Table of Contents, Show Frames, No Frames

Chapter 4
Processes

This chapter describes what a process is and how the Linux kernel creates, manages and deletes the processes in the system.

Processes carry out tasks within the operating system. A program is a set of machine code instructions and data stored in an executable image on disk and is, as such, a passive entity; a process can be thought of as a computer program in action.

It is a dynamic entity, constantly changing as the machine code instructions are executed by the processor. As well as the program's instructions and data, the process also includes the program counter and all of the CPU's registers as well as the process stacks containing temporary data such as routine parameters, return addresses and saved variables. The current executing program, or process, includes all of the current activity in the microprocessor. Linux is a multiprocessing operating system. Processes are separate tasks each with their own rights and responsibilities. If one process crashes it will not cause another process in the system to crash. Each individual process runs in its own virtual address space and is not capable of interacting with another process except through secure, kernel managed mechanisms.

During the lifetime of a process it will use many system resources. It will use the CPUs in the system to run its instructions and the system's physical memory to hold it and its data. It will open and use files within the filesystems and may directly or indirectly use the physical devices in the system. Linux must keep track of the process itself and of the system resources that it has so that it can manage it and the other processes in the system fairly. It would not be fair to the other processes in the system if one process monopolized most of the system's physical memory or its CPUs.

The most precious resource in the system is the CPU, usually there is only one. Linux is a multiprocessing operating system, its objective is to have a process running on each CPU in the system at all times, to maximize CPU utilization. If there are more processes than CPUs (and there usually are), the rest of the processes must wait before a CPU becomes free until they can be run. Multiprocessing is a simple idea; a process is executed until it must wait, usually for some

system resource; when it has this resource, it may run again. In a uniprocessing system, for example DOS, the CPU would simply sit idle and the waiting time would be wasted. In a multiprocessing system many processes are kept in memory at the same time. Whenever a process has to wait the operating system takes the CPU away from that process and gives it to another, more deserving process. It is the scheduler which chooses which is the most appropriate process to run next and Linux uses a number of scheduling strategies to ensure fairness.

Linux supports a number of different executable file formats, ELF is one, Java is another and these must be managed transparently as must the processes use of the system's shared libraries.

4.1 Linux Processes

So that Linux can manage the processes in the system, each process is represented by a `task_struct` data structure (task and process are terms that Linux uses interchangeably). The `task` vector is an array of pointers to every `task_struct` data structure in the system.

This means that the maximum number of processes in the system is limited by the size of the `task` vector; by default it has 512 entries. As processes are created, a new `task_struct` is allocated from system memory and added into the `task` vector. To make it easy to find, the current, running, process is pointed to by the `current` pointer.

As well as the normal type of process, Linux supports real time processes. These processes have to react very quickly to external events (hence the term ``real time'') and they are treated differently from normal user processes by the scheduler. Although the `task_struct` data structure is quite large and complex, but its fields can be divided into a number of functional areas:

State
: As a process executes it changes *state* according to its circumstances. Linux processes have the following states: [1]

Running
: The process is either running (it is the current process in the system) or it is ready to run (it is waiting to be assigned to one of the system's CPUs).

Waiting
: The process is waiting for an event or for a resource. Linux differentiates between two types of waiting process; *interruptible* and *uninterruptible*. Interruptible waiting processes can be interrupted by signals whereas uninterruptible waiting processes are waiting directly on hardware conditions and cannot be interrupted under any circumstances.

Stopped
: The process has been stopped, usually by receiving a signal. A process that is being debugged can be in a

stopped state.

Zombie
This is a halted process which, for some reason, still has a `task_struct` data structure in the `task` vector. It is what it sounds like, a dead process.

Scheduling Information
The scheduler needs this information in order to fairly decide which process in the system most deserves to run,

Identifiers
Every process in the system has a process identifier. The process identifier is not an index into the `task` vector, it is simply a number. Each process also has User and group identifiers, these are used to control this processes access to the files and devices in the system,

Inter-Process Communication
Linux supports the classic Unix ™ IPC mechanisms of signals, pipes and semaphores and also the System V IPC mechanisms of shared memory, semaphores and message queues. The IPC mechanisms supported by Linux are described in Chapter IPC-chapter.

Links
In a Linux system no process is independent of any other process. Every process in the system, except the initial process has a parent process. New processes are not created, they are copied, or rather *cloned* from previous processes. Every `task_struct` representing a process keeps pointers to its parent process and to its siblings (those processes with the same parent process) as well as to its own child processes. You can see the family relationship between the running processes in a Linux system using the pstree command:

```
init(1)-+-crond(98)
        |-emacs(387)
        |-gpm(146)
        |-inetd(110)
        |-kerneld(18)
        |-kflushd(2)
        |-klogd(87)
        |-kswapd(3)
        |-login(160)---bash(192)---emacs(225)
        |-lpd(121)
        |-mingetty(161)
        |-mingetty(162)
        |-mingetty(163)
        |-mingetty(164)
```

```
|-login(403)---bash(404)---pstree(594)
|-sendmail(134)
|-syslogd(78)
`-update(166)
```

Additionally all of the processes in the system are held in a doubly linked list whose root is the `init` processes `task_struct` data structure. This list allows the Linux kernel to look at every process in the system. It needs to do this to provide support for commands such as ps or kill.

Times and Timers

The kernel keeps track of a processes creation time as well as the CPU time that it consumes during its lifetime. Each clock tick, the kernel updates the amount of time in `jiffies` that the current process has spent in system and in user mode. Linux also supports process specific *interval* timers, processes can use system calls to set up timers to send signals to themselves when the timers expire. These timers can be single-shot or periodic timers.

File system

Processes can open and close files as they wish and the processes `task_struct` contains pointers to descriptors for each open file as well as pointers to two VFS inodes. Each VFS inode uniquely describes a file or directory within a file system and also provides a uniform interface to the underlying file systems. How file systems are supported under Linux is described in Chapter filesystem-chapter. The first is to the root of the process (its home directory) and the second is to its current or *pwd* directory. *pwd* is derived from the Unix ™ command pwd, *print working directory*. These two VFS inodes have their `count` fields incremented to show that one or more processes are referencing them. This is why you cannot delete the directory that a process has as its *pwd* directory set to, or for that matter one of its sub-directories.

Virtual memory

Most processes have some virtual memory (kernel threads and daemons do not) and the Linux kernel must track how that virtual memory is mapped onto the system's physical memory.

Processor Specific Context

A process could be thought of as the sum total of the system's current state. Whenever a process is running it is using the processor's registers, stacks and so on. This is the processes context and, when a process is suspended, all of that CPU specific context must be saved in the `task_struct` for the process. When a process is restarted by the scheduler its context is restored from here.

4.2 Identifiers

Linux, like all Unix ™ uses user and group identifiers to check for access rights to files and images in the system. All of the files in a Linux system have ownerships and permissions, these permissions describe what access the system's users have to that file or directory. Basic permissions are *read*, *write* and *execute* and are assigned to three classes of user; the owner of the file, processes belonging to a particular group and all of the processes in the system. Each class of user can have different permissions, for example a file could have permissions which allow its owner to read and write it, the file's group to read it and for all other processes in the system to have no access at all.

REVIEW NOTE: *Expand and give the bit assignments (777).*

Groups are Linux's way of assigning privileges to files and directories for a group of users rather than to a single user or to all processes in the system. You might, for example, create a group for all of the users in a software project and arrange it so that only they could read and write the source code for the project. A process can belong to several groups (a maximum of 32 is the default) and these are held in the `groups` vector in the `task_struct` for each process. So long as a file has access rights for one of the groups that a process belongs to then that process will have appropriate group access rights to that file.

There are four pairs of process and group identifiers held in a processes `task_struct`:

uid, gid
: The user identifier and group identifier of the user that the process is running on behalf of,

effective uid and gid
: There are some programs which change the uid and gid from that of the executing process into their own (held as attributes in the VFS inode describing the executable image). These programs are known as *setuid* programs and they are useful because it is a way of restricting accesses to services, particularly those that run on behalf of someone else, for example a network daemon. The effective uid and gid are those from the setuid program and the uid and gid remain as they were. The kernel checks the effective uid and gid whenever it checks for privilege rights.

file system uid and gid
: These are normally the same as the effective uid and gid and are used when checking file system access rights. They are needed for NFS mounted filesystems where the user mode NFS server needs to access files as if it were a particular process. In this case only the file system uid and gid are changed (not the effective uid and gid). This avoids a situation where malicious users could send a kill signal to the NFS server. Kill signals are delivered to processes with a particular effective uid and gid.

saved uid and gid
: These are mandated by the POSIX standard and are used by programs which change the processes uid and gid via system calls. They are used to save the real uid and gid during the time that the original uid and gid have been changed.

4.3 Scheduling

All processes run partially in user mode and partially in system mode. How these modes are supported by the underlying hardware differs but generally there is a secure mechanism for getting from user mode into system mode and back again. User mode has far less privileges than system mode. Each time a process makes a system call it swaps from user mode to system mode and continues executing. At this point the kernel is executing on behalf of the process. In Linux, processes do not preempt the current, running process, they cannot stop it from running so that they can run. Each process decides to relinquish the CPU that it is running on when it has to wait for some system event. For example, a process may have to wait for a character to be read from a file. This waiting happens within the system call, in system mode; the process used a library function to open and read the file and it, in turn made system calls to read bytes from the open file. In this case the waiting process will be suspended and another, more deserving process will be chosen to run.

Processes are always making system calls and so may often need to wait. Even so, if a process executes until it waits then it still might use a disproportionate amount of CPU time and so Linux uses pre-emptive scheduling. In this scheme, each process is allowed to run for a small amount of time, 200ms, and, when this time has expired another process is selected to run and the original process is made to wait for a little while until it can run again. This small amount of time is known as a *time-slice*.

It is the *scheduler* that must select the most deserving process to run out of all of the runnable processes in the system.

A runnable process is one which is waiting only for a CPU to run on. Linux uses a reasonably simple priority based scheduling algorithm to choose between the current processes in the system. When it has chosen a new process to run it saves the state of the current process, the processor specific registers and other context being saved in the processes `task_struct` data structure. It then restores the state of the new process (again this is processor specific) to run and gives control of the system to that process. For the scheduler to fairly allocate CPU time between the runnable processes in the system it keeps information in the `task_struct` for each process:

policy
: This is the scheduling policy that will be applied to this process. There are two types of Linux process, normal and real time. Real time processes have a higher priority than all of the other processes. If there is a real time process ready to run, it will always run first. Real time processes may have two types of `policy`, *round robin* and *first in first out*. In *round robin* scheduling, each runnable real time process is run in turn and in *first in, first out* scheduling each runnable process is run in the order that it is in on the run queue and that order is never changed.

priority
: This is the priority that the scheduler will give to this process. It is also the amount of time (in `jiffies`) that this process will run for when it is allowed to run. You can alter the priority of a process by means of system calls and the renice

command.

rt_priority
Linux supports real time processes and these are scheduled to have a higher priority than all of the other non-real time processes in system. This field allows the scheduler to give each real time process a relative priority. The priority of a real time processes can be altered using system calls.

counter
This is the amount of time (in `jiffies`) that this process is allowed to run for. It is set to `priority` when the process is first run and is decremented each clock tick.

The scheduler is run from several places within the kernel. It is run after putting the current process onto a wait queue and it may also be run at the end of a system call, just before a process is returned to process mode from system mode. One reason that it might need to run is because the system timer has just set the current processes `counter` to zero. Each time the scheduler is run it does the following:

kernel work
The scheduler runs the bottom half handlers and processes the scheduler task queue. These lightweight kernel threads are described in detail in chapter kernel-chapter.

Current process
The current process must be processed before another process can be selected to run.

If the scheduling policy of the current processes is *round robin* then it is put onto the back of the run queue.

If the task is `INTERRUPTIBLE` and it has received a signal since the last time it was scheduled then its state becomes `RUNNING`.

If the current process has timed out, then its state becomes `RUNNING`.

If the current process is `RUNNING` then it will remain in that state.

Processes that were neither `RUNNING` nor `INTERRUPTIBLE` are removed from the run queue. This means that they will not be considered for running when the scheduler looks for the most deserving process to run.

Process selection
The scheduler looks through the processes on the run queue looking for the most deserving process to run. If there are any real time processes (those with a real time scheduling policy) then those will get a higher weighting than ordinary processes. The weight for a normal process is its `counter` but for a real time process it is `counter` plus 1000. This means that if there are any runnable real time processes in the system then these will always be run before any

normal runnable processes. The current process, which has consumed some of its time-slice (its `counter` has been decremented) is at a disadvantage if there are other processes with equal priority in the system; that is as it should be. If several processes have the same priority, the one nearest the front of the run queue is chosen. The current process will get put onto the back of the run queue. In a balanced system with many processes of the same priority, each one will run in turn. This is known as *Round Robin* scheduling. However, as processes wait for resources, their run order tends to get moved around.

Swap processes

If the most deserving process to run is not the current process, then the current process must be suspended and the new one made to run. When a process is running it is using the registers and physical memory of the CPU and of the system. Each time it calls a routine it passes its arguments in registers and may stack saved values such as the address to return to in the calling routine. So, when the scheduler is running it is running in the context of the current process. It will be in a privileged mode, kernel mode, but it is still the current process that is running. When that process comes to be suspended, all of its machine state, including the program counter (PC) and all of the processor's registers, must be saved in the processes `task_struct` data structure. Then, all of the machine state for the new process must be loaded. This is a system dependent operation, no CPUs do this in quite the same way but there is usually some hardware assistance for this act.

This swapping of process context takes place at the end of the scheduler. The saved context for the previous process is, therefore, a snapshot of the hardware context of the system as it was for this process at the end of the scheduler. Equally, when the context of the new process is loaded, it too will be a snapshot of the way things were at the end of the scheduler, including this processes program counter and register contents.

If the previous process or the new current process uses virtual memory then the system's page table entries may need to be updated. Again, this action is architecture specific. Processors like the Alpha AXP, which use Translation Look-aside Tables or cached Page Table Entries, must flush those cached table entries that belonged to the previous process.

4.3.1 Scheduling in Multiprocessor Systems

Systems with multiple CPUs are reasonably rare in the Linux world but a lot of work has already gone into making Linux an SMP (Symmetric Multi-Processing) operating system. That is, one that is capable of evenly balancing work between the CPUs in the system. Nowhere is this balancing of work more apparent than in the scheduler.

In a multiprocessor system, hopefully, all of the processors are busily running processes. Each will run the scheduler separately as its current process exhausts its time-slice or has to wait for a system resource. The first thing to notice about

an SMP system is that there is not just one idle process in the system. In a single processor system the idle process is the first task in the `task` vector, in an SMP system there is one idle process per CPU, and you could have more than one idle CPU. Additionally there is one current process per CPU, so SMP systems must keep track of the current and idle processes for each processor.

In an SMP system each process's `task_struct` contains the number of the processor that it is currently running on (`processor`) and its processor number of the last processor that it ran on (`last_processor`). There is no reason why a process should not run on a different CPU each time it is selected to run but Linux can restrict a process to one or more processors in the system using the `processor_mask`. If bit N is set, then this process can run on processor N. When the scheduler is choosing a new process to run it will not consider one that does not have the appropriate bit set for the current processor's number in its `processor_mask`. The scheduler also gives a slight advantage to a process that last ran on the current processor because there is often a performance overhead when moving a process to a different processor.

4.4 Files

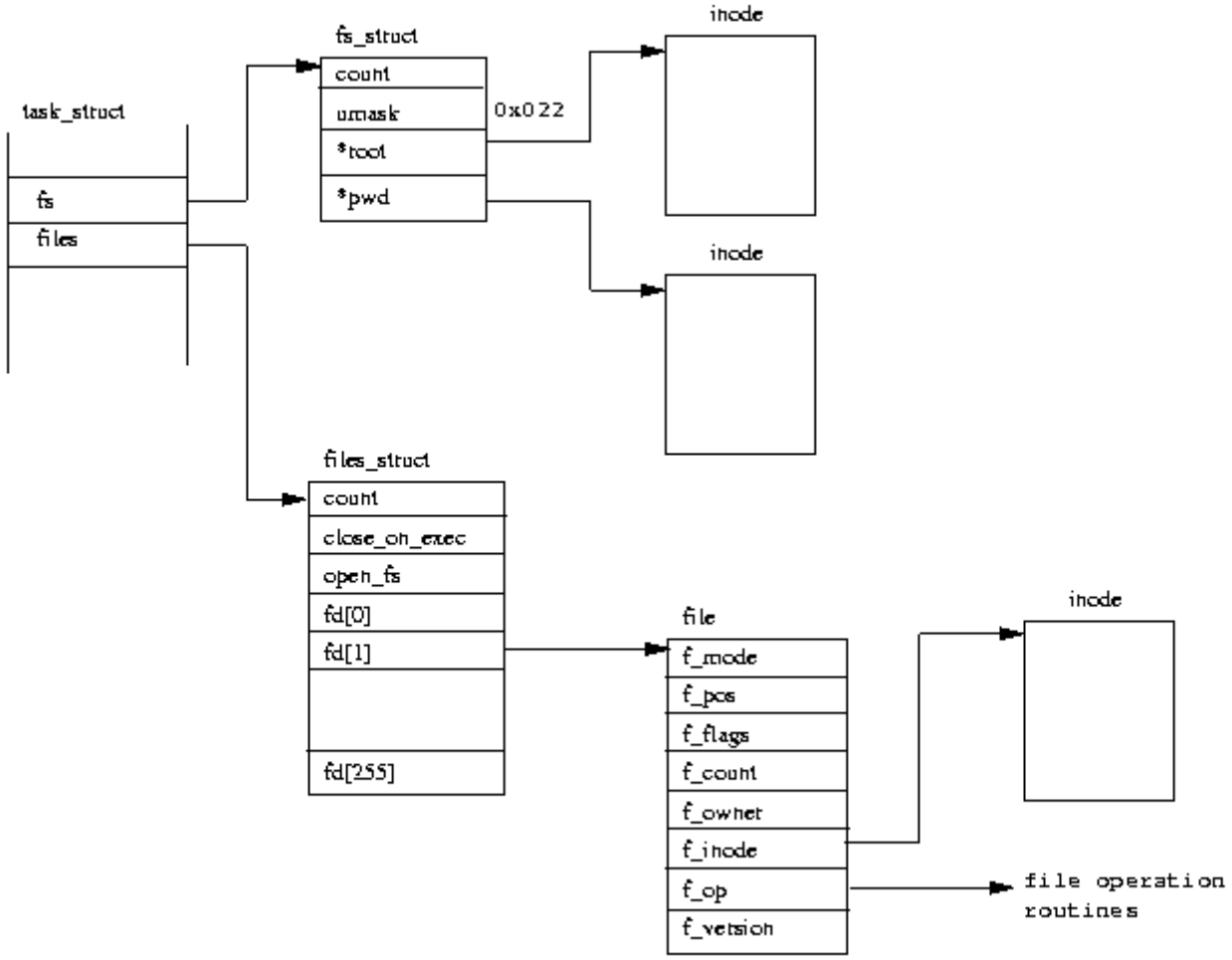

Figure 4.1: A Process's Files

Figure 4.1 shows that there are two data structures that describe file system specific information for each process in the system. The first, the `fs_struct`

contains pointers to this process's VFS inodes and its `umask`. The `umask` is the default mode that new files will be created in, and it can be changed via system calls.

The second data structure, the `files_struct`, contains information about all of the files that this process is currently using. Programs read from *standard input* and write to *standard output*. Any error messages should go to *standard error*. These may be files, terminal input/output or a real device but so far as the program is concerned they are all treated as files. Every file has its own descriptor and the `files_struct` contains pointers to up to 256 `file` data structures, each one describing a file being used by this process. The `f_mode` field describes what mode the file has been created in; read only, read and write or write only. `f_pos` holds the position in the file where the next read or write operation will occur. `f_inode` points at the VFS inode describing the file and `f_ops` is a pointer to a vector of routine addresses; one for each function that you might wish to perform on a file. There is, for example, a write data function. This abstraction of the interface is very powerful and allows Linux to support a wide variety of file types. In Linux, pipes are implemented using this mechanism as we shall see later.

Every time a file is opened, one of the free `file` pointers in the `files_struct` is used to point to the new `file` structure. Linux processes expect three file descriptors to be open when they start. These are known as *standard input*, *standard output* and *standard error* and they are usually inherited from the creating parent process. All accesses to files are via standard system calls which pass or return file descriptors. These descriptors are indices into the process's `fd` vector, so *standard input*, *standard output* and *standard error* have file descriptors 0, 1 and 2. Each access to the file uses the `file` data structure's file operation routines to together with the VFS inode to achieve its needs.

4.5 Virtual Memory

A process's virtual memory contains executable code and data from many sources. First there is the program image that is loaded; for example a command like ls. This command, like all executable images, is composed of both executable code and data. The image file contains all of the information neccessary to load the executable code and associated program data into the virtual memory of the process. Secondly, processses can allocate (virtual) memory to use during their processing, say to hold the contents of files that it is reading. This newly allocated, virtual, memory needs to be linked into the process's existing virtual memory so that it can be used. Thirdly, Linux processes use libraries of commonly useful code, for example file handling routines. It does not make sense that each process has its own copy of the library, Linux uses shared libraries that can be used by several running processes at the same time. The code and the data from these shared libraries must be linked into this process's virtual address space and also into the virtual address space of the other processes sharing the library.

In any given time period a process will not have used all of the code and data contained within its virtual memory. It could contain code that is only used during certain situations, such as during initialization or to process a particular event. It may only have used some of the routines from its shared libraries. It would be wasteful to load all of this code and data into physical memory where it would lie unused. Multiply this wastage by the number of processes in the system and the system would run very inefficiently. Instead, Linux uses a technique called *demand paging* where the virtual memory of a process

is brought into physical memory only when a process attempts to use it. So, instead of loading the code and data into physical memory straight away, the Linux kernel alters the process's page table, marking the virtual areas as existing but not in memory. When the process attempts to acccess the code or data the system hardware will generate a page fault and hand control to the Linux kernel to fix things up. Therefore, for every area of virtual memory in the process's address space Linux needs to know where that virtual memory comes from and how to get it into memory so that it can fix up these page faults.

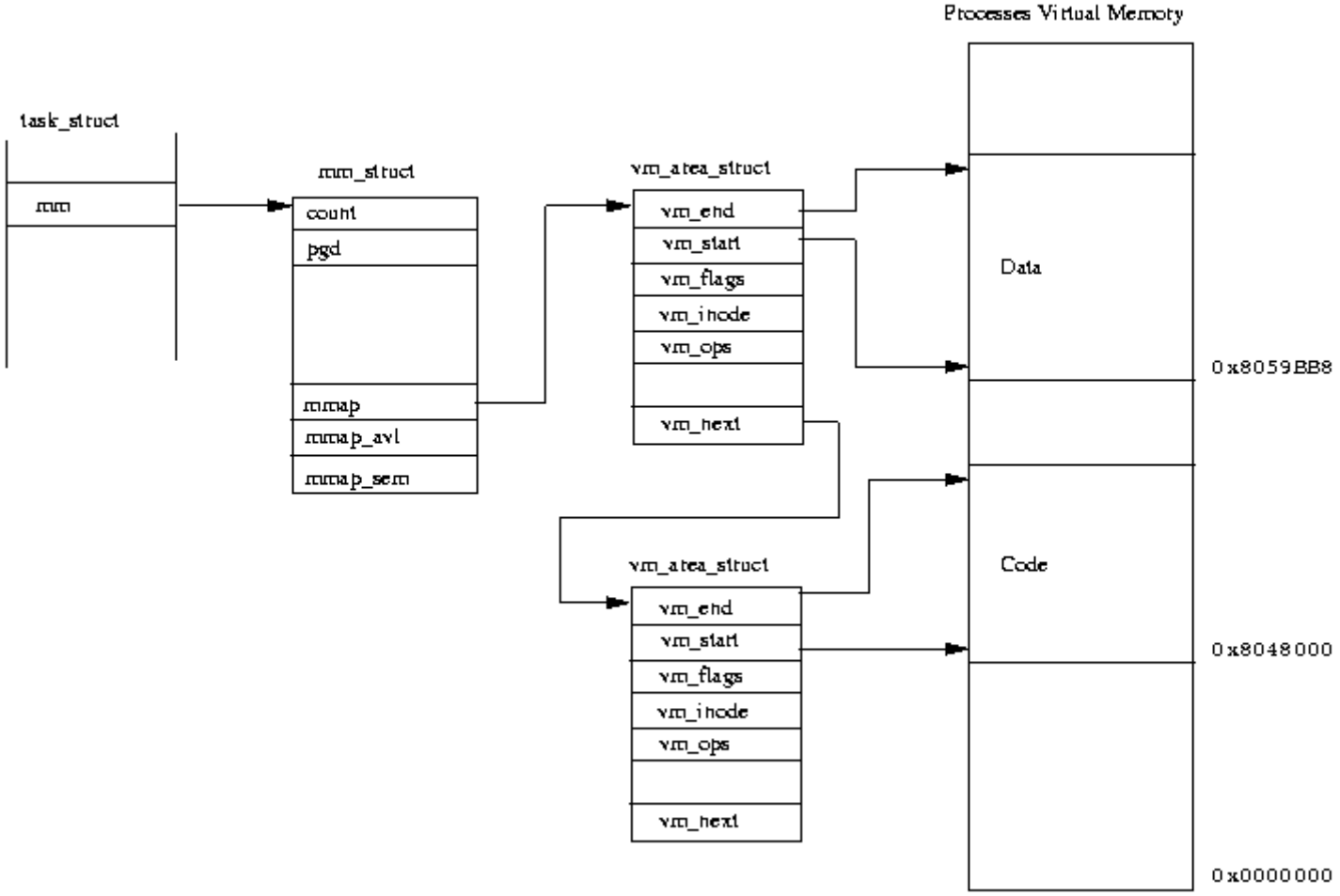

Figure 4.2: A Process's Virtual Memory

The Linux kernel needs to manage all of these areas of virtual memory and the contents of each process's virtual memory is described by a `mm_struct` data structure pointed at from its `task_struct`. The process's `mm_struct`

data structure also contains information about the loaded executable image and a pointer to the process's page tables. It contains pointers to a list of `vm_area_struct` data structures, each

representing an area of virtual memory within this process.

This linked list is in ascending virtual memory order, figure 4.2 shows the layout in virtual memory of a simple process together with the kernel data structures managing it. As those areas of virtual memory are from several sources, Linux abstracts the interface by having the `vm_area_struct` point to a set of virtual memory handling routines (via `vm_ops`). This way all of the process's virtual memory can be handled in a consistent way no matter how the underlying services managing that memory differ. For example there is a routine that will be called when the process attempts to access the memory and it does not exist, this is how page faults are handled.

The process's set of `vm_area_struct` data structures is accessed repeatedly by the Linux kernel as it creates new areas of virtual memory for the process and as it fixes up references to virtual memory not in the system's physical memory. This makes the time that it takes to find the correct `vm_area_struct` critical to the performance of the system. To speed up this access, Linux also arranges the `vm_area_struct` data structures into an AVL (Adelson-Velskii and Landis) tree. This tree is arranged so that each `vm_area_struct` (or node) has a left and a right pointer to its neighbouring `vm_area_struct` structure. The left pointer points to node with a lower starting virtual address and the right pointer points to a node with a higher starting virtual address. To find the correct node, Linux goes to the root of the tree and follows each node's left and right pointers until it finds the right `vm_area_struct`. Of course, nothing is for free and inserting a new `vm_area_struct` into this tree takes additional processing time.

When a process allocates virtual memory, Linux does not actually reserve physical memory for the process. Instead, it describes the virtual memory by creating a new `vm_area_struct` data structure. This is linked into the process's list of virtual memory. When the process attempts to write to a virtual address within that new virtual memory region then the system will page fault. The processor will attempt to decode the virtual address, but as there are no Page Table Entries for any of this memory, it will give up and raise a page fault exception, leaving the Linux kernel to fix things up. Linux looks to see if the virtual address referenced is in the current process's virtual address space. If it is, Linux creates the appropriate PTEs and allocates a physical page of memory for this process. The code or data may need to be brought into that physical page from the filesystem or from the swap disk. The process can then be restarted at the instruction that caused the page fault and, this time as the memory physically exists, it may continue.

4.6 Creating a Process

When the system starts up it is running in kernel mode and there is, in a sense, only one process, the initial process. Like all processes, the initial process has a machine state represented by stacks, registers and so on. These will be saved in the initial process's `task_struct` data structure when other processes in the system are created and run. At the end of system initialization, the initial process starts up a kernel thread (called `init`) and then sits in an idle loop doing nothing. Whenever there is nothing else to do the scheduler will run this, idle, process. The idle process's `task_struct` is the only one that is not dynamically allocated, it is statically defined at kernel build time and is, rather confusingly, called `init_task`.

The `init` kernel thread or process has a process identifier of 1 as it is the system's first real process. It does some initial setting up of the system (such as opening the system console and mounting the root file system) and then executes the system initialization program. This is one of `/etc/init`, `/bin/init` or `/sbin/init` depending on your system. The `init` program uses `/etc/inittab` as a script file to create new processes within the system. These new processes may themselves go on to create new processes. For example the `getty` process may create a `login` process when a user attempts to login. All of the processes in the system are descended from the `init` kernel thread.

New processes are created by cloning old processes, or rather by cloning the current process. A new task is created by a system call (*fork* or *clone*)

and the cloning happens within the kernel in kernel mode. At the end of the system call there is a new process waiting to run once the scheduler chooses it. A new `task_struct` data structure is allocated from the system's physical memory with one or more physical pages for the cloned process's stacks (user and kernel). A new process identifier may be created, one that is unique within the set of process identifiers in the system. However, it is perfectly reasonable for the cloned process to keep its parents process identifier. The new `task_struct` is entered into the `task` vector and the contents of the old (`current`) process's `task_struct` are copied into the cloned `task_struct`.

When cloning processes Linux allows the two processes to share resources rather than have two separate copies. This applies to the process's files, signal handlers and virtual memory. When the resources are to be shared their respective `count` fields are incremented so that Linux will not deallocate these resources until both processes have finished using them. So, for example, if the cloned process is to share virtual memory, its `task_struct` will contain a pointer to the `mm_struct` of the original process and that `mm_struct` has its `count` field incremented to show the number of current processes sharing it.

Cloning a process's virtual memory is rather tricky. A new set of `vm_area_struct` data structures must be generated together with their owning `mm_struct` data structure and the cloned process's page tables. None of the process's virtual memory is copied at this point. That would be a rather difficult and lengthy task for some of that virtual memory would be in physical memory, some in the executable image that the process is currently executing and possibly some would be in the swap file. Instead Linux uses a technique called ``copy on write'' which means that virtual memory will only be copied when one of the two processes tries to write to it. Any virtual memory that is not written to, even if it can be, will be shared between the two processes without any harm occuring. The read only memory, for example the executable code, will always be shared.

For ``copy on write'' to work, the writeable areas have their page table entries marked as read only and the `vm_area_struct` data structures describing them are marked as ``copy on write''. When one of the processes attempts to write to this virtual memory a page fault will occur. It is at this point that Linux will make a copy of the memory and fix up the two processes' page tables and virtual memory data structures.

4.7 Times and Timers

The kernel keeps track of a process's creation time as well as the CPU time that it consumes during its lifetime. Each clock tick, the kernel updates the amount of time in `jiffies` that the current process has spent in system and in user mode.

In addition to these accounting timers, Linux supports process specific *interval* timers.

A process can use these timers to send itself various signals each time that they expire. Three sorts of interval timers are supported:

Real
: the timer ticks in real time, and when the timer has expired, the process is sent a `SIGALRM` signal.

Virtual
: This timer only ticks when the process is running and when it expires it sends a `SIGVTALRM` signal.

Profile
: This timer ticks both when the process is running and when the system is executing on behalf of the process itself. `SIGPROF` is signalled when it expires.

One or all of the interval timers may be running and Linux keeps all of the neccessary information in the process's `task_struct` data structure. System calls can be made to set up these interval timers and to start them, stop them and read their current values. The virtual and profile timers are handled the same way.

Every clock tick the current process's interval timers are decremented and, if they have expired, the appropriate signal is sent.

Real time interval timers are a little different and for these Linux uses the timer mechanism described in Chapter _kernel-chapter. Each process has its own `timer_list` data structure and, when the real interval timer is running, this is queued on the system timer list. When the timer expires the timer bottom half handler removes it from the queue and calls the interval timer handler.

This generates the `SIGALRM` signal and restarts the interval timer, adding it back into the system timer queue.

4.8 Executing Programs

In Linux, as in Unix ™, programs and commands are normally executed by a command interpreter. A command interpreter is a user process like any other process and is called a `shell` [2].

There are many shells in Linux, some of the most popular are `sh`, `bash` and `tcsh`. With the exception of a few built in commands, such as cd and pwd, a command is an executable binary file. For each command entered, the shell searches the directories in the process's *search path*, held in the `PATH` environment variable, for an executable image with a matching name. If the file is found it is loaded and executed. The shell clones itself using the *fork* mechanism described above and then the new child process replaces the binary image that it was executing, the shell, with the contents of the executable image file just found. Normally the shell waits for the command to complete, or rather for the child process to exit. You can cause the shell to run again by pushing the child process to the background by typing `control-Z`, which causes a `SIGSTOP` signal to be sent to the child process, stopping it. You then use the shell command bg to push it into a background, the shell sends it a `SIGCONT` signal to restart it, where it will stay until either it ends or it needs to do terminal input or output.

An executable file can have many formats or even be a script file. Script files have to be recognized and the appropriate interpreter run to handle them; for example `/bin/sh` interprets shell scripts. Executable object files contain executable code and data together with enough information to allow the operating system to load them into memory and execute them. The most commonly used object file format used by Linux is ELF but, in theory, Linux is flexible enough to handle almost any object file format.

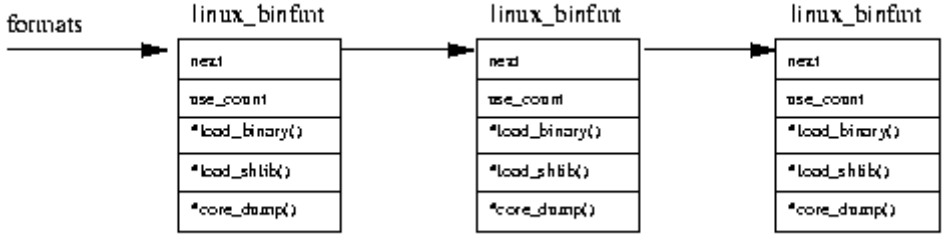

Figure 4.3: Registered Binary Formats

As with file systems, the binary formats supported by Linux are either built into the kernel at kernel build time or available to be loaded as modules. The kernel keeps a list of supported binary formats (see figure 4.3) and when an attempt is made to execute a file, each binary format is tried in turn until one works.

Commonly supported Linux binary formats are a.out and ELF. Executable files do not have to be read completely into memory, a technique known as demand loading is used. As each part of the executable image is used by a process it is brought into memory. Unused parts of the image may be discarded from memory.

4.8.1 ELF

The ELF (Executable and Linkable Format) object file format, designed by the Unix System Laboratories, is now firmly established as the most commonly used format in Linux. Whilst there is a slight performance overhead when compared with other object file formats such as ECOFF and a.out, ELF is felt to be more flexible. ELF executable files contain executable code, sometimes refered to as *text*, and *data*. Tables within the executable image describe how the program should be placed into the process's virtual memory. Statically linked images are built by the linker (ld), or link editor, into one single image containing all of the code and data needed to run this image. The image also specifies the layout in memory of this image and the address in the image of the first code to execute.

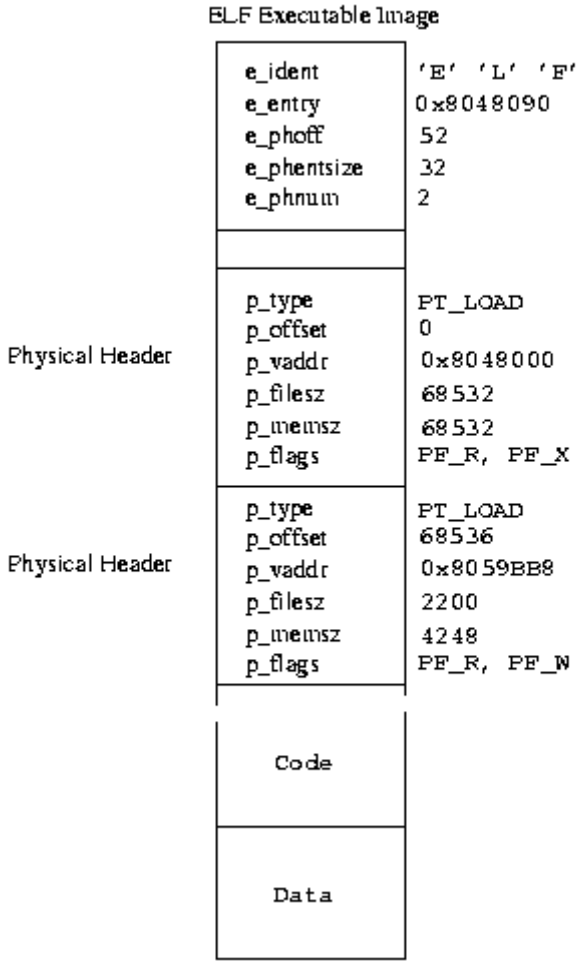

Figure 4.4: ELF Executable File Format

Figure 4.4 shows the layout of a statically linked ELF executable image.

It is a simple C program that prints ``hello world'' and then exits. The header describes it as an ELF image with two physical headers (`e_phnum` is 2) starting 52 bytes (`e_phoff`) from the start of the image file. The first physical header describes the executable code in the image. It goes at virtual address *0x8048000* and there is 65532 bytes of it. This is because it is a statically linked image which contains all of the library code for the `printf()` call to output ``hello world''. The entry point for

the image, the first instruction for the program, is not at the start of the image but at virtual address *0x8048090* (`e_entry`). The code starts immediately after the second physical header. This physical header describes the data for the program and is to be loaded into virtual memory at address *0x8059BB8*. This data is both readable and writeable. You will notice that the size of the data in the file is 2200 bytes (`p_filesz`) whereas its size in memory is 4248 bytes. This because the first 2200 bytes contain pre-initialized data and the next 2048 bytes contain data that will be initialized by the executing code.

When Linux loads an ELF executable image into the process's virtual address space, it does not actually load the image.

It sets up the virtual memory data structures, the process's `vm_area_struct` tree and its page tables. When the program is executed page faults will cause the program's code and data to be fetched into physical memory. Unused portions of the program will never be loaded into memory. Once the ELF binary format loader is satisfied that the image is a valid ELF executable image it flushes the process's current executable image from its virtual memory. As this process is a cloned image (*all* processes are) this, old, image is the program that the parent process was executing, for example the command interpreter shell such as `bash`. This flushing of the old executable image discards the old virtual memory data structures and resets the process's page tables. It also clears away any signal handlers that were set up and closes any files that are open. At the end of the flush the process is ready for the new executable image. No matter what format the executable image is, the same information gets set up in the process's `mm_struct`. There are pointers to the start and end of the image's code and data. These values are found as the ELF executable images physical headers are read and the sections of the program that they describe are mapped into the process's virtual address space. That is also when the `vm_area_struct` data structures are set up and the process's page tables are modified. The `mm_struct` data structure also contains pointers to the parameters to be passed to the program and to this process's environment variables.

ELF Shared Libraries

A dynamically linked image, on the other hand, does not contain all of the code and data required to run. Some of it is held in shared libraries that are linked into the image at run time. The ELF shared library's tables are also used by the *dynamic linker* when the shared library is linked into the image at run time. Linux uses several dynamic linkers, `ld.so.1`, `libc.so.1` and `ld-linux.so.1`, all to be found in `/lib`. The libraries contain commonly used code such as language subroutines. Without dynamic linking, all programs would need their own copy of the these libraries and would need far more disk space and virtual memory. In dynamic linking, information is included in the ELF image's tables for every library routine referenced. The information indicates to the dynamic linker how to locate the library routine and link it into the program's address space.

REVIEW NOTE: *Do I need more detail here, worked example?*

4.8.2 Script Files

Script files are executables that need an interpreter to run them. There are a wide variety of interpreters available for Linux; for example wish, perl and command shells such as tcsh. Linux uses the standard Unux ™ convention of having the first line of a script file contain the name of the interpreter. So, a typical script file would start:

```
#!/usr/bin/wish
```

The script binary loader tries to find the intepreter for the script.

It does this by attempting to open the executable file that is named in the first line of the script. If it can open it, it has a pointer to its VFS inode and it can go ahead and have it interpret the script file. The name of the script file becomes argument zero (the first argument) and all of the other arguments move up one place (the original first argument becomes the new second argument and so on). Loading the interpreter is done in the same way as Linux loads all of its executable files. Linux tries each binary format in turn until one works. This means that you could in theory stack several interpreters and binary formats making the Linux binary format handler a very flexible piece of software.

Footnotes:

[1] REVIEW NOTE: *I left out SWAPPING because it does not appear to be used.*

[2] Think of a nut the kernel is the edible bit in the middle and the shell goes around it, providing an interface.

Top of Chapter, Table of Contents, Show Frames, No Frames