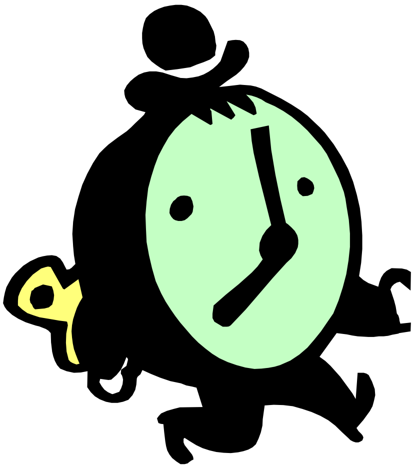
Real-Time Systems Programming

Summer-Semester 2002
Lecture 18
14 June 2002



Concurrent Programming conte

Contents

- Concurrency in
  - ***Ada*** (tasks)
  - ***Java*** (threads)
  - ***POSIX/C*** (processes and threads)
  - ***LegOS***
- Where to specify concurrency?
- A simple embedded system

Concurrency in Ada

- The unit of concurrency in Ada is called a ***task***
- Tasks must be ***explicitly declared***
  - No fork/join statement, COBEGIN/PAR etc
- Tasks may be declared ***at any program level***
- Tasks 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 monitor/conditional critical region)
  - ***Shared variables***

Ada 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

- 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 ***parameters***
  - a ***visible part*** defining entries and representation clauses
  - a ***private part*** defining hidden entries and representation clauses

Ada Example Task Specifications

```
task type Controller;


task type Agent(Param : Integer);


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

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

Task objects can be passed an integer parameter at their creation time

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

Objects will allow communication via two entries

Creation of Ada Tasks

- Can use array to create ***multiple*** instances
- Can create ***dynamic*** number of tasks via
  - giving non-static task array bounds – or
  - by using the **new** operator

```
Main_Controller : Controller;
Attendant1 : Garage_Attendant(2);

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);
```

Ada: 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;
```

An Ada 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
```

Activation, Execution & Finalisation

The execution of an Ada 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 initialized during this phase
  - In other languages called ***initialization***
- ***Normal Execution***
  - Execution of the statements within the body of the task
- ***Finalization***
  - Execution of any finalization code associated with any objects in its declarative part

Ada Task Activation

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

begin
  ...
end;
```

A is ***created*** when its declaration is elaborated

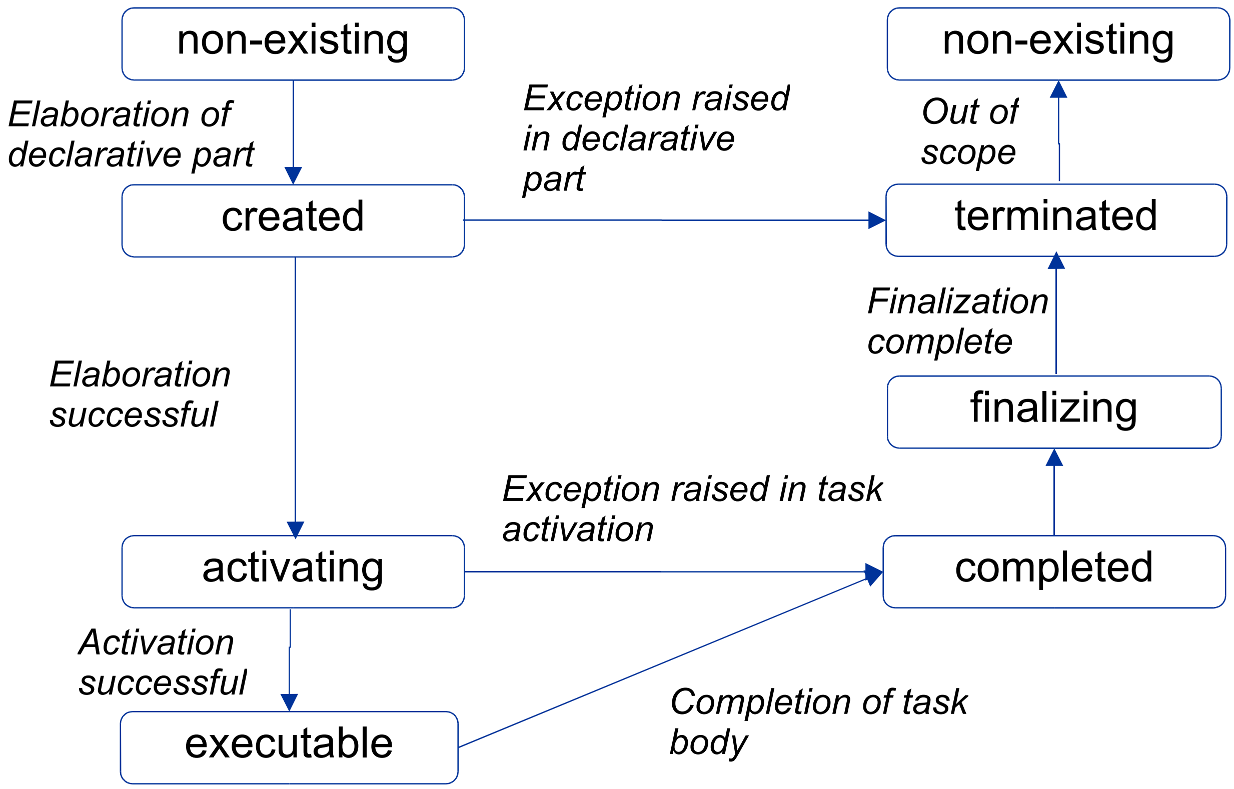
B and C are ***created*** when their declarations are elaborated

Tasks are ***activated*** when elaboration is finished

First statement ***executes*** once all tasks have finished their activation

- All static tasks created ***within a single declarative region*** begin their ***activation*** immediately after the region has elaborated
- The first statement following the declarative region is executed ***after*** all tasks have finished their activation
- Following activation:
  - ***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
  - Calling task is delayed until the called task is ready

Task States in Ada



R. v. Hanxleden SS 2002 – Real-Time Systems Programming – Lecture_18.sdd Foil 11

- ***If an exception is raised in the elaboration of a declarative part***:
  - ➢ ***No*** tasks created during that elaboration is activated, and instead will be terminated
- ***If an exception is raised during a task's activation***:
  - ➢ 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

Ada: Creation and Hierarchies

- ***Recall:*** The parent of a task is responsible for the creation of a child
- When a parent task creates a child:
  - Must wait for the child to finish activating
- This suspension occurs
  - Immediately after the action of the ***allocator*** (operator **`new`**), or
  - After it finishes elaborating the associated declarative part

Termination and Hierarchies

- ***Recall:*** 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;
```

`Parent_And_Master` 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

- Master of a task created by an allocator:
  - Declarative region containing ***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 A

Task Termination

A task ***terminates*** when

- it ***finishes execution*** of its body
  - 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***
- all is dependents have terminated

Unhandled Exceptions

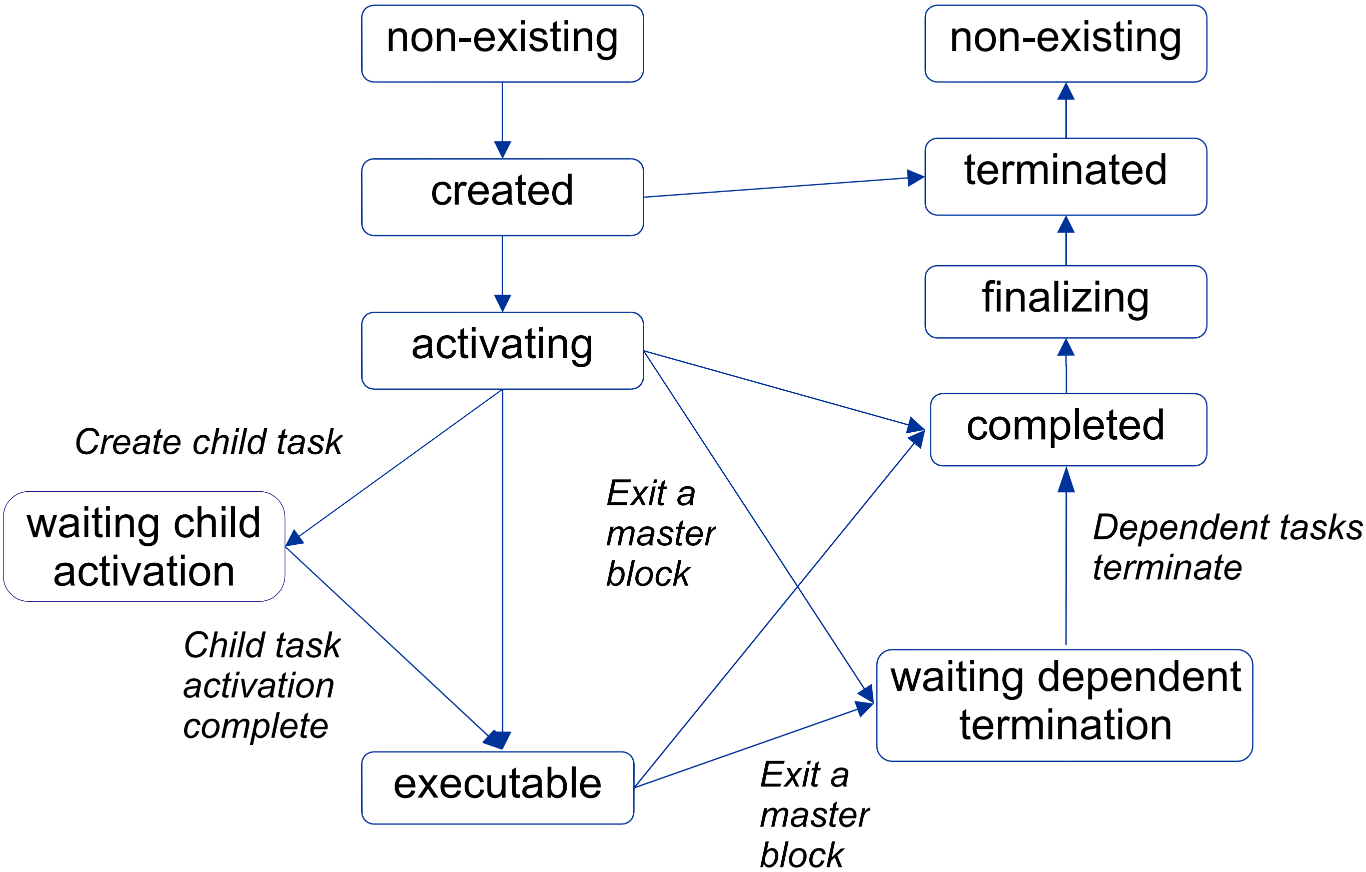
- The effect of an unhandled exception in a task is ***isolated to just that task***
- Another task can inquire (by the use of an attribute) if a task has terminated
- However, the enquiring task cannot differentiate between normal or error termination of the other task.

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

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
- 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

Task States in Ada



Concurrency in Java

- Predefined class `java.lang.Thread`
  - Provides thread/process creation mechanism
- To avoid all threads having to be child classes of `Thread`:
  - Standard interface

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

- Hence, any class which wishes to express concurrent execution must
  - Implement this interface
  - Provide the `run` method

The Java class `thread`

```
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 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 cont.

```
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();  // threads started
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 cont.

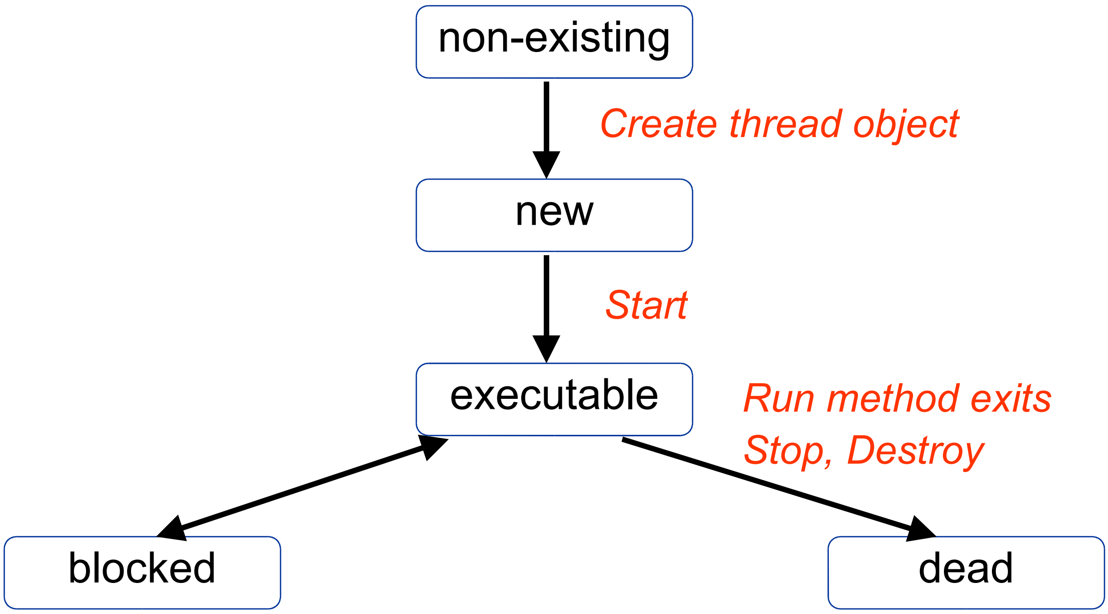
```
final int xPlane = 0;
final int yPlane = 1;
final int zPlane = 2;

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

- Allows ***dynamic thread creation***
- Allows ***arbitrary data*** to be passed as parameters
- Allows ***thread hierarchies*** and ***thread groups***
  - ***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
- **`join`** method:
  - Thread waits for other thread (the target) to terminate
- **`isAlive`** method:
  - Determines 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
- ... by its **`destroy`** method being called
  - destroy terminates the thread without any cleanup
  - never been implemented in the JVM
- ... via its **`stop`** method (depreciated)
  - inherently unsafe as it releases locks on objects and can leave those objects in inconsistent states

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

Thread Exceptions

- `NullPointerException`:
  - When a null pointer is passed to the `stop` method
- `InterruptException`:
  - When 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 return 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 */
}
```

Robot Arm in C/POSIX

```
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 */
}
```

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

*Need **`join`** as when a process terminates, all its threads are forced to terminate*

Threads in legOS

- A ***legOS*** thread is basically a memory area reserved on the stack and some allotted CPU time
- Similar to ***POSIX.1c*** threads
- Initially, there exist two threads:
  - the program manager
  - the energy manager

Thread Creation

```
pid_t execi(int (*code_start) (int, char**),
            int argc,
            char **argv,
            priority_t priority,
            size_t stack_size)
```

- Priority is fixed during existence of the thread
- ***Warning:*** the stack_size has to be made sufficiently large – otherwise arbitrary memory areas can be overwritten (a favorite cause of program crashes)

Further Thread Functions

```
// Makes rest of time slice available to
// other threads
void yield()

// Finishes thread with given return value
void exit(int retval)

// Terminates thread pid, as if this
// had called exit(-1)
void kill(pid_t pid)

// Terminates all threads with priority < p
void killall(priority_t p)
```

Suspending Threads

- Threads can specify a wake-up function that is tested each time a time-slice for the thread starts
- The thread becomes activated iff the wake-up function returns a non-zero value
- Should try to keep wake-up functions small – these are executed by the scheduler itself

```
wakeup_t wait_event(wakeup_t(*wakeup)(wakeup_t),
                    wakeup_t data)

// Example usage:
wait_event(dkey_pressed, KEY_ANY);
```

Multiple Task Example I

- The following program consists of two tasks:

```
#include "conio.h"
#include "direct-button.h"
#include "unistd.h"
#include "sys/tm.h"

pid_t pid;

int display_task(int argc, char **argv) {
  while(1) {
    cputs("Hello");
    lcd_refresh();
    sleep(1);
    cputs("nurse");
    lcd_refresh();
    sleep(1);
  }
  return 0;
}
```

Multiple Task Example II

```
int stop_task(int argc, char **argv) {
  msleep(200);
  while (!PRESSED(button_state(), BUTTON_RUN))
    ;
  kill(pid);
  return 0;
}

int main() {
  pid = execi(&display_task, 0, NULL, 0,
              DEFAULT_STACK_SIZE);
  execi(&stop_task, 0, NULL, 0, DEFAULT_STACK_SIZE);

  return 0;
}
```

Where to Specify Concurrency ?

- There is a debate over whether a ***language*** should define concurrency (or leave it up to the ***OS***)
- ***Pro*** (***Ada***, ***Java***):
  - Increases readability and maintainability
  - Improves portability across OSs
  - An OS may not be available to embedded computer
- ***Con*** (***C***, ***C++***):
  - Makes combination of different languages more difficult
  - May be difficult to implement on top of an OS
  - Emerging OS standards improve portability
- Integrated Modular Avionics APEX defines interface

Summary I

- 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 its ***expressive power*** and ***ease of use***
- 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 II

- The use of a concurrent programming language requires a ***run-time support system*** to manage process execution
- Processes have several ***states***:
  - non-existing
  - created
  - initialized
  - executable
  - waiting dependent termination
  - waiting child initialization
  - terminated

Summary III

A process model is characterized by its:

- ***Structure***
  - static
  - dynamic
- ***Level***
  - top level processes only (flat)
  - multilevel (nested)
- ***Initialization***
  - with or without parameter passing
- ***Granularity***
  - fine grain
  - coarse grain

Summary IV

A process model is further characterized by its:

- ***Termination***
  - Natural
  - Suicide
  - Abortion
  - Untrapped error
  - Never
- ***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
- ***Ada*** implements a guardian/dependent relationship (the master block); ***Java*** relies on garbage collection
- ***POSIX*** allows dynamic threads to be created with a ***flat*** structure
  - threads must explicitly terminate or be killed.

Problem Set 9 – Due: 26 June 2002

1.) To what extent can the Ada process state transition diagram be applied to processes in Java and in C/POSIX? **(2 pts)**

2.) Modify the robot built last week such that it *concurrently* performs the following tasks:

a) Check for crossings of dark lines

b) Check for obstacles on the way

c) Check whether a button is pressed

d) Check whether a pre-defined time-out occurs (1 minute)

The robot should be halted whenever a crossed line is thicker than 10cm, *or* an obstacle is encountered, *or* a button is pressed, *or* the time-out occurs; in each case a different alarm should sound.

You should provide two versions of the controller:

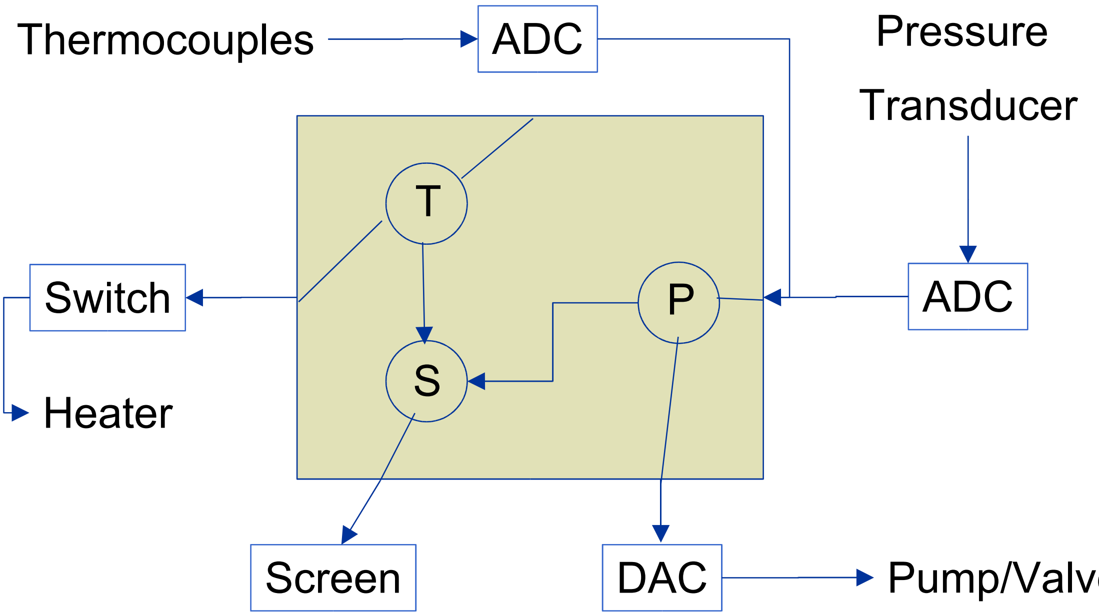
1) In C – using the threads provided by LegOS

2) In Java – using for example lejos (see http://www.informatik.uni-kiel.de/~kwi/programmierung/lejos.html)

a) *Documentation* (overview of approach and assumptions, commented source code, measurement of accuracy) **(2x3 pts)**

b) Functional *robot* **(2x3 pts)**

A Simple Embedded System



- The objective is to keep the temperature and pressure of a 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 ***OS 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
  - 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;


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;
```

Necessary type definitions

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_Ctrl(TR : Temp_Reading;
                      HS : out Heater_Setting);
  procedure Pressure_Ctls(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 -- infinite loop
    Read(TR);    -- from ADC
    Temp_Ctrl(TR,HS);
    Write(HS);   -- to switch
    Write(TR);   -- to screen
    Read(PR);
    Pressure_Ctrl(PR,PS);
    Write(PS);
    Write(PR);
  end loop;
end Controller;
```

No OS required !

Disadvantages of the Sequential Solution

- Temperature and pressure readings must be taken at the ***same rate***
- The use of counters and if statements would improve the situation
- But may still be necessary to split up the conversion procedures **`Temp_Ctrl`** and **`Pressure_Ctrl`**, and interleave their actions to balance work
- While waiting to read a temperature ***no attention*** can be given to pressure (and vice versa)
- A system failure that results in, e.g., control never returning from the temperature **`Read`** would also block the rest of the controller

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
- ***This approach still does not express the concurrency between the temperature and the pressure controller !***

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;
```

```
    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 III

```
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 both tasks are suspended*** there is no busy loop
- ***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***