciclo for para realizar un array con los burst time de los procesos. Luego, entra en un segundo ciclo de while donde se va realizando la operación hasta que se llega al último proceso. El código que se implementó para el desarrollo de este algoritmo es:

```

void roundRobin(int qt,PCB queue[100]) {
    // Cantidad de procesos
    int n = getQueueSize(queue);
    int i;
    int temp;
    // Turn Around Time
    int tat[100];
    // Waiting Time
    int wt[100];
    // Remaining Burst Time
    int rem_bt[100];
    // Counters
    int count = 0;
    // SQ
    int sq = 0;

    // Number of processes for and burst copying
    for (i = 0; i < n ; i++){
        rem_bt[i] = queue[i].burst;
    }

    while (1) {
        for (i = 0, count = 0; i < n ; i++)
        {
            temp = qt;

            if (rem_bt[i] == 0)
            {
                count++;
                continue;
            }

            if (rem_bt[i] > qt)
                rem_bt[i] = rem_bt[i] - qt;
            else
            {
                if (rem_bt[i] >= 0)
                {
                    temp = rem_bt[i];
                    rem_bt[i] = 0;
                }
                sq = sq + temp;
                tat[i] = sq;
            }
        }
        if (n == count)
            break;
    }

    printf("\nprocess\tburst time\tturnaround time\twaiting time\n");

    for ( i = 0; i < n ; i++){
        wt[i] = tat[i] - queue[i].burst;
        printf("\n%d\t\t%d\t\t%d\t\t%d", queue[i].PID,queue[i].burst,tat[i],wt[i]);
    }
}

```

III. HPF

Para la implementación del algoritmo de planificación de procesos llamado Highest Priority First (HPF). Se determina la cantidad de procesos que hay en la cola, recibida como parámetro. Primero se debe de realizar un reordenamiento de la cola recibida, que se realiza con el algoritmo de Bubblesort. Luego una vez se reorganiza la cola, se realiza el cálculo simple para el waiting time y turnaround time. El código que se implementó para el desarrollo de este algoritmo es:

```
void hpf(PCB readyQueue[100]) {
    // Cantidad de procesos
    int n = getQueueSize(readyQueue);
    // Contadores
    int i,j;
    // PCB Temporal
    PCB temp;
    // Waiting Time
    int wt[30];
    // Turn Around Time
    int tat[30];

    // Bubblesort por priority
    for (i = 0; i < n ; i++)
    {
        int pos=i;
        for(j=i+1;j<n;j++)
        {
            if(readyQueue[j].priority < readyQueue[pos].priority)
            {
                pos=j;
            }
        }
        temp=readyQueue[i];
        readyQueue[i]=readyQueue[pos];
        readyQueue[pos]=temp;
    }
    wt[0]=0;
    printf("process \t burst time\t priority \t waiting time \t turn around time\n");

    for (i=0 ; i < n; i++)
    {
        wt[i]=0;
        tat[i]=0;
        for (j=0;j<i;j++)
        {
            wt[i]=wt[i]+readyQueue[j].burst;
        }
        tat[i]=wt[i]+readyQueue[i].burst;
        // Iguala wt y tat en el PCB
        readyQueue[i].wt = wt[i];
        readyQueue[i].tat = tat[i];
        printf("%d\t\t\t %d\t\t\t %d\t\t\t %d\t\t\t %d\n", readyQueue[i].PID, readyQueue[i].burst, readyQueue[i].priority, readyQueue[i].wt, readyQueue[i].tat);
    }
}
```

IV. SJF

Para la implementación del algoritmo de planificación de procesos llamado Shortest Job First (SJF). Se determina la cantidad de procesos que hay en la cola, recibida como parámetro. Luego, se realiza un algoritmo de **Bubble Sort** para ordenar el array de acuerdo a la cantidad de tiempo que dure el proceso. Por último, simplemente se calcula el waiting time y turn around time por medio de un ciclo for. El código que se implementó para el desarrollo de este algoritmo es:

```
void sjf(PCB queue[100]) {
    // Cantidad de procesos
    int n = getQueueSize(queue);
    int i, j;
    PCB temp;
    // Waiting Time
    int wt[100];
    // Turn Around Time
    int tat[100];
    // Applying bubble sort technique to sort according to burst time
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n-i-1; j++)
        {
            if (queue[j].burst > queue[j+1].burst)
            {
                temp = queue[j];
                queue[j] = queue[j+1];
                queue[j+1] = temp;
            }
        }
    }

    printf("process \t burst time \t waiting time \t turn around time\n");

    for (i = 0; i < n; i++)
    {
        wt[i]=0;
        tat[i]=0;
        for (j=0;j<i;j++)
        {
            wt[i] = wt[i]+queue[j].burst;
        }
        tat[i]= wt[i]+queue[i].burst;
        queue[i].tat = tat[i];
        queue[i].wt = wt[i];
        printf("%d\t\t\t %d\t\t\t %d\t\t\t %d\n", queue[i].PID, queue[i].burst, queue[i].wt, queue[i].tat);
    }
}
```

Implementación de Sockets

Los sockets son mecanismos de comunicación entre procesos, es decir permiten que los procesos intercambien datos entre ellos. Los sockets implementados son bajo un esquema de Cliente-Servidor y de Tipo SOCK_STREAM. En el caso del servidor primero crea el socket, asigna IP y número de puerto al socket y después espera hasta que un cliente se conecte. En el caso del cliente primero crea el Socket y después conecta el Socket creado a una dirección IP y puerto especificado.

Servidor

Primero se hizo el include de las librerías necesarias para la creación y uso de los Sockets, las cuales son <sys/types.h>, <sys/socket.h>, <netinet/in.h>, <arpa/inet.h>. Se definen las variables y estructuras del socket (dos estructuras importantes de tipo **Socket Address** para cliente y servidor), además se definen unas variables de tipo char que son las que se utilizan para recibir y enviar mensajes al cliente. Además se hizo la definición de los datos necesarios para la estructura del servidor, como puerto e IP a utilizar.

```
int server_sockfd, client_sockfd; // descriptores de sockets
int server_len, client_len; //tamaños de las estructuras
struct sockaddr_in server_address; //declaracion de estructuras
struct sockaddr_in client_address;
char c[1024]; //cadena del cliente
char ch[1024]; //cadena del servidor
char msg[1024]; //cadena del servidor
int inicio = 0; //determina el inicio de sesion
char cs[1024]; //cadena del servidor
int bufs; //almacenamiento del tamaño cadena server
int ciclo = 1; //variable para ciclo de lectura escritura
int puerto; //variable para el puerto
```

Lo siguiente fue hacer la definición del socket, después se hace un aviso al sistema de que se ha creado un Socket y se establece el Socket en modo escucha.

```

server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
// se llena la estructura para el servidor con los datos necesarios
server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = inet_addr("127.0.0.1");
server_address.sin_port = 5000;
server_len = sizeof(server_address);

//avisamos al sistema operativo la creacion del socket
bind(server_sockfd, (struct sockaddr *)&server_address, server_len);

//decimos al server que quede escuchando
listen(server_sockfd,5);

```

Después se acepta la conexión con el cliente actual y se hace un ciclo While, que es donde se permite al servidor enviar y recibir mensajes, para recibir mensajes se hace uso de la función **recv** y para enviar mensajes se hace uso de la función **send**.

```

while(ciclo){
    //acepta la conexion con el cliente actual
    client_sockfd = accept(server_sockfd, (struct sockaddr *)&client_address, &client_len);
    if(inicio == 0) {
        printf("-----SESION INICIADA-----\n");
        printf("CLIENTE CONECTADO\n");
        strcpy(ch, "SERVIDOR CONECTADO...");
        send(client_sockfd, ch, 1024, 0);
        inicio = 1;
    }
    fflush(stdout);
    recv(client_sockfd, cs, 1024,0);
    processCounter++;

    // Crea el PCB
    PCB pcbTemp;
    pcbTemp.PID = processCounter;
    pcbTemp.burst = cs[0] - '0';
    pcbTemp.priority = cs[2] - '0';

    int lenReadyQueue = getQueueSize(readyQueue);

    readyQueue[lenReadyQueue] = pcbTemp;

    char PID[1024];
    sprintf(PID,"%d",processCounter);
    send(client_sockfd, PID, 1024,0);

    close(client_sockfd);
}

```

Cliente

Al igual que en el servidor, para el caso del cliente, primeramente se hace incluye de las librerías que se necesitan para la creación y uso de los Sockets. Después se hace la definición de la estructura **Socket Address** para los datos del servidor.

```
// VARIABLES GLOBALES
int sockfd;
int len;
struct sockaddr_in address;
int result;
char ch[1024];
char c[1024];
int buf;
int inicio = 0;
char cs[1024];
int bufs;
int ciclo =1;
char ipserver[16] = "127.0.0.1";
int puerto = 5000;
```

Al igual que en el Servidor lo siguiente fue definir el Socket y especificar los datos necesarios de la estructura Socket Address, como el puerto al que se debe conectar y la IP a utilizar.

```
char *msg = (char *)args;
// enviar msg por el socket

sockfd = socket(AF_INET, SOCK_STREAM,0);
//llenado de la estructura de datos
address.sin_family = AF_INET;
address.sin_addr.s_addr = inet_addr(ipserver);
address.sin_port = puerto;
len = sizeof(address);
```

Luego se conecta al servidor y se recibe un mensaje de bienvenida si la conexión fue exitosa, de lo contrario se devuelve un error de conexión.

```
//conectar con el server
result = connect(sockfd, (struct sockaddr *)&address, len);
if(result ==-1){
    perror("ERROR EN LA CONEXION\n");
    close(sockfd);
}

//validar el inicio de sesion
if(inicio==0){
    printf("----- SESION INICIADA ----- \n");
    recv(sockfd, ch, 1024,0);
    printf("%s\n",ch);
    inicio = 1;
}
send(sockfd,msg, 1024,0);

recv(sockfd, c, 1024,0);
printf("Proceso recibido! El PID asignado es el número: %s\n",c);
close(sockfd);
```

Una vez realizado lo anterior, al igual que en el servidor hicimos la creación de un ciclo while, que es donde se permite al cliente enviar y recibir mensajes, para recibir mensajes se hace uso de la función **recv** y para enviar mensajes se hace uso de la función **send**. Solo que en el caso del cliente lo hicimos distinto, con una función **sendData** que es donde está todo lo del Socket.

```
while(1)
{
    sleep(2);
    int randBurst = randNumber(r1->min,r1->max);
    int randPriority = randNumber(1,5);

    con[0] = randBurst + '0';
    con[1] = '|';
    con[2] = randPriority + '0';

    printf("Palabra:  %s",con);

    pthread_create(&pthread,NULL, sendData, (void *)&con);
    pthread_join(pthread,NULL);

    int randNum = randNumber(3,8);
    sleep(randNum);
}
```

Análisis de Resultados

Funcionalidad	Porcentaje de realización	Justificación
Implementación de los algoritmos FIFO, SJF, HPF, RR	100	
Implementación de esquema cliente-servidor	100	
Cliente Manual	100	
Cliente Automatico	100	
Creación y utilización de Sockets	100	
Creación y utilización de Threads	100	
Simulador(Server)	100	
Comunicación entre clientes y simulador(server)	100	

Job Scheduler	100	
CPU Scheduler	100	
Consultar cola Ready	100	
Detener simulación	100	
Desplegar resumen de procesos	100	

Lecciones aprendidas

Durante el desarrollo de este proyecto se lograron refrescar temas vistos en cursos anteriores, como en Estructuras de Datos y Programación Orientada a Objetos. En estos cursos, se vieron temas relacionados con el lenguaje de programación C y la implementación de hilos.

Además se logró comprender de manera más detallada el funcionamiento del planificador de la unidad de procesamiento central en un computador. Mediante la incorporación de los tres hilos en los que se basa el programa. Por su parte también pudimos terminar entendiendo de muy buena forma como se comportan los algoritmos FIFO, SJF, HPF y Round Robin.

Aprender sobre cómo se desarrolla bajo el esquema de Cliente-Servidor ha sido sumamente provechoso, ya que ninguno de los integrantes del grupo había tenido la oportunidad de trabajar con Sockets. Con esto aprendimos cómo se deben implementar los Sockets para el cliente y para el servidor, ha sido importante conocer primeramente como se debe definir la estructura del socket y posterior a eso crear el socket, además de conectar los sockets entre sí y ponerlos en modo escucha de manera ordenada, para que puedan enviar y recibir mensajes tanto cliente como servidor. En esta parte hemos podido resaltar la importancia de definir

el puerto y la IP de buena forma, además nos pareció interesante ver cómo pudimos conectar dos computadoras distintas para que se puedan mandar información entre sí.

Casos de pruebas

Caso 1

Datos del Archivo

```
BURST;Prioridad
5;3
6;2
7;5
7;2
1;3
```

En el caso de FIFO:

Menu Cliente Manual:

```
Bienvenido al menú del cliente
1. Cliente manual.
2. Cliente automático
3. Salir.

Introduzca opción que desee (1-3): 1
----- SESION INICIADA -----
SERVIDOR CONECTADO...
Proceso recibido! El PID asignado es el número: 1
Proceso recibido! El PID asignado es el número: 2
Proceso recibido! El PID asignado es el número: 3
Proceso recibido! El PID asignado es el número: 4
Proceso recibido! El PID asignado es el número: 5
```

Menú del Servidor:

```

Bienvenido al menú del servidor
1. FIFO.
2. SJF.
3. HPF.
4. RR.
5. Salir.

  Seleccione el algoritmo que desee (1-4): 1
  -----SESION INICIADA-----
  CLIENTE CONECTADO
  Proceso 1 con burst 5 entra en ejecución
  El proceso con ID: 1, ha finalizado

  Proceso 2 con burst 6 entra en ejecución
  El proceso con ID: 2, ha finalizado

  Proceso 3 con burst 7 entra en ejecución
  El proceso con ID: 3, ha finalizado

  Proceso 4 con burst 7 entra en ejecución
  El proceso con ID: 4, ha finalizado

  Proceso 5 con burst 1 entra en ejecución
  El proceso con ID: 5, ha finalizado

```

Resumen de Ejecución:

```

--- Resumen de ejecución ---
Cantidad de procesos ejecutados : 5
Cantidad de segundos con CPU ocioso: 15

```

PID	Burst Time	Waiting Time	Turn Around Time
1	5	0	5
2	30	0	40
3	1	23	41
4	6	16	47
5	5	17	52

```

Promedio de Waiting Time: 11.200000
Promedio de Turn Around Time: 37.000000

```

Caso 2

En el caso de SJF:

Datos del Archivo

```
BURST;Prioridad
5;4
30;5
1;1
6;2
5;1
```

Menu Cliente Manual:

```
Bienvenido al menú del cliente
1. Cliente manual.
2. Cliente automático
3. Salir.

Introduzca opción que desee (1-3): 1
----- SESION INICIADA -----
SERVIDOR CONECTADO...
Proceso recibido! El PID asignado es el número: 1
Proceso recibido! El PID asignado es el número: 2
Proceso recibido! El PID asignado es el número: 3
Proceso recibido! El PID asignado es el número: 4
Proceso recibido! El PID asignado es el número: 5
```

Cola de Ready segun el recibimiento de los procesos

Mientras se ejecuta el proceso con PID de 2, que dura 30 segundos llegan los procesos con PID 3, 4, y 5 respectivamente.

PID	Burst Time	Prioridad
2	30	5
3	1	2
4	7	3
5	5	11

Luego se hace un bubblesort cuando se termina de ejecutar el proceso con PID 2, y se transformaría la cola de Ready de la siguiente manera:

PID	Burst Time	Prioridad
3	1	2
5	5	11
4	7	3

Por último, se termina de ejecutar el algoritmo, ya que no llegan más procesos al “CPU”, y el resumen de ejecución concluirá de la siguiente manera:


```

--- Resumen de ejecución ---
Cantidad de procesos ejecutados : 5
Cantidad de segundos con CPU ocioso: 15

PID          Burst Time      Waiting Time      Turn Around Time
1             5              0                5
2            30              0               40
3             1              25              41
5             5              11              46
4             6              21              52

Promedio de Waiting Time: 11.400000
Promedio de Turn Around Time: 36.799999

```

Caso 3

En el caso de HPF:

Datos del Archivo

```

BURST;Prioridad
5;4
30;5
1;1
6;2
5;1

```

Menu Cliente Manual:

```

Bienvenido al menú del cliente
1. Cliente manual.
2. Cliente automático
3. Salir.

Introduzca opción que desee (1-3): 1
----- SESION INICIADA -----
SERVIDOR CONECTADO...
Proceso recibido! El PID asignado es el número: 1
Proceso recibido! El PID asignado es el número: 2
Proceso recibido! El PID asignado es el número: 3
Proceso recibido! El PID asignado es el número: 4
Proceso recibido! El PID asignado es el número: 5

```

Cola de Ready segun el recibimiento de los procesos

Mientras se ejecuta el proceso con PID de 2, que dura 30 segundos llegan los procesos con PID 3, 4, y 5 respectivamente. Si se imprime la cola de Ready se puede observar el ordenamiento de los procesos mediante otro algoritmo de bubblesort.

Cola de Ready previo al bubble sort

PID	Burst Time	Prioridad
2	30	5
3	1	1
4	6	2
5	5	1

Cola de Ready luego de aplicar el bubble sort

PID	Burst Time	Prioridad
2	30	5
3	1	1
4	6	2
5	5	1

Por último, se termina de ejecutar el algoritmo, ya que no llegan más procesos al “CPU”, y el resumen de ejecución concluirá de la siguiente manera:

```
--- Resumen de ejecución ---
Cantidad de procesos ejecutados : 5
Cantidad de segundos con CPU ocioso: 17

PID          Burst Time    Waiting Time    Turn Around Time
1            5              0               5
2            30              0              36
3            1              23             37
5            5              7              42
4            6              20             48

Promedio de Waiting Time: 10.000000
Promedio de Turn Around Time: 33.599998
```

Caso 4

En el caso de Round Robin:

Datos del Archivo

```
BURST;Prioridad
5;4
30;5
1;1
6;2
5;1
```

Este algoritmo se ejecutó con un quantum de 3.

Menu Cliente Manual:

```
Bienvenido al menú del cliente
1. Cliente manual.
2. Cliente automático
3. Salir.

Introduzca opción que desee (1-3): 1
----- SESION INICIADA -----
SERVIDOR CONECTADO...
Proceso recibido! El PID asignado es el número: 1
Proceso recibido! El PID asignado es el número: 2
Proceso recibido! El PID asignado es el número: 3
Proceso recibido! El PID asignado es el número: 4
Proceso recibido! El PID asignado es el número: 5
```

El funcionamiento de este algoritmo fue bastante complicado, ya que se debía manipular el burst time de acuerdo al tiempo real. Y a partir de ahí se tenía que ir modificando las posiciones de la cola de Ready, junto con los burst de cada PCB. En este caso, se ejecuta el primer proceso con PID, y luego llega el segundo con un segundo de tiempo ocioso, y así van llegando hasta completar los procesos del archivo. Concluyendo con el siguiente resumen de ejecución.

```
--- Resumen de ejecución ---
Cantidad de procesos ejecutados : 5
Cantidad de segundos con CPU ocioso: 27

PID          Burst Time      Waiting Time      Turn Around Time
1             5              0                5
2            30             17              47
3             1              3              20
4             6             32              38
5             5             44              49

Promedio de Waiting Time: 19.200001
Promedio de Turn Around Time: 31.799999
```

Comparativa entre los hilos de Java y Pthreads

Desde un inicio se sabe que la programación de hilos y multihilos es compleja, a pesar de que el concepto de hilos es el mismo tanto en Java como en Pthreads, al final de este proyecto hemos llegado a la conclusión de que hay diferencias importantes entre uno y otro. A continuación haremos una comparativa entre los hilos de Java y Pthreads desde las funcionalidades de Pthreads conocidas y utilizadas en la creación de este proyecto.

En el caso de Java para crear un hilo se debe instanciar un **objeto thread**, mientras que en el caso de Pthreads se crean con una llamada a la función **pthread_create()** y se le pasa las funciones específicas que serán ejecutadas por el hilo creado. En Pthreads los parámetros del nuevo hilo están limitados a uno, mientras que en Java ningún límite para la cantidad de parámetros que se le pasen al método **run()**, el cual es el método que empieza a ejecutar los hilos en java.

En Pthreads al crear un hilo este arranca automáticamente, para el caso de nuestro proyecto, primeramente tuvimos que crear los hilos y después utilizar una función **join** para que los hilos se sincronicen juntos entre sí. En Java esto es distinto ya que cuando se crea un hilo, este no arranca automáticamente, para que arranque hay que ejecutar la función **start()**. Los de Java en este caso tienen la

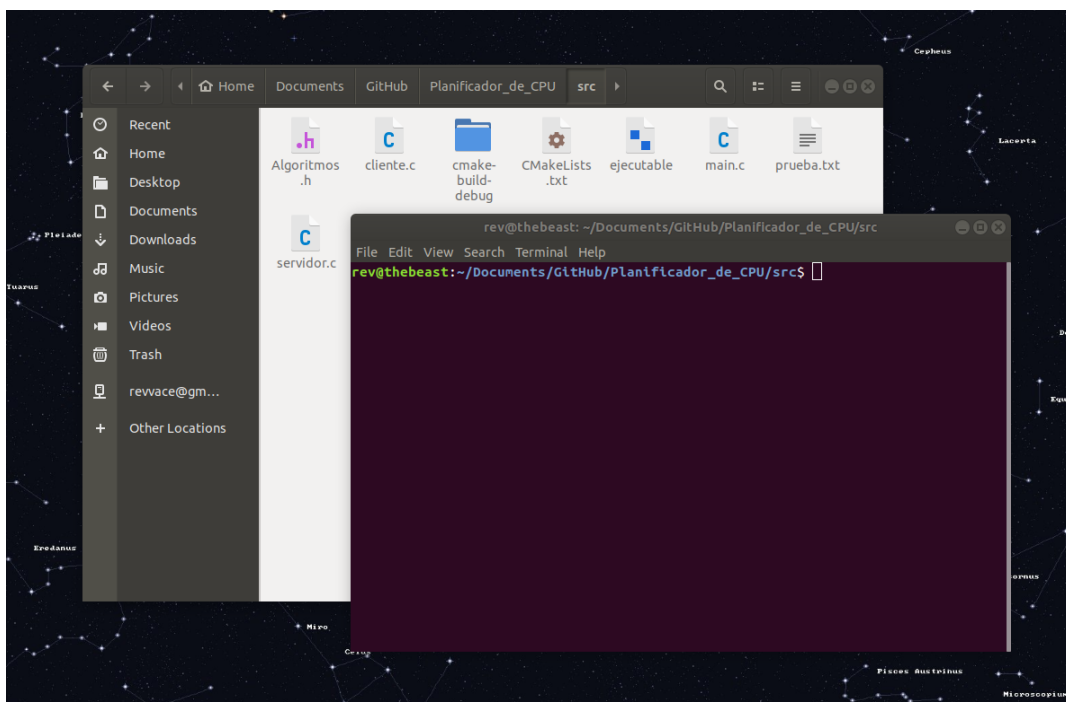
ventaja de que se pueden utilizar posteriormente donde se necesite (por que no arrancan automáticamente), pero en Pthreads donde se crean ahí se deben ejecutar, en otras palabras se deben crear en donde realmente se necesiten.

Pthreads tiene varias funciones que pueden detener un mientras que Java solo tiene un método implementado para esto, Pthreads tiene las funciones **pthread_exit()**, **pthread_kill()** y **pthread_cancel()**. Mientras que la de Java es **stop()**.

En conclusión ambos son útiles y adecuados para el desarrollo de hilos, siempre y cuando se saque el máximo provecho a cada uno, sin embargo, creemos que Java es un poco más sencillo por la flexibilidad que brinda.

Manual de usuario

Primero que todo se debe abrir la terminal, en el directorio donde están los programas. En este caso, se abre la terminal en la carpeta “src”.



Luego se tienen los siguientes comandos para compilar y ejecutar los programas. Al ser una aplicación cliente-servidor, primero se debe de compilar y correr el servidor.

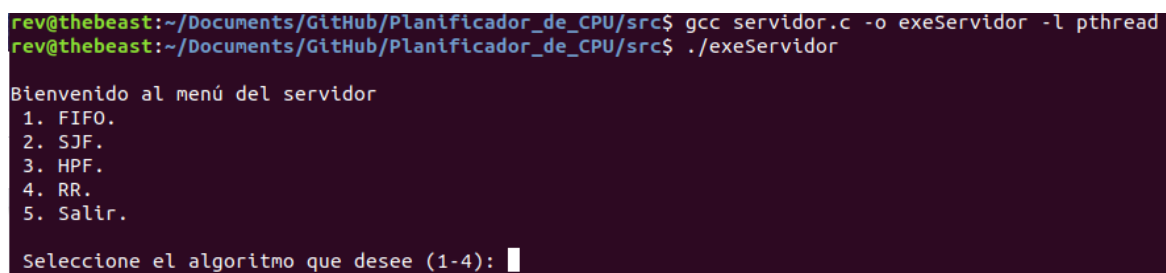
Servidor

Código para compilar, el cual crea un archivo ejecutable:

```
gcc servidor.c -o exeServidor -l pthread
```

Código para ejecutar el código compilado mediante el archivo creado anteriormente:

```
./exeServidor
```

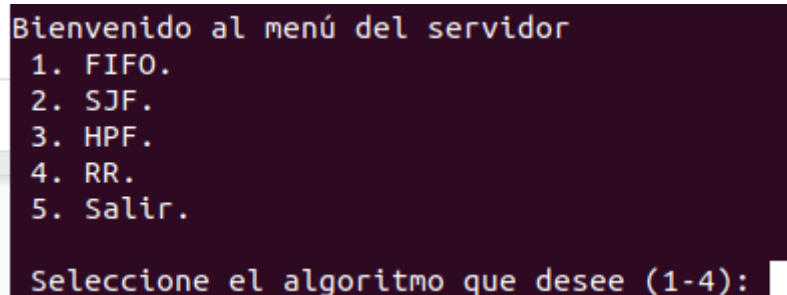


```
rev@thebeast:~/Documents/GitHub/Planificador_de_CPU/src$ gcc servidor.c -o exeServidor -l pthread
rev@thebeast:~/Documents/GitHub/Planificador_de_CPU/src$ ./exeServidor

Bienvenido al menú del servidor
1. FIFO.
2. SJF.
3. HPF.
4. RR.
5. Salir.

Seleccione el algoritmo que desee (1-4):
```

Una vez se ejecutan ambos, se obtiene el menú principal del servidor. En donde hay disponibles cuatro opciones numeradas del 1 al 4. Estas opciones corresponden al tipo de algoritmo de planificación que se implementará en el simulador siendo el número uno el First in - First Out (FIFO), el número dos el Shortest Job First (SJF), el número tres el Highest Priority First (HPF) y el número cuatro el Round Robin.



```
Bienvenido al menú del servidor
1. FIFO.
2. SJF.
3. HPF.
4. RR.
5. Salir.

Seleccione el algoritmo que desee (1-4):
```

Cliente

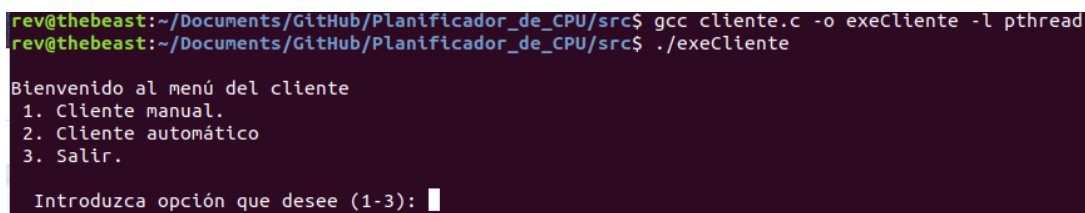
Al igual que en el servidor, se debe realizar un proceso similar para el cliente.

Código para compilar, el cual crea un archivo ejecutable:

```
gcc cliente.c -o exeCliente -l pthread
```

Código para ejecutar el código compilado mediante el archivo creado anteriormente:

```
./exeCliente
```

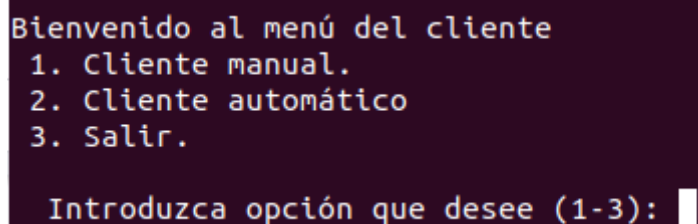


```
rev@thebeast:~/Documents/GitHub/Planificador_de_CPU/src$ gcc cliente.c -o exeCliente -l pthread
rev@thebeast:~/Documents/GitHub/Planificador_de_CPU/src$ ./exeCliente

Bienvenido al menú del cliente
1. Cliente manual.
2. Cliente automático
3. Salir.

Introduzca opción que desee (1-3):
```

Una vez se ejecutan ambos, se obtiene el menú principal del cliente. En donde hay disponibles dos opciones numeradas del 1 al 2. Estas opciones corresponden al tipo de cliente que desea ejecutar. Siendo el número del cliente manual y el número dos el cliente automático.



```
Bienvenido al menú del cliente
1. Cliente manual.
2. Cliente automático
3. Salir.

Introduzca opción que desee (1-3):
```

Una vez se indica que el cliente que se desea ejecutar, se muestra un anuncio de éxito de conexión al servidor.

```
Bienvenido al menú del cliente
1. Cliente manual.
2. Cliente automático
3. Salir.

Introduzca opción que desee (1-3): 1
----- SESION INICIADA -----
```

Cliente Manual

Dentro del cliente Manual se pueden ver los mensajes, de cuando un proceso es recibido y se le asigna un PID único.

```
Bienvenido al menú del cliente
1. Cliente manual.
2. Cliente automático
3. Salir.

Introduzca opción que desee (1-3): 1
----- SESION INICIADA -----
SERVIDOR CONECTADO...
Proceso recibido! El PID asignado es el número: 1
Proceso recibido! El PID asignado es el número: 2
Proceso recibido! El PID asignado es el número: 3
Proceso recibido! El PID asignado es el número: 4
```

Cliente Automatico

Dentro del cliente Automático se pueden ver los mensajes, de cuando un proceso es recibido y se le asigna un PID único.

```
Bienvenido al menú del cliente
1. Cliente manual.
2. Cliente automático
3. Salir.

Introduzca opción que desee (1-3): 2

Ingrese el valor minimo y maximo del rango, separados por un espacio: 3 5
----- SESION INICIADA -----
SERVIDOR CONECTADO...
Proceso recibido! El PID asignado es el número: 1
Proceso recibido! El PID asignado es el número: 2
```

Servidor con Cliente

```
Bienvenido al menú del cliente
1. Cliente manual.
2. Cliente automático
3. Salir.

Introduzca opción que desee (1-3): 2

Ingrese el valor minimo y maximo del rango, separados por un espacio: 3 5
----- SESION INICIADA -----
SERVIDOR CONECTADO...
Proceso recibido! El PID asignado es el número: 1
Proceso recibido! El PID asignado es el número: 2
```

Mientras se vaya ejecutando el algoritmo de planificación se puede ir viendo su proceso en la terminal del servidor.

```
Seleccione el algoritmo que desee (1-4): 1
-----SESION INICIADA-----
CLIENTE CONECTADO
Proceso 1 con burst 1 entra en ejecución
El proceso con ID: 1, ha finalizado

Proceso 2 con burst 9 entra en ejecución
El proceso con ID: 2, ha finalizado

Proceso 3 con burst 1 entra en ejecución
El proceso con ID: 3, ha finalizado

Proceso 4 con burst 2 entra en ejecución
El proceso con ID: 4, ha finalizado
```

También la terminal de servidor mientras se tenga un cliente ejecutando, se tienen las siguientes opciones:

-
1. Si el usuario hace click en la tecla “c”. Podrá ver un print de la cola de Ready, con los procesos que tenga dentro, si tiene.

```
-----  
PID                Burst Time          Prioridad  
2                  9                  5  
3                  1                  5  
-----
```

2. Una vez terminado el algoritmo de planificación se puede consultar por medio de un click en la tecla “d”. Esto, imprimirá un resumen de la ejecución, y finalizará el programa.

```
--- Resumen de ejecución ---  
Cantidad de procesos ejecutados : 4  
Cantidad de segundos con CPU ocioso: 26  
  
PID                Burst Time          Waiting Time          Turn Around Time  
1                  1                  0                  1  
2                  9                  0                  9  
3                  1                  0                  1  
4                  2                  0                  2  
  
Promedio de Waiting Time: 0.000000  
Promedio de Turn Around Time: 3.250000
```

Bitácora de trabajo

Se debe recalcar que para esta sección, no se realizaron consultas ni al asistente ni a la profesora a lo largo del desarrollo del proyecto.

Actividad	Fecha
Se realizó una reunión con el fin de evaluar las herramientas en las que se va a desarrollar el proyecto.	2 de abril 2022
Se repartieron los temas para investigar del desarrollo del proyecto.	5 de abril 2022

Se realizó una reunión para determinar la implementación de los hilos con el uso de la librería de pthreads en C.	6 de abril 2022
Se terminó el desarrollo de los algoritmos de planificación en C.	7 de abril 2022
Se terminó la investigación sobre la implementación de sockets en el proyecto.	8 de abril 2022
Se realizó una reunión para el desarrollo de la sincronización de los hilos y los sockets. Y se empezó a realizar la documentación.	9 de abril 2022
Se realizó la última reunión para implementar los algoritmos de planificación al proyecto final y terminar la documentación.	10 de abril 2022

Bibliografía

C language tutorial => calling a function from another c file. (2022). RIP Tutorial.

<https://riptutorial.com/c/example/3250/calling-a-function-from-another-c-file>

Fcfs program in c. (2018, 3 noviembre). [Vídeo]. YouTube.

<https://www.youtube.com/watch?v=dDBIBZMh0ic&t=0s>

Round robin scheduling program in c. (2020, 7 abril). [Vídeo]. YouTube.

https://www.youtube.com/watch?v=N0ET__T4sc0

Sjf program in c. (2018, 23 noviembre). [Video]. YouTube.

<https://www.youtube.com/watch?v=Z70bNxa2870>

Sockets en Linux Cliente-Servidor. (2020, 5 septiembre).[Video]. YouTube.

<https://youtu.be/AfDfmsh8OG0>

Sockets TCP, Linux, C. (2021, 21 enero). [Video]. YouTube.

<https://youtu.be/zFHjKCVwS48>