Chalmers University of Technology and Gothenburg University

Operating Systems
EDA093/092, DIT 401/400

Exam 2018-04-04

Date, Time, Place: Wednesday 2018/04/04, 14.00-18.00, “Maskin”-salar

Course Responsible: Vincenzo Gulisano (+46 31 772 61 47), Marina Papatriantafilou (031 772 54 13)

Auxiliary material: You may have with you
- An English-Swedish, Swedish-English dictionary.
- No other books, notes, calculators, PDA’s etc.

Grade-scale (”Betygsgränser”):
CTH: 3:a 30-39 p, 4:a 40-49 p, 5:a 50-60 p
GU: Godkänd 30-49p, Väl godkänd 50-60 p

Exam review (“Granskningstid”):
Will be announced after the exam.

Instructions

- Do not forget to write your personal number, if you are a GU or CTH student and at which program (”linje”).
- Start answering each assignment on a new page; number the pages and use only one side of each sheet of paper.
- Write in a **clear manner** and **motivate** (explain, justify) your answers. If it is not clear what is written, your answer will be considered wrong. If it is not explained/justified, even a correct answer will get **significantly** lower (possibly zero) marking.
- If you make **any assumptions** in answering any item, do not forget to clearly state what you assume.
- The exam is organized in groups of questions. The credit for each group of questions is mentioned in the beginning of the respective group. Unless otherwise stated, all questions in a group have equal weight.
- Answer questions in English, if possible. If you have large difficulty with that and you think that your grade can be affected, feel free to write in Swedish.

Good luck !!!!

1. (12 p)

   (a) (4 p) The manual for the fsync call says it “transfers all modified in-core data of (i.e., modified buffer cache pages for) the file referred to by the file descriptor fd to the disk device (or other permanent storage device) so that all changed information can be retrieved even after the system crashed or was rebooted”. Why is the OS maintaining modified in-core data instead of flushing it immediately itself?

       HINT: OS optimization (as many others) to reduce overhead.

   (b) (4 p) What is fragmentation? Which forms of fragmentation can be observed and how do they differ? Provide examples for each form of fragmentation you discuss.

       HINT: Please check lectures about memory management and file systems.

   (c) (4p) Discuss the pros and cons of keeping track of free space using the linked list or bitmap approach.

       HINT: With linked list each disk block can keep track of multiple blocks, only keeping track of free blocks. With bitmap, each block is represented by a bit (less space) but need to keep track of all the blocks in the disk. Each is better depending on the overall number of free blocks.

2. (12 p)

   (a) (4 p) Explain why concurrent and parallel execution can be achieved by a program by means of both multiple processes and multiple threads.

       HINT: multiple threads can run on different cores (parallelism) and concurrently on the same core too and so can do processes. Since processes can also communicate and work together (think about the example of chrome) parallelism and concurrent execution can be achieved by both threads and processes.

   (b) (4 p) Is the sentence printed by this code true or false? Discuss why.

```
int main() {
   pid_t pid1, pid2, pid3;
   pid1 = getpid();
   pid2 = fork();
   pid3 = getpid();
   if (pid3==pid1) {
      printf("I am the child process");
   }
}
```

       HINT: After fork (assuming it works correctly) pid1 and pid3 are the same for the parent but not for the child. The sentence is false because only the parent can print.

   (c) (4 p) What is printed by the following program? Explain why, assuming the fork is successful.

```

int a[5] = {0,1,2,3,4};

int main()
{

  int b[5] = {5,6,7,8,9};

  pid_t pid;

  for (int i=0;i<5;i++)
    a[i]*=2;
  for (int i=0;i<5;i++)
    b[i]*=2;

  pid = fork();

  if (pid == 0) {
    for (int i=0;i<5;i++)
      a[i]*=2;
  }
  else {
    wait(NULL);
    for (int i=0;i<5;i++)
      printf('%d ',a[i]);
    for (int i=0;i<5;i++)
      printf('%d ',b[i]);
  }

 return 0;
}
```

HINT: 0, 2, 4, 6, 8, 10, 12, 14, 16, 18. Because the arrays of the parent are not shared with the copy of the process created by fork.

3. (12 p)

   (a) (4 p) Is it possible for two distinct processes to share both the same virtual addresses and physical addresses? Motivate your answer.

   HINT: yes because of COW. No in general.

   (b) (4 p) Specify whether the following sentences are true or false (motivating why).

      i. In the page table, if the referenced frame for a certain page belongs to the process, the valid/invalid bit is set to valid for sure.

      ii. A page might need to be swapped to disk even if the dirty bit is not set (i.e., if the page is not dirty)

   HINT: false, can belong to process but be different page. false

   (c) (4 p) Which data structure can be used to implement LRU allocation so that the search for a victim frame does not require a full traversal of all used frames? Sketch the algorithm using such data structure.

   HINT: Using a stack, the victim frame is the one at the bottom. Notice that when accessing a page already stored in a frame, the corresponding frame should be lifted up in the stack.

4. (12 p)

   (a) (3 p) Consider three CPU-intensive processes, which require 10, 20 and 30 time units and arrive at times 0, 2 and 6, respectively. How many context switches are needed if the operating system implements (i) a preemptive shortest remaining time first scheduling algorithm and (ii) a preemptive longest remaining time first algorithm? Do not count the context switches at time zero and at the end. Support your answers with Gantt charts.

(b) (3 p) Explain a key trade-off with round-robin scheduling. How can it be balanced?

(c) (6 p) (i) Explain how multilevel feedback queue scheduling works.
(ii) Argue whether each of the following is a valid rule: A multilevel feedback queue scheduler assigns a long quantum to: high priority processes; low priority processes; new processes; old processes.

```
Answer hints
a. At time 0, P0 is the only available process so it runs.
At time 2, P1 arrives, but P0 has the shortest remaining time,
so it continues in (i) but a context switch happens in (ii).
Similarly, at time 6, P2 arrives, but P0 has the shortest remaining time,
so it continues in (i), but switches in (ii) and so on. ....

b. small quantum => better fairness, interactiveness,
large quantum 0 => better utilization;
balance with dynamic quanta or multilevel feedback queues.

c.  (i) in the book
(ii) following (i), low-priority procs usually get there because
they are cpu-bound and need the bigger quantum.
```

5. (12 p)

(a) (8 p) The atomic fetch-and-set(x, y) instruction unconditionally sets the memory location x to 1 and fetches its old value in y atomically, i.e. without allowing any intervening access to the memory location x. Consider the following implementation of wait and signal for a binary semaphore:

```
wait (binary_semaphore *s) {
  unsigned y;
  unsigned *x = &(s->value);
  do {
     fetch-and-set(x, y);
  } while (y);
}

signal (binary_semaphore *s) {
  s->value = 0;
}
```

(i) Is this a correct implementation of a binary semaphore in a system with with preemptive scheduling? Argue why or show a counter-example if it is not correct.
(ii) Is this a correct implementation of a binary semaphore in a system with non-preemptive scheduling? Argue why or show a counter-example if it is not correct.

(b) (4p) Given that we can create user-level code to control access to critical sections (e.g., Peterson's algorithm, Lamport's bakery algorithm), why is it important for an operating system to provide synchronization facilities such as semaphores in the kernel?

```
HINTS
a. (i) it is; show the semantics of a binary semaphore are satisfied,
i.e. wait loops if value of s was 1 before, changes to 0 with signal only
(same as the swap solution discussed in class)
(ii) loop can loop forever in uniprocessors; in multiprocessors still ok, as in (i).

b. efficiency reasons: eg no busy-wait => better cpu utilization; fairness;
avoids priority inversion
```