

Examen de Système – Solutions

14 janvier 2010

40 points could be reached altogether, but 38 already counted as 100 %.

1. For this task one could essentially go through the table of contents of the lecture and list the topics discussed in the course. A non-exhaustive list of examples:
 - Processes: To enable them running them concurrently and allowing communication, one requires at least a scheduler, signals, or other means of communication. One could also mention fork/threads etc here.
 - Memory management: to make sure that processes can peacefully coexist, they must agree on which parts of the memory each one uses, for which the OS provides interfaces. The presence of multiple users (not all of which are trustworthy) requires memory protection. Virtual memory fulfils both functions.
 - File system: also necessary for simpler operating systems, but in a multi-user system it must be extended with owner attribute, access rights. Due to multi-tasking, different accesses at the same time must be handled. One could also mention pipes (which in Unix are handled by the file system).
 - I/O: Both the multi-tasking and the multi-user aspects require to OS to guard access to all I/O ports, to govern common resources and to prevent malicious behaviour.
 - Hardware: Obvious candidates are the memory management unit, without which guarded access to memory/virtual memory is impossible, and the presence of a protected mode in the CPU, otherwise access restrictions cannot be guaranteed.

Altogether five points were awarded, ideally one for each of the aspects mentioned above (but with some flexibility, e.g. if some aspect was covered exhaustively). It wasn't necessary to mention everything listed above, but at least to mention an aspect and maybe give one example of how it is addressed.

2. Generally, in the “piped” version the two commands run in parallel whereas in the second, sequential version, they run one after another. Thus, in the parallel version, `cmd2` could influence `cmd1` via other channels (file system etc.), whereas in the sequential version it cannot.

Evidently, the sequential version will not work if the file `temp` cannot be written to (for which there can be various reasons). Also, the sequential version does not work if `cmd1` depends on communication (via other channels) with `cmd2` to proceed. (Admittedly, the latter case is obscure since in this case one would not generally use the pipe, so this answer wasn’t really expected.)

Whether the sequential version produces the same output depends on whether `cmd2` can influence the outcome of `cmd1`. If yes, then the results might not be the same, otherwise (the two commands are independent) they are. Moreover, the file `temp` is open to manipulation by third processes, whereas the pipe is accessible only to the two commands involved.

As for timing, the output of `cmd1` is communicated via pipes in the parallel version and via files in the sequential version. Both are handled by the file system, but the media used may be different, and hence the timing. (Normally, one would expect main memory vs hard drive, but it depends on the circumstances). Independently of this, the parallel version may be faster if `cmd1` gets blocked now and then, allowing `cmd2` to work during this time; in the sequential version, this time is simply lost.

One point was awarded for each of the three aspects (one half for at least partial answers), three in total.

3. One point for each subquestion, four in total:

- (a) There are two solutions: one is to use a temporary third file, like this:

```
mv titi tata; mv toto titi; mv tata toto
```

The drawback of this is that an existing file called `tata` would be overwritten. Although I did not subtract points for this, a cleaner solution is to make sure you use a file that does not exist. The command `mktemp` gives one such a filename (but knowledge of this was not expected):

```
i='mktemp'; mv titi $i; mv toto titi; mv $i toto
```

A solution without a third file in between is to use I/O redirection, e.g.:

```
(mv toto titi; cat > toto) < titi
```

This works because the shell will first open `titi` to provide input for the command in parentheses, for which it invokes another instance of the shell. That shell will delete the previous file `titi`, but since that file is still open in another process, it will stay around (actually, under another, temporary name, see the `unlink` manpage).

- (b) The easiest solution (probably) is `ls -l *.txt | wc -l`. Common errors were to just use `ls -c` (`ls` has no counting option, `-c` does sorting) or `wc *.txt`, which confuses the *input* on which `wc` is working with its *arguments*. Also, the `-l` was sometimes forgotten in either `ls` or `wc`, but I disregarded that.
- (c) Just do `cat *.txt > toto`. Many people used a `for` loop here, which is unnecessarily complicated. Also, if you want to use a loop, `for i in *.txt` will do, rather than `for i in `ls *.txt``.
- (d) A possible solution:

```
(for i in `cat titi`; do cat $i; done) > toto
```

Using parantheses saves the bother of repeatedly appending to `toto`, after emptying it first.

4. Three points for either subquestion, six points in total. Getting names of C functions wrong was not penalized, nor any punctuation etc. However, confusion between operations on pointers and the objects they point to was not tolerated. (Nor, of course, functional errors.)

- (a) A possible solution looks like this:

```
int prefix (char *s, char *t)
{
    while (*s && *s == *t) s++, t++;
    return !*s;
}
```

Of course, one could do it recursively, but that's not recommendable in terms of stack usage. A Common error was to let the pointers run out-of-bounds.

- (b) A possible solution:

```
int fcomp (char *s, char *t)
{
    char c,d;
    int result = 1;
    int f = open(s,O_RDONLY), g = open(t,O_RDONLY);

    while (1) {
        int m = read(f,&c,1), n = read(g,&d,1);

        if (m != n) break;
        if (!m) { result = 0; break; }
        if (c != d) break;
    }

    close(f); close(g);
}
```

```
    return result;
}
```

The use of the system calls for opening, reading from, and closing a file was expected, their precise syntax disregarded. Simply assuming a function for doing all of that at once was not accepted, as it simplifies the task too much. Also, one cannot treat the contents of a file as a string! Files may contain binary data, naturally also including zeros. A common functional error was to treat difference in size incorrectly. Half a point was deducted for not closing the files in the end – it's not just bad programming style, but can lead to problems when the function is used many times (overflowing the limit on open files, interfering with other processes, degrading performance).

5. Certainly one can describe some aspects of the interaction in various details. The salient points that should be covered are the following:
- The role of the OS to check access rights and determine the physical location of the data, using the file system.
 - The role of DMA; while retrieving the data, the hard drive first stores it in its own built-in buffer (unlike main memory, no interruptions can happen there), then copies it into the assigned part of the memory, then the OS distributes it to the user.
 - The interaction between the three parties, i.e. interrupts and scheduling (process is blocked by the OS, hard drive issues interrupt when finished, OS wakes up the process once the operation is finished).

A maximum of four points was awarded here, one each for covering those three aspects, another one flexibly, e.g., for detailed description or if additional aspects were covered, e.g., buffering or others.

Note: The notion of interrupt is twofold, hardware or software interrupt. A hardware interrupt is necessary for the hard drive to signal the end of its operation; the system call issued by the user process may be realised using a software interrupt but also in other fashion (e.g., this is done in MS-DOS).

6. $4 + 2 + 2 + 4$ points, 12 in total. The last question yielded only 0 or 4 points because there was no sensible way to give partial points for a false answer.
- (a) Let us say that node i is *selected* if it is either active, or critical, or the recipient of an incoming *token* message, or else *non-selected*, i.e. if i is either idle or waiting, and no *token* message is currently on the way to i . Evidently, the non-selected states are those with incoming arcs in the graph, and a selected node should be its root.
- The tree property is an invariant. At the beginning, 0 is selected, the others are not, and the tree consists of one edge from 0 to any other

node. To prove the invariant, we consider actions that change the graph; this may happen when a process changes its selected status (changing its state, or a *token* message being sent or received), or when some *fwd* variable is changed. First, let us note that at only an active node can send a *token* message, and only once, so as long as the invariant holds, there is at most one *token* message in the net at any one time.

Of the four actions that are possible, only one actually changes the tree:

- The first action changes one node from idle to waiting, not changing its selected status.
- The second action does not change the tree; previously, the process was the recipient of an incoming *token* message, now it becomes critical, then active, but it remains selected all the time.
- The fourth action does not perform any of the actions mentioned above.

The third action remains, let k be the node that receiving $sendto(i)$. By the invariant, k is the root of the tree. Sending the token makes i selected, and making k idle makes it non-selected (since no other *token* message can be underway). So we remove from the tree some edge (j, i) . Since k was previously the root, the removal gives us two separate trees, one rooted at k , the other at i . But we also add the edge (i, k) , re-uniting the tree with i as the root.

- (b) Mutual exclusion is directly implied by the tree property; all nodes but one have incoming edges, meaning they are idle or waiting.
- (c) The protocol is deadlock-free. (Well, at least for $n > 1$. But we don't really need one for $n = 1$...). Because the tree property is an invariant, at any point the root, say r , has some direct children. Any child c is either idle or waiting. If it is idle, it can always become waiting and issue $sendto(c)$ to r . If it is already waiting, then it has issued a $sendto$ message, and since only an active process can change its *fwd* value, that message must have gone to r . So either way, a $sendto(c)$ message will eventually arrive at r . Either r responds with the token, or r is no longer the root, in which case it has sent the token elsewhere. But either way, some process can become critical.
- (d) The protocol is not fair. Consider three processes 0, 1, and 2. Initially, 0 has the token. Now consider the following scenario: Both 1 and 2 want to become critical and send $M_1 = sendto(1)$ resp. $M_2 = sendto(2)$ to 0. M_1 arrives first, and 0 sends the token to 1, but before M_2 arrives, it wants to become critical and sends $M_0 = sendto(0)$ to 1, later forwards M_2 to 1. Now, 1 gets the token, becomes critical, then receives M_0 , so it sends the token to 0, then demands the token back by sending another instance of M_1 to 0. While waiting, it receives M_2 and forwards it to 0. Now, we have

the same scenario as before, which can repeat itself infinitely often, and 2 will wait indefinitely.

7. Either part gave three points, six in total.

- (a) The protocol is correct. Necessarily, one node, say m , has the largest identifier. Since message passing is reliable, its message $id(m)$ will eventually reach all nodes, making them inactive, and come back to m . Likewise, its resulting *winner* message will make a complete tour of the ring until it reaches m again. Moreover, no other *id* message will pass beyond m , so no other process can declare itself the winner. Thus, only a finite number of messages will be sent, all of them arrive eventually, and one and only one process will become the winner.
- (b) Consider the maximal node m again. Its *id* and *winner* messages will cause $2n$ messages altogether. There are no other *winner* messages, and all other *id* messages will be blocked at m or earlier. So in the worst case, they all travel all the way to m , the direct neighbour causing 1 message, the next but one 2 messages etc until $n - 1$. So we get an upper bound of $2n + \sum_{i=1}^{n-1} i = n \cdot (n + 3) / 2$. This upper bound is indeed achievable if the nodes are “sorted” in ascending order. The protocol of Dolev et al sends at most $\mathcal{O}(n \cdot \log n)$ messages, this one $\mathcal{O}(n^2)$, so Dolev et al’s is more efficient.