Operating Systems
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: 320202
Date: 2011-05-20
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	15		
F.3	20		
F.4	10		
F.5	15		
F.6	10		
F.7	10		
F.8	10		
Total	100		

Good luck!

Problem F.1: *file systems* (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	
☐	☐	A change of the permissions of a file is immediately reflected on all hard links of that file.
☐	☐	Unix file systems traditionally record for every file system object the creation time, the time of the last modification, and the time of the most recent access.
☐	☐	Soft links cannot cross file system boundaries.
☐	☐	A virtual file system allows applications to access different types of concrete file systems in a uniform way.
☐	☐	A logical volume manager allows system administrators to create partitions that can cross multiple physical storage devices.

Solution:

true	false	
☒	☐	A change of the permissions of a file is immediately reflected on all hard links of that file.
☒	☐	Unix file systems traditionally record for every file system object the creation time, the time of the last modification, and the time of the most recent access.
☐	☒	Soft links cannot cross file system boundaries.
☒	☐	A virtual file system allows applications to access different types of concrete file systems in a uniform way.
☒	☐	A logical volume manager allows system administrators to create partitions that can cross multiple physical storage devices.

Problem F.2: *processes and pipes* (2+5+8 = 15 points)

Assume that all system and library calls succeed at runtime. (All error handling code has been omitted for brevity.)

a) The following source code has been compiled into the executable file `inc`. Briefly explain what the program `inc` does.

```
#include <unistd.h>

int main(int argc, char *argv[])
{
    char buf[1];

    read(STDIN_FILENO, buf, 1);
    buf[0]++;
    write(STDOUT_FILENO, buf, 1);
    return 0;
}
```

b) What is the output produced by the following program? Provide an explanation!

```
#include <unistd.h>

int main(int argc, char *argv[])
{
    int xfds[2], yfds[2]; char *argv2[2] = { "inc", NULL };

    pipe(xfds); pipe(yfds);
    if (fork() == 0) {
        dup2(xfds[0], STDIN_FILENO);
        dup2(yfds[1], STDOUT_FILENO);
        execv("inc", argv2);
    }
    if (fork() == 0) {
        dup2(yfds[0], STDIN_FILENO);
        execv("inc", argv2);
    }
    write(xfds[1], "0", 1);
    return 0;
}
```

c) What is the output produced by the following program? Provide an explanation! Hint: What is the simplest to implement well defined semantic from a kernel programmer's perspective?

```
#include <unistd.h>

int main(int argc, char *argv[])
{
    int fds[2]; char *argv2[2] = { "inc", NULL };

    pipe(fds);
    if (fork() == 0) {
        dup2(fds[0], STDIN_FILENO);
        execv("inc", argv2);
    }
    if (fork() == 0) {
        dup2(fds[0], STDIN_FILENO);
        execv("inc", argv2);
    }
    write(fds[1], "01", 2);
    return 0;
}
```

Solution:

a) The program `inc` reads a byte from the standard input, increments it, and writes the result to the standard output before then exits. Note that the program processes exactly one byte from the standard input.

b) The program creates two pipes and two child processes. The first child process has the standard input connected to the first pipe (xfds) and the standard output connected to the second pipe (yfds). The second child process has its standard input connected to the second pipe. Both child processes execute the program `inc`. The parent process writes a byte representing the character `'0'` into the first pipe (xfds). The first child process reads the character `'0'`, increments it to `'1'` and passes it via the pipe (yfds) to the second child process. The second child process reads the character `'1'`, increments it to `'2'` and prints it on the standard output. Hence, the execution of the program produces the output `2`.

c) The program creates a pipe (fds) and two child processes. Both child processes connect the same end of the pipe to their standard input before executing the program `inc`. The parent process then writes two bytes representing the characters `'0'` and `'1'` into the pipe.

   What happens next may not be too obvious since we have a pipe filled with two characters (bytes) and two processes ready to read one byte each from the pipe. One could assume that both processes read the same data. However, this would complicate things in the kernel since the kernel would have to keep track which byte was read by which process before it can release the buffer. Hence, the kernel simply serializes the read requests to the shared pipe and as a consequence the first process reads the character `'0'`, increments it to `'1'` and prints it on the standard output. The second character `'1'` is then read by the other process, which increments it to `'2'` and prints it on the standard output. Hence, the execution of the program produces the output `12`.

Problem F.3: *policy to improve social life in the serveries* (20 points)

In order to strengthen the community spirit at Jacobs University and to improve the social life of students, the university leadership has adopted a new policy that students should not eat alone. The new policy allows a student to eat alone as long as there is no other student in the servery. However, if there are multiple students in the servery, then as long as a student has not finished eating, at least one other student has to be around. In particular, if only two students are left, then they either leave together after finishing the meal together or the student who finishes first has to wait until either the other student finishes (so they can leave together) or a new student arrives and starts eating. Due to their high workload, students have a natural desire to leave the servery after finishing eating. Students should only be blocked from leaving if there is exactly one student left eating who can't be left alone.

Below is a skeleton implementing a student thread. After getting food, by calling `food()`, a student calls `eat()` to consume the food before eventually calling `leave()` to leave the servery. Your task is to fill in the proper synchronization pseudo code using semaphores. For solving the problem, it is useful to keep track of how many students are eating (shared variable `eating`) and how many students are ready to leave (shared variable `ready_to_leave`). These shared variables are protected by the semaphore `mutex`. You are allowed to add additional shared variables and semaphores as you see necessary.

```
shared int eating = 0, ready_to_leave = 0;
semaphore mutex = 1;




void student()
{
    food();









    eat();









    leave();
}
```

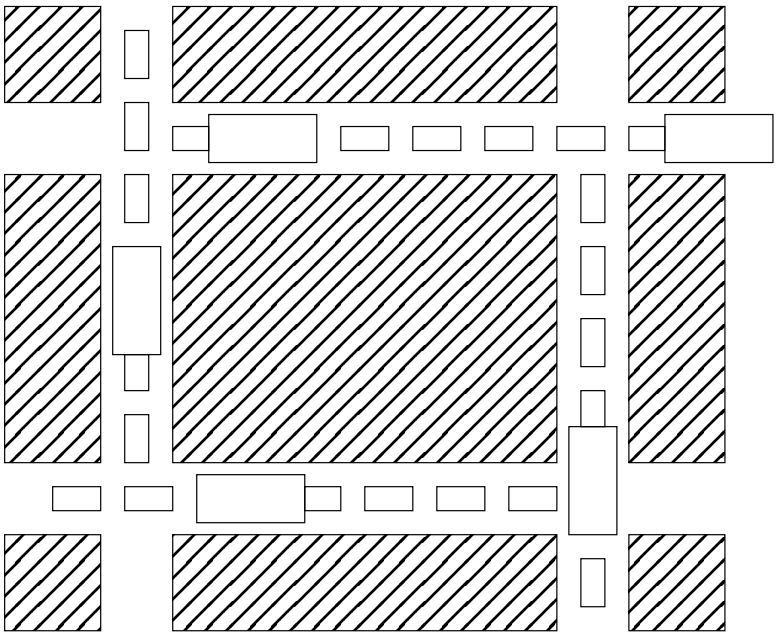
Solution:

```
shared int eating = 0, ready_to_leave = 0;
semaphore mutex = 1, ok_to_leave = 0;

void student()
{
    get_food();

    down(&mutex);
    eating++;
    if ((eating == 2) && (ready_to_leave == 1)) {
        up(&ok_to_leave);
        ready_to_leave--;
    }
    up(&mutex);

    eat();

    down(&mutex);
    eating--;
    ready_to_leave++;
    if ((eating == 1) && (ready_to_leave == 1)) {
        up(&mutex);
        down(&ok_to_leave);
    } else if ((eating == 0) && (ready_to_leave == 2)) {
        up(&ok_to_leave);
        ready_to_leave -= 2;
        up(&mutex);
    } else {
        ready_to_leave--;
        up(&mutex);
    }

    leave()
}
```

Problem F.4: *deadlocks conditions* (5+5 = 10 points)

a) Name the four necessary conditions that must hold simultaneously for a deadlock to occur. Provide a short explanation for each of them.

b) Consider the traffic deadlock depicted below. Show that the four necessary conditions for deadlocks indeed hold in this example. Is there a simple rule that will avoid deadlocks in this system?



Solution:

a)
1. Mutual Exclusion: Resources cannot be used simultaneously by several processes.
2. Hold and Wait: Processes apply for a resource while holding another resource.
3. No Preemption: Resources cannot be preempted, only the process itself can release resources.
4. Circular Wait: A circular list of processes exists where every process waits for the release of a resource held by the next process.

b) Consider each section of the street as a resource.

1. Mutual Exclusion: Only one vehicle can be on a section of the street at a time.
2. Hold and Wait: Each vehicle is occupying a section of the street, and waiting to move on to the next section of the street.
3. No Preemption: A section of the street that is occupied by a vehicle cannot be taken away from it.
4. Circular Wait: Each vehicle is waiting for a section of street held by the next vehicle in the traffic jam, in a circular fashion.

A simple rule to avoid the traffic deadlock is that a vehicle should only enter an intersection if it is assured that it will not have to stop inside the intersection.

Problem F.5: *banker's algorithm* (8+7 = 15 points)

a) A system has four processes and five allocatable types of resources. The current allocation and maximum needs are as follows:

$$Max = \begin{pmatrix} 1 & 1 & 2 & 1 & 3 \\ 2 & 2 & 2 & 1 & 1 \\ 2 & 1 & 3 & 1 & 0 \\ 1 & 1 & 2 & 2 & 1 \end{pmatrix} \qquad Alloc = \begin{pmatrix} 1 & 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

The available resources are given by $Avail = (0, 0, x, 1, 1)$. What is the smallest value of x (if it exists) for which this is a safe state?

b) What is the worst case complexity (big O notation) of the safe-state algorithm for m resource types and n processes? You may ignore the initialization and finalization steps.

Solution:

a) The calculation of the still needed resources gives us:

$$Need = \begin{pmatrix} 0 & 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \qquad Total = (5, 2, 4 + x, 5, 3)$$

Given $Avail = (0, 0, x, 1, 1)$ and the $Need$ matrix above, we can only satisfy the need of process number 4. For this, $x = 1$ is sufficient. Once process number 4 terminates, it will release all resources and hence the vector $Avail$ becomes $Avail = (1, 1, 1 + x, 2, 1)$. The needs of processes 1 and 2 still can not be satisfied. However, with $x = 2$, we can satisfy the resource needs of process 3. Once process number 3 terminates, it will release all resources and hence the vector $Avail$ becomes $Avail = (2, 2, 1 + x, 3, 1)$. With this, we can satisfy the resource needs of process 2. Once process number 2 terminates, it will release all resources and hence the vector $Avail$ becomes $Avail = (4, 2, 2 + x, 4, 2)$. We can now satisfy the resource needs of process 1 and after completion $Avail$ becomes $Avail = (5, 2, 4 + x, 5, 3)$. Hence, $x = 2$ is the smallest value of x for which this state is safe.

b) The outer loop of the algorithm tries to find an execution sequence allowing all n processes to complete. In each iteration of the outer loop, it is necessary to find a process which has not yet terminated and whose maximum resource requests can be satisfied. In the worst case, this requires again n steps. In order to determine whether the maximum resource requests are satisfied for a given process, m comparisons are needed (one for each resource type). Hence, the big O worst case complexity becomes $m \cdot n^2$.

Problem F.6: *programming related aspects* (3+3+4 = 10 points)

a) What is the difference between a statically linked program and a dynamically linked program? What are the advantages and disadvantages?

b) How is a segmentation fault detected and how is it communicated to the user space process causing the segmentation fault?

c) Consider the following C source code. In which data segments (text, data, heap, stack) is data and machine code stored?

```
char *ptr = NULL;

void foo()
{
   int i;

   ptr = malloc(42);
   if (ptr) {
     for (i = 0; i < 42; i++) {
       ptr[i] = ' ';
     }
   }
}
```

Solution:

a) A statically linked program has all libraries linked into the executable file. A dynamically linked program still has open references to libraries that must be resolved at program startup time.

Since statically linked programs have no references to other libraries, they can be more easily copied between systems and they have a fast startup time. However, since they carry all the library code needed, the program files tend to be much bigger. Dynamically linked programs require less disk space (and less memory space) at the expense of additional startup overhead. With dynamically linked programs, if is possible to update libraries (e.g., fixing bugs) without having to link the programs again.

b) A segmentation fault is detected by the memory management unit, which raises a hardware interrupt. The operating system kernel then takes control and identifies which process caused the invalid memory access. Once the process has been identified, the kernel delivers a signal (`SIGSEGV`) to the process. The signal handler by default prints an error message and terminates the process.

c) The pointer `ptr` is allocated in the data segment. The function `malloc()` allocates memory in the heap segment. The function local variable i is allocated on the stack and the machine code of the function `foo()` is allocated in the text segment.

Problem F.7: *network communication with sockets* (2+2+4+2 = 10 points)

a) Briefly explain the purpose of the `struct sockaddr_storage`.

b) What is the purpose of the `select()` system call?

c) Which sequence of socket related system calls is typically executed by a connection-oriented server until data can be read from / send to the first client connection?

d) Provide a reason why `getaddrinfo()` might return more than one element in the result list.

Solution:

a) The `struct sockaddr_storage` provides storage space that is guaranteed to be big enough to hold any address that can be used by the operating system. It is commonly used to allocate storage for socket addresses when the address family is not yet known.

b) The `select()` system call allows to examine whether sets of file descriptors are ready for reading, are ready for writing, or have an exceptional condition pending. An optional timeout can be provided to specify a maximum interval to wait.

c) A connection-oriented server creates a socket (`socket()`), binds it to an address (`bind()`), and finally tells the kernel to listen for incoming connection attempts (`listen()`). The `accept()` system call returns a file descriptor representing a client connection.

d) An Internet domain name such as `www.ietf.org` may resolve to both IPv4 and IPv6 addresses.

Problem F.8: *memory management* (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	
☐	☐	An operating system implementing the working set model will never suffer from thrashing.
☐	☐	Page replacement algorithms belonging to the class of stack algorithms can never exhibit Belady's anomaly.
☐	☐	Given a fixed frame size, the size of a page table is determined by the maximum possible logical address space while for an inverted page table, the size of the inverted page table is determined by the maximum possible physical address space.
☐	☐	Segmentation can be used to implement virtual memory.
☐	☐	With a Buddy System memory allocator, it is possible that a contiguous chunk of free physical memory satisfying a memory request exists that can't be assigned to the requester.

Solution:

true	false	
☐	☒	An operating system implementing the working set model will never suffer from thrashing.
☒	☐	Page replacement algorithms belonging to the class of stack algorithms can never exhibit Belady's anomaly.
☒	☐	Given a fixed frame size, the size of a page table is determined by the maximum possible logical address space while for an inverted page table, the size of the inverted page table is determined by the maximum possible physical address space.
☒	☐	Segmentation can be used to implement virtual memory.
☒	☐	With a Buddy System memory allocator, it is possible that a contiguous chunk of free physical memory satisfying a memory request exists that can't be assigned to the requester.