Exam in Operating Systems (EDAF35)
2018-08-30, 08:00–13:00

Inga hjälpmedel! No external resources allowed!

Examiner: Flavius Gruian, tel 046 2224518

25 out of 50p are needed to pass the exam.
You may answer in English/på svenska.

1. (6p) Define the following terms (1–2 sentences each):
   (a) (1p) kernel space
   (b) (1p) processor affinity
   (c) (1p) translation look-aside buffer (TLB)
   (d) (1p) file control block
   (e) (1p) port I/O
   (f) (1p) race condition

2. (6p) Describe/explain concepts:
   (a) (3p) Describe the concept of *journaling* in file systems.
   (b) (3p) Explain *deadlocks* and give strategies for managing them (at least two).

3. (12p) Compare/discuss:
   (a) (6p) Define and compare *user threads* vs. *kernel threads*. In this context, discuss advantages and drawbacks fo *one-to-one*, *many-to-one* and *many-to-many* mapping strategies.
   (b) (6p) In the context of memory management, compare *linear page tables*, *two-level page tables* and *hashed page tables*. Give at least one advantage and one drawback for each.

4. (10p) Assuming demand paging with three (3) frames, and the following page reference string
   `1 2 2 3 1 1 4 2 1 3 4 3 1 2 1 4 3 4 1 3`
   Show the page table contents for every access and count the page faults for
   (a) (4p) a LRU page replacement strategy, and for
   (b) (4p) an optimal replacement strategy.
   (c) (2p) Compare the results and the feasibility of the strategies.

5. (8p) Consider the less inspired `dots.c` program (next page) using `fork()` and POSIX pthreads (on Linux, kernel $\geq$2.6). The program is compiled into `a.out`. Assuming no errors occur,
   (a) (2p) Which lines can result in system calls? How about the `pthread_*` calls?
   (b) (3p) How many dots (".") does the program output when run with "`./a.out`"? Motivate.
   (c) (3p) What if you run it with "`./a.out 1 2 3 4`"? Why?

   Hint: Be extra-careful with `execv(...)`.

Listing 1: dots.c

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

void *run(void *ptr)
{
        char** ss = ptr;
        if(ss[1] != NULL) execv("./a.out",&ss[1]);
        fprintf(stderr, ".");
        return  ptr;
}

int main(int argc, char **argv)
{
        pthread_t thread[2];
        pthread_create(&thread[0], NULL, run, (void *) &argv[0]);
        pthread_create(&thread[1], NULL, run, (void *) &argv[0]);
        if(fork()) fprintf(stderr,"b");
        else fprintf(stderr,"a");
        pthread_join(thread[0],NULL);
        pthread_join(thread[1],NULL);
        return 0;
}
```

6. (8p) The *readers-writers* problem is a classic synchronization problem you should be familiar with already. Consider the following solution (pseudo-code) suggested by a forgetful engineer:

Listing 2: Erroneous Readers-Writers

```
/* code for a writer */
do {
        wait(rw_mutex);
                /* do some writing */
        signal(rw_mutex);
} while(true);

/* code for a reader */
do {
        wait(mutex);
        reads++; // zero initially
        if(reads==1) wait(rw_mutex);
                /* do some reading */
        reads--;
        if(reads==0) signal(rw_mutex);
        signal(mutex);
} while(true)
```

(a) (4p) What is the problem with this code? What is missing? (add two lines)

(b) (4p) When fixed according to (a), are there any issues with this solution?