Concurrent Programming

Real-Time Systems, Lecture 2

Martina Maggio

19 January 2016

Lund University, Department of Automatic Control

Content

[Real-Time Control System: Chapter 3]

1. Implementation Alternatives

2. Concurrent Programming

2.1 The need for synchronization

2.2 Processes and threads

2.3 Implementation

Implementation Alternatives

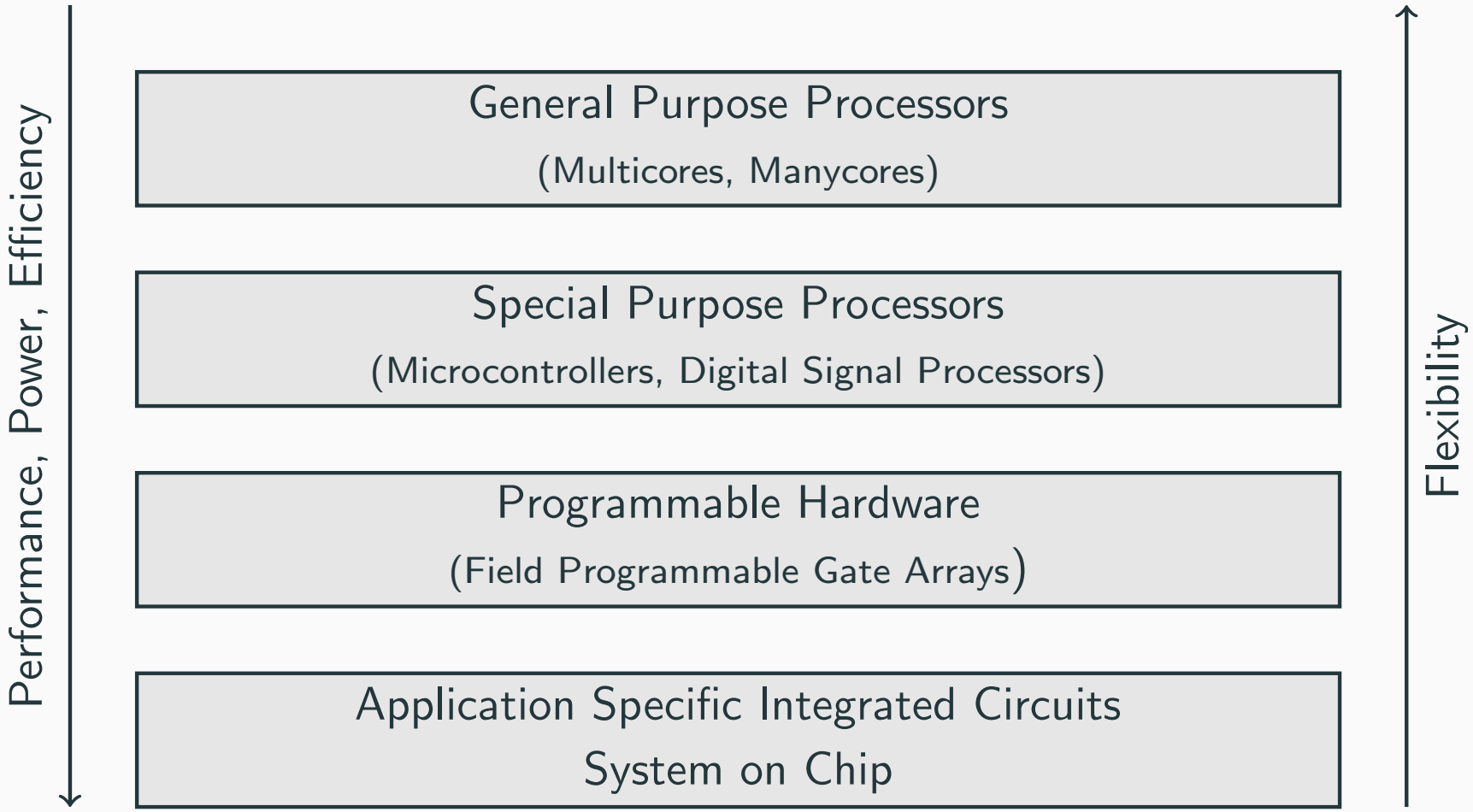
Historical Implementation Alternatives

Historically, controllers have been implemented in many ways:

- mechanical techniques;
- discrete analog electronics;
- discrete digital electronics.

These techniques are no longer used in industrial practice.

Current Implementation Alternatives



Microcontroller

- A small and cheap "computer on a chip".
  - In 2015, 8-bit microcontrollers cost \$0.311 (1,000 units), 16-bit \$0.385 (1,000 units), and 32-bit \$0.378 (1,000 units but at \$0.35 for 5,000)[1].
  - A typical home in a developed country is likely to have 4 general-purpose microprocessors and more than 35 microcontrollers.
  - A typical mid-range automobile has at least 30 microcontrollers.
- **Contains:** a processor core, memory and input/output peripherals.

[1] Numbers from https://en.wikipedia.org/wiki/Microcontroller and http://www.icinsights.com/news/bulletins/MCU-Market-On-Migration-Path-To-32bit-And-ARMbased-Devices/.

Small Microcontrollers

- Limited program memory.
- Limited data memory.
- Optimized interrupt latency:
  - main program and interrupt handlers;
  - (periodic) timers;
  - periodic controllers implemented in (timers) interrupt handlers;
  - limited amount of timers.

Large Microcontrollers

- Large amount of memory on chip.
- External memory available.
- Real-time kernel supporting multiple threads (and concurrent programming).

Multicores

- Single computing entity with two or more processor cores[2].
- Shared memory, tasks running on each of the cores can access data. Shared cashes, partitioned cashes or separate ones.
- **Concurrency** should be taken into account.

[2] Multicores typically have from 2 to 16 cores, many-cores exceed 16 cores, see http://www.argondesign.com/news/2012/sep/11/multicore-many-core.

Concurrent Programming

Concurrency

Concurrency is not parallelism[3]:

when people hear the word concurrency they often think of parallelism, a related but quite distinct concept. In programming, concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations.

[3] http://blog.golang.org/concurrency-is-not-parallelism, http://vimeo.com/49718712

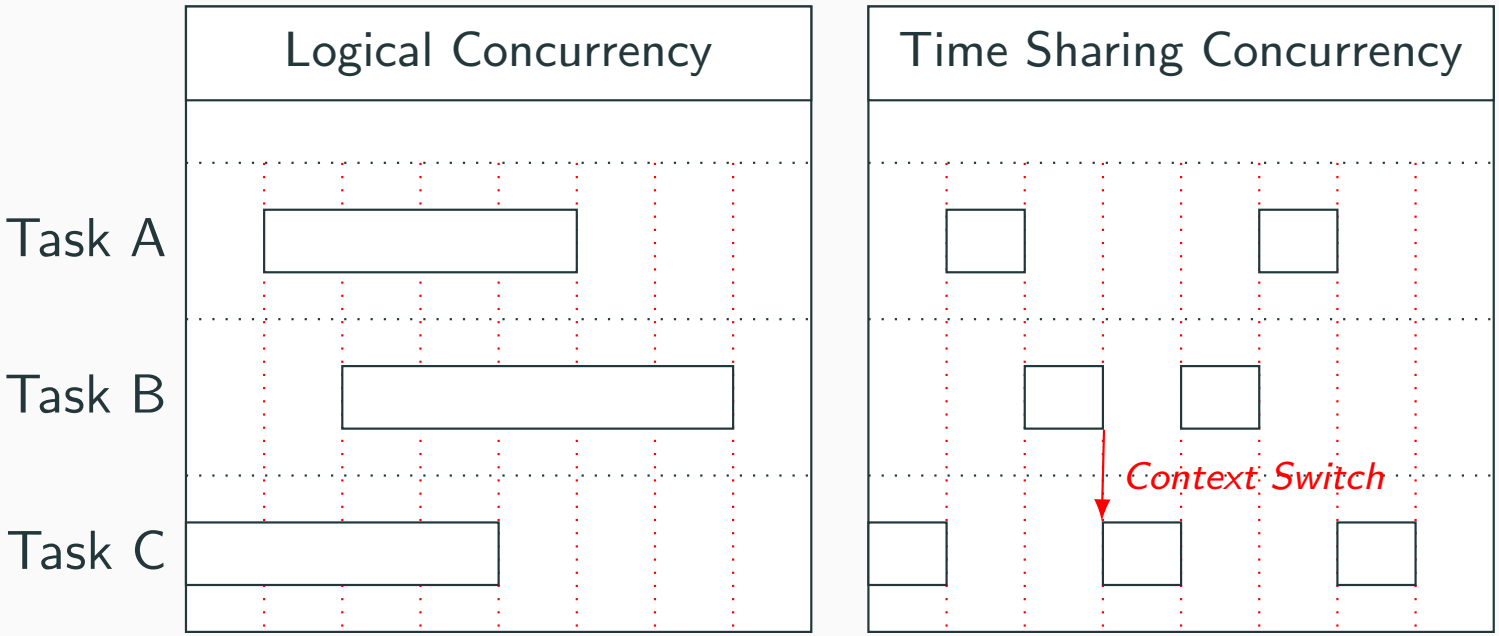
Concurrent Programming

Concurrent execution can take three different forms:

- (1) **Multiprogramming**: the processes multiplex their execution on a single processor
- (2) **Multiprocessing**: the processes multiplex their execution on tightly coupled processors (processors sharing memory)
- (3) **Distributed processing**: the processes multiplex their execution on loosely coupled processors (not sharing memory)

Concurrency, or *logical parallelism*: (1); Parallelism: (2) and (3)

Concurrent Programming



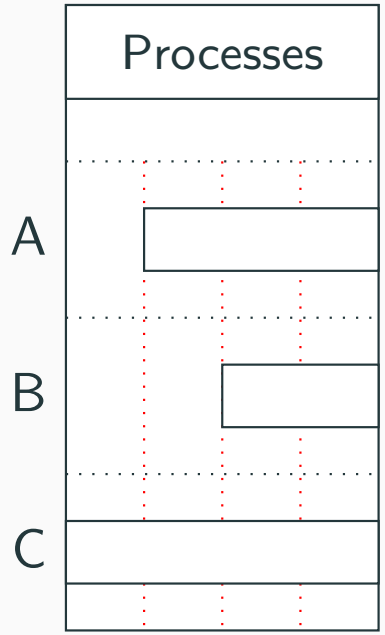
Concurrent Programming – Approaches

- Real-Time Operating Systems (RTOS)
  - Sequential Languages (like C) with real-time primitives
  - Real-Time kernel for process handling
- Real-Time Programming Languages (like Ada)
  - The language itself or its runtime kernel provides the functionality of a real-time kernel

Real-Time Programming

Real-Time Programming requires:

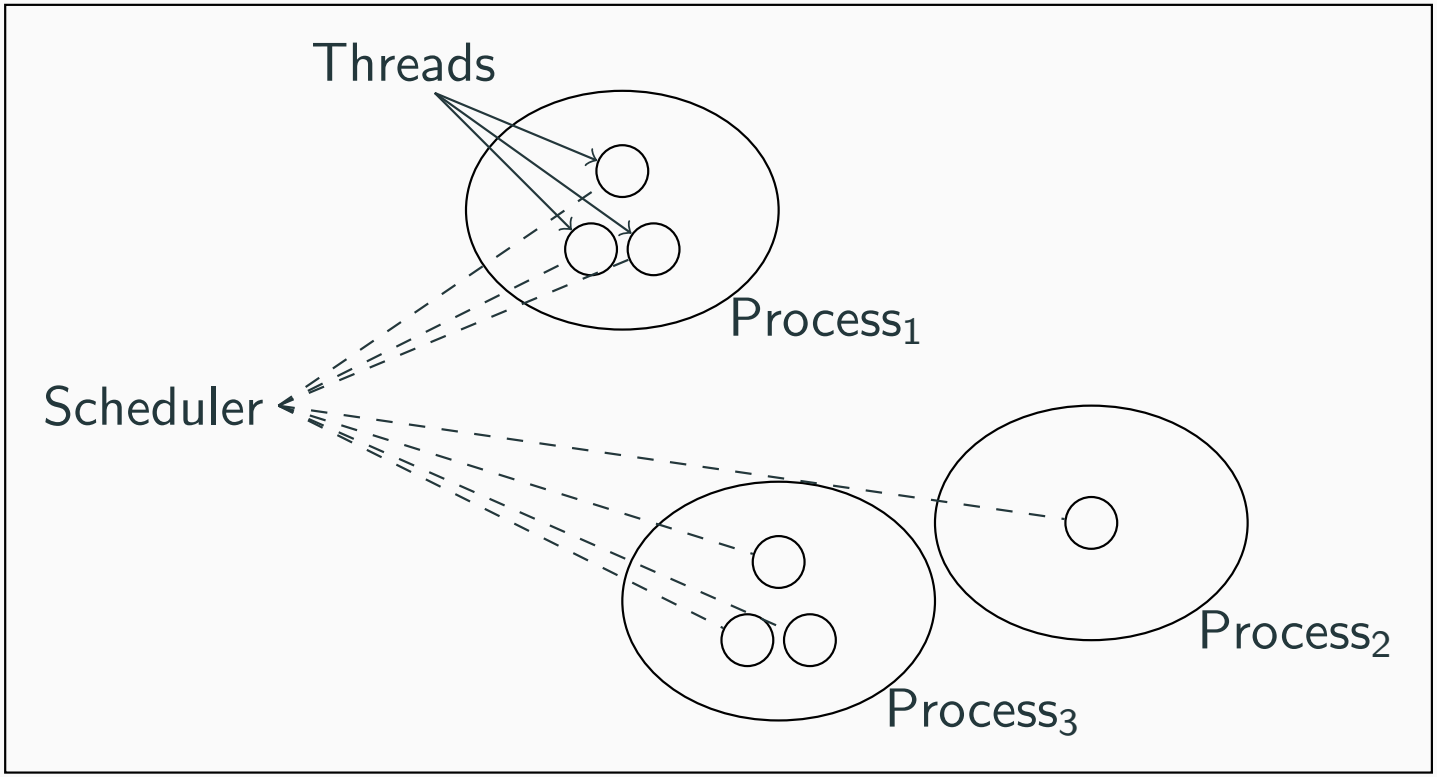
- Notion of **processes**
- Process **synchronization**
- Inter-process **communication** (IPC)



Processes – Memory Models

- **Shared address space**:
  it means that all the processes can access variables that are global in the address space (requires some care),
  lightweight processes or **threads**:
  less status information, less switching overhead
- **Separate address space**:
  it needs explicit ways to handle data that should be shared among the running processes (for example message passing)

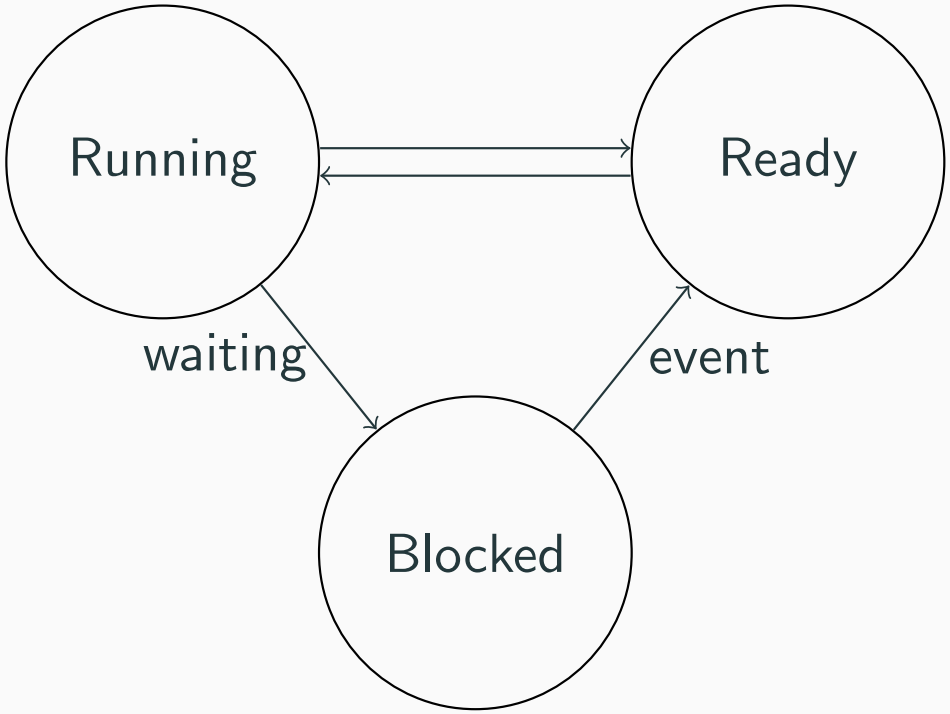
Processes vs Threads



Windows, Solaris, ...

Some opearting systems like Windows provide support for threads within processes. Linux does not differentiate between processes and threads.

Process States



- **Ready** — The process is ready to execute.
- **Running** — The process is currently executing.
- **Blocked** — The process is waiting for an event.

Context Switches

A **context switch** takes place when the system changes the running process (thread). The context of the process that is currently occupying the processor is **stored** and the context of the new process to run is **restored**.

The context consists in:

- the stack;
- the content of the programmable registers;
- the content of the status registers.

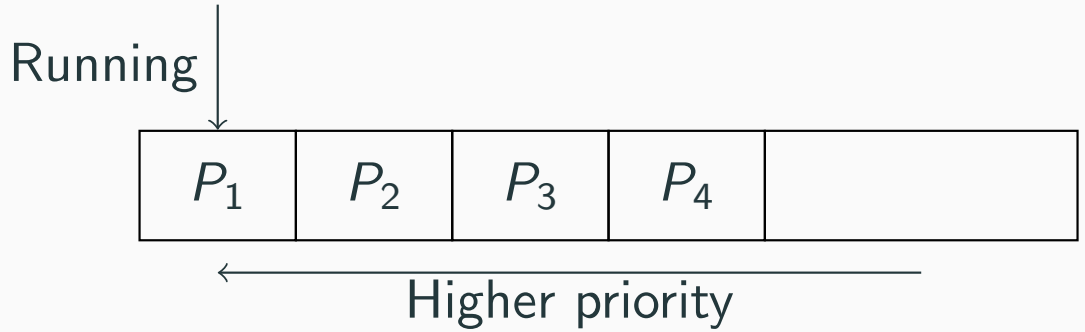
Priority

Each process is assigned a **priority** number that reflects the importance of its execution demands. The interval of possible priorities can vary, and they can be ordered from low numbers to high or from high numbers to low.

- STORK: low priority number = high priority
  (range 1 to max integer)
- Java: high priority number = high priority
  (range 1 to 10)
- Linux: high priority number = high priority
  (range 0, 1–99 and 100 depending on the scheduler)

Priority

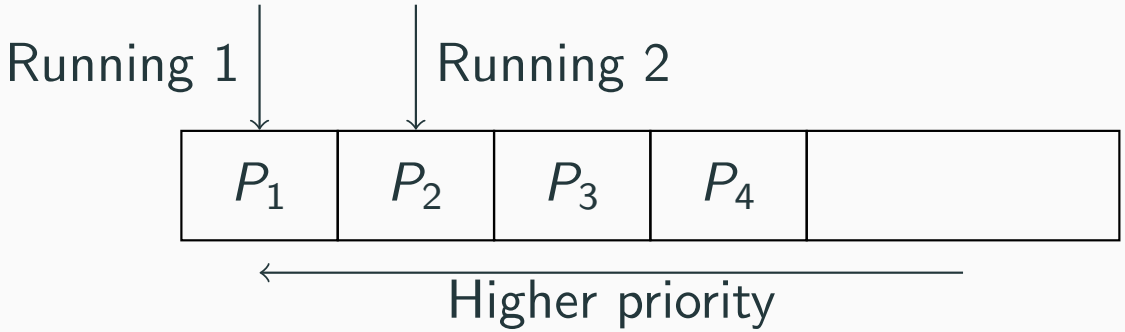
Processes that are ready to execute are stored in the ready queue according to their priorities.



Multicore Case

Two alternatives:

- **global** scheduling: a single ready queue with running processes.



- **partitioned** scheduling:
  - one ordinary ready queue per core;
  - often combined with the possibility to move (migrate) processes.

Preemption

A change in the ready queue may lead to a context switch. If the change (insertion, removal) results in a different process being the first in queue, the context switch takes place. Depending on the scheduler, this can happen in different ways:

- **preemptive** scheduling: the context switch takes place immediately (the running process is preempted), it happens in most real-time operating systems and languages (STORK, Java);
- **non-preemptive** scheduling: the running process continues until it voluntarily releases the processor, then the context switch takes place;
- **preemption-point** based scheduling: the running process continues until it reaches a preemption point, then the context switch takes place, it was the case in early versions of Linux (no preemption in kernel mode).

Assigning Priorities

Assigning priorities is non-trivial, requires global knowledge. Two ways:

- ad hoc rules:
  important time requirements $\Longrightarrow$ high priority
  time consuming computations $\Longrightarrow$ low priority
  conflicts, no guarantees;
- scheduling theory:
  often it is easier to assign deadlines than priorities.

Dynamic Priorities

Priority-based scheduling is **fixed-priority**, the priority of a process is fixed a priori. It may be better to have systems where it is the closeness to a deadline to decide which process should be executed: **earliest deadline first (EDF)** scheduling. The deadline can then be viewed as a dynamic priority that changes as the time proceeds (still unusual in commercial systems).

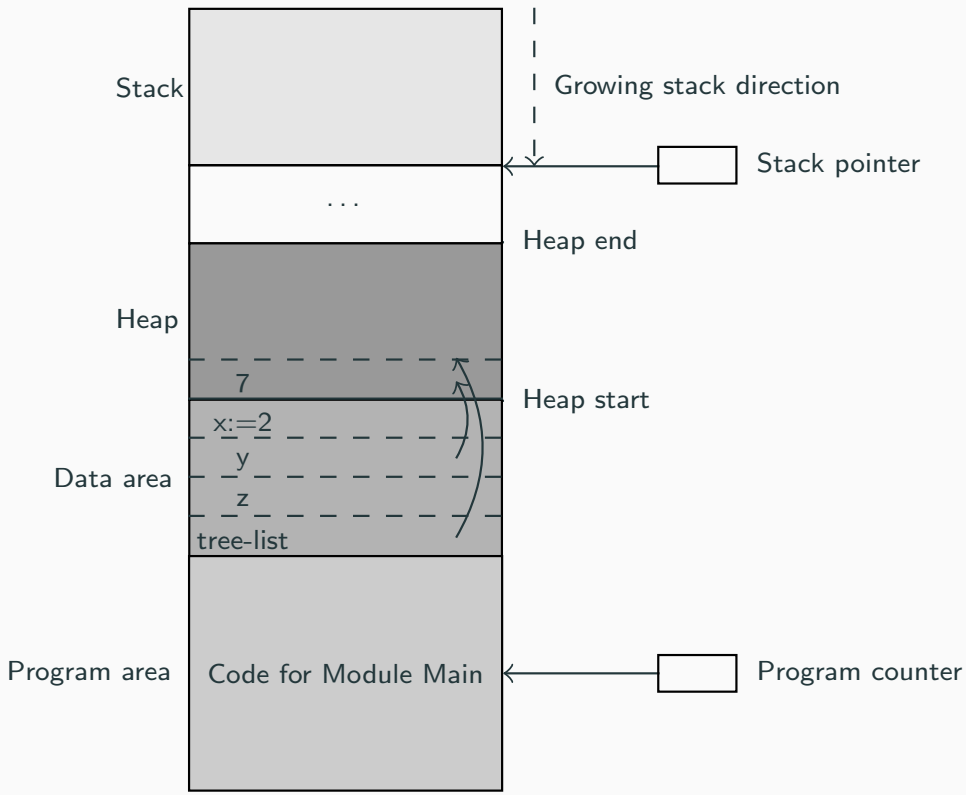
Process Representation

A process consists of:

- the code to be executed:
  in Java this is the `run` method;
- a stack:
  local variables of the process, arguments and local variables of procedures called by the process, (when suspended) storage space for the values of the programmable registers and program counter;
- a process record (process control block, task control block):
  administrative information, priority, a pointer to the code of the process, a pointer to the stack of the process, etc.

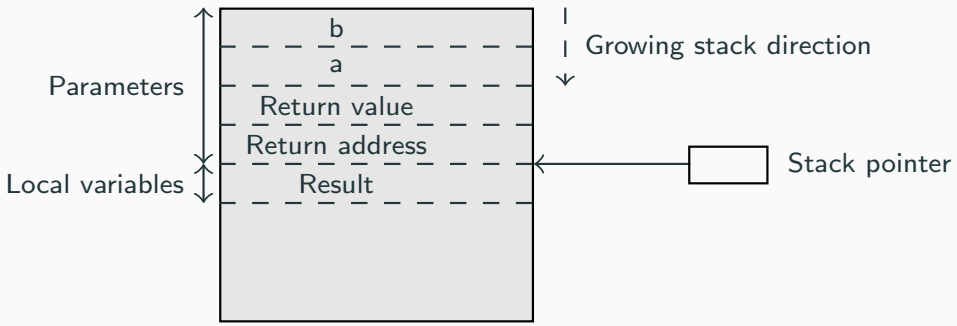
Memory Organization

```
MODULE Main;

VAR x, z: INTEGER;
    y: POINTER TO INTEGER;

PROCEDURE Demo(a: INTEGER, b: INTEGER): INTEGER;

  VAR Result: INTEGER;

  BEGIN
  Result := a*b - b*b;
  RETURN Result;
  END Demo;

BEGIN
x := 2;
NEW(y);
y^ := 7;
z := Demo(x, y^);
Write(z);
DISPOSE(y);
END Main.
```



Demo is called

When the `Demo` procedure is called, a stack frame is created and the parameters and the return value and address are given stack space. When the function executes, it can allocate local variables in the stack. When the procedure is terminated, the stack is cleared, the return values is returned and the program counter is set to the return address.

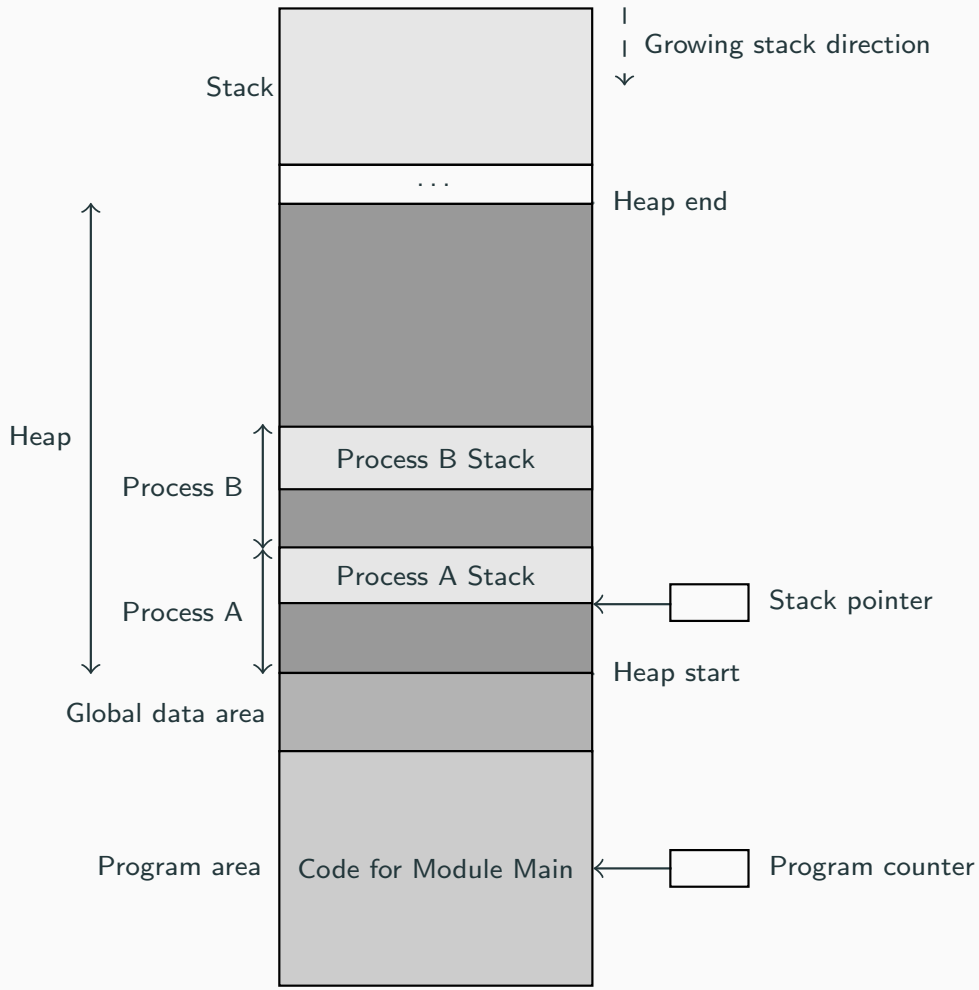


[STORK] Memory Organization: Concurrency

Process A is executing

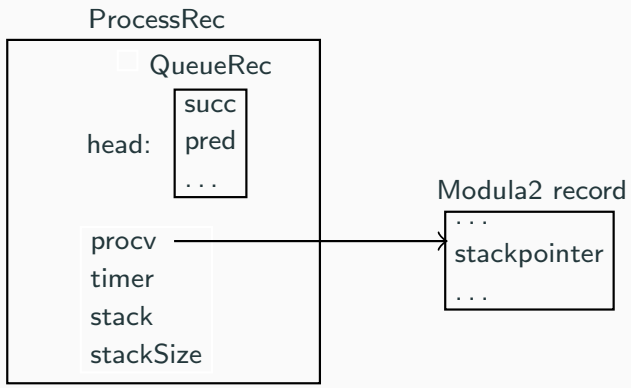
Process B is suspended

```
MODULE Main;

(* Process *) PROCEDURE ProcessA();
(* Local variable declarations *)
BEGIN
  LOOP
  Code for ProcessA;
  END;
END ProcessA;

(* Process *) PROCEDURE ProcessB();
(* Local variable declarations *)
BEGIN
  LOOP
  Code for ProcessB;
  END;
END ProcessB;

(* Global variable declarations *)
BEGIN
(* Code for creating process A and B *)
Wait statement;
END Main.
```



[STORK] Process

```
TYPE
  PROCESS = ADDRESS;
  ProcessRef = POINTER TO ProcessRec;
  Queue = POINTER TO QueueRec;

  QueueRec = RECORD
    succ, pred           : ProcessRef;
    priority             : Cardinal;
    nextTime             : Time;
  END;

  ProcessRec = RECORD
    head                 : QueueRec;
    procv                : PROCESS;
    timer                : CARDINAL;
    stack                : ADDRESS;
    stackSize            : CARDINAL;
  END;
```



In the global data area, there are two pointers: `ReadyQueue` (pointer to a double linked list of processes that are ready to run) and `running` (pointer to the currently running process). For each process:

- pointers part of `head` (`QueueRec`): successor and predecessor;
- `procv` points to the Modula2 process record;
- `timer` is used for round robin slicing.

[STORK] Process Creation

```
Procedure CreateProcess(proced: PROC, memReq: CARDINAL,
                        name: ARRAY OF CHAR);

BEGIN
  CreateProcess(ProcessA, 2000, "Process A");
  CreateProcess(ProcessB, 2000, "Process B");
  ...
END Main.
```

- Creates a process from the procedure `proced`;
- `memReq`: maximum memory bytes required by the process stack;
- `name`: used for debugging purposes.

The call initializes the process record, creates the process calling the Modula-2 primitive `NEWPROCESS`, allocates memory on the heap, set the process priority to 1 (highest priority), and insert the process into the `ReadyQueue`.

[STORK] Context Switching

Initiated by a call to `Schedule`

- directly by the running procedure;
- from within an interrupt handler.

It happens when:

- the running process voluntarily releases the CPU;
- the running process performs an operation that may cause a blocked process to become ready;
- an interrupt has occurred, which may have caused a blocked process to become ready.

[STORK] Schedule

```
PROCEDURE Schedule;

VAR
  oldRunning   : ProcessRef;
  oldDisable   : InterruptMask;

BEGIN
  oldDisable := Disable(); (* Disable interrupts *)
  IF ReadyQueue^.succ <> Running THEN
    oldRunning := Running;
    Running := ReadyQueue^.succ;
    TRANSFER(oldRunning^.procv, Running^.procv);
  END;
  Reenable(oldDisable);
END Schedule;
```

Operations:

- interrupts are disabled;
- the actual context switch takes place in the Modula-2 primitive TRANSFER
- using the Modula-2 process record pointer as argument;
- interrupts are enabled again.

TRANSFER is entered in the context of one process and left in the context of another process.

[STORK] Inside TRANSFER

Operations performed by the `TRANSFER` procedure:

- **Saving**: the state of the processor immediately before the switch is saved on the stack of the suspended process;
- **Switching**: the context switch operations
  - the value of the stack pointer is stored in the process record of the suspended process,
  - the stack pointer is set to the value that is stored in the process secord of the new process;
- **Restoring**: the state of the resumed process is restored (popped from its stack).

Hyperthreading

Many new architectures support hyperthreading:

- the processor registers are duplicated;
- one set of registers is used for the thread currently being executed, the other set (in case of two hyperthreads) is used for the thread that is next to be executed (next-highest priority);
- when context switching between these two, no need for saving and restoring the context;
- the processor only need to change the set of registers that it operates upon, which takes substantially less time.

[JAVA] Java in Real-Time

Three possibilities:

- Ahead-Of-Time (AOT) compilation
  - java or java bytecode to native code
  - java or java bytecode to intermediate language (C)
- Java Virtual Machine (JVM)
  - as a process in an existing OS (**green thread model**: the thread are handled internally by the JVM – vs – **native thread model**: the java threads are mapped onto the threads of the operating system)
  - executing on an empty machine (green thread model)
- direct hardware support (for example throug micro-code)

[JAVA] Execution Models

- Ahead-Of-Time compilation works essentially as STORK.
- JVM with native thread model:
  - each java thread is executed by a native thread,
  - similar to what seen for STORK.

[JAVA] Execution Models

- JVM with green thread model:
  - threads are abstractions inside the JVM,
  - the JVM holds within the thread objects all information related to threads (the thread's stack, the current instruction, bookkeeping information),
  - the JVM performs context switching between threads by saving and restoring the contexts,
  - the JVM program counter points at the current instruction of the running thread,
  - the global program counter points at the current instruction of the JVM.

[JAVA] Thread Creation

A thread can be created in two ways:

- by defining a class that extends (inherits from) the `Thread` class,
- by defining a class that implements the `runnable` interface (defines a `run` method).

The `run` method contains the code that the thread executes.
The thread is started by a call to the `start` method.

[JAVA] Thread Creation extending Thread

```
public class MyThread extends Thread {
  public void run() {
    // Here goes the code to be executed
  }
}
```

Thread Start:

```
MyThread m = new MyThread();
m.start();
```

[JAVA] Thread Creation implementing Runnable

Used when the object needs to extend some other class than `Thread`

```
public class MyClass extends MySuperClass implements Runnable {
  public void run() {
    // Here goes the code to be executed
  }
}
```

Thread Start:

```
MyClass m = new MyClass();
Thread t = new Thread(m);
t.start();
```

Drawback: non-static thread methods are not directly accessible inside the tun method.

[JAVA] Thread Creation: threads as variables

```
public class MyThread extends Thread {
  MyClass owner;
  public MyThread(MyClass m) {
    owner = m;
  }
  public void run() {
    // Here goes the code to be executed
  }
}
public class MyClass extends MySuperClass {
  MyThread t;
  public MyClass() {
    t = new MyThread(this);
  }
  public void start() {
    t.start();
  }
}
```

Makes it possible for an active object to contain multiple threads. The thread has a reference to the owner (not elegant).

[JAVA] Thread Creation: threads as an inner class

```
public class MyClass extends MySuperClass {
  MyThread t;

  class MyThread extends Thread {
    public void run() {
      // Here goes the code to be executed
    }
  }

  public MyClass() {
    t = new MyThread();
  }
  public void start() {
    t.start();
  }
}
```

No owner reference needed. MyThread has direct access to variables and methods of MyClass. The inner class is anonymous.

[JAVA] Thread Priorities

A new thread inherits the priority of the creating thread. The default priority is `NORM_PRIORITY` equal to 5. Thread priorities can be changed calling the nonstatic Thread method `setPriority`.

```
public class MyClass implements Runnable {
  public void run() {
    Thread t = Thread.currentThread();
    t.setPriority(10);
    // Here goes the code to be executed
  }
}
```

The static `currentThread` method is used to get a reference to thread of MyClass.

[JAVA] Thread Termination

A thread terminates when the `run` method terminates. To stop a thread from some other thread it is possible to use a flag.

```
public class MyClass implements Runnable {
  boolean doIt = true;

  public void run() {
    while (doIt) {
      // Here goes the code to be executed periodically
    }
  }
}
```

The thread is stopped by setting the `doIt` flag to false, either directly or providing some access method provided by MyClass. Sleeping threads will not terminate immediately.

[LINUX] Scheduling in Linux

Four scheduling classes (schedulers):

- `SCHED_OTHER` (previously called `SCHED_NORMAL`)
  - default scheduler
  - from 2.6.23 based on the Completely Fair Scheduler (CFS)
  - not a real-time scheduler
  - round robin-based
  - fairness and efficiency are major design goals
  - tradeoff between low lantecy (for example for input/output bound processes) and high throughput (for example for compute bound processes)
  - threads scheduled with this scheduler have priority 0

[LINUX] Scheduling in Linux

Four scheduling classes (schedulers):

- `SCHED_RR`
  - round robin
  - real-time scheduler
  - threads scheduled with this scheduler have priority always higher than threads scheduled with `SCHED_NORMAL`
  - increasing priorities from 1 to 99
  - after a `quantum` the thread releases the CPU to the scheduler and the scheduler assigns it to the highest priority ready thread
- `SCHED_FIFO`
  - works as `SCHED_RR` but the thread does not release the CPU until termination

[LINUX] Scheduling in Linux

Four scheduling classes (schedulers):

- `SCHED_DEADLINE`
  - added in Linux 3.14 (as a result of the EU project ACTORS led by Ericsson in Lund and with the Department of Automatic Control as partner)
  - Earliest-Deadline First scheduler
  - support for CPU reservations
  - threads scheduled with this scheduler have priority 100 (highest) – "if any `SCHED_DEADLINE` thread is runnable, it will preempt any thread scheduled under one of the other policies"[4]

[4] http://man7.org/linux/man-pages/man7/sched.7.html