USTH – ICT department – Teaching Unit 1.3

Systems and Networks

Acedemic year 2011-2012

Daniel Hagimont
Abderrahim Benslimane

Duration: 3 hours – Documents are not accepted during the exam

The two parts of the exam (SYSTEMS and NETWORKS) have to be produced on separate documents.

SYSTEMS

Question 1

In this exercice, we rely on Hoare monitors as presented in the lectures (this means priority to the signal receiver and implicit locking of the monitor).
We want to control the lifecycle of a software component (like a box). This component can be stopped and started. If the component is in the stopped state, there must not be any activity running inside the component.
The monitor which implements such a lifecycle for a component is composed of the following procedures:

- *enter()* : executed by each activity which wants to enter the component (must block if the component is stopped, until the component is started).
- *exit()* : executed by each activity which wants to exit the component.
- *stop()* : executed by an activity which wants to stop the component. This procedure returns only when there isn't any activity running in the component (must block if activities are running inside).
- *start()* : executed by an activity which wants to start the component. The *start()* procedure is always called by the activity which previously stopped the component.

We assume here that only one activity can invoke the *stop()* procedure (i.e. *stop()* cannot be invoked while the component is stopped).
You must program this monitor. Clarity is an important evaluation criteria.
Indication: the problem is very similar to a reader/writer scheme, you can implement it with two conditions: *start* and *stop*.

Question 2

We now consider that several activities can invoke *stop()* (i.e. *stop()* can be invoked while the component is stopped). In such a case, the second call to *stop()* is suspended until the former completes (and will resume the later).
You must update the previous monitor in order to address this requirement.
Indication : you can implement it with an additional condition: *stopstop*.

Question 3

We consider a virtual memory management system which relies on the following structures:

```
mem_addr_t PageTable[NB_PAGE];
disk_addr_t SwapTable[NB_PAGE];
```

mem_addr_t is an address in memory and disk_addr_t is an address on disk.
We assume that a page is either in memory or in the swap (on disk), else there's an addressing error.
PageTable is the page table of the virtual address space we have to manage (we manage only one virtual address space). SwapTable gives for each virtual page its location in the swap if the page was swapped out.
You have to implement the *void page_fault(int npage)* procedure which is invoked by the hardware when the *npage* entry in the page table is null.
To implement this procedure, you can use the following procedures (i.e. they are **available**, you **don't have to implement them**):

- **void memory_error()**: have to be called when a memory error is detected.
- **mem_addr_t memory_alloc()**: allocate a page in main memory. Return the address of the page in memory, null if memory is full.
- **int lru_select()**: select a page for replacement. Return the page number in PageTable. Be carefull, it does not return a page address in memory. The address of the page in memory (which is selected for replacement) has to be fetched from PageTable.
- **disk_addr_t swap_alloc()**: allocate a page in the swap. Return the address of this page on disk.
- **void swap_free(disk_addr_t disk_page)**: free a page from the swap.
- **void load_page(mem_addr_t mem_page, disk_addr_t disk_page)**: load a page from the swap into a page in memory.
- **void store_page(mem_addr_t mem_page, disk_addr_t disk_page)**: store a page from memory into a page in the swap.

NETWORKS

blablabla

Question 4

blablabla

Question 5

blablabla

Question 6

blablabla