Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

IC6600: Principios de sistemas operativos

Proyecto 1: Simulador de algoritmos de CPU scheduling

Profesora Ericka Solano

Estudiantes:
Luis Daniel Cordero
Jose Pablo Murillo
Roger Villalobos

Abril, 2018

Introducción

Los algoritmos de planificación de CPU son la base de los sistemas operativos multiprogramados. Mediante la comunicación de CPU entre varios procesos, el sistema operativo puede hacer que la computadora sea más productiva.

La multiprogramación tiene como objetivo poder tener continuamente varios procesos en ejecución, esto para maximizar el uso de CPU. Cuando un proceso debe esperar el sistema operativo retira el uso de CPU del proceso y se lo cede a otro proceso.

La planificación de procesos puede ser cooperativa, donde un proceso mantiene la CPU hasta que este termine o pase a estado de espera, o puede ser apropiativa, donde el planificador puede desalojar el proceso durante su ejecución y cambiarlo por otro.

El módulo de despachador es el que le proporciona el control de CPU a los diferentes procesos seleccionados por el planificador.

Los algoritmos de planificación de CPU toman diferentes criterios en cuenta a la hora de diseñarse. Dichos criterios son:

- Utilización de CPU: se busca tener la CPU tan ocupada como sea posible
- **Tasas de procesamiento**: número de procesos ejecutados por unidad de tiempo
- **Tiempo de ejecución**: tiempo desde que se manda a realizar ejecución hasta que termine
- **Tiempo de espera**: tiempo que proceso espera a ser ejecutado
- **Tiempo de respuesta**: tiempo que tarda en devolver la primera respuesta

Los algoritmos de planificación que se estudiaron para la elaboración de este proyecto son:

- FCFS (First Come First Server): donde se asigna CPU al primero que lo solicite. Los procesos a como van llegando se van agregando al final de la cola.

- SJF (Shortest Job First): cuando el CPU está libre se asigna al proceso cuyo burst sea el menor

- HPF (Highest Priority First): donde se asocia un valor de prioridad a cada proceso. Cuando el CPU está libre, se asigna al proceso cuyo valor de prioridad sea el menor.

- Round Robin: se usa un valor especificado de quantum donde cada proceso solo se puede ejecutar por ese quantum. En caso de que no termina de ejecutarse se vuelve a agrear al final de la cola.

El objetivo que se buscaba de este proyecto era crear un simulador de atención de procesos del sistema operativo bajo los diferentes algoritmos de planificación. Como requerimiento se pidió que sirviera bajo la modalidad cliente-servidor. En esta modalidad, el cliente es quien crea procesos (ya sean aleatorios o leídos de un archivo) y se los manda al servidor para que este los agregue en su cola de procesos y los ejecute de acuerdo al algoritmo especificado por el usuario. El usuario debe recibir una notificación de cuando su proceso enviado al servidor es agregado a la cola de procesos y se le es asignado un ID.

Estrategia de solución

A. Librerías utilizadas

Para la solución diseñada se utilizaron las siguientes bibliotecas y funciones incluidas:

- stdlib.h
    - atoi: para convertir un array de caracteres en un entero
    - malloc: para reservar espacio de memoria para una variable
    - free: para liberar la memoria previamente pedida para reservar
    - exit: causa que el programa termine su ejecución

- stdio.h
    - Usado para realizar operaciones relacionadas con el manejo de I/O
    - fopen y fclose para la manipulación y lectura de arcviso
    - printf: para mostrar datos en la terminal
    - fgets: usado para leer de una entrada y guardar el resultado en un arreglo.

- string.h
    - Usado para manipular de manera eficiente los arreglos de caracteres

    - strcpy: para copiar un arreglo a un buffer de datos
    - strcat: para concatenar hileras
- ctype.h
    - Usado para hacer operaciones sobre caracteres
    - isdigit: para saber si un caracter es numérico

- arpa/inet.h y sys/socket.h
    - Usados para montar un socket para permitir la comunicación entre la terminal de cliente y la del servidor, por default se conecta a la dirección 127.0.0.1
    - send: para mandar un mensaje entre cliente servidor
    - recv: para recibir un mensaje entre cliente servior

- unistd.h y pthread.h
    - Usados para la definición y creacion de hilos con sus respectivas funciones.
    - pthread_create: para crear un hilo
    - pthread_join: para agregar el hilo al programa y empezar a ejecutarlo

B. Organización de archivos

- makefile: usado para realizar la compilación de ambos programas en una sola corrida

- funcionesAuxiliares.c: contiene funciones genéricas que pueden ser usadas en futuros proyectos como determinar si un arreglo de caracteres es numerico o para convertir un entero en un arreglo de caracteres numéricos

- client.c: incluye todas las funciones y definiciones de variables necesarias para la lógica y funcionamiento correcto del programa cliente.

- servidor.c: incluye todas las fucniones y definiciones de variables necesarias para la lógica y funcionamiento correcto del programa servidor.

    - Tanto el archivo client.c como servidor.c incluyen hilos para poder recibir datos por medios del socket y mostrarlos en pantalla

    - El archivo servidor.c ademas incluye hilos que hacen el proceso de jobScheduler y CPUScheduler

- ListaEspera.c: define la estructura de la lista de espera, el PCB y las funciones que realizan operaciones sobre la lista de espera.

- El PCB diseñado contiene la siguiente información:
  - id
  - burst
  - burstFaltante (esto para llevar lo que falta por procesar en el RR)
  - Prioridad
  - TAT (tiempo total hasta el momento)
  - WT (Tiempo que lleva esperando)

Análisis de resultados:

Tarea/ funcionalidad	Porcentaje completado	Justificación
Conexión de aplicación cliente con aplicación servidor	100%	El programa se conecta con éxito cuando el servidor está levantado
Se pide al usuario un rango de burst de los procesos	100%	Los datos de burst mínimo y máximo son ingresados con éxito y se guardan los valores dentro de la aplicación cliente. Además se realiza una validación de datos para que la aplicación se salga en caso de ser inválidos
Se realiza el modo manual donde se le pide al cliente un archivo de entrada y se va leyendo de este para solicitar crear un proceso al servidor con los datos leídos. Además se realiza un sleep entre cada línea de archivo procesada.	100%	El archivo se pide por medio de línea de comandos y se hace validación de que el archivo se pueda abrir, de lo contrario se sale de la aplicación. El hilo que procesa la lectura de archivo realiza un escaneo de la línea actual y luego manda a procesar la línea para determinar si los datos son válidos.

		En caso de que el dato de burst en la línea no tenga un dato válido para prioridad, se le asigna por defecto un 5. En caso de que el dato de burst sea inválido, la línea se ignora y se continua leyendo del archivo.
Se realiza el modo automático donde se pide una tasa de creación de procesos y se crean procesos aleatorios hasta que el usuario termine la función	100%	Se crean datos para el burst y prioridad aleatorios y se manda a crear un proceso al servidor. La creación de procesos termina una vez que el usuario cierra la terminal Se realiza una validación del parámetro de tasa de creación donde la entrada debe ser un número entero, de lo contrario la aplicación termina de correr.
Se le retorna al cliente un ID del proceso creado por parte del servidor	100%	Una vez que se han mandado los datos al servidor, este crea un PCB y lo agrega dentro de la lista, se manda un ID de proceso creado al cliente que corresponde a una variable que incrementa cada vez que se crea un PCB. El mensaje dice “ID proceso: <PID> fue agregado con exito”
Al momento de levantar el servidor, se le pide al usuario el modo en que desea correr el simulador.	100%	Se presenta un menú con las opciones de algoritmos de planificación y se realiza

		una validación de datos en caso de que la opción no es válida. En caso de que la opción no es válida, la aplicación servidor termina su ejecución.
Se implementa la creación y agregado de procesos a una estructura de datos (JobScheduler)	100%	Se crean nodos tipo PCB dentro de una lista enlazada que sirve como la cola de procesos en espera de ser ejecutados El hilo de jobScheduler está esperando a recibir datos de la entrada estándar por medio de un socket.
Se implementa el algoritmo de FCFS en la aplicación servidor	100%	Los procesos son agregados al final de la cola sin tomar en cuenta el burst ni la prioridad
Se implementa el algoritmo SFJ en la aplicación servidor	100%	Los procesos que van a ser agregados se les valida el burst para determinar dónde colocarlos en la cola
Se implementa el algoritmo HPF en la aplicación servidor	100%	Los procesos que van a ser agregados se les valida la prioridad para determinar donde colocarlos en la cola.
Se implementa el algoritmo Round Robin en la aplicación servidor	100%	Se pide y se valida el valor de quantum al usuario. Los procesos son agregados pero en caso de no terminar de ejecutarse en el quantum especificado, se agregan al final de la cola.
Se implementa el context switch cuando un proceso	100%	La aplicación servidor muestra el mensaje:

entra en ejecución		“Ejecutando proceso: <PID> con burst de <burst> y prioridad< prioridad>
Cuando un proceso ha terminado su ejecución, se muestra en pantalla la finalización	100%	La aplicación servidor muestra el mensaje: “finalizó proceso: <PID>”
Se muestra la cola de procesos en espera cuando el usuario lo solicite al servidor	100%	Por medio del comando “M” se muestran los procesos que aún están en la lista de espera
El planificador de CPU puede dejar de correr en cualquier momento si el usuario lo especifica. En ese momento se muestra en pantalla: - cantidad de procesos ejecutados - Cantidad de segundos con el CPU ocioso - Tablas de TAT (Turnaround time) y WT (Waiting time) para cada proceso ejecutado - Promedio total de waiting time	100%	Por medio del comando “S”, la aplicación servidora termina de correr y por lo tanto no corre ningún otro hilo. Se muestra la cantidad de procesos, los segundos que el procesador estaba ocupado y, el promedio de tiempo de espera y todos los procesos terminados con su tiempo total en el procesador más su tiempo de espera.

Lecciones aprendidas

A nivel técnico:

- Se aprendió el uso e implementación de sockets en C para que se puedan desarrollar aplicaciones más complejas donde se pueden ejecutar dos programas al mismo tiempo y estos puedan intercambiar información entre ellos.

- Se aprendió el uso e implementación de hilos para aprovechar los recursos de la máquina y poder ejecutar procesos simultáneos dentro de la misma aplicación. Esto ayuda a repartir ciertas tareas (como recibir de la entrada estándar, agregar procesos, ejecutar procesos, etc.) a subprogramas para que estos solo se enfoquen en ejecutar su tarea.
- Se aprendió el uso e implementación de algunas funcionalidades de string.h lo cual facilitó la manipulación de arreglos de caracteres.

A nivel personal

- El uso de documentación interna dentro del código ayuda a que los otros integrantes del grupo sepan sobre el funcionamiento de las funciones, lo que se espera que reciba y devuelva. La documentación interna y nombres significativos ayuda a que los integrantes puedan entender con facilidad un trozo de código.

- La decisión mutua de los integrantes sobre separar cada subsistema del programa ayuda a que se puedan asignar tareas específicas y así es más fácil detectar la fuente de los errores.

- Se hacían reuniones, ya sean presenciales o por medio de messenger, donde se definía lo que se debía modificar para que cualquier integrante tuviera la capacidad y entendimiento sobre realizar los cambios discutidos. Esto ayudaba a aligerar la carga entre los integrantes debido a que no podían dedicarle tiempo al desarrollo durante algunos días debido a su involucramiento con otros proyectos y clases. Las reuniones ayudaron a explorar posibilidades de soluciones y fomentaba la participación de los integrantes.

- El uso de un repositorio ayudaba a llevar un control sobre los cambios realizados y también saber quién era el responsable por cambiar el código para que si ocurría algo, el responsable debía cambiarlo o al menos informar a los demás sobre qué estaba intentando realizar para poder cooperar y avanzar con el desarrollo.

- El printf() es nuestro mejor amigo para realizar depuración y pruebas de funciones.

Casos de prueba

Prueba número 1: Funcionamiento correcto de modo manual de la aplicación cliente

Entrada	Resultado esperado	Resultado actual
“prueba”	“Oops parece que ese archivo no pudo ser accesado”	“Oops parece que ese archivo no pudo ser accesado”
“prueba.txt”	Se muestra el mensaje: “Analizando línea de archivo” y en caso de mandar a crear un proceso se muestra el mensaje: “ID proceso: <PID> fue agregado con éxito”	Se muestra el mensaje: “Analizando línea de archivo” y en caso de mandar a crear un proceso se muestra el mensaje: “ID proceso: <PID> fue agregado con éxito”

Prueba número 2: funcionamiento de algoritmo FCFS en el servidor cuando se usa modo manual y el archivo: prueba.txt

Resultado esperado	Resultado actual
“Ejecutando proceso 0 con burst de 8 y prioridad 3 finalizo proceso: 0 Ejecutando proceso 11 con burst de 7 y prioridad de 2 Finalizo proceos: 1”	“Ejecutando proceso 0 con burst de 8 y prioridad 3 finalizo proceso: 0 Ejecutando proceso 11 con burst de 7 y prioridad de 2 Finalizo proceos: 1”

Comparación

Los hilos creados en C ofrecen una mayor cantidad de opciones para la sincronización de procesos. Entre las opciones están: semáforos, mutex, locks de lectura-escritura, entre otros. Los hilos en C se prefieren sobre los de Java debido a su eficiencia y cercanía con el sistema operativo.

Los hilos en Java son menos eficientes ya que Java es un lenguaje de más alto nivel debido a la máquina virtual JVM, esto hace que esté más lejos del sistema operativo. Los mecanismos de sincronización que use dependen del sistema operativo en el que esté utilizando. Java también implementa opciones de sincronización pero estas deben ser traducidas dependiendo del sistema operativo. Hay una relación muchos a muchos entre los hilos de Java y los hilos del sistema operativo.

Bitácora

Fecha	Actividad realizada

26/03/2018	Implementación de aplicación cliente donde se piden datos de entrada. Implementación inicial de servidor y socket de comunicación
27/03/2018	Implementación inicial donde la información sobre burst y proceso se mandan por medio de socket. Implementación de modo manual y modo automático
05/04/2018	Definición de funciones y archivo listaEspera.c
07/04/2018	Implementación de algoritmos de planificación. Mejora en separación de archivos cliente y servidor. Implementación de hilo JobScheduler
08/04/2018	Corrección de lectura de datos de archivo en modo manual

Bibliografía

Alfonso, R. (Producer). (2015, August 14). *Sockets (parte 1/4): Crear un servidor*[Video file]. Accesado Marzo 26, 2018, de https://www.youtube.com/watch?v=Zd_YJtVScuA

Multithreading in C. (2017, December 28). Accesado Marzo 28, 2018, de https://www.geeksforgeeks.org/multithreading-c-2/

C Linked List. (n.d.). Accesado Abril 5, 2018, de http://www.zentut.com/c-tutorial/c-linked-list/#Add_a_node_at_the_beginning_of_the_linked_list