Week 3

The Producer-Consumer Problem. Let us consider the producer-consumer problem to see how these two primitives are used. The processes share a common buffer. One of them called a producer writes information into the buffer, and the other one, called consumer, reads and deletes it out. It is clear that, nothing can be written by the producer if the buffer is full. Similarly, nothing can be read and deleted by consumer is buffer is empty. In the first case the producer must sleep until consumer reads and information from buffer. Respectively, in the second case the consumer must sleep until the producer writes and information into the buffer. In both cases the active process awakes the process slept.

There may be a situation when a wake signal is sent to the process that is not slept yet. If so, the signal will be lost. In such case both processes will sleep forever. E.W. Dijkstra in 1965 suggested using an integer variable to count the number of wake-ups stored for future use. The idea is based on using a new variable type called semaphore. A semaphore could have the value 0, indicating that no wake-ups were saved, or some positive value if one or more wake-ups were sending.

Important notes about the producer-consumer problem:

- Two processes share a common, fixed-size buffer.
- One of them, the Producer, puts information into the buffer, and the other, the Consumer, takes information out.
- Trouble arises when the Producer wants to put a new item in the buffer, but it is already full.
- The solution is for the Producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the Consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the Producer puts something in the buffer and wakes it up.

Dijkstra proposed having two operations DOWN and UP (generalizations of SLEEP and WAKEUP). The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value and just continues. If the value of 0, the process is put to sleep. The UP operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier DOWN operation, one of them is chosen by the system, and is allowed to complete its DOWN operation.

There are many software situations which match this scenario!
Non-semaphore solution. The following solution contains a fatal race condition. The essence of the problem is that access to the count variable is not mutually exclusive.

```c
#define N 100
int count = 0;

void producer(void)
{
    int item;
    while(TRUE) {
        produce_item(&item);
        if (count == N) sleep();
        enter_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}

void consumer(void)
{
    int item;
    while(TRUE) {
        if (count == 0) sleep();
        remove_item(&item);
        count = count - 1;
        if (count == N-1) wakeup(producer);
        consume_item(item);
    }
}
```

Notes:

- The two processes producer and consumer are running continuously could even extend it to m producer and n consumer processes
- &item is the memory address of where the item of data to be placed on the buffer is stored, and where the item of data read from the buffer is to be placed.
- enter_item and remove_item are two functions that place/remove the item of data to/from the buffer
- produce_item is a function that does something to create the item of data ready to be placed in the buffer and
- consume_item is a function that uses the item of data after reading it from the buffer.

Problem:
• buffer is empty and consumer reads count = 0
• scheduler performs a context switch to give the CPU to
  the producer
• producer enters an item in the buffer and increment count
to 1
• producer then tries to wakeup consumer - except consumer
  is not asleep at this point, so the wakeup is lost
• producer keeps producing until the buffer is full, then
  is goes to sleep
• a some point whilst the producer is producing the
  scheduler will switch back to the consumer
• consumer then immediately executes the next instruction
  from when it was interrupted, i.e. goes to sleep now both
  processes are asleep - forever?

Semaphore-based Solution

typedef int semaphore;

// shared variables
semaphore mutex = 1; // controls access to critical region
semaphore empty = N; // counts empty buffer slots
semaphore full = 0; // counts full buffer slots

void producer(void)
{
  int item;
  while(1) {
    produce_item(&item);
    down(&empty); // decrement empty count
    down(&mutex); // enter critical region
    enter_item(&item); // put new item in buffer
    up(&mutex); // leave critical region
    up(&full); // increment count of full slots
  }
}

/*--------------------------------------------*/
void consumer(void)
{
  int item;
  while(1) {
    down(&full); // decrement full count
    down(&mutex); // enter critical region
    remove_item(&item); // take item from buffer
    up(&mutex); // leave critical region
    up(&empty); // increment count of empty slots
    consume_item(item);
  }
}
Notes:

- Semaphores put processes to sleep on the semaphore's queue when the value of the semaphore is decreased to < 0.
- Processes are woken up from the queue when the value of the semaphore is incremented. This algorithm uses that property of semaphores to correctly put processes to sleep and wake them up.
- empty is a counting semaphore to count the number of empty slots in the buffer initialised to the value N empty slots as all slots are empty full is also a counting semaphore to keep track of the number of full slots in the buffer initialised to empty = 0.
- mutex is a binary semaphore used to ensure only one process is ever accessing its critical region - the buffer.

How does this fix the problem outlined in the first solution? Work through it to find out.

The Dining Philosophers Problem. In 1965, Dijkstra posed and solved a synchronization problem called the dining philosophers problem. The problem can be stated as follows. Five philosophers are sitting around a table. Each philosopher has a plate of spagetti. A philosopher needs two forks to eat it. There is a fork between each couple of plates. The life of a philosopher consists of alternate periods of eating and thinking. When a philosopher gets hungry, he/she tries to acquire his/her left and right forks, one at a time, in either time. If successful in acquiring two forks, a philosopher eats for a while, then puts down the forks and continues to think. The key question is: can you write a program for each philosopher that does what it is supposed to do and never gets stuck? Unfortunately, there may be situation when all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock. Deadlock is a situation when all processes would stay blocked forever and no more work would be done.

One of the suggestions is that after taking the left fork, the program checks top see if the right fork is available. If it is not, the philosopher put down the left one, waits for some time, and then repeats the whole process. But it is not a good solution, since all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, etc., forever. A situation like this is called starvation. But if the philosophers would just wait a random time instead of the same time after falling to acquire the right-hand fork, the chance
that everything would continue in lockstep for even an hour is very small.

One way to avoid deadlock and starvation is by use a semaphore type variable MUTEX. Before to acquire forks, a philosopher would do a DOWN an MUTEX. After replacing the forks, she would do an UP on MUTEX. From a practical one, it has a performance bug: only one philosopher can be eating at any instant. With five forks available, we should be able to allow two philosophers to eat at the same time.

Let us consider the method, which allows the maximum parallelism for arbitrary number of philosophers. It uses an array to keep track of whether a philosopher is eating, thinking or hungry. A philosopher may move into eating state if neither neighbor is eating. Note that each philosopher has two neighbors: left and right. The program uses an array of philosophers, one per philosopher, so hungry philosophers can be blocked if the needed forks are busy.

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited resources.

```c
#include <stdio.h>

#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2

typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosophers (int i)
{
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}

void take_forks(int i)
```
{  
down(&mutex);
state[i]=HUNGRY;
test(i);
up(&mutex);
down(&s[i]);
}

void put_forks(i)
{
  down(&mutex);
  state[i]=THINKING;
test(LEFT);
test(RIGHT);
  up(&mutex);
}

void test(i)
{
  if (state[i]==HUNGRY && state[LEFT]!=EATING &&
      state[RIGHT]!=EATING)
  {
    state[i]=EATING;
    up(&s[i]);
  }
}