Operating Systems
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: 320202
Date: 2013-05-22
Duration: 120 minutes

Final Examination

The Jacobs University's Code of Academic Integrity applies to this examination. Please fill in your name (please write readable) and sign below.

Name:	
Signature:	

This exam is **closed book**. In addition, you are not allowed to use any electronic equipment such as computers, smart phones, cell phones, or calculators.

Please answer the questions on the problem sheets. If you need more space, feel free to write on the back of the pages. Please keep the papers stapled.

Problem	Max. Points	Points	Grader
F.1	10		
F.2	10		
F.3	20		
F.4	25		
F.5	10		
F.6	15		
F.7	10		
Total	100		

Good luck and a nice relaxing summer break!

Problem F.1: *deadlocks* (2+2+2+2+2 = 10 points)

Indicate which of the following statements are correct or incorrect by marking the appropriate boxes. For every correctly marked box, you will earn two points. For every incorrectly marked box, you will loose one point. Statements which are not marked or which are marked as true and false will be ignored. The minimum number of points you can achieve is zero.

true	false	
☐	☐	Resources that can be de-allocated by the operating system (without causing side effects) after they have been allocated to a process are not causing deadlocks.
☐	☐	A cycle in a resource allocation graph is a sufficient condition for the presence of a deadlock.
☐	☐	The Banker's algorithm can be used to avoid deadlocks if the maximum number of resources requested by a process during its lifetime is known.
☐	☐	Deadlock prevention generally leads to very inefficient solutions and hence is not widely used.
☐	☐	A deadlock assumes that the processes involved in the deadlock block and wait for a condition that is never becoming true. A livelock is a situation where the processes involved do execute instructions but stay in a loop and never progress out of the loop.

Problem F.2: *processes and pipes* (3+3+2+2 = 10 points)

Take a look at the C code (fa-pipe.c) shown below. Assume that all system and library calls succeed at runtime. (All error handling code has been omitted for brevity.)

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    pid_t pid1, pid2;
    int fds[2];

    char *argv1[] = { "ls", "-l", "/usr/bin", NULL };
    char *argv2[] = { "more", NULL };

    pipe(fds);

    pid1 = fork();
    if (! pid1) {
        close(fds[0]);
        dup2(fds[1], STDOUT_FILENO);
        close(fds[1]);
        execvp(argv1[0], argv1);
    }

    pid2 = fork();
    if (! pid2) {
        close(fds[1]);
        dup2(fds[0], STDIN_FILENO);
        close(fds[0]);
        execvp(argv2[0], argv2);
    }

    close(fds[0]);
    close(fds[1]);
    waitpid(pid2, NULL, 0);

    return EXIT_SUCCESS;
}
```

a) Describe what the program is doing.

b) Show the open file tables of the processes involved right before the calls to `execve()` and `waitpid()`.

c) What happens if the programmer mistakenly passes `pid1` instead of `pid2` to the `waitpid()` call and runs the program from his command shell?

d) Assume the first child process experiences a runtime error and it produces an error message. What will happen to the error message? Can the user page back to the error message?

Problem F.3: *synchronization (running competition)* (20 points)

A sports club is organizing a running competition. The club has a total of N electronic chips that are used to measure the performance of the runners between the start and the finish line. Unfortunately, the number of runners showing up at the event by far exceeds the number of chips owned by the club. As such, only some runners can start together while the others have to wait for other runners to finish before they can pick up a chip.

The competition also supports relay running teams. An arriving relay runner joins the next possible team. Once a team of T relay runners is complete, the team leader (e.g., the last member of the team) picks up T chips and then the relay runners proceed together to the start.

Write a solution (in pseudo code like on the course slides) for this problem using semaphores. Individual runners execute the function `runner()` to obtain a chip, run, and return the chip. Relay runners execute the function `relay()` to join a team of relay runners. Once a team is complete, the team leader obtains T chips and then the team members proceed to run. Every relay runner returns his chip independently after finishing.

Problem F.4: *pthread programming* (20+3+2 = 25 points)

Take a look at the C code shown below. Assume that all system and library calls succeed at runtime. (All error handling code has been omitted for brevity.)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#include <unistd.h>

#define N 5

static long int a[N];

typedef struct {
    pthread_t thread;
    int       tid;
    long int  value;
} thread_state_t;

static void* weird(void *arg)
{
    thread_state_t *ts = (thread_state_t *) arg;
    int me = ts->tid;
    int ne = (me > 0) ? me - 1 : N - 1;

    while (1) {
        if (a[ne] == -1) {
            a[me] = a[ne];
            break;
        } else if (a[ne] == ts->value) {
            a[me] = -1;
            printf("\n%d: The lucky number is: %ld\n", me, a[ne]);
            break;
        } else if (a[ne] > a[me]) {
            a[me] = a[ne];
        } else {
            /* do nothing */
        }
    }

    return NULL;
}

int main(int argc, char **argv)
{
    int i;
    thread_state_t ts[N];

    srandom(getpid());          /* initialize the random number generator */
    for (i = 0; i < N; i++) {
        ts[i].value = a[i] = random();  /* a random non-negative number */
        ts[i].tid = i;                  /* an id in the range [0..N-1] */
        printf("%d: %12ld\n", ts[i].tid, ts[i].value);
    }

    for (i = 0; i < N; i++) pthread_create(&ts[i].thread, NULL, weird, &ts[i]);
    for (i = 0; i < N; i++) pthread_join(ts[i].thread, NULL);

    return EXIT_SUCCESS;
}
```

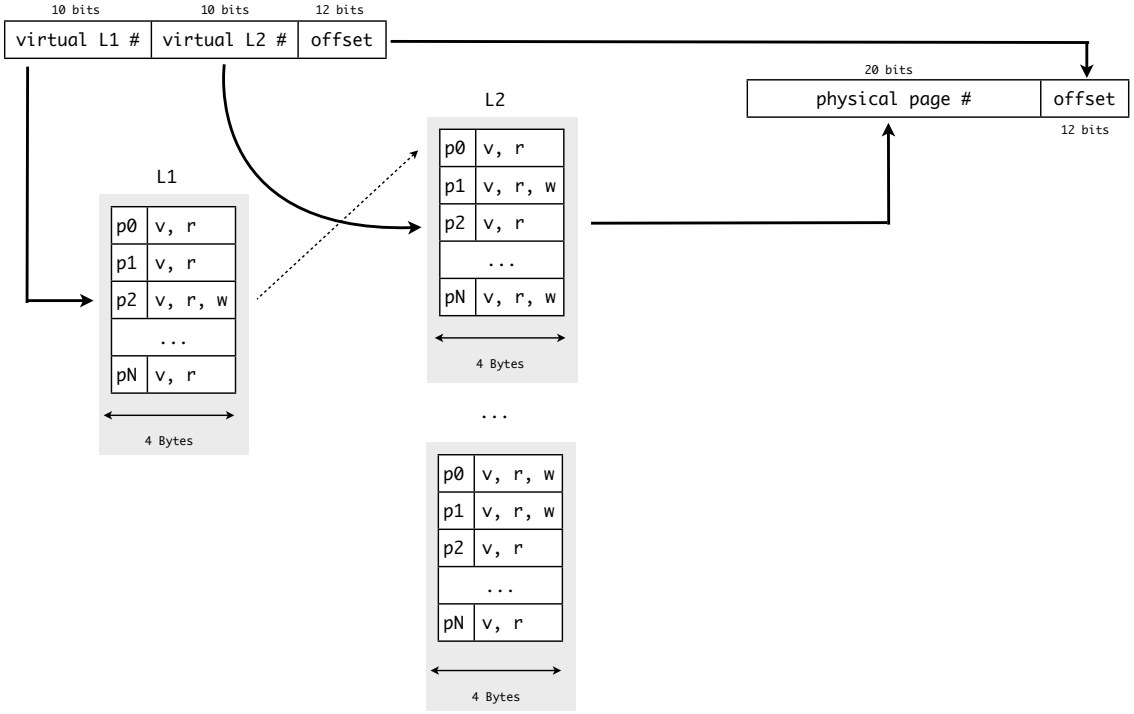
a) Explain what the program is doing. What is a lucky number and which thread is going to print the lucky number? Hint: Try an example with N=3 to see what the algorithm does.

b) The threads operate on shared data and hence there might be synchronization problems. What is the shared data? How can the potential synchronization problem be fixed? Insert statements as you see fit (the precise syntax is not relevant as long as the idea is clear).

c) Is the program always going to produce the right result?

Problem F.5: *virtual memory and two-level page tables* (2+2+2+2+2 = 10 points)

Consider the two-level paging system shown below.



a) What is the size of a page? Explain.

b) How many page tables are possible at the second level? Explain.

c) What is the size of each page table at L2? Explain.

d) How many pages are possible in total? Explain.

e) Can a stack/heap span beyond a page? If so, would it be contiguous in physical memory? Explain.

Problem F.6: *files and filesystems* (4+2+3+3+3 = 15 points)

a) Unix file systems typically use *directories*, *inodes*, and data blocks to organize the file system. Which information is stored in a directory and which information is stored in an inode?

b) Assume the following shell commands have been executed successfully:

```
mkdir d
mkdir d/a
mkdir d/b
mkdir d/c
```

What is the link count of the directory d? Explain why.

c) Explain the difference between hard links and symbolic links. How do POSIX operating systems react when symbolic links create a loop (see the example below)?

```
ln -s a a
cat a
```

d) Assume you plan to write a program that collects information in a temporary file. The temporary file should be hidden so that other users (who of course know about `ls -a`) cannot find it. How can this be accomplished? (Note: The hidden file may be visible for a very short period of time.)

e) POSIX systems support memory mapped files. Explain what memory mapped files are. What is the motivation for memory mapping files and what are typical use cases?

Problem F.7: *devices and efficient low-level I/O* (2+2+2+2+2 = 10 points)

Indicate which of the following statements are correct or incorrect by marking the appropriate boxes. For every correctly marked box, you will earn two points. For every incorrectly marked box, you will loose one point. Statements which are not marked or which are marked as true and false will be ignored. The minimum number of points you can achieve is zero.

true	false	
☐	☐	Vectored I/O (also known as scatter/gather I/O) is an I/O method where a single system-call (`writev()`) sequentially writes data from multiple buffers to a file descriptor and a single system-call (readv()) reads data from a file descriptor to multiple buffers.
☐	☐	On POSIX systems, devices are represented as special files (typically located under `/dev`) with an associated major and minor device number: the major number identifies the device driver and the minor number identifies a particular device (possibly out of many) that the driver controls.
☐	☐	Logical volume management (LVM) is a form of storage virtualization where several physical volumes can form a (virtual) volume group and logical volumes can be easily allocated out of the volume groups.
☐	☐	The driver of a terminal device (`tty` or `ptty`) operating in cooked mode will interpret control characters such as Control-C and Control-D by generating a signal or an end of file indication.
☐	☐	The contents of the environment variable `TERM` is commonly used by application programs to lookup the escape and control sequences understood by the controlling terminal.