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

TENTAMEN / *EXAM*

TDDB68

Processprogrammering och operativsystem /
Concurrent programming and operating systems

24 aug 2015, 14:00–18:00, 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 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.

1. (7.5 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) We introduced several multithreading models (i.e., mapping between user-level and kernel-level threads). Describe one model that is appropriate for multiprocessor systems, and explain why. (1p)

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

   (d) Two main methods for inter-process communication in a computer are *shared memory* and *message passing*.

   For each of them, give a short explanation of how it works and how the operating system is involved, i.e., which important system calls are to be used and what they do.

   Which of the two methods is likely to have less overhead if two processes communicate frequently with each other, and why? (2.5p)

   (e) How does a computer system prevent a user process from executing privileged instructions? What hardware support is required for that? (1p)

2. (7 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	6	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;

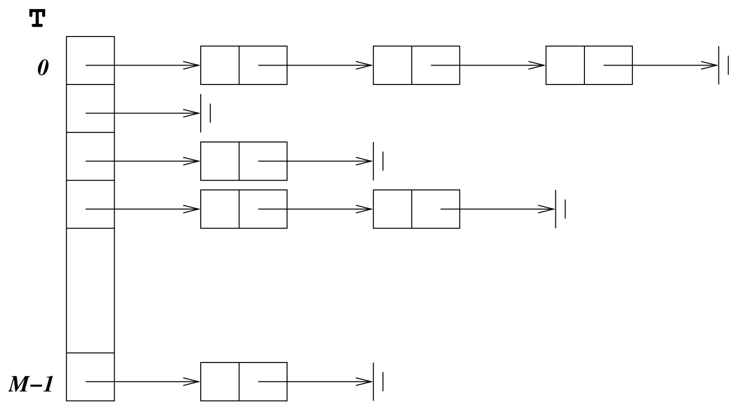
   (iv) Priority Scheduling *without* preemption.

   (v) Priority Scheduling *with* preemption.

   (b) Strictly priority-based schedulers introduce the problem of process *starvation*. What does this mean? Shortly describe one technique (there exist several ones) that can reduce or completely remove this starvation problem. How does this technique interfere with the priority mechanism? (2p)

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

Consider an ordinary *hash table* T of size M designed for a single-threaded environment. A hash function h that maps data items to the range $\{0, ..., M-1\}$ is given. Data items u, v hashed to the same hash value $h(u) = h(v) = i$ are chained in a singly-linked list, such that list $T[i]$ contains all items with hash value i (see the picture). Each list item contains a data element and a `next` pointer to the next item in the list (NULL for the last element in a list). Each $T[i]$ points to the first item in its list (or $T[i]$ is NULL if the ith list is empty), $i = 0, ..., M-1$.



A sequential implementation is given as follows. List elements are represented by the following data type:

```
struct htelem {
   struct dataitem *item;
   struct htelem *next;
};
```

The operation `insert`(T, u) inserts a data item u into hash table T at the beginning of list $T[h(u)]$:

```
void insert ( struct htelem *T[], struct dataitem *u )
{
  int i = h(u);
  struct htelem *e = (struct htelem *) malloc(sizeof(struct htelem));
  e->item = u;
  e->next = T[i];
  T[i] = e;
}
```

The boolean operation `lookup`(T, u) checks whether item u is currently stored in T or not:

```
int lookup ( struct htelem *T[], struct dataitem *u )
{
  int i = h(u);
  struct htelem *p;
  for (p = T[i]; p!=NULL; p = p->next )
     if (equal( u, p->item ))
        return 1;
  return 0;
}
```

where the boolean function equal(u, v) tests for equality of two data items u and v.

(a) Give an example scenario of a *race condition* with two threads that concurrently insert items into an unprotected shared hash table on a multi-tasking single-processor system with preemptive scheduling. I.e., give two different interleavings of the execution of two threads that lead to incorrect behavior, possibly even to a corrupted data structure or a run-time error. (1p)

Your task will now be to make the hash table implementation thread-safe to allow a hash table stored in shared memory to be accessed properly by multiple threads:

(b) Identify the critical section(s) in your pseudocode and suggest how to protect them properly against race conditions by using a single mutex lock. Show the revised (pseudo) code. (2p)

(c) What is the disadvantage of a single-lock solution if there are many threads accessing T frequently, and why? (0.5p)

Describe a multi-lock approach that solves that problem. Why is the multi-lock approach better? (1p)

(d) Threads executing an `insert` operation modify the status of T, while threads executing a `lookup` do not change it. Assume that `lookup` operations occur much more often than `insert` operations. Suggest a lock-based protection technique that exploits this fact to improve the degree of concurrency compared to your previous solution. (No details, no code. Just give the technical term and the basic idea how it works.) (1p)

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

(a) Consider a page-based virtual memory system with a page size of $2^9 = 512$ bytes where virtual memory addresses have 30 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. show how a TLB can be used to accelerate address calculation. (1p)

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

(c) Characterize an important property of pages that are particularly suitable to choose as a victim when freeing a used frame in order to reduce page replacement overhead. Explain why. What kind of data structure could help with this choice? (1.5p)

(d) What is the *working set size* of a process? (1p)

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

(a) What is the difference between *deadlock* and *starvation*? (1p)

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

void main ( void )
{
 initialize mutex locks l1, l2, l3;
 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)

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

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

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

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

   (d) What is the basic idea and motivation of the Unix *inode* structure? (1.5p)

   (e) Sometimes it may become necessary to run a file system consistency check. What does ”inconsistency” of a file system mean, and what could possibly have caused it? (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. (2 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)

Good luck!