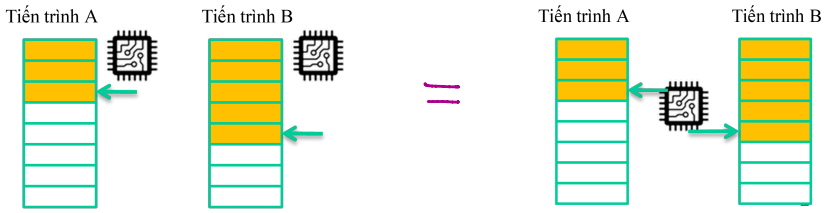
1. Processes, threads synchronization

Saturday, March 14, 2020 8:59 PM

1.1 Some basic concepts

Multiple processing model

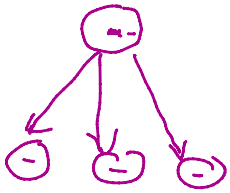
Conceptual model **Emulation model**



Parent and child processes

Unix

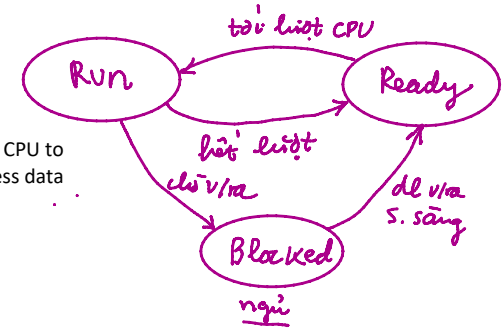
- A process can generate child processes
- In turn, child processes may have their own children
- All these processes can run in parallel



DOS

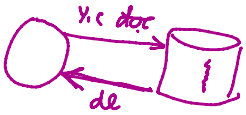
- Only one process can actually run at a time
- A process has to stall in order to load and run another process

Process states



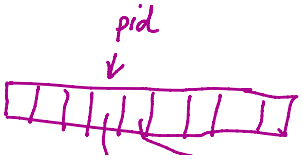
using CPU to process data

Input data is not ready



input data is ready; wait until next CPU turn

Process management

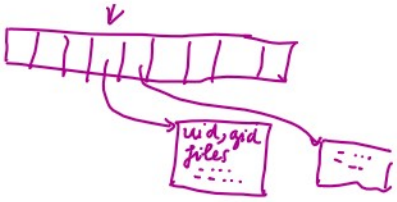


Process state table

- Process' specific data

Process state table

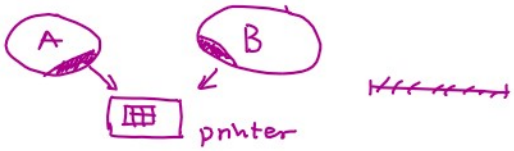
- Process' specific data
- Current memory usage
- Current files in use



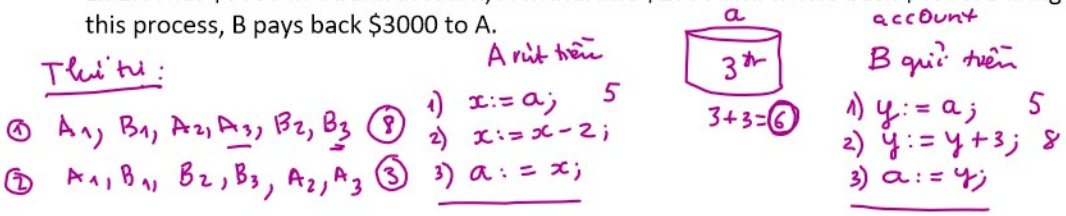
Priority

- Processes with equal priority will be placed in the same queue
- Queues are managed with different priorities

Race condition



Race condition is the situation when 2 or more processes access the same exclusive shared resource concurrently and give unexpected result.
Ex 1: A and B both put data into the printer buffer, which results in data loss
Ex 2: A has $5000 in a bank account, A withdraws $1000 and writes back $4000. During this process, B pays back $3000 to A.



Critical section / critical region is the part of a program if running in parallel with other codes, which access the same shared resource, may cause a race condition. To avoid the race condition, the critical section of a program needs to be protected, not to be allowed to run in parallel with other critical codes.

1.2 Alternation method

Description

Two processes will alternately use the shared data resource in turn. After each usage, turn will be given to the other process.

Code

Shared data
Int turn; // ==0 A's turn, ==1 B's turn

Process A

```
while (TRUE) {
    while (turn != 0);
    critical_section_A();
    turn = 1; // give turn to B
    non_critical_section_A();
}
```

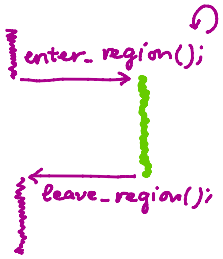
Process B

```
while (TRUE) {
    while (turn !=1);
    critical_section_B();
    turn = 0; // give turn to A
    non_critical_section_B();
}
```

Remarks

- The number of processes is limited to 2.
- Does not separate process code from race condition handling code. Turn is hard coded into the program.
- Turn has to be alternate and has to happen even when the other process is not in need.
- Waste of CPU. Processes without resource still demand for CPU, just to check whether its turn has come.

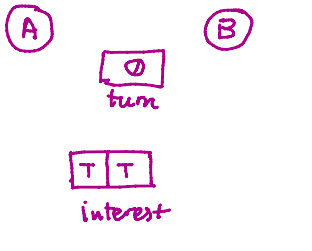
1.3 Peterson method



Description

- Processes need to express their interest if they want to get into the critical region (to use the shared resource)
- Then, they have to call **enter_region()** to get permission to the critical region
- If passed, the process will run its critical section code
- After being done with the shared resource, the process needs to call **leave_region()** to leave the critical region and gives turn to another process

Code



Shared data
```
#define N 2; // number of processes
int turn;  // ==0 A's turn, ==1 B's turn
int interest[N];  // ==T Yes, ==F No
```

Hàm enter_region()
```
void enter_region(int pid) {
    int other = 1 - pid;

    interest[pid] = TRUE;
    turn = pid;

    while (turn==pid) && (interest[other]==TRUE);
}
```

Hàm leave_region()
```
void leave_region(int pid) {
    interest[pid] = FALSE;
}
```

Remarks

- Race condition handling code is not hard coded into process code
  For example, codes for process A and B now look like below
  ```
  process(int pid) {
      enter_region(pid);
      criticial_section();  // đoạn mã cạnh tranh
      leave_region(pid);
      non_critical_section();  // đoạn không cạnh tranh
  }
  ```
- Turn will be given only to processes that are in need of the shared resource
- The number of processes is still limited to 2
- Waste of CPU. Processes without resources still demands for CPU, just to check whether their turn have come

1.4 Test and Set Lock (TSL) using hardware

An example of using a conditional flag


```
int flag;  // ==0 free, ==1 occupied

L: if (flag==0) {  // free
       flag := 1;  // mark as occupied
       critical_section();
       flag := 0;  // mark as free
}
else goto L;
```

Test and Set Lock solution

Syntax: TSL register, flag;

equivalent to the following instructions, if executed consecutively

```
register := flag;
flag := 1;
```

Code

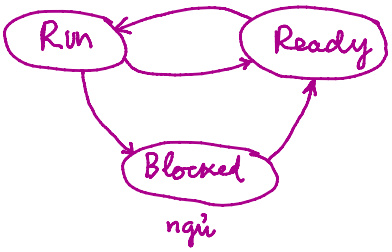
```
enter_region:
      tsl register, flag;
      cmp register, 0;  // compare register với 0
      jnz enter_region;  // Jump if Not Zero
      ret

leave_region:
      mov flag, 0;    // flag:=0 mark the resource free
      ret
```

Remarks

- Separate race condition handling code from process code
- Turn will only be given to processes with interest of using the shared resource
- No more limitation to the number of processes
- Still, waste of CPU. Processes without resources still demand for CPU, just to check whether their turn have come

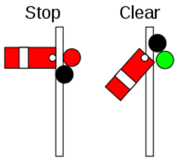
1.5 Overall remarks on the 3 above approaches



None of the above solutions turn a process into blocked state when I/O data is not available

1.4 Semaphore

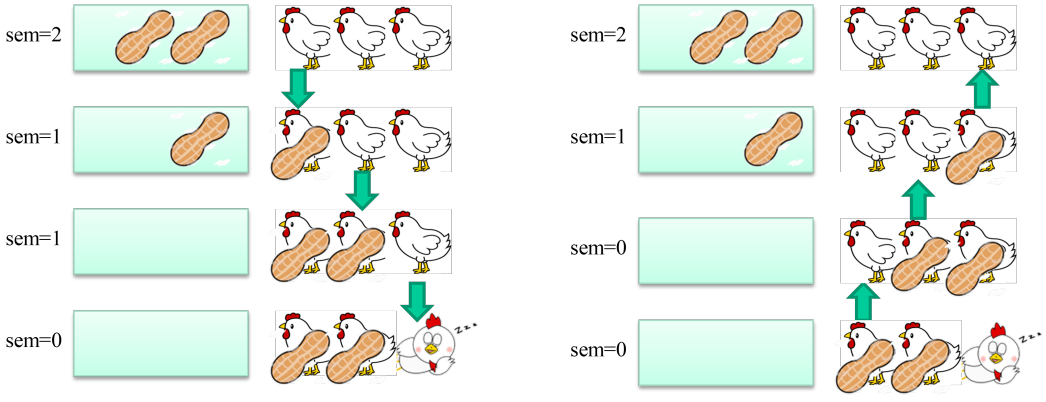
Concept



- Proposed by Edsger Dijkstra in 1962
- Is a non-negative integer variable
- Its value can only be changed via 2 actions up() and down()

- **Down(sem)**
  - If sem=0 the process turns into blocked state, down action is deferred
  - Otherwise, sem is reduced by 1
- **Up(sem)**
  - If sem=0, wake up a process in blocked state (sleeping on the semaphore wating for the resource)
  - Increase sem by 1

Example



1.5 Procucer - Consumer problem

Consumer:
- Go to the warehouse, check for an occupied slot
- If avaialbe, get an item
- Consume the item

Producer
- Produce an item
- Go to the warehouse, check for an empty slot
- If available, put the item in

An appoach

```
#define N 10 // Number of slots in the warehouse
int count;     // count the number of occupied slots

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

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

A problem with this approach is that race conditions may happen.

Using semaphore

Shared data

```
#define N 10  // N is the number of slots
#define int semaphore
semaphore mutex=1;    // Mutual Exclusion, a binary semaphore
semaphore empty = N;  // number of free slots
semaphore full;       // number of occupied slots
```

Consumer code

```
void consumer(void) {
    int item;
    while (TRUE) {
        down(&full);   // nếu kho không có hàng thì ngủ
        down(&mutex);
        get_item(&item);
        up(&mutex);
        up(&empty);  // tăng ngăn rỗng, đánh thức nsx
        consume_item(item);
    }
}
```

Producer code

```
void producer(void) {
    int item;

    while (TRUE) {
        produce_item(&item);
        down(&empty); // k.tra ngăn rỗng, nếu không có thì ngủ
        down(&mutex);
        enter_item(item);
        up(&mutex);
        up(&full);  // tăng ngăn có hàng, đánh thức ntd
    }
}
```

Notes: Do not perform down(&sem) within a critical section wrapped by down(&mutex) and up(&mutex) if all those semaphores relate to the same resource

1.6 Dinning philosophers problem

5 chinese philosophers sitting around a table, 5 plates of noodles in front, 5 chopsticks between the plates. A philosopher's activities:

- Thinking
- Feeling hungry, get 2 chopsticks next to him
- Eat
- Put the chopsticks down

Requirement: write code for philosophers to simulate the above activities.

An approach

Deadlock (bế tắc)

Mỗi NHT lấy được đũa trái và đều chờ đũa phải.

take_chopsticks(i);

```
#define N  5   // 5 philosophers
void philosopher(int i) {
      while (TRUE) {
            think();
            take_chopstick(i);          T
            take_chopstick( (i+1)%N);   P
            eat();
            put_chopstick(i);
            put_chopstick( (i+1)%N);
      }
}
```

Using semaphore

```
// Shared data
#define N 5  // 5 philosophers
#define LEFT(i)   (i-1+N)%N
#define RIGHT(i)  (i+1)%N
#define THINKING  0
#define HUNGRY    1
#define EATING    2
#typedef int  semaphore

int state[N];         // state of each philosopher
```

```
semaphore s[N];  // s[i] == 1 both chopsticks are ready, ==0 not ready for philosopher i
semaphore mutex=1;   // Mutual Exclusion semaphore
```

Philosopher code

```
void philosopher(int i) {
    while (TRUE) {
        think();
        take_chopsticks(i);   // get 2 chopsticks or wait
        eat();
        put_chopsticks(i);    // put 2 chopsticks
    }
}
```

take_chopsticks function

mutex : bán

down(&mutex) ≡ freeze

up(&mutex) ≡ free

```
void take_chopsticks(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);   // check if 2 chopsticks are available. If yes, set semaphore s[i] up
    up(&mutex);
    down(&s[i]);  // get the 2 chopsticks if s[i] is up, sleep otherwise
}
```

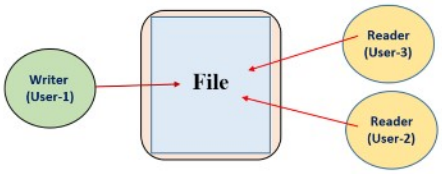
test() function

```
void test(int i) {
    if ( state[i] == HUNGRY &&
        state[LEFT(i)] != EATING &&  // left chopstick avail.
        state[RIGHT(i)] != EATING)  {  // right chopstick avail.
          state[i] =  EATING;
          up(&s[i]);   // set s[i] up if both chopsticks are avail.
    }
}
```

drop_chopsticks() function

```
void put_chopsticks(int) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT(i));  // wake up the phil. on the left if sleeping
    test(RIGHT(i)); // wake up the phil on the right if sleeping
    up(&mutex);
}
```

1.7 Reader - Writer problem



	R	W
R	✓	✗
W	✗	✗

Reader:

- 1st process: get WRITE permission
- Last process: release WRITE permission
- otherwise: just read the resource

Writer:
- Get WRITE permission
- Write to the resource
- Release WRITE permission

Shared data

```
semaphore mutex=1;  // for multual exclusion
semaphore db = 1;   // guard the write permission
int rc=0;           // reader counter
```

Reader process

```
void reader(void) {
    while (TRUE) {
        down(&mutex);
        rc++;     // tăng số đếm t.tr đọc
        if (rc==1) down(&db);   // giành quyền ghi
        up(&mutex);
        read_data();
        down(&mutex);
        rc--;     // giảm số đếm t.tr đọc
        if (rc==0) up(&db);  // trả lại quyền ghi
        up(&mutex);
        use_data();
    }
}
```

Writer process

```
void writer(void) {
    while (TRUE) {
        thinkup_data(); // chuẩn bị dl
        down(&db);      // giành quyền ghi
        write_data();
        up(&db);        // trả lại quyền ghi
    }
}
```

Question: we execute down(&db) within the scope of down(&mutex) .... up(&mutex). Is there a risk that all processes will be blocked waiting for each other?
Answer: no, as semaphore mutex guards for the resource rc, semaphore db guards for the file resource. These 2 semaphore are independent.

1.8 Sleeping barber problem

A barbershop with n chairs
The barber:
- Call a waiting client for the haircut, sleep if none
- If there is a sleeping client, wake him/her up
- Cut client's hair
- Back to the 1st step

Customers
- Check for any availalble chair, leave otherwise
- Seat and wait on a chair, call for the barber and sleep if not available
- Have the hair cut
- Go home

Using semaphore

Shared data

```
#define CHAIRS 5  // 5 chairs

semaphore customers = 0;   // no of customers waiting on the chairs
semaphore barbers = 0;     // no of available barbers
semaphore mutex = 1;       // mutual exclusion semaphore
int waiting = 0;           // same value as customers
```

Barber process

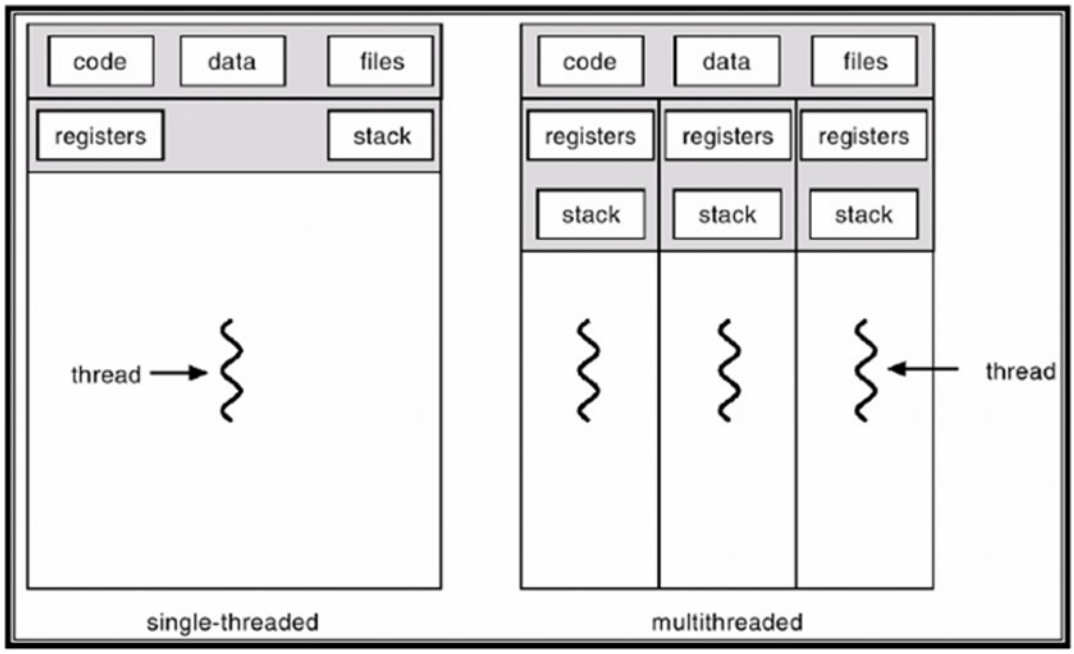
```
void barber(void) {
    while (TRUE) {
        down(&customers);   // sleep if no customers
        down(&mutex);       // get access to 'wating'
        waiting--;
        up(&barbers);
        up(&mutex)
        cut_hair();
    }
}
```

Customer process

```
void customer(void) {
    down(&mutex);
    if (waiting < CHAIRS) {  // chairs available
        waiting++;
        up(&customers);  // wake up a sleeping barber
        up(&mutex);
        down(&barbers);  // sleep if no available barbers
        get_hair_cut();
    }
    else up(&mutex);       // leave the barbershop
}
```

Threads synchronization using monitors and condition variables

Threads



A process consists of

- program code;
- process data including global variables;
- open files/resources;
- program stack containing arguments, local variables when a sub function is called;
- a set of registers that CPU is working on, including the instruction pointer that points to the next instruction.

A thread is a sub environment in a process consisting of:

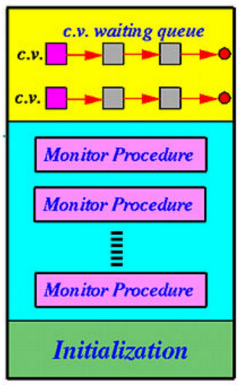
- Shared environment between threads: code, data segments and open resources
- Private environment:
  - A set of register values which is specific to each thread. All threads have the same code segment but each has its own intruction pointer
  - A stack keeping the value of sub function arguments and local variables

Monitors

A monitor is an class object with following structure. It is where concurrent programming meets object programming.

```
monitor <monitor name> {
        shared data;
        condition variables;

        procedure init(…) {…};
        procedure p_1(…) {…};
        …
        procedure p_n(…) {…};
}
```



Monitor procedures p_1 … p_n contain critical sections to access common resource labeled as shared data. Condition variables are used to control when to execute a critical section by using the sleep and wake up mechanism. In a monitor, only one procedure is allowed to run at a time.

Monitors are implemented at the programming language level, not at the kernel level as with semaphores. Monitors are used to synchronize threads within a process.

Example: account deposit and withdrawal (without condition variables) written in java

```
class account {
        int balance;
        public synchronized void deposit(int amount) {
                int x = balance;
                x += amount;
                balance = x;
        }
        public synchronized void withdraw(int amount) {
```

```
            int y = balance;
            y -= amount;
            balance = y;
        }
    };
```

To guarantee that only one procedure is running at a moment, an automatic mutex locking/unlocking mechanism is implemented at the programming language level. The actual code should look like the one below

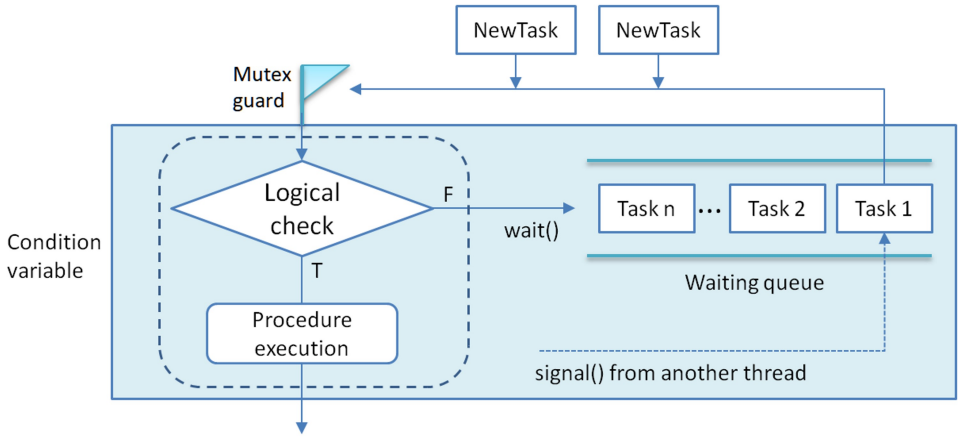
```
public synchronized void deposit(int amount) {
    int x;
    lock(this.mutex);
    x = balance;
    x += amount;
    balance = x;
    unlock(this.mutex);
}
```

```
public synchronized void withdraw(int amount) {
    int y;
    lock(this.mutex);
    y = balance;
    y -= amount;
    balance = y;
    unlock(this.mutex);
}
```

Condition variables

A condition variable is a queue structure attached to a logical condition. The queue contains threads waiting till their turn. When the logical condition is satisfied, the thread at the beginning of the queue will be woken up to run. The following functions can be used with condition variables

- wait() block the calling thread and add to the waiting queue
- signal() wake up a waiting thread (if any), otherwise the signal will be ignored



Sender priority / Mesa semantics

There are 2 different semantics when signal() is called

- Hoare semantics (receiver priority): Block the calling thread and immediately give CPU control to the woken thread. It is difficult to implement in practice.
- Mesa semantics (sender priority): Wake up a sleeping thread from the queue and move it to ready state waiting till CPU turn. The signaling thread is allowed to complete its task and exits the monitor. It is easier to implement but may cause a race condition problem as another thread may access the monitor before the woken thread. A solution is to check the logical condition again after being woken up.

Example: Producers - Consumers problem

Hoare sematics / receiver priority	Mesa semantics / sender priority

monitor ProducerConsumer {
  int fullSlots=0;
  cond empty, full;
 	
  producer() {
  if (fullSlots==N)
  wait(empty);
  … // fill a slot
  fullSlots++;
  signal(full);
  }
  consumer() {
  if (fullSlots==0)
  wait(full);
  … // empty a slot
  fullSlots---;
  signal(empty);
  }
}
	monitor ProducerConsumer {
	  int fullSlots=0;
	  cond empty, full;
	
	  producer() {
	  while (fullSlots==N)
	  wait(empty);
	  … // fill a slot
	  fullSlots++;
	  signal(full);
	  }
	  consumer() {
	  while (fullSlots==0)
	  wait(full);
	  … // empty a slot
	  fullSlots---;
	  signal(empty);
	  }
	}

Implementation in C/C++

C/C++ does not provide an automatic locking/unlocking mechanism so programmers have to do it manually.

Functions for monitor programming in C/C++

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init (pthread_mutex_t *mutex, const
                        pthread_mutex_attr *attr);
int pthread_mutex_lock (pthread_mutex_t *m);
int pthread_mutex_unlock (pthread_mutex_t *m);

pthread_cond_t vc = PTHREAD_COND_INITIALIZER;
int pthread_cond_init (pthread_cond_t *vc, const pthread_cond_attr *attr);
int pthread_cond_wait (pthread_cond_t *vc, pthread_mutex_t *m);
int pthread_cond_signal (pthread_cond_t *vc);
int pthread_cond_broadcast (pthread_cond_t *vc);
```

Remarks:

- pthread_cond_wait() blocks the calling thread and then release mutex lock so other threads can access the monitor.
- When a woken thread continues to run, mutex lock will be set back so the thread can run its code without being intervened by other threads.

Example: Producers - Consumers

void * producer (void * arg) {	void * consumer (void * arg) {

```
    int item;
    produce_item(&item);

    pthread_mutex_lock(&mutex);
    while (count == N)
        pthread_cond_wait(&empty, &mutex);
    enter_item(item);
    pthread_cond_signal(&full);
    pthread_mutex_unlock(&mutex);
}
```

```
    int item;
    pthread_mutex_lock(&mutex);
    while (count == 0)
        pthread_cond_wait(&full, &mutex);
    item = get_item();
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&mutex);

    consume_item(item);
}
```

Semaphores vs monitors

Semaphores are implemented at the kernel level, can be used for synchronization of both processes or threads.	Monitors are implemented at the programming language level, can be used for threads synchronization.
Programming with semaphores can be tricky and may cause deadlocks if a process is blocked within down(&mutex) … up(&mutex).	Progamming with monitors may look simpler. Mutex locking and unlocking are automatically inserted by the compiler.
Semaphores have their own memory to keep its value. A call to sem_post() increases the semaphore value by 1 whether or not a process is sleeping on the semaphore.	A condition variables has no memory. If there is no thread sleeping on a condition variable, then a call to pthread_cond_signal() on that condition variable will simply be ignored.

Despite the differences, the 2 methods are powerfully equivalent in solving race condition problems. It is possible to implement semaphores using monitors and vice versa.

Implement semaphores using monitors

```
monitor Semaphore () {
    int count;
    condition positive;

    procedure init(int value) {
        count = value;
    }
    procedure down() {
        count--;
        if (count < 0) positive.wait();
    }
    procedure up() {
        count++;
        positive.signal();
    }
}
```

Implement monitors using semaphores

Mesa semantics (sender priority): To run a monitor procedure, a thread has to lock mutex and then unlock it when being done. Then, the woken thread can lock mutex back to continue its run. There may be a situation where another thread locks mutex succesfully before the woken thread continues

to run, thus it may cause a race condition conflict. So with Mesa semantics, the thread when being woken up has to check the logical condition again, by using while() instead of if().

```
Shared data
    Semaphore mutex = new Semaphore(1);
    Semaphore cond = new Semaphore(0);

A monitor procedure
    mutex.down();       // lock mutex
    <procedure body>
    mutex.up();         // unlock mutex when done

cond.signal in a monitor procedure
    if (!cond.empty()) { // if there are still thread sleeping on cond
        cond.up();      //    wake a thread up
    }

cond.wait in a monitor procedure
    mutex.up();         // before going to sleep, mutex must be realsed
    cond.down();        // sleep on cond
    mutex.down();       // lock mutex back, before the thread continues
```

Hoare semantics (receiver priority): A new semaphore, waitings, is introduced to control threads waiting to get access again to the monitor to complete their task. The executing thread still keeps mutex in the locked state and gives control to a waiting thread (sleeping on waitings). Mutex is only unlocked when there are no more threads sleeping on semaphore waitings. After that, other threads may lock mutex and get access to the monitor.

```
Shared data
    Semaphore mutex = new Semaphore(1);
    Semaphore waitings = new Semaphore(0);
    Semaphore cond = new Semaphore(0);

A monitor procedure
    mutex.down();
    <procedure body>
    if (!waitings.empty())   // if there are threads to resume its run
        waitings.up();      //    wake it up, but keep mutex in locked state
    else mutex.up();        // only unlock mutex when waiting threads are done

cond.signal in a monitor procedure
    if (!cond.empty()) {     // if there are threads sleeping on cond
        cond.up();          //    wake it up
        waitings.down();  // the executing thread then sleeps, waiting for turn to get access to the
                          //  monitor again
    }

cond.wait in a monitor procedure
    if (!waitings.empty())  // if there are threads to resume its run
        waitings.up();      //    wake it up, but keep mutex in locked state
    else mutex.up();        // only unlock mutex when waiting threads are done
    cond.down();            // sleeing on cond
```