Universidad Simón Bolívar
CI4821 - Sistemas de Operación II
Septiembre-Diciembre 2019
Profesora: Yudith Cardinale, PhD

Simulador del Proceso de Administración de CPU de Linux
Entrega Final

Integrantes:
Gustavo Castellanos
Constanza Abarca
Rubmary Rojas

Descripción del *scheduler* de Linux

Desde la versión 2.6.23 del kernel de Linux, de Octubre de 2007, el planificador o *scheduler* de procesos de Linux utiliza el **Completely Fair Scheduler** (CFS) para darle fracciones de tiempo equitativas a los distintas procesos del usuario y el sistema.

Para asegurar la equitatividad del tiempo de CPU, cuando un proceso está "fuera de balance" (lleva menos tiempo de ejecución) se le asigna tiempo de CPU. Para saber cuando un proceso está fuera de balance, se utiliza un valor asignado a él llamado *Virtual Runtime* el cual consiste del tiempo total de ejecución que ha tenido el proceso multiplicado por un peso determinado por la prioridad del mismo (entre menos prioridad, más crece su virtual runtime). Los procesos tienen prioridad estática, asignada en la creación, y prioridad dinámica.

Para planificar a los procesos y escoger el siguiente proceso a asignarle el CPU, se utiliza un Árbol Rojo-Negro [1] cuyos índices son los virtual runtime o `vruntime`. El siguiente proceso a ser ejecutado es el más a la izquierda, el cual es borrado del árbol, se le calcula el nuevo `vruntime` y es insertado de nuevo en el árbol.

Un proceso tiene como tiempo máximo de ejecución el tiempo que ha estado esperando para ser ejecutado dividido entre el número de procesos.

Las tareas en el sistema operativo Linux se representan con la estructura `task_struct`, la cual incluye el estado del proceso, su identificador, su padre, sus hijos, sus archivos abiertos, su pila, sus flags, su prioridad estática y dinámica, y mucha más información. Además tiene un atributo de tipo `sched_entity`, el cual almacena el nodo del árbol rojo-negro correspondiente al proceso.

Linux implementa el ***load balancing*** o *equilibrio de carga* utilizando tanto *pull migration* (un procesador sin procesos pendientes toma un proceso que esperaba por un procesador ocupado) como *push migration* (una tarea específica del sistema periódicamente mueve procesos de procesadores muy sobrecargados a procesadores con menos carga).

Implementación del simulador

Entrada

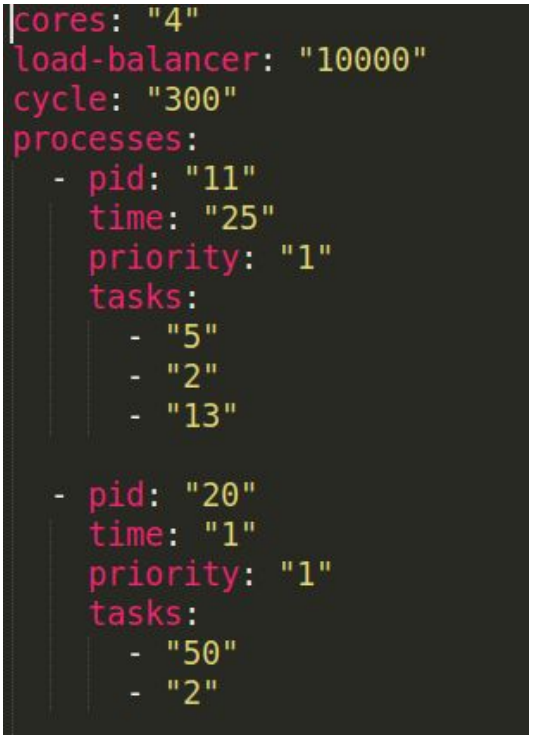
La entrada se proporcionan mediante un archivo en formato Yaml, el nombre del archivo se debe proporcionar como argumento al ejecutar el programa. Si el archivo existe en la carpeta correspondiente será utilizado como entrada, en caso contrario se creará un archivo con los parámetros especificados en los argumentos o los parámetros por defecto. Estos parámetros son:

- Nombre del archivo: `args.yaml` por defecto.
- Número de CPUs: 4 por defecto.
- Número de procesos: 15 por defecto.
- Tiempo de espera para realizar balanceo de carga: 5000 por defecto
- Tiempo de espera entre cada ciclo del reloj: 300 por defecto.

Si el archivo con el nombre especificado no existe, es creado utilizando la `CreateFile` , con los parámetros proporcionados. Cada proceso tiene la siguiente información:

- `pid`: process id
- `time`: tiempo de creación del proceso
- `priority`: prioridad, un entero entre 1 y 10.
- `tasks`: número de ciclos de reloj y ciclos de entrada/salida de forma alternada.

Esta información es creada de forma aleatoria, salvo el id de los procesos que es creada de forma secuencial.



Sistema Operativo

El sistema operativo está representado por la clase `OperatingSystem`, recibe como parámetros los cpus, la información para crear los procesos, el reloj y el resource. El sistema operativa crea el `timer` (el hilo para inicializa el reloj y lo incrementa según el tiempo establecido) y crea los procesos según el tiempo de inicialización de cada uno sincronizado al reloj del `timer`. Cuando un proceso es creado es agregado al CPU que tenga una carga más baja (la carga es medida según la cantidad de procesos que estén utilizando/esperando al CPU). Además, también inicializa el `loadBalancer`, que se encarga de balancear las cargas entre los CPU's cada cierta cantidad de tiempo especificada en el archivo YAML.

CPUs

Para representar a los distintos CPUs disponibles se utilizará la clase `CPU`. Esta clase crea un hilo por cada CPU instanciado. Tiene como atributo principal `processTree`, que es un árbol rojo-negro que incluye los procesos esperando por este CPU ordenados según el `vruntime`. Cada CPU obtiene el proceso que se encuentre en el árbol con el `vruntime` de menor valor para asignarle tiempo de CPU. Un proceso tiene como tiempo máximo de ejecución el tiempo que ha estado esperando para ser ejecutado dividido entre el número de proceso o el tiempo que necesite antes de necesitar por recurso. Si el CPU se queda libre, inicia *Pull Load Balancing*, donde intenta tomar un proceso del CPU con más carga.

Procesos

Para representar a los distintos procesos del sistema se utilizará la clase `Process`, la cual tendrá como atributos un entero `pid` que corresponde al identificador del proceso, `priority` su prioridad estática y `vruntime`. También se mantiene una *deque* o *cola doblemente terminada*, para llevar control de las tareas (especificadas en el YAML) pendientes del proceso. Los principales métodos son:

- `run(maxTimeToRun)` el cual es un ciclo que mantiene ocupado al proceso durante el tiempo `maxTimeToRun` o hasta que el proceso tenga que esperar por recursos (en este caso I/O).
- `waitForCPU()`: agrega el proceso al árbol del CPU que tiene asignado cuando el proceso está listo para seguir corriendo o cuando es interrumpida su ejecución en el CPU por alcanzar el tiempo máximo.
- `waitForResource()`: agrega el proceso a la cola de de los procesos que están esperando por el Recurso de I/O.

Push Load balancing

Como se mencionó previamente, uno de los atributos de la clase `OperativeSystem`, es una instancia de la clase `LoadBalancer`, el cual cada cierto tiempo equilibrará la carga

entre los procesadores, moviendo procesos del árbol de un CPU con mucha carga a otro árbol de un CPU con menos carga.

Resource

La clase Resource, la cual implementa `Runnable`, abstrae un recurso por el cual los procesos compiten, en este caso el recurso de I/O es el único que se tiene en el simulador. Esta clase posee un atributo llamado `pQueue`, que es una cola de los procesos que están esperando por el recurso, además de dos métodos:

- `enqueue(process)`: agrega un proceso a la cola de procesos esperando por el recurso.
- `run()`: ciclo que, en cada iteración, asigna el recurso al primer proceso en la cola durante el tiempo que tiene asignado.

Algunos monitores

Se implementaron varios monitores para poder realizar las distintas tareas entre los componentes del simulador. Entre los monitores se encuentran:

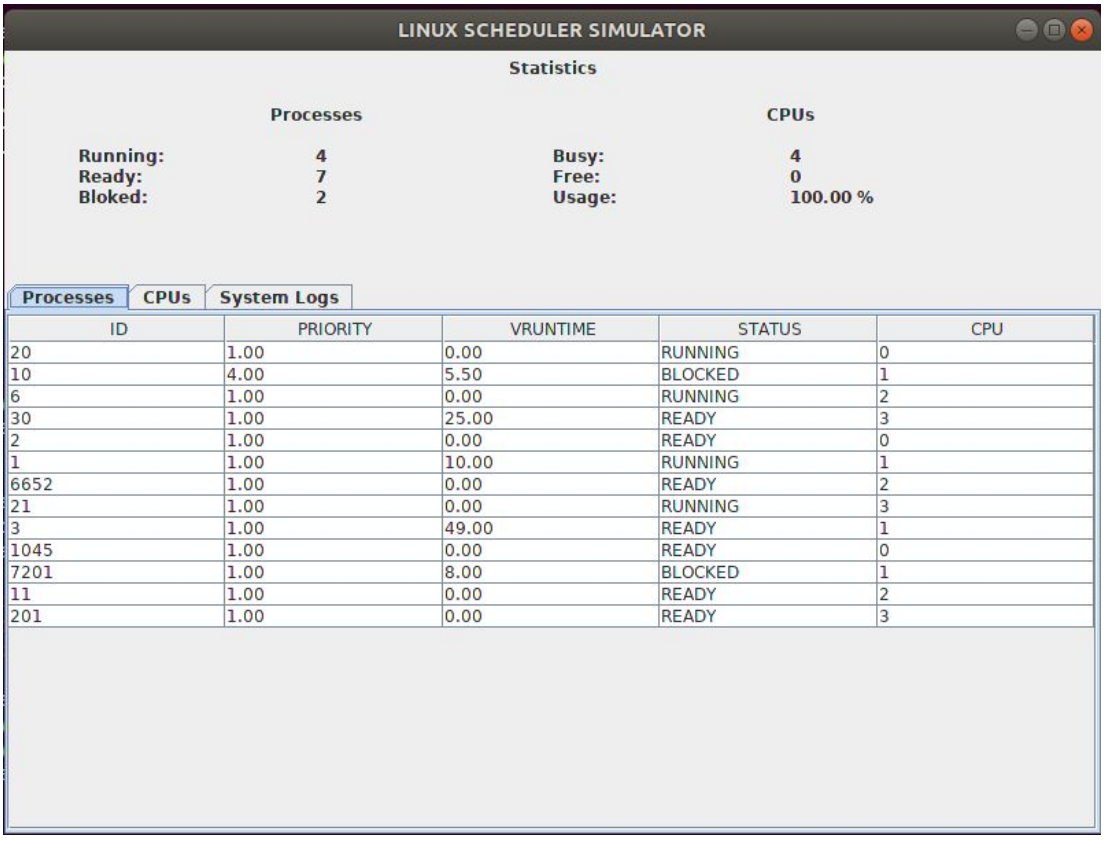
- `AllocatedCPUMonitor`: Una estructura que mapea cada proceso a su CPU asignado.
- `CPUTreeMonitor`: Un conjunto de CPUs ordenado por sus cargas.
- `StatusMapMonitor`: Una estructura que mapea cada proceso a su Status.

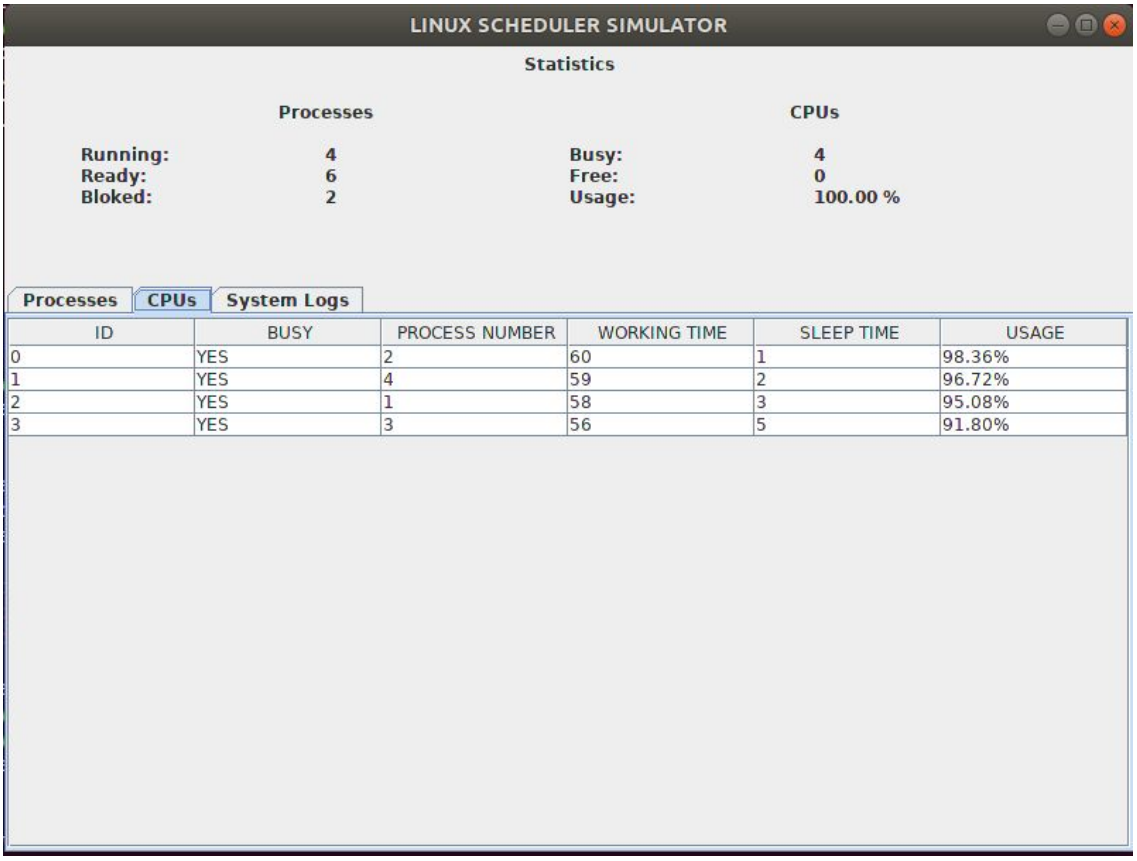
Interfaz Gráfica

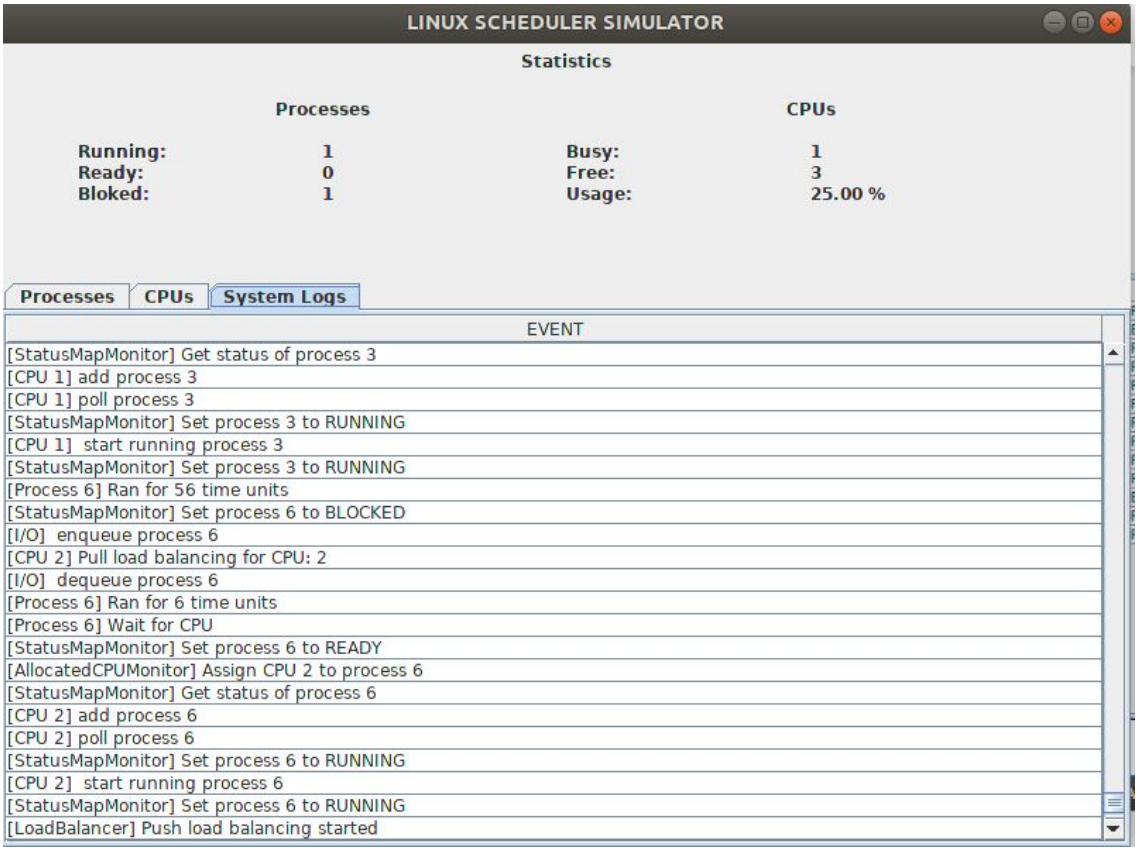
La interfaz gráfica consta de 4 partes: estadísticas generales que se encuentran en la parte superior de la ventana y 3 pestañas para indicar el estado de cada proceso, el estado de cada CPU y el log general del sistema.

Las estadísticas generales consta de dos partes: estadísticas generales de los procesos, donde se muestran la cantidad de procesos por cada estado, y estadísticas generales de los CPUs, donde se indican la cantidad de CPUs ocupados, la cantidad libre y el porcentaje de los CPUs en uso en el momento. Estas estadísticas se actualizan en cada ciclo del reloj del simulador.

La pestaña de procesos tiene una fila por cada proceso que fue creado en el sistema operativo, mostrando el pid, la prioridad, el vruntime, el status y el CPU asignado. Cuando el proceso termina es marcado como FINISHED en la columna de STATUS. La pestaña de CPUs muestra una fila por cada CPU, mostrando su ID, si está ocupado o no y la cantidad de procesos correspondientes al CPU, que están corriendo o esperando que se libere el CPU. Además muestra el tiempo que ha estado ocupado el CPU, el tiempo que ha estado libre y el porcentaje que ha estado en uso hasta ese momento. La última pestaña muestra el log de cada evento que ocurre.


LINUX SCHEDULER SIMULATOR
Statistics
Processes
Running: 4
Ready: 7
Bloked: 2
CPUs
Busy: 4
Free: 0
Usage: 100.00 %
Processes
CPUs
System Logs
ID PRIORITY VRUNTIME STATUS CPU
20 1.00 0.00 RUNNING 0
10 4.00 5.50 BLOCKED 1
6 1.00 0.00 RUNNING 2
30 1.00 25.00 READY 3
2 1.00 0.00 READY 0
1 1.00 10.00 RUNNING 1
6652 1.00 0.00 READY 2
21 1.00 0.00 RUNNING 3
3 1.00 49.00 READY 1
1045 1.00 0.00 READY 0
7201 1.00 8.00 BLOCKED 1
11 1.00 0.00 READY 2
201 1.00 0.00 READY 3


LINUX SCHEDULER SIMULATOR
Statistics
Processes
Running: 4
Ready: 6
Bloked: 2
CPUs
Busy: 4
Free: 0
Usage: 100.00 %
Processes
CPUs
System Logs
ID BUSY PROCESS NUMBER WORKING TIME SLEEP TIME USAGE
0 YES 2 60 1 98.36%
1 YES 4 59 2 96.72%
2 YES 1 58 3 95.08%
3 YES 3 56 5 91.80%


LINUX SCHEDULER SIMULATOR
Statistics
Processes
CPUs
Running: 1
Ready: 0
Bloked: 1
Busy: 1
Free: 3
Usage: 25.00 %
Processes
CPUs
System Logs
EVENT
[StatusMapMonitor] Get status of process 3
[CPU 1] add process 3
[CPU 1] poll process 3
[StatusMapMonitor] Set process 3 to RUNNING
[CPU 1] start running process 3
[StatusMapMonitor] Set process 3 to RUNNING
[Process 6] Ran for 56 time units
[StatusMapMonitor] Set process 6 to BLOCKED
[I/O] enqueue process 6
[CPU 2] Pull load balancing for CPU: 2
[I/O] dequeue process 6
[Process 6] Ran for 6 time units
[Process 6] Wait for CPU
[StatusMapMonitor] Set process 6 to READY
[AllocatedCPUMonitor] Assign CPU 2 to process 6
[StatusMapMonitor] Get status of process 6
[CPU 2] add process 6
[CPU 2] poll process 6
[StatusMapMonitor] Set process 6 to RUNNING
[CPU 2] start running process 6
[StatusMapMonitor] Set process 6 to RUNNING
[LoadBalancer] Push load balancing started

Bibliografía

1. *Introduction to Algorithms.* Thomas H.. Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein. 2001. Pág 308-339.