This is still a “familiarize yourself” assignment without concrete deliverables other than convincing your tutor that you are on top of these concepts.

**Fixed Priority Scheduling (FPS)**

We will discuss many finer points of fixed priority scheduling later in the course, yet for the sake of this lab exercise, we can define fixed priority scheduling as a preemptive scheduling method which switches to one (or multiple, depending on available resources) runnable, higher priority task. Switching means here to preempt and suspend a currently lower priority task and hand over a CPU resource to a runnable, higher priority task. Tasks on the same priority are handled on a round robin basis, yet take note that there are no timers involved in this process and hence there is no “time-slicing” happening. Handing over a CPU to a same-priority task only happens if a task becomes blocked (or offers its own temporary suspension).

There are other scheduling methods used in real-time systems (which we will also discuss later in the course), yet FPS is a common, default starting point in real-time systems. You can also rely on the fact that every real-time system will provide you with a FPS method, hence we start with this scheduler.

**A few essential scheduling packages and pragmas in Ada**

**Priority types**

Valid priority values are defined in the package `System`. (The number of task and interrupt priorities is hardware/implementation dependent, yet the types defined there (e.g. `Priority`) are mandatory and the number of priorities shall be at least 30.) Also take note that a task which has not been assigned a priority by the program will be set to a mid-level priority (`Default_Priority`). All your tasks so far have been running on this same, mid-level priority. Higher numbers imply higher priorities, hence `Priority'Last` will be the highest priority in the system.

**Assigning priorities**

Priorities can be assigned statically by adding a pragma `Priority (<expression>)` inside a task definition (usually right after the line with the `task` name, but always before the according `end` statement). Thus a very simple priority assignment could look like this:

```ada
with System; use System;

(_:)
```

1 | ANU College of Engineering and Computer Science  August 2013
task Not_so_important is
  pragma Priority (Priority'First);
  (...) -- entries for this task come here
end Not_so_important;

(...)  

task Pretty_important is
  pragma Priority (Priority'Last);
  (...) -- entries for this task come here
end Pretty_important;

Priorities can also be dynamic (changed at run-time). This is achieved with the methods defined in Ada.Dynamic_Priorities. The package with its two, dead-giveaway-named procedures Set_Priority and Get_Priority speaks for itself. (If you have not yet found out how to look up standard packages, check Help -> GNAT Runtime inside your GPS environment.) Note the default task assignment, which means that if you do not specify which task’s priority you want to change, you will change the priority of the task this call is being executed in.

Don’t be too worked up by the confusing concept that you can change priorities in a ‘Fixed Priority Scheduling’ environment. In cases where priorities can change at runtime, the method will be referred to as ‘Priority Scheduling’. Dynamic priorities can be used to emulate any scheduling method (e.g. if your environment has not already defined the scheduler which you need), as they give you full control over the scheduling process. Certification processes usually ban its usage though, and in fact dynamic priorities are forbidden in Ada-pre-defined, high-integrity language subsets.

Hanging out for a bit

Tasks can also put themselves to sleep for a defined amount of time or until a specific absolute time. In high-integrity systems only the latter will be allowed, so don’t become too attached to the former. The Ada syntax for those statements is delay <relative-time-expression> and delay until <absolute-time-expression> respectively. As you can judge from the course title, you will be using those time related statements a lot later and in all possible contexts, so don’t spend too much time on those here, but rather focus on basic scheduling first. If you are already curious, then have a look at the package Ada.Real_Time. The <absolute-time-expression> for the delay until statement is of type Time as defined there. Note that the <relative-time-expression> is of type Duration (a subtype of Float) and is not defined in Ada.Real_Time, yet conversion routines between the real-time-type Time_Span and the general type Duration are provided there.

Switching between tasks on the same priority

Preempting tasks is a complex operation which should only be applied on a necessity basis in a real-time system. In other words: the concept of “provide all tasks with a fair share of CPU time just because it seems more fair” does not apply. Tasks will be only preempted based on priorities.

Yet sometimes tasks themselves will know that they reached a lesser critical part of their computation, and that it would be “all right” now to release the CPU if it could be made use of elsewhere in the system. This leads ultimately to “cooperative scheduling” (a scheduling method used in some high-integrity systems) which we will discuss later in the course as well.

A task may indicate that it is willing to be suspended if the CPU can be deployed for another, same-priority task at the moment by the statement delay 0.0, or slightly less clumsy by calling Yield from the package Ada.Dispatching. If there are currently more (or an equal number of) CPU cores available then runnable, same-priority tasks, those calls have no effect. Otherwise
the current CPU core will be handed over to the next (in a first-in-first-out fashion) runnable task on the same priority.

**Apply and experiment with all of the above**

Use the service-task and the multiple-queue, protected object example as a starting point to experiment with priority scheduling and its effects. You may want to add more tasks to explore the situation when your hardware actually runs out of available CPU cores. You probably already found that some of the examples will lead to unpredictable sequences on the terminal. While this is not a problem per se, you can nicely experiment here by influencing the scheduling behaviours. Keep in mind that alternating the scheduling is not the only way to control the timing and synchronization behaviour of a task set. After you changed the task behaviours by means of priorities and cooperation, try to reach the same effect by means of additional synchronization points (typically task or protected object entries). Blocking tasks until all other related tasks have reached the same stage is for instance a classical form of synchronization here. As a way to convince yourself that you understand all concepts before attempting the same on your tutor: always predict what you think will happen before you actually run a program … and make those tests interesting enough for yourself.