
Teaching multitasking to electronics and computing students

A. F. Armitage

Faculty of Engineering and Computing, Napier University, Edinburgh, Scotland

E-mail: a.armitage@napier.ac.uk

Abstract Although widely used in the solution of engineering problems, multitasking is conceptually as complicated as parallel processing. A simple system that covers the key features of multitasking in a short time is presented. A description is given of its use with a final-year undergraduate class of electronics students who take computing options.

Keywords computer engineering; multitasking; real-time

Engineers who will be working with real-time systems, embedded microprocessors and computer interfaces have to be conversant with the hardware and software of the systems they will be dealing with. There is a tendency for this area to fall between the topics normally taught in an electronics degree and a computing degree. At Napier University, students on the four-year Electronics and Electrical Engineering degree take a common first two years before specialising. Some go on to do two years of Electronics and Computing. This produces a graduate able to cope with the specialised area of the interface between hardware and software in computer systems. These graduates find employment in a variety of roles, especially in telecommunications companies, companies using embedded microprocessors, and companies using computers directly interfaced to hardware for control, test and measurement.

In the final year (the fourth year of the Scottish degree), these students take a module intended to introduce the basic concepts of multitasking. The purpose of this article is to describe the reasons for teaching this topic, where it fits into the course, the methods used, and some of the successes and problems we have encountered.

Why teach multitasking?

Multitasking programs behave effectively as if they were made up of parallel tasks operating independently. This leads to complications compared with ordinary serial code (deadlock, communication between tasks, and so on). Why then do we feel it to be important to teach this topic?

The main reason is that many students from this course go on to jobs where they are writing multitasking code. This is often for embedded processors that interact directly with hardware (one of the main areas of employment for graduates with expertise in electronics and computing). Also, PC-based systems are increasingly using multitasking. For instance, the good old DOS that used to run PCs couldn't multitask. Windows 3.1 (of about ten years ago) could multitask poorly. Windows 98 does it reasonably well, NT 4.0 does it better,

and Windows 2000 does it extremely well. Consequently, students equipped with an understanding of multitasking are more able to cope with programming effectively in these environments. Newer languages, such as Java, also use multithreading, a lightweight form of multitasking. Writing efficient Java code requires an understanding of this process.

Even those who might not program in the conventional sense might come across multitasking systems. It is now common for test software to be written, not with a traditional programming language, but with a graphical programming language, such as LabVIEW, or HP-VEE. This allows test and measurement software to be constructed by connecting together blocks in graphical wiring diagrams. The latest release of LabVIEW (version 5) has just incorporated multithreading.

The future belongs not just to multitasking/multithreading applications running to a single processor, but to parallel code running on multiple processors. Ten years ago we were teaching Occam in the sure and certain knowledge that Occam and transputers were the way of the future. As it happened, this is not how things worked out. Although Occam is still a wonderful educational language, the transputer has died the death of a revolutionary processor that did not have the backing of a major company such as Intel or Motorola. What has happened instead is that 'mainstream' processors have taken about a decade to catch up in the level of parallelism supported. However, they have now reached this level, to the extent that Windows 2000 Server comes in different versions depending on how many processors you want to support. To cope with these changes, it is important that students working in the area between electronics and computing become at least familiar with some form of multitasking or parallelism (conceptually the two are almost identical).

Where does it fit into a degree course?

The four-year Honours degree at Napier University is based on a modular, semester system. Typically, the student takes four modules per semester with two lectures per week and two hours of laboratory/tutorial time per week for each module. Each semester lasts 15 weeks, with two weeks at the end given over to revision and exams. In the first two years, common to all electrical and electronic students, one computing module is taken per year. Although there is more that could be taught, the crowded syllabus limits the time that can be devoted to microprocessor and computing topics. The first year module is a general introduction to PCs, and can be taught in any general-purpose PC lab. The second year module covers interfacing. A dedicated hardware lab is used, with 20 PC workstations connected to motors, sensors and so on via interface hardware (ADCs and DACs, digital I/O, etc.). Student numbers are such that the lab has to be repeated a number of times during the week, but the cost of the special interfacing hardware is high, and we can only maintain a single dedicated laboratory (general-purpose PC labs with no interface hardware are much cheaper to maintain).

The third and fourth years are taken only by students on the Electronics and Computing route. This means that numbers are smaller, and it is possible to include more modules on computing and microprocessor subjects. In the third year, two modules are taken. One covers software subjects and software engineering. The other is concerned with embedded processor systems and microprocessor hardware. Microprocessor circuitry is covered in some depth, building on digital electronics subjects taken in the first two years. For the embedded processor, we use an Analog Devices 2100 series microprocessor, which has DSP features useful for other sections of the course. Finally, a fourth-year module covers some advanced computer architecture issues and the multitasking that is the subject of this article. Because the module covers general computer architecture issues as well as multitasking, the time available for the multitasking is severely limited. One lecture per week, and about 8 weeks of laboratory work are scheduled. The practical work (very important for reinforcing concepts) culminates in a practical exercise that counts for 40% of the module marks. There is then a small section on this topic in the end of semester exam, though the bulk of the exam is devoted to the more theoretical computer architecture. Although lab work is found to be useful for reinforcing learning, it also worth mentioning that lab work is important to engage the students' interest and make the subject appear relevant.

What to teach?

There are a number of key concepts that have to be introduced when covering multitasking or parallel computing. Most students have been exposed to serially executing code, and will find some of the ideas very unfamiliar. This is the area where practical lab work comes in, as this is the most effective way of instilling deep learning. For instance, one problem encountered is deadlock, a situation where two or more processes cannot progress because each is waiting for another. It does not matter how much this is explained in the classroom; until the student has experienced it for herself in the lab (and had to work out how to cure it) the concept will not be grasped.

In the limited time available, we feel that there are three main areas that need to be covered. These are mutual exclusion, synchronisation, and communication between tasks. Additionally, the idea of design for multitasking/parallelism can be introduced. Relatively speaking this is an area that is poorly served by design tools, and in a short course this area is only lightly touched on. However, as the aim is to produce engineers who have a good systems approach, rather than just programmers, it is important to give at least an introduction to design methods.

Mutual exclusion

Tasks in a multitasking system run independently. Occasionally, these tasks will need to access resources that are shared. These could be blocks of memory, or input and output devices. Problems can be encountered if more than one

task can access a shared resource at the same time. Of course, on a single processor the access cannot really be simultaneous, since only one process runs at any one time. In practice execution can be switched between tasks at any time, and to all intents and purposes the system behaves as if the tasks were running in parallel. In some situations it becomes essential to enforce mutual exclusion. In other words, while one task accesses a shared resource, other tasks have to be denied access to the resource (though they can continue running if they do not need access to that particular resource). Semaphores are a common way of implementing mutual exclusion, and this is an important feature to teach. In fact, the first laboratory given to the students involves running two simple tasks that both try to access the screen at the same time, resulting in garbled output. Once semaphores are added to enforce mutual exclusion, the output becomes clear.

Synchronisation

Tasks may need to synchronise for a number of reasons. For instance, a consumer task may be designed to use data produced by another task. Obviously, the consumer cannot run until the other task has produced the data. Or a master task may 'farm' out work to a number of other tasks. Until all these tasks are complete, the master task must not proceed. There must be some way of synchronising tasks so that progress in task A can be halted until task B has reached a certain stage. There are several methods for doing this, such as wait-signal pairs. In practice, the semaphores mentioned in the previous section can be used for this, and we have found that students who have been using semaphores for mutual exclusion soon get the hang of using them for synchronisation.

Communication

In a multitasking system, tasks are scheduled at run time. This means that, at the time they are written, the order of running is not known. This complicates inter-task communication: when one task wants to send data to another, the sending task does not know what stage the receiving task has got to. There are a number of ways round this problem, involving mechanisms such as the ADA rendezvous, Tony Hoare's Communicating Sequential Processes, and the use of dedicated communications links, such as mailboxes, buffers and shared memory areas. We have found that only one or two of these methods should be covered in detail in lectures: it is best to let the student get experience in using at least one method practically, rather than exposing them briefly to too many alternatives in the lectures. Having said that, it is important to make the students aware of the range of communication alternatives available in commercial systems.

How to teach

Some possibilities

As has already been mentioned, lab work is an essential part of the educational process when teaching this subject. Lectures on this subject are relatively

straightforward, labs less so. Part of the problem is that there are many ways of introducing multitasking in lab work, but some are complex and not suitable for a short course.

Originally, we investigated the use of commercial operating systems. There are a large number of these, and many provide useful information and demos via their websites (see references). The problem we found with these operating systems was that they were too complex to learn quickly. For example, the call to create a process under OS-9 (a Unix-like operating system) looks like this:

```
Os9fork(modname,parmsize,paramptr,type,lang,datasize,prior)
```

Where modname is a pointer to the module name, parmsize is the size of the parameter list, paramptr is a pointer to the parameter list, type is the type of module, lang is the module language, datasize is the additional memory for the process in bytes, and prior is the initial priority of the process.

Unfortunately, this level of complexity is far too great for our students to handle. It would take the best part of a lecture just to cover this one call. Having looked at a number of these commercial systems over a number of years, it was decided that they were all too complex for a short course. This has not always been the case. In discussing a multitasking lab at Purdue University, Schultz¹ describes the use of DCX, a small real-time executive for early Intel microcontrollers. This had the advantage of only having 10 system calls. Unfortunately from an educational point of view, this sort of executive has now been replaced by much more sophisticated systems designed to make full use of current microcontrollers and microprocessors.

More recently, Windows itself has become a possibility for teaching multitasking. Version 3.1 used co-operative multitasking in a rather inefficient way (it was very prone to crashing). However, recent versions of Windows multitask in a more sophisticated manner. Also, Windows CE has been released for the embedded market. It has not yet achieved the dominance in the embedded environment that other versions have achieved in the PC environment (and probably never will). However, it uses essentially the same programming environment as its desktop relatives. It was certainly worth investigating. If the students could learn some embedded Windows programming, and were able to apply that directly to desktop programming, that would be a huge bonus. On closer inspection, this advantage was completely outweighed by the complexity of the environment. For instance, the description in the manual of the process call to create a new process (the equivalent of the OS-9 fork above) is:

```
BOOL CreateProcess (LPCTSTR lpApplicationName, LPTSTR
lpCommandLine, NULL, NULL, FALSE, DWORD dwCreationFlags,
NULL, NULL, NULL, LPPROCESS_INFORMATION
lpProcessInformation);
```

This is then followed in the manual by a detailed description of all the parameters and their use. I do not need to reproduce them here to convey the idea that this is a complicated system! While it is useful in the lectures to illustrate

to students the complexities of some real systems, the various versions of Windows are not ideal teaching tools.

A simple system

Obviously, what is needed is a simpler system that can be used to let the students practise some multitasking without getting involved in the complexities of a full commercial operating system. It is possible to write a multitasking kernel, as has been done at Queen's University.² Such a system was introduced in an article some years ago by Craig Lindley.³ He produced a small multitasking system as an add-on to Borland's Turbo Pascal. As this was the main teaching language we were using at the time, we looked at it, and decided it worked admirably as a teaching tool (and had the added advantage that it was free). It is very limited in scope, as it uses co-operative multitasking without priority. However it is beautifully simple, having 3 initialisation procedures and 9 function calls. To create a new task called Task1, for example, the code is

```
fork;
if child_process then Task1;
```

The call to fork takes no parameters, all that needs to be done is that the value of the variable child_process has to be checked to ensure the creation of the new process was successful. Other functions are:

yield;	— voluntarily yield this task to let others run
pause(n);	— pause for <i>n</i> quarter second ticks
wait; send;	— synchronise tasks by sending a signal
put_byte(byte_val,buffer_name);	— send data to a buffer
byte_val := get_byte(buffer_name);	— get data from a buffer
alloc(semaphore_name);	— book a semaphore
dealloc(semaphore_name)	— release a semaphore

This simple set of multitasking constructs is perfectly adequate to let the students explore some of the more interesting features of multitasking, without getting too bogged down in details. Using this system, the students explore communication, synchronisation and mutual exclusion through a sequence of short laboratory exercises. Having worked through these labs, the students are then given an extended assignment over a number of weeks. In past years, these systems have included data acquisition systems, d.c. motor control systems, and the control of multiple stepper motors. This last exercise has been particularly useful at showing the advantages of multitasking. Having developed the code for one stepper motor, it is relatively simple to duplicate the task, with a few minor changes, to control another stepper motor. In fact, the assignment became so simple that we have had to make it more challenging by specifying different control strategies (position and velocity control) for different motors.

Limitations of the simple system

Some limitations (such as a lack of priority) are inherent to the system. However, there are other limitations. For instance, it is not a true multitasking kernel or operating system, but an add-on to Turbo Pascal. To switch from one task to another, a few special calls, such as `yield` and `wait`, are added. These are function calls that use a small amount of assembly language to mess around with the stack. Instead of returning to the calling function, these jump to the next task in a list of ready tasks. This manipulation of the stack has some unfortunate side effects: a slight decrease in reliability, and the inability of the debug tools to decipher what is going on. This can make the multitasking code harder to debug. Another unfortunate side effect is that the original system (designed for version 3.0 of Turbo Pascal) needed some modification to work with later versions. This was because the parameter passing to functions, using the stack, changed between versions. The current version works with version 5.0 of Turbo Pascal. This is not the most recent version of the language, but it is the one we happen to have installed in our interfacing/embedded system laboratory.

The language of the system, Pascal, is excellent for teaching. However, the final year of the course hosts a number of direct-entry students from continental European countries. These students tend not to be familiar with Pascal. We have therefore been re-writing the multitasking system in C, a language many of the direct-entry students are familiar with. This is an on-going project, but the hope is that for the next academic session, students will be able to pick C or Pascal as they prefer, but work with essentially the same multitasking commands.

Looking ahead

Inevitably (but regrettably), Pascal is going to fade out of our labs as we bow to the continuing pressure to move to C/C++. The limitations of the current system are such that, as we look to changing our language, we are also looking at alternative multitasking systems. One, which looks very promising, is the Tempo kernel developed at Carleton University in Canada, and described in the book by Buhr and Bailey.⁴ This is a small real-time kernel based on a subset of C++. The source code is freely available,⁵ together with some examples. We have not yet used it in undergraduate laboratories, but it appears to have many of the features we are looking for. Most importantly, it is relatively simple, with only 23 calls. Using the call to create a new task for comparison with previously discussed systems, we find that it is simple:

```
Pid=create_process(root_function, priority);
```

It is slightly more complicated than the Pascal system currently in use, in that it needs an initial priority. However, it is possible to miss even that out if the user wishes to bypass the priority system. All in all, this looks like being a very useful system for teaching at this level.

Design methods

This is a difficult area to cover in an undergraduate course. In general, parallel processing and multitasking is an area of software engineering that is relatively poorly supported by design tools and methods. At low levels, Petri nets can be useful for analysing the design of a multitasking system.⁶ Data flow diagrams can give a useful overview of the parallel flows of data within a system. These diagrams form a mainstay of traditional structured analysis methods. In the real-time world, the most widely used structured methods have been variants of the Yourdon method. Two particular variants, Ward-Mellor, and Hatley-Pirbhai (also referred to as Boeing-Hatley), are widespread enough to be supported by CASE (Computer Aided Software Engineering) tools from a number of vendors. For instance, we have been using EasyCASE, an inexpensive CASE tool that supports both of these methods.⁷ An advantage of these methods is that they naturally incorporate state transition diagrams, a modelling tool used in sequential logic design, and hence familiar to many electronics engineers. The use of such CASE tools in electronic engineering education has been described by Cooling.⁸

More recently though, the main debate in methodologies has been whether to switch to object-oriented methods. These methods, particularly when used with languages such as C++ and Java, have become widespread in many areas of software engineering. However, they are perhaps not as well suited to embedded systems (for instance they tend to be memory-hungry). A recent article by Moore⁹ highlights the difficulties of trying to get a multitasking problem to fit into an object-oriented methodology. Having said that, it looks like the Unified Modelling Language (UML) is going to be widely adopted in the object-oriented community. Although not specifically aimed at real-time systems, it does include some valuable tools, such as very comprehensive support of state diagrams. The interested reader is referred to the book by Douglas¹⁰ for further details.

Teaching resources

For those interested in reading further on this subject, I have found a number of sources useful. The websites of the major commercial real-time/embedded software organisations listed in the references provide the most up-to-date information. A number of books cover the area. There are a lot of books on operating systems (for instance, Tanenbaum, Milenkovic, Silberschatz).¹¹⁻¹³ Unfortunately, many of them are written from the point of view of a computing science course, often using Unix as an illustration. This is not very helpful for a course specialising in the sort of embedded systems most likely to be of relevance to electronics engineers. Milenkovic is good at covering real-time systems, but is now rather dated. Silberschatz has a good section on Process (i.e. task) management, and considers real-time systems, if somewhat briefly. This book also takes a detailed look at Java, including multiple threads.

Although Java has not yet been widely adopted as a real-time language, this may change over the next few years. It will certainly be a popular development tool for internet-enabled embedded systems.

Several books that are definitely aimed more at the electronics engineer working with computer systems are those by Bennett,⁶ and Olsson and Piani.¹⁴ Bennett has several chapters that are particularly relevant. As well as several chapters on real-time systems, he is one of the few authors to cover design methodologies for real-time systems. He chooses the most widely used real-time variants of the Yourdon structured analysis method, and Mascot, as used in UK defence work (though not so widely nowadays). Olsson and Piani have less of relevance to this subject area, but do have a good chapter on real-time and multitasking considerations. They devote more space to control aspects and interfacing than Bennett does. The multitasking Pascal system referred to in this article is not a commercial product, but the author would be happy to hear from people who would be interested in using it.

References

- 1 T. W. Schultz, 'Peripheral hardware and a hands-on multitasking lab', *IEEE Micro*, (Feb. 1991), 30–81.
- 2 N. Manjikian, 'Educational applications and benefits of a compact multitasking kernel for microcontrollers', in *Proc. IEEE Canadian Conf. On Electrical and Computer Eng.*, Edmonton, 9–12 May 1999, pp. 421–426.
- 3 C. A. Lindley, 'Multitasking with Turbo Pascal', *Dr. Dobb's Journal*, (July 1987), 42–73.
- 4 R. J. A. Buhr and D. L. Bailey, *An Introduction to Real-Time Systems* (Prentice Hall, Englewood Cliffs, 1999).
- 5 <http://www.sce.carleton.ca/ftp/pub/RealTimeBook/>
- 6 S. Bennett, *Real-Time Computer Control: An Introduction*, 2nd edn (Prentice Hall, Englewood Cliffs, 1994).
- 7 EasyCASE Version 4.23, Evergreen Software Tools, Inc., 15444 NE 95th Street, Suite 244, Redmond WA 98052, USA.
- 8 J. E. Cooling, 'Methodology and CASE tools in real-time embedded systems', *Int. J. Elec. Enging. Educ.*, **33** (1996), 165–178.
- 9 A. Moore, 'Why tasks aren't objects and how to integrate them in your design', *Embedded Systems*, (February 2000), 18–30.
- 10 B. P. Douglas, *Real-Time UML: Developing Efficient Objects for Embedded Systems* (Addison Wesley, Harlow, 1998).
- 11 A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*, 2nd edn (Prentice Hall, Englewood Cliffs, 1997).
- 12 M. Milenkovic, *Operating Systems* (McGraw-Hill, New York, 1987).
- 13 A. Silberschatz, P. Galvin and G. Gagne, *Applied Operating Systems Concepts* (John Wiley, Chichester, 2000).
- 14 G. Olsson and G. Piani, *Computer Systems for Automation and Control* (Prentice Hall, Englewood Cliffs, 1992).

Appendix

Websites of commercial real-time software suppliers

<http://www.qnx.com>

<http://www.lynx.com>

<http://www.microware.com>
<http://www.enea.com>
<http://www.hiware.com>
<http://www.express.logic.com>
<http://www.smxinfo.com>
<http://www.ghs.com>
<http://www.avocetsystems.com>
<http://www.wrs.com>