Försättsblad till skriftlig tentamen vid Linköpings Universitet

(fylls i av ansvarig)

Datum för tentamen	4 jun 2013
Sal	TER1
Tid	14-18
Kurskod	TDDB68
Provkod	TEN1
Kursnamn/benämning	Processprogrammering och operativsystem
Institution	IDA
Antal uppgifter som ingår i tentamen	9
Antal sidor på tentamen (inkl. försättsbladet)	7
Jour	Christoph Kessler, 013-282406, 0703-666687
Besöker salen ca kl.	16:00
Examinator/kursansvarig	Christoph Kessler, IDA
Kursadministratör **(namn + tfnnr + mailadress)**	Carita Lilja, 013-281463, carita.lilja@liu.se
Tillåtna hjälpmedel	Engelsk ordbok, miniräknare
Övrigt **(exempel när resultat kan ses på webben, betygsgränser, visning, övriga salar tentan går i m.m.)**	

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

TENTAMEN / *EXAM*

TDDB68

Processprogrammering och operativsystem /
Concurrent programming and operating systems

4 jun 2013, 14:00–18:00

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 9 assignments and 6 pages, including this one.
  Read all assignments carefully and completely before you begin.
- It is recommended that you use a new sheet of paper for each assignment, because they will be corrected by different persons.
  Sort the pages by assignment, number them consecutively and mark each one on top with your exam ID and the course code.
- 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.
- Grading: U, 3, 4, 5. The preliminary threshold for passing is 20 points.

1. (7 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) We discussed different thread models with different relations (mappings) between user and kernel threads. Describe one of these models that is appropriate for use on a *multiprocessor architecture*, and explain why. (1.5p)

   (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. (2.5p)

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	7	5
P_2	1	5	3
P_3	4	1	4
P_4	7	3	2
P_5	11	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.

   (b) What is the purpose of
   (i) the long-term / medium-term scheduler and
   (ii) the short-term scheduler in an operating system? (1p)

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 that p 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++;
  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
 ...
 barrier();
 ...
}
```

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

(c) Can you guarantee correct execution without using mutex locks, by using atomic *fetch-and-add* instead? If yes, show how to modify the code above and explain. If not, explain why. (1.5p)

(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? A correct solution that works for an arbitrary number of barriers using the minimum number of counters gets a +1p bonus.)*

4. (2 p.) **More synchronization**

   (a) Explain how mutual exclusion synchronization can interfere with priority based scheduling. (2p)

5. (3 p.) **Deadlocks**

   (a) Consider the following pseudocode:

```
mutex_lock_t l1, l2, l3;

void T1( void )
{
 mutex_lock( &l1 );
 mutex_lock( &l2 );
 ...
 mutex_release( &l2 );
 mutex_release( &l1 );
 mutex_lock( &l3 );
 ...
 mutex_release( &l3 );
}

void T2( void )
{
 mutex_lock( &l2 );
 mutex_lock( &l3 );
 ...
 mutex_release( &l3 );
 mutex_release( &l2 );
}

void T3( void )
{
 mutex_lock( &l3 );
```

```
  mutex_lock( &l1 );
  ...
  mutex_release( &l1 );
  mutex_release( &l3 );
}

void main ( void )
{
  create 3 threads that execute T1(), T2(), T3() respectively
}
```

There are 3 threads that concurrently execute the functions `T1`, `T2` and `T3` respectively, which need to acquire and release mutual exclusion locks in order to perform their work (...).

Is this program deadlock-free?

If yes, give a formal argument why.

If not, give a formal argument why, and a counterexample. (2 p)

(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)

6. (8 p.) **Memory management**

   (a) Consider a page-based virtual memory system with a page size of 256 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); (1p)

      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)

      iv. calculate the expected time for a paged memory access if a physical memory access costs 100ns on average and a TLB access costs 5ns, and the TLB hit rate is 80%. (0.5p)

   (b) Explain how segmented virtual memory supports sharing of memory segments between processes. (1p)

   (c) Given a virtual memory system with 4 page frames, how many page faults occur with the *Least-Recently Used* replacement strategy when pages are accessed in the following order:

      1, 2, 3, 4, 5, 1, 3, 4, 2, 3, 1, 5, 4.

      (Justify your answer. Just guessing the right number is not acceptable.) (1.5p)

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

7. (3.5 p.) **File systems**

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

   (b) Where is the FCB contents stored after a file has been opened? (0.5p)

   (c) Is the file allocation method *indexed allocation* susceptible (prone) to *external* fragmentation? Give a short explanation. (1p)

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

8. (1.5 p.) **OS Structures and Virtualization**

   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)

9. (3 p.) **Protection and Security**

   (a) Name two different (software or hardware) measures to prevent *buffer-overflow* vulnerabilities. Explain briefly (1 line each) how they work. (1p)

   (b) How does memory segmentation support protection? (1p)

   (c) Why can, in general, the *microkernel* approach increase the security of a system compared to a traditional OS structure? (1p)

Good luck!