IRIX is the Bestix: An Overview of the IRIX Operating System

CS 351 – Operating Systems

Spring 2001

Section 002

GROUP B:
Matthew Boyer
Christian Dunlap
Steven Hughes
Patrick Quentmeyer
Shelley Sloane
Michael Williams

Table of Contents

Overview

IRIX was first conceptualized in 1982 by the Silicon Graphics Institute out of a need for a for an extremely stable but yet extraordinarily powerful graphics platform. Since it's inception, however, IRIX has found its niche to be in many other areas of computing besides just graphical applications, especially the natural sciences. This is probably in part due the fact that IRIX was the first commercially available Unix operating system to support symmetric multi-processing. Another crucial reason was that IRIX systems, long before Windows or any other operating system for that matter, implemented 32-bit and 64-bit environments. Early on, scientist and mathematicians found these super computing capabilities necessary for carrying out extremely precise computations on very large amounts of data.

Besides being a robust, reliable yet powerful operating system IRIX is all so a very flexible. These aspects are apparent in from just a quick overview of the system. First IRIX is a multi-user operating system, which allows multiple users to work on private files on the same system at the same time. Second, being a multi-tasking operating system IRIX allow users to run multiple applications a the same time. Third, IRIX has powerful networking capabilities which allow transparent file transfers. Fourth, a wide variety of hardware, such as terminals, disk drives, and printers, can be add without the need of any software at all.

Version 6.5 the most current release of IRIX, is fully compliant with Unix V Releas4 as well as the Open Groups standard which include Year 2000, UNIX 95, POSIX, X11, Motif, and OpenGL®. All new versions of IRIX 6.5 take full advantage of the new MIPS processor, one of the most powerful processors in the industry. According to SGI:

> "IRIX is the most scalable, feature-rich, high-performance OS available. For high-end scalability, big I/O performance, real-time performance, and superior graphics capabilities, IRIX is the premier choice. No OS, including Windows® 2000, Linux®, or Solaris®, is capable of matching IRIX in these respects."

SGI has listed 5 major goals of IRIX 6.5 which include:

1. IRIX must be applicable on all platforms, that is no platform specific IRIX releases
2. IRIX updates will be made available on a set schedule which will be strictly followed.
3. Reliability is crucial, therefore all IRIX release will strive for 100% reliability. Fixing any found bugs is the highest priority for future updates.
4. All applications developed in IRIX 6.5.X must have binary compatibility so that the application will run on any previous or future releases of the operating system.
5. The performance of IRIX must not differ depending of complexity of the system or that of it's load.

SGI's chief scientist John Mashey, said:

> "There are researchers that do important work with IRIX because, in some cases, nothing else can do the job. Some IRIX users carry out grand initiatives in fundamental and advanced research to improve health, save lives, and advance our understanding of climate and our ability to predict it. Others use it to extend our ability to create and use new kinds of supercomputing capabilities, build safer and more efficient cars and aircraft, contribute to national security, and, through cosmology, extend our mental fingertips to explore deep space."

Process States

An IRIX process represents a thread of execution. The virtual address space of a process, the contents of its user structure and proc table entry, and the values contained in machine registers when the process is running constitute the context of the process (Silicon Graphics, Inc. 1).

To support multiple processes, IRIX implements a process-scheduling algorithm that assures a fairly equitable division of processor time among all processes (Silicon Graphics, Inc. 1). This algorithm is non-preemptive, that is, the running process cannot be preempted by another process but can be preempted by the kernel. The running process can yield to another process "voluntarily," by making a system call such as an I/O request that causes it to sleep, in which case another process is selected to run. The running process can also be preempted by the kernel to handle an exception, in which case the process is rescheduled to resume immediately after the exception handler is finished (Silicon Graphics, Inc. 1). The kernel also enforces limits on the amount of time a process can monopolize the processor using time slicing (Silicon Graphics, Inc. 1).

The command ps prints information about active processes. Without options, information is printed about processes associated with the controlling terminal. The output consists of a short listing containing only the process ID, terminal identifier, cumulative execution time, and the command name (Indiana University 1). If the option S is added to the ps command then the state of the process will also be printed out
The following are possible states of an IRIX process; there are eight possible states (Indiana University 1).

1. Process is running on a processor
2. Process is sleeping, waiting for a resource
3. Process is running.
4. Process is terminated and parent not waiting
5. Process is stopped.
6. Process is in intermediate state of creation.
7. Process is creating core image after error.
8. Process is waiting for memory

In Stallings' book there are 5 basic states of a process and nine intermediate states. There is much similarity when comparing the Stallings' states with the IRIX states mentioned above. The first state mentioned is when the process is running on the processor. This would be most applicable to the Stallings' state of Running as would the third IRIX state of just running. This is when the process is currently being executed. The IRIX state of Sleeping is most applicable to the Stallings' transition state of Running to Blocked. In Stallings' book, when a process needs a I/O resource it is put into the Blocked state if it must wait. The same applies to the IRIX system only its calling Sleeping. The IRIX process of waiting for memory will also have the same effect as the Blocked state in Stallings' book since the process is waiting for a resource. The IRIX state of Stopped is the same as the Stallings' state of Ready. The process is no longer executing, however it may still have more operations that need to take place and needs to wait for its time share on the processor. The IRIX state of Terminated is the same as the Exit state in Stallings' book. The process is released from the execution pool and the parent is made aware of this. In IRIX the process can not be terminated by another process, only the kernel is allowed to make a termination decision. The New state would be the same as the process being in some intermediate state of creation in the IRIX system. When the process is in an intermediate state of creation it has not been added to the pool of executable processes as mentioned in Stallings' book. There is no definite mapping of the IRIX state of creating core image after error to a

Stallings' state. The most applicable Stallings' state would probably be Running to Ready state because this gives the process a chance to recover itself after an error. The process does not die however it cannot execute at this time.

Threads

IRIX uses POSIX-style threads, which are used on most Unix based operating systems. A *thread* is an independent execution state; that is, a set of machine registers, a call stack, and the ability to execute code. When IRIX creates a process, it also creates a thread to execute that process. However, it is possible to write a program that creates many more threads to execute in the same address space. (man realtime)

There is only one type of thread used in an IRIX system. It is called a pthread, which stands for POSIX thread. However, the attributes of the pthread can be changed either when the pthread is created or by using certain commands such as **pthread_attr_setscope()**. (sgi man) This command sets the initial scheduling scope of the thread by passing one of the scope constants, PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS. By default, the process scope is selected but thread scheduling by the kernel is provided with the system scope attribute. System scope threads run at real-time policy and priority. These types of threads can only be created by privileged users. The other attributes that can be changed are the automatic-detach, inherited attributes, starting thread priority, scheduling policy, scheduling scope, stack size, stack guard size, address of memory to use as stack, and the deallocation of a thread. (sgi man)

In IRIX, there is another way to execute a program, and that is through lightweight processes. These differ from threads in that they are less portable and create higher overhead when created with the **sproc()** command. Threads share all resources and attributes of a single process. If you want each executing entity (thread or lightweight process) to have its own file descriptors, you must use lightweight or normal processes. Also, threads allow for the modification of data which is shared with other threads, while lightweight processes do not. (serpentine)

Mutual Exclusion

This section will explain the methods the IRIX Operating System uses to implement mutual exclusion on system resources. There are four main ways IRIX implements mutual exclusion:

- Locks
- Semaphores
- Test and Set
- Barriers

The IRIX Operating System has a multitude of ways in which it handles mutual exclusion. It uses mutual exclusion to ensure that each system resource is utilized by only one process at a time so that different processes, which require the same resource, cannot perform conflicting operations on that same resource at the same time. Mutual exclusion is essential for coordination between process execution and resource allocation in concurrent multiprocessing systems. IRIX has different methods for implementing mutual exclusion to handle a variety of operating schemes and circumstances. By operating schemes, IRIX uses different approaches to mutual exclusion depending on the processor environment. It implements different mutual exclusion schemes for uniprocessor environments and multiprocessor environments. Under different circumstances, IRIX use of a mutual exclusion method depend on which level of abstraction the operating

system is using to force mutual exclusion and which system call is triggering the necessity of mutual exclusion for a resource by a process.

Locks

Locks are used by IRIX to lock access to a resource to only one process so that another process cannot access the resource. When a process requests a resource from the operating system, the process is issued a lock to that resource. All other processes are essentially locked out of using that particular resource until the process that is using the resource release the lock on it. It then becomes available for other process to use. These locks are used in IRIX semaphores to create an organized method for allowing different processes' requests to the same resource to be handled in an orderly manner so that each process will get its turn in utilizing the resource. When the locks are used directly, IRIX issues a busy wait on processes that try and use a resource that is already locked to another process for multiprocessor machines. For uniprocessor machines, IRIX suspends the process that requests a resource that is locked. Locks can be used for any resource on the machine and are used by other mutual exclusion methods to ensure that only one process at a time gets a resource exclusively while implementing mutual exclusion in a more organized and system beneficial way.

Semaphores

The use of semaphores in IRIX applies to those system resources that can only be used by one process at a time and can incur a backlog of processes waiting to use that resource such that the resource will be used completely before being released back to the system (i.e., disk write/read etc.). The semaphores use a last-in-first-out queue to allow only one resource allocation per process and to permit all processes that are requesting the resource access to that resource on a first come first serve basis depending on which process has requested the resource first. The other processes then wait until those processes ahead on the queue have finished with the resource and relinquish it to the system so that they can in turn utilize the resource. The semaphores use the locks discussed above to allow only one process access to a resource at a time. Using semaphores organizes the lock method in a way that benefits the overall management of process and resources coordination by making sure that all processes requesting a resource will get access to that resource in an organized manner.

IRIX has two types of semaphores: normal and polled. The normal semaphore blocks a process requesting the resource if the semaphore count becomes negative and the polled semaphore does not block the process, but rather sends it a return code that the process must then send to a poll which allows the semaphore to synchronize multiple resource use among many different processes instead of a single resource

Test and Set

IRIX has two functions that allow access to the hardware test and set methods for mutual exclusion in the MIPS architecture. These functions permit direct hardware control for mutual exclusion and pertain especially to data shared in memory by more than one process. There is a Load Link function and a Store Conditional function. These functions ensure that for multiprocessor machines, shared data is permissibly modifiable by different processes so that each process can correctly modify the data, as per its needs, without interference from the other process.

Barriers

Barriers are a unique type of mutual exclusion. They are a way to collectively ensure that parallel processes begin execution at the same time. The barriers collect the parallel processes as they load until all necessary processes are ready. IRIX uses barriers on multiprocessor machines to coordinate such parallel processes. Processes which need to be in sync with one another before executing make calls to barrier so that they will wait until all the requisite processes have made a barrier call and are likewise waiting and ready to begin, at which time the processes commence execution. Barriers are mostly used on multiprocessor machines because true parallel processing cannot occur with a single processor. IRIX is typically run on multiprocessor systems because it is designed for intense graphic applications.

(Cortesi, Ch 4 – Mutual Exclusion)

Memory Management

IRIX 6.5 usually runs on a multiprocessor machine because it is mainly used for complex graphics programs, which require high computing speeds. For most applications, the operating system is capable of producing reasonable levels of locality via dynamic page migration and replication; however, because of the multiprocessor environment, it is beneficial to have a way for each application to control how memory is allocated and managed in order to minimize access and write times.

When the operating system needs to execute an operation to manage a section of a process' address space, it uses the methods specified by the Policy Module that is attached to each section. A Policy Module contains the policy methods used to handle the operations initial allocation, dynamic relocation and paging. For initial allocation there are three policies: placement, page size, and fallback. For dynamic relocation there are two policies, which are migration and replication. Paging only has one policy, which is the paging policy.

When a programmer wants to create a Policy Module they can use the following code:

```
typedef struct policy_set {
      char* placement_policy_name;
      void* placement_policy_args;
      char* fallback_policy_name;
      void* fallback_policy_args;
      char* replication_policy_name;
      void* replication_policy_args;
      char* migration_policy_name;
      void* migration_policy_args;
      char* paging_policy_name;
      void* paging_policy_args;
      size_t page_size;
                  short page_wait_timeout;
                  short policy_flags;
} policy_set_t;

pmo_handle_t pm_create(policy_set_t* policy_set);
```

This is the data structure that holds all the policy types that are to be used when allocating memory for pages.

To allocate a physical page, the virtual memory system physical allocator first calls the method provided by the Placement Policy that determines from where the page should be allocated. Internally, this method returns a handle identifying the node from which memory should be allocated. The physical memory allocator determines the page size to be used for the current allocation by checking the Page Size Policy. With this information, the system can now try to find a space on the node that is large enough to hold the page. If there is then the operation completes, but if not then the Fallback Policy is called. The fall back method provided by the s policy decides whether to try the same page size on a different node, a smaller page size on the same source node, sleep or just to fail.

The Fallback Policy that is chosen is determined by the kinds of memory access patterns an application exhibits. The page size is given higher priority is the applications working set is large but still has reasonable cache behavior. If the application generates a lot of cache misses locality is given precedence over page size. Once a page has been allocated to a node it stays there until it needs to be migrated, paged out, or faulted back in.

The Migration Policy controls the migration of a page. Some applications will not make use of page migration because they present a very uniform memory access pattern from beginning to end and the initial placement of the pages is good enough to be used for the duration of the program. However, some programs may benefit from dynamic page migration in which the pages are attracted to the nodes where most of the memory is being used.

The Replication Policy determines the degree of replication of the text. Only read-only text can be replicated. Text that is shared by lots of processes, which are running on different nodes may benefit from several replicas of that text. This is to provide high locality and minimize the interconnect contention.

Finally, the Paging Policy controls all the paging activity. When a page is about to be evicted or replaced by another page, the pager uses the Paging Policy Methods in the corresponding Policy Module to determine whether the page can really be stolen or not.

(SGI Man MMCI)

File Management

This section will explain some important concepts about hard disk file management through filesystems for the IRIX operating system.

Every IRIX system disk contains some standard directories. These directories contain operating system files organized by function. A filesystem is a data structure that organizes files and directories on a disk partition so that they can be easily retrieved. Only one filesystem can reside on a disk partition. A file is a one-dimensional array of bytes with no other structure implied. Information about each file is stored in structures called inodes. Files cannot span filesystems. A directory is a container that stores files and other directories. It is merely another type of file that the user is permitted to use, but not allowed to write; the operating system itself retains the responsibility for writing directories. Directories cannot span filesystems. The combination of directories and files make up a filesystem.

The starting point for any filesystem is an unnamed directory that serves as the root for that particular filesystem. In the IRIX operating system there is always one filesystem that is itself referred to by that name, the root filesystem. Filesystems are attached to the directory hierarchy by the 'mount' command, resulting in an IRIX directory structure as in **Figure 1.1**.

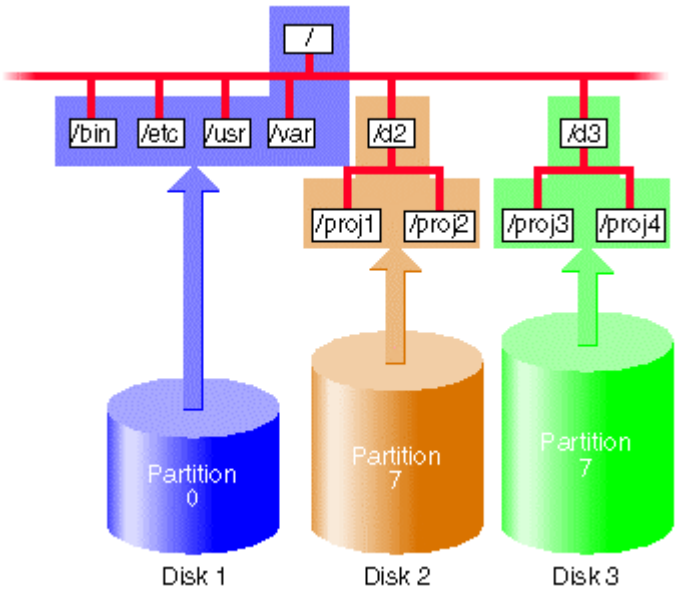

Figure 1.1 – IRIX Filesystem
Image courtesy of SGI

Two or more disk partitions can be joined to create a logical volume. The logical volume can be treated as if it were a single disk partition, so a filesystem can reside on a logical volume and hence is the only way for a single filesystem to span more than one disk.

(SGI, Ch. 5 – Filesystem Concepts, p. 2-7)

Inodes

Information about each file is stored in a structure called an inode. The word inode is an abbreviation of the term index node. An inode is a data structure that stores all information about a file except its name, which is stored in the directory. Each inode has an identifying inode number, which is unique across the filesystem that includes the file. An inode contains the following information:

- The type of the file.
- The access mode of the file. The mode defines the three access permissions read, write, and execute. The mode may also contain security labels and access control lists.
- The number of hard links to the file.
- The owner's user-ID number and the group-ID number.
- The size of the file in bytes.
- The date and time the file was last accessed and last modified.
- Information for finding the file's data within the disk partition or logical volume.
- The pathname of symbolic links.

Inodes do not contain the name of the file or its directory.

(SGI, Ch. 5 – Filesystem Concepts, p. 7-8)
(Stallings, p. 552)

Hard Links and Symbolic Links

Information about each file is stored in an inode for the file. The name of the file is stored in the file's directory and associating the filename with an inode number creates a link to the file. This kind of link is called a hard link.

Although every file is a hard link, this term is normally applied when two or more filenames are associated with the same inode number. Since inode numbers are unique within a filesystem, hard links cannot be created across filesystems boundaries.

As an example, consider a current directory contains the file called 'origfile,' and there is a hard link, called 'linkfile', created to 'origfile'. Since 'origfile' and 'linkfile' are now two names for the same file, changes in the content of the file are visible when using either filename. If one of the links is removed there will be no resulting effect on the other link. The file will not be removed until the file's 'link count', stored in the inode, is zero.

The second type of link is the symbolic link. The symbolic link is really a file. This file contains the pathname of another file or directory as a text string. Since the symbolic link is actually a file, it has its own owners and permissions. The symbolic link can point to a file or directory in another filesystem. The symbolic link can become useless if the file or directory it points to is removed. When this happens to the target of the symbolic link, the link becomes known as a dangling symbolic link.

(SGI, Ch. 5 – Filesystem Concepts, p. 8-9)

Filesystem Names

Filesystems do not have actual names; instead they are identified by location on a disk or the position in the directory structure. The filesystems can be located and identified with the following information:

- The block and character device file names of the disk partition or logical volume that holds the specific filesystems.
- A mnemonic name for the disk partition or logical volume that holds the filesystem.
- The mount point for the filesystem.

(SGI, Ch. 5 – Filesystem Concepts, p. 9-10)

XFS Filesystems

This section will provide a closer look into IRIX, specifically a brief overview of one type of filesystem created for SGI's version of IRIX, named XFS.

XFS is a type of IRIX filesystem that was created for use on Silicon Graphics systems. The XFS filesystem can be used on both desktop computers and supercomputers. Some of XFS's important features are:

- Full 64-bit file capabilities.
- Efficient support of large, sparse files, files that have 'holes'.
- Rapid and reliable recovery after system crashes because of journaling technology.

- Integrated, full-function volume manager, known as the XLV Volume Manager.
- Very high I/O performance that scales well on multiprocessing systems.
- A guaranteed-rate I/O for multimedia and data acquisitions usage.
- Compatibility with existing applications
- User-specified filesystem block sizes ranging from 512 bytes up to 64KB.
- Small directories and symbolic links of 156 characters or less use no space.

XFS applies database journaling technology to allow for quick recovery and high reliability. A recovery can be made within a few seconds of a system crash without using a filesystem checker.

XFS is designed to be a very high performance filesystem. Under certain conditions, throughput exceeds 100 MB per second. The performance of the XFS filesystem is scaled to complement the Challenge MP architecture and the Origin 2000 architecture. While traditional filesystems suffer from reduced performance as they grow in size, the XFS filesystem has no performance penalty.

Filesystems can be created with block sizes ranging from 512 bytes to 64 KB. For real-time data, the maximum extent size is 1 GB. The filesystem extents, providing for contiguous data within a file, are automatically created for normal files and can be configured at the file's creation time. Extents are multiples of a filesystem block. Inodes are created as needed by XFS filesystems. Another feature of XFS filesystems called extended attributes enable users and applications to associate name and value pairs to files, directories, symbolic links, and inodes.

(SGI, Ch. 5 – Filesystem Concepts, p. 10)

References

Cortesi, David. "Topics in IRIX Programming".
URL: http://www.vislab.usyd.edu.au/ebt-bin/nph-dweb/dynaweb/SGI_Developer/T_IRIX_Prog/

Indiana University (1996). "Monitoring Memory Performance."
URL: http://www.uwsg.indiana.edu/usail/tuning/stat-mem.html

Silicon Graphics, Inc.(1995). "IRIX 6.1 Technical Report."
URL: http://www.biochemtech.uni-halle.de/~sgi/irix61_doc.html

Silicon Graphics Inc (2001). "IRIX Admin: Disks and Filesystems."
URL: http://techpubs.sgi.com/library/index.html

Stallings, William (2001). *Operating Systems*. Upper Saddle River, New Jersey: Prentice Hall.
ISBN 0-13-031999-6