Guardian: A fault-tolerant operating system environment

Architecture is nothing new. Real building architecture has been around for thousands of years, and some of the most beautiful examples of building architecture are also thousands of years old. Computers haven't been around that long, of course, but here too there have been many examples of beautiful architectures in the past. As with buildings, the style doesn't always persist. In this chapter I describe one such architecture, and consider why it had so little impact.

Guardian is the operating system for Tandem's fault-tolerant "NonStop" series of computers. It was designed in parallel with the hardware to provide fault tolerance with minimal overhead cost.

This chapter describes the original Tandem machine, designed between 1974 and 1976 and shipped between 1976 and 1982. It was originally called "Tandem/16", but after the introduction of its successor, "NonStop II", it was retrospectively renamed "NonStop I". Tandem frequently use the term "T/16" both for the system and later for the architecture.

I worked with Tandem hardware full-time from 1977 until 1991. Working with the Tandem machine was both exhilarating and unusual. In this chapter, I'd like to bring back to life some of the feeling that programmers had about the machine.

The T/16 was a fault-tolerant machine, but that wasn't its only characteristic. In this discussion I mention many aspects that don't directly contribute to fault tolerance—in fact a couple detract from it! So prepare for a voyage into the past, about 1980, starting with one of Tandem's marketing slogans.

Tandem/16: Some day all computers will be built like this

Tandem describes the machines as single computers with multiple processors, but from the perspective of the 21st century they're more like a network of computers operating as a single machine. In particular, each processor works almost completely independently from the others, and the system can recover from the failure of any single component, including processors. The biggest difference from conventional networked processors is that the entire system runs from a single kernel image.

Hardware

Tandem's hardware is designed to have no potential for a "single point of failure": any one component of the system, hardware *or* software, can fail without causing the entire system to fail. Beyond this, it is designed for *graceful degradation*: in most cases, the system as a whole can continue running despite multiple failures, though this depends greatly on the nature of the individual failure.

The first implication of this architecture is that there must be at least two of each component, in case one should fail. In particular, this means that the system requires at least two CPUs.

But how should the CPUs be connected? The traditional method, then as now, is for the CPUs to communicate via shared memory. At Tandem we call this *tightly coupled multiprocessors*. But if the processors share memory, that memory could be a single point of failure.

Theoretically it is possible to duplicate memory—a later Tandem architecture actually did that—but it's very expensive, and it creates significant timing problems. Instead, at the hardware level Tandem chose a pair of high-speed parallel buses, the "Interprocessor Bus" or *IPB*, sometimes also referred to as *Dynabus*, which transfer data between the individual CPUs. This architecture is sometimes called *loosely coupled multiprocessors*.

There's more to a computer than the CPU, of course. In particular, the I/O system and data storage are of great importance. The basic approach here is also duplication of hardware; we'll look at it further down.

The resultant architecture looks something like this, the so-called *Mackie diagram*, named after Dave Mackie, a vice-president of Tandem:

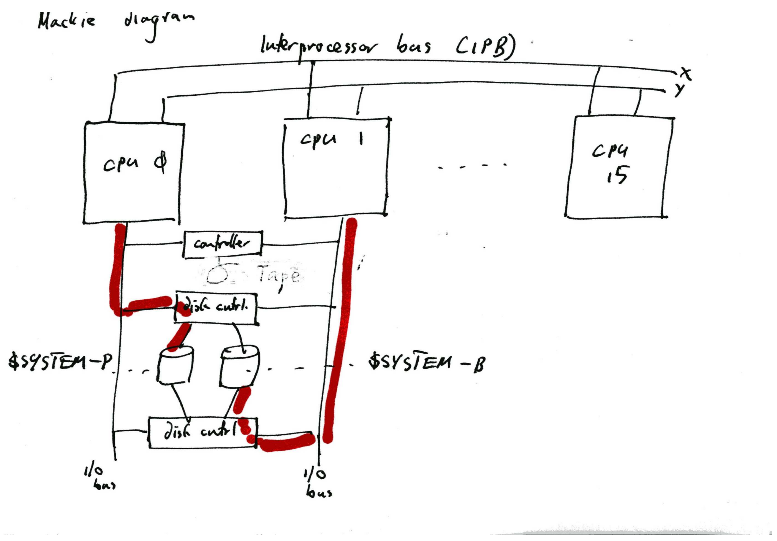


Figure 1

Illustrators: I refer to this diagram in a number of places. In particular, the red markings show the access paths to the disks. Let me know how you represent this so that I can change the text accordingly.

The dotted lines between `$SYSTEM-P` ***and*** `$SYSTEM-B` ***are supposed to indicate that the disks have these names.***

This could easily have led to at least doubling the cost of a system, such as is the case with "hot standby" systems, where one component is only present to wait for the failure of its partner. Tandem chose a different approach for the more expensive components, such as CPUs. Each CPU is active: instead, the operating system processes provides the hot standby function.

Diagnosis

The operating system needs to find out when a component fails. In many cases, there's not much doubt: if it fails catastrophically, it stops responding altogether. But in many cases, a failed component continues to run, but generates incorrect results.

Tandem's solution to this problem is neither particularly elegant nor efficient. The software is designed to be paranoid; at the first suggestion that something has gone wrong, the operating system stops the CPU—there's another to take over the load. If a disk controller returns an invalid status, it is taken offline—there's another to continue processing without interruption. But if the failure is subtle, it could go undetected, and on rare occasions this results in data corruption.

It's not enough for a CPU to fail, of course; other CPUs have to find out that it has failed. The solution here is a watchdog: each CPU broadcasts a message, the so-called "I'm alive" message, over both buses every 1.2 seconds. If a CPU misses two consecutive "I'm alive" messages from another CPU, it assumes that that CPU had failed. If the CPUs share resources (processes or I/O), the CPU that detects the failure then takes over the resources.

Repair

It's not enough to take a defective component offline; to maintain both fault tolerance and performance, it needs to be brought back on line ("up") as quickly as possible, and of course without taking any other components off line ("down").

How this happens depends on the component and the nature of the failure. If the operating system has crashed in one CPU (possibly deliberately), it can be rebooted ("reloaded") on line. The standard way to boot a system is to first boot one processor from disk, then boot all other processors across the IPB. Failed processors are also rebooted via the IPB.

If, on the other hand, the hardware is defective, it needs to be replaced. All system components are *hot-pluggable*: they can be removed and replaced in a running system with power up. If a CPU fails because of a hardware problem, the appropriate board is replaced, and then the CPU is rebooted across the bus as before.

Mechanical layout

The system is designed to have as few boards as possible, so all boards are very large, about 50 cm square. All boards use low power Schottky TTL logic.

The CPU consists of two boards, the processor and the MEMPPU. The MEMPPU contains the interface to memory, including virtual memory logic, and the interface to the I/O bus. The T/16 can have up to 512 kW (1 MB) of semiconductor memory or 256 kW of core memory. Memory boards come in three sizes: 32 kW core, 96 kW and 192 kW semiconductor memory. This means that there is no way of getting exactly 1 MB of semiconductor memory with fully populated boards. Core memory has word parity protection, while semiconductor memory has ECC protection, which can correct a single bit error and detect a double bit error.

Processor cabinets are about 1.8 metres high and house four CPUs with semiconductor memory or 3 CPUs with core memory. The processors are located at the top of the cabinet, with the I/O controllers located in a second rack directly below. Below that are fans, and at the bottom of the cabinet there are batteries to maintain memory contents during power failures.

Most configurations have a second cabinet with a tape drive. The disk drives are free-standing 14" units. There is also a system console, a DEC LA-36 printing terminal.

Processor architecture

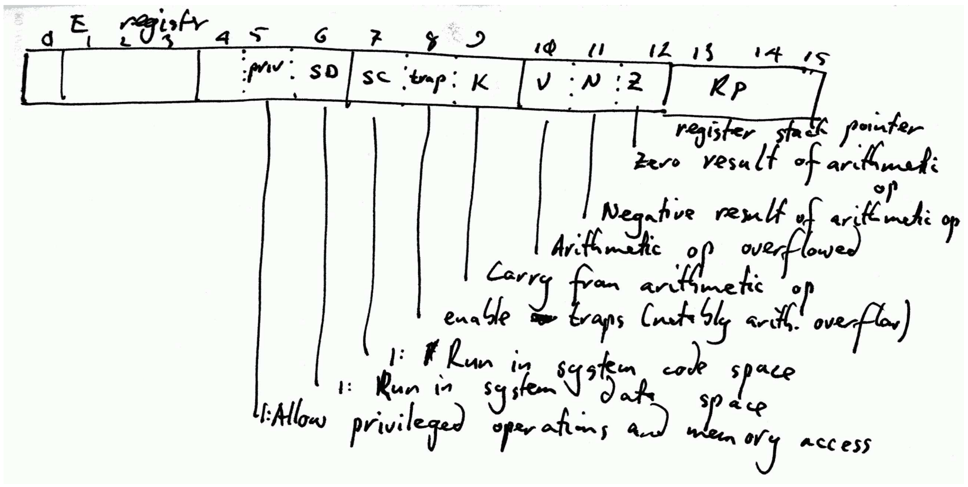
The CPU is a custom TTL design which shows significant similarities with the Hewlett-Packard 3000. It has virtual memory with a 2 kB page size, a stack-based instruction set and fixed-width 16 bit instructions. Raw processor speed is about 0.8 MIPS per processor, giving 13 MIPS in a fully equipped 16 processor system.

Memory addressing

The T/16 is a 16 bit machine, and the address space is limited to 16 bits width. Even in the late 1970s, this is beginning to become a problem, and Tandem addresses it by providing a total of four address spaces at any one time:

- User code. This address space contains the executable code. It is read-only and shared between all processes that use it. Due to the architecture (separate memory for each CPU), the code can only be shared on a specific CPU.
- User data, the data space for user processes.
- System code, the code for the kernel.
- System data, the kernel data space.

With one exception, only one data space and one code space is accessible at any one time. They are specified in the *Environment Register*, which contain a number of flags describing the current CPU state:


E register
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
priv
SD
SC
trap
K
V
N
Z
RP
register stack pointer
Zero result of arithmetic op
Negative result of arithmetic op
Arithmetic op overflowed
Carry from arithmetic op
enable traps (notably arith. overflow)
1: Run in system code space
1: Run in system data space
1: Allow privileged operations and memory access

The SD bit determines the data space, and the SC bit determines the code space. The SG-relative addressing mode is an exception to this rule: it always addresses system data.

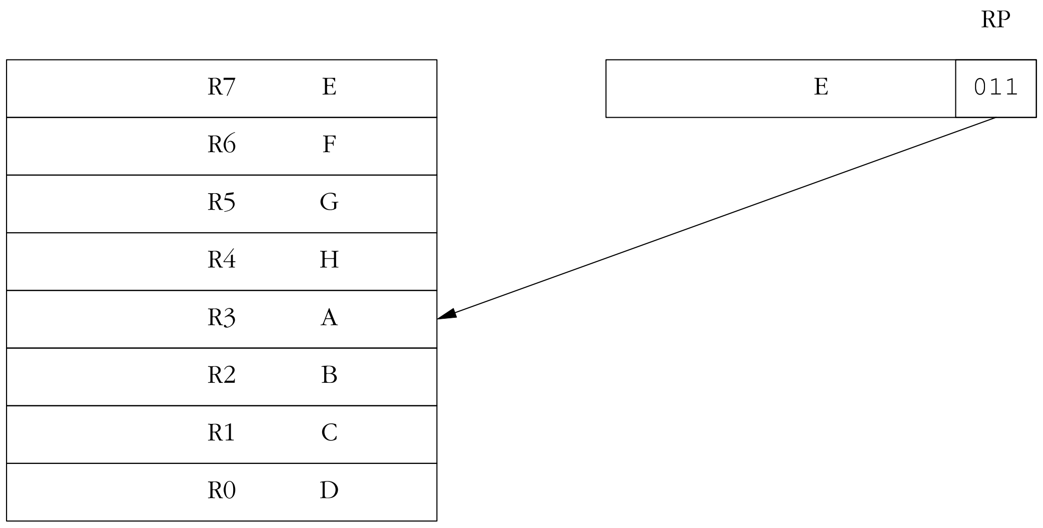
In keeping with the aim of reliability and data integrity, the trap bit in the E register enables, amongst other things, traps on arithmetic overflow. There are "logical" equivalents of the arithmetic instructions that do not set the condition codes.

The CPU has a hardware stack addressed by two registers, the *S register* or stack pointer, and the *L register*, which points to the current stack frame. The L register is a relatively new idea: it points to the base of the current frame. Unlike the S register, it does not change during the execution of a procedure.[1] The stack is limited by addressing considerations to the first 32 kB of the current data space, and unlike some other machines, it grows upwards.[2]

In addition to the hardware stack, there is a *register stack* of 8 16 bit words. The registers are numbered R0 to R7, but the instruction set uses them as a circular stack, where the top of stack is defined by the RP bits of the E register. In the following example, RP is set to 3, making R3 the top of stack, referred to as the A register:

1. This is the same thing as the base pointer register used in most 21st century processors.

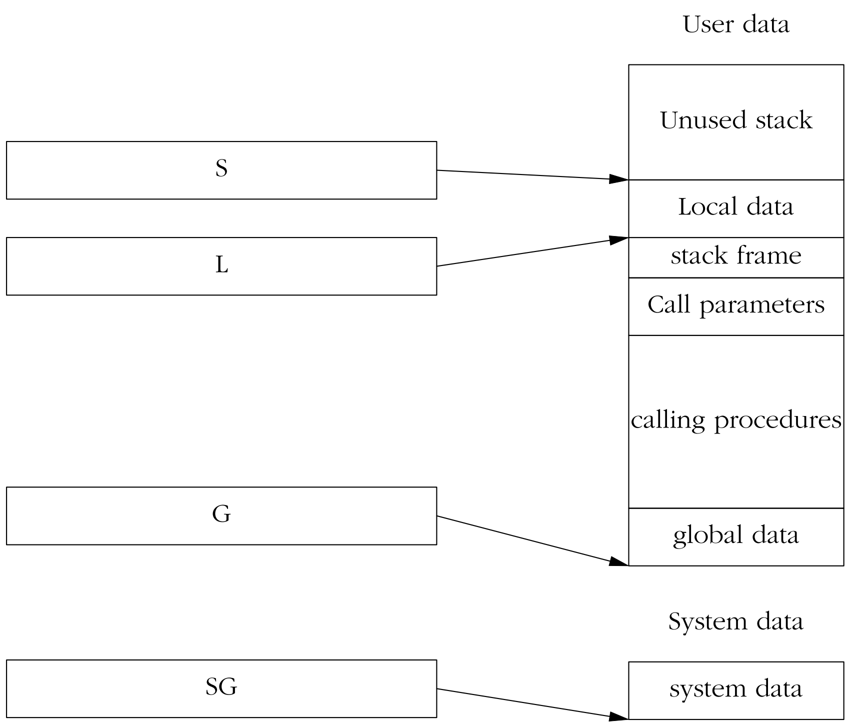
2. Stacks were quite a new idea in the 1970s. Like its predecessor, the HP 3000, Tandem's support for stacks went significantly beyond that of systems like DEC's PDP-11, the most significant other stack-based machine of the time.



Assuming that the register stack is “empty” at the beginning, a typical instruction sequence might be:

```
LOAD    var^a           -- push var^a on stack (R0)
LOAD    var^b           -- push var^b on stack (R1)
ADD                     -- add A and B (R1 and R0), storing result in R0 (A)
STOR    var^c           -- save A to var^c
```

Instructions are all 16 bits wide, which does not leave much space for an address field: it is only 9 bits wide. To work around this problem, Tandem bases addressing on offsets from a series of registers:



Only the following memory areas can be addressed directly (without indirect addressing):

- The first 256 words of the current data space, referred to as “`G`” (*global*) mode. These are frequently used for indirect pointers.
- The first 128 words positive offset from the `L` register, called `L+`. These are the local variables of the current procedure invocation, which would be called *automatic* variables in C.
- The first 64 words of system data (“`SG+`” mode). System calls run in user data space, so the CPU needs some means for privileged procedures to access system data. They are not accessible, even read-only, to unprivileged procedures.
- The first 32 words below the current value of the `L` register. This includes the caller stack frame (3 words) and up to 29 words of parameters passed to the procedure.
- The first 32 words below the top of the stack (`S-` addressing). These are used for *subprocedures*, procedures defined inside another procedure, which are called without leaving a stack frame. This address mode thus handles both the local variables and the parameters for a subprocedure.

These address modes are encoded in the first few bits of the address field of the instruction:

0	*offset*	Global (0:%377)
10	*offset*	L+ (0:%177)
110	*offset*	SG (0:%77)
1110	*offset*	L- (-%37:0)
1111	*offset*	S- (-%37:0)

Illustrators: These boxes are 9 bits wide, and each of the digits in the first part represents one bit.

The `%` symbol represents octal numbers, like `%377` (decimal 255, or hexadecimal `7F`). Tandem does not use hexadecimal.

The instruction format also provides single-level indirection: if the `I` bit is set in the instruction, the retrieved data word is taken as the address of the final operand, which has to be in the same address space. The address space and the data word are both 16 bits wide, so there is no possibility for multi-level indirection.

One problem with this implementation is that the unit of data is a 16 bit word, not a byte. The instruction set also provides "byte instructions" with a different addressing method: the low-order bit of the address specifies the byte in the word, and the remainder of the address are the low-order 15 bits of the word address. For data accesses, this limits byte addressability to the first 32 kB of the data space; for code access, it limits the access to the same half of the address space as the current instruction. This has given rise to the restriction that a procedure cannot span the 32 kB boundary in the code space.

There are also two instructions, `LWP` (*load word from program*) and `LBP` (*load byte from program*) that can access data in the current code space.

Procedure calls

Tandem's programming model owes much to the Algol and Pascal world, so it reserves the word *function* for functions that return a value, and *procedure* for those that do not. Two instructions are provided to call a procedure: `PCAL` for procedures in the current code space, and `SCAL` for procedures in the system code space. `SCAL` fulfils the function of a *system call* in other architectures.

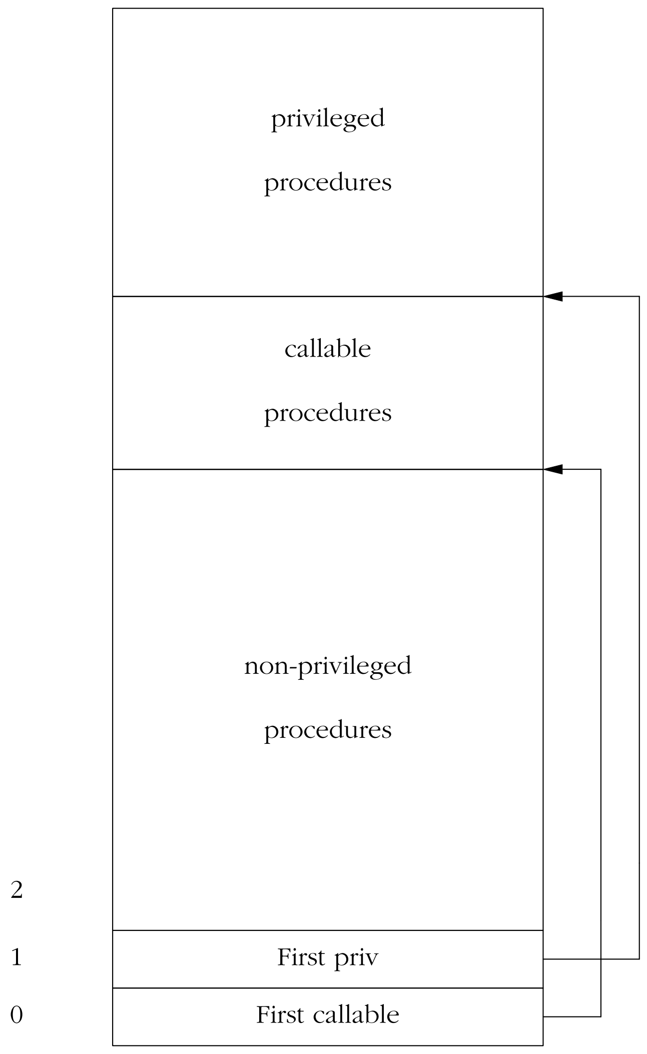
All calls are indirect via a *Procedure Entry Point Table* or *PEP*, which occupies up to the first 512 words of each code space. The last 9 bits of the `PCAL` or `SCAL` instruction are an index in this table.

This approach has dangers and advantages: the kernel uses exactly the same function call methods as user code, which simplifies things. On the other hand, at least in theory, the `SCAL` instruction enables any user program to call any kernel function.

The system protects access to sensitive procedures based on the `priv` bit in the E register. It distinguishes between three kinds of procedures:

- Non-privileged procedures, which can be called from any procedure, whether privileged or not.
- Privileged procedures, which can be called only from other privileged procedures.
- *Callable* procedures, which can be called from any procedure, but which set the `priv` bit once called. They provide the link between the privileged and the non-privileged procedures.

The distinction between privileged, non-privileged and callable procedures is dependent on their position in the PEP. Thus it is possible to have non-privileged library procedures in the system PEP, sometimes called the *SEP*. The table has the following structure:



Action of the `PCAL` and `SCAL` instructions

The `PCAL` instruction performed the following actions:

- If the `priv` bit in the E register is not set (meaning that the calling procedure is non-privileged), check the "first priv" value (word 1 of the code space). If the offset in the instruction is greater or equal, the procedure is trying to call a privileged procedure. Generate a protection trap.

- If the `priv` bit in the E register is not set, check the “first callable” value (word 0 of the code space). If the offset in the instruction is greater or equal, set the `priv` bit in the E register.
- Push the current value of the P register (program counter) on to the stack.
- Push the *old* value of the E register on to the stack.
- Push the current L register value on the stack.
- Copy the S register (stack pointer) to the L register.
- Set the RP field of the E register to 7 (empty).
- Load the contents of the PEP word addressed by the instruction into the P register.

The `SCAL` instruction works in exactly the same way, except that it also sets the `SC` bit in the E register, thus ensuring that execution continues in kernel space. The data space does *not* change.

The `PCAL` and `SCAL` instructions are very similar, and the programmer normally does not need to distinguish between them. That is done by the system at execution time. Thus library procedures can be moved between user code and system code with no recompilation.

The interprocessor bus

All communication between CPUs goes via the *interprocessor bus* or *IPB*. There are in fact two busses, called X and Y (see Figure 1), in case one fails. Unlike other components, both busses are used in parallel when they’re up.

Data is passed across the bus in fixed-length packets of 16 words. The bus is fast enough to saturate memory on both CPUs, so the client CPU performed it synchronously in the *dispatcher* (scheduler) using the `SEND` instruction. The destination (server) CPU reserves buffer space for a single transfer at boot time. On completion of the transfer the destination CPU receives a *bus receive* interrupt and handles the packet.

Input/Output

Each processor has a single I/O bus with up to 32 controllers. All controllers are dual-ported and connected to two different CPUs. At any one time, only one CPU has access to any specific controller. This relationship between CPU and controller is called *ownership*: the controlling CPU "owns" the controller. The backup path is not used until the primary path fails or the system operator manually switches to it (a so-called *primary switch*).

Disks are a particularly sensitive issue: many components could fail. It could be a disk itself, the physical connection (cable) to the disk, the disk controller, the I/O bus or the CPU to which it is connected. As a result, in addition to the dual-ported controllers, each disk is physically duplicated—at least in theory—and it is also dual-ported and connected to two different controllers, both connected to the same two CPUs The restriction remains that only one CPU can access each controller at any one time, but it is possible for one of the CPUs to own one of the controllers, and the other CPU to own the other controller. This is also desirable from a performance point of view.

Figure 1 shows a standard configuration: the system disk `$SYSTEM` is in fact a pair of disks called `$SYSTEM-P` (primary) and `$SYSTEM-M` (mirror). The primary disk is controlled from CPU 0 by one controller, and the mirror disk is controlled from CPU 1 by the other controller (XXX red markings XXX).

That's the theory, anyway. In practice, disks and drives are expensive, and many people run at least some of their disks in degraded mode, without duplicating the drive hardware. This works as well as you would expect, but of course there is no longer any further fault tolerance: effectively, one of the disks has already failed.

Process structure

Guardian is a process-based microkernel system: apart from the low-level interrupt handlers (a single procedure, `IOINTERRUPT`) and some very low-level code, all the system functions are performed by system processes which run in system code and data space. The more important ones are:

- The *system monitor*, PID 0 in each CPU, is responsible for starting and stopping other processes and for miscellaneous tasks such as returning status information, generating hardware error messages and maintaining the time of day.

- The *memory manager*, PID 1 in each CPU, is responsible for I/O for the virtual memory system.
- The I/O processes are responsible for controlling I/O devices. All access to I/O devices from anywhere in the system goes via its dedicated I/O process. The I/O controllers are connected to two CPUs, so each device is controlled by a pair of I/O processes running in those CPUs, a *primary* process that performs the work and a *backup* process that tracks the state of the primary process and waits to fail or to hand over control to it voluntarily ("primary switch").

  The main issue in the choice of primary CPU is the CPU load, which needs to be balanced manually. For example, if you have 6 devices connected between CPUs 2 and 3, you would probably put the primary process of 3 of them in CPU 2, and the primary process of the other 3 in CPU 3.

Process pairs

The concept of pairs of processes is not limited to I/O processes. It is one of the cornerstones of the fault-tolerant approach. To understand the way they work, we need to understand the way messages are passed in the system.

Message system

As we've seen, the biggest difference between the T/16 and conventional computers is the lack of any single required component. Any one part of the system can fail without bringing down the system. This makes it more like a network than a conventional shared memory multiprocessor machine.

This has far-reaching implications for the operating system design. A disk could be connected to any two of sixteen CPUs. How do the others access it? Modern networks use file systems such as NFS or CIFS, which run on top of the network protocols, to handle this special case. But on the T/16 it isn't a special case; it is the norm.

File systems aren't the only thing that required this kind of communication: interprocess communication of all kinds required it too.

Tandem's solution to this issue is the *message system*, which runs at a very low level in the operating system. It is not directly accessible to user programs.

The message system transmits data between processes. In many ways it resembles the later TCP or UDP. The initiator of the message is called the *requestor*, and the object is called the *server*.[1]

All communication between processes, even on the same CPU, goes via the message system. The following data structures implements the communication:

- Each message is associated with two *Link Control Blocks* or *LCBs*, one for the requestor and one for the server. These small data objects are designed to fit in a single IPB packet. If more data is needed than would fit in the LCB, a separate buffer is attached.

- To initiate a transfer, the requestor calls the procedure `link`. This procedure sends the message to the server process and queues the LCB on its *message queue*. At this point the server process has not been involved, but the dispatcher awakes the process with an `LREQ` (*link request*) event.

  On the requestor side, the call to `link` returns immediately with information to identify the request; the requestor does not need to wait for the server to process the request.

- At some time the server process sees the `LREQ` event and calls `listen`, which removes the first LCB from the message queue.
- If a buffer is associated with the LCB, and it includes data to be passed to the server, the server calls `readlink` to read in the data.
- The server then performs whatever processing is necessary and then calls `writelink` to reply to the message, again possibly with a data buffer. This wakes the requestor on `LDONE`.

  > A process can sleep waiting for multiple specific events, which are specified as a bit mask. When woken, it receives information about which event has occurred. A number of these events are related to the message system.

- The requestor sees the `LDONE` event and examines the results and terminates the exchange by calling `breaklink`, which frees the associated resources.

Only other parts of the kernel use the message system directly. The *file system* uses it to communicate with the I/O devices and other processes: interprocess communication is handled almost identically to I/O, and it also is used for maintaining fault-tolerant *process pairs*.

This approach is inherently asynchronous and multi-threaded: after calling `link`, the requestor continues its operations. Many requestors can send requests to the same server, even when it's not actively processing requests. The server does not need to respond to the link request immediately. When it replies, the requestor does not need to acknowledge the reply immediately. Instead, in each case the process is woken on an event which it can process when it is ready.

Process pairs, revisited

One of the requirements of fault tolerance is that a single failure must not bring the system down. We've seen that the I/O processes solve this by using process pairs, and it's clear that this is a general way to handle the failure of a CPU.

1. These names correspond closely in function to the modern terms *client* and *server*.

Guardian therefore provides for creation of user-level process pairs.

All process pairs run as a *primary* and a *backup* process. The primary process performs the processing, while the backup process is in a "hot standby" state. From time to time the primary process updates the memory image of the backup process, a process called *checkpointing*. If the primary fails or voluntarily gives up control, the backup process continues from the state of the last checkpoint. A number of procedures implement checkpointing, which is performed by the message system:

- The backup process calls `checkmonitor` to wait for checkpoint messages from the primary process. It stays in `checkmonitor` until the primary process goes away or relinquishes control. During this time, the only use of the CPU is message system traffic to update its data space and calls to `open` and `close` to update file information,
- The primary process calls `checkpoint` to copy portions of its data space and file information to the backup process. It is up to the programmer to decide which data and files to checkpoint, and when.
- The primary process calls `checkopen` to checkpoint information about file opens. This effectively results in a call to `open` from the backup process. The I/O process recognizes that this is a backup open and treats it as equivalent to the primary open.
- The primary process calls `checkclose` to checkpoint information about file closes. This effectively results in a call to `close` from the backup process.
- The primary process may call `checkswitch` to voluntarily release control of the process pair. When this happens, the primary and backup processes reverse their roles.

When the backup process returns from `checkmonitor`, it has become the new primary process. It returns to the location of the old primary's last call to `checkpoint`, not to the location from which it was called. It then carries on processing from this point.

In general, the life of a process pair can look like:

Primary	Backup
Perform initialization `call newprocess` to create backup process	
	Perform initialization `call checkmonitor` to receive checkpoint data

`call checkpoint`	wait in `checkmonitor`
`call checkopen`	`call open` from `checkmonitor`
processing	wait in `checkmonitor`
`call checkswitch`	wait in `checkmonitor`
processing	wait in `checkmonitor`
voluntary switch: `call checkswitch`	take over
`call checkmonitor` to receive checkpoint data	processing
wait in `checkmonitor`	`call checkpoint`
wait in `checkmonitor`	processing
wait in `checkmonitor`	`call checkpoint`
wait in `checkmonitor`	*CPU fails*
take over	*(gone)*
process	

Synchronization

This approach proves very reliable. It can deliver reliability superior to that of a pure lock-step approach: in some classes of program error, notably race conditions, a process that is running in lock-step will run into exactly the same program error and crash as well. A more loosely coupled approach can often avoid the exact same situation and continue functioning.

A couple of issues are not immediately obvious:

- Checkpointing is CPU-intensive. How often should a process checkpoint? What data should be checkpointed? This decision is left to the programmer. If he does it wrong, and forgets to checkpoint important data, or does it at the wrong time, the memory image in the backup process will be inconsistent, and it may malfunction.
- If the primary process performs externally visible actions, such as I/O, after performing a checkpoint but before failing, the backup process will repeat them after takeover. This could result in data corruption.

In practice, the issue of incorrect checkpointing has not proved to be a problem, but duplicate I/O most certainly is a problem.

The system solves this problem by associating a sequence number called a *sync id* with each I/O request. The I/O process keeps track of the requests, and if it receive a duplicate request, it simply returns the completion status of the first call to the request.

Networking: EXPAND and FOX

The message system of the T/16 is effectively a self-contained network. That puts Guardian in a good position to provide wide-area networking by effectively extending the message system to the whole world. The implementation is called *EXPAND*.

From a programmer's point of view, EXPAND is almost completely seamless. Up to 255 systems can be connected together.

System names

Each system has a name starting with a backslash, such as `\ESSG` or `\FOXII`, along with a node number. The node numbers are much less obvious than modern IP addresses: from the programmer's perspective they are necessary almost only for encoding file names, which we'll see below.

EXPAND is an extension of the message system, so most of the details are hidden from the programmer. The only issues are the difference in speed and access requirements, which we'll look at below.

FOX

From purely practical constraints it is difficult to build a system with more than 16 CPUs; in particular, hardware constraints limit the length of the interprocessor bus to a few metres, so a realistic limit is 16 CPUs. Beyond that, Tandem supplies a fast fibre-optic connection capable of connecting up to 14 systems together in a kind of local area cluster. In most respects it is a higher-speed version of EXPAND.

File system

Tandem uses the term *file system* to mean the access to system resources that can supply data ("read") or accept it ("write"). Apart from disk files, the file system also handles devices, such as terminals, printers and tape units, and processes (interprocess communication).

File naming

There is a common naming convention for devices, disk files and processes, unfortunately complicated by many exceptions. Processes may have names, but only I/O processes and paired processes *must* have a name. In all cases, the file "name" is 24 characters long and consists of three 8 byte components. Only the first component is required; the other two are used only for disk files and named processes.

Unnamed processes use only the first 8 bytes of the name. Unpaired system processes, such as the monitor or memory manager, have the format:

			0			CPU	PIN
0	1	2	3	4	5	6	7

Unpaired user processes have the format:

2			timestamp			CPU	PIN
0	1	2	3	4	5	6	7

Illustrators: these pictures attempt to show a sequence of 8 bytes. The last two bytes contain CPU and PIN (described in the text). In the first illustration, bytes 0-5 contain 0, in the second the first 2 bits contain the value 2, and in the 5 bytes and 6 bits in the middle are either a timestamp or 0.

The combination *CPU* and *PIN* together forms the process ID or *PID*. The PIN is the *process identification number* within the CPU. This limits each CPU to 256 processes.

Real names start with a $ sign. Devices only use the first 8 bytes, disk files use all three; the individual components look like disk, directory and file names, though in fact there is only one directory per disk volume. Processes can also use the other two components for passing information to the process.

Typical names are:

`$TAPE`	tape drive
`$LP`	printer
`$SPLS`	Spooler process
`$TERM15`	terminal device
`$SYSTEM`	System disk
`$SYSTEM SYSTEM  LOGFILE`	system log file, on disk `$SYSTEM`
`$SPLS   #DEFAULT`	Default spooler print queue
`$RECEIVE`	Incoming message queue, for interprocess communication

Illustrators: The names in the left column must contain the correct number of spaces. `SYSTEM` ***and*** `#DEFAULT` ***start in the 9th character position,*** `LOGFILE` ***in the 17th (i.e. the names consist of groups of 8 characters).***

If a component is less than 8 bytes long, it is padded with ASCII spaces. Externally, names are represented in ASCII with periods, for example `$SYSTEM.SYSTEM.LOGFILE` and `$SPLS.#DEFAULT`.

There are still further quirks in the naming. Process subnames must start with a hash mark (#), and user process names (but not device names, which are really I/O process names) have the PID at the end of the first component:

$	S	P	L	S		CPU	PIN
0	1	2	3	4	5	6	7

The PID in this example is the PID of the primary process. It limits the length of user process names name to 6 characters, including the initial `$`.

As if that wasn't enough, there is a separate set of names for designating processes, disk files or devices on remote systems. In this case, the initial `$` sign is replaced by a \ symbol, and the second byte of the name is the system number, shifting the rest of the name one byte to the right. This limits the length of process names to 5 characters if they are to be net-visible. So from another system the spooler process we saw above might have the external name `\ESSG.$SPLS` and have the internal format:

\	173	S	P	L	S	CPU	PIN
0	1	2	3	4	5	6	7

The number 173 is the node number of system `\ESSG`.

Asynchronous I/O

One of the important features of the file system interface is the strong emphasis on asynchronous I/O. We've seen that the message system is intrinsically asynchronous in nature, so this is relatively simple to implement.

Processes can choose synchronous or asynchronous ("no wait") I/O at the time they open a file. When a file is opened no-wait, an I/O request will return immediately, and only errors that are immediately apparent will be reported, for example if the file descriptor isn't open. At a later time the user calls `awaitio` to check the status of the request. This gives rise to a programming style where a process issues a number of no-wait requests, then goes into a central loop to call `awaitio` and handle the completion of the requests, typically issuing a new request.

Interprocess communication

At a file system level, interprocess communication is a relatively direct interface to the message system. This causes a problem: the message system is asymmetrical. The requestor sends a message and may receive a reply. There's nothing that corresponds to a file system `read` command. On the server side, the server reads a message and replies to it; there's nothing that corresponds to a `write` command.

The file system provides `read` and `write` procedures, but `read` only works with I/O processes, which map them to message system requests. `read` doesn't work for interprocess communication level, and in practice `write` is also not used much. Instead, the requestor uses a procedure called `writeread` to first write a message to the server and then get a reply from it. Either the message or the reply can be null (zero length).

These messages find their way to the server's message queue. At a file system level, the message queue is a pseudo-file called `$RECEIVE`. The server opens `$RECEIVE` and normally uses the procedure `readupdate` to read a message. At a later point it can reply with the procedure `reply`.

System messages

The system uses `$RECEIVE` to pass messages to processes. One of the most important is the *startup message*, which passes parameters to a newly started process. The following example is written in TAL, Tandem's low-level system programming language (though the name stands for "Tandem Application Language"). TAL is derived from HP's SPL, and it is similar to Pascal and Algol. One of the more unusual characteristics is the use of the caret (`^`) character in identifiers: the underscore (_) character is not allowed. This example should be close enough to C to be intelligible. It shows a process which starts a child server process and then communicates with it:

Parent process (requestor)

```
call newprocess (program^file^name,,,,,, process^name); -- start the server process
call open (process^name, process^fd);            -- open process
call writeread (process^fd, startup^message, 66); -- write startup message
while 1 do
  begin
  read data from terminal
  call writeread (process^fd,
                  data, data^length,             -- write data
                  reply, max^reply,              -- read data back
                  @reply^length);                -- return real reply length
  if reply^length > 0
    write data back to terminal
  end;
```

Child process (server)

```
call open (receive, receive^fd);
do
  call read (receive^fd, startup^message, 66);
until startup^message = -1;             -- first word of startup message is -1.
while 1 do
  begin
  call readupdate (receive^fd, message, read^count, count^read);
  process message received, replacing buffer contents
  call reply (message, reply^length);
  end;
```

The first messages that the child receives are system messages: the parent `open` of the child sends an `open` message to the child, then the first call to `writeread` sends the startup message. The child process handles these messages and replies to them. It can use the open message to keep track of requestors, or to receive information passed in the last 16 bytes of the file name. Only then does the process receive the normal message traffic from the parent. At this point, other processes can also communicate with the child. Similarly, when a requestor closes the server, it receives a `close` system message.

Device I/O

It's important to remember that device I/O, including disk file I/O, is handled by I/O processes, so "opening a device" is really opening the I/O process. Still, I/O to devices and files is implemented in a slightly different manner, though the file system procedures are the same. In particular, the typical procedures used to access files are the more conventional `read` and `write`, and normally disk I/O is not no-wait.

Security

In keeping with the time, the T/16 is not an overly secure system. In practice, this hasn't cause any serious problems, but one issue is worth mentioning: the transition from non-privileged to privileged procedures is based on the position of the procedure entry point in the PEP table and the value of the `priv` bit in the E register. Early on exploits became apparent: if you could get a privileged procedure to return a value via a pointer, and get it to overwrite the saved E register on the stack in such a way that the `priv` bit was set, the process would remain privileged on return from that procedure. It is the responsibility of callable procedures to check their pointer parameters to ensure that they don't have any addressing exceptions, and that they return values only to the user environment. A bug in the procedure `setlooptimer`, which sets a watchdog timer and optionally returns the old value, made it possible to become the `SUPER.SUPER` (the root user, with ID 255,255, or -1):

```
proc make^me^super main;
begin
int .TOS = 'S';                          -- top of stack address

call setlooptimer (%2017);               -- set a timer value
call setlooptimer (0, @TOS [4]);         -- reset, return old value to saved E reg
pcb [mypid.<8:15>].pcbprocaid := -1;     -- dick in my PCB and make me super
end;
```

The value `%2017` gets written to the E register, in particular setting the `priv` bit, which leaves the process in privileged state. It then uses SG-relative addressing to modify the user information in its own PCB. `mypid` is a function returning the current processe's PID, and the last 8 bits (`<8:15>`) were the PIN, which is used as an index in the PCB table.

This bug was quickly fixed, of course, but it showed a weakness in the approach: it is up to the programmer to check the parameters passed to callable procedures. Throughout the life of the architecture, such problems have reoccurred.

File access security

Tandem's approach to file access security is similar to that of UNIX, but users can belong only to a single group, which is part of the user name. Thus my user name `SUPPORT.GREG`, also written numerically as `20,102`, indicates that I belong to the `SUPPORT` group (20)—only, and that within that group my user ID is 102. Each of these fields is 8 bits long, so the complete user ID fits in a word.

Each file has a number of bits describing what access the owner, the group or all users have to the file. Unlike UNIX, however, the bits are organized differently: the four permissions are *read*, *write*, *execute* and *purge*. *Purge* is the Tandem name for *delete*, and it's necessary because directories don't have their own security settings.

For each of these access modes, there is a choice of who is allowed to use them:

- *Owner* means only the owner of the file.
- *Group* means anybody in the same group.
- *All* means anybody.

All of these relate only to the same system as where the file is located. A second set of modes was introduced with networking to regulate access from users on other systems:

- *User* means only a user with the same user and group number as the owner of the file.
- *Class* means anybody with the same group number as the owner of the file.

- *Network* means anybody, anywhere.

There is no security whatsoever for devices, and user processes have to roll their own. The former is a particular disadvantage in a networked environment. At a security seminar in early 1989, I was able to demonstrate stealing the `SUPER.SUPER` (root) password on system `\TSII`, in the middle of the management area in Cupertino, simply by putting a fake prompt on the system console. I was in Düsseldorf at the time.

Folklore

Coming back into the present, early 21st century, it's easy to forget the sheer fun it was working with the computer. Tandem was a fun company, and it looked after its employees. One Friday in late 1974, early in the development of the system, the founders finally got the software to work on the hardware; up to this point the software had been developed on simulators. You can imagine the excitement. The story goes that one of the VPs went out and brought in a crate of beer, and they all sat round the crate, celebrating the event and discussing the future. One thing they decided was that the crate of beer should be a weekly event: the Tandem Beer Bust was born, and it really did continue into the 1990s, during which it became increasingly politically incorrect and was finally cancelled.

Tandem gave rise to lots of slogans and word plays, of course—the name "Tandem" itself was one. In those days we had T-shirts with slogans like "So nice, so nice, we do it twice", "There's no stopping us" and "Tandem users do it with mirrors". And, of course, the standard answer when anybody came up with an excess of just about anything: "It's there in case one fails".

This last slogan was more than just word play. It sat deep in our thought processes. Early on, after returning from 5 weeks of Tandem training, I was faced with the sad discovery that our cat had run away. After establishing that she wasn't going to return, we went out and got—two new cats. It wasn't until much later that I learnt that this was the result of successful brain-washing. Even today I have a phobia of rebooting a computer unless it's absolutely unavoidable.

Disadvantages

The T/16 was a remarkably successful machine for its intended purpose—at one time over 80% of all ATMs in the USA were controlled by Tandem systems—but of course there were disadvantages as well. Some, like the higher cost in comparison with conventional systems, are inevitable. Others were not so obvious to the designers.

Performance

Tandem was justifiably proud of the near-linear scaling of performance when hardware was added. *[Horst, Chou, 1985]*, which refers to a later system, the TXP, shows how a FOX cluster can scale linearly from 2 to 32 processors.

[Bartlett 1982] shows the down side: the performance of the message system limited the speed even of small systems. A single message with no attached data takes over 2 ms to transmit, and messages with 2000 bytes of data in each direction take between 4.6 ms (same CPU) and 7.0 ms (different CPUs). This is the overhead for a single I/O operation, and even in its day it was slow. The delay between sequential I/O requests to a file was long enough that they would not occur until the data had passed the disk head, meaning that only one request could be satisfied per disk revolution. A program that sequentially reads 2 kB from disk and processes it (for example, the equivalent of *grep*) would get a throughput of only 120 kB/s. Smaller I/O sizes, such as 512 bytes, could limit the throughput to floppy disk speeds.

Hardware limitations

As the name "Tandem/16" suggests, the designers had a 16 bit mind set. That is fairly typical for the mid-1970s, but the writing was already on the wall that "real" computers had a 32 bit word. Over the course of time a number of them were addressed. In 1981, Tandem introduced the *NonStop II* system with an upwards compatible instruction set and fewer hardware limitations. Over the next 10 years, a number of compatible but faster machines were introduced. None were extremely fast, but they were fast enough for online transaction processing. In addition, the operating system was rewritten to address the more immediate problems, and over the course of time additional improvements were made. The changes included:

- Introduce a 31 bit address mode to give user processes "unlimited" memory space. This mode used byte addresses, but it didn't remove the limitations on stack size and code spanning the 32 kB boundary, since the old instruction formats were still in use.

- Increase the number of hardware virtual memory maps. The T/16 had only four, for the four code and data spaces. Later machines also had a system library and user library space, which increased the total space available to 384 kB. A total of 16 memory maps meant that the processor could directly address up to 2 MB without involving the memory manager.

  One of these maps was used as a kind of translation lookaside buffer to handle the 31 bit extended addresses.

  Still later, the number of library spaces was increased from 2 (system and user) to up to 62 (31 each system and user) by segment switching: only a single user library and system library map could be active at any one time.

- Message queue size proved to be a problem. The monitor processes sent status messages at regular intervals to every process that wanted them. If the process didn't read the messages, large numbers of resources (LCBs and message buffers) could be used. To address this problem, a *messenger* was introduced that would keep a single copy of these status messages and send them to a process when it called `listen`.

Missed opportunities

The T/16 was a revolutionary machine, but it also offered an environment that few other machines of the day did. Ultimately, though, it was the small things that got in the way. For example, device independence is one of the most enduring aims of operating systems, and Tandem went a long way towards this goal. Ultimately, though, they missed their full potential because of naming issues and almost gratuitous incompatibilities. Why was it not possible to use `read` in interprocess communication? Why did process names have to differ in format from device names? Why did they need a # character in the 9th byte?

Split brain

A more serious issue was with the basic way of detecting errors. It worked fine as long as only one component failed, and usually quite well if two failed. But what if both interprocessor buses failed? Even in a two-CPU system, the results could be catastrophic. Each CPU would assume that the other had failed and take over the I/O devices, not once, but continually. Such circumstances did occur, fortunately very rarely, and they often resulted in complete corruptions of the data on disks shared between the two CPUs.

Posterity

From 1990 on, a number of factors contributed to a decline in Tandem's sales:

- Computer hardware in general was becoming more reliable, which narrowed Tandem's edge.
- Computer hardware was becoming *much* faster, highlighting some of the basic performance limitations of the architecture.

In the 1990s, the T/16 processor architecture was replaced by a MIPS-based solution, though much of the remaining architecture remained in place. The difference in performance was big enough that as late as 2000 Tandem was still using the MIPS processors to emulate the T/16 instructions. One of the reasons was that most Tandem system-level software was still written in TAL, which was closely coupled to the T/16 architecture. Moves to migrate the code base to C were rejected because of the cost involved.

For such a revolutionary system, the Tandem/16 has made a surprisingly small impression on the industry and design of modern machines. Much of the functionality is now more readily available—mirrored disks, network file systems, the client-server model or hot pluggable hardware—but it's difficult to see anything that suggests that Tandem was leading the way. This may be because the T/16 was so different from most systems, and of course the purely commercial environment in which it was developed didn't help either.

Further reading

Hewlett-Packard has a number of papers on its web site; start looking at the Tandem Technical reports at *http://www.hpl.hp.com/techreports/tandem/*. In particular,

[Bartlett 1982] http://www.hpl.hp.com/techreports/tandem/TR-81.4.html?jumpid=reg_R1002_USEN, "A NonStop Kernel" by Joel F. Bartlett, gives more information about the operating system environment.

[Horst, Chou, 1985], "The Hardware Architecture and Linear Expansion of Tandem NonStop Systems", April 1985, *http://www.hpl.hp.com/techreports/tandem/TR-85.3.html.*

[Bartlett et al 1990] "Fault Tolerance in Tandem Computer Systems" *http://www.hpl.hp.com/techreports/tandem/TR-90.5.html* describes the hardware in more detail.

[Gray 1988], "The cost of messages" *http://www.hpl.hp.com/techreports/tandem/TR-88.4.html* describes some of the performance issues from a theoretical

point of view.