A "NonStop"* Operating System

Joel F. Bartlett
Tandem Computers Inc.
19333 Vallco Parkway
Cupertino, California

ABSTRACT

The Tandem/16 computer system is an attempt at providing a general-purpose, multiple-computer system which is at least one order of magnitude more reliable than conventional commercial offerings. Through software abstractions a multiple-computer structure, desirable for failure tolerance, is transformed into something approaching a symmetric multiprocessor, desirable for programming ease. Section 1 of this paper provides an overview of the hardware structure. In section 2 are found the design goals for the operating system, "Guardian". Section 3 provides a bottom-up view of Guardian. The user-level interface is then discussed in section 4. Section 5 provides an introduction to the mechanism used to provide failure tolerance at the application level and to application structuring. Finally, section 6 contains a few comments on system reliability and implementation.

1. INTRODUCTION

1.1 Background

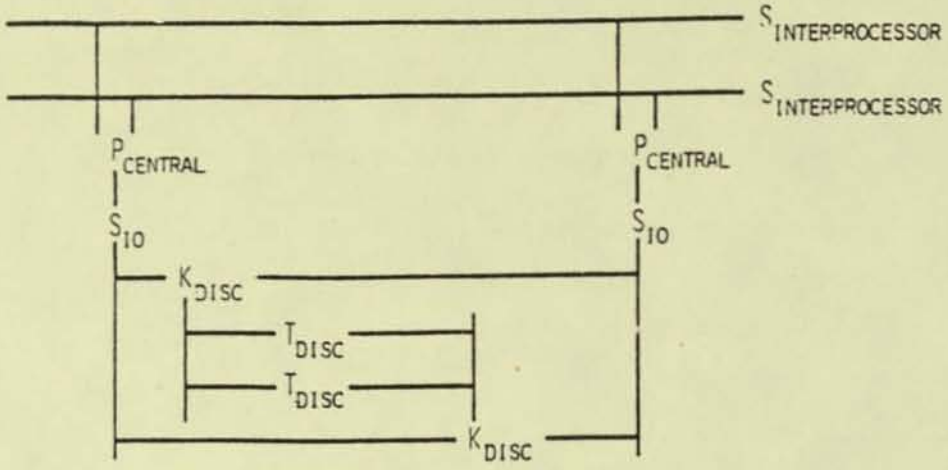
On-line computer processing has become a way of life for many businesses. As they make the transition from manual or batch methods to on-line systems, they become increasingly vulnerable to computer failures. Whereas in a batch system the direct costs of a failure might simply be increased overtime for the operations staff, a failure of an on-line system results in immediate business losses.

1.2 System Overview

The Tandem/16 (1,2) was designed to provide a system for on-line applications that would be significantly more reliable than currently available commercial computer systems. The hardware structure consists of multiple processor modules interconnected by redundant interprocessor buses. A PMS (3) definition of the hardware is found in Figure 1.

Each processor has its own power supply, memory, and I/O channel and is connected to all other processors by redundant interprocessor buses. Each I/O controller is redundantly powered and connected to two different I/O channels. As a result, any interprocessor bus failure does not affect the ability of a processor to communicate with any other processor. The failure of an I/O channel or of a processor does not cause the loss of an I/O device. Likewise, the failure of a module (processor or I/O controller) does not disable any other module or disable any inter-module communication. Finally, certain I/O devices such as disc drives may be connected to two different I/O controllers, and disc drives may in turn be duplicated such that the failure of an I/O controller or disc drive will not result in loss of data.

* "NonStop" is a trademark of Tandem Computers Inc.



Hardware Structure
Figure 1

The system is not a true multiprocessor (4), but rather a "multiple computer" system. The multiple computer approach is preferable for several reasons. First, since no module is shared by the entire system, it increases the system's reliability. Second, a multiple computer system does not require the complex hardware needed to handle multiple access paths to a common memory. In smaller systems, the cost of such a multiported memory is undesirable; and in larger systems, performance suffers because of memory access interference.

On-line repair is as necessary as reliability in assuring system availability. The modular structure of the Tandem/16 system allows processors, I/O controllers, or buses to be repaired or replaced while the rest of the system continues to operate. Once repaired, they may then be reintegrated into the system.

The system structure allows a wide range of system sizes to be supported. As many as sixteen processors, each with up to 512k bytes of memory, may be connected into one system. Each processor may also have up to 256 I/O devices connected to it. This provides for tremendous growth of application programs and processing loads without the requirement that the application be reimplemented on a larger system with a different architecture.

Finally, the system is meant to provide a general solution to the problem of providing a failure-tolerant, on-line environment suitable for commercial use. As such, the system supports conventional programming languages and peripherals and is oriented toward providing large numbers of terminals with access to large data bases.

2. SYSTEM DESIGN GOALS

2.1 Integrated Hardware/Software Design

The Tandem/16 system was designed to solve a specific problem. This problem was not stated in terms of hardware and software requirements, but rather in terms of system requirements. The hardware and software designs then proceeded in tandem to provide a unified solution. The hardware design concerned itself with the contents of each module, their interconnections to the common buses, and error detection and correction within modules and on the communication paths. The software design was given the problem of control; that is, selection of which modules to use and which buses to use to communicate with them. Furthermore, as errors are detected, it was the responsibility of the software to control recovery actions.

2.2 Operating System Design Goals

The first and foremost goal of the operating system, Guardian, was to provide a failure-tolerant system. This translated into the following design "axioms":

- the operating system should be able to remain operational after any single detected module or bus failure
- the operating system should allow any module or bus to be repaired on-line and then reintegrated into the system.

- the operating system should be implemented in a reliable manner. Increased reliability provided by the hardware architecture must not be negated by software problems.

A second set of requirements came from the great numbers and sizes of hardware configurations that are possible:

- the operating system should support all possible hardware configurations, ranging from a two-processor, discless system through a sixteen-processor system with billions of bytes of disc storage.

- the operating system should hide the physical configuration as much as possible such that applications could be written to run on a great variety of system configurations.

3. OPERATING SYSTEM STRUCTURE

To satisfy these requirements, the operating system was designed to have the appearance of a true multiprocessor at the user level. The design of the system was strongly influenced by Dijkstra's work on the "THE" system (5), and Brinch Hansen's implementation of an operating system nucleus for a single-processor system (6). The primary abstractions are processes, which do work, and messages, which allow interprocess communication.

3.1 Processes

At the lowest level of the system is the basic hardware as earlier described. It provides the capability for redundant modules, i.e. I/O controllers, I/O devices, and processor modules consisting of a processor, memory, and a power supply. These redundant modules are in turn interconnected by redundant buses. Error detection is provided on all communication paths and error correction is provided within each processor's memory. The hardware does not concern itself with the selection of communication paths or the assignment of tasks to specific modules.

The first abstraction provided is that of the process. Each processor module may have one or more processes residing in it. A process is initially created in a specific processor and may not execute in another processor. Each process has an execution priority assigned to it. Processor time is allocated on a strict priority basis to the highest priority ready process.

Process synchronization primitives include "counting semaphores" and process local "event" flags. Semaphore operations are performed via the functions PSEM and VSEM, corresponding to Dijkstra's P and V operations. Semaphores may only be used for synchronization between processes within the same processor. They are typically used to control access to resources such as resident memory buffers, message control blocks, and I/O controllers.

When certain low-level actions such as device interrupts, processor power-on, message completion or message arrival occur, they result in "event" flags being set for the appropriate process. A process may wait for one or more events to occur via the function WAIT. The process is activated as soon as the first WAITed for event occurs. Events are signaled via the function AWAKE. Event signals are queued using a "wake up waiting" mechanism so that they are not lost if the event is signaled when the process is not waiting on it. Like semaphores, event signals may not be passed between processors. Event flags are predefined for eight different events and may not be redefined.

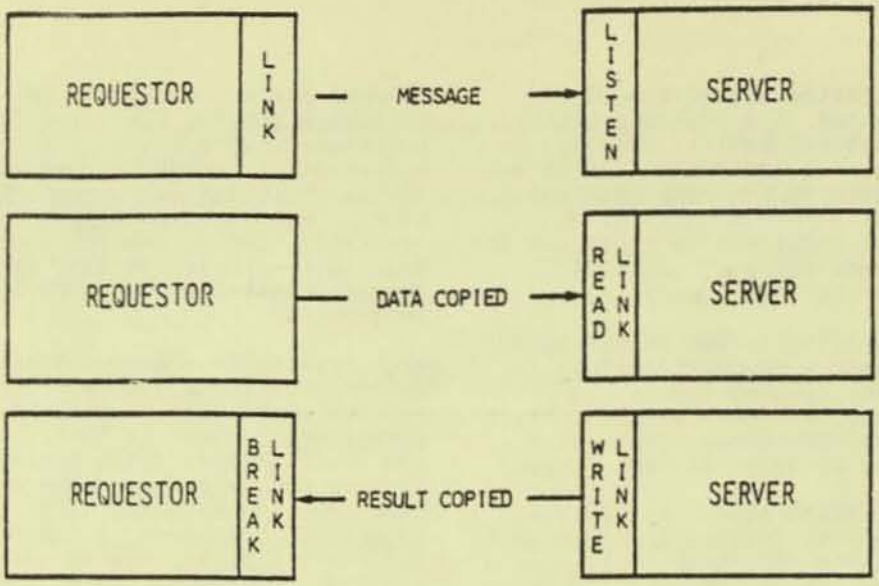
When a process blocks itself to wait for some event to occur or for a semaphore to be allocated to it, it may specifiy a maximum time to block. If the time limit expires and the event has not occurred or the resource has not been obtained, then the process will continue execution but an error condition will be returned to it. This timeout allows "watch dog" timers to be easily placed on device interrupts or on resource allocations where a failure may occur.

Each process in the system has a unique identifier or "processid" in the form: <cpu #,process #>, which allows it to be referenced on a system-wide basis. This leads to the next abstraction, the message system, which provides a processor-independent, failure-tolerant method for interprocess communication.

3.2 Messages

The message system provides five primitive operations which can be illustrated in the context of a process making a request to some server process, Figure 2. The process' request for service will send a message to the appropriate server process via the procedure LINK. The message will consist of parameters denoting the type of request and any needed data. The message will be queued for the server process, setting an event flag, and then the requestor process may continue executing.

When the server process wishes to check for any messages, it calls LISTEN. LISTEN



Message System Primitive Operations
Figure 2

returns the first message queued or an indication that no messages are queued. The server process will then obtain a copy of the requestor's data by calling the procedure READLINK.

Next, the server process will process the request. The status of the operation and any result will then be returned by the WRITELINK procedure, which will signal the requestor process via another event flag. Finally, the requestor process will complete its end of the transaction by calling BREAKLINK.

A communications protocol was defined for the interprocessor buses that would tolerate any single bus error during the execution of any message system primitive. This design assures that a communications failure will occur if and only if the sender or receiver processes or their processors fail. Any bus errors which occur during a message system operation will be automatically corrected in a manner transparent to the communicating processes and logged on the system console. The interprocessor buses are not used for communication between processes in the same processor, which can be done faster in memory. However, the processes involved in the message transfer are unable to detect this difference.

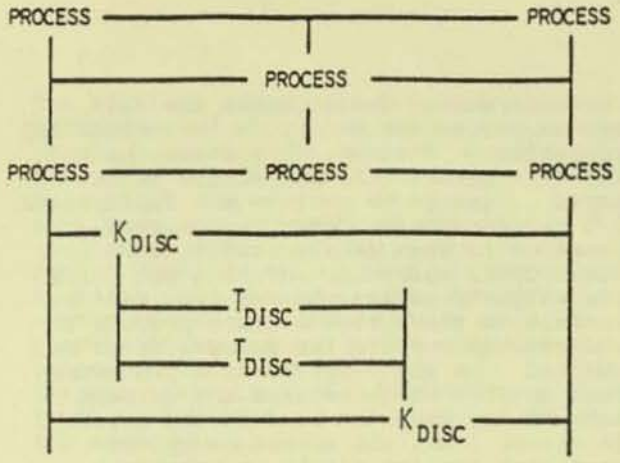
The message system is designed such that resources needed for message transmission (control blocks) are obtained at the start of a message transfer request. Once LINK has been successfully completed, both processes are assured that sufficient resources are in hand to be able to complete the message transfer. Furthermore, a process may reserve control blocks to guarantee that it will always be able to send messages to process a request that it picks up from its message queue. Such resource controls assure that deadlocks can be prevented in complex producer/consumer interactions, if the programmer correctly analyzes and anticipates potential deadlocks within the application.

3.3 Process-pairs

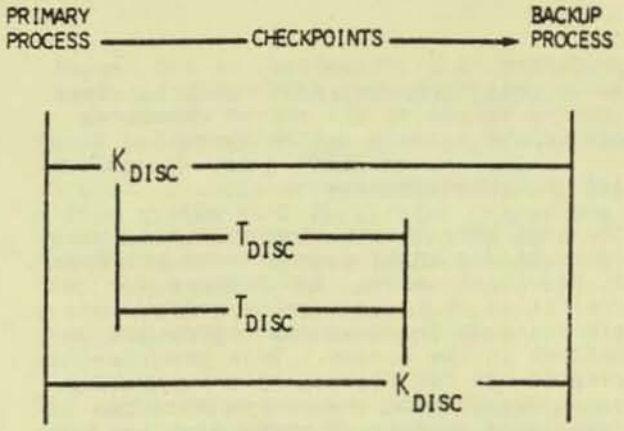
With the implementation of processes and messages, the system is no longer seen as separate modules. Instead, the system can be viewed as a set of processes which may interact via messages in any arbitrary manner, as shown in Figure 3.

By defining messages as the only legitimate method for process-to-process interaction, interprocess communication is not limited by the multiple-computer organization of the system. The system then starts to take on the appearance of a true multiprocessor. Processor boundaries have been blurred, but I/O devices are still not accessible to all processes.

System-wide access to I/O devices is provided by the mechanism of "process-pairs". An I/O process-pair consists of two cooperating processes located in two different processors that control a particular I/O device. One of the processes will be considered the "primary" and one will be considered the "backup". The primary process handles requests sent to it and controls the I/O device. When a request for an operation such as a file



System Structure After the Addition of Processes and Messages
Figure 3



Process-pair for a Redundantly-Recorded Disc Volume
Figure 4

open or close occurs, the primary will send this information to the backup process via the message system. These "checkpoints" assure that the backup process will have all information needed to take over control of the device in the event of an I/O channel error or a failure of the primary process' processor. A process-pair for a redundantly-recorded disc volume is illustrated in Figure 4.

Because of the distributed nature of the system, it is not possible to provide a block of "driver" code that could be called directly to access the device. While potentially more efficient, such an approach would preclude access to every device in the system by every process in the system.

The I/O process-pair and associated I/O device(s) are known by a logical device name such as "$DISC1" or by a logical device number rather than by the processid of either process. I/O device names are mapped to the appropriate processes via the logical device table (LDT) in every processor, which supplies two processids for each device. A message request made on the basis of a device name or number results in the message being sent to the first process in the table. If the message cannot be sent or if the message is sent to the backup process, an error indication will be returned. The processid entries in the LDT will then be reversed and the message resent. Note two things: first, the error recovery can be done in an automatic manner; and second, the requestor is not concerned with what process actually handled the request. Error recovery cannot always be done automatically. For example, the primary process of a pair controlling a line printer fails while handling a request to print a line on a check. The application process would prefer to see the process failure as an error rather than have the request automatically retried, which might result in two checks being printed.

The two primitives, processes and messages, blur the boundaries between processors and provide a failure-tolerant method for interprocess communication. By defining a method of grouping processes (process-pairs), a mechanism for uniform access to an I/O device or other system-wide resource is provided. This access method is independent of the functions performed within the processes, their locations, or their implementations. Within the process-pair, the message system is used to checkpoint state changes so that the backup process may take over in the event of a failure. This checkpoint mechanism is in turn independent of all other processes and messages in the system.

The system structure can be summarized as follows. Guardian is constructed of processes which communicate using messages. Fault tolerance is provided by duplication of components in both the hardware and the software. Access to I/O devices is provided by process-pairs consisting of a primary process and a backup process. The primary process must checkpoint state information to the backup process so that the backup may take over on a failure. Requests to these devices are routed using the logical device name or number so that the request is always routed to the current primary process. The result is a set of primitives and protocols which allow recovery and continued processing in spite of bus,

processor, I/O controller, or I/O device failures. Furthermore, these primitives provide access to all system resources from every process in the system.

3.4 System Processes

The next step in structuring the system comes in assigning functions to processes. As previously shown, I/O devices are controlled by process-pairs. Another process-pair known as the "operator" is present in the system. This pair is responsible for formatting and printing error messages on the system console. Here is an example of where Guardian has not followed a strict level structure. The operator makes requests to a terminal process to print the messages, yet the terminal process wishes to send messages to the operator to report I/O channel errors. An infinite cycle is prevented by having the terminal process not send messages for errors on the operator terminal and having I/O processes never wait for message completions when sending errors to the operator. While it may be preferable to prevent cycles of any type in system design, they have been allowed in Guardian when it can be shown that they will terminate. The ability to reserve message control blocks assures that no cycle will be blocked because of resource problems.

Each processor has a "system monitor" process which handles such functions as process creation and deletion, setting time of day, and processor failure and reload cleanup operations.

A memory management process is also resident in each processor. This process is responsible for allocating a page of physical memory and then sending messages to the appropriate disc processes to do the actual disc I/O. Pages are brought in on a demand basis and pages to overlay are selected on a "least recently used" basis over the entire memory of the processor.

The choice of relatively unsophisticated algorithms for scheduling and memory management was a result of the fact that the system was not intended to be a general-purpose timeshare system. Rather, it was to be a system which supported multiple processes and terminals in an extremely flexible manner.

3.5 Application Process Interface

Above the process and communication structure there exists a library of procedures which are used to access system resources. These procedures run in the calling process' environment and may or may not send messages to other processes in the system. For example, the file system procedures do not do the actual I/O operations. Instead, they check the caller's parameters, and if all is in order a message is sent to the appropriate I/O process-pair. Likewise, process creation is seen as a procedure call to NEWPROCESS, which does nothing but check the caller's parameters and then send a message to the system monitor process in the processor where the process is to be created. On the other hand, a procedure such as TIME which returns the current time of day does not send any messages. In either case, the access to system resources appears simply as procedure calls, effectively hiding the process structure, message system, hardware organization, and associated failure recovery mechanisms.

3.6 Initialization and Processor Reload

System initialization starts with one processor being cold loaded from some disc on the system. The load file contains a memory image of the operating system resident code and data, with all system processes in existence and at their initial states. The system monitor process then creates a command interpreter process.

Guardian may be brought up even though a processor or peripheral device is down. This is possible because operating system disc images may be kept on multiple disc drives, I/O controllers may be accessed by two different processors, and the terminal that has the initial command interpreter on it is selected by using the processor's switch register.

After a cold load, the system logically consists of one processor and any peripherals attached to it. More processors and peripherals may be added to the system via the command interpreter command:

```
:RELOAD  1,$DISC
```

This command will read the disc image for processor 1 from the disc $DISC and send it over either interprocessor bus to processor 1. Once it is loaded, all processes residing in other processors in the system will be notified that processor 1 is up.

This command is also used to reload a processor after it has been repaired. Guardian does not differentiate between an initial load of a processor and a later reload. In each case, resources are being logically added to the system and processes must be notified so that they may make use of them.

The previous example of a reload message being sent to all processes is an example of how functions are split in Guardian. A mechanism is provided for informing a process of a system status change. It may then take some unspecified action (including doing nothing). Similarly, a system power-on simply sets the PON event flag for all processes. The operating system kernel must only insure that the process structure and message system are correctly saved and restored. It is then the responsibility of individual processes to do such things as reinitialize their I/O controllers.

3.7 Operating System Error Detection

Besides the hardware-provided single error detection and correction on memory, and single error detection on the interprocessor and I/O buses, additional software error checks are provided. The first of these is the detection of a down processor. Every second, each processor in the system sends a special "I'm alive" message over each bus to all processors in the system. Every two seconds, each processor checks to see that it has received one of these messages from each processor. If a message has not been received, then it assumes that that processor is down.

Additionally, the operating system makes checks on the correctness of data structures such as linked lists when operations are done on them. Any processor detecting such an error will halt.

All I/O interrupts are bracketed by a "watch dog" timer such that the system will not hang up if an I/O operation does not complete with the expected interrupt. If an I/O bus error occurs then the backup process will take over control of the device using the second I/O bus.

As previously noted, the interprocessor bus protocol is designed to correct single bus errors. In addition to this, extensive checks are made on the control information received over the buses to verify that it is consistent with the state of the receiving processor.

Power-fail/automatic restart to provided within each processor. A power-failure is detected independently by each processor module and as a result is not a system-wide, synchronous event. The system was designed to recover from either a complete system power-fail, or a transient which will cause some of the processors to power-fail and then immediately restart.

4. USER-LEVEL SYSTEM INTERFACE

Tools are provided for interactive program development using COBOL or a block-structured implementation lapguage, T/TAL. A file system with facilities comparable to or exceeding those offered by other "midi" computer systems allows access to disc files and other I/O devices. Process creation, intercommunication, and checkpointing primitives are also implemented.

The application process level facilities and the interactive program development tools have been heavily influenced by the HP 3000 (7) and by UNIX (8).

4.1 Interactive System Access

General-purpose, interactive access to the system is provided by the command interpreter, COMINT, similar in many ways to the Shell of UNIX. Normally a command interpreter is run interactively from a terminal, but commands may be read from any type of file. The command interpreter is seen by the operating system as simply another type of application process.

Commands are read from the terminal, prompted by a colon (":"):

```
: command / process parameters / arguments
```

If the command is recognized, it will be directly executed. A command of this type is:

```
:LOGON  SOFTWARE.JOEL
```

which is used to gain access to the system. If the command is not recognized, then a process will be created using the program file "$SYSTEM.SYSTEM.command" and the arguments for the command will be sent to this new process. The command interpreter will then suspend itself until a message is received indicating that the process has stopped. If this process cannot be created, then an error message is printed. For example, the text editor is accessed by typing EDIT followed by any command string:

```
:EDIT  FILE
```

This will result in a process being created using the program file $SYSTEM.SYSTEM.EDIT and the command string, "FILE", being sent to it. Also a part of this command string message are the names of the files that are being used for input and output by the command interpreter. These are then used by the process for its input and output. If the previous command was typed at a terminal, the input and

output files would be the device name of the terminal. Alternative names for the input and output files may be specified. For example:

```
:EDIT   /IN COMMANDS/
```

will create an editor process and pass it the file name "COMMANDS" for the input file and the terminal's file name, the default, for the output file. Finally, the processor to use and the priority at which to run the process may also be specified:

```
:EDIT  /PRI 100, CPU 3/
```

This will create an editor process in processor three with a priority of 100.

Additional features allow multiple processes to be started from one command interpreter and allow the previously typed command line to be edited.

4.2 Programming Languages

Compilers have been implemented for two languages, T/TAL, and ANSI 74 COBOL. T/TAL is a block-structured implementation language. Its capabilities are similar to those offered by C on UNIX or SPL on the HP3000. All Tandem software is written in T/TAL as are most user applications.

Code generated by either compiler may be shared by multiple processes in the same processor. Both compilers generate an object file which may be immediately run without any intervening link edit operation. However, the object file also contains enough information so that an object editor, UPDATE, may combine the objects produced by several compilations or selectively replace procedures in an object file.

4.3 Tools

Program development tools include an interactive text editor, object file editor, text formatter, and interactive debugger. A screen generation program and access routines are provided to facilitate application interaction with page mode CRT terminals. File utilities exist which allow file backup and restore, file copying and dumping, and initial loading of key-sequenced files. A peripheral utility is provided to do such operations as disc formatting, disc track sparing, and mounting or demounting disc volumes.

4.4 Process Creation and Deletion

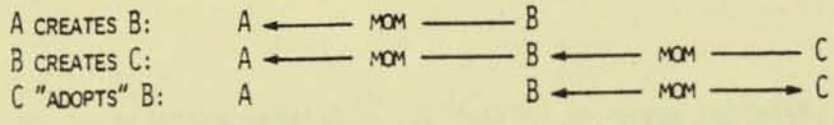
Processes are created by the command interpreter or by an application process call to the procedure NEWPROCESS. Parameters supplied include the name of the file holding the object code for the process, the processor number to use, and the priority at which to run the process. The parameters will be checked and then sent to the system monitor process in the appropriate processor. The system monitor will then create the process and return a "creationid" identifying the new process to the calling process. Part of this value is the processid previously defined, and the rest is the value of the processor clock at the time of process creation. The clock is kept as a 48 bit value which is the number of 10ms intervals since 12 a.m. on December 31, 1975, which assures that creationid's will be unique over the life of the system.

Processes are not grouped in classical ancestry trees. No process is considered subservient to any other process on the basis of parentage. Two processes, one created by the other, will be treated as equals by the system. When a process, A, creates another process, B, no record of B is attached to A. The only record kept is in process B where the creationid of A is saved. This creationid is known as B's "mom". When process B stops, process A is sent a stop message indicating that process B no longer exists. A process's mom is flexible and a process may adopt another process. For example, (Figure 5), process A creates process B. Process B in turn creates a cooperating process, C. Since C would like to know if B stops, C will adopt B.

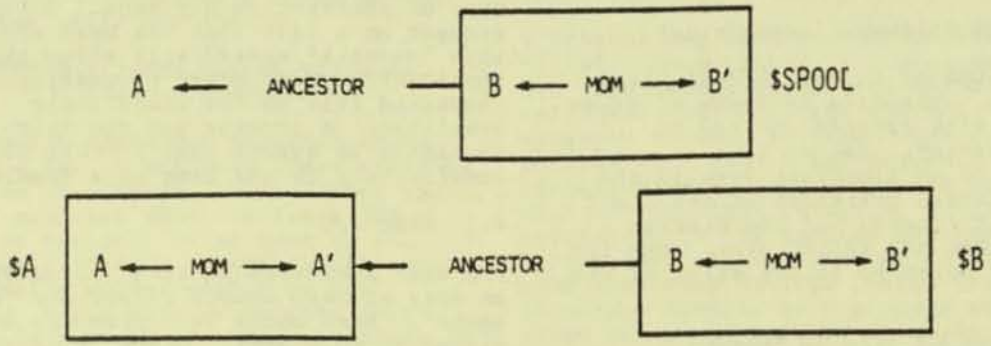
A process may stop itself or some other process by calling STOP. Process deletion is again a function of the system monitor process. Resources will be released and a stop message will be sent to the process' mom. If the mom process does not exist, then no message will be sent.

4.5 Application Process-pairs

The process-pair concept introduced earlier is a powerful method for making some resource available to all processes in the system in a fault-tolerant manner. It is extended to the application processes as follows. When a process is created via NEWPROCESS, a process-pair name may be supplied. The creationid returned for this process consists of the processid and the process name rather than the processor clock value. For example, (Figure 6), process A wishes to create a process with the name "$SPOOL". Once B has been created, any process in the system may send a message to that process via the name "$SPOOL".



Flexible Process Relationships
Figure 5



Application Process-pairs
Figure 6

Process B may now wish to create a process B' in another processor to be its backup. B would then call NEWPROCESS, supplying the name "$SPOOL". Process B will keep B' updated via checkpoints so that B' may become the primary if B fails. B and B' each wish to receive an indication if the other process is deleted. Therefore, B and B' will be automatically set to be each other's moms.

When the last process with the name "$SPOOL" is deleted, process A will be sent a message. Process A is known as the "ancestor" by the fact that this process was the one which created the first process by the name of "$SPOOL". Process A in turn may be a named process, in which case A's name will be sent the termination message. This allows a process-pair, "$A" consisting of processes A and A', to create a named process, "$B" consisting of B and optionally B', and guarantee that it will be sent a message when the process name $B no longer exists. This will occur even if the process which first created $B no longer exists.

4.6 File System

The Guardian file system provides a uniform method for access to disc files, unit record devices, and processes. All files are named: disc files have names such as "$DISC1.SUBVOL.FILE" and unit record devices and processes have names such as "$LP". Access by name allows any process running in any processor to access any file in the system. Direction to the appropriate process of the process-pair is handled by the file system in a manner transparent to the requesting process.

Files of all types are opened by calling:

```
CALL OPEN(filename,filenum,...)
```

The calling process supplies the file name. Security will be checked and then a file number will be returned to the calling process. This file number is then used for all furthur accesses to the file. A file may be opened for "wait" or "no-wait" access. If "wait" access is chosen, the process will be suspended until the requested operation on the file has been completed. On the other hand, if the "no-wait" access is requested then once the operation has been initiated, the requesting process may continue processing.

4.7 Disc Files

Each disc file is composed of between one and sixteen partitions. Each partition resides on a specific disc volume and is in turn composed of up to sixteen extents. Each extent is one or more contiguous disc pages of 2048 bytes each. Disc files comes in several types. The first is "unstructured", similar to UNIX, where the file is treated as a contiguous set of bytes. A current file pointer is kept which is the byte address of the beginning of the next transfer. After each read or write operation:

```
CALL READ(filenum,buffer,cnt,transfercnt)
CALL WRITE(filenum,buffer,cnt,transfercnt)
```

the file pointer is incremented by the number of bytes transferred. The file pointer may be moved explicitly by:

```
CALL POSITION(filenum,fileposition)
```

The second type of file is a "relative-record" file. The file consists of fixed or variable-size records and may be randomly accessed. Rather than positioning to an arbitrary byte in the file, the process positions to the start of a specific record. If the process reads less than the record size, then the file pointer advances to the start of the next record.

The third type of file is "entry-sequenced". Records written to this file may be of varying lengths and are always appended to the end of the file. This type of file is normally used as a log file.

The final type of disc file is "key-sequenced". A file of this type may have a unique primary key and up to 255 alternate keys. Entry-sequenced and relative-record files may also have alternate keys. Each key may be up to 255 bytes long. Access to this file may be done on any key using the procedure:

```
CALL KEYPOSITION(filenum,key,keytag,
                 keylen,positionmode)
```

The parameter "keytag" identifies which key is being used. The pointer "key" designates the value of the key which is "keylen" bytes long. The "positionmode" describes what type of access is to be made to the file. The first type of access is "approximate". Using this, successive reads to the file will return all records whose key values are greater than or equal to the "key" for "keylen". The second type of positioning is "generic". Here, successive reads will return all records whose key value is equal to "key" for "keylen". The final type of positioning is "exact". Successive reads will return all records whose keys are "keylen" long and equal to "key".

Files or individual records may be locked by:

```
CALL LOCKFILE(filenum)
CALL LOCKRECORD(filenum)
```

Record locking and unlocking may be combined with the actual I/O operation desired for increased efficiency:

```
CALL READUPDATELOCK(filenum,...)
CALL WRITEUPDATEUNLOCK(filenum,...)
```

The distributed nature of the system does not allow efficient detection of deadlocks caused by file locking. As a result, this type of checking is not done. A lock request on a file that has been opened with "no-wait" access will allow the application to do other processing if the requested file is not immediately available. A process may use this mechanism to assure that it will not wait indefinitely in the case of a deadlock.

4.8 Disc I/O

The disc processes in each processor share an area of main memory called the "disc cache". Each block read from the disc is placed in this area. Space is reused on a weighted "least recently used" basis. Thus, such items as index blocks for key sequenced files are kept available in memory so that successive accesses do not require that they be reread.

A logical disc volume, "$DISC1", may be recorded onto two different disc drives using two different I/O controllers. This second, or "mirror" volume provides a transparent duplication of data which protects a data base against loss via a failed disc drive or controller. All file writes are performed on both disc drives and file reads may be done from either drive. When a failed drive has been repaired, it may be "revived" while the application continues accesses and updates to files on that logical device.

4.9 Device I/O

I/O operations are done to unit record devices in a similar manner to disc file accesses. Here, Guardian does not support a record-structured, device-independent mode of access and as a result operations such as unblocking tape records must be done by the application program. While this lack of device-independent I/O can be considered a liability in some applications, it allows easy addition of new types of I/O devices to the system without requiring changes to the file system and allows device-dependent control by the application program.

Read and write operations are done in an identical manner for all files. Device-dependent operations such as skipping on vertical form channels on a line printer may be done by:

```
CALL CONTROL(filenum,control#,parameter)
```

Enabling or disabling terminal parity checking or other such access options are done by:

```
CALL SETMODE(filenum,modetype,...)
```

Guardian provides an extremely general purpose interface to asynchronous RS-232 or current loop devices. The file system and asynchronous terminal process provide a read after write operation:

```
CALL WRITEREAD(filenum,buffer,writecnt,
               readcnt,countread)
```

which allows a character sequence to be output to a device followed immediately by a read from the device. This allows the character sequence which causes a CRT terminal to transmit to be sent to it. The line will then be turned around and the terminal's buffer read into the processor. Since this write/read turn-around is done in the device controller, no data is lost because the read could not be started soon enough.

Normally, operating systems wish to enforce certain terminal characteristics such as inserting a carriage return and linefeed after each line written or interpreting certain characters on input for such operations as line and character delete. While Guardian provides such facilities, they may be disabled at the time the system is configured or after the file is opened. Other characteristics such as type of connection, character size, parity, speed, and character echoing are completely configurable. This allows arbitrary character sequences to be input and output without any interpretation or character editing being done by the operating system.

Communication software is also provided to handle multi-point asynchronous terminals. Point-to-point and multi-point Bisync software is also provided. Rather than attempting to emulate specific devices, the application program is allowed to specify the line control used.

4.10 Interprocess I/O

Each process in the system may have messages from other processes queued for it. Access to this message queue is provided via the file "$RECEIVE". A read on this file will return the first message. A process may check to see if any messages are queued and then continue processing if none are present. A process may ascertain the identity of the sending process via the procedure:

```
CALL LASTRECEIVE(sender)
```

This returns the "creationid" of the sending process. It is supplied by the operating system and thus may not be forged by the sending process. A process will receive indication of such events as a process being stopped, a processor failing or being reloaded, or the break key being pressed on a terminal that this process has open in the form of messages read from this file.

A process may open another process as a "file". Once opened, the process may use the file system procedures WRITE, WRITEREAD, SETMODE, and CONTROL to send messages to that process. The receiving process will read these requests from its "$RECEIVE" file. It will then process them and possibly return an error indication to the sending process. This allows the "server" process to simulate some arbitrary device. Using these tools, an output spooler or a process which could allow access to labeled magnetic tapes written on some other system can be constructed. The requesting process believes that it is communicating with a device, and the server process is able to simulate that device without requiring special privileged "hooks" in the file system.

5. APPLICATION PROGRAMS

5.1 Application Program Checkpointing

Application process-pairs are used to provide some service on a failure-tolerant basis. Requests are processed by the primary process and results are returned to the requestor process. On a failure of the primary process, the backup must be able to continue offering this service. This requires that any state changes in the primary process be sent (checkpointed) to the backup process. While such checkpoints could be sent on an instruction-by-instruction basis, this is clearly not feasible because of the overhead involved. Instead, a process need only checkpoint its state when it is about to make a non-retryable request to another process.

For example, at time T1, when the primary process and its backup are in the same state, the primary process starts some operation. Later, at time T2, when it is ready to write the result to a disc file, it will checkpoint changes made since time T1 to its backup. The processes will then again be in the same state. If the primary process failed at any point before T2, the backup process could restart at the last checkpoint, made at T1. The selection of states to checkpoint is analogous to defining restart points for jobs in a batch processing system. In a batch environment, these checkpoints are saved in a disc file; in process-pairs they are saved in a backup process.

Guardian provides system functions for checkpointing process state information

between processes of a process-pair. The first type of item checkpointed is portions of the process' data space. This includes global data and/or the current stack, holding procedure return addresses, procedure local variables, and procedure parameters. Consider the following program segment, written in T/TAL, whose purpose is to output to a terminal the first 100 items of an array, "buffer":

```
FOR  i := 1 TO 100  DO
   BEGIN
   CALL WRITE(terminal,buffer[i],itemlen);
   END;
```

This operation could be made failure-tolerant by two calls to the CHECKPOINT procedure. The first checkpoint copies the entire buffer to the backup process. This need only be done once as the data is not changed by later processing. The second checkpoint, before each write, saves the current process state, including the variable "i". This allows the backup process to take over the operation, duplicating at most one line of output.

```
CALL CHECKPOINT(,buffer[1],buffersize);
FOR  i := 1 TO 100  DO
   BEGIN
   CALL CHECKPOINT(stackbase);
   CALL WRITE(outfile,buffer[i],itemlen);
   END;
```

When the primary process fails, the backup will take over at the last checkpoint. The next logical extension to the original segment would be if the process were copying the one hundred values to be output from some disc file:

```
FOR  i := 1 TO 100  DO
   BEGIN
   CALL READ(infile,buffer,itemlen);
   CALL WRITE(outfile,buffer,itemlen);
   END;
```

In this case, not only would the process' data space contents need to be checkpointed as before, but so would the current file pointers for the input and output files. This ensures that they are correctly set when the backup process takes over. In order for file pointers to be checkpointed, both processes of the process-pair must have the files open. Special functions are provided which allow the primary process to cause a file to be opened or closed by the backup process:

```
CALL CHECKOPEN(filename,...)
CALL CHECKCLOSE(filename,...)
```

In the sample program, CHECKOPEN would be called following the call to OPEN when the primary process started. The program segment would now look like:

```
CALL CHECKPOINT(,buffer[1],buffersize);
FOR  i := 1 TO 100  DO
   BEGIN
   CALL CHECKPOINT(stackbase,,infile,,
                   outfile);
   CALL READ(infile,buffer,itemlen);
   CALL WRITE(outfile,buffer,itemlen);
   END;
```

If a failure occurred after the read but before the write, the backup would take over and repeat the read using the same file pointer as was used by the primary. In both of these examples, a failure following the write but preceding the next checkpoint could result in a record being written twice. This would cause no problem if the record was being written to some absolute position in the file; however, an error would occur when writing to a key-sequenced disc file. In this case, the primary would successfully write the record to the file, but its backup process would get a "duplicate key" error when repeating the write. This problem is solved by having Guardian automatically

ACTION	SEQUENCE VALUES AFTER ACTION: PRIMARY	BACKUP	DISC PROCESS
CHECKPOINT(stackbase, ,infile, , outfile) sequence # of primary sent to backup	0	0	0
CALL WRITE(outfile, buffer, itemlen) sequence #'s match, operation is done, sequence #'s advanced	1	0	1
*** PRIMARY PROCESS FAILS, BACKUP TAKES OVER ***			
CALL WRITE(outfile, buffer, itemlen) sequence #'s don't match, operation is not done, backup's sequence # is advanced		1	1

File System Sequence Numbers
Figure 7

generate an optional sequence number for disc file writes.

A part of the information copied to the backup process when a file is checkpointed is the sequence number for the next write to the file. When a write is done to a file that has been opened with this option, the sequence number passed by the file system is compared with the copy held by the disc process. If it is the same, then the operation is done and the status (error indication and transfer count) is returned to the application process and a copy is saved by the disc process. On the other hand, if sequence numbers do not agree, then no operation is done and a copy of the previous operation's status is returned. Using the previous example, the use of file sequence numbers is shown in Figure 7.

When a process-pair has a file open, any records locked in the file will be considered locked by the process-pair. When the primary fails, the backup may finish file modifications with locks still in effect, preserving the integrity of the data base.

While the primary process operates, the backup process receives the checkpoint information via a call to the procedure CHECKMONITOR. When the primary process sends a checkpoint message via a call to CHECKPOINT, this procedure moves checkpointed portions of the primary process' data space into the backup's data space and saves the latest file information. If a message is directed to the backup process and the primary process still exists, it is rejected with a "ownership" error which informs the sender that the message is to be sent to the other member of the process-pair. Finally, when the primary process fails, CHECKMONITOR will transfer control to the correct restart point.

The Tandem implementation of COBOL provides a similar checkpointing facility. In each case, checkpointing is not an automatic operation. Careful attention during the application design phase will result in fewer checkpoints and will yield a checkpoint scheme that can be analyzed for correctness. Consideration must also be given to how the application will recover from failures occuring while a write operation is in progress to non-disc devices. Recovery when accessing a CRT terminal could be automatically done by rewriting the entire screen. Recovery while printing checks on a line printer would require some manual intervention and operator interaction with the application program.

5.1 Application Structuring

The process, process-pair, and inter-process communication primitives of Guardian provide extremely general tools for application structuring. For example, consider an inquiry application such as hotel reservations. Requests come in from various types of terminals for reservations, cancellations, and hotel registration. Other requests come in for items such as management reports showing the number of rooms available at some hotel on some date. The application could be structured as follows.

Process-pairs will be defined for each type of terminal to handle actual terminal I/O (including any required line protocol and character conversion) and initial request verification. Each process-pair will be designed to handle some number of terminals. When a valid request has been received from a terminal, the terminal process-pair will route the message to the appropriate server process-pair.

Each server process-pair will be assigned a certain part of the application. In some cases, multiple copies of a server program will be run to allow multiple requests to be processed in parallel.

There are several advantages to this approach. First, the handling of terminals and processing of requests have been cleanly separated. New types of terminals can be added by simply adding a type of terminal control process-pair. New types of requests can be handled by adding another type of server process-pair. Likewise, software modifications and testing can be done on a modular basis. Finally, nowhere in this structure is there any requirement for a specific number of processors in the system or for the relative locations of processes. As the system load or the application changes, the number of processors, amount of memory, or physical location of processes may be changed without disturbing the application's internal structure.

6. SOFTWARE RELIABILITY

When design of the operating system was started, we hoped to eliminate as much as possible the archetypal system crash. That is, once or twice a day, or week, the system crashes in a non-repeatable fashion. Our experience on an in-house system used primarily for software development and manual writing shows that we have achieved that goal. During a three-month period in the summer of 1977, a processor failed because of a software

problem on two occasions. In each case, the problem was found at that time and the failure could be repeated by running a particular program.

I propose the following explanation of this reliability. First, the system was very carefully structured and much time was spent in initially specifying primitives. As experience was gained in trying to apply these primitives at higher levels, problems were found. This resulted in design changes at lower levels rather than "kludges" at a higher level. Implementation was also forced to stay within the designed structures because of the distributed nature of the hardware. If a problem could not be solved using processes interacting via messages, then it could not be "kludged" by turning off interrupts and changing some flag in memory. Given a single processor system, there is a very strong temptation to solve difficult problems in this manner.

Second, as the operating system and hardware were developed at the same time, another vendor's system was used to provide interactive text editing, a cross T/TAL compiler, a Tandem/16 processor simulator, and a downloader for the Tandem/16 prototypes. Implementation and checkout were not impeded by unreliable prototypes and as each level of the system was implemented, it could be extensively checked. These tools allowed initial implementation and checkout of all functions of the system through the level of the command interpreter. The wisdom of this approach can best be shown by the fact that when the first prototype processors were made available to the operating systems group, all operating system functions which ran on the simulator ran on the prototypes.

Third, debugging tools were built into the operating system from the start. A low-level interactive debugger was implemented which allowed breakpoints to be set at any level of the operating system, including interrupt handlers. Once this low-level debugger is entered in one processor, clocks in all other processors in the system are stopped so that they will not decide that the first processor is down. When the first processor continues, so will the rest of the system. A full maintenance panel only had to be used to track problems that managed to destroy the low-level debugger. Consistency checks were also coded into low-level routines. For example, before an element is inserted in a doubly-linked list, the list links of the element that the new element is being inserted behind are verified. These checks have proved to be extremely valuable in tracking problems or when implementing new features in the system. Even when extensive changes are being made to the system, it has the property that it will stop at one of these consistency checks very soon after something has gone wrong, allowing the problem to be rapidly found.

Fourth, formal testing was carried out at all levels of the system as they were implemented. A third person, whose only job was testing, was added to the initial project well before completion. By testing not just the external specifications of the system but also the underlying system primitives, it was assured that all system functions at all levels could be checked.

Finally, the primary design goal of the entire system was reliability. When the system design goals are clearly defined and understood by all involved, they can control implementation on a daily basis. Implied goals on the other hand are often forgotten when seemingly small decisions are made.

7. CONCLUSIONS

The innovative aspects of Guardian lie not in any new concepts introduced, but rather in the synthesis of pre-existing ideas. Of particular note are the low-level abstractions, process and message. By using these, all processor boundaries can be hidden from both the application programs and most of the operating system. These initial abstractions are the key to the system's ability to tolerate failures. They also provide the configuration independence that is necessary in order for the system and applications to run over a wide range of system sizes.

Guardian provides the application programmer with extremely general approaches to process structuring, interprocess communication, and failure tolerance. Much has been said about structuring programs using multiple communicating processes, but few operating systems are able to support such structures.

Finally, the design goals of the system have been met to a large degree. Systems, with between two and ten processors, have been installed and are running on-line applications. They are recovering from failures and failures are being repaired on-line.

8. ACKNOWLEDGEMENTS

An operating system is the work of many people. In particular I would like to acknowledge the contributions of Dennis

McEvoy, Dave Hinders, Jerry Held, and Robert Shaw in its design, implementation, and testing.

9. REFERENCES

1. Katzman, J. A., System Architecture for NonStop Computing, Compcon, (February 1977), pp 77-80.

2. Tandem Computers Inc., Tandem/16 System Description, 1976.

3. Bell, C. G. and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, Inc., (1971), pp 15-36.

4. Enslow, P. H. Jr., Multiprocessor Organization - a Survey, Computing Surveys 9, (March 1977), pp 103-129.

5. Dijkstra, E. W., The Structure of the "THE" Multiprogramming System, Comm. ACM 11, (May 1968), pp 341-346.

6. Brinch Hansen, P., The Nucleus of a Multi-programming System, Comm. ACM 13, (April 1970), pp 238-241, 250.

7. Hewlett-Packard Journal, January January 1973.

8. Thompson, K. and Ritchie, D. M., The UNIX Time-Sharing System, Comm. ACM 17, (July 1974), pp 365-375.

Biography. The author received his M.S. degree in Computer Science : Computer Engineering and B.S. degree in Statistics at Stanford University in 1972. Before joining Tandem Computers in 1974, he was employed by Hewlett-Packard. Member of the ACM.