

Unit 1

Introduction and Basic Concepts

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads**
- 1.7 Architectures providing concurrency**
- 1.8 Java Threads**
- 1.9 Pascal FC**

1.1 Baseline definitions

1.2 Benefits and issues of concurrency

1.3 Correctness

1.4 Atomic statements and volatile variables

1.5 Specification of Concurrent Execution

1.6 Processes vs. Threads

1.7 Architectures providing concurrency

1.8 Java Threads

1.9 Pascal FC

1.1 Baseline definitions

- A **program** is a set of instructions written in one or several files.

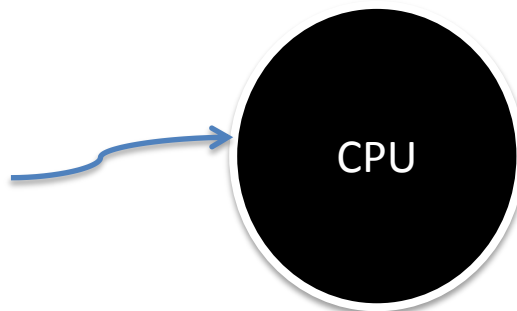
```
def process_row(row):
    first = row[0].strip()
    last = row[1].strip()
    address = row[2].strip()

    try:
        (first, middle) = first.split(' ')
        (address, middle) = first.split(' ')
    except ValueError:
        pass

    #print first, last
    return (first, last, address)

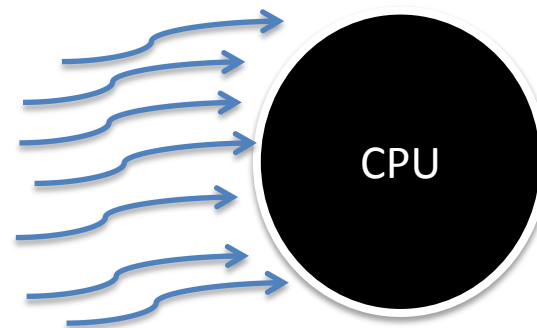
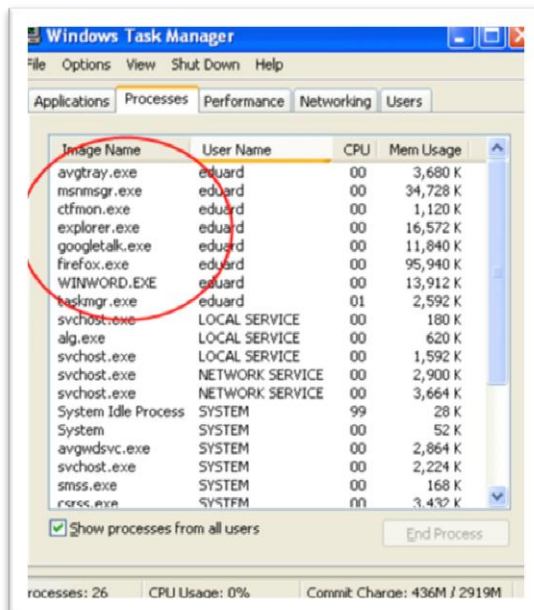
def process_file(f):
    voters = []
    for row in f:
        try:
            if (len(row) > 0):
                (first, last, address) = process_row(row)
                first = first.upper()
                last = last.upper()
                voters.append((first, last, address))
        except IndexError:
            print "Error in row"
            print "[", row, "]"
            raise
    print voters
```

- When you compile and run the program, you create a **process** which is the dynamic execution of the compiled instructions in the CPU.



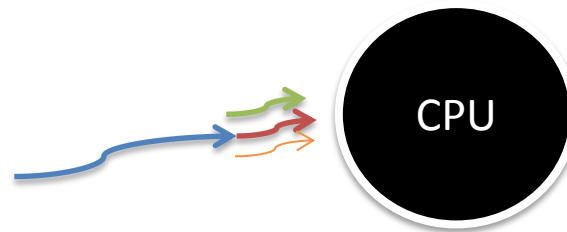
1.1 Baseline definitions

- The process needs some help to run successfully: Program Counter (PC), CPU registers, stack, stack pointer, main memory for global variables.
- Each time you run a compiled program, you create one new process with new PC, stack, registers and memory addresses.
- So you have one process running in your computer per program and instance:
Multitask Operating Systems.



1.1 Baseline definitions

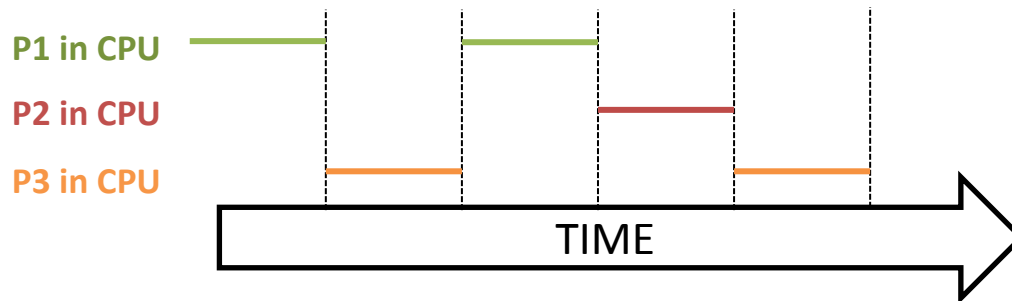
- Moreover, each process may contain several sub-processes or threads.



- These threads share the same main memory for global variables. But they still have different PC, stack and registers.
- Blue process: Intelligence test
 - Green thread: display questions and get input
 - Read thread: timer
 - Orange thread: pop-ups advertising

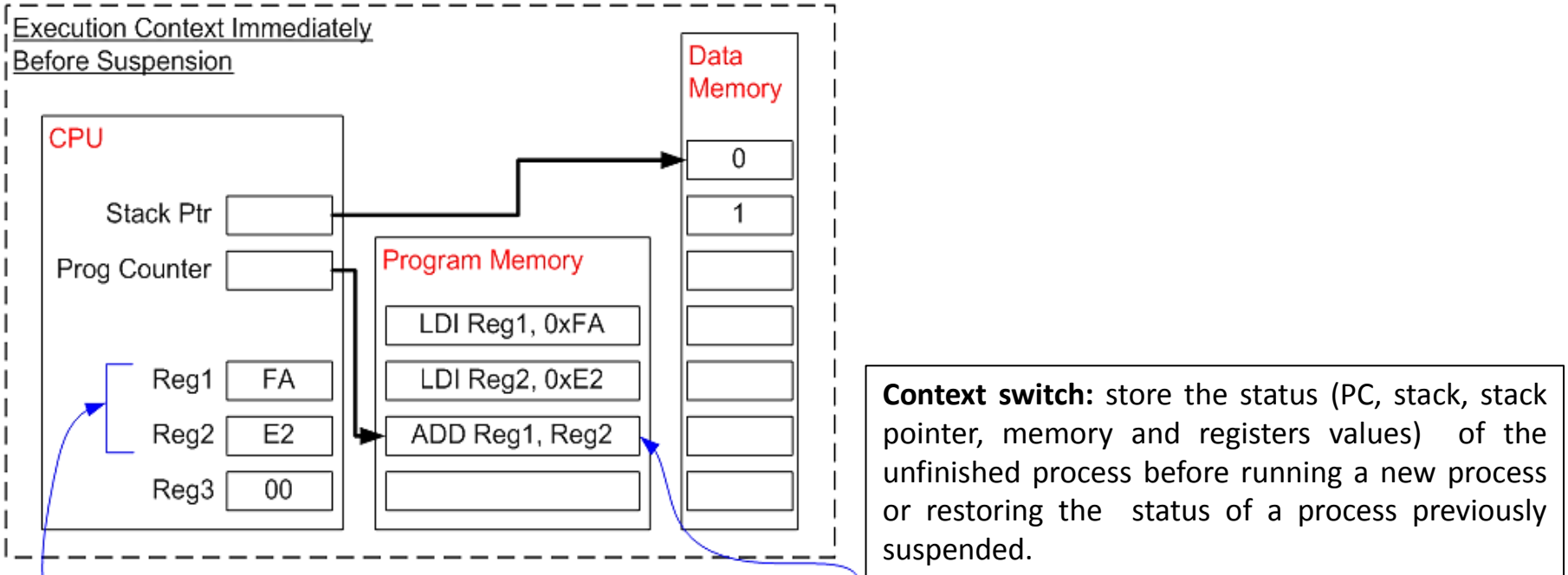
1.1 Baseline definitions

- The **scheduler** is a built-in process in the kernel of the operating system which decides when a process/thread enters or exits the CPU.



- Each period of time during which a process is granted the CPU is called **time-slice**.
- A process may not be finished when the scheduler decides that another process must enter in CPU. So what about the unfinished process?
 - Can the new running process use the same memory addresses or registers?
 - Should it start from the beginning again once it re-enters the CPU?

1.1 Baseline definitions



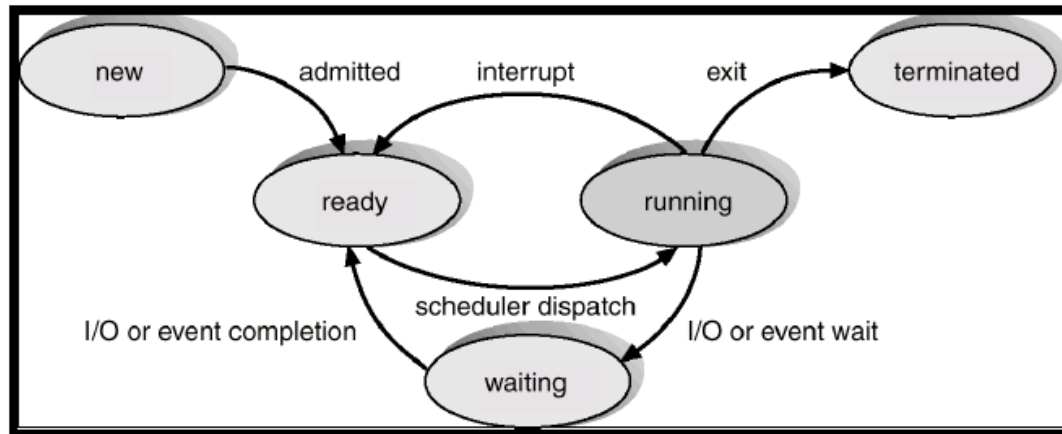
The task gets suspended as it is about to execute an ADD.

The previous instructions have already set the registers used by the ADD. When the task is resumed the ADD instruction will be the first instruction to execute. The task will not know if a different task modified Reg1 or Reg2 in the interim.

Image obtained from www.freertos.org

1.1 Baseline definitions

- Thus, a process may be running, waiting, stopped... A generic state diagram for a process is:



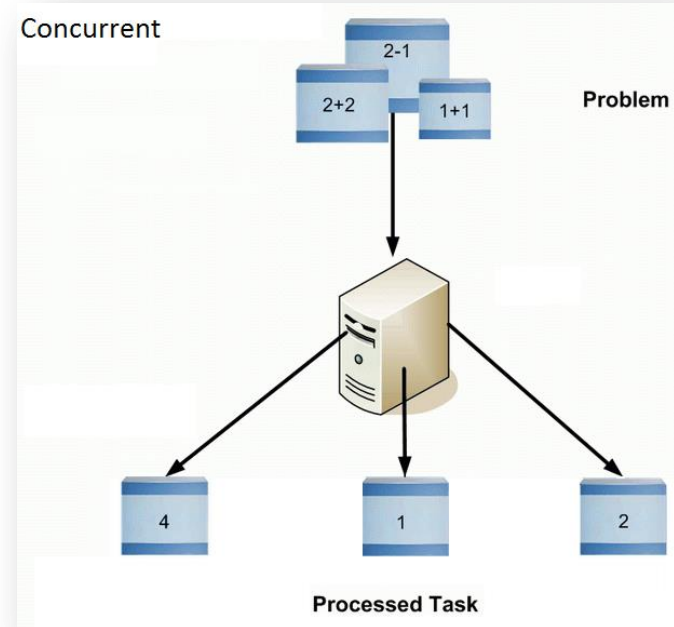
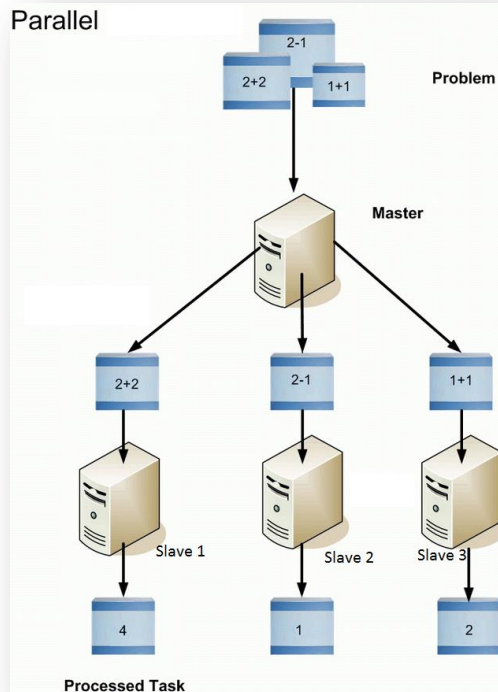
- **Ready**: the process wants to start/resume its execution.
- **Running**: the process is inside CPU and the current context is its own.
- **Waiting**: the process is waiting for a signal, or it has been interrupted by an I/O device.
- *Admitted*: the process enters in the ready-queue managed by the OS scheduler
- *Interrupt*: the scheduler decides to give a new time-slice to another process.
- *Sch. dispatch*: the scheduler gives a new time-slice to a ready process.

All transitions to or from *running* perform a context switch.

1.1 Baseline definitions

- CPUs have such a high clock-frequency that we are not usually aware of context switches, thus having the abstraction or the appearance that all processes and I/O are being executed at the same time: operating system, spreadsheet, word processor, internet browser,...
- **Parallel execution of processes:** two or more processes running at the same time (one CPU per process). Physically simultaneous processing.
- **Concurrent execution of processes:** two or more processes which need to share at least one CPU to run their machine code instructions. Logically simultaneous processing.
- **Concurrent programming:** application of techniques which help us manage the underlying problems arising from concurrent execution: mutual exclusions and synchronization of processes .
- The same definitions hold for execution of threads or sub-processes.
- Your lab assignments apply concurrent programming of threads. That is, 1 process with several threads which share 1 CPU.

1.1 Baseline definitions

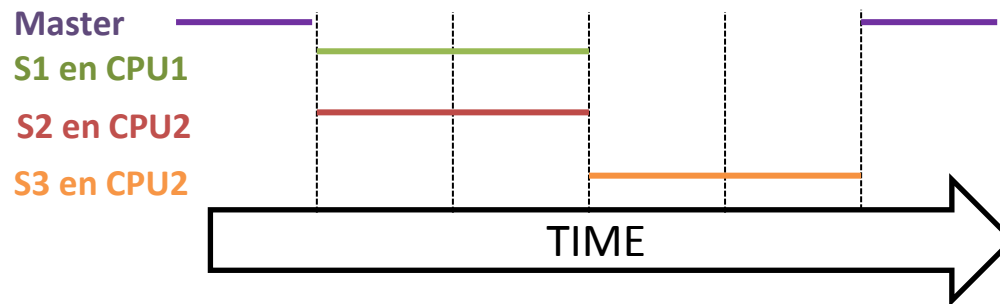


- In parallel computation, memory may be shared or not. In this example, it is not.
- Concurrent computation always uses shared memory.
- Whenever **memory** is **shared**, access must be controlled.
- When processes work together to solve a given problem, **synchronization** is necessary.
- From now on, with *concurrent* we refer to any computation with shared memory or need for synchronization.

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness
- 1.4 Atomic statements and volatile variables
- 1.5 Specification of Concurrent Execution
- 1.6 Processes vs. Threads
- 1.7 Architectures providing concurrency
- 1.8 Java Threads
- 1.9 Pascal FC

1.2 Benefits and issues of concurrency

- **Performance** gain from multiprocessing hardware
 - Parallelism for numeric computations
 - Several slave processes and one master which merges the results.
 - E.g.: In the previous example of parallel computation, if we only had 2 CPUs for slave processes:



- If we only had 1 CPU, the processing would take 2 extra time-slices.

1.2 Benefits and issues of concurrency

- Increased application **throughput**
 - an I/O call blocks the thread in the corresponding CPU without delaying the others.
 - E.g.: A process for a video-game with 2 threads: moving the enemy and moving Player 1. The enemy thread is always working, while the thread for player 1 is blocked waiting for the input device (mouse, keyboard, sensor,...)



1.2 Benefits and issues of concurrency

- Increased application **responsiveness**

- high priority threads.
- E.g.: the main thread is not responding and it would never release the CPU, but a higher priority thread gains access and gives us the opportunity to stop it.



- More **appropriate structure**

- for programs which interact with the environment (sensors, data analysis, alarms), control multiple activities and handle multiple events (chats).
- Databases: several readers, one writer.

1.2 Benefits and issues of concurrency

- Depending on the programming techniques, the resulting high-level code may result very **difficult to understand or maintain**.
- **Error-prone**: as you will find in your lab assignments, most of the concurrent programming techniques used for controlling shared-memory or synchronization can be distributed along all your code, making it difficult to get rid of execution-time errors:
 - E.g.: access to memory is never unlocked
 - E.g.: signals for synchronization do not reach the appropriate process
- **Indeterminism** in execution: the scheduler cannot be controlled, so:
 - **Interleaving**
 - we cannot predict the execution order for threads
 - we cannot predict how many lines of code a thread will run in each time-slice
 - Thus, different runs of a concurrent program produces different interleaving of its threads access to CPU.
 - But, if the concurrency is correctly controlled/programmed, the result is always the same.
 - Program expecting the worst case: each line of code is interleaved.

1.2 Benefits and issues of concurrency

- Example of indeterminism in execution without controlling access to memory:

Shared memory	
int x;	
Process p	Process q
x = 2	x = 1
int y = x + 3	int z = x + 4
print y	print z

- Instructions p1 and q1 are atomic: 1 machine instruction after compilation.
- Instructions p2 and q2 are not atomic: 3 machine instructions: LOAD, ADD and STORE which can be interleaved due to the scheduler. But they do not affect x.
- Try to understand why the possible resulting values for variable y are: 5 and 4.
- Try to understand why the possible number of context switches are: 1 to 9.

1.2 Benefits and issues of concurrency

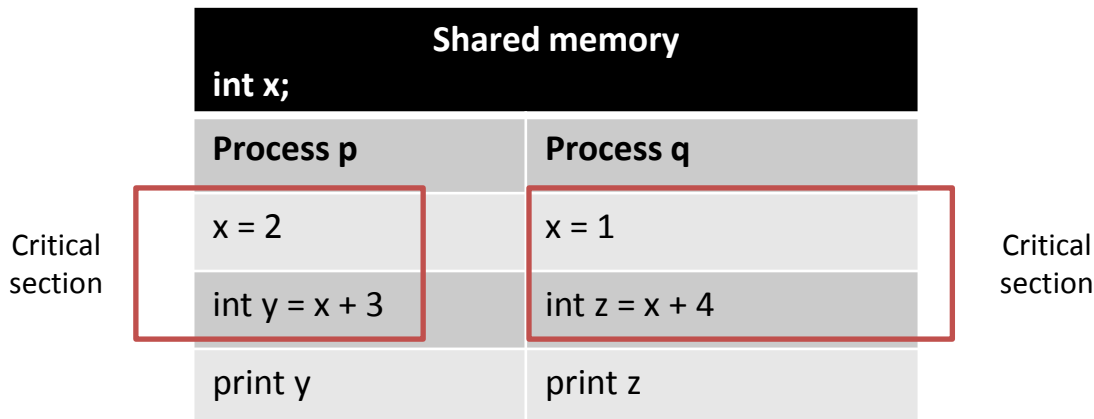
- Example of indeterminism in execution controlling access to memory:

Shared memory	
int x;	
Process p	Process q
<i>lock access to x</i>	<i>lock access to x</i>
x = 2	x = 1
int y = x + 3	int z = x + 4
<i>unlock access to x</i>	<i>unlock access to x</i>
print y	print z

- Try to understand now why y is always 5.
- When one process holds the lock, the other is suspended until the lock is released.
- Number of (working) context switches: 1 to 3. (switch to a process which cannot hold the lock, makes it be suspended and switch again)
- The order is still non-deterministic, but the result is always the same.

1.2 Benefits and issues of concurrency

- **Critical section:** it is the portion of code which accesses a shared resource which might be changed by some process.
- If more than 1 process is running code from the critical section, unexpected results occur.
- In the previous example, without controlling access to memory, the critical section are those instruction which access variable x.



- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables
- 1.5 Specification of Concurrent Execution
- 1.6 Processes vs. Threads
- 1.7 Architectures providing concurrency
- 1.8 Java Threads
- 1.9 Pascal FC

1.3 Correctness

- As we said, concurrent programming is error-prone.
- If you run your concurrent program once and you get the **expected result, this does not mean it is correct**. If you run several times, some error may appear due to a wrong protection of the critical section or wrong synchronization.
- A concurrent program is correct if properties of **safety** and **liveness** hold:
 - a liveness property must eventually become true
 - a safety property must always be true.
- Due to the unpredictability of interleaving:
 - it is **impossible to debug** a concurrent program to find the source of an error.
 - Correctness can only be **proved with formal specification** methods (out of the scope of this course).

1.3 Correctness

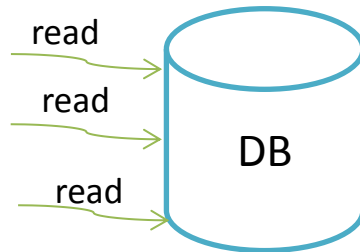
- Safety properties (or problems):

1) Mutual exclusion: only 1 process can be running inside the critical section.

E.g. Several readers and writers and 1 database.

No writers

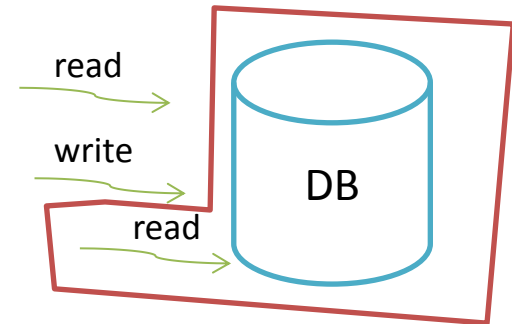
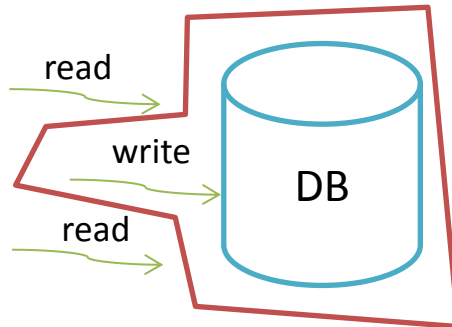
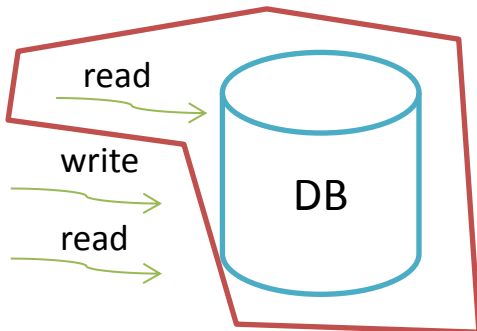
There is no need to define a critical section



1 writer

Critical section is access to database.

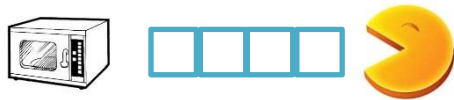
Should 2 readers gain access without blocking to each other?



1.3 Correctness

2) **Synchronization**: when a process must wait for an I/O event or for another process to do something.

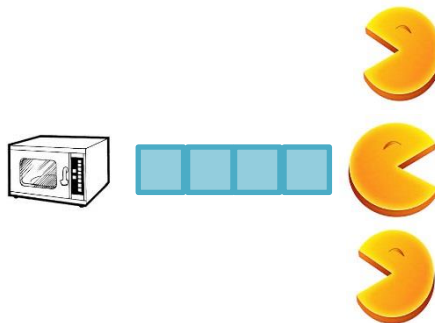
E.g.: Producer-Consumer problem: you cannot consume if the buffer is empty, and you cannot produce if the buffer is full.



Thread Consumer waits for the Producer to insert something into the buffer.
If it does not wait --> **null pointer!**



Thread Producer waits for the Consumer to take one element out of buffer.
If it does not wait --> **index out of bounds!**

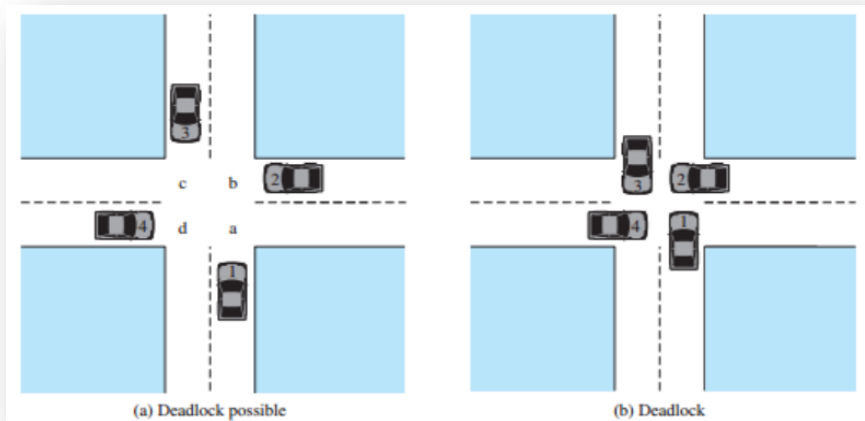


We may have several consumers... and even several producers.

Mutual exclusion and Synchronization may get very hard depending on the tools that our concurrent programming high-level language provides.

1.3 Correctness

3) Deadlock: process A is waiting for process B to do something, but process B is waiting for process A (or a process which depends on A) to do another thing. Process A and B will be always waiting.



Car 1 cannot go on until 2 goes on
Car 2 cannot go on until 3 goes on
Car 3 cannot go on until 4 goes on
Car 4 cannot go on until 1 goes on
DEADLOCK

Operating Systems: Internals and Design Principles. Ch. 6. W. Stalling.

- Once in deadlock, processes are blocked forever.
- There is **no general solution for deadlock**. In programming time, you must think of possible deadlocks.
- Possible solutions:
 - Assign a maximum time of wait (cars go backwards after 10 seconds blocked)
 - Mutual exclusion (the cross-road is a critical section, so only 1 car is allowed at a time)
 - Assign a permanent order in which threads must gain access to the resource

1.3 Correctness

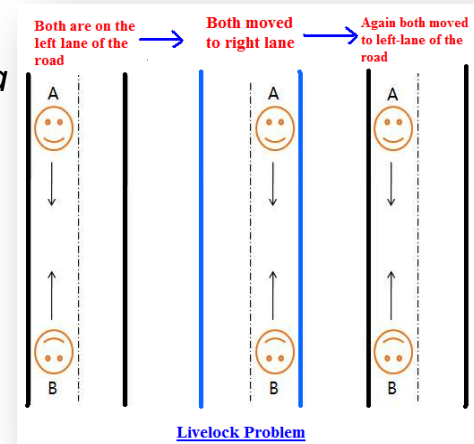
- The **safety** of a concurrent program is assured if it **always** meets these criteria:
 - our code guarantees mutual exclusion of critical sections,
 - processes are synchronized, and
 - processes never reach a deadlock status.

and then we say that our program is **thread-safe**.

- Liveness properties (or problems):

1) **Livelock**: two processes enter in livelock when they are doing some work responding to each other, but none of them makes any progress. E.g.:

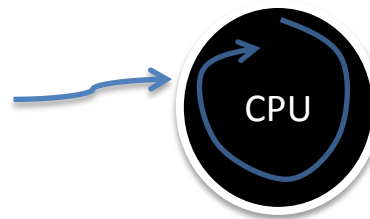
- *Thread A needs the light on to wake up, but thread B needs the light off to sleep. When A switches the light on, B switches it off, and then A switches on again. They will be looping forever.*
- *Two cars on a road, or two people walking through a corridor, moving left or right at the same time.*



1.3 Correctness

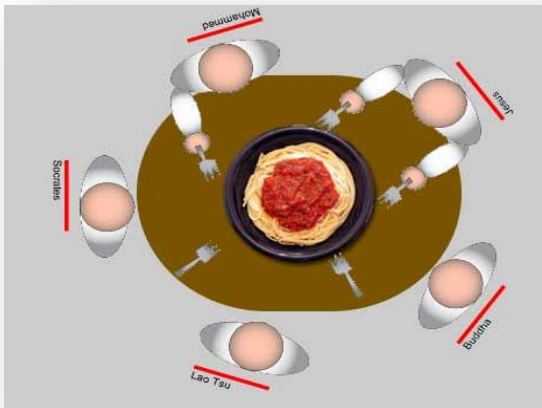
2) Starvation: a process is virtually dead by starvation when it never gets access to CPU meanwhile other threads do. That is, it never gets out of the *ready* status. This may happen per several **reasons**:

- Other thread never releases the CPU

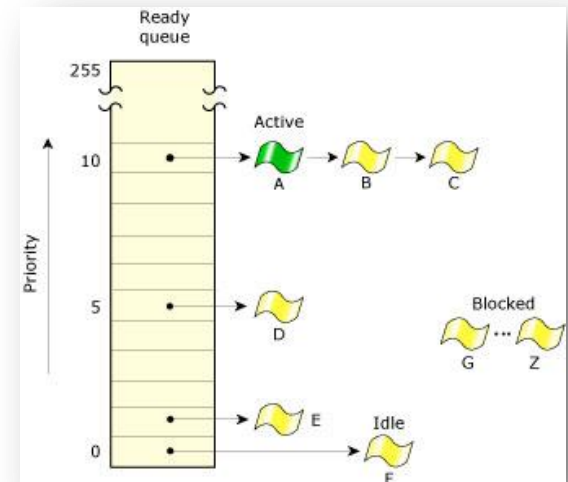


- Other thread never releases a resource which grants access to CPU (locks, semaphores,...)

- Other threads have higher priority



www.danfinlay.com/



www.qnx.com

1.3 Correctness

- If it may happen that our processes or threads fall sometimes in livelock or starvation, then our program does not provide **liveness**.
- A concurrent program cannot (should not) be debugged using traditional methods, since liveness problems may happen from time to time. Moreover, the use of a **debugger makes changes in the scheduler** so they might never appear.
- **Careful design** is encouraged, as well as the use of the **highest-level tools** available to solve our concurrent problem.
- A program is **correct** if it is thread-safe and free of liveness problems.

1.3 Correctness

- If there is not any relation between the activity of 2 processes, we say they are **independent**.
- We can find two kinds of interaction between processes which might make correctness fail:
 - **Competence**: several processes must share common resources from the system (processor, memory, disk, printers,...), so they need to compete to get them. When the shared resource is a variable in memory, the competition is known as **race condition**. The value of such variable might be different depending of which process gains access to it first.
 - **Cooperation**: several processes must work on different parts of a problem to solve it together.

1.3 Correctness

Competence and cooperation happen by means of one or more of the following activities:

- **Communication:** interchange of information between processes. “I want to print!”, or “here is the result of your inquiry”.
- **Synchronization:**
 - **Conditional synchronization:** one or more processes wait until another process or processes do some work. “I will print the document when the mail server sends it to me”.
 - **Mutual Exclusion:** only 1 process can be running inside the critical section or using a shared resource. “I am printing now, so the others jobs need to wait”.

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution
- 1.6 Processes vs. Threads
- 1.7 Architectures providing concurrency
- 1.8 Java Threads
- 1.9 Pascal FC

1.4 Atomic and Volatile

- A statement is **atomic** if its resulting machine code is executed without any interleaving.

- An assignment like

$x = y$

is compiled to STORE and WRITE instructions, so interleaving may happen.

- Some languages, like Java, assure us that **assignment** and evaluation of boolean conditions are executed in an atomic manner.

$x = 3$ is atomic

$x = x + 3$ is not atomic!!

- A **combination** of atomic statements is not atomic!

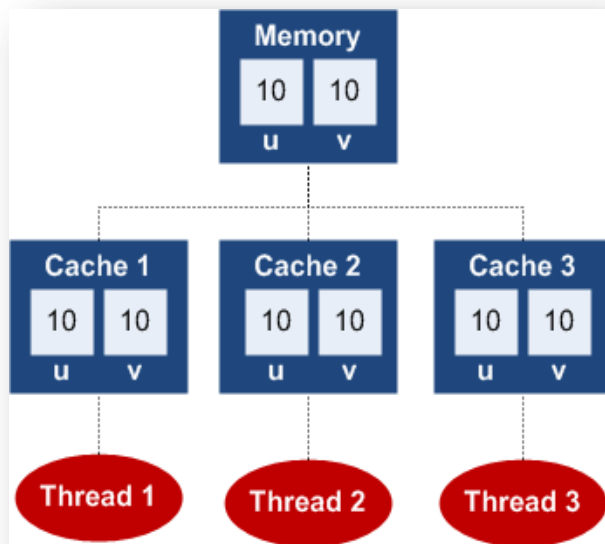
if(condition) $x = 3$

if *condition* is true, another thread may set it to false before x is set to 3.

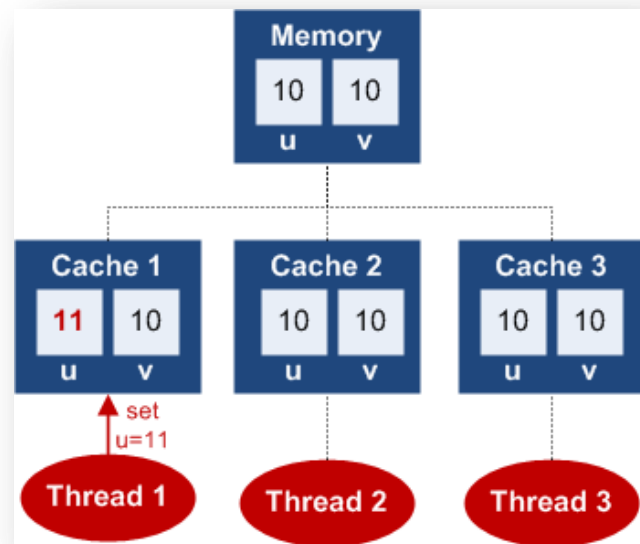
- Assignment statements of **long** and **double** variables are not atomic! (they need 2 words to store their value, and words reading may suffer of interleaving)

1.4 Atomic and Volatile

- Shared variables are first created in main memory.
- In compiling time, the compiler optimizes our code by making each process **cache a copy of all variables** or move them to registers.
- So, what happens if 3 threads work with a copy of a shared variable?

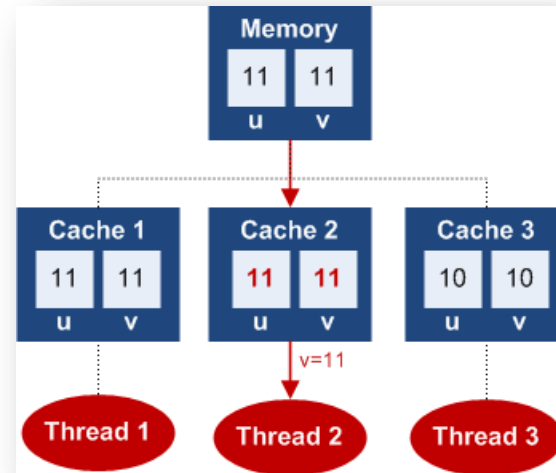
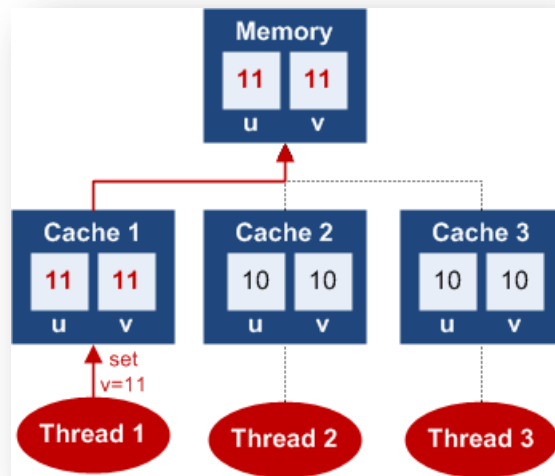


<http://igoro.com/>



1.4 Atomic and Volatile

- In order to avoid this, Java lets us declare variables as *volatile* to tell the compiler that a variable is accessed by 2 or more threads.
- Thus, sets and gets are now volatile. Some languages, like C#, make volatile reads and writes by default.
- Of course, this makes the computation with such variables slower because each access to it derives in reading or writing in main memory.



- Variable *v* is volatile, but *u* is not. However, the set of a new value to *v* flushes all cache values in main memory.
- A get call to *v* in Thread 2 flushes all main memory values in its cache.

1.4 Atomic and Volatile

- Reads and writes of volatile variables are atomic, including long and double.
- Imagine 10 processes running the following code on a volatile shared variable which is initiated as *double d=0* :

```
for(int i=0; i<5;i++) d++;
```

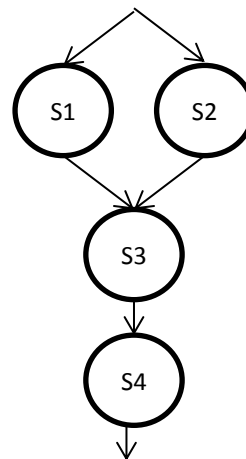
- Can we say that the resulting value of d is 50?

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads
- 1.7 Architectures providing concurrency
- 1.8 Java Threads
- 1.9 Pascal FC

1.5 Specification of Concurrent Execution

- There exist several methods to specify the order of execution of the instructions in our concurrent program. Two well-known methods are:
 - Precedence graphs
 - Cobegin-coend statements
- **Precedence Graphs:** an acyclic graph, in which a node represents a set of instructions. An arrow from node A to node B means that B cannot start until A ends. Two parallel nodes means they can be executed concurrently.

S1 \rightarrow a := x + y;
S2 \rightarrow b := z - 1;
S3 \rightarrow c := a - b;
S4 \rightarrow w := c + 1;



1.5 Specification of Concurrent Execution

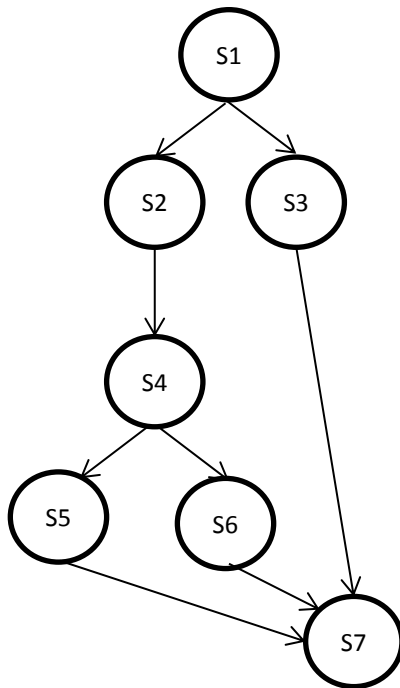
- **Cobegin/coend block:** instructions which can be executed in parallel are written inside a cobegin/coend block. Instructions inside these blocks can be run in any order (concurrently), the rest is run sequentially.

S1 \rightarrow a := x + y;
S2 \rightarrow b := z - 1;
S3 \rightarrow c := a - b;
S4 \rightarrow w := c + 1;

```
begin
  cobegin
    a := x + y;
    b := z - 1;
  coend;
  c := a - b;
  w := c + 1;
end;
```

1.5 Specification of Concurrent Execution

Use the precedence graph to write the specification of concurrent execution with cobegin/coend statements.

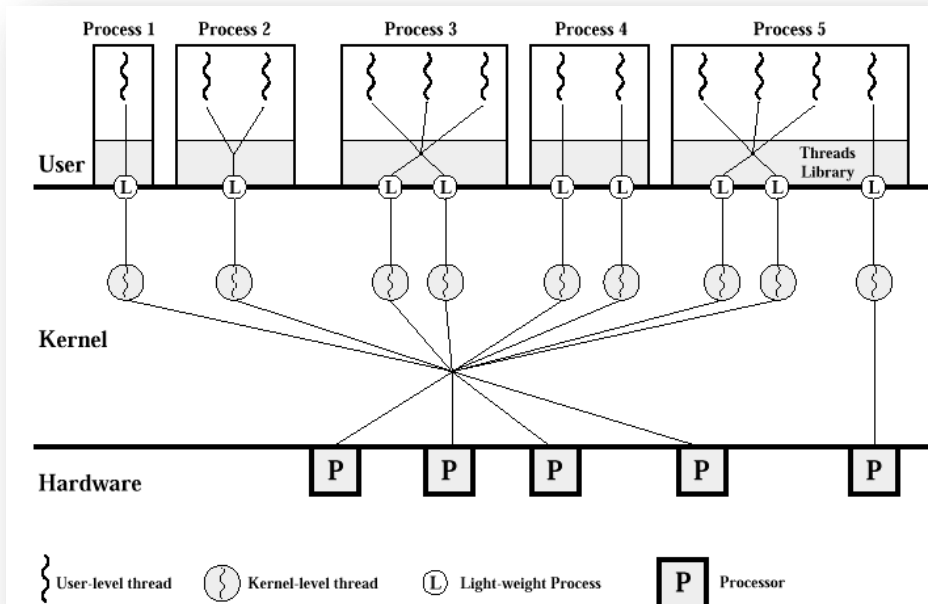


S1;

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads**
- 1.7 Architectures providing concurrency
- 1.8 Java Threads
- 1.9 Pascal FC

1.6 Processes vs. Threads

- Processes are run by the operating system.
- Threads** are independent running sequences inside a process.
- Both processes and threads can be run concurrently:
 - 1st level of concurrency: processes
 - 2nd level of concurrency: threads
- Context **switch is lighter in threads** than in processes:
 - Some of the context information of a process belongs to the OS kernel
 - All the information related to a thread belongs to the OS user space

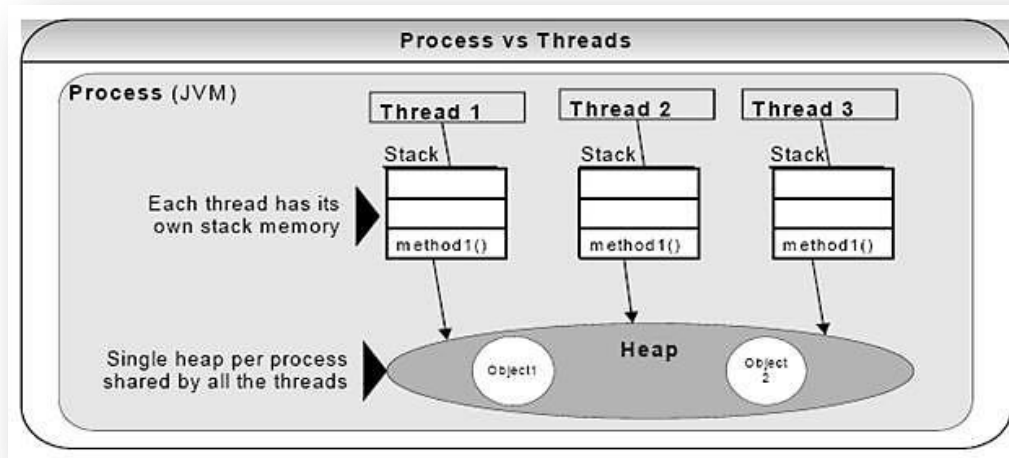


1st level scheduling: user threads compete for access to a kernel thread

2nd level scheduling: system threads compete for access to CPU

1.6 Processes vs. Threads

- Different **processes** use **different memory** addresses.
- **Threads** of the same process **share memory** addresses.
- A process allocates a **shared memory** space (heap space) for the **shared variables** of all its threads. Although each thread has its own **stack (local variables from methods)**.

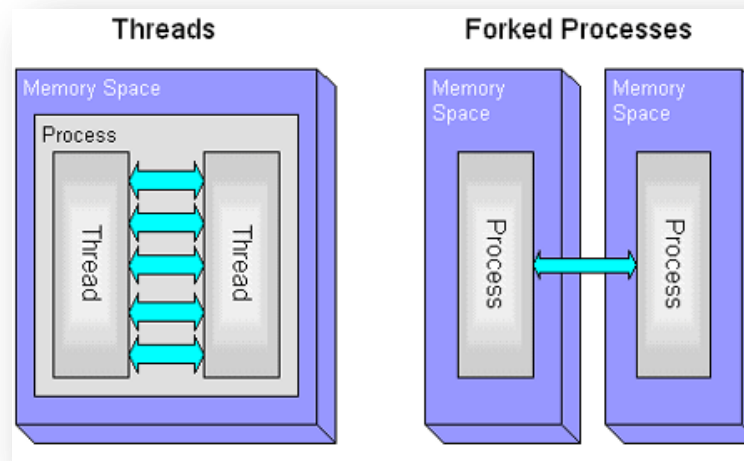


<http://www.java-forums.org/>

- Threads can **control** other threads (kill, create,...). Processes can only manage their children.
- Threads are also called *light weight processes*.

1.6 Processes vs. Threads

- Threads can communicate with each other directly (signals), while processes need calls to the operating systems, pipes, ...
- Children and parent threads share heap space.



<http://www.perl.com/>

- There are two **levels of threads-programming**:
 - System (kernel) threads
 - User threads

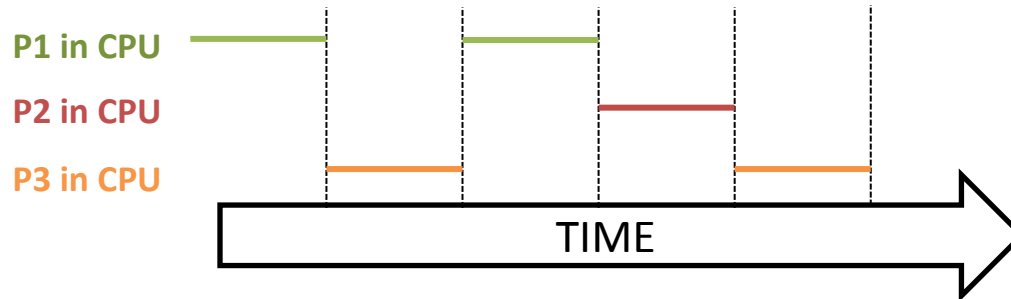
1.6 Processes vs. Threads

- **User threads:** threads created inside the user space of the OS. These are created from our high-level programming language and are used to create concurrent programs.
- **System threads:** threads provided by the operating system to give support to user threads. There exist 3 standards of system threads:
 - Win32 (proprietary), implemented in the OS kernel
 - OS/2 Win32 (proprietary), implemented in the OS kernel
 - POSIX (UNIX and Linux), implemented in the user space of the OS
- The way the programming language uses the native system threads is transparent for the developer.

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads**
- 1.7 Architectures providing concurrency**
- 1.8 Java Threads
- 1.9 Pascal FC

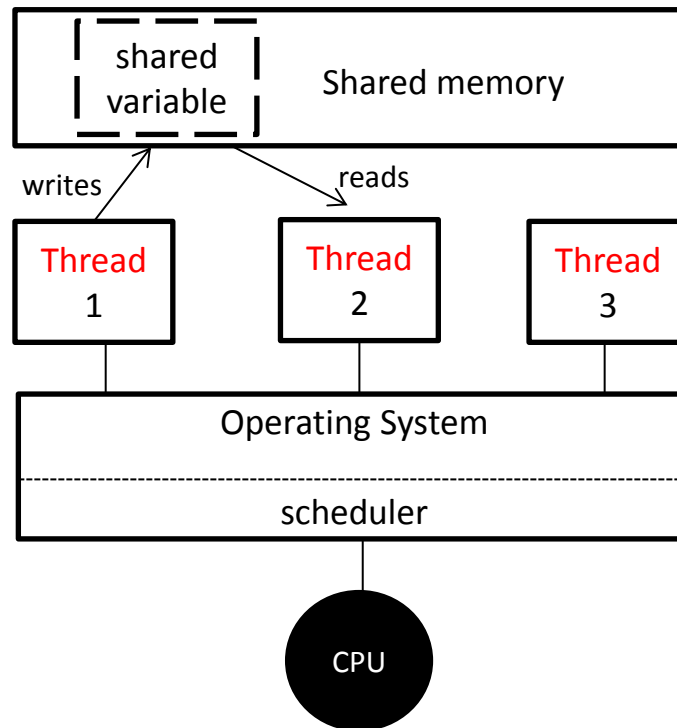
1.7 Architectures providing concurrency

- There are three kinds of hardware architecture which provide concurrency:
 - Uniprocessor: 1 computer with 1 processor
 - Multiprocessor: 1 computer with more than 1 processor
 - Distributed Systems: several computers (uni or multiprocessor) in a network.
- Uniprocessor:
 - Processes share the processor by interleaving.



1.7 Architectures providing concurrency

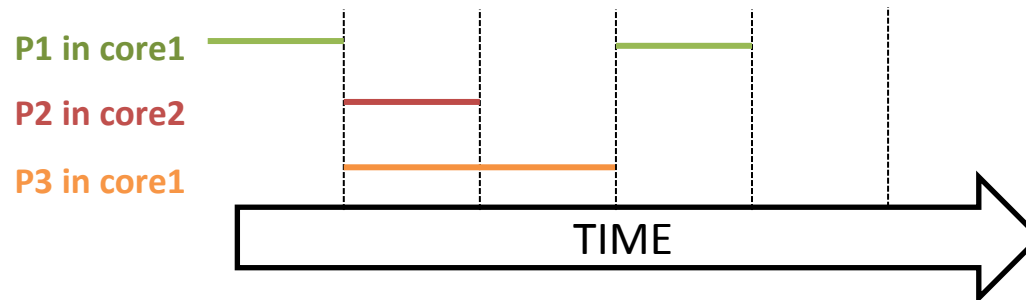
- Interleaving is controlled by the scheduler of the operating system
- **Threads** share the same memory: communication and synchronization is performed by shared variables.



Concurrency in uniprocessor architecture

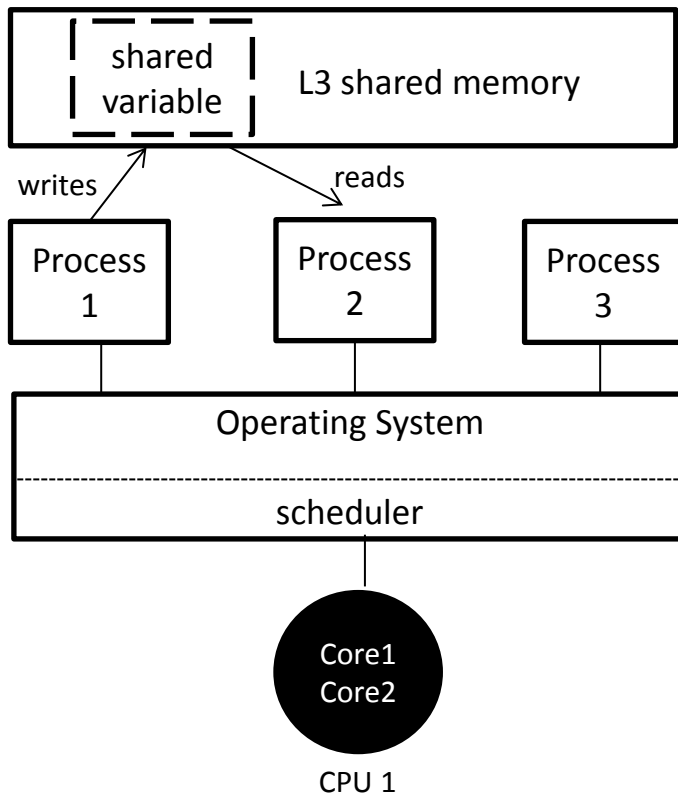
1.7 Architectures providing concurrency

- Multiprocessor and Multicore:
 - Now one processor has several cores, each capable of running parallel instructions. They are integrated in one chip: multicore
 - There might be several multicore chips: multicore multiprocessor (cluster)
 - Real parallelism happens, but interleaving is still necessary (commonly, the number of processes is higher than the number of cores)

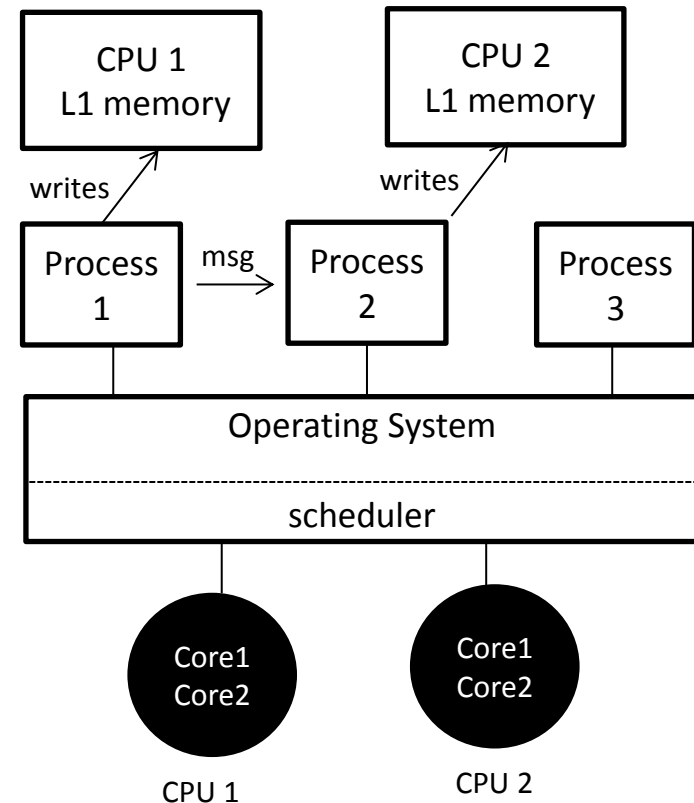


1.7 Architectures providing concurrency

- Cores in one processor may share the same memory or have different levels each one.
- Communication and synchronization may be performed by shared memory or message passing, depending on the architecture.



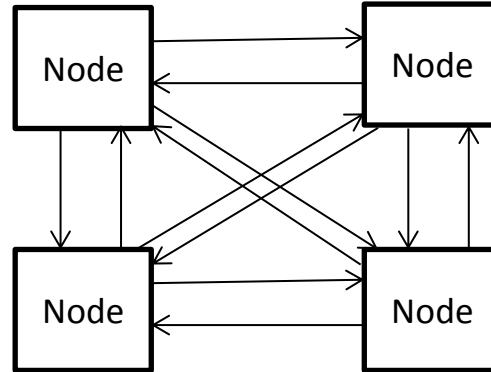
Concurrency in multicore architecture



Concurrency in multicore multiprocessor

1.7 Architectures providing concurrency

- Distributed systems:
 - Nodes (processors) are connected to each other through a network
 - Communication and synchronization by message passing.
 - Each node may contain processors with different architecture or systems.
 - Parallelism and concurrency occur.



Message passing in a distributed system

1.7 Architectures providing concurrency

- Depending on the architecture, we define 3 **kinds of scheduling**:
 - **Multiprocessing**: several cores or processors are available, and shared memory is used. It happens in multicore and multiprocessor architecture.
 - **Distributed processing**: several cores or processors are available. It happens in distributed systems.
 - **Multiprogramming**: only one processing unit is available. Shared memory used, and it happens in uniprocessor architecture. Parallelism is not possible.

1.7 Architectures providing concurrency

- In order to avoid correctness problems, and to help us to get rid of low level details, we should implement concurrent programs from the **concurrent programming abstraction**:

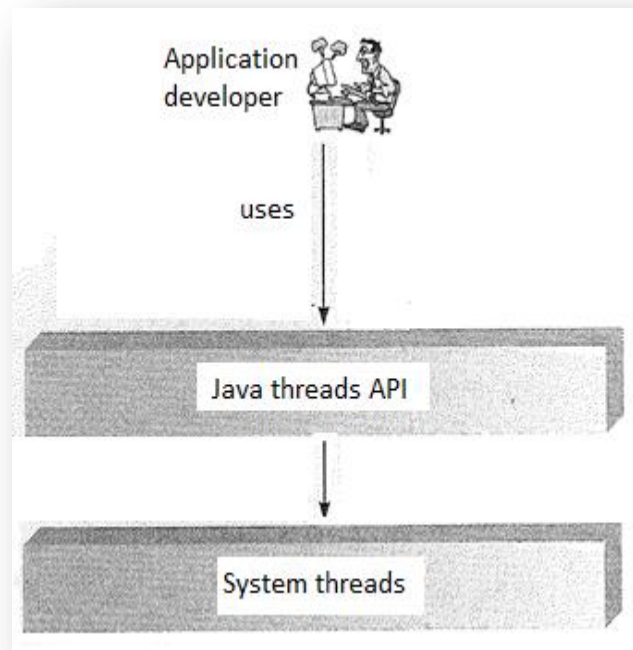
The execution of a concurrent program proceeds by executing a sequence of atomic statements obtained by random interleaving of the atomic statements of each process.

- So we should think or assume that:
 - The final execution is a **single sequential program**, which is made of atomic statements of all processes, randomly interleaved.
 - Since there exist only 1 CPU, the **clock frequency is not important**.
 - And, most importantly, INTERLEAVING MAY HAPPEN AT ANY TIME. THUS, USE **CONCURRENT PROGRAMMING TOOLS** TO PROTECT CRITICAL SECTIONS AND CORRECTLY PERFORM SYNCHRONIZATION.

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads**
- 1.7 Architectures providing concurrency**
- 1.8 Java Threads**
- 1.9 Pascal FC**

1.8 Java Threads

- Java threads are implemented on the Java Virtual Machine (JVM), which is built on the corresponding operating system threads.



Palma et al. Ch. 2. 2003

- JAVA makes concurrent programming possible without taking into account the underlying system threads library.

1.8 Java Threads

- When you run the main method in a Java program, you create one thread (**Main Thread**).
- From the main thread, you can create **new threads**. And from each new thread, you can create new threads as well.
- When the only existing thread is the Main Thread, you can be sure of the order of execution: **sequential**
- Once you create a second thread and start it, you can never know the order of execution of time-slices for the coexisting threads: you may find **indeterminism and non thread-safe** situations if you do not control access to critical sections and synchronization.

1.8 Java Threads

- In Java, a thread is represented by a `java.lang.Thread` object. The **two ways to create a thread** are:
 1. Instantiate a class which extends `java.lang.Thread` and overrides method `run()`
 2. Instantiate `Thread` passing as argument a class implementing method `run()` of interface `java.lang.Runnable`

```
1 package lectures.unit1;
2
3 public class HelloThread extends Thread {
4
5     public void run() {
6         System.out.println("Hello from a thread!");
7     }
8
9     public static void main(String args[]) {
10         (new HelloThread()).start();
11     }
12
13 }
```

```
1 package lectures.unit1;
2
3 public class HelloRunnable implements Runnable {
4
5     public void run() {
6         System.out.println("Hello from a thread!");
7     }
8
9     public static void main(String args[]) {
10         (new Thread(new HelloRunnable())).start();
11     }
12
13 }
14 }
```

- Extending `Thread` is more simple and intuitive, but your new class cannot extend anymore classes (Java does not allow multiple inheritance)
- The second method is more complex but your thread can extend another class.

1.8 Java Threads

- The official number of states for a Java thread is 6, from version 1.5:

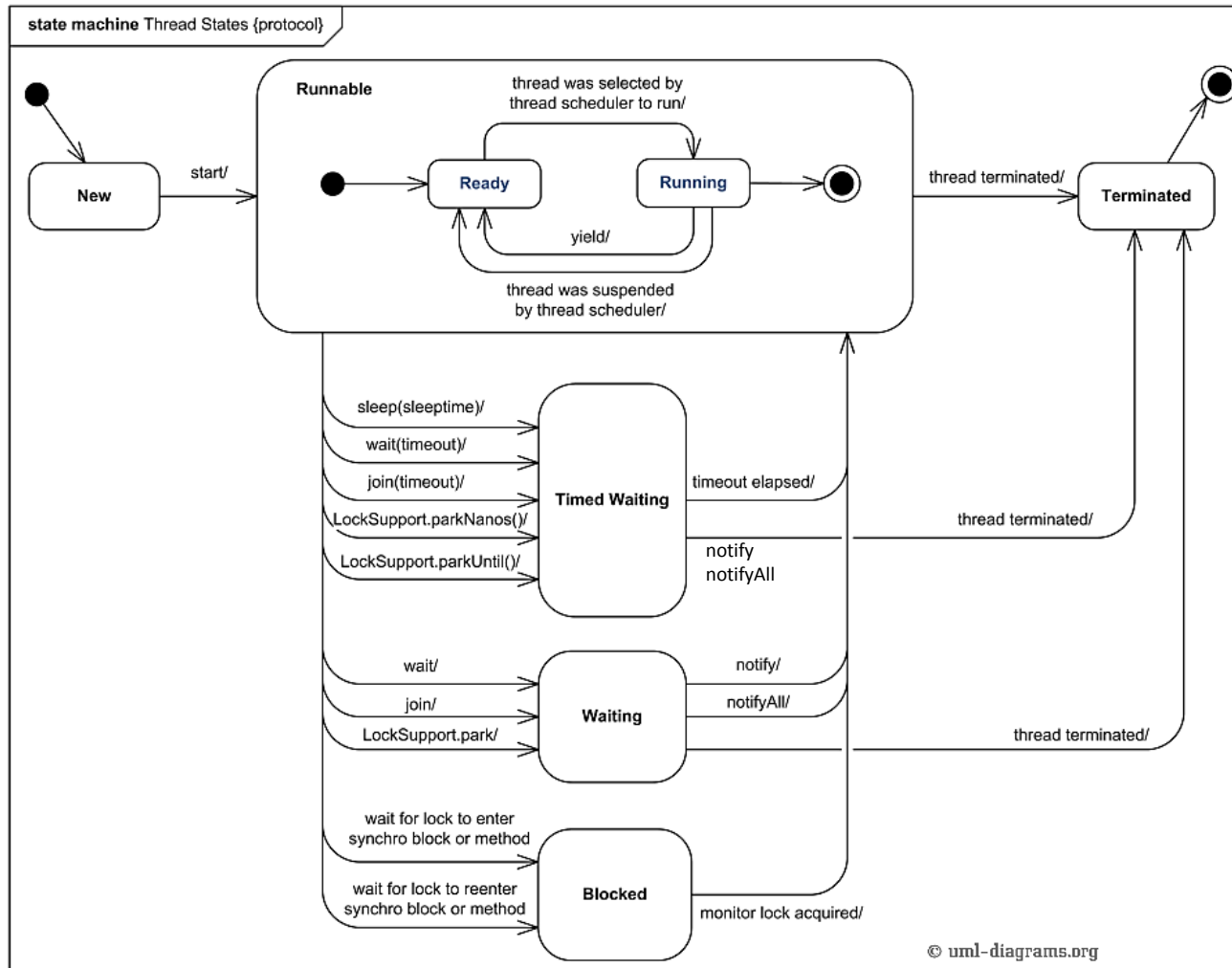
Enum Constant and Description	
BLOCKED	Thread state for a thread blocked waiting for a monitor lock.
NEW	Thread state for a thread which has not yet started.
RUNNABLE	Thread state for a runnable thread.
TERMINATED	Thread state for a terminated thread.
TIMED_WAITING	Thread state for a waiting thread with a specified waiting time.
WAITING	Thread state for a waiting thread.

<http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html>

- These are the states inside the JVM. That is, we cannot ask if the thread is Running or Sleeping using method `Thread.getState()`.

1.8 Java Threads

- The Java thread status does not **lets us always know** how the thread reached it.



In Runnable, we do not know when it is actually running. We do not really need it, when it Runs your code it means it is running!

A thread can be killed while waiting

A thread *waiting* is waiting for another thread to send a signal

A thread is blocked when waiting for another thread to release a lock

1.8 Java Threads

- We can **change the state** of a thread by calling **methods** provided by Thread and Object.
 - To wait state:
 - *sleep(milliseconds)*: current thread waits the given period of time
 - *join()*: current thread waits until the thread on which this method is called is terminated
 - *wait()*: current thread waits until it receives a signal from other thread.
 - To running state:
 - *yield()*: current thread tells the scheduler that it wants to release the processor
 - Indicate that it should go to terminate state:
 - *interrupt()*: current thread sets the *interrupted* flag of the thread on which this method is called. The interrupted thread should check this flag and finish when it corresponds.
- Careful, do not use deprecated methods!: *stop()*, *destroy()*, *suspend()*, *resume()*

1.8 Java Threads

- In order to call a method on a thread different to the current thread, we need to keep references when they are instantiated. Think what happens in the following examples:

```
class extendedThread extends Thread{

    @Override
    public void run() {
        for(int i=0;i<1000;i++){
            System.out.println("hi there from "+Thread.currentThread());
        }
    }
}

public class Calling {

    public static void main(String args[]){

        extendedThread t1=new extendedThread();
        extendedThread t2=new extendedThread();

        t1.start();
        t2.start();

    }
}
```

1.8 Java Threads

```
public static void main(String args[]) throws Exception{

    extendedThread t1=new extendedThread();
    extendedThread t2=new extendedThread();

    t1.start();
    t1.join();
    t2.start();

}
}
```

```
class extendedThread extends Thread{

    @Override
    public void run(){
        for(int i=0;i<1000 && !Thread.currentThread().isInterrupted();i++){
            System.out.println("hi there from "+Thread.currentThread());
        }
    }
}

public class Calling {

    public static void main(String args[]) throws Exception{

        extendedThread t1=new extendedThread();
        extendedThread t2=new extendedThread();

        t1.start();
        t2.start();
        t1.interrupt();

    }
}
```

1.8 Java Threads

```
class extendedThread extends Thread{

    @Override
    public void run(){
        for(int i=0;i<1000;i++){
            System.out.println("hi there from "+Thread.currentThread());
            if(Thread.currentThread().getName().equals("T1")) yield();
        }
    }
}

public class Calling {

    public static void main(String args[]) throws Exception{

        extendedThread t1=new extendedThread();
        extendedThread t2=new extendedThread();

        Thread.sleep(3000);
        t1.setName("T1");
        t2.setName("T2");
        t1.start();
        t2.start();
    }
}
```

1.8 Java Threads

- When you print a thread using *Thread.currentThread()* method, 3 values are printed: [name of thread, priority, threads group]
- The default priority is 5, which can be changed calling method *setPriority(int n)* being n from 1 to 10 (min to max priority).
- In theory,
 - a thread with higher priority will gain access to CPU before a lower priority thread. However,
 - A ready thread will not switch context with a running higher-priority threadhowever, Java does not guarantee this is true at any moment.

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads**
- 1.7 Architectures providing concurrency**
- 1.8 Java Threads**
- 1.9 Pascal FC**

1.9 Pascal FC

- In this course, we will learn concurrent programming using Pascal-FC and Java.
- Pascal FC is based on Pascal, which is enhanced and reduced to support concurrent programming and to be used in educational contexts.
- Developed by Alan Burns and Geoff Davies, at the University of York.
- The official webpage maintained by the authors is
<http://www-users.cs.york.ac.uk/~burns/pf.html>



Pascal-FC

by Alan Burns and Geoff Davies

1.9 Pascal FC

- Pascal-FC was developed to provide the most common tools to achieve correctness in our concurrent programming language, whose primitive commands or objects may not contain the desired tools.
- Program structure:

```
program name;  
(* global declarations:*)  
(* variables, processes, monitors,... *)  
  
begin  
  (* statements *)  
end
```

1.9 Pascal FC

- Declaration and use of 3 processes:

```
program threeprocesses;
  process type MYPROCESS(I : integer);
  begin
    writeln(I);
  end;
var
  P1, P2, P3: MYPROCESS;
begin
  (*... statements executed sequentially*)
  cobegin
    P1(1);
    P2(2);
    P3(3);
  coend
  (*... statements executed sequentially*)
end.
```

Processes P1, P2 and P3:

- are type MYPROCESS
- are run concurrently (we do not know the order)
- cannot start until the sequential statements prior to **cobegin** are finished.
- must finish before the sequential statements after **coend** start.

1.9 Pascal FC

- Program which defines two processes. Each one prints its id 5 times.

```
program printID;
process First;
var
    i: integer;
begin
    for i:=1 to 5 do
        writeln(1);
end;
process Second;
var
    i: integer;
begin
    for i:=1 to 5 do
        writeln(2);
end;
begin
    writeln('This is executed sequentially');
    writeln('and the following cobegin/coend');
    writeln('block concurrently');
    cobegin
        First;
        Second;
    coend;
    writeln('When the 2 processes end,');
    writeln(', this is run sequentially');
end.
```

Instead of defining the process type, since we only want 1 occurrence of each type, processes are defined directly.

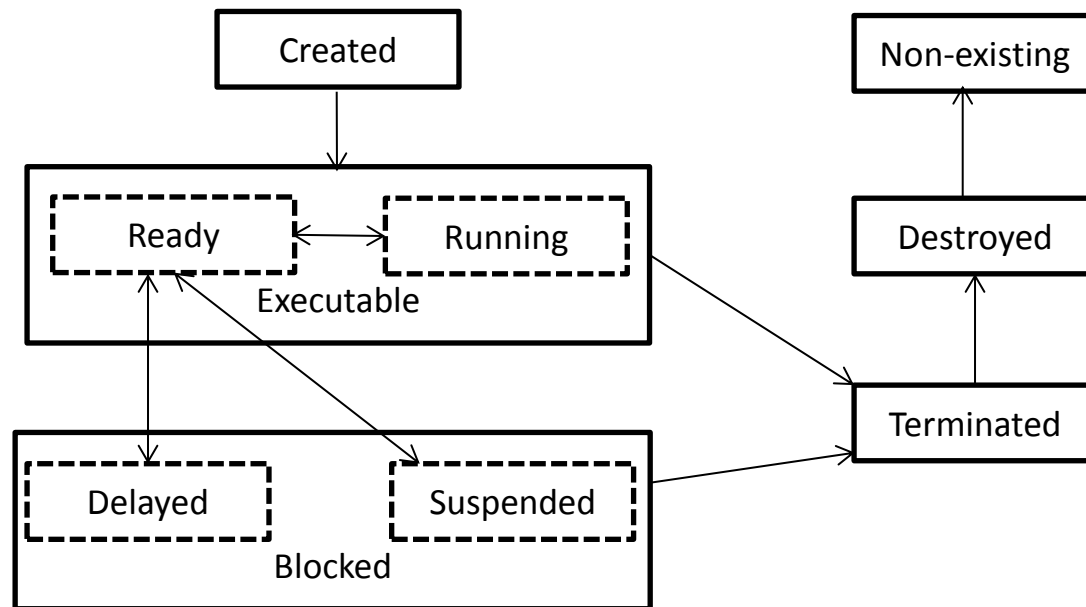
Burns et al. Ch2. 1993.

1.9 Pascal FC

- Modify the previous program so that we only need to define 1 kind of process: declare the type of process, instantiate as many as necessary, use parameters.

1.9 Pascal FC

- States diagram of a process in Pascal-FC



A process is delayed by ***sleep()***. It returns to Ready state after a given time.

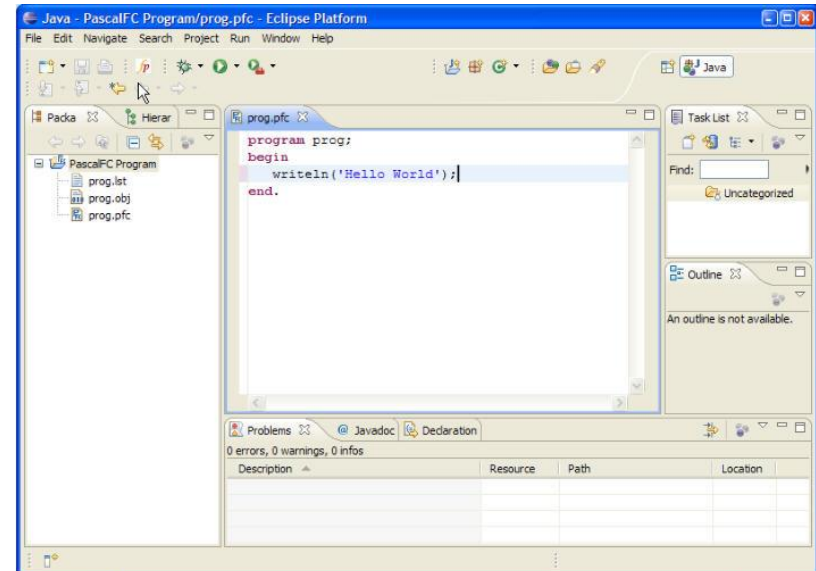
A process is suspended by calling a primitive which blocks it: read channel, request semaphore... **A suspended process can only go back to Ready state by the action of another process.** A suspended process is Terminated because it is selected as alternative (advanced topic).

1.9 Pascal FC

- Pascal-FC is designed to be run in OS without support to concurrency. In order to achieve this, it compiles all processes in **one single sequential program**.
- This means that if the code of one process halts, all the others halt. Do not misunderstand with state Blocked. By 'halt' we mean the process cannot go on, for example due to a deadlock problem or waiting for I/O which never happens.
- We can choose **2 kinds of execution**:
 - **Unfair** (without time-slices): one process cannot start until other is Terminated
 - **Fair** (time-slices): pieces of code are interleaved in the compiled *sequential* program.

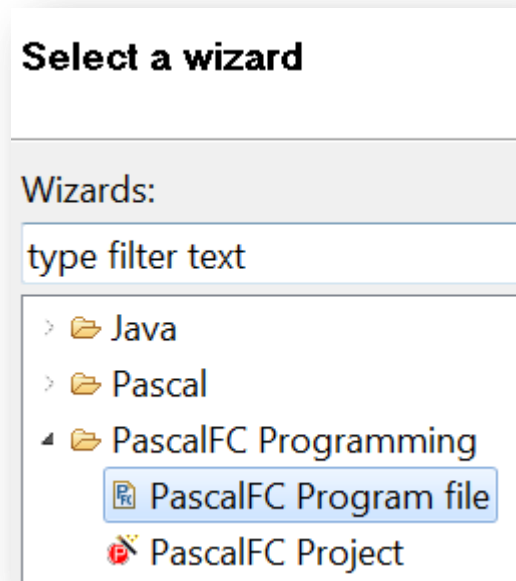
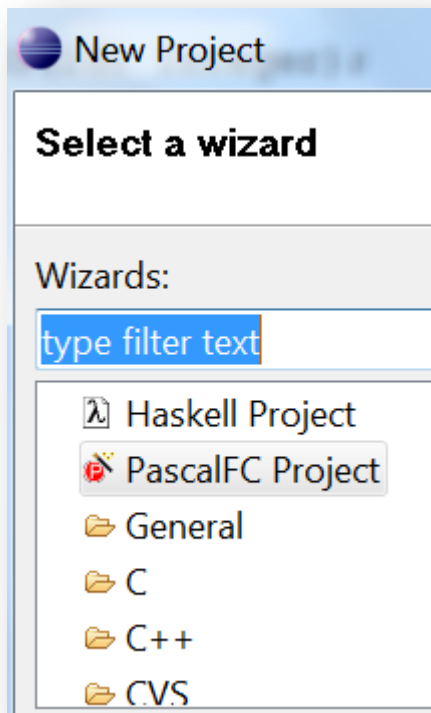
1.9 Pascal FC

- The official compiler and interpreter can be downloaded at <http://www-users.cs.york.ac.uk/~burns/pf.html>, but they work by command line.
- Compile and run Pascal FC programs using the Eclipse Gavab version. This is not available on its official webpage: <http://www.gavab.es/eclipsegavab>
- But you can browse the Internet to find it, for example at: <http://eclipsegavab.software.informer.com/2.0/>



1.9 Pascal FC

- Eclipse Gavab: new project of type PascalFC, then new File→Other→PascalFC Program file
- You can create folders in a project to sort your programs



Potential Midterm Exam Questions

1. What is the operating system scheduler? Where is it running?
2. What is the difference between parallel and concurrent execution? Do you need to previously know the kind of execution when doing concurrent programming?
3. What do we mean when we say that concurrency implies indeterminism?
4. What is a critical section?

Potential Midterm Exam Questions

5. Identify the critical section in this code:

```
class Counter {
    int value;

    Counter(int v) {
        value = v;
    }

    public void increment() {
        value++;
    }
}

class LoopingThread extends Thread {
    Counter counter;

    LoopingThread(Counter c) {
        counter = c;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (int i = 0; i < 5; i++) {
            counter.increment();
        }
    }
}
```

```
public class Question5 {

    public static void main(String args[]) {
        Counter c = new Counter(0);
        LoopingThread t1 = new LoopingThread(c);
        LoopingThread t2 = new LoopingThread(c);
        t1.start();
        t2.start();
    }
}
```

Potential Midterm Exam Questions

- 5. Identify the critical section in this code:

```
class LoopingThread extends Thread {
    int value;










    LoopingThread(int x) {
        value=x;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (int i = 0; i < 5; i++) {
            value++;
        }
    }
}

public class Question6 {

    public static void main(String args[]) {
        int v=0;
        LoopingThread t1 = new LoopingThread(v);
        LoopingThread t2 = new LoopingThread(v);
        t1.start();
        t2.start();
    }
}
```

Keywords phonetics

- synchronization /ˌsɪŋkrənaɪˈzeɪʃən/ 
- starvation /stɑːˈveɪʃən/ 
- initiate /ɪˈnɪʃieɪt/ 
- variable /ˈveəriəbl/ 
- instantiate /ɪnˈstæŋʃieɪt/ 
- inheritance /ɪnˈherɪtəns/ 
- yield /jiːld/ 
- architecture /ˈɑːkɪtektʃər/ 
- concurrent /kənˈkʌrənt/ 
- precedence /ˈpresədəns/ 