DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I - Schedulling

Sistemas Operativos
Primer Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Leandro Javier Raffo	945/12	
Abel Delgadillo	74/12	
Lautaro Alvarez	268/14	lautarolalvarez@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.dc.uba.ar

Índice

Ejercicio 1.

Para este punto escribimos el siguiente código.

```
void TaskConsola(int pid, vector<int> params) {
    size_t n = params[0];
    size_t bmin = params[1];
    size_t bmax = params[2];
    std::random_device rd;
    std::mt19937 mt(rd()); //Distribuye en el rango pedido
    std::uniform_int_distribution<int> dist(bmin, bmax);
    for (size_t i = 0; i < n; i++) {
        uso_IO(pid, dist(mt));
    }
}
```

Donde usamos el segundo y tercer parámetro del vector para obtener el rango de valores random de espera, y el primer parámetro para saber cuantas veces ibamos a esperar. Seedeamos el pseudo generador de números random mt19937 con random_device y luego generamos los mismos distribuidos uniformemente en el rango dado.

Probamos la tarea con el siguiente lote, usando el First Come First Served scheduler

```
TaskConsola 3 1 6
TaskConsola 5 1 24
TaskConsola 2 1 3
TaskConsola 8 1 7
```

Obteniendo el siguiente resultado

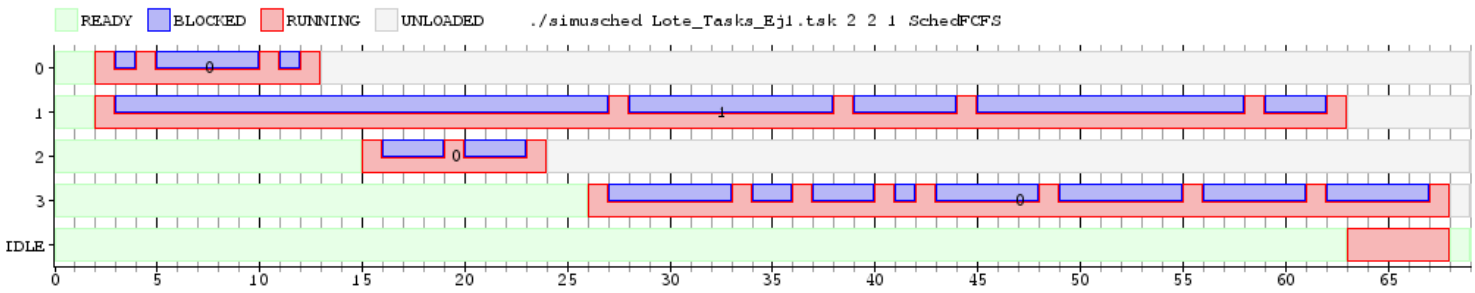


Figura 1: FCFS Scheduler con 4 tareas de consola.

1. Ejercicio 2

Las tareas que ejecuta el grupo de competencia de Data Mining se pueden escribir así:
TaskCPU 500
TaskConsola 10 1 4
TaskConsola 20 1 4
TaskConsola 30 1 4

La primer tarea simula el algoritmo que usa intensivamente la cpu, las otras tres simulan a los tres usuarios que se conectan al servidor. La simulación de las ejecuciones de estas tareas
para uno, dos y cuatro núcleos con un scheduler FCFS, son las siguientes respectivamente.

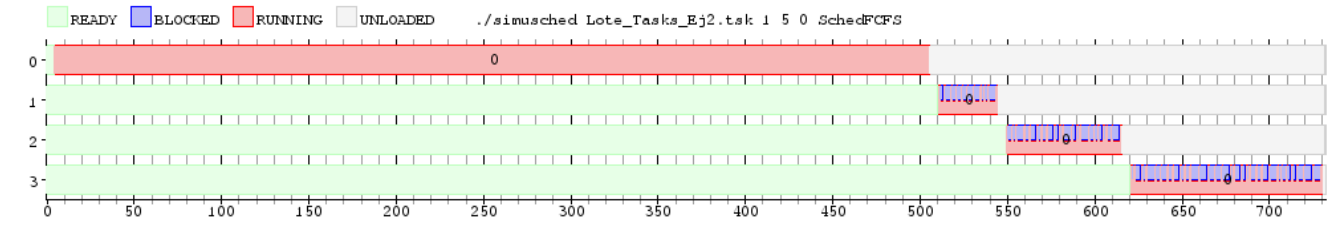


Figura 2: Ejecución con un núcleo

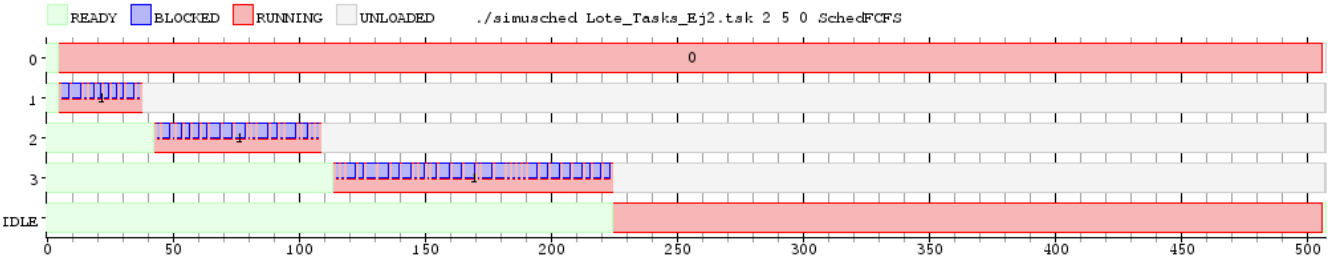


Figura 3: Ejecución con dos núcleos

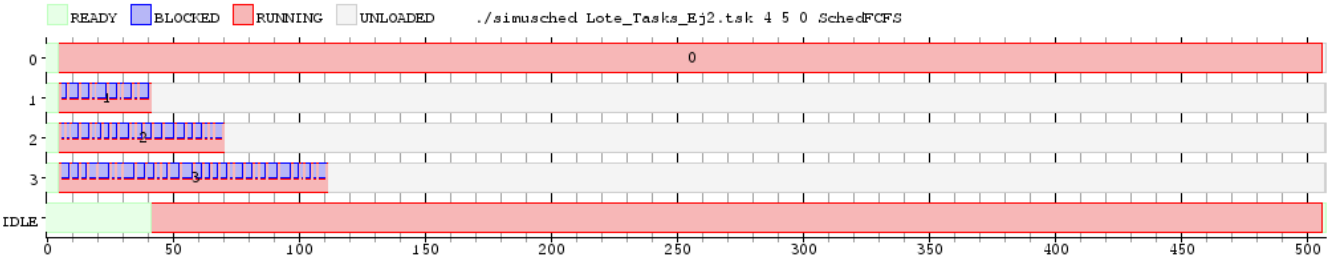


Figura 4: Ejecución con cuatro núcleos

Latencias de las tareas:

Con un núcleo:
Primera tarea: 5
Segunda tarea: 510
Tercera tarea: 550
Cuarta tarea: 620

Con dos núcleos:
Primera tarea: 5

Segunda tarea: 5
Tercera tarea: 42
Cuarta tarea: 112

Con cuatro núcleos:
Primera tarea: 5
Segunda tarea: 5
Tercera tarea: 5
Cuarta tarea: 5

En base a los gráficos obtenidos y los cálculos realizados, podemos concluir que First Come First Served no es una buena política de scheduling para ejecutar tareas interactivas.

2. Ejercicio 3.

Código de la tarea de batch

```
const char bloquea = 1;
const char no_bloquea = 0;

void TaskBatch(int pid, vector<int> params) {
    size_t total_cpu = params[0];
    size_t cant_bloqueos = params[1];
    // Vamos a guardar en un vector que indica si es cpu o IO
    std::random_device rd;
    std::mt19937 mt(rd()); //Distribuye en el rango pedido
    std::uniform_int_distribution<int> dist(0, total_cpu - 1);
    // Vector de bloqueos para saber en que ciclos bloquear
    // Si es true bloquea 4 ciclos de IO, sino es un ciclo de CPU.
    std::vector<char> bloqueos;
    bloqueos.reserve(total_cpu);

    // Llenamos el vector con los bloqueos necesarios y el resto con no bloqueos
    for (size_t i = 0; i < total_cpu; ++i) {
        if (i < cant_bloqueos)
            bloqueos.push_back(bloquea);
        else
            bloqueos.push_back(no_bloquea);
    }

    // Mesclamos los elementos del vector que contiene bloqueos y nobloqueos
    // usando el generador pseudo aleatorio
    for (size_t i = 0; i < total_cpu; ++i) {
        int temp;
        int idx = dist(mt);
        int idx2 = dist(mt);
        temp = bloqueos[idx];
        bloqueos[idx] = bloqueos[idx2];
        bloqueos[idx2] = temp;
    }

    for(size_t i = 0; i < bloqueos.size(); ++i) {
        if (bloqueos[i] == bloquea) {
            uso_IO(pid, 4);
        } else {
            uso_CPU(pid, 1);
        }
    }
}
```

Para este problema usamos el segundo parámetro, es decir el segundo valor del vector, para obtener el número de llamadas bloqueantes a realizar y el primer parametro para saber la cantidad de ciclos consumodos. Para asignar si va a ser una llamada bloqueante o uso de CPU, llenamos un vector de tamaño ciclos_cpu con la cantidad de llamadas bloqueantes pasadas en el segundo valor del vector por argumento y el resto con llamadas no bloqueantes a su vez vamos a usar el mismo prng del ejercicio 1, esta vez distribuyendo uniformemente sobre el tamaño del vector. Para obtener la aleatoriedad recorrimos hasta el final del vector tirando dos valores aleatorios por iteracion y swapeando los valores apuntados por esos indices.

Para probar la tarea batch escribimos el siguiente lote

```
TaskBatch 22 4
TaskBatch 10 2
TaskBatch 15 3
TaskBatch 30 7
```

Corriendolo sobre FCFS conseguimos

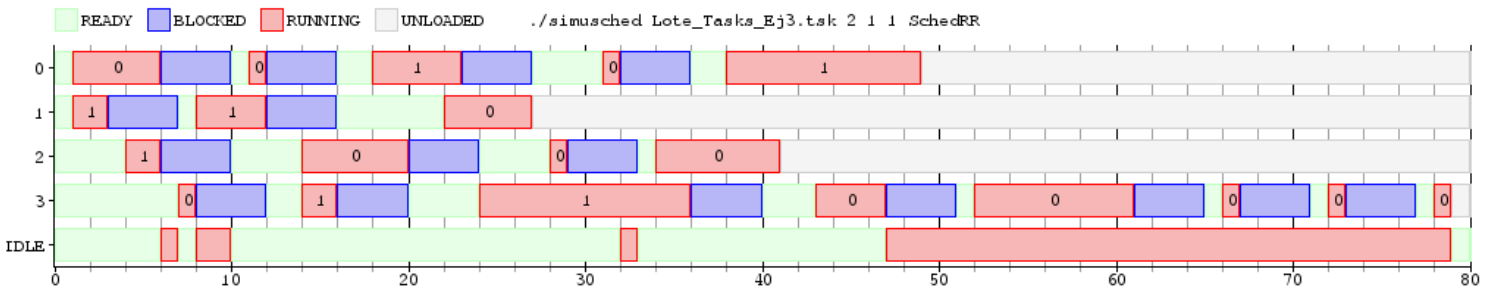


Figura 5: FCFS Scheduler con 4 tareas Batch.

3. Ejercicio 4.

Vamos a explicar brevemente la implementacion del sched_rr. Para la parte privada mantuvimos dos vectores, donde el índice marca un cpu y el valor marca, en uno de los vectores los ticks restantes y en el otro los ticks por defecto. Por otro lado tenemos una cola de tareas a ejecutar. Además si una tarea esta ejecutandosé, bloqueandosé o fue terminada esta no va a estar en la misma.
El constructor, destructor, unblock y load de sched_rr son tan simples que no hace falta explicación. Veamos la función tick.

```
int SchedRR::tick(int id_cpu, const enum Motivo m) {
    int pid_ret = IDLE_TASK;
    int pid_c = current_pid(id_cpu);
    if (m == TICK && pid_c != IDLE_TASK) {
        cpu_quantum[id_cpu]--;
        if (cpu_quantum[id_cpu] == 0 && !q.empty()) {
            q.push(pid_c);
            pid_ret = q.front();
            q.pop();
            cpu_quantum[id_cpu] = def_quantum[id_cpu];
        } else {
            pid_ret = pid_c;
            if (!cpu_quantum[id_cpu]) cpu_quantum[id_cpu] = def_quantum[id_cpu];
        }
    } else {
        if (!q.empty()) {
            pid_ret = q.front();
            q.pop();
            cpu_quantum[id_cpu] = def_quantum[id_cpu];
        }
    }
    return pid_ret;
}
```

Como se ve lo que hacemos es fijarnos si actualmente esta corriendo una tarea idle o pidió un block/exit. En tal caso lo único que hace falta es intentar cargar una nueva tarea al cpu, de no poder hacerlo, porque no hay tareas en la cola, dejamos la idle. Si era un tick y además no era la tarea idle, vemos si fue el ultimo tick, en tal si hay álguna tarea en la cola de procesos, la cargamos, y sino devolvemos la idle.

Luego para probar que andaba corrimos el scheduler con el loteEj5.tsk que es

```
TaskBatch 12 4
TaskCPU   34
TaskBatch 25 3
TaskCPU   8
```

Obteniendo

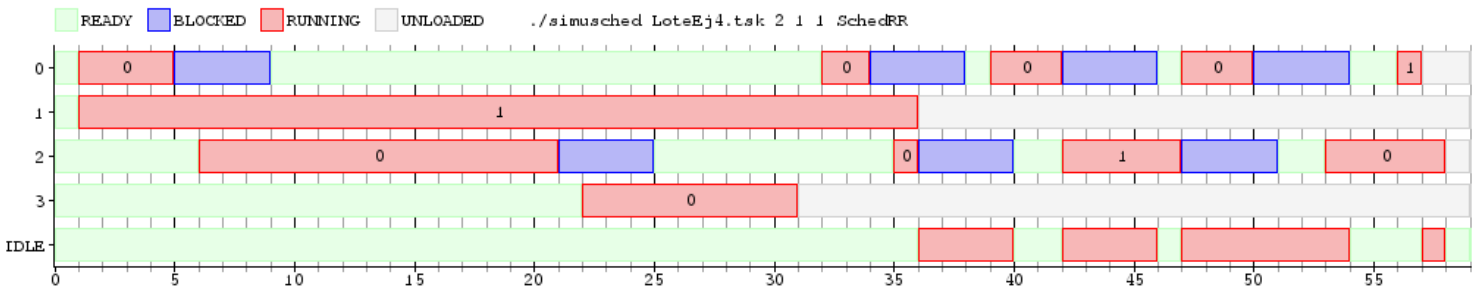


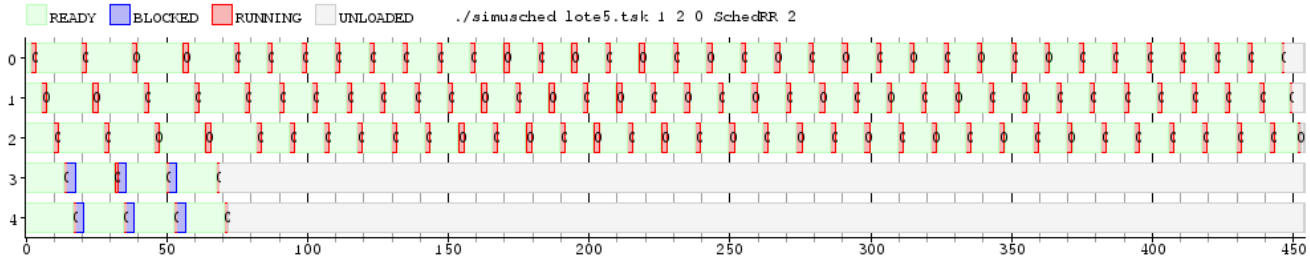
Figura 6: Round Robin Lote4.

3.1. Ejercicio 5

En este ejercicio se comparan algunas métricas del algoritmo de scheduling Round Robin con un sólo núcleo con distintos valores de quantum, ejecutando siempre el mismo lote:

TaskCPU 70
TaskCPU 70
TaskCPU 70
TaskConsola 3 3 3
TaskConsola 3 3 3

En la primer figura quantum = 2.



Métricas de las tareas

Tarea 1
Tiempo de espera: 376
Latencia: 2
Tiempo de ejecución: 446

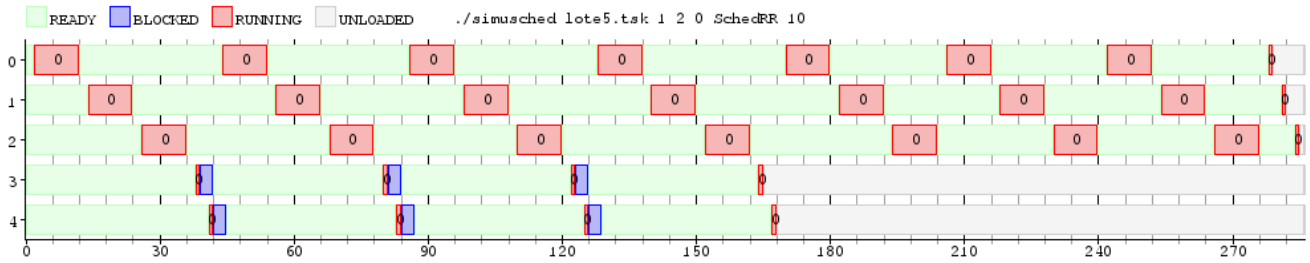
Tarea 2
Tiempo de espera: 379
Latencia: 6
Tiempo de ejecución: 449

Tarea 3
Tiempo de espera: 382
Latencia: 10
Tiempo de ejecución: 452

Tarea 4
Tiempo de espera: 55
Latencia: 14
Tiempo de ejecución: 68

Tarea 5
Tiempo de espera: 58
Latencia: 17
Tiempo de ejecución: 71

Con quantum = 10



Métricas de las tareas

Tarea 1
Tiempo de espera: 209
Latencia: 2
Tiempo de ejecución: 279

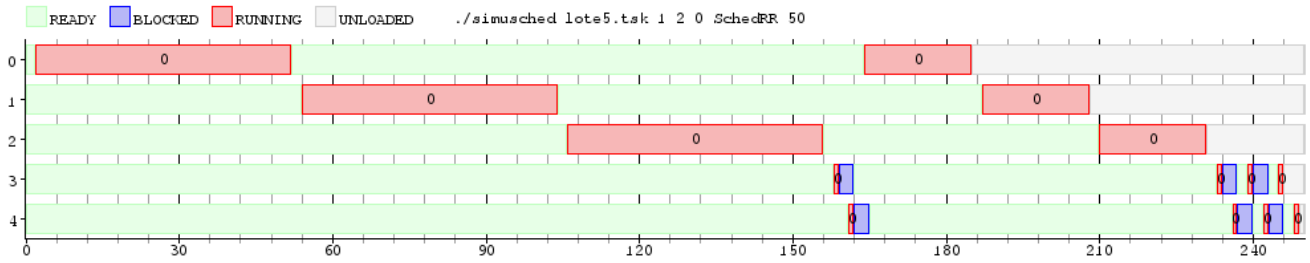
Tarea 2
Tiempo de espera: 212
Latencia: 14
Tiempo de ejecución: 282

Tarea 3
Tiempo de espera: 215
Latencia: 26
Tiempo de ejecución: 285

Tarea 4
Tiempo de espera: 162
Latencia: 38
Tiempo de ejecución: 175

Tarea 5
Tiempo de espera: 165
Latencia: 41
Tiempo de ejecución: 178

Con quantum = 50



Métricas de las tareas

Tarea 1
Tiempo de espera: 115
Latencia: 2
Tiempo de ejecución: 185

Tarea 2
Tiempo de espera: 137
Latencia: 54
Tiempo de ejecución: 207

Tarea 3
Tiempo de espera: 159
Latencia: 106
Tiempo de ejecución: 229

Tarea 4
Tiempo de espera: 233
Latencia: 158
Tiempo de ejecución: 246

Tarea 5
Tiempo de espera: 236
Latencia: 161
Tiempo de ejecución: 249

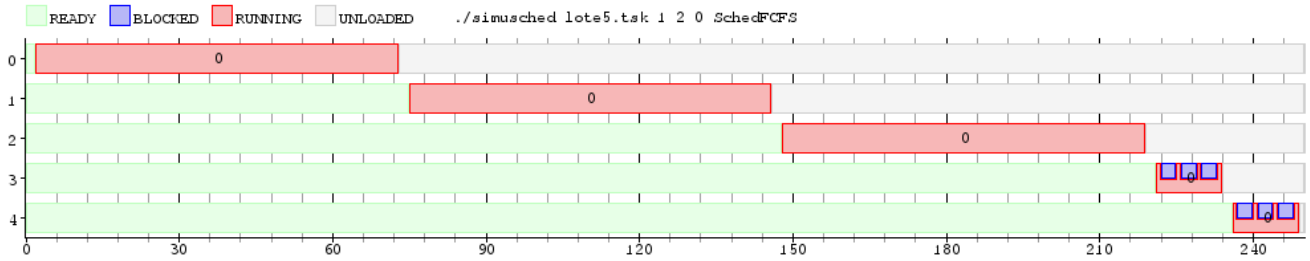
Con quantum = 2 todas las tareas pueden comenzar a ejecutarse rápidamente, por eso las latencias son bajas. En cuanto al tiempo de ejecución, son elevados para las tareas 1 a 3. La cpu debe realizar un cambio de contexto, que consume 2 ciclos, cada vez vez que una tarea pierde su turno, es decir cada 2 ciclos. Esto genera que el tiempo de espera de las tareas 1 a 3 sea elevado, y por esta razón los tiempos de ejecución de dichas tareas es también elevado. En el caso de las tareas 4 y 5, no realizan muchas llamadas bloqueantes, y el tiempo de bloqueo es corto. Esto genera que el tiempo de ejecución y el tiempo de espera de estas tareas no sea elevado.

Con quantum = 10, las tres métricas con similares para todas las tareas. Esto no es conveniente para las tareas 4 y 5, ya que son interactivas y en cada bloqueo deben pasar más de 30 ciclos para volver a ejecutarse. De hecho con este quantum, el tiempo de ejecución de estas tareas es peor que con quantum 2.

Con quantum = 50, el timepo de ejecución de las tareas 1 a 3 es menor que con los otros valores de quantum. Esto se debe a que se realizan pocos desalojos de tareas y por lo tanto se pierde menos tiempo en hacer cambios de contexto. Las tareas 4 y 5 no se benefician tanto, ya que sus latencias son elevadas y al igual que con quantum 10

deben pasar más de 30 ciclos para ejecutarse, y por lo tanto, sus tiempos de espera y ejecución también lo son.

Ejercicio 6



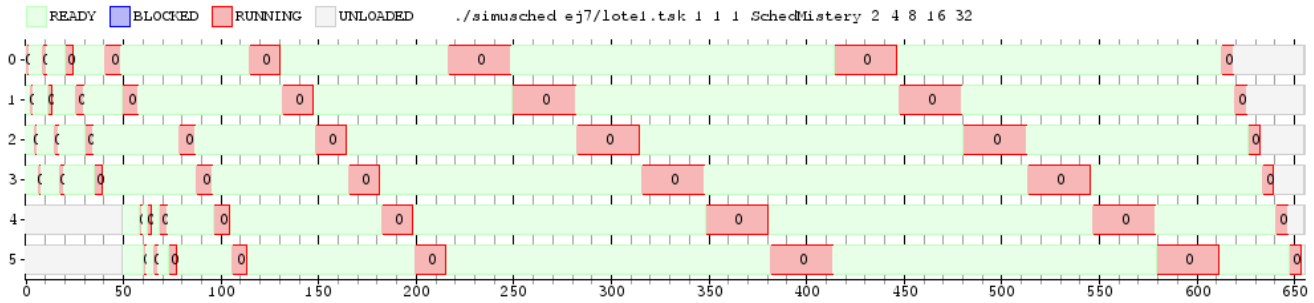
El gráfico representa la ejecución del lote del ejercicio 5 con un scheduler FCFS. En este caso las tareas interactivas se ejecutan recién cuando las demás tareas terminaron. Para que esto no sucediera, se tendrían que haber cargado antes que las demás tareas. Algo similar sucedía con Round Robin con quantum 50. Pero con quantum 2, las tareas interactivas podían terminar de ejecutarse rápidamente. Entonces Round Robin permite que todas las tareas puedan ejecutarse sin necesidad de que las demás terminen. En cambio en FCFS las tareas se ejecutan en orden, por lo que si una tarea de larga duración se ejecuta primero, las demás pueden demorar mucho en empezar a ejecutarse. Esto es un problema especialmente para las tareas interactivas.

4. Ejercicio 7

Para comprender su funcionamiento comenzamos a hacer pruebas y observamos las salidas entregadas por el scheduler SchedMistery.

Lote de Prueba 1

Como primer caso, utilizamos un lote con seis procesos sin llamadas bloqueantes y con idéntica longitud (cantidad de quantums a consumir). Cuatro de ellos comienzan en el tiempo cero, mientras que los otros dos llegan en el tiempo 50. Esta diferenciación nos servirá para ver si el scheduler les da algún trato diferente a los que llegan. Vamos a pasarle como parámetros los valores '2', '4', '8', '16' y '32' e intentaremos determinar para qué utiliza estos valores.



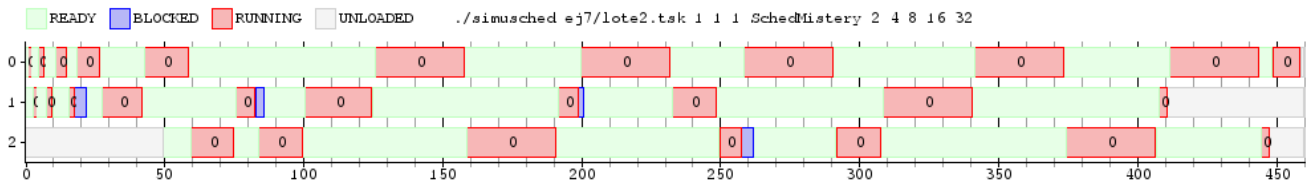
En la imagen podemos observar los resultados del primer lote. Lo primero que notamos es que el scheduler tiene un funcionamiento similar al Round Robin, dándole un quantum a cada proceso. Pero estos quantums no tienen siempre la misma longitud. Viendo en detalle los valores vemos que el primer quantum de cada proceso tiene longitud 1, mientras que después toma los valores pasados por parámetro. Con esto podemos insinuar que tiene distintas colas de prioridad: una principal con quantum 1 y luego una por cada parámetro de entrada (con quantum de su valor); y al finalizar su quantum se los mueve a una cola de prioridad menor.

Notamos también que al ingresar los dos procesos mas tarde ejecutan sus quantums hasta llegar a la cola de prioridad de los otros cuatro y se posicionan luego de estos. De este detalle podemos determinar que las colas de prioridad son FIFO.

Otro detalle es que al llegar a la última cola de prioridad se mantienen en ella. Es algo predecible, pero vale la pena tenerlo en cuenta.

Lote de Prueba 2

Como segundo caso utilizaremos un lote con tres procesos, dos con llamadas bloqueantes y un tercero sin llamadas bloqueantes. Veremos de qué forma se maneja el scheduler con respecto a los procesos con llamadas bloqueantes. Utilizaremos el proceso sin llamadas bloqueantes para comparar los quantums otorgados a cada proceso.



En la imagen se observan los resultados del segundo lote. Podemos observar que al regresar de un bloqueo los procesos tienen quantums de diversas longitudes. A simple vista no podemos afirmar nada, así que veremos en detalle los momentos previos y posteriores a las llamadas bloqueantes de cada proceso.

FILA	Proceso	Número de bloqueo	Prioridad al bloquearse	Longitud del próximo quantum
1	1	1	3	14
2	1	2	5	24
3	1	3	6	16
4	2	1	6	16

Vamos a analizar las filas de la tabla para tratar de definir el funcionamiento del scheduler.

- En la fila 1 el quantum posterior es de longitud 14: 2 + 4 + 8. Con esto podemos decir que al volver de su bloqueo se lo colocó en la cola de prioridad 2. Al no haber nadie en las cola de prioridad 2, 3 y 4 pudo ejecutar 3 veces seguidas. Luego el scheduler lo colocó al final de la cola de prioridad 5 (detrás del proceso 0). En resumen, el proceso pasó a estado bloqueado luego de estar en la cola de prioridad 3 y al volver de su bloqueo el scheduler lo colocó en la cola de prioridad 2.
- En la fila 2 el quantum posterior es de longitud 24: 8 + 16. Con esto podemos decir que el scheduler colocó al proceso en la cola de prioridad 4. Al estar vacías las colas de prioridad 4 y 5 pudo ejecutar dos veces seguidas. Al llegar a la cola de prioridad 6 se posicionó al final (luego del proceso 0). En resumen, el prceso hizo una llamada bloqueante luego de estar en al cola de prioridad 5 y al volver el scheduler lo colocó al final de la cola de prioridad 4.
- En la fila 3 el quantum posterior es de longitud 16. con esto podemos decir que el scheduler colocó al proceso en la cola de prioridad 5 (con longitud 16). El proceso ejecutó una vez y luego se colocó al final de la cola de prioridad 6 (luego del proceso 0). En resumen, el proceso realizó una llamada bloqueante luego de estar en la cola de proridad 6 y al volver el scheduler lo colocó en la cola de prioridad 5. Luego de ejecutar su quantum de longitud 16 el scheduler lo colocó en la cola de prioridad 6 (detrás de los procesos 2 y 0).
- El caso de la fila 4 es muy similar al de la fila 3. El proceso realizó una llamada bloqueante luego de estar en la cola de prioridad 6 y al volver el scheduler lo colocó al final de la cola de prioridad 5. Ejecutó un quantum de longitud 16 y luego fué colocado al final de la cola de prioridad 6 (detrás de los procesos 1 y 2).
- Vale aclarar que no encontramos importancia al número de bloqueo, ya que no notamos cambios con respecto a la primer llamada, la segunda o la i-ésima.

Viendo en detalle los casos antes listados podemos insinuar que cuando un proceso vuelve de un bloqueo, el scheduler lo coloca al final de la cola de prioridades un número menor a la prioridad que tenía al bloquearse.

Luego de correr otros lotes de prueba similares pudimos confirmar las insinuaciones antes listadas y realizamos el scheduler SchedNoMistery que responde a las características del scheduler SchedMistery.

No adjuntamos las imágenes salidas de este scheduler porque eran iguales a las del scheduler a copiar y no resultaban relevantes. Estas pruebas pueden reproducirse utilizando el Makefile con el parámetro 'ejercicio7'.

5. Ejercicio 8

El scheduler SchedRR2 es una variación del tipo de scheduler Round-Robin que no permite migración de procesos entre nucleos. Para determinar en qué nucleo correrá un proceso se tienen en cuenta la cantidad de procesos activos totales de cada nucleo (blocked + running + ready) y se elije el que tenga la cantidad menor.

La característica de no permitir migración de procesos entre nucleos afecta el rendimiento notablemente con respecto a un scheduler que si lo permita: en algunos casos lo beneficia y en otros lo perjudica. A continuación vamos a analizar un caso para el cual el scheduler sale beneficiado y otro para el que no.

Caso beneficioso

Un caso en el que el scheduler saldría beneficiado al no permitir la migración de procesos entre nucleos creemos que es un servidor que constantemente recibe peticiones (entre un grupo acotado). Al llegar una petición, se inicia un proceso que responda a ella. Estos procesos van a ir distribuyéndose entre los distintos nucleos. Al ser estos procesos de similar tamaño no va a ser necesaria la migración entre nucleos, ya que es dificil que se acumulen varios procesos en un nucleo y que el resto de los nucleos se liberen. Además, en caso de que esto ocurra lo mas probable es que a los procesos les quede poco tiempo de ejecución y hacer un cambio de nucleo (con los costos que esto genera) será mas perjudicial que beneficioso.

Creemos que para este caso el throughput va a ser un valor alto, ya que constantemente se inician procesos (para responder las peticiones) de tamaño similar, lo que implica que terminarán en un tiempo similar, dando como resultado constantes finalizaciones de procesos. El turnaround depende del promedio de la cantidad de procesos que lleguen, pero en general creemos que va a ser un tiempo bajo. El waiting time también depende de la cantidad de procesos que vayan llegando, pero al ser en su esencia un Round-Robin el proceso va a ser ejecutado cada poco tiempo. El tiempo de respuesta va a ser casi inmediato, el proceso sólo debe esperar que finalize la ronda de Round-Robin del nucleo al que fué asignado. Creemos que va a haber justicia (Fairness) en cuanto a la distribución de la CPU a los procesos por el solo hecho de que se trata de un Round-Robin, y el CPU se distribuyen de a quantums iguales.

Veamos un ejemplo de este caso. Creamos un lote de prueba con muchos procesos de similar tamaño que llegan cada tiempos reducidos y vamos a correr el scheduler para ver como se comporta.

En la figura 5 podemos ver que los nucleos se mantuvieron ocupados de forma similar (casi sin momentos inactivos) y se liberaron en tiempos cercanos.

En la figura 5 podemos ver los distintos procesos y sus estados a lo largo de su tiempo de vida. Vemos que el lote cuenta con procesos similares y terminan con un formato similar al de la entrada (en escalera).

Analicemos el rendimiento del scheduler:

- **Fairness:**
  - 8 procesos reciben 2 quantums.
  - 7 procesos reciben 3 quantums.

  Como vemos los quantums se distribuyen de forma bastante justa entre los procesos.
- **Tiempo de respuesta:**
  Promedio: $\frac{1+1+1+1+6+4+6+4+11+7+11+7+25+19+25}{15} = 8,6$
- **Throughput:**
  Promedio: $\frac{1+1+1+1+1+1+1+2+1+1+1+1+1+1}{82} = 0,1829$
  Vale aclarar que si contamos sólo a partir de la unidad de tiempo 35 el número cambia significativamente: $\frac{1+1+1+1+1+1+1+2+1+1+1+1+1+1}{47} = 0,319$. Esto se debe a que en un comienzo no se encuentra ejecutando ningún proceso previo, entonces se dan esas 35 unidades de tiempo donde ningún proceso finaliza. Sirve tener en cuenta este valor porque luego de mucho tiempo de ejecución el throughput tendería a ser mas cercano a 0,319 que a 0.1829.
- **Turnaround:**
  Promedio: $\frac{(36-0)+(31-1)+(38-2)+(55-3)+(54-4)+(73-5)+(56-6)+(59-7)+(63-8)+(77-9)+(65-10)+(63-11)+(72-12)+(81-13)+(74-14)}{15} =$
  $52,8$
  El número no parece ser desfavorable teniendo en cuenta que el lote contaba con procesos con longitud de entre 5 y 65 (y algunos con llamadas bloqueantes intercaladas).
- **Waiting time:**
  Promedio: $\frac{(1+19)+(1+12+26)+(1+19)+(1+12+17)+(6+28)+(4+21)+(6+28)+(4+12)+(11+28)+(7+23+18)+(11+28)+(7+16)+(25+19)+(19+20+}{15}$
  $33,53$

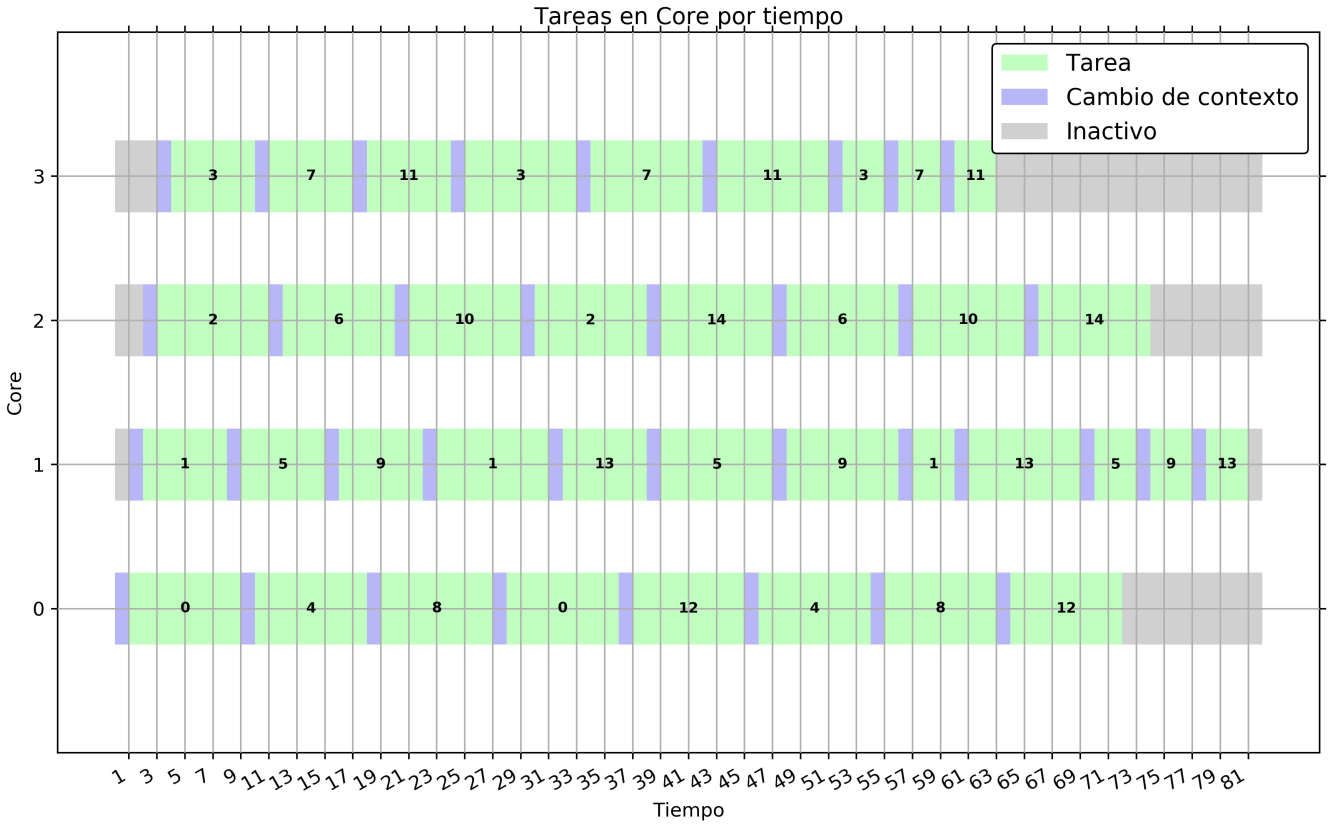


Figura 7: Diagrama de uso de nucleos del lote 1.

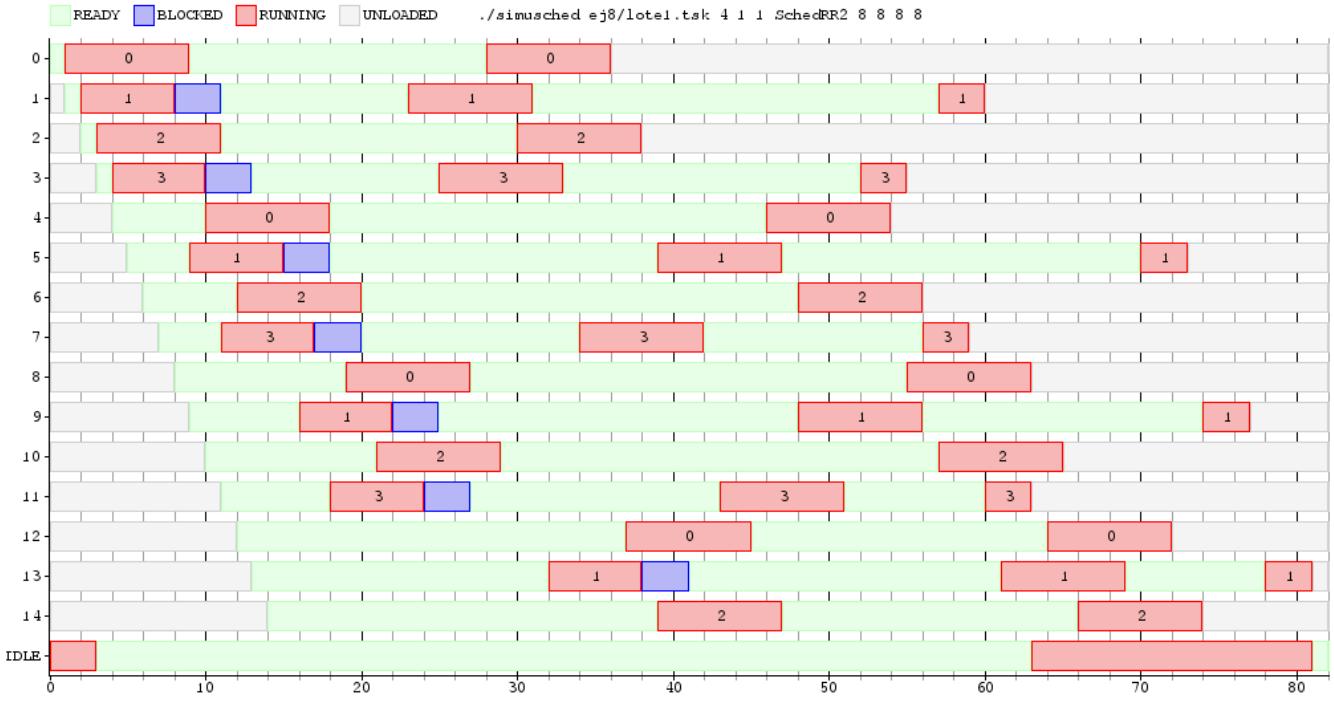


Figura 8: Diagrama de procesos del lote 1.

Caso no beneficioso

Un caso para el cual este scheduler no saldría beneficiado sería el de una computadora de escritorio de uso doméstico que corre procesos de diversos tipos y tamaños. El problema al que queremos apuntar es que pueden quedar varios procesos largos y pesados compartiendo un único nucleo, liberarse los otros y no poder distribírlos (migrarlos) entre estos nucleos libres.

Creemos que para este caso el troughput y el turnaround van a tomar valores altos. El waiting time y el turnaround también van a tomar valores altos, debido a que los procesos pueden variar en longitud. Al tratarse de una variación del Round-Robin el fairness se va seguir respetando.

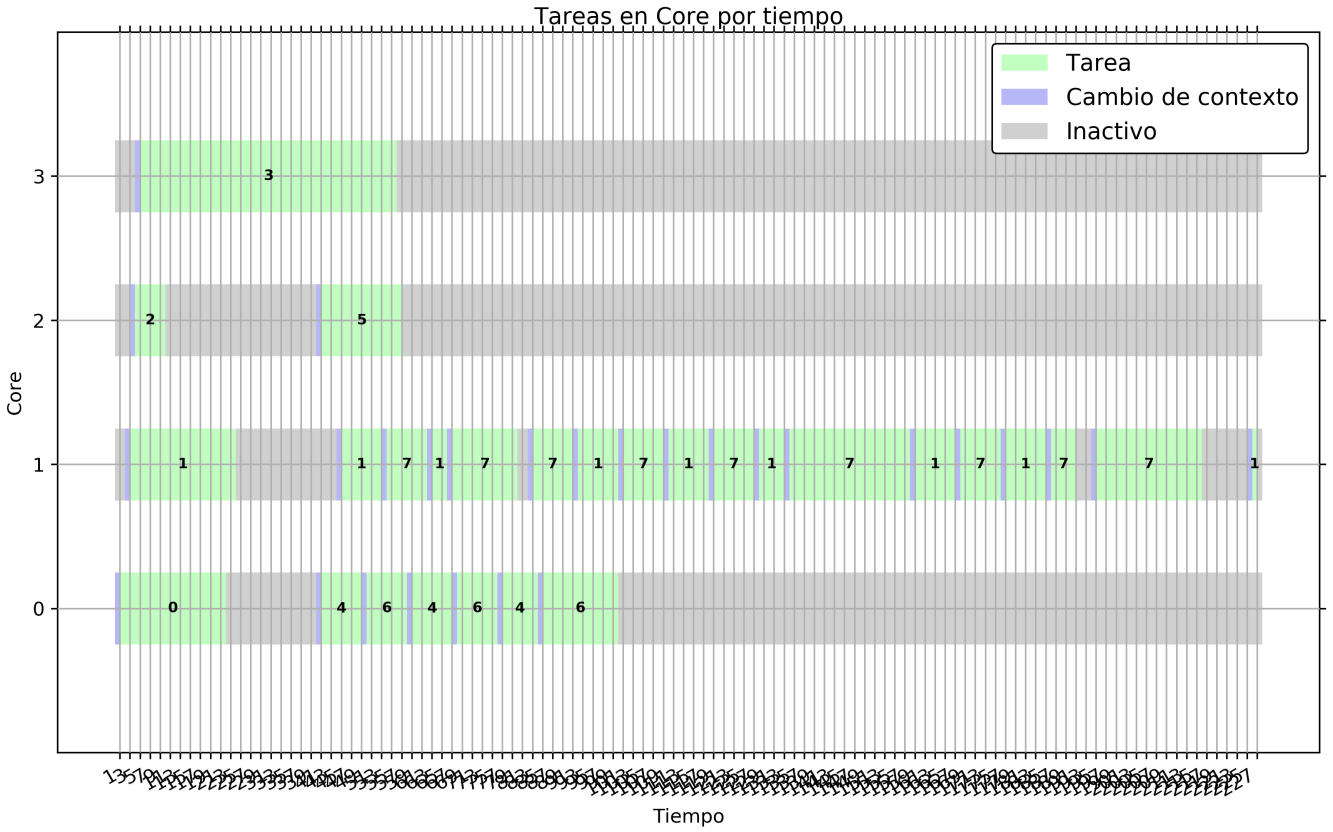


Figura 9: Diagrama de uso de nucleos del lote 2.

En la figura 5 podemos ver que los nucleos 0, 2 y 3 se liberan rápidamente mientras que el 1 continúa ejecutando por largo tiempo. Esto se debe a que quedan dos procesos pesados (el 1 y el 7) corriendo en el nucleo 1 y el scheduler no migra alguno de estos a otro nucleo.

En la figura 5 podemos ver los procesos y sus estados con el paso del tiempo. Aquí también podemos notar que los procesos 1 y 7 son los que utilizan mas tiempo el CPU (y antes vimos que justo caen en el mismo nucleo).

Analicemos el rendimiento del scheduler:

- **Fairness:**
  Se trata de un scheduler Round-Robin y podemos ver que distribuye un quantum a cada uno de la cola en órden.
- **Tiempo de respuesta:**
  Promedio: $\frac{1+1+1+1+1+1+7+9}{8} = 2,75$
- **Throughput:**
  Promedio: $\frac{1+1+1+1+1+1+1+1}{228} = 0,035$
- **Turnaround:**
  Promedio: $\frac{(22-0)+(227-2)+(10-3)+(56-4)+(84-40)+(57-40)+(100-43)+(216-45)}{8} = 74,375$
  El número no parece ser favorable porque la mayoría de los procesos tienen una longitud corta. Pero el proceso mas largo tiene longitud aproximada 90, por lo que no parece tan alejado del valor obtenido.

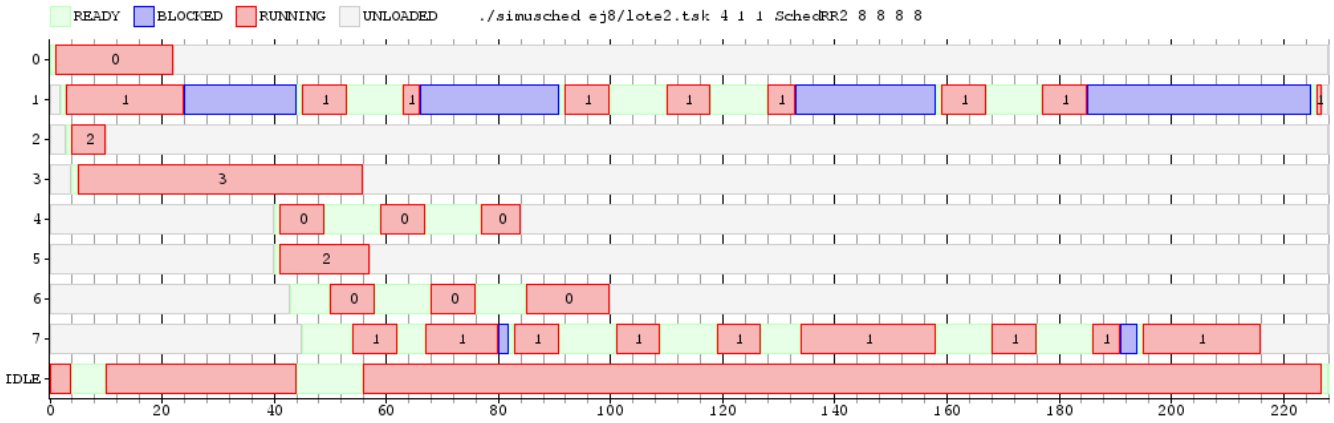


Figura 10: Diagrama de procesos del lote 2.

- **Waiting time:**
  Promedio: $\frac{1+(1+1+10+10+10+1+10+1)+1+1+(1+10+10)+1+(7+10+9)+(9+5+1+10+10+7+10+10+1)}{8} = 19,75$

La diferencia mas grande se nota en el valor de Throughput. El ejemplo del caso beneficioso al scheduler nos daba un valor de $0,1829$, mientras que el ejemplo del caso no beneficioso al scheduler nos daba un valor de $0,035$ (5 veces menor). Esto se debe a lo que comentábamos de la migración de procesos entre nucleos. En general se trata de evitar, pero para casos como el que tomamos como ejemplo no beneficioso puede ser útil y ayudar a aliviar un nucleo.

En la figura 5 se ve claramente el uso de un nucleo mientras que el resto se encuentran liberados y podrían ser utilizados para alivianar la carga sobre este.