Concurrency: Mutual Exclusion and Synchronization - Part 2

To avoid all kinds of problems in either software approaches or hardware approaches, people then turned to build concurrency support in operating systems and programming languages.

1 Semaphores

1.1 The origin of the idea

Semaphores are first proposed by Dijkstra in 1960s. According to him, an operating system should be designed as a collection of cooperating sequential processes with efficient and reliable mechanisms for supporting cooperation. He brought forward a principle for cooperation mechanism as follows: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal.

In the purpose of signaling, special variables called *semaphores* are used. And two primitives, *signal(s)* and *wait(s)* associated with a semaphore *s*, are available respectively for the signal sender and the signal receiver. Through the invocation of these two primitives the cooperation among processes is implemented.

In more details, to achieve the desired effect, the functions of semaphores and the associated primitives may be described as follows:

- A semaphore can be viewed as an integer variable, whose initial value indicates the number of resources of concern available.
- The *wait* operating decrements the semaphore value, indicating the desire for resource. If the value becomes negative, then the running process is blocked because no resource is available at the moment. Thus the absolute value of the negative integer also shows the number of blocked processes due to unsuccessful requests.
- The *signal* operation increments the semaphore value, indicating that a resource becomes available. If the updated value is not positive, i.e. zero or negative, there must be a process blocked before the update. Since now at least a source is available for allocation, then a process blocked by the corresponding *wait* operation should be unblocked.

Figure 1 shows the formal definition of semaphores and their primitives in terms of variables and procedures.

```
struct semaphore {
        int count;
        queueType queue;
}

void wait(semaphore s)
{
        s.count--;
        if (s.count < 0)
        {
            place this process in s.queue;
            block this process
        }
}
void signal(semaphore s)
{
        s.count++;
        if (s.count <= 0)
        {
            remove a process P from s.queue;
            place process P on ready list;
        }
}
```

Figure 1: A definition of semaphore primitives

```
struct binary_semaphore {
        enum (zero, one) value;
        queueType queue;
};

void waitB(binary_semaphore s)
{
    if (s.value ==  1)
          s.value = 0;
    else
      {
          place this process in s.queue;
          block this process;
      }
}
void signalB(semaphore s)
{
    if (s.queue.is_empty())
          s.value = 1;
    else
    {
          remove a process P from s.queue;
          place process P on ready list;
    }
}
```

Figure 2: A definition of binary semaphore primitives

Sometimes, the number of the resources of our concern is 1, thus as shown in Figure 2, we may have a more restricted version of semaphore, called **binary semaphore**. The binary semaphore may only take on the values 0 and 1.

For both semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore. Thus a question arises about how the blocked processes are removed from the queue when resources become available. The fairest policy is first-in-first-out (FIFO). The process that has been blocked the longest time is unblocked first. This strategy can surely avoid the starvation of blocked processes.

Note that *wait* and *signal* are atomic; that is they cannot be interrupted and each of them can be treated as an indivisible step. Actually, the semaphore itself is a critical resource and the primitives are all critical sections, while they are used for controlling access to other shared

resources. It seems we go back to the starting point since the implementation of the proposed solution involves a problem that we are trying to solve!

This is partly true. On the one hand, this signaling mechanism cannot solve the mutual exclusion problem itself and does need help from other facilities. On the other hand, if the mechanism is available, we can see through the following examples that it does bring convenience. Thus we may treat it as a wrapper around other awkward solutions.

1.2 Mutual exclusion

Figure 3 shows a straightforward solution to the mutual exclusion control problem using a semaphore *s*.

```
/* program mutual exclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true)
    {
        wait(s);
        /* critical section */;
        signal(s);
        /* remainder */;
    }
}
void main()
{
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 3: Mutual exclusion using semaphores

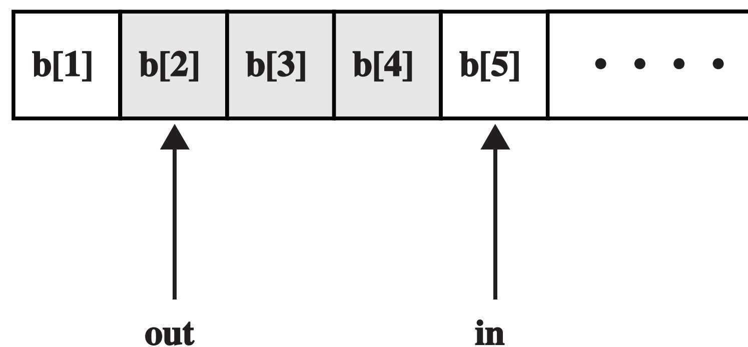
In the solution, the semaphore is initialized to 1. Thus the first process that executes a *wait* will be able to enter the critical section immediately, setting the value of *0*. Any other processes attempting to enter the critical section will find it busy and will be blocked, setting the value of *s* to -1. Any number of processes may attempt entry, each such unsuccessful attempt results in a further decrement of the value of *s*. When the process that initially entered the critical section departs, *s* is incremented, and one of the blocked processes (if any) is removed from the queue of blocked processes associated with the semaphore and put in a Ready state. When it is next scheduled by the operating system, it may enter the critical section.

1.3 The producer/consumer problem

Now we consider one of the most classic problems in concurrent processing - the producer/consumer problem. In this problem, there are two groups of entities, *producers* and *consumers*, as well as a buffer between them. Producers produce products and store them in the buffer if there is space left, while consumers fetch products from the buffer if there is any and consume them. In this problem, the buffer acts as the shared resource. It is accessible to both producers and consumers, but not at the same time.

Infinite buffer

Let us first consider the case in which the buffer is infinite. As illustrated in Figure 4, two pointers, *in* and *out*, are used respectively indicating the next space for producers to put a newly created product and the next product that is available for consumers to consume.



Note: shaded area indicates portion of buffer that is occupied

Figure 4: Infinite buffer for the producer/consumer problem

The behaviors of producers and consumers may be defined as below:

```
Producer:
  while (true) {
    /* produce item v */
    buffer[in] = v;
    in++;
  }
```

```
Consumer:
  while (true) {
    while (in <= out)
      /* do nothing */
    w = buffer[out];
    out++;
    /* consume item w */
  }
```

Thus the producers and consumers cooperate by the access to the buffer. Then how

to use the semaphore method to control the exclusive access to the buffer at the same time the cooperation relationship is maintained? That is we need to deal with both competition and cooperation. The former refers to the access to the buffer, while the latter is about the relationship between their activities. Figure 5 gives the first attempt.

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
    while (true)
    {
        produce();
        waitB(s);
        append();
        n++;
        if (n==1)
                signalB(delay);
        signalB(s);
    }
}
void consumer()
{
    waitB(delay);
    while (true)
    {
        waitB(s);
        take();
        n--;
        signalB(s);
        consume();
        if (n==0)
                waitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

Figure 5: An incorrect solution to the infinite-buffer producer/consumer problem using binary semaphores

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
    while (true)
    {
        produce();
        waitB(s);
        append();
        n++;
        if (n==1) signalB(delay);
        signalB(s);
    }
}
void consumer()
{
    int m; /* a local variable */
    waitB(delay);
    while (true)
    {
        waitB(s);
        take();
        n--;
        m = n;
        signalB(s);
        consume();
        if (m==0) waitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

Figure 6: A correct solution to the infinite-buffer producer/consumer problem using binary semaphores

Instead of two pointers, *in* and *out*, an integer variable, n ($= in - out$), is used to indicate the number of products available for consumption and a set of functions are provided for convenience: *produce()*, *consume()*, *append()*, and *take()*. The semaphore *s* is used to enforce the mutual exclusion; the semaphore *delay* is used to force the consumer to wait if the buffer is empty.

It seems the solution is pretty straightforward, however there is a flaw in it. The following lines guarantee that the consumer will be blocked if no product is available in the buffer and will not be waked up until a new item is produced.

```
if (n == 0)
  waitB(delay);
```

However, suppose we have two consumers in the system and they have used up all the products in the buffer. And suppose the two consumers, either Ready or Running, are just before the above lines.

If they proceed to do the checking on *n*, then they will surely be blocked by invoking `waitB(delay)`. But let us consider another possible case. A producer is now scheduled by the operating system, a new item is produced, and the value of *n* is incremented to be 1. If immediately after this producer exits its critical section, the operating system alternately schedules both consumer processes. Then surprisingly, both will pass the checking on *n*, and thus are not blocked. If they continue to run, then the value of *n* will possibly be decremented twice. That is the value of *n* becomes -1. However, there is only one product available in the buffer previously, but now two consumers have consumed "successfully". One of them has consumed an item from the buffer that does not exist!

Can we simply move the *if* into the critical section? No, that will cause a deadlock, in which a consumer may be blocked in the critical section waiting for a producer to produce a new item while a product is blocked waiting for the consumer to exit from the critical section so that it may proceed to produce.

A fix for the problem is to introduce an auxiliary variable that can be set in the consumer's critical section for use later on. This is shown in Figure 6.

Unfortunately, although the above correct solution is obtained finally, it is obviously too subtle and error-prone. A much better solution is using general semaphores (also called *counting semaphores*). As Figure 7 shows, *n* is now a semaphore. Its value still is equal to the number of items in the buffer, but we don't need any longer to update its value explicitly in the user programs or put processes blocked when specific values are confirmed.

```
/* program  producer/consumer */
semaphore n = 0;
semaphore s = 1;
void producer()
{
    while (true)
    {
        produce();
        wait(s);
        append();
        signal(s);
        signal(n);
    }
}
void consumer()
{
    while (true)
    {
        wait(n);
        wait(s);
        take();
        signal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Figure 7: A solution to the infinite-buffer producer/consumer problem using semaphores

```
/* program bounded buffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1;
semaphore n= 0;
semaphore e= sizeofbuffer;
void producer()
{
    while (true)
    {
        produce();
        wait(e);
        wait(s);
        append();
        signal(s);
        signal(n)
    }
}
void consumer()
{
    while (true)
    {
        wait(n);
        wait(s);
        take();
        signal(s);
        signal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Figure 8: A solution to the bounded-buffer producer/consumer problem using semaphores

Compared with the infinite buffer, a finite buffer is more realistic. In this case, the producers cannot append items to the buffer without restriction. Similar to the consumers, the producers will be blocked when they try to append items to a full buffer. The behaviors of producers and consumers in this case may be defined as follows:

```
Producer:
  while (true) {
    /* produce item v */
    while ((in + 1) % n == out)
      /* do nothing */
    buffer[in] = v;
    in = (in + 1) % n;
  }
```

```
Consumer:
  while (true) {
    while (in == out)
      /* do nothing */
    w = buffer[out];
    out = (out + 1) % n;
    /* consume item w */
  }
```

Figure 8 shows a solution using general semaphores. The semaphore e has been added to keep track of the number of empty spaces.

1.4 Implementation of semaphores

As we discussed above, the semaphore mechanism needs support from other solutions to concurrency problems. We have covered both software approaches and hardware approaches. Either may be used to meet our need. Figure 9 shows two possible implementations of semaphores based on respectively the testset instruction and interrupt disabling.

```
wait(s)
{
    while (!testset(s.flag))
        /* do nothing */;
    s.count--;
    if (s.count < 0)
    {
        place this process in s.queue;
        block this process (must also set s.flag to 0)
    }
    else
        s.flag = 0;
}

signal(s)
{
    while (!testset(s.flag))
        /* do nothing */;
    s.count++;
    if (s.count <= 0)
    {
        remove a process P from s.queue;
        place process P on ready list
    }
    s.flag = 0;
}
```

(a) Testset Instruction

```
wait(s)
{
        inhibit interrupts;
        s.count--;
        if (s.count < 0)
        {
                place this process in s.queue;
                block this process and allow interrupts
        }
        else
            allow interrupts;
}

signal(s)
{
        inhibit interrupts;
        s.count++;
        if (s.count <= 0)
        {
                remove a process P from s.queue;
                place process P on ready list
        }
        allow interrupts;
}
```

(b) Interrupts

Figure 9: Two possible implementations of semaphores

2 Monitors

As we can see from the above example, semaphores provide a primitive yet powerful and flexible tool for mutual exclusion and cooperation. However it is also difficult to finish with a correct program since the *wait* and *signal* operations may scatter throughout the program, which makes the possible cases intricate. *Monitor* is then brought forward as a programming language construct with equivalent functionality but easier to control.

2.1 Monitor with signal

A monitor, based on the idea of object orientation, is a software module consisting of:

- Local data: It may be the shared resources or the access points leading to those resources.
- Procedures: The procedures make up of the interface to the above local data, which is invisible to any external procedure.
- Initialization: The monitor may include code to initialize local data.

A monitor requires that only one process may be executing in the monitor at a time; any other process that has invoked procedures of the monitor is suspended, waiting for the monitor to become available. This requirement makes a monitor a mutual exclusion facility for accessing the shared resources.

Besides mutual exclusion, a monitor also supports cooperation (or synchronization) between concurrent processes by the use of *condition variables*, which like the local data reside within the monitor and accessible only within the monitor. Two functions operation on condition variables:

- *cwait(c)*: Suspend execution of the calling process on condition *c*. To avoid circular waiting, the monitor is then made available for use by another process.
- *csignal(c)*: Resume execution of some process suspended after a *cwait* on the same condition. Different from semaphores' *signal()*, *csignal(c)* will do nothing if no such process.

Usually queues are used to contain processes that have been suspended due to either waiting for accessing the monitor or waiting over a condition variable.

Figure 10 shows an example of the use of a monitor to solve the bounded-buffer producer/consumer problem.

```
/* program producer/consumer */
Monitor boundedbuffer {

  /* local data */
  char buffer[N];
  int nextin, nextout;
  int count;

  /* condition variables */
  cond notfull, notempty;

  /* procedures */
  void append(char x) {
    if (count == N)
      cwait(notfull);
    buffer[nextin] = x;
    count++;
    csignal(notempty);
  }

  void take(char x) {
    if (count == 0)
      cwait(notempty);
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count++;
    csignal(notempty);
  }

  /* initialization */
  static {
    nextin = 0;
    nextout = 0;
    count = 0;
  }
}

void producer() {
  char x;
  while (true) {
    produce(x);
    boundedbuffer.append(x);
  }
}

void consumer() {
  char x;
  while (true) {
    boundedbuffer.take(x);
    consume(x);
  }
}

void main() {
  parbegin(producer, consumer);
}
```

Figure 10: A solution to the bounded-buffer producer/consumer problem using a monitor

The advantage of monitors is that all procedures are confined to monitors themselves, so once monitors have been correctly designed, the related processes may be easily developed.

Nachos project #1 requires to implement condition variables based on semaphores.

2.2 Monitor with Notify and Broadcast

By definition, the above *csignal(c)* resumes the execution of one suspended process if there is one. This puts an implicit requirement on the process scheduling: the resumed process must be dispatched immediately otherwise other processes may enter

the critical section and the condition under which the process was activated could change.

Lampson and Redell, in 1974, improved the above approach by using *cnotify* instead of *csignal*. *cnotify* is interpreted in the following way: When a process executing in a monitor executes *cnotify(x)*, it causes the *x* condition queue to be notified, but the signaling process continues to execute. And the process at the head of the condition queue will be resumed at some convenient future time when the monitor is available. Accordingly, since there is no guarantee that some other process will not enter the monitor before the waiting process, the waiting process must recheck the condition. For example, the *if* statements in both *append()* and *take()* should be replaced with *while* ones.

Another improvement is the introduction of the *cbraodcast()* primitive, which causes all processes waiting on a condition to be placed in a Ready state. This is useful when there is no way to know how may other processes should be reactivated. In Java, *notifyall* is used instead.

3 Message passing

As we know, some concurrent processes share global resources without awareness of the existence of each other, while some others interact with each other directly. In the latter case, *message passing* is a common approach to enforce mutual exclusion and communication. It is superior to the above approaches in the sense that it works in distributed systems as well as in shared-memory multiprocessor and uniprocessor systems.

The function of message passing is normally provided in the form of a pair of primitives:

```
send(destination, message)
receive(source, message)
```

A process sends information in the form of a *message* to another process designated by a *destination*. A process receives information by executing *receive* primitive, indicating the *source* process and the *message*.

Many issues need to be considered regarding message passing and will be examined in the following sections.

3.1 Synchronization

The synchronization aspect of message passing is regarding the behavior of processes that execute either of the two primitives. Two choices are available: *blocking* or *nonblocking*.

Consider the *receive* primitive. When a process issues a *receive* primitive, there are two possibilities:

1. If a message has previously been sent, the message is received and execution continues.
2. If there is no waiting message, then either
   - (blocking) the process is blocked until a message arrives, or
   - (nonblocking) the process continues to execute, abandoning the attempt to receive.

Thus both the sender and the receiver can be blocking or nonblocking. Theoretically there are totally four combinations regarding the blocking issue, but the following three are common:

- **Blocking send, blocking receive**: Both the sender and the receiver are blocked until the message is delivered. This mode is also referred to as *rendezvous*.
- **Nonblocking send, blocking receive**: This is probably the most natural and useful combination. For example, a server process typically waits until a request comes in, and sends out the response later on.
- **Nonblocking send, nonblocking receive**: Neither party is required to wait.

3.2 Addressing

The parameters in the primitives, *destination* and *source*, raise the issue of addressing. There are two categories of addressing: *direct addressing* and *indirect addressing*.

With direct addressing, a specific identifier is used to refer to a process. Sometimes it is impossible for a receiver to specify the source of anticipated messages. In this case, the *source* parameter of the *receiver* possesses a value returned when the receive operation has been performed.

With indirect addressing, as Figure 11 shows, messages are not sent directly from sender to receiver but rather are sent to a shared data structure, called *mailbox*, consisting of queues that can temporarily hold messages. The advantage of this mechanism is that decoupling the sender and the receiver brings greater flexibility in the use of messages.

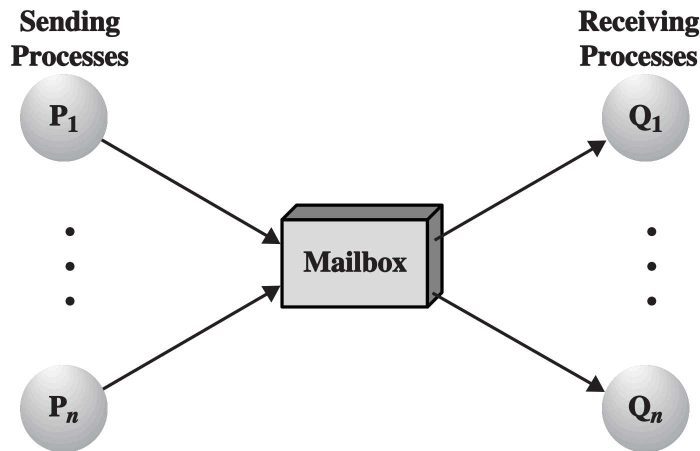


Figure 11: Indirect message passing

A mailbox may be created by a process and then owned and associated with it, or by the operating system on request of processes.

3.3 Message format

Based on the requirements of the applications, messages may take different formats. Figure 12 depicts a typical format of a message:

- **header**: contains the addresses of the source and the intended destination, message length, and some other fields for control purposes.
- **body**: contains the actual contents of the message.

Messages are typically stored in queues before they are delivered. The simplest dispatching scheme is FIFO. Sometimes different priorities may be assigned to messages according to the context.

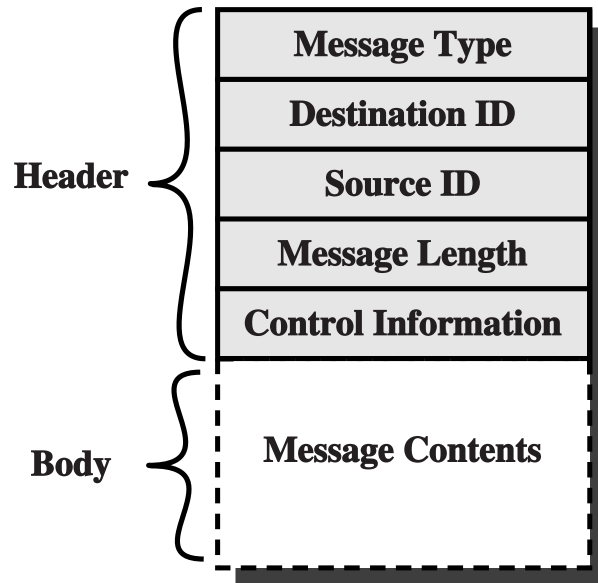


Figure 12: General message format

3.4 Mutual exclusion

Figure 13 shows how message passing is used to support mutual exclusion. In the example, a mailbox, *mutex*, is created first and initialized to contain a token message for processes to access critical resources. Each process that is going to access has to invoke *receive* first to obtain the token before moving on. If permitted, when it exits the critical section, it again sends the token back to the mailbox.

Another example shown in Figure 14 is a solution to the bounded-buffer producer/consumer problem using message passing. There are two mailboxes instead, *mayproduce* and *mayconsume*. Every single location of the buffer is associated with two token messages, one for a producer to get permission to produce and the other for a consumer to get permission to consume.

4 A Barbershop Problem

This problem is worth discussion due to its similarity to the real problems in the operating systems.

The barbershop of concern has 3 chairs, 3 barbers, and a waiting area that can accommodate 4 customers on a sofa and that has standing room for additional customers, as Figure 15 shows. And it is supposed that the maximal number of customers allowed

```
/* program mutualexclusion */
const int n = /* number of processes */;
void P(int i)
{
    message msg;
    while (true)
    {
        receive (mutex, msg);
        /* critical section */;
        send (mutex, msg);
        /* remainder */;
    }
}
void main()
{
    create_mailbox (mutex);
    send (mutex, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 13: Mutual exclusion using messages

to be in the barber shop is 20.

The customers' actions include:

- Wait: the sofa, the standing area, and the outside.
- Haircut: the chairs
- Pay: the cash register

The barbers' actions include:

- cut hair:
- accept payment:
- wait:

4.1 Analysis

Processes

```
const int
    capacity =  /* buffering capacity */ ;
    null =    /* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true)
    {
        receive (mayproduce, pmsg);
        pmsg = produce();
        send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true)
    {
        receive (mayconsume, cmsg);
        consume (cmsg);
        send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++)
        send (mayproduce, null);
    parbegin (producer, consumer);
}
```

Figure 14: A solution to the bounded-buffer producer/consumer problem using messages

What processes should be created? Only customers? Or customers and barbers? Actually barbers may be viewed as resources and only customer processes are used. In this case, customers may have a barber and a barber chair as a combination for service and release both resources after they finish.

Resources

Room capacity, sofa capacity, barber chair capacity, cashier capacity

Relationship

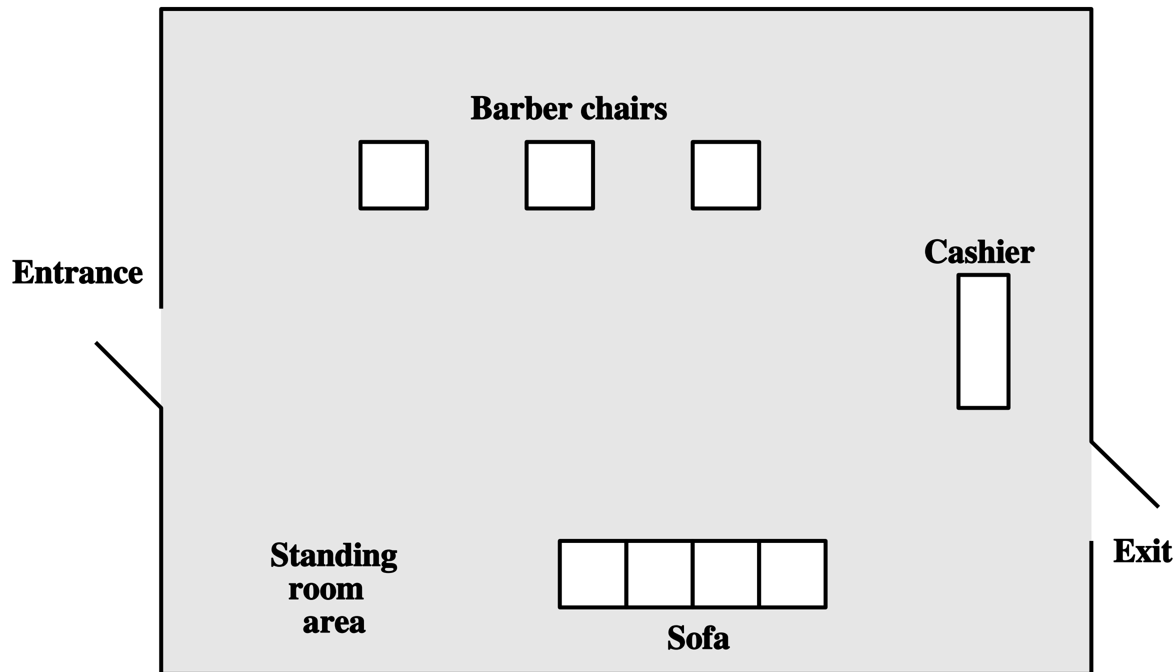


Figure 15: The barbershop

- Mutual exclusion: capacity limits
  - room capacity
  - sofa capacity
  - barber chair capacity
  - cash register capacity: The program given in the textbook does not consider the mutual exclusion of cash register capacity, which may lead to the mistake that receipts are given to the wrong customer if two customers have paid and are waiting for receipts.
- synchronization:
  - * barbers $\Longrightarrow$ customers: you may take chairs. (*barber_chair*)
  - customers $\Longrightarrow$barbers : I have take the chair and you may cut. (*cust_ready*)
  - barbers $\Longrightarrow$ customers: I have finished cutting and you may leave. (*finished*)
  - customers $\Longrightarrow$ barbers: I have left the chair and you may proceed to do whatever you want. (*leave_b_chair*)
  - customers $\Longrightarrow$ barbers: Here is the money. (*payment*)
  - barbers $\Longrightarrow$ customers: Here is the receipt. (*receipt*)

> Since the *wait* and *signal* do not guarantee the synchronized action is performed right away, thus when the action is finished, another synchronization is needed so that both parties know the action has indeed been done after it is given green light.

Note that it is awkward to put the behaviors of barbers into two processes: `barber()` and `cashier()`. Actually since a *while* loop is utilized in every process, once a process is created, it will never exit, which make it impossible for a barber to cut hair for a while and then to accept payment. A more proper approach is to merge them together into a barber process. However it seems there is lack of some query functions (like `top()` for a stack object with `pop()` and `push()`) for a barber to check if some customer is waiting at the cash register or at a barber chair.

4.2 An unfair barbershop using semaphores

Figure 16 gives an implementation of the barbershop.

4.3 A fair barbershop

There are some problems with the above implementation. For example, the customers who are sitting in the barber chairs, have all been waiting for *finish* signal. Due to the queue organization, the first customer to have the chair will definitely receive the signal first. However the barbers may operate in different speeds, so it is not always true that the first that comes will be finished first. Figure 17 made minor revisions on the previous implementation and finishes with a fair barbershop. where customers are distinguished from one another by each assigned a unique number. And accordingly, new semaphores are used, one for each number. Thus when a customer waits for the finish of haircut, he/she will wait over the corresponding semaphore. And the barber will also signal to the process waiting over the semaphore.

For communication between the barbers and customers so that the former know the number of the customer they will serve, a queue is used. Before a customer notifies the barbers that he/she is ready for haircutting, he/she first places his/her number into the queue; the barber that receives the signal will then obtain the customer number from the queue.

To make sure the customers are served strictly in the order they enqueue their numbers, another new semaphore is used as well.

```
/* program barbershop1 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3;
semaphore coord = 3;
semaphore cust_ready = 0, finished = 0, leave_b_chair = 0, payment= 0, receipt = 0;

void customer ()
{
    wait(max_capacity);
    enter_shop();
    wait(sofa);
    sit_on_sofa();
    wait(barber_chair);
    get_up_from_sofa();
    signal(sofa);
    sit_in_barber_chair;
    signal(cust_ready);
    wait(finished);
    leave_barber_chair();
    signal(leave_b_chair);
    pay();
    signal(payment);
    wait(receipt);
    exit_shop();
    signal(max_capacity)
}

void barber()
{
    while (true)
    {
        wait(cust_ready);
        wait(coord);
        cut_hair();
        signal(coord);
        signal(finished);
        wait(leave_b_chair);
        signal(barber_chair);
    }
}

void cashier()
{
    while (true)
    {   wait(payment);
        wait(coord);
        accept_pay();
        signal(coord);
        signal(receipt);
    }
}

void main()
{
    parbegin (customer, . . . 50 times, . . . customer, barber, barber, barber,          cashier);
}
```

Figure 16: An unfair barbershop

Another method to avoid the unfairness is to number the barber chairs so that less semaphores are needed, but how? Think about it!

```
/* program barbershop2 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3, coord = 3;
semaphore mutex1 = 1, mutex2 = 1;
semaphore cust_ready = 0, leave_b_chair = 0, payment = 0, receipt = 0;
semaphore finished [50] = {0};
int count;

void customer()
{
    int custnr;
    wait(max_capacity);
    enter_shop();
    wait(mutex1);
    count++;
    custnr = count;
    signal(mutex1);
    wait(sofa);
    sit_on_sofa();
    wait(barber_chair);
    get_up_from_sofa();
    signal(sofa);
    sit_in_barber_chair();
    wait(mutex2);
    enqueue1(custnr);
    signal(cust_ready);
    signal(mutex2);
    wait(finished[custnr]);
    leave_barber_chair();
    signal(leave_b_chair);
    pay();
    signal(payment);
    wait(receipt);
    exit_shop();
    signal(max_capacity)
}

void barber()
{
    int b_cust;
    while (true)
    {
        wait(cust_ready);
        wait(mutex2);
        dequeue1(b_cust);
        signal(mutex2);
        wait(coord);
        cut_hair();
        signal(coord);
        signal(finished[b_cust]);
        wait(leave_b_chair);
        signal(barber_chair);
    }
}

void cashier()
{
    while (true)
    {
        wait(payment);
        wait(coord);
        accept_pay();
        signal(coord);
        signal(receipt);
    }
}

void main()
{   count := 0;
    parbegin (customer, . . . 50 times, . . . customer, barber, barber, barber,
        cashier);
}
```

Figure 17: An fair barbershop