Synchronization and Communication (part I)

Real-Time Systems, Lecture 3

Martina Maggio

20 January 2017

Lund University, Department of Automatic Control
www.control.lth.se/course/FRTN01

Content

[Real-Time Control System: Chapter 4]

1. Common Resources and Mutual Exclusion
2. Semaphores
3. Monitors

Common Resources and Mutual Exclusion

Communication and Synchronization

Concurrent processes are not independent.

Communication refers to the transfer of information between multiple processes. When processes are communicating, **synchronization** is usually required, as one process must wait for the communication to occur. In some cases, synchronization is the only necessary activity and no communication takes place.

Communicating using Shared Memory

In a multi-threaded application, the threads can use shared memory to communicate. One thread writes a variable in the shared memory and another one reads from the same variable.

Process A	Process B
...	...
`n := n+1;`	`n := n-1;`
...	...

Assume $n = 5$ and that A and B executes once. What is the value of n?

Communicating using Shared Memory

Process A	Process B
...	...
`LOAD R1, N`	`LOAD R2, N`
`ADD R1, 1`	`SUB R2, 1`
`STORE R1, N`	`STORE R2, N`
...	...

The high-level instructions are not atomic. Interrupts that cause context switches may occur in between every machine instruction. In Java the same situation occurs with Java statements and byte code statements. For example, write operations with `double` and `long` variables are not atomic. However, if the data attributes are declared as `volatile` or if special `Atomic` variable classes are used, then the atomicity is extended.

Communicating using Shared Memory

Process A	Process B	N	R1	R2
...				
LOAD R1, N		5	5	
ADD R1, 1			6	
	LOAD R2, N			5
	SUB R2, 1			4
	STORE R2, N	4		
	...			
STORE R1, N		6		
...				

Communicating using Shared Memory

Process A	Process B	N	R1	R2
	...			
	LOAD R2, N	5		5
LOAD R1, N			5	
ADD R1, 1			6	
STORE R1, N		6		
...				
	SUB R2, 1			4
	STORE R2, N	4		
	...			

Race Condition

A **race condition** happens when the output is dependent on the sequence or timing of other uncontrollable events.

We must in some way guarantee that only one thread at a time access the shared variable. This is called **mutual exclusion**. The shared variable can be viewed as a common resource, the code that manipulates it is known as the **critical section**.

Shared Resources

Resources that are common to several proesses:

- shared variables;
- external units (keyboard, printer, screen);
- non-reentrant code[1].

It is important to guarantee exclusive access to shared resources.

[1] A subroutine is called reentrant if it can be interrupted in the middle of its execution and then safely called again ("re-entered") before its previous invocations complete execution. The interruption could be caused by an internal action such as a jump or call, or by an external action such as a hardware interrupt or signal. Once the reentered invocation completes, the previous invocations will resume correct execution.

Mutual Exclusion

Mutual Exclusion is a mechanism that allows a process to execute a sequence of statements (a *critical section*) indivisibly.

It can be implemented **disabling the interrupts**.

- Process A disable the interrupts,
- Process A access the critical section,
- Process A terminates the critical section,
- Process A re-enable the interrupts.

The drawback is that no other process can execute (while we would like that no other process can be simultaneously in a critical section). This approach only works in uni-processor case.

Mutual Exclusion

Mutual Exclusion can also be implemented **using flags** (polling, busy wait).

Process A

```
...
REPEAT until free;
free = false;
access critical section;
free = true;
...
```

Process B

```
...
REPEAT until free;
free = false;
access critical section;
free = true;
...
```

However, one flag is not enough. The flag is also a global variable and as such it is subject to the same read and modify problems. Using Dekker's algorithm[2] it is possible to synchronize two processes using three flags.

[2] http://en.wikipedia.org/wiki/Dekker's_algorithm

Mutual Exclusion

The previous approach would work if the test on the free flag and the assignment were a single **atomic operation**. Atomic test-and-set operations are common in many processors. They read a variable from memory and they assign to it a new value in a single operation, so that it is possible to achieve mutual exclusion with a flag check.

Semaphores

Semaphore

A semaphore is a non-negative counter that can be used for two **atomic** operations.

- wait (while s = 0, do busy wait, end; then s = s-1;)
- signal (s = s+1;)

Atomicity is obtained by disabling interrputs. Implemented with priority-sorted wait queues to avoid busy wait.

Semaphores for Mutual Exclusion

Process A
```
...
wait(mutex);
access critical section;
signal(mutex);
...
```

Process B
```
...
wait(mutex);
access critical section;
signal(mutex);
...
```

The semaphore `mutex` is initialized to 1. The mutex semaphore will only have the values 0 or 1 (known as a binary semaphore).

Semaphores for Synchronization

Asymmetric synchronization

Process A
```
LOOP
 ...
 signal(Aready);
 waitTime(time);
END;
```

Process B
```
LOOP

 wait(Aready);
 ...
END;
```

The semaphore `Aready` is initilized to 0 and may take any non-negative value. It is also called *counting semaphore*. Sometimes different data types are provided for binary and counting semaphores.

Semaphores for Synchronization

Symmetric synchronization

Process A
```
LOOP
 ...
 signal(Aready);
 wait(Bready);
 ...
END;
```

Process B
```
LOOP
 ...
 signal(Bready);
 wait(Aready);
 ...
END;
```

[STORK] Semaphores

```
TYPE
  Semaphore = POINTER TO SemaphoreRec;
  SemaphoreRec = RECORD
    counter : CARDINAL;
    waiting : Queue;
    (* Queue of waiting processes *)
  END;
```

wait(sem);

```
IF sem^.counter = 0 THEN
  insert Running into waiting queue;
ELSE sem^.counter = sem^.counter - 1;
```

signal(sem);

```
IF waiting is not empty THEN
  move the first process in waiting to the ready queue
ELSE sem^.counter = sem^.counter + 1;
```


[STORK] Semaphores

```
PROCEDURE Wait(sem: Semaphore);
VAR
  oldDisable : InterruptMask;

BEGIN
  oldDisable := Disable();
  WITH sem^ DO
    IF counter > 0 THEN
      DEC(counter);
    ELSE
      MovePriority(Running, waiting);
      Schedule;
    END;
  END;
  Reenable(oldDisable);
END Wait;
```


[STORK] Semaphores

```
PROCEDURE Signal(sem: Semaphore);
VAR
  oldDisable : InterruptMask;

BEGIN
  oldDisable := Disable();
  WITH sem^ DO
    IF NOT isEmpty(waiting) THEN
      MovePriority(waiting^.succ, ReadyQueue);
      Schedule;
    ELSE
      INC(counter);
    END;
  END;
  Reenable(oldDisable);
END Signal;
```


[STORK] Semaphores

```
PROCEDURE New
  (VAR semaphore    : Semaphore;
       initialValue : INTEGER;
       name         : ARRAY OF CHAR);
(* Creates the semaphore and initializes
   it to its 'initialValue', 'name' is used for debugging *)
END New;

PROCEDURE Dispose
  (VAR semaphore    : Semaphore);
(* Deletes the semaphore. If there are processes waiting
   for it, an error is reported *)
END Dispose;
```


Semaphores (improved)

The standard way of implementing semaphores can be improved.

Process A (high priority)	Process B (low priority)
`wait(mutex);`	`...`
`...`	`wait(mutex);`
`signal(mutex);`	`...`
`wait(mutex);`	`...`

- A waits (the mutex is free) context switch from A to B
- B waits (inserted into the waiting queue) context switch from B to A
- A signal, B moved into the ready queue, no context switch
- A waits (inserted into the waiting queue) context switch from A to B

Semaphores (improved)

A has higher priority, so it would be better if A hold the mutex.

wait(sem);

```
LOOP
  IF sem^.counter = 0 THEN
    insert Running into waiting queue;
  ELSE
     sem^.counter = sem^.counter - 1;
     EXIT;
  END; (* IF *)
END; (* LOOP *)
```

signal(sem);

```
IF waiting is not empty THEN
  move the first process in waiting to the ready queue
ELSE sem^.counter = sem^.counter + 1;
```

Semaphores (improved)

- A waits (the mutex is free, counter 0) context switch from A to B
- B waits (inserted into the waiting queue) context switch from B to A
- A signal, B moved to the ready queue, no context switch, counter 1
- A waits (counter decreased to 0), A holds the semaphore
- context switch from A to B, B checks again if it holds the semaphore
- B inserted into the waiting queue, context switch from B to A
- A signal, B moved to the ready queue, no context switch, counter 1
- context switch from A to B, B checks again if it holds the semaphore
- B sets the counter to 0 and holds the semaphore

[JAVA] Atomic Classes

In Java there is a small set of classes that allows atomic reads, writes and test-and-set operations. They belong to `java.util.concurrent.atomic`.

- `AtomicInteger`, `AtomicBoolean`, `AtomicLong`, `AtomicReference` and arrays of these
- `boolean compareAndSet(expectedValue, updateValue);`
  test-and-set primitive
- `int get();`
  read method on `AtomicInteger`
- `void set(int value);`
  write method on `AtomicInteger`

[JAVA] Semaphores

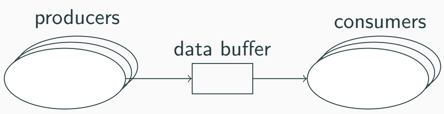
Semaphores were originally not a part of Java. Added in Java 1.5. They are part of `java.utils.concurrent`.

- `Semaphore` class
- `acquire();` corresponds to `wait();`
- `release();` corresponds to `signal();`

It is also possible to implement a Semaphore class using synchronized methods (we will use that, in this course).

Condition Synchronization

A combination of access to common data under mutual exclusion with synchronization of type "data is available". Checking some logical condition on the shared data, when the condition becomes true there is an event.



[STORK] Critical Section

```
TYPE CriticalSection = RECORD
  mutex, change : semaphore;
  waiting       : INTEGER;
  dataBuffer    : buffer;
END;
VAR R: Critical Section;

(* Producer Process *)
...
WITH R DO
  Wait(mutex);
  enter data into buffer;
  WHILE waiting > 0 DO
    DEC(waiting);
    Signal(change);
  END;
  Signal(mutex);
END;
...
```

```
(* Consumer Process *)
...
WITH R DO
  Wait(mutex);
  WHILE NOT "data available" DO
    INC(waiting);
    Signal(mutex);
    Wait(change);
    Wait(mutex);
  END;
  get data from buffer;
  Signal(mutex);
END;
```


[STORK] Critical Section

The condition test must be performed under mutual exclusion.

The while construct is necessary because there might be several consumer process that are waken up at the same time.

A better solution to this problem is obtained via **monitors**.

Semaphores (summary)

- A low-level real-time primitive that can be used to obtain mutual exclusion and synchronization.
- Requires programmer discipline. A misplaced wait or signal is difficult to detect and may have unpredictable and disastrous results.
- Condition synchronization with semaphores is complicated.

Monitors

Monitors

Monitors are a communication mechanism that combines mutual exclusion with condition synchronization. Sometimes they are called `mutex`. A monitor consists of:

- Internal data structures (hidden)
- Mutually exclusive operations upon the data structure (STORK: monitor procedures, Java: synchronized methods)

[STORK] Monitor Procedure

Mutually esclusive section enclosed in an `enter-leave` pair. Acts as a mutual exclusion semaphore.

```
VAR mutex : MONITOR;

(* Monitor *) PROCEDURE Proc();
BEGIN
  Enter(mutex);
  ...
  Leave(mutex);
END Proc;
```


Condition Variables

Condition synchronization is obtained via condition variables (monitor events, event variables). A condition variable is associated with a monitor and has a queue of processes waiting for the event).

A thread can decide to wait for an event. A thread can notify other threads that the event has occurred. These operations are called within the monitor. If a thread decided to wait for an event the monitor is released and the thread reenters the monitor when the event occurs.

[STORK] Condition Variables

Condition variables are represented by variables of type `Event`.

`PROCEDURE Await(ev: Event);`
Blocks the current process and places it in the queue associated with the event. Performs an implicit `Leave`. May only be called from within a monitor procedure.

`PROCEDURE Cause(ev: Event);`
All processes that are waiting in the queue of the event are moved to the monitor queue and inserted according to their priority. If no process is waiting, cause corresponds to a null operation. May only be called from within a monitor procedure.

[STORK] Condition Variables

`PROCEDURE NewEvent (VAR ev : EVENT;`
` mon : MONITOR;`
` name: ARRY OF CHAR;`

Initializes the event and associates it with the monitor guarded by `mon`.

`PROCEDURE DisposeEvent (ev : EVENT);`

Deletes the event.

[STORK] Critical Section Monitor for Producer-Consumer

```
TYPE CriticalSectionMonitor = RECORD
  mon        : Monitor;
  change     : Event;
  databuffer : buffer;
END;
VAR R: CriticalSectionMonitor;

(* Producer Process *)
...
WITH R DO
  Enter(mon);
  enter data into buffer;
  Cause(change);
  Leave(mon);
END;
...
```

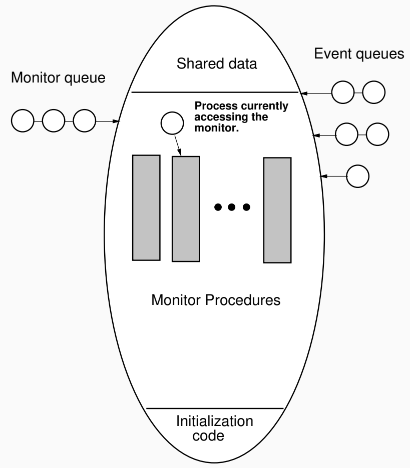
```
(* Consumer Process *)
...
WITH R DO
  Enter(mon);
  WHILE NOT "data available" DO
    Await(change);
  END;
  get data from buffer;
  Leave(mon);
END;
```


Spurious Wakeups

Threads that are waiting for an event should always check that the condition that they are waiting for still is true when they are resumed.

- Several threads may be woken up and it is not sure that the condition still is true when a thread eventually executes.
- Certain platforms, e.g., certain Java platforms and POSIX platforms may generate *spurious wakeups* (wakeups caused by the coputing platform).

[STORK] Monitor



[STORK] Monitor Implementation

```
TYPE
  Monitor = POINTER TO MonitorRec;
  Event = POINTER TO EventRec;

  MonitorRec = RECORD
    waiting  : Queue;
    blocking : ProcessRef;
    events   : Event;
  END;

  EventRec = RECORD
    evMon   : Monitor;
    waiting : Queue;
    next    : Event;
  END;
```


[STORK] Monitor Implementation (Enter)

```
PROCEDURE Enter(mon: Monitor);

VAR
  oldDisable : InterruptMask;

BEGIN
  WITH mon^ DO
    oldDisable := Disable();
    LOOP
      IF blocking = NIL THEN
        blocking := Running;
        EXIT;
      ELSE
        MovePriority(Running,waiting);
        Schedule;
      END;
    END;
  Reenable(oldDisable);
  END;
END Enter;
```

[STORK] Monitor Implementation (Leave)

```
PROCEDURE Leave(mon: Monitor);

VAR
  oldDisable : InterruptMask;

BEGIN
  WITH mon^ DO
    oldDisable := Disable();
    blocking := NIL;
    IF NOT IsEmpty(waiting) THEN
      MovePriority(waiting^.succ,ReadyQueue);
      Schedule;
    END;
    Reenable(oldDisable);
  END;
END Leave;
```


[STORK] Monitor Implementation (Await)

```
PROCEDURE Await(ev: Event);

VAR
  oldDisable : InterruptMask;

BEGIN
  oldDisable := Disable();
  Leave(ev^.evMon);
  MovePriority(Running,ev^.waiting);
  Schedule;
  Reenable(oldDisable);
  Enter(ev^.evMon);
END Await;
```


[STORK] Monitor Implementation (Cause)

```
PROCEDURE Cause(ev : Event);

VAR
  oldDisable : InterruptMask;
  pt         : ProcessRef;

BEGIN
  oldDisable := Disable();
  LOOP
    pt := ev^.waiting^.succ;
    IF ProcessRef(ev^.waiting) = pt THEN
      EXIT (* Event queue empty *)
    ELSE
      MovePriority(pt,ev^.evMon^.waiting);
    END;
  END;
  Reenable(oldDisable);
END Cause;
```


[JAVA] Synchronized Methods

Monitors are implemented as Java objects with *synchronized methods*. The Java platform maintains a lock for every object that has synchronized methods. Before a thread is allowed to start executing a synchronized method it must obtain the lock. When a thread finishes executing a synchronized method the lock is released. Threads waiting to acquire a lock are blocked.

Java does not specify how blocked threads are stored or which policy that is used to select which thread that should acquire a newly released lock. Often a priority-sorted queue is used.

[JAVA] Synchronized Methods

```
public class MyMonitor {

  public int x;

  public synchronized void method1(...) {
    // Code for method 1
  }

  public synchronized void method2(...) {
    // Code for method 2
  }
}
```


[JAVA] Synchronized Methods

```
public class MyMonitor {

  public int x;

  public synchronized void method1(...) {
    // Code for method 1
  }

  public void method2(...) {
    // Code for method 2
  }
}
```

If `method2` is unsynchronized it can access data without prodtection. Public attributes can be accessed directly with the dot-notation without protection. May be dangerous.

[JAVA] Synchronized Methods

Java locks are reentrant. A thread holding the lock for an object may call another synchronized method of the same lock. In STORK this would lead to a deadlock. Static methods can also be synchronized.

- Each class has a class lock, the class lock and the instance lock are distinct, unrelated, locks.

[JAVA] Synchronized Blocks

Synchronization can be provided for smaller blocks of code than a method.

```java
public void MyMethod() {
  // unsychronized code
  synchronized (this) { // could also be synchronized (obj)
    // synchronized code
  }
  // unsychronized code
}
```

Acquires the same object lock as if it had been the whole method that had been synchronized. Using synchronous blocks it is also possible to synchronize on other objects. The code of the synchronized block is, from a synchronization point of view, executed as if it instead had been a call to a synchronized method of the object `obj`.

[JAVA] Condition Synchronization

However, Java only supports a single, anonymous condition variable per locked object.

The Java method `wait()` corresponds to STORK's `Await(ev : Event)`:

- method of class `Object`;
- no argument (single, anonymous condition variable);
- may only be called within synchronization;
- the calling thread releases the lock and becomes blocked;
- Java does not specify how the blocking is implemented, however, in most implementations a priority-sorted queue is used.

[JAVA] Condition Synchronization

The Java method `wait()` corresponds to STORK's `Await(ev : Event)`:

- throws the runtime exception `IllegalMonitorStateException` if the current thread is not the owner of object's lock;
- throws the checked exception `InterruptedException` if another thread has interrupted the current thread;

```java
try {
  wait();
} catch (InterruptedException e) {
  // Exception handling
}
```

- takes an optional timeout argument, `wait(long timeout)`;
- the thread will wait until notification or until the timeout period has elapsed.

[JAVA] Condition Synchronization

The Java method `notifyAll()` corresponds to STORK's `Cause(ev : Event)`:

- method of class `Object`;
- no argument;
- may only be called within synchronization;
- all threads waitining for the anonymous event for the object are woken up (moved to the "waiting" queue of the object)

The Java method `notify()` just wakes up one thread:

- not available in STORK;
- not specified which thread is woken up;
- in most implementations it is the first one in the queue;
- may only be called within synchronization.

[JAVA] Anonymous Condition Variables

Having only one condition variable per synchronized object can lead to inefficiency. Assume that several threads need to wait for different conditions to become true. With Java synchronized objects the only possibility is to notify all waiting threads when any of the conditions become true. Each thread must then recheck its condition, and, perhaps, wait anew. May lead to unnecessary context switches.

Java design flaw!

[JAVA] Producer-Consumer

Multiple producers and consumers. Buffer of length 1 containing an integer. Four classes: Buffer, Consumer, Producer, Main.

```java
public class Buffer {
  private int data;
  private boolean full = false;
  private boolean empty = true;
```


[JAVA] Producer-Consumer

```java
  public synchronized void put(int inData) {
    while (full) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    data = inData;
    full = true;
    empty = false;
    notifyAll();
  }
```


[JAVA] Producer-Consumer

```java
  public synchronized int get() {
    while (empty) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    full = false;
    empty = true;
    notifyAll();
    return data;
    }
}
```


[JAVA] Producer-Consumer

```java
public class Consumer extends Thread {
  private Buffer b;

  public Consumer(Buffer bu) {
    b = bu;
  }

  public void run() {
    int data;
    while (true) {
      data = b.get();
      // Use data
    }
  }
}
```


[JAVA] Producer-Consumer

```java
public class Producer extends Thread {
  private Buffer b;

  public Producer(Buffer bu) {
    b = bu;
  }

  public void run() {
    int data;
    while (true) {
      // Generate data
      b.put(data);
    }
  }
}
```


[JAVA] Producer-Consumer

```java
public class Main {

  public static void main(String[] args) {

    Buffer b = new Buffer();
    Producer w = new Producer(b);
    Consumer r = new Consumer(b);

    w.start();
    r.start();

  }
}
```

[JAVA] Semaphores

Semaphores can be implemented using synchronized methods.

```java
public final class Semaphore {
  // Constructor that initializes the counter to 0
  public Semaphore();

  // Constructor that initilizes the counter to init
  public Semaphore(int init);

  // The wait operation (Wait is a Java keyword)
  public synchronized void take();

  // The signal operation
  public synchronized void give();
}
```


[JAVA] Condition variables

Condition variables can also be implemented using synchronization. Can be used to obtain condition synchronization in combination with class Semaphore.

```java
public class ConditionVariable {

  // Constructor that associates the condition variable
  // with a semaphore
  public ConditionVariable(Semaphore sem);

  // The wait operation
  public void cvWait();

  // The notify operation
  public synchronized void cvNotify();

  // The notifyAll operation
  public synchronized void cvNotifyAll();
}
```


Homework

Study the implementation of the classes Semaphore and ConditionVariable in the text book.

[JAVA] Producer-Consumer II

Using classes Semaphore and ConditionVariable. Only the class Buffer changes.

```java
public class Buffer {
  private Semaphore mutex;
  private ConditionVariable nonFull, nonEmpty;
  private int data;
  private boolean full = false;
  private boolean empty = true;

  public Buffer() {
    mutex = new Semaphore(1);
    nonEmpty = new ConditionVariable(mutex);
    nonFull = new ConditionVariable(mutex);
  }
```


[JAVA] Producer-Consumer II

```java
  public void put(int inData) {
    mutex.take();
    while (full) {
      try {
        nonFull.cvWait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    data = inData;
    full = true;
    empty = false;
    nonEmpty.cvNotifyAll();
    mutex.give();
  }
```


[JAVA] Producer-Consumer II

```java
  public int get() {
    int result;

    mutex.take();
    while (empty) {
      try {
        nonEmpty.cvWait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    result = data;
    full = false;
    empty = true;
    nonFull.cvNotifyAll();
    mutex.give();
    return result;
    }
}
```

[JAVA] Monitors

The following basic programming rules are good practice to follow.

- Do not mix a thread and a monitor in the same object/class:
  - a monitor should be a passive object with synchronized access methods;
  - you may use a passive monitor object as an internal object of another, possible active, object.
- Do not use synchronized blocks unnecessarily.

Monitors (summary)

- A high-level primitive for mutual exclusion and condition synchronization.
- Implemented using synchronized methods/blocks in Java.
- Semaphores and condition variables can be implemented.

[C-Linux] Synchronization

Linux supports synchronization in many ways. Part of it is provided by the Linux Kernel and is intended to be used primarily by the Kernel but can be used also by userspace processes (often inefficient). Part of it is provided by the Posix (`pthread`[3]) library and is intended to be used by userspace applications.

[3] https://computing.llnl.gov/tutorials/pthreads/

[C-Linux] Kernel Synchronization

- Spin Lock (Kernel): similar to a binary semaphore, but a thread that wants to take a lock held by another thread will wait via spinning (busy waiting); assumes that the thread holding the lock can be preempted; inefficient use of the CPU; should only be held for very short periods of time.
- Semaphores (Kernel): counting semaphores; operations (`up()` for wait and `down()` for signal.

[C-Linux] Pthread Synchronization: Thread Creation

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 4

void *PrintHello (void *threadid) {
  long tid; tid = (long)threadid;
  printf("Hello World! It's me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  int rc; long t;
  for (t=0; t<NUM_THREADS; t++) {
    printf("In main: creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
    if (rc) { printf("ERROR\n"); exit(-1); }
  }
  pthread_exit(NULL);
}
```


[C-Linux] Pthread Synchronization: Mutex

```c
#include <pthread.h>
int main (int argc, char *argv[]) {
  pthread_mutex_t mx;
  pthread_mutex_init(&mx, NULL);
  ...
  /* Locking and unlocking the mutex */
  pthread_mutex_lock (&mx);
    // Here goes the critical section code
  pthread_mutex_unlock (&mx);
  ...
  pthread_mutex_destroy(&mx);
  exit(0);
}
```

[C-Linux] Pthread Synchronization: Condition Variables

- `pthread_cond_wait` unlocks the mutex and waits for the condition variable to be signaled;
- `pthread_cond_timedwait` lace limit on how long it will block;
- `pthread_cond_signal` restarts one of the threads that are waiting on the condition variable cond;
- `pthread_cond_broadcast` wakes up all threads blocked by the specified condition variable.