System and device programming
14 September 2011

(Theory: no textbooks and/or course material allowed)

(15 marks) The final mark is the sum of the 1st > 8 and the 2nd part > 10

The final mark cannot be refused, it will be registered (no retry for marks >= 18)

1. (3.0 marks) List and explain the steps, the data structures, and the processes involved in the Unix OS **to make the login windows or prompt appear in your screen**. List the steps, the files and the processes involved **during** the login to finally obtain your shell prompt.

2. (3.0 marks) What is thrashing? Which are its causes? It is possible for the system to detect and eliminate thrashing?

3. (4.0 marks) Complete/modify the following routines for push/pop from a stack, in order to work on a multithreaded Win32 environment, where multiple threads can use a shared stack (for both push and pop operations).

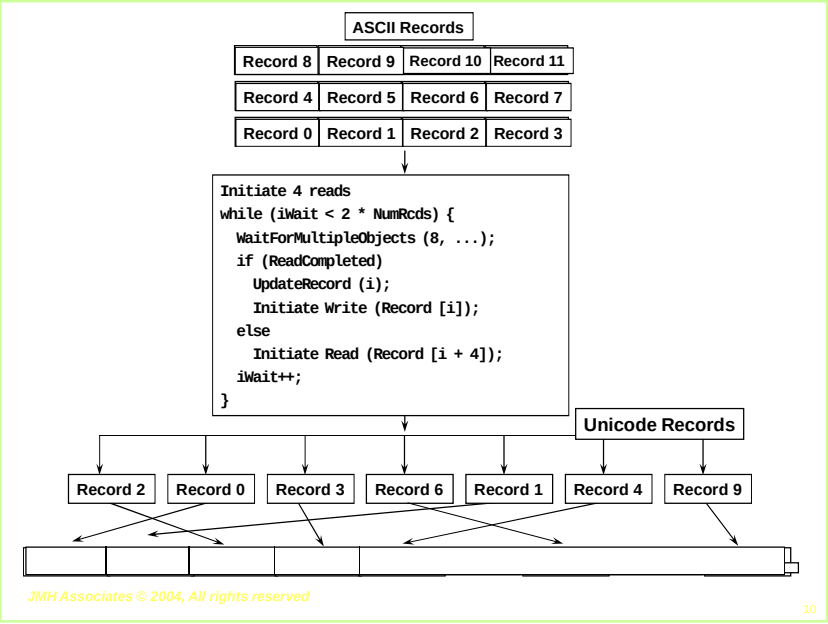
```
typedef struct {
  LPDWORD data; /* array of stack items */
  int index;    /* index to the stack top: for insertion */
  int size;     /* size of the stack (max. number of slots available */
  ...
} stack_t;
...
BOOL push (stack_t *s, DWORD item) {
   ...
   s->data[s->index++] = item;
   return TRUE;
}
BOOL pop (stack_t *s, LPDWORD itemp) {
   ...
   *itemp = s->data[--s->index] = item;
   return TRUE;
}
```

   Synchronization data should be added to the **`stack_t`** struct, in order to guarantee correct access in mutual exclusion. Push and pop should be implemented in two ways:
   - blocking, so that the requesting thread will wait in case of stack not available.
   - Non blocking, with function returning TRUE upon correct completion, FALSE if stack not available.

   Is deadlock possible (motivate the answer) ? If yes, provide a solution to avoid it.

4. (2 marks) Describe the main features of heaps in Win32. Why can we get benefits from using multiple heaps, with respect to standard memory management in libc? Which are the main functions for heap management in Win32 ?

5. (3 points) Explain the pseudo-code shown in figure, taken from concurrent ASCII-Unicode conversion of a file, using asynchronous I/O. In particular, describe how to implement concurrent updates using overlapped structures. Why is the main loop bounded **`by 2*NumRcds`** ? What does **`WaitForMultipleObjects`** wait for ?
   Is it possible for record #21 to be written on file before record #16 ? and before record #17 ? *(motivate your answers)*



System and device programming
14 September 2011

(C program: textbooks and/or course material allowed)

(18 marks) The final mark is the sum of the 1st > 8 and the 2nd part > 10
The final mark cannot be refused, it will be registered (no retry for marks >= 18)

Write a concurrent C program in the **UNIX** environment which creates **K** threads (**K** given as the first command line argument).

Each thread instance belongs to a randomly selected class among *C1, C2,* and *C3*. The index of the class *i* defines the priority class of the thread, 3 being the maximum priority.

Priority means that when a thread finishes using its critical region, and there are one or more threads waiting to enter their critical regions, **the waiting thread with highest priority has to be unblocked**. The threads must implement this rule using normal semaphores and global variables.

A thread **cyclically** makes a request to access its critical region, and waits a random time (between 2 and 10 seconds) before issuing the next access request.

The thread sends on a pipe, shared with the main thread, its relevant status information, i.e. "I'm requesting access to my critical region, or I'm inside my critical region". Thus the status is defined as the triple

`<thread_identifier, priority_class, N>`

where **N** is an integer (**`N=0`** when the thread is requesting access to its critical region, **`N=1`** the thread is inside its critical region).

The main thread waits on the pipe the sent information sent by the created threads and **prints the received information and the current time in a log file** (name given as the second argument of the command line), so that one can assess the correct behaviour of the concurrent threads.