COMPUTER TECHNOLOGY

The design of a real-time operating system for experimental psychology

KENNETH J. BURKHARDT

Department of Electrical Engineering, Rutgers University, Piscataway, New Jersey 08854

This paper describes the structure of the operating system that was utilized in the implementation of the experimental programming language, PL/E. Although the system was originally designed to support programming of psychological experiments on a specific machine (Data General Nova), the operating system can be ported with a minimum of effort to other machines. This paper describes the internal algorithms and control structures used in the system and suggests how it can be moved to other computers.

Given the ever decreasing cost of computer hardware, it is becoming quite clear that the cost of implementing any application on a computer is greatly affected by the degree of effort required to develop system software. In many small computer environments, system hardware costs are often negligible when compared with software costs. Thus, in order to minimize total system cost, it is necessary to utilize appropriate design aids for efficient software development. Although many such tools exist in conventional data processing and scientific environments (for example, higher level languages such as FORTRAN and COBOL), there are a limited number of tools available for real-time system design.

This paper describes the structure of an operating system utilized in the implementation of an experimental language known as PL/E. PL/E is a loosely defined higher level language that has been designed to function as an aid in the implementation of real-time systems software. Its primary use has been in applications associated with on-line experimental psychology. PL/E, however, has also been utilized in several commercial data base management systems.

This paper has four major sections. The first describes software requirements for real-time systems in general and justifies the development of a special-purpose operating system for psychology. The second section briefly describes the features of PL/E and the run-time interpreter that supports the language. The third section describes the control structures and algorithms utilized in the operating system. The final section describes requirements for moving the operating system to other computers.

Development of the PL/E system was supported by the National Institute of Mental Health, Grant MH-21795, Earl Hunt, principal investigator, and the National Institute of Education, Grant NIE-6-74-0104, Clifford Lunneborg, principal investigator, to the University of Washington. John Palmer, Zelda Zabinsky, and Barry Yanoff were all instrumental in the implementation of the original system.

SOFTWARE REQUIREMENTS FOR REAL-TIME SYSTEMS

Software requirements for real-time applications such as experimental psychology must be examined at two levels: the level of the applications software and the level of the operating system software. At the applications level, the problems of real-time programming are quite complex. The wide disparity between potential applications makes even specification of a typical environment extremely difficult, if not impossible. Conceptually, real-time programs require little more complexity than conventional programs. The only constructs that have to be added to non-real-time languages are capabilities for synchronization and capabilities for controlling tasks as a function of time. From a practical standpoint, however, there is one potentially large difference between conventional and real-time programs. In real-time systems the applications programmer must be concerned with operating systems concepts and timing considerations that are transparent in non-real-time applications. This not only makes the task of program validation more complex due to timing considerations, but it also makes the basic task of programming more complex, in that the applications programmer must be aware of system operations at a lower level.

At the operating systems level, the problems of real-time software have been extensively investigated. Brinch Hansen (1977), Wirth (1977), and others (Dijkstra, 1968; Hoare, 1974) have proposed and implemented a variety of higher level languages and other facilities for the design and construction of reliable and efficient real-time executives for both small and large computers. Generally, these software tools include capabilities for controlling concurrent processes, interprocess communication, and general process synchronization. Unfortunately, many of these facilities are inadequately supported by the vendors of mini- and

microcomputer systems. The greatest difficulty encountered in commercial operating systems is the uncertainty of execution speed for many operations. This can seriously hinder the capability to control psychological experiments, in which there is often a desire to control display intervals or measure subject response times to within several milliseconds. Other problems often encountered in commercial systems are difficulty in interfacing nonstandard hardware, limited scheduling capabilities, and lack of support for appropriate higher level languages.

This paper describes a system that was designed to alleviate these problems. The primary emphasis is to describe the structure and algorithms utilized in the operating system that supports the experimental programming language, PL/E. The operating system has removed the majority of the problems associated with real-time programming from the applications programmer and supports facilities to accurately estimate timing requirements associated with various operations.

The Applications Language

PL/E is an algebraic language designed originally for human experimental psychology. The main features of the language are the abilities to process a variety of different variable types, to control external apparatus (e.g., experimental stations, CRTs, response keys, etc.), and to control a large number of subjects simultaneously. Rather than being a systems programming language such as MODULA (Wirth, 1977) or CONCURRENT PASCAL (Brinch Hansen, 1977), PL/E is strictly an applications-level language, with virtually all timing control and other synchronization tasks performed by the underlying operating system (for a more detailed description of the system, see Burkhardt, 1976, 1977; Palmer, McCloud, & Loftus, 1978).

PL/E is not directly executed. It is an intermediately interpreted language that is compiled before execution takes place interpretively under control of the run-time system. The code generated by the compiler is in a three-address prefix format. For example, $I = X + Y$ is compiled as

```
ADD
@ X
@ Y
. @ I
```

where ADD corresponds to an integer constant pointing to an interpreter branch table, and X, Y, and I are pointers to descriptors of the appropriate variables. The calculation of variable addresses is performed by a single system subroutine (GETARG) that examines the descriptor to determine the data address, verifies that the address is valid, and returns the address of the argument.

This very simple organization makes it easy to add

either pseudoinstructions (commands) or new data types to the PL/E language without having to modify existing software. New commands can be added to the PL/E simply by adding the command name and information about its parameters to a system data file read by the compiler at run-time. New data types or structures can also be added to the system with relative ease. For example, structured data records (similar to COBAL or PL/1) have been added to PL/E simply by modifying a single systems routine and adding a new data-type definition command to the compiler.

The Operating System

To a large extent the system can be described as an operating system without an operating system. The only required element of the resident system is a 100 instruction kernel, the scheduler. The scheduler consists of two modules: the interrupt service routine and the background scheduler, or interpreter. The interrupt service routine assigns priorities to the various devices and, on detection of interrupts, passes control to the appropriate device handlers. If after processing interrupts, the interrupt service routine determines that the system was previously idle, control is passed to the background scheduler.

Before describing the actual operation of the operating system in detail, it is useful to define the control structures utilized in its realization.

Control structures. The control structures of a system are the basic data structures manipulated by the operating system during execution of a program. The control structures define the status of individual subject programs and the status of individual input-output (I/O) devices. The four basic structures utilized in the PL/E system are semaphores, program status vectors (PSV), doubly linked lists, and dope vectors. The first three structures are utilized internally by the operating system and are totally transparent to user programs. Dope vectors are generated by the compiler and are utilized to pass user program variables to lower level routines.

Semaphores. Semaphores are special integer variables proposed by Dijkstra (1968) as synchronizing flags to be utilized in the allocation of resources. A semaphore is associated with any device that can be simultaneously accessed by more than a single task (e.g., disk, console TTY). A device is referenced internally by means of two basic operations: allocate (P) and release (V). The implementation of the P and V operations is quite similar to that proposed by Dijkstra (1968). Whenever a task requires a device, the appropriate P operation is executed and, if the device is free, it is allocated to the task. When finished with a device, the task releases the device and reschedules any suspended tasks by means of the V operation.

The P and V operations are implemented according to the following algorithms:

```

      Procedure P (Device)
Disable Interrupts
IF (Device-flag.Eq. 0)
    THEN Device-flag = 1
          Enable Interrupts
          Return
    ELSE Device-flag = Device-flag + 1
          Append PSV on Device-list
          Enable Interrupts
          Go to Scheduler
      DONE

      Procedure V (Device)
Disable Interrupts
Device-flag = Device-flag - 1
IF (Device-flag.Eq1. 0)
    THEN Enable Interrupts
          Return
    ELSE TEMP = POP (Device List)
          Append TEMP to XEQ-List
          Enable Interrupts
          Return

```

Note that interrupts are disabled during both P and V. This is to insure mutually exclusive access to the resource scheduler. Without mutual exclusion, the system can only process a single subject at a time.

Program status vector. The PSV (Figure 1) is a sequence of variables that defines the status of the system at any point in time. Included in this list of variables is the pseudoprogram counter, task identification, high- and low-memory limits (for the implementation of storage protection), and various other status information. The PSV is altered primarily by the scheduler during the start-up and suspension of tasks.

Doubly linked lists. A doubly linked list (Figure 2) is a data structure consisting of a list of entries, each

containing forward and backward address links together with sufficient storage to save a task's PSV. The primary purpose of the doubly linked list structure is to act as a scheduling aid for each of the hardware resources in the PL/E system. There is a doubly linked list associated with each I/O device, with the timer and with the CPU itself.

By utilizing a series of system subroutines, a process can insert, delete, append, or remove the first entry from a doubly linked list. Thus, this data structure allows a list to be utilized as a stack, a queue, or an ordered queue. The methodology of queue manipulation defines the resource-scheduling algorithm. For example, the schedule assigns a task the right to the CPU by removing the first entry from the execution list (XEQ) and copying the associated status to the active PSV.

Dope vector. The dope vectors are the variable descriptors that are utilized by command routines to access user program variables. The dope vectors are

Subject Number
Pseudoprogram Counter
Maximum Memory Address
Minimum Memory Address
I/O Return Address
I/O Device in Use
Format Specification Address
Expiration Time
Response Key Mask
Response Key Data
Temporary Storage

Figure 1. The process status vector.

Figure 2. The doubly linked list data structure.

set up initially by the compiler to define variable type, storage limits, and mode of addressing. The primary reason for utilizing dope vectors, rather than storage addresses is to be able to support parallel program execution (via relative addressing) and complex data types, such as arrays and strings.

The Schedulers

Given the basic control structures and assorted system-level routines for manipulating these structures, the implementation of the operating system is fairly straightforward.

Background scheduler. The basic scheduling algorithm of the system is first come, first served. A subject retains control of the machine until issuing a request for an I/O operation or a delay. The device driver then suspends the subject until the I/O operation is complete.

The background scheduler is activated either after an interrupt has occurred (when the system was previously idle) or after a subject has been suspended.

The scheduling algorithm is as follows;

```

If (XEQ-list .EQ. EMPTY)
  Then PSV = IDLE
  ELSE PSV = POP (XEQ-list)
  DONE.

```

If a subject is waiting for execution, the first entry of the execution list will be copied to the PSV, and control is passed to that subject. If no subject is ready to proceed, the system simply enters an idle mode.

Interrupt scheduler. The operation of the interrupt scheduler varies depending on the interrupt processing hardware of the associated processor. In the case of the Data General Nova, which has only a single true level of interrupt priority, the interrupt scheduler has the tasks of saving registers, determining what device caused the interrupt, and then passing control to the appropriate device drivers. In this case the priority of a device is determined by the order in which the interrupt scheduler tests for its having caused an interrupt. On exit from an interrupt, control is returned to either the interrupted program or the background scheduler as a function of whether or not the system was previously in an idle mode.

Device handlers. All hardware resources in the system have at least two levels of supporting software: the background monitor and the interrupt-level device driver. The background monitor allocates devices, suspends the subject program, and physically starts the device. The interrupt driver responds to device interrupts and, when appropriate, reschedules the background monitor.

The background monitor is entered directly from the subject program. The following is a sample control sequence for a background monitor and an interrupt driver.

Background Monitor

```

P (Device); Allocate Device;
Insert PSV at head of Device List
START I/O
WAIT for Response
V (Device)
Go to Scheduler

```

Interrupt Driver

```

Clear Interrupt
TEMP = POP Device-List
Append TEMP to XEQ-List
EXIT INTERRUPT

```

Note that only a single list is required to keep track of the subject that has actually been allocated a device and subjects waiting for a device. The P operation moves a subject trying to gain access to an allocated device to the end of the device list. The background and interrupt drivers, however, utilize the first list entry for controlling the subject actually allocated to the device.

Timing control. The device drivers utilized with the real-time clock are handled somewhat differently from those of regular I/O devices. Since the real-time clock can support more than a single subject simultaneously, timer list entries are ordered by time. The frame of reference utilized for timing control is time in milliseconds from system start-up. When a subject issues a WAIT (time) command, the background handler calculates absolute expiration time and then inserts a list entry at the appropriate position in the timer list.

The interrupt-level driver then checks the first entry in the list each time the timer generates an interrupt (every millisecond). The primary advantage of ordering entries by time is to minimize system overhead in the timer interrupt handler.

Response key. Response keys are also handled differently from other I/O devices. At start-up time, each subject is allocated a unique keyboard; thus there is no concern with shared access to a device. A subject program accesses a keyboard either by waiting for response or by waiting for a response to occur within a certain period of time. In both situations, a mask is set in the response list entry that defines allowable responses. In the case in which the device may time-out, a second entry is created for the timer list. When a response key is pressed, the interrupt driver locates the appropriate entry and, if a valid response has occurred, moves the entry to the XEQ list.

Formatted I/O. The formatted I/O handler is the most complex portion of the PL/E run-time system. It requires a second level of interpretation to interface device drivers to the format specifications outline in subject programs. The format specifications are treated as instructions, whereas the normal instruction stream is treated as data. Although a detailed description of the formatted I/O handler is beyond the scope of this paper, it is useful to examine its operation at a top level. Consider the following PL/E statement sequence:

WRITE - TT FORM 1. RESPONSE
FORM1: FORMAT (<X<5> I<5> X<1> 'MS')

This sequence might correspond to printing some preliminary results on a Teletype. The two statements would be compiled as follows:

```
WRITE - TTY
FORM1
@ RESPONSE
FORM 1: X<5>      ; ASCII string corresponding
      I<5>        to I/O format specification
      X<1>
      'M
      S '
      0
```

Execution of the WRITE-TTY statement causes control to be passed to the top-level Teletype monitor, where the device is allocated to this subject by means of the P(TTY) operation.

Control is then passed to the formatted I/O handler, which examines the ASCII string stored at FORM1 one character at a time. When a format specifier (X,I,') is detected, control is passed to the appropriate lower level routine that performs code conversions based on in-line format specifications. If data are required to complete this conversion, they are acquired from the normal input stream.

Physical I/O is handled by the device drivers and is handled either a format field or a buffer at a time, depending on the device.

Timing considerations. Timing estimates for various operations within the PL/E system have varying degrees of accuracy, depending on the priority of the operation. Timing and response key related operations offer 1-msec accuracy because they are executed at the highest priority and have a minimal amount of software overhead. Display refresh operations can also be accurately controlled because external hardware insures a refresh operation every 10 msec. True execution limits for low-priority operations such as data formatting and arithmetic operations are more difficult to obtain due to the multiprogrammed nature of the system. Theoretical execution times range from under 50 microsec to several hundred microseconds. The actual time required, however, varies significantly, depending on the number of subjects active. In the majority of cases with minimal calculations, delays caused by multiple subjects are well within the 1-msec accuracy of the system. Some problems have been encountered in attempting to minimize interdisplay intervals (cf. Palmer et al., 1978). This difficulty, however, is due primarily to display hardware, rather than to a limitation of the software.

PORTABILITY

Although initially implemented in assembly language on a Data General Nova, the PL/E operating system is sufficiently flexible to be moved to other machines. Basically, the major requirement is to be able to support the control structures described earlier in this paper. These structures can be implemented on almost any machine in use today. At Rutgers, a project is presently moving the operating system to an Intel 8080-based hobbyist computer. In this situation the system will be coded in a higher level language (PL/M) to speed system implementation and simplify validation.

Moving the compiler for PL/E is a more complex task. Although it has been written in FORTRAN on Nova and PDP-11/60 computers, many small systems do not support the recursive features required by the parser section of the compiler. An alternative to porting the compiler is to cross-compile applications programs on a timesharing system.

SUMMARY

This paper has described the internal algorithms and control structures utilized in an operating system for computer-controlled psychology. Although the higher level language, PL/E, supported by the operating system is fairly powerful, the simplicity of the underlying system has acted to both simplify system implementation and aid in the validation of experimental programs.

REFERENCES

- BRINCH HANSEN, P. *The architecture of concurrent programs*. Englewood Cliffs, N.J.: Prentice Hall, 1977.
- BURKHARDT, K. J. EMPP: An extensible multiprogramming system for experimental psychology. *Behavior Research Methods & Instrumentation*, 1976, 8, 239-244.
- BURKHARDT, K. J. PL/E—An extensible higher level language for real-time systems application. *Proceedings of the Fifth International Symposium on Mini- and Microcomputers*, 1977, MIMI 77, 35-40.
- DIJKSTRA, E. W. Cooperating sequential processes. In F. Genuys (Ed.), *Programming languages*. New York: Academic Press, 1968.
- HOARE, C. A. Monitors: An operating system structuring concept. *Communications of the ACM*, 1974, 10, 549-557.
- PALMER, J. C., McLEOD, C. M., & LOFTUS, G. R. PLE: A high-level multiprogramming language for psychology. *Behavior Research Methods & Instrumentation*, 1978, 10, 764-772.
- WIRTH, N. Modula: A language for modular multiprogramming. *Software Practice and Experience*, 1977, 1, 3-35.

(Received for publication June 27, 1979;
revision accepted July 24, 1979.)