Lecture 2: Implementation Alternatives & Concurrent Programming

[RTCS Ch. 3]

- Implementation Alternatives
- Processes and threads
- Context switches
- Internal representation

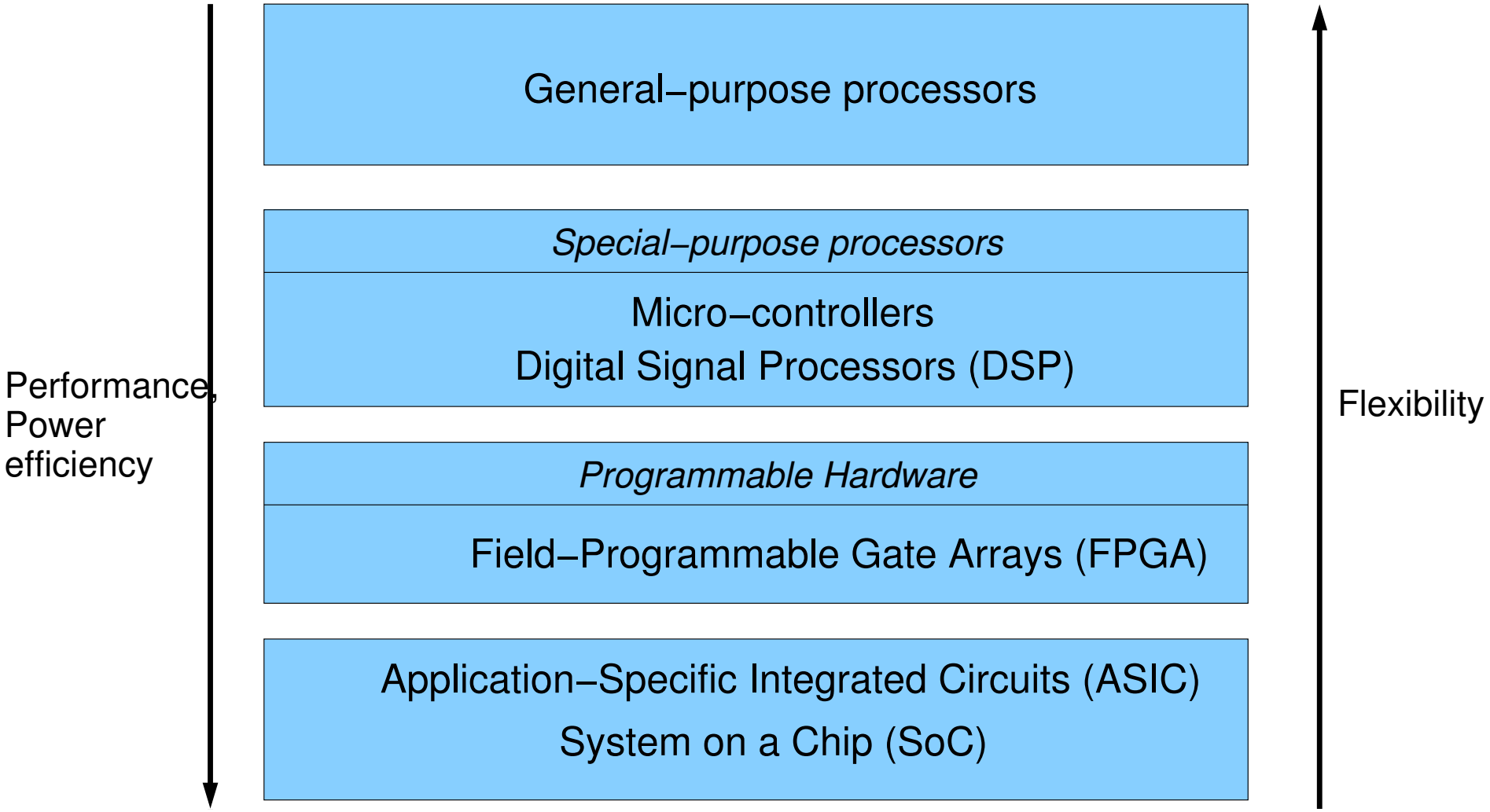
Implementation Alternatives

Controllers can be implemented using a number of techniques

Several of these are rarely used any longer in industrial practice, e.g.

- mechanical techniques
- discrete analog electronics
- discrete digital electronics

Current Implementation Alternatives



Microcontroller (MCU)

- ”Computer-on-a-chip”
- A microprocessor containing all the memory and interfaces required for a simple (control) application
  - Processor (4bit – 64bit)
  - IO interfaces, e.g., UARTs and AD/DA converters
  - Peripherals (e.g., timers, watchdogs)
  - Serial communications interfaces, e.g., I^2C, CAN
  - ROM, EPROM, or EEPROM (Flash) memory for program storage
  - RAM memory for data storage
  - Clock generator

Small Microcontrollers

Limited program memory (e.g. 8-16 kByte)

Limited data memory (e.g. 1-4 kByte)

Processor:

- Main program + interrupt handlers
- (Periodic) timers
- Periodic controllers implemented in interrupt handlers associated with periodic timers
- Only a limited number of timers

Large Microcontrollers

Large amount of memory on chip

External memory

Processor:

- Real-time kernel supporting multiple threads, i.e., concurrent programming

Multicores

Several processors (cores) on the same chip

Multicore – typically 2-16 cores

Manycore – $>$ 16 cores

Shared main memory

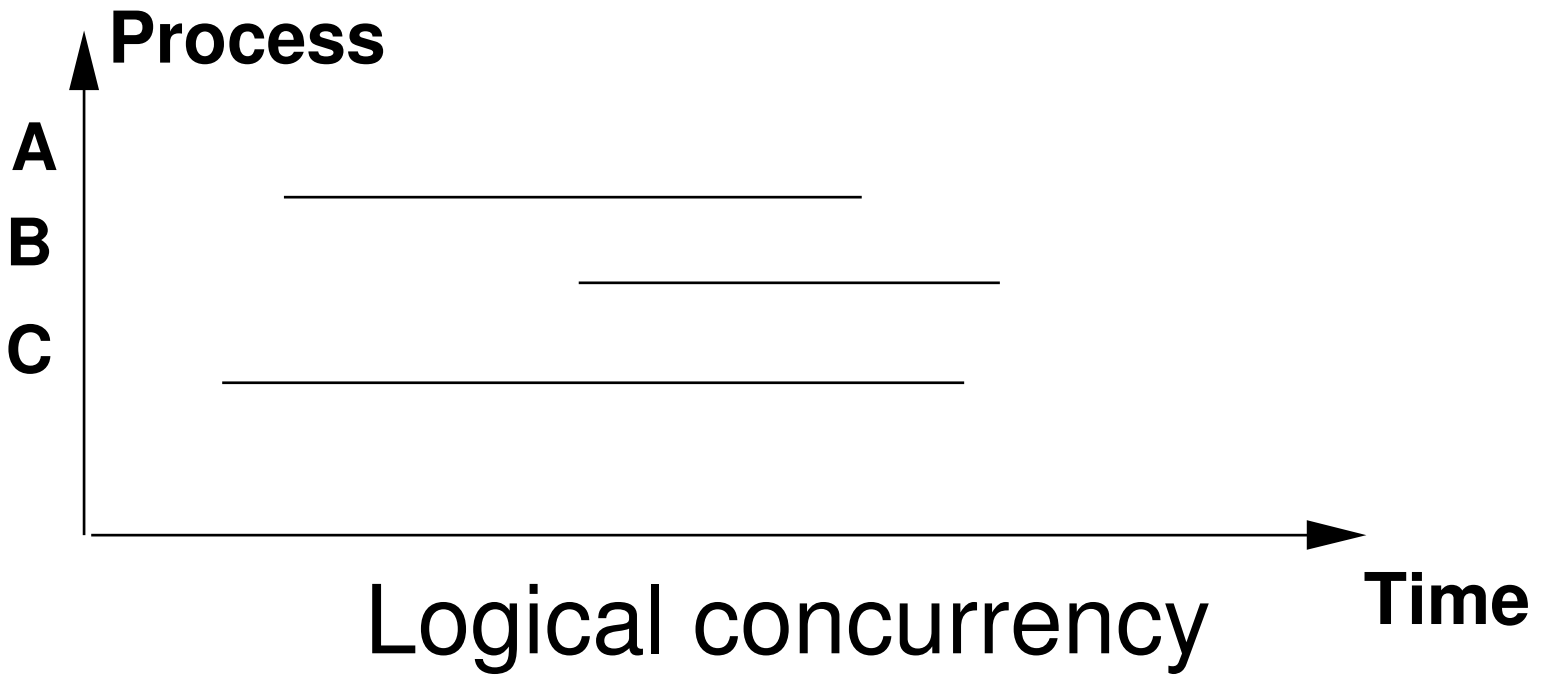
Shared or separate caches

Concurrent Programming

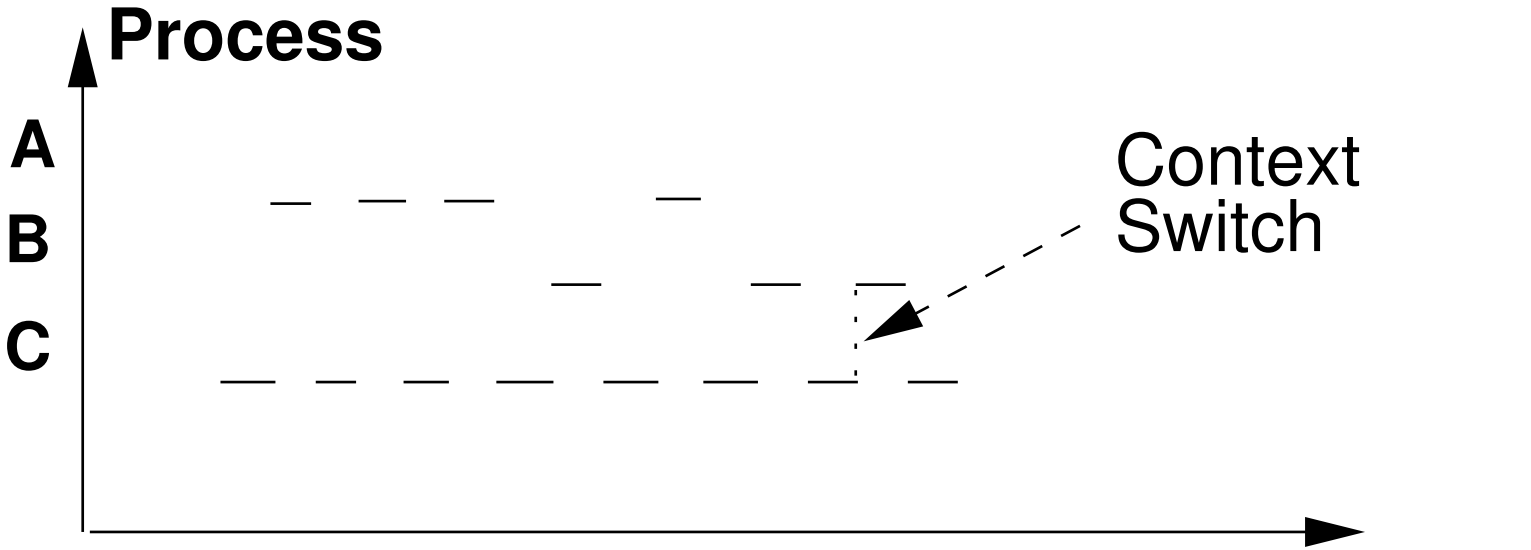
1. **Multiprogramming:** the processes multiplex their execution on a single CPU
2. **Multiprocessing:** the processes multiplex their execution on a multiprocessor system with tightly coupled processors, e.g., a multi-core platform
3. **Distributed processing:** the processes multiplex their execution on several CPUS connected through a network

True parallelism: (2) and (3)

Logical parallelism (pseudo-parallelism): (1) – the main topic



Logical concurrency



Time sharing concurrency

Approaches

- Real-Time Operating System (RTOS)
  - sequential language (C) with real-time primitives
  - real-time kernel for process handling
- Real-time programming language (e.g. Ada)
  - the language itself or the run-time system provides the functionality of a real-time kernel

Memory models

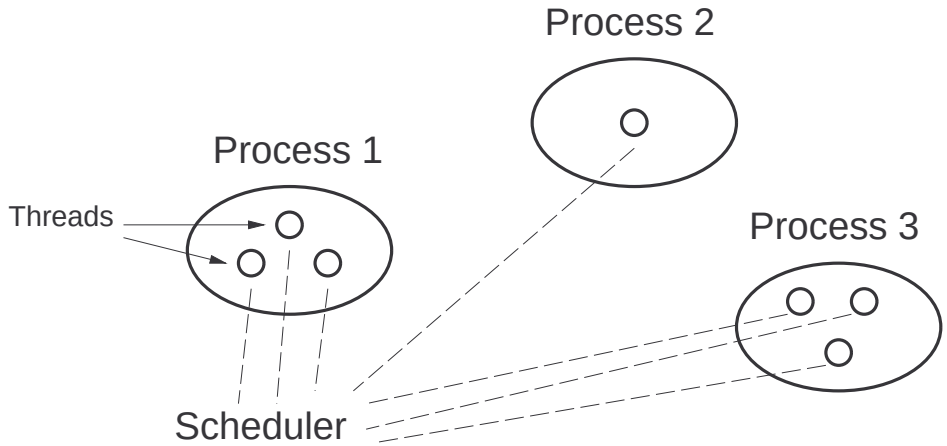
Processes may have

- shared address space
  - **thread** (sometimes also called light-weight process)
  - shared variables and code
  - process = procedure (Modula 2) or `run` method (Java)
  - used in the course
- separate address space
  - independent of each other
  - cp. Windows
  - message-passing

Processes vs threads

A thread (light-weight process) resides within an “ordinary” process and shares it address space.

The scheduler operates on threads.



Windows, Solaris

In the course we will only deal with threads. However, for historical reasons we, incorrectly, call the STORK threads for processes (not unusual).

Processes and Threads in Linux

Linux does not differentiate between processes and threads

All threads are implemented as processes

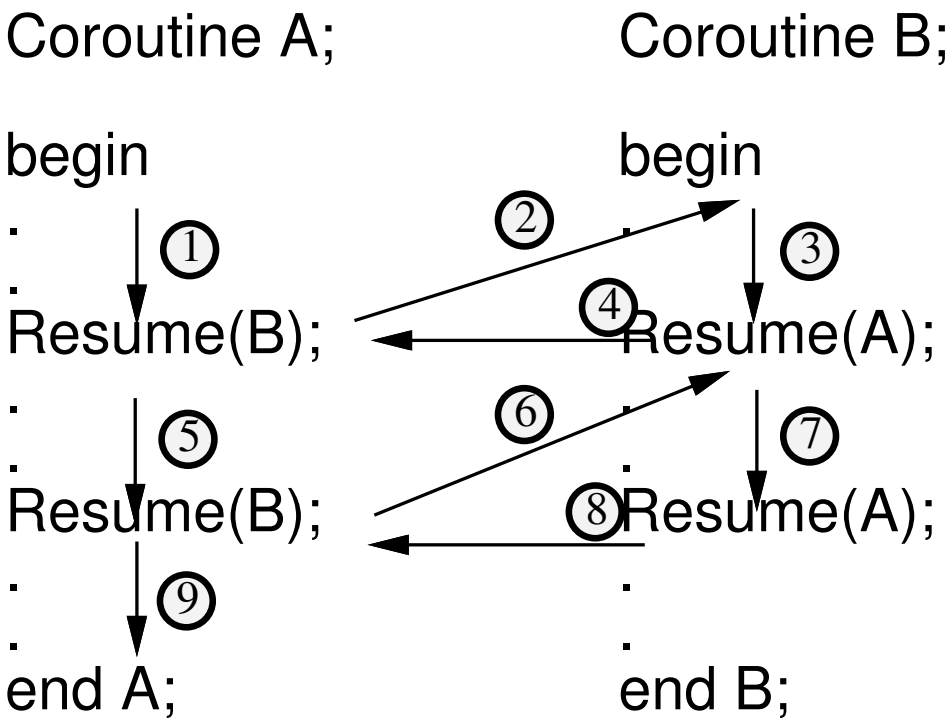
A thread is simply a process that shares certain resources, e.g., memory, with other processes

Coroutines

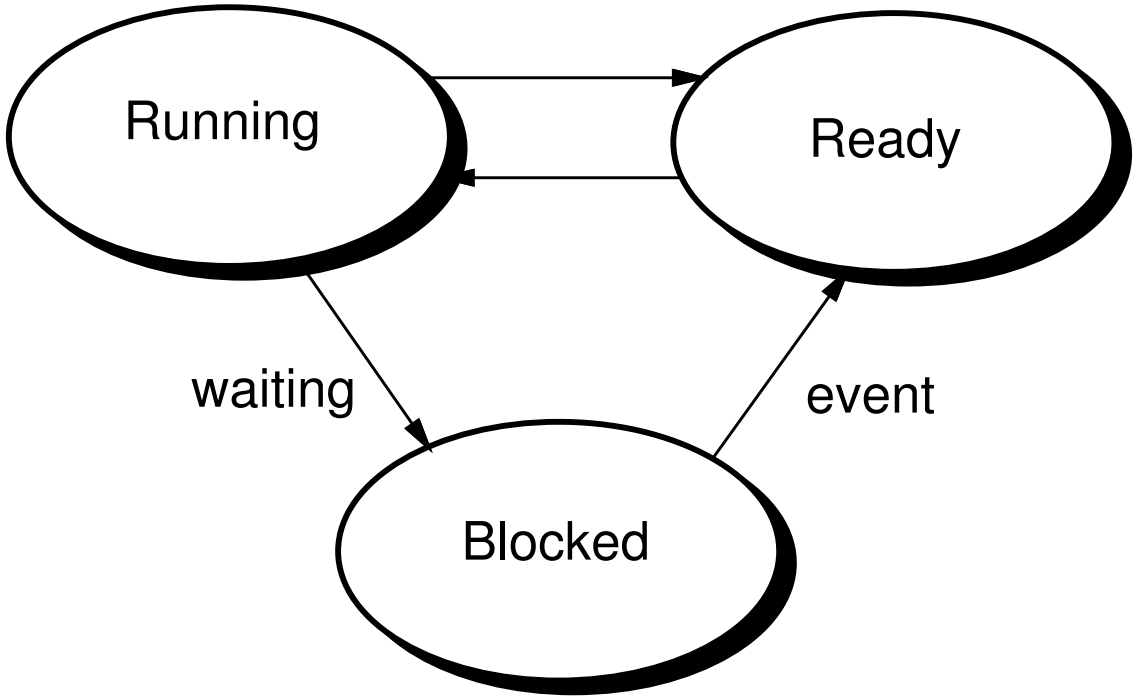
A mechanism for expressing concurrent execution.

A procedure (subroutine) that, instead of returning, resumes another coroutine.

The context (local data) of the coroutines remains in between invocations.



Internal 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 previously running process is **stored** and the context of the process to run next is **restored**.

The context consists of

- the stack that is used
- the contents of the programmable registers
- the contents of the status registers

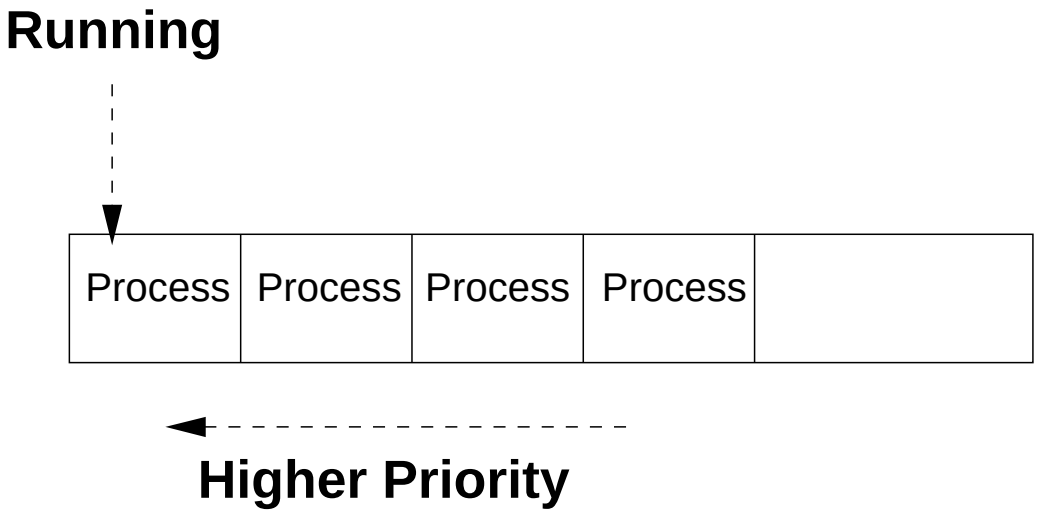
Priorities

Each process is assigned a number that reflects the importance of its time demands.

Many RTOS (incl Stork): Low priority number = high priority. Priority range: 1 - max integer

Java: High priority number = high priority. Priority range: 1 - 10

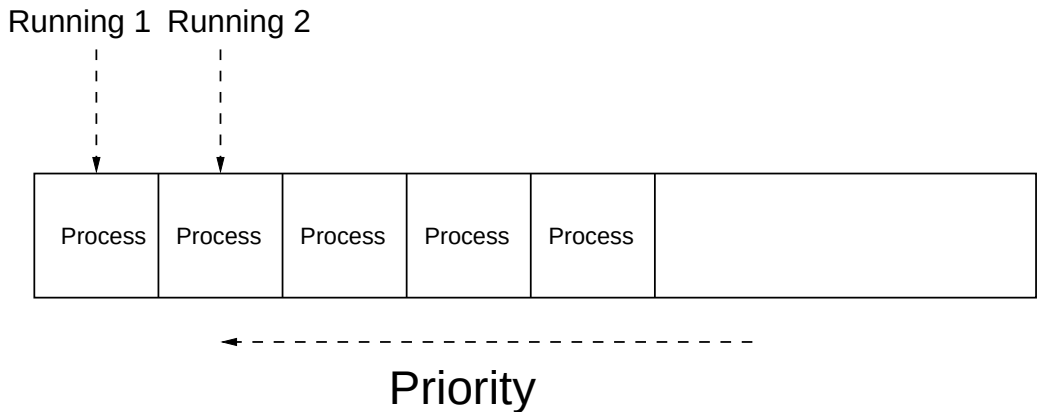
Using priority-based scheduling, processes that are Ready to execute are stored in `ReadyQueue` according to priority.



Multicore Case

Two alternatives: Global scheduling and Partitioned scheduling

Global Scheduling – a single ready queue



Partitioned scheduling

- one ordinary ready queue per core
- often combined with the possibility to move (migrate) processes among cores

A change (insertion, removal) in `ReadyQueue` may lead to a context switch.

If the change results in a new process being the first in `ReadyQueue`, a context switch takes place.

- preemptive scheduling
  - the context switch takes place immediately (the running process is preempted)
  - Most RTOS and real-time languages (e.g. STORK and Java)
- nonpreemptive scheduling
  - the running process continues until it voluntarily releases the CPU, then the context switch takes place
- scheduling based on preemption points
  - the running process continues until it reaches an preemption point, then the context switch takes place
  - early versions of Linux (no preemptions in kernel mode)

Assigning priorities

Non-trivial

Global knowledge

Two ways:

- ad hoc rules
  - important time requirements $\Rightarrow$ high priority
  - time consuming computations $\Rightarrow$ low priority
  - conflicts, no guarantees
- scheduling theory

Often easier to assign deadline than to assign priority

Dynamic priorities

With priority-based scheduling the priority of a process is fixed.

“Fixed priority scheduling”

In the real-time system community it is increasingly common to instead study systems where it is the closeness to the deadline of the processes that decides which process that should execute.

“Earliest Deadline First (EDF) scheduling”

The deadline can be viewed as a dynamic priority that changes as time proceeds.

Still not usual in commercial systems.

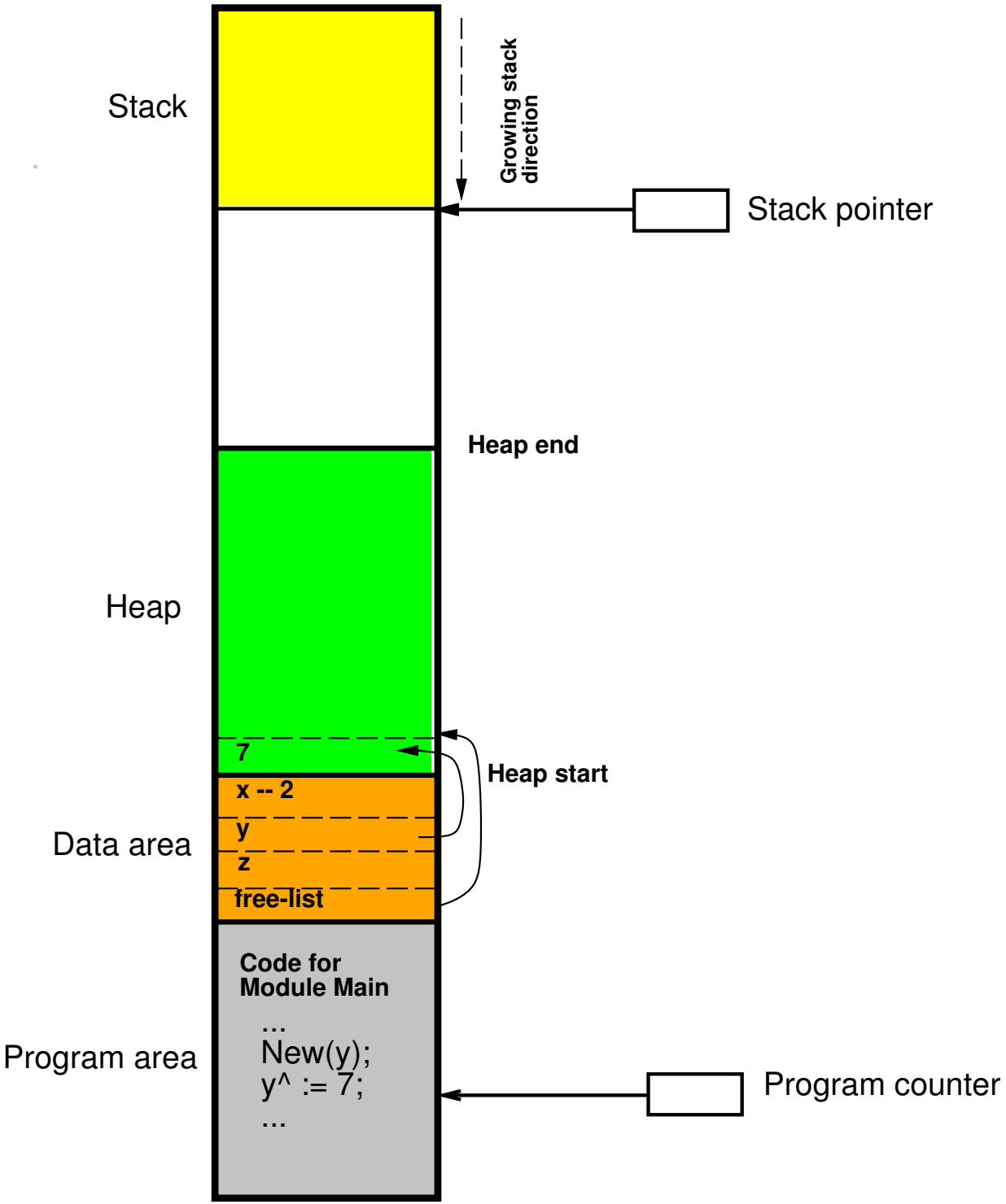
Process Representation

Conceptually a process/thread consists of:

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

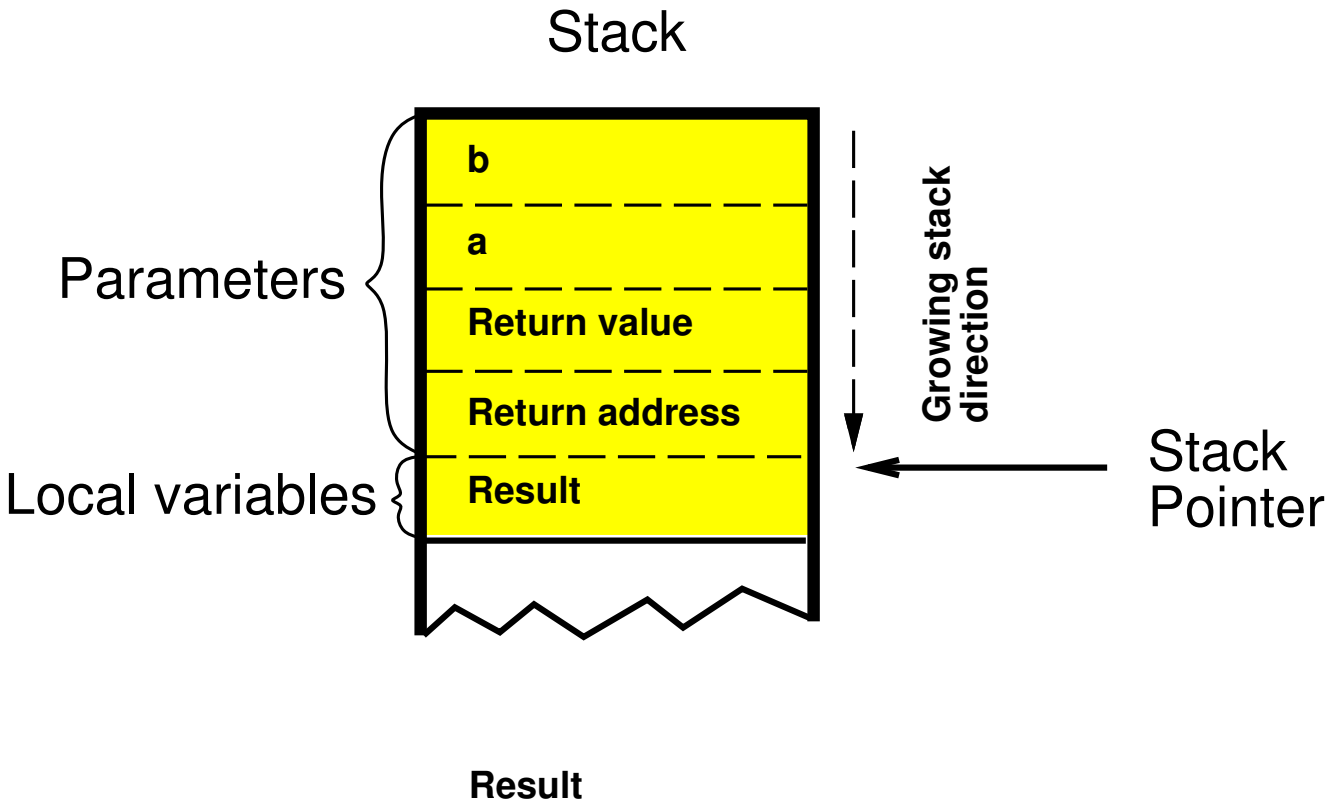
Memory Organization: Sequential Program

```
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.
```


Stack
Growing stack direction
Stack pointer
Heap end
Heap
7
Heap start
x -- 2
y
z
free-list
Data area
Code for Module Main
...
New(y);
y^ := 7;
...
Program area
Program counter

Stack contents when `Demo` is called.

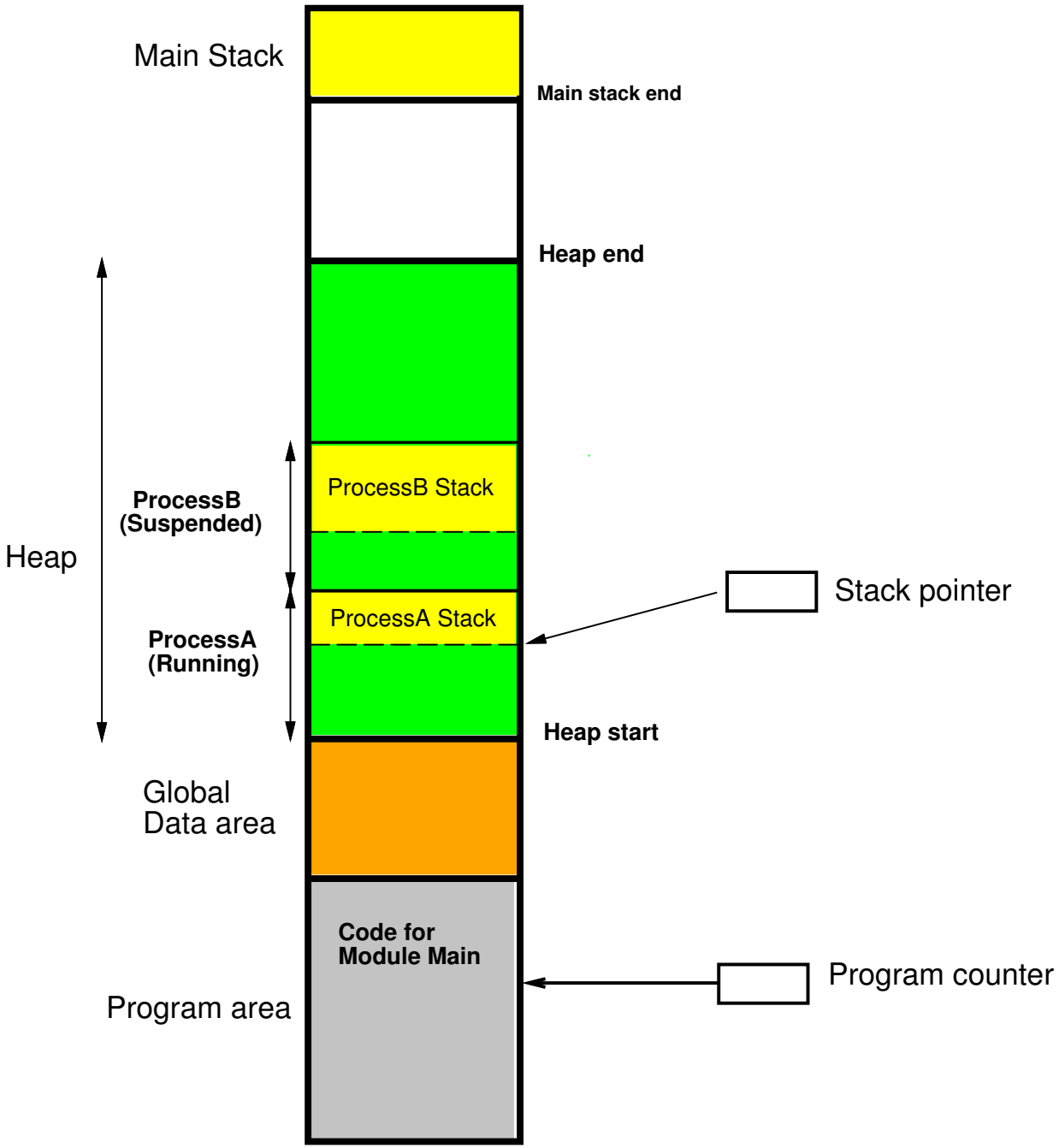
A stack frame is created.



Memory Organization: Concurrency

```
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.
```

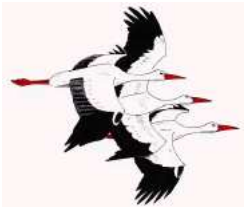
STORK


Main Stack
Main stack end
Heap end
Heap
ProcessB
(Suspended)
ProcessB Stack
ProcessA Stack
ProcessA
(Running)
Stack pointer
Heap start
Global
Data area
Code for
Module Main
Program area
Program counter

Process Record

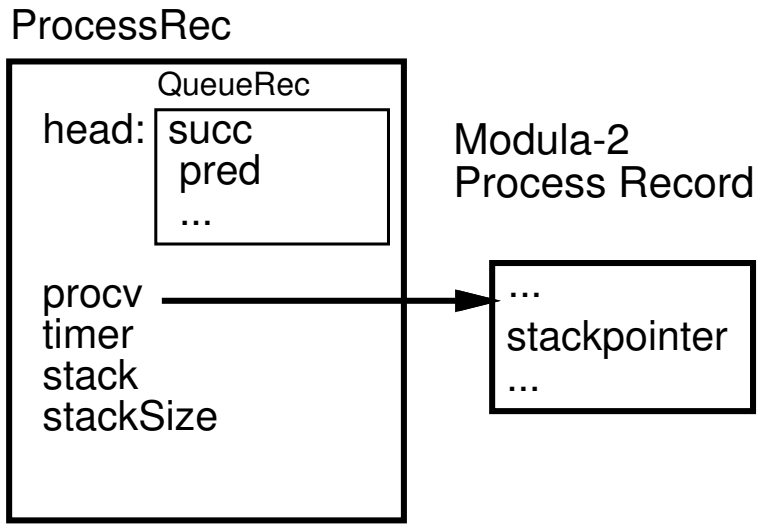
STORK

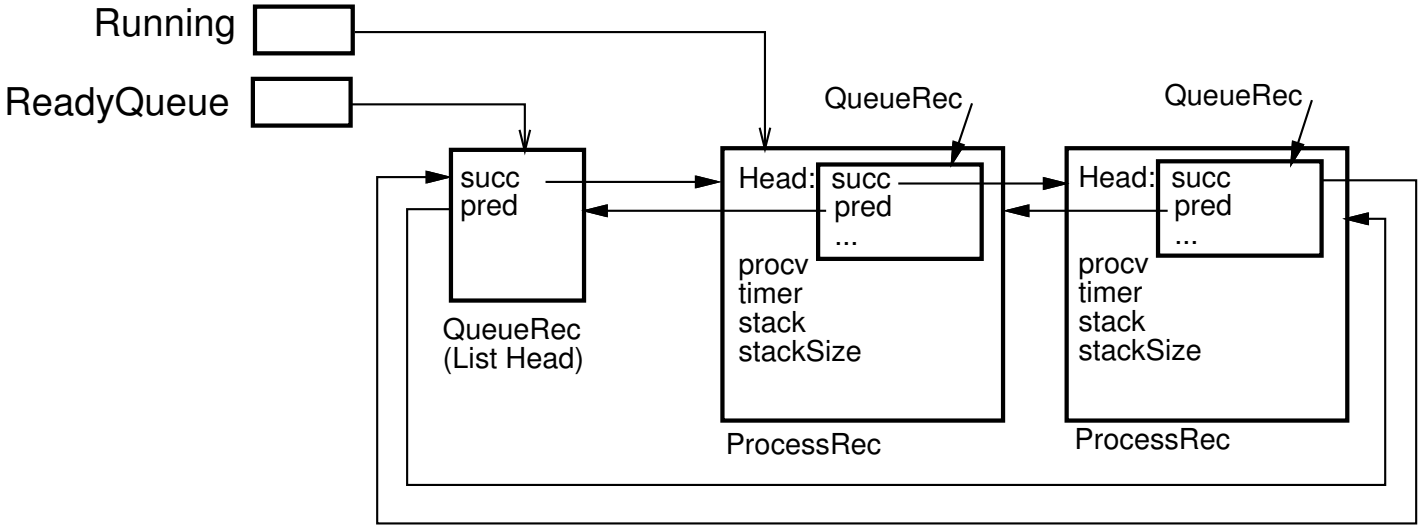
```
TYPE
  PROCESS = ADDRESS;
  ProcessRef = POINTER TO ProcessRec;
  Queue = POINTER TO QueueRec;

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

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



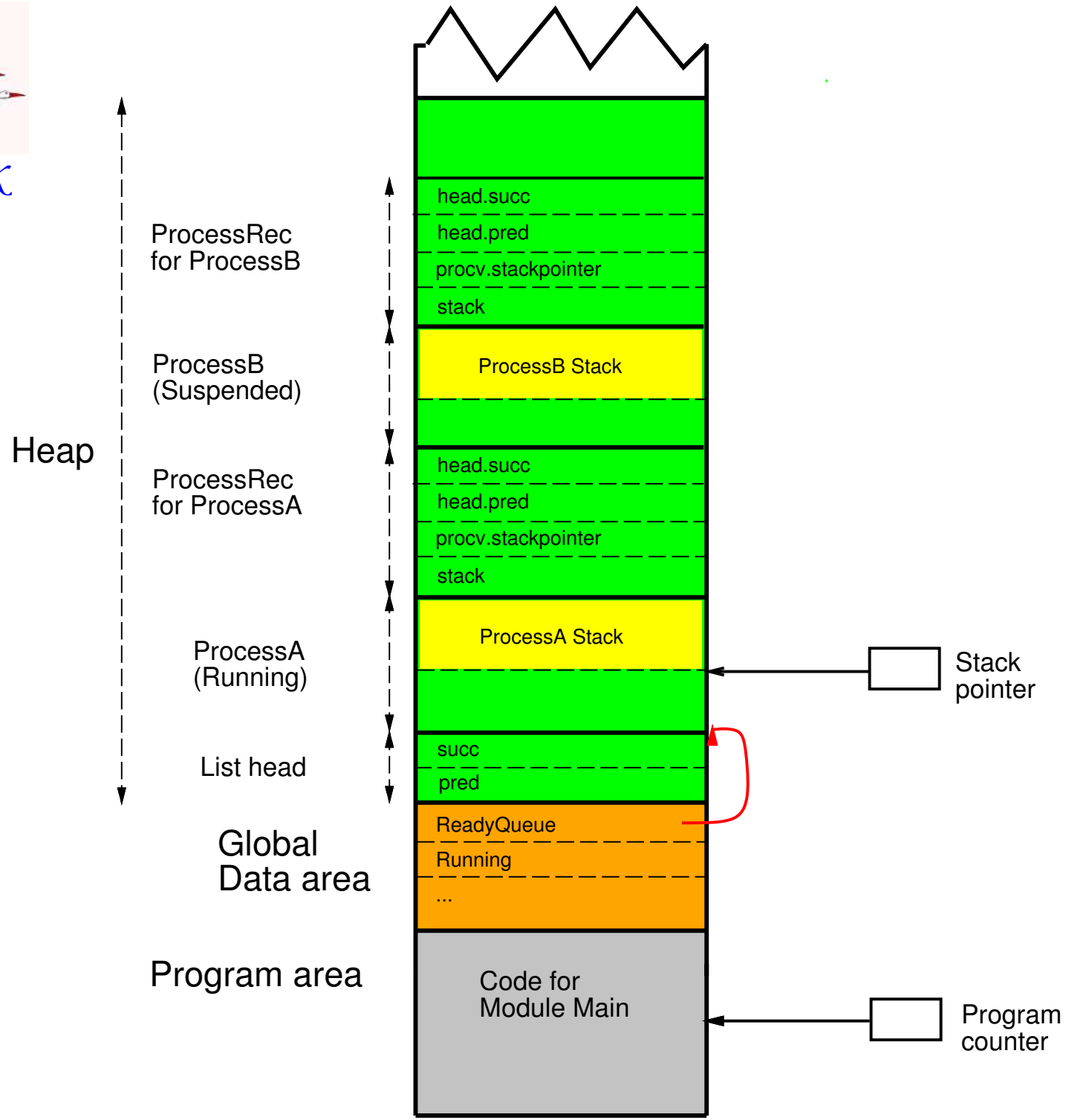
STORK

Real-Time Kernel
Process Record

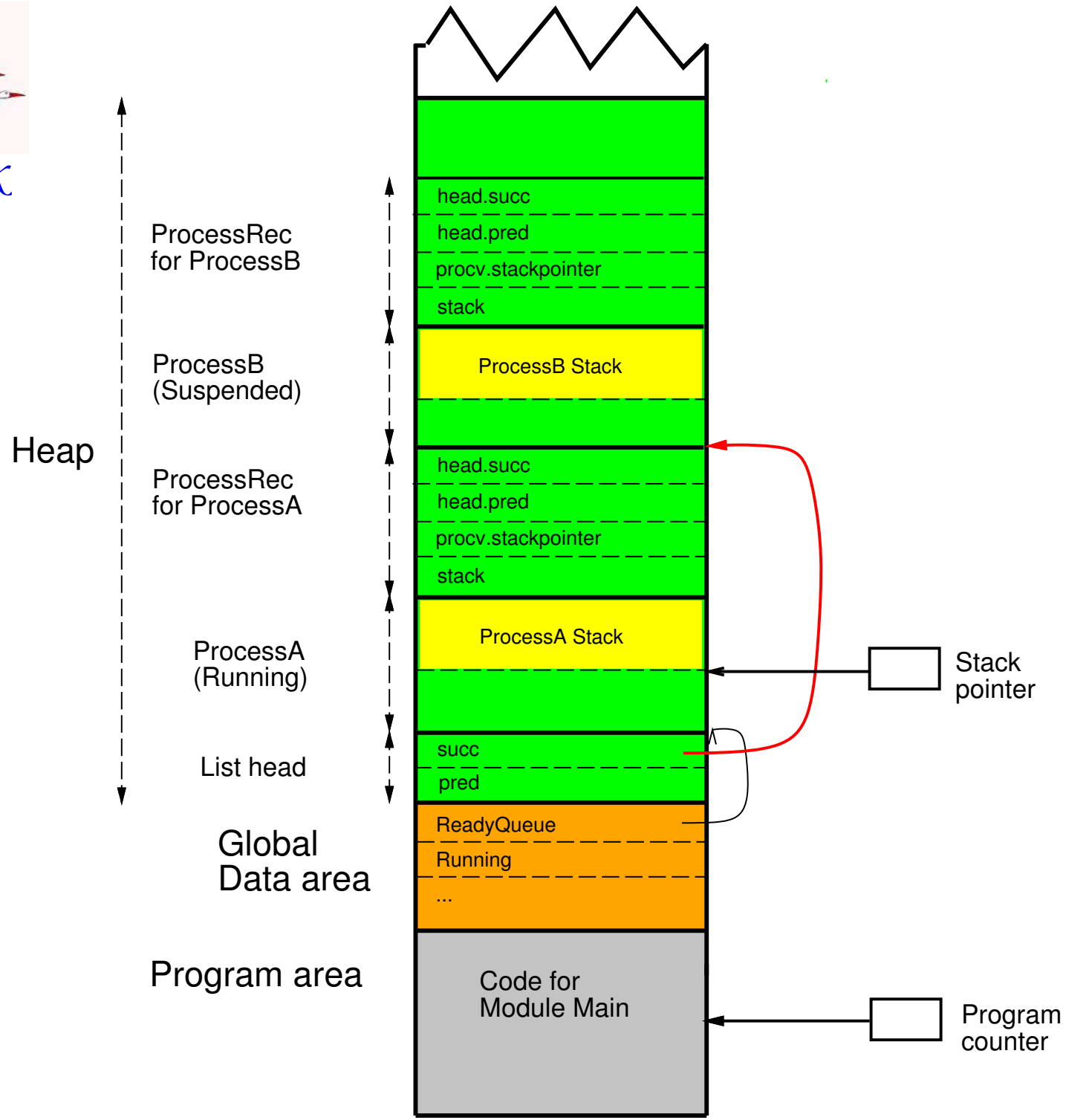




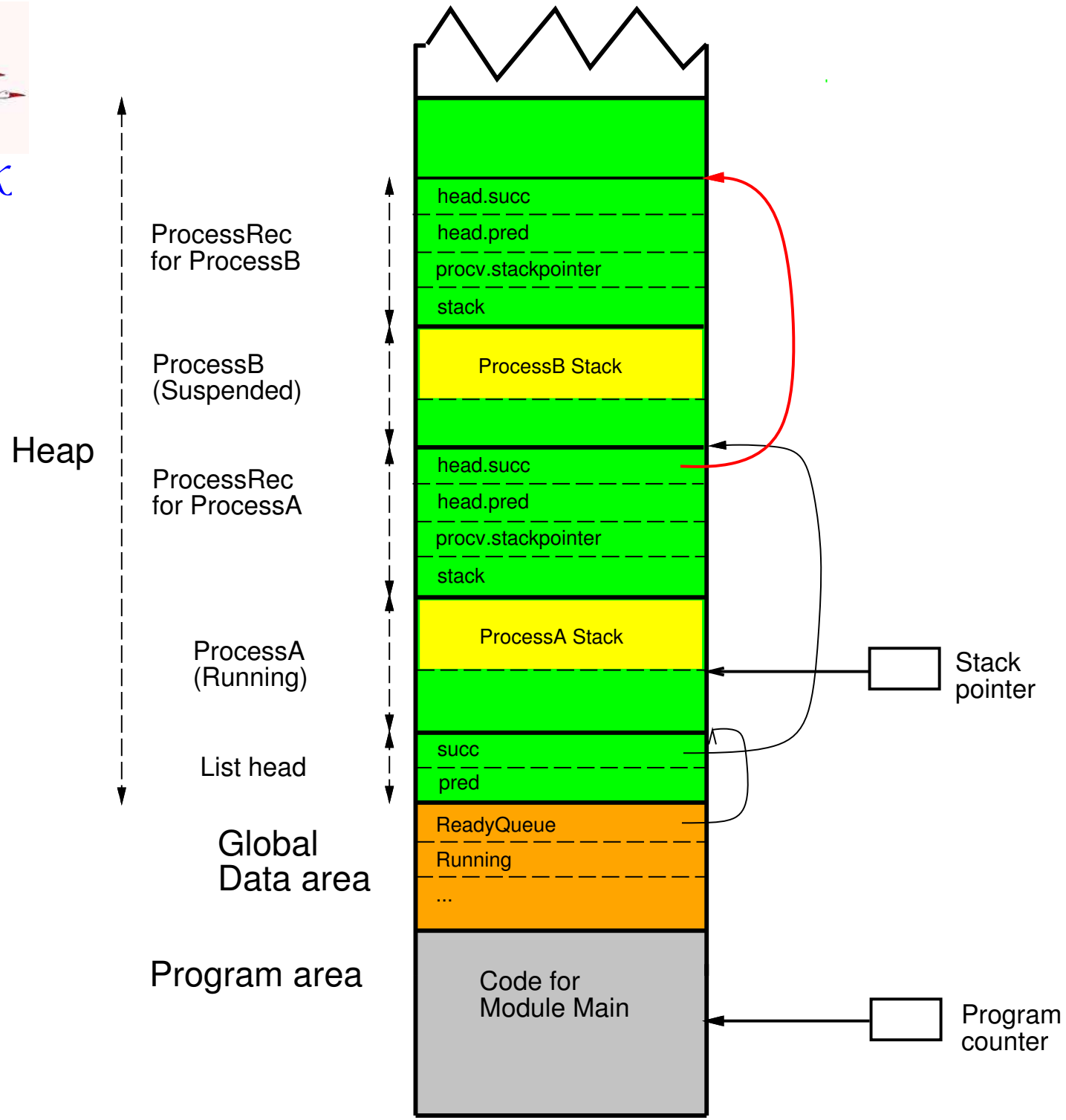
STORK


Heap
ProcessRec
for ProcessB
ProcessB
(Suspended)
ProcessRec
for ProcessA
ProcessA
(Running)
List head
Global
Data area
Program area
head.succ
head.pred
procv.stackpointer
stack
ProcessB Stack
head.succ
head.pred
procv.stackpointer
stack
ProcessA Stack
succ
pred
ReadyQueue
Running
...
Code for
Module Main
Stack
pointer
Program
counter

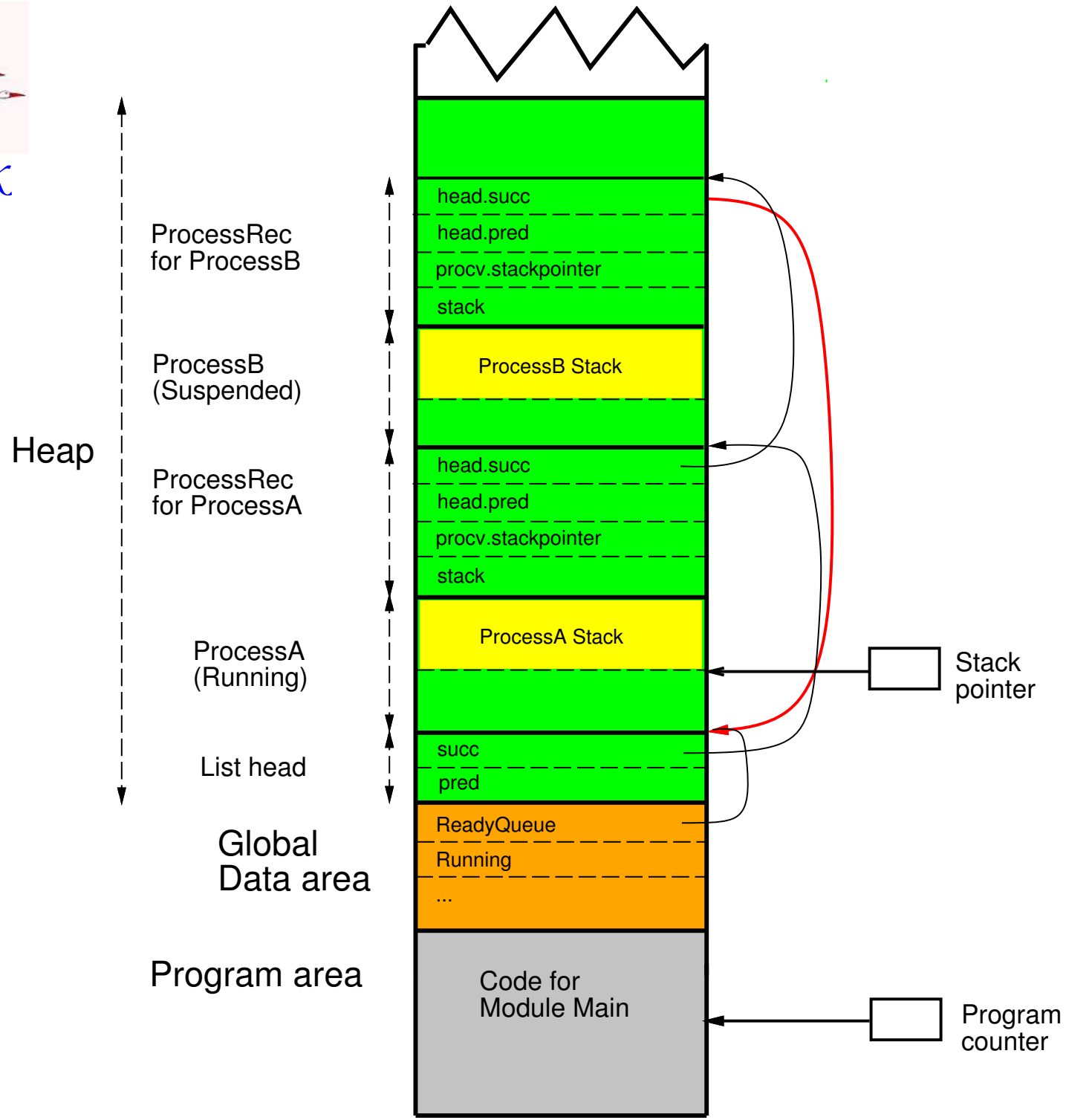
STORK


Heap
ProcessRec
for ProcessB
ProcessB
(Suspended)
ProcessRec
for ProcessA
ProcessA
(Running)
List head
Global
Data area
Program area
head.succ
head.pred
procv.stackpointer
stack
ProcessB Stack
head.succ
head.pred
procv.stackpointer
stack
ProcessA Stack
succ
pred
ReadyQueue
Running
...
Code for
Module Main
Stack
pointer
Program
counter

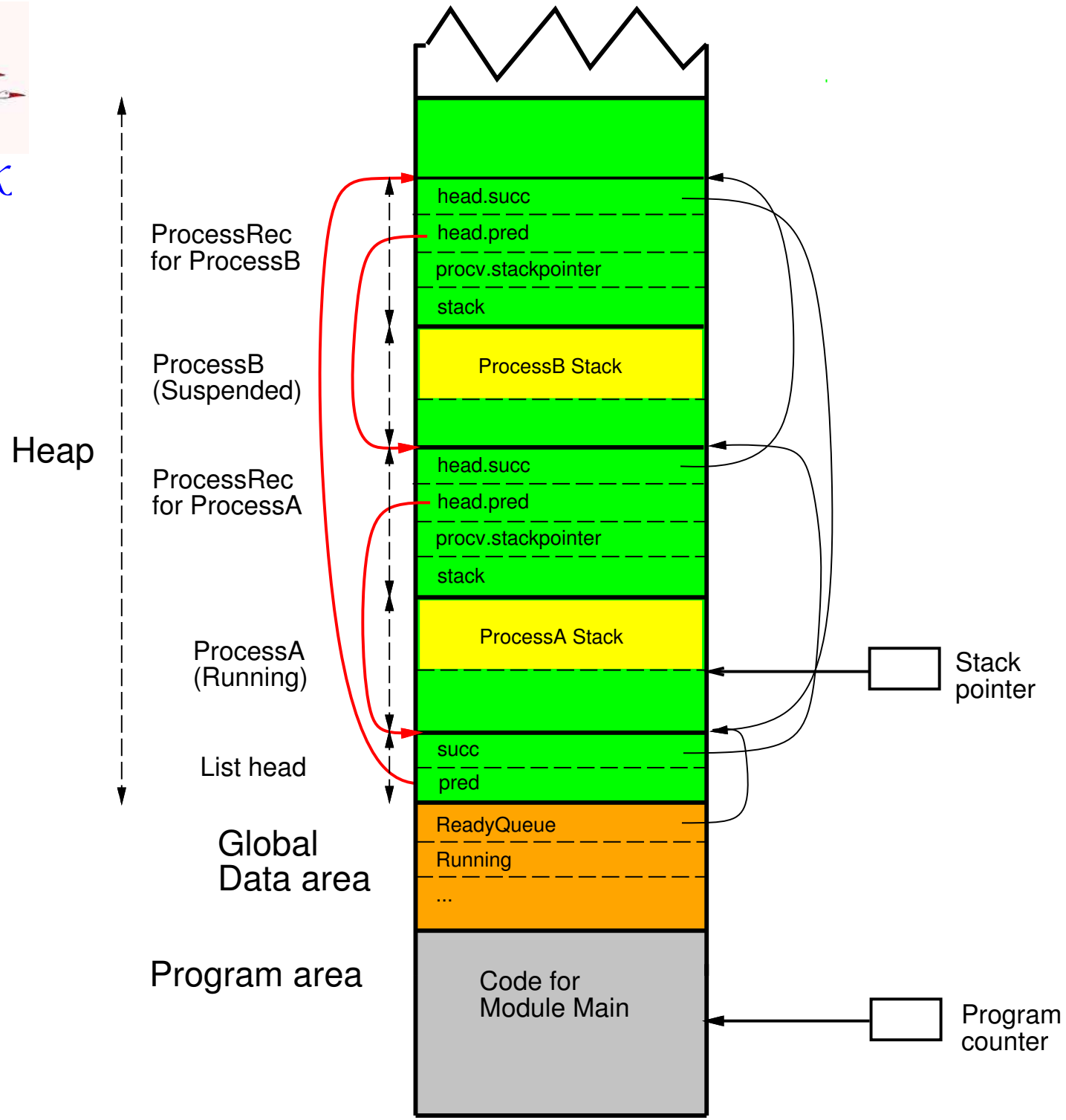
STORK


Heap
ProcessRec
for ProcessB
ProcessB
(Suspended)
ProcessRec
for ProcessA
ProcessA
(Running)
List head
Global
Data area
Program area
head.succ
head.pred
procv.stackpointer
stack
ProcessB Stack
head.succ
head.pred
procv.stackpointer
stack
ProcessA Stack
succ
pred
ReadyQueue
Running
...
Code for
Module Main
Stack
pointer
Program
counter

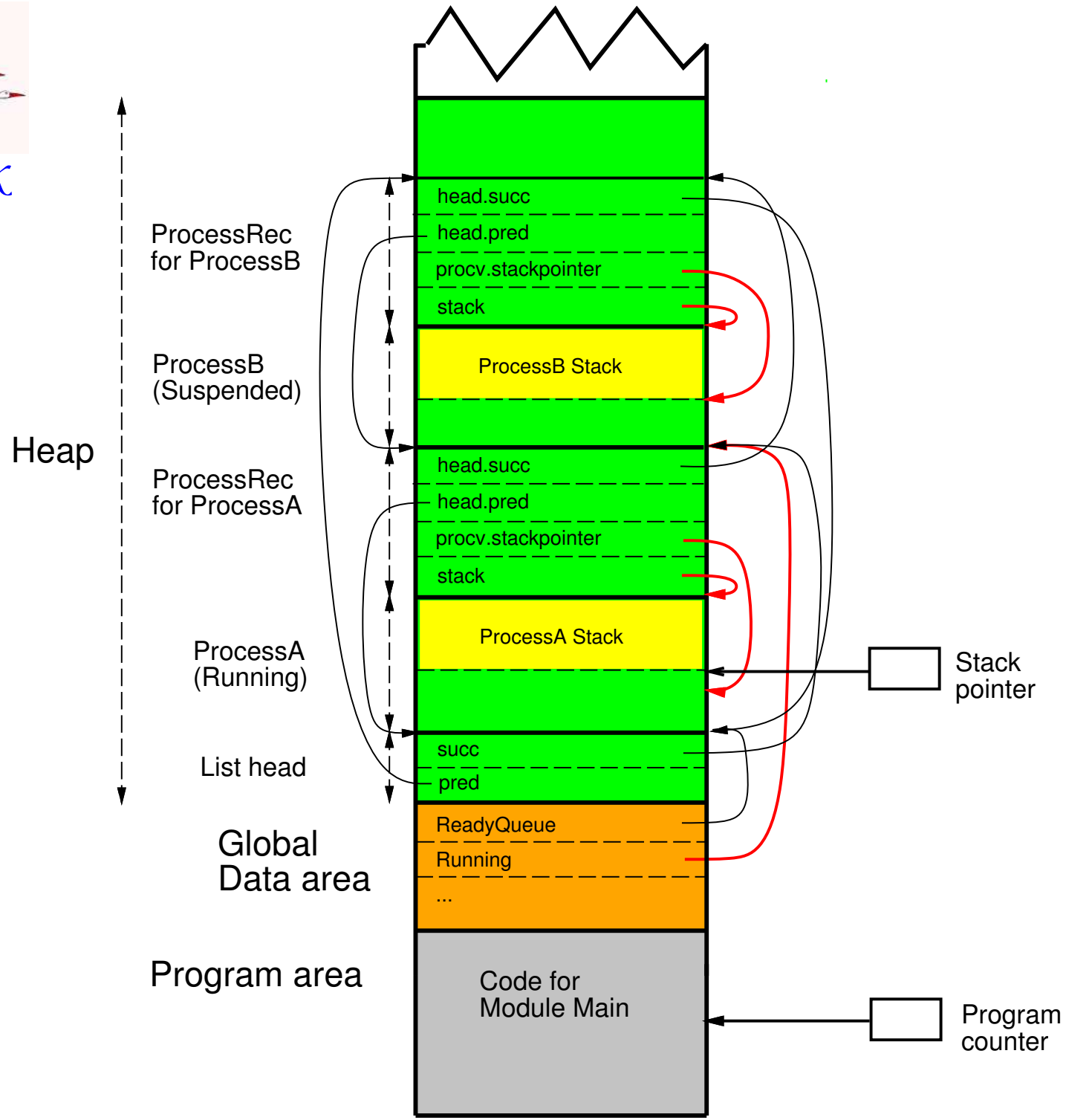
STORK

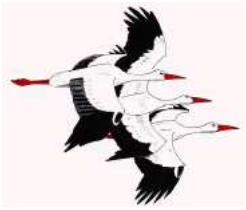

Heap
ProcessRec
for ProcessB
ProcessB
(Suspended)
ProcessRec
for ProcessA
ProcessA
(Running)
List head
Global
Data area
Program area
head.succ
head.pred
procv.stackpointer
stack
ProcessB Stack
head.succ
head.pred
procv.stackpointer
stack
ProcessA Stack
succ
pred
ReadyQueue
Running
...
Code for
Module Main
Stack
pointer
Program
counter

STORK


Heap
ProcessRec
for ProcessB
ProcessB
(Suspended)
ProcessRec
for ProcessA
ProcessA
(Running)
List head
Global
Data area
Program area
head.succ
head.pred
procv.stackpointer
stack
ProcessB Stack
head.succ
head.pred
procv.stackpointer
stack
ProcessA Stack
succ
pred
ReadyQueue
Running
...
Code for
Module Main
Stack
pointer
Program
counter

STORK


Heap
ProcessRec
for ProcessB
ProcessB
(Suspended)
ProcessRec
for ProcessA
ProcessA
(Running)
List head
Global
Data area
Program area
head.succ
head.pred
procv.stackpointer
stack
ProcessB Stack
head.succ
head.pred
procv.stackpointer
stack
ProcessA Stack
succ
pred
ReadyQueue
Running
...
Code for
Module Main
Stack
pointer
Program
counter



STORK

Context Switching

Initiated by a call to `Schedule`

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

Happens when:

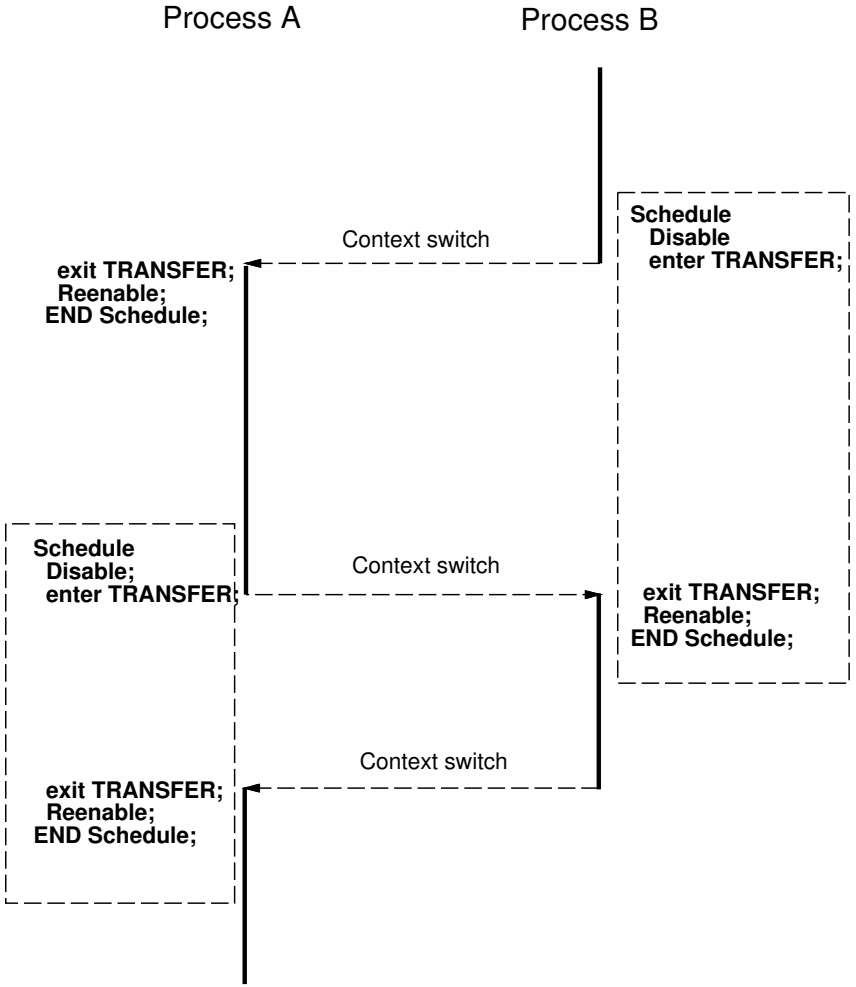
- when the running process voluntarily releases the CPU
- when the running process performs an operation that may cause a blocked process to become ready
- when an interrupt has occurred which may have caused a blocked process to become ready

Schedule

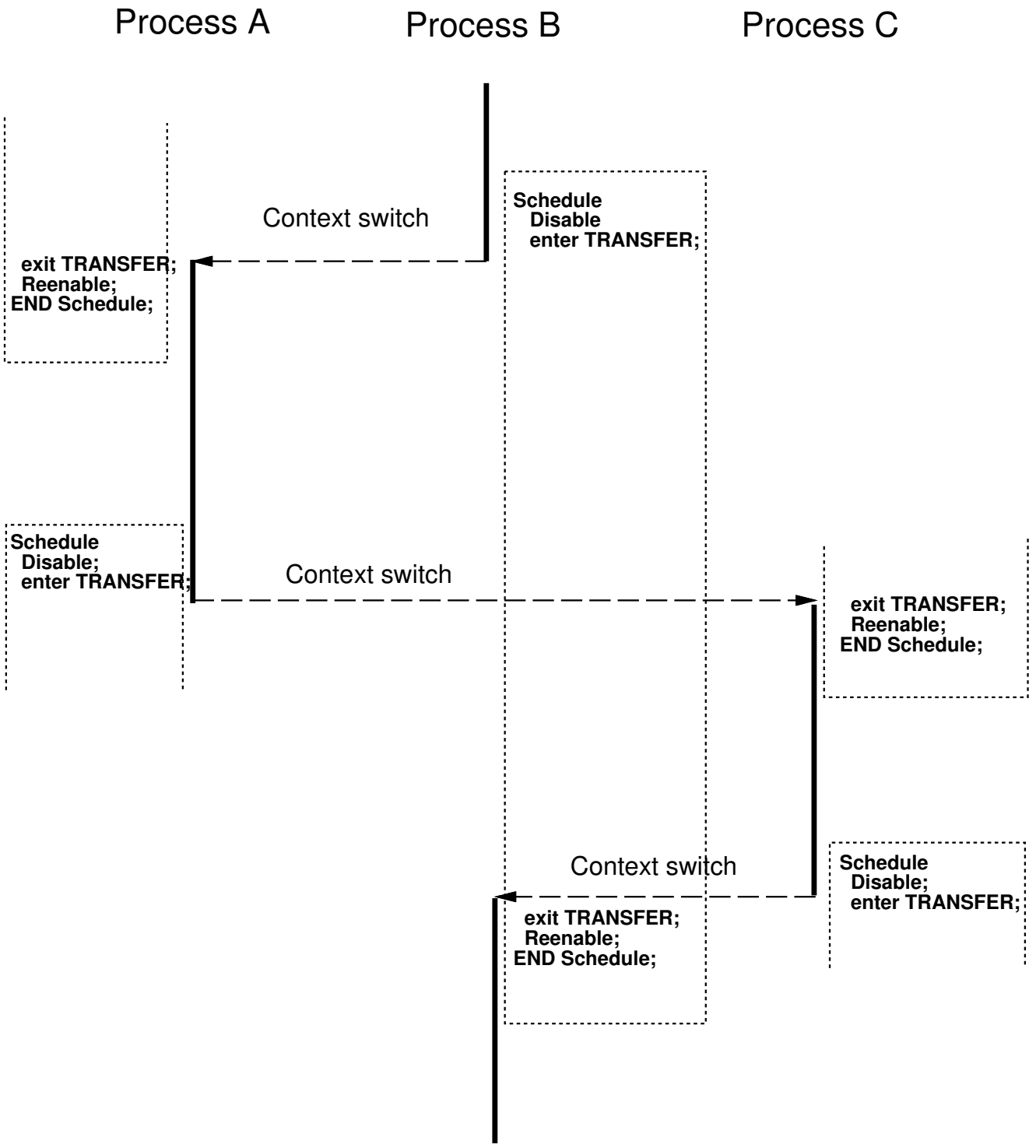
```
PROCEDURE Schedule;

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;
```

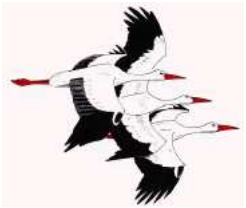
- interrupts disabled
- the actual context switch takes place in the Modula-2 primitive `TRANSFER`
- the Modula-2 process record pointer as argument

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



STORK


Process A
Process B
Process C
Schedule
Disable
enter TRANSFER;
Context switch
exit TRANSFER;
Reenable;
END Schedule;
Schedule
Disable;
enter TRANSFER;
Context switch
exit TRANSFER;
Reenable;
END Schedule;
Context switch
Schedule
Disable;
enter TRANSFER;
exit TRANSFER;
Reenable;
END Schedule;

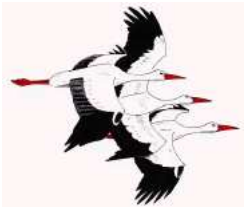


STORK

Inside `TRANSFER`

The actual context switch

- **Saving**: the state of the processor immediately before the switch is saved on the stack of the suspended process
- **Switching**: the context switch
  - 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 record of the new process
- **Restoring**: the state of the resumed process is restored (popped from its stack)



STORK

Saving	Turn off all interrupts Push the program counter on the stack; Push the programmable registers on the stack; Push the status registers on the stack;	Old context
Switching	Save the value of the stack pointer in the process record; Set the value of the stack pointer to the value stored in the new process;	
Restoring	Pop the status registers from the stack; Pop the programmable registers from the stack; Pop the program counter from the stack; Turn on all interrupts;	New Context

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 the case of two hyperthreads) is used for the thread that is next to be executed, e.g., the thread with the next-highest priority.

When context-switching between these two threads no context need to be saved and restored.

The processor only needs to change which set of register that it operates upon, which takes substantially less time.

Java in Real-Time

Three possibilities:

1. Ahead-of-time (AOT) Compilation
   - Java or Java bytecode to native code
   - Java or Java bytecode to intermediate language (C)
2. Java Virtual Machine (JVM)
   - as a process in an existing OS
     - green thread model — the threads are handled internally by the JVM
     - native thread model — the Java threads are mapped onto the threads of the OS
   - executing on an empty machine
     - green thread model
3. direct hardware support, e.g., through micro-code

Java Execution Models

Compiled Java:

- works essentially in the same way as presented for STORK

JVM with native-thread model:

- each Java thread is executed by a native thread
- similar to what has been presented before

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 the threads
  - the thread's stack
  - the current instruction of the thread
  - bookkeeping information
- the JVM performs context switching between threads by saving and restoring thread contexts
- the JVM program counter points at the current instruction of the executing thread
- the global program counter points at the current instruction of the JVM

Thread creation

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.

A Java thread *active object*, as opposed to a *passive object*.

Thread creation: Extending `Thread`

```
public class MyThread extends Thread {
  public void run() {
    // Code to be executed
  }
}
```

Start of the thread:

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

Thread creation: Implementing `Runnable`

This way is used when the active object needs to extend some other class than `Thread`.

```
public class MyClass implements Runnable {
  public void run() {
    // Code to be executed
  }
}
```

Since an instance of MyClass is no longer a Thread object, the starting of the thread is slightly more complicated.

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

Drawback: non-static Thread methods are not directly accessible inside the run method (example later on)

Thread creation: Thread as a variable

```
public class MyThread extends Thread {
  MyClass owner;

  public MyThread(MyClass m) {
    owner = m;
  }
  public void run() {
    // 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 execution threads.

Thread creation: Thread as an inner class

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

  class MyThread extends Thread {
     public void run() {
       // Code to be executed
     }
  }

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

No explicit `owner` reference needed. MyThread has direct access to variables and methods of MyClass.

The inner class need not have a name – an anonymous class (Exercise 1)

Thread Priorities

A newly created thread inherits the priority of the creating thread.

The default priority is `NORM_PRIORITY` which is 5

Thread priority is changed by calling the nonstatic Thread method `setPriority`.

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

The static `currentThread` method is used to get a reference to thread of MyClass. This reference is then used to call the nonstatic `setPriority` method.

Thread termination

A thread terminates when its `run` method terminates.

To stop a thread from some other thread the recommended solution is to use a flag.

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

  public void run() {
    while (doIt) {
       // Code to be executed periodically
    }
  }
}
```

The thread is stopped by setting the `doIt` flag to false, either directly or by using some access method provided by MyClass.

Sleeping threads will not terminate immediately

Scheduling in Linux

Four different scheduling classes (schedulers)

- SCHED_NORMAL
  - the default scheduler
  - from 2.6.23 based on the Completely Fair Scheduler (CFS)
  - not a real-time scheduler
  - round robin-based
  - fairness and efficiency major design design goals
  - tradeoff between low latency (e.g., IO-bound processes) and high throughput (e.g., for compute-bound processes)
- SCHED_FIFO and SCHED_RR
  - two quite similar real-time scheduling policies
  - always have priority over SCHED_NORMAL tasks
  - behaviour similar to, e.g. STORK

Scheduling in Linux, cont

- SCHED_DEADLINE
  - Earliest-Deadline-First Scheduler (more in later lectures)
  - Support for CPU reservations
  - Has priority over all the other scheduling policies
  - Added in Linux 3.14, March 2014
  - A result from the EU project ACTORS led by Ericsson in Lund and with our department as partner