PEOPLE AND COMPUTERS

If we are to discuss how to run computers (which is the purpose for which operating systems were invented), then we need some way of defining just how we want them to be run. It is, after all, possible to make a computer do useful things with no noticeable operating system at all – which is how people managed when computers were first built. If we want any more than that, we have to say what sort of facilities we need; and a good way of finding out the answer to that question is to analyse how we want to use the machines, and what sort of behaviour we would like.

In this course, we shall look specifically at computer systems which are used by people to perform fairly conventional computing tasks. Both ends of that specification impose constraints on the course material, and it is appropriate to show what these are.

PEOPLE.

We choose to concentrate on "computer systems which are *used by people*". In a sense, of course, all computers are used by people; but we intend to exclude those computers used primarily to control machines, or networks, or telephone exchanges, or whatever, which are normally intended to be invisible to all but a few people knowledgeable in the relevant field. That means that we rule out of consideration most of the world's computers. (There are still quite a lot left.) This might seem to be a rather drastic simplification, but it turns out to be sensible enough for the course for three reasons :

- First, it focuses attention on an important sort of system, and helps us to narrow down the field in a consistent way;
- Second, the low-level characteristics of operating systems are common throughout the field;
- Third, there isn't much overlap between the special requirements of control and real-time systems and the conventional systems we treat in the course.

CONVENTIONAL COMPUTING.

We use this term to mean, roughly, activities which can be essentially completed *within the computer*. We do not include external activities, except insofar as these can be controlled by the computer programme – so the course will include material on driving common external devices but not on any details of their internal operation.

WHAT DO PEOPLE WANT TO DO WITH COMPUTERS ?

At a detailed level, that's a silly question, because people want to do many different things with computers. Any attempt to list all the requirements would be silly, but to illustrate the range here are a few examples :

Commercial computing people want accountancy, banking, administration;

Technological computing people want real-time calculation, mathematics, simulation;

Recreational computing people want silly games, music;

Rehabilitation computing people want special user interfaces, robot controls. (Rehabilitation computing is the study of computer applications which might be of use to people with disabilities. It's one of our interests, and will turn up from time to time.)

Not surprisingly, all the requirements are different – and the whole point of the *general-purpose* digital computer is that it will work for all of them. Equally unsurprisingly, the diversity isn't a good start for building an operating system which must work in a general-purpose digital computer; we'd like a single, simple, universal goal which gives a clear guide as to how we should design our system. We therefore look for a general statement which will encompass all of these special cases, and any others which might

come along – and we can find one. We are not sure whether this is an astonishing coincidence or boringly predictable.

The unifying principle is obvious once you see it : with rather few exceptions, people do not want to run programmes on computers – they want to *get work done*. They use a computer as a means to an end, not as an end in itself. They would use a crystal ball, or a pack of trained monkeys, or Santa Claus and his reindeer if they were guaranteed to do the job quickly and reliably and cheaply. If the operating system is to be effective in helping people to use the computer, it must make this task as easy as possible. How can the task be formulated ? Consider this suggestion :

TO PRODUCE RESULTS AS INSTRUCTED.

That emphasises two features of the task : the system has to be told what to *do*, and the system has to be *told* what to do. How the task is completed is not necessarily a part of the instruction – though if the person desiring the results wishes to give such a specification, it should be possible.

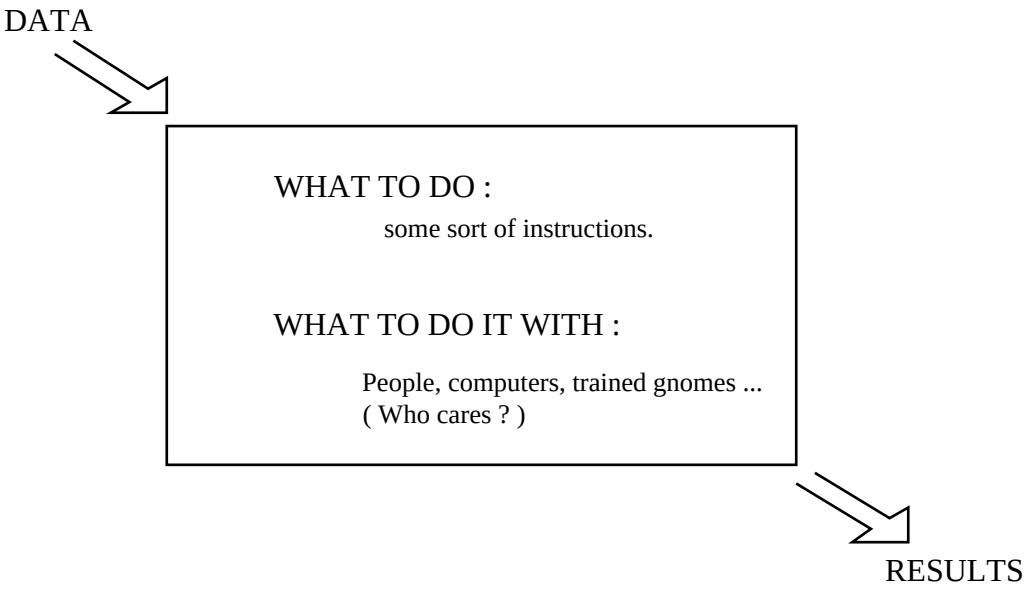
It's also worth noticing that "telling the computer what to do" is expressed as "telling it what results we want" – not what programmes to run, which fits in with the ideas we expressed earlier. Computists often find this idea hard to accept, because so many of them spend a large proportion of their time worrying about their programmes, and how to make them work. If you are dubious about or claims, you might like to reflect on these two points :

1 : They are *your* programmes, but (unless you're just indulging yourself) they're likely to be used with *someone else*'s data. The system has to work for the someone else, who is quite likely to be supremely uninterested in anything about your programme except whether it's correct, how easy it is to use, and how fast the results appear.

2 : When you are writing your programmes, what do you expect the computer to do for you ? You expect it (among other things) to convert your source programme into executable code. While you're doing that, and assuming that it works well, do you spend a lot of time thinking about how the compiler was written ? We don't; we assume that the compiler will do its job, and that the system will get it and set it going in whatever way is appropriate. We want the computer to produce results as instructed.

It is therefore the task of the operating system to take our instructions, and to cause those things to happen which are needed to obey them. It must first accept and "understand" instructions in some form, then it must execute programmes (because it can't do anything else) to perform the required job. We'll have to put in – somehow or other – instructions to convey what we want the machine to do next, and also any information it needs to do the job, and we'll expect to get out some sort of result. Whatever our operating system does, it must provide for these two operations of input and output. A pictorial view of this proposition appears on the next page.

This view emphasises the importance of the communication between the people and the computer system, and draws our attention to the means by which this communication is carried out. The collection of devices, software, and methods which make up this part of the computer system is often called the *user interface*, and that's where we shall start our discussion of the system.



THE USER INTERFACE.

After our prodigious feat of simplification which led us to express all of computing in a single line of text, it is something of a disappointment that things immediately become more complicated. The user interface is a good example of this tendency; here are some of the topics which are important in designing the interface :

Instructions : The interface must be able to convey instructions to the operating system for running programmes, and instructions to the programmes themselves, and instructions to the system for internal operations such as file movements.

Information : The interface should provide information which will help people to use the system. This might be diagnosis of faults which have occurred, information on how to use programmes, news about system availability. etc.

Ease of use : The interface should be designed so that it is easy to learn and to use when performing the required tasks.

It is not easy to design a good user interface, but it is the key to constructing an effective operating system. The table on the next page shows some guidelines[REQ1] which apply to all sorts of interfaces, and repay careful study.

As a final comment, it's important that communications between person and computer should be understood. It's all done by symbols of one sort or another : the software displays a symbol on the screen, and the person must understand it; the person conveys a symbol to the interface device (by touching a key or moving a mouse or whatever), and the system must understand. The symbols used for these purposes should be carefully designed, or misunderstandings can happen. Symbols (more precisely called *signs*) are the raw material of *semiotics*, in which the relationships between signs and how we understand them are studied.

We won't tell you much about semiotics (because we don't know much), but one interesting notion which has been clarified is the twofold nature of a sign. It is a combination of *signifier* (the picture on the screen, the key press) and *signified* (what is understood as the meaning of the signifier). The relationship between the two can be strongly modified by context – so an exclamation mark (!) can mean several different things in different circumstances :

Ordinary text	Exclamation
Road sign	Hazard
Mathematics	Factorial
C language	Negation

If you use signs in the wrong context, misunderstanding is likely; if you design a user interface, make sure that the vocabulary used (both input and output) is appropriate to the context, which includes the sort of work being carried out, the experience of the person using the system, and so on.

Simple and natural dialogue : Dialogues should not contain information that is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility. All information should appear in a natural and logical order.

Speak the user's language : The dialogue should be expressed clearly in words, phrases, and concepts familiar to the user, rather than in system-oriented terms.

Minimize the user's memory load : The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

Consistency : Users should not have to wonder whether different words, situations, or actions mean the same thing.

Provide feedback : The system should always keep users informed about what is going on through appropriate feedback within reasonable time.

Provide clearly marked exits : Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue.

Provide shortcuts : Clever shortcuts – unseen by the novice user – may often speed up the interaction for the expert user such that the system caters to both inexperienced and experienced users.

Good error messages : They should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Prevent errors : Even better than good error messages is a careful design that prevents a problem from occurring in the first place.

COMPARE :

Lane and Mooney[INT3], Chapter 1.

REFERENCE.

REQ1 : J. Nielsen : "Traditional dialogue design applied to modern user interfaces", *Comm.ACM* **33#10**, 109 (October 1990).

QUESTIONS.

Is "getting work done" the same as "running a programme" ?

If it isn't the same, does it make any difference to the operating system ? How should we go about helping people to "get work done" more effectively ?
