Mittuniversitetet
MID SWEDEN UNIVERSITY

Final exam

DT011G Introduction to Operating Systems

Daniel Bosk
daniel.bosk@miun.se
Phone: 060 - 14 8709

2013-01-18

Contents

Instructions

Carefully read the questions before you start answering them. Note the time limit of the exam and plan your answers accordingly. Only answer the question, do not write about subjects remotely related to the question.

Write your answers on separate sheets, not on the exam paper. Only write on one side of the sheets. Start each question on a new sheet.

Make sure you write your answers clearly, if I cannot read an answer the answer will be awarded no points – even if the answer is correct. The questions are *not* sorted by difficulty.

Time 6 hours.

Aids Dictionary, course literature, graded assignments and personal notes.

Maximum points 25

Questions 5

Preliminary grades

E $\geq$ 50% with no question having zero points, D $\geq$ 60%, C $\geq$ 70%, B $\geq$ 80%, A $\geq$ 90%.

Aim

The aim of the exam is to examine that you have fulfilled the requirements specified in the course syllabus.

An excerpt from the life of a systems administrator, or: the exam questions

(4p) 1. A colleague at your work-place is setting up a web-server. He is working with the Apache web-server. The Apache web-server is modular by design, meaning it has loadable modules to get different behaviour on different systems. For instance, it has one module which spawns a child process for each client to handle its requests. Another of Apaches modules creates a thread for each client to handle its requests.

Your colleague asks you: "What are the advantages and disadvantages of each approach? Which one should I pick?"

> **Suggested solution** Threads share memory of a single process [SGG09, p. 153–154], hence less memory is used to handle many clients. This could be the difference between having to swap to disk or not. It will definitely be faster to create a thread for each client than to fork an entire process [SGG09, p. 154] (although the forking in UNIX-like systems is heavily optimized for this).
>
> The disadvantages include more complex programming (concurrent programming is thus more error prone) which increases the risk of buggy code, which of course can be exploited. Another disadvantage is that if a client manages to crash its server, i.e. a thread, then the entire web-server goes down – for all clients! E.g. generating a memory error in one thread will make the operating system terminate the process, not just a single thread.
>
> Furthermore, since memory is shared among threads, this means that the thread of one client can access the memory of a thread of another client. Thus a client could access the web data of another client, containing e.g. credit-card data. This is of course provided there exists a vulnerability which can be exploited, but as stated above: multi-threaded programs are more complex and hence more error prone.

(6p) 2. After work you are visiting a friend, this friend of yours is not very tech-savy (although he himself thinks that he is). When you arrive he says to you: "You know what, I managed to deadlock a ssh and tar pipe the other day." You look strangely at him as he continues, "well, I ran `tar -zcf - <files> | ssh user@server tar -zxf -`," a trick you taught him, "and then I accidentaly disconnected my access point. The program was deadlocked for several minutes before it finally quit."

Explain to your friend what a deadlock really is and why this is not a deadlock.

> **Suggested solution** The problem here is not that of a deadlock, it is a common misuse of terminology. For a deadlock to occur you need the following four conditions to hold [SGG09, pp. 285–287]:
>
> 1. Mutual exclusion: at least one resource must be held by any process in an unsharable way, such that no other process can access the resource while being held by another process.
> 2. Hold and wait: a process must be holding at least one resource and must be waiting for at least one more which is held by another process.
> 3. No preemption: there can be no preemption, i.e. there cannot be any process which may interrupt and force another process to release any resource. Hence we cannot deallocate a resource from one process to reallocate it to another. Thus no kind of priority system may exist.
> 4. Circular wait: there must exist a set of processes $\{P_1, \ldots, P_n\}$ such that P_1 is waiting for P_2, which is waiting for P_3, and so on until P_n which is waiting for P_1.

If these four conditions hold we have a deadlock.

Then why is our scenario above not a deadlock? First, there is no resource for which we have mutual exclusion.

Second, there is no process holding a resource and waiting for another held by a different process. We have one process waiting for another process, the `tar(1)` process on the server waiting for data from that on the client, but not in a way required by our condition.

Third, as there is no resource for which we have mutual exclusion there is no sense in speaking of preemption for any such resource either.

Fourth, there is no circular wait involved. Sure, as stated above the server process waits for data from the client process, but there is no resource which is being held which is waited for.

It is thus inappropriate use of terminology to call this a deadlock. Using an analogy of Sten-Erik Winborg: if you consider the Dining Philosophers Problem, what happened in our scenario is equivalent of removing the bowl in the middle of the table from which the philosophers get their food.

(6p) 3. A little embaressed for his misunderstanding of what a deadlock is (remember, he considers himself as tech-savy) he quickly changes the conversation. "Oh, I have to tell you, I just upgraded my legacy 32-bit hardware for my in-closet server to a bit more modern 64-bit hardware." Whereby you reply: "Nice, did you know something cool, the 64-bit address space is so large that hierarchical paging systems are in general too ineffective to use, one has to resolve to hashed or inverted page tables instead." Your friend looks at you in disbelief and says: "And why is that? How can it be effective for 32-bit systems but not for 64-bit systems? That makes no sense."

How do you answer him? (A tip is to give a convincing example and calculating the effective access time for that example, there is no arguing against that as long as it is a valid example and your calculations are correct.)

Suggested solution First, let us assume that each memory access takes 100 ns. Let us further assume that we use 32-bit physical addresses in a 32-bit system and 64-bit physical addresses in a 64-bit system. Also assume a page size of 4 KiB, i.e. 2^{12} bytes.

For clarity we make no use of TLBs.

In a 32-bit system, we use hierarchical paging because otherwise the page table of each process would be too large: we would need contigous memory to hold $2^{32}/2^{12} = 2^{20}$ entries, each of which being 32 bits, or 4 bytes. That is, we would need 4 MiB of contigous memory. This would counter the benefits of using paged memory and is thus unacceptable. When we introduce hierachical paging we want to fit page tables into single pages (and corresponding frames) [SGG09, p. 337]. By dividing the 20 bits of page number into two 10-bit parts we can use the first part (most significant bits) as the outer page. There are 2^{10} entries which will fit nicely into a single page. Each of these entries contain a page number, we use the inner page number as an offset into this page. The address at this offset is another page, and this is the actual page which we are looking for. Notice that we need to perform three memory references. One for the outer page, one for the inner page, and then the final one to access the actual page. With our assumptions this would mean that the effective access time will be 100 ns + 100 ns + 100 ns = 300 ns. In the single-level paging scheme we first access the page table and then the page, yielding an effective access time of only 200 ns.

Looking at the 64-bit system we realise that we would need a six-level page table: $2^{64}/2^{12} = 2^{52}$ entries to store, i.e. 4 TiB of memory is required to store the page table. For this we need five 10-bit outer pages and one 2-bit outer page. This would require a total of seven memory accesses and hence would yield an effective access time of 700 ns – not as slow as a hard-disk drive, but we are closing in. And alas, we must find another way to do this, we need a lower effective access time.

(3p) 4. Obviously your friend has not taken a course on operating systems concepts. And quite frankly you begin to see his jealousy, especially as he continues: "Well, since you seem to know it all, can you also explain why my files in `/pub` disappears when I try to mount my external harddrive in that directory as well? I would like to see both the files in the directory and those on my external harddrive in there simultaneously."

Explain to your friend why this is not possible.

Suggested solution As this is commonly implemented a directory contains directory entries, each corresponding to a file or subdirectory and has a flag indicating whether it is a mount point or not [SGG09, pp. 444–445]. We could allow the mount-point flag to be set and not ignore the directory entries, but here follows why we do not do this.

The problem becomes obvious when you ask yourself the question *In which file system would we create a new file?*. Changing existing files is not a problem, as the operating system can open them for reading and writing. Although, both directories, one in each file system, would have to be searched for the file. The problem appears when the user wants to create a non-existent file, the operating system cannot possibly know in which directory to create an entry for the file. As the mounting of a file system is transparent to the user the user does not even know that there are two possible file systems and thus cannot notify the operating system in any way which file system to use.

A possible solution would be to always create new files in the directory of the file system using the other directory as a mount point. Since we did mount another file system here it is reasonable to assume we are interested in using it; but this is, of course, not always true.

A further problem is: what happens if the same filename occurs in both directories? Because of the transparency for the user the operating system cannot make a difference between these two files.

(6p) 5. The next day, back in your work-place, you see two strangers, a man and a woman seeming a little lost. To you they look like two stiff hippies walking about the IT-department, feeling very out-of-place. You see them speaking to one of your colleagues and he points towards you. They come over to you and the woman starts talking: "We heard that you are the one to talk to about distributed file system infrastructure." You look back at them with a smile as this is one of your favourite topics. The man continues, "well, you see, we want to connect our Sundsvall, Stockholm, London and Sydney offices with a shared file system. But I say this is not a good idea as the file server would probably end up in London or Sydney, and it is slow as it is already when it is in the same building." He adds "I do editing of raw film material, which is quite large files, and me and my colleagues share this material in our network-mapped home directories right now. I fear that with this system each read and write would require data to travel to Sydney and back again." The woman interrupts him by saying: "I have tried to convince him that there are solutions to this, that there are more advanced file systems than this. You know the details better than me, please tell him."

Explain to them how you would solve the problem to ensure performance to ease his worries, and also how your solution could provide some sort of redundancy in case of a short temporary network failure between any two offices.

Suggested solution The thing which comes to mind is a DFS with location independence. This means a system where the physical location of a file can change from one place to another without any visible change in its logical location [SGG09, p. 707]. One system which implements this is AFS. To see why this helps our two hippie friends we have to overview the implementation.

In AFS terminology there are client machines and server machines. The servers all present the clients with a homogeneous, identical, and location-transparent file hierarchy – the shared name space [SGG09, p. 719].

More than this AFS features client-side caching with cache consistency [**?**, p. 718] which means that the client can cache files used often and that these cached versions are guaranteed to be

consistent across the system. It also includes server-side caching in the form of replicas, with high availability through automatic switchover to a replica if the source server is temporaily unavailable [SGG09, p. 718]. (It also provides authentication using Kerberos.)

At last, it is finally Friday evening.

The end.

References

[SGG09] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons Inc, Hoboken, N.J., 8 edition, 2009. International Student Version.