UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

# INFR09015 OPERATING SYSTEMS

**Thursday 1st May 2014**

**09:30 to 11:30**

Year 3 Courses

Convener: S. Viglas
External Examiners: A. Cohn, T. Field

## INSTRUCTIONS TO CANDIDATES

**Answer any TWO questions.**

**All questions carry equal weight.**

**CALCULATORS MAY NOT BE USED IN THIS EXAMINATION**

1. (a) Give **brief** (one or two sentence) definitions of the following four terms in memory management: *virtual memory*, *page table*, *secondary storage*, *segment register*. [*8 marks* ]

   (b) Imagine you are part of a team designing the virtual memory model for a new highly reliable architecture intended for use in safety- or service-critical systems. As there is no need for very large amounts of in-memory data, the architecture will be 32-bit. The amount of physical memory available will be limited by modern standards, owing to its expense. The architecture is intended to support a standard modern process and thread model.

      i. It has been decided that the memory model should be segmented, and you are asked how many and what kind of segments are appropriate. Give, and justify, your opinions, bearing in mind particularly the need for security and safety, as well as the need to make good use of physical memory. [*6 marks* ]

      ii. A colleague suggests that if you have segments, there is no need to have paging as well. Another colleague replies that if Intel provide both segmentation and paging, there must be a good reason to do it.
      Write a brief explanation for your colleagues of how segmentation and paging solve different issues, and give your opinion on whether paging is appropriate, and in what form. [*6 marks* ]

      iii. On the basis of your recommendations in parts (i) and (ii), describe how the bits of the 32-bit logical address will be interpreted by the hardware. [*5 marks* ]

2. (a) Briefly define the notion of *process*. Sketch the standard five-state diagram of process states and transitions, giving one-sentence definitions of each state and transition. [*8 marks* ]

   (b) Until recently, the Linux kernel code defined five possible values for the `state` field of the task structure:

   - `RUNNING`: the task is on the run queue
   - `INTERRUPTIBLE`: the task is waiting for an event, and may be scheduled earlier if a signal arrives
   - `UNINTERRUPTIBLE`: the task is waiting for an event, and cannot proceed until it happens
   - `ZOMBIE`: the task has exited, but its parent has not yet noticed
   - `STOPPED`: the task has been stopped by a signal, and cannot be scheduled until explicitly re-started

   For each state, discuss whether it corresponds to a state of part (a). Suggest reasons for the differences between the two state sets. [*7 marks* ]

   (c) The `UNINTERRUPTIBLE` state has long been a source of frustation: if a process is in an uninterruptible wait for an event which does not happen (e.g., waiting for a read from a file on a networked file system which has failed), the process remains stuck until system reboot.

   i. Why would the original designers have made the `UNINTERRUPTIBLE` state so absolutely uninterruptible? If it were not, what risks would arise? [*5 marks* ]

   ii. In modern Linuxes, an additional state `KILLABLE` has been added: this is like `UNINTERRUPTIBLE`, except that the process will respond to a `KILL` signal, which forces immediate termination. Why ought this be a sensible thing to add? Under what circumstances would it still be necessary to use `UNINTERRUPTIBLE`? [Hint: the circumstances are more to do with the history of kernel code than with external events.] [*5 marks* ]

3. (a) Give Peterson's algorithm for mutual exclusion, and justify informally its correctness (namely, that it enforces mutual exclusion and does not deadlock or livelock). [*10 marks* ]

   (b) Suppose that an operation `set(b)` is available which atomically sets a boolean variable `b` to `true`: that is, `set` returns `true` only if it changed `b` from `false` to `true`, and returns `false` if `b` was already `true`. Suppose that also that an operation `waitfor(b)` is provided, which will sleep until `b` is set true by another process calling `set`.

   Using pseudo-code as necessary, show how to implement a safely shared stack between two processes. You may assume standard stack routines `push(item)`, `pop()`, `isempty()` to manipulate the raw stack; your task is to provide routines `safepush()` etc. which wrap the stack routines in mutual exclusion. You may assume that `pop()` returns `NULL` if and only if called on an empty stack, but your `safepop()` must not return `NULL`: if called on an empty stack, it should wait until an item is `safepush()`ed on to the stack, and then pop that. [*10 marks* ]

   (c) Consider the problem of part (b), and suppose that a bug in one of the raw stack routines causes a process to be terminated while changing the raw stack. What effect will this have on the other process, and what might be done to avoid it? [*5 marks* ]