Linköpings universitet
IDA Department of Computer and Information Sciences
Prof. Dr. Christoph Kessler

TENTAMEN / *EXAM*

TDDB68

Processprogrammering och operativsystem /
Concurrent programming and operating systems

18 mar 2015, 14:00–18:00, TER1, TER2, TER3, TERE

Jour: Christoph Kessler (070-3666687, 013-282406); visiting ca. 16:00

Hjälpmedel / *Admitted material:*

– Engelsk ordbok / *Dictionary from English to your native language*;
– Miniräknare / *Pocket calculator*

General instructions

- This exam has 8 assignments and 5 pages, including this one.
  Read all assignments carefully and completely before you begin.
- Please **use a new sheet of paper for each assignment**, because they will be corrected by different persons.
  Sort the pages by assignments and number them consecutively.
- You may answer in either English or Swedish. **English is preferred** because not all correcting assistants understand Swedish.
- Write clearly. Unreadable text will be ignored.
- Be precise in your statements. Unprecise formulations may lead to a reduction of points.
- Motivate clearly all statements and reasoning.
- Explain calculations and solution procedures.
- The assignments are *not* ordered according to difficulty.
- The exam is designed for 40 points. You may thus plan about 5 minutes per point.
- **How much to write?** No general policy, but as a rule of thumb: Questions for 0.5p can typically be answered properly in a single line. Correct and concise answers to questions for 1p usually require a few lines. Code and figures should be commented properly.
- Grading: U, 3, 4, 5. The preliminary threshold for passing is 20 points.

1. (6 p.) **Interrupts, processes and threads**

   (a) Define the terms *process* and *thread*.
   In particular, what are the main differences between processes and threads, and what do they have in common? Be thorough! (2p)

   (b) Why do user-level threads (in contrast to kernel-level threads) promote portability of applications? (1p)

   (c) Write a simple Unix-style program (pseudocode using appropriate system calls) that *spawns exactly two (2) child processes*, each of which shall write `''Hello World''` to the standard output, and that writes `''Goodbye''` after the two child processes have terminated. Explain your code. (2p)

   (d) Two main methods for inter-process communication in a computer are *shared memory* and *message passing*. Which of the two methods is likely to have less overhead if two processes communicate frequently with each other, and why? (1p)

2. (6 p.) **CPU Scheduling**

   (a) Given a single-CPU system and the following set of processes with arrival times (in milliseconds), expected maximum execution time (ms), and priority (1 is highest, 5 is lowest priority).

Process	Arrival time	Execution time	Priority (as applicable)
P_1	0	5	5
P_2	2	3	4
P_3	4	2	2
P_4	7	3	3
P_5	9	2	1

   For each of the following scheduling algorithms, create a Gantt chart (time bar diagram, starting at $t = 0$) that shows when the processes will execute on the CPU. Where applicable, the time quantum will be 3 ms. Assume that a task will be eligible for scheduling immediately on arrival. If you need to make further assumptions, state them carefully and explain your solution. (5p)

   (i) FIFO;

   (ii) Round-robin;

   (iii) Shortest Job First *without* preemption;

   (iv) Priority Scheduling *without* preemption.

   (v) Priority Scheduling *with* preemption.

   (b) There exist several strategies for CPU scheduling on a multiprocessor system. One of them is *affinity-based scheduling*. How does it work (basic idea, no details) and what is the motivation for it? (1p)

3. (6 p.) **Synchronization**

Electronic voting server

An electronic voting system uses as central data structure a global array

```
unsigned int votes[N];    // N = number of parties
```

for counting the number of votes for each of the N parties, initialized to zeroes.

The voting server program, originally single-threaded, processes one vote request received from an external client at at time, by calling the function

```
void cast_vote ( unsigned int i )
{
  votes[i] = votes[i] + 1; // count vote for party i
}
```

In order to scale up to many millions of voters, the voting program should be multi-threaded (with the `votes` array residing in shared memory of the voting server process) and run on a multicore server running a modern multithreaded operating system with preemptive scheduling, so that multiple calls to `cast_vote` can execute concurrently. It is your task to make the program thread-safe and preserve its correct behavior.

(a) Using a simple contrived scenario (Hint: use $N = 2$ parties and few votes), show with a concrete example (interleaving of shared memory accesses) that, without proper synchronization, race conditions are possible that could even cause that the wrong party wins the election. (1p)

(b) Identify the critical section(s) and protect the program against race conditions with a *single* mutex lock. Show the (pseudo)code. (1p)

(c) Now extend your implementation of (b) by adding the shared global variable

```
int current_leader = 0;  // init. to 0 at program startup
```

that shall, at any time, contain the *index* of the currently leading party. Routine `cast_vote` needs to be extended to check if the just incremented vote counter exceeds `votes[current_leader]` and, if so, update `current_leader` accordingly. Show the resulting `cast_vote` pseudocode and explain how this change affects the critical section compared to (b). (1p)

(d) Now add a *separate* thread that is responsible only for continuously monitoring `current_leader` and, in case of a change in the leader, updating a graphical display of the *currently leading party*. A naive solution using busy polling of `current_leader` could use a thread function like this one:

```
void *leader_monitoring ( void * arg )
{
 int last_observed_leader = 0;
 while (1) { // endless loop for continuous monitoring:
    if (current_leader != last_observed_leader) { // change:
       last_observed_leader = current_leader;
       update_display( current_leader );
    }
 }
 return NULL;
}
```

Suggest a more suitable solution (using a special synchronization construct, *mention its name and explain what it does*) such that the leader-monitoring thread does not waste CPU time by busy polling of `current_leader`, and such that no display updates are done that any new vote doesn't make necessary. (Assume that it is not critical if the update of the leader visualization is delayed a bit, or if a short-time change in the leader and back is not displayed.) *Show and explain* the resulting modified pseudocode of `cast_vote` and of `leader_monitoring`. (3p)

4. (5 p.) **Deadlocks**

   (a) There are four conditions that must hold for a deadlock to become possible. Name and describe them briefly. (2p)

   (b) You are given a system with 4 types of resources, A, B, C and D. There are 4 instances of A, 5 instances of B, 1 instance of C and 7 instances of D. Currently, 5 processes P_1...P_5 are running, and for each process, the resources currently held and its total maximum resource need (including the already held ones) for each type are given as follows:

Process	Already held	Maximum total need
	A B C D	A B C D
P_1	0 1 0 1	0 2 0 4
P_2	0 0 0 0	4 3 0 5
P_3	1 2 1 0	2 2 1 3
P_4	1 0 0 2	2 2 0 4
P_5	1 0 0 2	2 0 1 4

   I.e., currently, process P_1 holds already one B and one D out of the 2 Bs and 4 Ds that it eventually may need in the worst case, etc. One A is currently still available, etc.

   (i) Show that the system is currently in a safe state (calculation). (1.5p)

   (ii) In the situation given above, process P_1 now asks for 1 instance of D, in addition to the B and D that it already has. Is it safe to grant the request? Why or why not? (calculation) (1.5p)

5. (8 p.) **Memory management**

   (a) Define the term *external memory fragmentation* and give an example of a memory management technique (technical term, no details) that completely avoids external fragmentation. (1p)

   (b) Consider a page-based virtual memory system with a page size of 1024 bytes where virtual memory addresses have 32 bit size. If using *multi-level paging*,

      i. determine how many levels of paging are required, and describe the structure of the virtual addresses (purpose, position and size of its bit fields); (1.5p)

      ii. explain (annotated figure) how in this case the physical address is calculated by multi-level paging from a virtual address; (1p)

      iii. show how a TLB can be used to accelerate address calculation. (1p)

   (c) Explain how the possibility of sharing memory pages among multiple processes can help to speed up the start-up of a child process at a `fork` system call. (1.5p)

   (d) What is *thrashing* in a virtual memory system? How does it occur? And what can be done about it? (2p)

6. (5 p.) **File systems**

   (a) Can a file system create a *hard link* to a file in a mounted file system? Justify your answer! (1p)

   (b) What information is usually contained in a *file control block* (FCB)? (At least 4 different items are expected) (1p)

   (c) Describe one technique to extend indexed allocation for large files. (1p)

   (d) How does the file system implementation keep track of unused disk space? Sketch one possible technique for free-space management. (1p)

   (e) What is the purpose of *disk scheduling*? (1p)

7. (2 p.) **OS Structures and Virtualization**

   (a) Give one of the main disadvantages of strict layering (with more than just very few layers) in operating systems. (0.5p)

   (b) What does a *hypervisor* (also known as *virtual machine monitor, VM implementation*) do?

      Illustrate your answer with a commented figure that shows where the hypervisor is positioned in the system software stack and with which other system entities it interacts. (1.5p)

8. (2 p.) **Protection and Security**

   (a) What is an *access control list (ACL)*? What does it contain, and how is it used? (1p)

   (b) How does memory segmentation support protection? In particular, which kind of security attack could it help to prevent? (1p)

Good luck!