p—A Timesharing Operating System for Laboratory Automation

ANDERS LINDGÅRD
Chemistry Laboratory III, H. C. Ørsted Institute, København Universitet, Universitetsparken 5, DK-2100 København Ø, Denmark

SUMMARY

An operating system supporting laboratory automation and interactive use of a computer has been designed and implemented for the multiprogrammed RC4000 computer. The emphasis in the design has been to give the user interactive access to as many system resources as possible, and to facilitate writing of process control programs in a high-level language, i.e. ALGOL. The operating system is used to run control and data collection programs for a variety of experiments in physical chemistry. Laboratory automation programs can be started and removed dynamically, and may even be restarted automatically after a system failure. The operating system is a 1500 line ALGOL program. The first version was designed, implemented and debugged in about 4 person-months. The operating system has been running day and night for approximately 5 years without errors despite heavy use.

KEY WORDS Operating system Time sharing Laboratory automation Multiprogramming Real-time control On-line access

INTRODUCTION

The purpose of the operating system is to provide experimentalists with a flexible and convenient tool for the interactive development of laboratory automation programs. This will encourage experimentalists to use the computer for existing experiments and enable them to carry out experiments not otherwise feasible. With this aim in mind, this operating system has to be easy-to-learn, easy-to-use and yet powerful enough to handle most practical situations.

Rapid development of hardware and software for experiments requires an interactive mode of working. The experimentalists should have the possibility of using a terminal in the laboratory for checking the experiment and getting meaningful messages about the behaviour of the experiment and the program.

A computer equipped with a disc, lineprinter, floating point hardware, etc. is expensive even if the central processor is cheap. Expensive and fast peripheral units are needed for speedy program development, to hold data from experiments, and to analyse and refine data. It is of economic importance if these peripheral units can be shared between several users and experiments. In general such a computer will have a surplus of computing power, even if the number of experiments is fairly high. (This surplus can be conveniently utilized for ordinary data processing provided the computer configuration has the peripheral units normally found in a small computing centre and the computer has the necessary software.) Even large scale computations as found in quantum chemistry may benefit from such a computer system,[1, 2] using the normally surplus night and weekend time.

0038–0644/79/1209–0971$01.00 *Received 2 February 1978*

The system described in this paper gives users fully interactive facilities from typewriter terminals. It minimizes the load on the computer from laboratory automation programs. Processes running laboratory automation programs and other interactive programs are created and removed dynamically. Automatic start-up facilities are provided after system deadstart. The system runs on a 24 hours a day, 7 days a week basis.

Characteristics of the RC4000 computer

The RC4000 computer system at the H. C. Ørsted Institute was funded in 1970 to provide the departments of chemistry, physics and mathematics with ordinary computer service to enable experimentalists in chemistry to automate experiments.

The RC4000 configuration has all the peripheral units normally found in an edp-centre plus a fairly large number of peripheral units for control and data collection from experiments (see Table I).

Table I. Hardware configuration

Conventional hardware:
24-bit central processor. Instruction time 4 μs; 48-bit floating point
48K words primary store (magnetic core, 1·5 μs)
384K words drum (15 ms access)
9M words disc file (50 ms access)
1 line printer (670 lines/min, full ISO alphabet)
1 card reader (1,500) cards/min)
2 magnetic tape units (800 bpi, 36,000 words/s)
1 paper tape reader (2,000 characters/s)
1 paper tape punch (150 characters/s)
2 Calcomp plotters (200 steps/s, 300 steps/s)
3 storage displays (4,800 baud, 9,600 baud)
18 directly connected typewriter terminals and alphanumeric displays (110 baud–1,200 baud)
Manufacturer designed hardware for process control:
8 digital input units (24 bit)
7 digital output units (24 bit)
3 interrupt expanders (24 bit)
1 stepping motor controller (8 motors)
8 binary counters (12 bit)
2 D/A converters (12 bit)
1 A/D converter (12 bit, 48 channels, 4 gain ranges)
H. C. Ørsted Institute designed hardware:
4 programmable pulse generators (24 bit and 12 bit)
1 general purpose DMA input unit (24 bit)
1 controller for a Intel 8080 microcomputer
1 multimicrocomputer unit with 4 microcomputers (2 Motorola M6800, 1 Intel 8080, 1 Zilog Z80)
5 local microcomputer systems connected through the multimicrocomputer system (Motorola M6800)

It is a multiprogrammed computer where a program, called 'monitor',[3, 4] provides (i) short-term scheduling of processes, (ii) a set of procedures for controlling processes and (iii) a set of procedures for communication between processes using message buffers. For technical reasons the file system and the input–output procedures are part of the monitor. Input–output procedures and their private variables are called 'peripheral processes'. This environment is not regarded as an operating system, but rather as a software extension of the hardware, which makes it possible to create operating systems in an orderly manner. An 'operating system' is defined here as a program capable of initiating and controlling new processes on demand and allocating resources between processes. The created processes are called 'child processes', and the operating system is the 'parent process'. A process has to be a contiguous fixed area in primary store throughout its lifetime as there are no hardware base or relocation registers. In this system it is possible to have a number of operating systems with different strategies active simultaneously, and even to replace an operating system dynamically by another without interfering with other activities in the computer. The monitor does not distinguish between operating systems and other programs. An operating system can be written as any ordinary program in any of the available programming languages, the only difference being the resources it has allocated and the unusual task it has to perform. The monitor is a monitor in the sense of Hoare.[5, 6]

The interface between compilers, user programs, other files and the users is handled by a job control language interpreter, called the 'file processor'.[7] This is executed as a program in the user area. An important feature of the file processor is that it does not matter which device (process) supplies the commands (input), nor which process has to receive the output, that is the interface is standard.

Characteristics of the experiments

Large groups of experiments in physical chemistry are slow compared to a modern computer's instruction execution time. Typical response time requirements which an experiment has to a computer is in the range from seconds to minutes. It is even common that the experiment can wait until the computer is ready, thus having no response time requirements at all. Experiments which do have significant response time requirements to a computer generally only have these for a relatively short time, e.g. less than 10 s. Examples can be found in chemical kinetics where a transient signal is measured or in modern thermodynamics where stochastic signals are measured. The number of data points required in a single experiment is typically less than 1,000.

In computer systems for a factory or where a large number of experimental set-ups of the same kind[8] have to be serviced by one computer, the usual strategy is to collect data from every data source periodically (polling technique). The experiments controlled at the H. C. Ørsted Institute have fairly low duty cycles and many will only run for a relatively short time, e.g. an hour, while others will run day and night but will not need much service, except for a few short time periods. An event-driven system, where the experiments signal when they need service, is more appropriate in an environment where the experiments differ much in behaviour and are completely independent of each other.

Interactive use of a computer

The experimentalist naturally prefers a turnaround time for the collection, refinement and analysis of data which is only slightly longer than the time for doing the data collection itself. On-line interaction with the experiment is considered valuable and is used to change set points or to display results.

Experiments are rarely static in a research environment. Experimentalists have a need to modify the experiment itself or to add new equipment or to change program strategy. This requires changes in programs and debugging of the changed programs and experiment simultaneously. This debugging task is more complicated than for ordinary programs as it is often difficult to determine whether an error is in the program part or in the experimental hardware. An extremely valuable tool with undebugged experimental equipment is the ability to monitor the program controlling an experiment from another program running from the same terminal.

REVIEW OF SOME OPERATING SYSTEMS FOR LABORATORY AUTOMATION

The three operating systems discussed below handle a class of experiments equivalent to that found at the H. C. Ørsted Institute. They are implemented on a medium-size multiprogrammed computer.

Concurrently with the operating system described in this paper, the manufacturer was developing an operating system BOSS2 for the RC4000[9] which may be called a remote batch system. It is basically a batch operating system having two active user programs in primary store at the same time. Furthermore, it gives a restricted time-sharing access from terminals as in the Cambridge multiple access system[10] although a full on-line access is possible for one single user, having the program occupying one of the primary store areas permanently while being on-line. Swopping of process areas in primary store is used to pre-empt programs and makes it possible to guarantee reasonable turnaround times for batch jobs.[11]

BOSS2 has been used for process control of experiments at the University of Århus, Department of Chemistry, but very little information about this part of the system is found in the literature.[12] In BOSS2[11] all process control peripheral processes are simulated and a message to any of these causes a swop. BOSS2 provides the necessary buffer space, if any is needed, while the input–output operation takes place. This is not a fair strategy against a process control job which may need to handle a number of peripheral processes in parallel. However, there is no direct way an operating system can be activated when a child starts to wait.

The interface for the experiments has to be rather extensive in order to synchronize experiment and user programs[13] and can be characterized as small special purpose computers. Some experiments have to use a 'core-lock' facility preventing swopping until a 'core-open' message is sent. The use of the 'one-line' and 'core-lock' facilities degrades the performance of the computer drastically. 'Core-lock' has to be used even in cases where it ought not to be necessary.

In the NBS system[14] data collection programs are part of a foreground area of primary store. Programs are initiated and may be removed by pushbutton signals from the experimental interface in the laboratory. All program input comes from thumbwheels in the laboratory interface, but output may come on the terminal in the laboratory. The data collection programs are resident in primary store until the program itself executes a swop request to the operating system or is terminated. The foreground jobs are not protected against each other and neither is the operating system, thus they must be error free. The background area of primary store is used for batch jobs which may be non-debugged data collection programs. The foreground area is protected against malfunctioning programs in the background area. The system serves a number of experiments with fairly low data rates and some giving bursts of data at a high rate. The system lacks inter-user protection, and there is no real possibility for using a terminal interactively. The foreground jobs will reside in store even if their duty cycle is low. The data collection programs can for space reasons only perform very simple data handling on the collected data.

The ARGOS operating system[15] is another example employing a much more powerful hardware configuration. Data collection programs are small assembly language programs executed as part of a foreground area of primary store. They are resident for long periods of time. Usually 10 programs of an average size of 1·5 K, 32-bit words are active at a time. Each data collection program has a unique priority for interrupt handling. The system gives a restricted interactive support allowing users to call precompiled programs from terminals. Batch service is supported using the card reader as the only input-medium and even long-term computations are now running on this computer.[1]

DESIGN OF THE OPERATING SYSTEM

The elements which have to be considered are the peripheral processes used for data collection, the operating system itself and the strategy for handling the terminals.

Peripheral processes for data collection and control

The problems of data collection for certain timescales and the problem of time jitter in the collected data are handled by the fairly general data collection peripheral processes (input–output procedures). The peripheral process concept is used to make all process control devices completely independent and they contribute to the users' illusion that no other activities take place in the computer. No specific assumptions about any of the connected experimental setups have been used in the design of the peripheral processes.[16] They appear as solutions to a fairly general class of data collection problems given the hardware characteristics of the peripheral units and the characteristics of the monitor. Data collection programs cannot execute input/output instructions directly, but send messages to peripheral processes.

The operating system and programming language

The primary design goal for the operating system described here is to achieve possibilities to use the computer in a fully interactive mode from terminals, close to what can be obtained on a monoprogrammed computer with many peripherals, i.e. the user should have access to all compilers, file handling programs, the editor and precompiled programs. All types of peripherals should be interactively available, e.g. line printer, plotters, magnetic tape and process control devices.

With the rather small store of 10 K, 24-bit words available for a process control operating system and the processes it is going to control (see Figure 1), swopping of processes between primary store and the drum appears to be a good solution. Figure 1 shows the store layout of p. The file processor and its input/output buffers (in, out) are only included when a test output from p is needed. Segments of p contain the code organized as segments of 256 words. Whenever a new segment is needed from the backing store the least recently used segment is overwritten. Ten segments are enough to hold the central loop, so that only commands to p, etc. will cause segments to be transferred into primary store. This gives a worse response time for an input, compute, output sequence by a user program than having all programs in primary store at the same time. The larger process size which can be obtained by swopping allows the user to run the editor, all compilers and most of the other programs interactively from a terminal. Swopping gives cheap interuser protection between the processes controlled by the operating system, otherwise it would have been necessary for some of the data collection programs to be error free, as the hardware protection system of the RC4000[17] is rather crude.

Division of space in primary store instead of time multiplexing would have forced us to write all data collection programs in assembly language, as in the ARGOS system. This approach was tried using the primitive operating systems,[3, 4] but the time investment in developing and especially in debugging the assembly language programs was excessive. The

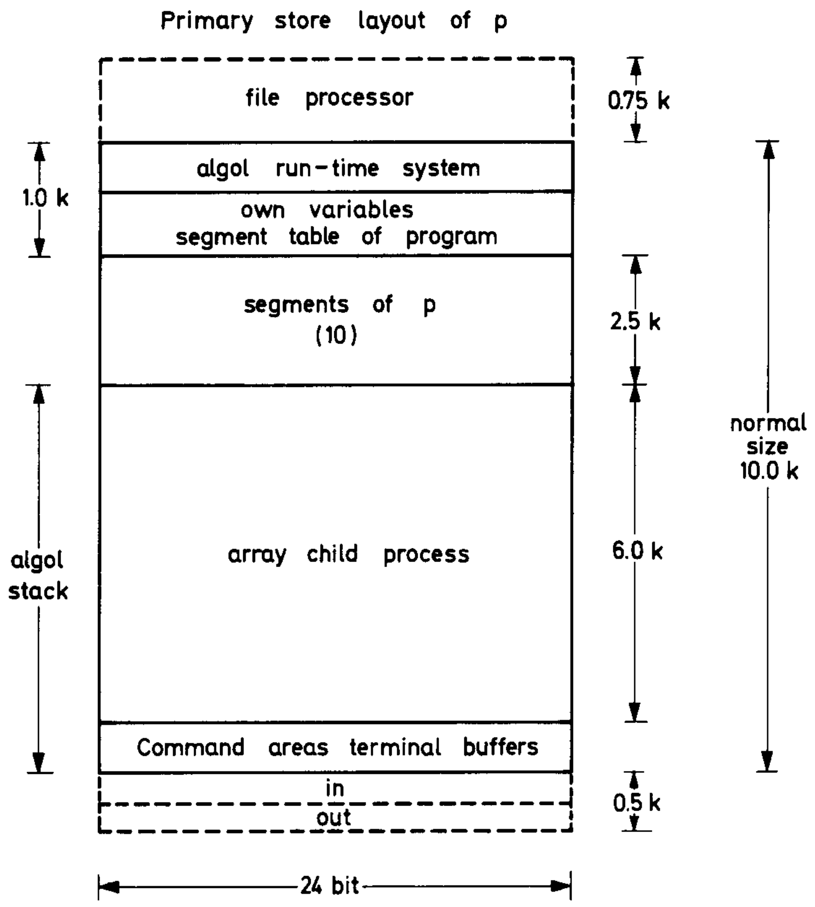

Figure 1. Store layout of p

editor cannot run in a small process area nor can the assembler. So either we had to use the batch system for debugging or we had to get a larger process from the operating system s when debugging a program.

The minimum process size to run the editor, the compilers, user ALGOL[18] programs with around 1K real variables (48 bit) is 6K words, leaving 4K words to the operating system code and variables out of the available 10K words. Four K words is sufficient to run translated ALGOL programs having a fair number of variables, provided the job control language interpreter has been removed (see figure 2); thus it was possible to get enough space in primary store so that the operating system could be written in ALGOL. Figure 2 shows the layout of the primary store of the RC4000 computer and a sketch of how p uses the drum. The batch operating system has only 13K words in normal working hours, but outside these the time-sharing operating system t is removed and this gives the batch-operating system 21K words. The file on the drum used for swopping has the name pdumparea. The code for p is placed on another file prun and segments of code are transferred from prun to the primary store area for p on demand. In this machine ALGOL programs have software virtual store for the generated code, organized as segments on the backing store, while the stack, the own variables and run-time system are store resident.[19] So an ALGOL program may run in a few K words of store, practically independent of the number of statements written. The software virtual store for ALGOL programs is organized as pieces of re-entrant code, called segments. Thus there is no need ever to read a segment back to the backing store. The behaviour of the ALGOL compiler and the run-time system were very

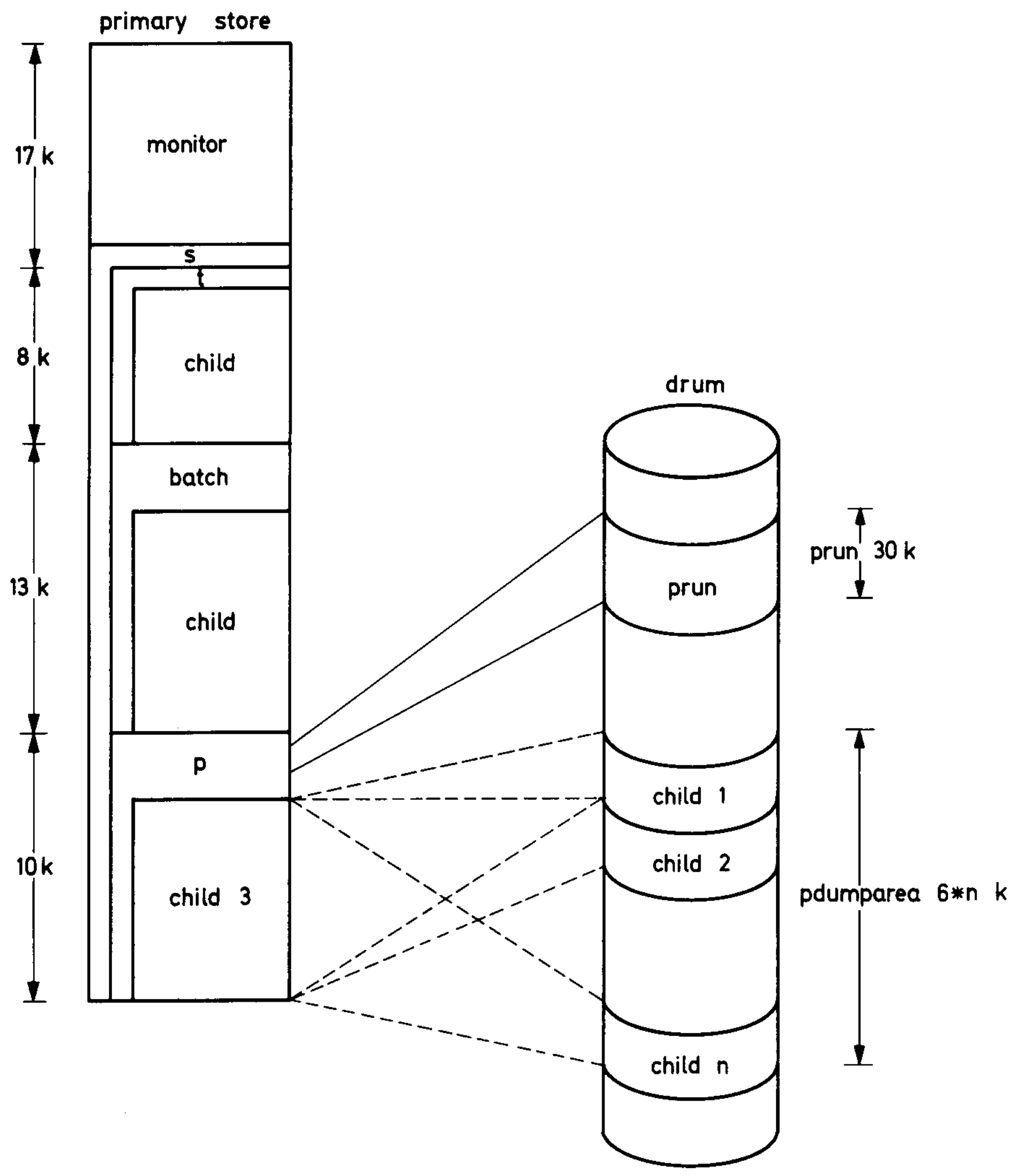

Figure. 2. Layout of the primary store of the RC4000 computer and a sketch of how p uses the drum

important design factors for this operating system. The FORTRAN compiler behaves in the same way as the ALGOL compiler, but nearly all programming is done in ALGOL. A minimum of code corresponding to only 20–40 ALGOL lines needs to be in primary store. This allows us to use most of the process area for variables. As laboratory automation programs typically make light use of the central processor, while often requiring many variables, this approach is quite good.

The primary reason for selecting ALGOL as the language for the operating system was to shorten the implementation time and facilitate maintenance of the system. At the time of design, ALGOL had been used by the manufacturer to write a batch-operating system.The standard facilities for communication and calling the monitor in ALGOL required excessive array handling for setting up parameters. This operating system was later replaced by the more extensive and sophisticated batch-operating system BOSS2.[9, 11] We developed a set of procedures for calling the monitor which are much simpler and a set of procedures to make assembly language features available with ALGOL (see Appendix), i.e. procedures using physical addresses. These procedures are inefficient in the sense that it may take 50 instructions to execute what is equivalent to a single machine instruction. As an operating system will only use a small fraction of the available central processor time in any case, this overhead is only of marginal importance and is far outweighed by the advantages in system development.

Scheduling of processes

In foregound–background scheduling,[20] background jobs will only get central processor time if no foregound jobs are active. The foreground jobs may even have priorities, causing a lower priority job to be pre-empted if a higher priority job needs service. We doubt whether complex priority schemes are really useful in a time-sharing environment, unless it is required to press timing requirements for interrupt-driven data collection for a single job to the limit. A large number of interrupts will dramatically reduce the usefulness of a time-sharing system, as the terminal users cannot get a reasonable response. In the RC4000 the system degrades dramatically when a 2 kHz signal is sent into even the lowest priority interrupt bit. This implies to us that the number of interrupts should be kept at a low level, as our expensive computer should not waste much time on trivial interrupt handling, and that priorities should not be used to solve the scheduling of time in primary store between user processes. An ordering of requests for doing a number of operations within a time interval should use semaphore operations, where the critical phases are complete time intervals. Priorities only determine which operation shall fail, as priorities cannot resolve a conflict between two equal priority jobs trying to be active at the same time.

Operating system p uses a modified round-robin scheduling for the active processes; p performs only scheduling of time in primary store (medium-term scheduling). The short-term scheduling of the central processor is performed by monitor on a round-robin basis. The normal maximum time period in primary store between roll in and roll out is 5 s. This time period can be changed dynamically by a command to p. The period may be shorter if the process starts to wait for an event. In normal working hours users should be punished for running programs which use the central processor heavily under operating system p, as this degrades the response time for other users. Whenever p detects that a process has used more than a certain fraction of the available central processor time within its 5 s in primary store, it is removed from the queue of active processes for a certain time interval. This time interval can be changed by a command to p.

Terminal input/output

The hardware for controlling the terminals uses the instruction-controlled data channel only. As an input/output machine instruction can only transfer one single character every character generates an interrupt. This implies that a process can only communicate directly with a terminal when the process is in primary store to provide the necessary buffer space. To smooth the terminal input/output, p has a buffer for terminal input/output, for every possibly active process. For every block of characters transferred from a process to a terminal or *vice versa* two messages are sent and waited for. The first one from the child process to p, and then from p to the terminal process; p has to copy the information between its terminal buffer and the terminal buffer inside the child process. This is a time consuming and inelegant way of handling terminal input/output, forced upon us by the primitivity of the hardware. At present we are building a terminal multiplexer as part of a multi-microcomputer system.[21] Its primary purpose is to relieve the central processor of the RC4000 computer from trivial interrupt handling and to avoid having terminal buffers in the operating systems.

Optimization of the use of primary store

An operating system which uses swopping for store multiplexing has to be very concerned about the behaviour of the child processes. A child process should be swopped whenever it starts to wait for some event, which is expected to occur later than the time for rolling a child process out and a new child process in. An operating system in the RC4000 system performs a swop by first attempting to stop the child process. A child is at a complete stop when all pending data transfers using the direct memory access channel (DMA) are finished. Low-speed data transfers using the instruction-controlled data channel cannot delay a stop. The rolling out is performed as just another transfer of data from internal store to a file. The next process is rolled in and started. The complete operation will, using the drum as swop medium, take about 400 ms. The program's use of peripheral units like discs or drums ought not to cause a swop even if there is a queue of requests for access. Input from terminals is the opposite case, as one will always expect this to take a long time. Output from a process to a terminal should not necessarily cause a swop. If the terminal buffer contains 50 characters a 1,200 baud terminal will output these in time for a swop. Waiting for a process control device must produce a swop as the waiting time must be considered as undefined. Waiting for an answer to a message sent to the internal clock, which may be done with a time resolution worse than 100 ms, is a case for swopping.

Any peripheral process may be simulated by the operation system which makes it easy for the operating system to check when a message is sent to a peripheral process. This is done in p for terminals only.

A child process in the RC4000 system has a process description as part of the monitor, which contains a head of the queue of message buffers, a working register dump, the state of the process and the instruction counter. p checks every second the state of a running child process. If it is waiting, the last instruction executed and the content of the dumped working registers gives all the information needed about the event waited for. If the event has not yet arrived and may be expected to last a long time this child process is swopped; p keeps the information about the event, and whenever this child process is a candidate for rolling in, the message buffer queue of this child process is examined. If an event which will activate this child process has arrived it is rolled in again and participates in the normal schedule, otherwise the next process is examined. This regular inspection of the child process is a form of busy wait, but it is very cheap in resources, as other programs running under the two other operating systems may use the central processor. It further makes it possible for a child process to use low-speed devices directly without an unreasonable degradation in performance, e.g. the 2,000 character/s paper tape reader. This saves buffer space in the operating system, which otherwise should have simulated the device.

Structure of the ALGOL text for p

Operating system p is organized in three parts: (1) Initialization code, which checks that the process is created correctly and creates and initializes the arrays used for controlling the child processes, the terminal buffers and the array where the child process is actually executed. A message is sent to all terminals giving the time of start up. The processes described in an initialization file are created. (2) A number of procedures used rather infrequently, i.e. the interpreter for commands from terminals, the procedure for process removal, the interpreter for certain messages from child processes called parent messages, (3) The central loop, which is a set of coroutines taking care of swopping and of terminal input/output. There is a single waiting point where it is determined whether the event can be taken care of by the central loop or some of the slower procedures have to be invoked.

Features of the operating system p

A session using p for a simple task is shown in Figure 3. Other examples may be found elsewhere.[22, 23] Figure 3 shows an example of job session using p for writing a simple interactive program and executing this. The : in the middle of the figure and what is to the left are later inserted comments. The log-in scheme is very simple. A BELL character will

```
att p                                   : getting in contact with p
pass ali                                : writing a command to p (log in)
                                        :
ali started  21 12 76  13 58 28         : a process is started and the
                                        : file processor (FP) loaded
to ali                                  : written by the terminal process
r = algol                               : FP reads a command to
                                        : execute ALGOL compiler
begin                                   : compiler starts reading
real s;                                 :
s: = reader(<:value:>);                 :
write (out, <:sinus = :>, sin(s));      :
end                                     : the final end terminates input
algol end 26                            : translation OK
r                                       : FP reads a command to execute r
value = 0.2                             : program writes value = and user 0.2
sinus = 0.1987                          : execution of write statement
end 17                                  : termination of ALGOL program
finis                                   : FP reads a command to execute finis
                                        : which send a message to p (log-out)
from p                                  :
ali removed  21 12 76  14 00 14         : message from p; process is removed
Run time 1.25 s.                        : central processor time used
```

Figure 3. Example of job session using p

activate the computer and att is written. The user may get in contact with any operating system or other process by typing the name of the process. The user then types pass to p and the password. Following to ali all input is to the child process ali where a job control language interpreter is running; r = algol loads the ALGOL compiler, which will produce the binary output in the working file r, and take input from the terminal. The characters from begin to end are read by the ALGOL compiler. In the line algol end 26, the compiler signals compilation OK and transfers control back to the file processor; r is executed by writing its name. The program outputs value = and reads in a real value which is assigned to s in the program. The write statement is executed and the program terminates writing end 17; finis sends a log-out message to p which then writes the last two lines.

The operating system supports a number of experiments as well as normal program development. A process control user will be assigned resources after a log-in and will loose the resources, except for permanent files at a log-out. The only difference between a computational user and a process control user is that the latter has access to process control peripheral processes. The dynamic allocation of resources is in general advantageous, but a user may run into the problem that it is impossible to obtain a process as the total number of processes is rather limited.

The plotting system[24] has been designed to be used interactively under the two timesharing operating systems as well as under the batch system. It makes it easy even for very inexperienced persons to control the plotter.[23]

It is easy from a process to create jobfiles for the batch-operating system and to submit these. It is further possible through the batch-operating system to get files on the backing store printed on the high-speed line printer.

Any computer system may crash due to failures in the hardware. Development of basic software and of new peripheral units sometimes requires an upstart of the software system from scratch. p has an initialization file where commands may be inserted, which has the same effect as creating a process from a terminal. The process created will read normal job control language statements from a file, making it possible to execute automatically any sequence of programs.

The terminal is a limited resource, though this computer has more terminals than possible active processes. From a terminal one can have a number of processes active at the same time. This is advantageous in program debugging or when tracking down errors in the experimental hardware. In one process the usual process control program or a test program may be executed, and from the other process connected to the same terminal, it is possible to examine queues, states of peripheral processes, etc. A monitor modification makes it possible for the user to know to which process input is made and from which process output comes, though all input/output to terminals is performed by the operating system.

Terminals may break down, and this should not harm a process. It is possible to change the terminal of a process to another terminal by a command to p. A process may not even need a terminal at all, but just a file on backing store for input and another for output, and this is supported by the operating system.

It is possible for a child process to create a new parallel child process. This is useful when using the plotting system for making drawings which take a long time. It is not really possible, due to the structure of the plotting system, to run a process control program which performs a number of parallel actions and plots at the same time.

Facilities for process control

Most of the specific facilities for process control are not part of the operating system, but have been built into the peripheral processes, which are small pieces of re-entrant code, about 50 instructions for each type, connected to a process description of typically 15 words per process. Using peripheral processes, rather than the operating system, has the advantage that the former may be used under any of the operating systems.

A process exists which can prevent any operating system from stopping a child process for a period of 15 s. This core-lock feature makes it possible to do interrupt-driven data collection, using the child-process own store area as the buffer area, provided it can be performed within the 15 s. To prevent repeated locking, it is impossible to use the core-lock process for 15 s following locking. Computational programs cannot utilize the core-lock mechanism, as a child process cannot use the central processor after initiation of a stop.

Most of the peripheral processes are capable of handling information flow, where all the information is stored in the small message buffers. Message buffers are part of the monitor and are thus never swopped. Message buffers are queued by monitor to processes when used. Thus multibuffering is immediately available. As an example, 15 message buffers, each capable of storing two data words and the associated times for data collection, can keep pace with a signal requesting data collection every second. The associated time values, which are measured indivisible with the data point, make it easy to check that data collection was performed within the required time limits.

DEVELOPMENT OF THE OPERATING SYSTEM

The operating system was fully operative and error free on 15 September 1972. The time invested was around 4 person-months, of which 1 month was for development of the small assembly language coded procedures and minor changes in the monitor. The small time invested in developing the operating system may be attributed to several factors: (1) Only minor changes had to be performed in the monitor, file processor, etc. due to their modularity, and a clean interface to possible operating systems. (2) No accounting facilities or scheduling of peripheral devices had to be part of the operating system. Laboratory automation programs will only use the peripheral processes they need, and no mistakes have yet taken place as peripheral processes have names and not numbers. (3) Information secrecy is not of concern at a university. (4) The system was written in a high-level language, which includes (low-level language features (physical addresses in primary store). To add features at the machine language level to a high-level language of course partially spoils the idea of using a high-level language. It was, however, necessary due to the primitive ALGOL data structures and due to the fact that some of the information needed could only be known outside the ALGOL level, i.e. by reading the monitor data structures and reading information in the controlled process. Moreoever, it is much easier to write an operating system in a language having both kinds of features than in an assembly language. We do not have to bother about registers or how far relative addresses can span. We do keep an extensive syntax check on the major part of the program. At run-time it is possible to use an index check on array indices. It is easy to insert test output in the program. All these high-level language features were extremely valuable in the debugging phase of the operating system. A high-level language approach makes it much easier later to add new features to the operating system and to examine whether strategy changes will affect the performance. We would of course have preferred to write the operating system in a more appropriate high-level language like Concurrent Pascal[25] or Module,[26] but these were not available at the time.

There have been some later minor additions of features to the system and the central loop has been assembly language coded. This has not caused any essential change in characteristics or performance. The system is practically error free. The very few errors seen could just as well be attributed to our hardware development effort of new peripherals.

PERFORMANCE

The operating system uses about 4 per cent of the available central processor time. Rewriting the central loop in assembly language did not change this figure. There is an immediate response to simple requests, when few users are actively using the operating system editing, compiling small programs and running small programs. At higher loads there are normally also high loads on the other operating systems, which gives a queueing of requests to the disc. It has not been possible to detect queues on the drum due to swopping. The queues on the drum come from the job control language interpreter which uses a segmentation scheme similar to a running ALGOL program for its lengthy routines.

An input buffer to a terminal is one line or less than 76 characters. An output buffer is less than 76 characters. The number of output buffers from a user process is approximately 100 times larger than the number of input buffers, as measured on the RC4000. Users do use their terminals for listing of smaller programs (1–2 pages) and for output from running programs, and this forces the monitor to spend a rather large amount of time in the terminal peripheral process.

Use of the central processor, the drum and the disc has been monitored by examining queue lengths using a small program permanently in primary store. The pattern seen is highly irregular and seems to depend rather critically on the job mix. The job mix varies so much during the normal working hours that we see the system being bound on any of the three resources monitored, but hardly ever on more than a single one of the resources at a time. With three operating systems active, one batch and two time sharing swopping the load are characteristically about 90 per cent on the central processor, when all systems are fully occupied. We consider 90 per cent very satisfactory and do not believe it is worth trying to improve the system software to get more throughout. The only realistic way to improve matters considerably for the users is to develop a better terminal control peripheral unit and to purchase more backing store devices and a faster central processor.

DISCUSSION

Time-sharing systems described in the literature seem to avoid direct input to running programs[27–29] and to optimize the use of the central processor. In our system, where there is a parallel batch system and another time-sharing operating system, it is to be expected that the central processor time which cannot be used by a child process in one of the operating systems will be used by the child processes in the other. These assumptions seem justified as we see idle times for the central processor of less than 10 per cent during the high-use period.

The two-level segmentation or paging scheme introduced here is a little unusual. The ALGOL programs, the compiler and the job control language interpreter all use a software demand paging scheme internally, and the operating system uses swopping to pre-empt processes running these programs. However, it restricts thrashing problems compared with a computer having simple hardware paging. An individual program may in our system experience thrashing internally, by consecutively loading the ALGOL segments of a too large loop into primary store. This will not affect other programs very much, neither the ones running under the same operating system nor under the other. It only increases the queue lengths to the disc moderately. If several programs thrash simultaneously, each under a different operating system, the users will experience a slow computer. This is not likely to happen with a number of completely independent operating systems. Contrary to hardware paging systems, thrashing basically only depends on the user's own programs, and not on the system as a whole. What they have to do is either minimize the amount of code in the critical loop or get a larger primary store area. Running a program under the batch system nearly doubles the primary store compared to the time-sharing systems. If the user is willing to run the program after working hours the store available nearly triples. From a monitoring process we have examined experimentally the thrashing problem. The results seem to indicate that the two time-sharing operating systems and their child processes can never cause thrashing to a degree where the performance of the total system is drastically reduced. Batch jobs, which in our system are never pre-empted, may cause trouble if they access the disc causing a large number of head movements.

The idea of minimizing the load on the computer from process control programs may seem odd for a process control operating system. However, other users gain, including other process control users. Further, and not least important, it makes it economically feasible to automate experiments which otherwise would never have been considered.

The operating system described here and the peripheral processes for data collection cannot solve response time and synchronization problems to a degree which may be obtained with a monoprogrammed computer. Very fast data collection is, however, just as well performed by a multiprogrammed as a monoprogrammed computer. The connection of microcomputers as slave computers to a multiprogrammed computer, for use as user-programmable peripheral processes, can solve response time and synchronizing problems at a low cost.[23] This is done without losing the advantages of having a large multiprogrammed computer with its extensive facilities.

For users who do computations only, it is of course less attractive to have three operating systems sharing the primary store in fixed partitions than to have one operating system capable of giving the user nearly the whole store. Outside normal working hours, users of this RC4000 system will get more space in primary store by removal of the other time-sharing operating system. Programs with large primary store requirements tend to have long run times, so the policy is rather fair. Even the busy–wait strategy for examination of whether child processes are waiting or not does not seem to harm the performance at all.

Not only do experimental set-ups benefit from a timesharing operating system such as p. A minicomputer is used for long-term computations in statistical mechanics.[30] An ALGOL program running under p acts as an operating system for this slave computer. The automatic start-up facilities in p make it possible to run the long-term computations without break-downs, independent of breakdowns of the RC4000.

Comparison with BOSS2

BOSS2 is a much better batch-operating system than the one we are using at present. BOSS2, however, needs to be the only operating system in the computer mainly due to the hierarchical administration of the backing store firmly implemented as part of BOSS2. It is less attractive for this reason to an installation where the primary reason for having a computer is process control of independent experiments. Operating system p has its force in process control over BOSS2 for the following reasons: (1) On-line use with direct terminal input–output to the user process is well supported and does not degrade other uses of the computer, e.g. batch. (2) Multiple buffering and use of several peripherals in a parallel fashion is supported, allowing us to use very simple process control peripheral hardware and simple general input–output driver programs for these peripherals. (3) A time-limited 'core-lock' mechanism gives the possibility of using the primary store of the calling process as a buffer store for semi-fast input–output. (4) Other operating systems may run in parallel as handling of the backing store resources is done by the monitor instead of the operating systems. (5) Automatic start-up facilities for process control and other programs after breakdown.

Transfer of p to other computers

Is it possible to transfer a system like p to another computer with different software ? This must be answered with a no unless the monitor is transferred too; p has heavy dependence on the general input–output scheme, and the control mechanisms of the monitor. The implementation in ALGOL really only depends on the possibility of including small machine-coded procedures for address handling (see Appendix) and on the possibility of direct communication with monitor within the language.

ACKNOWLEDGEMENTS

The operating system p was designed and implemented by K. Ludvigsen, P. Graae Sørensen and the author. H. Bjerregaard later rewrote the central loop in assembly language. R. Moss and H. T. Wheeler very carefully read this paper and made it much more readable.

APPENDIX

Assembly language features of the ALGOL system;

integer procedure first_address (var);
comment finds the address of a simple variable or the first address of an array independent of its type;

procedure redef-array (a, address, bytes);
comment defines the declared array to be placed in store from address and forward. The store for array a in the stack is inaccessible.
Eventual index check of bounds is still performed;

integer procedure wordload (address);
comment delivers the content of the cell in store having address as physical address;

procedure wordstore (address, content);
comment stores content in the cell in store having address as physical address;

These few procedures make it possible to access all data structures within the running program and monitor in a primitive way. The procedure redefarray is very useful for accessing data structures in the monitor, e.g. a process description without having to copy the information. In some computers it is possible to access half-words, double-words with a single machine instruction, and it may be convenient to implement the corresponding procedures.

REFERENCES

1. A. F. Wagner, P. Day, A. C. Vanbuskirk and A. C. Wahl, 'Minicomputers and large-scale computations' *ACS Symposium Series* (Ed. P. Lykos), **57**, 200–217, 1977.
2. J. M. Norbeck and P. R. Certain, *ACS Symposium Series* (Ed. P. Lykos), **57**, 191–199, 1977.
3. P. Brinch Hansen, *Operating Systems Principles*, Prentice-Hall, Englewood Cliffs, 1973.
4. P. Brinch Hansen, *Commun. ACM*, **13**, 238–243 (1970).
5. C. A. R. Hoare, *Operating Systems Techniques* (Ed. C. A. R. Hoare and R. H. Perrott), Academic Press, New York, 1972.
6. E. W. Dijkstra, *Acta Informatica*, **1**, 115–138 (1971).
7. S. Lauesen, *RC 4000 Software, File Processor*, RCSL 55-D21, Regnecentralen, Copenhagen, 1969.
8. E. Ziegler, P. Henneberg and G. Schamburg, *Anal. Chem.* **42**, 51A–56A (1970).
9. S. Lauesen, *Commun. ACM*, **18**, 377–389 (1975).
10. M. V. Wilkes, *Software—Practice and Experience*, **3**, 323–332 (1973).
11. S. Lauesen, *Acta Informatica*, **2**, 1–11 (1973).
12. H. J. Skov, L. Kryger and D. Jagner, *Anal. Chem.* **48**, 933–937 (1976).
13. A. Lindgard and J. Oxenboll, *Anvendelse af BOSS2 til processkontrol p-å*, H. C. Ørsted Institutet, 1974 (unpublished) (in Danish).
14. J. R. Devoe, R. W. Shideler, F. C. Ruegg, J. P. Aronson and P. S. Schoenfeld, *Anal. Chem.* **46**, 509–520 (1974).
15. P. Day and J. Hines. *Operating Systems Review*, **7**, 28–37 (1970).
16. P. Graae Sorensen and A. Lindgard, *Computers in Chemical Research and Education*, (Ed. D. Hadzi), **III**, 5/39–5/58, Elsevier, Amsterdam, 1973.

17. P. Brinch Hansen, *BIT*, **7**, 191–199 (1967).
18. S. Lauesen, *RC4000 Software, Algol 5*, RCSL 55–D42, Regnecentralen, Copenhagen, 1969.
19. P. Naur, *BIT*, **3**, 124–140 (1963).
20. A. P. Sayers, *Operating Systems Survey*, Auerbach, New York, 1971.
21. A. Lindgard, J. Oxenboll and H. Bjerregard (to be published).
22. A. Lindgard, R. Moss and J. Oxenboll, *Comp. and Chem.* **1**, 7–11 (1976).
23. A. Lindgard, P. Graae Sorensen and J. Oxenboll, *J. Phys. E*, **10**, 264–270 (1977).
24. A. Lindgard and R. Moss (to be published).
25. P. Brinch Hansen, *IEEE Trans. Software Engineering*, **1**, 199–207 (1975).
26. N. Wirth, *Software—Practice and Experience*, **7**, 3–84 (1977).
27. J. W. Meeker, N. R. Crandell, F. A. Dayton and G. Rose, *AFIPS Conf. Proc.* **34**, 241–248 (1969).
28. C. C. Foster. *Comput. Surv.* **3**, 23–48 (1971).
29. R. M. Needham, in *Operating Systems Techniques* (Ed. C. A. R. Hoare and R. H. Perrott), Academic Press, New York, 1972.
30. A. Lindgard, P. Graee Sorensen and J. Oxenboll, *Comput. and Chem.* **1**, 277–285 (1977).