Laboratorio 2 - Sistemas Operativos

Stephano Wurttele Johan Tanta Cesar Madera

Octubre 2020

1 Análisis

El problema a resolver es el siguiente:

- En un barbería se encuentran N barberos y la lista de espera tiene K sillas.
- Sea cualquier barbero, al observar que la lista de espera se encuentra vacía, este barbero procede a dormir.
- Cuando una persona entra a la barbería, si la lista de espera se encuentra llena, este procede a retirarse. En el caso contrario, entra a la lista de espera, ocupando uno de los K asientos.
- El final de día es considerado cuando entran a la barbería M personas, es decir, se cuentan las personas que fueron atendidas y las que se retiraron por encontrar la lista de espera llena.

Luego de evaluar el problema, en nuestro programa debemos tener tres formas de controlar o llevar cuenta ciertos datos importantes. Primero, cuantos barberos hay disponibles en la barbería. Luego, cuantas personas se encuentran en la lista de espera y, finalmente, cuantas personas del total quedan por pasar por la barbería. Asimismo, la modificación de estas variables sería la región crítica de nuestro programas, pues son los datos sensibles que se compartirían durante el proceso.

Para llevar a cabo el control de dichas situaciones y las regiones críticas, se usa semáforos y mutexes, pues estos son importantes para la solución del problema. El uso de mutex es utilizado para evitar que varias threads se encuentren en su misma región crítica, esta dada por determinar cuantos se encuentran en la lista de espera durante. Igualmente, el uso de semáforos son de utilidad ya que cuando una persona se encuentra en la lista de espera, el hilo de esta persona debe bloquearse hasta que un barbero pueda atenderla y en la otra perspectiva, cuando la lista de espera está vacía, el hilo del barbero debe bloquearse. Al lograr esto, resolvemos el problema de manera óptima y sin conflictos entre hilos. La correctitud de nuestro algoritmo se comprueba utilizando dos métodos, uno tradicional y otro con una herramienta. Primero, para fines prácticos y durante

la implementación, se utilizo la función printf, la cual muestra como el proceso empieza, ejecuta sus tareas intermedias, y termina sin problemas. Sin embargo, ante la impredecibilidad del comportamiento de los threads, utilizamos una herramienta adicional que nos ayuda a confirmar la ausencia de errores, Valgrind. Con esta herramienta comprobamos que no hay errores.

2 Implementación

Para la implementacion se utilizaron las siguientes variables:

- **sem_t barberos**: Semáforo que señala cuantos barberos están disponibles para atender.
- **sem_t num_personas**: Numero de personas que ingresarán a la barbería. Si este contador es cero, significa que es el final del día, pues todos los clientes planeados para la ejecución del programa ya pasaron por la barbería, siendo atendidos o no.
- **sem_t waiting_list**: Numero de personas que se encuentra en la lista de espera, o que ocupa una de las sillas. Este semáforo despertará a los barberos cuando por lo menos su valor sea mayor a 0, lo cual significa que existe por lo menos 1 persona en la waiting list esperando a ser atendido.
- **pthread_mutex_t waiting**: Este mutex nos ayuda a que un solo hilo entre a su región crítica y modifique los contenidos de la variable **waiting_list** sin que ningún otro valor lo lea y ocasione una posible incongruencia de resultados.

En base a estas variables, la funcion para el barbero seria:

```
1  void *barber(void* id){
2      int ID = *(int*) id;
3      while (sem_trywait(&num_personas) != -1) {
4          /* will stay in loop until num_personas is empty */
5          sem_post(&barberos); /* one barber is now ready to cut hair */
6          sem_wait(&waiting_list); /* decreases customer being attended
7          from waiting seats, else it waits for available customers */
8          printf("Cortando cabello, barber id: %d\n", ID);
9          /* cut hair (outside critical region */
10     }
11     printf("FIN DEL DIA\n");
12 }
```

Se observa que cada uno de los barberos tiene su propio identificador para fines ilustrativos. La condicional del bucle está dada para que el barbero sepa cual es el final de la jornada laboral y esto sucede cuando las M personas ingresan a la barbería, sean o no atendidas. En primer lugar, se informa a los clientes que se encuentra un barbero disponible (linea 5). Luego, este barbero, revisa si es que existe algún cliente en la lista de espera(linea 6). En caso que si exista, pasa a cortar el cabello, en caso contrario, se duerme hasta que algún cliente llegue.

Por otra parte, la funcionalidad de los clientes sería:

```
1  void *person(void* id){
2      int ID = *(int*) id;
3      pthread_mutex_lock(&waiting);
4      // Locks waiting mutex to access waiting list as reader and writer
5      int temp = 0;
6      sem_getvalue(&waiting_list, &temp);
7      //gets number of customers in waiting list at the moment
8      if (temp < SEATS) { /* if there are no free chairs, leave */
9          sem_post(&waiting_list);
10         // Increases customers in waiting seats
11         pthread_mutex_unlock(&waiting);
12         // Unlocks waiting lock for other threads to use waiting list
13         sem_wait(&barberos);
14         /* Decreases number of available barberos, or waits until
15         there is one available */
16
17     } else {
18         sem_trywait(&num_personas);
19         // Decreases total number of people
20         pthread_mutex_unlock(&waiting);
21         // Unlocks waiting lock for other threads to use waiting list
22     }
23 }
```

A la persona también se le asigna un identificador. En primer lugar, este revisa la *waiting_list* para verificar si es que existe un espacio disponible (linea 5 a 8). Por ser un recurso compartido se debe restringir para que que varios hilos se encuentren en su región crítica al mismo tiempo. Debido a ello, nos apoyamos de un *mutex_lock* en *waiting* (linea 3). En caso de que haya un espacio disponible, se procede a aumentar el valor del contador del semáforo de *waiting_list*, (linea 9). Asimismo, se indica que hay un nuevo cliente esperando a ser atendido. Luego, se libera el lock *waiting* para que los demás hilos, si lo requieren, puedan acceder al recurso de la lista. Ya esperando el cliente solo queda que alguno de los barberos se desocupe, verificación que hacemos haciendo un *sem_wait* en *barberos*, esperando a que su valor sea mayor a 0, o se queda esperando hasta que lo sea (linea 13). En caso la verificación del número de espacios libres falle, es decir que no hallan asientos libres, el cliente se retira, lo que significa que *num_personas* decrecerá, pues hay uno menos del total de personas (linea 18). En este caso alternativo, también se necesita liberar el lock de waiting por el motivo previamente establecido.

3 Resultados

Ejecutando el algoritmo y revisando los prints dentro del código, se pudo ver lo siguiente:

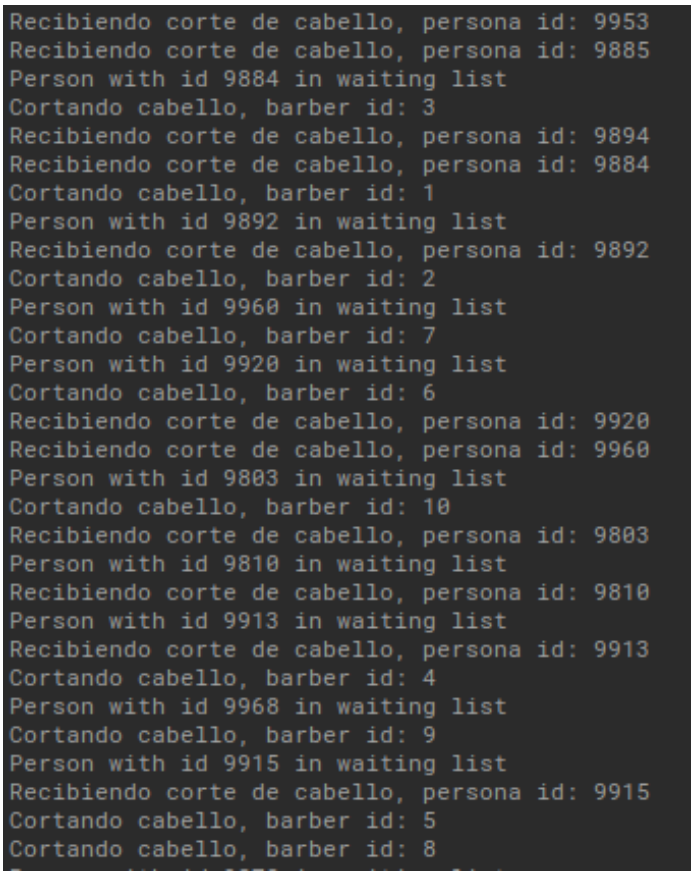

Figure 1: output de la ejecución del código, donde personas reciben su corte de cabello

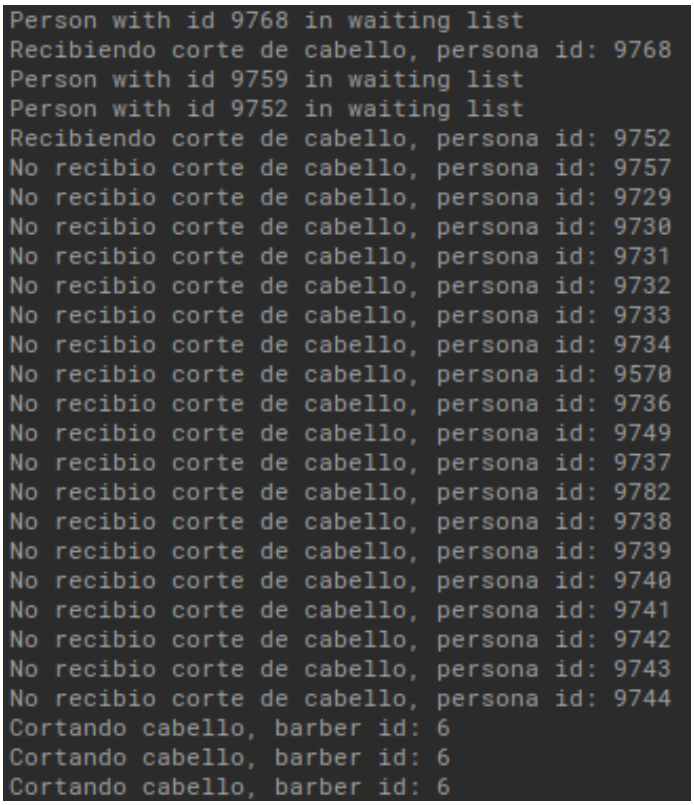

Figure 2: output de la ejecución del código, donde personas no reciben su corte de cabello

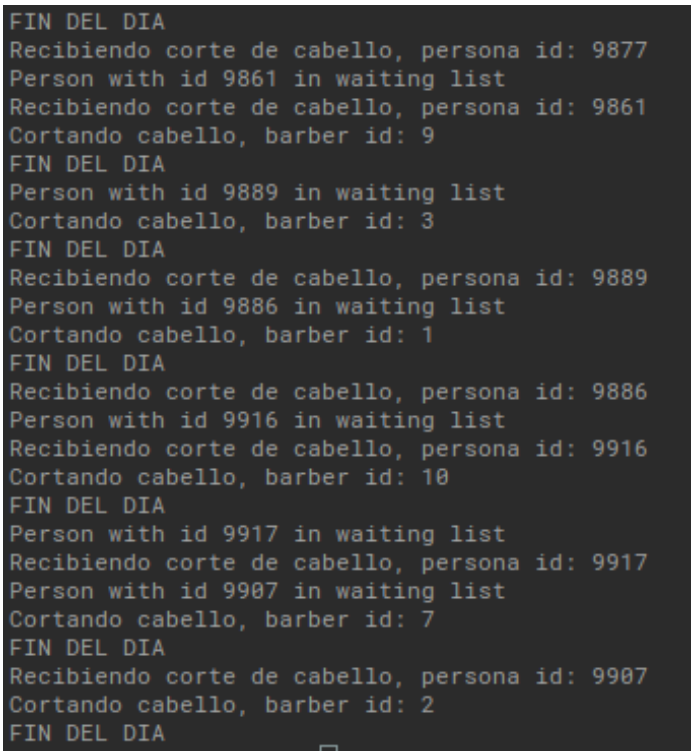

Figure 3: output de la ejecución del código, donde termina la jornada laboral de cada barbero

Se ha ejecutado el código con el siguiente escenario: 10000 personas, 10 barberos y 5 sillas. De esta manera, se puede observar primero que aparecen los clientes y luego eventualmente son atendidos por los barberos concurrentemente, hasta que se atiende a todos los clientes. Por ello, un dato importante que se puede notar es que algunos barberos terminan su ejecución ("FIN DEL DIA") antes que los otros. Esto sucede debido a que mientras uno termina de cortar el cabello y trata de revisar si es que existe un siguiente cliente, no lo encuentra y termina su ejecución. En cambio otros barberos podrían estar cortando el cabello en ese mismo instante, por lo que terminan sus ejecución. Otro dato interesante es que si se hace demorar al barbero mientras ejecuta la acción de cortar el cabello, se puede apreciar mejor cuando es que ciertos clientes se van retirando de la barbería, ya que actualmente aparece un grupo relativamente bajo de clientes que no han sido atendidos. También se corroboraron los resultados con la herramienta Valgrind, dando como resultado una ejecución limpia y concurrente:

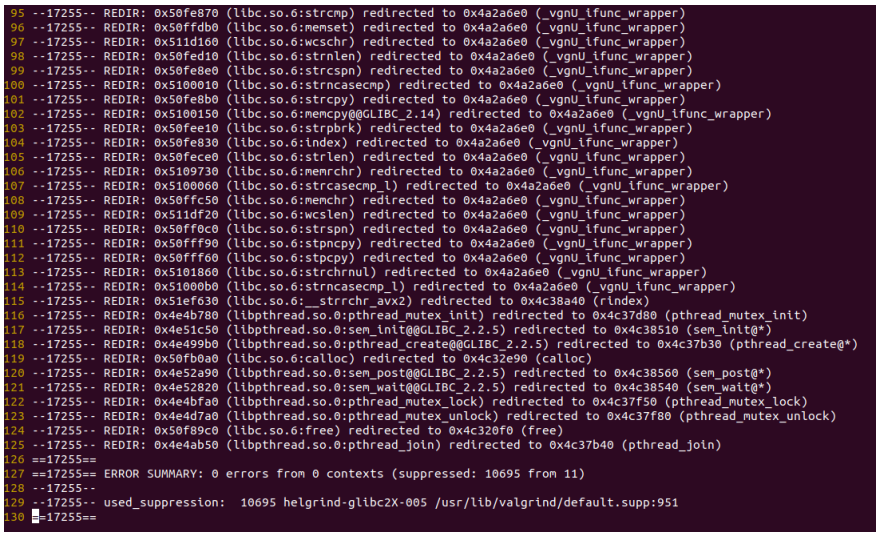

Figure 4: output de la ejecución del código con Valgrind

4 Conclusiones y observaciones

De los resultados encontrados y con la ayuda de la herramienta de Valgrind, logramos concluir que una concurrencia completa con un número elevado de hebras es posible, manteniendo exclusión mutua en los espacios importantes, llamadas secciones críticas, de forma que no obtenemos resultados inesperados o incontrolables.

Podemos observar, sin embargo, que el funcionamiento continuo es de alguna forma ficticio. Para emular un comportamiento más regular, funciones como *sleep()* pueden ser utilizadas, pues dichas funciones representan una demora en la atención de uno de los clientes y consecuentemente resultan en menos clientes siendo atendidos.