TENTAMEN / *EXAM*

TDDB68

Processprogrammering och operativsystem / *Concurrent programming and operating systems*

2018-08-27

Examiner: Mikael Asplund (0700895827)

Hjälpmedel / *Admitted material:*

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

General instructions

- This exam has 6 assignments and 8 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.

Good luck!

1. **Multiple choice quesions (10p)**

Below are 10 multiple choice questions. Please answer them by removing the last page of the exam, fill the appropriate boxes, and hand it in together with the rest of your answers. Please note that there might be more than one correct option. Each question below can give 0 or 1 point(s), and to get 1 point you must have identified exactly the right set of choices.

(a) Consider the program below, called *fork-test*.

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
  int pid1 = 0;
  pid1 = fork();
  int pid2 = 0;
  pid2 = fork();
  printf("PID1: %d, PID2: %d\n", pid1, pid2);
  return 0;
}
```

Which of the lines below could have been produced by the *first* process that was started when running fork-test?

A) PID1: 0, PID2: 0
B) PID1: 0, PID2: 28252
C) PID1: 28250, PID2: 0
D) PID1: 28250, PID2: 28251

(b) Which of the following events will *always* trigger an interrupt (hardware or software).

A) A user program requesting to read data from a file.
B) A user program requesting to read data from main memory.
C) A key being pressed on the keyboard.
D) A byte being transferred from disk to main memory.

(c) Which of the following requirements are included in the Critical section problem as defined by the course book.

A) Mutual Exclusion
B) Hold and Wait
C) First come, first served
D) Progress

(d) Which of the following statements are true about deadlocks:

A) If there is only a single instance of every resource, a cycle in the resource allocation graph means that there is a deadlock.

B) All four Coffman conditions must me met for there to be a deadlock.

C) Banker's algorithm is used to detect and remove deadlocks.

D) Banker's algorithm guarantees freedom from starvation.

(e) Which of the following statements (relating to memory management) are true?

A) TLB is short for Transitive Local Buffer.

B) A page fault is a critical failure that the OS cannot recover from.

C) Paging and segmentation can be combined.

D) Belady's page replacement algorithm requires knowing the future memory access pattern.

(f) Which of these parameters will affect the effective access time (EAT) of demand paging?

A) Page fault rate (probability of page fault)

B) Segment table length

C) Time to swap page in from disk

D) Disk size

(g) Which of the following properties are true for semaphores?

A) The wait and signal operations are atomic.

B) They guarantee freedom from deadlock.

C) They can be used to solve the critical section problem.

D) They can only be implemented on a system with special hardware.

(h) Consider the following code snippet (adapted from thegeekstuff.com):

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char buff[15];
    int pass = 0;

    printf("\n Enter password : \n");
    gets(buff);

    if(strcmp(buff, "secret")) {
        printf ("\n Wrong Password \n");
    } else {
        printf ("\n Correct Password \n");
        pass = 1;
    }

    if(pass) {
```

```
        printf ("\n Access granted\n");
    }

    return 0;
}
```

What are the *possible* outcomes of entering a string of more than 15 characters into the program?

A) Nothing in particular, the program outputs "Wrong Password".
B) The program outputs "Wrong Password" and then "Access granted".
C) Stack protection mechanisms detects a buffer overflow and terminates the program.
D) The gets function will not accept a string that exceeds the buffer size so it will throw an exception.

(i) Which of the following are possible states for a system process (as described in the course literature):

A) Waiting
B) Ready
C) Forking
D) Controlling

(j) Which of the following are algorithms that can be used for process scheduling?

A) Shortest-job-first
B) Round robin
C) Priority inversion
D) First-come, first-served

2. **Synchronization (10p)**

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 multithreaded (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. (2p)

(b) Identify the critical section(s) and protect the program against race conditions with a *single* mutex lock. Show the (pseudo)code. (2p)

(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). (2p)

(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`. (4p)

3. **Deadlocks (6p)**

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

(b) Consider the following resource allocation problem in a system with 3 resources (R1-R3), and 4 processes (P1-P4). The table indicates the currently allocated resources and in parenthesis the maximum possible demand.

	R1	R2	R3
P1	0 (1)	0 (0)	0 (0)
P2	0 (6)	0 (1)	2 (7)
P3	2 (3)	0 (0)	1 (1)
P4	0 (2)	0 (0)	0 (9)

The currently available resources are: [5, 1, 6]. Use Banker's algorithm to determine if the request [1, 0, 0] from Process P3 should be granted. (4p)

4. **Processes and scheduling (6p)**

   (a) Define the terms *process*, *kernel-level thread* and *user-level thread*, and explain the differences between them. (3p)

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

   (c) What is a *process control block (PCB)*?
   What is its purpose?
   What data is contained in the PCB of a single-threaded process? (at least 4 relevant items are expected) (2p)

5. **Memory Management (4p)**

   (a) Explain how paging supports sharing of memory between processes. (2p)

   (b) How does a simple segmented memory management system work, and what hardware support should be provided by the memory management unit (MMU)? (2p)

6. **Security (4p)**

   (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) (2p)

   (b) How does memory segmentation support protection? In particular, which kind of security attack could it help to prevent? (2p)

Intentionally empty page.

Multiple choice form for answering question 1. Please put X:s in the appropriate cells:

	A	B	C	D
1 a)				
1 b)				
1 c)				
1 d)				
1 e)				
1 f)				
1 g)				
1 h)				
1 i)				
1 j)				