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

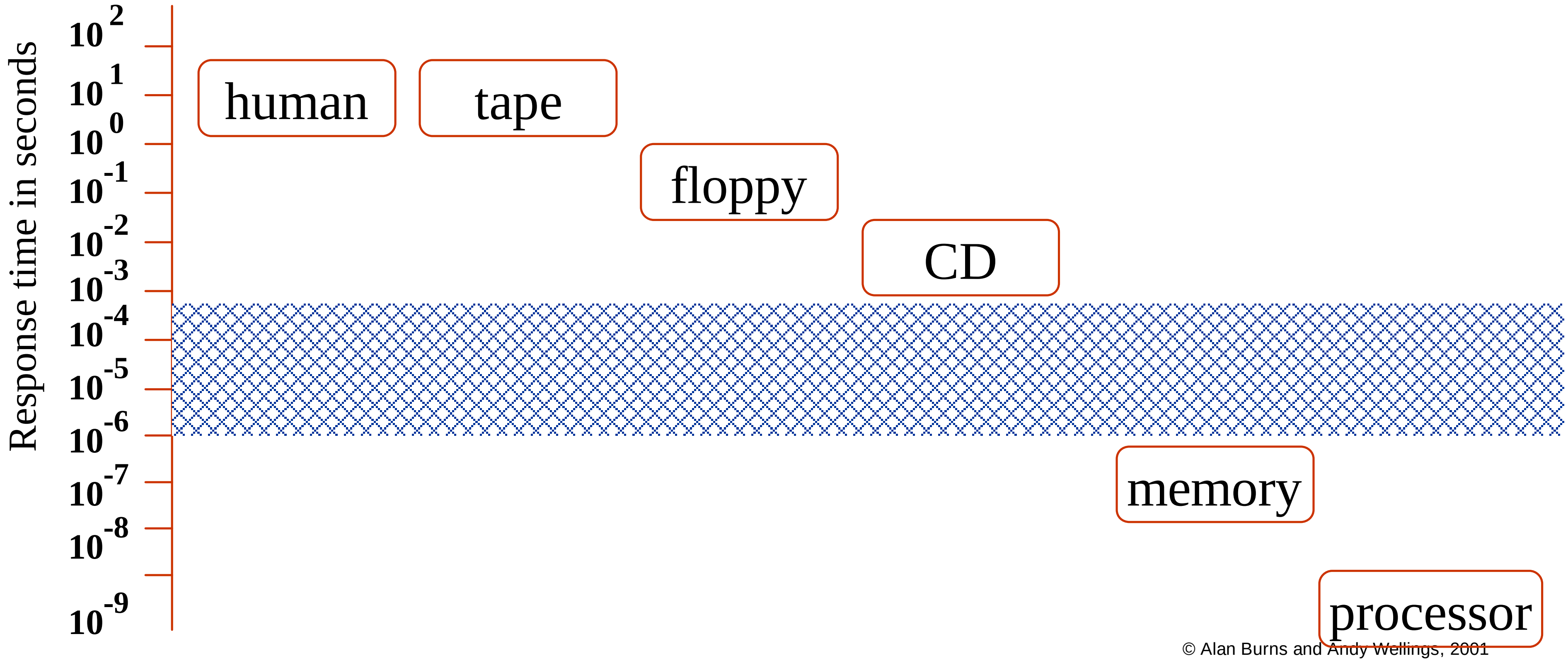
- To illustrate the requirements for concurrent programming
- To demonstrate the variety of models for creating processes
- To show how processes are created in Ada (tasks), POSIX/C (processes and threads) and Java (threads)
- To lay the foundations for studying inter-process communication

Concurrent Programming

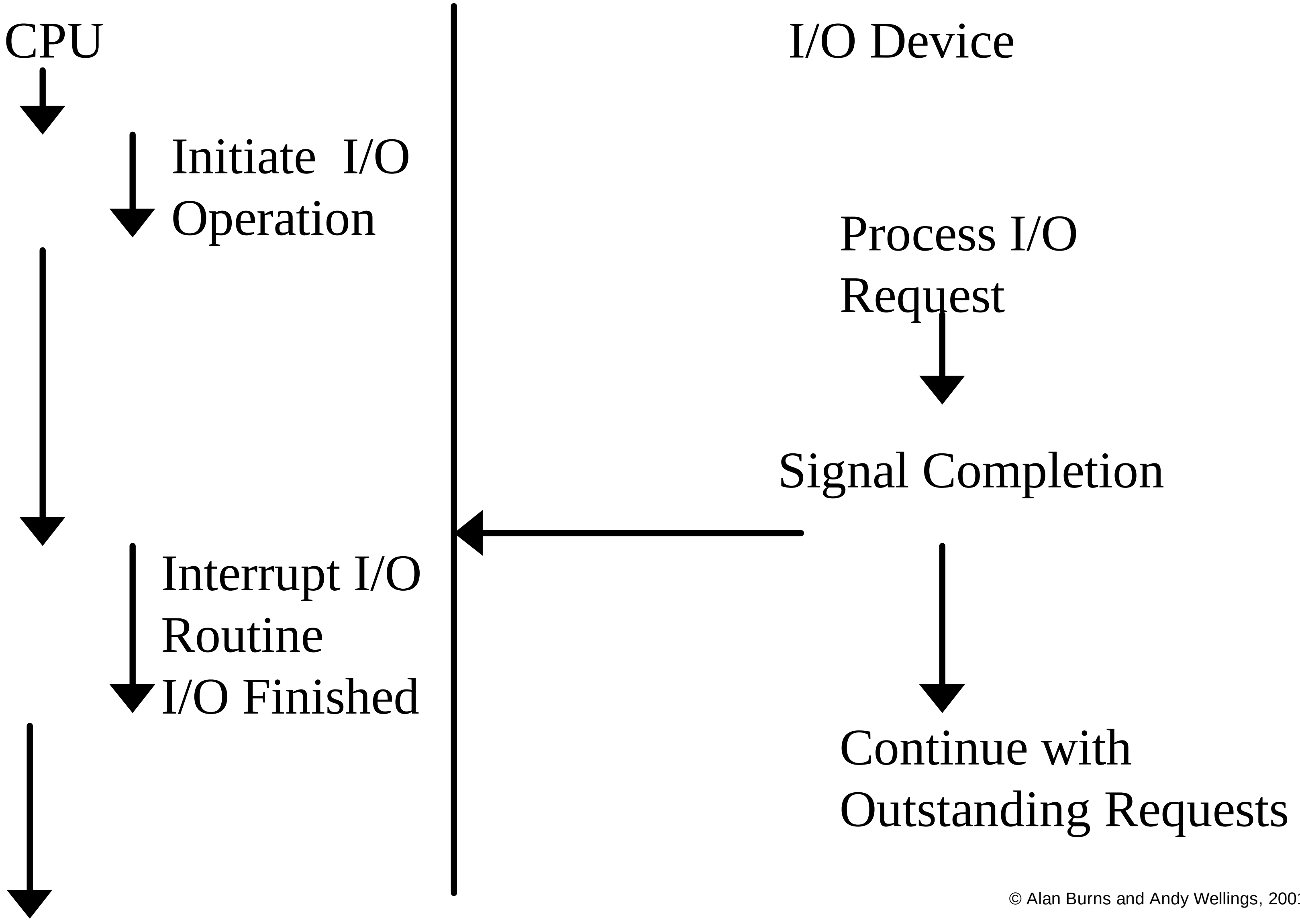
- The name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems
- Implementation of parallelism is a topic in computer systems (hardware and software) that is essentially independent of concurrent programming
- 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



Parallelism Between CPU and I/O Devices

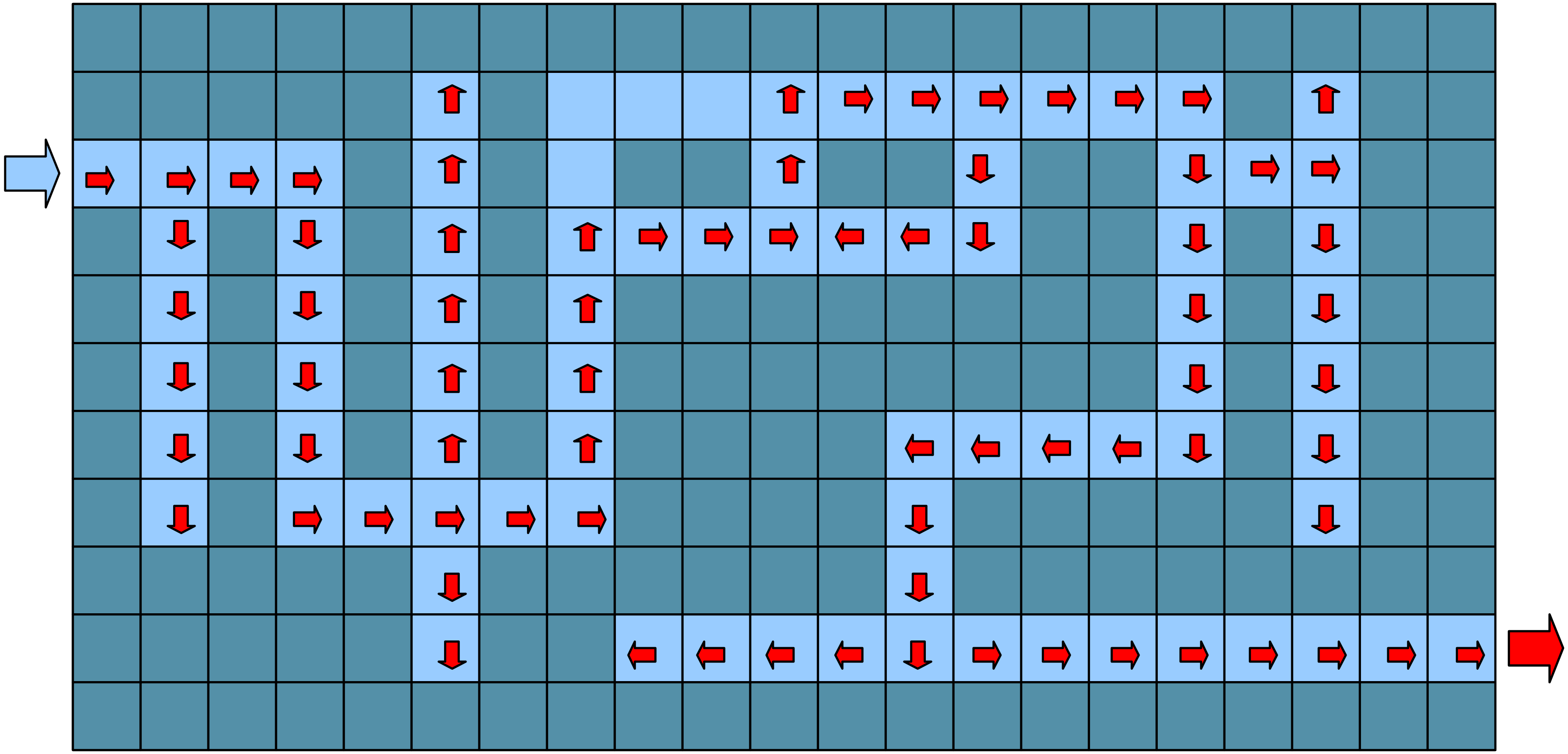


Why we need it

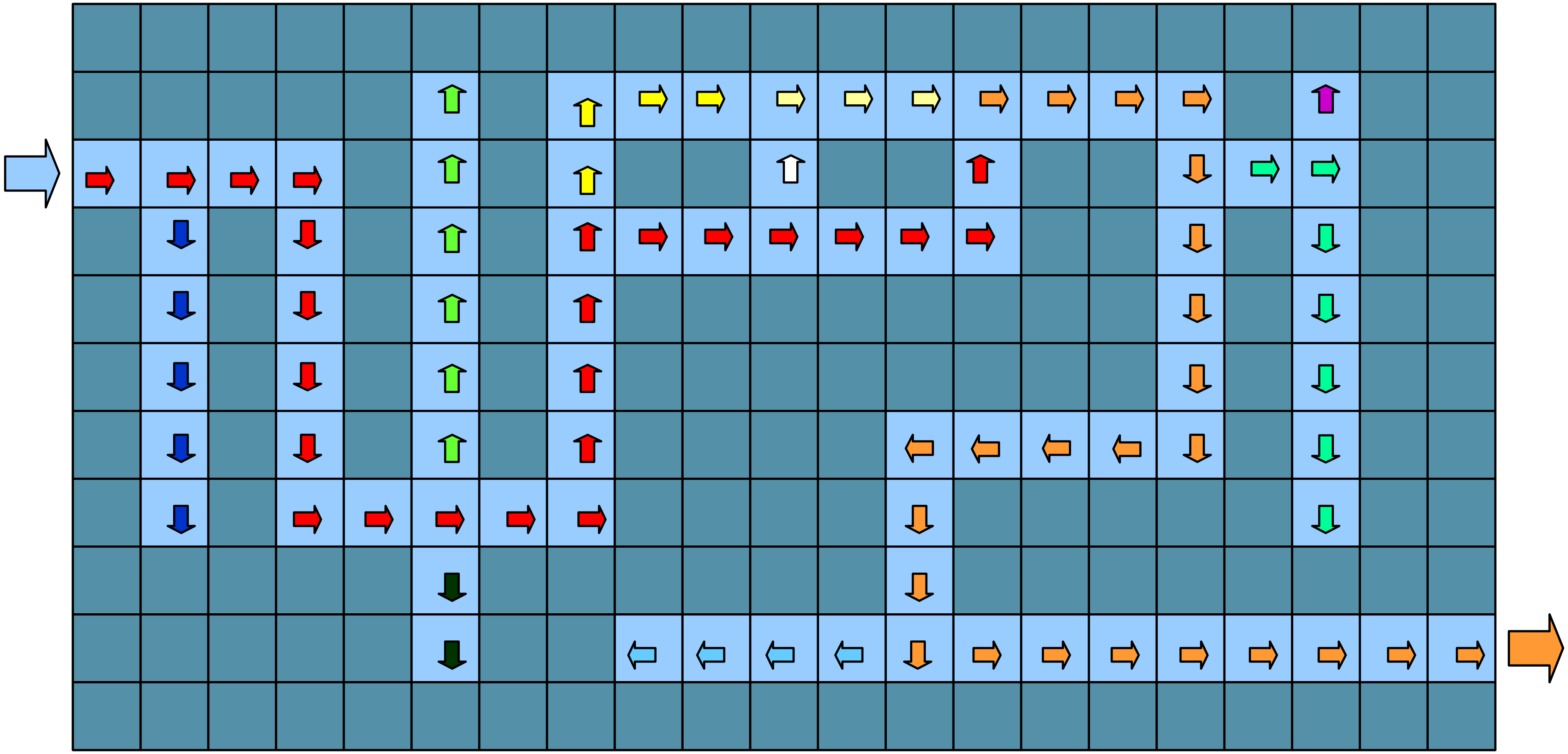
- To allow the expression of potential parallelism so that more than one computer can be used to solve the problem

- Consider trying to find the way through a maze

Sequential Maze Search



Concurrent Maze Search

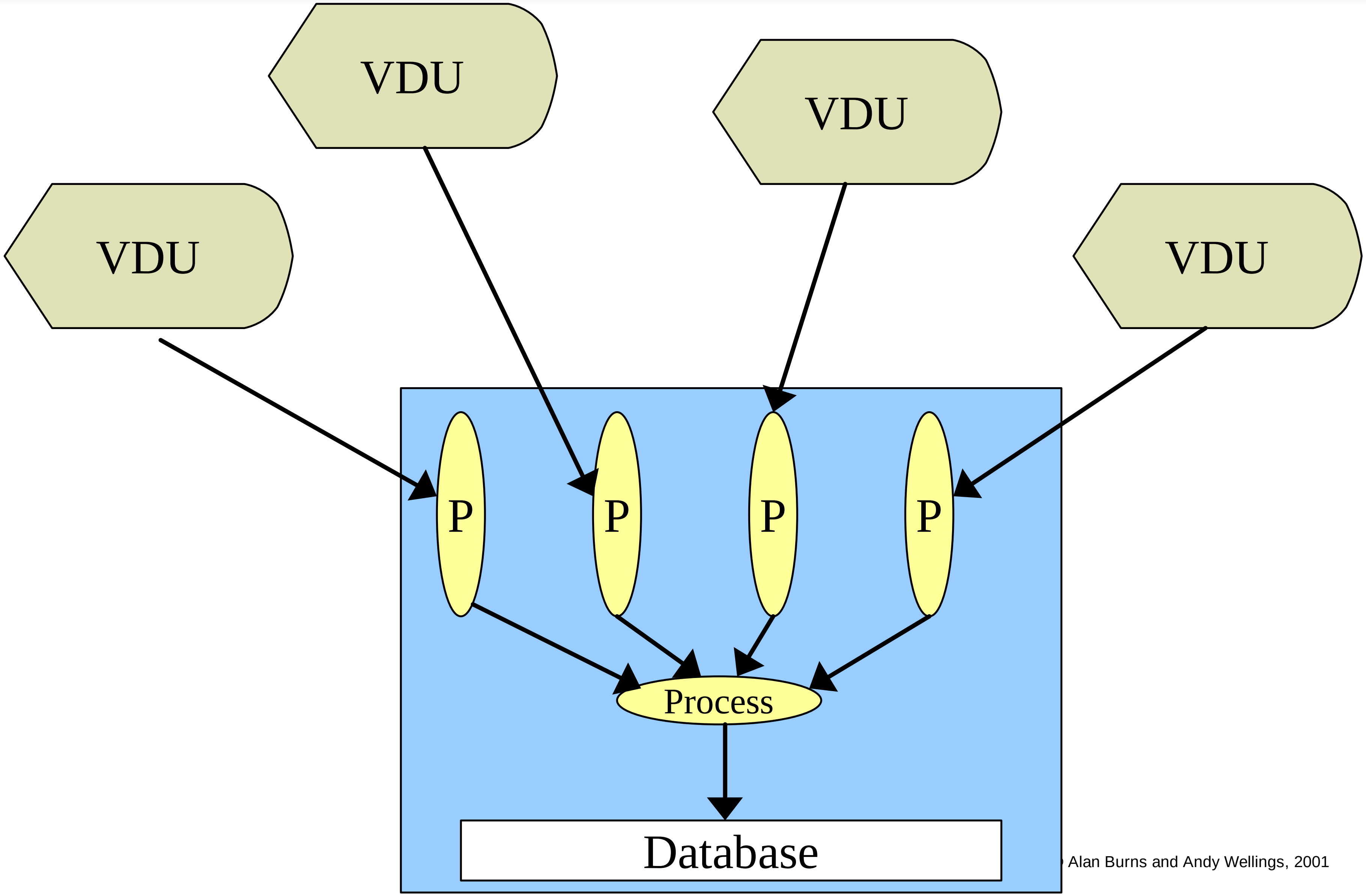


Why we need it

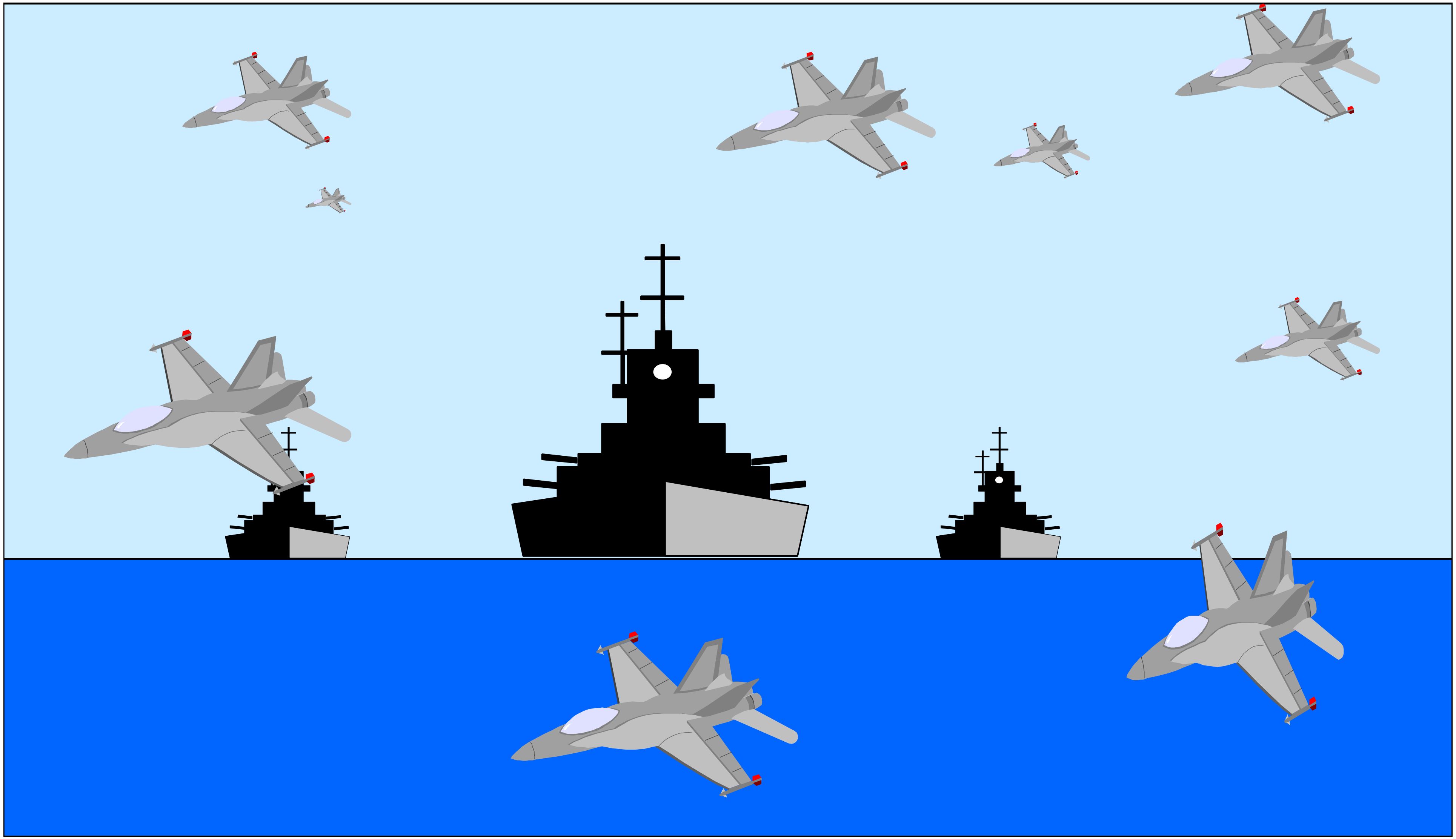
- To model the parallelism in the real world

- Virtually all real-time systems are inherently concurrent — devices operate in parallel in the real world
- This is, perhaps, the main reason to use concurrency

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 a collection of autonomous sequential processes, executing (logically) in parallel
- Each process has a single thread of control
- 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

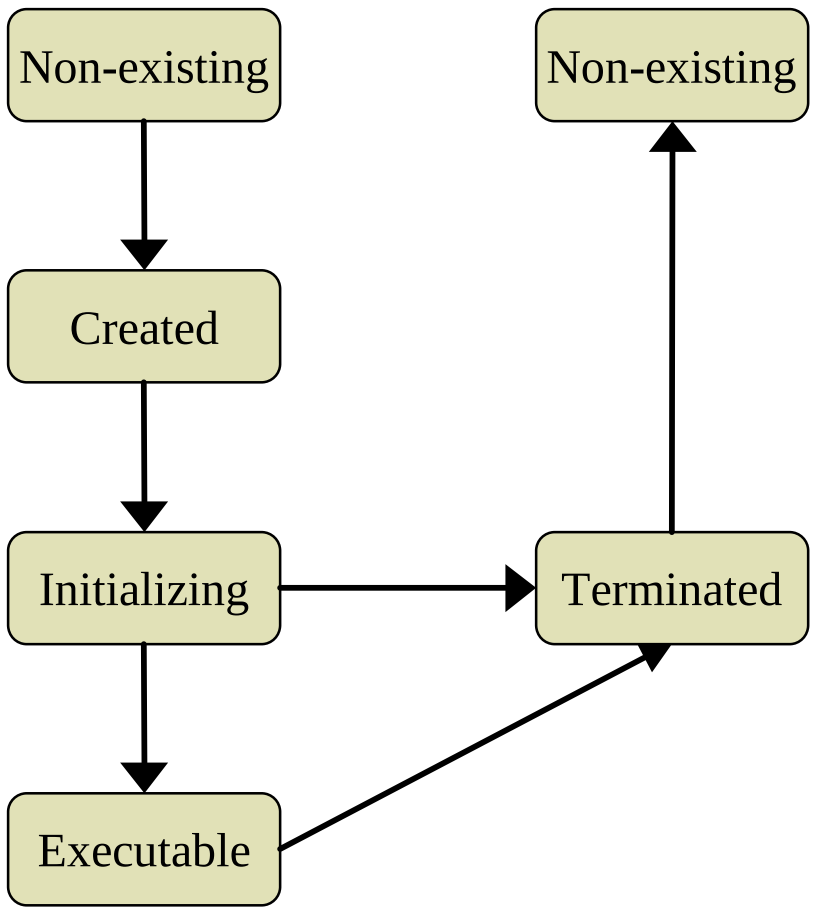
Multiprocessing

- 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

- An RTSS has many of the properties of the scheduler in an operating system, and sits logically between the hardware and the application software.
- In reality it may take one of a number of forms:
  - A software structure programmed as part of the application. This is the approach adopted in Modula-2.
  - A standard software system generated with the program object code by the compiler. This is normally the structure with Ada programs.
  - A hardware structure microcoded into the processor for efficiency. An occam2 program running on the transputer has such a run-time system.

Processes and Threads

- All operating systems provide processes
- 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 O.S.
  - Ada and Java provide concurrency
  - C, C++ do not

Concurrent Programming Constructs

Allow

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

Processes may be

- independent
- cooperating
- competing

Concurrent Execution

Processes differ in

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

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.

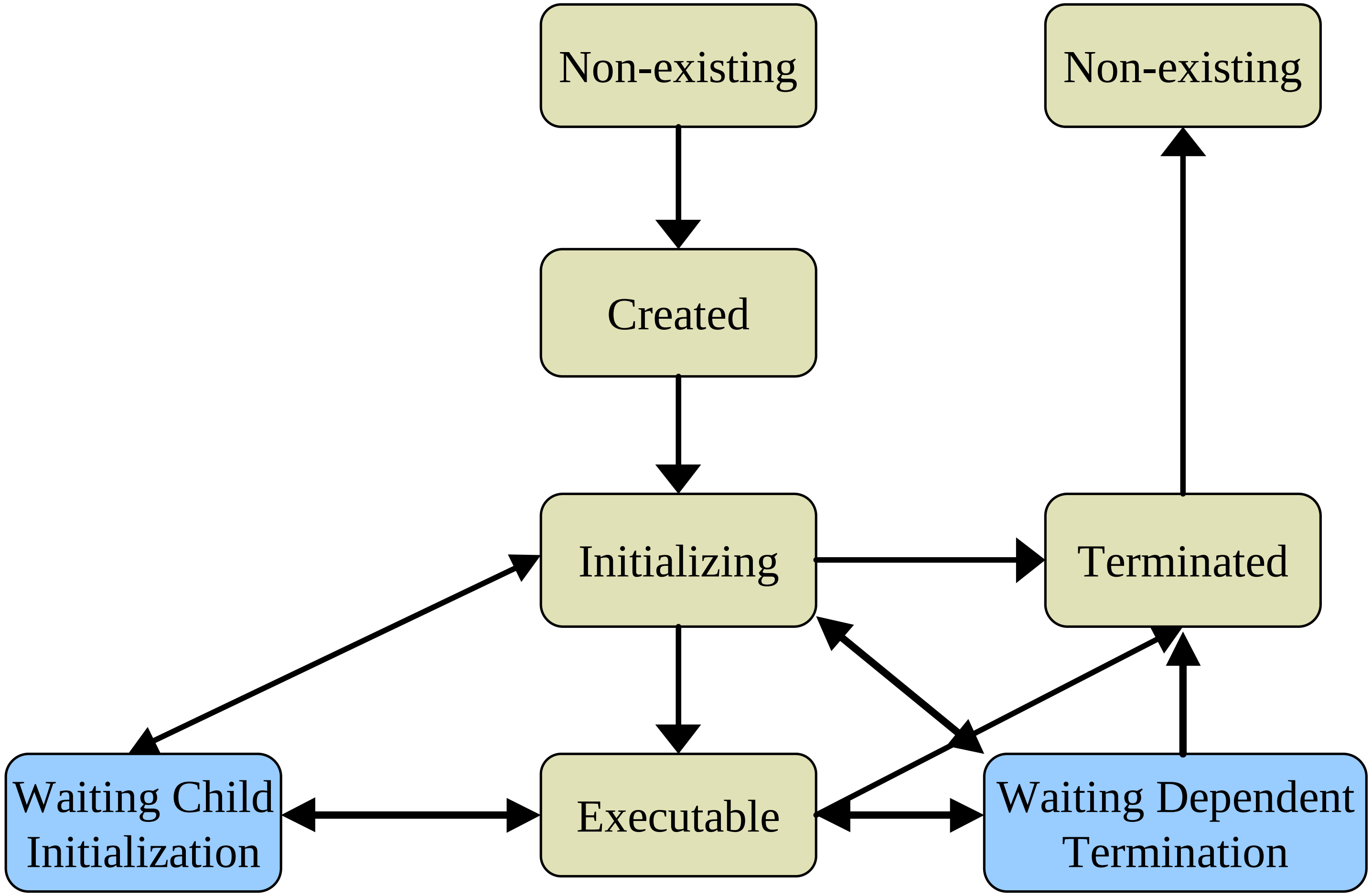
Nested Processes

- 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



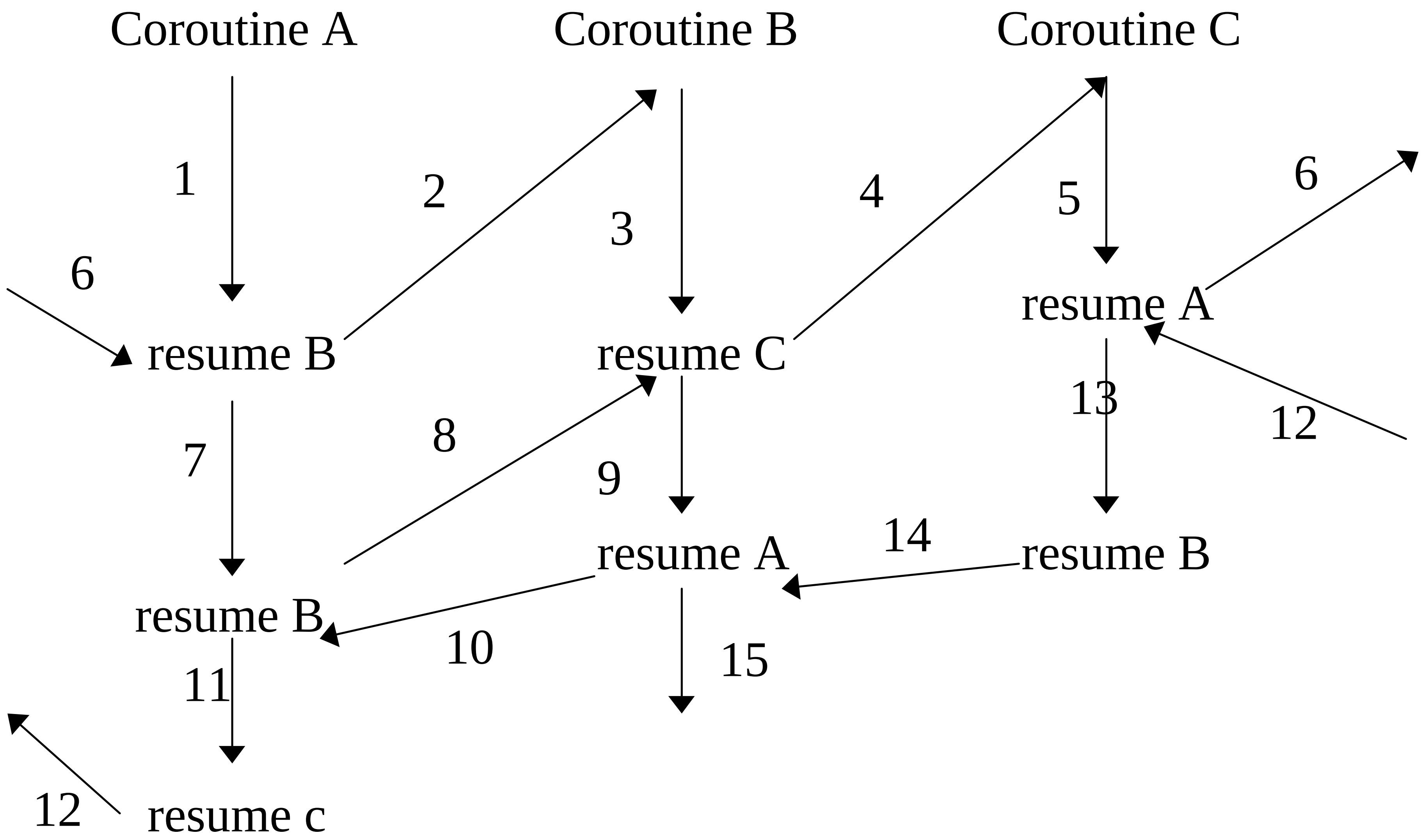
Processes and Objects

- 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 resource — passive resource controller
- Server — active resource controller

Process Representation

- 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 persit 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 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;
  S3;
  .
  .
  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/PAR 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

Example Task Structure

```
task type Server (Init : Parameter) is
  entry Service;
end Server;

task body Server is
begin
  ...
  accept Service do
    -- Sequence of statements;
  end Service;
  ...
end Server;
```

specification

body

Example Task Specifications

```
task type Controller;
```

this task type has no entries; no other tasks can communicate directly

```
task type Agent(Param : Integer);
```

this task type has no entries but task objects can be passed an integer parameter at their creation time

```
task type Garage_Attendant(
     Pump : Pump_Number := 1) is
  entry Serve_Leaded(G : Gallons);
  entry Serve_Unleaded(G : Gallons);
end Garage_Attendant;
```

objects will allow communication via two entries; the number of the pump to be served is passed at task creation time; if no value is passed a default of 1 is used

An Example

```
type User is (Andy, Neil, Alan);
task type Character_Count(Max : Integer := 100;
                          From : User := Andy);
task body Character_Count is
  use Ada.Text_IO, User_IO, Ada.Integer_Text_IO;
  Digit_Count, Alpha_Count, Rest : Natural := 0;
  Ch : Character;
begin
  for I in 1 .. Max loop
  Get_From_User(From, Ch);
    case Ch is
      when '0' .. '9' =>
        Digit_Count := Digit_Count + 1;
        ...
    end case;
  end loop;
  -- output values
end Character_Count;
```

Creation of Tasks

```
Main_Controller : Controller;
Attendant1 : Garage_Attendant(2);
Input_Analyser : Character_Count(30, Andy);
type Garage_Forecourt is array (1 .. 10) of
                               Garage_Attendant;
GF : Garage_Forecourt;

type One_Pump_Garage(Pump : Pump_Number := 1) is
  record
    P : Garage_Attendant(Pump);
    C : Cashier(Pump);
  end record;
OPG : One_Pump_Garage(4);
```

Robot Arm Example

```
type Dimension is (Xplane, Yplane, Zplane);
task type Control(Dim : Dimension);
C1 : Control(Xplane);
C2 : Control(Yplane);
C3 : Control(Zplane);

task body Control is
  Position : Integer;      -- absolute position
  Setting : Integer;       -- relative movement
begin
  Position := 0;               -- rest position
  loop
    New_Setting (Dim, Setting);
    Position := Position + Setting;
    Move_Arm (Dim, Position);
  end loop;
end Control;
```

Warning

- Task discriminant do not provide a general parameter passing mechanism. Discriminants can only be of a discrete type or access type

```
type Garage_Forecourt is array (1 .. 10) of
                                   Garage_Attendants;
GF : Garage_Forecourt;
```

- All `Garage_Attendants` have to be passed the same parameter (the default)
- How can we get around this problem?

Work-around for Task Arrays and Discriminants

```
package Count is
  function Assign_Pump_Number return Pump_Number;
end Count;
package body Count is
  Number : Pump_Number := 0;
  function Assign_Pump_Number return Pump_Number is
  begin
    Number := Number + 1; return Number;
  end Assign_Pump_Number;
end Count;
```

```
task type New_Garage_Attendant(
     Pump : Pump_Number := Count.Assign_Pump_Number) is
  entry Serve_Leaded(G : Gallons);
  entry Serve_Unleaded(G : Gallons);
end Garage_Attendant;
type Forecourt is array (1..10) of New_Garage_Attendant;
Pumps : Forecourt;
```

A Procedure with Two Tasks

```
procedure Example1 is
  task A;
  task B;

  task body A is
    -- local declarations for task A
  begin
    -- sequence of statement for task A
  end A;

  task body B is
    -- local declarations for task B
  begin
    -- sequence of statements for task B
  end B;
begin
      -- tasks A and B start their executions before
      -- the first statement of the procedure's sequence
      -- of statements.
      ...
end Example1; -- the procedure does not terminate
              -- until tasks A and B have
              -- terminated.
```

Dynamic Task Creation

- By giving non-static values to the bounds of an array (of tasks), a dynamic number of tasks is created.
- Dynamic task creation can be obtained explicitly using the "new" operator on an access type (of a task type)

```
procedure Example2 is
  task type T;
  type A is access T;
  P : A;
  Q : A:= new T;
  . . .
begin
  ...
  P := new T;
  Q := new T;   -- What happens to old Q.all?
  ...
end example2;
```

This creates a task that immediately starts its initialization and execution; the task is designated Q.**all**

Activation, Execution & Finalisation

The execution of a task object has three main phases:

- Activation — the elaboration of the declarative part, if any, of the task body (any local variables of the task are created and initialised during this phase)
- Normal Execution — the execution of the statements within the body of the task
- Finalisation — the execution of any finalisation code associated with any objects in its declarative part

Task Activation

```
declare
   task type T_Type1;
   task A;
   B, C : T_Type1;
   task body A is ...;
   task body T_Type1 is
   ...

begin
   ...
end;
```

created when declaration is elaborated

B and C created when elaborated

tasks activated when elaboration is finished

first statement executes once all tasks have finished their activation

Task Activation

- All static tasks created within a single declarative region begin their activation immediately the region has elaborated
- The first statement following the declarative region is not executed until all tasks have finished their activation
- Follow activation, the execution of the task object is defined by the appropriate task body
- A task need not wait for the activation of other task objects before executing its body
- A task may attempt to communicate with another task once that task has been created; the calling task is delayed until the called task is ready

Dynamic Task Activation

- Dynamic tasks are activated immediately after the evaluation of the allocator (the new operator) which created them
- The task which executes the allocator is blocked until all the created task(s) have finished their activation

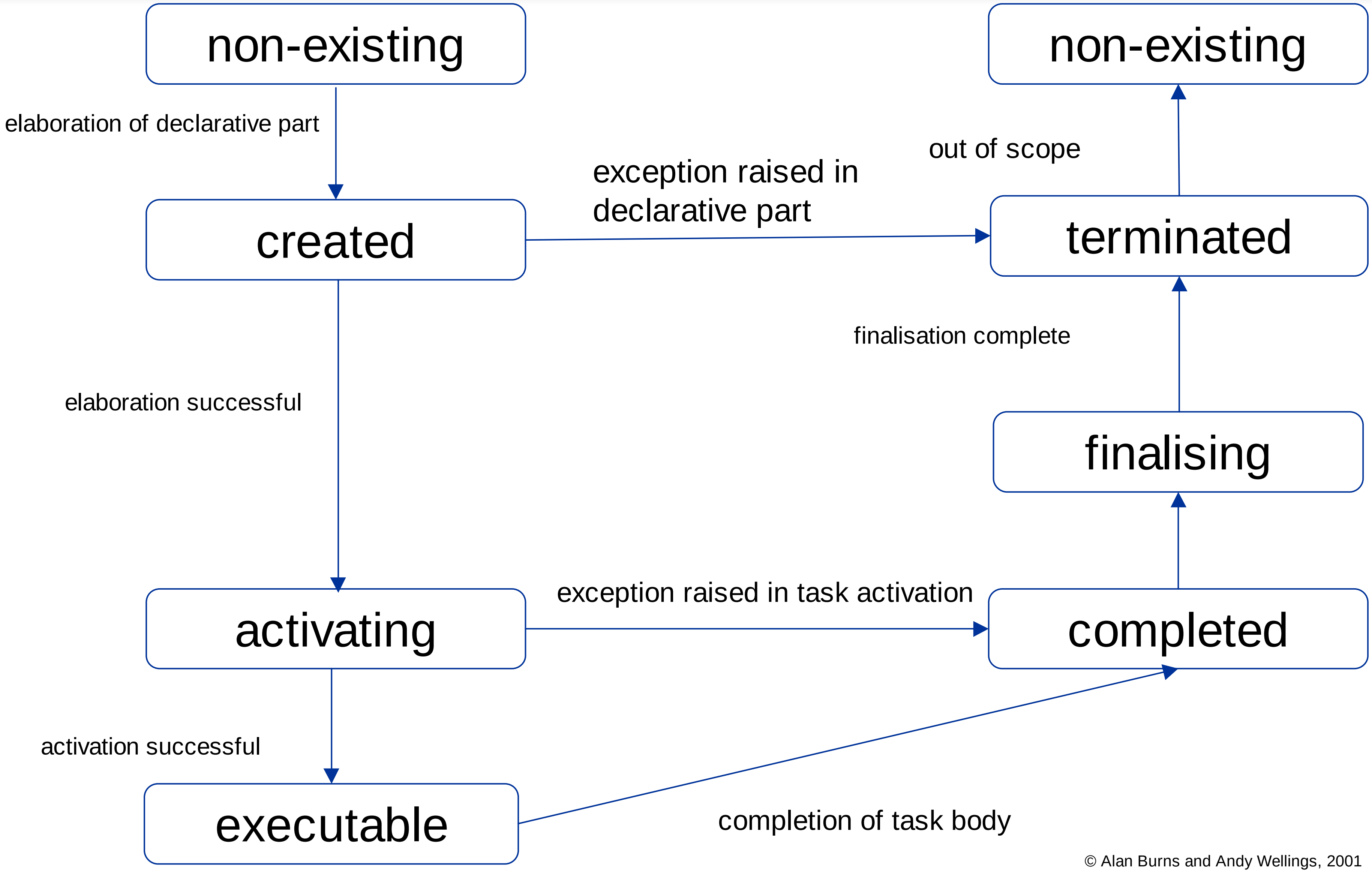
```
declare
  task type T_Type;
  type T_Type_Ptr is access T_Type;
  Ref1 : T_Type_Ptr;
  task body T_Type is ...;
begin
      Ref1 := new T_Type;
end;
```

creation and activation of **Ref1.all**

Exceptions and Task Activation

- If an exception is raised in the elaboration of a declarative part, any tasks created during that elaboration are never activated but become terminated
- If an exception is raised during a task's activation, the task becomes completed or terminated and the predefined exception `Tasking_Error` is raised prior to the first executable statement of the declarative block (or after the call to the allocator); this exception is raised just once
- The raise will wait until all currently activating tasks finish their activation

Task States in Ada



Creation and Hierarchies

- A task which is responsible for creating another task is called the parent of the task, and the created task is called the child
- When a parent task creates a child, it must wait for the child to finish activating
- This suspension occurs immediately after the action of the allocator, or after it finishes elaborating the associated declarative part

Termination and Hierarchies

- The parent of a task is responsible for the creation of a child
- The master of a dependent task must wait for the dependent to terminate before itself can terminate
- In many cases the parent is also the master

```
task Parent_And_Master;
task body Parent_And_Master is
  task Child_And_Dependent;
  task body Child_And_Dependent is
  begin ... end;
begin
  ...
end Parent_And_Master;
```

task becomes Completed, it Terminates when Child_And_Dependent terminates

Master Blocks

```
declare -- internal MASTER block
  -- declaration and initialisation of local variables
  -- declaration of any finalisation routines
  task Dependent;
  task body Dependent is begin ... end;
begin -- MASTER block
  ...
end; -- MASTER block
```

- The task executing the master block creates `Dependent` and therefore is its parent
- However, it is the MASTER block which cannot exit until the `Dependent` has terminated (not the parent task)

Termination and Dynamic Tasks

- The master of a task created by the evaluation of an allocator is the declarative region which contains the access type definition

```
declare
  task type Dependent;
  type Dependent_Ptr is access Dependent;
  A : Dependent_Ptr;
  task body Dependent is begin ... end;
begin
  ...
  declare
    B : Dependent;
    C : Dependent_Ptr := new Dependent;
  begin
    A := C;
  end;
end;
```

must wait for B to terminate but not C.all; C.all could still be active although the name C.all is out of scope; the task can still be accessed via

Termination and Library Units

- Tasks declared in library level packages have the main program as their master (in effect)
- Tasks created by an allocator whose access type is a library level package also have the main program as their master
- The main program cannot terminate until all library level tasks have terminated
- Actually, there is a conceptual task called the Environment Task which elaborates the library units before it calls the main procedure

Library Tasks

```
package Library_Of_Useful_Tasks is
  task type Agent(Size : Integer := 128);
  Default_Agent : Agent;
  ...
end Library_Of_Useful_Tasks;-- a library package.

with Library_Of_Useful_Tasks;
use Library_Of_Useful_Tasks;
procedure Main is
  My_Agent  : Agent;
begin
  null;
end Main;
```

Note: an exception raised in `Default_Agent` cannot be handled by the `Main` program

Completion versus Termination

- A task completes when
  - finishes execution of its body (either normally or as the result of an unhandled exception).
  - it executes a "terminate" alternative of a select statement (see later) thereby implying that it is no longer required.
  - it is aborted.
- A task terminates when all is dependents have terminated.
- If an unhandled exception in a task is isolated to just that task. Another task can enquire (by the use of an attribute) if a task has terminated:

```
if T'Terminated then    -- for some task T
  -- error recovery action
end if;
```

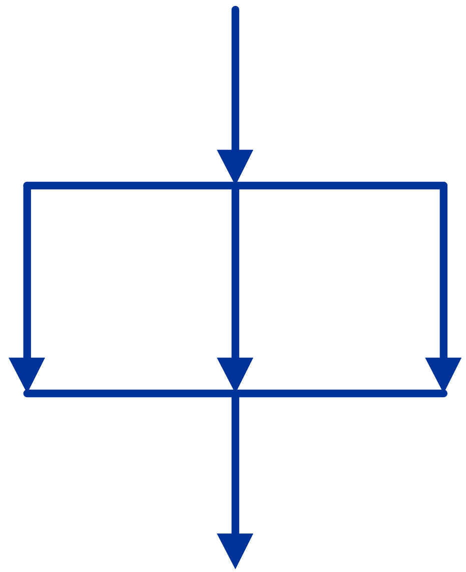
- However, the enquiring task cannot differentiate between normal or error termination of the other task.

Task Abortion

- Any task can abort any other task whose name is in scope
- When a task is aborted all its dependents are also aborted — why?
- The abort facility allows wayward tasks to be removed
- If, however,a rogue task is anonymous then it cannot be named and hence cannot easily be aborted. How could you abort it?
- It is desirable, therefore, that only terminated tasks are made anonymous

COBEGIN and PAR

- Ada 95 has explicit process declaration for its model of concurrency. Other languages (e.g. occam) use a COBEGIN or PAR structure



```
PAR
   B   C
   B   C   B
   B   C   C
```

How can this structure be represented in Ada 95?

Example Exam Question

- Explain fully the following relationships between processes (tasks) in the context of concurrent programming
  - parent <=> child
  - guardian (or master) <=> dependent
- Indicate in your answer the difference between a guardian process and a guardian block.
- Draw the state transition diagram for a process which during its life time can be a child, a parent, a guadian and a dependent.

Exam Problem

- For every task in the following Ada program, indicate its parent and guardian (master) and , if appropriate its children and dependents. Also indicate the dependents of the Main and Hierarchy procedures

```
procedure Main is
  procedure Hierarchy is
    task A;
    task type B;
    type PB is access B;
    pointerB : PB;
    task body A is separate;
    task body B is
    begin
      -- sequence of statements
    end B;
  begin . . . end Hierarchy;
begin
  Hierarchy;
end Main;
```

```
task body A is
  task C;
  task D;
  task body C is
  begin
    -- seq of statements including
    pointerB := new B;
  end C;
  task body D is
    another_PointerB : PB;
  begin
    another_PointerB := new B;
  end D;
begin
  -- sequence of statements
end A;
```

What happens?

```
procedure Main is
begin
  declare
    task Y;
    task body Y is
      I :Integer_Subtype := Read_Int;
    begin
      I := Read_Int;
    exception
      when others => . . .;
    end Y;
  begin
    . . .
    exception
      when Constraint_Error => . . .;
      when Tasking_Error => . . .;
  end;
exception
  when Constraint_Error => . . .;
  When Tasking_Error => . . .;
end;
```

Returns a value outside the subtype range

Returns a value outside the subtype range

What happens

```
declare
  A : TaskType1; -- successfully completes its activation
  B : TaskType2; -- Raises an exception during its activation
begin
 . . .
 . . .
exception
   when Tasking_Error => . . .;
   when others => . . . ;
end;
```

tasks begin activation here

Execution of the block cannot start until tasks have finished activation

Task Identification

- In some circumstances, it is useful for a task to have a unique identifier
- E.g, a server task is not usually concerned with the type of the client tasks. However, there are occasions when a server needs to know that the client task it is communicating with is the same client task with which it previously communicated
- Although the core Ada language provides no such facility, the Systems Programming Annex provides a mechanism by which a task can obtain its own unique identification. This can then be passed to other tasks

Task Id

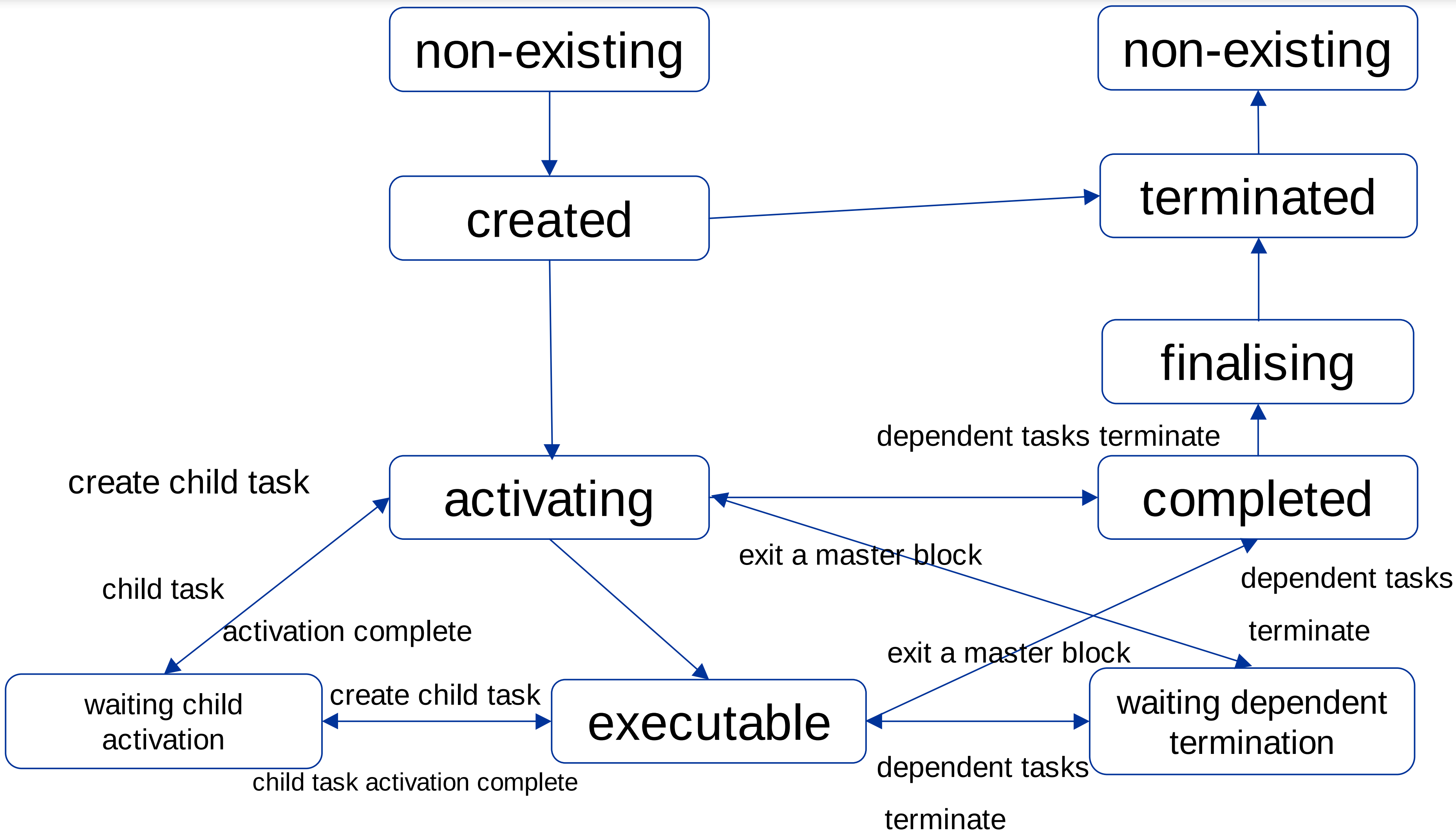
```
package Ada.Task_Identification is
  type Task_Id is private;
  Null_Task_Id : constant Task_Id;
  function "=" (Left, Right : Task_Id)
               return Boolean;
  function Current_Task return Task_Id;
    -- returns unique id of calling task
  -- other functions not relevant here
private
  ...
end Ada.Task_Identification;
```

Attributes

- The Annex supports two attributes:
  - For any prefix `T` of a task type, `T'Identity` returns a value of type `Task_Id` that equals the unique identifier of the task denoted by `T`
  - For any prefix `E` that denotes an entry declaration, `E'Caller` returns a value of type `Task_Id` that equals the unique identifier of the task whose entry call is being serviced

Care must be taken when using task identifiers since there is no guarantee that, at some later time, the task will still be active or even in scope

Task States in Ada



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

```
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 --- DEPRECIATED
  public final void setDaemon();
  // throws SecurityException, IllegalThreadStateException
  public final boolean isDaemon();
  // Note, RuntimeExceptions are not listed as part of the
  // method specification. Here, they are shown as comments
}
```

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();
```

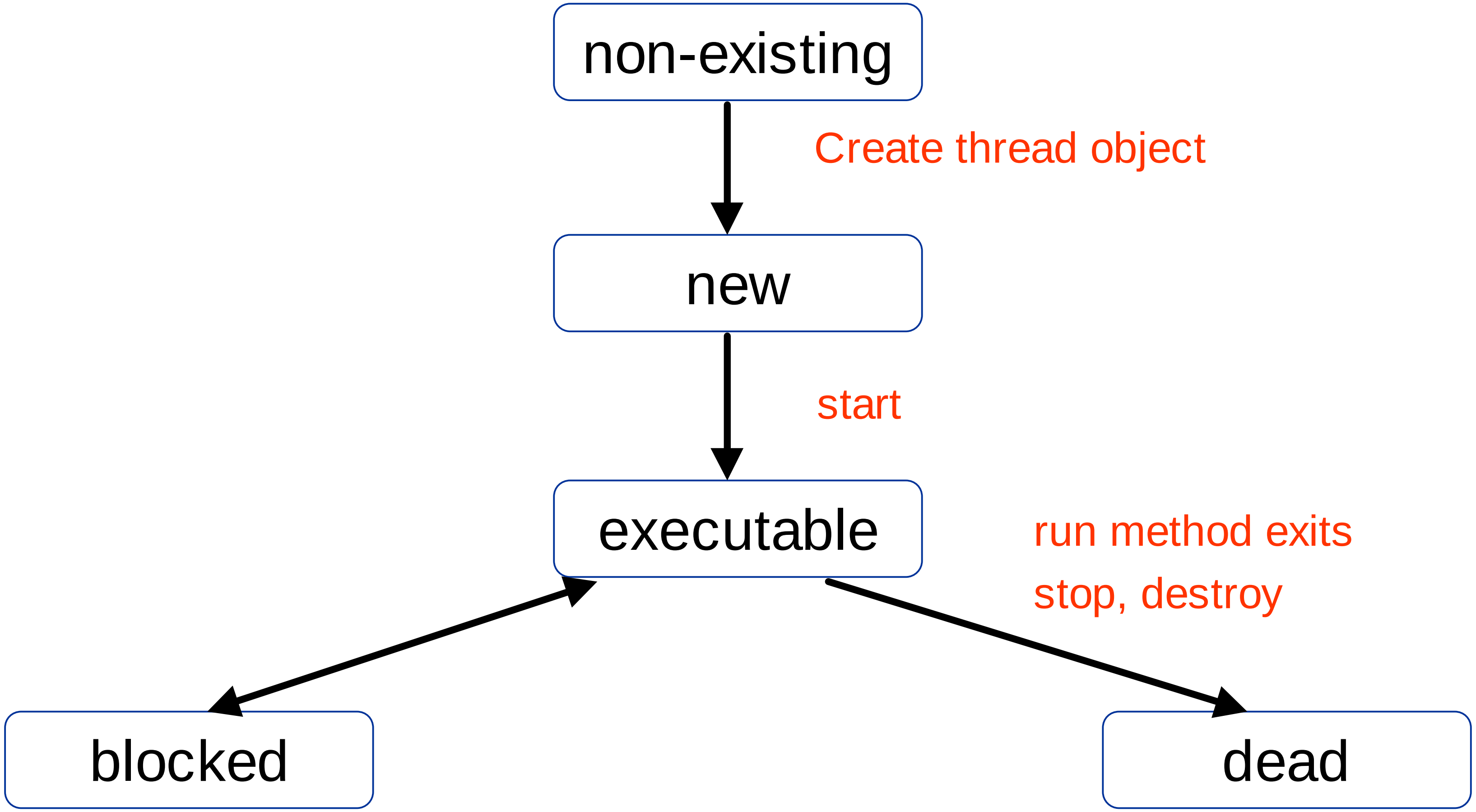
Alternative Robot Control

```
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 (by means of constructor methods) allows arbitrary data to be passed as parameters
- 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 the target 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 (depreciated) and should not be used
- by its destroy method being called -- destroy terminates the thread without any cleanup (never been implemented in the JVM)

Daemon Threads

- Java threads can be of two types: user threads or daemon threads
- Daemon threads are those threads which provide general services and typically never terminate
- When all user threads have terminated, daemon threads can also be terminated and the main program terminates
- The `setDaemon` method must be called before the thread is started
- (Daemon threads provide the same functionality as the Ada ``or terminate'' option on the select statement)

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 `NullPointerException` is thrown when:
  - A null pointer is passed to the `stop` method
- 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

Concurrent Execution in POSIX

- Provides two mechanisms: fork and 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)
- To manipulate these you use attribute objects
- Threads are created using an appropriate attribute object

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 */

pthread_t pthread_self(void);
```

All functions returns 0 if successful, otherwise an error number

Robot Arm in C/POSIX

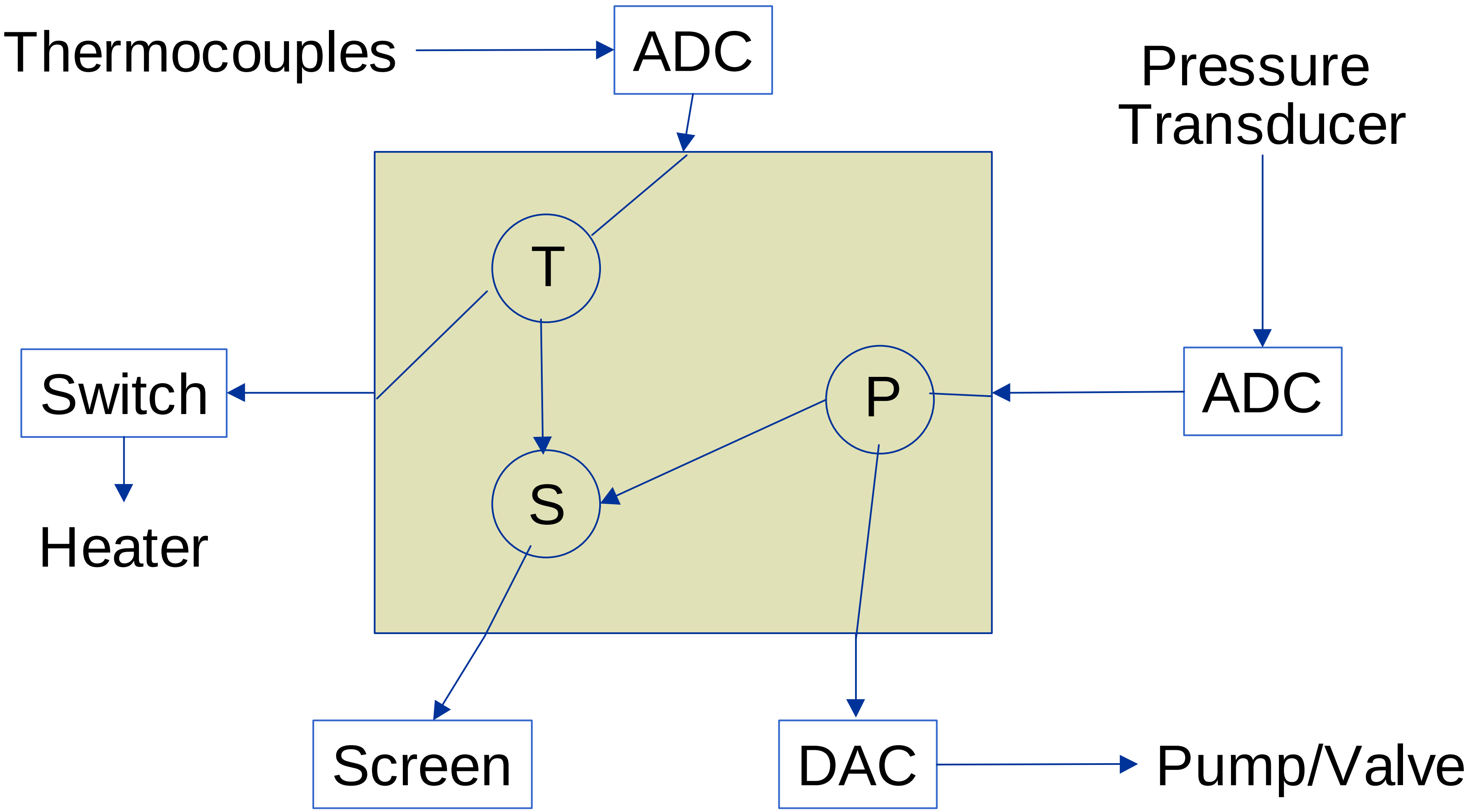
```
#include <pthread.h>

pthread_attr_t attributes;
pthread_t xp, yp, zp;

typedef enum {xplane, yplane, zplane} dimension;

int new_setting(dimension D);
void move_arm(int D, int P);

void controller(dimension *dim)
{
  int position, setting;

  position = 0;
  while (1) {
    setting = new_setting(*dim);
    position = position + setting;
    move_arm(*dim, position);
  };
  /* note, process does not terminate */
}
```

```
int main() {
  dimension X, Y, Z;
  void *result;

  X = xplane,
  Y = yplane;
  Z = zplane;
  PTHREAD_ATTR_INIT(&attributes);
  /* set default attributes */

  PTHREAD_CREATE(&xp, &attributes, (void *)controller, &X);
  PTHREAD_CREATE(&yp, &attributes, (void *)controller, &Y);
  PTHREAD_CREATE(&zp, &attributes, (void *)controller, &Z);
  PTHREAD_JOIN(xp, &result);
  /* need to block main program */

  exit(-1); /* error exit, the program should not terminate */
}
```

Need JOIN as when a process terminates, all its threads are forced to terminate

SYS_CALL style indicates a call to sys_call with a check for error returns

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 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 is the best approach?

Useful Packages

```
package Data_Types is
  type Temp_Reading is new Integer range 10..500;
  type Pressure_Reading is new Integer range 0..750;
  type Heater_Setting is (On, Off);
  type Pressure_Setting is new Integer range 0..9;
end Data_Types;
```

necessary type definitions

```
with Data_Types; use Data_Types;
package IO is
  procedure Read(TR : out Temp_Reading); -- from ADC
  procedure Read(PR : out Pressure_Reading);
  procedure Write(HS : Heater_Setting);-- to switch
  procedure Write(PS : Pressure_Setting); -- to DAC
  procedure Write(TR : Temp_Reading);  -- to screen
  procedure Write(PR : Pressure_Reading);-- to screen
end IO;
```

procedures for data exchange with the environment

Control Procedures

```
with Data_Types; use Data_Types;
package Control_Procedures is
  -- procedures for converting a reading into
  -- an appropriate setting for output.
  procedure Temp_Convert(TR : Temp_Reading;
                         HS : out Heater_Setting);
  procedure Pressure_Convert(PR : Pressure_Reading;
                             PS : out Pressure_Setting);
end Control_Procedures;
```

Sequential Solution

```
with Data_Types; use Data_Types; with IO; use IO;
with Control_Procedures;  use Control_Procedures;
procedure Controller is
  TR : Temp_Reading;
  PR : Pressure_Reading;
  HS : Heater_Setting;
  PS : Pressure_Setting;
begin
  loop
    Read(TR);     -- from ADC
    Temp_Convert(TR,HS);
    Write(HS);    -- to switch
    Write(TR);    -- to screen
    Read(PR);
    Pressure_Convert(PR,PS);
    Write(PS);
    Write(PR);
  end loop; -- infinite loop
end Controller;
```

No O.S. Required

Disadvantages of the Sequential Solution

- Temperature and pressure readings must be taken at the same rate
- The use of counters and if statements will improve the situation
- But may still be necessary to split up the conversion procedures `Temp_Convert` and `Pressure_Convert`, 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 vice versa)
- 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

An Improved System

```
with Data_Types; use Data_Types; with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is
  TR : Temp_Reading;   PR : Pressure_Reading;
  HS : Heater_Setting; PS : Pressure_Setting;
  Ready_Temp, Ready_Pres : Boolean;
begin
  loop
    if Ready_Temp then
      Read(TR); Temp_Convert(TR,HS);
      Write(HS);  Write(TR);
    end if;
    if Ready_Pres then
      Read(PR); Pressure_Convert(PR,PS);
      Write(PS);  Write(PR);
    end if;
  end loop;
end Controller;
```

What is wrong with this?

Problems

- The solution is more reliable
- Unfortunately the program now spends a high proportion of its time in a busy loop polling the input devices to see if they are ready
- Busy-waits are unacceptably inefficient
- Moreover programs that rely on busy-waiting are difficult to design, understand or prove correct

The major criticism with the sequential program is that no recognition is given to the fact that the pressure and temperature cycles are entirely independent subsystems. In a concurrent programming environment this can be rectified by coding each system as a task.

Using O.S. Primitives I

```
package OSI is
  type Thread_ID is private;
  type Thread is access procedure;

  function Create_Thread(Code : Thread)
           return Thread_ID;
  -- other subprograms
  procedure Start(ID : Thread_ID);
private
   type Thread_ID is ...;
end OSI;
```

Using O.S. Primitives II

```
package Processes is
  procedure Temp_C;
  procedure Pressure_C;
end Processes;

with IO; use IO;
with Control_Procedures; use Control_Procedures;
package body Processes is
  procedure Temp_C is
    TR : Temp_Reading;  HS : Heater_Setting;
  begin
    loop
      Read(TR); Temp_Convert(TR,HS);
      Write(HS); Write(TR);
    end loop;
  end Temp_C;
```

Using O.S. Primitives III

```
  procedure Pressure_C is
    PR : Pressure_Reading;
    PS : Pressure_Setting;
  begin
    loop
      Read(PR);
      Pressure_Convert(PR,PS);
      Write(PS);
      Write(PR);
    end loop;
  end Pressure_C;
end Processes;
```

Using O.S. Primitives IV

```
with OSI, Processes; use OSI, Processes;
procedure Controller is
  TC, PC : Thread_ID;
begin
   TC := Create_Thread(Temp_C'Access);
   PC := Create_Thread(Pressure_C'Access);
   Start(TC);
   Start(PC);
end Controller;
```

Better, more reliable solution

For realistic OS, solution becomes unreadable!

Ada Tasking Approach

```
with Data_Types; use Data_Types; with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is

  task Temp_Controller;
  task body Temp_Controller is
    TR : Temp_Reading;
    HS : Heater_Setting;
  begin
    loop
      Read(TR);
      Temp_Convert(TR,HS);
      Write(HS); Write(TR);
    end loop;
  end Temp_Controller;

  task Pressure_Controller;
  task body Pressure_Controller is
      PR : Pressure_Reading;
      PS : Pressure_Setting;
    begin
      loop
        Read(PR);
        Pressure_Convert(PR,PS);
        Write(PS); Write(PR);
      end loop;
    end Pressure_Controller;

begin
  null;
end Controller;
```

Advantages of Concurrent Approach

- 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 versus 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 are beginning to emerge
- The Ada/Java philosophy is that the advantages outweigh the disadvantages

Summary of Concurrent Programming

- 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 states
    - non-existing
    - created
    - initialized
    - executable
    - waiting dependent termination
    - waiting child initialization
    - terminated

Variations in the Process Model

- structure
  - static, dynamic
- level
  - top level processes only (flat)
  - multilevel (nested)
- initialization
  - with or without parameter passing
- granularity
  - fine or coarse grain
- termination
  - natural, suicide
  - abortion, untrapped error
  - never, when no longer needed
- representation
  - coroutines, fork/join, cobegin, explicit process declarations

Ada, Java and C/POSIX

- Ada and Java provide a dynamic model with support for nested tasks and a range of termination options.
- POSIX allows dynamic threads to be created with a flat structure; threads must explicitly terminate or be killed.