CSS 503
Programmer's Chang Liu
Email: cliu2015a@gmail.com

Design

1. Program description:

The purpose of this program is to extend the original-so-called sleeping-barber problem to a multiple sleeping barbers problem where many customers visit a barbershop and receive a haircut service from any one available among barbers in the shop.

This program has three files, driver.cpp, Shop_org.h and Shop_org.cpp. The Shop_org class simulates a barbershop which consists of a waiting room with n chairs and a barber room with m barber chairs.

If there are no customers to be served, all the barbers go to sleep. If a customer enters the barbershop and all chairs (including both, waiting and barber chairs) are occupied, then the customer leaves the shop. If all the barbers are busy but chairs are available, then the customer sits in one of the free waiting chairs. If the barbers are asleep, the customer wakes up one of the barbers.

Note that when all waiting chairs are occupied, but a barber chair is available, I make the customer go to sit on the barber's chair in this case. I double checked this question with the Doctor Parson, she thinks that should let that customer leave in that case. However, I compared the giving output with my output thoroughly, and found that when input more barbers and less waiting chair (for instance, 1 waiting chair), following the condition the teacher said will lead more customer leave the shop since it only considers the number of waiting chairs. It seems different from the sample output. Therefore, I change the condition to when all waiting chairs are occupied but a barber chair is available, I make the customer go to sit on the barber's chair. Please contact me if this is inappropriate condition. I have already notice this problem. I can submit another version of program2 with the condition which the teacher said.

2. Implementation:

2.1 Avoid of race condition

the sleeping barber problem is a classic inter-process communication and synchronization problem between multiple operating system processes. There are many data (service_chair, in_service, and money_paid) might be accessed and modified by different threads. Without process synchronization, the race condition will very likely happen to cause data inconsistency. For example, while the driver keeping generate customer threads one after another, the new customer thread will try to grab the available barber chair and set to its id and erase the original data. For another example, the barber and customer will “racing” to change the money_paid variable and cause data inconsistency.

2.2 Use monitor to create synchronization

This program uses pthread mutex to implement a monitor (which is the barber shop) to synchronize the sleeping barber program. The whole shop has only one **mutex**. Customer and barber threads will enter their critical section only if they get the mutex. Therefore, execute the program in an orderly manner.

Depends on the condition, the thread which get the mutex sometimes gives back the mutex and go to wait condition until a specific condition variable is signaled, then, this thread comes back to critical section to continue to execution.

Details look at 2.4 2.1 Program flow

2.3 Implementation details – array of conditional variable

In this program, I use three arrays of condition variables: cond_customer_served[nBarbers], cond_barber_paid[nBarbers] and cond_barber_sleeping[nBarbers]. They are each specified by index of barberId (0~ nBarber-1). Threrefore, each barber will have his own cond_barber_paid[barberId] and cond_barber_sleeping[barberId] to identify which specific barber should go to sleep or wait until the customer pay. The cond_customer_served[barberId] indicate which customer is being served by the specific barber identify by barberId. When in_serving[barberId] is true, that barber go to wait.

Besides, all the customers are sharing a same cond_customers_waiting. In this case, when barber signal next customer in waiting, this conditional variable will signal the first customer waiting with this variable. Therefore, follow the order of first come first serve.

2.4 Program flow

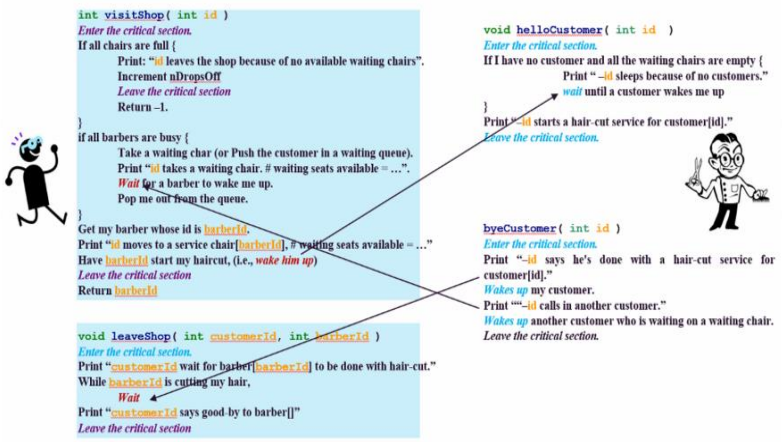
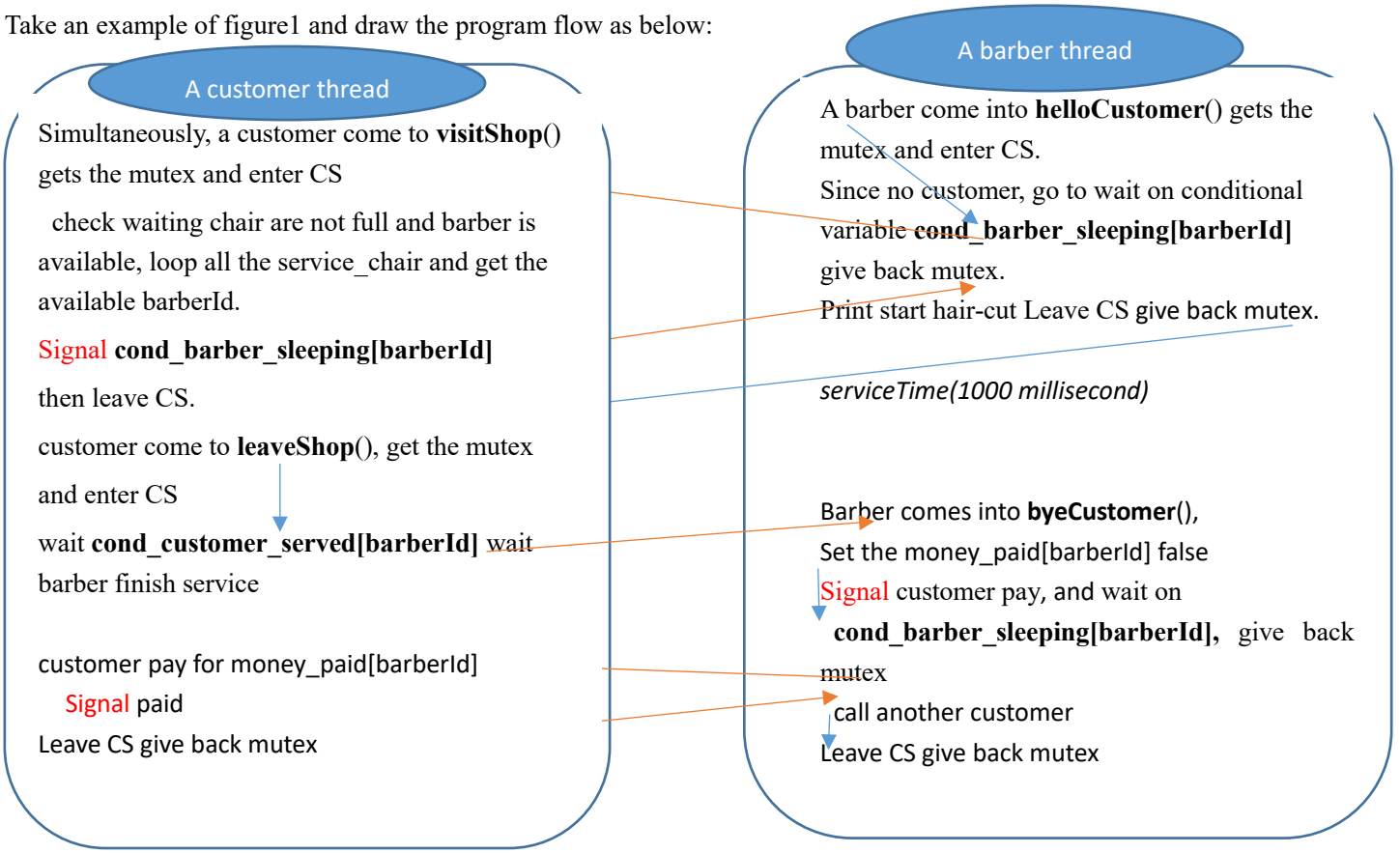


Figure 1

Take an example of figure1 and draw the program flow as below:



Discussion

1. Limitation and possible extension

1.1 performance rely on hard ware

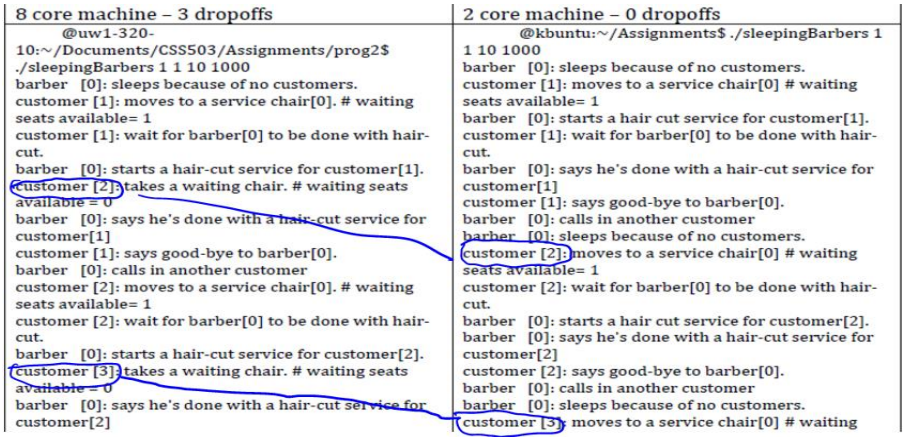
The sleeping barber problem is a synchronization problem between multiple operating system processes. The program generates multi-thread and all threads are running Simultaneously. However, since this program requires executing in an orderly manner, it only allow one thread enter its critical section(CS) at a time. Therefore, comparing with a multi-core processor, a single-core (or less core) processor might be forced to have more context switch to allow thread which wish to enter CS to run first, and thread which run remain instruction might be postponed. In this case, using a single core (or less core) processor run the program might increase the running time (decrease performance), but decrease nDropsOff customers in this program.

This is mainly because pthread mutex follows the ***critical section requirement***. The second requirement is "progress" which is "If no process is executing in its CS && there exist some processes that wish to enter their CS. The selection of the processes that will enter the CS next cannot be postponed indefinitely." This means that if there is thread which wishes to enter CS is waiting, the processor will schedule the CPU resource to let this waiting thread to run first. Therefore, with a single-core (or less core) processor, the thread which finished the CS or run No-CS instructions has to yield and give back the entire processor to allow another thread to enter its CS. A single-core (or less core) processor might be forced to have more context switch, and the No-CS instruction may only run in the period of no thread wait to enter its CS.

In this program, customer thread (which run visitShop() and leaveShop()) and barber thread (which run helloCustomer() and byeCustomer()) need to enter CS, and main (witch generate customers with interval of random millisecond) is running No-CS instructions.

Theoretically, comparing with multi-core processor, a single-core(or less core) processor gives less chance for main to generate a new customer and tends to give more chance to the customer thread and barber thread to work on their CS instructions and finish the CS.

On the country, on a multi-core processor machine, the main which is the thread run No-CS instruction may easier get CPU resource and take an available processor to execute its remain instruction. The usleep(rand() % 1000) gets more chance to run, and the interval seems shorter than the single-core(or less core) processor.



8 core machine – 3 dropoffs	2 core machine – 0 dropoffs
@uw1-320-10:~/Documents/CSS503/Assignments/prog2$ ./sleepingBarbers 1 1 10 1000 barber [0]: sleeps because of no customers. customer [1]: moves to a service chair[0]. # waiting seats available= 1 customer [1]: wait for barber[0] to be done with hair-cut. barber [0]: starts a hair-cut service for customer[1]. customer [2]: takes a waiting chair. # waiting seats available = 0 barber [0]: says he's done with a hair-cut service for customer[1] customer [1]: says good-bye to barber[0]. barber [0]: calls in another customer customer [2]: moves to a service chair[0]. # waiting seats available= 1 customer [2]: wait for barber[0] to be done with hair-cut. barber [0]: starts a hair-cut service for customer[2]. customer [3]: takes a waiting chair. # waiting seats available = 0 barber [0]: says he's done with a hair-cut service for customer[2]	@kbuntu:~/Assignments$ ./sleepingBarbers 1 1 10 1000 barber [0]: sleeps because of no customers. customer [1]: moves to a service chair[0] # waiting seats available= 1 barber [0]: starts a hair cut service for customer[1]. customer [1]: wait for barber[0] to be done with hair-cut. barber [0]: says he's done with a hair-cut service for customer[1] customer [1]: says good-bye to barber[0]. barber [0]: calls in another customer barber [0]: sleeps because of no customers. customer [2]: moves to a service chair[0] # waiting seats available= 1 customer [2]: wait for barber[0] to be done with hair-cut. barber [0]: starts a hair cut service for customer[2]. barber [0]: says he's done with a hair-cut service for customer[2] customer [2]: says good-bye to barber[0]. barber [0]: calls in another customer barber [0]: sleeps because of no customers. customer [3]: moves to a service chair[0] # waiting

Figure 2

Figure 2 compares the sample code, the more-core machine output shows that a new customer tends to come sooner than less-core machine. This might cause more customers going to waiting chair. Therefore, when input parameters are all the same, the more core machine might have more nDropsOff than the less core processor machine.

Overall, this program give different result when running with different number of processor.

1.2 for loop find the first barber implementation

I use a for loop to check the service_chair[barberId] array. Whenever find the service_chair[barberId] == 0, that is the first available barber to service this customer. This might cause the barber who wakes up this customer is different to the one who cut this customer's hair. Moreover, the barbers who have smaller barberId index have higher possibility to set to customer than barbers who have bigger barberId index. When you run my code, you might see that barber[0] takes more jobs than barber[1] and barber[2]. If we have many barber in this program, for example 6, and not many customer, for example 20, the barber[0] and barber[1] might take all the customers hair-cut, but barber[2], barber[3], barber[4] and barber[5] are keeping sleeping and never wake up.

This drawback can be improved by using a queue to save all the barbers who is available. When customer come, pop the queue in front. All the barber thread will take turns to serve customer.

2. Discussion on step5 & step 6:

2.1 step 5

```
# customers who didn't receive a service = 0
chang15@uw1-320-13:~/program2$ ./sleepingBarbers 1 100 200 1000
```

I need 100 chairs to serve all 200 customers. This result is like the sample code for eight core. I have mainly talk about the reason in 1.1 that more cores processor allow main method has more chance to run since it is no-CS instruction. Therefore, the customer arrives faster and the shop need more waiting chairs to commentate these customers in order to serve all of them.

In my case, my machine probably has four or more cores. Thus, I have a similar result like the sample output or eight cores.

2.2 step 6

Please take look the file of "typescript". I need 3 barber to server all customers if the other parameter is 0 200 1000.

The underline design can't take 0 as the input number of waiting chairs. If the parameter of nChairs is not bigger than 0, The shop will take the default value which is 3 to set to the number of waiting chairs.

Therefore, the situation changes to with 3 waiting chairs, 200 customers and 1000 millisecond service time, how many barbers needed to serve all the customer (nDropOff == 0). It turns out that 3 barbers are enough to finish the work.

Well this is the result of testing on the machine with four or more cores. I suppose that if using a less core machine, then the customer generating slower, we can use even less barbers to serve all the customers. The sample output of 2 core machine shows that with input of "2 0 200 1000", the 2 barber can take care of all the customers.

On the other hand, if testing on a machine with even more cores, I suppose the number of barbers need to be increase to full fill all the service.