

# Operating Systems ID2206

## English version

2017-01-14 08:00-12:00

Name: \_\_\_\_\_

### Instruction

- You are, besides writing material, only allowed to bring one self hand written A4 of notes. Mobiles etc, should be left to the guards.
- All answers should be written in these pages, use the space allocated after each question to write down your answer.
- Answers should be written in Swedish or English.
- You should hand in the whole exam.
- No additional pages should be handed in.

### Grades for 6 credits

The exam is divided into a number of questions where some are a bit harder than others. The harder questions are marked with a star *points\**, and will give you points for the higher grades. The exam is thus divided into basic points and points for higher grades. First of all make sure that you pass the basic points before engaging with the higher points.

Note that, of the 40 basic points only at most 36 are counted, the points for higher grades will not make up for lack of basic points. The limits for the grades are as follows:

- Fx: 21 basic points
- E: 23 basic points
- D: 28 basic points
- C: 32 basic points
- B: 36 basic points and 12 higher points
- A: 36 basic points and 18 higher points

The limits could be adjusted to lower values but not raised.

### Gained points

Don't write anything here.

<b>Uppgift</b>	1	2	3	4	5	6	7	8	9
<b>Max G/H</b>	4/0	2/2	4/2	4/2	4/2	4/2	4/2	2/2	12/8
<b>G/H</b>									

**Total number of points:**

Name: \_\_\_\_\_

## 1 Operating systems

### 1.1 files och rights [2 points]

In a Unix operating system files and directories protected in a way that limits the rights to use an object. We can see these rights when we list the content in a directory. Describe which rights are given to the file called `foo` below.

```
> ls -l foo
> -rwxr-x--- 1 kalle trusted 234 dec 26 13:18 foo
```

### 1.2 commands in a shell [2 points]

Give a short description of the commands below.

- `wc`
- `grep`
- `mkdir`
- `pwd`

## 2 Processes

### 2.1 `fork()` [2 points]

What is printed when we run the program below and why do we get this result?

Name: \_\_\_\_\_

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

int x = 42;

int main() {
    if (fork() == 0) {
        x++;
        printf("x = %d\n", x);
    } else {
        x++;
        printf("x = %d\n", x);
    }
    return 0;
}
```

## 2.2 IDT [2 points\*]

What does the IDT (Interrupt Descriptor Table) contain and what happens

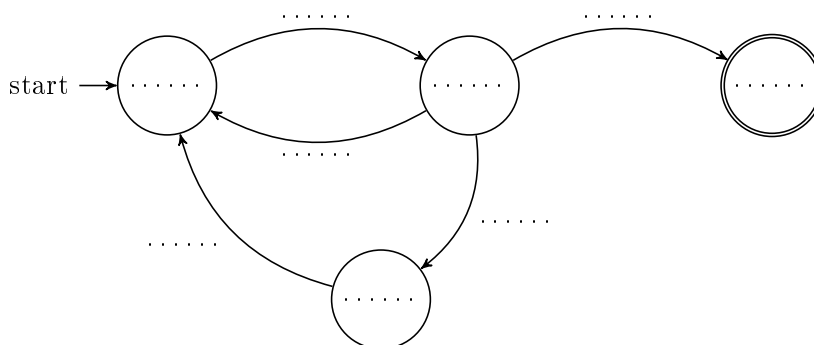
Name: \_\_\_\_\_

when a user process executes the instruction INT (x86 assembler). Give a short description.

### 3 Scheduling

#### 3.1 state diagram [2 points]

Here follows a state diagram for scheduling of processes. Enter the marked states and transitions to describe what states means and when a process is transferred between different states.



#### 3.2 shortest job first [2 points]

Assume we have a scheduler that implements "shortest job first" i.e. not able

Name: \_\_\_\_\_

to preempt jobs. If we have three jobs that will take 10ms, 20ms and 30ms it's a better strategy than taking the jobs in random order, show why.

### **3.3 fair scheduler [2 points\*]**

There is a group of scheduleres where each process will get a “fair” share of the processing power. Some of these are based on a “lottery” but we can also

Name: \_\_\_\_\_

implement a complete deterministic scheduler. What is this method called?  
Give a short description of how it works.

## 4 Virtual memory

### 4.1 base register [2 points]

Why do you not want to implement a segmented memory using only a *base register* to describe a segment?

### 4.2 a tree [2 points]

When representing a *page table* a tree structure is used. Why use a tree structure, it would be faster to access an entry if the table was represented

Name: \_\_\_\_\_

as an array with direct access to the entries. A tree will only give us one or more indirection so why use a tree?

### **4.3 segmented memory x86 [2 points]**

When is segmented memory used in a Linux system on a x86 architecture?

## **5 Memory management**

### **5.1 parking lots [2 points]**

When they arranged for parking space along Sveavägen (central Stockholm) there were two alternatives: 1/ have painted parking lots of 6m in length or 2/ let cars park with 25 cm distance without the limitation of painted lots. If we, for simplicity, assume that cars are between 4.0 and 5.5 meters and



Name: \_\_\_\_\_

that everyone can park a car in a slot with half a meter of extra space, then what is the problem with each of the solutions?

### **5.2 list of free blocks [2 points]**

If we when implementing `malloc()` and `free()` choose to save the free blocks in a linked list that is ordered by their address, we will have a certain advantage. When we free a block we can insert it in the list and perform an operation that reduces the external fragmentation. What can we do and why is it an advantage to have the blocks order by address?

### **5.3 buddy allocation [2 points]**

The so called “Buddy algorithm” to handle free space has a clear advantage but also a deficiency. Assume we have a memory divided into 4K-blocks and

Name: \_\_\_\_\_

we have 16K that are free in block 0b10100. What do we do when get 8K free in block 0b11000? How does this illustrate the deficiency?

## 6 Concurrent programming

### 6.1 count [2 points]

If we execute the procedure `hello()` below in two threads concurrently, the result will be that `count` obtains the value ...- which values can `count` hold after both of the threads have completed the execution? Why is this possible?

```
int loop = 10;
int count = 0;

void *hello() {
    for(int i = 0; i < loop; i++) {
        count++;
    }
}
```

Name: \_\_\_\_\_

### **6.2 own stack [2 points]**

If we divide a process in two threads, the two threads can read from each others stacks - true or false? Motivate your answer.

### **6.3 total store order[2 points\*]**

How does *total store order* differ from *sequential consistency*?

## **7 File systems**

### **7.1 zombies [2 points]**

If one happens to remove the last link to a file the file becomes a so called *zombie file* - true or false? Motivate your answer.

Name: \_\_\_\_\_

### **7.2 bit maps [2 points]**

When implementing a file system we can divide a disk into: super block, bit maps, inodes and data blocks. What are the bit maps used for?

### **7.3 a directory [2 points\*]**

Which information does a directory contain and how is it represented? Describe in terms of inodes and data blocks, which information is where? Draw a simple picture that describes the structure.

Name: \_\_\_\_\_

## 8 Virtualization

### 8.1 user mode [2 points]

When virtualizing a complete operating system the virtualized operating system will execute in *user mode* - true or false? Motivate your answer.

### 8.2 emulators [2 points\*]

A regular *hypervisor* is used to virtualize one or more operating systems allowing all to run on the same hardware. The so called *emulators*, such as QEMU, have an additional advantage - which?

## 9 Implementation

### 9.1 memory map [2 points]

Below is a, somewhat shortened, printout of a memory mapping of a running process. Briefly describe the role of each segment marked with ???.

```
> cat /proc/13896/maps
```

```
00400000-00401000 r-xp 00000000 08:01 1723260          .../gurka ???
00600000-00601000 r--p 00000000 08:01 1723260          .../gurka ???
00601000-00602000 rw-p 00001000 08:01 1723260          .../gurka ???
022fa000-0231b000 rw-p 00000000 00:00 0              [???]
7f6683423000-7f66835e2000 r-xp 00000000 08:01 3149003          .../libc-2.23.so ???
```

Name: \_\_\_\_\_

```
      :
7ffd60600000-7ffd60621000 rw-p 00000000 00:00 0      [???]
7ffd60648000-7ffd6064a000 r--p 00000000 00:00 0      [vvar]
7ffd6064a000-7ffd6064c000 r-xp 00000000 00:00 0      [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

## 9.2 execlp() [2 points\*]

In the program below we call the library procedure `execlp()`. What will this procedure do and when and in what order will “Humle” and “Dumle” be written? Motivate your answer.

```
int main() {

    int pid = fork();

    if(pid == 0) {
        execlp("gurka", "gurka", NULL);
        printf("Humle\n");
    } else {
        wait(NULL);
        printf("Dumle\n");
    }
    return 0;
}
```

Name: \_\_\_\_\_

### 9.3 modify the kernel [2 points]

If we should add a functionality to a Linux kernel, we must recompile the kernel and restart the system - true or false? Motivate your answer.

### 9.4 `proc_create()` [2 points\*]

There are several ways in which we can communicate with a kernel module, the code below shows one way. Here we initialize certain data structures that then allow other processes to easily access the module. Which is the visible interface that processes will use? Give a brief explanation.

```
static int __init skynet_init(void) {
    proc_create("skynet", 0, NULL, &skynet_fops);
    printk(KERN_INFO "Skynet in control\n");
    return 0;
}

static void __exit skynet_cleanup(void) {
    remove_proc_entry("skynet", NULL);
    printk(KERN_INFO "I'll be back!\n");
}
```

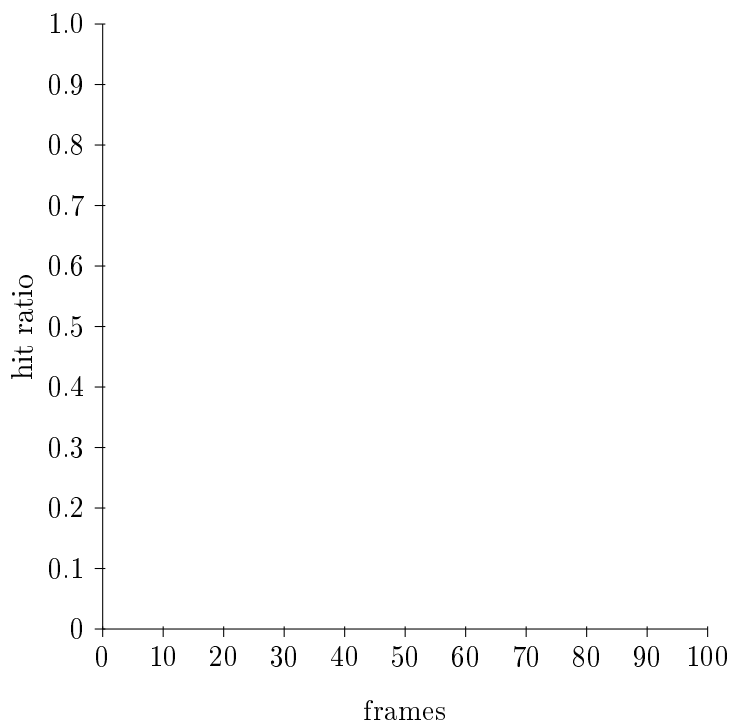
Name: \_\_\_\_\_

### 9.5 a graph [2 points]

Assume that we do an experiment where we compare three methods to implement a *swapping algorithm* i.e. to throw out pages from memory when more frames are needed. In our experiment we have a virtual memory of 100 pages and simulate a memory of up to 100 frames. The experiment simulates a sequence of memory operations with temporal locality.

In the diagram below you should plot justifiable graphs for the following three strategies:

- RND: choose a page by random
- OPT: choose a page that will not be used in a long time
- LRU: choose a page that has not been used for a long time





Name: \_\_\_\_\_

### 9.6 an expensive operation [2 points]

Below is an extract from a program that implements *Least Recently Used* (LRU). The code shows why LRU is expensive to implement and why one probably instead chooses to approximate this strategy. What is the code doing and when is it executed?

```

:
if (entry->present == 1) {

    if (entry->next != NULL) {

        if (first == entry) {
            first = entry->next;
        } else {
            entry->prev->next = entry->next;
        }
        entry->next->prev = entry->prev;

        entry->prev = last;
        entry->next = NULL;

        last->next = entry;
        last = entry;
    }
} else {

:
}
```

### 9.7 tick-tack [2 points\*]

When implementing the *clock algorithm* it is sufficient to have the list single

Name: \_\_\_\_\_

linked i.e. there is no need for a double linked list. Why can we manage with a single linked list?

### 9.8 lock() [2 points]

Assume that we define `try()` as the code below. The procedure `__sync_val_compare_and_swap(int *loc, int old, int new)` will in an atomic operation compare the value of `*loc` and if it is equal to `old` write `new` in its place. The procedure returns the value of `*loc` before the operation.

```
int try(volatile int *mutex) {
    return __sync_val_compare_and_swap(mutex, 0, 1);
}
```

How can we implement a simple *spin lock*? Use the code below and assume that we call `lock()` and `unlock()` with a pointer to a global lock that holds 0 in its unlocked state.

```
void lock(volatile int *mutex) {
    :
    :
    :
}

void unlock(volatile int *mutex) {
    :
    :
    :
}
```

### 9.9 futexp\_wait() [2 points]

We can improve the characteristics of a *spin lock* by using so called *mutex*. The code that follows defined two procedures `mutex_wait()` and `mutex_wake()`.

```
int mutex_wait(volatile int *mutex) {
    return syscall(SYS_mutex, mutex, MUTEX_WAIT, 1, NULL, NULL, 0);
}
```

Name: \_\_\_\_\_

```
void futexp_wake(volatile int *futexp) {  
    syscall(SYS_futex, futexp, FUTEX_WAKE, 1, NULL, NULL, 0);  
}
```

What do these procedures do and how are they used?

### 9.10 list updates [2 points\*]

Assume we have a single linked sorted list that we wish to update by adding or removing elements. We want several threads to be able to perform operations in parallel and therefore implement a list where each cell is protected by a lock. The operations require that we hold two locks when we remove an item but this is easily implemented.

We can choose to use `pthread_mutex_lock()` to implement the lock operations but we could also use simple *spin-locks*. What is the advantage and disadvantage of using *spin-locks* and how are the conditions changed if we have more threads and less cores?