Operating Systems ID1206

(ID2200/06 6hp)

Exam

2019-01-11 14:00-18:00

Instruction

- You are, besides writing material, only allowed to bring one self hand written A4 of notes. The notes are handed in and can not be reused.
- 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 and the hand written page of notes. No additional pages should be handed in.

Grades

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.

Questions with multiple sub-questions are normaly awarded 2p for all correct and 1p for one wrong answer.

Note that, of the 24 basic points only at most 22 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: 12 basic points
- E: 13 basic points
- D: 16 basic points
- C: 20 basic points
- B: 22 basic points and 6 higher points
- A: 22 basic points and 10 higher points

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

Name:_____ Persnr:_____

Answer: The provided answers are not necessarily answers that would result in full points, longer explanations could be required.

1 Processer

1.1 stack or heap [2 points]

What is done in the procedure below and where should `gurka` be allocated? Why? Complete the code so that `gurka` is allocated space.

Answer: The procedure switches value of the two arguments. The variable should be allocated on the stack since it has a known size and is not needed after the call to *tomat()*. This is done by adding a local declaration `int gurka;`.

```
void tomat(int *a, int *b) {
  // allocate room for gurka


  gurka = *a;
  *a = *b;
  *b = gurka;
}
```

1.2 fork() [2 points]

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

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

void foo(int *x) {
  (*x)++;
  }

int main() {
  int global = 17;
  int pid = fork();
  if(pid == 0) {
    foo(&global);
  } else {
```

Name:_____ Persnr:_____

```
    foo(&global);
    wait(NULL);
    printf("global = %d \n", global);
  }
  return 0;
}
```

Answer: The output will be `global = 18` once. The two processes will have their own copy of the stack and hence `global`. Since the updates are independent of each other the final result in both cases is 18. Only the mother process will output the result.

Name:_____ Persnr:_____

1.3 a stack, a bottle and.. [2 points]

You have written the program below to examine what is on the stack.

```
void zot(unsigned long *stop ) {
  unsigned long r = 0x3;
  unsigned long *i;
  for(i = &r;  i <= stop; i++){ printf("%p          0x%lx\n", i, *i);  }
}

void foo(unsigned long *stop ) {
  unsigned long q = 0x2;
  zot(stop);
}

int main() {
  unsigned long p = 0x1;
  foo(&p);
 back:
  printf("  p: %p \n", &p);
  printf("  back: %p \n", &&back);
  return 0;
}
```

This is the print out. Explain what is found at the locations indicated by arrows.

Answer: Previous stack base pointer (EBP) and return address from zot.

```
0x7ffca03d1748      0x3
0x7ffca03d1750      0x7ffca03d1750
0x7ffca03d1758      0xb93d7906926a7d00
0x7ffca03d1760      0x7ffca03d1790        <-------
0x7ffca03d1768      0x55cdac31d78c        <-------
0x7ffca03d1770      0x7ffca03d17d8
0x7ffca03d1778      0x7ffca03d17b0
0x7ffca03d1780      0x1
0x7ffca03d1788      0x2
0x7ffca03d1790      0x7ffca03d17c0
0x7ffca03d1798      0x55cdac31d7c2
0x7ffca03d17a0      0x55cdac31d810
0x7ffca03d17a8      0x12acac31d5f0
0x7ffca03d17b0      0x1
  p: 0x7ffca03d17b0
  back: 0x55cdac31d7c2
```

Name:_____ Persnr:_____

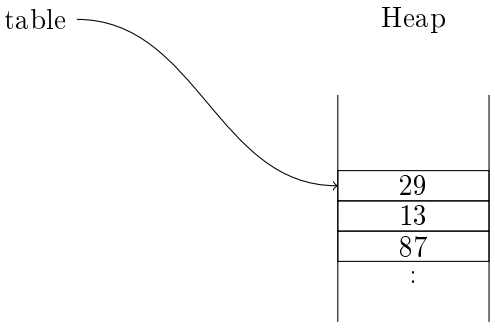
1.4 the size of the block? [2 points]

When implementing *free()* we need to know the size of the block that should be freed. How do we know the size? Draw and explain in the figure below what an implementation could look like.

Answer: You can hide a header before the allocated block where the size of the block is recorded.

```
int *new_table(int elements) {
  return (int*)malloc(sizeof(int)*elements);
}

int main() {)
  int *table = new_table(24);
    :
   table[0] = 29;
   table[1] = 13;
   table[2] = 87;
    :
  free(table);
  return 0;
}
```



Name:_____ Persnr:_____

1.5 IDTR [2 points*]

When executing the `INT` instruction, there is a jump to given position in the IDT. At the same time there is a transition from user mode to kernel mode. This is done to allow the operating system to access its own data structures. What prevents a user from changing IDTR so that it is referring to a table that it controls?

Answer: To change the IDTR is a privileged instruction (LIDT) that can only be done in kernel mode. If this is done in user mode, an interrupt is raised and the operating system is in control (through a jump in the IDT). That the table is placed in kernel space by the operating system, protects the table but not the register per se.

1.6 library call vs system call [2 points*]

An operating system that implements POSIX should provide specified functionality to a user process. Is this provided by system calls, library procedures or a combination of both? Explain the difference between system calls and procedure calls and which parts belong to the operating system.

Answer: POSIX is partly provided by system calls and partly be library procedures, they both belong to the operating system. Library procedures are executed in user mode and can thus work without a costly context switch. Since library procedures are more efficient, as much as possible should be implemented using them. Library procedures are however limited in that they do not have access to the global data structures of the operating system and can thus not provide all functionality. Even if not all can be done i user space it is, as in malloc/free, an advantage to do most of the work there.

Name:_____ Persnr:_____

2 Communication

2.1 a buffer [2 points]

We implement a buffer with one element as shown below (get() defined in similar way). We will have several threads that produce and consume values from the buffer. The buffer is protected by a lock and the threads are synchronized using a conditional variable. The program below does not work (the calls to the pthread procedures are not even legal), why does it not work (even if they where legal)?

Answer: The problem is that we release the lock and then suspend on the conditional variable in two operations. If the process is interrupted between these operations, another process might: call get(), take the lock, remove the item from the buffer and signal. If we now suspend on the conditional we will never wake up.

The code also has the problem that if we do wake up the lock is not held. We also have a problem that a producer can wake up another producer instead of a consumer.

```
#define TRUE 1
#define FALSE 0

volatile int buffer = 0;
volatile int empty = TRUE;

pthread_mutex_t lock;
pthread_cond_t signal;

void put(int value) {
  pthread_mutex_lock(&lock);
  while(TRUE) {
    if(empty) {
      buffer = value;
      empty = FALSE;
      pthread_cond_signal(&signal);
      pthread_mutex_unlock(&lock);
      break;
    } else {
      pthread_mutex_unlock(&lock);
      pthread_cond_wait(&signal);
    }
  }
}
```

Name:_____ Persnr:_____

2.2 pipes [2 points]

The program below opens a pipe and iterates a number of times (ITERATIONS) where each iteration sends a number (BURST) of messages ("0123456789"). We need to handle the situation where the receiving process will not keep up with the sender; how do we implement flow-control to avoid buffer overflow?

Answer: Pipes have builtin flow control so we do not have to do anything.

```
int main() {
  int mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
  mkfifo("sesame", mode);
  // add flow control


  int flag = O_WRONLY;
  int pipe = open("sesame", flag);

  /* produce quickly */
  for(int i = 0; i < ITERATIONS; i++) {
    for(int j = 0; j < BURST; j++) {
      write(pipe, "0123456789", 10);
      // add flow control


    }
    printf("producer burst %d done\n", i);
  }
  printf("producer done\n");
}
```

2.3 SOCK_WHAT [2 points*]

When you create a socket you can choose to create a *SOCK_STREAM* or *SOCK_DGRAM*. Which properties differ and when is it an advantage to choose one over the other.

Answer: The big difference is that *SOCK_STREAM* is a double direction connection providing a sequence of bytes while *SOCK_DGRAM* is a one directional channel for messages of limited size.

The advantage *SOCK_DGRAM* is that the receiver will receive one message at a time and need not think about how to divide a sequence of bytes into messages. If the messages are of limited size it is almost always better to use

Name:_____ Persnr:_____

SOCK_DGRAM. The order is however not guaranteed nor that messages actually arrive. If this is important one has to implement a protocol to keep the order and request resending.

Name:_____ Persnr:_____

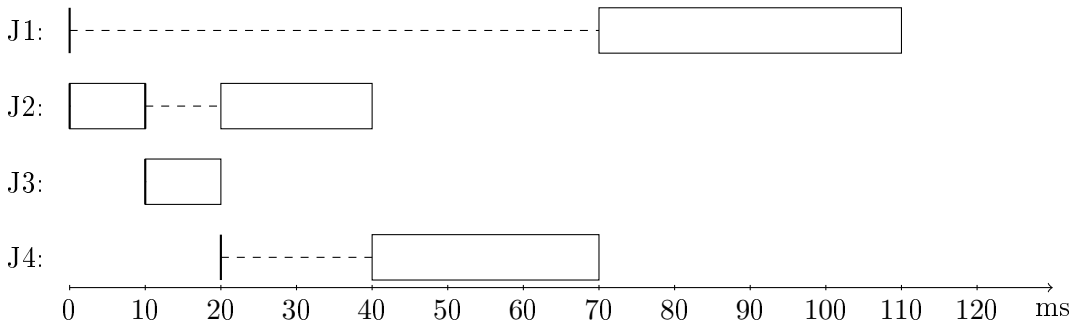
3 Scheduling

3.1 Bonnie Tylor [2 points]

Assume that we have a scheduler that implements *shortest time to completion first* or as it is also called *preemptive shortest job first*. We have four jobs described below as ⟨*arrive at*, *execution time* ⟩ in ms. Draw a time diagram and specify the turnaround time for each of the jobs.

Answer:

- J1 : ⟨0,40⟩ 110 ms
- J2 : ⟨0,30⟩ 40 ms
- J3 : ⟨10,10⟩ 10 ms
- J4 : ⟨20,30⟩ 50s

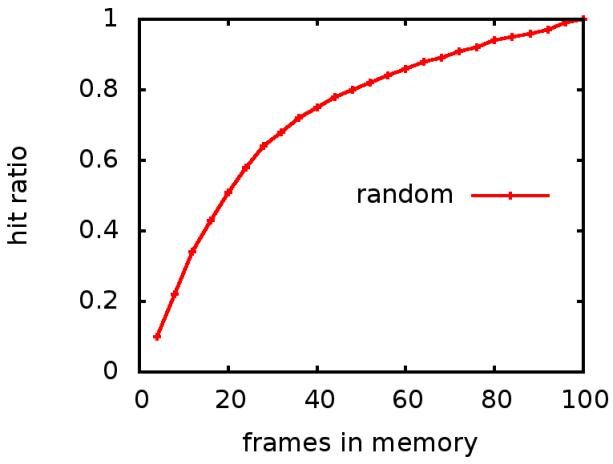


Name:_____ Persnr:_____

3.2 random not that wrong [2 points]

When benchmarking an implementation for swapping one found that a completely random algorithm worked better than expected. In the graph below you can see the ratio of page hits and page references when the size of the memory is changed. The process that runs uses 100 pages and the different runs shows how the hit ratio varies with the size of the memory.

One would expect 50% hit ratio when the memory is half of what the process needs, but the hit ratio is better than this. What could a plausible explanation to the observed behavior be?



Answer: A plausible explanation is that the process does not reference the pages randomly. It could be the case that some pages are referenced more often or that pages are often referenced repeatedly in a sequence (time locality).

Name:_____ Persnr:_____

3.3 priority inversion [2 points*]

When one uses a scheduler with strict priority and uses locks at the same time one can encounter some problems. Describe how a high priority process can wait forever on a lower priority process although the scheduler guarantees that higher priority processes always have precedence before lower priority processes.

Answer: If a process with the lowest priority takes a lock and is then preempted by a higher priority process that also wants the lock, the high priority process will be suspended waiting for the lock. As long as there are processes with higher priority than the low priority process that holds the lock, the high priority process will be blocked.

Name:_____ Persnr:_____

4 Virtual memory

4.1 aproximate what? [2 points]

The clock algorithm, that is used to swap pages from memory, is described as an approximation. What is it that it tries to approximate? Describe a scenario when it does not do the right choice because of the approximation.

Answer: It approximates LRU (*least recently used*). It only notes that a page has been used since the last turn, not how often or when it was accessed. If it is time to swap a page and all pages are marked as being used, the algorithm will clear the markers one by one and then swap the first page. It could be that this page was the page that was access most recently and thus should not be swapped.

4.2 implement the buddy algorithm [2 points]

Assume that you should implment the buddy-algorithm for memory management. To help you a function that locates the buddy of a block of size k is given. Assume that all blocks are taged as either free ortaken and have a field that gives the size of the block. Free blocks also have two pointers that links the block in a double linked list of free blocks of its size.

Assume that you should free a block and have found its buddy - what do you have to check before you can coalece the block with it's buddy? If the blocks can me coaleced, what are the operations that you need to perform.

Answer: You have to check if the buddy is **free** and of the **same size**. If this is the case they can be coalesced. This is done by **removing the buddy** from its free-list, determine which block is the mayor one, change the size of this block. One then must free the coalesced block **recursively**. When no free buddy is found the block is inserted first in the list of its size.

Name:_____ Persnr:_____

4.3 x86_32 addressing 512 byte pages [2 points*]

Assume that someone wants to create a x86-architecture for embedded systems where a large virtual address space is not essential. You're asked to propose a virtual memory architecture.

The world length is 32 bits and we should of course use a paged virtual memory. One requirement is that the page size is 512 bytes so this will set some constraints on your solution.

How do we divide a virtual address using a hierarchical page table based on a page size of 512 bytes? Give an example on how a virtual address should be decoded; the virtual address space need not be fully 32 bits.

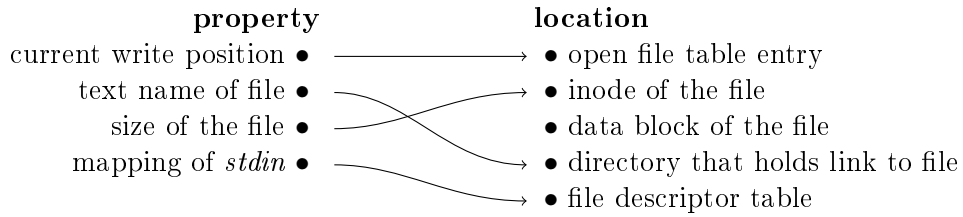
Answer: One proposal is to use 9 bits as offset and then have three levels in the tree with an index of 7 bits per level. Nine bits in the offset to address a page of 512 bytes. The seven bits in the index would allow 128 entries in each page table. If we have 4 bytes per entry this is page table size of 512 bytes which is perfect. The virtual address space is $3 * 7 + 9 = 30$ bits.

Name:_____ Persnr:_____

5 File systems and storage

5.1 which goes where [2 points]

A file has many properties; where do we find the below listen properties? Connect the properties to the left with the correct locations to the right. Several properties might be found at the same location but each property is only found at one location (many to one).



Answer: As shown by arrows above.

5.2 different types [2 points]

A directory can hold links of different types, describe the type of the following five links:

```
drwxrwxr-x 2 johanmon johanmon 4096 dec 21 17:43 bar.doc
-rwxrwxr-x 1 johanmon johanmon 8464 dec 21 22:25 cave
lrwxrwxrwx 1 johanmon johanmon    7 dec 21 17:43 foo.pdf -> ./gurka
-rw-rw-r-- 1 johanmon johanmon    7 dec 21 17:42 gurka
prw-r--r-- 1 johanmon johanmon    0 dec 21 22:25 sesame
```

Answer: *bar.doc* is a directory, *cave* is an executable file, *foo.pdf* is a symbolic (soft) link, *gurka* is a regular file and *sesame* is a *pipe*

Name:_____ Persnr:_____

5.3 journal based fs [2 points*]

Assume that we have a journal based file system. Which of the following statements below is/are correct and which ones is/are false - explain why.

- It is important the the transactions are checkpointed in the same order as theyr were created.
- A transaction is considered to be valid ones it is written to the journal.
- In a restart after a crash, we must be careful not to checkpoint a transaction twice.

Answer:

- Correct:if several transactions changes the same block the result is the last performed.
- Correct: a halfway written transaction is not valid, a fully written transaction is. This is independent of if we have performed the checkpoint, changing the content of the real blocks.
- False: since the operations are idempotent it does not matter if we perform a transaction twice (the only thing that is important is that they are performed in the right order).