Escuela de Ingeniería en Computación

Principios de Sistemas Operativos - IC6600

Planificador de CPU

Equipo de Proyecto:

Andres Masis Rojas 2020127158

Juan Sebastián Gamboa Botero 2020030303

Leonardo David Fariña Ozamis 2020045272

Nikholas Ocampo Fuentes 2020061243

Profesor:

Erika Marín Schumann

I Semestre del 2023

Fecha: 03 de Abril, 2023

Introducción

El uso y buen manejo del Job y CPU Scheduler son fundamentales en el desarrollo de sistemas operativos y aplicaciones que requieren la ejecución de procesos. Estos planificadores de procesos son responsables de determinar el orden en el que los procesos deben ser ejecutados y cuánto tiempo deben permanecer en el CPU.

La importancia del uso y buen manejo del Job y CPU Scheduler radica en que el rendimiento y la eficiencia de un sistema operativo o aplicación dependen en gran medida de cómo se gestionan los procesos. Si los procesos se planifican de manera incorrecta, pueden producirse cuellos de botella y pérdida de rendimiento. Además, un mal manejo del planificador puede provocar bloqueos, fallos y errores en el sistema.

El Job Scheduler es responsable de recibir los procesos y ponerlos en cola de espera, mientras que el CPU Scheduler se encarga de seleccionar qué proceso debe ejecutarse en la CPU. Es importante que ambos planificadores trabajen de manera eficiente y coordinada, y que se utilice el algoritmo de planificación adecuado para el tipo de aplicación y procesos que se están ejecutando.

Por eso, en este proyecto vamos a desarrollar un simulador de planificador de CPU utilizando los algoritmos de planificación FIFO, SJF, HPF y Round Robin con un quantum específico definido por el usuario. El proyecto también incluirá un esquema cliente-servidor para enviar información de los procesos a ejecutar mediante sockets.

Este proyecto se enfoca en mejorar las habilidades de programación en C y en el uso de herramientas como sockets y pthreads, así como en la comprensión de los algoritmos de planificación de CPU y su aplicación en un entorno cliente-servidor.

Estrategia de Solución

El presente proyecto ha sido diseñado para simular un planificador de CPU en el sistema operativo Linux, en el cual se han implementado varios algoritmos de planificación como FIFO, SJF, HPF y Round Robin con un quantum especificado por el usuario. El proyecto se ha desarrollado en un esquema cliente-servidor, donde el servidor recibe la información de los procesos a ejecutar mediante sockets y los pone en una cola de espera para su posterior ejecución.

Para la solución del proyecto primero debemos empezar por las estructuras creadas para luego ir a los procesos grandes, ya que estas estructuras confirman dichos procesos.

La primera estructura será la del Job Scheduler, que esta utiliza la estructura jobscheduler.h (anexo 1), donde esta estructura tiene el PCB con su, pid, burst, priority, starTime, entre otros datos importante que puede contar el PCB. Además cuenta con dos estructuras iguales, una para el ReadyQueue y otra para los FinishQueue.
Algunas funciones en las que esta estructura se usan serían de crear, mover, eliminar jobs, así como obtener el job más corto, el de mayor prioridad y algunas funciones de impresión.

Luego está el programa que se encarga del CPU Scheduler (anexo 2), en donde se crea cada uno de los algoritmos de planificación, esto nos indica que aca esta el algoritmo de FIFO, SJF, HPF, RR, es acá donde se hacen ciclos, movimientos de burst, ejecución de los jobs, obtención del Ready Queue entre otros procesos.

También se definen estructuras de cómo funciona un Job (anexo 3), que todo job tiene que tener un burst y un priority, esto es muy útil para cuando se leen archivos poder definir los jobs, este struct se utiliza para el envío del job de los clientes al servidor.

Ahora toca explicar la solución al problema, en donde se llaman y hacen uso estas estructuras. Empezando por el archivo server.c (anexo 4), este lo que hace es crear

los sockets en donde se va a conectar los clientes, y estar pendiente de si algún cliente está intentando conectarse al servidor.
Este servidor, además de manejar los diferentes hilos de las conexiones, maneja todo tipo de comandos, como para ver la queue, detener y reiniciar los procesos, e imprimir toda la tabla de datos necesarios para analizar el correcto funcionamiento, como observar el TAT o el WT de los jobs.

Pero en resumen lo que realiza es llamar al algoritmo de planificación que se solicita como parámetro ponerlo a correr, manejar las conexiones de los clientes para mandar al ready queue, llevar un temporizador TIMESF el cual simula los tiempos del cpu y manejar la interfaz de usuario como los comandos que se reciben y los mensajes que se muestran.

Se cuentan con dos tipos de clientes, estos son específicos en sus tareas, podemos explicar primero el cliente automático (anexo 6). Este cliente se conecta al servidor, crea jobs, pero estos jobs son creados a partir de un parámetro o dos que recibe cuando se ejecuta el código.
Estos parámetros son, la cantidad de ratio que tiene los jobs para ser creados y el máximo de burst que pueden llegar a tener los jobs, a partir de acá el servidor llama a su función según fue ejecutado y va recibiendo jobs hasta que el usuario detenga esta función.

A su vez, tenemos un cliente manual (anexo 7), que hace la misma función, conectarse al servidor y mandar jobs, la diferencia es que este cliente recibe como parámetro un nombre de archivo el cual lo lee, ejecutando los jobs que ya tienen definido un burst junto a una prioridad.

Estos clientes a su vez van recibiendo notificaciones de la creación de estos jobs y el estatus de estos por parte del servidor.

Con esto podemos concluir las ideas principales de la solución del problema, haciendo uso de estructuras, hilos, sockets, clientes y lectura de archivos, todo esto utilizado de una manera eficiente y eficaz.

Análisis de Resultados

Después de haber desarrollado el simulador de planificador de CPU con los diferentes algoritmos y la implementación del cliente-servidor con sockets en C utilizando pthreads, se realizaron pruebas exhaustivas para analizar su rendimiento.

Para evaluar la eficiencia del planificador de CPU se realizaron pruebas en diferentes configuraciones de CPU, memoria y en diferentes situaciones de carga. Se midieron la utilización de CPU, el tiempo de respuesta de los procesos y la cantidad de procesos atendidos correctamente.

En general, se observó que el planificador de CPU funcionó correctamente y proporcionó resultados precisos. Los diferentes algoritmos de planificación funcionaron de acuerdo a lo esperado y mostraron resultados coherentes en relación a los procesos.

En cuanto al rendimiento, se observó que el planificador de CPU funcionó de manera eficiente incluso en situaciones de alta carga, gracias a la implementación de hilos con pthreads que permitió una gestión óptima de los procesos en tiempo real.

Por último, se observó que el protocolo de comunicación de los sockets funcionó de manera adecuada, permitiendo la conexión entre los diferentes clientes y el servidor sin problemas de conectividad.

A continuación se presentan los análisis de resultados según lo indicado en la descripción detallada del proyecto:

1. Cliente Manual: Recibe de un archivo con una lista de procesos : 100%
2. Cliente Automático: Recibe valores de Burst y Tasa de creación : 100%
3. Notificación de recibido por parte del server : 100%

4. Comunicación: Por medio de sockets : 100%
5. Simulador (Server) : 100%
    a. Job Scheduler : 100%
    b. CPU Scheduler : 100%
6. Documentación : 100 %

Lecciones aprendidas

Este proyecto aportó beneficios a la formación como ingenieros de todos los miembros del equipo. Hubo aprendizaje técnico donde se profundizó lo estudiado en el aula. Se complementa la teoría dada por la profesora con fuentes en Internet y además se puso en práctica todo este conocimiento, lo que ayudó a reforzar aún más. También se trabajan habilidades blandas tanto intrapersonales como interpersonales.

Se mejoró el trabajo en equipo y la organización de cada miembro. Con respecto a las metodologías de trabajo, se probó aplicar la delegación. Esto significa que a cada miembro se le asignaba un avance y este debió compartirlo con el resto del grupo apenas lo terminara. Es importante que esto se aplique solo para tareas más sencillas como elementos de la documentación. Para tareas más complejas se dio un trabajo entre todos los miembros donde todos aportan ideas y se ayudaban mutuamente. Por ejemplo, en la creación del CPU Scheduler y el manejo de los diferentes algoritmos como FIFO, SJF, HPF y Round Robin.

Adicionalmente a esto, también se aprenden lecciones técnicas como el manejo adecuado de recursos, ya que la implementación debe asegurar un manejo adecuado de los recursos del sistema, como los procesos y los sockets. Si estos no se gestionan adecuadamente, pueden surgir problemas de rendimiento.

Tanto como en la parte de planificación y diseño se hizo una comprensión clara de los requisitos, los algoritmos de planificación de CPU, los hilos y la comunicación de sockets antes de comenzar la implementación, junto con pruebas exhaustivas ya

son necesarias para detectar y corregir errores en el sistema antes de desplegarlo en producción.

Para este proyecto se dio un uso intensivo de medios de comunicación digitales como WhatsApp o Discord. Debido a los horarios y actividades personales de cada miembro, fue muy difícil que todos estuvieran disponibles en el mismo momento y en el mismo lugar. La ventaja es que se fortaleció el uso de estas herramientas que cada vez son más comunes en el ámbito profesional, como el sistema de control de versiones GitHub y el uso de las ramas de trabajo individual y grupal. Adicionalmente, siempre se procuró una participación de todos los miembros y la colaboración a pesar de ser medios virtuales.

Afortunadamente no se tuvieron problemas a lo interno del grupo. Solo se fortalecieron habilidades como la cooperación, trabajo en equipo y organización de tareas. Estas son características muy importantes para satisfacer el mercado laboral.

Por otro lado, este proyecto fue especialmente beneficioso para reforzar los conceptos vistos en clase donde todos los miembros pudimos aplicar estos conceptos y técnicas aprendidas.

En términos generales la comunicación y trabajo en equipo siempre fue presente ya que nos apoyamos de buenas herramientas que impulsan esta práctica como Kanban, ya que es importante visualizar el trabajo en todo momento. Kanban permite a los equipos ver el estado actual de los proyectos en tiempo real, lo que ayuda a identificar cuellos de botella y problemas de flujo de trabajo. Esto puede ayudar a los equipos a priorizar tareas y a asegurarse de que se están enfocando en las áreas más críticas del proyecto.

Casos de pruebas

En este contexto, se deben realizar pruebas para los distintos componentes del sistema, incluyendo el servidor, el cliente manual, el cliente automático y los algoritmos de planificación FIFO, SJF, HPF y RR. Para cada uno de estos componentes se deben diseñar casos de prueba que permitan verificar su correcto

funcionamiento, asegurando que los resultados obtenidos sean los esperados y que el sistema cumpla con los requisitos establecidos en la descripción del proyecto.

Los casos de prueba deben ser diseñados de manera exhaustiva y sistemática, cubriendo diferentes escenarios y posibles situaciones en las que el sistema podría fallar. De esta manera, se asegura que el sistema funcione correctamente en todo momento y se minimiza el riesgo de errores o fallos inesperados. En el siguiente apartado, se presentarán los casos de prueba diseñados para cada uno de los componentes del sistema.

Cliente Automatico

Se prueba el cliente automático para una tasa de creación de 3 y un burst máximo de 3:

Jobs enviados por el cliente

```
./automatic_client 3 3
/ / / Client started / / /
Job sent - burst: 2 - priority: 4
Job recv - pid: 1
Job sent - burst: 3 - priority: 4
Job recv - pid: 2
Job sent - burst: 2 - priority: 1
Job recv - pid: 3
Job sent - burst: 1 - priority: 5
Job recv - pid: 4
Job sent - burst: 1 - priority: 3
Job recv - pid: 5
Job sent - burst: 1 - priority: 4
Job recv - pid: 6
Job sent - burst: 3 - priority: 2
Job recv - pid: 7
Job sent - burst: 1 - priority: 2
Job recv - pid: 8
Job sent - burst: 1 - priority: 2
Job recv - pid: 9
Job sent - burst: 3 - priority: 1
Job recv - pid: 10
Job sent - burst: 1 - priority: 3
Job recv - pid: 11
```

```
Job sent - burst: 2 - priority: 1
Job recv - pid: 12
Job sent - burst: 1 - priority: 5
Job sent - burst: 3 - priority: 3
Job sent - burst: 1 - priority: 5
Job sent - burst: 1 - priority: 2
```

Podemos analizar la enviada de datos constante de parte del cliente, como tiene un constante envío gracias a la baja tasa de creación que se mandó, también se puede observar que el burst de cada job no sobrepasa el número 3 y por último se puede ver que el cliente sigue enviando jobs aunque el servidor esté detenido.

El server responde:

```
./server FIFO

[JS] PID: 0 at 10
[CPU]: PID 0 start at 10
                                                    [JS] PID: 1 at 12
[CPU]: PID 0 finished at 12
[CPU]: PID 1 start at 12
                                                    [JS] PID: 2 at 14
[CPU]: PID 1 finished at 15
[CPU]: PID 2 start at 15
                                                    [JS] PID: 3 at 17
[CPU]: PID 2 finished at 17
[CPU]: PID 3 start at 17
                                                    [JS] PID: 4 at 18
[CPU]: PID 3 finished at 18
[CPU]: PID 4 start at 18
[CPU]: PID 4 finished at 19
                                                    [JS] PID: 5 at 20

[CPU]: PID 5 start at 20
[CPU]: PID 5 finished at 21
                                                    [JS] PID: 6 at 23
[CPU]: PID 6 start at 23
                                                    [JS] PID: 7 at 25
                                                    [JS] PID: 8 at 26
[CPU]: PID 6 finished at 26
[CPU]: PID 7 start at 26
[CPU]: PID 7 finished at 27
[CPU]: PID 8 start at 27
[CPU]: PID 8 finished at 28
                                                    [JS] PID: 9 at 29
[CPU]: PID 9 start at 29
                                                    [JS] PID: 10 at 31
[CPU]: PID 9 finished at 32
[CPU]: PID 10 start at 32
```

```
[JS] PID: 11 at 32
```

Como respuesta del servidor se puede ver el tiempo de llegada de cada job, importante recordar que este valor de llegada siempre es random, además de eso se puede ver el PIB de cada job junto con su tiempo de inicio y tiempo de finalización.

Para este ejemplo se aplicaron dos comandos, el freeze para detener la ejecución de procesos y el stop, para mostrar la tabla de procesos

```
Server: freeze
Server: stop

Stop CPU:
# of jobs executed: 10
Time in secs of idle CPU: 14
Job: 0, TAT: 2, WT: 0, B: 2, P: 4, AT: 10, ET: 12
Job: 1, TAT: 3, WT: 0, B: 3, P: 4, AT: 12, ET: 15
Job: 2, TAT: 3, WT: 1, B: 2, P: 1, AT: 14, ET: 17
Job: 3, TAT: 1, WT: 0, B: 1, P: 5, AT: 17, ET: 18
Job: 4, TAT: 1, WT: 0, B: 1, P: 3, AT: 18, ET: 19
Job: 5, TAT: 1, WT: 0, B: 1, P: 4, AT: 20, ET: 21
Job: 6, TAT: 3, WT: 0, B: 3, P: 2, AT: 23, ET: 26
Job: 7, TAT: 2, WT: 1, B: 1, P: 2, AT: 25, ET: 27
Job: 8, TAT: 2, WT: 1, B: 1, P: 2, AT: 26, ET: 28
Job: 9, TAT: 3, WT: 0, B: 3, P: 1, AT: 29, ET: 32
------------------
AVG. WAT: 0
AVG. TAT: 2
```

Aca podemos observar la cantidad de jobs ejecutados, el tiempo que estuvo el CPU ociosa, además de cada Turn around time(TAT), waiting time (WT), Burst (B), Priority(P), Tiempo de llegada (AT) y tiempo de finalización (ET), de los jobs específicos, y al final de la tabla se puede ver el promedio del Waiting Time y Turn Around Time de los jobs.

Concluyendo así que el cliente automático tiene un funcionamiento correcto como se puede observar que se fueron ejecutando en su orden de llegada.

Cliente Manual

A continuación se presenta el cliente manual y el servidor en cada uno de los diferentes algoritmos, para esto se crean diferentes archivos de texto con diferentes procesos, con el fin de poder ver la información más precisa.

FIFO

Archivo de prueba:

```
BURST Prioridad
3 5
3 3
24 1
10 2
2 4
```

Respuesta del cliente:

```
./manual_client FIFO.txt

/ / / Client started / / /
Job sent - burst: 3 - priority: 5
Job recv - pid: 1
Job sent - burst: 3 - priority: 3
Job recv - pid: 2
Job sent - burst: 24 - priority: 1
Job recv - pid: 3
Job sent - burst: 10 - priority: 2
Job recv - pid: 4
Job sent - burst: 2 - priority: 4
Job recv - pid: 5
```

Nuevamente podemos analizar la enviada de datos constante de parte del cliente hasta que se completen todos los procesos que se crearon con el archivo.

Respuesta del Servidor:

```
Server: FIFO

                                                        [JS] PID: 0 at 11
[CPU]: PID 0 start at 11
[CPU]: PID 0 finished at 14
                                                        [JS] PID: 1 at 18
[CPU]: PID 1 start at 18
[CPU]: PID 1 finished at 21
                                                        [JS] PID: 2 at 24
[CPU]: PID 2 start at 24
                                                        [JS] PID: 3 at 28
                                                        [JS] PID: 4 at 36
[CPU]: PID 2 finished at 48
[CPU]: PID 3 start at 48
[CPU]: PID 3 finished at 58
[CPU]: PID 4 start at 58
[CPU]: PID 4 finished at 60
```

De nuevo podemos ver como el servidor recibe cada job y dura el tiempo exacto según su burst indicado, importante tener en cuenta que los tiempos de llegada son siempre aleatorios.

Tabla completa:

```
Server: stop

Stop CPU:

# of jobs executed: 5
Time in secs of idle CPU: 27
Job: 0, TAT: 3, WT: 0, B: 3, P: 5, AT: 11, ET: 14
Job: 1, TAT: 3, WT: 0, B: 3, P: 3, AT: 18, ET: 21
Job: 2, TAT: 24, WT: 0, B: 24, P: 1, AT: 24, ET: 48
Job: 3, TAT: 30, WT: 20, B: 10, P: 2, AT: 28, ET: 58
Job: 4, TAT: 24, WT: 22, B: 2, P: 4, AT: 36, ET: 60
------------------
AVG. WAT: 8
AVG. TAT: 16
```

Concluyendo así que el cliente manual en forma de FIFO tiene un funcionamiento correcto como se puede observar que se fueron ejecutando en su orden de llegada.

SJF

Archivo de prueba:

```
BURST Prioridad
20 5
15 3
17 1
18 2
10 4
```

Respuesta del cliente:

```
./manual_client SJF.txt

/ / / Client started / / /

Job sent - burst: 20 - priority: 5
Job recv - pid: 1
Job sent - burst: 15 - priority: 3
Job recv - pid: 2
Job sent - burst: 17 - priority: 1
Job recv - pid: 3
Job sent - burst: 18 - priority: 2
Job recv - pid: 4
Job sent - burst: 10 - priority: 4
Job recv - pid: 5
```

Nuevamente podemos analizar la enviada de datos constante de parte del cliente hasta que se completen todos los procesos que se crearon con el archivo.

Respuesta del servidor:

```
Server : SJF
                                                    [JS] PID: 0 at 8
[CPU]: PID 0 start at 8
                                                    [JS] PID: 1 at 15
                                                    [JS] PID: 2 at 21
                                                    [JS] PID: 3 at 25
```

```
[CPU]: PID 0 finished at 28
[CPU]: PID 1 start at 28
                                                [JS] PID: 4 at 33
[CPU]: PID 1 finished at 43
[CPU]: PID 4 start at 43
[CPU]: PID 4 finished at 53
[CPU]: PID 2 start at 53
[CPU]: PID 2 finished at 70
[CPU]: PID 3 start at 70
[CPU]: PID 3 finished at 88
```

Con la respuesta del servidor podemos analizar lo siguiente, teniendo en cuenta que estamos ejecutando el algoritmo de SJF, esto significa que el proceso con menor burst será el que se ejecute primero, según el archivo enviado este algoritmo se ejecuta de manera correcta, primero ejecuta el proceso 0 y a su vez va recibiendo otros jobs.

Luego ejecuta el proceso 1 cuando termina el 0, es en este momento que llega el proceso 4, con un burst menor a todos, termina el proceso 1 y ahora si, ejecuta el proceso 4, terminando este, ejecuta los otros procesos pendientes, pero en forma de SJF, mostrando esto que el algoritmo se ejecuta de manera correcta.

Tabla completa:

```
# of jobs executed: 5
Time in secs of idle CPU: 14

Job: 0, TAT: 20, WT: 0, B: 20, P: 5, AT: 8, ET: 28
Job: 1, TAT: 28, WT: 13, B: 15, P: 3, AT: 15, ET: 43
Job: 4, TAT: 20, WT: 10, B: 10, P: 4, AT: 33, ET: 53
Job: 2, TAT: 49, WT: 32, B: 17, P: 1, AT: 21, ET: 70
Job: 3, TAT: 63, WT: 45, B: 18, P: 2, AT: 25, ET: 88
------------------
AVG. WAT: 20
AVG. TAT: 36
```

Concluyendo así que el cliente manual en forma de SJF tiene un funcionamiento correcto.

HPF

Archivo de pruebas:

```
BURST Prioridad
20 3
15 1
10 3
4 4
8 2
```

Respuesta del cliente:

```
./manual_client HPF.txt

/ / / Client started / / /

Job sent - burst: 20 - priority: 3
Job recv - pid: 1
Job sent - burst: 15 - priority: 1
Job recv - pid: 2
Job sent - burst: 10 - priority: 3
Job recv - pid: 3
Job sent - burst: 4 - priority: 4
Job recv - pid: 4
Job sent - burst: 8 - priority: 2
Job recv - pid: 5
```

Nuevamente podemos analizar la enviada de datos constante de parte del cliente hasta que se completen todos los procesos que se crearon con el archivo.

Respuesta del servidor:

```
Server: HPF

                                                    [JS] PID: 0 at 9
[CPU]: PID 0 start at 9
```

```
                                                    [JS] PID: 1 at 16
                                                    [JS] PID: 2 at 22
                                                    [JS] PID: 3 at 26
[CPU]: PID 0 finished at 29
[CPU]: PID 1 start at 29
                                                    [JS] PID: 4 at 34
[CPU]: PID 1 finished at 44
[CPU]: PID 4 start at 44
[CPU]: PID 4 finished at 52
[CPU]: PID 2 start at 52
[CPU]: PID 2 finished at 62
[CPU]: PID 3 start at 62
[CPU]: PID 3 finished at 66
```

Con la respuesta del servidor podemos analizar lo siguiente, teniendo en cuenta que es un algoritmo HPF, se ejecuta el proceso con mayor prioridad, se ejecuta el proceso 0 y 1, porque todavía no han llegado otros procesos con menor prioridad, pero durante la ejecución de estos procesos llegaron el resto de jobs que faltaban.

Es acá donde se muestra la ejecución del algoritmo, el proceso 4 se ejecuta sobre el 3 y 2, porque tiene una prioridad de 2, luego se ejecuta el proceso 2, porque tiene prioridad de 3 y por último el proceso 3 que tiene una prioridad de 4. Mostrando esto la correcta ejecución del algoritmo.

Tabla completa:

```
Server: stop

Stop CPU:

# of jobs executed: 5
Time in secs of idle CPU: 92

Job: 0, TAT: 20, WT: 0, B: 20, P: 3, AT: 9, ET: 29
Job: 1, TAT: 28, WT: 13, B: 15, P: 1, AT: 16, ET: 44
Job: 4, TAT: 18, WT: 10, B: 8, P: 2, AT: 34, ET: 52
Job: 2, TAT: 40, WT: 30, B: 10, P: 3, AT: 22, ET: 62
Job: 3, TAT: 40, WT: 36, B: 4, P: 4, AT: 26, ET: 66
------------------
AVG. WAT: 17
AVG. TAT: 29
```

Concluyendo así que el cliente manual en forma de HPF tiene un funcionamiento correcto.

RR

Archivo de pruebas:

```
BURST Prioridad
6 3
17 3
8 2
7 4
12 1
```

Además de este archivo el algoritmo de RR, recibe como parámetro el quantum, en este caso se envía un 2 para hacer estos ciclos.

Respuesta del cliente:

```
./manual_client RR.txt

/ / / Client started / / /

Job sent - burst: 6 - priority: 3
Job recv - pid: 1
Job sent - burst: 17 - priority: 3
Job recv - pid: 2
Job sent - burst: 8 - priority: 2
Job recv - pid: 3
Job sent - burst: 7 - priority: 4
Job recv - pid: 4
Job sent - burst: 12 - priority: 1
Job recv - pid: 5
```

Respuesta del servidor:

```
Server: RR
                                                    [JS] PID: 0 at 9
[CPU]: PID 0 start at 9
[CPU]: PID 0 with Burst Left 4
[CPU]: PID 0 start at 11
[CPU]: PID 0 with Burst Left 2
[CPU]: PID 0 start at 13
[CPU]: PID 0 finished at 15
                                                   [JS] PID: 1 at 16
[CPU]: PID 1 start at 16
[CPU]: PID 1 with Burst Left 15
[CPU]: PID 1 start at 18
[CPU]: PID 1 with Burst Left 13
[CPU]: PID 1 start at 20
                                                   [JS] PID: 2 at 22
[CPU]: PID 1 with Burst Left 11
[CPU]: PID 2 start at 22
[CPU]: PID 2 with Burst Left 6
[CPU]: PID 1 start at 24
                                                   [JS] PID: 3 at 26
[CPU]: PID 1 with Burst Left 9
[CPU]: PID 2 start at 26
[CPU]: PID 2 with Burst Left 4
[CPU]: PID 3 start at 28
[CPU]: PID 3 with Burst Left 5
[CPU]: PID 1 start at 30
[CPU]: PID 1 with Burst Left 7
[CPU]: PID 2 start at 32
```

Con estos procesos hasta ahora, podemos analizar que en efecto se van ejecutando los procesos cada 2 ciclos, esto nos confirma la cantidad que burst que le quedan a los procesos y a su vez podemos ver como los procesos se van ejecutando en orden de llegada.

Ready Queue hasta el momento:

```
PID: 2, Burst: 8, Priority: 2, AT: 22
PID: 3, Burst: 7, Priority: 4, AT: 26
PID: 1, Burst: 17, Priority: 3, AT: 16
```

Con la tabla de ready queue, en este momento podemos analizar lo siguiente, el proceso 0 ya terminó de ejecutarse, el proceso 4, no ha llegado, y se puede ver el burst inicial, la prioridad y el tiempo de llegada de cada proceso.

Continuar con el proceso:

```
[CPU]: PID 2 start at 33
                                                [JS] PID: 4 at 34
[CPU]: PID 2 with Burst Left 1
[CPU]: PID 3 start at 35
[CPU]: PID 3 with Burst Left 3
[CPU]: PID 1 start at 37
[CPU]: PID 1 with Burst Left 5
[CPU]: PID 4 start at 39
[CPU]: PID 4 with Burst Left 10
[CPU]: PID 2 start at 41
[CPU]: PID 2 finished at 42
[CPU]: PID 3 start at 42
[CPU]: PID 3 with Burst Left 1
[CPU]: PID 1 start at 44
[CPU]: PID 1 with Burst Left 3
[CPU]: PID 4 start at 46
[CPU]: PID 4 with Burst Left 8
[CPU]: PID 3 start at 48
[CPU]: PID 3 finished at 49
[CPU]: PID 1 start at 49
[CPU]: PID 1 with Burst Left 1
[CPU]: PID 4 start at 51
[CPU]: PID 4 with Burst Left 6
[CPU]: PID 1 start at 53
[CPU]: PID 1 finished at 54
[CPU]: PID 4 start at 54
```

Siguiendo con el proceso, podemos seguir corroborando que cada proceso se ejecuta cada 2 ciclos y se ejecutan en orden de llegada.

Ready Queue hasta el momento:

```
PID: 4, Burst: 12, Priority: 1, AT: 34
```

Esto nos indica que ya se terminaron todos los procesos menos el proceso 4, el cual llegó de último, verificando con las impresiones anteriores, confirmamos que esto es correcto.

Terminando así la ejecución del programa.

```
[CPU]: PID 4 start at 56
[CPU]: PID 4 with Burst Left 2
[CPU]: PID 4 start at 58
[CPU]: PID 4 finished at 60
```

Tabla completa:

```
Server: stop

Stop CPU:

# of jobs executed: 5
Time in secs of idle CPU: 14
Job: 0, TAT: 6, WT: 0, B: 6, P: 3, AT: 9, ET: 15
Job: 2, TAT: 20, WT: 12, B: 8, P: 2, AT: 22, ET: 42
Job: 3, TAT: 23, WT: 16, B: 7, P: 4, AT: 26, ET: 49
Job: 1, TAT: 38, WT: 21, B: 17, P: 3, AT: 16, ET: 54
Job: 4, TAT: 26, WT: 14, B: 12, P: 1, AT: 34, ET: 60
------------------
AVG. WAT: 12
AVG. TAT: 22
```

Concluyendo así que el cliente manual en forma de RR tiene un funcionamiento correcto.

En conclusión, se llevaron a cabo con éxito los casos de prueba propuestos en el proyecto del simulador de planificador de CPU. Se comprobó el correcto funcionamiento del servidor, cliente manual y cliente automático, así como la implementación de los algoritmos de planificación FIFO, SJF, HPF y RR con el quantum especificado por el usuario.

Se pudo verificar que los resultados obtenidos en la ejecución de cada algoritmo corresponden a los esperados en términos de tiempos de espera, tiempos de respuesta y turnaround time para los procesos.

Además, se realizaron pruebas de estrés y carga en el simulador, y se pudo comprobar que el sistema responde adecuadamente y se comporta de manera estable.

En general, se considera que los casos de prueba fueron completos y exhaustivos, cubriendo todos los aspectos relevantes del proyecto. Como resultado, se puede asegurar que el simulador de planificador de CPU está listo para su implementación y uso en aplicaciones reales.

Comparación

En la programación de sistemas y aplicaciones de alto rendimiento, la elección de la herramienta adecuada para la gestión de hilos es crucial para garantizar un alto rendimiento y una correcta utilización de los recursos del sistema. En este contexto, los hilos de Java y Pthreads son dos de las opciones más populares. Ambos ofrecen una amplia gama de características y opciones de configuración, pero difieren en su uso, detalles técnicos y rendimiento.

Java y POSIX threads (pthreads) son dos enfoques diferentes para la programación concurrente en sistemas operativos. En esta investigación comparativa, explicaremos las diferencias entre Java y pthreads en términos de usabilidad, detalles técnicos y rendimiento.

Usabilidad

Java es un lenguaje orientado a objetos de alto nivel que proporciona una interfaz de programación de aplicaciones (API) fácil de usar para la programación concurrente. Java utiliza una implementación de hilos nativos que se ejecutan en el sistema operativo subyacente y se administran mediante una API de alto nivel. Java

también proporciona características de sincronización y bloqueo integradas en el lenguaje, lo que facilita la creación de programas seguros y robustos en un entorno multi-hilo.

Pthreads, por otro lado, son una biblioteca de programación C para la creación y el manejo de hilos. Pthreads requiere que los programadores administren los hilos manualmente, lo que puede resultar complicado y propenso a errores. Los programadores también deben preocuparse por la sincronización y el bloqueo manualmente, lo que puede ser propenso a errores y difícil de depurar.

Detalles técnicos

Java utiliza una implementación de hilos nativos que se ejecutan en el sistema operativo subyacente y se administran mediante una API de alto nivel. Los hilos de Java son administrados por el recolector de basura y están protegidos por la memoria administrada. Los hilos de Java también son escalables y pueden aprovechar múltiples núcleos de CPU en un sistema.

Pthreads, por otro lado, son una biblioteca de programación C que requiere que los programadores administren los hilos manualmente. Los hilos de pthreads se ejecutan en el espacio de usuario y no están protegidos por la memoria administrada. Esto puede hacer que los programas sean más propensos a errores de memoria, como fugas de memoria y corrupción de memoria. Además, pthreads no proporciona una escalabilidad nativa y no puede aprovechar múltiples núcleos de CPU en un sistema.

Rendimiento

Java es conocido por tener una sobrecarga adicional debido a su recolector de basura, que puede ralentizar la ejecución de programas. Sin embargo, los hilos de Java pueden aprovechar múltiples núcleos de CPU en un sistema y proporcionar

una escalabilidad nativa. Además, la implementación de hilos nativos de Java es generalmente más eficiente que la administración manual de hilos en pthreads.

Pthreads, por otro lado, puede proporcionar un mejor rendimiento en sistemas con un solo núcleo de CPU debido a su baja sobrecarga. Sin embargo, pthreads no puede aprovechar múltiples núcleos de CPU en un sistema y puede requerir una administración manual de hilos que resulte en programas propensos a errores.

En resumen, Java proporciona una API fácil de usar y una implementación de hilos nativos escalable y eficiente, mientras que pthreads puede proporcionar un mejor rendimiento en sistemas con un solo núcleo de CPU pero requiere una administración manual de hilos propensa a errores.

Manual de usuario

Este manual está planeado para alguien que no sabe cómo fue que se programó la solución y ya tiene una versión completa y estable del producto.

Este simulador de planificador de CPU es un proyecto que permite implementar diferentes algoritmos de planificación de procesos tales como FIFO, SJF, HPF y Round Robin, y para ello se utiliza la librería de C llamada ncurses.
Es importante mencionar que, antes de iniciar con la ejecución de este proyecto, es necesario realizar la instalación de esta librería en su sistema operativo. A continuación, se presentarán los pasos necesarios para llevar a cabo la instalación de la librería y la ejecución del simulador.

Paso 1: Instalación de la librería ncurses

Para instalar la librería de ncurses en su sistema operativo, utilice el siguiente comando en su terminal:

```
sudo apt-get install libncursesw5-dev
```

Paso 2: Compilación del servidor

Para compilar el servidor, utilice el siguiente comando en su terminal:

```
gcc -o server server.c -lncurses -lpthread
```

Paso 3: Compilación de los clientes

Existen dos tipos de clientes que se pueden compilar para conectarse al servidor, el cliente automático y el cliente manual. Para compilar el cliente automático, utilice el siguiente comando en su terminal:

```
gcc -o automatic_client automatic_client.c -lpthread
```

Y para compilar el cliente manual, utilice el siguiente comando en su terminal:

```
gcc -o manual_client manual_client.c -lpthread
```

Paso 4: Ejecución del servidor

Una vez compilado el servidor, se puede ejecutar la información de los algoritmos de planificación de procesos. Para ello, utilice el siguiente comando en su terminal:

```
./server tipo de algoritmo
```

Los diferentes tipos de algoritmos que se pueden utilizar son FIFO, SJF, HPF y RR. En el caso de RR, es necesario indicar cada cuántos ciclos se ejecutará, de la siguiente manera:

```
./server RR #ciclos
```

Ejemplos:

```
./server FIFO
./server RR 2
```

Paso 5: Ejecución del cliente manual

Para ejecutar el cliente manual, se debe abrir una nueva terminal y ejecutar el siguiente comando en su terminal:

```
./manual_client nombre del archivo txt
```

El archivo txt debe contener el código de prueba, tal como se explicó en una sección anterior. Una vez ejecutado este comando, se empezarán a ejecutar los jobs con sus burst de acuerdo al algoritmo de planificación que se está utilizando.

```
Ejemplo: ./manual_client FIFO.txt
```

Paso 6: Ejecución del cliente automático

Finalmente, se puede ejecutar el cliente automático en otra terminal. Este cliente recibe dos parámetros, el primero indica la frecuencia con la que se crean jobs (un número más pequeño creará jobs más rápido) y el segundo indica el máximo de burst que puede tener cada job. Para ejecutar este cliente automático, utilice el siguiente comando en su terminal:

```
./automatic_client frecuencia máximo burst
```

Ejemplo: `./automatic_client 1 9`

Comandos

Estos comandos son exclusivos del servidor y solo se pueden utilizar mientras el cliente está ejecutando los diferentes trabajos.

- 'help': Este comando muestra una lista de los comandos disponibles que se pueden utilizar en el servidor.
- 'stop': Detiene la CPU y muestra información relevante, como la cantidad de trabajos ejecutados, la cantidad de segundos de CPU ociosa y una tabla con:

    - Turn around Time (TAT)
    - Waiting time (WT)
    - El Burst (B)
    - La prioridad (P)
    - El tiempo de llegada y el tiempo final de los trabajos.

    Y abajo de esto
    - Promedio de Waiting Time (WAT)
    - Promedio de Turn Around Time (TAT)

- 'queue': Consulta la cola de trabajos listos (Ready Queue), es decir, los trabajos que ya han llegado al sistema y están esperando a ser procesados.
- 'restart': Reinicia el programa del servidor, lo que es útil en caso de que se necesite limpiar el sistema y comenzar de nuevo.
- 'freeze': Congela la pantalla del servidor para detener temporalmente la visualización de la información.
- 'unfreeze': Reanuda la pantalla del servidor después de haberla congelado.

- 'exit': Detiene el servidor por completo y finaliza la ejecución del programa. También es útil para salir de la interfaz del servidor.

Estos comandos son útiles para monitorear el progreso del servidor y obtener información relevante sobre los trabajos procesados.

Es importante tener en cuenta que estos comandos solo se pueden utilizar mientras se está ejecutando el cliente y los diferentes procesos, y solo se deben utilizar cuando sea necesario y con precaución, ya que algunos de ellos pueden detener la ejecución de los procesos y afectar la simulación del planificador de CPU.

Bitácora

A continuación se muestra una tabla con las actividades realizadas durante el desarrollo de este proyecto, las fechas corresponden a los días que se hizo reunión; no obstante, se excluye el tiempo utilizado para la búsqueda de información y referencias.

Bitacora		
Fecha	Actividad	Detalles relevantes
13/03/2023	Reunión Inicial	Estudiar el proyecto, ver requisitos y distribuir tareas
18/03/2023	Revisar estructura y manejo de pthreads	Gran parte del proyecto se utilizan los threads, por eso se hizo una reunión específica para estudiarlos y aplicarlos
20/10/2023	Revisión y Pruebas del Servidor	Se corrobora que la información solicitada y el código creado por el equipo sobre el servidor sea la correcta y tenga funcionamiento Conclusión : Aprobada
22/03/2023	Revision de Clientes	Se verifica que la información enviada por el servidor sea recibida y pueda ser manejada por los clientes Conclusión : Aprobada
24/03/2023	Documentación y revisar avances	Se empieza la documentación, además de esto se revisan los avances del Job Scheduler y CPU scheduler

		Conclusión : Errores en el JS con el PCB y en el CS con SJF, HPF y RR
26/03/2023	Corrección de errores	Se corrigen los errores y se verifican los errores anteriores de JS y SJF Conclusión : Pendiente HPF y RR
30/04/2023	Corrección de errores	Se corrigen errores de CS como el HPF y RR
31/04/2023	Pruebas	Se corren pruebas en general para revisar el funcionamiento completo del programa. Conclusión : Se encuentran errores en comandos del servidor y en manejo de la cola, en CS, específicamente FIFO y RR. Diseño de impresión de datos
01/04/2023	Corrección de errores	Se corrigen todos los errores anteriores. Conclusion : Programa terminado
02/04/2023	Documentación	Se concluye la documentación con las ultimas pruebas del codigo

Bibliografía

GeeksforGeeks. (2023, 20 febrero). *Socket Programming in C C*.

https://www.geeksforgeeks.org/socket-programming-cc/

pthread_create() — Create a thread. (s. f.).

https://www.ibm.com/docs/en/zos/2.3.0?topic=functions-pthread-create-create-thread

pthread_create(3) - Linux manual page. (s. f.).

https://man7.org/linux/man-pages/man3/pthread_create.3.html

Wolovick, N. (s. f.). *Lab3: Programando con Hilos*.

https://cs.famaf.unc.edu.ar/%7Enicolasw/Docencia/so2002/lab3.html

GitHub

https://github.com/JGamboaB/SO-CPU_Planner

Anexos

Anexo 1 Job Scheduler

```
#ifndef JOBSCHEDULER_H
#define JOBSCHEDULER_H

//Envía un mensaje de confirmación al cliente con el valor del PID.
//Cada vez que un proceso termina completamente su ejecución y deja de estar
en espera debe desplegarlo en pantalla.

typedef struct PCB{
   int pid;
   int burst;
   int priority;
   int startTime;
   int endTime;
   int waitingTime;
   int turnaroundTime;
   int burstLeft;
   int finish; //0: ready, 1: finished
   struct PCB* next;
} PCB;

typedef struct ReadyQueue{
   int cpuOcioso;
   int finishedJobs;
   PCB* head;
   PCB* last;
} ReadyQueue;

typedef struct FinishQueue{
   int cpuOcioso;
   int finishedJobs;
   PCB* head;
   PCB* last;
} FinishQueue;

PCB* insert(ReadyQueue *RQ, int pid, int burst, int priority, int starTime){

   //Create PCB
   PCB* pcb = (PCB*)malloc(sizeof(PCB));
   pcb->pid = pid;
   pcb->burst = burst;
   pcb->burstLeft = burst;
   pcb->priority = priority;
```

```
    pcb->next = NULL;
    pcb->startTime = starTime;
    pcb->finish = 0;
    pthread_mutex_lock(&cpu_mutex);
    if (RQ->head == NULL){ //empty
        RQ->head = pcb;
        RQ->last = pcb;
    } else {
        if (RQ->head->next == NULL){ //only 1 element in RQ
            RQ->head->next = pcb;
            RQ->last = pcb;
        } else {
            RQ->last->next = pcb;
            RQ->last = pcb;
        }
    }
    pthread_mutex_unlock(&cpu_mutex);
    return pcb;
}

void delete(ReadyQueue *RQ,FinishQueue *FQ, PCB *pcb){
    //Create PCB
    PCB* pcb2 = (PCB*)malloc(sizeof(PCB));
    pcb2->pid = pcb->pid;
    pcb2->burst = pcb->burst;
    pcb2->priority = pcb->priority;
    pcb2->next = pcb->next;
    pcb2->startTime = pcb->startTime;
    pcb2->endTime = pcb->endTime; //////
    pcb2->turnaroundTime = pcb->turnaroundTime; /////
    pcb2->waitingTime = pcb->waitingTime; /////
    pcb2->finish = 1;
    pthread_mutex_lock(&cpu_mutex);
    if (FQ->head == NULL){ //empty
        FQ->head = pcb2;
        FQ->last = pcb2;
    } else {
        if (FQ->head->next == NULL){ //only 1 element in RQ
            FQ->head->next = pcb2;
            FQ->last = pcb2;
        } else {
            FQ->last->next = pcb2;
            FQ->last = pcb2;
        }
    }
    pthread_mutex_unlock(&cpu_mutex);

    pthread_mutex_lock(&cpu_mutex);
```

```
    if (RQ->head == pcb){ //first element

        RQ->head = pcb->next;
        if (RQ->last == pcb)//only element
            RQ->last = NULL;

        free(pcb);
        pthread_mutex_unlock(&cpu_mutex);
        return;
    }
    pthread_mutex_unlock(&cpu_mutex);

    PCB* tmp = RQ->head;

    pthread_mutex_lock(&cpu_mutex);
    while(tmp != NULL){ //search
        if (tmp->next == pcb){
            tmp->next = pcb->next;
            if (pcb->next == NULL)
                RQ->last == pcb;
            free(pcb);
            pthread_mutex_unlock(&cpu_mutex);
            return;
        }
        tmp = tmp->next;
    }
    pthread_mutex_unlock(&cpu_mutex);
}


void moveFirstToLast(ReadyQueue *RQ){
    pthread_mutex_lock(&cpu_mutex);
    RQ->last->next = RQ->head; //the one before the last one has a connection
to the new last one
    RQ->last = RQ->last->next; // The first becomes the last
    RQ->head = RQ->head->next;  // The first job becomes the next one
    RQ->last->next = NULL; //Remove link to the new "first" job of the last
one
    pthread_mutex_unlock(&cpu_mutex);
}

PCB* getSJF(ReadyQueue *RQ){ //Get Shortest Job
    pthread_mutex_lock(&cpu_mutex);
    PCB* tmp = RQ->head;
    PCB* SJ = tmp;
    int shortest = tmp->burst;

    while(tmp != NULL){
```

```
        if (tmp->burst < shortest){ //Found someone shorter
            shortest = tmp->burst;
            SJ = tmp;
        } tmp = tmp->next;
    }
    pthread_mutex_unlock(&cpu_mutex);
    return SJ;
}

PCB* getHPF(ReadyQueue *RQ){ //Get Highest Priority
    pthread_mutex_lock(&cpu_mutex);
    PCB* tmp = RQ->head;
    PCB* HP = tmp;
    int priority = tmp->priority;

    while(tmp != NULL){
        if (tmp->priority < priority){ //Found someone with more priority
            priority = tmp->priority;
            HP = tmp;
        } tmp = tmp->next;
    }
    pthread_mutex_unlock(&cpu_mutex);
    return HP;
}

void printRQ(ReadyQueue *RQ){
    pthread_mutex_lock(&cpu_mutex);
    printf("Ready Queue:\n");
    PCB* tmp = RQ->head;

    if (tmp == NULL){
        printf("\\ \\ Empty \\ \\");
    }

    while(tmp != NULL){
        printf("-> PID: %d, Burst: %d, Priority: %d ", tmp->pid, tmp->burst,
tmp->priority);
        tmp = tmp->next;
    }printf("\n");
    pthread_mutex_unlock(&cpu_mutex);
}

#endif
```

Anexo 2 CPU Scheduler

Anexo 2.1 Struct

```
//
// Created by andre on 31/3/2023.
//

#ifndef SO_CPU_PLANNER_CPUSCHEDULER_H
#define SO_CPU_PLANNER_CPUSCHEDULER_H

// #include <unistd.h>  // For sleep
#include "jobscheduler.h"

#define TIME 1  // Used for processing simulation time


volatile sig_atomic_t stop = 0;

/*
* @author Andres
* @dev this function takes the first job of the ready queue and executes it
until it finishes
* @param *arg: since it works with another thread, it needs to receive the
parameters as a void pointer
*   Following, see the parameters that this pointer contains
* @cpuinfo: a struct that contains the ready queue, the finish queue and the
output window
* @param readyQueue: a queue of structs Job to work on
* @param FQ: a queue of structs Job that contains the finished jobs
* @param output: the output window to print the messages
* */
void *fifo(void *arg) {
   // Unpacks the parameters
   CPUINFO *cpuinfo = (CPUINFO *)arg;
   ReadyQueue *readyQueue = cpuinfo->RQ;
   FinishQueue *FQ = cpuinfo->FQ;
   WINDOW *output = cpuinfo->output;

   // Infinite loop to always check changes in the ready queue
   while(true){
       if (stop){
           sleep(1);
       } else {
           while(true){
               //to kill infinite loops
               if(stop){
                   break;
```

```
                }

                // There are no jobs in the ready queue, simulates idle time
                while(readyQueue->head == NULL){
                    int tempTime = TIMESF;
                    double time = 0;
                    while (tempTime == TIMESF){
                        if(readyQueue->head != NULL)
                        {
                            break;
                        }
                        sleep(0.01);
                        time += 0.01;
                    }
                    if(time >= 1){
                        readyQueue->cpuOcioso++;
                    }
                }

                // Found a job, keeps loading jobs until there are no more
left
                while(readyQueue->head != NULL) {
                    if(stop){
                        break;
                    }
                    PCB* job = readyQueue->head;  // Takes the first job of
the queue

                    // Prints that the job entered CPU
                    char message[100];
                    pthread_mutex_lock(&win_mutex);
                    sprintf(message, "[CPU]: PID %d start at %d\n", job->pid,
TIMESF);
                    waddstr(output, message);
                    wrefresh(output);
                    pthread_mutex_unlock(&win_mutex);

                    // Simulates the job execution time
                    int i = 0;
                    while(i < job->burst){
                        if(stop){
                            break;
                        }
                        int tempTime2 = TIMESF;
                        while (tempTime2 == TIMESF){
                            sleep(0.1);
                        }
                        i++;
```

```
                    }

                    // To kill infinite loops
                    if(stop){
                        break;
                    }

                    // The job finished, takes its statistics
                    job->endTime = TIMESF;
                    job->turnaroundTime = (job->endTime) - (job->startTime);
                    job->waitingTime = (job->turnaroundTime) - (job->burst);
                    readyQueue->finishedJobs++;

                    // Prints that the job finished
                    char message2[100];
                    pthread_mutex_lock(&win_mutex);
                    sprintf(message2, "[CPU]: PID %d finished at %d\n",
job->pid, TIMESF);
                    waddstr(output, message2);
                    //mvwprintw(win->input, 0, 0, "Command: ");
                    wrefresh(output);
                    pthread_mutex_unlock(&win_mutex);

                    delete(readyQueue,FQ, job);  // The job finished, so is
removed from the queue

                    // Goes for the next job
                }
            }
        }
    }
}

/*
* @author Andres
* @dev this function takes the job with the shortest burst from the ready
queue and executes it until it finishes
* @param *arg: since it works with another thread, it needs to receive the
parameters as a void pointer
*   Following, see the parameters that this pointer contains
* @cpuinfo: a struct that contains the ready queue, the finish queue and the
output window
* @param readyQueue: a queue of structs Job to work on
* @param FQ: a queue of structs Job that contains the finished jobs
* @param output: the output window to print the messages
* */
void shortestJobFirst(void *arg) {
    // Unpacks the parameters
```

```
    CPUINFO *cpuinfo = (CPUINFO *)arg;
    ReadyQueue *readyQueue = cpuinfo->RQ;
    FinishQueue *FQ = cpuinfo->FQ;
    WINDOW *output = cpuinfo->output;

    // Infinite loop to always check changes in the ready queue
    while(true){
        if (stop){
            sleep(1);
        } else {
            while(true){
                //to kill infinite loops
                if(stop){
                    break;
                }

                // There are no jobs in the ready queue, simulates idle time
                while(readyQueue->head == NULL){
                    int tempTime = TIMESF;
                    double time = 0;
                    while (tempTime == TIMESF){
                        if(readyQueue->head != NULL)
                        {
                            break;
                        }
                        sleep(0.01);
                        time += 0.01;
                    }
                    if(time >= 1){
                        readyQueue->cpuOcioso++;
                    }
                }

                // Found a job, keeps loading jobs until there are no more
left
                while(readyQueue->head != NULL) {
                    if(stop){
                        break;
                    }

                    PCB* job = getSJF(readyQueue); // Works with the job with
the shortest burst

                    // Prints that the job entered CPU
                    char message[100];
                    pthread_mutex_lock(&win_mutex);
                    sprintf(message, "[CPU]: PID %d start at %d\n", job->pid,
TIMESF);
```

```
                    waddstr(output, message);
                    wrefresh(output);
                    pthread_mutex_unlock(&win_mutex);

                    // Simulates the job execution time
                    int i = 0;
                    while(i < job->burst){
                        if(stop){
                            break;
                        }
                        int tempTime2 = TIMESF;
                        while (tempTime2 == TIMESF){
                            sleep(0.1);
                        }
                        i++;
                    }

                    // To kill infinite loops
                    if(stop){
                        break;
                    }

                    // The job finished, takes its statistics
                    job->endTime = TIMESF;
                    job->turnaroundTime = (job->endTime) - (job->startTime);
                    job->waitingTime = (job->turnaroundTime) - (job->burst);
                    readyQueue->finishedJobs++;

                    // Prints that the job finished
                    char message2[100];
                    pthread_mutex_lock(&win_mutex);
                    sprintf(message2, "[CPU]: PID %d finished at %d\n",
job->pid, TIMESF);
                    waddstr(output, message2);
                    //mvwprintw(win->input, 0, 0, "Command: ");
                    wrefresh(output);
                    pthread_mutex_unlock(&win_mutex);

                    delete(readyQueue,FQ, job);  // The job finished, so is
removed from the queue

                    // Goes for the next job
                }
            }
        }
    }
}
```

```
/*
* @author Andres
* @dev this function takes the job with the best priority from the ready
queue and executes it until it finishes
* @param *arg: since it works with another thread, it needs to receive the
parameters as a void pointer
*   Following, see the parameters that this pointer contains
* @cpuinfo: a struct that contains the ready queue, the finish queue and the
output window
* @param readyQueue: a queue of structs Job to work on
* @param FQ: a queue of structs Job that contains the finished jobs
* @param output: the output window to print the messages
* */
void highestPriorityFirst(void *arg) {
   // Unpacks the parameters
   CPUINFO *cpuinfo = (CPUINFO *)arg;
   ReadyQueue *readyQueue = cpuinfo->RQ;
   FinishQueue *FQ = cpuinfo->FQ;
   WINDOW *output = cpuinfo->output;

   // Infinite loop to always check changes in the ready queue
   while(true){
       if (stop){
           sleep(1);
       } else {
           while(true){
               //to kill infinite loops
               if(stop){
                   break;
               }

               // There are no jobs in the ready queue, simulates idle time
               while(readyQueue->head == NULL){
                   int tempTime = TIMESF;
                   double time = 0;
                   while (tempTime == TIMESF){
                       if(readyQueue->head != NULL)
                       {
                           break;
                       }
                       sleep(0.01);
                       time += 0.01;
                   }
                   if(time >= 1){
                       readyQueue->cpuOcioso++;
                   }
               }
```

```
                // Found a job, keeps loading jobs until there are no more
left
                while(readyQueue->head != NULL) {
                    if(stop){
                        break;
                    }

                    PCB* job = getHPF(readyQueue); // Works with the job with
the best priority

                    // Prints that the job entered CPU
                    char message[100];
                    pthread_mutex_lock(&win_mutex);
                    sprintf(message, "[CPU]: PID %d start at %d\n", job->pid,
TIMESF);
                    waddstr(output, message);
                    wrefresh(output);
                    pthread_mutex_unlock(&win_mutex);

                    // Simulates the job execution time
                    int i = 0;
                    while(i < job->burst){
                        if(stop){
                            break;
                        }
                        int tempTime2 = TIMESF;
                        while (tempTime2 == TIMESF){
                            sleep(0.1);
                        }
                        i++;
                    }

                    // To kill infinite loops
                    if(stop){
                        break;
                    }

                    // The job finished, takes its statistics
                    job->endTime = TIMESF;
                    job->turnaroundTime = (job->endTime) - (job->startTime);
                    job->waitingTime = (job->turnaroundTime) - (job->burst);
                    readyQueue->finishedJobs++;

                    // Prints that the job finished
                    char message2[100];
                    pthread_mutex_lock(&win_mutex);
                    sprintf(message2, "[CPU]: PID %d finished at %d\n",
job->pid, TIMESF);
```

```
                    waddstr(output, message2);
                    //mvwprintw(win->input, 0, 0, "Command: ");
                    wrefresh(output);
                    pthread_mutex_unlock(&win_mutex);

                    delete(readyQueue,FQ, job);  // The job finished, so is
removed from the queue

                    // Goes for the next job
                }
            }
        }
    }
}

/*
* @author Andres
* @dev this function takes the first job from the ready queue and executes
in quantum of time.
* If the process finishes in the quantum everything is ok, if it finishes
before the quantum ends, the quantum is just cut earlier and goes with the
next quantum
* If the process could not finish in the quantum, bad luck, it goes to the
end of the queue and has to wait for its next chance to continue executing
* @param *arg: since it works with another thread, it needs to receive the
parameters as a void pointer
*   Following, see the parameters that this pointer contains
* @cpuinfo: a struct that contains the ready queue, the finish queue and the
output window
* @param readyQueue: a queue of structs Job to work on
* @param FQ: a queue of structs Job that contains the finished jobs
* @param output: the output window to print the messages
* @param q, tells the length of the quantum
* */
void roundRobin(void *arg) {
    // Unpacks the parameters
    CPUINFO *cpuinfo = (CPUINFO *)arg;
    ReadyQueue *readyQueue = cpuinfo->RQ;
    FinishQueue *FQ = cpuinfo->FQ;
    int q = cpuinfo->rrQ;
    WINDOW *output = cpuinfo->output;

    // Infinite loop to always check changes in the ready queue
    while(true){
        if (stop){
            sleep(1);
        } else {
            while(true){
```

```
                //to kill infinite loops
                if(stop){
                    break;
                }

                // There are no jobs in the ready queue, simulates idle time
                while(readyQueue->head == NULL){
                    int tempTime = TIMESF;
                    double time = 0;
                    while (tempTime == TIMESF){
                        if(readyQueue->head != NULL)
                        {
                            break;
                        }
                        sleep(0.01);
                        time += 0.01;
                    }
                    if(time >= 1){
                        readyQueue->cpuOcioso++;
                    }
                }

                // Found a job, keeps loading jobs until there are no more
left
                while(readyQueue->head != NULL) {
                    if(stop){
                        break;
                    }

                    PCB* job = readyQueue->head;  // Takes the first job of
the queue

                    char finished = 0;  // This works as a flag that tells if
the job finished in that quantum or needs to wait for the next one

                    // Prints that the job entered CPU
                    char message[100];
                    pthread_mutex_lock(&win_mutex);
                    sprintf(message, "[CPU]: PID %d start at %d\n", job->pid,
TIMESF);
                    waddstr(output, message);
                    wrefresh(output);
                    pthread_mutex_unlock(&win_mutex);

                    // Simulates the job execution time
                    int i = 0;
                    while(i < q){
                        if(stop){
```

```
                        break;
                    }
                    int tempTime2 = TIMESF;
                    while (tempTime2 == TIMESF){
                        usleep(100000);
                    }
                    i++;

                    job->burstLeft--;  // Decreases the job burst time
                    if(job ->burstLeft == 0){  // If the job finished
                        finished = 1;  // Activates the flag
                        break;  // No point of continuing with this
quantum if the job ended
                    }
                }

                // To kill infinite loops
                if(stop){
                    break;
                }

                // Checks if the job was able to finish in the quantum
                if (finished) {
                    // The job finished, takes its statistics
                    job->endTime = TIMESF;
                    job->turnaroundTime = (job->endTime) -
(job->startTime);
                    job->waitingTime = (job->turnaroundTime) -
(job->burst);
                    readyQueue->finishedJobs++;

                    // Prints that the job finished
                    char message2[100];
                    pthread_mutex_lock(&win_mutex);
                    sprintf(message2, "[CPU]: PID %d finished at %d\n",
job->pid, TIMESF);
                    waddstr(output, message2);
                    //mvwprintw(win->input, 0, 0, "Command: ");
                    wrefresh(output);
                    pthread_mutex_unlock(&win_mutex);

                    delete(readyQueue, FQ, job);  // The job finished, so
is removed from the queue

                } else {  // The job could not finish in the quantum, has
to stay in the queue and wait for the next quantum
                    // Moves the job from head to last
                    char messageRR[100];
```

```
                        pthread_mutex_lock(&win_mutex);
                        sprintf(messageRR, "[CPU]: PID %d with Burst Left
%d\n", job->pid, job->burstLeft);
                        waddstr(output, messageRR);
                        //mvwprintw(win->input, 0, 0, "Command: ");
                        wrefresh(output);
                        pthread_mutex_unlock(&win_mutex);

                        moveFirstToLast(readyQueue);
                    }

                    // Goes for the next job
                }
            }
        }
    }
}




/*
* UNUSED PREEMPTIVE FUNCTIONS
* @author Andres
* @dev this function takes the job with the shortest burst from the ready
queue.
* Since it is preemptive it executes for just 1 time unit, it does not
matter if it could finish or not, and checks again
* @param readyQueue: a queue of structs Job to work on
* */
/*
void shortestJobFirstPreemptive(ReadyQueue *readyQueue) {
   // Keeps loading jobs until there are no more left
   while(readyQueue->head != NULL) {
       PCB* job = getSJF(readyQueue); // Works with the job with the
shortest burst

       // We cannot execute all the burst, we must go 1 by 1 to check if one
with a highest priority came
       printf("\nSe ejecuto por 1 el proceso %d", job->pid);
       sleep(TIME);  // Simulates it has taken 1 time unit
       job->burst--;  // Since it has advanced, the process is 1 unit closer
to end so its burst has to decrease

       // Checks if the job ended or still has burst to execute
       if(job->burst == 0){
           endJob(readyQueue, job);
       }
```

```
        // Goes for the next job
    }
}*/


/*
* @author Andres
* @dev this function takes the job with the best priority from the ready
queue.
* Since it is preemptive it executes for just 1 time unit, it does not
matter if it could finish or not, and checks again
* @param readyQueue: a queue of structs Job to work on
* */
/*
void highestPriorityPreemptive(ReadyQueue *readyQueue) {
    // Keeps loading jobs until there are no more left
    while(readyQueue->head != NULL) {
        PCB* job = getHPF(readyQueue); // Works with the job with the highest
priority

        // We cannot execute all the burst, we must go 1 by 1 to check if one
with a highest priority came
        printf("\nSe ejecuto por 1 el proceso %d", job->pid);
        sleep(TIME);  // Simulates it has taken 1 time unit
        job->burst--;  // Since it has advanced, the process is 1 unit closer
to end so its burst has to decrease

        // Checks if the job ended or still has burst to execute
        if(job->burst == 0){
            endJob(readyQueue, job);
        }

        // Goes for the next job
    }
}
*/



#endif //SO_CPU_PLANNER_CPUSCHEDULER_H
```

Anexo 2.2 Functions

```
#include <unistd.h>  // For sleep
#include "jobstruct.h"

#define TIME 2  // Used for processing simulation time

/*
* @author Andres
* @dev this function takes the first job of the ready queue and executes
it until it finishes
* @param readyQueue: a queue of structs Job to work on
* */
void fifo(struct ReadyQueue readyQueue) {
   // Keeps loading jobs until there are no more left
   while(!readyQueue.isEmpty()) {
       struct Job job = readyQueue.pop(0);  // Takes the first job of
the queue, pops it to also removed from the queue

       // Simulates the burst of the process
       while(job.burst > 0){
           sleep(TIME);  // Simulates it has taken 1 time unit
           job.burst--;  // Since it has advanced, the process is 1 unit
closer to end so its burst has to decrease
       }
       // Goes for the next job
   }
}

/*
* @author Andres
* @dev this function takes the job with the shortest burst from the
ready queue and executes it until it finishes
* @param readyQueue: a queue of structs Job to work on
* */
void shortestJobFirst(struct ReadyQueue readyQueue) {
   // Keeps loading jobs until there are no more left
   while(!readyQueue.isEmpty()) {
       // First it searches for the shortest burst
       int minBurst = 100;  // This variable will be used to compare
which job from all has the shortest burst
       int shortestJobIndex = 0;  // This tells the index of  the job we
will chose, to later access it
       // Goes job by job checking if we can get a shortest burst
       for(int i = 0; i < readyQueue.length(); i++) {
           int currentBurst = readyQueue.get(i).burst;

           // Checks if the current job has a shortest burst than the
```

```
shortest we have found so far
            if(currentBurst < minBurst) {
                shortestJobIndex = i;  // Updates the position of the job
with the shortest burst of all
                minBurst = currentBurst;  // Updates to the even shorter
burst we just found

                if(minBurst == 1){
                    break;  // There cannot be a shortest burst than 1,
with 1 we reached the best so there is no point of continue searching
                }
            }
        }

        // Works with the job with the shortest burst
        struct Job job = readyQueue.get(shortestJobIndex);  // Takes the
shortest job of the queue, pops it to also removed from the queue
        // Simulates the burst of the process
        while(job.burst > 0){
            sleep(TIME);  // Simulates it has taken 1 time unit
            job.burst--;  // Since it has advanced, the process is 1 unit
closer to end so its burst has to decrease
        }
        // Goes for the next job
    }
}

/*
* @author Andres
* @dev this function takes the job with the shortest burst from the
ready queue.
* Since it is preemptive it executes for just 1 time unit, it does not
matter if it could finish or not, and checks again
* @param readyQueue: a queue of structs Job to work on
* */
void shortestJobFirstPreemptive(struct ReadyQueue readyQueue) {
    // Keeps loading jobs until there are no more left
    while(!readyQueue.isEmpty()) {
        // First it searches for the shortest burst
        int minBurst = 100;  // This variable will be used to compare
which job from all has the shortest burst
        int shortestJobIndex = 0;  // This tells the index of  the job we
will chose, to later access it
        // Goes job by job checking if we can get a shortest burst
        for(int i = 0; i < readyQueue.length(); i++) {
            int currentBurst = readyQueue.get(i).burst;
```

```
            // Checks if the current job has a shortest burst than the
shortest we have found so far
            if(currentBurst < minBurst) {
                shortestJobIndex = i;  // Updates the position of the job
with the shortest burst of all
                minBurst = currentBurst;  // Updates to the even shorter
burst we just found

                if(minBurst == 1){
                    break;  // There cannot be a shortest burst than 1,
with 1 we reached the best so there is no point of continue searching
                }
            }
        }

        // Works with the job with the shortest burst
        struct Job job = readyQueue.get(shortestJobIndex);  // Takes the
shortest job of the queue, pops it to also removed from the queue

        // We cannot execute all the burst of the selected process, we
must go 1 by 1 to check if a shortest one came
        sleep(TIME);  // Simulates it has taken 1 time unit
        job.burst--;  // Since it has advanced, the process is 1 unit
closer to end so its burst has to decrease

        // Checks if the job ended or still has burst to execute
        if(job.burst > 0){
            readyQueue.push(job);  // Puts it back at the end of the
queue to wait for a chance to continue with its unfinished execution,
remember at the we popped it so we removed form the queue
        }

        // Goes for the next job
    }
}

/*
* @author Andres
* @dev this function takes the job with the best priority from the ready
queue and executes it until it finishes
* @param readyQueue: a queue of structs Job to work on
* */
void highestPriorityFirst(struct ReadyQueue readyQueue) {
    // Keeps loading jobs until there are no more left
    while(!readyQueue.isEmpty()) {
```

```
        // First it searches for the best priority
        int bestPriority = 100;  // This variable will be used to compare
which from all job has the best priority
        int mostImportantJobIndex = 0;  // This tells the index of  the
job we will chose, to later access it
        // Goes job by job checking if we can get a better priority
        for(int i = 0; i < readyQueue.length(); i++) {
            int currentPriority = readyQueue.get(i).priority;

            // Checks if the current job has a better priority than the
best we have found so far
            if(currentPriority < bestPriority) {
                mostImportantJobIndex = i;  // Updates the position of
the job with the shortest burst of all
                bestPriority = currentPriority;  // Updates to the even
shorter burst we just found

                if(bestPriority == 1){
                    break;  // There cannot be a better priority than 1,
with 1 we reached the best so there is no point of continue searching
                }
            }
        }

        // Works with the job with the bestPriority
        struct Job job = readyQueue.get(mostImportantJobIndex);  // Takes
the shortest job of the queue, pops it to also removed from the queue
        // Simulates the burst of the process
        while(job.burst > 0){
            sleep(TIME);  // Simulates it has taken 1 time unit
            job.burst--;  // Since it has advanced, the process is 1 unit
closer to end so its burst has to decrease
        }

        // Goes for the next job
    }
}

/*
* @author Andres
* @dev this function takes the job with the best priority from the ready
queue.
* Since it is preemptive it executes for just 1 time unit, it does not
matter if it could finish or not, and checks again
* @param readyQueue: a queue of structs Job to work on
* */
```

```
void highestPriorityPreemptive(struct ReadyQueue readyQueue) {
   // Keeps loading jobs until there are no more left
   while(!readyQueue.isEmpty()) {
       // First it searches for the best priority
       int bestPriority = 100;  // This variable will be used to compare
which from all job has the best priority
       int mostImportantJobIndex = 0;  // This tells the index of  the
job we will chose, to later access it
       // Goes job by job checking if we can get a better priority
       for(int i = 0; i < readyQueue.length(); i++) {
           int currentPriority = readyQueue.get(i).priority;

           // Checks if the current job has a better priority than the
best we have found so far
           if(currentPriority < bestPriority) {
               mostImportantJobIndex = i;  // Updates the position of
the job with the shortest burst of all
               bestPriority = currentPriority;  // Updates to the even
shorter burst we just found

               if(bestPriority == 1){
                   break;  // There cannot be a better priority than 1,
with 1 we reached the best so there is no point of continue searching
               }
           }
       }

       // Works with the job with the bestPriority
       struct Job job = readyQueue.get(mostImportantJobIndex);  // Takes
the shortest job of the queue, pops it to also removed from the queue

       // We cannot execute all the burst of the selected process, we
must go 1 by 1 to check if one with a highest priority came
       sleep(TIME);  // Simulates it has taken 1 time unit
       job.burst--;  // Since it has advanced, the process is 1 unit
closer to end so its burst has to decrease

       // Checks if the job ended or still has burst to execute
       if(job.burst > 0){
           readyQueue.push(job);  // Puts it back at the end of the
queue to wait for a chance to continue with its unfinished execution,
remember at the we popped it so we removed form the queue
       }

       // Goes for the next job
   }
```

```
}


/*
* @author Andres
* @dev this function takes the first job from the ready queue and
executes in quantum of time.
* If the process finishes in the quantum everything is ok, if it
finishes before the quantum ends, the quantum is just cut earlier and
goes with the next quantum
* If the process could not finish in the quantum, bad luck, it goes to
the end of the queue and has to wait for its next chance to continue
executing
* @param readyQueue: a queue of structs Job to work on
* @param q, tells the length of the quantum
* */
void roundRobin(struct ReadyQueue readyQueue, int q) {
   // Keeps loading jobs until there are no more left
   while (!readyQueue.isEmpty()) {
       struct Job job = readyQueue.pop(0);  // Takes the first job of
the queue

       char finished = 0;  // This works as a flag that tells if the job
finished in that quantum or needs to wait for the next one
       // Simulates the burst of the quantum
       for (int i = 0; i < q; i++) {
           sleep(TIME);  // Simulates it has taken 1 time unit
           job.burst--;  // Since it has advanced, the process is 1 unit
closer to end so its burst has to decrease

           // Checks if the process ended
           if (job.burst == 0) {
               finished = 1;  // Updates the flag
               break;  // Since the job finished, there is no point to
continue with this quantum
           }
       }

       // Since we popped the job in the first line, it is no longer in
the queue, however it may needs to return to the queue because it did
not finish
       if(!finished){
           readyQueue.push(job);  // Puts it back at the end of the
queue to wait for a chance to continue with its unfinished execution
       }
```

```
        // Goes for the next job
    }
}
```

Anexo 3 Job Struct

```
#ifndef JOBSTRUCT_H
#define JOBSTRUCT_H

typedef struct Job{
    int burst;
    int priority;
} Job;

#endif
```

Anexo 4 Server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include "jobstruct.h" //Custom Job struct
#include <ctype.h>
#include <limits.h>

#include <ncurses.h>  //ADD AT THE END OF COMPILING THE FILE -lncurses
#include <pthread.h>
#include <signal.h>
#include <errno.h>   // for errno


#define PORT 8080


// global pid
int pid_count = 0;
volatile sig_atomic_t flagRun = 1;
volatile sig_atomic_t algor = 0;
char *algorithm;
```

```c
static _Atomic int TIMESF = 0;


// JobTr
typedef struct JobTr{
   int sock_fd;                // socket
} JobTr;

// Client Connection and Server connection
typedef struct Win{
   WINDOW *input;              // input window
   WINDOW *output;             // output window
} Win;

// Client Connection and Server connection
typedef struct Connection{
   int sock_fd;                // socket server
   int new_socket;             // socket client
   struct Win *win;             // windows
} Connection;

// Send info CPU
typedef struct CPUINFO{
   struct ReadyQueue *RQ;      // Ready Queue
   struct FinishedQueue *FQ;   // Finished Queue
   WINDOW *output;             // output window
   int rrQ;
} CPUINFO;

pthread_mutex_t cpu_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t win_mutex = PTHREAD_MUTEX_INITIALIZER;

#include "cpuscheduler.h"
ReadyQueue RQ = {NULL, NULL};
FinishQueue FQ = {NULL, NULL};

int create_server_socket(){
   int server_fd = 0;
   if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) { //
socket(domain, type, protocol)
       perror("Server creation error");
       exit(EXIT_FAILURE);
   }

   int opt = 1;
   // Attach socket to the port 8080
```

```
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
&opt, sizeof(opt))) {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }

    printf("/ / / Server started / / /\n");
    return server_fd;
}


void init_server_address(struct sockaddr_in* address){
    memset(address, '0', sizeof(*address));
    address->sin_family = AF_INET; // IPv4
    address->sin_addr.s_addr = INADDR_ANY; //server will accept
connections to any of the IP addresses of the server.
    address->sin_port = htons(PORT);
}

// Bind the socket to the address and port
void bind_socket_to_address(int server_fd, struct sockaddr_in address){
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) <
0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    printf("/ / Binding succesfull / /\n");
}

// Listen for incoming connections
void listen_for_incoming_connections(int server_fd, int max_queue){
    if (listen(server_fd, max_queue) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }
}

void accept_incoming_connection(int server_fd, struct sockaddr_in
address, int* new_socket, int* addrlen){
    if ((*new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t *)addrlen)) < 0) {
        perror("accept");
        exit(EXIT_FAILURE);
    }
}
```

```c
void *handle_connection(void *arg) {
    Connection *connection = (Connection *)arg;
    int sock_fd = connection->sock_fd;
    int new_socket = connection->new_socket;
    struct Win *win = connection->win;

    while(true) {
        Job *job = malloc(sizeof(Job));
        if (recv(sock_fd, job, sizeof(Job), 0) == -1) {
            printf("Receive failed job\n");
            break;
        }
        else{
            int tempTime = TIMESF;
            while (tempTime == TIMESF){
                sleep(0.1);
            }

            if (flagRun == 0){
                break;
            }

            if(job->burst == 0 || job->priority == 0){
               //printf("Client desconect\n"); usar ncurses
                break;
            }

            while (stop){
                sleep(1);
            }

            pthread_mutex_lock(&win_mutex);
            char message0[100];
            for (int i = 0; i <= COLS/2; i++){
                waddstr(win->output, " ");

            }
            sprintf(message0, "[JS] PID: %d at %d\n",pid_count, TIMESF);
            waddstr(win->output, message0);
            wrefresh(win->output);
            pthread_mutex_unlock(&win_mutex);

            //mvwprintw(win->input, 0, 0, "Command: ");

            //printf("Received job with burst = %d, priority = %d\n",
```

```
job->burst, job->priority);

            // here you need to add to the ready queue

            insert(&RQ, pid_count, job->burst, job->priority,TIMESF);

            //add critical section for incrementing pid like example
            pthread_mutex_lock(&cpu_mutex);
            pid_count++;
            pthread_mutex_unlock(&cpu_mutex);

            // notify client of the pid
            if (send(sock_fd, &pid_count, sizeof(pid_count), 0) < 0) {
                printf("Send failed pid\n");
                break;
            }
        }
    }

    close(sock_fd);
    pthread_detach(pthread_self());
}

void *window_thread(void *arg) {
    Win *window = (Win *)arg;
    WINDOW *input = window->input;
    WINDOW *output = window->output;
    bool done = FALSE;
    char bufferWin[1024];
    char buffer[2048];
    char *com =  "Command: ";

    while(!done) {
        mvwprintw(input, 0, 0, com);
        if (wgetnstr(input, bufferWin, COLS - 4) != OK) {
            break;
        }

        pthread_mutex_lock(&win_mutex);
        werase(input);
        waddch(output, '\n');   /* result from wgetnstr has no newline */
        waddstr(output, "Server");
        waddstr(output, ": ");
        waddstr(output, bufferWin);
        wrefresh(output);
        pthread_mutex_unlock(&win_mutex);
```

```
        done = (*bufferWin == 4);  /* quit on control-D */

        // Here you put the commands

        if (strcmp(bufferWin, "exit") == 0) {
            sprintf(buffer, "%s\n", bufferWin);
            flagRun = 0;
            //send(socketfd, buffer, strlen(buffer), 0);
            break;
        }
        else if (strcmp(bufferWin, "help") == 0) {
            pthread_mutex_lock(&win_mutex);
            waddstr(output, "\n\nHelp ---------------------:\n");
            waddstr(output, "\n\t'help': Display available commands");
            waddstr(output, "\n\t'stop': Stop CPU and display
information");
            waddstr(output, "\n\t'queue': Consult Ready Queue");
            waddstr(output, "\n\t'restart': Restart Program");
            waddstr(output, "\n\t'freeze': Freeze the screen");
            waddstr(output, "\n\t'unfreeze': Unfreeze the screen");
            waddstr(output, "\n\t'exit': Stop Server\n");
            wrefresh(output);
            pthread_mutex_unlock(&win_mutex);
        }
        else if (strcmp(bufferWin, "stop") == 0) {
            //sprintf(buffer, "%s\n", bufferWin);
            stop = 1;

            pthread_mutex_lock(&win_mutex);
            werase(input);
            waddch(output, '\n');   /* result from wgetnstr has no
newline */
            waddstr(output, "\nStop CPU:\n");
            char message[100];
            sprintf(message, "\n# of jobs executed: %d",
RQ.finishedJobs);
            waddstr(output, message);
            waddstr(output, "\n");
            sprintf(message, "Time in secs of idle CPU: %d",
RQ.cpuOcioso);
            waddstr(output, message);
            waddstr(output, "\n");
            wrefresh(output);
            pthread_mutex_unlock(&win_mutex);

            if (RQ.finishedJobs != 0){
```

```
                PCB *tmp = FQ.head;
                int i = 0;
                int sumTAT = 0;
                int sumWT = 0;
                while( i != RQ.finishedJobs){
                    bzero(message, sizeof(message));
                    pthread_mutex_lock(&win_mutex);
                    sprintf(message, "Job: %d, TAT: %d, WT: %d, B: %d, P:
%d, AT: %d, ET: %d", tmp->pid, tmp->turnaroundTime, tmp->waitingTime,
tmp->burst, tmp->priority, tmp->startTime, tmp->endTime);
                    waddstr(output, message);
                    waddstr(output, "\n");
                    wrefresh(output);
                    pthread_mutex_unlock(&win_mutex);
                    usleep(500000);
                    sumTAT += tmp->turnaroundTime;
                    sumWT += tmp->waitingTime;
                    i++;
                    tmp = tmp->next;
                }
                pthread_mutex_lock(&win_mutex);
                sprintf(message, "------------------");
                waddstr(output, message);
                sprintf(message, "\nAVG. WAT: %d", sumWT/i);
                waddstr(output, message);
                waddstr(output, "\n");
                sprintf(message, "AVG. TAT: %d", sumTAT/i);
                waddstr(output, message);
                waddstr(output, "\n");
                wrefresh(output);
                pthread_mutex_unlock(&win_mutex);
            }

        } else if (strcmp(bufferWin, "queue") == 0) {
            int before = 0;
            if (stop){
                before = 1;
            }
            stop = 1;

            pthread_mutex_lock(&win_mutex);
            waddch(output, '\n');   /* result from wgetnstr has no
newline */
            waddstr(output, "\nReady Queue");
            waddstr(output, ": \n");
            pthread_mutex_unlock(&win_mutex);
```

```
            char message[100];

            PCB *tmp = RQ.head;

            while( tmp != NULL){
                bzero(message, sizeof(message));
                pthread_mutex_lock(&win_mutex);
                sprintf(message, "\n\t\t\tPID: %d, Burst: %d, Priority:
%d, AT: %d", tmp->pid, tmp->burst, tmp->priority, tmp->startTime);
                waddstr(output, message);
                wrefresh(output);
                pthread_mutex_unlock(&win_mutex);
                usleep(500000);
                tmp = tmp->next;
            }
            waddstr(output, "\n");
            wrefresh(output);

            stop = before;
        } else if (strcmp(bufferWin, "restart") == 0) {
            pthread_mutex_lock(&win_mutex);
            waddch(output, '\n');   /* result from wgetnstr has no
newline */
            waddstr(output, "\nCPU Restarted\n------------------\n");
            wrefresh(output);
            pthread_mutex_unlock(&win_mutex);
            stop = 0;
        } else if (strcmp(bufferWin, "freeze") == 0) {
            pthread_mutex_lock(&win_mutex);
            waddch(output, '\n');   /* result from wgetnstr has no
newline */
            wrefresh(output);
            pthread_mutex_unlock(&win_mutex);
            stop = 1;

        } else if (strcmp(bufferWin, "unfreeze") == 0) {
            pthread_mutex_lock(&win_mutex);
            waddch(output, '\n');   /* result from wgetnstr has no
newline */
            wrefresh(output);
            pthread_mutex_unlock(&win_mutex);
            stop = 0;
        }else {
            sprintf(buffer, "%s\n", bufferWin);
            // send(socketfd, buffer, strlen(buffer), 0);
```

```
        }
        // consultar ready queue
        // detener simulacion

        bzero(bufferWin, 1024);
    }

    flagRun = 0;
    endwin();
    exit(EXIT_FAILURE);
    pthread_detach(pthread_self());
}


void *start_time_thread(void *arg) {
    while(stop){
        sleep(1);
    }

    while(stop == 0){
        sleep(1);
        pthread_mutex_lock(&cpu_mutex);
        TIMESF++;
        pthread_mutex_unlock(&cpu_mutex);
    }

    pthread_t time_id;
    pthread_create(&time_id, NULL, start_time_thread, NULL);

    pthread_detach(pthread_self());
}

int main(int argc, char **argv) {

    if(argc < 2){
        printf("\nERROR: Missing Algorithm argument");
        printf("\nExample run: ./server FIFO\n");
        printf("\nExample run: ./server SJF\n");
        printf("\nExample run: ./server HPF\n");
        printf("\nExample run: ./server RR 2\n"); //Round Robin
        return EXIT_FAILURE;
    }

    algorithm = argv[1];

    int server_fd, new_socket;
```

```
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) <
0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    WINDOW *input, *output;
    initscr();
    cbreak();
    echo();
    input = newwin(1, COLS, LINES - 1, 0);
    output = newwin(LINES - 1, COLS, 0, 0);
    wmove(output, LINES - 2, 0);    /* start at the bottom */
    //scroll(output);
    scrollok(output, TRUE);

    Win *window = (Win *)malloc(sizeof(Win));
    window->input = input;
    window->output = output;

    pthread_t thread_id;
    pthread_create(&thread_id, NULL, window_thread, (void*)window);


    pthread_t time_id;
    pthread_create(&time_id, NULL, start_time_thread, NULL);


    pthread_t cpu_id;
```

```
    CPUINFO *cpuinf = (CPUINFO *)malloc(sizeof(CPUINFO));
    cpuinf->RQ = &RQ;
    cpuinf->FQ = &FQ;
    cpuinf->output = output;

    char message[100];
    // cases for the different algorithms and compare the text
    if (strcmp(algorithm, "FIFO") == 0) {
        sprintf(message, "\nServer: FIFO\n");
        waddstr(output, message);
        wrefresh(output);
        mvwprintw(input, 0, 0, "Command: ");
        pthread_create(&cpu_id, NULL, fifo, (void*)cpuinf);
    } else if (strcmp(algorithm, "SJF") == 0) {
        sprintf(message, "Server: SJF\n");
        waddstr(output, message);
        wrefresh(output);
        mvwprintw(input, 0, 0, "Command: ");
        pthread_create(&cpu_id, NULL, shortestJobFirst, (void*)cpuinf);
    } else if (strcmp(algorithm, "HPF") == 0) {
        sprintf(message, "Server: HPF\n");
        waddstr(output, message);
        wrefresh(output);
        mvwprintw(input, 0, 0, "Command: ");
        pthread_create(&cpu_id, NULL, highestPriorityFirst,
(void*)cpuinf);
    } else if (strcmp(algorithm, "RR") == 0) {

        int rrQ = atoi(argv[2]);
        cpuinf->rrQ = rrQ;
        sprintf(message, "Server: RR\n");
        waddstr(output, message);
        wrefresh(output);
        mvwprintw(input, 0, 0, "Command: ");
        pthread_create(&cpu_id, NULL, roundRobin, (void*)cpuinf);
    } else {
        sprintf(message, "Server: Invalid algorithm\n");
        waddstr(output, message);
        wrefresh(output);
        mvwprintw(input, 0, 0, "Command: ");
        return 0;
    }

    while (flagRun) {
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                                 (socklen_t *)&addrlen)) < 0) {
```

```
            perror("accept");
            exit(EXIT_FAILURE);
        }
        pthread_t thread_id;
        Connection *connection = (Connection
*)malloc(sizeof(Connection));
        connection->sock_fd = new_socket;
        connection->new_socket = server_fd;
        connection->win = window;
        pthread_create(&thread_id, NULL, handle_connection,
(void*)connection);
    }

    shutdown(server_fd, SHUT_RDWR); // closing the listening socket

    return 0;
}

// desplegar el ready queue
```

Anexo 6 Cliente Automatico

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <ctype.h>
#include <limits.h>
#include "jobstruct.h" //Custom Job struct

#include <ncurses.h>  //ADD AT THE END OF COMPILING THE FILE -lncurses
#include <pthread.h>
#include <signal.h>
#include <errno.h>   // for errno
#include <time.h>

#define PORT 8080

volatile sig_atomic_t flag = 0;

// Procs
typedef struct Procs{
```

```
    int rate;               // rate
    int maxBu;                  // max burst
    int sock_fd;                // socket
} Procs;


int create_socket() {
    int sock_fd = 0;
    if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) { //
socket(domain, type, protocol)
        printf("Client creation error\n");
        exit(EXIT_FAILURE);
    }
    printf("/ / / Client started / / /\n");
    return sock_fd;
}

void init_server_address(struct sockaddr_in *serv_addr) {
    memset(serv_addr, '0', sizeof(*serv_addr));
    serv_addr->sin_family = AF_INET; // IPv4
    serv_addr->sin_port = htons(PORT);
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr->sin_addr) <= 0) {
        printf("Invalid address/ Address not supported\n");
        exit(EXIT_FAILURE);
    }
}

void connect_to_server(int sock_fd, struct sockaddr_in serv_addr) {
    if (connect(sock_fd, (struct sockaddr *)&serv_addr,
sizeof(serv_addr)) < 0) {
        printf("Connection Failed\n");
        exit(EXIT_FAILURE);
    }
}


void *send_job_aut(void *arg){
    time_t current_time;
    time(&current_time);
    srand(current_time);

    Procs *procs = (Procs *)arg;
    Job job = {rand() % procs->maxBu + 1, rand() % 5 + 1};

    if (send(procs->sock_fd, &job, sizeof(job), 0) < 0) {
        printf("Send proc failed\n");
```

```
        exit(EXIT_FAILURE);
    }
    printf("Job sent - burst: %d - priority: %d\n", job.burst,
job.priority);
    int pid;
    if (recv(procs->sock_fd, &pid, sizeof(pid), 0) == -1) {
        printf("Receive failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Job recv - pid: %d \n", pid);
    pthread_detach(pthread_self());
}

int main(int argc, char **argv) {

    if(argc != 3){
        printf("\nERROR: Missing Creation Rate or Max Burs");
        printf("\nExample run: ./automatic_client 4 10\n");
        return EXIT_FAILURE;
    }

    int rate = atoi(argv[1]);
    int burst = atoi(argv[2]);

    int sock_fd = create_socket();
    struct sockaddr_in serv_addr;
    init_server_address(&serv_addr);
    connect_to_server(sock_fd, serv_addr);

    Procs * procs = (Procs *)malloc(sizeof(Procs));
    procs->rate = rate;
    procs->maxBu = burst;
    procs->sock_fd = sock_fd;

    while (true) {
        sleep(rand() % procs->rate + 1); // 1 - rate segs
        pthread_t proc_thread;
        if(pthread_create(&proc_thread, NULL, &send_job_aut,
(void*)procs) != 0){
            printf("\e[91;103;1m Error pthread send  proc\e[0m\n");
            return EXIT_FAILURE;
        }
    }

    close(sock_fd);
```

```
    return 0;
}
```

Anexo 7 Cliente Manual

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <ctype.h>
#include <limits.h>
#include "jobstruct.h" //Custom Job struct

#include <ncurses.h>  //ADD AT THE END OF COMPILING THE FILE -lncurses
#include <pthread.h>
#include <signal.h>
#include <errno.h>   // for errno

#define PORT 8080

volatile sig_atomic_t flag = 0;

// Procs
typedef struct Procs{
    int priority;                   // priority
    int burst;                  // burst
    int sock_fd;                // socket
} Procs;


int create_socket() {
    int sock_fd = 0;
    if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) { //
socket(domain, type, protocol)
        printf("Client creation error\n");
        exit(EXIT_FAILURE);
    }
    printf("/ / / Client started / / /\n");
    return sock_fd;
```

```
}

void init_server_address(struct sockaddr_in *serv_addr) {
   memset(serv_addr, '0', sizeof(*serv_addr));
   serv_addr->sin_family = AF_INET; // IPv4
   serv_addr->sin_port = htons(PORT);
   if (inet_pton(AF_INET, "127.0.0.1", &serv_addr->sin_addr) <= 0) {
       printf("Invalid address/ Address not supported\n");
       exit(EXIT_FAILURE);
   }
}

void connect_to_server(int sock_fd, struct sockaddr_in serv_addr) {
   if (connect(sock_fd, (struct sockaddr *)&serv_addr,
sizeof(serv_addr)) < 0) {
       printf("Connection Failed\n");
       exit(EXIT_FAILURE);
   }
}

void *send_job_man(void *arg){
   sleep(2);
   Procs *procs = (Procs *)arg;
   Job job = {procs->burst, procs->priority};

   if (send(procs->sock_fd, &job, sizeof(job), 0) < 0) {
       printf("Send proc failed\n");
       exit(EXIT_FAILURE);
   }
   printf("Job sent - burst: %d - priority: %d\n", job.burst,
job.priority);
   int pid;
   if (recv(procs->sock_fd, &pid, sizeof(pid), 0) == -1) {
       printf("Receive failed\n");
       exit(EXIT_FAILURE);
   }
   printf("Job recv - pid: %d \n", pid);
   pthread_detach(pthread_self());
}

int main(int argc, char **argv) {

   if(argc != 2){
       printf("\nERROR: Missing Path to file");
       printf("\nExample run: ./manual_client file.txt\n");
       return EXIT_FAILURE;
```

```
}

char *filename = argv[1];

int sock_fd = create_socket();
struct sockaddr_in serv_addr;
init_server_address(&serv_addr);
connect_to_server(sock_fd, serv_addr);



char line[100];

FILE* fp = fopen(filename, "r");

if (fp == NULL) {
    perror("Error opening file");
    //exit(EXIT_FAILURE);
    return EXIT_FAILURE;
}

while (!feof(fp)) {
    fgets(line, sizeof(line), fp);
    line[strcspn(line, "\n")] = '\0'; // remove newline character
    //printf("%s\n", line);

    if(isdigit(line[0])){
        //printf("%s\n", line);
        char b[100], *p;
        char *temp;
        strcpy(b, line);
        strtok_r(b, " ", &p);
        //printf("%s <> %s\n", b, p);

        sleep(rand() % 6 + 3); // 3 - 8 segs

        //define struct job
        Job job = {strtol(b, &temp, 10), strtol(p, &temp, 10)};

        Procs * procs = (Procs *)malloc(sizeof(Procs));
        procs->sock_fd = sock_fd;
        procs->burst = job.burst;
        procs->priority = job.priority;

        pthread_t proc_thread;
        if(pthread_create(&proc_thread, NULL, &send_job_man,
```

```
(void*)procs) != 0){
                printf("\e[91;103;1m Error pthread send  proc\e[0m\n");
                return EXIT_FAILURE;
            }
        }

    }
    sleep(12);
    fclose(fp);

    close(sock_fd);

    return 0;
}
```