Computer Science 340

Operating Systems

Assignment 1

Due date : 21 August 1998

MIDDLEWARE : MIDDLE, PERHAPS, BUT WHERE ?

This is a nice easy assignment to get you going; do not regard it as setting a pattern for the whole semester. In fact, it's so easy that you should all get full marks, and the only reason I'm bothering with the marking at all is that the markers need a nice easy introduction too. So please make it really easy for them – give all the right answers.

The assignment is about who should do what in computer systems, with some emphasis on the position of "middleware". Read on ...

Except for some very simple tasks, computing work nowadays is rarely done by a single programme. Perhaps it never was, though in the early days it might have seemed so to the computer, because all the connections between the parts were handled by people. As time went by, it was recognised that quite a lot of this organisation and coordination could be done just as well, if not better, by the computer, and job-control languages were developed so that people could write down the instructions needed to coordinate the activities of the programmes which they wanted to use in their jobs.

One result was that one no longer had to write enormous programmes which did all the work of a big job by themselves; instead, one could bring together a number of specialist programmes designed to perform some parts of the job, perhaps writing new software if necessary to carry out activities peculiar to the specific overall task required. The most enthusiastic implementation of this idea was probably the Unix shell, which was specifically designed so that several "filters" (each one a specialised programme) could be chained together in a "pipeline" to perform quite complicated operations on character streams. This is all very modern, and one can rave enthusiastically about encapsulation and reuse of components; at the time (around 1970 ± several) it was just the obvious thing to do. It has worked well ever since.

The system isn't perfect. A significant drawback is the need to switch between programmes in the middle of what might really be a single job. To do so, it is typically necessary to pack up any important data structure in some form which will survive being written to a file and read back again, so that it can be reconstituted at a later stage of the job. (Incidentally, the Unix pipelines work because they are restricted to character streams, which *either* means that there's very little structure in the data *or* forces you to encode the structure you want in a character stream.) While that has always been so, and we are accustomed to recoding our structures for long-term file storage, it has become more difficult as the structures used have become more complicated, and to be forced to do it in the middle of a job is irksome and time-consuming.

For this and other reasons, therefore, it has fairly recently been realised that it would be worthwhile to extend the principle of cooperation between specialist programmes to a smaller scale ("finer granularity" is a term sometimes used), so that the services of different programmes (or whatever – modules, objects, servers, etc.) can be called on without leaving another programme which has built up some complex structure. A good example is the incorporation of services from other programmes into word-processors, so that graphics, spreadsheets, sound, and animation can be included in documents. There are different ways of organising such systems, but the principle of active cooperation between different, and essentially independent, software units is common to them all. A subsidiary, but also important, principle is that there should be a standard interface between the cooperating parts; as well as simplifying the linkage, this makes it possible for new facilities to be used immediately in programmes which were not designed with them in mind.

Middleware has evolved to support this activity. Its function is to provide and manage the links between the top-level software, visible to people using the system, and the specialised units – programmes, objects, or whatever – which carry out the specialised tasks. At the moment, middleware services come as self-contained packages (OLE, changing into something else; OpenDoc, apparently moribund; CORBA, very trendy; etc.).

In this assignment, I want you to think about a quite simple question : should middleware support be provided as part of the operating system ? More precisely, middleware is just an excuse, though it's an interesting one; what I really want you to do is think about what the criteria are for including software in an operating system.

THE TASK.

After all that, it will be an anticlimax to find that we are not providing you with a middleware system to use in the assignment. Perhaps it's just as well that it's only an excuse. Instead, we shall provide one job-control language, and ask you to attempt a task with and without the job-control language – and then to contemplate the advantages which might be offered by the middleware system we don't have. This puts the emphasis on thinking rather than just doing things, which we think is appropriate to a university.

In fact, it's worse than that; middleware hardly appears at all. In the original version of the assignment, I asked you to write down and submit for assessment your opinion on whether a middleware system would be useful for the task to be described, and on whether or not it should be regarded as part of the operating system. Having worked out a good part of the answer I'd have liked you to give, I reluctantly concluded that as an assignment it wouldn't work; you had to imagine too much, and short of giving you a manual for a middleware system I couldn't provide the sort of information you'd need. What you have here, therefore, is a comparatively simple assignment in which you experiment with carrying out a task in two simple ways. Nevertheless, the intention is still to draw to your attention issues of how we can control the flow of work in operating systems, how we can use the facilities available to automate complex operations, and how this depends on the exchange of information between different parts of the system.

The task is a real one, though somewhat simplified. It comes from the annual exercise of preparing the department's handbooks. This is quite complicated, and involves most of the members of the department. From the computing point of view, it involves software for electronic mail, file management, simple text editing, and formatting; graphics is next on the list, but not yet included as an automated part of the system. All these bits are tied together by a set of supervisory "job-control programmes", some of which are still executed manually. The intention is to automate as much of it as we can manage.

Part of this system is the collection of course files. There is a file for each lecture course, each containing a fairly standard set of information about the course. These files are used in several different ways, but for the assignment we shall concentrate on the production of the summary of coursework allocation (page 56 of this year's handbook –

or http://www.cs.auckland.ac.nz/~alan/handbook/ughandbook.9.html), in which one or two items of information from each file must be collected and tabulated. We shall ignore the final "practical" column. This isn't very difficult, but will suffice (with a bit of imagination) to give the raw material for the assignment.

The task itself, therefore is this :

You are provided with three course files, all formatted in the same pattern. From these files, you are to produce a summary file containing one line for each course file, with each line containing the course number (to be taken from the file, not from the filename) and the percentages of the three coursework components in the order given. The items on the line are to be separated by a mark-up symbol for tabulation, which is "|!!tabulate!|" – for example,

```
415.370|!!tabulate!|65%|!!tabulate!|10%|!!tabulate!|25%
```

THE ASSIGNMENT.

(a) Perform the task in two different ways :

1 : Copy the files to a conventional GUI system (Macintosh preferred, but reasonable equivalent acceptable), and perform the task by hand. Write down a set of instructions which someone could follow to repeat the task beginning with a different set of files with the same format.

2 : Copy the files to a Unix system (C shell preferred, but reasonable equivalent acceptable), and write a shell script called "summary" which will carry out the task automatically on being started with the instruction "summary course*".

In both cases, use the facilities available to you to produce a reasonably efficient result, but don't agonise over choices between different methods which look fairly similar in execution efficiency. If there are obvious errors which are easy to detect, do so – don't do anything clever, just put a plausible and visible error message in the summary file. Don't spend a long time worrying over things you can't easily detect.

(b) Suppose that the file names had not been assigned systematically – for example, suppose that the 333, 340, and 350 files had been called FP, OS, and MF respectively. Work out what changes you would have had to make to the instructions from 1 and the shell script from (a) to ensure that a properly sorted file was produced. Again, don't try too hard to optimise the methods. Implement the revised shell script.

(c) The course files are acquired by this process :

Information coordinator sends this year's version by E-mail to course supervisor, with a request to edit it to produce the correct version for next year and return it by E-mail;
Course supervisor replies by E-mail;
The new course file is extracted from the reply and stored.

That's simple – provided that the course supervisor follows the instructions. The new file can then simply be extracted from the E-mail reply (perhaps with some line prefixes like > removed, but that's easy). Unfortunately, course supervisors are all highly intelligent academics who make up their own minds about things, and some decide that they know better than the information coordinator how to do it. They might send replies like the contents of the alternative 340 course file. (That isn't a real example – the 340 course supervisor isn't like that – but it's close to some real cases.) In the current system, this forces a manual step between receiving the reply and further automatic processing. It would be much better to have an automatic system which would handle normal cases by itself but call on the information coordinator to correct a course file if it seems to be faulty.

Think about how you might change the two methods (manual and shell script) to incorporate this suggestion.

NOTES ABOUT THE ASSIGNMENT.

- An API is an Application Programmer Interface – a software interface to a more-or-less self-contained piece of code which gives access to its facilities from other software. Typically, it looks like a collection of procedure calls, each providing a particular well-defined sort of service.

- In writing the shell scripts, do not strive unduly hard to achieve the maximum possible efficiency. (If that's what you wanted, you wouldn't use a shell script anyway.) Aim rather for reasonable efficiency and comprehensibility – remember that the marker has to understand what you've done. Do not make the sort of self-unpacking scripts I've given you in the examples; just make it work.

- Please restrict yourself to the Unix features which appear in the examples. That's not to be deliberately obstructive; it's in the hope that the Unix and word-processor versions might thereby be made more easily comparable.

- Unix documentation can be found in textbooks, the Unix manual, the Unix `man` instruction, http://www.cs.auckland.ac.nz/cgi-bin/man-cgi (a web version of `man`), http://www.cs.auckland.ac.nz/tech-support/software/unix/help/ (an introduction and help). If you use `sed` (which you can hardly avoid if you stick to what's in the examples), it's worth reading the `man` information carefully.

NOTES ABOUT ANYTHING WHATEVER.

- Anyone keen to find out more about middleware might like to look at a review : P. Bernstein : "Middleware : a model for distributed system services", *Communications of the ACM* **39#2**, 86-98 (February, 1996). As the title suggests, the article leans strongly towards the use of middleware in distributed systems, but the principles still work in local systems. You don't need to read the article to do the assignment.

- Anyone incomprehensibly interested in the department's handbook system, and with too much spare time, can find out more from http://www.cs.auckland.ac.nz/~alan/informat/introdoc.htm and references therein. That's not required for the assignment. Note that the description given in this sheet is cooked for the assignment; it is a mixture of current implementation and future plan, though it remains essentially realistic.

SUBMIT FOR ASSESSMENT :

SUBMISSION 1 : The instructions for manual execution of the task, from alternative 1 of (a), and the shell script, and any other relevant material, from alternative 2 of (a).

SUBMISSION 2 : Two diagrams, one describing alternative 1 of (a) and one describing alternative 2 of (a), which look the same as the middleware diagram in the appendix below, but with each text label replaced by the equivalent from the step concerned.

SUBMISSION 3 : A comparison of the two methods, based on the differences between your diagrams.

SUBMISSION 4 : Both new sets of instructions from (b), following the pattern of SUBMISSION 1, with in each case an explanation of why the changes are necessary.

SUBMISSION 5 : The summary files you get from the shell script written in (a) using the three original course files, and the summary file you get if you replace the original 340 file by the alternative version provided.

SUBMISSION 6 : The results of your thinking under (c). Comment on whether it is possible to solve the problems in each of the two methods, and *either* explain how it can be done *or* explain why it can't be done. Give your answers in terms of the interactions between parts of the system which appear in your diagrams for SUBMISSION 2.

WHAT LEVEL OF ANSWER IS APPROPRIATE ?

SUBMISSION 1 : For the manual method : Instructions comprehensible to someone reasonably accustomed to using a word processor. Example :

> Open file xyz;
> Find "dog";
> Select from "dog" to "cat";
> Copy;
> Go back to the top;
> Find "3.1";
> Paste after the line with the "3.1".

For the shell script, etc. : the shell script, etc. Relevant material will include any files required other than the shell script.

SUBMISSION 2 : Don't write essays for the labels, but describe the components in terms similar to those used in the middleware diagram. The intention is to identify similarities and differences in these areas of the overall system.

SUBMISSION 3 : State the topics in which you find differences, then explain them in terms of the labels of the diagrams. Please tabulate your answer; that will help the markers, and make them more benevolent :

Subject	*Word processor*	*Shell script*
<Finding the fish>	Harder because the <word processor equivalent of middleware> has fewer <wheels>	Easier because the <shell equivalent of middleware> has more <wheels>
....	....	....

(You are supposed to replace the bits written as <...> by something relevant to the question. If I gave you a real example, that would answer some of the question, wouldn't it ?)

SUBMISSION 4 : Both instructions and explanations should be set at the levels of question 1.

SUBMISSION 5 : Simply present the files as they come. We just want to be assured that you've done it.

SUBMISSION 6 : Give two short answers, one for each case.

THE THREE COURSE FILES.

The course files follow. There are two end-of-line "characters" (which in some environments are pairs of characters) between fields in the records; there is one end-of-line "character" after the field markers, which all look like `|!|...!|`. In the files themselves, there are no end-of-line "characters" before lines which are indented in the listing below; whether or not end-of-line characters appear after you've copied and pasted depends on the details of the copy-and-paste method, but it's easy enough to write your shell script etc. so that it doesn't matter. Apart from copying and pasting, or whatever else you choose to do, don't mess about with the text of the files; the markers will assume that these are the files you've used, so any deviation from the expected result will be taken as a fault in your programmes. The header lines (415.333 ===...) are not part of the files, and neither are the blank spaces just before them.

415.333 ==

```
|!|number!|
415.333

|!|title!|
Functional Programming and Language Implementation

|!|prerequisites!|
415.212 and 415.233

|!|restrictions!|
415.360, 415.330, 415.733

|!|examination!|
70%

|!|tests!|
0%

|!|assignments!|
30%

|!|where!|
Tamaki

|!|when!|
First Semester

|!|lecturers!|
Dr S. Manoharan (50%)
Dr John Hosking (Supervisor) (50%)

|!|texts required!|
Aho, Sethi, Ullman. Compilers - Principles, Techniques, and Tools,
     Addison Wesley.
Simon Thompson. Haskell: The Craft of Functional Programming, Addison
     Wesley.

|!|description!|

This paper provides an introduction to language implementation and
     studies an alternative style of programming in the form of a
     modern functional language.

|!|contents!|

An overview of the process of compilation and interpretation of
     computer languages. Lexical analysis. Use of lex, a lexical
     analyzer generator. Syntax analysis. Use of yacc, a parser
     generator. Functional programming. Types and polymorphism. Lists.
     Higher-order functions. Recursion and induction. Strictness versus
     laziness. Infinite lists.
```

415.340 ==

|!|number!|
415.340

|!|title!|
Operating Systems

|!|prerequisites!|
(415.210 or 415.212) and (415.231 or 415.233) and (415.232 or 415.234)

|!|restrictions!|
415.341

|!|examination!|
70%

|!|tests!|
10 %

|!|assignments!|
20%

|!|where!|
City and Tamaki

|!|when!|
Second semester, 3 lectures per week

|!|lecturers!|
Dr Alan Creak (supervisor) (50%)
Robert Sheehan (50%)

|!|texts required!|
A. Silberschatz and P.B. Galvin, Operating Systems Concepts (4th or 5th Edition) (Addison-Wesley).

|!|description!|

A computer's operating system is the collection of software which deals with the essential organisational tasks which must be carried out if the machine is to deliver services to people who need them. Its responsibilities range from dealing with the people who use the system, to moment-by-moment allocation of resources, to managing different activities in progress. The paper describes the functions which the operating system must perform, and critically examines ways in which these requirements can be satisfied.

|!|contents!|

Why we need operating systems, and how they have developed. Making the system usable: the computer-human interface. Making the system safe: protection, security. Managing data: memory management, file management. Managing computing: processes, processor management, concurrent processes. Making the system work: driving devices, scheduling.

415.350 ==

```
|!|number!|
415.350

|!|title!|
Mathematical Foundations of Computer Science

|!|prerequisites!|
415.105 and (445.225 or 280.201). Students with a low grade in 445.225
     are strongly discouraged from enrolling.

|!|examination!|
70%

|!|tests!|
10%

|!|assignments!|
20%

|!|where!|
City

|!|when!|
First semester

|!|lecturers!|
Professor Cristian S. Calude (supervisor) (33%)
Dr Bakh Khoussainov (33%)
Assoc Prof Fred Kroon (Philosophy) (33%)

|!|texts required!|
M. Sipser. Introduction to the Theory of Computation, PWS Publishing
     Comp., Boston, 1997.

|!|texts recommended!|
B. Khoussainov, A. Nerode. Automata and Transition Structures,
     Birkhauser, Boston, in preparation.

|!|description!|

The aim of this paper is to present mathematical models for computers
     and computation, and derive results about what can and cannot be
     computed. It deals with idealised computers (automata) which
     operate on idealised input and output (formal languages). For
     example, we prove that it is impossible to write a computer
     program that takes as input any computer program and tells us
     whether or not that program will end up in an endless loop (the
     halting problem). Non-standard models of computation, as DNA and
     quantum paradigms, will also be briefly described. Familiarity
     with abstract mathematics is assumed.

|!|contents!|

Regular languages and finite automata. Tree automata. Context-free
     grammars. Computable functions. Computations on infinite objects.
     DNA and quantum computing.
```

End of normal examples ==

THE ALTERNATIVE 340 FILE.

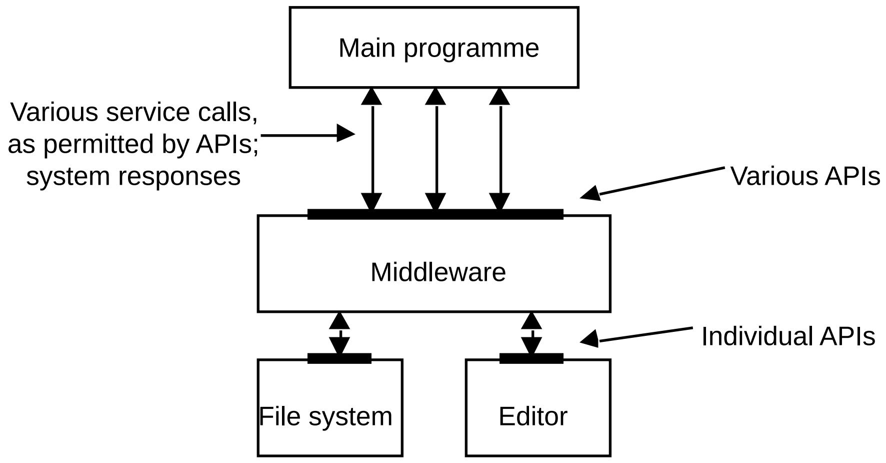
415.340 alternative ==

```
Thanks, that looks good. Just add "or fifth edition, 1998" to the
    recommended book.
```

End of example ==

WHAT ABOUT MIDDLEWARE ?

A middleware system intended to carry out the same task might be organised something like this :



For the simple task described, the "main programme" is a simple programme which does much the same as the shell script but is written in a conventional programming language. It would be likely to have a user interface which doesn't have a lot to do, but might sensibly be used for error reports. In a more complicated system such as the proposed handbook generator, the "main programme" might coordinate the activities of several server programmes, each running as a separate process, perhaps with its own user interface. The appearance of the file system and an editor as servers illustrates the purpose of the middleware; we're accustomed to expecting that the file system will always be there ready and waiting to do things for us, and with middleware we can use other software in the same way.

This includes preserving the state of the "server" process. We don't expect that the file system will start from scratch every time we want to use it, but that it will operate in parallel with our own software. In this way of working, we expect that all the components will continue as processes throughout the lifetime of the cooperation, so the editor can keep running (or waiting) while the rest of the system does other things.

Communications are obviously important, and a primary task of the middleware system. Just what is provided depends on the system, but it is reasonable to assume that the API for the editor service will include (something equivalent to) a set of editing instructions such as you might give with a textual interface can be sent to the editor, and that it should return signals indicating the completion of an instruction, with an indication of what error, if any, occurred.

If you've followed that description carefully, you might have realised that really middleware does rather little that a shell script can't do – but it does it a lot more conveniently. You get the advantages of calling upon – in principle – any programme at any time, but in a high-level language rather than a shell script, and process state is preserved so you don't have to keep writing things to disc. The example is really too simple to make the point well, but the potential advantages of middleware show up best in the sorting example.

A question which was going to appear in the original version of the assignment is : "Should middleware be supplied as part of the operating system, or should it be left as an independent stand-alone programme ?". It's an interesting question both from the point of view of the status of middleware and because it leads one to think hard about what should, or shouldn't, be included in the operating system. My view is that it should be provided by the operating system as a means of managing processes (compare the files system, which is a means of managing files), because it provides a valuable service, and because it is intimately connected with communication between processes. If you look at the reference I gave, you might also be tempted to argue that middleware should be made part of the operating system in self-defence – anything must be better than the proliferation of many, many slightly differing versions of middleware packages listed in the article.

Alan Creak,
July, 1998.