Information page for written examinations at Linköping University

Examination date	2016-08-22
Room (1)	TER2
Time	14-18
Course code	TDDB68
Exam code	TEN1
Course name **Exam name**	Concurrent Programming and Operating Systems (Processprogrammering och operativsystem) Examination (Tentamen)
Department	IDA
Number of questions in the examination	8
Teacher responsible/contact person during the exam time	Christoph Kessler
Contact number during the exam time	013-282406
Visit to the examination room approximately	15:30
Name and contact details to the course administrator (name + phone nr + mail)	Elin Brödje, 013-284767, Elin.Brodje@liu.se
Equipment permitted	Engelsk ordbok / english dictionary, Miniräknare / pocket calculator
Other important information	No exam review for re-exams. After reporting, exams will be archived in the IDA student expedition in the E house where they can be inspected on request. Due to assistants being on travel/on leave in the coming weeks we expect that the grading will be ready around 12 september.
Number of exams in the bag	

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

TENTAMEN / *EXAM*

TDDB68

Processprogrammering och operativsystem / *Concurrent programming and operating systems*

22 aug 2016, 14:00–18:00, TER2

Jour: Christoph Kessler (070-3666687, 013-282406); visiting ca. 15:30

Hjälpmedel / *Admitted material:*

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

General instructions

- This exam has 8 assignments and 6 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.

  Due to assistants being on travel/on leave in the coming weeks we expect that the grading will be ready around 12 september.

1. (8 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) Draw the general life cycle diagram (finite state machine) for a process in a system with *preemptive scheduling*, as introduced in the lecture. For each state (node) explain shortly what it represents. For each possible state transition (arrow) annotate which event(s) trigger(s) the transition. (2p)

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

   (d) 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)

   (e) 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. (5 p.) **CPU Scheduling**

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	6	5
P_2	2	3	2
P_3	4	1	4
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 2 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.

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

A *barrier synchronization* is a function that does not return control to the caller until all p threads of a multithreaded process have called it.

A possible implementation of the barrier function uses a shared `counter` variable that is initialized to 0 at program start and incremented by each barrier-invoking thread, and the barrier function returns if `counter` has reached value p.

We assume here that p is fixed and can be obtained by calling a function `get_nthreads()`, that load and store operations perform atomically, and that each thread will only call the barrier function once.

The following code is given as a starting point:

```
static volatile int counter = 0;   // shared variable

void barrier( void )
{
  counter = counter + 1;
  while (counter != get_nthreads())
     ;  // busy waiting
  return;
}
```

(a) Show by an example scenario with $p = 2$ threads (i.e., some unfortunate interleaving of thread execution over time) that this implementation of `barrier` may cause a program calling it (such as the following) to hang. (0.5p)

```
void main( void )
{
 ... // create p threads in total
 ...
 barrier();
 ...
}
```

(b) Identify the critical section(s) in this implementation, and use a *mutex lock* to properly protect the critical section(s), without introducing a deadlock. Show the resulting C code. (1.5p)

(c) Today, many processors offer some type of atomic operation(s). Can you use here an atomic *fetch-and-add* operation instead of the mutex lock to guarantee correct execution? If yes, show how to modify the code above and explain. If not, explain why. (1.5p)

(continued on next page...)

(d) The counter-based barrier solution as given above can only be used once in a program execution. Why?

Suggest a way to generalize the above solution (properly synchronized) so that it works even if there occur *several* barrier synchronizations in the same program, such as in

```
void main( void )
{
 ... // create p threads
 ...
 barrier();
 ...
 barrier();
 ...
}
```

Explain your solution, and motivate why it works correctly and will not hang. (2.5p) *(Hint: Is a single counter variable sufficient? Two?)*

4. (8 p.) **Memory management**

(a) For a paging system with a page size of N bytes, what is the maximum amount of internal fragmentation that can occur when allocating memory for a process? (0.5p)

(b) Consider a page-based virtual memory system with a page size of $2^{10} = 1024$ bytes where virtual memory addresses have 32 bit. 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. When using a TLB to accelerate address calculation, calculate the expected time for a paged memory access if a physical memory access costs 100ns on average and a TLB access costs 1ns, and the TLB hit rate is 90%. (1p)

(c) Explain how paging supports sharing of memory between processes. (1p)

(d) LRU is a popular strategy for page replacement in virtual memory. However it is just a heuristic technique. What is the (theoretical) *optimal* page replacement strategy, why is it not applicable in practice, and how does it differ from LRU? (2p)

(e) Why is it useful for a virtual memory system to be able to estimate the current *working set sizes* of all executing processes? (1p)

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

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

   (b) Most current operating systems do not implement the Banker's algorithm for deadlock avoidance but instead shift this task to the application programmer. Name 2 limitations of the Banker's algorithm that are the main reason for this. (1p)

   (c) How can the occurrence of a deadlock be *detected* when only one instance of each resource type exists in a system? (1p)

6. (4 p.) **File systems**

   (a) Does a *soft link* to the file `exam.pdf` still work after the command `mv exam.pdf archive/exam.pdf`? Why or why not? (1p)

   (b) Can, in principle, the same file be opened by multiple processes?
   If yes, explain (draw a commented figure) how the internal data structures for opened files in the operating system provide this possibility.
   If not, explain why it is not possible. (2p)

   (c) Why is the management of unused disk space by the file system more complicated with contiguous file allocation? (1p)

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

   (a) What is the main idea of the *microkernel* approach to OS structuring, what is its main advantage and what is its main drawback? (2p)

8. (3 p.) **Protection and Security**

   (a) The *Heartbleed* bug discovered 2014 in OpenSSL was a so-called *buffer-overread* vulnerability. What is a buffer-overread vulnerability, and how could an attacker benefit from exploiting such a vulnerability? (principle only, no details of OpenSSL) (1p)
   Given a segmented memory system, could it be prevented by careful setting of segment access rights? Explain why or why not. (1p)

   (b) How can using *virtual machines* increase the security of a system? (1p)

Good luck!