A MULTIPROGRAMMING SYSTEM FOR EDUCATION AND RESEARCH

by

Alan C. Shaw

and

Nelson Weiderman

Technical Report

No. 70-81

November, 1970

Department of Computer Scienc
Cornell University
Ithaca, N. Y. 14850

ABSTRACT

A Multiprogramming System for Education and Research

by

Alan C. Shaw and Nelson Weiderman

Department of Computer Science

Cornell University

Ithaca, N. Y. 14850

U. S. A.

A novel multiprogramming case study which has proven useful in operating systems education and research is presented. The operating system and its unusual hypothetical machine are specified. The paper then describes experience with the case study as a class project involving hardware simulation, and systems design and implementation; and as a research tool for testing new ideas in operating systems primitives and design methodologies. It is concluded that case studies of this type will be widely used in the future.

A MULTIPROGRAMMING SYSTEM FOR EDUCATION AND RESEARCH*

by

Alan C. Shaw and Nelson Weiderman

1. INTRODUCTION

Research and development efforts in computer operating systems have produced a number of techniques and concepts that are both useful and intellectually interesting [3,11]. As a result, the subject is now firmly established as a legitimate area of study in an academic computer science program. However, there are several qualitatively different approaches to teaching operating systems. We can distinguish the following three approaches, all of which have the same educational objectives [1,2]:

(1) a detailed case study of one (or more) real systems, augmented by programming projects,

(2) a design and implementation of a real system, and

(3) a lecture course, covering general principles and techniques, augmented by one or more large programming projects.

Consider the characteristics of real systems. Most real systems are written in assembly language for machines with complex instruction sets and hardware characteristics; time and space efficiency is achieved

*This research was supported in part by the National Science Foundation, Grant GJ-108.

through many <u>ad</u> <u>hoc</u> tricks which make the code difficult to understand. Generally, there are few basic ideas relative to the size of the system and to the amount of labor required to comprehend or implement it. These reasons lead the first author, when teaching a course in fall 1968, to reject (1) and (2) above, as well as projects comprising extensions or subsystems for real systems in (3). Instead, a large project [10] was designed to augment a normal lecture course. The project involves the design and implementation of a multiprogramming operating system (MOS) for a specified hardware configuration that can be easily simulated.

Research in operating systems has been devoted for the most part to the development of practical concepts and tools (advances in "software engineering") as opposed to abstract theory. New ideas are traditionally tested by simulations or by applying them in a real system. This approach, at least for initial testing, suffers from the same problems mentioned above. The MOS project has also been valuable for rapid testing of new software ideas [12,13].

This paper describes the project and discusses its application in education and research. We believe that the novel type of systems abstraction exemplified by the MOS and machine will be a common educational and research tool in the future.

2. MACHINE SPECIFICATIONS

The virtual machine (VM), that is, the computer as viewed by a normal user, is a very primitive single address computer (Fig. 1). With the small instruction set, a batch of compute bound, input-output bound, and balanced programs may be quickly written.

The components of the real machine are illustrated in Figure 2. The CPU may operate in either master or slave mode. In master mode, instructions from supervisor storage are directly processed by the higher level language processor; in slave mode, this processor interprets a "microprogram" in read-only memory which simulates (emulates) the CPU of the VM and accesses VM programs in user storage. This organization allows the VM emulator and the MOS to be coded in a higher level language while maintaining some correspondence with real computers. Currently, the higher level language is BCPL/360 augmented with coroutine primitives [8,9,14].

User storage contains a number of ten-word storage blocks, each word identical to a VM word. User storage is large enough to accomodate several virtual memory spaces to permit multiprogramming of user jobs; a typical size is 30 blocks, but this number may be changed to determine the effects on MOS performance. Addressing occurs through dynamic relocation hardware; a page table register in the CPU points to a page table in user storage associated with the currently executing user process. This permits a VM program to be non-contiguously allocated on a block basis; we do not provide for demand paging. Supervisor storage is loosely defined as that amount of storage required for the MOS.

The auxiliary storage drum is organized in ten-word tracks and is used to store user jobs and output; it typically consists of 100 tracks, but it is again convenient to make this number a parameter. The card reader, printer, and drum are each connected to the system by separate channels which operate independently of each other and the CPU.

Execution times are specified for each of the VM instructions and input-output operations over the channels. In addition, there is an interval timer which may be interrogated and set in master mode. Time runs only while in slave mode with the exception that a master mode program may "wait" for a specified number of time units.

An interrupt mechanism is specified which detects four classes of interrupts: program, supervisor, input-output, and timer. Interrupts occur only at the end of VM instruction cycles in slave mode, but their presence may be tested in master mode. Program interrupts are generated by user storage protection violations and invalid operation codes; supervisor interrupts are caused by a VM read, write, or halt instruction; input-output interrupts result from the completion of a channel operation; and a timer interrupt occurs when the interval timer is decremented to zero. When an interrupt-causing event occurs, its description is stored in a register. The hardware performs the following actions when a slave mode program is interrupted:

(1) save the current slave mode state,

(2) switch to master mode, and

(3) transfer control to a fixed location in supervisor storage associated with the interrupt class.

3. THE OPERATING SYSTEM

The primary objective of the MOS is to efficiently process a batched stream of user jobs. This is accomplished by multiprogramming systems and user processes.

Each user job consists of a job header followed by a VM program and, possibly, data; the job header contains a job identifier, and time and line limits. A job J will pass sequentially through the following phases:

(1) Input spooling

J enters from the card reader and is transferred to the drum.

(2) Main processing

The program part of J is loaded from the drum into user storage. J is then ready to run. Until J terminates either normally or as a result of an error, its status will generally switch many times between:

(a) ready - waiting for the CPU to be allocated to it,

(b) running - executing on the CPU, and

(c) blocked - waiting for completion of an input-output request.

J's input-output requests are translated by the MOS into drum input-output operations.

(3) Output spooling

J's output, including charges, systems messages, and its original program, is printed from the drum.

In general, many jobs will simultaneously be in the main processing phase.

The operating system is documented and programmed as a set of cooperating sequential processes [4,5]. A typical design would have the following major processes:

ReadInCards:	Read cards into supervisor storage.
JobToDrum:	Create a job descriptor and transfer a job to the drum.
Loader:	Load a job into user sotrage.
GetPutData:	Process VM input-output instructions.
JobEnd:	Perform a job termination.
LinesFrom Drum:	Read output lines from drum into supervisor storage.
PrintLines:	Write output lines on the printer.

A process scheduler, invoked by interrupt handling routines and systems processes, is responsible for allocating the CPU to ready processes.

A major task of the MOS is the management of hardware and software resources; Table 1 contains the producer-consumer relationships for some of the resources with respect to the above processes. The MOS is also responsible for maintaining statistics on hardware resource utilization and job characteristics. BCPL code for a version of the Loader process is presented in Figure 3.

4. USE IN EDUCATION

This case study has been used at Cornell for three successive years as the main design and programming project in a one semester graduate computer science course on operating systems. The course is not a detailed study of a particular system but covers a variety of concepts and techniques with appropriate examples from real or proposed systems. The role of the project is to consolidate many of these notions and to provide some experience in their application.

Students are given a complete description of the hardware, and a general outline of the functions and main components of the MOS at about the level of detail of the last section. They are then required to

(1) write the "microprogram" emulator for the VM,

(2) write a simulator for the hardware (everything except the higher level language processor), and

(3) design and implement an MOS for the machine.

The project is handed out about halfway through the semester and is due near the end of the term - a period of approximately two months. Presently, students work in small groups of two or three, each group doing the complete project. The emulator, hardware simulator, and MOS consist of approximately 1000 cards of program with most (about 4/5) of the code representing the MOS. A batch stream of 60 jobs is used for testing and evaluation. At this time, 20 systems have been written. Enough experience has been gained to convince us of the usefulness of the project.

The educational value of the MOS lies in the almost realistic yet tractable setting within which students are forced to come to grips with the following issues, concepts, and techniques:

(1) the organization of large programs and projects,

(2) input-output control and buffering,

(3) synchronizing of processes,

(4) process scheduling,

(5) main and auxiliary storage management,

(6) deadlock problems,

(7) interrupt handling,

(8) data structrues for processes and resources, and

(9) higher level languages for systems programming.

As a result of this experience, and the course lectures and homework, students seem to feel quite comfortable in using, evaluating, modifying, and designing real systems.

There are certainly a number of limitations and abstractions from reality in such a project; some of these are mentioned in the concluding section. However, the limitations present no educational difficulties provided that they are carefully specified and that the project is used to complement a broader study of systems.

5. USE IN RESEARCH

The MOS has also been useful as a research tool for preliminary testing of new ideas. The applicable areas of research include data structures and primitive operations for process and resource management, design methodologies, higher level languages for describing and implementing systems, and performance monitoring.

In one investigation, the MOS has been extensively used to test the sufficiency and convenience of a single pair of primitives, designated REQUEST and RELEASE (Fig.3), and an associated set of data structures proposed for both process communication and resource allocation [13]. All processing required for these two activities is embedded within the primitives rather than distributed throughout the system; REQUEST and RELEASE are defined to be the <u>only</u> mechanisms that switch the status of processes between logically running and blocked. The proposed primitives are generalizations of others with which we are familiar, such as Dijkstra's P and V [4], Hansen's message communications operations [6], and IBM's ENQ, DEQ, WAIT, and POST [7].

A second area of research which uses the MOS as a convenient test case is in the development and evaluation of design methodologies. Dijkstra [5] has employed a hierarchical structure of sequential processes such that each level represents an abstract machine whose logical correctness can be verified. An iterative multi-level methodology was proposed by Zurcher and Randall [16] in which, at each level, the external behavior (input-output relationship) of a subsystem is systematically specified before the design of its internal logic at more detailed levels; each level is evaluated and refined before another level is added. The same authors suggest that executable versions of the design may be implemented as part of a simulation following the above design sequence; this would allow performance prediction while writing the system. A combination of Dijkstra's scheme and the Zurcher/Randall proposals has been developed, and used to design and implement one version of the MOS [12]. The current plan is to further confirm the validity of this methodology by writing a multiprogramming subsystem for a real machine.

A final point concerns the specifications of higher level languages for systems programming. Experience with the MOS has justified our enthusiasm about BCPL as a superior language for this project. However, a number of additions would seem appropriate for more general use; these include primitives for process creation, synchronization, and termination; operations for defining and processing common data structures, such as stacks, queues, and tables; and some machine dependent facilities for accessing various hardware registers, such as those in PL360 [15]. Some of these features may conceivably be borrowed from simulation languages. Clearly, more research is needed in this direction.

6. CONCLUSIONS

The MOS and machine deviate from reality in simplifying some feature of real systems and omitting others. Significant features that are lacking include a more general virtual machine permitting multi-step jobs and the use of language translators; a system to organize and handle a broader variety of data files; an operator communication facility; and master mode operation of the CPU in non-zero time to allow measurements of systems overhead. The project spcifications could be expanded to include some of the above, but the marginal utility of the effort appears to be small. However, it is conjectured that similar case studies emphasizing other aspects of operating systems, such as file systems or time-sharing, could be designed.

The MOS case study has proved to be an effective tool for learning and applying some basic concepts about operating systems, and for preliminary testing and evaluation of several new ideas. The study demonstrates that a system not too far removed from reality can be designed and implemented with a reasonable expenditure of effort by a very small group (one to three people), and that there are educational and research advantages in doing so. We believe this will become a widely used approach for the study of operating systems.

REFERENCES

[1] ACM Curriculum Committee on Computer Science. Curriculum 68 - Recommendations for academic programs in Computer Science. Comm. ACM 11,6(March 1968), 151-197.

[2] ACM Curriculum Committee on Computer Science. Report on programming systems workshop, A.L. Engel (ed.), ACM Special Interest Committee on Computer Science Education SICCSE Bulletin 1,3(Oct. 1969), 9-26.

[3] ACM Symposium on Operating System Principles. Gatlinburg, Tenn., October 1967, Comm. ACM 11,5(May 1968).

[4] Dijkstra, E.W. Cooperating Sequential Processes. Technological U., Eindhoven, The Netherlands, September 1965.

[5] Dijkstra, E.W. The structure of "THE" - multiprogramming system. Comm. ACM 11,5(May 1968), 341-346.

[6] Hansen, P.B.. The nucleus of a multiprogramming system. Comm. ACM 13,4(April 1970), 238-250.

[7] IBM Corporation. IBM System/360 Operating System Supervisor and Data Management Services. Form C28-6646, IBM Systems Reference Library, September 1968.

[8] Kelley, J.R. and Wilcox, T.R. BCPL/360 Reference Manual. Dept. of Computer Science, Cornell U., Ithaca, N.Y., February 1970.

[9] Richards, Martin. BCPL: A tool for compiler writing and system programming. AFIPS 1969 Spring Joint Comput. Conf., Vol. 34, AFIPS Press, pp. 557-566.

[10] Shaw, Alan C. A Multiprogramming System. CS413, Dept. of Computer Science, Cornell U., Ithaca, N.Y., Fall 1968.

[11] Watson, Richard W. Timesharing System Design Concepts. McGraw-Hill Book Co., New York, 1970.

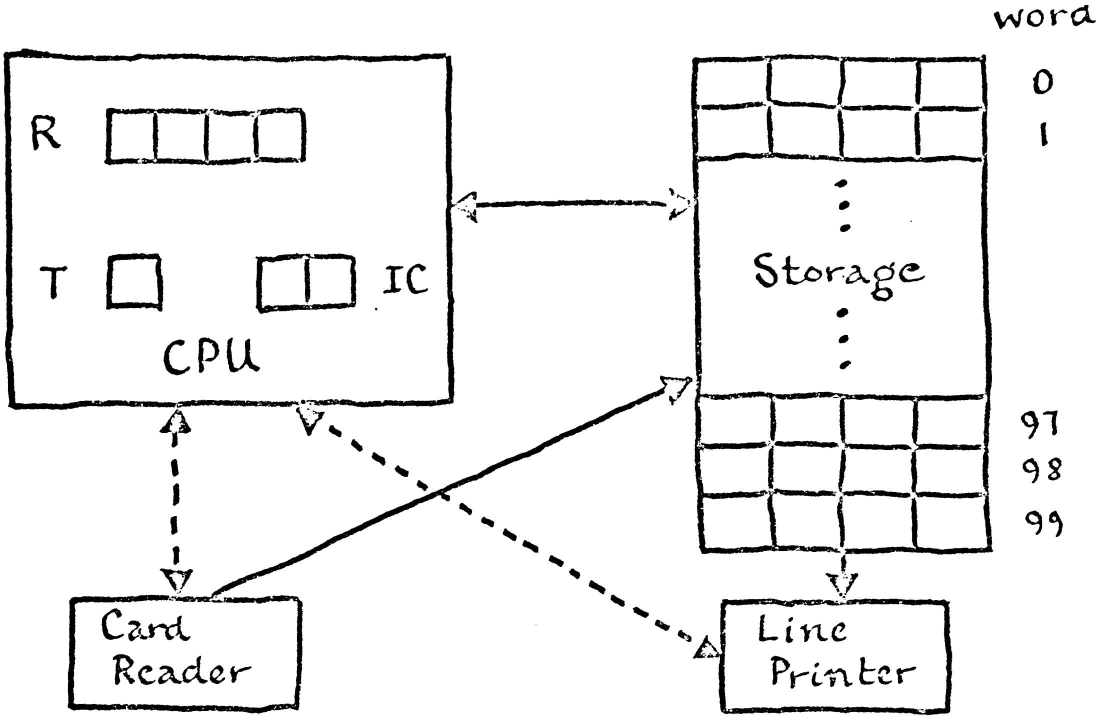
[12] Weiderman, Nelson. A Simulation Technique for Operating System Design and Implementation. Dept. of Computer Science, Cornell U., Ithaca, N.Y., June 1970.

[13] Weiderman, Nelson. Synchronization Primitives and Their Relation to Resource Allocation. Dept. of Computer Science, Cornell U., Ithaca, N.Y., January 1970.

[14] Wilcox, Thomas R. BCPL Coroutine Primitives. Dept. of Computer Science, Cornell U., Ithaca, N.Y., Fall 1969.

[15] Wirth, Niklaus. PL360, a programming language for the 360 computers. Journal of the ACM 15,1(Jan. 1968), 37-74.

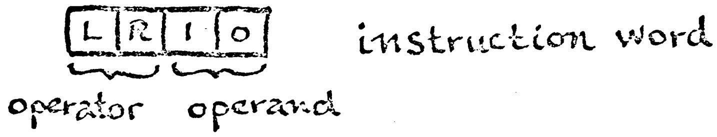
[16] Zurcher, F.W. and Randell, B. Iterative multi-level modeling - a methodology for computer system design. IFIP Congress 68, Edinburgh, Scotland, August 1968.



R: General Register
IC: Instruction Counter
T: Two-state Toggle

□ ≡ 1 Byte (Character)
——— Information
----- Control

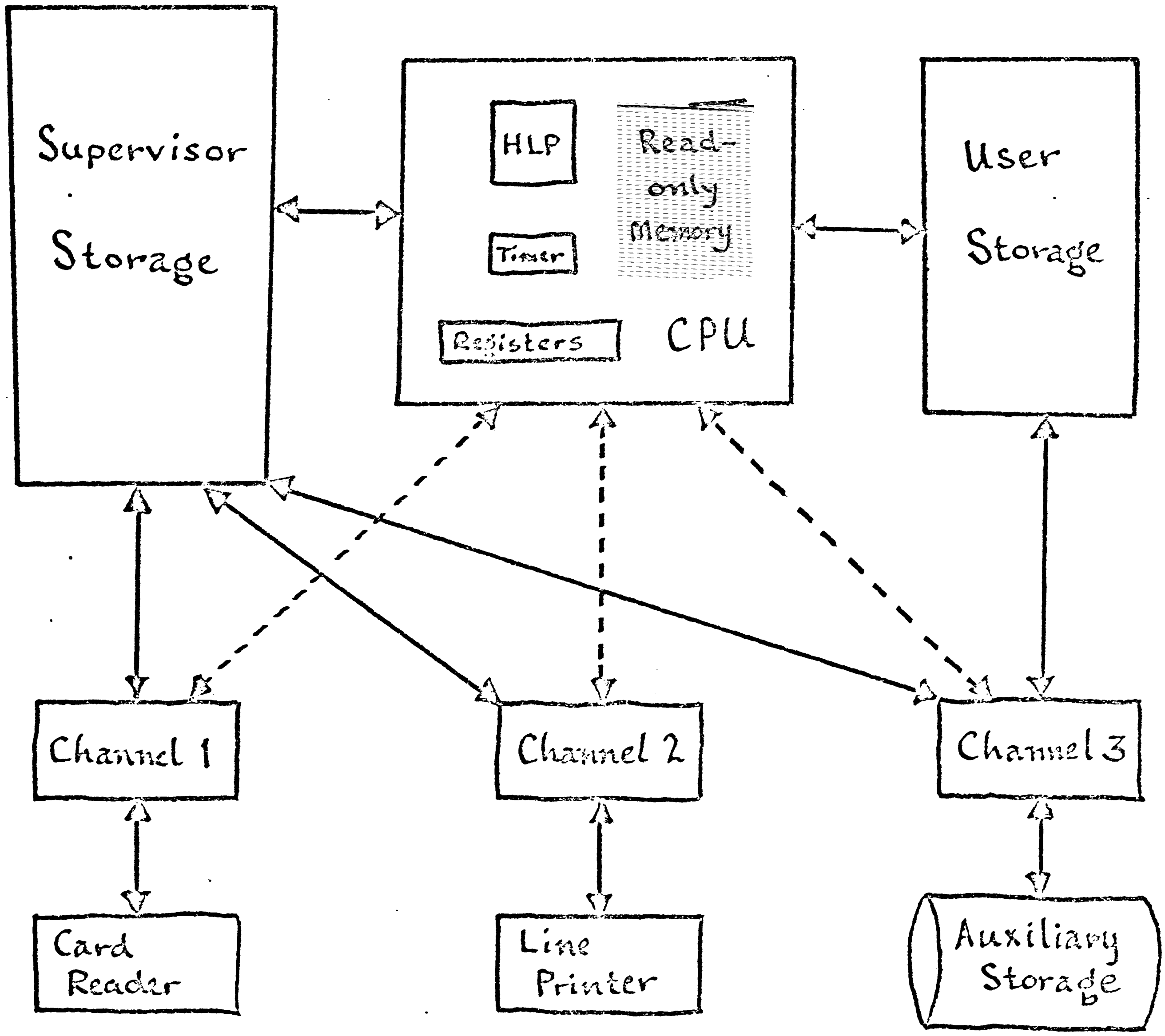
(a) The Virtual Machine



1. Load R (LR)
2. Store R (SR)
3. Compare R to Storage.
   If equal then set T, else reset T (CR)
4. Branch if T is set (BT)
5. Read a Card (GD)
6. Print a Line (PD)
7. Halt (H)

(b) Instruction Set.

Fig. 1: Specifications of the Virtual Machine



HLP: Higher level Language Processor
---- Control Path
—— Information and Control Path

Fig 2: The Real Machine

```
LET LOADER() BE
  $1  LET COMMAND = VEC 5                          || DEFINE TEMPORARY VECTOR
      LET JOB = REQUEST(RESIDENTJOB)               || JOB POINTS TO CONTROL BLOCK
      LET DRUMTRACK = JOB.(FIRSTPROGRAMCARD)       || DRUMTRACK IS DRUM ADDRESS
      JOB.(PAGE_TABLE) := REQUEST(USERCORE)        || ALLOCATE A PAGE TABLE
      JOB.(PAGES) := 0
      WHILE NOEOF(JOB) & JOB.(PAGES) < 10 DO
        $2  COMMAND.(TO@) := REQUEST(USERCORE) || SET UP CHANNEL COMMAND
            COMMAND.(FROM@) := DRUMTRACK
            RELEASE(CHANNEL3,COMMAND) || QUEUE COMMAND FOR
                                      || CHANNEL 3 I/O PROCESSING
            REQUEST(IOCOMPLETION)     || WAIT FOR COMPLETION
            JOB.(PAGES) := JOB.(PAGES) + 1
            ENTER(COMMAND.(TO@),JOB.(PAGE_TABLE)) || UPDATE PAGE TABLE
            DRUMTRACK := DRUMTRACK.(NEXT) || TRACKS ARE CHAINED
        #2
      IF JOB.(PAGES) = 0  DO JOB.(COMPLETIONCODE) := 6 || ERROR
      IF    NOEOF(JOB)    DO JOB.(COMPLETIONCODE) := 7 || ERROR
      RELEASE(READYJOB,JOB) || PUT LOADED  JOB ON READY LIST
  #1
```

Notes: A.(I) denotes vector application

F(x) denotes function application

|| denotes beginning of comment field

$,# <u>begin</u>, <u>end</u> brackets

FIG.: 3 BCPL CODE FOR A LOADER PROCESS

Table 1

PRODUCER AND CONSUMERS OF TYPICAL RESOURCES*

Resource	Processes: Consumer	Processes: Producer
free buffers**	ReadInCards, LinesFromDrum	JobToDrum, PrintLines
card buffers**	JobToDrum	ReadInCards
line buffers**	PrintLines	LinesFromDrum
free core blocks	Loader	JobEnd
free drum tracks	JobToDrum, GetPutData	GetPutData, LinesFromDrum
drum resident jobs	Loader	JobToDrum
channel 3	JobToDrum, Loader, GetPutData, LinesFromDrum	JobToDrum, Loader, GetPutData, LinesFromDrum

---

*This tabular form of presentation was originally suggested by T. Wilcox.

**Part of a buffer pool in supervisor storage