Concurrent Systems

Doing many things at the same time

Computadores II / 2004-2005

Characteristics of RTS

- *Large and complex*
- **Concurrent control of separate system components**
- Facilities to interact with special purpose hardware.
- Guaranteed response times
- Extreme reliability
- Efficient implementation

Aim

- To illustrate the requirements for concurrent programming
- To demonstrate the variety of models for creating processes
- To show how processes are created in Ada (tasks), POSIX/C (processes and threads) and Java (threads)
- To lay the foundations for studying inter-process communication

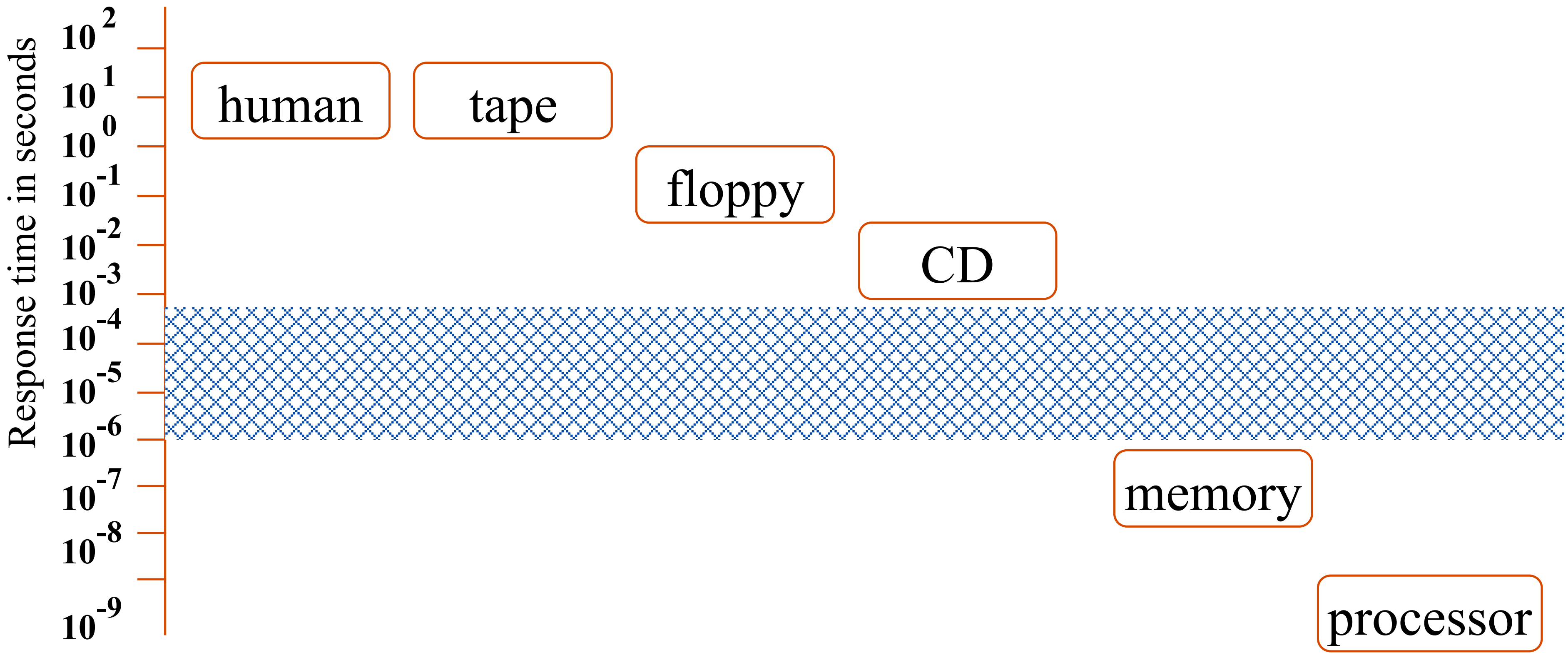
Concurrent Programming

- The name given to programming notation and techniques for **expressing potential parallelism** and solving the resulting synchronization and communication problems

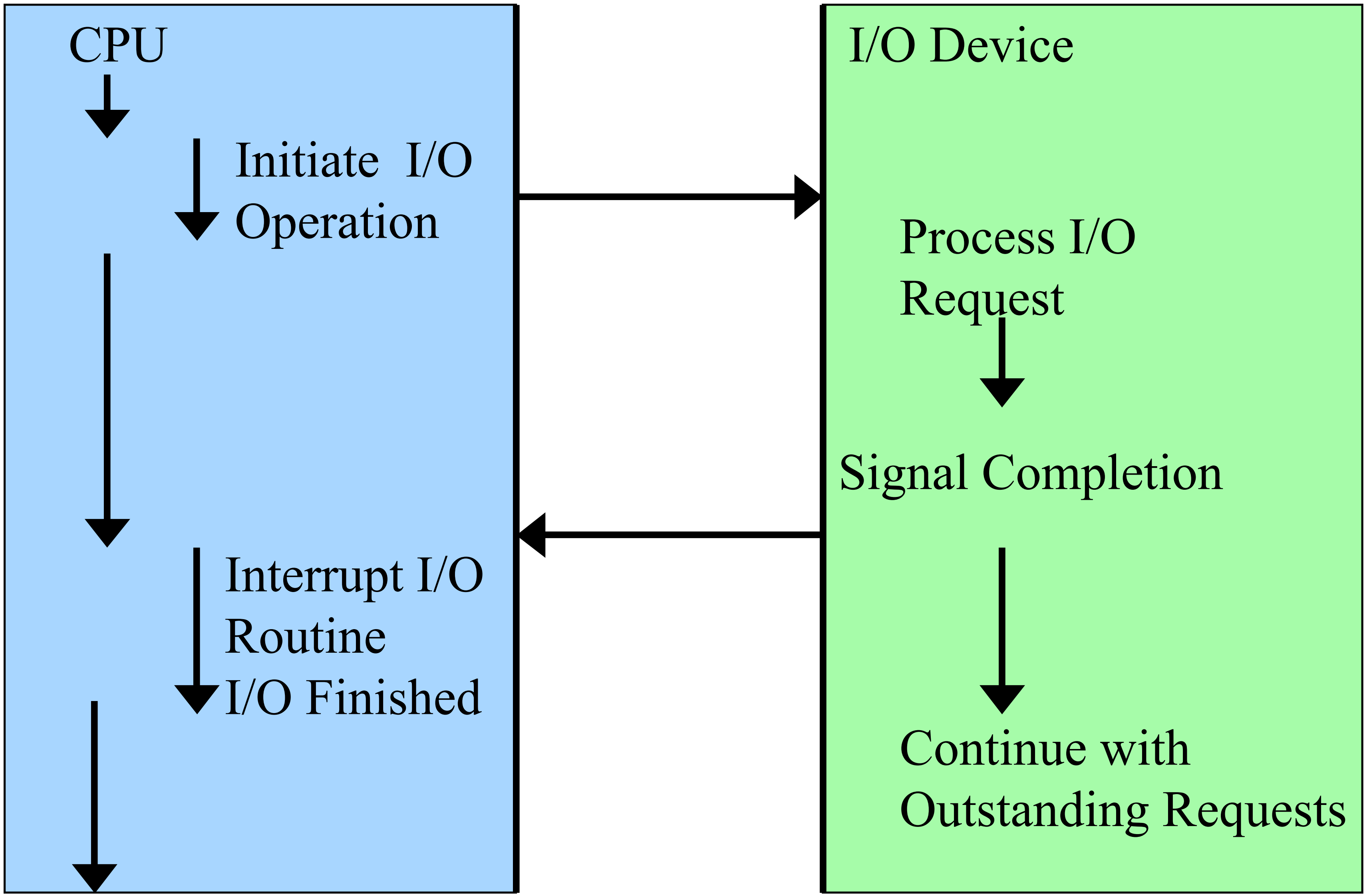
- Implementation of parallelism is a topic in computer systems (hardware and software) that is essentially independent of concurrent programming
- Concurrent programming is important because it provides an abstract setting in which to study parallelism without getting bogged down in the implementation details

Why we need it

- To **fully utilise the processor**(s)



Parallelism Between CPU and I/O

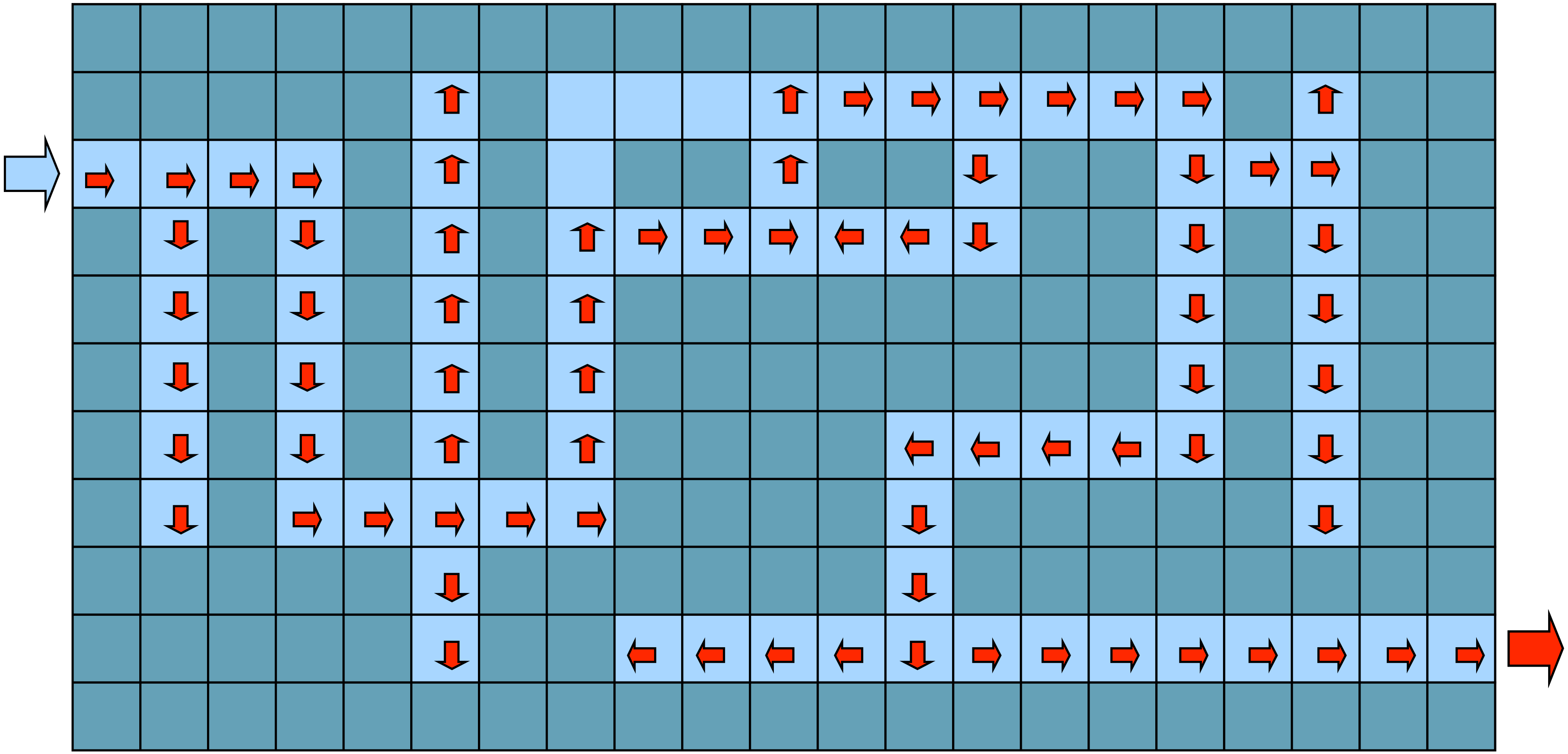


Why we need concurrency

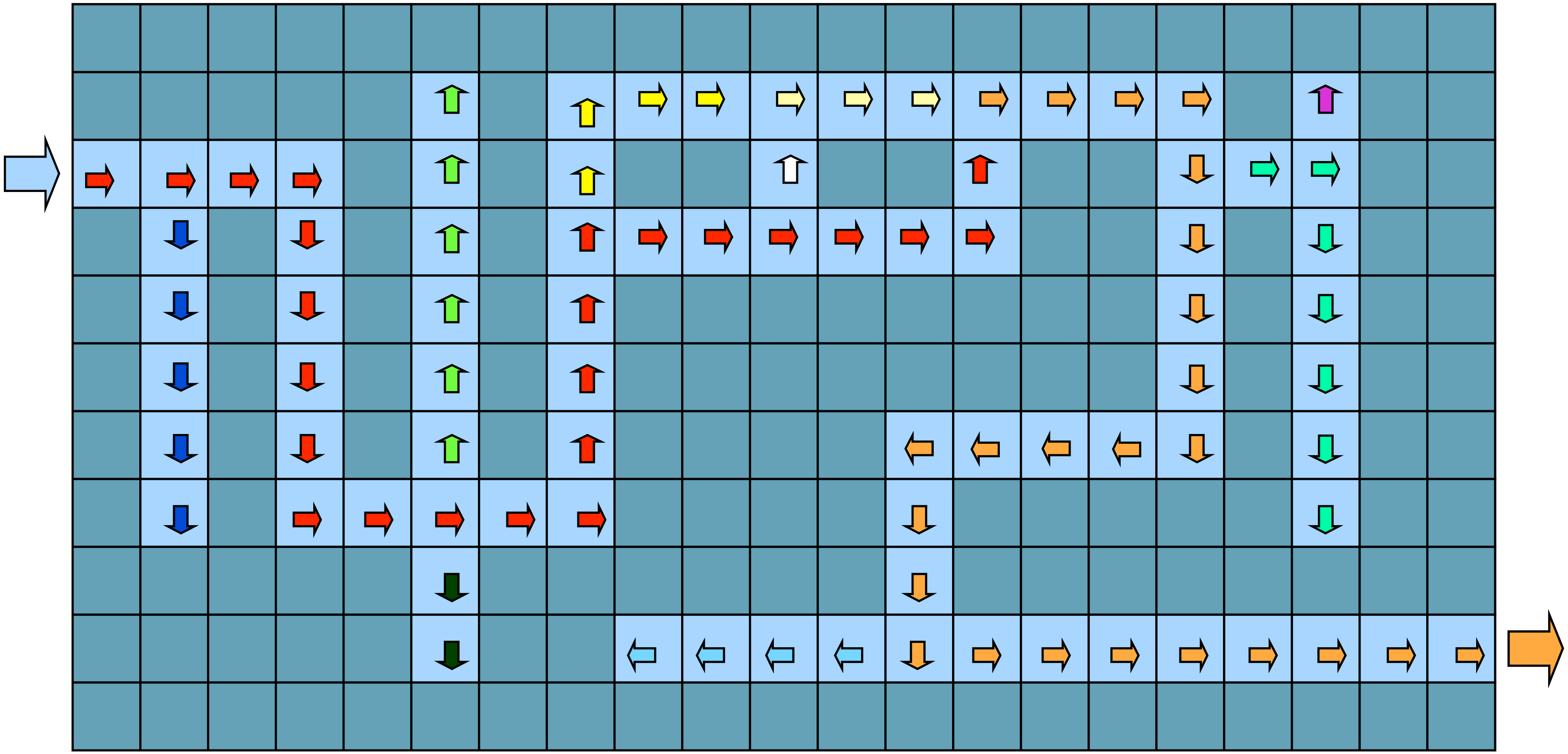
- To allow the **expression of potential parallelism** so that more than one computer can be used to solve the problem

- Consider trying to find the way through a maze

Sequential Maze Search



Concurrent Maze Search



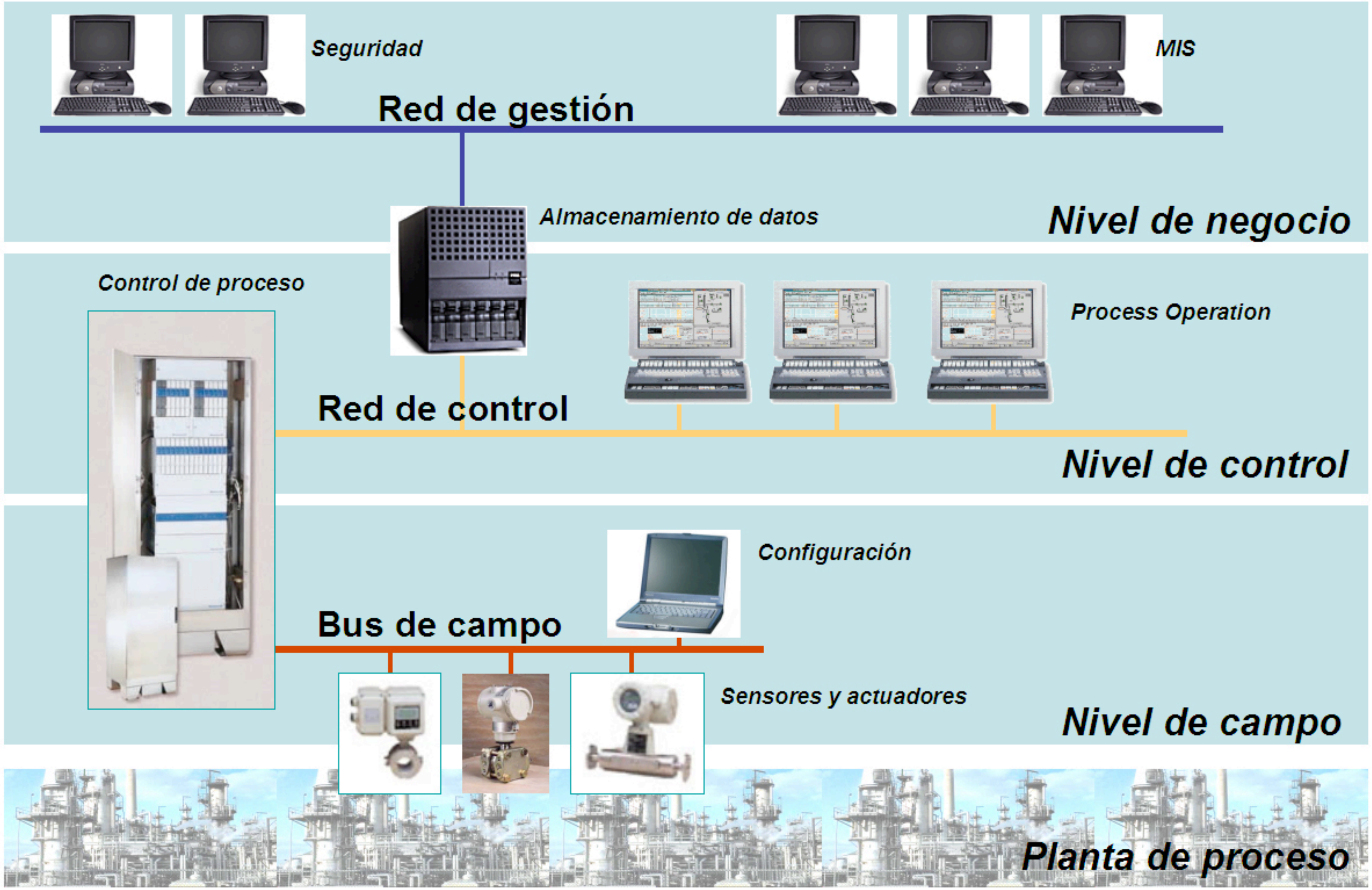
Why we need it

- To model the **parallelism in the real world**

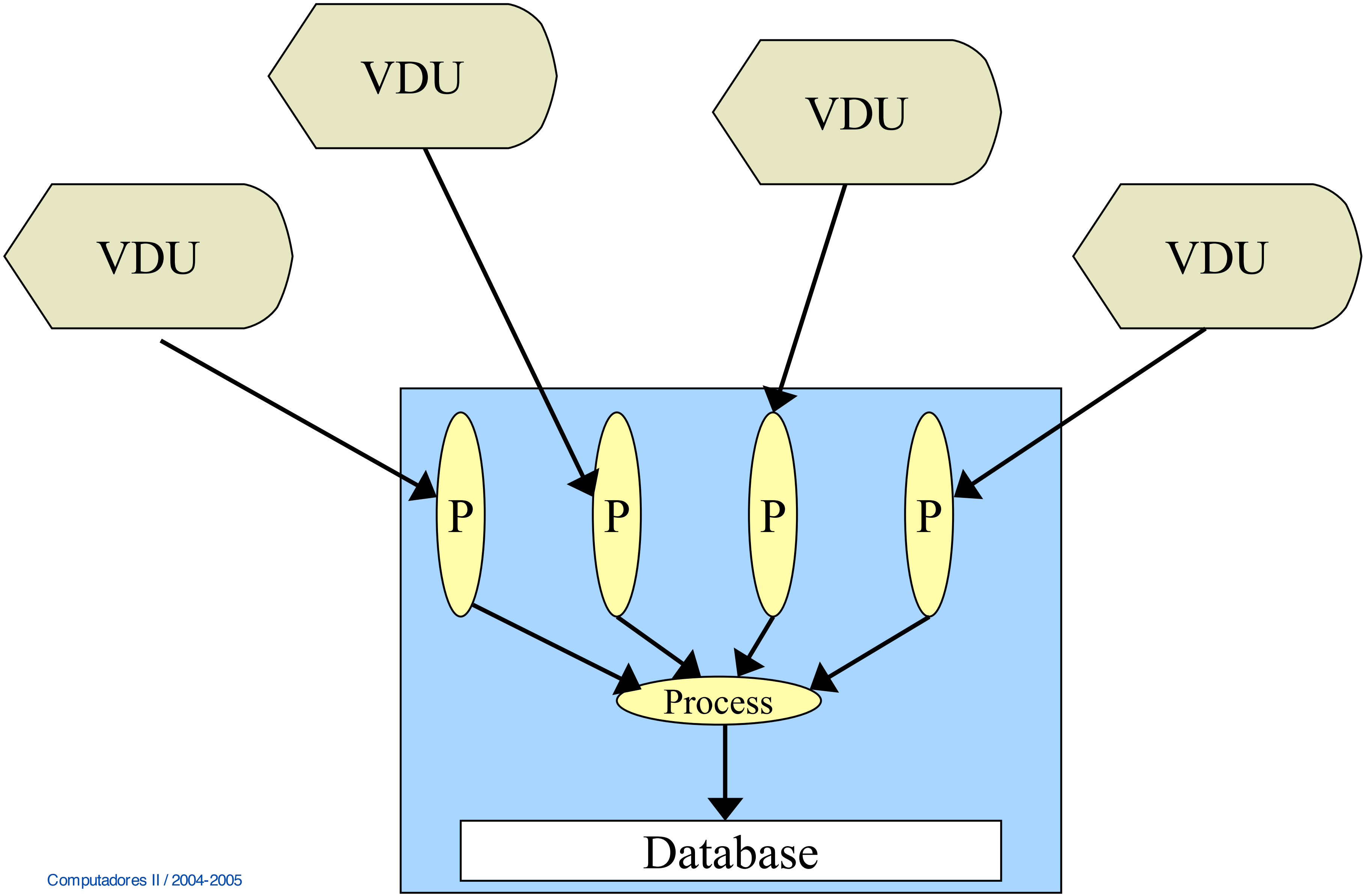
- Virtually all real-time systems are inherently concurrent
  - Physical devices operate in parallel in the real world

- This is, perhaps, the main reason to use concurrency in control systems

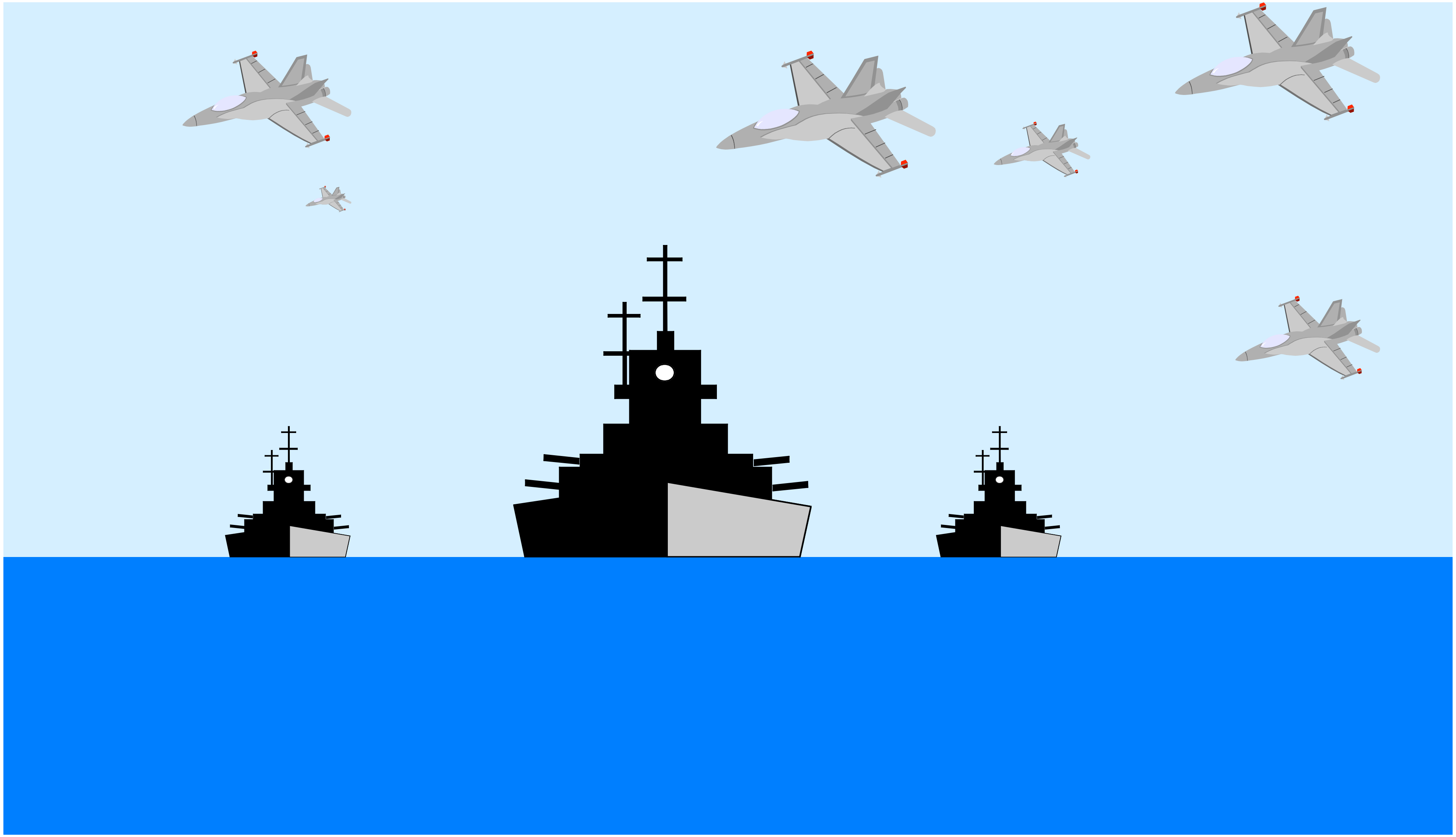
Process Control



Airline Reservation System



Air Traffic Control



Why we need it

- The alternative is to use **sequential programming** techniques
  - The programmer must construct the system so that it involves the cyclic execution of a program sequence to handle the various concurrent activities
  - This complicates the programmer's already difficult task and involves him/her in considerations of structures which are irrelevant to the control of the activities in hand
  - The resulting programs will be more obscure and inelegant
  - It makes decomposition of the problem more complex
  - Parallel execution of the program on more than one processor will be much more difficult to achieve
  - The placement of code to deal with faults is more problematic

Terminology

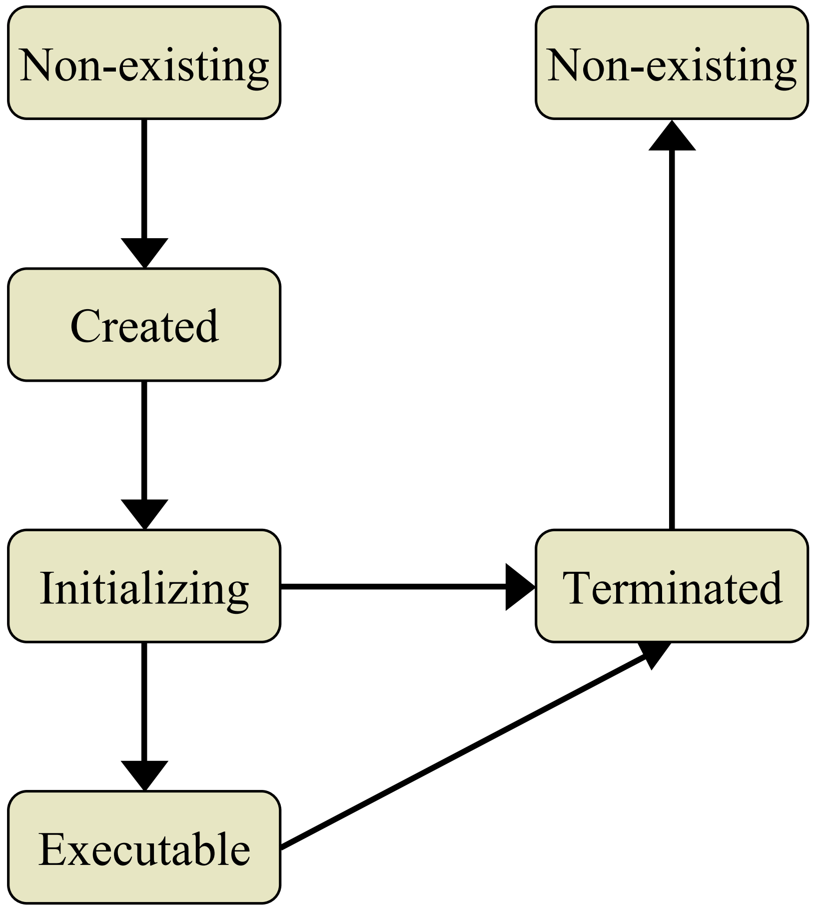
- A **concurrent program** is a collection of autonomous sequential **processes**,
- Processes execute (logically) in **parallel**
- Each process has a single **thread of control**

Implementation

The actual implementation (i.e. execution) of a collection of processes usually takes one of three forms:

- **Multiprogramming**
  - processes multiplex their executions on a single processor
- **Multiprocessing**
  - processes multiplex their executions on a multiprocessor system where there is access to shared memory
- **Distributed Processing**
  - processes multiplex their executions on several processors which do not share memory

Process States



Run-Time Support System

- To execute a concurrent program a **Run-time Support System is Necessary** (RTSS)

- The RTSS handles the execution (multiplexing) of the processes in the processors
- An RTSS has many of the properties of the scheduler in an operating system, and sits logically between the hardware and the application software.

RTSS Structures

- A software structure programmed as part of the application.
  - This is the approach adopted in Modula-2.
- A standard software system linked to the program object code by the compiler.
  - This is normally the structure with Ada programs.
- A separate platform (virtual machine) that executes applications.
  - This is the Java approach
- A hardware structure microcoded into the processor for efficiency.
  - An occam2 program running on the transputer has such a run-time system.
  - The aJile Java processor is another example.

Processes and Threads

- All operating systems provide **processes/tasks**
- Processes execute in their **own virtual machine** (VM) to avoid interference from other processes
- Recent OSs provide mechanisms for creating **threads** within the same virtual machine; threads are sometimes provided transparently to the OS
- Threads have unrestricted access to their VM
- The programmer and the language must provide the protection from interference
- Long debate over whether language should define concurrency or leave it up to the OS:
  - Ada and Java provide concurrency
  - C, C++ do not (rely on OS for that)

CP Ideas

CP Allow

- The expression of concurrent execution through the notion of process
- Process synchronization
- Inter-process communication

Processes may be

- Independent
- Cooperating
- Competing

Processes differ in

- Structure — static, dynamic
- Level — nested, flat

Concurrent Execution

Language	Structure	Level
Concurrent Pascal	static	flat
occam2	static	nested
Modula	dynamic	flat
Ada	dynamic	nested
C/POSIX	dynamic	flat
Java	dynamic	nested

Concurrent Execution

- Granularity
  - coarse (Ada, POSIX processes/threads, Java)
  - fine (occam2)
- Initialization — parameter passing, IPC
- Termination
  - completion of execution of the process body;
  - suicide, by execution of a **self-terminate** statement;
  - abortion, through the explicit action of another process;
  - occurrence of an untrapped error condition;
  - never: processes are assumed to be non-terminating loops;
  - when no longer needed.

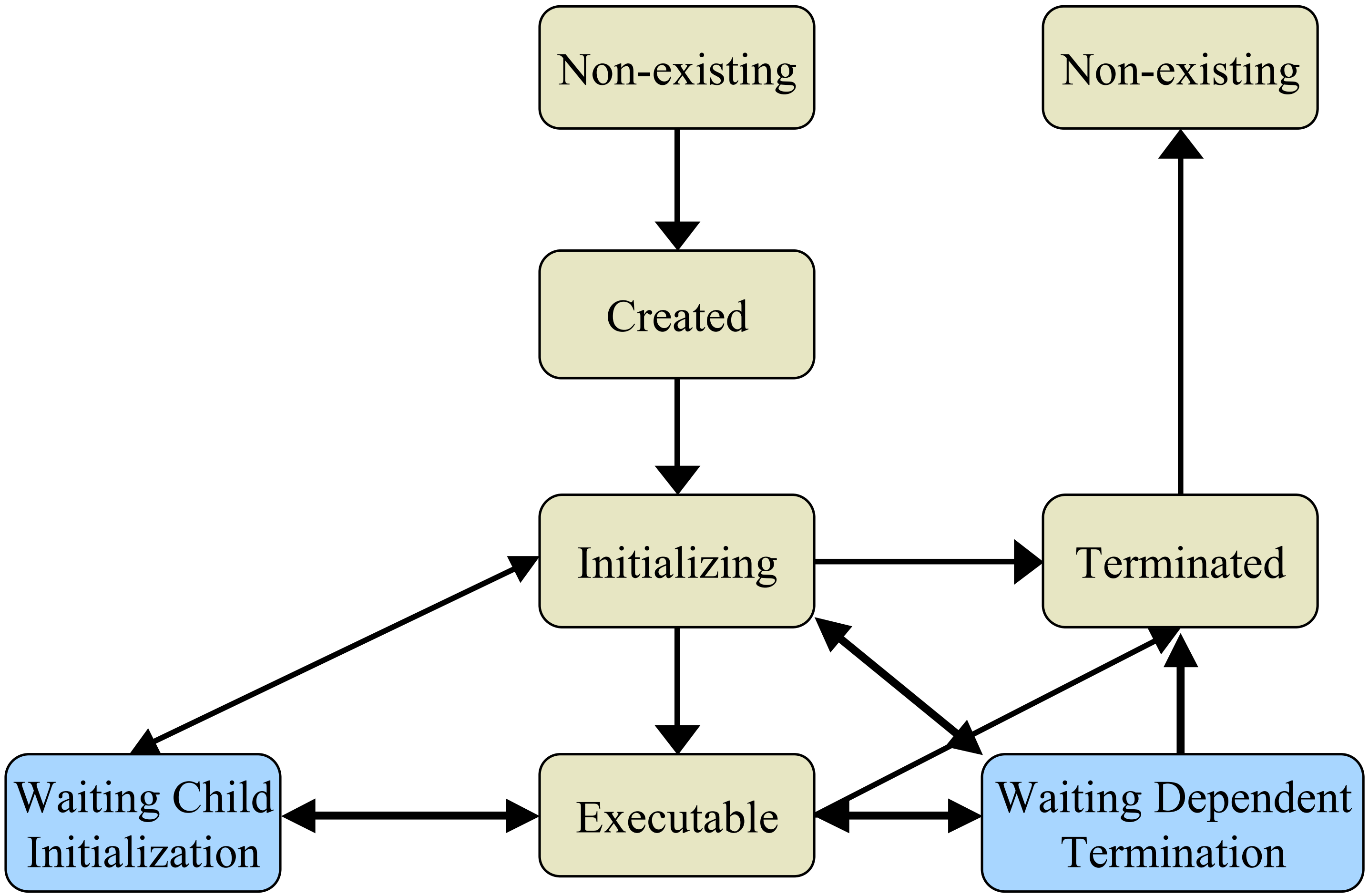
Process Hierarchies

- Hierarchies of processes can be created and inter-process relationships formed
- For any process, a distinction can be made between the process (or block) that created it and the process (or block) which is affected by its termination
- The former relationship is know as **parent/child** and has the attribute that the parent may be delayed while the child is being created and initialized
- The latter relationship is termed **guardian/dependent**. A process may be dependent on the guardian process itself or on an inner block of the guardian
- The guardian is not allowed to exit from a block until all dependent processes of that block have terminated

Nested Processes

- A guardian cannot terminate until **all** its dependents have terminated
- A program cannot terminate until **all** its processes have terminated
- A parent of a process may also be its guardian (e.g. with languages that allow only static process structures)
- With dynamic nested process structures, the parent and the guardian may or may not be identical

Process States



Processes and Objects

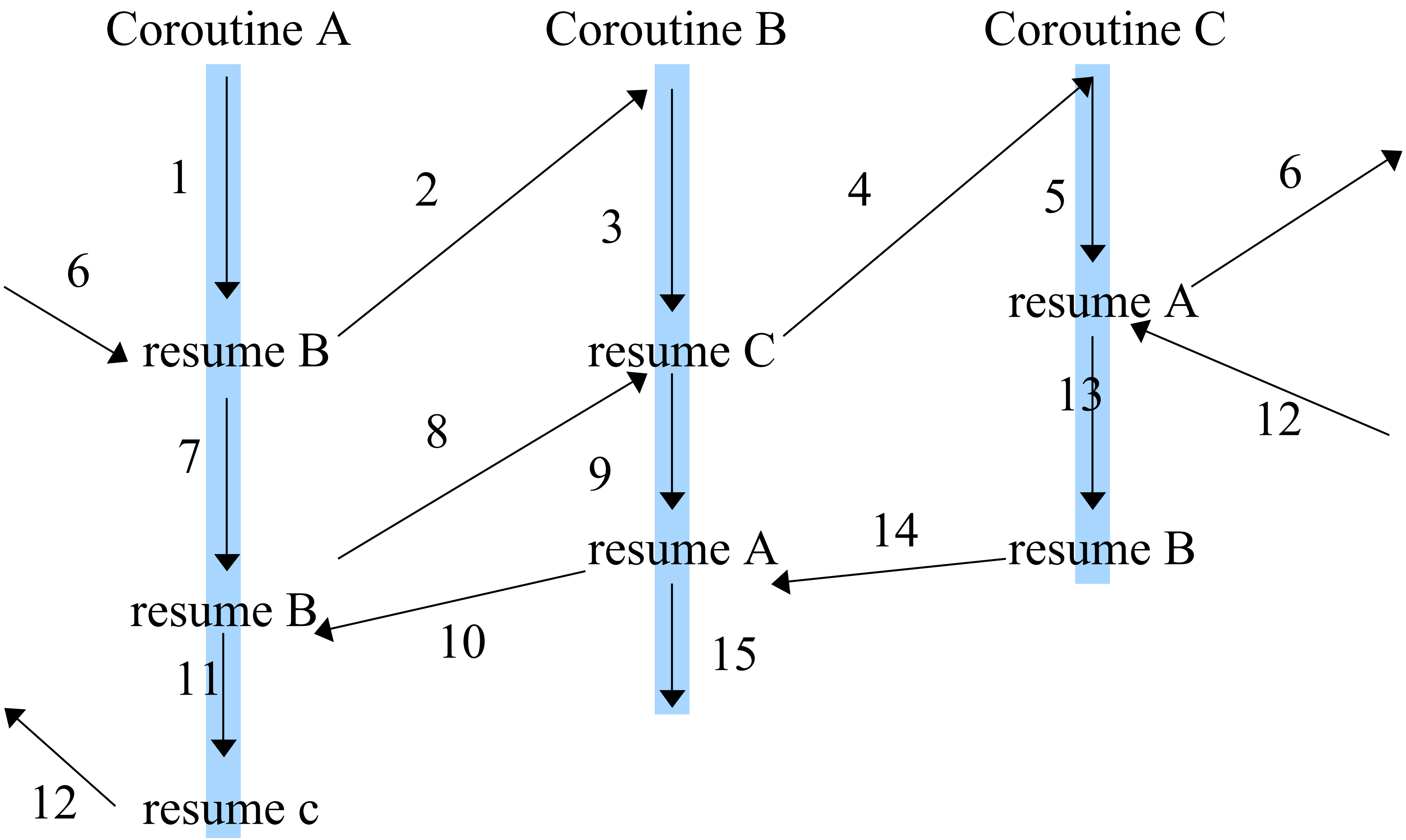
- **Active** objects
  - undertake spontaneous actions
- **Reactive** objects
  - only perform actions when invoked
- **Resources**
  - reactive but can control order of actions
- **Passive**
  - reactive, but no control over order
- **Protected** resources
  - passive resource controller
- **Server**
  - active resource controller

Process Representation

- Many program constructs to express concurrence

- Coroutines
- Fork and Join
- Cobegin
- Explicit Process Declaration

Coroutine Flow Control



Note

- No return statement — only a resume statement
- The value of the data local to the coroutine persist between successive calls
- The execution of a coroutine is supended as control leaves it, only to carry on where it left off when it resumed

Do coroutines express true parallelism?

Fork and Join

- The fork specifies that a designated routine should start executing concurrently with the invoker
- Join allows the invoker to wait for the completion of the invoked routine

```
function F return is ...;
procedure P;
     ...
     C:= fork F;
     ...
     J:= join C;
     ...
end P;
```

- After the fork, P and F will be executing concurrently. At the point of the join, P will wait until the F has finished (if it has not already done so)
- Fork and join notation can be found in Mesa and UNIX/POSIX

UNIX Fork Example

```
for (I=0; I!=10; I++) {
  pid[I] = fork();
}
wait . . .
```

How many processes are created?

Cobegin

- The cobegin (or parbegin or par) is a structured way of denoting the concurrent execution of a collection of statements:

```
cobegin
  S1;
  S2;
  .
  .
  Sn
coend
```

- S1, S2 etc, execute concurrently
- The statement terminates when S1, S2 etc have terminated
- Each Si may be any statement allowed within the language
- Cobegin can be found in Edison and occam2.

Explicit Process Declaration

- The structure of a program can be made clearer if routines state whether they will be executed concurrently
- Note that this does not say when they will execute

```
task body Process is
begin
   . . .
end;
```

- Languages that support explicit process declaration may have explicit or implicit process/task creation

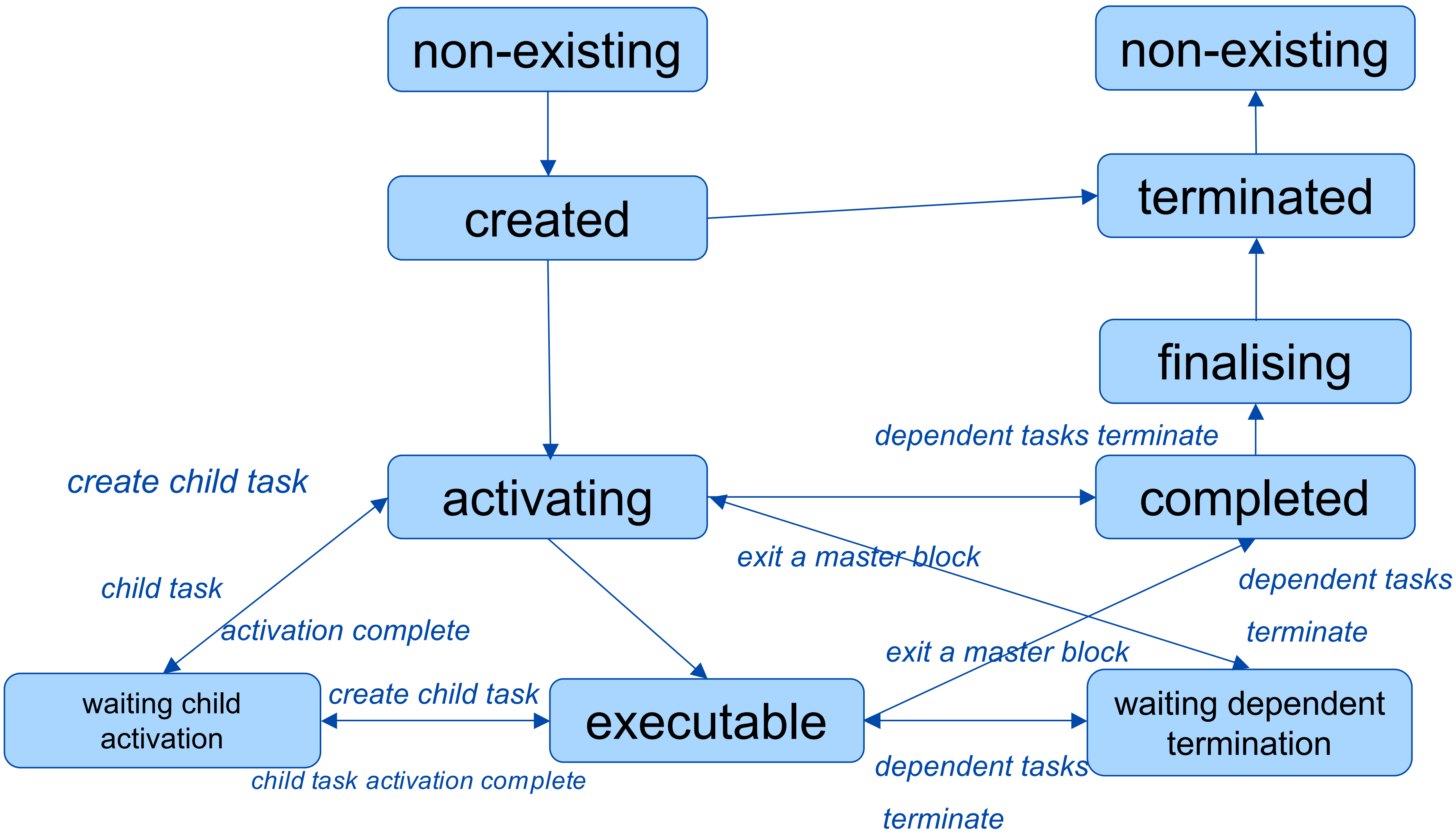
Tasks and Ada

- The unit of concurrency in Ada is called a **task**
- Tasks must be explicitly declared, there is no fork/join statement, cobegin/coend, etc.
- Tasks may be declared at any program level; they are created implicitly upon entry to the scope of their declaration or via the action of an allocator
- Tasks may communicate and synchronise via a variety of mechanisms:
  - rendezvous (a form of synchronised message passing),
  - protected units (a form of monitor/conditional critical region),
  - and shared variables

Task Types and Task Objects

- A task can be declared as a type or as a single instance (anonymous type)
- A task type consists of a specification and a body
- The specification contains
  - the type name
  - an optional discriminant part which defines the parameters that can be passed to instances of the task type at their creation time
  - a visible part which defines any entries and representation clauses
  - a private part which defines any hidden entries and representation clauses

Task States in Ada



Concurrent Execution in POSIX

- Two main mechanisms:
  - Coarse grained: **fork**
  - Fine grained: **pthreads**.
- **fork** creates a new process
- **pthreads** are an extension to POSIX to allow threads to be created
- All threads have attributes (e.g. stack size) that can be manipulated
- Threads are created using an appropriate attribute object
- Threads can communicate using POSIX IPC mechanisms

Typical C POSIX interface

```
typedef ... pthread_t;  /* details not defined */
typedef ... pthread_attr_t;

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);

int pthread_attr_setstacksize(..);
int pthread_attr_getstacksize(..);

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
               void *(*start_routine)(void *), void *arg);
  /* create thread and call the start_routine with the argument */

int pthread_join(pthread_t thread, void **value_ptr);
int pthread_exit(void *value_ptr);
  /* terminate the calling thread and make the pointer value_ptr
     available to any joining thread */
```

Concurrency in Java

- Java has a predefined class **`java.lang.Thread`** which provides the mechanism by which threads (processes) are created.
- However to avoid all threads having to be child classes of `Thread`, it also uses a standard interface

```
public interface Runnable {
  public abstract void run();
}
```

- Hence, any class which wishes to express concurrent execution must implement this interface and provide the **`run`** method

Java Thread Class

```
public class Thread extends Object implements Runnable
{
  public Thread();
  public Thread(Runnable target);

  public void run();
  public native synchronized void start();
  // throws IllegalThreadStateException
  public static Thread currentThread();
  public final void join() throws InterruptedException;
  public final native boolean isAlive();
  public void destroy();
  // throws SecurityException;
  public final void stop();
  // throws SecurityException

...
}
```

Robot Arm Example

```
public class UserInterface
{
  public int newSetting (int Dim) { ... }
  ...
}

public class Arm
{
  public void move(int dim, int pos) { ... }
}

UserInterface UI = new UserInterface();

Arm Robot = new Arm();
```

Robot Arm Example

```
public class Control extends Thread
{
  private int dim;

  public Control(int Dimension) // constructor
  {
    super();
    dim = Dimension;
  }

  public void run()
  {
    int position = 0;
    int setting;

    while(true)
    {
       Robot.move(dim, position);
       setting = UI.newSetting(dim);
       position = position + setting;
    }
  }
}
```

Robot Arm Example

```
final int xPlane = 0;  // final indicates a constant
final int yPlane = 1;
final int zPlane = 2;

Control C1 = new Control(xPlane);
Control C2 = new Control(yPlane);
Control C3 = new Control(zPlane);

C1.start();
C2.start();
C3.start();
```

Alternative Robot Control

```
public class Control implements Runnable
{
  private int dim;

  public Control(int Dimension)  // constructor
  {
    dim = Dimension;
  }

  public void run()
  {
    int position = 0;
    int setting;

    while(true)
    {
       Robot.move(dim, position);
       setting = UI.newSetting(dim);
       position = position + setting;
    }
  }
}
```

Alternative Robot Control

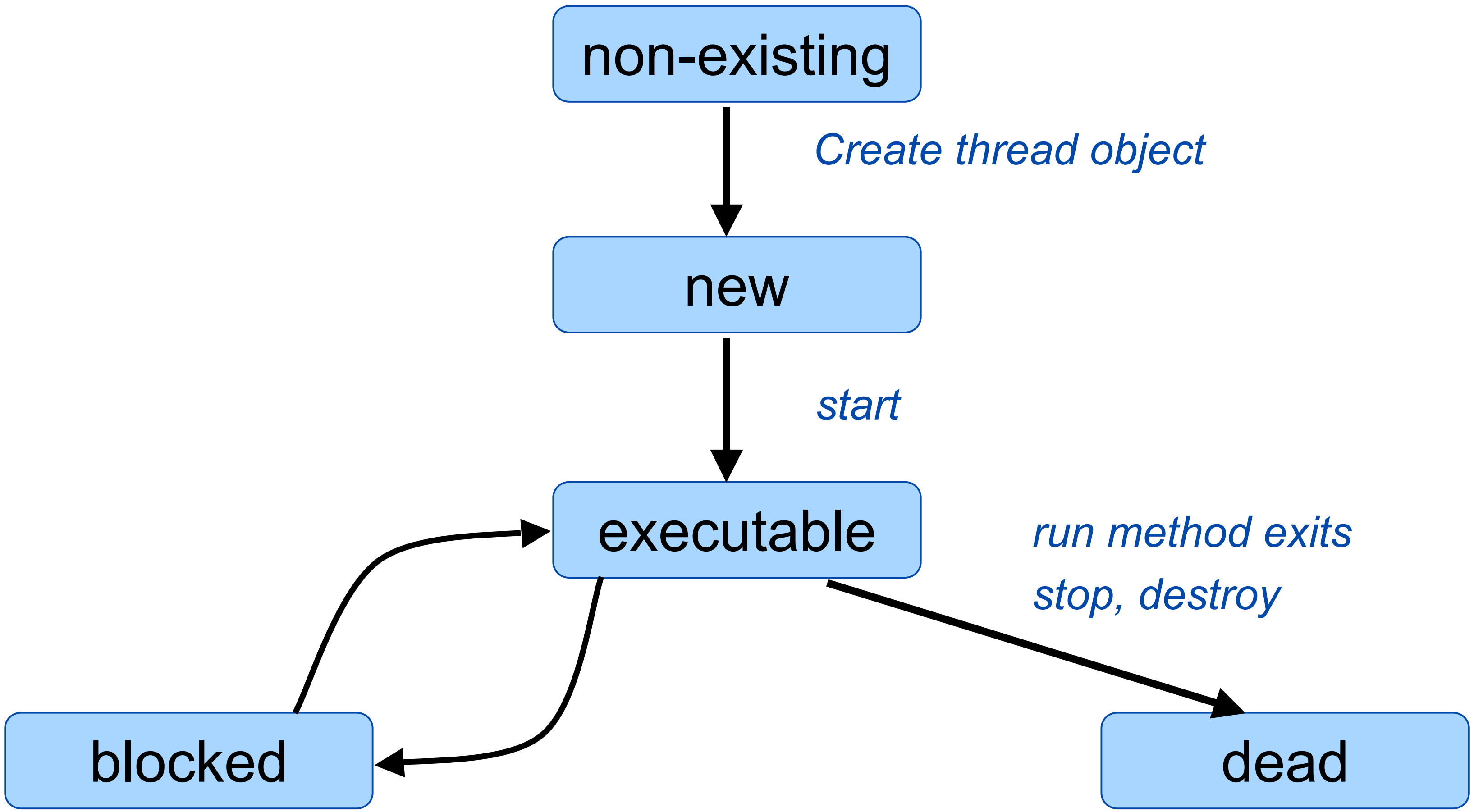
```
final int xPlane = 0;
final int yPlane = 1;
final int zPlane = 2;

Control C1 = new Control(xPlane); // no thread created yet
Control C2 = new Control(yPlane);
Control C3 = new Control(zPlane);

// constructors passed a Runnable interface and threads created
Thread X = new Thread(C1);
Thread Y = new Thread(C2);
Thread Z = new Thread(C2);

X.start(); // thread started
Y.start();
Z.start();
```

Java Thread States



Points about Java Threads

- Java allows dynamic thread creation
- Java allows arbitrary data to be passed as parameters during construction
- Java allows thread hierarchies and thread groups to be created but there is no master or guardian concept
- Java relies on garbage collection to clean up objects which can no longer be accessed
- The main program in Java terminates when all its user threads have terminated (see later)
- One thread can wait for another thread (the target) to terminate by issuing the **`join`** method call on the target's thread object.
- The **`isAlive`** method allows a thread to determine if another thread has terminated

A Thread Terminates:

- when it completes execution of its **`run`** method either normally or as the result of an unhandled exception
- via its **`stop`** method — the **`run`** method is stopped and the thread class cleans up before terminating the thread (releases locks and executes any finally clauses)
  - the thread object is now eligible for garbage collection.
  - if a **`Throwable`** object is passed as a parameter to **`stop`**, then this exception is thrown in the target thread; this allows the run method to exit more gracefully and cleanup after itself
  - **`stop`** is inherently unsafe as it releases locks on objects and can leave those objects in inconsistent states; the method is now deemed obsolete (deprecated) and should not be used
- via its **`destroy`** method — destroy terminates the thread without any cleanup (not implemented in Sun's JVM)

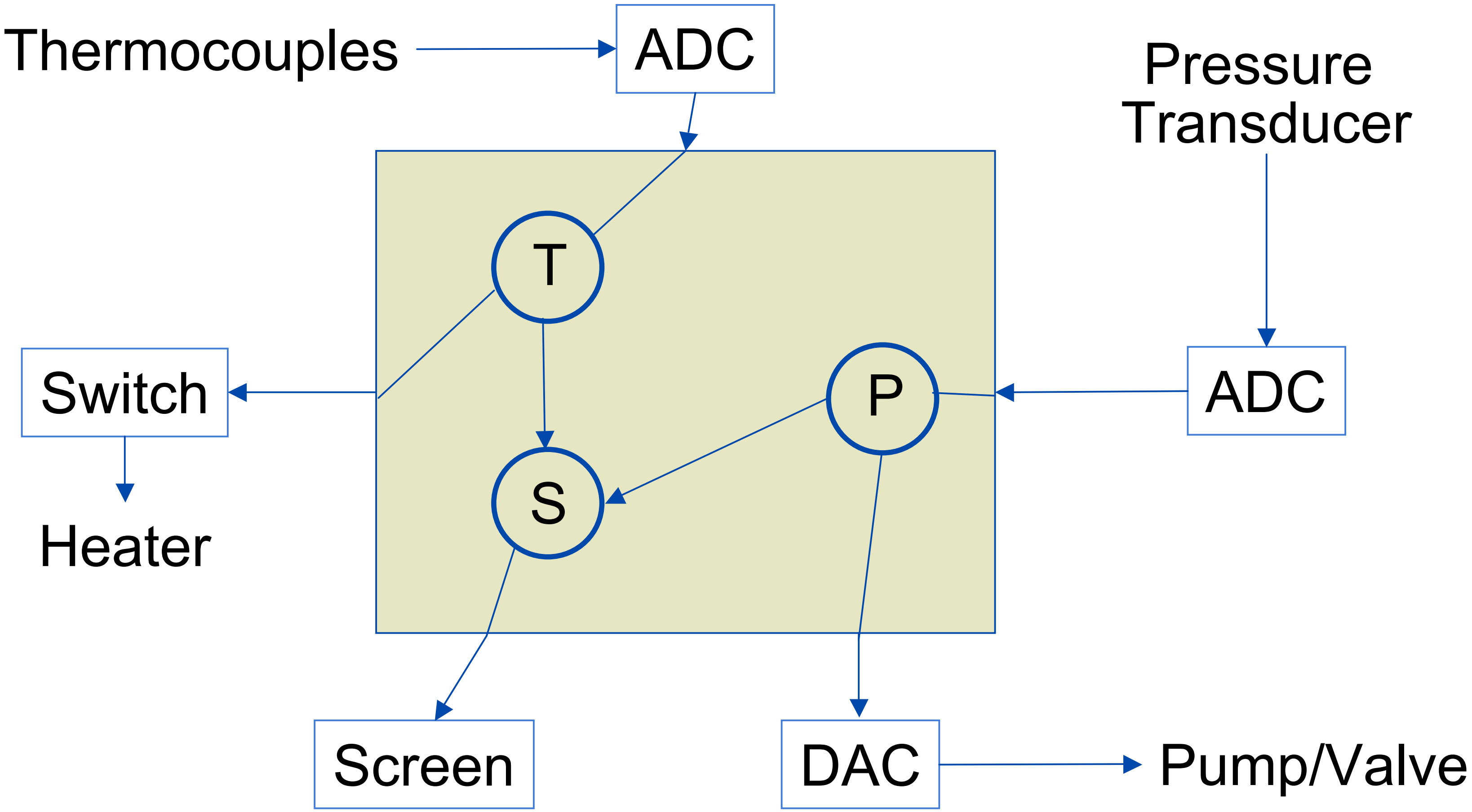
Daemon Threads

- Java threads can be of two types:
  - **user** threads
  - **daemon** threads
- Daemon threads are those threads which provide general services and typically never terminate
- The **`setDaemon`** method must be called before the thread is started to mark it as daemon
- When all user threads have terminated, daemon threads can also be terminated and the main program terminates

Thread Exceptions

- The `IllegalThreadStateException` is thrown when:
  - the `start` method is called and the thread has already been started
  - the `setDaemon` method has been called and the thread has already been started
- The `SecurityException` is thrown by the security manager when:
  - a `stop` or `destroy` method has been called on a thread for which the caller does not have the correct permissions for the operation requested
- The `InterruptException` is thrown if a thread which has issued a `join` method is woken up by the thread being interrupted rather than the target thread terminating

A Simple Embedded System



- Overall objective is to keep the temperature and pressure of some chemical process within well-defined limits

Possible Software Architectures

- A **single sequential program** is used which ignores the logical concurrency of T, P and S; no operating system support is required
- T, P and S are written in a sequential programming language (either as separate programs or distinct procedures in the same program) and **operating system primitives** are used for program/process creation and interaction
- A **single concurrent program** is used which retains the logical structure of T, P and S; no operating system support is required although a run-time support system is needed

Which one is the best approach?

Disadvantages of Single Sequential

- Temperature and pressure readings must be taken at the same rate (the use of counters and if statements may improve the situation)
- But may still be necessary to split up the conversion procedures, and interleave their actions so as to meet a required balance of work
- While waiting to read a temperature no attention can be given to pressure (and viceversa)
- Moreover, a system failure that results in, say, control never returning from the temperature read, then in addition to this problem no further calls to read the pressure would be taken

Advantages of Concurrency

- Controller tasks execute concurrently and each contains an indefinite loop within which the control cycle is defined
- While one task is suspended waiting for a read the other may be executing; if they are both suspended a busy loop is not executed
- The logic of the application is reflected in the code; the inherent parallelism of the domain is represented by concurrently executing tasks in the program

Disadvantages

- Both tasks send data to the screen, but the screen is a resource that can only sensibly be accessed by one process at a time
- A third entity is required. This has transposed the problem from that of concurrent access to a non-concurrent resource to one of resource control
- It is necessary for controller tasks to pass data to the screen resource
- The screen must ensure mutual exclusion
- The whole approach requires a run-time support system

OS vs Language Concurrency

- Should concurrency be in a language or in the OS?
- Arguments for concurrency in the languages:
  - It leads to more readable and maintainable programs
  - There are many different types of OSs; the language approach makes the program more portable
  - An embedded computer may not have any resident OS
- Arguments against concurrency in a language:
  - It is easier to compose programs from different languages if they all use the same OS model
  - It may be difficult to implement a language's model of concurrency efficiently on top of an OS's model
  - OS standards (POSIX, W32) are available
- The Ada/Java philosophy is that the advantages outweigh the disadvantages

Summary

- The application **domains** of most real-time systems are inherently **parallel**
- The inclusion of the notion of **process** within a real-time programming language makes an enormous difference to the **expressive power** and ease of use of the language
- Without concurrency the software must be constructed as a single control loop
- The structure of this loop cannot retain the **logical distinction** between systems components. It is particularly difficult to give process-oriented timing and reliability requirements without the notion of a process being visible in the code

Summary Continued

- The use of a concurrent programming language is not without its **costs**. In particular, it becomes necessary to use a **run-time support system** to manage the execution of the system processes
- The behaviour of a process is best described in terms of **process states**
  - non-existing
  - created
  - initialized
  - executable
  - waiting dependent termination
  - waiting child initialization
  - terminated