

ESPRIT LTR 21917 (Pegasus II)

Deliverable 4.5.2: Unix functionality

Rolf Neugebauer and Michael Dales
Department of Computing Science
University of Glasgow

Wednesday 31st March 1999

Abstract

Traditional operating systems present fixed, high-level abstractions to application developers and users. These are part of standard APIs, such as POSIX or X/OPEN, which are typically implemented as a thin library layer on top of monolithic kernels. Recent efforts in operating system research, however, have focussed on providing more flexibility and new functionality to applications by lowering the abstraction level to a minimal kernel interface. Higher-level abstractions are provided through user-level servers or, more recently, through shared libraries. These library based operating systems allow the design and implementation of arbitrary high-level abstractions as user-level shared libraries on top of a minimal kernel interface.

Nemesis is a library based operating system which offers genuine support for multi-media data stream types by providing Quality of Service guarantees for all shared resources in the system. In *Nemesis*, the libraries implementing the high level abstractions are carefully designed to avoid interactions between different processes for shared state. Abstractions which rely on traditional stateful APIs are handled using library componets called *personalities*, described in (Neugebauer & Black 1998).

This deliverable report describes the design and implementation of a personality offering Unix-like functionality for the *Nemesis* operating system. This effort was motivated on two grounds: first, to research the techniques and feasibility of providing such functionality in a single address space system such as *Nemesis*, and second, by the desire to take advantage of the vast amount of existing application code available for Unix systems.

1 Introduction

The *Nemesis* operating system was designed with the aim of providing genuine support for multi-media data stream types by providing Quality of Service (QoS) guarantees for all shared resources in the system. In the early stages of the Pegasus I (ESPRIT BRA 6865) project (Mullender et al. 1994), it was decided that this objective could be best achieved not by attempting to provide any backward compatibility with existing operating systems, but instead by developing a new operating system completely *ab initio*. As a result, *Nemesis* is implemented as a single address space operating system (SASOS) allowing shorter context switch times and facilitating rich sharing of text and data at a very fine granularity. This is supported by a sophisticated, object-based programming model providing dynamic linkage between strongly typed interfaces at run-time (Roscoe 1995, Roscoe 1994). However, there is a huge body of popular software which relies on industry standard Unix-compatible Application Programming Interfaces (APIs). In order to

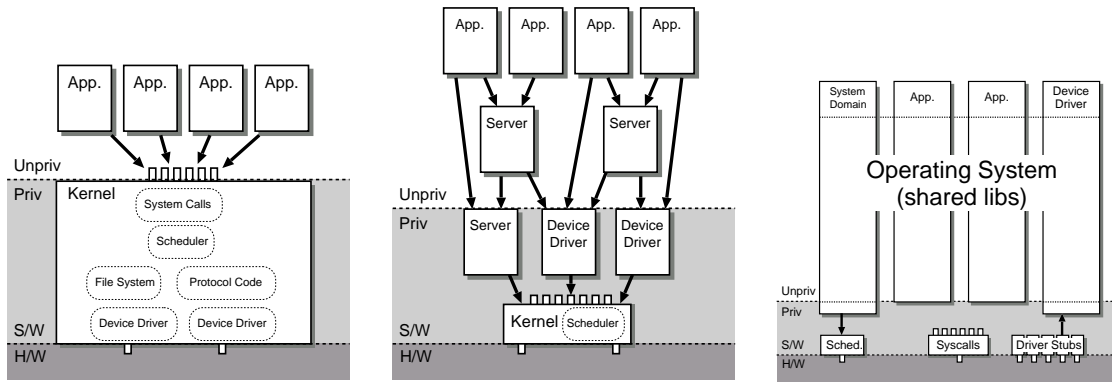


Figure 1: OS architectures: (a) monolithic kernel, (b) microkernel, and (c) vertically structured.

increase scope for dissemination activities it was determined to provide a degree of backward compatibility with existing APIs, such as POSIX (ISO/IEC 1996), within the Pegasus II project.

Initially, problems relating to the use of a single address space, as opposed to the multiple address spaces implicitly assumed by the Unix operating system and the POSIX standard, were addressed. We developed a *Stateful API Architecture* (Neugebauer & Black 1998) which allows us to emulate implicit Unix-like linkage in a single address space at source-code level, without sacrificing desirable properties of the standard Nemesis programming model, such as dynamic linkage at run-time, and fine-grained sharing of code.

This report describes the architecture and implementation status of a subsystem (also known as a personality) emulating standard Unix functionality based on standard Nemesis abstractions. At this point it has to be stressed that the aim of this deliverable is to provide a reasonable subset of Unix functionality, which allows us to port Unix applications and “standard” utilities with relative ease. Support for complete Unix functionality is well beyond the resources of the project¹ and is of little research interest. Instead we focus on a useful subset of the X/Open Release 4 (Open Group 1994), which is the predecessor of the current Single Unix Specification, version 2 (Josey 1997). We chose this standard since it combines parts of the BSD and the System VR4 API and also contains an API for networking, something the POSIX standard (ISO/IEC 1996) doesn’t cover.

One interesting research issue is how to design a Unix subsystem which extends to standard Unix applications the advanced resource allocation mechanisms which the Nemesis operating system provides to its “native” applications. Section 2 describes our architecture of the Unix personality subsystem, based on the general design principles of the Nemesis operating system, which enables us to provide QoS guarantees to existing Unix applications. The Unix personality is constructed from a number of components each forming individual linkage units and providing well defined functionality typical of a Unix system. These components are described in detail in sections 3 to 8. In section 9 we report on our experience with porting Unix applications to the Unix personality. Section 10 provides a brief evaluation of performance and presents an initial evaluation of our claim to provide QoS to unaltered Unix applications. In section 11, we present our conclusions.

2 Architecture

The Unix operating system was designed as a system based on a monolithic kernel. Within this kernel, privileged operations are performed and data structures are maintained. User level applications communicate with the kernel through a well defined system call interface. Higher level abstractions, such as the POSIX API or the standard C library (ISO/IEC 1997) are typically provided as a relatively thin layer on top of the system call interface, as shown in figure 1(a).

¹And in some cases impossible as described in later sections of this report.

With the advent of microkernel based systems (Liedtke 1996) such as MACH (Golub et al. 1990), L3 (Liedtke 1993), or recently L4 (Härtig et al. 1997) the kernel was reduced to a smaller entity implementing only basic operating system functionality such as interrupt handling, enforcement of protection, and scheduling. Higher level abstractions are typically implemented by a number of servers. Clients access this functionality using an inter-process communication mechanism implemented by the kernel. This is schematically depicted in figure 1(b). A number of personalities have been implemented on top of microkernel architectures. The implementation of Unix servers had been particularly popular, e.g., an implementation of Unix on the Mach microkernel (Golub et al. 1990), Unix for the V-Kernel (Cheriton et al. 1990), an implementation of SunOS on the Spring operating system (Khalidi & Nelson 1993), and the port of Linux to L4 (Härtig et al. 1997). Other abstractions have also been implemented, e.g., a port of OS/2 to Mach (Phelan et al. 1993).

Throughout the Pegasus project it has been argued (Mullender et al. 1994, Leslie et al. 1996) that neither monolithic kernel nor microkernel architectures are suitable for operating systems with genuine support for multi-media streams; in both architectures individual applications can negatively influence the performance of other applications by congesting shared resources such as the kernel, or servers performing time-dependent operations on behalf of applications. This phenomena has been termed QoS-crosstalk. In the Nemesis operating system this problem has been solved by structuring the operating system *vertically*. A minimal kernel implements only the necessary functionality for basic interprocess communication and deals with the scheduling of the CPU resource. Other shared resources are structured so that clients negotiate a share of that resource *out of band* and then perform all time critical data operations themselves (Barham 1996). This approach has been demonstrated to be feasible for all resources managed by an operating system, including disk access (Barham 1997), the network interface (Black et al. 1997), the framebuffer (Barham 1996, Black 1998), the virtual memory subsystem (Hand 1999), and the audio interface (Reed 1998). From this architecture-provided, fine-grained, low-level access to shared resources, high-level abstractions are implemented as user-level shared libraries. This leads to the vertical structure depicted in figure 1(c).

The definition of the Unix API has been developed based on a monolithic kernel structure as depicted in figure 1(a). Unfortunately, some implicit assumptions about this structure are made in the POSIX standard. The POSIX API is best and easiest implemented on a monolithic kernel, encapsulating the state related to processes, virtual memory management, and filesystems, and providing individual address spaces to processes. This, for example, can be demonstrated by the specification of the `fork()` and the `exec()` system calls used for process creation. The `fork()` system call creates a new process executing in its own address space, which is an exact copy of its parent address space including its text and data segment, its file-descriptor table and process state. The `exec()` system call replaces the current calling process with a new program, and the new program starts executing at its `main()` function. Part of this functionality, namely the copying of the parent's address space to the child's address space, is impossible to implement in a single address space environment – references to data and absolute jumps and branches would refer to the same data in all processes. See section 5 for a more detailed description of related problems and the solution adopted for the implementation of the Unix personality.

The management of process and filesystem related state, however, is independent of the choice of address space architecture. We considered two possible approaches, depicted in figure 2. The first option (figure 2(a)) would be the implementation, or port, of a popular Unix system to Nemesis, with a server replacing the kernel. This has been proven viable with the port of the Linux operating system to the L4 microkernel (Härtig et al. 1997). The architecture-dependent part of the Linux kernel was emulated using basic mechanisms provided by the L4 microkernel, whilst the architecture-independent part of the kernel was left (almost) unchanged². Whilst having the potential of providing almost the full functionality of a powerful and popular Unix system, this approach was rejected early on in the design of the Unix subsystem for Nemesis. The port of an existing Unix kernel to Nemesis would have been possible, but we did not consider it interesting enough from a research perspective. The main objection, however, was that with this approach all *data path* operations (e.g., reads and writes to files) would be performed by the

²Other microkernel systems implement a multi server approach, e.g., (Golub et al. 1990, Hamilton & Kougiouris 1993), whereby multiple servers implement parts of the Unix functionality, such as process control, filesystems, and virtual memory management.

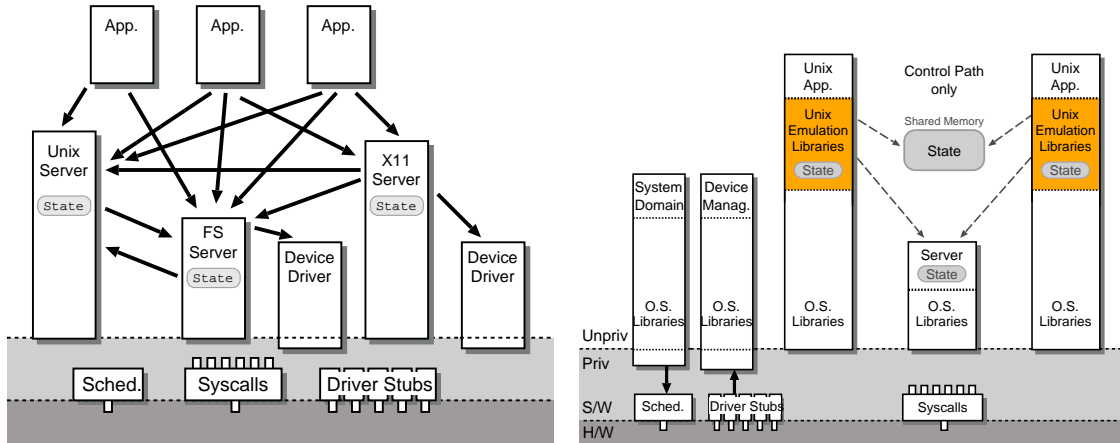


Figure 2: Options for a Unix implementation: (a) server based, and (b) library based

server implementing the Unix kernel on behalf of its clients. This would introduce a potential point of congestion, preventing the provision of QoS to ported applications.

Instead, we decided to base the architecture of the Unix personality on the design principles of the Nemesis operating system by choosing a *vertical* structure (figure 2(b)). The Unix personality is structured as a set of shared libraries implementing different parts of standard Unix functionality.

The advantage of implementing Unix functionality in a set of servers is that there is an obvious place where state shared between different processes is stored – in the corresponding server. If the same functionality is implemented as a set of shared libraries then there are a number of options for maintaining the state: in a shared memory segment, in a small server solely responsible for providing access to the state, or by allowing applications read-only access to a shared memory segment and performing all updates through a server. As pointed out in the diagram it is important, that the shared state is only accessed in *out of band* control operations and is not part of the data path of operations; this is necessary if the implementation is to provide QoS to individual applications.

A traditional Unix kernel maintains many data structures, such as file descriptor tables, file table, process table, etc. Access to them is protected through kernel code. Some shared state is essential to support the full and correct semantics of a Unix system and hence a full library based implementation has to emulate protected access to this state. The authors are aware of only a few library based implementations: OpenBSD on the Exokernel system (Kaashoek et al. 1997), UWIN, a port of the Unix API to Windows NT (Korn 1997), and the very similar Cygwin32 system by Cygnus Solutions (Noer 1998). These pursue slightly different approaches. UWIN and Cygwin32 maintain shared state in a shared memory segment. Access to it is not protected from accidental or deliberate alteration or corruption. As a small “security” measure, the region of shared memory is mapped into an address range far away from the data segments of the programs so that accidental corruption of the data is less likely. In the Exokernel system, most shared data structures are protected using hierarchically-named capabilities (Mazières & Kaashoek 1997) while others are kept in unprotected shared memory. In all three systems the shared data structures are modelled after their Unix equivalents. These are still potential sources for application QoS-crosstalk since they are accessed and modified during data-path operations.

Throughout this paper, we argue that this shared state can be greatly reduced and even completely elided provided one is not attempting to provide *full* Unix functionality; the lack of shared state has the benefit of enabling Nemesis’ resource guarantees to be made to individual applications. Our current implementation maintains no shared state and provides a reasonable and workable subset of Unix functionality, sufficient to allow a wide range of applications to be ported. In the subsequent sections we describe how this was achieved and which restrictions we impose on Unix applications wanting to use the Unix subsystem.

2.1 Components

Although there are restrictions, as described above, the general design of the system is flexible enough to model almost complete Unix functionality. This is achieved through a highly modularised design of the libraries implementing the Unix API; the Unix functionality is built from a number of components, each implementing a separate part of the functionality. For example, we have components implementing basic `libc` support, a file descriptor table, and the `stdio` API. The linkage between these components is based on closures, modelled after the Nemesis default programming model (Roscoe 1994). Each function call is passed a closure³ in addition to its standard parameters. This forms an interface around the component which abstracts from its actual implementation by encapsulating its state and effectively makes the standard Unix API “object-based”. To achieve source code compatibility (the standard Unix API is not object based) we deploy the stateful API architecture developed as part of the deliverable 4.5.1; applications are statically linked with small stub libraries, converting standard API calls into object-based ones (see (Neugebauer & Black 1998) for a detailed description).

At compile time, application source code is linked only against these small stub libraries, which essentially is a linkage against the interfaces of the components. At runtime each component is initialised using a “proper” Nemesis interface defining the “constructor” method for that component. The initialisation acquires a reference to the constructor interface through the standard Nemesis namespace. This constitutes a dynamic linkage at runtime between an interface and a particular implementation of it.

The use of different, self contained components with a flexible linkage model allows us to replace the functionality of individual components without re-compilation of applications. Moreover, it allows applications to be executed in different run-time environments. Consider a set of components implementing the same interfaces modelling process creation and process relationships. One could implement a simplified model which doesn’t maintain any shared state between different processes and hence doesn’t provide full Unix functionality. A second implementation may implement a process table as found in classic Unix kernels in a small server process. For most applications it doesn’t matter against which of these components they are linked at run time. A few applications, e.g., command shells, require a more complete implementation of the processes control functionality and would choose the latter of the two implementations. Processes which are created from these applications would then have to use the implementation used by its parent process.

This not only enables us to evolve the components independently⁴, but also allows us to provide the minimal runtime environment for individual applications while still allowing them to execute in richer environments without alteration or re-compilation. This could be useful where applications are being developed and tested in resource rich environments to run in environments with limited resources, e.g., set-top boxes.

The next sections describe the individual components, depicted in figure 3, of the Unix personality in detail. Section 3 describes a component providing basic support to run POSIX programs under Nemesis. It can be considered as much a part of the C runtime environment as part of the Unix environment. Section 4 describes the general file I/O implementation. Section 5 describes the POSIX process model and how it is currently implemented in the Unix personality. Section 6 presents our design of the component providing terminal I/O functionality. Unix’s Inter Process Communication primitives and how our design includes support for them is described in section 7. Finally, section 8 presents the design for networking support in the Unix personality.

Not all of these components are fully implemented yet. Some of the implementations are not fully compliant to the Unix standard. In these cases we describe in detail how the full functionality could be implemented. However, our experience with porting software to the Unix personality (see section 9) indicates that the functionality provided so far is sufficient to support a wide range of applications and tools.

³A record containing a reference to a vector containing pointers to the other functions of the component, the method suite, and a reference to the state of that component.

⁴This in itself is a significant software engineering advantage over the maintainence effort required for standard monolithic `libc` implementations such as the 19 Megabytes of source code in `glibc2`.

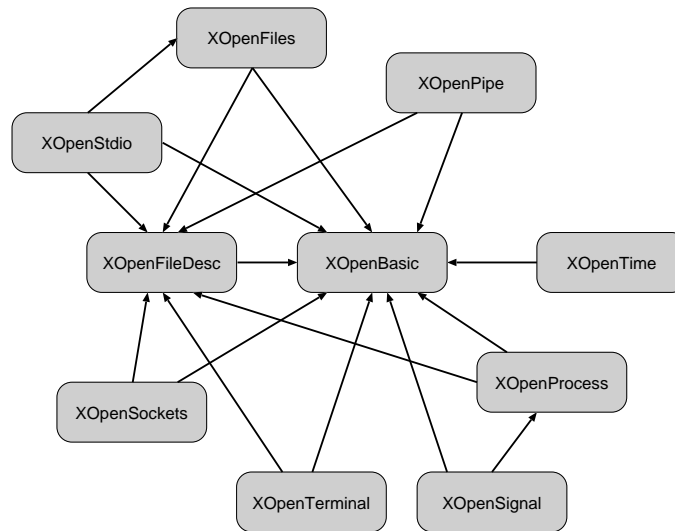


Figure 3: Components and their relationships

3 Basic support

Unix programs require a set of standard functions and facilities which are not provided by the C runtime environment as implemented by the `libc` veneer library under Nemesis. These include support for `errno`, environment variables and argument handling. A component, `XOpenBasic`, has been implemented to provide this basic support to programs. There are two interesting issues which are solved with this component: access to a standard set of global variables such as `errno`, and the passing of parameters such as the arguments or the environment from one process to another. Both are described below.

As argued in (Neugebauer & Black 1998), access to process global variables from shared code is particularly difficult to manage in a single address space operating system. The stateful API architecture provides the framework which was used not only to implement the mapping from the standard API to a closure based implementation (see section 2), but also to provide access to process global variables. Global variables are declared in the statically linked stub library which provides the mapping from standard API function calls to the closure based implementation. This allows application code to access them since the linker can resolve references to them at compile time.

However, shared code contained in components needs to access these global variables as well. Reference to global variables in shared code cannot be resolved at compile time by the linker. Instead, pointers to the per process instances of global variables are passed into the component at runtime through the components constructor method. Shared code then can access the global variables indirectly.

Each global variable is associated with exactly one component – they are considered to be part of the component’s private state. The `errno` variable, for example, is associated with the `XOpenBasic` component. Other components need to access this variable in order to indicate error conditions. For this purpose, the `XOpenBasic` component implements a set of methods allowing other components to access the `errno` variable. In order to invoke these methods the other components need to hold an object reference to the `XOpenBasic` component. These object references are passed in as arguments to the constructor methods of components; thus defining dependencies between components. The resulting dependency graph is acyclic and relatively small and can easily be managed manually. Figure 3 illustrates the components with arrows indicating dependencies between them.

The second problem, the passing of parameters from one process to another, arises from the fact, that in Nemesis, by default, very little information is passed from the creator of a new process to the newly created process. However, rather than using the inflexible mechanism provided by a Unix system, the creator of a new process can freely configure the entire namespace of the new process – thus the creator has complete control over the initial runtime environment of the new process. Since this mechanism provides a powerful but also very basic mechanism, a process *Builder* is provided as part of Nemesis which facilitates

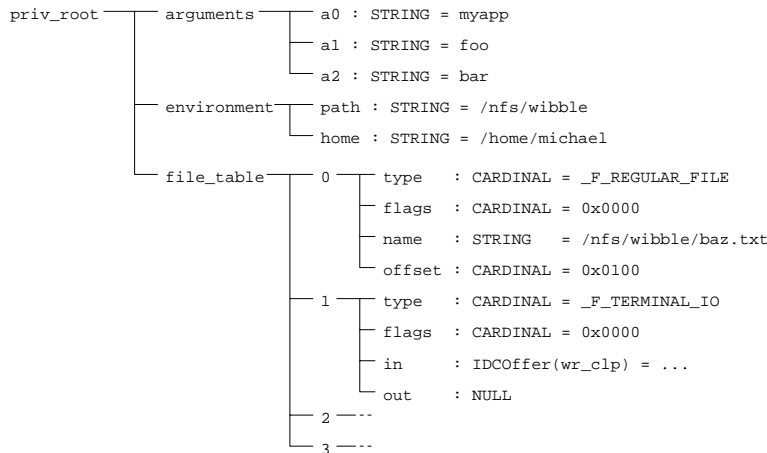


Figure 4: Namespace passed from parent to child

the creation of a new process and its runtime environment by implementing a standard set of *conventions* on top of the basic mechanism (Roscoe 1995).

We use functionality of the Nemesis *builder* to pass the arguments and environment variables from the parent process to the child process. In the parent process, we store the arguments and the environment variables in a particular namespace, known as `priv_root` which is deep-copied and then mapped into the child's namespace.

When a child process starts up in the Unix personality, it first initialises all its components and then extracts the arguments from the namespace (see figure 4 for a sample namespace). This is performed in the equivalent of `crt0.o`, i.e., before the `main()` function is executed. The arguments, obtained from the namespace are then simply put in the standard `argv` data structure and the `main()` function of the program is called. When the `XOpenBasic` component is initialised, it extracts the environment variables from the namespace and sets up the environment pointer. This mechanism is also used by other components to pass information from parent to child process (e.g., see section 4.1).

When the call to `main()` returns, `exit()` is called. This first executes all functions registered by applications using the `atexit()` and `on_exit()` API calls, then performs general cleanup functions such as freeing up resources before terminating the process. Components which require component specific cleanup function to be called can register functions with `on_exit()` when they are initialised. Since functions registered with `atexit()` or `on_exit()` are called in reverse order during `exit()` it is guaranteed that each *component destructor function* is called after all functions registered by the application have been executed and before the destructor function of any component on which it may rely.

4 File Input and Output

Processes and files are the most important abstractions provided by a Unix system. Processes are described in the next section, in this section we focus on files. Files are not only used to provide persistent storage for data in a filesystem, but also for terminal access, general device access, inter process communication, and networking. Apart from providing a powerful abstraction for general I/O related tasks, files also form an important part of a process' context. In this section we describe how the file abstraction is implemented in the Unix personality, focusing mainly on regular files. We also describe our implementation of the standard I/O library which is implemented using the file abstraction of the Unix personality. The relationship between files and processes is described in section 5.1.1. In subsequent sections we describe how other types of files fit in our file framework.

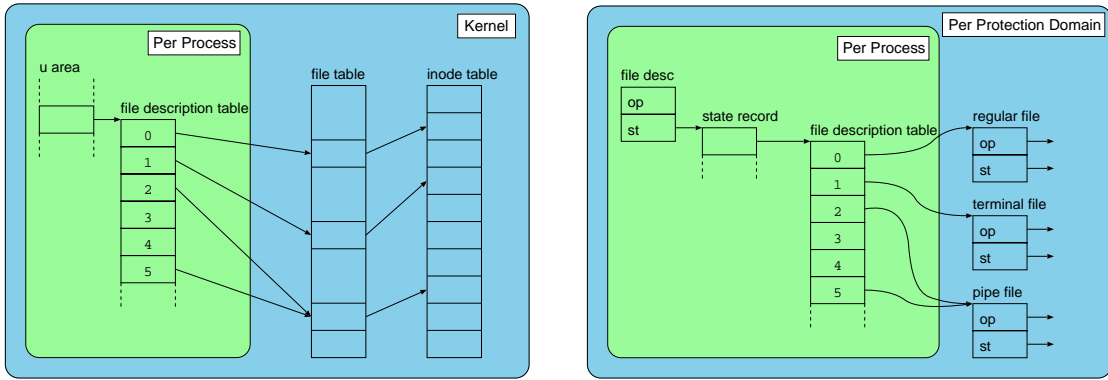


Figure 5: (a)File Descriptor Tables in Unix and (b)XOpenFileDesc

4.1 File Descriptors

In a Unix system user processes can interact with a wide range of different type of files through a well-defined, procedural interface. The interface exports a small number of abstractions to user processes: *files*, *directories*, *file descriptors*, and *file systems*. The abstraction which “glues” these together is the file descriptor. Each file and directory opened by a process is represented through a file descriptor, a small positive integer. The kernel maintains a table of open files for each process, known as the *file descriptor table*. Entries in these process private tables map from the per-process file descriptors to a kernel data structure shared between all processes, known as the *file table*. In this table information about all open file objects in the system is maintained, for example access flags (e.g., read and write access) and a current offset in the corresponding file. Multiple entries in a single file descriptor table may refer to the same entry in the file table (through the use of the `dup()` system call) or entries from different processes’ file descriptor tables may refer to the same entry in the file table (through child processes inheriting the file descriptor table of their parent process). Entries in the file table refer to the actual file objects which, in older implementations, are represented by *inodes*, holding information about the physical location of a file on a given file system (Bach 1986). The relationship between these objects is illustrated in figure 5(a). In most modern variants of Unix the inode level has been replaced by an more object oriented *vnode/vfs* interface (Kleiman 1986).

One possible way to implement the functionality of the file descriptor table is to mimic the data structures described above in the Unix personality. However, we do not wish to have shared state (as represented by the file table) involved in data path operations of file I/O. In a Unix system, the data maintained in the file table is updated on almost every access to a file. This potentially introduces QoS crosstalk between different processes. Instead, we implement a per-process file descriptor tables – thus providing the semantics of the Unix API. Access to the table and to the objects it contains is performed through an interface implemented in the `XOpenFileDesc` component. Instead of containing references to entries in a global file table, our file descriptor table contains direct references to generic file objects similar to the object oriented *vnode/vfs* architecture.

This architecture is illustrated in figure 5(b). The `XOpenFileDesc` component maintains, as part of its state, the file descriptor table. Entries in this table are references to file objects of different types, such as regular file objects, pipes, terminals, or sockets. The file descriptor table is a per process data structure whilst the file objects are maintained for each *Protection Domain*. This allows file objects to be shared between processes executing as threads in the same protection domain (see section 5 for details).

Each of the different types of files is required to implement a common interface, on which the code in the `XOpenFileDesc` component operates. Different types of files may implement other interfaces to support file type specific operations. The `XOpenFileDesc` component implements all common operations on files, such as `read()`, `write()`, or `close()`, by providing a mapping from an invocation using a numeric file descriptor to a method invocation on the appropriate object. Consider, for example, an application calling `read()`. The implementation of `read()` in the file descriptor component retrieves the object reference

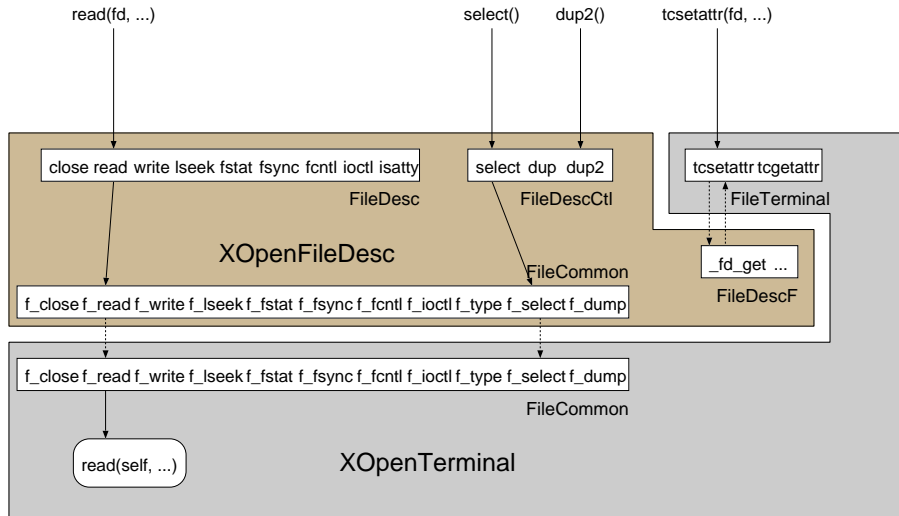


Figure 6: Code Structure for File Descriptor Table and File components

corresponding to the file descriptor from the file descriptor table. It then invokes the `f_read()` method on the common file interface implemented by the retrieved file object. This process is illustrated in figure 6. For simplicity, this diagram only contains the `XOpenFileDesc` component and the `XOpenTerminal` component. The latter provides an implementation of the common file interface for terminal I/O as described in section 6.

In addition, the `XOpenTerminal` component also implements an interface providing terminal file type specific operations, e.g., `tcsetattr()`, for setting terminal attributes. File type specific operations on files differ from the implementation of the common operations on files. They are entirely implemented by the components providing support for that file type. The file descriptor component provides an internal interface (labelled `FileDescF` in figure 6), to support these other components. Analogous to the description of the `read()` function call, consider an application calling the terminal specific function `tcsetattr()` for an open terminal file. Its implementation in the `XOpenTerminal` component first retrieves the file object corresponding to the file descriptor from the `XOpenFileDesc` components using the internal `FileDescF` interface. It then checks if the retrieved file object is a terminal file using the `f_type()` operation of the file’s common file interface. If the retrieved file is a terminal file, the terminal attributes are set and the function returns. If it is not a terminal file, an error is returned.

The `XOpenFileDesc` component also implements some file descriptor table management operations. The `dup()` and `dup2()` operations are easy to implement by simply copying the references to file objects to new entries in the file descriptor table (`dup2()` may additionally need to invoke the close operation if the new entry is already in use).

The `select()` system call takes a group of file descriptors as arguments and only returns if data becomes available on one of the file descriptors or a specified time-out is reached. This functionality ought to be relatively easy to implement; but there is a complication. In Nemesis, all basic components, including the basic kernel interface, the I/O system and the Inter-Domain Communication (IDC) system, were designed to operate asynchronously using *events* as the communication mechanism. An implementation of `select()` could simply block on a number of asynchronous communication endpoints, corresponding to the file descriptors passed in as arguments, waiting until data becomes available or a timeout is reached. Unfortunately, the access to the low level, asynchronous mechanisms is “hidden” behind higher level interfaces. We are currently investigating, whether it is feasible to regain access to the low-level mechanisms. If this is infeasible, we intend to implement `select()` following the approach described in (Noer 1998). Upon entering the function, each of the files is polled to discover if data is available. If data is available, the function simply returns. The more complex case involves waiting for data to become available. This can be implemented by blocking the main thread after starting separate threads for each file descriptor; these

new threads poll the corresponding files until data becomes available or until the time-out is reached. Once one or more threads find data available the main thread is unblocked and reconsiders the file descriptors.

4.2 Files and Directories

In the previous section, we described how the general file I/O framework based on file descriptors is implemented in the Unix personality. In this section the handling of regular files and directories is described.

Nemesis currently provides access to file systems through the `FSDir` and the `FileIO` interfaces. These interfaces have been implemented for access to NFS, the Linux native `ext2` file system, and a simple file system known as `sfs`. The `FSDir` interface provides access to the meta information of files and directories and allows the manipulation of files, such as creating, deleting, and opening of files and directories. An extended interface, `FSDirUnix`, provides access to additional meta-information only relevant for Unix file systems. Data path operations are performed using the `FileIO` interface returned by the `open` operation.

Access to these facilities from within the Unix personality is provided by the `XOpenFiles` component. It implements the API for accessing files and directories as defined by (ISO/IEC 1996, Section 5). Most of this functionality is implemented as small wrappers around calls to the `FSDir` interface.

The Unix API for performing file I/O is implemented in a file object providing the common file interface used by the file descriptor table. For regular files this generic interface has been extended in the fashion described above to implement function calls which are specific to regular files, such as `fchmod()`. The standard file I/O operations, such as `read()`, `write()`, and `lseek()` are implemented as small wrappers around calls to the `FileIO` interface.

4.3 Standard I/O

The Unix interface for performing file I/O is based on the numeric file-descriptors which map to open files in the file-descriptor table. The C Standard (ISO/IEC 1997) defines a stream based I/O model which is known as Standard I/O or `stdio`. In a Unix system, this is naturally implemented using the Unix file abstraction. Standard I/O provides buffered output, a set of standard streams, and a wide range of formatting facilities by means of which output can be formatted and structured input can be read more easily. In total, the 4.4 BSD version of the Standard I/O library defines over 60 functions, implemented with over 6000 lines of code.

Rather than implementing the standard I/O library ourselves, we decided to port an existing implementation for the Unix personality. We choose the 4.4 BSD implementation because of its relatively clean design. We also considered the GNU and the Linux version, but these appeared more complicated in design and implementation. In an initial attempt, we simply compiled the source into a statically linked library. This compiled and worked with only minor source code modification⁵ necessary. We then fitted the implementation into the *Stateful API Architecture* to provide standard I/O functionality as a shared library to the Unix personality. This required the identification of the library's internal state, the change of all internal function calls to closure based invocations, and the generation of a statically linked stub library to provide the standard API. Due to the relatively clean design, this task was accomplished in a couple of days, and is now contained in the `XOpenStdio` component.

The C Standard defines a set of standard streams to be present at program start. These are streams for standard input (`stdin`), standard output (`stdout`) and the standard error stream (`stderr`). During the initialisation of the `XOpenStdio` component, the file descriptor table is checked for open files with the corresponding file descriptors (`STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`). If these are found, the standard streams are simply set up. If the parent process didn't provide these files, e.g., it was a standard Nemesis application, we default the standard streams to the basic input and output streams provided for every Nemesis application.

The early port of the `XOpenStdio` component has proven to be very helpful in spotting incorrect or incomplete implementations of the underlying file I/O implementation and is necessary for porting even simple Unix programs. In addition, having ported the mature BSD 4.4 implementation guarantees that the full standard is implemented.

⁵For example, to replace some BSD specific functions such as `writenv()` and to conditionally exclude floating point operations not supported on some Nemesis platforms.

5 Processes

A process is an active entity executing one or more sequences of instructions in an address space. The address space comprises a set of memory locations a process may access and is implemented as a private virtual address space. Processes are protected from each other and are run on a private virtual machine. The kernel maintains a number of shared data structures, such as the *process table*, which models relationships between processes (parent – child, process groups etc.), and the *u area* containing process specific information such as the owner of a process etc. (Bach 1986).

Nemesis does not implement such a process model, i.e., there are no parent child relationships. In Nemesis different aspects combined in the Unix process abstraction are deliberately separated. Nemesis distinguishes between Scheduling Domains (SDOMs), Activation Domains (ADOMs), and Protection Domains (PDOMs). SDOMs are entities the scheduler allocates the CPU resource to. Each SDOM contains one or more ADOMs which the scheduler explicitly activates via an activation vector (Anderson et al. 1992). Based on this mechanism a variety of user-level thread schedulers have been implemented. Usually there is a one-to-one mapping between SDOMs and ADOMs, however, it is anticipated that a SDOM can contain more than one ADOM which have to be scheduled by the SDOM. PDOMs are the entities on which memory protection is enforced. They form the closest equivalent of an address space in a multiple address space operating system. Since there is also, usually, a one to one mapping between ADOMs and PDOMs (in order to provide memory protection between applications) the combined triplet is normally referred to as a *Domain* or *Nemesis Process*.

There are three different ways to sensibly map the Unix process abstraction to Nemesis' domain concept:

1. A Unix process could be formed from a one-to-one mapping between an SDOM, ADOM, and PDOM. This would make individual Unix processes subject to Nemesis' CPU resource allocation. Since each process executes in its own PDOM protection between processes is enforced.
2. A Unix process could be formed from a one-to-one mapping between an ADOM and a PDOM, with multiple processes being managed by one SDOM. This would make a group of processes, i.e., those running in the same SDOM, subject to a single CPU resource allocation. A second level scheduler would have to multiplex the CPU resource given to the SDOM over all processes in some way. Protection between processes is enforced since processes execute in their own PDOM.
3. A Unix processes could execute as a user-level thread within an ADOM. In this alternative, processes are not protected from each other, since all threads within an ADOM have to execute in the same PDOM. The advantage of this alternative is that it incurs very little overhead on process creation.

Each of the alternatives has its tradeoffs. Option 1 is appropriate for compute intensive batch processing jobs and multi-media applications. Resources can be managed on the granularity of individual applications. Option 2 is interesting for “normal” applications and login sessions since it provides a mechanism to allocate resources to a group of processes but still enforces protection between them. One could imagine a complex, long running make job executing in such an environment, limiting the resources the whole job receives. Option 3 is ideal for small system utilities, such as `ls` or `cat`, which typically run only for a short period of time and are sufficiently bug free that the lack of protection between processes can be neglected. Another possible use is for server daemons which create new processes to service each incoming request. In an multi-threaded environment it is preferable to create threads; option 3 would allow us to achieve this without changing the source code of the traditional server daemons. Since each of these options has its benefits, we decided to provide support for all three in a flexible manner (see section 5.2 for details). Unfortunately, in the current implementation of the domain concept under Nemesis, ADOMs and SDOMs are not separated.⁶ As a result, option 2 is currently not available in our Unix personality implementation. We are convinced that, should the separation between ADOM and SDOM become available, support for it in the Unix personality can be implemented straightforwardly.

⁶Neil Stratford of the Computer Lab at Cambridge separated the two domains in a proof-of-concept implementation in early 1998. Unfortunately his implementation didn't make it into the mainstream Nemesis code.

5.1 Process Semantics

In POSIX compliant systems, processes are created using the `fork()` system call; this creates a new process which is an almost⁷ identical copy of the original process. This copy is composed of the data structures describing the process and its relationship with other processes as well as its address space. The latter posed a particular performance penalty on early implementation of the BSD variant of Unix since it was implemented by making physical copies of all pages of the parent process. For this reason BSD variants of Unix also provide the `vfork()` system call which executes in the same address space as the parent (Bach 1986). After being created with `vfork()` the child is expected to call `exec()` or `exit()` shortly after. During this period the parent is blocked and the address space is shared. Modern variants of Unix typically perform a copy-on-write of the parent's address space for the child, improving the performance of `fork()` significantly. The `exec()` system call replaces the current process image with a new process image; the new image being constructed from a file specified as an argument. The process state remains unchanged with the exceptions specified in (ISO/IEC 1996, Section 3.1.2).

As argued in section 2, it is impossible to model the full `fork()` functionality in a single address space operating system. The definition of `fork`, producing an exact copy of the parents address space, inherently assumes multiple private address spaces, one for each process. An implementation of `vfork()` is potentially possible in a single address space. The child could execute as a thread in the protection domain of the parent process while the parent process is blocked. However, most programs typically call `fork()` closely followed by a call to `exec()`. This had been observed in the past (Korn 1997, Noer 1998) and led to the definition of the `spawn()` call (Delorie 1997) in environments where the implementation of the `fork()/exec()` system calls for process creation is either impractical or impossible. We decided, that support for the `spawn()` system call is sufficient for our purposes and implemented a variant of it for our Unix personality (See section 5.2 for details).

5.1.1 Information Inheritance

In a Unix system the per process file descriptor table is copied from a parent process to the child process during the `fork()` system call; thus the child process inherits the open files of the parent process. In the Unix personality, we use the namespace to pass information from the parent process to the child process as described in section 3. As argued in section 4.1, the Unix personality doesn't deploy a shared file table to allow the direct sharing of opened files between child and parent process. Instead, we are using the namespace to pass information about opened files from parent to child process. The parent process has to provide enough information about open files to allow the child process to reopen the files and place them in its own file descriptor table (see figure 4 for an example).

This is achieved through the introduction of a new context in the namespace of the child process, called `file_table`. It contains an entry for each of the open files of the parent process. The information placed inside the namespace for each file is specific to the type of the file (for example, regular files will not want to store the same information as a terminal would). For this purpose, the interface for the common file operations (see section 4.1) contains a method to store the information necessary to reopen the file in a namespace. Each specific file type has to implement this operation. On the creation of a new process the file descriptor table component traverses the file descriptor table and invokes the method for storing the information in a namespace for each file object in the table. Files with the *close-on-exec* flag set are ignored during this process. When the child process is started, as each component handling a specific file type is instantiated, it will examine the `file_table` branch of the namespace for entries of its type. If an entry is found, the initialiser attempts to reopen the file with the information provided, and, on success, stores it in the file descriptor table using the internal `FileDescF` interface. This mechanism separates the management of file descriptors from the concrete type of the files and thus allows us to introduce new types of files without altering other parts of the system.

The removal of the shared file table from our implementation has another implication. Under Unix, inherited open files are referring to the same file object in the file table. This means, that the files shared between parent and child process also share the same offset in the file. The mechanism described above does not allow this. However, we consider this a minor restriction, since the main use for sharing entries to

⁷See (ISO/IEC 1996, Section 3.1.1) for details.

open files in the file table between different processes is for multiple instances of the same process writing information to a log-file. This is mainly used by server processes forking a new process to service incoming requests. In a multi-threaded environment, however, this type of application behaviour can be much better implemented using different threads for servicing incoming requests.

The process model described above allows for processes being created as threads executing in the same protection domain as their parent process. In this case, file objects may be shared between processes. Instead of storing information allowing the child process to reopen inherited files, object reference to the open files are stored in the child process' namespace. Thus, parent and child process access the same file object; including a shared offset in the file. This obviously raises concurrency issues, which are addressed by protecting the internal state of each file object with a mutex. To prevent inherited files from being closed by one process, while a "related" process is still referencing it, we maintain a reference count in the encapsulated state of each file object. A file is only closed if its reference count equals zero.

5.1.2 Process state

Apart from the file descriptor table the Unix kernel maintains other state associated with each process in the *process table* and the *u area* as described above. These data structures contain information about process relationships, user credentials, process limits, and accounting information.

Our current implementation does not maintain any such state. So far this has not been necessary. We are able to compile most stand-alone Unix applications with little change to the source code necessary. However, we anticipate that, with a wider range of applications being ported, we will have to face the need for more accurate support of the Unix process model.

In our current design we plan to implement a *process table* and an equivalent of the *u area* in a shared memory region allowing all processes read access to these data structures. Write access has to be performed through a privileged server process preventing accidental (or deliberate) corruption of the data. The server won't introduce QoS-crosstalk since write operations to the process table are *out-of-band* operations.

Note that we can easily change the implementation of the process model without affecting the rest of the Unix personality. All process-related function calls are encapsulated in the `XOpenProcess` component, allowing us to locally change their implementation.

5.2 spawn

Spawn works like a combined `fork()` and `exec()`, and is used on platforms where `fork()` and `exec()` cannot be mapped onto lower level functionality. This has been done in both Cygwin32 and UWIN. With both systems it has been demonstrated that `spawn()` is a viable alternative for a wide range of applications. For a description of the `spawn` syntax and semantics see appendix A.

Our `spawn()` implementation currently provides for the creation of two different types of processes as discussed above: child processes are either created as separate domains or as threads executing in the same protection domain as their parents. The first type is typical of how Nemesis applications are launched, while the second type is useful for executing small simple programs with little overhead (e.g., calling `ls` or `cat` from a shell). For both types of processes we implemented synchronous and asynchronous variants – the calling process either waits for the completion of the child process or continues directly after creating the new process.

Because the distinction between domain and thread is not found on other platforms, the standard `spawn()` family of calls offers no technique to allow the caller to specify whether an application it is launching should be created in a new domain or a new thread. A possible solution is to specify an extra flag to pass to `spawn()` which instructs it whether to spawn a thread or a new domain. Whilst this is most flexible it has the disadvantage that each application needs to maintain a list of which programs should be loaded using which method. Under ancient Unix systems a file's *sticky* bit (one of its permission bits) was used as a hint to the virtual memory system to indicate small programs that should have their text segment kept in memory after being loaded for the first time because they are frequently used. This notion of small frequently used programs is what we are looking for too. The `spawn()` implementation checks if the sticky bit of an executable is set; if this is the case, the program will be run as a thread, otherwise, the program is started in a new domain. We provide a set of flags for the `spawn()` function call to override this

default behaviour. We also use flags to specify whether new processes should be created synchronously or asynchronously.

As described in previous sections, the Unix personality uses the namespace to pass information from parent to child processes. During the execution of the `spawn()` call, the namespace of the child process is constructed. This includes arguments to the `main()` function of the child process and environment variables as described in section 3, as well as the information on opened files to be inherited by the child process as described in section 5.1.1.

Unix programs frequently manipulate the file descriptor table between the calls to `fork()` and `exec()`, for example, to redirect the standard output of the process to a file. The standard `spawn()` interface does not allow the application to specify this type of behaviour. For this particular purpose, we introduced new members to the family of `spawn()` calls which allow us to perform the necessary manipulation of the inheritance of open file from parent process to child process during the execution of `spawn()`. For example the `spawnv()` call was extended to:

```
int spawnvN(int mode, const char *path, const char **argv, int posc, int posv[]);
```

Two extra arguments, an array of integers, `posv`, of length `posc`, were introduced. The array `posv` represents the child processes file descriptor table. The value for each entry specifies which of the parent processes file descriptors should be mapped to this entry. This information is passed to the `XOpenFileDesc` component when it is requested to store its information in the namespace of the child process. Other variations of `spawn()` were extended in the same fashion.

5.3 Signals

In Unix, signals are used as a primitive mechanism of interprocess communication and synchronisation between processes. The process of signalling is divided into two phases – signal generation and signal delivery. Each signal has a *default action* which is performed on delivery. For some signals, applications can register signal handler functions which are executed instead of the default actions. Some signals are delivered as asynchronous events originating from other processes (e.g., generated with the `kill` command) while others are delivered synchronously (e.g., on accessing an illegal address). A process may block certain signals temporarily in which case the signal will not be delivered until it is unblocked. All signal actions can only be taken by the receiving process itself, which means that the receiving process has to be scheduled. A signal will interrupt the normal execution of the running process as soon as it is unblocked. The receiving process then executes the signal handler, if installed, on the same stack as its main thread of execution and resumes execution at the point where it was preempted.

We implement support for signals in the `XOpenSignals` module. Currently, we only deal with synchronous intra-process signals, i.e., signals generated by an application and destined for the same process are handled. We plan to deliver signals between different processes using Nemesis' IDC mechanism. The receiving process handles IDC calls in a separate thread. This thread can perform the default actions such as *abort*, *exit*, *ignore*, *stop*, and *continue*. The latter two actions can simply be implemented by suspending and resuming the main thread of execution. The first two actions can also easily be implemented by exiting the process. The IDC system is currently being extended to include security features⁸ which we plan to use to authenticate the sender of a signal.

Note that this design does not allow for application specific signal handlers to execute on the same stack as the main thread of execution. This could be accomplished in a similar fashion as described in (Khalidi & Nelson 1993)⁹, where a helper thread is used to manipulate the stack of the main thread, so that it executes application specific signal handlers. However, we regard this particular aspect of signalling under Unix as fundamentally broken. For example,¹⁰ consider the case where a signal gets delivered to a process while it is allocating memory on its heap using `malloc()` and the signal handler also needs to call `malloc()`. This sequence of events may result in inappropriate concurrent re-entry of the heap while in an inconsistent state. Similar problems can occur when both the process and the signal handler use functions which are not reentrant.

⁸This work is being undertaken by a Research Student at Cambridge.

⁹Other non genuine Unix systems follow similar approaches, e.g., (Korn 1997, Briceño 1997, Noer 1998).

¹⁰Taken from (Stevens 1993, Chapter 10.6)

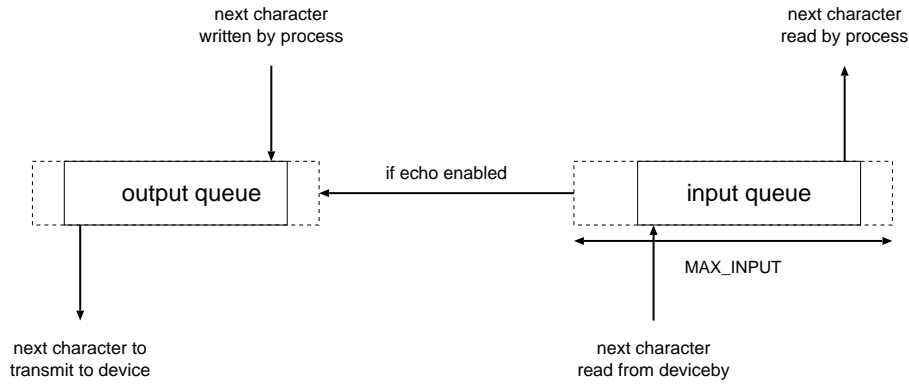


Figure 7: Schematic terminal device

We therefore conclude that an implementation in which signals are handled by a separate thread is sufficient for our purposes, i.e., it will cover the majority of uses of signals.

5.4 Session management

Unix provides the notion of process groups to control terminal access and to support login sessions. Historically, there have been big differences in the implementation of terminal control in the various flavours of Unix. However, there are common concepts between the different implementations as identified by (Vahalia 1996). Each process belongs to a *process group* which is identified by the *process group id* `pgrp`. A group may have a *group leader*, which is the process whose process id is same as its process group id. Each process may have a *controlling terminal*. All processes in the same group have to share the same controlling terminal. Each terminal is associated with one process group at a time, called the *controlling group*. Keyboard generated signals, such as `SIGINT` and `SIGQUIT` are sent to all processes of this group by the terminal device driver. A *job control* mechanism manages access to a terminal between process groups.

Modern flavours of Unix, such as SVR4 or 4.4 BSD, introduced the concept of a *session*, solving a variety of problems with earlier implementations of process management. Jobs and sessions are separated but related abstractions. A *session object* represents a login session while a job represents a process group. Each process belongs to a session and a process group. Each controlling terminal is associated with a session and a foreground process group. A *session leader* is responsible for managing the login session. The foreground process group has unlimited access to the terminal and acts as the terminal's controlling group. Both, SVR4 and 4.4 BSD, provide comparable powerful session management functionality, although there are subtle differences in the implementation.

We do not intend to implement session management as outlined above. It is of little research interest. We anticipate that basic job control support, as offered by earlier versions of the various variants of Unix such as SVR3 or 4.3 BSD, could be added to the system relatively easily. However, we argue that this issue should be addressed for the Nemesis operating system as a whole and not just within the Unix personality.

6 Terminal I/O

Terminals perform two distinct function in a Unix system. They are responsible for performing character based input and output, offering a standard file abstraction to clients, and they play an important role in the implementation of session management. This section is concerned with the I/O aspect of terminals. Terminal I/O is a complex subsystem of every Unix system. Part of this complexity can be attributed to the wide range of devices supported, ranging from physical terminals, hardwired lines between computers, modems, printers, virtual terminals such as provided by a windowing system, and so on. For the implementation of terminal I/O for the Unix personality we can simplify this complexity considerably, since it is not our aim to provide the full range of terminal devices a Unix system typically supports.

In figure 7 (adapted from (Stevens 1993)) a simplified view of a terminal is given. It consists of two queues: an output queue to which a process may write characters and an input queue from which a process may read characters. The device driver transmits characters from the output queue to the device and reads characters from the device into the input queue. If a local echo is enabled for the terminal, there is an implied link between the input queue and output queue; input characters are immediately transmitted back to the output queue when they are received from a terminal.

There are two modes of operation for terminal I/O:¹¹ canonical and non-canonical mode. In canonical mode, terminal input is processed as lines and the terminal driver returns at most one line per request. In non-canonical mode the input characters are not assembled in lines. The default mode is canonical mode, but some applications, e.g., full screen editors, use non-canonical mode, since they need to manage the whole screen and need to process individual characters.

The terminal driver processes the input for the occurrence of special input characters. The POSIX standard (ISO/IEC 1996) defines 11 such input characters, 9 of which can be changed by an application. Different variants of Unix add more special characters to this. The actions taken by the device driver may depend on the current mode of operation of device (e.g., *erase* and *kill* characters are not processed). A list and a detailed description of the special input characters may be found in (ISO/IEC 1996, Section 7.1.1.9).

Implementing basic terminal I/O support for the Unix personality is relatively straightforward. The Nemesis client render windowing system (Black 1998) provides a basic text window known as `WSterm`. This provides a raw character stream interface in form of a Reader/Writer abstraction which performs no processing on the streams. A second module, `CLine`, is used to perform processing of the input stream. It essentially provides a canonical mode to raw “terminals”. We are currently extending `CLine` to implement a more generic terminal abstraction. As an initial step we have already added a control interface `CLineCtl` to enable and disable local echo. More functionality for this interface will be added to allow for switching from canonical to non-canonical mode and to redefine the special input characters. This functionality is provided in the form of standard Nemesis interfaces so that non-Unix applications can use these features as well.

The mapping from this generic terminal functionality to the Unix API calls is implemented by the `XOpenTerminal` component. It performs two tasks. First, it implements a generic file object for the file descriptor table providing standard file I/O operations such as `read()` and `write()` for terminals as described in section 4.1. Secondly, it provides the API for setting the modes of operations and the attributes of the terminal “driver” such as `tcsetattr()`. These are simple wrappers around the `CLineCtl` interface.

As argued in section 5.4, we currently don’t intend to provide session management for the Unix personality. If this were to be implemented, further changes to the `XOpenTerminal` component would be necessary. At the very least, the terminal module would need to be able to find out about the controlling process group it is associated with. It would also need to be aware of some session related information, since it would have to send signals to processes not in the controlling process group which attempt to access the terminal.

We do, however, provide a basic facility, whereby processes of the same process group share the same terminal. For this purpose, we make the Nemesis interfaces to the terminal (`CLine` and `CLineCtl`) available to all child processes of the group leader. In the case where the child processes are threads within the same domain we simply pass the interface references to them. In the case where the child processes are executing in a separate domain, the interface is exported to and bound by the child process using the standard IDC mechanism provided by Nemesis. Since this doesn’t implement any job or session control it is similar to the implementation of terminal management in SVR3 (Vahalia 1996, Section 4.9.2).

Access to the controlling terminal via the alias `/dev/tty` would require some support from the `XOpenFiles` component, since it implements the `open()` API call. The implementation of `open()` would need to be extended to deal with this special case, in which it would have to acquire a reference to the file object representing the controlling terminal from the `XOpenTerminal` component.

As a reference terminal driver, we implemented a generic terminal program `genterm`, which uses the `WSterm` and `CLine` modules to create a terminal window. As a command line argument it can take any Unix application, which is executed using the `spawn()` system call (see section 5.2). If no argument is given it currently defaults to a shell (`ash`, see section 9.2). This default behaviour could be altered, so that

¹¹BSD based systems support three modes for terminal input. See (McKusick et al. 1996) for details.

the terminal application would spawn a login process, forming something similar to a standard Unix login sequence.

7 Basic Inter Process Communication

Different vendors' Unix offer a variety of different mechanisms for Inter-Process Communication (IPC). Examples are: pipes, FIFOs (named pipes), sockets, stream pipes, named stream pipes, message queues, semaphores, shared memory, and streams (see (Stevens 1993) for a comprehensive description). The most widely supported mechanisms are pipes. Our design for implementing them for the Unix personality is described in this section. We believe that the other IPC mechanisms could be implemented easily but currently have had no requirement to do so.

Pipes are the oldest form of Unix IPC. They provide simplex communications channels between processes. This means that one end of the pipe can only be written to, and the other end can only be read from. Pipes can only be used between processes that have a common ancestor since they use the mechanism of "inheriting" the file descriptor table from parent to child to establish the end-points of the communication. The first restriction was lifted with the introduction of *stream pipes* provided by modern releases of System V and BSD Unix. FIFOs solve the second limitation by storing communication endpoints as special files in the filesystem.

The `pipe()` call returns two file-descriptors, one for read and one for write access. A pipe in a single process is pointless. Normally the process that creates a pipe calls `fork()` creating a channel between parent and child. Depending on the direction the data should flow, the child closes one of the files returned by `pipe()` (either the write end or the read end) and the parent the other.

A component, `XOpenPipe`, will provide the functionality of pipes for the Unix personality. Pipes will be implemented using a shared memory region as a fifo with two associated event counts operating both as synchronisation primitives and as indicators for the head and tail end of the fifo, similar to the fifos deployed in the RBuf mechanism used in Nemesis for stream bulk data transfer (Black 1995). This fifo will be presented to the Unix personality with the common file interface as described in section 4.1.

Pipes will be inherited from parent to child process in different ways depending on whether or the child process is executing as a thread in the same protection domain as the parent. If the child is executing as a thread, then the event counts marking the head and tail of the fifo can be simply shared between parent and child process. If the child is executing in a different domain, the event counts will be replaced by an *Event Channel*.

The typical scenario to create a pipe between two processes can be implemented using the `spawn()` call and the *close-on-exec* flag. The parent would create a pipe and set the *close-on-exec* flag for either the read or the write end. The subsequent `spawn()` call would close that file-descriptor for the child process while passing the other file-descriptor to the child as described above. After the `spawn()` call returns, the parent closes the other descriptor. Since this is a common operation, the standard I/O library (see section 4.3) contains the `popen()` call, which handles all this automatically.

Support for duplex pipes (also known as *stream pipes*) as supported by modern flavours of Unix such as SVR4 and 4.4 BSD can be added to this design. This would require the creation of two fifos during the `pipe()` call, one in each direction. Both fifos would then be passed to the child process.

8 Networking

Unlike the POSIX standard, the X/Open group standard defines an API for networking. This API is based on the original BSD networking known as *Berkeley sockets* (McKusick et al. 1996, Chapter 11) and is supported by virtually all modern Unix implementations and many other systems. Whilst this interface defines a procedural API, an object based interface providing standard Nemesis interfaces has been designed and implemented for Nemesis (Ahlgren & Voigt 1998). Since the Nemesis interfaces were designed to match the procedural API used on Unix system, we don't anticipate any problems in integrating the BSD sockets framework in the Unix personality.

The `XOpenSockets` component will implement a thin veneer encapsulating calls to the Nemesis interfaces. The network I/O operations would be implemented in a file object managed by the file descriptor

table component as described in section 4.1. As with other file types, inheritance of open “socket” files can be easily achieved, if parent and child process execute as threads in the same protection domain – both processes simply access the same object. We currently don’t have plans for developing a mechanism to inherit open socket files from parent to child process if they execute in different protection domains.

9 Ported software

In this section we describe some of the applications and utilities ported to Nemesis using the Unix personality. These served two useful purposes: firstly, providing a clear demonstration of what had been achieved so far, and secondly, highlighting areas of the API that needed to be completed. Allowing real programs to guide the development helped to identify the commonly used parts of the API which would be required when porting other common applications.

The usual way to launch applications in Nemesis is from the Nash shell. A major problem with Nash is that it works asynchronously. This meant that the initial applications that were ported had no user interaction. Once basic terminal support in the form of `genterm` (see section 6) had been implemented, interactive applications could be demonstrated.

9.1 File utilities

We started with a number of file based Unix utilities since this was the first major component to be implemented. The 4.4 BSD version of `cat` provided the first simple test utility of the Unix file and file descriptor components. It worked without any source code modifications.

The `grep` utility was chosen as a fairly complex and popular utility, which doesn’t require any user interaction or terminal support. It only operates on files and prints its results to standard output. Again, the standard 4.4 BSD source code didn’t need any modification and compiled without problems.

The port of the 4.4 BSD variant of the standard Unix utility `ls` was used as a test case for the directory and file characteristic related API calls. The standard 4.4 BSD source code only needed one minor modification since it produces different a slightly different output format when invoked by root.

We believe that these utilities represent a good set of API calls used by a wide variety of other utilities and applications, and don’t expect any problems porting similar utilities. In particular, we don’t expect problems with other utilities from the standard 4.4 BSD distribution. Our experience to date has shown that these are implemented more cleanly than, say, the GNU or Linux equivalents.

One consequence of our choice of the 4.4 BSD code is that our Unix personality provides a special BSD component, `XOpenBSD`. This implements BSD specific functions, such as the `fts` functions used for traversing file hierarchies, which are frequently used in BSD source code, but are not part of the X/Open standard.

9.2 ash – A simple shell

To evaluate the usefulness of our process creation primitive we choose to port a Unix shell to the Unix personality. Command shells are typically fairly complicated programs utilising and relying on a lot of standard Unix functionality. This includes input/output redirection, job control, signal creation, and initialisation of IPC channels between processes. Given this, we knew from the outset that not all the functionality of a normal shell could be provided as we had no intention of implementing the full functionality. However, two issues made such a port interesting. First, we were interested in how much modification such a relatively complicated program would require to sensibly execute in the Unix personality. Secondly, it would provide us with a test application with which we could test the functionality we intended to add to the Unix personality. In particular, a command shell provides an excellent test application for the `spawn()` call to create processes.

It was decided to port the `ash` shell – a minimal shell mainly developed for running scripts. Because we knew we would have to modify the code base, this simplicity was an appealing factor. Other shells were considered, but they were either much bigger than `ash` (e.g., `bash`) or non-standard in operation (e.g., `esh`).

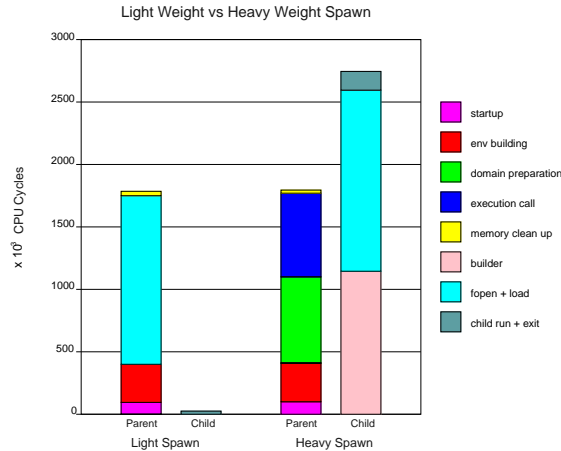


Figure 8: CPU cycles spent in parent and child for a light and heavy spawn

In the first stage of the porting effort, we identified the sections of the source code related to process control, I/O redirection, pipes, etc., and replaced them with stubs that showed what was being called. Once this was done only minor additions were needed to the existing Unix implementation to make the shell compile (e.g., adding missing `errno` error codes). This gave us a shell, with a few basic built-in commands, which is executed in the `genterm` terminal window.

The `ash` shell had to be modified to use the `spawn()` family of function calls for process creation. `Spawn` works slightly different from the `fork()/exec()` technique that `ash` would normally use. By carefully examining the code we were able to replace all instances of calls to `fork()` and `exec()` with calls to `spawn()`. The only change required was to account for the fact that `exec` never returns, whereas `spawn` does. This was solved by doing the same long jump back to the main loop that the error handler does if `exec` fails.

9.3 X Applications

Along with simple text based applications, several X-Window based applications were ported as part of the deliverable 4.5.3. While the purpose of these applications was to test the Nemesis X module, they too make use of the Unix API described in this report. None of the applications required any changes to the Unix related part of their source code. Applications ported include:

- `xli` - Image viewing application
- `mpeg_play` - MPEG video player
- `xpacman` - Simple video game
- `heretic` - 3D adventure game

10 Performance

10.1 Process Creation

In implementing `spawn` it was noted that small programs should be launched in a new thread rather than in a new domain, in order to reduce the process creation/destruction overhead (see section 5.2). Some simple performance analysis of the `spawn` operation was carried out to examine the difference in execution time for the two methods.

An important difference between the `spawn` implementation under Nemesis and an implementation under a conventional OS is that no work is done in the kernel (where process creation is traditionally

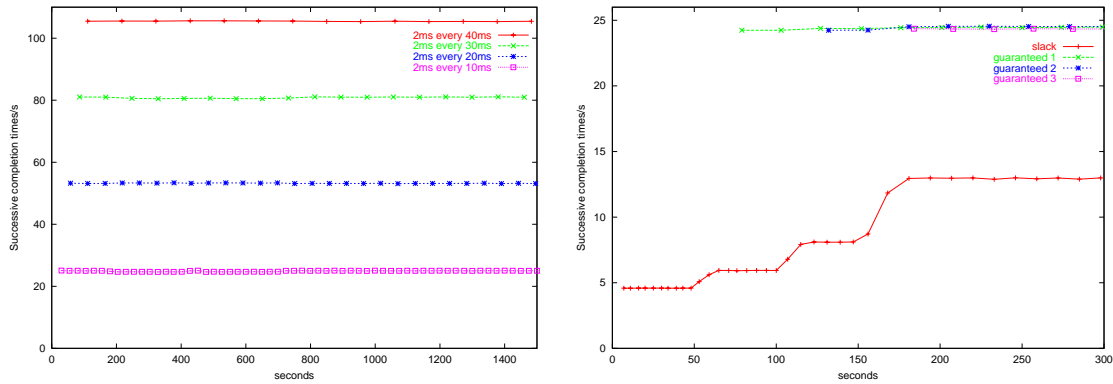


Figure 9: SPEC CPU95 – 099.go completion times: (a) QoS levels, and (b) slack time

handled) – the work is carried out in both the child and parent processes. Because of this, in addition to measure the raw performance of the spawn call, it is interesting to see where that time is being spent.

We implemented a test of process creation performance similar to the ones used in `lmbench` (McVoy & Staelin 1996) and `hbench` (Brown & Seltzer 1997). The test comes in two parts – a parent application and a child application. The child application is a very simple C program which simply returns without doing anything. The parent application repeatedly spawns the child, blocking until the child returns. In our test the parent application is run twice – once with the child being spawned as a new thread of the parent (the light weight spawn) and once with the child being spawned as a new domain (the heavy weight spawn). In order to get a better breakdown of where time was being spent we used more test points than found in either `lmbench` or `hbench`.

The graph in figure 8 shows the results of the tests, with the time spent in the parent and child processes separated for clarity. This supports our original assertion that the light weight spawn carries less overhead than the heavy weight spawn. This can be attributed to the overhead of creating a new domain compared to the overhead of forking a new thread.

It is important to note that in the case of the light weight spawn, the overhead of creating a child process is entirely accountable to the parent process and is dependent on the size of the executable of the new process – the parent process is responsible for loading the executable. For the heavy weight spawn, the overhead for the parent process is constant, since the executable is loaded in the newly created domain.

10.2 Quality of Service

To verify the ability of the Unix personality to provide QoS to unmodified Unix applications we ran an initial experiment based on a “Go” simulator taken from the SPEC CPU95 benchmark suit (Giladi & Ahituv 1995). The simulator plays the game of “Go” against itself in an infinite loop¹². The execution time is CPU bound and it only performs I/O during the initialisation of each game.

We ran two experiments; in each we started 4 concurrent identical copies of the simulator. These instances differed only in the Quality of Service guarantees for the CPU.

In the first experiment, each instance has a fixed guarantee and is ineligible for surplus (best-effort) CPU time. *Successive* times for completion of each instance are plotted in figure 9(a) with each point on each line representing a completion of that particular instance. Clearly the time for an instance to complete one iteration of the simulation is inversely proportional to the guarantee that it has received. What is more significant is that the stability of the completion times shows that the underlying Nemesis guarantees are translating directly into guarantees for the Unix application.

This can also be seen in the second experiment (see figure 9(b)) where multiple instances are started in a phased order. The first instance runs only on best-effort CPU time and subsequent instances have guarantees. It can be seen that as each new instance starts, only the performance of the first instance degrades.

¹²The algorithm is deterministic, i.e., for a given starting situation it always requires the same number of CPU cycles to complete.

These experiments indicate that the use of the Unix personality for deploying this application has not introduced any cross-talk with other processes as would have occurred had a traditional “Unix server” technique been deployed.

11 Conclusion

In this report we have described the design and implementation of a Unix personality for the Nemesis operating system. The main objective, of providing a useful subset of Unix functionality, which allows Unix applications to be ported with relative ease, has been achieved. Where functionality in key areas is currently missing, we presented detailed designs of how an implementation can be achieved. This is supported by a flexible architecture which allows for gradual evolution of the implementation.

Our guiding design principle was to develop a vertically structured Unix personality, avoiding shared state and server processes in time critical data path operations. This was motivated by the desire to make unmodified Unix applications subject to the advanced resource management facilities Nemesis provides to its native applications. With this background we justified the decisions we made, in particular with respect to file I/O, that resulted in implementations with slightly non-standard semantics. We reported that, to date, we have not found these minor differences to be a significant hindrance.

We presented some initial performance measurements based on traditional micro- and macro- benchmarks as well as an initial set of experiments supporting our claim that our implementation allows us to make Unix applications subject to Nemesis resource allocation mechanisms. Clearly a more detailed evaluation is needed and we are currently working on this; our intention is to present this alongside our demonstration of the Unix personality at the end of the project.

References

- Ahlgren, B. & Voigt, T. (1998), IP version 4 on Nemesis, PEGASUS II Deliverable Report 5.1.2, Swedish Institute of Computer Science.
- Anderson, T. E., Bershad, B. N., Lazowska, E. D. & Levy, H. M. (1992), ‘Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism’, *ACM Transactions on Computer Systems* **10**(1), 53–79.
- Bach, M. J. (1986), *The Design of the Unix Operating System*, Software Series, Prentice Hall, Englewood Cliffs, NJ, USA. ISBN: 0-13-201757-1.
- Barham, P. R. (1996), Devices in a Multi-Service Operating System, PhD thesis, University of Cambridge Computer Laboratory. Available as Technical Report No. 403.
- Barham, P. R. (1997), A fresh approach to File System Quality of Service, in ‘Proceedings of the 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)’, St. Louis, MO, USA.
- Black, R. (1998), Client Renders Window System, PEGASUS II Deliverable Report 4.4.1, Department of Computing Science, University of Glasgow.
- Black, R., Barham, P., Donnelly, A. & Stratford, N. (1997), Protocol Implementation in a Vertically Structured Operating System, in ‘The Annual Conference on Computer Networks (LCN)’, Vol. 22, IEEE Computer Society, pp. 179–188.
- Black, R. J. (1995), Explicit Network Scheduling, PhD thesis, University of Cambridge Computer Laboratory. Available as Technical Report no. 361.
- Briceño, H. (1997), Decentralizing UNIX Abstractions in the Exokernel Architecture, Master’s thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.
- Brown, A. & Seltzer, M. (1997), Operating System Benchmarking in the wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture, in ‘Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems’, pp. 214–224.
- Cheriton, D. R., Whitehead, G. R. & Sznyter, E. W. (1990), Binary Emulation of UNIX using the V Kernel, in ‘Proceedings of the Summer 1990 USENIX Conference’, Anaheim, CA, USA, pp. 73–85.
- Delorie, D. J. (1997), *DJGPP: C Library reference manual*.
*<http://www.delorie.com/djgpp/doc/libc-2.01/>
- Giladi, R. & Ahituv, N. (1995), ‘SPEC as a performance evaluation measure’, *IEEE Computer* **28**(8), 33–42.
- Golub, D., Dean, R., Forin, A. & Rashid, R. (1990), UNIX as an Application Program, in ‘Proceedings of the Summer 1990 USENIX Conference’, Anaheim, CA, USA.
- Hamilton, G. & Kougiouris, P. (1993), The Spring Nucleus: A Microkernel for Objects, in ‘Proceedings of the Summer 1993 USENIX Conference’, Cincinnati, OH, USA, pp. 147–159. Also published as Sun Microsystems Laboratories Technical Report No. 93-14.
- Hand, S. (1999), Self-Paging in the Nemesis Operating System, in ‘Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI’99)’, New Orleans, LA, USA.
- Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S. & Wolter, J. (1997), The performance of μ -kernel-based systems, in ‘Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review’, Saint-Malo, France, pp. 66–77.

- ISO/IEC (1996), 'ISO/IEC 9945-1: Information Technology – Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]', International Standard.
- ISO/IEC (1997), 'ISO/IEC 9899:1990 Programming languages – C', British Standards Institute BS EN 29899 : 1993 issue 2. Including amendments 0 and 1 to International Standard ISO/IEC 9899 : 1990.
- Josey, A., ed. (1997), *Go Solo 2: The authorized guide to version 2 of the single UNIX Specification*, The Open Group.
- Kaashoek, F., Engler, D., Ganger, G., Briceño, H., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J. & Mackenzie, K. (1997), Application Performance and Flexibility on Exokernel Systems, in 'Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review', Saint-Malo, France, pp. 52–65.
- Khalidi, Y. A. & Nelson, M. N. (1993), An Implementation of UNIX on an Object-oriented Operating System, in 'Proceedings of the Winter 1993 USENIX Conference', San Diego, CA, USA, pp. 469–479. Also published as Sun Microsystems Laboratories Technical Report No. 92-3.
- Kleiman, S. R. (1986), Vnodes: An Architecture for Multiple Filesystem Types in Sun UNIX, in 'Proceedings of the Summer 1986 USENIX Technical Conference', pp. 238–247.
- Korn, D. (1997), Porting UNIX to Windows NT, in 'Proceedings of the USENIX 1997 Annual Technical Conference', Anaheim, CA, USA.
- Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R. & Hyden, E. (1996), 'The Design and Implementation of an Operating System to Support Distributed Multimedia Applications', *IEEE Journal on Selected Areas In Communications* **14**(7), 1280–1297.
- Liedtke, J. (1993), A persistent system in real use – experiences of the first 13 years, in 'Proceedings of 3rd International Workshop on Object-Oriented in Operating Systems (IWOOS)', Asheville, NC, USA, pp. 2–11.
- Liedtke, J. (1996), 'Toward real microkernels', *Communications of the ACM* **39**(9), 70–77.
- Mazières, D. & Kaashoek, F. (1997), Secure Applications need flexible Operating Systems, in 'Proceedings of the 6th Workshop on Hot Topics in Operating Systems', pp. 56–61.
- McKusick, M., Bostic, K., Karels, M. & Quarterman, J. (1996), *The Design and Implementation of the 4.4 BSD Operation System*, Addison Wesley. ISBN: 0-201-54979-4.
- McVoy, L. & Staelin, C. (1996), Imbench: Portable Tools for Performance Analysis, in 'Proceedings of the 1996 USENIX Technical Conference', San Diego, CA, USA, pp. 279–295.
- Mullender, S. J., Leslie, I. M. & McAuley, D. R. (1994), Operating-System Support for Distributed Multimedia, in 'Proceedings of Summer 1994 USENIX Conference', Boston, MA, USA, pp. 209–220. Also available as Pegasus Paper 94–6.
- Neugebauer, R. & Black, R. (1998), Stateful API Architecture, PEGASUS II Deliverable Report 4.5.1, Department of Computing Science, University of Glasgow.
- Noer, G. J. (1998), Cygwin32: A free win32 porting layer for unix applications, in 'Proceedings of the 2nd USENIX Windows NT Symposium', Seattle, WA, USA.
- Open Group (1994), 'The X/Open Release 4 CAE Specification, System Interfaces and Headers', X/Open, Ltd.
- Phelan, J. M., Arendt, J. & Ormsby, G. R. (1993), An OS/2 personality on Mach, in 'MACH III Symposium, April 19–21, 1993. Sante Fe, NM', USENIX, Sante Fe, NM, USA, pp. 191–201.

- Reed, D. (1998), A new audio device driver abstraction, in 'Proceedings of the 8th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)', Cambridge, UK, pp. 87–90.
- Roscoe, T. (1994), 'Linkage in the Nemesis Single Address Space Operating System', *ACM Operating Systems Review* **28**(4), 48–55.
- Roscoe, T. (1995), The Structure of a Multi-Service Operating System, PhD thesis, University of Cambridge Computer Laboratory. Available as Technical Report no. 376.
- Stevens, W. R. (1993), *Advanced Programming in the Unix Environment*, Addison-Wesley. ISBN: 0-201-56317-7.
- Vahalia, U. (1996), *Unix internals: The new frontier*, Prentice-Hall. ISBN: 0-13-101908-2.

A Spawn

This appendix contains the specification for the `spawn` function found in Posix implementations that can not support the usual `fork` and `exec` primitives, which we implement in our Posix library (see section 5.2). The description below is taken from the `spawn` man page included with the DJGPP Library with runs under MS-DOS (Delorie 1997).

A.1 Syntax

```
#include <process.h>

int spawnl(int mode, const char *path, const char *argv0, ...);
int spawnle(int mode, const char *path, const char *argv0, ... /*, const char **envp */);
int spawnlp(int mode, const char *path, const char *argv0, ...);
int spawnlpe(int mode, const char *path, const char *argv0, ... /*, const char **envp */);

int spawnv(int mode, const char *path, const char **argv);
int spawnve(int mode, const char *path, const char **argv, const char **envp);
int spawnvp(int mode, const char *path, const char **argv);
int spawnvpe(int mode, const char *path, const char **argv, const char **envp);
```

A.2 Description

These functions run other programs. The `path` points to the program to run. The extension is optional — if not given, and `path` is not found neither in the current directory nor along the `'PATH'`, the extensions `'.com'`, `'.exe'`, `'.bat'`, `'.btm'`, `'.sh'`, and `'.ksh'` are checked. `'.com'` programs are invoked via the usual DOS calls; DJGPP `'.exe'` programs are invoked in a way that allows long command lines to be passed; other `'.exe'` programs are invoked via DOS; `'.bat'` and `'.btm'` programs are invoked via the command processor given by the `'COMSPEC'` environment variable; `'.sh'`, `'.ksh'` programs and programs with any other extensions that have `#!` as their first two characters are assumed to be Unix-style scripts and are invoked by calling a program whose pathname immediately follows the first two characters. (If the name of that program is a Unix-style pathname, without a drive letter and without an extension, like `'/bin/sh'`, the `spawn` functions will additionally look them up on the `'PATH'`; this allows to run Unix scripts without editing, if you have a shell installed somewhere along your `'PATH'`.)

Note that built-in commands of the shells can *not* be invoked via these functions; use `system` instead.

The programs are invoked with the arguments given. The zeroth argument is normally not used, since MS-DOS cannot pass it separately. There are two ways of passing arguments. The `l` functions (like `spawnl`) take a list of arguments, with a zero at the end of the list. This is useful when you know how many argument there will be ahead of time. The `v` functions (like `spawnv`) take a pointer to a list of arguments. This is useful when you need to compute the number of arguments at runtime.

In either case, you may also specify `e` to indicate that you will be giving an explicit environment, else the current environment is used. You may also specify `p` to indicate that you would like `spawn*` to search the `PATH` (in either the environment you pass or the current environment) for the executable, else it will only check the explicit path given.

Note that these function understand about other DJGPP programs, and will call them directly, so that you can pass command lines longer than 126 characters to them without any special code. DJGPP programs called by these functions will *not* glob the arguments passed to them; other programs also won't glob the arguments if they suppress expansion when given quoted filenames.

See the `exec(3)` man page.

A.3 Return Value

If successful and `mode` is `P_WAIT`, these functions return the exit code of the child process in the lower 8 bits of the return value. Note that if the program is run by a command processor (e.g., if it's a batch file), the exit code of that command processor will be returned. `'COMMAND.COM'` is notorious for returning 0 even if it couldn't run the command.

If successful and `mode` is `P_OVERLAY`, these functions will not return.

If there is an error (e.g., the program specified as `argv[0]` cannot be run, or the command line is too long), these functions return -1 and set `errno` to indicate the error. If the child program was interrupted by Ctrl-C or a Critical Device error, `errno` is set to `EINTR` (even if the child's exit code is 0), and bits 8-17 of the return value are set to `SIGINT` or `SIGABRT`, accordingly. Note that you must set the signal handler for `SIGINT` to `SIG_IGN`, or arrange for the handler to return, or else your program will be aborted before it will get chance to set the value of the return code.

A.4 Example

```
char *environ[] = {
    "PATH=c:\dos;c:\djgpp;c:\usr\local\bin",
    "DJGPP=c:/djgpp",
    0
};

char *args[] = {
    "gcc",
    "-v",
    "hello.c",
    0
};

spawnvpe(P_WAIT, "gcc", args, environ);
```