CHANGING THE CONFIGURATION.

Computer systems rarely remain static for any length of time. New facilities are required, machines are replaced by newer and slightly different – or sometimes very different – models, changing circumstances lead to changes in patterns of work, and – on a smaller time scale – both software and hardware might have to be temporarily withdrawn for repairs or maintenance. With the increasing popularity of networks and distributed systems, significant characteristics of the overall system can change without warning at any time.

An operating system designed to make it easy to cope with these changes is a great aid to smooth operations. Many early systems – and some not so early systems – effectively hard-wired themselves during initial configuration, so they were immutable, and you had to stop them and restart with a new configuration to change anything.

(That's still very common is microcomputer systems, but it shouldn't be. The inability to change something in the configuration without saving what you're doing and restarting is bad enough within a programme, and can be a serious problem in a system. It can be intolerable in a microcomputer which is multiprogrammed, or which provides some service to a network or other distributed system, because the effect of the restart is not restricted to the system component which requires reconfiguring.)

Restarting should be unnecessary for any but the most serious changes in modern systems. We can design systems to make the changes comparatively easy by bearing in mind that it might be desirable at some time to change pretty well any parameter which must be set during configuration, and then designing the system accordingly. This in turn is easier than it used to be. Much of the rigidity of the older systems was a consequence of a configuration technique which built up the operating system as a single large programme running in a large flat memory. Once the system was built, each component had its own subroutine at some fixed position in memory, the address of which was embedded in the code at every reference to the component. Any structural change to such an edifice would probably be harder, and almost certainly more dangerous, than rebuilding it from scratch, so that's what they did. Since that era, operating systems have become steadily more modular, as is seen by developments such as object-oriented systems and the trend towards microkernels. Different components are brought into memory and used as required, in much the same way as any other programmes, so the purely geographical problems have disappeared. We can compare the two approaches :

THE OLD WAY :	A BETTER WAY :
Stop the system;	Shut down the component affected;
	Don't accept any more work;
	When current work is finished, adjust system tables, queues, etc.
Make the change;	Make the change;
Restart the system.	Bring back the component affected.
	Adjust system tables, etc.;
	Make the new version available.

We must still find ways of managing the change. In so doing, it is helpful to bear in mind the obvious, but illuminating, principle that any change can be regarded as composed of two stages : withdrawing the initial state, and activating the final state. In some cases, of course, one or other of these stages will be null, but that doesn't change the principle. It is also immaterial – again, in principle, though practical consequences are likely – in which order the two stages are performed. We shall consider the stages one by one. We shall discuss only the case of a single computer, but similar principles apply to linked machines, where completely automatic operation is very important.

Withdrawing the old version : This is not quite as simple as it sounds because, ex hypothesi, we want to do it while the system is in full operation, so we must decide what to do about people who are using the part of the system affected by the change. There are two sorts of answer to this question, depending on the sort of change concerned. If the requirement is to replace an old version of some hardware object by a new one, then some sort of discontinuity might be inevitable (the two objects might require the use of the same socket); if, on the other hand, the change involves only software, or a parameter of some operation, then continuity usually can be preserved, and should be preserved.

We first consider the discontinuous case. In some cases – if a component is demonstrably faulty, for example – it might be justifiable simply to stop all relevant operations, but such events are unusual. More commonly, the old version works well enough, and new version is simply different in some way. A common expedient is to publicise the change, and warn people that if they use the component during the critical time, then they're asking for trouble, but this could be seen as detracting from the user-friendly image which we always try to project.

What we really want to do, and what we *can* do with careful design, is to allow people to continue to use the old version if they have already started, but to switch new requests to the new version. Withdrawal is now a matter of monitoring usage of the old version, and reporting as appropriate when all current users have completed their tasks, at which point it can safely be unplugged, switched off, deleted, or whatever other variant of the terminating ceremony is appropriate. Meanwhile, new work must be queued awaiting the completion of the changeover.

This is likely to be possible if all software involved runs as one or more independent processes, communicating with other parts of the operating system by well defined paths and without using identifiers which are rigidly associated with specific devices or memory locations – so, for example, it is better to use a device type identifier rather than a device table index until a specific device is firmly identified and allocated. Provided that any such transactions required are clearly identified by process identifier, the possibility of confusion is small.

This approach can also be used for such tasks as clearing out material from discs which are to be maintained, or even releasing processors which must be withdrawn from a multiprocessor system.

When a continuous change is required, a rather different approach is possible. Consider, for example, a change in allocation of disc areas, perhaps to reserve more spooling space over a weekend. It should be possible to manage most, if not all, such changes invisibly. Once the desired change is specified, the system can begin to move towards the target by transferring existing material which would be affected to a more appropriate position; then, when the operation is complete, formally effecting the desired change. In the same sort of way, in the case of hardware changes continuous service is simplified if the new and old versions can both be attached to the machine at once. In this case, the two versions can run in parallel for some time, and the old one can be withdrawn once no one is using it any more. Provided that devices and software are reached through tables under control of the system, and that there is no physical obstacle to connecting old and new devices to the computer at the same time, there should be no insoluble problem in implementing a smooth change of the sort described.

Of course, it can only do so if the requirement has been foreseen and appropriate code incorporated into managing software.

Installing the new version : Again, we can consider the two variants discussed in the previous section.

For the discontinuous case, the change is likely to require changes to system tables, such as the device table or file directory. As we shall require the older version to remain accessible for running processes for some time, it might be necessary to accommodate two very similar entries in the table simultaneously, perhaps even with the same name. Commonly, though, the old *name* will not be required any more, as it will only be used when searching for the item; once the new version is installed, all searches should lead to that version, so the old name can be removed. Consider, for example, replacing a device such as a printer. When a process wants to use a printer, it will search the device table for a device of the appropriate sort, but thereafter need not search again, as the process's file information block will contain some sort of explicit link to the correct device descriptor in the device table. If this cannot be arranged, then it might be necessary to augment various tables with an additional attribute which explicitly records the device's *being withdrawn* state.

For a continuous change, nothing so drastic is needed. The new state is attained once any necessary reorganisation – such as the movement of disc material mentioned above – is complete. This must be monitored so that some appropriate report may be made when the operation is complete, but no further action is needed.

MAKING IT HAPPEN.

We've discussed a number of issues connected with the replacement of a system component, and have commented that much depends on the provision in the rest of the system for appropriate operations. Perhaps it will help to clarify these implications if we now follow through an example with comments on how other system components must be manipulated to maintain the required service. The aim throughout is to ensure that transactions intended for the old version are satisfactorily completed, software changed to that needed to manage the new device, and communication with the system reestablished for normal running.

Closing down the old device : It must be possible to indicate in the device table that the device is temporarily unavailable, so that new attempts to open the device can be detected and handled by the system. What the system does in such a case depends on circumstances; possibilities include using an alternative equivalent device (if there is one), queuing the request until the changeover is complete (sensible for a batch system), engaging in dialogue with someone wishing to use the device interactively, and so on. Each of these in turn requires that appropriate software be included somewhere, initially in the **open( )** procedure for the device. Transactions already in progress can be completed; if the change isn't urgent,

processes which have begun to use the device may be permitted to continue until they close it.

Removing the old device : The device might require special closing-down operations; these would normally already be incorporated into the device's normal **close( )** procedure. Also, as well as physically unplugging the device and carting it away, it must be removed from the system tables. Its device table entry must be removed, and its interrupt handler (if any) must be unlinked from the interrupt handling system and deleted. Finally, the device driver (and interrupt handler) must be deleted. Both the device table and the interrupt handling mechanism must therefore be changeable without reconfiguring the system, and appropriate operator instructions must be provided to initiate these actions.

Installing the new device : This operation is the inverse of the previous one. The new interrupt handler and device driver must be loaded into the system, and linked into the input-output system as appropriate. Again, the device table and interrupt handling mechanism must be changeable, and the device installation procedure must be able to incorporate the required changes.

Making the new device available : If this device is to be used as a full replacement for the old one, its device table entry must appear to be identical to a process using the device. The nature of the device is typically recorded as some sort of device type field, set by operator instruction when the device is installed, and interrogated by the file system when a request to open a file using a specific device is received. With this field set, the device can be made available, and all should proceed properly. In a case where the new device is sufficiently different from the old to be identified by a different device type, more care might be necessary when managing the incoming requests for service; perhaps it will not be possible simply to queue the work for the new device. This decision will affect the course taken in the first step of this sequence.

The immediate implications are that there must be a modular system structure to make it possible, operator instructions to make it happen, and device-specific code to provide detailed instructions. In more detail, we can summarise the implications for the system architecture :

- There must be system instructions which are executable by the operator to remove a device from service and to install a new device. (A single instruction to suspend and replace a device would be possible, but its only "advantage" would be that the new device descriptor could be made to occupy the same device table slot as the old one, and this isn't particularly helpful unless you want to keep the device indices in the table permanently constant, which is generally not a very useful idea.) During the execution of this instruction, many system components will be affected, and each of these must be constructed to permit the necessary changes without affecting its performance.

- The device table must include an availability flag for the device which is interrogated before any stream connected with the device is opened. It must also include some indication that the device is currently in use. It must be possible to mark a device table slot as empty

- The device descriptor must be composed largely of pointers, for the procedures for the different types of device might be of different sizes, so cannot be preallocated specific memory areas.

- The device driver must include any special procedures needed to close down the device, and the system must be able to execute these as part of the closing-down sequence. (Perhaps each device will have a closing procedure in the device table.)

- The interrupt handler must be detachable from the interrupt management structures of the system, and the new interrupt handler must be insertable.