The NMFECC Cray Time-Sharing System

KIRBY W. FONG

National Magnetic Fusion Energy Computer Center at the University of California, Lawrence Livermore National Laboratory, Livermore, California 94550, U.S.A.

SUMMARY

The National Magnetic Fusion Energy Computer Center (NMFECC) at the Lawrence Livermore National Laboratory (LLNL) has implemented a simple, yet powerful interactive operating system, the Cray Time-Sharing System (CTSS), on a Cray-1 supercomputer. CTSS augments the multi-programming batch facilities normally found in supercomputer systems with many of the interactive services typical of interactive minicomputer systems. This paper gives some of the historical background leading to CTSS and gives an overview of the system that emphasizes the strong points or unusual features such as multiple channels, decentralized control of resources, priorities and program scheduling, system recovery, and on-line documentation.

KEY WORDS Supercomputers Operating systems Time-sharing Interactive

OVERVIEW AND ASSUMPTIONS

To derive the greatest benefit from a supercomputer, one must have an idea of how it can and should be used. The NMFECC believes that interactive computing is the foundation for full beneficial use of supercomputers as well as for smaller computers. Not all supercomputers are suitable for interactive use;[1] however, the Cray-1 is suitable. If a machine can interact, it should. Since interactive use of supercomputers is at variance with the conventional wisdom about these machines, we begin by explaining the reasons for this approach.

First, interactivity opens the door to more productive use of the user's time if the appropriate program development tools are available. This reason[2] applies to computers of any size. Although text editors then become feasible, text editing is not the principal reason for wanting an interactive system. After all, one could use an interactive front-end machine to repair one's source code. The real reason is to reduce the calendar time needed to develop programs. This goal can be achieved if the system has a symbolic, dynamic debugging program that assists users in finding errors rapidly. A secondary benefit of fast debugging is that users are inclined to be more adventurous and innovative in their programming if they know that new features can be implemented quickly. Debugging of major programs frequently cannot be done on any other machine than the supercomputer because they are usually machine or system dependent or use library routines available only on the supercomputer. Any attempt to debug through a front-end machine usually degenerates into the insertion of debug print statements in the source or the taking of memory dumps from the supercomputer.

0038–0644/85/010087–17$01.70 *Received 21 January 1983*
© 1985 by John Wiley & Sons, Ltd. *Revised 20 September 1983 and 21 May 1984*

Secondly, interactivity saves the expense of a front-end machine, reduces file transport traffic, and avoids the time required to translate file formats and character codes when two machines are involved.

Thirdly, interactivity allows the power of a supercomputer to be used for other resource-intensive services besides number crunching. One example is graphics postprocessing that allows the user to inspect a graphics file or to compute elaborate pictures for display on a graphics terminal. It is common for the hidden-line removal or shading in a graphics postprocessing phase of a mechanical engineering analysis to need as much CPU power (though for a shorter period of time) as the analysis itself. A second example is symbolic algebra—an activity that is normally interactive but sometimes requires large amounts of real memory.

Fourthly, some number crunching codes are themselves interactive. These are hydrodynamics simulations where the user observes the course of the calculation on a graphics display and intervenes when modifying the course of computation is needed.

To sum up, interactivity makes the machine more versatile, more convenient, and more useful—exactly the same arguments that apply to minicomputers. Furthermore, interactivity does not preclude using the supercomputer for batch processing. It simply allows the machine to be more flexible in adapting to the needs of users.

Supercomputers can perform more economically many of the tasks that are commonly done on minicomputers. One reason is that there is an economy of scale in the purchase, maintenance, and operation of a supercomputer over the equivalent host of minicomputers. The NMFECC, for example, also has a DEC-10 system, which is representative of the more powerful minicomputer systems. Although the Cray-1 costs about twenty times as much, it is also about eighty times faster. A second reason is that supercomputers are generally kept busy all the time whereas minicomputers are frequently allowed to go idle at nights and weekends—a clear waste of capital resources.

A central computer centre should do more than accept the large programs that users cannot run on their own minicomputers. It must provide an attractive facility by turning supercomputers from mere number crunchers into convenient research tools. Since supercomputers already have the cost advantage, it remains only to provide a competitive programming environment. The significance of achieving this goal is twofold. One is that users receive good service as well as high performance hardware for their money. The second is that it demonstrates that a centralized computer centre can viably provide general scientific computing services as opposed to being a special facility, housing special equipment to run a handful of large production codes.

LTSS: A CTSS FORERUNNER

When the CDC 6600 computer became available in the early 1960s, the local computer centre at LLNL decided to write a time-sharing operating system for it. This was done at a time when vendors of mainframes supplied only batch processing systems, and time-sharing systems such as the MIT CTSS[2] (Compatible Time Sharing System) and MULTICS[3] were research efforts. This system was the first in a series of systems collectively known as the Livermore Time Sharing System (LTSS). This first system (1965) was written in assembly language, a normal practice for that time.

During the same period, FORTRAN was rising to prominence as a scientific programming language. Since it was clear that a high-level language enabled users to become more productive, the laboratory decided to incorporate some extensions into FORTRAN to facilitate systems programming. The resulting language was christened LRLTRAN,[4] after the laboratory's name at that time (Lawrence Radiation Laboratory). This permitted supporting only one compiler, not separate compilers for users and systems programmers. A second CDC 6600 time-sharing system[5] (1967) was then written in LRLTRAN. Also the LRLTRAN compiler was rewritten in LRLTRAN. LRLTRAN was used for systems programming because no other compilers appropriate for writing systems software were readily available on the CDC 6600 then, and FORTRAN was the only high-level language of interest to the first users of the CDC 6600.

Although the use of a high-level language was intended primarily to ease software maintenance,[6] there was a second benefit. The advent of the CDC 7600 computer meant that the entire programming environment on the CDC 6600 needed to be bootstrappped onto the CDC 7600. The similarity of the two architectures and the fact that most software (including the compiler) was written in LRLTRAN enabled the laboratory to move software to the CDC 7600 by modifying the code generator of the compiler and making changes in source to use the two-level memory of the CDC 7600. This third system[7, 8] was placed on the CDC 7600 in 1969.

Needless to say, LTSS did not spring forth complete with all the desired facilities and free of operational problems at its inception. Recognizing, understanding and correcting problems has been an ongoing process. Also, the appropriate tools for editing, debugging, and graphics display were initially unknown and could be developed only as the laboratory gained insight through actual experience.

THE NMFECC

The NMFECC came into being because magnetic fusion researchers supported by the U.S. Department of Energy realized that they would need access to supercomputers to carry out the numerical simulations needed to study the magnetic confinement of plasmas. These researchers were distributed at different laboratories throughout the U.S.; however, not all research sites had locally accessible supercomputers, and it was clearly not feasible to buy supercomputers for every laboratory. Thus, the Department of Energy chose to build a central computer centre along with a nationwide communications network.

LLNL was designated as the site of the NMFECC in 1973. The NMFECC began operation a year later with a borrowed CDC 6600 from the laboratory's local computer centre. With the acquisition of its own CDC 7600 in 1975, the NMFECC became a completely separate computer centre, both physically and administratively. Naturally, the interactive operating system used on the laboratory's CDC 6600 and CDC 7600 computers was also used at the NMFECC since, initially, the only access for remote users was dial-up connections for terminals. In 1976, the NMFECC installed its data communications network consisting of 50-kilobit land lines. These lines connected DEC-10 computers at four major remote sites to the CDC 7600 at the central site. The DEC-10s served as remote terminal concentrators and also drove remote printers in addition to providing some local computing capability.

The demand for service soon outstripped the capacity of a single CDC 7600. In 1977, after a competitive bid for the latest supercomputer, the U.S. Department of

Energy selected a Cray-1 computer[9] for the NMFECC. The contract, signed in November 1977, called for delivery of a Cray-1 in May 1978.

THE BEGINNINGS OF CTSS

The NMFECC could integrate the Cray-1 into its network in one of two ways. One way would be to use the vendor's multi-programming batch operating system and have the existing, interactive CDC 7600 serve as a front-end to stage jobs in and out of the Cray-1. The second way was to write an interactive system for the Cray-1. The second way required considerably more work. It was a serious alternative, however, for the three reasons discussed in the first section and because most of the necessary software was already written in a single high-level language.

A self-sufficient Cray-1 also offered three further benefits. One is that the Cray-1, consisting of fewer discrete components than the CDC 7600, should be relatively more reliable than the CDC 7600. We did not want to depend on a front-end that was less reliable than the Cray-1 and whose problems could cut off our only access to a functioning Cray-1. Secondly, the natural evolution of a computer centre is to replace the oldest, least cost effective equipment with more modern equipment. We did not want to place critical functions on our CDC 7600 that would artificially prolong its life. Thirdly, we wanted to encourage users to convert swiftly to the Cray-1. To do this, we would have to provide a programming environment that was not only convenient but one which was similar to the CDC 7600 environment so that users would not have to learn about an utterly different system. Consequently, we decided to produce a new LTSS suitable for the Cray-1. We named this system the Cray Time Sharing System (CTSS) to avoid confusion with the previous versions of LTSS.

We began work in 1977 on an LRLTRAN cross-compiler that would run on the CDC 7600 and produce relocatable binaries for the Cray-1. Also, we installed a Cray-1 simulator on the CDC 7600 to check out our Cray-1 code. Concurrently, we started work on the operating system, cross-loader and I/O library. In 1978, additional staff started work on basic utility routines, a text editor, a debugger, the batch processing subsystem and network and permanent file connections. We also recognized that we could not have everything ready by May 1978; in particular, we could not begin real debugging until we had a Cray-1 on which to run our software even though we had compiled and simulated code on the CDC 7600 as early as February 1978. Therefore, we simultaneously wrote software to allow our CDC 7600 to serve as a front-end for the vendor's system on the Cray-1. The machine was installed as scheduled at the beginning of May 1978 along with the vendor's system and was immediately available to users. Beginning initially at two hours per day, we used the machine to run and debug CTSS and its associated software. In June 1978, CTSS executed its first program. Time-sharing gradually increased to eight hours per day by midsummer. Meanwhile, users began code conversion to CTSS while still having the vendor's system available for batch jobs at night. In mid-September 1978, CTSS was running full time, and we phased out use of the vendor's system.

Although it appears remarkable that we could develop a usable time-sharing system and programming environment in less than one year, closer inspection reveals that this accomplishment was not really so astonishing. For example, the CDC 7600 I/O library and batch processing subsystem had been rewritten in 1977 to improve user friendliness, maintainability and transportability, so that these major pieces of software were already restructured to permit easier conversion. They were of course written in LRLTRAN, not assembly language. Furthermore, all the logic for handling system tables, accounting and scheduling transferred from the CDC 7600 to the Cray-1, so that no major design work was necessary. Finally, nearly all of the essential CTSS utility routines already existed in CDC 7600 versions in LRLTRAN, so that large amounts of CDC 7600 developed code were applicable to the Cray-1. Much of the conversion work was to accommodate eight 8-bit characters per word instead of ten 6-bit characters. From our point of view, the development of CTSS was not really a gamble because we were not attempting to implement any unproven concepts. The only question was when CTSS would work reliably enough for users to use it productively.

The initial CTSS effort involved eleven programmers working full time or nearly full time. Of these, only four were involved in the operating system itself. The remainder plus seven others who worked part time on the CTSS effort took care of compilers, libraries and service routines. In addition, four people worked full time on documentation. Thus, twenty-two person-years was the upper limit on the effort needed to produce the initial release of CTSS. Between 1978 and 1982, we have invested another twenty to thirty person-years on system enhancements, the development of more service routines and libraries, and documentation.

CTSS FROM THE USER'S PERSPECTIVE

The first unusual aspect users notice about CTSS is that when they log in, they are required to give a *suffix* in addition to the expected user number, account number and password. A suffix is one of the letters 'a' to 'e' and is the vestige of an early attempt to allow users to run up to five concurrent and independent programs by providing a software switch so that a single terminal could have up to five virtual paths into the same work space. The user can switch from one suffix to another and interact with the programs running under each suffix. Most users log in to suffix 'a'. The user *byes* (switches) to another suffix by typing (CTRL-e), the desired suffix and carriage return. To log out completely, the user types (CTRL-d). Terminal I/O (as in UNIX[10]) is full duplex.

When the user logs in, CTSS indicates which, if any, of the suffixes are active (*i.e.* have programs still running) and how much time is left in the *bank account* that the user may use. The bank account is an important concept because it is used to manage computer time allocations to users. A bank account is measured in minutes and is generally shared among a group of a dozen or more users who are working on a common project. Each bank account has at least one administrator who is, typically, one of the users of that account. We normally initialize each bank account on Monday morning with the amount of time that bank account has been allocated for that week. For bank accounts with few users, the administrator for that bank account may allow any user to drain the entire bank account on the assumption that some members may not compute during a particular week and that the others should be able to use the allocation. The administrator of a bank account with many users may assign maximum percentages to each member in the bank account to prevent any one user from using too much in one week. The key aspect of bank account percentages is that the users' supervisors, not the computer centre staff, set the percentages. Those in the best position to know how much time each user should be using then have the power to set individual limits. This decentralization of resource management was also a MULTICS[3] goal.

Users discover that there are very few commands they can issue directly to the operating system. On CTSS, all non-trivial actions are carried out by programs that the user must execute. All commands executed directly by the system start with the control character (CTRL-e) followed by a few more characters to specify the command. Commands fall into two categories—those that ask for information and those that change something. Typical enquiries ask for the user's available time, the remaining bank account time, the accumulated charge for the currently running program, the remaining time limit for the currently running program, the priority of the currently running program, the amount of time the user has used since the last bank account update, the state of the currently running program, the length of the active queue (*i.e.* how many programs currently want to use the CPU), and the length of the inactive queue (*i.e.* the total number of programs currently in some state of execution). Typical changes are to abort the currently running program, to discard a stream of terminal messages being sent by the currently running program, to change the priority of the currently running program, to interrupt the currently running program, to switch to another suffix and to send a message to the computer operator.

To begin executing a program, the user must specify at least one of three pieces of information. The one required piece is the name of the file that contains the executable memory image of the program. The second is a message for the program, and the third is information to CTSS on the time limit and priority for the program. The user types all of this information on a single line. The first symbol on the line is interpreted by CTSS as the name of an executable file. The sequence ' / *t v*' at the end of the line, where *t* and *v* are numbers, is the optional specification of time *t* and value *v* in minutes. Any remaining characters are considered to be a message to the program.

Basic form: program message / *t v*
Example: CFT I = SRC, B = BIN, L = LIST / 1 2

The example executes a compiler giving it a message telling what files to read and create.

The time *t* is the maximum number of minutes the program should be allowed to execute. The value *v* is the maximum number of minutes the user is willing to spend from the bank account. The program's *priority* is its value divided by its time. Thus a user who *bids* ' / 2·5 4' wants to run at priority 1·6. If the program actually terminates after 2 min, the user's bank account will be decremented by 3·2 min (the charge time multiplied by the priority). We permit priorities in the range 0·1 to 2·0 and standby priority of 0·03. We explain below how CTSS uses the bid priority in determining how quickly the program will be serviced. Users who are in a hurry can bid a high priority and rapidly deplete their bank account. They are trading part of their allocation for immediate service. Conversely, users who bid a low priority are willing to wait for service and are rewarded by getting more computer time than their nominal bank account allocation. If users do not specify the time and value, CTSS uses defaults of 1 and 1.

The user may attempt to execute any file; however, those that do not contain an executable memory image will soon abort. For reasons described below, executable files are called *controllees*. If there is a message on the execute line, the system holds it in an input message buffer until the program makes a system call asking for the message. The system does not interpret the message in any way. The user must know what kind of message the program expects. If a program is interactive, it will generally prompt the user for more input or commands as well as complain about incorrect input. Subsequent lines typed by the user are not considered to be execute lines (since a program is running) but are taken by CTSS as input messages for the running program. When the program terminates, the system generates an 'all done' message at the user's terminal.

As users start using CTSS, they begin to create files whose names and sizes can be displayed by executing FILES. This program typically displays the attributes of a user's *private* files. These files are accessible only to the user, unless copies are given to other users.

Private files are distinct from *public* files, which are accessible to everyone, and *political* files, which are accessible to sets of users. When the user types an execute line, CTSS first looks at the user's private file set for the executable file. If the file is not there, the system searches the set of public and political files. Users can execute political files if they are in the set of users having access to them. Files essential to the operation of CTSS are public files. Most of the commonly needed programs (such as compilers and editors) and libraries are in political files that happen to have universal access. (This split allows the operating system to be deadstarted when public files are restored but before all the utilities and libraries are restored.)

The need for large amounts of scratch disk space by production programs has greatly influenced the file system developed in CTSS (and its predecessors). Only the public and political files are permanent. Users' private files are supposed to be transitory working files that are in constant use during program development. CTSS is intended for use in a network where other equipment or systems handle the long term storage of files. Intimate knowledge of the permanent file facility is not built into CTSS itself but into a utility program that is privileged to send and retrieve copies of files between the Cray-1 and the storage facility. Since users were presumed to have only a modest number of working files, we thought a flat file structure would be adequate. This is in contrast to a hierarchical file structure commonly available in other operating systems.[11–13] The flat structure also leads to a simple file index structure that minimizes the overhead of system access to the file index.

Although this file system does work, there are some side effects. One is that switching suffixes does not change the file space, so programs running under two suffixes can interfere with each other. It would have been better to implement at least one level of file directory and have the concept of switching to different directory spaces rather than different suffixes. The second is that the absence of a unified file directory structure encompassing both CTSS and the long term file storage facility means CTSS cannot easily migrate unused private files out to storage. CTSS consequently must purge private files if they persist for some time without being used. We find purge times of 32 hours on weekdays and 72 hours on weekends to be workable. Users who work every day are able to retain their files so that they do not have to retrieve everything from the file store each day.

There is a special kind of private file, called a *dropfile*, and it is the file into which CTSS writes the memory image of a program when the program is swapped out of memory. All publicly available programs and most user programs ask the system to create a distinct dropfile in the user's private file space before the expiration of their first time slot in memory. Since Cray-1 memory is not paged or segmented, CTSS must roll an entire memory image of a program out to disk. One normally does not want to overwrite the original executable file with the memory image, so one creates a dropfile.

A number of benefits accrue from allowing the dropfile to be accessible to the user as a private file. If the user aborts a program and later decides to resume execution, the dropfile receives the memory image at the time of the abort and is itself an executable file. The user can then type the dropfile name as an execute line in order to resume running the program. A second benefit is that the user may restart the dropfile in case of a system crash in which running programs were lost. A third benefit is that very long running programs need not make restart dumps. When a program exceeds its time limit, its final memory image is left in the dropfile. If the dropfile and all necessary input and output files are retained (*e.g.* written out to the permanent file store and retrieved later), the dropfile can be restarted another day when the user has more bank account time to continue. A fourth and the most useful benefit of the dropfile is that it enables the user to debug the program dynamically without having to reload the program. A symbolic debugging utility can set breakpoints in the dropfile and run it as a controllee, allowing the user to stop at selected subroutines and labels to interrogate for the values of variables.

Finally, the casual user notices that we frequently refer to *utilities* or *utility routines* rather than commands, verbs or control cards. Virtually all actions of any consequence are cast in the form of executable programs that are placed in public files. These programs of general use are referred to as utilities or utility routines. Different members of the computer centre staff generally are responsible for and maintain the various public and political files. Even such important programs as compilers and loaders are utilities whose political files are in no way special to CTSS. There is no tight binding (another MULTICS[3] idea) between CTSS and most of the programs executed by the user. Much of the user's impression of CTSS really depends on the behaviour of the various utilities and libraries. Another computer centre could install CTSS and greatly alter its appearance and performance by using different utilities and libraries.

File characterization

Files are treated as strings of 64-bit words. Words are individually addressable and randomly accessible. CTSS does not (*e.g.* by file name extension) maintain information about the nature of data in a file. CTSS also does not maintain a position pointer for files. Every file access must therefore specify the starting word address and the number of words to be transferred. This very basic structure permits users to impose the data structure and organization that best suit their needs.

Naturally, the FORTRAN I/O library and other utilities co-operate in imposing a structure of *text* files and *sequential record* (binary) files, but their conventions are of no concern to CTSS. It is the responsibility of I/O routines loaded into a program's memory image to maintain any necessary I/O buffers (also contained in the program's memory image) as well as manage the control information in a disk file (such as inter-record separators) that define the file's (conventionally agreed upon) format. A file not in text or sequential record format is considered to be in *absolute* (word addressable) format. In fact, major production programs generally do little FORTRAN formatted or unformatted I/O, but do absolute I/O instead, reading and writing large blocks of data. CTSS was kept simple and relatively small so that large programs could have complete control of their I/O and have a lot of memory.

CTSS permits asynchronous I/O; that is, a program can use the CPU and do I/O concurrently. Although our FORTRAN I/O routines could double buffer I/O to take advantage of this concurrency, we have chosen to implement simple single buffering. This recognizes that the major Cray-1 programs primarily use absolute I/O and already overlap CPU and I/O. Indeed, some programs actually do two or more I/O transfers and compute concurrently. Of course, the standard FORTRAN I/O routines also do absolute I/O since that is all CTSS supports, but they will give up the CPU until I/O completes before they resume computing. We consider this loss to be tolerable since cutting I/O buffers into halves in order to double buffer will tend to double the number of disk accesses made by a program using standard FORTRAN I/O. Doubling the sizes of buffers would avoid increasing the number of disk accesses, but would make programs larger so that they would be more difficult to fit into memory.

The standard FORTRAN I/O routines move multiples of a disk sector's worth of data (512 words) and read and write on a sector boundary in the disk file since this is the most efficient transfer. CTSS will, however, support transfers as short as one word between any memory location and any disk location in either direction.

CTSS carefully maintains the privacy of information in users' private files. A user cannot examine the contents of another user's private files. When a user destroys a file, CTSS writes a pattern over every word in the file before releasing the disk space for reuse. Therefore, a newly created file is filled with this pattern, not the previous owner's information.

Files have a number of attributes, most of which are common to other systems. Principal attributes are the *name*, the *size*, the read/write/execute *access*, the *security level*, the *time of last reference*, the *time of last charge*, the *trust level*, the *load length*, the *type* (private, public or political), the *owner*, a possible *free* use, and the *active I/O count*. The *time of last reference* is used to tell which files have been unused long enough to be purged. The *time of last charge* is used by the accounting task that charges users for their disk space. *Trust levels* are used to give utility routines various privileges that should not be available to user programs. The *load length* tells the system how much of a file actually needs to be placed in memory to begin execution. It is common to place an executable image plus related files into one large file, and this attribute saves the system from having to load the entire file into memory and then discover that only part of it is the executable image. Some utilities are designated *free* so that bankrupt users can still execute them at a normal priority. The *active I/O count* is the total number of programs that have the file in use. A file cannot be destroyed or given away until its active I/O count is zero.

Running programs

A program requires space in memory not only for code and data storage but also for I/O buffers. The space required for the latter two sometimes cannot be determined until run time because it is problem dependent. CTSS has two system calls for supporting this requirement. One is for creating a new (larger) dropfile, and the other is for expanding (or contracting) memory field length. The FORTRAN I/O library automatically takes care of field length expansion and I/O buffer assignments for the casual user. For the majority of users who require nothing more complicated than the enlargement of blank common at run-time, we provide a library routine they can call that takes care of creating a larger dropfile, expanding field length, and moving the I/O buffers to the end of the new field length. Our normal layout is to place dynamically created I/O buffers beyond the last memory address that the program ought to be using. There is no need for the user to tell CTSS the maximum field length a program will need before it starts running. As the running program changes its size, it will be written to disk. Its new size then becomes known to the program scheduler, whose job is to fit programs into memory.

Another useful capability of programs running under CTSS is that they can initialize and run other programs. The first is called the *controller* of the second, and the second is called the *controllee* of the first. The designations of controller and controllee are relative. One can create a chain that is up to ten levels deep. A program in the middle is the controller of the program beneath it and the controllee of the program above it. Currently, only one member of such a chain is allowed to be in memory at a time. Note that each member of such a chain is a self-contained program with its own dropfile. The simplest case is a chain with only one program. Its controller is considered to be the terminal. Although it is seldom that a user needs to write a program that will run another program as a controllee, we have placed routines to do so in the FORTRAN I/O library. It is generally utility routines that use this capability. CTSS does not permit a program to have multiple controllees nor does it permit controllees in a chain to run asynchronously.

The mechanism for determining which program in a chain is to go into memory is the set of system calls for passing messages (of up to 512 characters) between programs. The four fundamental system calls are to send/receive a message to/from a controller/controllee. When a program sends a message, it is dropped from memory, and the controller or controllee to which it sent the message is scheduled to run. Likewise when a program asks to receive a message, it is dropped from memory (if there is no message ready) and the program (controller or controllee) from which the message is sought is scheduled to run. (In the future, we will remove the restriction that only one member of a chain can be in memory.) It is clear that a deadlock can develop if every program asks for a message and none sends one. Either a controller–controllee pair must understand how the other will behave or programs must be written in a defensive way (using special options in system calls) to avoid potential deadlocks. There is, of course, no problem when a single program is run from a terminal. It can only send or receive messages from its controller since it has no controllee. In this case, sending a message to the controller causes the message to print on the terminal. If the program asks for a message from the controller and there is none, it can choose to do without (*i.e.* resume running) or wait (be dropped from memory) until the user types an input message.

Thus far we have described message passing between programs that are in a controller–controllee chain. This means all the programs are running on the same suffix on the same user number. CTSS also has a way for two programs on different suffixes (or even two user numbers) to exchange messages. A program can make a system call asking to become a *process*. The system enters the program in a pool of known processes and assigns it a process number. The program can then use *interprocess communication* (IPC) to send and receive messages to and from other processes. Thus a process is nothing more than an executing program that has been given an identification number (its process number) and the privilege to send and receive messages to and from other processes. Generally, two processes need to know something about each other's behaviour in order to communicate successfully. Certain system service programs have pre-assigned process numbers. Non-privileged programs are simply given the next available number in a cyclically used sequence of numbers. There are system calls for determining the dropfile names, user numbers and process numbers of other processes on the system. The destination for an IPC message must have a host number as well as process number. This allows the Cray-1 to be used as one of many hosts in a network. Since the system service programs have pre-determined process numbers, they can communicate with each other across mainframes. Programs in separate mainframes must have a certain *trust level* in order to communicate. Since user programs are not normally trusted, they can communicate with processes only on the same host. IPC is used primarily by the batch processing subsystem (because it is distributed over several user numbers) and the permanent file system (because it is distributed over several machines). We have recently started using IPC to collect usage information on selected library subroutines. The instrumented routines send an IPC message to a collecting process giving the routine's name. We can then tell which of the (instrumented) routines in a library are actually being used. The IPC message facility is primitive (compared, say, to DEMOS[14]), but we have found it adequate for our needs.

Programs can be in any one of a number of states that the user can observe. The user can type CTRL-e, lower case 's', and carriage-return to ask CTSS the state of the currently running program. The system responds with a three character state abbreviation and the dropfile name of the currently running program. (The latter piece of information is important in determining which program in a controller–controllee chain is running.) Six states are commonly seen: waiting to get into memory, in memory waiting to run, waiting for an incoming message (usually from a controller), waiting to send a message (usually to a slow device like a terminal), waiting for an incoming IPC message, and suspending voluntarily (usually to wait for some programmer defined event to occur).

CTSS supports one level of user program interrupt. A program can tell the system it wishes to be interrupted when a message (*e.g.* from the terminal) arrives, at the end of a prescribed time interval, or when its outstanding I/O completes. CTSS will preserve the program's registers at the point of interrupt and transfer control to the (previously specified) address for processing the interrupt. Since most software on the Cray-1 is not re-entrant, it is important that none of the program's computing at *interrupt level* disturb return addresses or variables that are needed after the return from interrupt level. One can return from interrupt level either by resuming the computation (and registers) in progress at the moment of interrupt or by declaring the previous computation to be abandoned and designating the current state no longer to be at interrupt level. In either case, there is a system call for returning from interrupt level.

The message interrupt is useful for programs that do not want to ask and wait for a message from the terminal, but are willing to change whatever they are doing if the user types something. The most common use is when a program (*e.g.* a text editor) is sending a lot of information to the terminal. If the users tire of seeing the output, they should be able to type something that causes the program to stop and ask for another command. It would be inappropriate to abort the program just because it was printing too much.

The time interrupt has been used by an interesting set of subroutines that help in optimizing programming. The user calls one routine to start the package and another to terminate. In between, the user does usual computing. The package causes an interrupt to occur every four milliseconds at which time it samples the program counter (address of the current instruction) for the regular level of computing. The samples are dumped into a file so that another program with the help of the user program's symbol table can construct a histogram showing where the user spent the most time. The histogram has subroutine names and statement labels, which facilitates user optimization of the program.

The CTSS charging algorithm charges users for CPU, I/O and memory residence times. All three times are multiplied by associated weighting factors and by the program's bid priority before being deducted from the user's bank account. The various weighting factors for the individual components are chosen in order to reach a compromise between several requirements. One is that the separate charges for CPU, memory, and I/O should reflect the relative cost of the resources. Second is that the total charges for all users without regard for priority should approximate the wall clock time. Third is that the factors should encourage users to configure their programs efficiently. The algorithm is always subject to fine tuning, so we describe it only qualitatively.

The CPU charge is generally less than the real CPU time since we expect I/O and memory charges to bring the total charge time up to wall clock time. Programs using a large amount of memory are given a larger discount than small programs in their CPU charge because the large program also incurs a large memory charge, there is only one CPU, and while it is being used by a small program, much of the memory is wasted, and one presumably buys a Cray-1 for running large programs. The I/O charge is a fixed fraction of the real channel time. Since programs can compute and perform I/O simultaneously, they can incur CPU and I/O charges simultaneously. The memory charge is proportional to the amount of memory used as well as the length of time it is used. Since a program cannot use the CPU or perform I/O without being in memory, it must pay for memory along with CPU or I/O. The purpose of a separate memory charge is to encourage users to overlap CPU and I/O usage and thereby reduce the total time spent in memory.

Program scheduling

An important system task is the scheduling of memory and CPU. Our goal is to give production programs most of the CPU, to give high priority interactive programs quick service, and to postpone the rest of the programs if necessary. We do consider priority[15–20] in scheduling programs; however our priorities take into account only the bid priority and memory size and otherwise do not change dynamically. We use a variable quantum[20–23] that depends on priority and size rather than on system load. Scheduling is pre-emptive;[19, 24] a program can be removed from memory after the expiration of its quantum. As in the p[25] system, CPU time is given out in slices smaller than a quantum. Finally, we use round-robin[19, 24] to service the programs in memory. Since our scheduling seems to be unusual (the closest procedure seems to be Schwetman's Algorithm 3[18]), we elaborate upon it.

Two separate system tasks are involved in memory scheduling—the program scheduler and the preemption scheduler. Roughly speaking, the program scheduler looks for the highest priority program in the wait queue that will fit into memory. It examines the *load priority* rather than the user's *bid priority*. The *load priority* is generally proportional to *bid priority*, but includes a slight bias to favour large programs (unlike systems[2, 19] that discriminate against large programs). Since memory is likely to become fragmented, it is easier to find space for a small program than a large program, and this bias partly compensates for the tendency to pick smaller programs of equal bid priority. Since the program scheduler picks programs that will fit in memory, it can bypass higher priority large programs. The pre-emption scheduler acts as a counterbalance by trying to evict lower priority programs from memory on behalf of the highest priority program in the wait queue. If several programs are at the maximum load priority, they will pre-empt each other in round-robin fashion.

Once a program is allowed into memory, it is given a guaranteed minimum *time slot*, which is roughly proportional to its size and bid priority. The *time slot* is like a *quantum* except that it counts real I/O channel time as well as real CPU time. Proportionality to priority enables the pre-emption scheduler to evict low-priority programs more quickly. A program does not receive its time slot all at once. The time slot is further divided into *time slices*. Time slices are also approximately proportional to program size and bid priority; however, various factors are chosen so the ratio of the largest to smallest time slice is much smaller than the ratio of the largest to smallest time slot. The system then serves all programs in memory in round-robin, giving each its appropriate time slice.

The time slice is necessary for interactive computing because highly interactive programs seldom use up their time slot before taking some action that removes them from memory. By moving the CPU from program to program on a scale of time slices rather than slot times, we take less wall-clock time to provide the interactive program the little service it needs and remove it from memory so that some other program might be loaded. On the other hand, the time slice is not an adequate quantum for a large, CPU intensive program. Such programs need to use up their slot time to compensate for the overhead of swapping them in and out of memory. This two level scheme allows CTSS to service small interactive programs and large production programs with only modest compromises in level of service. Unfortunately, we have no solution for large, high priority, interactive programs. This type of program tends to increase system overhead and cause deterioration of service to other programs, but we allow such programs because they (presumably) represent a legitimate use of the Cray-1.

The only mechanism that prevents users from bidding the maximum priority is the finite size of their bank accounts. Users are forced to be reasonable in their bid priorities to conserve their resources, but they can (and do) briefly bid high priorities when they truly need quick service. The priorities then guide CTSS into serving the most important programs at any moment.

Deadstarts

Another facility of CTSS is its range of deadstart and recovery capabilities.[26] The kinds of computers for which CTSS and its precursors were written were state-of-the-art supercomputers whose reliability was unknown. CTSS has inherited a number of levels of restart to cope with different situations. The highest level is truly a restart and not a deadstart. This happens after the Cray-1 is taken down for routine maintenance during which all files and running programs are preserved. The next step down is a *hot* start in which all files and running programs are saved except for the user program that was using the CPU. This kind of crash and deadstart is typically caused by an unrecoverable memory parity error in the user program. Next there are two kinds of *warm* start in which system tables are all recovered from back-up copies on disk and no running programs are restarted. (Their dropfiles remain available for a manual restart by the user.) A CPU failure affecting the handling of tables of running programs would require this kind of restart. A *lukewarm* start is like a warm start but first requires a restoration of the public files from tape. Finally, there is the *cold* deadstart, a truly catastrophic and seldom occurring event. This happens when some failure causes destruction of the file index or other tables to the point that even the back-up copies on disk are suspect. All user files are lost, and users have to retrieve copies from the (external) permanent file system. Public files might have to be restored from tape. (We make copies of public files once a week and political files twice a week.) One must remember that the CPU, memory, I/O channels and disks are all subject to failure. CTSS is alert for such failures. Not only does the system itself stop if something in its tables appears to be inconsistent, but the system runs hardware diagnostics every fifteen minutes (time-shared along with users' programs) to look for developing problems. CTSS can also inhibit file creation on a failing disk unit so that users can access their files while the unit is gradually dried up to be taken down for maintenance. We can also move files off of a disk unit that is to be taken down. When users destroy files, CTSS overwrites the files with pattern and removes from reuse any sectors where the write fails. CTSS, in short, looks out for problems, tries to protect users from problems, and salvages as much of the users' work as it can should it finally succumb.

Other facilities

CTSS and its utility routines provide most of the services commonly available on time-sharing systems, including batch processing, symbolic debugging, electronic mail, electronic bulletin boards, online documentation,[27] and real-time computer conferencing. The online documentation contains more than the usual help packages; an interactive utility provides access to (currently) 223 write-ups comprising 17,218 pages. Recently developed services include a screen editor and a graphics design utility that use a personal microcomputer as an intelligent terminal. Co-operating programs run simultaneously on the Cray-1 and the microcomputer, the latter carrying out trivial commands that modify the display, and notifying the former of the corresponding changes to make to a file. This combination allows the Cray-1 to provide some of the highly interactive functions for which its architecture is inefficient.

STATISTICS

It is impossible to show that CTSS provides substantial benefits or convenience beyond the vendor's operating system since there are too few users sufficiently familiar with both systems to conduct a representative survey. There are, however, at least two indirect ways to gauge the success of CTSS.

One way is to look at the program mix on the system and measure how the CPU is distributed. Our measurements show that at night the system overhead fluctuates between 1 and 4 per cent. During periods of heavy interactive use, the overhead fluctuates between 2 and 8 per cent with an average of about 5 per cent. In examining the load on the system, we use percentages of the CPU that remain for users after deducting the overhead. We partition all programs into two sets—those that use no more than 10 seconds and no more than 100,000 (decimal) 64-bit words, and those exceeding 10 seconds or exceeding 100,000 words of memory. The first set is clearly the interactive type of program. It comprises about 94 per cent of the total. The nature of the second set is not so clear. It includes all the production jobs, but may also contain some very large (hence sluggishly interactive) programs. This 6 per cent of the programs uses about 85 per cent of the CPU on weekdays and 90 per cent on weekends. If we count only those jobs exceeding 150,000 words regardless of execution time, we find on weekdays that they represent 3 per cent of all programs and use 50 per cent of the CPU. On weekends, they are 2 per cent of all programs and use 60 per cent of the CPU. Clearly, the large or long running programs are getting most of the resources, but just as clearly, the overwhelming number of programs are both short and small. We take this as indirect evidence that the Cray-1 can and is being used heavily for interactive computing.

Incidentally, even after allowing for repair, maintenance, and system development time, we generally can make the Cray-1 available to users for 94 per cent of the wall-clock time each month. Typical idle times are less than 0·1 per cent. This means there is almost always some program waiting to run.

A second way to gauge the success of CTSS is to see if any other Cray-1 owners are interested in using it. Not surprisingly, the local computer centre at LLNL uses CTSS on their Cray-1 computers. The Los Alamos National Laboratory began switching from their locally written multi-programming batch system[14] to CTSS in 1979 and completed the transition on all their Cray-1 by the end of 1980. The Air Force Weapons Laboratory at Kirtland AFB switched from the vendor system to CTSS in 1982. Two other owners have also been interested enough to deadstart CTSS on their Cray-1s to see how it works. It is a serious and complicated decision to change operating systems, and we doubt anyone would switch to CTSS unless they saw substantial advantages in doing so.

To give an idea of the size and usage of a Cray-1 with CTSS, the following information summarizes the use of the smaller of our two Cray-1 computers.

2,000	total number of users
254	maximum simultaneous users
1,048,576	(64-bit) words of memory
700,000	maximum words of memory for any one program
6600	million bytes of disk space

We do not count the number of user programs executed, but we do record the use of generally available utility routines. Average daily use (24-hour day, 7-day week basis) is given for selected categories of programs.

539	Fortran compilers
2044	text editors
229	electronic mail utilities
189	interactive debugging utility
415	batch jobs (includes direct execution of COSMOS by users)

OBSERVATIONS AND CONCLUSIONS

A supercomputer can effectively time-share even though its CPU may be much faster than its I/O subsystem provided it has enough memory to use as I/O buffers for programs being swapped to and from disk. Even then, memory must be carefully scheduled. Imbalance between CPU and I/O performance may mean that it is impractical to run programs that must be swapped in for character-at-a-time input. Virtually all CTSS utilities are designed for line-at-a-time input.

For a supercomputer, time-sharing augments, but cannot replace, batch processing. Scheduling based on user specified priorities allows the system to provide quicker service to more urgent (presumably interactive) programs at the expense of other (presumably batch) programs.

The combination of a large machine and time-sharing leads to many users and many files; thus a system crash can disrupt a massive amount of work. It is most important that the system have several levels of recovery in order to minimize the damage from equipment (and power) failures.

A supercomputer system can be written in a high-level language without serious loss of efficiency. Not only is the system more easily maintained, but the increased transportability makes conversion of the system to other computers feasible.

Most of the interactive facilities that make minicomputers convenient to use can also be implemented on a Cray-1. Dispensing with interactive front-end machines is advantageous to users who are uninterested in learning how to use additional operating systems. The development and postprocessing phases (which are usually interactive) of their programming are then smoothly integrated with the production phase.

Ten to fifteen years ago, computer scientists debated the merits of batch processing versus time-sharing. For the most part, the question has been resolved in favour of time-sharing, since such systems could also support batch processing. Supercomputers seem to have been exempted from this reasoning because they were perceived as very expensive and special tools. It then followed that they should be treated as computational engines to be accessed through general-purpose, front-end computers. CTSS demonstrates that the Cray-1 (as LTSS did for the CDC 6600 and CDC 7600) is usable as a general-purpose scientific computer, capable of supporting all the auxiliary computing that comes along with large production programs. We conclude that supercomputers may be used effectively in ways never envisioned by their manufacturers and that purchasers of such machines must uncover and exploit such latent powers.

ACKNOWLEDGEMENTS

The following people contributed information or comments used in this paper: Lawrence Berdahl, Pamela Farnstrom, Dieter Fuss, Terry Girill, Bruce Griffing, Barry Howard, Walter Hutchinson, Morris Jette, Clement Luk, Cynthia Phillips, Judith Sarna, George Spix, Ronald Sykora and Bing Young. We also acknowledge the work done by the authors of References 4, 5, 7 and 8 without which there could not have been a CTSS.

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

REFERENCES

1. N. Lincoln, 'Supercomputers = colossal computations + enormous expectations + renowned risk', *Computer*, **16** (5), 38–47 (1983).
2. F. J. Corbato, M. M. Daggett and R. C. Daley, 'An experimental time-sharing system', *Proc. SJCC*, 335–344 (1962).
3. F. J. Corbato and V. A. Vyssotsky, 'Introduction and overview of the Multics system', *Proc. FJCC*, 185–196 (1965).
4. P. J. Du Bois and J. T. Martin, 'The Lrltran language as used in the Frost and Floe time-sharing operating systems', *SIGPLAN Notices*, **6** (9), 92–104 (1971).
5. M. D. Richards, H. G. Coverston, P. J. Du Bois, P. E. Lund, N. A. Storch and C. L. Streeter, 'CDC 6600 system calls and i/o requests', *LTSS Chapter 9, edition 3*, Lawrence Livermore National Laboratory, 1969.
6. J. J. Horning, 'Yes, high level languages should be used to write systems software', *Proc. Natl ACM*, 206–208 (1975).
7. G. G. Sutherland, H. G. Coverston, P. J. Du Bois, D. R. Emery, D. A. Kent, P. E. Lund, G. H. Powles, F. D. Storch, L. O. Tennant and D. L. Von Buskirk, 'Livermore time-sharing system for the CDC 7600', *Oper. Syst. Rev.*, **5** (2), 6–26 (1971).
8. P. J. Du Bois, H. G. Coverston, F. D. Storch and G. H. Powles, 'CDC 7600 system calls and i/o requests', *LTSS Chapter 10, edition 6*, Lawrence Livermore National Laboratory, 1974.
9. R. M. Russell, 'The Cray-1 computer system', *Communs Ass. comput. Mach.*, **21**, 63–72 (1978).
10. D. M. Ritchie, 'Unix time-sharing system: a retrospective', *Bell Syst. Tech. J.*, **57**, 1947–1969 (1978).
11. R. C. Daley and P. G. Neumann, 'A general-purpose file system for secondary storage', *Proc. FJCC*, 213–229 (1965).
12. M. L. Powell, 'The Demos file system', *Oper. Syst. Rev.*, **11**, 33–42 (1977).
13. D. M. Ritchie and K. Thompson, 'The Unix time-sharing system', *Bell Syst. Tech. J.*, **57**, 1905–1929 (1978).
14. F. Baskett, J. H. Howard and J. T. Montague, 'Task communication in Demos', *Oper. Syst. Rev.*, **11**, 23–31 (1977).
15. J. M. Babad, 'A generalized multi-entrance time-sharing priority queue', *J. Ass. comput. Mach.*, **22**, 231–247 (1975).
16. C. G. Bell, A. Kotok, T. N. Hastings and R. Hill, 'The evolution of the DECsystem-10', *Communs Ass. comput. Mach.*, **21**, 44–63 (1978).
17. B. R. Borgerson, M. L. Hanson and P. A. Hartley, 'The evolution of the Sperry Univac 1100 series: a history, analysis, and projection', *Communs Ass. comput. Mach.*, **21**, 25–43 (1978).
18. H. D. Schwetman, 'Job scheduling in multiprogrammed computer systems', *Software—Practice and Experience*, **8**, 241–255 (1978).
19. K. Thompson, 'Unix time-sharing system: Unix implementation', *Bell Syst. Tech. J.*, **57**, 1931–1946 (1978).
20. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson, 'Tenex, a paged time-sharing system for the PDP-10', *Communs Ass. comput. Mach.*, **15**, 135–143 (1972).
21. U. N. Bhat and R. E. Nance, 'An evaluation of CPU efficiency under dynamic quantum allocation', *J. Ass. comput. Mach.*, **26**, 761–778 (1979).
22. M. A. Kartsev, 'The M-10 computer', *Soviet Phys. Dokl.*, **24**, 149–151 (1979).
23. D. Potier, E. Gelenbe and J. Lenfant, 'Adaptive allocation of central processing unit quanta', *J. Ass. comput. Mach.*, **23**, 97–102 (1976).
24. J. C. Browne, J. Lan and F. Baskett, 'The interaction of multi-programming job scheduling and CPU scheduling' *Proc. FJCC*, 13–21 (1972).
25. A. Lindgard, 'p—a timesharing operating system for laboratory automation', *Software—Practice and Experience*, **9**, 971–986 (1979).
26. C. H. Luk and F. D. Storch, 'Exception-handling for a time sharing system', *Joint Fault Tolerant Computing and Software Engineering Workshop*, December 1978.
27. T. R. Girill and C. H. Luk, 'Document: a comparative study of an interactive, online solution to four documentation problems', *Communs Ass. comput. Mach.*, **26**, 328–337 (1983).