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.

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

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
```

Answer: The owner `kalle` has read, write and execute rights. Members of the group `trusted` can read and execute. Other users have no rights.

1.2 commands in a shell [2 points]

Give a short description of the commands below.

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

Answer: Have a look using `man`.

2 Processes

2.1 fork() [2 points]

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

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

int x = 42;
```

Name:_____

```
int main() {

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

Answer: The output will be `x = 43` and `x = 43` since the two processes have their own copy of `x`.

2.2 IDT [2 points*]

What does the IDT (Interrupt Descriptor Table) contain and what happens when a user process executes the instruction `INT` (x86 assembler). Give a short description.

Answer: The IDT is set up by the kernel and contains pointers to procedures that should be executed by different interrupts. When a user process executes for example `INT 80` the process enters *kernel mode* and jumps to the procedure indicated by position 80 (hex).

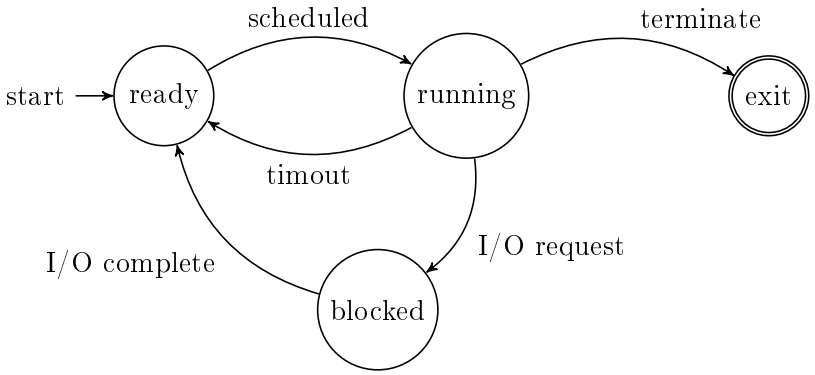
3 Scheduling

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.

Answer:

Name:_____



3.2 shortest job first [2 points]

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

Answer: If we take them is SJF-order we will have a mean turnaround time of $(10 + 30 + 60)/3 = 33$ ms. If we take them in the "wrong" order we will have a turnaround time of $(30 + 50 + 60)/3 = 47$ ms.

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

Answer: It's called stride scheduling. All processes have a value and a stride. The process with the smallest value is scheduled but then will have its stride added to its value.

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?

Answer: A process could use an offset that will address a location outside of the segment. To prevent this a register is used to set a *bound* on the offset.

Name:_____

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

Answer: We don't need to represent the whole table but only a fraction of the virtual address space that is used. This will give us a much smaller data structure to manage - important for a 32-bit address space and completely decisive for a 64-bit space.

4.3 segmented memory x86 [2 points]

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

Answer: It is used to implement memory specific to a thread or core.

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 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?

Answer: In the first alternative we will have internal fragmentation since we loose in average 75 cm in each lot. The second alternative will risk having external fragmentation since empty spaces can be two small for most cars.

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?

Answer: We will immediately be able to tell if the adjacent blocks can be merged with the new block to form a larger block. If the blocks were not ordered by address we would have to search through all blocks.

Name:_____

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 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?

Answer: The two blocks are adjacent to each other but we will not be able to merge them into a 24K block.

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++;
  }
}
```

Answer: The variable `count` can have a final value in the range $[2, 20]$. This is since the `++` operation is not atomic. One thread can read the value, the other do one or more updates and then the first overwrites the changes.

6.2 own stack [2 points]

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

Answer: True - the two threads share the address space and have the same read and write privilege. The have their own stacks but nothing prevents them (all though not advisable) to read or write to each others stacks.

6.3 total store order[2 points*]

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

Name:_____

Answer: In sequential consistency all operations are executed in a total order that preserves that process order. In *total store order* this only applies to write operations, read operations do not have to be performed in the process order. A read operation can be performed even if a write operation to another memory location has not been performed.

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.

Answer: False, a *zombie* is a terminated process where the mother process has not yet collected the result. A file is simply deleted when the last link is removed.

7.2 bit maps [2 points]

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

Answer: The bit maps describe which inode and data blocks that are free to use.

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.

Answer: A directory is represented by an inode. The inode contains general information about the directory: when created, last modified, owner, rights etc. The inode also has one ore more pointers to data blocks that contain a mapping from names to type and inodes. Each entry is a link to a file sub-directory, symbolic link etc.

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.

Name:_____

Answer: True, the operating system executes in user mode to allow all interrupts to be controlled by the *hypervisor* that is supervising the virtualized operating system.

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?

Answer: They can virtualize another hardware than the emulator is running on.

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 ???
    :
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]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

Answer: The first three segments are: code, read-only data and global data for the running process `gurka`. Then there is a segment for the *heap*. The segment marked with `lib-2.23.so` is a shard library. In the uppermost region we find the segment of the *stack*.

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.

Name:_____

```
int main() {

  int pid = fork();

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

Answer: The procedure `exexlp()` will replace the process memory segment with the code and data from the program `gurka`. "Humle" will never be printed unless `execlp()` fails. "Dumle" is written after the termination of the program `gurka`.

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.

Answer: False - you can load a kernel module into a running kernel using the command `insmod`.

9.4 proc_create() [2 points*]

There are several ways in which we can communicate with at 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:_____

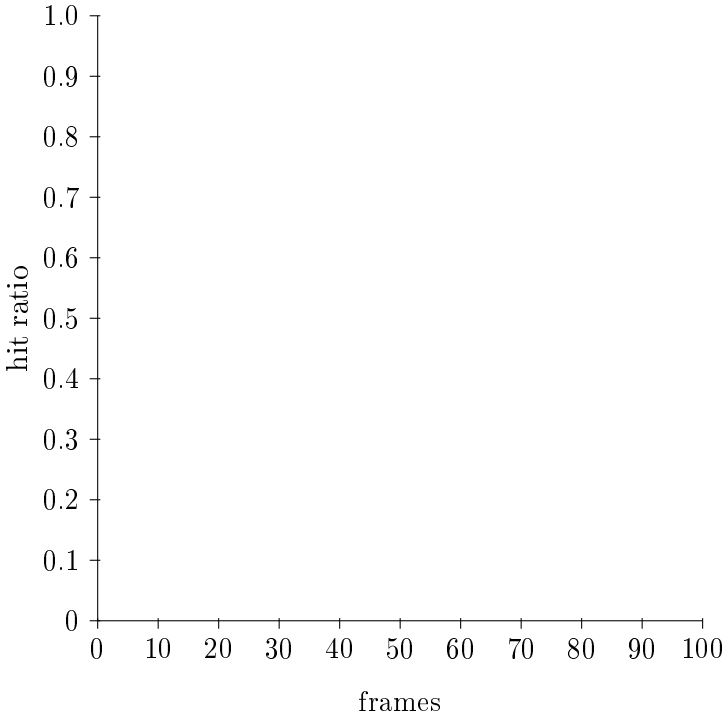
Answer: We register the module as a file under `/proc`. Processes can now use regular file operations to communicate with the module.

9.5 a graph [2 points]

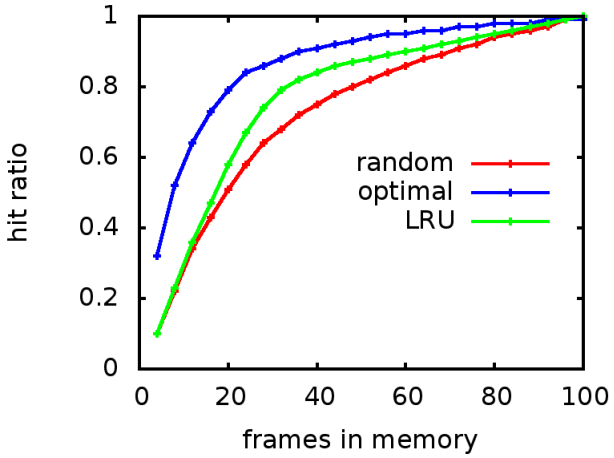
Assume that we do an experiment were we compare three methods to implement a *swapping algorithm* i.e. to throw out pages from memory when more frames are needed. In our experiment we have a virtual memory of 100 pages and simulate a memory of up to 100 frames. The experiment simulate a sequence of memory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



Answer:

9.6 an expensive operation [2 points]

Below is a extract from a program that implements *Least Recently Used* (LRU). The code shows why LRU is expensive to implement and why one probably instead choose 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 {

  :
}
```

Name:_____

Answer: The code unlinks an entry and places it last in a list that should be updated with the least used pages first. This operation must be done every time a page is referenced.

9.7 tick-tack [2 points*]

When implementing the *clock algorithm* it is sufficient to have the list single linked i.e. there is no need for a double linked list. Why can we manage with a single linked list?

Answer: We never remove an element at random but always from the current position of the dial. We can easily keep track of its position and its immediate predecessor.

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) {
  :
  :
  :
}
```

Answer:

```
void lock(volatile int *mutex) {
  while(try(mutex) != 0);
}
```

Name:_____

```
void unlock(volatile int *mutex) {
  *mutex = 0;
}
```

9.9 futex_wait() [2 points]

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

```
int futex_wait(volatile int *futexp) {
  return syscall(SYS_futex, futexp, FUTEX_WAIT, 1, NULL, NULL, 0);
}

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

What do these procedures do and how are they used?

Answer: The procedure `futex_wait()` will suspend the process if the lock is still held. The procedure `futex_wake()` will wake one suspended process (if there are any suspended).

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?

Answer: The advantage with *spin-locks* would be that the locks are held for a short time and a taken lock is probably released in a few clock cycles. If we spin we can avoid suspending a process. The down side is of course if the process that holds the lock is suspended. If we increase the number of threads or have fewer cores the risk for this increase which would then be an argument against using *spin-locks*.