MÁSTER UNIVERSITARIO EN SISTEMAS ESPACIALES
POR LA UNIVERSIDAD POLITÉCNICA DE MADRID

POLITÉCNICA

2021.04.12

Data handling

Concurrency and tasking

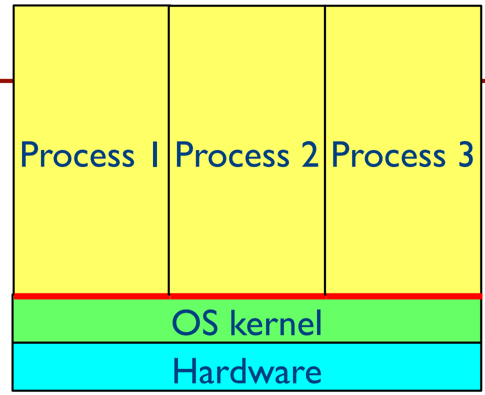
Juan Antonio de la Puente <juan.de.la.puente@upm.es>
Alejandro Alonso <alejandro.alonso@upm.es>

STRAST

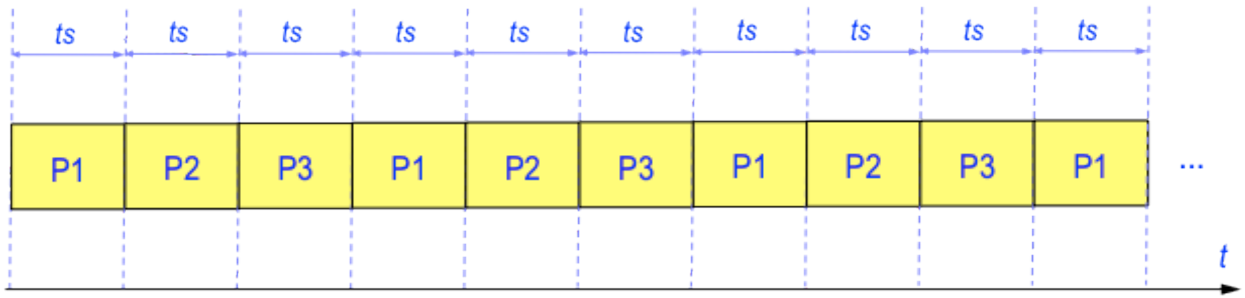
Processes and

Tasks/threads

Process

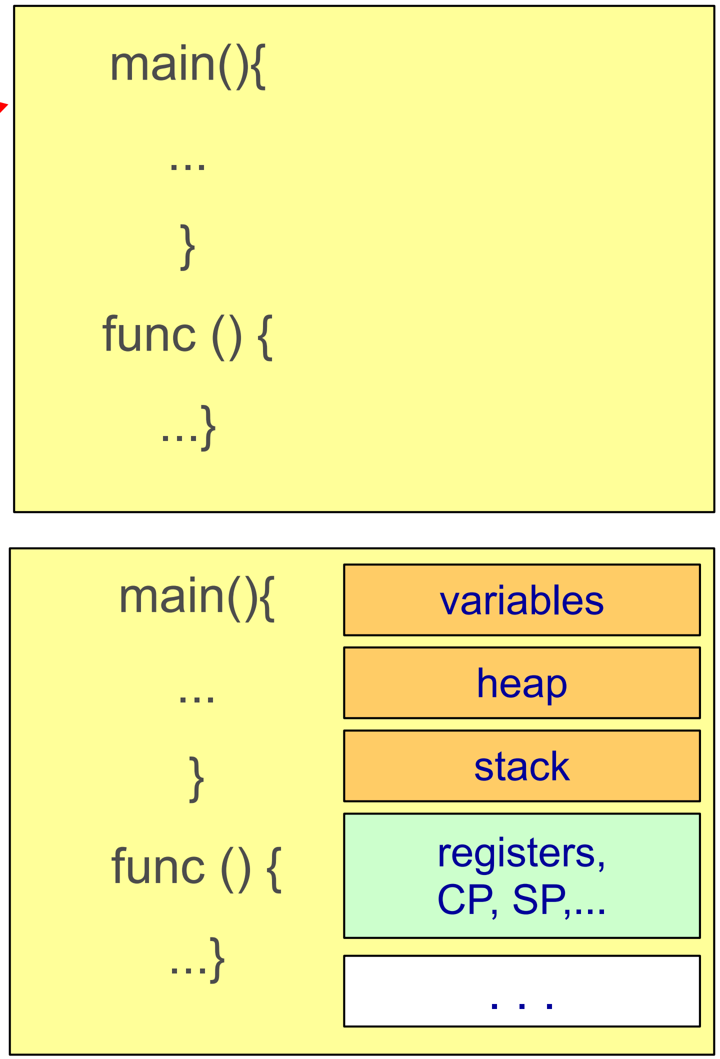


- Most operating systems support multiprogramming
- A process: a program in execution
  - Needs resources: CPU time, memory, files, E/S devices
- Processes are isolated:
  - A process cannot access directly to another
  - Interprocess: invoking OS system calls
- The OS multiplex process for using the CPU time
  - Processes creating and dispatching are time consuming
  -



Process management

- **Process: program running**
- **Program: code and y static data**
- **Process image in an OS**
  - Code
  - Process state:
    - Memory area: values of variables, stack, heap, etc.
    - Registers: program counter, stack pointer
    - . . .

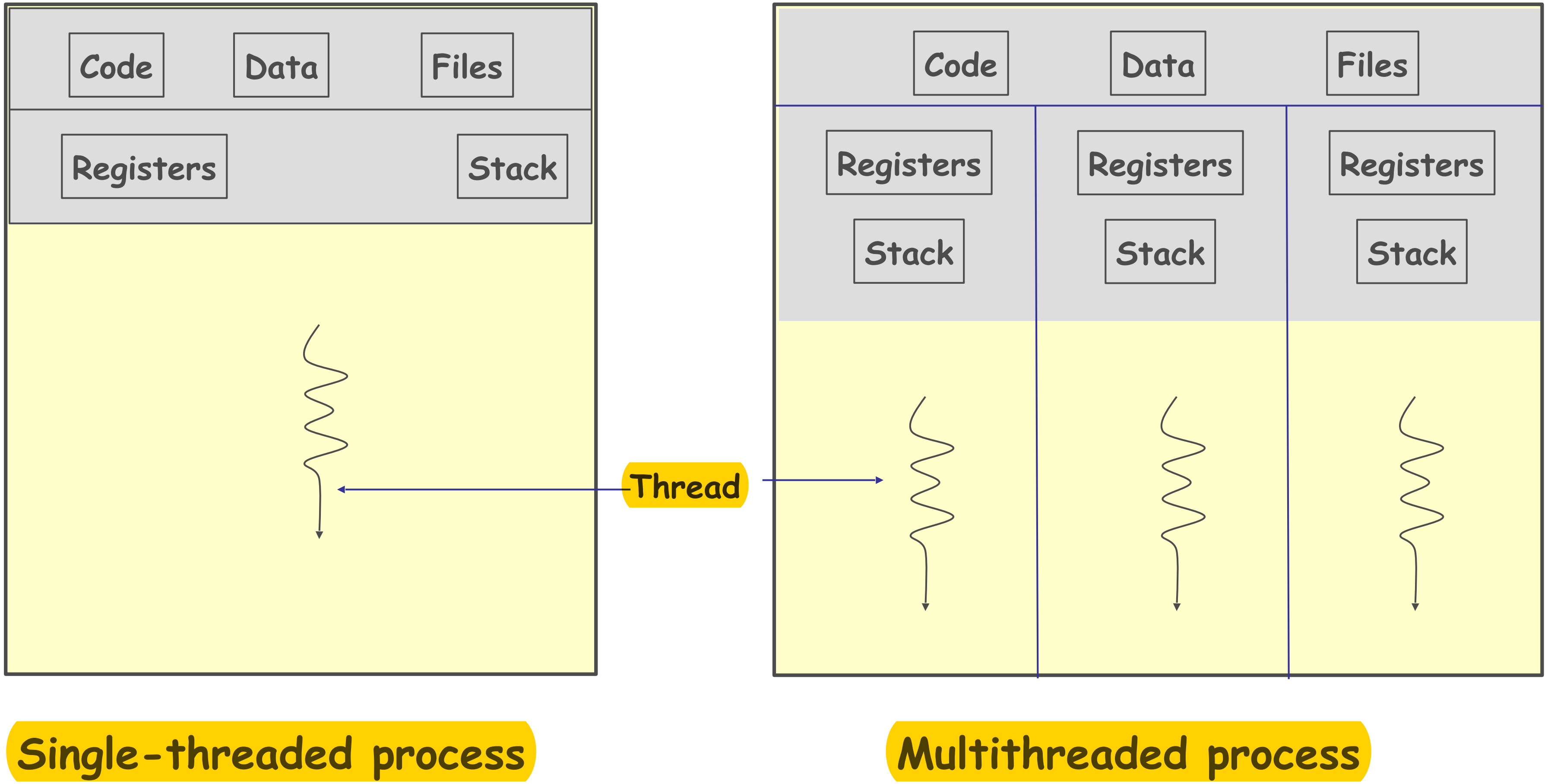


Thread

- Most operating systems support multithreading
- A process can include a set of running threads
  - Threads are running in concurrency
  - Thread state: registers, CP, SP
  - Threads in a process share the additional resources
    - Are less safe than processes.
- Threads creating and multiplexing are efficient

- Embedded systems: usual to have only threads

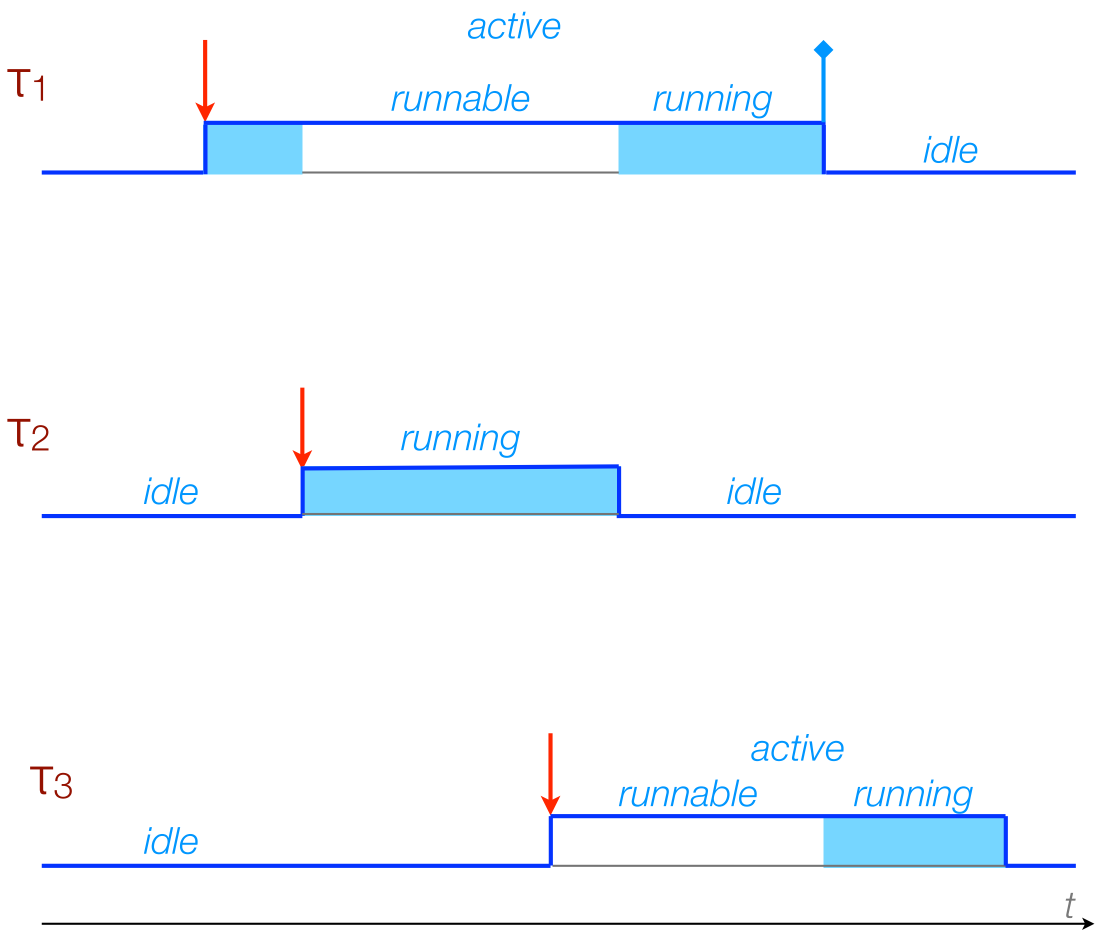
Processes/Threads



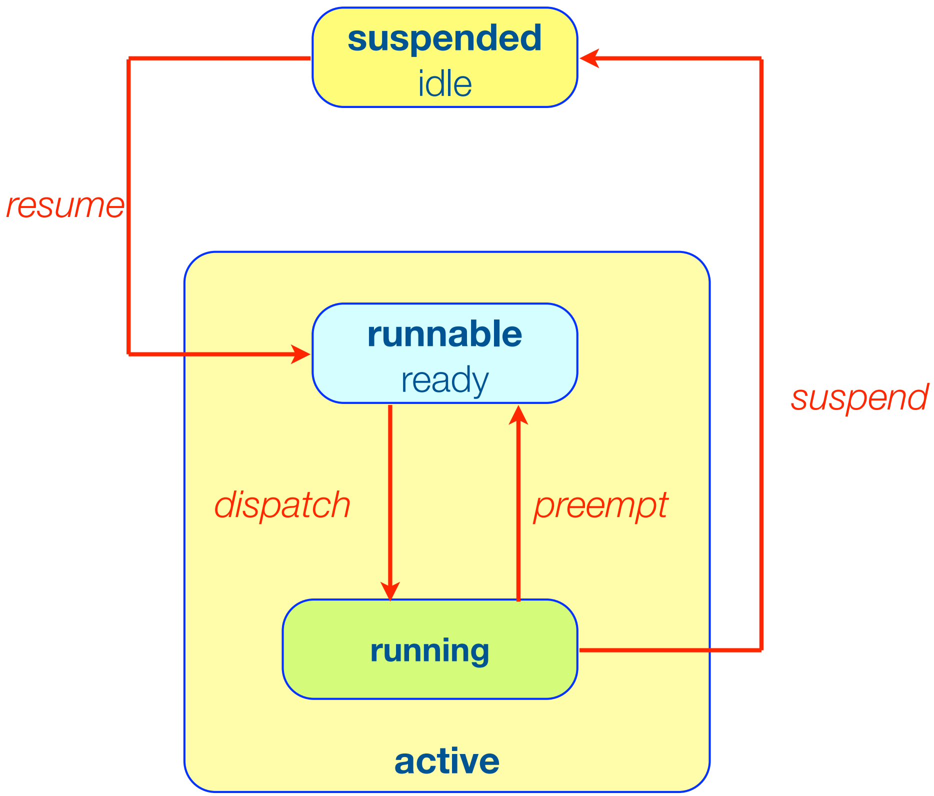
Concurrent tasks

Concurrent tasks

- Tasks are most often implemented using *threads*
  - concurrent flows of execution that progress asynchronously
  - processor time is multiplexed among active threads
  - requires operating system or run-time system support
    - e.g. POSIX threads, Ada tasks



Task states



- A task may be in different states
- Runnable tasks are dispatched for execution according to a scheduling method
- Scheduling is implemented by the OS kernel or the RTS

Tasks in Ada

- Ada tasks are program units
  - but not compilation units
  - must be compiled within packages
- Composed of specification and body
  - specification contains the visible interface
    - often empty
  - body contains the executable statements
- Tasks belong to a task type
  - explicit or anonymous

Example

```
package Temperature_Control is
   task type Temperature_Controller; -- specification
end Temperature_Control;

package body Temperature_Control is
   task body Temperature_Controller is -- body
      ...                              -- local declarations
   begin
      ...                              --  sequence of statements
   end Temperature_Controller;

   task Driver;                        -- single task object
   task body Driver is                 -- body
      ...
   begin
      ...
   end Driver;
end Temperature_Control;
```

Example (continued)

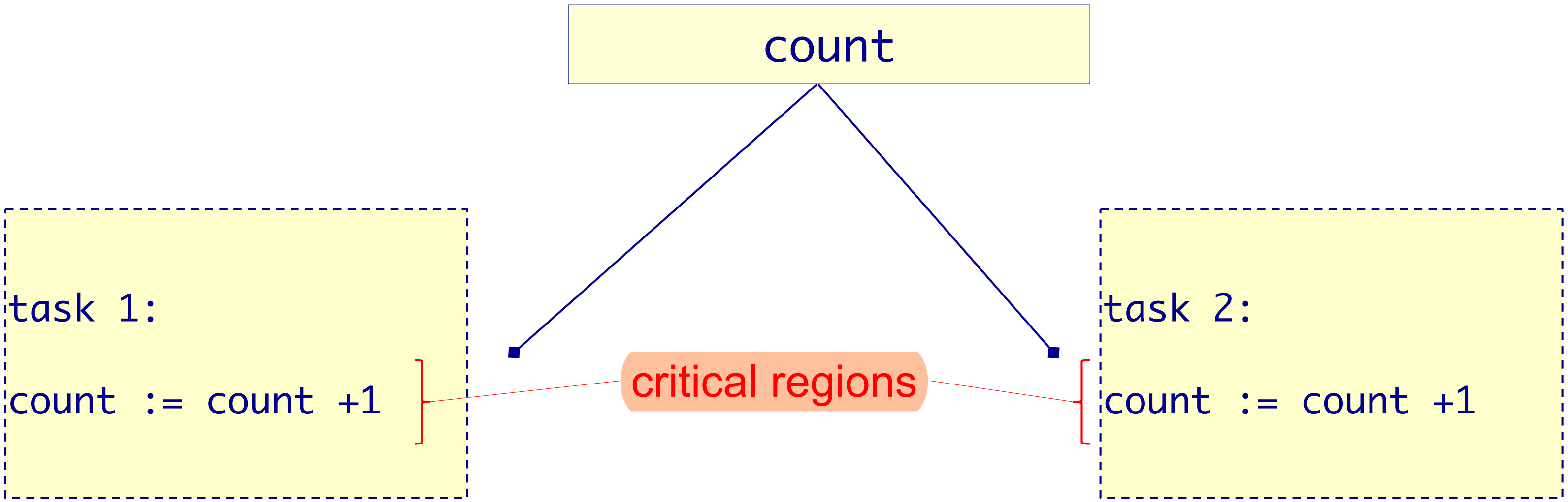
```
with Temperature_Control;
procedure Control is              -- main procedure
   use Temperature_Control;
   TC1, TC2 : Temperature_Controller;
begin         -- TC1, TC2, PC, and Driver run in parallel
 null;        -- the main procedure does nothing
end Control; -- and does not terminate until TC1, TC2,
             -- and Driver complete
```

Shared data and synchronisation

Shared data

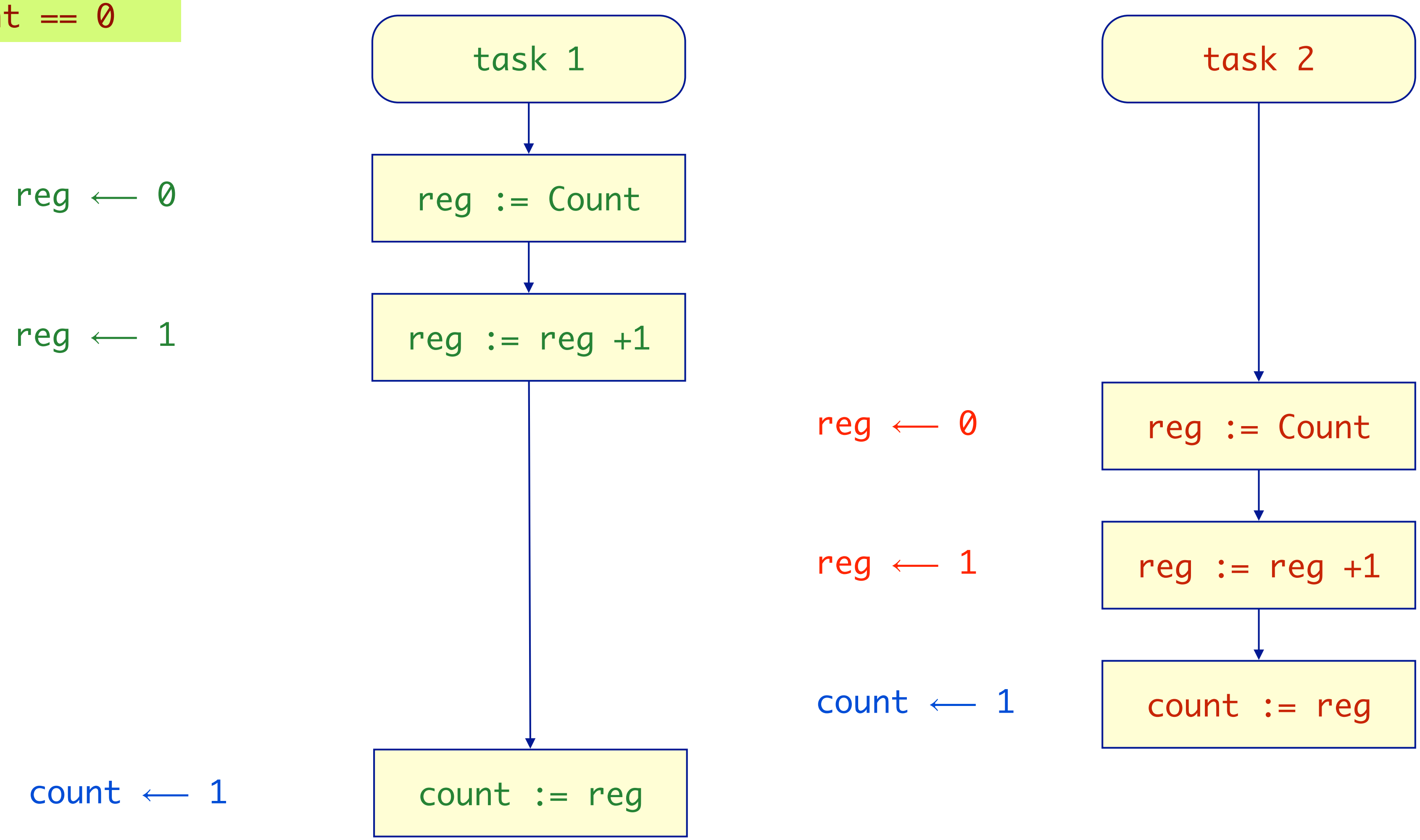
- Concurrent access to shared data may lead to race conditions
  - the value of the data may depend on the relative speeds of concurrent threads
- In order to prevent race conditions, access to shared data must be carried out in mutual exclusion

Example: shared counter



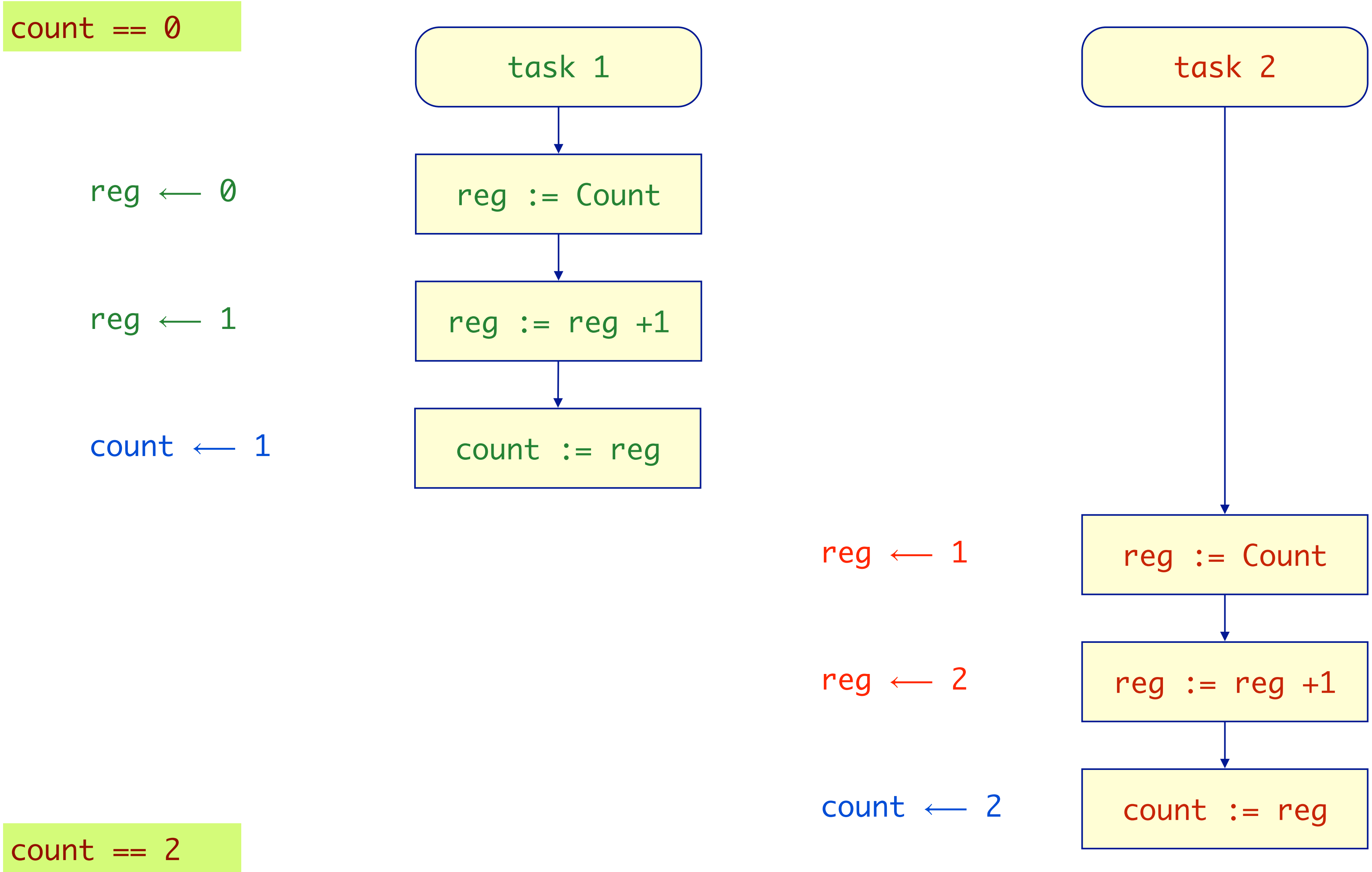
Shared counter (1)

count == 0



count == 1

Shared counter (2)



Race condition

- The result may be 1 or 2
  - depending on the respective speeds of task execution
    - "who runs faster"
- Such a program is incorrect
  - each execution is different
    - behaviour is not deterministic
  - the outcome of the program cannot be verified
- Solution: mutual exclusion
  - while one task is using the shared data, no other task can do
  - critical regions are mutually exclusive

Protected objects in Ada

- Protected objects encapsulate shared data and operations that can be carried out on them
- Mutual exclusion is automatically provided
- POs are instances of (possibly anonymous) protected types
- Protected types are declared in two separate parts
  - specification: visible interface of the protected objects
    - name of the type and, optionally, parameters
    - structure of protected data
    - specifications of protected operations
  - body: implementation of the protected operations

Example

```ada
type Operating_Mode is (Off, Safe, Nominal, Latency);

protected type Mode_Manager (Initial_Mode : Operating_Mode) is
   function  Current_Mode return Operating_Mode;
   procedure Set_Mode (New_Mode : Operating_Mode);
private
   Mode : Operating_Mode := Initial_Mode;
end Mode_Manager;

protected body Mode_Manager is
   function Current_Mode return Operating_Mode is
   begin
      return Mode;
   end Current_Mode;

   procedure Set_Mode (New_Mode : Operating_Mode) is
   begin
      Mode := New_Mode;
   end Set_Mode;

end Mode_Manager;
```

Example (continued)

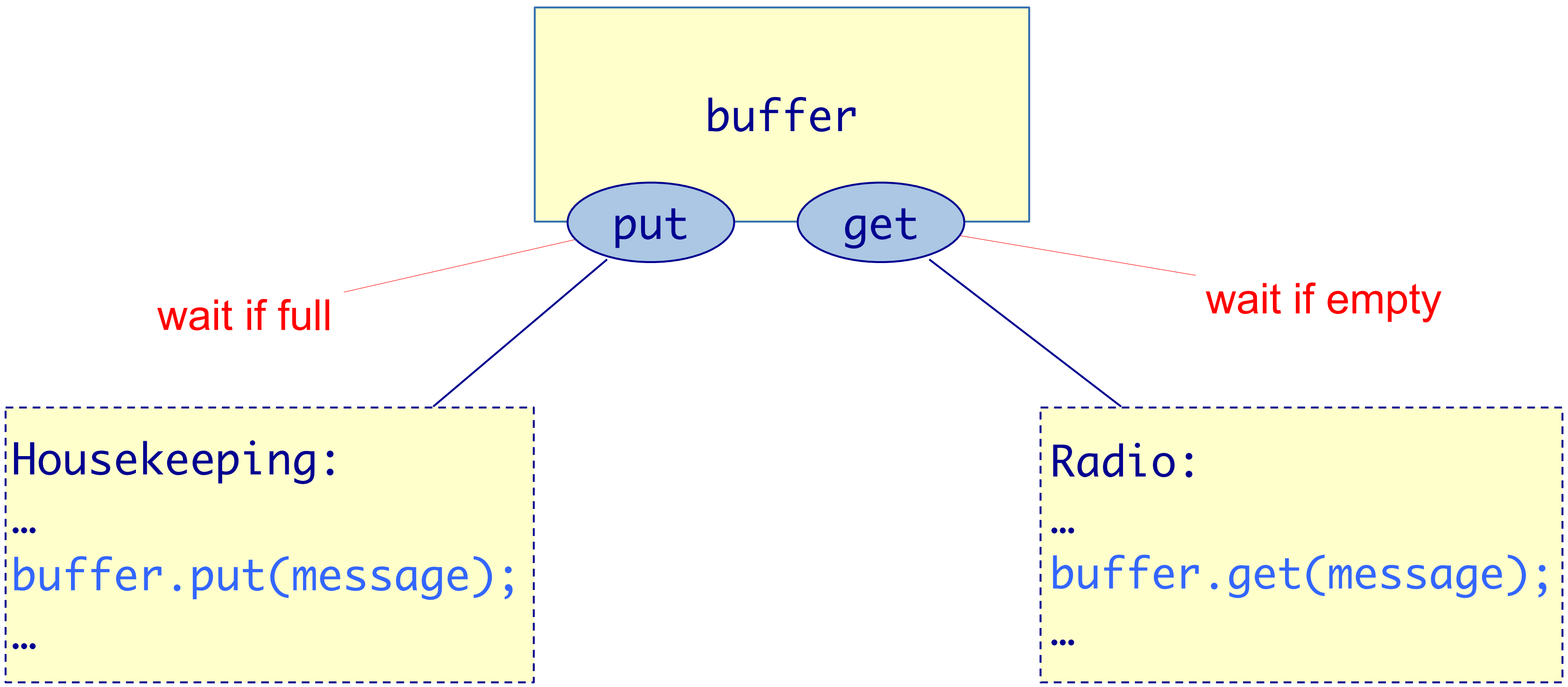
```
System_Mode : Mode_Manager (Off);

case System_Mode.Current_Mode is
   when Off => null;
   when Safe => ...;
   when Nominal => ...;
   when Latency => ...;
end case;

if (Battery_Low) then
   System_Mode.Set_Mode(Safe);
end if;
```

Conditional synchronisation

- Some times a task must wait for another task to fulfil some condition
  - e.g. message buffer

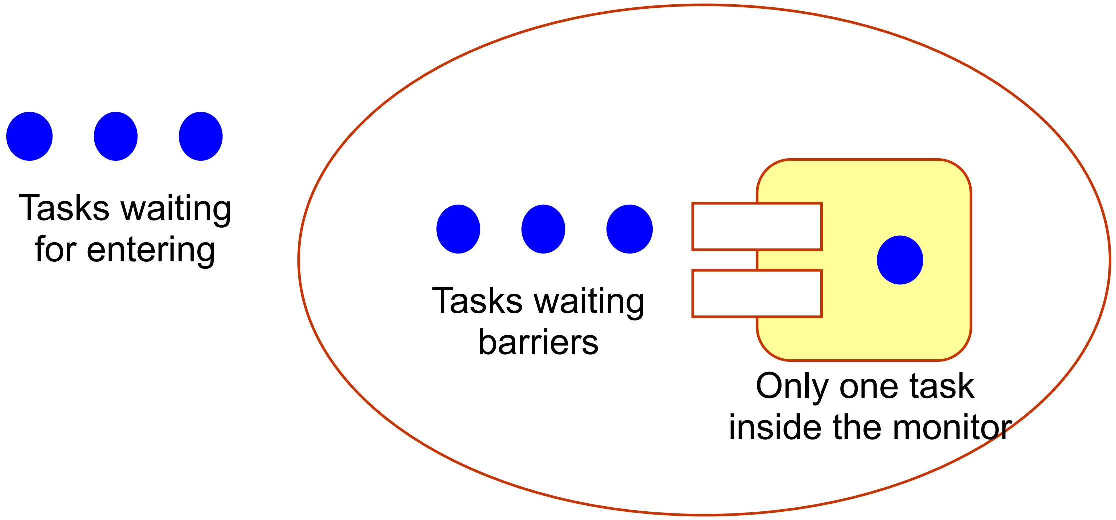


Conditions and entry barriers

- Conditional synchronisation is achieved by using special operations called entries
- Each entry has a barrier (a Boolean expression)
  - a task calling an entry with a closed (false) barrier is suspended
  - whenever another operation on the same object ends the barriers are re-evaluated and opened if true
  - one task waiting on a previously closed barrier is resumed
  - the resumed task executes the entry body

Mutual exclusion and barriers

- Tasks waiting on a barrier have priority for accesing the monitor
  - This approach avoids race conditions
- This model is called eggshell



Example: message buffer

```
type Message is ...;
Size : constant Positive := 32;
type Index is mod Size;
type Message_Queue is array (Index) of Message;

protected type Message_Buffer is   -- specification
   entry Put(M : in  Message);
   entry Get(M : out Message);
private
   First    : Index := Index’First;
   Last     : Index := Index'First;
   Number   : Natural := 0;
   Store    : Message_Queue;
end Message_Buffer;
```

Example: message buffer (continued)

```
protected body Message_Buffer is

   entry Put(M : in  Message) when Number < Size is
   begin
      Store(Last) := M;
      Last    := Last + 1;
      Number := Number + 1;
   end Put;

   entry Get(M : out Message) when Number > 0 is
   begin
      M := Store (First);
      First  := First + 1;
      Number := Number - 1;
   end Get;

end Message_Buffer;
```