What is an operating system?

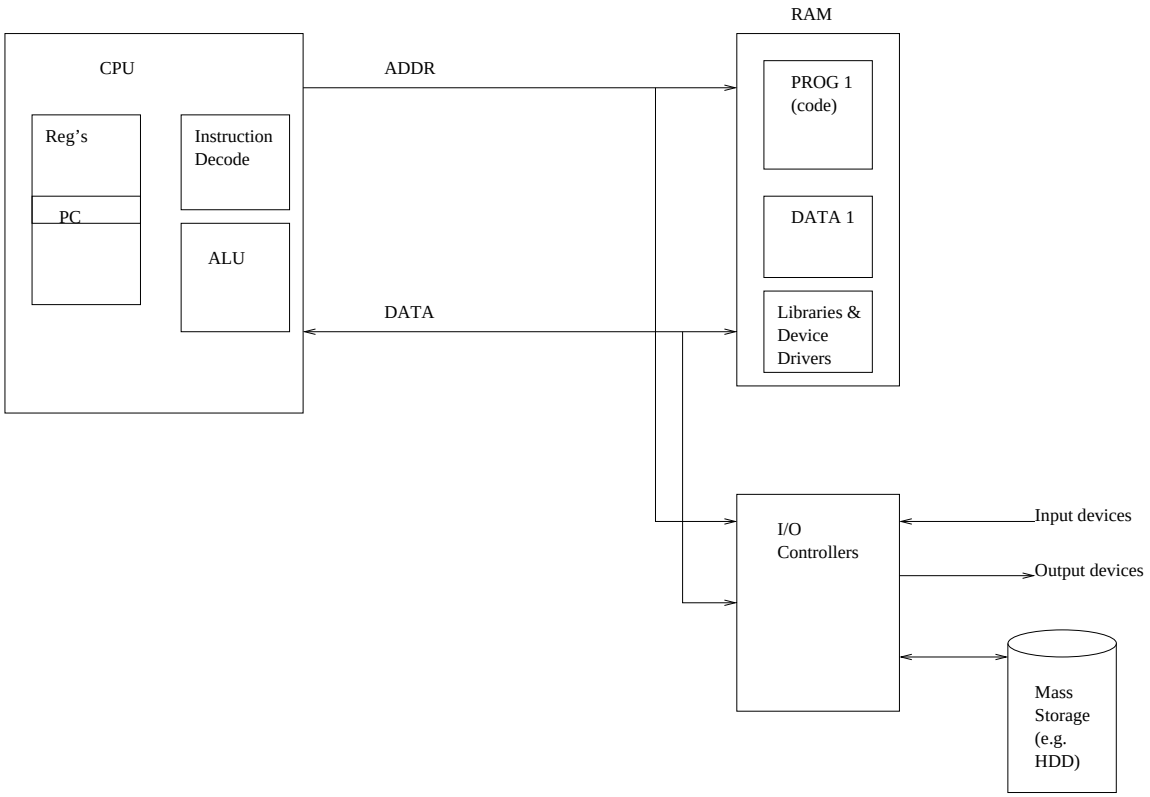
The term "operating system" means different things to different people. In order to understand the scope and role of an Operating System as it pertains to this course, let us briefly examine the history of computing. *The more things change, the more things remain the same*. The computing industry is the leader in wheel re-invention. Many of the classic problems in computing, including operating systems design, are "re-discovered" every few decades.

The earliest "computers" were really just calculators. They could perform a fixed task, such as summing a list of numbers, and some could be re-programmed in a sense by changing the wiring, but they were not capable of accepting and running an arbitrary software program.

With the introduction during World War II of Von Neummann architecture, the concept of the "computer program" was born. The operation of the computing machine could be described as a sequence of primitive instructions, or "opcodes", which could be represented as binary numbers. Therefore, it became possible to consider a computer program as another form of data which could be stored, processed and manipulated. Initial entry of a program into memory was typically in the form of punched paper tape or card stock media, or through manipulation of front-panel switches. Magnetic media such as drums, disks and tapes were later introduced to store and transport large amounts of data.

During these early days, any subroutines necessary for the operation of the machine itself, such as device drivers for I/O devices, had to be included with each program. Obviously this introduced considerable repetitive bulk. So, the very first "operating systems" were this collection of device driver subroutines and other commonly used routines that were kept semi-permanently in the machine's memory. (*It might be interesting to learn that most computer memory of this time period was magnetic "core" arrays, which had the property of retaining their contents with power off.*)

In this "batch" mode of operation, the OS remained resident while each user's "job" was loaded into memory, executed, and the output produced, typically on a line printer or card puncher. Only one job could be run at any given time, leading to long queues and poor development cycles (imagine waiting 45 minutes to find a syntax error!). A lot of a job's time was spent idly waiting for slow I/O operations to complete. The model below depicts, schematically, such a simple computer that does one program at a time.



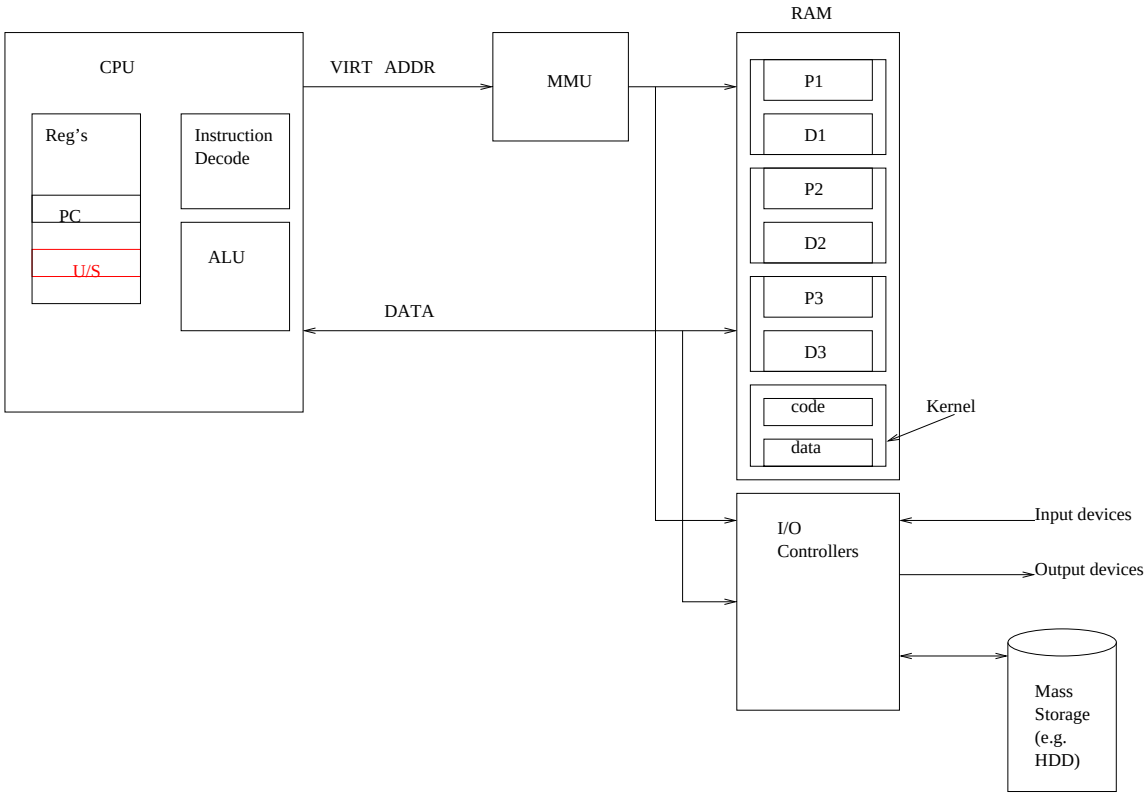
The advent of "multiprogramming", later called "multitasking", made use of this idle I/O wait time. Instead of being restricted to a single task, multiple jobs could be loaded into the machine. They would take turns executing; when a job blocked on I/O, another job would be selected to run. After the I/O operation finished, the first job would become eligible to run again and would run as soon as its turn came up. The operating system under a multitasking system has the responsibility of switching the use of the processor between jobs.

With multiple jobs, frequently belonging to multiple users, being executed at the same time on the same machine, the errant operation of a program was no longer a mere inconvenience. It became a serious problem in that if one program crashed, the entire machine crashed. In addition, a malfunctioning program could accidentally execute instructions or access memory locations or device registers that could physically damage the machine (many electromechanical peripherals such as printers and disk drives were controlled directly, such that it was possible to e.g. run the disk drive head off the end of the platter) or cause the destruction or corruption of data. Unacceptable!

The solution was to introduce hardware-based protection. Computers with such protection run in one of two modes. In the "Supervisor", aka "Kernel", aka "System", aka "Privileged", aka "Real" mode the system behaves identically to an unprotected system. In "User" mode, certain instructions and direct access to the machine's hardware are disallowed, including, obviously, the ability to switch the machine back into Supervisor

mode. Further protection is afforded by Memory Management hardware (MMU) which partitions the physical random-access memory (RAM) of the machine into sections. An attempt to read or write to an address outside of a program's assigned address space causes the operation to be denied and an error raised.

The Operating System could now be clearly viewed as having a "kernel", meaning the collection of operating system code which is run in Supervisor mode. When the machine is first bootstrapped, the kernel is loaded into memory and execution begins with the machine in Supervisor Mode. When the kernel selects a user program to run, it transfers control to the user program while at the same time de-activating Supervisor mode. The user program runs in User Mode, whereby it is prevented from causing harm to the system through errant operation.



As an aside, note that the rise of the "Personal Computer" in the late 1970s and early 1980s in many ways represented a regression. The Personal Computer was, in terms of architecture and sophistication, equivalent to the more primitive machines of the early 1960s. It was distinctly personal, single-user and single-tasking. The ability to do more than one thing at a time and the idea that a single bad program should not crash the entire machine took many years to re-enter the computing mainstream, although such concepts had been understood and accepted for over 20 years prior.

Today, operating systems are found on a wide variety of computing devices with varying amounts of memory, storage, processing and connectivity. Traditional categorization

based on market target (such as "desktop" or "server") is confusing at best because almost all operating systems have some crossover components. But here are some "tags" and their commonly understood meanings:

• Personal: designed to provide services to a human user who is locally connected to the computing device via human input/output devices such as keyboard, display or mouse/touchscreen.

• Server: designed to provide services to remote human users or non-human programs running on other devices, by means of a communications network.

• Embedded: designed primarily to monitor and/or control real-world physical systems. E.g. automotive engine controls, traffic lights, biomedical.

• Mobile: designed to provide connectivity and content services to mobile human users.

In this course, the majority of discussion will be of server and personal operating systems. We will concentrate primarily on the kernel, both its internal construction and the programming interfaces to it. Although there are software components, such as system libraries, which are often thought of as part of the operating system, we will attempt to keep the distinction clear between what happens within the kernel, and what occurs outside of it.

Services Provided by the Operating System

When people speak informally about Operating Systems, they often include software components which are not part of the kernel. These components may be libraries that are supplied by the system, programs which run in the background to provide services, or even entire applications. Here are some of the broad categories of services which are provided by the kernel itself, and which we will be examining during the course:

• Bootstrap/Initialization: Bringing the machine from an initial state after power-up or reboot to a functioning state.
• Device Drivers: Interfacing with devices at the hardware level and isolating the programmer or end user from those details.
• Storage/Filesystems: Managing bulk storage devices such as disks and organizing them to provide directories, files, etc.
• Memory Management: Controlling the pool of physical memory and allocating it as needed using the virtual memory illusion.
• Scheduling: Controlling the illusion of virtual processors and allocating processor resources fairly among competing processes.
• Security: Determining the identity and authorization profiles of users and programs, and enforcing the desired protections against unauthorized access.

• Communication: Allowing programs and users, either on the same machine or across a network, to communicate.

Illusions

The operating system kernel assumes control of the computer at boot time and remains in command as the system runs. The kernel periodically relinquishes control, in a deliberate and regulated manner, to user-mode processes. The kernel, in conjunction with the processor hardware, is able to maintain several critical *illusions* with respect to the standpoint of these user-level programs.

The first illusion is that of **virtual processors**, or **multi-tasking**. The use of the processor (for now, to simplify the material, we will assume a single-processor system, but later in the course we will fully explore modern multi-processor designs) is shared among the many user-level programs currently running on the computer. In effect, each program has the use of a virtual processor.

This sharing is done by **time-slicing**, such that program A runs on the processor for a number of instruction cycles, then another program B runs, then program C, etc. and eventually program A runs again. When execution resumes in program A, all registers and other critical state information are restored and execution continues just where it had left off, through a software technique known as *context switching.* Thus, the fact that other programs had run in the intervening time is transparent to the program's code.

The second illusion is **virtual memory**. When user-level code is executing, all addresses used in the program are virtual addresses. A component of the processor hardware called the **memory management unit** (MMU) sits between the virtual addresses which are generated by program execution (instruction fetches and data read/writes) and the physical addresses which are output on the address pins of the processor [ignoring the reality that the MMU is typically integrated on the same chip as the CPU.]

The kernel, by using the processor registers which control the MMU, is able to establish the mapping between virtual and physical addresses. This can be changed by the kernel as each program executes. Hence the kernel has the ability to control completely all of the memory which a user-level program can "see."

The kernel can also use the MMU and its control over virtual address space to allow the grand total of RAM in use by all of the programs on the computer exceed the actual amount of physical RAM. It does this by using disk or other storage to hold the data. When an area of a program's virtual address space does not currently have an actual place in physical memory to hold it, the MMU flags that area as missing. Then, when the program attempts to access the missing area, an interrupt is generated. This is known as a **page fault**. The kernel allocates an available chunk of physical RAM and maps it in, then resumes execution of the program.

Therefore, we see that the kernel has complete control over the world view of any

program on the computer. This combination of virtual processor and virtual address space forms a virtual equivalent of the the old-fashioned single-purpose computer for each program. Such a "virtual computer" (note: this is not a reference to virtual machine (VM) software such as VMWare or VirtualBox) is known as a **process**. The process is the cornerstone of multi-user, multi-tasking computing. In UNIX operating systems, each command launched from the shell such as `ls` is executed in a new process. We can think of a process as a brand-new virtual computer into which the program is loaded and executed. When the program has completed, the virtual computer is discarded.

Because of time-slicing, the actual run time of a program as observed by an external observer may vary from run to run, depending on how many other processes shared the use of the CPU. The operating system records the total amount of time for which a given process has use of the processor. This time is divided into two measurements: **user time** and **system time**. The former is the time the process spent executing its user-level code, and the latter is the time spent executing kernel-level code on the behalf of the process (e.g. responding to system calls as described below). The sum of these two measurements is the actual time spent doing useful work, and will always be less than or equal to the **real time** elapsed, which is also called **wall time**, i.e. the time spent by the user staring at the wall waiting for the program to complete.

We will explore processes, time-slicing and virtual memory at great length in subsequent units.

Kernel Re-entry: The System Call

A user-mode program finds itself executing within the protective bubble of a process. It is not able to touch control registers, I/O devices or other sensitive resources directly. There must be some means of penetrating that bubble in a controlled manner and gaining access to required resources and services. We shall see that there are three basic mechanisms, all of which are supported by hard-wired logic built in to the CPU. These are: hardware interrupt, instruction fault, and system call. In this unit, we will consider the last mechanism, as an introduction to user-kernel interfacing.

A System Call is a controlled transition from user mode back into kernel mode. This requires execution of hardware-specific, specialized opcodes. Because the method of making a system call will vary from platform to platform, this detail is isolated from the programmer by the system libraries. When writing a program in the C language, a system call appears to be just another function call. It is the convention among UNIX-like systems that system calls are documented in section (2) of the man pages, while other standard library functions are in section (3).

The set of system calls available to the user program is well-defined and well-documented. The kernel must be extra-careful to verify all of the parameters passed to it during the system call and avoid accidental or deliberate corruption of the system through garbage data.

The following is an overview of a system call. In later units we will explore these items in greater detail:

• The user-mode program makes an ordinary function call to a function of a specified, standardized name, such as `open`. This function is provided by the system libraries (`libc`) and is a hybrid function: it is coded partially in C, and partially in assembly language.

• The system call takes the parameters which are passed as ordinary function call parameters, and places them in the proper place (usually registers) where the kernel expects them to be. It then invokes a special assembly language instruction which performs the system call.

• The CPU responds to this instruction by simultaneously elevating back to Supervisor mode and jumping to a pre-arranged location in the kernel.

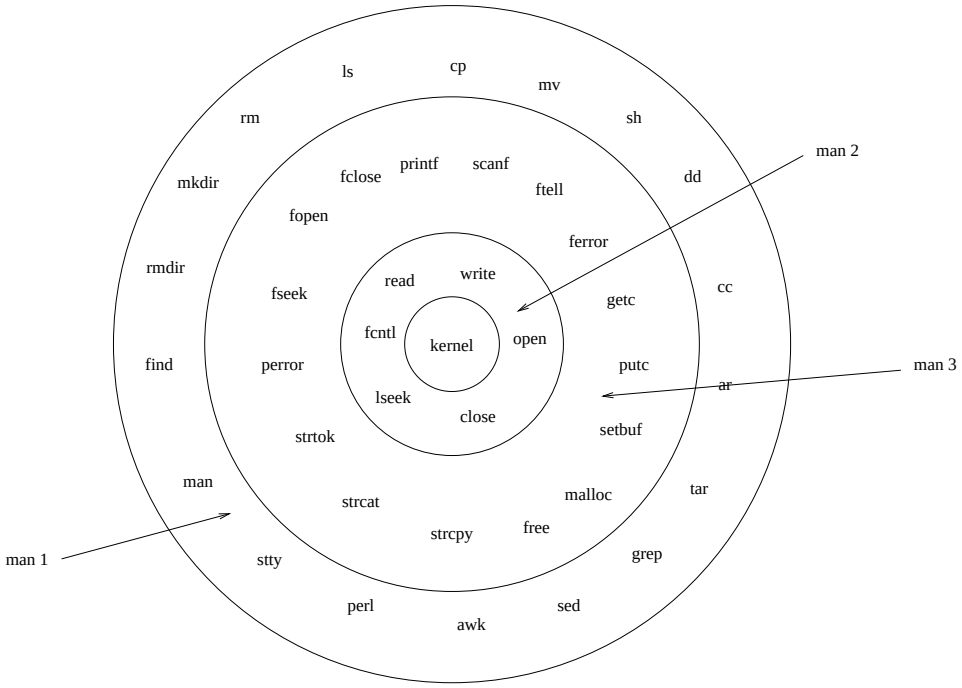
• The kernel's system call entry handler verifies all the parameters and if they are correct, executes the system call as a C function with a corresponding name. E.g. in the Linux kernel, the `open` system call in user mode corresponds to a function `sys_open` that is invoked in kernel mode.

• The system call may take some time to complete. During that time, other processes may be running. When the system call does complete, it has either succeeded or failed. The failure condition is reflected by a negative number being returned from the system call kernel function (e.g. `sys_open`). This number also encodes a specific integer error code. If the system call succeeds, the non-negative return value may have additional meaning, e.g. the number of bytes that have been read from a file.

• The kernel returns from the system call handler. Hardware restores to the User mode and execution resumes in the user-level library function, which takes the system call return value and makes it appear to be the return value from the ordinary C function. If the return value from the kernel is a negative number, that number is placed into a global variable `errno` and, generally, the user program sees a -1 return value. Otherwise, the value returned from the kernel is returned as-is to the user program.

UNIX Software component layers & man pages

The figure below depicts how the standard I/O and standard C library functions which are documented in man section 3, call system call functions which are in section 2 of the man pages, providing access via the system call interface to the kernel itself. The commands which are typically associated with the UNIX command-line environment, in section 1 of the man pages, are built on top of the standard library functions.



The list of defined error numbers is provided in an include file, which provides symbolic `#define` names for these constants. The return value used to flag an error, as well as which error codes may be returned, is part of the documentation for each system call. To call up, e.g., the documentation for the `open` system call, use the command `man 2 open`. The numeral refers historically to section 2 of the UNIX Reference Manual, which at one time was a printed document that came with your UNIX system. All system calls are contained in section 2. Under some UNIX systems, such as Solaris, the syntax would be `man -s2`.

It is convention that system commands are in section 1 of the manual. System calls are in section 2. Standard C library functions and functions in other libraries, such as the math library, are in section 3. These 3 sections are universal among all UNIX variants. Linux uses section 4 for information on specific hardware devices. Section 5 documents system file formats. Section 6 is for games. Section 7 documents network protocols, how the system bootstraps, character sets, and how certain kernel features works. Section 8 documents system maintenance and administration commands. You may find that sections 4 and higher are organized differently on different UNIX variants such as Solaris or BSD.

The UNIX I/O model

The set of system calls for performing I/O under UNIX is somewhat analogous to the stdio library routines. To open a file:

```
#include <fcntl.h>           /* Defines O_XXX constants */

int fd;                     /* Must be signed integer */
char *fname;

          fd=open(fname,O_RDONLY);     /* Similar to fp=fopen(fname,"r") */
          fd=open(fname,O_WRONLY|O_CREAT|O_TRUNC,0666);   /* fopen(fname,"w") */
          fd=open(fname,O_RDWR);                 /* fopen(fname,"r+") */
          fd=open(fname,O_RDWR|O_CREAT|O_TRUNC,0666);     /* fopen(fname,"w+") */
          fd=open(fname,O_WRONLY|O_CREAT|O_APPEND,0666);  /* fopen(fname,"a") */
          fd=open(fname,O_RDWR|O_CREAT|O_APPEND,0666);    /* fopen(fname,"a+") */
```

The value returned from open is known as a *file descriptor*. The file descriptor is essentially a "handle" or a "cookie" which can be used by the program in subsequent system calls to refer to the open file. It is a small integer with a range of 0 to N, where N is at least 19 on every variant of UNIX known to mankind, and is frequently much larger. (E.g. on Linux systems, the maximum number of open files is a configurable resource limit. You can observe these limits with the command `ulimit -a`. The default limit on file descriptors is 1024)

Once a program is done with the file, it can call `close(fd)`. This is analogous to `fclose(fp)`. All open files are also automatically closed when a program exits. It is often a good idea to explicitly close a file when you are done with it, and check the close for error, because there are some error conditions which can occur when writing to a file or file-like entity such as a network connection, which will not be detectable until the file is closed. Once closed, the file descriptor number may be given out again by a subsequent open system call. In fact, the lowest available (non-open) file descriptor number is generally given out.

We see that the open(2) system call can take either 2 or 3 parameters. The meaning of the third parameter, which sets the file access modes for newly created files, will be explained in a subsequent unit.

The second parameter to open(2) is an integer which is a bitwise combination of flags defined as O_XXXX in the header file `<fcntl.h>`. For example, on a particular Linux system, these constants are defined thus (some definitions have been ommited for clarity).

```
#define O_RDONLY             00
#define O_WRONLY             01
#define O_RDWR               02
#define O_CREAT            0100
#define O_EXCL             0200
#define O_TRUNC           01000
#define O_APPEND          02000
```

We see that the bits are defined as octal constants, and can be combined using the logical OR operator. This idiom is very common for UNIX system calls.

The `<fcntl.h>` header file is one which is packaged with the operating system. It must be coordinated so that the same bit definitions which were used to build the kernel

are also the ones exposed to user-level programmers.

Standard I/O streams

In the UNIX model, three file descriptors are special. Descriptors 0, 1 and 2 are known as standard input, standard output and standard error, respectively. The analogous stdio names are `stdin`,`stdout`,`stderr`.

By default, when the program is started from a command-line shell environment, standard input refers to the keyboard and standard output and standard error the screen. Use of the shell redirection syntax (such as > or |) changes which file is associated with particular standard streams, and is one of the most basic building blocks of the power of the UNIX operating system. More will be said about the mechanisms of I/O redirection in a later unit.

The three standard streams (0,1,2) may be assumed to always be open when the program begins, and should generally not be closed by the program. Doing so would defeat the intentions of the invoking user. However, we will see examples later of instances where it is necessary and proper to close and/or re-open these streams.

Error reporting

When a UNIX system call fails, it returns an out-of-the-ordinary value to indicate its failure, and sets a **global** (ick ick ick) integer variable called `errno` to indicate the nature of the error. As there is only one such variable, subsequent failed system calls will overwrite the error information of earlier calls.

Note: `errno` is only set when a system call fails. Since it is a global integer, the initial value is 0, which is "No error." If a particular system call fails, it will set `errno`. Subsequent system calls that do not fail will **not** overwrite that value. This global variable model becomes a problem with multi-threaded programming, but that is far beyond the scope of this first unit.

A program should always check for errors. It can attempt to take some action based on the specific error code:

```
#include <errno.h>          /* Defines error numbers */
#include <fcntl.h>          /* For O_RDONLY def */

int fd;
char *fname;

          if  ((fd=open(fname,O_RDONLY))<0)
          {
 /* The code below is meant as an example of making a decision based
          on errno.  DON'T BLINDLY COPY IT!!! */
```

```
                if (errno == ENOENT)
                        fprintf(stderr,"Sorry, %s does not appear to be
                                the name of a valid file\n",fname);
                else
                        fprintf(stderr,"Can't open file %s for reading:%s\n",
                                fname,strerror(errno));
                return -1;
        }
```

In the above example, we see the use of the symbolic constant `ENOENT` to refer to the specific error ("No such file or directory"). We also see the use of the standard library function `strerror` which returns the error description. This illustrates an important point in error handling and reporting: Report as much information as possible! Don't hide information that might be useful in determining the reason for a problem. Once that information is discarded, it is lost forever. At the very least, for failed system calls, provide:

The operation that was being attempted ("open")
The object being operated on (the file name)
Other relevant parameters ("for reading")
The error returned (strerror(errno))

The textual error message which is generated by `strerror`, `perror` and the like, may vary from system to system or may not even be in the English language if "localization" is turned on. The assignment of specific `errno` integers to specific errors is also subject to change. The symbolic error names, such as `ENOENT`, however, are part of the standardized UNIX programming interface. Programmers should always use these when making decisions based on specific error results. The `man` pages document which error codes may be expected for a given system call (although beware, sometimes the documentation is not exhaustive). The Linux kernel defines over 100 error codes. As an example, the open(2) system call can return more than a dozen different error codes.

Although the example above illustrates looking at `errno` to make error-specific decisions, **generally that is not necessary**. In most cases, system calls should be checked for error and the error should be reported with `strerror`, etc.

Reading and Writing

The system call provided to read data from a file is called, oddly enough, `read`. Here is an example:

```
char buf[4096];
int j,lim;
        lim=sizeof buf;
        j=read(fd,buf,lim);
```

The first parameter to read is an open file descriptor, The second is a buffer (an array of characters) into which the data will be read. Sometimes the buffer is a static array, as above, sometimes the buffer has been dynamically allocated with malloc, but it better be something that actually has some memory associated with it! Finally, the third argument is a limit to the number of bytes to read into the buffer. Obviously, the buffer should be at least as large as this limit. The operating system will never try to write more than that number of bytes into the buffer. Note that the result is NOT terminated by a NUL (\0) character.

We can classify the return value from read into 3 categories:

j<0: An error has occurred.
j==0: The end of file has been reached (see discussion below)
0<j<=lim: j bytes of data have been read into the buffer.

The converse of read is write:

```
char buf[4096];
int n,j;

        strcpy(buf,"test");
        n=strlen(buf);
        j=write(fd,buf,n);
```

`write` attempts to write the `n` bytes residing at the address `buf` to the file `fd`. The return value falls into 4 categories:

j<0: An error occurred. The data were not written.
j==0: No data were written. This is almost always an error and can be treated the same as a negative result. Under some circumstances, such as non-blocking I/O on devices or network connections, this condition is a transient one and the write can be re-tried. This is beyond the scope of this introduction.
j==n: All of the data were written.
0<j<n: The first j bytes of data were written successfully.

The last case is an interesting one, and one in which many programmers make mistakes. This so-called "short write" can happen for a variety of reasons, which we shall explore in future units. It doesn't indicate a "hard error," but it *does* mean that bytes j .. n-1 were **not** written. (remember, bytes in a file, like elements of an array in C, start counting at 0 not 1). If you overlook this condition and only check for write returning a negative value, you will potentially lose data. On the other hand, if you over-react and treat this short-write as a system call error, you'll find that there is no error, and you will terminate your efforts prematurely. Proper recovery is to re-try the write for the remaining n-j bytes starting at buf+j.

Just a bunch of bytes

The UNIX file model may be said to be "just a bunch of bytes". The UNIX kernel does not impose any kind of typing or structure on files. Every file is created and treated equally as an arbitrary and *opaque*(meaning that UNIX doesn't care what the contents are) sequence of bytes. There are no special characters; each of the 256 possible byte values is of equal stature. In particular, note that the NUL character (`'\0'`) has no special significance. Although it is used by C as a string terminator, it is neither needed nor noticed by read or write.

Also, there is no EOF (End Of File) character! The UNIX kernel knows where the end of a file is because it knows exactly how many bytes are in the file. By default, reads and writes are sequential. The kernel maintains a record for each file descriptor of where the last read or write left off, and the next one picks up there. A write beyond the existing end of file simply grows the file. A read past end of file returns 0 (after all, there are 0 bytes available to be read).

The current read/write position in an open file can be queried or reset through the `lseek` system call. Thus a file can be accessed either sequentially or randomly. The position is 0 when the file is first opened, although as a further wrinkle, when a file has been opened with the **O_APPEND** bit flag, all writes include an implicit and automatic seek to the current end of file at the moment of the write. This mode is often used for log files to ensure that data can never be overwritten.

When reading from the terminal, as is the case by default when reading from standard input, the event of hitting the end of file can be simulated when the user hits Control-D at the beginning of the line. This causes read to return 0 when it reaches this point. Subsequent reads will not continue to return 0, as would be the case when reading from an actual file, so it is possible to input multiple "files" from the keyboard. It is important to note that the actual Control-D character is NOT part of the input stream and will never be seen by the program. The fact that Control-D is the keyboard end-of-file is an arbitrary convention. Any character can be made the EOF character (read the man page `stty(1)`), often with amusing results.

The stdio library

The C programming language was designed as the first "portable" high-level language. This was a great challenge considering the vast collection of operating systems and machine architectures that existed, far more than have survived into common usage in the present day. Each operating system could be counted on to have its own way of doing what is a very universal function: input and output of files.

The Standard I/O library (stdio) is a wrapper designed to isolate the programmer from these details and allow hir to write code which can be readily compiled and executed on a multitude of systems. Underneath, the wrapper makes OS-specific system calls to accomplish its job. It also provides some "value-added" features such as error handling

and buffering.

Other libraries have been layered on top of the stdio library, most notably the `iostream` object library of the C++ run-time model.

It is assumed here that the reader is fully conversant in C programming with the stdio library and so no explanation of matters such as `printf` syntax will be given.

Under UNIX, the stdio library makes use of the UNIX file I/O system calls such as open, close, read and write.

Buffering and the stdio library

Buffering is a key concept in operating systems design. The principle is simple: given that you've done something, you're likely to do something very similar very soon. One of the most common uses of buffering is in the stdio library. Consider a program to change all linefeeds to <CR><LF> (in other words, converting from UNIX to Windows/DOS EOL conventions):

```
main()
{
char buf[1];
int n;
          while ((n=read(0,buf,1))>0)
          {
                    if (buf[0]=='\n')
                              write(1,"\r",1);
                    write(1,buf,1);
          }
}
```

The equivalent program using stdio runs faster: (**NOTE: Both versions of the program are seriously flawed in that they fail to check for or report errors !!!**)

```
#include <stdio.h>

main()
{
int c;
          while ((c=getc(stdin))!=EOF)
          {
                    if (c=='\n')
                              putc('\r',stdout);
                    putc(c,stdout);
          }
}
```

When you use the stdio functions such as putc and getc, or printf (which is internally built on a variant of putc), you are not actually making a system call. The `FILE *` is a pointer to a struct which is allocated by fopen. This struct contains a pointer to a buffer,

which is also allocated at user-level by fopen, and other related fields such as a count of the characters in the buffer. putc copies the supplied single character into the next space in the buffer. When the buffer is full, only then is the `write` system call made.

Let's say the buffer is 4,096 bytes (a common choice). Now instead of making 4,096 system calls, one character at a time, just one will be made. This saves the overhead of making a system call, which as we will see can involve a long code path with a lot of instructions. Of course, if the buffer is very large, there will be relatively little gain in economy, and the price will be excessive memory usage. The default buffer size is tuned to the optimal performance in most cases. For most programs, this layer of isolation is ideal. It allows the programmer to separate what he or she is trying to do from the low-level implementation details. In some cases, especially with programs that do system-level things (like formatting disks) this isolation gets in the way, and it is necessary to use the system calls directly.

In the case of input, a read system call is only made when the buffer is empty. Thereafter, functions such as getc, fgets, scanf, etc. simply copy the characters out of the buffer, until it gets empty again.

Sometimes we don't want to wait for an entire buffer before writing. The stdio library supports 3 output buffering modes:

- _IONBF: Unbuffered. Data are written immediately. This is the default mode for stderr--you really want to see your errors as soon as they happen.

- _IOFBF: File buffering. Buffer is flushed when it is full.

- _IOLBF: Line buffering. Buffer is flushed when an end-of-line character is written. This is intended for terminals, and is the default for stdin and stdout.

The `setvbuf` library call can be used to adjust the buffering mode of a FILE, and the size of the buffer. `fflush` can be used to force a buffer flush.

The File System

Any operating system must provide a framework for the storage of **persistent** data, i.e. those data which live beyond the execution of particular programs and which are expected to survive reboots. UNIX-like operating systems place a great deal of emphasis on the file system and present a fairly simple but robust interface.

The file system arose historically as a way of managing bulk storage devices. When data were stored on punched cards or punched tape, the grouping of individual records into files and of files into "folders" was purely physical, as these paper data were kept in specialized file cabinets. With the advent of magnetic media (disk, drum and tape), it was now incumbent upon the operating system to manage these raw bits and organize them so that users could store and retrieve their files.

Today, persistent storage is provided primarily by one of two technologies. The Hard Disk (or sometimes Disc) Drive (HDD) uses one or more spinning platters of magnetic material with tiny read/write heads flying a few thousandths of an inch above the platter surface. The other major technology is persistent solid-state memory, provided by "Flash" electrically erasable programmable read-only memory (EEPROM) chips in various configurations. Solid-state drives (SSDs) are flash devices optimized for use to replace HDDs as the primary storage for a desktop, laptop or server system, while other flash products such as the ubiquitous USB memory "stick" or SD cards used in cameras and phones, are designed and priced with less writing activity in mind.

In this unit, we won't go much further into the construction of HDDs or SSDs but will focus on how the operating system manages their contents. We can think of a disk as a randomly addressable array of bytes. However, since a byte is a very small amount of storage, all mass storage devices have a minimum addressable element size which is known as a **disk block** or **sector**. The latter term is preferred since disk block could mean something else, as we'll see later. Historically most devices used a 512 byte sector size. With hard drives breaking through the 2TB mark, this has required a shift to 2048 byte sectors. (If the sector number is considered as a 32-bit unsigned number, with 2^32 maximum sectors and a sector size of 2^9, the total disk size would be limited to 2^41 or 2 TB).

To give things a clear name, we'll call each such randomly-addressable array of sectors (whatever their size) a **volume**. A volume may be an entire hard disk, a removable storage device, a partition, or a "logical volume." For the purposes of this unit, all of these are equivalent.

How, then, do we go about organizing those bytes of the disk to form a permanent data structure, much as we organize the bytes of RAM to form in-memory data structures? Each operating system historically had both its own layout scheme of bytes on a disk, and its own interface details, such as how files and directories were named, and what other

information was kept.

There were also radical variations in how operating systems presented an interface for users or programs to explore and manipulate its system of files. We have already seen that UNIX takes the simple "just a bunch of bytes" approach. This was not so with many other operating systems, which often had built-in notions of file "types" (e.g. a program source file was a certain type of file for which different operations were possible from, say, an executable file).

What the File System provides

Every type of filesystem, from the simplest to the most robust, must provide certain basic services to the user. These services are delivered via the kernel system call interfaces.

• **Naming**: We must have a way of giving human-readable names to the files (and/or other objects) stored in the filesystem.

• **Data Storage**: Obviously the filesystem must provide a way to read and write the actual data contained in the files.

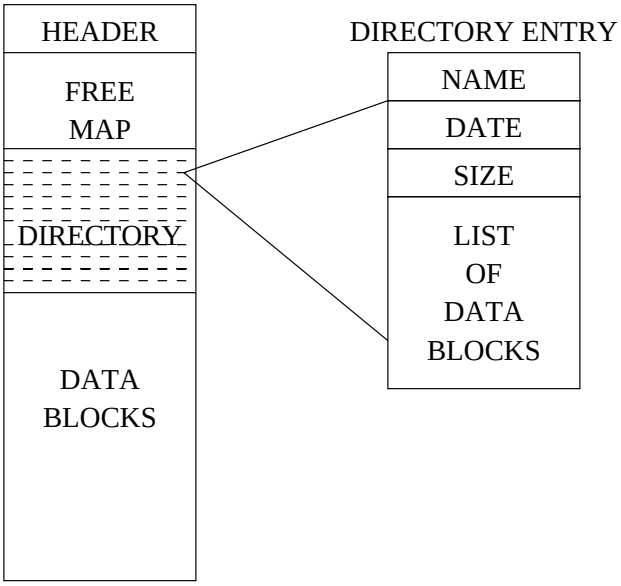
• **Meta-data**: The filesystem tracks other data which are not the actual contents of the files, but are data about the data, or "meta" data. These data may include the size of the file, time of last modification, the creator, etc.

• **Free space management**: The disk on which the files are stored is finite. We must have a way of determining how much empty space is left on the disk, and how much space each file is taking up.

Flat file systems

The simplest file system is a flat one, in which there are no subdirectories, and the number of files is limited to a fixed number. Historically, the last major general-purpose operating system to use a flat filesystem was MSDOS version 1.

However, today flat filesystems may still be found where storage needs are simple and adding the complexity of a directory hierarchy is prohibitively expensive, for example, many embedded devices, such as digital cameras or network switches/routers. As processor power and ROM size grows with embedded processors, flat filesystems are becoming more rare.



This volume is divided into 4 distinct parts. A header block serves to name to volume, give its size, and provide other summary information such as how much free space is available.

The contents of the files are stored within the data block section. The free map section maintains an overview of which data blocks are currently being used, and which are free to be allocated to files as they are written. Common implementations of the free map are a linked list of free blocks, or a bitmap with each bit representing the status of one block.

In the FAT filesystem (aka the MSDOS filesystem), the free map area (or "File Allocation Table") contains many linked lists. One is the list of free blocks, and additional lists chain together the data blocks comprising each file. This strategy does not perform well for random-access to file contents.

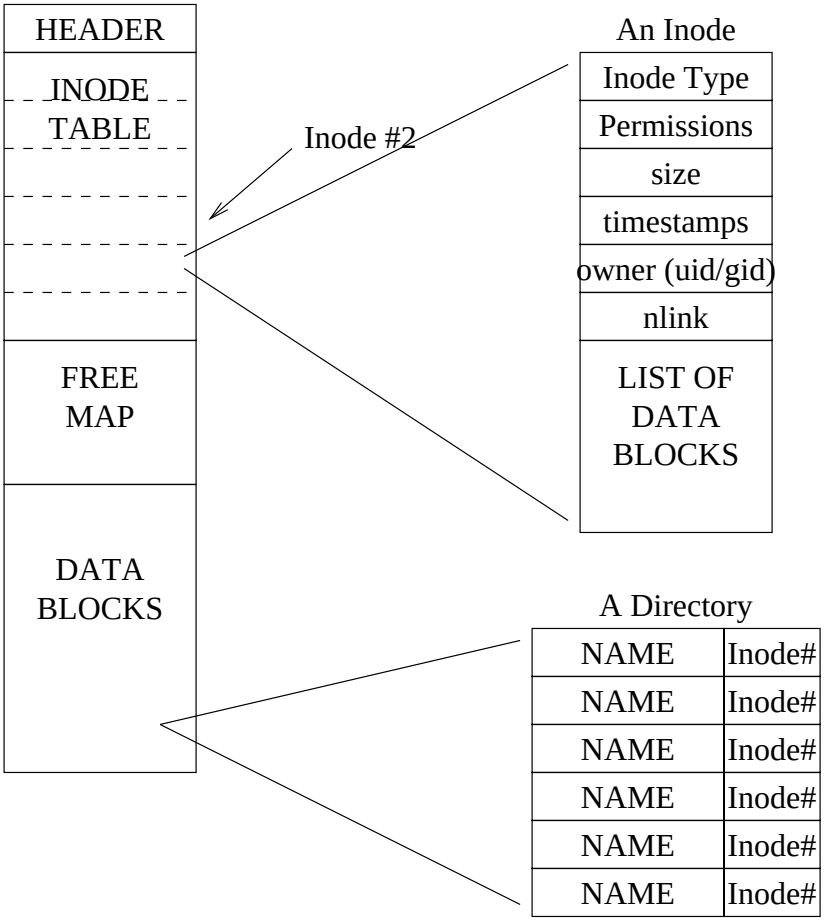
The directory lists the names of the files on the volume. Note that in this flat filesystem, the number of directory slots is fixed, and each directory entry contains everything there is to know about a file: its name, its **metadata** (such as the file size and time of last modification), and the mapping of the file contents to specific data blocks. The metadata are not the actual contents of the file, but are still important to users and programs, and are thus accessible through the system call interfaces (e.g. `stat(2)` in the UNIX kernel). Other data within the directory entry slot, e.g. the list of data blocks, is not meaningful at the user level, and is thus typically not accessible except by viewing the raw underlying on-disk data structure.

The MSDOS filesystem had a fixed file name size of 8 characters plus a 3-character extension, which determined how the file was treated. Common extensions included .COM and .EXE for executable programs, .BAT for MSDOS batch (shell script) language, .SYS for system files, .TXT for plain text files, etc. This 8+3 filename structure still haunts us today. E.g. TIFF files are .TIF, and JPEG files are .JPG

The simple flat filesystem can be trivially extended, e.g. the MSDOS/FAT filesystem, by adding a flag to a directory entry which causes that "file" to be treated as another directory. This allows a fully hierarchical view. We're not going to spend any more time on these primitive filesystems, and the curious reader is invited to consult online resources which describe FAT/VFAT/FAT32 filesystems more completely.

UNIX Filesystems

The UNIX file system model is derived from the actual data structures which were used on disk many years ago in the earliest UNIX operating systems. The approach taken back then was flexible enough that modern UNIX systems can use the same interface to "seamlessly integrate" many volumes of many different file system organizational types into one hierarchy. Let's begin by exploring, abstractly, how UNIX organizes a volume.



Once again, the volume comprises 4 distinct areas, the size of each of which is fixed at volume initialization time. The UNIX command to create a filesystem on a volume is called `mkfs`. This command accesses the disk directly, on a byte-by-byte (sector-by-sector) raw basis, and lays out the filesystem data structure.

The volume header contains miscellaneous information about the entire volume, such as a descriptive name, the number of active files within the volume, the time of last use, and other critical information. For historical reasons, this header data structure is often called the "superblock". The superblock describes the size and layout of the rest of the volume. If the superblock were to be lost or corrupted, then no data could be accessed as it would be impossible, e.g., to discern where the data block section was. For this reason, UNIX operating systems keep redundant copies of the superblock at other well-known locations in the volume.

The data block section can be thought of as a resource pool, divided into **filesystem allocation blocks** of a certain size. Historically, UNIX used 512 byte blocks, and this size still creeps into certain dark corners of the operating system. However, larger block sizes such as 1K, 2K, 4K or 8K are more common. Given the block size, we can think of the data block area as an array of blocks, indexed by a block number. (For reasons which may become clearer after reading some kernel source code, the first block number is not 0, but some larger number). Filesystem blocks are not necessarily the same size as sectors. They are usually larger, although some pathological cases could exist (e.g. creating an old System-V type filesystem with 512 byte blocks on a 2+TB drive with 2K sectors).

The filesystem data block is the smallest unit of storage allocation in the filesystem. If the block size is 1K, then a file which is 518 bytes long still consumes 1024 bytes of disk space. There is a tradeoff between space efficiency and time efficiency and the selection of block size can be tuned accordingly.

What was unique about the UNIX approach was the treatment of directories as just another type of file. Thus the directories are stored in the same data block resource pool as the file contents.

Another unique feature of the UNIX filesystem was the divorcing of metadata information from the directory entries. A directory in a UNIX filesystem is simply a list of filenames. Information about a single file or directory is kept within a data structure known as an **inode**.

The inode table is conceptually an array of these inodes (a typical on-disk inode size is 128 bytes and therefore several inodes are kept per sector), indexed by **inode number**. Again, for historical and kernel programming reasons, the first inode is usually numbered 2 (because 0 was used to indicate an empty directory slot, and 1 was reserved for the boot block).

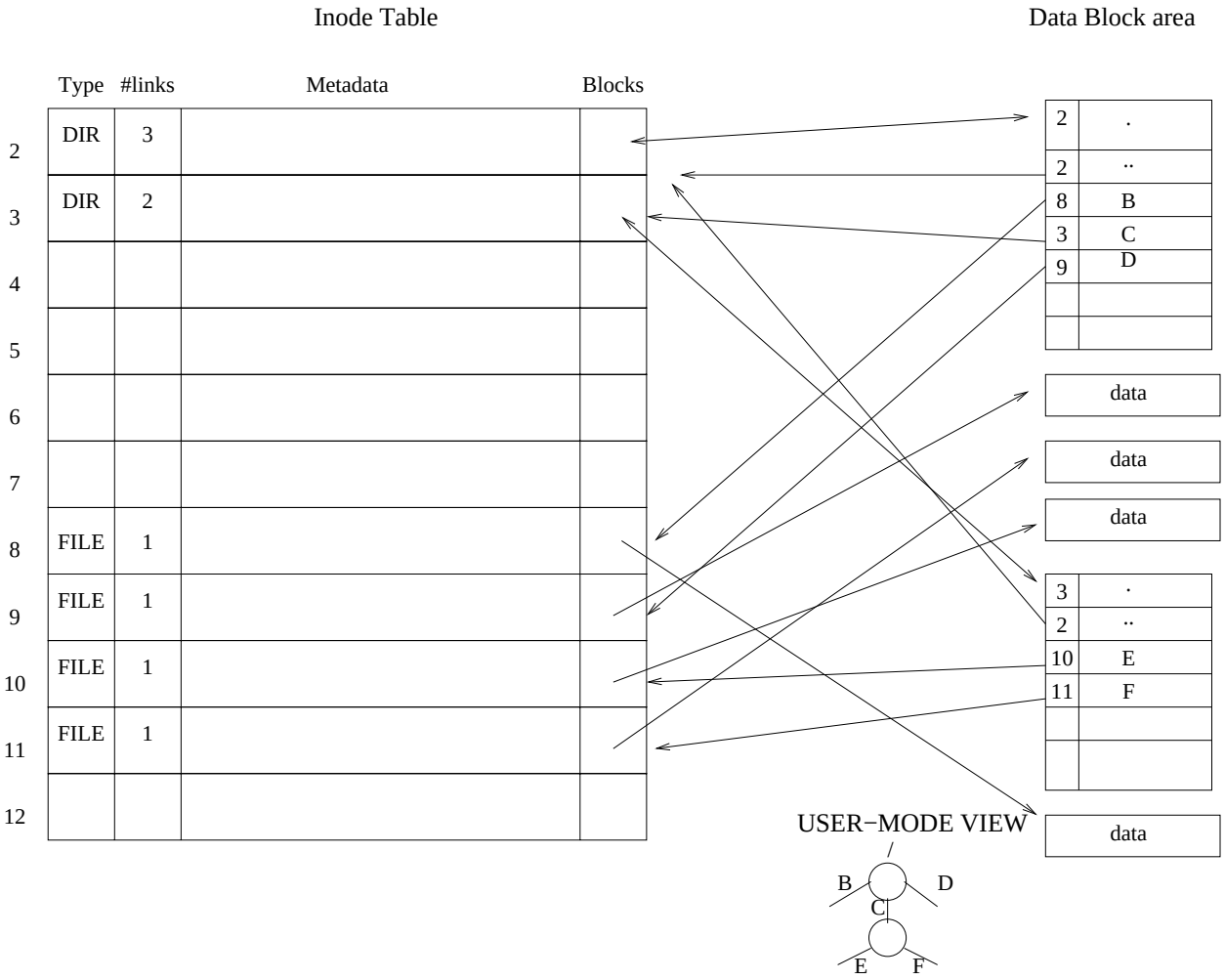
The inode contains all of the **metadata**, such as the size, owner, permissions and times of last access. It also contains (conceptually) the list of block numbers which comprise the contents of the file or directory. Another important bit of metadata is the **inode type**, e.g. is this a regular file or a directory (there are other types too, as we'll see)

The free map provides an overview of which data blocks are in use, and which are free to

be allocated to files. Most modern UNIX systems use a bitmap representation of the free map.

Directories and inodes

A directory is simply another instance of a file, specially identified as such via the Type field in its inode. Logically, a directory is an unordered list of names and inode numbers. In actual implementation, directory entries may be of fixed or of variable length, and in some cases more sophisticated data structures such as hashes or trees are introduced to make searching them faster. In the original UNIX filesystem, the name field of the directory entry was fixed at 14 characters, and the inode number was 2 bytes long, yielding a fixed 16-byte directory entry size, and also imposing an annoying 14-character limit on file names. Most modern UNIX filesystems allow at least 255 characters per path component name, and at least 1024 characters for a complete path name.



Note that the directory entry does not contain any other metadata information. In order to retrieve that, or to figure out where the associated file's contents are in the data section,

the inode must be consulted. Effectively, the inode number in the directory entry is a **pointer** to the inode. Note that the inode, in turn, does not have a "Name" field. A file is only given a name in the sense that there exists a path that refers to it. Given a particular inode, there is no way to find the path or paths associated with it except through exhaustive search of the filesystem.

Inode numbers can be considered "cookies". They are integers with definite bounds that can be compared for equality, but no other semantics can be inferred. In particular, the fact that one inode has a lower number than the other does not necessarily imply that it was created first. As a filesystem ages and files are created and removed, inode numbers will get recycled. Similarly, even though, e.g., inodes 69 and 71 exist, there is no basis to assume that inode 70 exists.

Pathnames and Wildcards

There must be a starting point, a root of the tree. By convention, the first inode of the filesystem (which has inode #2) is a directory and forms the "root directory" of the filesystem. UNIX pathnames that begin with "/" are evaluated starting from the root. These are known as **absolute**, or **fully-qualified** paths. Otherwise, they are **relative** paths and are evaluated starting at the **current working directory** (`cwd`, which is a state variable associated with each process).

A pathname under UNIX is a series of component names delimited by the **pathname separator** character, forward slash (/) . Each component is a string of **any** characters, **except for the / character or the NUL terminator (\0) character**. The length of each component name has a finite limit, typically 255 characters. Each component of a pathname, except for the last, must refer to a directory. While there is no limit on the number of components in a pathname, there may be limits as to the total length of the pathname string, such as 1024 characters (the `pathconf` library function can be used to determine this limit). This is half a screen of text so one would not want to have to type such a long pathname too often.

Doubtless the reader is familiar with UNIX wildcard syntax, such as `rm *.c`. Wildcard expansion is performed by the **shell**, which is the UNIX command-line interpreter. In a later unit, we will see how the shell functions. From the standpoint of the operating system kernel and system calls, there are no wildcards. The * or ? characters have no significance and are valid path component name characters. So are spaces, control characters, hyphens and a host of other characters which often cause confusion to novices.

Note also that UNIX does not have a notion of a file "extension" as does the DOS/WINDOWS family of operating systems. There is a naming convention which some programs follow. E.g. the C compiler expects that C source code files end in `.c`. This is strictly an application-level issue, and is not the concern of the kernel in any way.

The component names . and . . are reserved. They are always found in any directory, even an freshly-created empty one, and refer to the directory itself and to the directory's parent directory, respectively.

Because empty component names do not make any sense, any sequence of forward slashes in a pathname is equivalent to just one. E.g. "/C/////E" is the same as "/C/E". As a result of this, and the . and .. names, **there is (essentially) an unbounded number of possible pathnames which refer to the same node** (subject to pathname length limits). The existence of hard links makes this statement even more important.

Hard Links

To draw the analogy to the world of paper records, the UNIX file system is a file cabinet where the individual files are given arbitrary serial numbers (the inode number), and there is a separate card catalog index which allows us to determine the number associated with a particular naming. It is often useful to retrieve a given physical file under multiple names or subjects, e.g. the file for product "X1000" might be filed under Products/ Widgets/ X1000; also under Product Recall Notices/Safety Critical; and Patents/Infringement Claims.

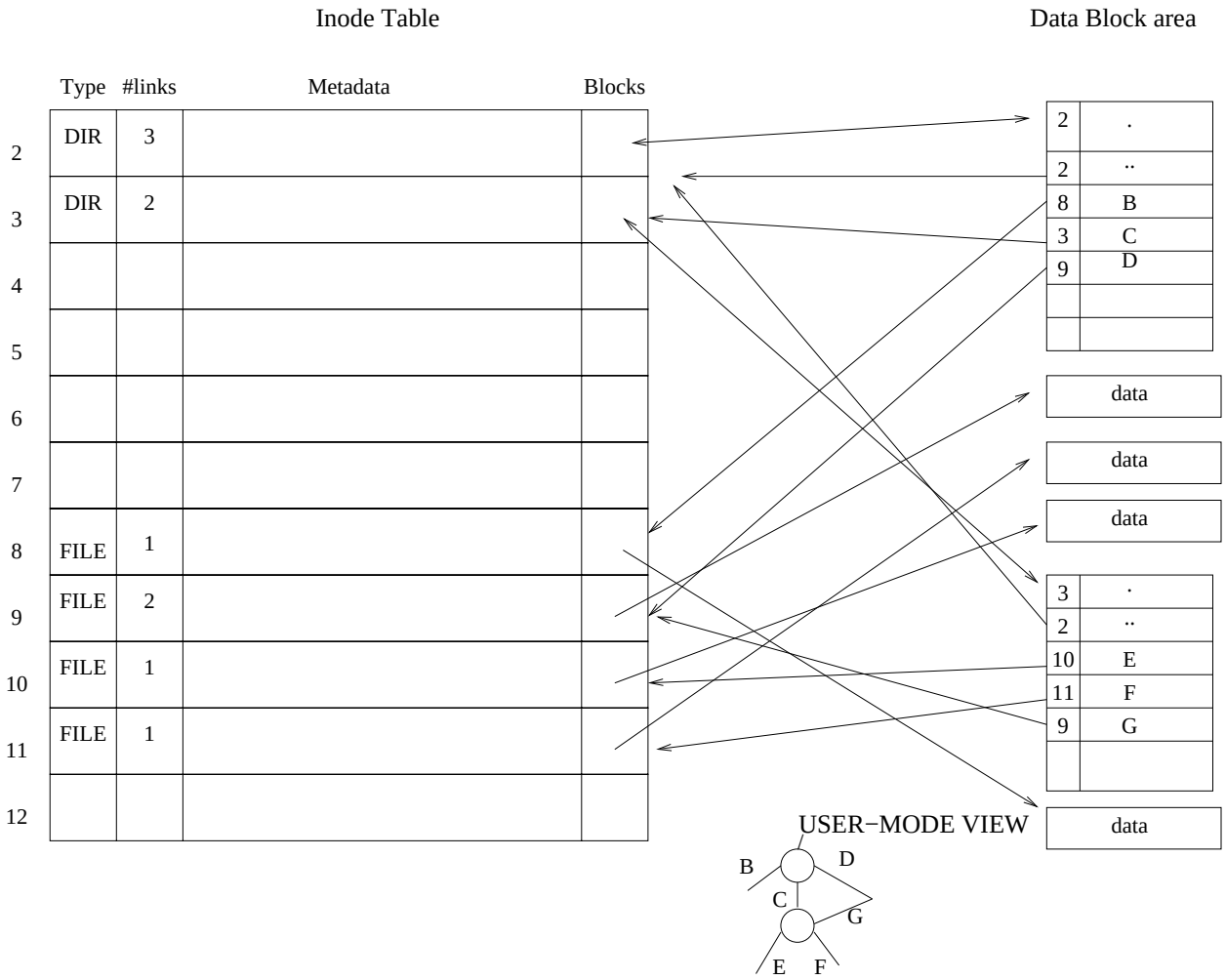
There are two ways to look at this, which are analogous to Hard Links vs Symbolic Links in UNIX. In the latter case, we think of one of the names of the file as being canonical, while all the others are "aliases." In the former case, all of the names are equal in stature.

In the UNIX filesystem, for a given inode number, there can be multiple directory entries throughout the volume which refer to that inode number. In other words, there can be multiple paths (above and beyond those created syntactically by the use of . .. and multiple slashes) that resolve to the same inode.

The system call `link` creates a hard link. Consider:

```
link("/D","/C/G");
```

This creates another name ("/C/G") for an existing file ("/D"). The existing file must actually exist and the new name must not already exist prior to the `link` system call. Here is the situation after executing this call:



Once the link has been completed, the "old" and the "new" names are indistinguishable. There is no way to learn which was in fact the old name and which the new. This is a by-product of the UNIX philosophy which de-couples the name of the file from the actual file.

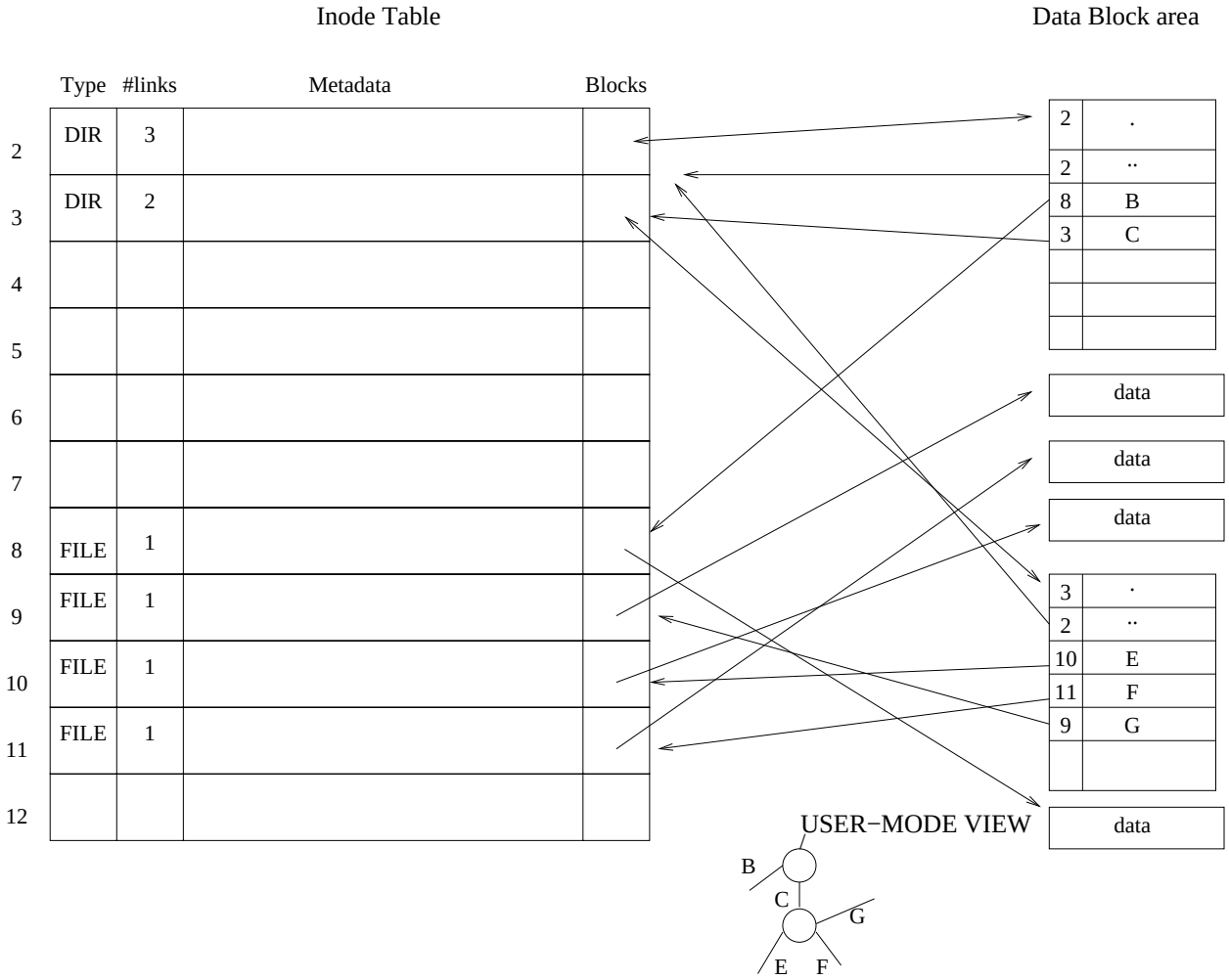
Notice that the inode #9 field *links* has gone up. The operating system must maintain this counter in order to be aware of how many paths exist in the filesystem which point to this inode. We could now remove the pathname "/D", by executing the UNIX command:

```
rm /D
```

This command will, in turn, execute the underlying `unlink` system call:

```
unlink("/D");
```

After which, the filesystem looks like this:



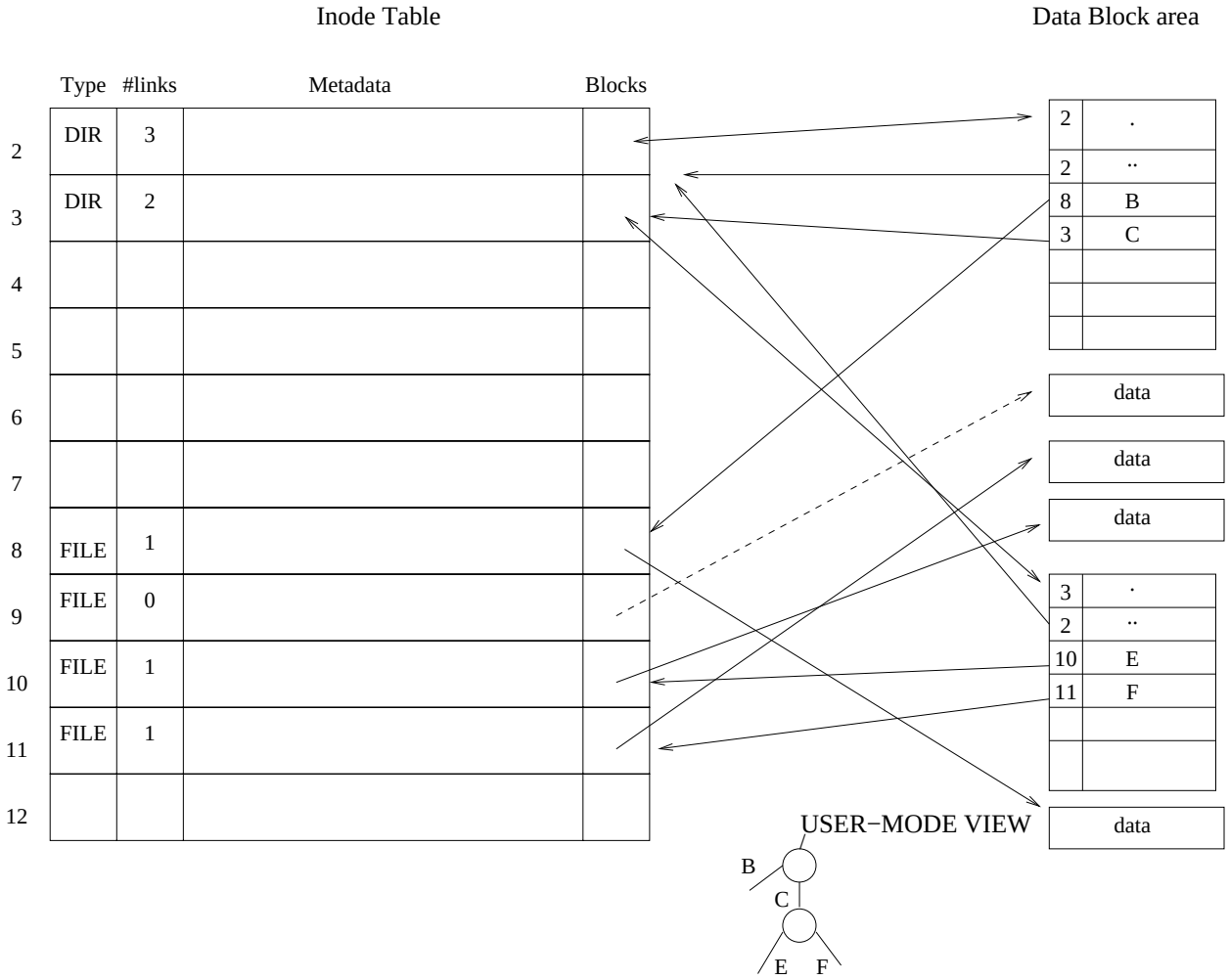
Even though we have issued the **rem**ove command, the actual file represented by inode #9 has not been removed! There is still another pathname which refers to this file ("/C/G"). The file will not actually be deleted until its link count falls to 0, i.e. there are no more pathnames that point to the file. This is why there is an `unlink` system call in UNIX, but no `remove`, `delete`, `erase` or similarly named system call.

File Deletion

Let us execute:

```
unlink("/C/G");
```

The file system then looks like:



Now there are no pathnames pointing to inode #9. There is no way to access the file ever again. Its link count falls to 0 and it is considered deleted. [Linux calls these "orphan" inodes] Its data blocks are marked as free in the free map and the inode structure itself is marked as free and allowed to be re-used when needed. Note that the deletion of the file does not destroy the actual data, but merely marks those data blocks as free for subsequent use. This can be a security exposure if the file contains sensitive data. [Because the data blocks pointers in the inode are not erased when the link count reaches 0, an unlinked file could be recovered intact if one knows the inode number (or can find it by searching through all of the deleted files for a given pattern). There is no system call to do this, but using a byte-level filesystem editor such as `debugfs` is is fairly easy. However, if the volume has had significant write activity since the accidental deletion, it becomes increasingly likely that either the inode has been re-used or the data blocks have been partially or entirely re-used. This would depend on how much writing has taken place and how tight the volume is on free inodes and free data blocks.]

BUT, if any process happened to have the file controlled by inode #9 open at the time, the resources would not be freed, and the file would continue to be active, until that process

exits or closes the file. This leads to the interesting phenomenon of "ghost files" under UNIX, in which files continue to exist and consume disk space, but can not be opened new. There are some systems programming tricks which exploit this, e.g. the ability to have a temporary working file which is guaranteed to disappear when the process exits.

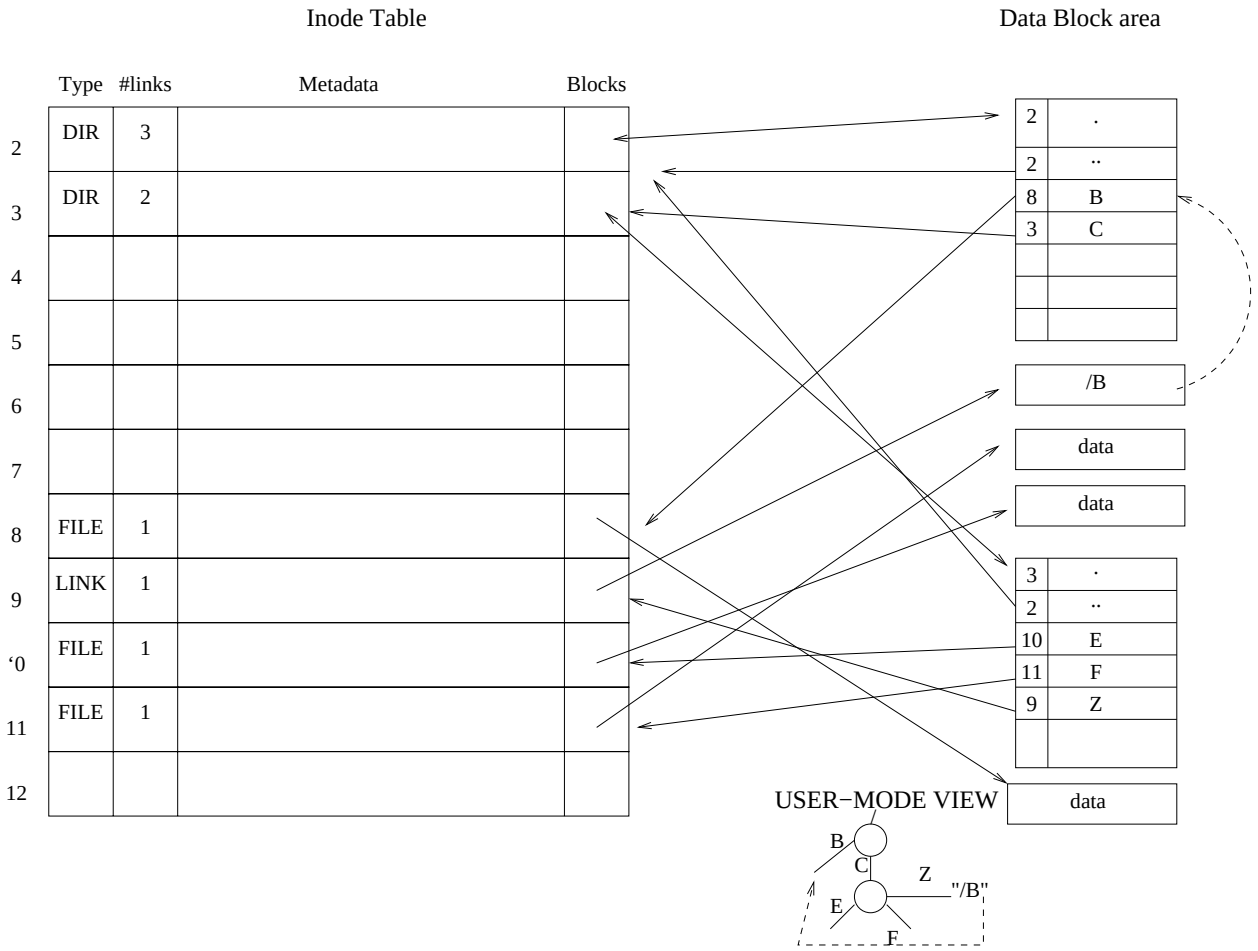
Symbolic Links

Symbolic links, aka soft links, aka symlinks, was a kluge-on feature added to the UNIX filesystem. All modern UNIX variants support it. Unlike a hard link, a symlink is asymmetrical. There is a definite a link and a definite target. The symlink is merely another inode type (S_IFLNK). The data associated with that inode form a string that is substituted into the pathname when the link is traversed. A symlink is created with the `symlink` system call:

```
symlink("/B","/C/Z");
```

The first argument is the target of the link (the existing node), the second is a path name which will be created as the symlink. Note that the "existing" name need not presently exist; it is permissible and even useful to make a symbolic link to a non-existent target. The symlink can start with a leading / in which case it is absolute, otherwise it is relative to the symlink inode's place in the filesystem tree. It is a common idiom to have symlinks with relative pathnames, such as "../../bin/foo"

After executing the system call above, our filesystem looks like this:



Most system calls "follow" symlinks, i.e. they transparently substitute the contents of the link into the pathname and continue traversal. (`open(2)` follows symlinks, unless the flag `O_NOFOLLOW` is given.) `unlink` does not follow symlinks. An attempt to `unlink` a node which is a symlink will cause that symlink to be deleted, but will not have any effect on the target. Also note that no count is kept on the number of symlinks pointing to a particular target. That's why it's a soft link. It is possible to create a circularity of symlinks. This will not be detected until an attempt is made to traverse this loop, at which point the operating system will give an error ELOOP. Most UNIX-like kernels use a fairly dumb algorithm for symlink loop detection which places a static limit on the number of symlink expansions allowed at path evaluation time.

To retrieve the metadata associated with the symlink itself, without following it, use the `lstat` system call, which is identical to `stat` except it does not follow symlinks. To retrieve the text value of the symlink (e.g. `/B`), use the `readlink` system call.

Although the S_IFLNK inode has an owner(uid), group(gid) and permissions mode, these do not quite work the same as other inodes. The permissions mode is never checked and is always 0777. The `chmod` system call always follows the symlink and it is not possible to change the mode of the symlink inode itself. The gid is never checked. The uid is

checked only when attempting to unlink the symlink inode itself and the containing directory has the "sticky bit" on. The symlink inode has the usual atime/mtime/ctime timestamps. The atime is updated whenever the symlink is "read" (either explicitly with `readlink` or implicitly when the kernel follows the link). Since there is no "writelink" syscall, the mtime is always the time of link creation. The ctime is affected the same as other inode types.

Symlinks are useful when it is desired to preserve the distinction between the "real" file and its "alias". Most other operating systems provide an equivalent mechanism. In fact, the hard link is fairly unique to UNIX. A restriction of the hard link, which a symlink overcomes, is that it is not possible to make a hard link across **volumes**. The reason for this will become clear very shortly.

Directories and link counts

A new directory is created with the mkdir system call:

```
mkdir("/foo/bar",0755);
```

The second parameter is the **mode**, or permissions mask, which we will explore below. `mkdir` creates an empty directory with but two entries, "." and ".." In traversing a UNIX pathname, the component "." refers to the same directory, and ".." refers to the parent directory. For historical reasons having to do with simplifying the path name evaluation routine in the kernel, every directory contains an entry for "." and an entry for ".."

Therefore, an empty directory has a link count of 2: one link is the "." entry of the directory itself, the other is the entry in the parent directory pointing to child directory. Whenever a subdirectory is created, the ".." entry of the subdirectory effects a link back to the parent directory and increments the parent directory’s link count by 1.

Probably one of the first problems discovered with hierarchical filesystems is that inadvertent removal of a directory will leave dangling and stranded all subdirectories and files beneath the removed directory, thus having the supplementary effect of recursively removing all of these nodes! Therefore, the `unlink` system call is not valid for directory inodes, and will fail returning the error `EISDIR`. A separate system call is provided:

```
rmdir("/foo/bar");
```

In order for `rmdir` to succeed, the target must be an empty directory, i.e. it must only contain the entries "." and ".." If not, the error `EEXIST` will be returned. To remove a populated directory, one must first explicitly unlink all of the children of that directory (which may involve recursion), then remove the directory itself, thus forestalling the cries of "oops, I didn’t mean to do that!"

Under some versions of UNIX, it was possible to create a hard-link to a directory, but because of the potential confusion, such behavior is strongly discouraged and is disallowed by most modern UNIX kernels. Exploring the amusing consequences of hard-linked directories is left as an exercise to the reader.

Reading directories

Directories, as we have seen, are special files that equate path component names to inode numbers. Directories can be read using a set of standard C library functions:

```
#include <dirent.h>


r(char *fn)
{
 DIR *dirp;
 struct dirent *de;
        if (!(dirp=opendir(fn)))
        {
                fprintf("Can not open directory %s:%s\\n",fn,strerror(errno));
                return;
        }
        while (de=readdir(dirp))
        {
                printf("%s\n",de->d_name);
        }
        closedir(dirp);
}
```

More information about these calls can be gleaned from the man pages. These functions are in section 3 of the man pages as they are technically not system calls. Just as `fopen(3)` is a stdio library layer on top of the `open(2)` system call, `readdir(3)` is an abstraction of the `getdents(2)` system call. Since the behavior of `getdents(2)` is awkward and non-portable, the use of readdir is preferred in all directory scanning applications. The use of the `readdir(3)` family of calls isolates the application from implementation-specific details of directory structure.

The stat system call

Metadata are informational data about a file, directory or other object in the filesystem, distinct from the data, i.e. contents of that object. UNIX provides the `stat` and `fstat` system calls to retrieve the metadata of a node:

```
#include <sys/types.h>
#include <sys/stat.h>

struct stat st;
int fd;

stat("/path/name",&st);                 // This does NOT open /path/name
fd=open("/foo/bar",O_RDONLY);
fstat(fd,&st);
```

The stat structure provides the following information:

```
struct stat {
          dev_t              st_dev;
          ino_t              st_ino;
          umode_t            st_mode;
          nlink_t            st_nlink;
          uid_t              st_uid;
          gid_t              st_gid;
          dev_t              st_rdev;
          off_t              st_size;
          blksize_t st_blksize;
          blkcnt_t  st_blocks;
          time_t             st_atime;
          time_t             st_mtime;
          time_t             st_ctime;
}
```

Most of these fields are stored in the metadata section of the on-disk inode data structure.

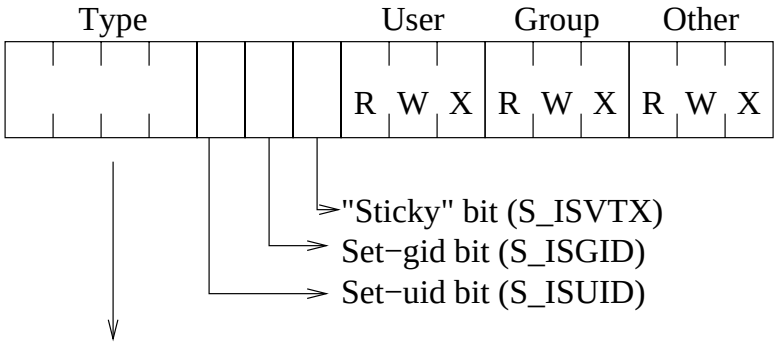
- `st_mode`: The inode type and permissions (see below)
- `st_nlink:` Number of pathnames linking to inode
- `st_uid`: The user id which owns the inode
- `st_gid`: The group owner of the inode
- `st_rdev`: "Raw" Device number (character and block device special inodes only)
- `st_size`: The size of the data, in bytes, if applicable (some inode types do not have a size, such as device inodes). The `st_size` is one greater than the byte position of the last byte in the file. However, it is possible in UNIX to have sparse files, e.g. bytes 0 and 65536 have been written, but all contents in between are undefined. Undefined areas do not occupy storage space and return 0 when read.
- `st_blocks`: The number of blocks of storage space occupied by the file, measured in units of 512 byte sectors. It will generally be a little larger than `st_size` because of the granularity of disk block allocation. However, for sparse files, disk space consumption may be less than `st_size`.
- `st_blksize`: The "best" buffer size to use with operations on this inode. This is

generally the filesystem allocation disk block size.

• `st_atime, st_mtime, st_ctime`: There are 3 timestamps contained within the inode. Each is a UNIX Time, i.e. the number of seconds since midnight January 1, 1970 UTC. [On some filesystems, higher-resolution timestamps are available] The `st_mtime` is the last time a write operation was performed to the *contents* of the file or directory. `st_atime` is the time of the last read operation. `st_ctime` gets touched whenever one of the *metadata* are modified. The `utime` system call can be used to directly modify the atime and mtime stamps, but the ctime field can not be changed. The `touch` command uses the `utime` system call to update timestamps.

`st_blksize`, `st_dev` and `st_ino` do not appear anywhere in the on-disk inode. They are are added by the operating system. The blocksize is a property of the particular volume in which this inode appears. `st_ino` is the inode number, which is inferred by the inode's position in the inode table. `st_dev` identifies the volume on which this inode resides. This will be discussed later in this unit.

The `st_mode` field is a 16 bit bitmask, as follows:



0000	Deleted/Free inode	
0001	Named pipe (FIFO)	(S_IFIFO)
0010	Character Device	(S_IFCHR)
0100	Directory	(S_IFDIR)
0110	Block Device	(S_IFBLK)
1000	Regular File	(S_IFREG)
1010	Symlink	(S_IFLNK)
1100	Network/IPC socket	(S_IFSOCK)

The top nybble of the mode field identifies the type of node. Macros are provided in <sys/stat.h> to give symbolic names to these types:

```
if ((st.st_mode&S_IFMT)==S_IFDIR)
{
        printf("Directory\n");
}
```

Inode Types

There are 15 possible inode types. Inode type 0 is generally reserved to mark a free or deleted inode. The seven inode types depicted in the figure above represent the major, universal types.

• S_IFREG (type==8): A regular file.
• S_IFDIR (type==4): A directory.
• S_IFLNK (type==10): A symbolic link.
• S_IFCHR (type==2) and
S_IFBLK (type==6): UNIX gives names to devices and provides access to them through the filesystem. E.g. `/dev/sda1` is a file-like node in the filesystem which provides direct access to the first partition of the first hard drive. Within the kernel, devices are identified by an integer device number. The Character Special and Block Special inode types provide a mapping from a pathname to a device number, using the `st_rdev` field. This will all be discussed in a subsequent unit.
S_IFIFO (type==1): A FIFO or "pipe". To be discussed in a later unit.

• S_IFSOCK (type==12): A networking socket. To be discussed in a later unit.

In addition to these, some inode types are/were used only in certain variants of UNIX, and are considered non-portable:
• S_IFMPC (type=3): Obsolete multiplexed character special device
• S_IFNAM (type=5): Obsolete XENIX named file
• S_IFMPB (type=7): Obsolete multiplexed block special device.
• S_IFCMP (type=9): Compressed file, proprietary to Veritas, or "network special" file on proprietary HP-UX operating system.
• S_IFSHAD (type=B): Shadow inode for ACL extensions under some versions of Solaris. Never seen by user-mode programs.
• S_IFDOOR (type=D): Proprietary IPC mechanism under Solaris.
• S_IFWHT (type=E): "Whiteout". An obscure topic which falls outside of the traditional filesystem model, and comes into play with "union mounts".
• S_IFPORT (type=E): Solaris (version 10 and higher) uses this type for an "event port", which provides a way for a process to place watchpoints on various system events or filesystem accesses.

The "sticky bit" S_ISVTX was historically used as a hint to the virtual memory system but now has a different meaning associated with directories (see below). The set-uid and set-gid bits, when applied to executable files, cause the effective user or group id to be changed. This allows a non-superuser to gain controlled access to superuser powers through specific commands. The set-gid bit is also used, on non-executable files, to indicate that file and record locking should be strictly enforced. This subject is beyond the scope of this introduction. Additionally, when the set-gid bit is set for a directory, nodes created in that directory take the group ownership associated with that directory, rather than the gid of the running process.

The remaining 9 bits determine the permissions associated with the node.

The UNIX file permissions model

Every user of the UNIX operating system has an integer **user id**. For the purposes of group collaboration, users may also be lumped into groups identified by an integer **group id**. Historically, uids and gids are 16 bit numbers, although modern Linux systems support 32-bit values. Each running program, or **process**, has associated with it the user id of the invoking user, the group id of the user's primary group, and a list of groups (including the primary group) to which the user belongs.

Every inode has an individual owner, `st_uid` and a group owner `st_gid`. This ownership is established when the node is first created. The uid of the node when created is the (effective) uid of the process, and the gid is the (effective) primary gid of the process (but see below about the set-gid bit and directories).

When a system call operation requires checking of filesystem permissions, the first step is to determine which of the 3 sets of 3-bit permissions bit masks to extract from the `st_mode` field:
• If the user attempting an operation matches the owner of the file, the user portion of the permissions mask is checked.
• Otherwise, if the group ownership of the file is among the list of groups to which the current process belongs, the group portion of the mask is checked.
• Otherwise, the "other" portion is used.

• Once the appropriate mask is selected, the read, write or execute bit is consulted based on the operation being attempted.
• For files, permissions are checked once, when the file is opened or execution of the file is attempted. If the file is being opened O_RDONLY, read permission must be present. If the file is being opened O_WRONLY, write permission must be present, and in the case of O_RDWR, both permissions are needed. Once the file is opened successfully, changing the permissions on the file has no effect on programs that already have the file open. Execute permission is checked when one attempts to use a file as an executable program (e.g. a.out). This will be covered in a subsequent unit.
• What should write permission for a directory mean? Being able to write to a directory implies the ability to create new directory entries, or to modify or remove existing ones. I.e. directory write permission allows creation, renaming, or unlinking of the nodes within. This original interpretation was found to be problematic in shared directories, (such as /tmp which is generally 777 mode) in that another user might be able to delete a file which s/he did not own. The presence of the "sticky bit" in the permissions mode modifies the semantics of writable directories such that only the owner of a file can rename or unlink it.
• Read permission on a directory implies the ability to search the directory and learn the contents.
• Execute permission is the ability to traverse the directory, that is, to reference an element of the directory. One can have execute permission but not read permission on a directory, allowing one to access files or subdirectories as long as their name is known.

The uid and gid of a node can be changed (at the same time) with the `chown` system call. The permissions of a file can be changed with the `chmod` system call. In both cases, the user attempting the operation must already be the owner of the file (uids match). Furthermore, on most UNIX systems, to avoid problematic interactions with quotas, file "giveaways" are not permitted for ordinary users, i.e. an ordinary user can change the group id of their files but can’t change the individual ownership to another user.

The user with uid 0 is the **superuser**, or **root** account, aka the system administrator. When the process uid is 0, all of these permissions checks are bypassed.

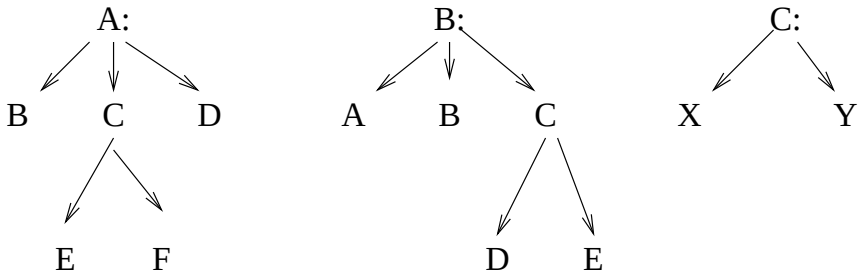
Most UNIX systems support a more elaborate way of expressing filesystem permissions known as **Access Control Lists**. Their application is not widespread because the

traditional 3-tiered UNIX permissions model is sufficient for most applications.

Mounted Volumes

Of course a system that supports just a single random-access storage device is not very useful. We have defined a **volume** to be one instance of such a device. Each volume is an independent filesystem data structure which can be detached from the system and attached to another system. Some types of volumes are designed to be removable (e.g. a flash drive) while others require more effort to relocate (e.g. a hard disk).

When a volume is attached to a system and available to users as a file store, it is said to be **mounted**. Many operating systems take the "forest of trees" approach to multiple volumes. For example, Microsoft operating systems such as DOS and Windows assign drive letters starting with A: to each volume:

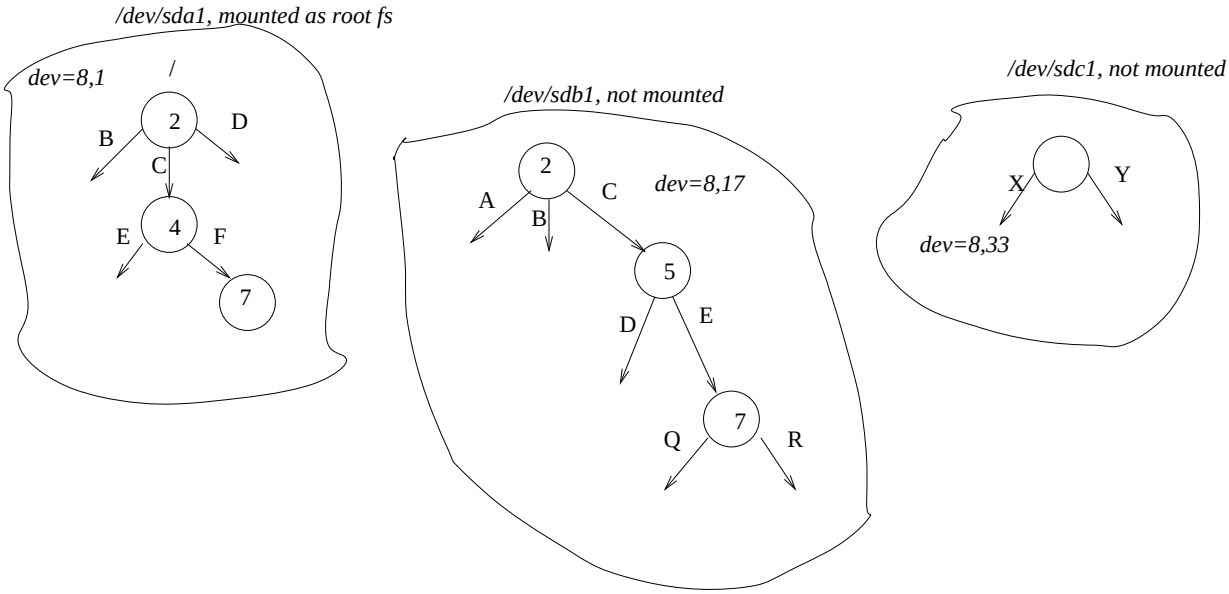


Each volume is an independent tree, and the collection of all such trees forms a flat namespace.

UNIX takes a "big tree" approach:

When a UNIX system first comes up, there is only one volume mounted. This is known as the **root filesystem**. The root of this volume is "/", the root of the entire namespace. Additional volumes get mounted over empty place-holder directories in the root filesystems (or recursively: a volume can be mounted on another volume which is in turn mounted on the root volume, etc.)

Below we see a system where the root filesystem resides on disk partition `/dev/sda1`. Two other partitions on other drives, `/dev/sdb1` and `/dev/sdc1` are present but are not yet mounted and thus not visible. (For clarity, inode #s for non-directory nodes are omitted)

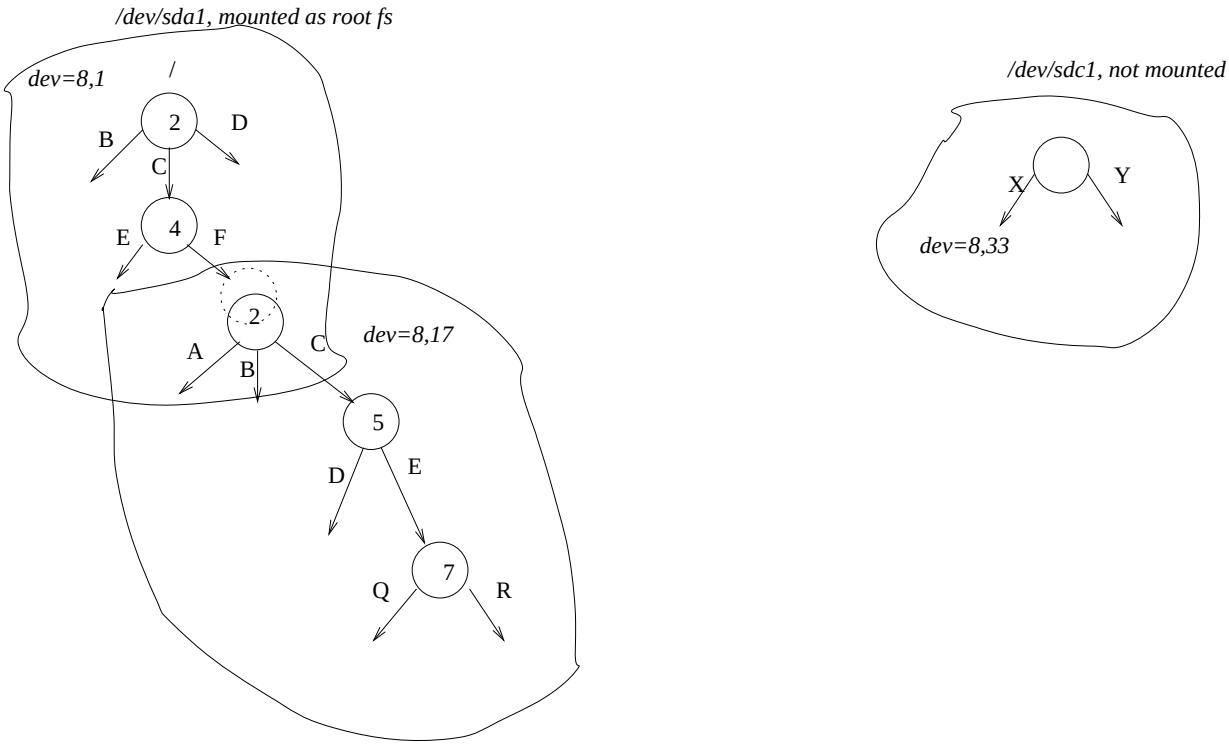


Now we execute the command:

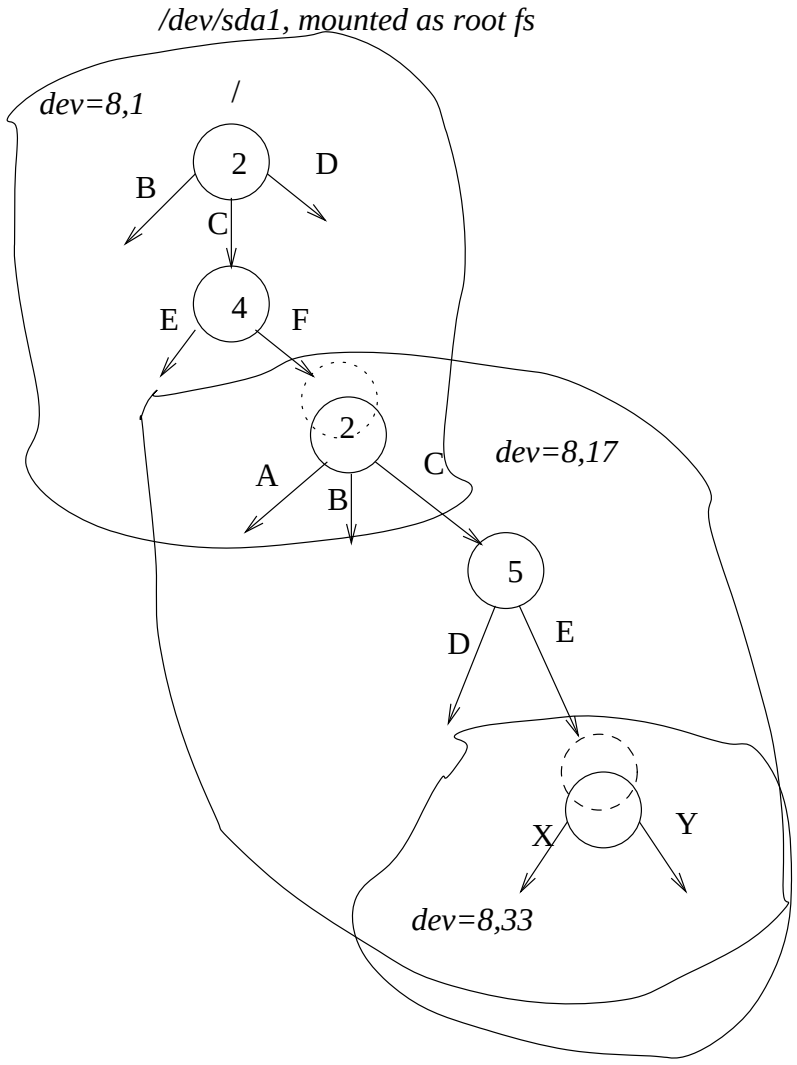
```
mount /dev/sdb1 /C/F
```

The pathname `/dev/sdb1` is in a special part of the filesystem in which the nodes are device special inodes. Disks are of type S_IFBLK. When the kernel translates this pathname, it produces a major,minor device number pair, e.g. 8,17. This uniquely identifies the first partition of the second SCSI/SATA hard disk on the system.

Pathname /C/F becomes the **mount point**. It was inode #7 in the root filesystem. That inode now becomes obscured, and is replaced with the root inode of the *mounted* volume, which is inode #2:



A mount point must be an existing directory in an already-mounted part of the path name space. Normally, it is an empty directory. But if the directory had contents, they become obscured by the mount:



Here we have performed `mount /dev/sdc1 /C/F/C/E`. The files Q and R, formerly visible through /C/F/C/E (inode #7 in device 8,17) are no longer accessible. If we later `umount /dev/sdc1`, these files will once again be visible.

Note that many UNIX kernels support the concept of a "union" or "overlay" mount in which both the newly mounted and the mounted-over volumes are visible. This feature is useful, e.g., when working off a large read-only volume such as a DVD-ROM while needing to make changes on the fly. We will not be considering this further in this course.

The kernel keeps track of which inodes are being used as mount points, and whenever the path name resolution algorithm within the kernel traverses a mount point inode, that inode is not considered further, but instead is replaced in the traversal by the root inode of the mounted volume. This applies in both downward and upward traversals. Consider what happens if we had entered the directory /C/F/C and then accessed the path "../../E".

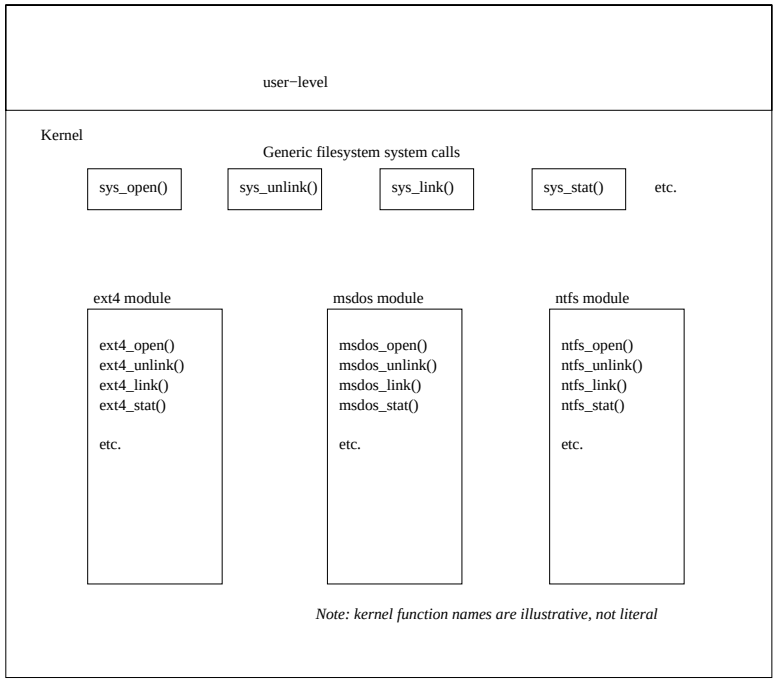
This mounting of a volume onto the existing filesystem hierarchy is not permanent. There is a corresponding `umount` operation, and a corresponding `umount` command, which removes the volume from the filesystem and unveils the original mount point again. In order to unmount a volume, there must not be any open dependencies on it (e.g. a process with an open file, executable or current working directory that is within that volume, or another volume mounted within the volume being unmounted).

Alien Filesystems / Virtual Filesystem Layer

A given volume need not always be mounted at the same place in the filesystem. When the media containing the volume is moved to a different machine, the volume may even be mounted by a different operating system. Issues can arise, e.g. with byte order, mapping of user identifiers, differences in path name conventions, and other semantics. The UNIX kernel creates an "inode" interface to these alien filesystems. Some features which may be present in other operating systems, such as "resource forks" in traditional Macintosh OS, may not map cleanly to UNIX semantics, and new system calls were added to the traditional UNIX calls to provide access to these "alien" semantics.

The filesystem in UNIX is heterogeneous, i.e. the different volumes need not follow the same data structure. This allows a running system to mount volumes which were created under older versions of the operating systems, or entirely different operating systems. For example, while the current default filesystem for Linux is of the EXT3 type, it is just as easy to use the older EXT2 filesystem type, or the very latest EXT4 type. There are other filesystem types which attempt to optimize for certain situations, e.g. ReiserFS. Linux can also deal with the native filesystem layouts of other UNIX variants, such as UFS (BSD and Solaris), HPFS (HP-UX), and non-UNIX systems such as MSDOS, NTFS, HFS (older Macs), etc. There is support for adding new filesystem types to the kernel dynamically (after the system is already booted), and even to allow a user-level process to implement a filesystem ("FUSE").

This integration of different filesystem types into the overall hierarchy is known as the **Virtual Filesystem** layer of the kernel. Because all file system interface access flows through the kernel, it can keep track of exactly what parts of the naming hierarchy correspond to what type of filesystem, and delegate to the various filesystem modules which are loaded as part of the kernel. Traversal of a mount point is thus transparent to the user. The UNIX pathname space is a flexible one which is independent of the constraints of physical disks.



Pseudo- and Network- filesystems

A mounted filesystem is not necessarily located on a physical disk device on the local machine. This is known as a **Pseudo-filesystem**. **Network filesystems** are pseudo-filesystems which make parts of a filesystem on a remote machine appear local. Two major examples of this are the NFS (the most popular network filesystem for UNIX-like systems) and the SMBFS (the network filesystem used by Windows).

Other pseudo-filesystems provide a filesystem-like interface to things which have nothing to do with files, hard disks and the like. For example, the `procfs` filesystem found under `/proc` in Linux allows one to enter a directory whose name is equal to a process id, and once in there see the processes's memory map, command line invocation, resource usage, and many other interesting things.

Traditionally, mounting of filesystems is a global operation which is performed by a process running as the super-user (uid==0), and all processes see identical namespaces. In some modern variants of UNIX, there are capabilities to associate a different namespace with different processes or users. This can be used to some advantage in securing an application and restricting its access to the filesystem.

Loopback and RAM filesystems

Most UNIX kernels support a "loopback" mount where a regular file that currently exists

someplace in the filesystem tree can be treated as a raw disk image. E.g. one can download a `.iso` file which is an image of a CD or DVD-ROM and then mount that file directly, instead of "burning" the CD/DVD.

Most UNIX kernels support a "ramdisk" where an area of RAM is treated as if it were a volume. This is especially prevalent in "live" media, e.g. a CD/DVD or USB stick that contains a "live", bootable Linux system. We need a place for the root filesystem, including places to write things, but we don't necessarily want to rely on existing hard disks. The union or overlay mount technique can be used here to merge an initial filesystem read-only image from the boot medium with the ramdisk. Any changes are simply discarded when the system reboots.

The Mount Table

Conceptually, the kernel keeps a table in kernel memory to track each mounted volume:

Device	Mounted on	FSType	Flags/Properties
/dev/sda1	/	ext4	rw
/dev/sdb1	/C/F	ext2	rw, noatime
/dev/sdc1	/C/F/C/E	vfat	ro

An example above illustrates the 4 fields of this table. The device field identifies the underlying physical device. The next column is the mount point. The third column is the filesystem type. The last column is a list of flags or properties. E.g. the "rw" flag means that read/write is allowed to the filesystem, whereas "ro" means the kernel will never attempt to write to the volume. The "noatime" property says that when files are read, the usual update of the inode `atime` will not take place. This is often set to improve performance where keeping track of atime has no benefit.

One can query the mount table either by using the `mount` command with no arguments or (on Linux systems) `cat /proc/mounts`. You will find a lot of other lines in this table that correspond to pseudofilesystems, such as the `/proc` filesystem itself, that are necessary for system function.

Device Numbers & Inode Numbers

As we have seen, the UNIX kernel associates an integer **device number** with each volume on the system. The device number is not something which would ever be found on the volume itself, rather it is a tracking number maintained by the kernel. The stat field `st_dev` is filled in with the device number of the volume on which the inode in question resides. Like inode numbers, device numbers should be treated as cookies: they can be compared for equality, but no other assumptions should be made about their properties. By convention, UNIX device numbers are split bit-wise into a most significant word, which is the **major device number**, and a least significant **minor**

device number. Typically these are represented as a pair of decimal numbers, e.g. in the device number 8,16 8 is the major part and 16 is the minor part. If the kernel in question is using 16-bit device numbers, this would be 0x0810. The significance of major vs minor numbers is beyond the scope of this unit.

By examining the `st_dev` field, the user can determine if two paths reside on the same volume. In the example above, stat'ing `/C/F` would yield the `st_dev` device number of the second volume, not the root volume, because the original mount point in the root volume is inaccessible.

Because each volume is a self-contained independent data structure, the inode numbers (`st_ino`) are unique only within the same volume. A consequence of this is that hard links can not be made across volumes, because the inode number in the first volume would have no validity in the second volume. The combination of `st_ino` with `st_dev` uniquely identifies any node within the pathname space at the time that comparison is made. Of course, if volumes are later mounted or unmounted by the system administrator, that might invalidate such a test.

Move/Rename

The `rename` system call:

```
int rename (char *oldpath, char *newpath)
```

is used either to rename or move a file within the same volume(filesystem). The same result could be accomplished by using link(oldpath,newpath) following by unlink(oldpath), and indeed very early versions of UNIX did not have an explicit rename syscall. However, consider what would happen if the process dies (e.g. the user hits Control-C -- we'll talk about signals and process termination in a few more weeks) between the link and unlink. The rename system call is atomic. The mv(1) command uses rename(2). Note: it is not possible to move a file using rename(2) from one volume to another, because it is not possible to make a hard link from one volume to another. The mv(1) command hides this annoyance from the user. If oldpath and newpath are on different volumes, the command instead does the equivalent of `cp oldpath newpath ; rm oldpath`. This is not atomic, instead the mv command also uses the utime, chown, and chmod system calls to make newpath as much as possible have the same metadata as oldpath. Note that since the ctime can not be changed via system call, that will be at least one imperfection.

Is that filesystem, or file system, or filesystem?

Unfortunately the terminology pertaining to the file system is often inconsistent, ambiguous and confusing. In operating system literature, "filesystem" can mean:

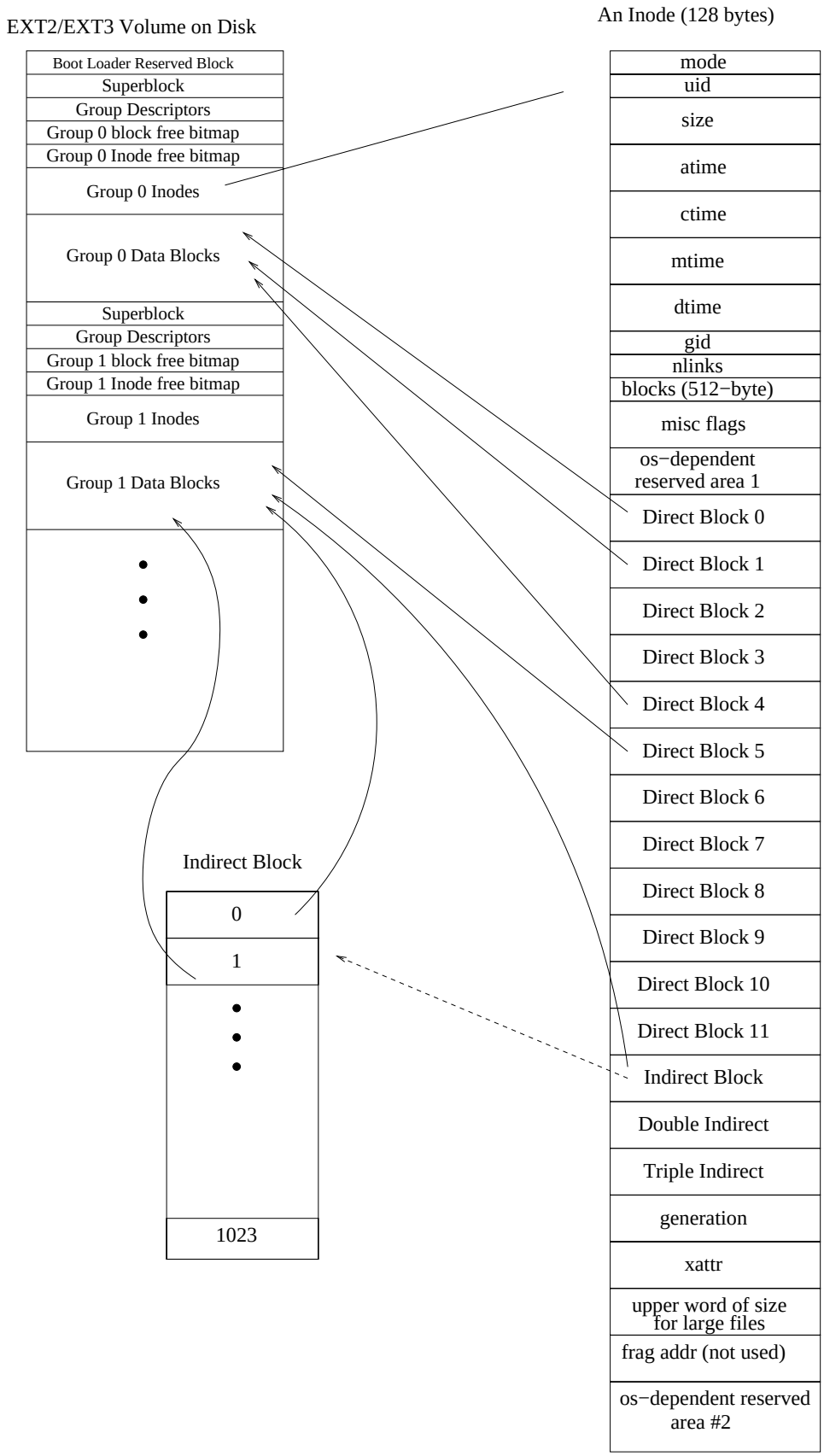
• The overall file system, its semantics and interfaces. e.g. "The UNIX filesystem

provides a simple, clean interface."
• A particular schema for organizing data within a volume, e.g. "The Reiser filesystem performs better than the EXT2 filesystem when there are many, small files."
• A code module within the kernel for implementing a filesystem, in the sense of "filesystem" given in the last item.
• A particular instantiation of such an organization of data. We called this a "volume" in these notes, which term is also used in the literature.

Locality and fragmentation

With traditional hard disks that have moving heads, it takes a longer time to access two sectors in a row that are far apart from each other on the disk than if they are close together, and the time grows with distance. Consider the reference model of a UNIX filesystem presented earlier with Header, Inodes, Free Map and Data Blocks. In order to perform an operation on a file, one needs to access the inode, possibly the free map (if the file is being written to) and the data blocks. It would be nice to keep these things close to each other.

All modern UNIX filesystems use **cylinder groups** or **block groups** (these are essentially equivalent terms):


EXT2/EXT3 Volume on Disk
Boot Loader Reserved Block
Superblock
Group Descriptors
Group 0 block free bitmap
Group 0 Inode free bitmap
Group 0 Inodes
Group 0 Data Blocks
Superblock
Group Descriptors
Group 1 block free bitmap
Group 1 Inode free bitmap
Group 1 Inodes
Group 1 Data Blocks
Indirect Block
0
1
1023
An Inode (128 bytes)
mode
uid
size
atime
ctime
mtime
dtime
gid
nlinks
blocks (512−byte)
misc flags
os−dependent reserved area 1
Direct Block 0
Direct Block 1
Direct Block 2
Direct Block 3
Direct Block 4
Direct Block 5
Direct Block 6
Direct Block 7
Direct Block 8
Direct Block 9
Direct Block 10
Direct Block 11
Indirect Block
Double Indirect
Triple Indirect
generation
xattr
upper word of size for large files
frag addr (not used)
os−dependent reserved area #2

Each block or cylinder group is like a mini-filesystem, containing a portion of the overall inode table and data blocks, and a free block bitmap covering just the data blocks in that cylinder group. The kernel will attempt to keep a file allocated all within the same cylinder group. This improves locality and filesystem performance. Inode numbers and block numbers continue to have global meaning, but the inodes and blocks are now spread out among the groups instead of being concentrated together.

You will also notice that Linux uses a second bitmap to keep track of free vs in-use inodes. This improves performance..the inode bitmap often fits within a single disk block, which remains cached in kernel memory anyway. Finding a free inode is a quick in-memory bitmap search, rather than a series of disk accesses to examine each inode to see if it is free.

You may also notice that a space for the volume Superblock appears in each block group. In practice, several replicas of the superblock are stashed in block groups in addition to group #0. Whenever the superblock is flushed to disk, these additional disk writes are also needed, so we don't want to actually replicate the superblock in *every* group. Having replicas means it is still possible to recover the volume if the main superblock gets corrupted or if that sector of the disk goes bad.

Fragmentation is defined as the case where a file's contents are not stored in contiguous data blocks. Fragmentation is impossible to avoid unless one wants to sacrifice a lot of disk space by employing what is known known as a **contiguous filesystem**. This is only used in very limited circumstances, such as extremely limited embedded environments.

Excessive fragmentation will cause a lot of disk seek activity to access a file and is undesirable. To combat fragmentation, the kernel can avoid allocating space for a new or growing file which is right up against the space of another file. But as the total space used on the volume approaches the total available space, it becomes harder to avoid that. Many system administrators tune filesystems with a **reserve factor** of 5-10%. On a volume with 1GB of data block space and a 10% reserve factor, it would appear from the `df` command that there is only 900MB free. Ordinary users will start to get "disk full" errors when the space in use exceeds 900MB, but in reality 100MB is being held in reserve for system processes (which usually run as "superuser"). With 10% of the volume still free, the kernel is usually able to avoid excessive fragmentation. Volumes that are running below 5% free space typically start to have performance issues from fragmentation.

A filesystem with a lot of fragmentation can be "de-fragmented" if there is sufficient free space. This involves first moving smaller files around on the volume so as to open up larger contiguous regions of free space, to which larger files are moved. This process can take many hours and could potentially cause filesystem corruption if it crashes in the middle. It is best done of volumes that are not currently mounted.

Keeping track of disk blocks in the inode

Inodes which are Directory, Regular File or Symlink store data in the data blocks part of the filesystem. We will now take a peek at how UNIX operating systems translate offsets within a file (or directory or symlink) into disk data block numbers so that the file (or directory or symlink) can be read or written. Traditionally, many UNIX filesystems, including the original (Version 7 UNIX), System V, UFS (BSD) and EXT2/EXT3 have used the following model (which is illustrated several pages back):

There are 12 **direct block** slots which give the block numbers of the first 12 block-sized chunks of the file. E.g. when the filesystem block size is set to 4K (the default on Linux), then slot[0] gives the block number which holds bytes 0..4095 of the file. slot[1] gives bytes 4096..8191, etc.

Now, if the file size exceeds 12*block_size, the **single indirect** block comes into play, which is found at slot[12]. This gives the block number of a block in the data block section, which is marked as allocated in the allocation bitmap, but doesn't actually store any file data. Instead, this block contains an array of block numbers. Let us say that block numbers are represented as 32-bit unsigned integers and the block size is 4K. The first indirect block then maps 1K * 4K = 4M worth of the file, from offset 48K to 48K+4M.

OK, what happens if the file is bigger than 4,243,456 bytes? Now we go on to the **double indirect** block at slot[13]. It contains the block number of a block which contains an array of block numbers, each of which is a single indirect block. Again, if block numbers are 32 bit and blocks are 4K, the double indirect block covers 1K*4M or 4GB of the file.

Twenty years ago, the idea of a file larger than 4GB+4M+48K was ludicrous when a 9GB hard drive cost a thousand dollars. But of course today such large files are common. So we get to slot[14] which is the **triple indirect** block, which is an array of double indirect block numbers, each of them being an array of single indirect block #s, ad nauseam. Keeping in line with our example numbers, the triple indirect block covers 1K*4G=4TB. 4TB+4GB+4M+48K would then seem to be the largest file that could be kept under this representation with 4K blocks.

*Aside: If one searches for "maximum file size" for the older EXT2/EXT3 Linux filesystems, the result is 2TB. This is because there is a field in the on-disk inode called i_blocks which is 32 bits long and which is in units of sectors (512 bytes), not disk blocks. This field describes the number of 512 byte sectors that the file consumes. 2^32 * 2^9 = 2^41 or 2TB. Because this field would overflow beyond 2TB, the kernel prevents a file from exceeding that point. EXT2/EXT3 use 32-bit block numbers, thus with a 4K disk block size, the largest volume (filesystem) which can be created is 16TB. EXT4 addresses*

some of these limitations.

Inline file storage

In the Linux EXT2/EXT3 implementations, block numbers are 32 bits (4 bytes) and the block map portion of the inode is therefore 60 bytes long. As a further optimization, files which are 60 bytes or less could be stored directly in that 60-byte area and not require any additional disk accesses, beyond the one to retrieve the inode. This optimization is particularly helpful for symlinks which are often very short.

Sparse Allocation

Consider the following code:

```
fd=open("file",O_CREAT|O_TRUNC|O_WRONLY,0666);
write(fd,"X",1);
lseek(fd,16384,SEEK_SET);
write(fd,"Y",1);
```

The file contains an X at offset 0 and a Y at offset 16384. What is in between? Nothing has ever been written there. The philosophy that UNIX takes is that this is a "sparse" file, i.e. a file with a "hole" in it. The data from 1..16383 are not defined, but for the sake of consistency and security, they read back as all-0 bytes. Now, since nothing is really there, do we need to allocate disk data blocks? UNIX does not. In the inode, the [0] slot and the [4] slot would have valid block numbers, but the [1] [2] and [3] slots would have 0 as the block number, indicating that the corresponding 4K region of the file is not allocated.

The field `st_blocks` in the stat structure will report 16 (remember that this field is in units of 512-byte blocks), i.e. only 8K is being allocated to the file. On the other hand, **st_size** will be 16385, which at first glance would seem to require five 4K blocks. Try it at home, folks.

Extent-based allocation

The block map (direct/indirect/double/triple) method has worked very well for many, many years. It is particularly efficient for small files (e.g. under 48K in our example above). However, for larger files, it requires additional disk accesses to get at the indirect blocks. Indirect blocks are cached (see below under "buffer/block cache") but for random-access to a large file in the worst case, it could require several disk accesses per data block access.

In the Linux world, the EXT4 filesystem has moved to extent-based allocation. Other filesystems also use this. An **extent** is defined as a *contiguous* group of disk blocks. Therefore it can be described by an **extent descriptor**:

```
                          Linux EXT4 Extent Descriptor
                    ***********************************************
start_file_block_offset    32bits    The offset of this extent within the file's
                                          address space, measured in units of
                                          FS block size
start_disk_block_num       48 bits   Disk block number of start of extent,
                                          units of fs blocks
block_count                15 bits   Number of contiguous blocks
flag                       1 bit     Flag to indicate empty extent
```

Note that the starting disk block number is a 48-bit quantity, which overcomes the limitations of EXT2/3 of 16TB volume size. With these numbers, and a block size of 4K, the volume can be 2^12 * 2^48 = 2^60 =1 Exabyte! A single extent descriptor can cover a maximum of 2^15 * 2^12 = 2^27 = 128MB of file address space (with 4K block sizes.)

In this implementation, each extent descriptor is 12 bytes long. EXT4 also uses a 12-byte header for the entire extent data structure. Therefore, within the existing 60-byte area used for the block map, EXT4 can store 4 extent descriptors (4*12=48 + 12 for the header = 60). Now, whereas the block map approach always required a fixed number of 4-byte block numbers to hold the map for a file of a given size, with extent-based allocation, the number of 12-byte extent descriptors is variable depending on the fragmentation of the file. If the file is allocated contiguously, these 4 extent descriptors could map the first 512MB of the file. Extent descriptors are always maintained in sorted order by start_file_block_offset, so a simple binary search will find the descriptor that maps the area of the file that we are interested in.

To map larger files and/or if fragmentation requires a greater number of extent descriptors, the extent data structure becomes a special form of tree with the property that it is of uniform depth (i.e. at any given moment, the number of internal nodes traversed to reach a leaf node is constant for all leaf nodes). The leaf nodes of this tree are the extent descriptors. A tree of depth 0 is the case where 4 extent descriptors are stored inside the inode. At depth 1, the inode contains 4 internal nodes which contain (start_file_block_offset,next_disk_block). Again, these internal nodes are sorted by start_file_block_offset, but now they contain "pointers" to disk blocks which contain one 12-byte header followed by an array of leaf nodes (extent descriptors) which are sorted by start_file_block_offset. A 4K disk block holds 340 leaf nodes with 4 bytes wasted at the end (these 4 bytes can be used for a checksum to improve filesystem integrity checking).

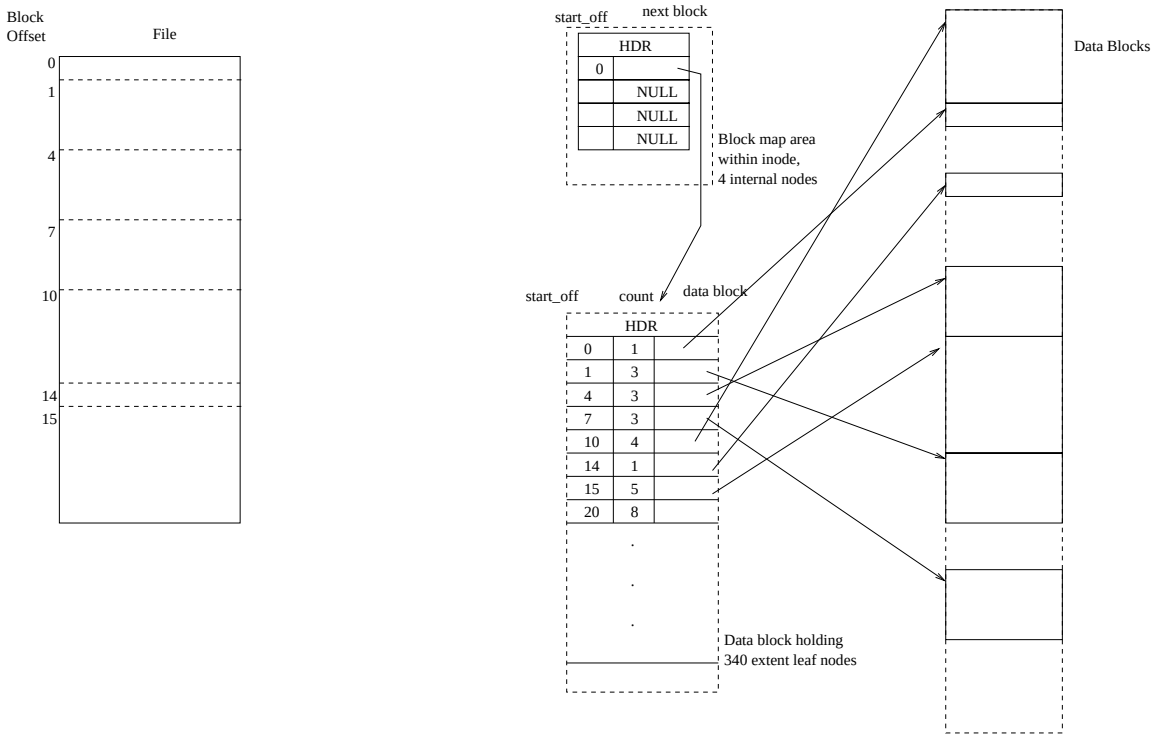
If the file requires more than 1360 extent descriptors, now we have to move to a 2-level tree. The 4 entries in the inode now each point to disk blocks which contain 340 internal nodes which point to additional disk blocks which finally contain 340 leaf nodes (extent descriptors). Where does it end?

The theoretical worst case is a file that is completely fragmented, such that each extent

descriptor has a block_count of just 1. Since start_file_block_offset is a 32-bit quantity, we would then need 2^32 leaf nodes (of course the filesystem would be useless at this level of fragmentation). A 4-level tree would hold 4*340^3 or 157,216,000 block numbers which is too small, so a 5-level tree is needed which could hold 53,453,440,000 block numbers. This is the maximum tree depth that would be needed, but of course if the file can be mapped in fewer levels, it is.

This data structure would be a poor choice in the general case but for file allocation, once an area of the file has been allocated, the block numbers assigned to that area will generally not change. If the file grows, new extent descriptors can simply be appended, since the previously allocated ones are already in file offset order. When we run out of room to represent the extent tree with the current number of levels, the entire existing tree can remain and a new higher level is created with a pointer to the existing tree.

Extent-based allocation is also able to support sparse allocation. In this case, a flag in the extent descriptor indicates that the corresponding number of blocks represent a "hole", and the start_disk_block_num field is not valid.



Caching

In general, the CPU is faster than RAM and RAM is faster than mass storage. **Caching** is a concept which will arise repeatedly in this course. The basic premise is that if you do something once, you are likely to do it again, or something very much like it or near it, in

the near future. Therefore, it usually pays to keep a copy of something which is kept in slower, plentiful, cheaper storage temporarily in faster, scarcer, pricier storage.

When talking about filesystems, caching is going on at multiple levels.
• Individual sectors (or disk blocks) are cached in the **buffer cache**
• The superblock (volume header) is cached by keeping the "live" copy in kernel memory whenever a volume is mounted.
• Individual inodes are cached by keeping a live copy in kernel memory whenever the inode is needed, e.g. when the file is open, during directory traversal, during a `stat` system call, and many other cases.
• Directory entry lookups are cached in the **dentry cache** (Linux terminology, other non-Linux UNIX kernels have called this the namei cache)
• Reads and writes of regular files are cached by keeping copies of the file contents in kernel memory. We'll also learn in Unit 5 about file/memory equivalencies.

Each of these caches (with the exception of the superblock which is only caching one thing) is, like any cache, an associative array. The cache is addressed by a "key" which results in either a hit, in which case the requested item is in memory, or a "miss" which requires loading it from disk. In some cases, the key is simply the direct "address" of the object (such as a sector number) but in other cases the key is part of the object's data itself (such as directory entry names).

All caches have overflow issues: if there is no room to cache a new object, some older object must be released from the cache, possibly writing it back to the disk. All of these filesystem caches will release all of their in-memory resources and synchronize the data back to disk when the volume is unmounted. We'll now look at each cache briefly.

Buffer Cache

One view of the volume is as just a collection of sectors. The buffer cache (or "block cache") is keyed by (`device_num,sector_num`) and objects in the cache are the sectors (or "disk blocks" which may be multiple sectors). Since most of the sectors of a filesystem represent more high-level objects, such as inodes, directories or files, the buffer cache is often superceded by the other caches below. The buffer cache does come into play with caching indirect blocks and inode/data block allocation bitmaps.

Inode Cache

Whenever any system call is performed on a node in the filesystem, the kernel needs read (and potentially write) access to the corresponding inode fields. The inode cache is keyed on (`device_num,inode_num`) and contains "in-core inodes". This data structure, kept in kernel memory, has all of the fields of an on-disk inode, plus numerous other

fields that the kernel needs, such as pointers to the correct virtual filesystem modules for the filesystem which contains that inode. Once an inode is brought into kernel memory, it generally is locked there until no longer needed. In particular, this means that any open file has an in-core inode.

Dentry Cache

In the Linux kernel, the dentry cache is keyed on (`inode,component_name`) where `inode` is a pointer to the in-core inode which is the directory containing the entry in question. Another way of viewing this is that the key is (`device_num,inode_num,component_name`). Whenever a directory must be searched, such as during path name evaluation, the (device_num,inode_num) of that directory is presented to the dentry cache along with the component_name being searched. Unlike the other caches we have seen, the dentry cache is also a **negative cache**. If there is no dentry object corresponding to the component_name, this results in the filesystem module's directory lookup function being called on the directory inode. How this is implemented is obviously filesystem-dependent (a directory lookup on an MSDOS/FAT filesystem will be very different from ext4fs). The result of this function call is either the inode number (if the component was found) OR it is nil (not found). In either case, the result must be cached (negative caching).

The dentry cache must be notified whenever a filesystem operation happens that affects a directory entry. I.e. an unlink or rename system call must cause the corresponding cached dentry to be either changed to a nil, or renamed. Therefore the dentry cache is a complex and confusing part of the kernel source code!

File Data Cache

We'll see in Unit #5 that areas of memory and areas of regular files have an equivalency. Prior to that revelation, let us just say that when read and write system calls are made, that causes the corresponding areas of the file to be cached as areas of memory (it turns out these areas are always in 4K chunks, for reasons that will become apparent in Unit 5). Generally speaking, this improves the efficiency of small reads and writes. Rather than having to make say four disk requests to read 4K of data with 1024 buffers in the read system call, just one disk request of 4K is needed. The first read system call takes a little while, but the next three are satisfied from the cached copy in memory, and are much faster.

When writing, it is important to realize that the write system call does not generally result in an immediate write to the disk. Instead, the bytes that are supplied to the write system call are copied into the in-memory image of that area of the file. The write back to the disk happens at a later time. This is now discussed further.

Cache modes & sync

In general, there are three modes to caching: **uncached** (or "raw") in which all object accesses result in direct and immediate disk accesses, **write-through** where reads are satisfied from cache but writes cause immediate write-through to disk, and **write-back** where the write to disk takes place at some indeterminate later time. Raw mode would generally not be used except for system utility programs. Write-through mode has the advantage of keeping the disk in a consistent state and minimizes the risk of data loss or filesystem corruption. However it causes a large amount of disk write activity. Therefore, most systems are run in write-back mode.

In write-back mode, a system call such as `write` will complete before the data are written to the disk. In fact, the `close` will complete too. The precious data are sitting someplace in kernel memory, awaiting write-back to the disk. If the system crashes (sudden power loss, kernel bug, hardware issue) at this point, the data will be lost, but more importantly, the program (and the end-user) will be under the false impression that the data were saved, because no system call errors were raised.

If this is not acceptable, there are a few solutions (better than turning to write-through caching). The `fsync` system call takes a file descriptor and commands the kernel to flush all data and metadata associated with that file descriptor to the disk. The system call blocks until the disk writes have completed. The file could also be opened with the `O_SYNC` flag, in which case all write system calls will block until the corresponding disk writes have completed. Finally, there is a `sync` system call and a corresponding user-level command which causes ALL cached data to be written back to disk.

Outside of these explicit flushes, the Linux kernel periodically scans memory for cached objects and flushes them to disk. The rate at which this is done depends on system load, but generally on an idle system, flushes will be done at least once a minute. A system that is quiet and crashes, therefore, is not likely to lose any data. In addition, when a volume is unmounted, all cached objects are flushed, and the `umount` system call blocks until these disk writes are complete. This is particularly important for removable storage devices -- it ensures that once you give the `umount` command and get your shell prompt back, it is safe to remove the device!

Filesystem corruption, recovery & journaling

The filesystem is a data structure built out of disk blocks. As with any complex data structure, there are transient moments when it can be in an inconsistent state. Consider how the kernel conceptually handles the creation of a new file:

Step 1) Search the inode allocation bitmap for a free inode, say we pick #9999
Step 2) Mark the inode as non-free by changing its type field to S_IFREG (and also set the uid, gid and ctime fields, and set nlink to 1)
Step 3) Mark the inode non-free in the inode bitmap
Step 4) Search the directory containing the new file for a free slot
Step 5) Write the path component name and inode number 9999 to the directory slot

Let us say that somewhere between steps 2 and 5, the system loses power and therefore the directory is not updated. When the system comes back up and the volume is mounted again, we now have a "phantom" inode which is not free but is also not linked anywhere in the filesystem. This is just one example of many kinds of corruption which can arise from the underlying problem that filesystem operations reduce to a **non-atomic** series of writes to different disk blocks. The filesystem corruption problem is different from the cache consistency problem : filesystem corruption could occur even if there were no caching and all filesystem blocks were written through immediately.

When the system comes back up after a crash, there must be a way of determining if a filesystem (volume) is corrupt, and if so, to correct before mounting. Otherwise, the corruption could lead to further corruption and data loss. The first step is that all UNIX filesystems maintain a flag in their superblock known as the "clean" status. When the volume is dismounted properly, using the `umount` system call/command, or automatically as part of a controlled system shutdown, the superblock on disk is written to and the CLEAN status is set. While the volume is mounted, the status is set to DIRTY.

If the system crashes, the volumes will not be explicitly unmounted, and the status of the on-disk superblock will be DIRTY. The traditional way of correcting the corruption is via a user-level program known as `fsck` which examines the on-disk filesystem on a block-by-block basis. It can do this since the raw contents of the volume are available via a special file name, e.g. `/dev/sda1`. fsck is a complex, multi-pass program. It does a recursive descent exploration of the pathname tree, visiting each node, and cross-checking that view with the view represented by the inode table and free block map. If a consistency error is detected, fsck prompts before taking action to correct it. I.e. fsck is a read-only operation unless the user invoking it gives permission for writing. In batch mode, e.g. when the system is being booted, fsck is normally run in a mode where it will always attempt to fix problems, unless certain serious problems are detected, or the number of problems is abnormally high, at which point the booting process will stop pending system administrator interaction.

Often fsck will find inodes that might represent valid files, or might be invalid or deleted files. Because the filesystem is inconsistent, it has no way of knowing for sure. These inodes are landed in a pre-existing subdirectory of the root directory of the volume, called `lost+found`.

Filesystem Journal

fsck can be a very time-consuming process, since it needs to visit every part of the filesystem data structure, including parts that correspond to unallocated space, e.g. unused inodes. As hard disk capacities have increased, this has caused fsck times to be unacceptably long. Most modern UNIX systems have gone to journaling to address this.

A **journal** is a circular array of blocks somewhere in the filesystem, generally at a fixed location and generally contiguous. The journal contains **journal entries** which are ordered from oldest to newest. Since the journal is of finite size, eventually newer entries overwrite older ones. (Linux by default uses a journal size of 128MB) That may cause operations to hang until disk write flushes complete. While there are many different ways of implementing a journal, they all accomplish a similar goal: they make it possible to quickly recover a corrupted filesystem without having to use the very slow `fsck`. They do this at the expense of additional disk write traffic. Blocks that are critical enough to deserve journaling are written twice: once to the journal, and once again at a later time when the applicable cache is flushed.

Here is an example of hypothetical entries in a journal for our previous example of creating a new file:

```
--BEGIN TRANSACTION ID#1234--
Copy of disk block containing inode #9999
Copy of disk block containing entry 9999 in free inode bitmap
Copy of disk block containing new directory entry referring to ino9999
***** potentially other, unrelated transactions ******
--COMMIT TRANSACTION ID #1234--
```

The kernel, when journaling is in use, first assembles all of the elements of the atomic transaction and writes them to the journal. Then, only after the COMMIT TRANSACTION record has been written to the journal, the kernel schedules the various disk blocks to be flushed (sync'd) to disk. This might happen a "long time" later (several seconds) during which time additional journal entries might be written for other, unrelated transactions. Depending on kernel settings, the kernel might cause the system call (e.g. `creat`) to block until the COMMIT TRANSACTION record has been written. This ensures that when the system call returns, the operation will actually happen, even if the system crashes.

When the volume is mounted after a system crash and is found to be DIRTY, rather than invoking `fsck`, the journal is examined in order from oldest to newest entry. For each transaction ID number mentioned, there are two possible outcomes:

1) The BEGIN entry is there but no COMMIT. This means that the system crashed while the transaction record was being written to the journal. Since there is no COMMIT entry, we don't know if we have the entire transaction. No valid action can be taken. Since disk

flushes don't happen until the COMMIT entry is written, no partial transaction was written to the disk. In other words, this transaction never happened.

2) The BEGIN and COMMIT entries are both there. This transaction is then "replayed", i.e. the copies of blocks for the transaction that are in the journal are written to their places in the filesystem. It does not matter that they may have already been written. There is no harm in writing the same data again, other than a slight delay.

Recovery via the journal is very, very fast, typically less than one second!

Journal modes

In Linux, a particular mounted volume which supports journaling can be set to one of three modes (this is configured at each mount and is not a permanent attribute of the volume) which controls how the data itself (as opposed to metadata) are handled in the journal:

• `data=journal`: In this, the most paranoid mode, all data writes are written into the journal first, i.e. the data have the same level of protection as the metadata. This can have a very heavy toll on performance since all writes effectively require two disk accesses (one to the journal, the other to the actual data block) and because data write traffic is high-volume and may overflow the small journal area causing a stall until disk writes are completed. This mode guarantees that the order in which data are written to a file is consistent with the order in which files are created, renamed or deleted.

• `data=ordered`: Data are written back before corresponding metadata are written into the journal. The data blocks themselves are not written to the journal. This is the default mode. Consider a file that is being written with new data. As it grows, its size field in the inode has to increase, the block maps have to be updated, and the free map bits have to be marked as in-use. If these metadata are put into the journal first and then the system crashes, upon journal recovery the file would *appear* to contain the new data, but in fact the contents of those data blocks as described in the block map will be garbage. By forcing data write-back prior to metadata journal commit, this situation is avoided.

• `data=writeback`: This mode will have the best performance but the worst integrity. The write-back of data blocks is not synchronized at all with metadata writes to the journal. The situation described above can happen.

This discussion of journaling is necessarily a brief one. There are many complex issues which affect performance and reliability. Journaling is still an active area of research and development.

A review of filesystem-related system calls & library functions seen thus far

```
open:     Create an ordinary file, and/or open a file for I/O.

read:     Read data from an open file

write:    Write data to an open file

lseek:    Change the current position within an open file

close:    Close an open file

unlink:   Destroy a path to a file (or symlink)

mkdir:    Create a directory node

rmdir:    Destroy a directory node (must be empty)

link:     Clone a file, producing a hard link

symlink:  Create a soft (symbolic) link

readlink: Query the value of a symlink

stat:     Retrieve the metadata information about a path

fstat:    Retrieve the metadata information about an open file

lstat:    Retrieve the metadata information without following symlinks

chown:    Change the uid and gid ownership of a node

chmod:    Change the permissions mask of a node

utime:    Change the timestamps

opendir:  Pseudo-system call to open a directory

readdir:  Pseudo-system call to read the next directory entry

closedir: Close a directory opened by opendir
```

The UNIX Process

The UNIX process is a **virtual computer**, that is to say the combination of a virtual address space and a virtual processor (or task). The kernel provides system calls to create new processes, to destroy processes, and to change the program which is running within the process. The purpose of this unit is to make an introductory exploration of these mechanisms.

We will be looking at 3 important system calls which behave oddly, from the standpoint of conventional programming. These are fork, exit and exec. You call these functions once, but they return twice, never, or once but in a different program, respectively!

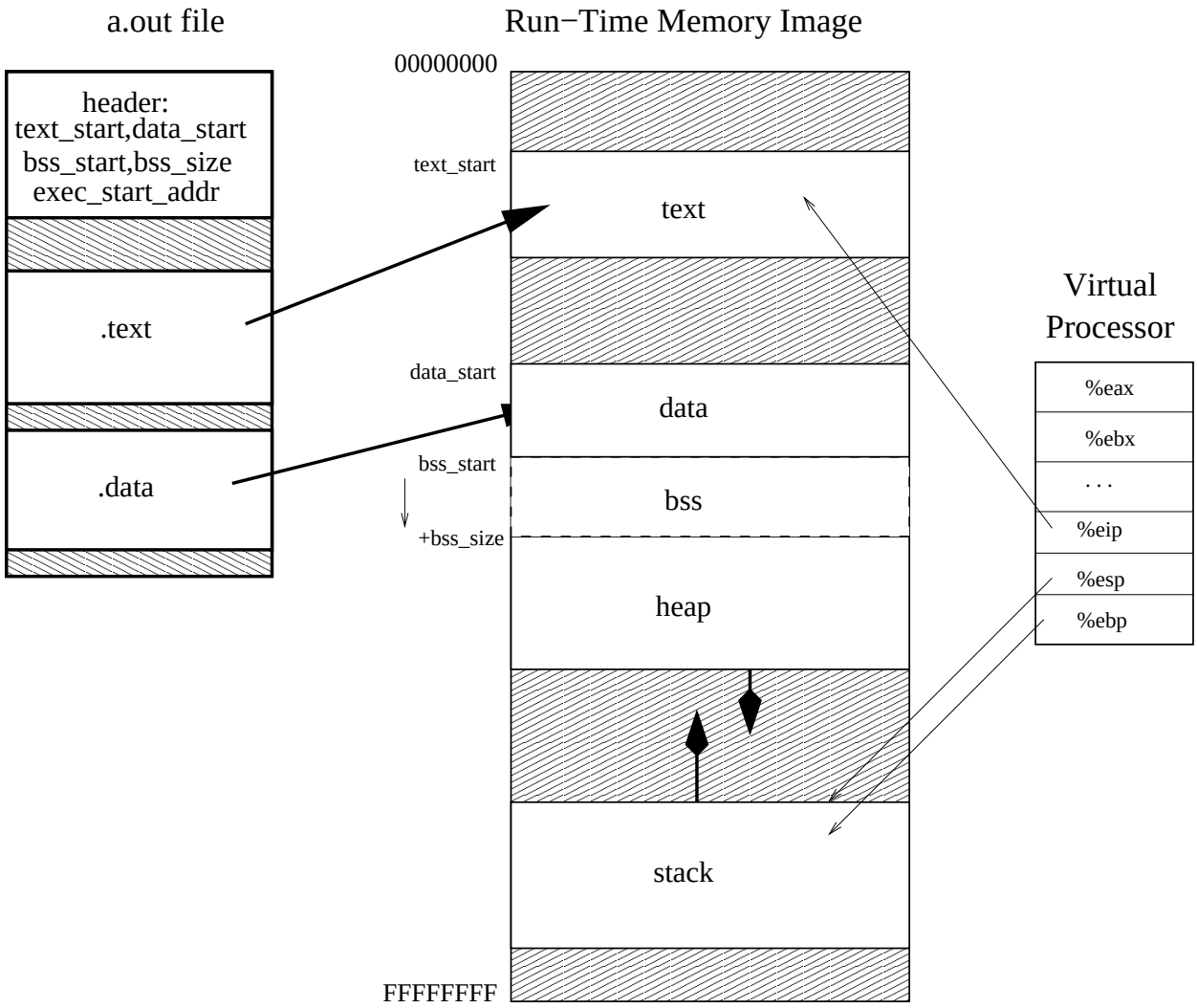
Processes are identified by an integer **Process ID (pid)**. All processes have a parent which caused their creation, and thus the collection of processes at any instant forms an ancestry tree. The pid of the current running process can be retrieved with the `getpid` system call, and the `getppid` system call returns the parent's process id.

There are interfaces to get the list of all running processes (pids) on the system. The `ps` command is commonly used for this. On Linux systems, it in turn uses the `/proc` pseudo-filesystem. There is an entry under this directory for each running process, e.g. `/proc/123` is a subdirectory which contains more information about pid #123.

Process #1 is always at the root of the tree, and is always running a specialized system utility program called `init`. `init` is started by the kernel after bootstrap, and it in turn spawns off additional processes which provide services and user interfaces to the computer.

The Virtual Address Space of a Process

All UNIX processes have a virtual address space which consists of a number of **regions** aka **virtual memory areas** aka segments (however the term segment should not be confused with hardware address segmentation as practiced on the x86 family of processors). For a given UNIX operating system variant and processor type, there is a typical virtual memory layout of a process. Recall that virtual addresses are meaningful within a given process only. Thus there is no conflict when the same virtual addresses are used in different processes.



For the purposes of simplicity, we will assume a 32-bit architecture, and therefore virtual address space ranges from 0 to 0xFFFFFFFF. Not all of this address space is populated. Traditionally, all UNIX systems use 4 regions: text, data, bss and stack.

• The `text` region is the executable code of the program. Other read-only data are sometimes placed in this region, such as string literals in the C language. The program counter register (%eip on X86-32 architecture) will generally be pointing into this region.

• The `data` region contains initialized global variables.

• The `bss` region contains uninitialized globals. Lacking an explicit initializer, these variables are implicitly set to 0 when the program starts. According to the original authors of UNIX, "bss" was the name of an assembly-language pseudo-opcode "block started by symbol", and was used to define an assembly symbol representing a variable or array of fixed size without an initializer. The bss region is grown by requesting more memory from the kernel, and this dynamically-allocated memory is often called "the

heap".
• The `stack` region is the function call stack of the running program. Function arguments (X86-32 only) and return addresses are pushed and popped on this stack. A different stack is used when the process is running in kernel mode, however that discussion will have to wait until a subsequent unit. The %esp and %ebp registers on X86-32 are pointing within the stack region.

There are additional memory regions which can be created as well, such as shared libraries, and memory-mapped files. In Unit #5, we will explore the properties of virtual memory in much greater detail.

Installing a new program with exec

The `exec` system call replaces the currently running program with a new one. It does not change the process ID, but it does conceptually delete the entire virtual address space of the process and replace it with a brand new one, into which the new program is loaded and executed.

We'll review the exec(2) system call very shortly. In order to load and execute a new program into an existing process, the UNIX kernel most be given the following:
• The pathname of the executable file
• A list of arguments (the familiar C-style argv[] array)
• A list of strings known as the **environment** which will be discussed below.

• *Conceptually*, the exec system call, after making a copy of the 3 vital pieces of information above into kernel memory, discards the entire virtual address space of the process as it currently exists. Again, conceptually, the kernel loads the executable image into (virtual) memory beginning at some specific absolute virtual address. The executable file, or `a.out`, contains:
• The loading virtual address and size of the text and data regions.
• The virtual address and initial size of the bss region.
• The entrypoint (virtual address of first opcode) of the program

The kernel creates the four basic regions (text, data, bss, stack) according to the information in the `a.out` file. The text and data regions are initialized by loading their image from the `a.out`. The bss region is initialized as all 0 bytes (meaning that any global variables lacking an explicit initializer are implicitly initialized to 0). An initial stack region is created (we will see in Unit #5 that it grows on demand) and a small portion of the stack, at the very highest address, is typically used to pass the environment variables and argument strings. The stack pointer and frame pointer registers are set to point to the correct place within the stack. The kernel establishes a stack frame as if the entrypoint function had been called with `(int argc, char *argv[], char`

```
*envp[])
```

If you examine the values of these pointers argv and envp, you'd find that they fall within the stack region. The kernel sets up the stack, starting from the highest address (because stacks grow towards low-numbered addresses), allocating space for the arguments and the environment. The kernel then sets the stack pointer (%esp on X86-32) register to the next free address and begins execution. Since the arguments and environment are below the stack frame for the startup function, they are "stable" and may be passed around freely throughout the program without fear that the associated memory may disappear or be used for something else.

After the memory regions are created and initialized, execution of the program begins when the kernel sets the program counter register to the start address which is contained in the a.out file, and then releases the virtual processor to begin executing instructions.

Although the traditional view is that execution of a C or C++ program begins with the main() function, in fact there are numerous hidden startup routines which execute first. These are provided by the standard library to initialize various modules of the library, such as the stdio subsystem.

During exec, some attributes of the process are retained for the next program, and others are reset. Of primary importance to this discussion is the fact that the virtual memory space is reset to a fresh state for the incoming program, while the set of open files, current directory, process id, parent process id, uid, and gid are retained across the exec boundary.

Exec system call

The `exec` system call replaces the currently running program with another program. There are actually several variants of the `exec` call, and under the Linux operating system, most are actually C library wrappers for the underlying system call, which is `execve`.

```
int execve (char  *path, char *argv [],char *envp[]);
int execv (char *path, char *argv[]);
int execvp (char *file, char *argv[]);
int execl (char *path, char *arg, ...);
int execlp (char *file, char *arg, ...);
int execle (char *path, char *arg , ...,char * envp[]);
```

The 'l' variants accept the argv vector of the new program in terms of a variable argument list, terminated by NULL. The 'v' variants, on the other hand, take a vector. Although it is convention that `argv[0]` is the name of the program being invoked, it is entirely possible for the caller to "lie" to the next program about `argv[0]`!

The first argument to any exec call is the name of the program to execute. The variants without ’p’ require a specific pathname (e.g. "/bin/ls"). The ’p’ variants will also accept an unqualified name ("ls") and will search the components of the colon-delimited environment variable `PATH` until an executable file with that name is found (this action is performed by the standard C library, not the kernel).

Exec errors

The invoking user must have execute permission for the executable file. This means not only that the file has execute permission set for the user, but also that all directory components in the path to that file are traversable (execute permission is granted). Read permission on the executable file (or intermediate directories) is not required for exec.

The executable file must be of the correct format to load on this operating system. This means that the binary processor architecture, addressing model, run-time model, and other issues must be compatible. I.e. the executable must have either been compiled on a similar system, or have been cross-compiled with the target system type in mind. E.g. a Windows .EXE file can not be run on a Linux system, even if both are 64-bit X86 processors, because the run-time environment is not compatible (but there are tools which interpose the correct environment and allow Windows programs to run under Linux, and vice-versa). A Linux a.out file compiled for an ARM processor is not going to run on an X86 processor. The kernel determines executable format compatability by examining the header of the a.out file.

There are several other errors which might cause the exec system call to fail, which are documented by `man 2 execve`.

If exec is successful, from the standpoint of the calling program, it appears never to return. On error, exec returns -1. The kernel does not get to the point of discarding the old address space until it has done enough checking to have reasonable assurance that the new executable can actually be loaded. Otherwise there would be no calling program to return -1 to!

Executing via an interpreter

The executable must either be a native binary (consisting of machine language instructions that can be executed by that system), or an interpreted script. In the latter case, the executable file will begin with:

```
#!/path/to/interpreter arg
```

`/path/to/interpreter` must be a qualified path (the PATH environment variable will not be searched) and must be a native binary executable file (not another interpreter). It will be executed with argv[0] set to "`interpreter`" (i.e. the last component of the interpreter pathname). If `arg`, which is optional, is present in the #! line, it will be inserted as the next argument (argv[1]). Then the entire argv vector of the invoked

program is appended to argv. This means that the name of the script file becomes argv[1] (argv[2] if the optional `arg` was specified in the #! line), and, in a break with tradition, it is the fully qualified pathname of the script file, rather than just the base name. This allows the interpreter to open this file and begin to interpret (execute) it.

For historical reasons, if the executable file has execute permissions, but is not a binary file, and does not contain an explicit #! interpreter invocation, it is interpreted with the shell /bin/sh, as if it had started with `#!/bin/sh`.

Linux and most other UNIX systems support **binary interpreters**. A special section of the `a.out` file directs the kernel to exec a specified interpreter, much like the `#!` mechanism above, but now the a.out file can remain a binary file instead of a line-by-line text file such as a shell, perl, awk, python, etc. script. The binary interpreter mechanism is heavily used: most commands are dynamically linked and the dynamic linker `ld.so` is in fact the program that completes the exec process. However, this detail is difficult to explain at this point until we have explored memory mappings in Unit #5.

The Environment

The environment is a set of strings of the form `variable=value` which is used to pass along information from one program to the next. The environment represents **opaque data** to the kernel, i.e. the kernel does not inspect or interpret its contents. There are UNIX conventions that environment variables have uppercase names, and certain names have certain functions. PATH contains the search patch for executables. PS1 contains the shell prompt string. TERM is the terminal type of the controlling terminal. HOME is the home directory of the current user. The shell command `env` displays the current environment variables and values. The shell command `export VARIABLE=value` creates a new environment variable.

The standard C library routines `getenv` and `putenv` can be used to query and create environment variable settings. The entire vector is also available as the global variable:

```
extern char **environ;
```

The 'e' variants of exec accept a vector, analogous to `argv[]`, specifying the **environment** of the new program. The non-'e' variants pass along the current environment.

The environment is established by the kernel prior to calling the program's start function, and has the same NULL-terminated array of strings format as argv. Storage for the environment and argument vectors is allocated by the kernel at the high end of the stack region.

Starting a new process with fork

While the `exec` system call replaces the currently running program with a new program, it does so inside the same virtual computer container (or process). The method which UNIX uses to create new processes is often confusing at first, because it creates a new process which is a copy of the current process at that moment, but does NOT change the running program. The `fork` system call is used to create a new process. The process which called `fork` is the **parent** process, and the new, **child** process is an **exact duplicate** of the parent process, including the entire virtual address space and the register set of the virtual cpu, with three exceptions:

• The child process will be assigned a new process id.
• The **parent process id (ppid)** of the child will be set to the pid of the parent.
• The `fork` system call will return 0 to the child process, and will return the child's process id to the parent.

Note that fork does not provide for a change in the currently running program. This results in the strange programmatic sensation of calling a function which returns **twice**. Another way to view this is that the child process comes to life executing at the exact point of returning from the fork system call.

The fork system call is fairly unique to UNIX. Most other operating systems provide a system call that combines fork with `exec` to both create a new process and associate it with a new program at the same time, i.e. a "spawn" system call. This would be useful because, as we'll see, the most common system call to follow fork is exec. We'll also see, in later units, how the UNIX kernel optimizes this.

```
int i;

f()
{
 int pid;
        i=10;
        switch (pid=fork())
        {
         case -1:
                perror("fork failed");exit(1);
                break;  /*NOTREACHED*/
         case 0:
                printf("In child\\n");
                i=1;
                break;
         default:
                printf("In parent, new pid is %d\n",pid);
                break;
        }
        printf("i==%d\\n",i);
}
```

If fork fails, then no child process has been created, and a value of -1 (which can never be a legal pid as pids are positive) is returned.

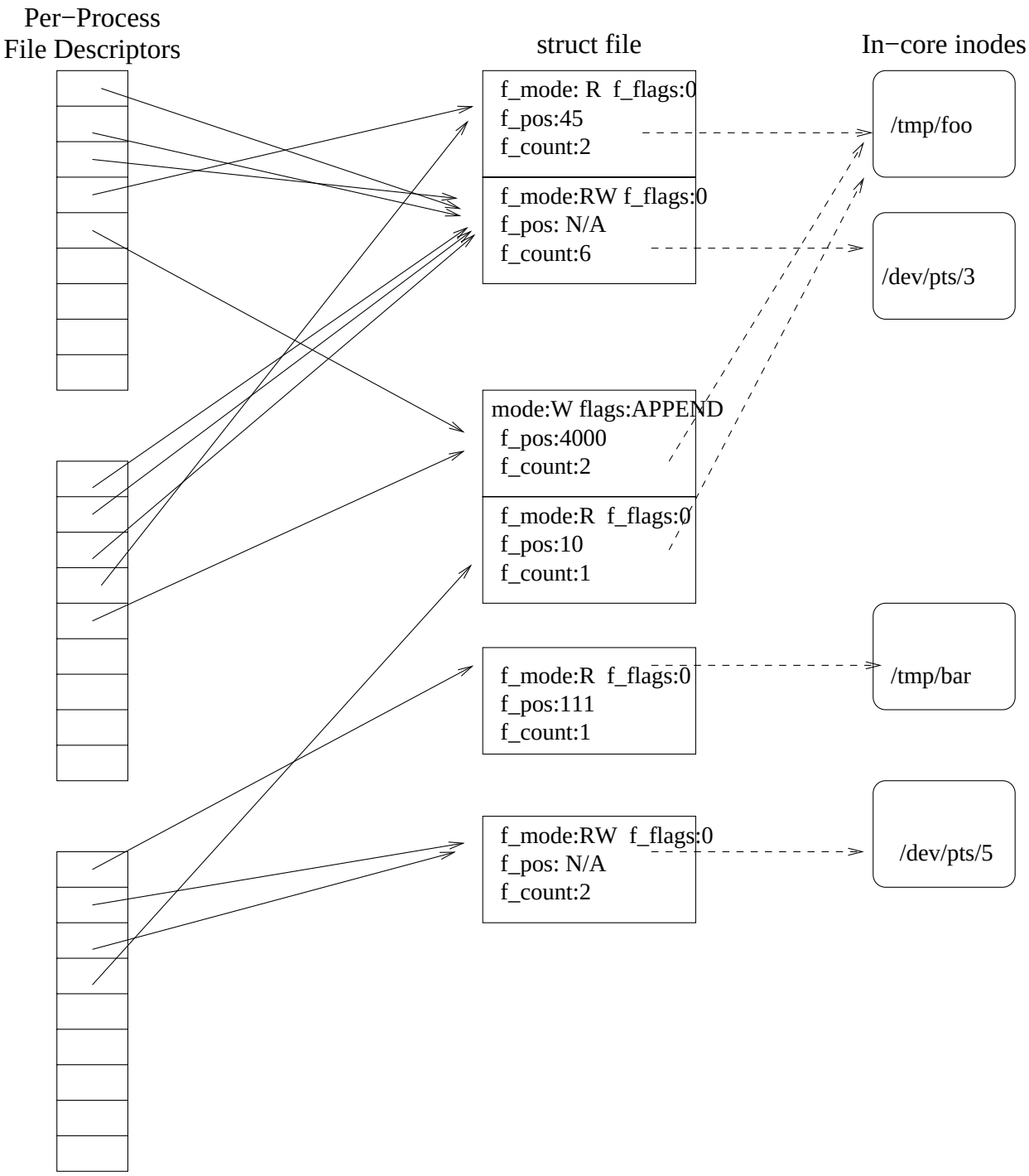
Although the child is an exact copy of the parent, it is nonetheless an independent entity

and has an independent virtual memory space which starts off as an exact copy of the parent's (again, we will see in a later unit how the kernel optimizes this and avoids actually copying physical memory until it is necessary). Therefore, in the example above, the child's modification of variable `i` does not affect the parent's copy.

Note that is **indeterminate** whether, after the fork system call completes, the parent runs first, the child runs first, or (on a multiprocessor system) both processes run simultaneously. Programmers should not assume any particular case. Nor should any assumption be made based on examining pid numbers that a particular process was forked before or after another process. While pids are generally assigned sequentially, there is a finite limit, whereupon the pid number "rolls over" and the next lowest unused number is selected.

The file table and file descriptors

There are in fact two layers of tables between the file descriptor numbers used by a process for I/O calls (such as open, read, write, etc.) and the actual inodes. Each process maintains a file descriptor table, the entries of which in turn point to kernel data structures which are called (in the Linux kernel) `struct file`.



`struct file` contains many fields. Right now, we are concerned with the following:

• `f_mode`: The mode under which the file was opened (RDONLY, WRONLY, RDWR).

• `f_flags`: The remainder of the second argument to the open system call. There are many esoteric flags, such as the ability to request non-blocking I/O (O_NOBLOCK). The only important one at this point in the course is `O_APPEND`, which causes all write requests to first seek to the current end of file.

• `f_count`: The reference count of how many entries in process file descriptor tables are pointing to this particular `struct file`.
• `f_pos`: The byte offset in the file where the last read or write left off.
• Through an intermediate data structure, the kernel can find an in-memory copy of the inode for the file, which is necessary for actually performing read or write operations.

The `f_pos` field maintains a **cursor** into the file. It is initialized to 0 when the file is first opened. Normally, a read or write system begins at byte offset `f_pos`, and then `f_pos` is incremented by the number of bytes read or written. Therefore, reads and writes appear to be sequential. `f_pos` can be queried or changed using the `lseek` system call. When the file has been opened with `O_APPEND`, all writes automatically begin at the current size of the file, i.e. all writes will append to the file and never over-write any part of it. After the append, `f_pos` contains the new size of the file.

The act of `open`ing a file creates both a new file descriptor and and new `struct file`. A fork makes an exact copy of the parent's file descriptor table, resulting in an additional reference to each file table entry (see dup below). This sharing of open files means that when e.g. the child process reads from a file, the parent process will see a change in the file position (e.g. through `lseek`).

A `close` on a file descriptor (assuming the file descriptor actually refers to a valid open file) NULLS out that file descriptor table entry **for the calling process only** and removes one active reference to the corresponding `struct file`. When the number of references falls to 0, the `struct file` itself is destroyed. That deletes one particular instance of having the inode open, but as illustrated above, there may be other `struct file` objects which reference the same inode and thus hold it open.

In the diagram above, one process, running on terminal /dev/pts/3, had apparently opened the file /tmp/foo twice, once O_RDONLY, and the second time O_RDWR|O_APPEND. This process forked, and so the top and middle per-process file descriptor tables are identical at this moment. Another process is running on /dev/pts/5. It has also opened /tmp/foo, O_RDONLY. We also see that its standard input has been redirected to the file /tmp/bar. This mechanism to do this will now be explained.

Dup and I/O redirection

The `dup` system call allocates a new file descriptor table entry for the process and points it to the same `struct file` as an existing file descriptor. The new file descriptor is **exactly equivalent** to the original one, as can be inferred by the diagram above. There are strong analogies here to `link`.

dup comes in two flavors: original `dup`, which picks a file descriptor for you (as usual, the lowest available fd number is chosen), or `dup2` which allows you to pick the new file descriptor number, which is first `close`d if already open.

The most frequent application of dup is to redirect standard input, standard output or standard error:

```
if ((fd=open(logfnm,O_CREAT|O_APPEND|O_WRONLY,0666))<0)
{
          fprintf(stderr,"Can't open log file %s",logfnm);
          perror("");
          return -1;
}
if (dup2(fd,2)<0) {
          perror("Can't dup2 logfile to stderr");
          return -1;
}
close(fd);
if (execlp("/usr/local/bin/nextprog","nextprog","arg1","arg2",NULL)<0)
{
          perror("Whoops, can't exec /usr/local/bin/nextprog!");
          return -1;
}
```

In this example, nextprog is invoked with stderr redirected to a log file whose name is contained in the char * variable logfnm. Note the close(fd). After the dup2 call, both file descriptor fd AND file descriptor 2 (standard error) point to the newly-opened file `logfnm`. It would be a "bad idea" to start the new program with an extra reference to this file.

fork and the file descriptor table

The effect of a fork is to create, in the child process, a file descriptor table which is an exact copy of the parent process. The reference counts in the `struct file` structures are incremented accordingly. In the diagram above, the top and middle processes have forked, and share all file descriptors. It is as if the `struct files` have been `dup`'d, except the referencing file descriptor table entries are in a different process.

Typical shell I/O redirection

The shell uses dup or dup2 to establish I/O redirection for spawned commands. To isolate possible errors from the main shell process, generally the `fork` is done first, and the opening of files and redirection of file descriptors is performed in the child process.

In the classic UNIX environment, the only way two processes can share an open file instance (`struct file`) is if they share a common ancestor which performed the open, and the referencing file descriptors were thus inherited through forks. In modern UNIX kernels, there are other mechanisms, beyond the scope of this lecture, which can violate this principle.

Expected file descriptor environment

It is a UNIX programming convention that, unless otherwise specified, a program expects to start life with just the 3 standard file descriptors open. This means that any output or errors which the program produces will go somewhere, and there is someplace from which to solicit input if needed.

To have extra file descriptors open when the program begins is generally an error, and may cause problems. These extra open file descriptors create, from the standpoint of the program, an unexpected connection to something else on the system, and from the standpoint of the system administrator, dangling and dead references which might prevent resources from being freed.

It is likewise an error if the standard 3 file descriptors are not open when a program starts, or are not open correctly (e.g. fd#1 is opened with O_RDONLY mode). This will cause unexpected errors when attempted to read/write to/from the standard descriptors.

Process termination

Processes terminate either when they call the `exit` system call or they receive certain types of **signals** (which will be covered in the next unit).

The `exit` system call takes a single integer argument, which is called the **return code**. By convention, a return code of 0 is used to flag the normal and successful conclusion of a program, anything else indicates an error or abnormal end ("ABEND" for any old mainframers out there). Equivalently, when the function `main()` returns, it is equivalent to calling `exit`, and the return value of main is used as the return code. Good programming practice calls for main to have an explicit `return` so that a consistent return code is generated, typically 0 since a normal return from main is usually a good sign.

Although it is commonly stated that C program execution begins with the function `main`, that is not entirely true. The **entrypoint** of a program is the virtual address at which execution begins, and is found in the executable file. When a program has been compiled with the standard C library, the entrypoint is a function called `__start`, which performs any required library initializations and then invokes `main`. When `main` returns, library cleanup is performed. In particular, note that stdio buffers are flushed here, so that even when a programmer has been sloppy and has allowed `main` to return without calling `fclose`, data are not lost.

```
_start(int argc,char **argv,char **envp)
{
int rc;
extern char **environ;
          /* perform initialization of stdio and other libs */
          environ=envp;
```

```
        rc=main(argc,argv,envp);
        exit(rc);

void exit(int rc)           /* The exit(3) library fn */
{
        /* execute atexit callbacks */
        /* close and flush all stdio streams */
        /* other library cleanup */
        _exit(rc);          /* The real exit system call */
}
```

On many UNIX systems, a mechanism called `atexit` is provided. Additional cleanup functions can be registered by calling atexit:

```
f_cleanup(void)
{
        fprintf(stderr,"I'm going away now\n");
}

main()
{
        ....
        atexit(f_cleanup);
        ...
}
```

The registered cleanup routines are called in reverse order of their registration.

`exit(3)`, which is a standard library function. There is a system call which is the "real" exit, `exit(2)`. This system call forces the **immediate** termination of the process. In contrast, the library function `exit(3)` first executes all of the cleanup and atexit routines, then calls _exit. The result is that both calling `exit` or returning from `main` have identical semantics.

A process can also be terminated when it receives a **signal**. A signal is the virtual computer equivalent of an interrupt. It can be sent from another process, or can be raised against the process by the operating system because the process performed an illegal operation, attempted to access a bad memory location, or for various other reasons. Signals do not always result in termination. Some signals may be ignored, deferred, or handled. Signals will be covered in depth in subsequent units.

Processes that die because of a signal will not have a chance to run the standard library exit functions, therefore stdio buffers will not be flushed, etc. This is one of the reasons why `stderr` is, by default, unbuffered. In the event that the process is killed, it is beneficial to see all of the error messages leading up to that point.

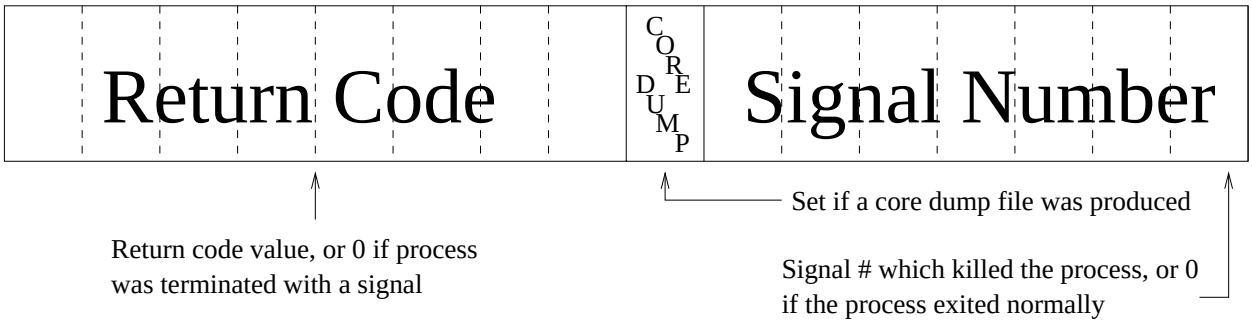
Regardless of the termination reason, when a process terminates, all file descriptors are closed by the kernel as if `close` had been called on them. All resources used by the process (such as memory) are freed (unless they are also being shared by other extant processes). Other state information (such as locks held by the process) is also adjusted. This means that explicit `close` system calls are not needed before a process exits

(although you might want to do this to catch any errors associated with the file descriptor), nor is it necessary to explicitly `free` memory that was allocated with `malloc`.

The exiting process becomes a **zombie**, consuming no resources, but still possessing a process id. The function of the zombie is to hold the statistics about the life of the process.

If the exiting process has any surviving children, they become orphans. Their parent process id (ppid) is reset to 1. This, you may recall, is the process id of the init process, which inherits all orphaned processes on the system.

Typically, the parent process claims its zombie child by executing the `wait` system call. The exit status of the process will be packed into a 16-bit integer. It will indicate either that the process terminated by calling exit, and will supply the return code (truncated to 8 bits), or that the process terminated from a signal. There are macros to decode this status word, for example:



```
#include <sys/wait.h>
#include <wstat.h>

pid_t cpid;
unsigned status;

if ((cpid=wait(&status))== -1)
{
          perror("wait failed");
}
else
{
          fprintf(stderr,"Process %d ",cpid);
          if (status!=0)
          {
                    if (WIFSIGNALED(status))
                    {
                              fprintf(stderr,"Exited with signal %d\n",
                                        WTERMSIG(status));
                         else
                                  fprintf(stderr,"Exited with nz return val %d\n",
                                           WEXITSTATUS(status));
                         return -1;
                }
          }
          else
                    fprintf(stderr,"Exited normally\n");
}
```

Another form of wait is `wait3` which can be used to obtain the resource usage information for the child process:

```
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

struct rusage ru;
int cpid;
unsigned status;
          if (wait3(&status,0,&ru)== -1)
          {
                    perror("wait3");
          }
          else
          {
                    fprintf(stderr,"Child process %d consumed
                              %ld.%.6d seconds of user time\n",
                              pid,
                              ru.ru_utime.tv_sec,
                              ru.ru_utime.tv_usec);
          }
}
```

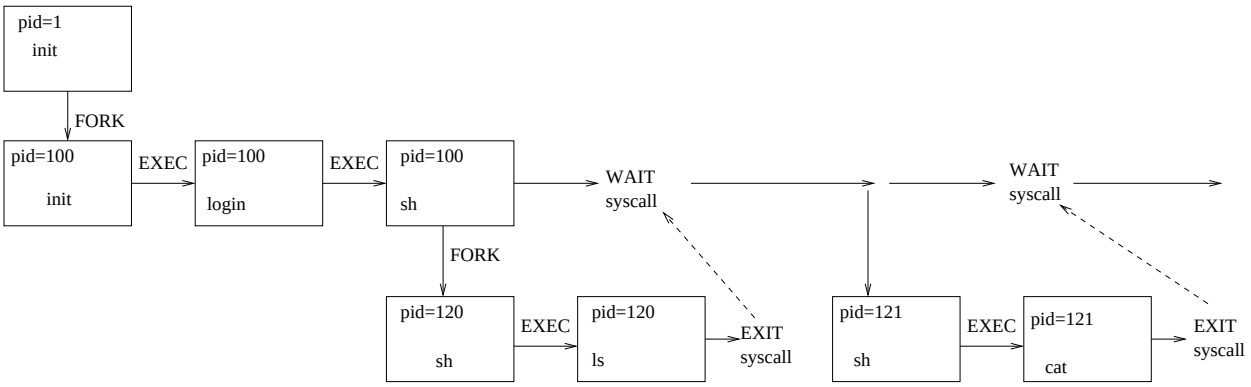
Among the resource usage information kept for each process is the total **user CPU time**

and **system CPU time**. User time is time accumulated executing user-level code. I.e. the total amount of time that the virtual processor (the process) had use of a physical CPU, in user mode. Likewise, system time is the time accumulated executing kernel code on behalf of the process. The sum of user+system time is the total amount of CPU time that the process consumed during its lifetime. This will always be less than the **real time** elapsed between process start and process termination, since the physical processor (or processors) is shared among numerous virtual processors, as well as system overhead functions.

There are additional calls such as `waitpid` which will not return until a specific child process has exited (as opposed to `wait` which will return when any child has exited), and `wait4` which is like `wait3` with the semantics of `waitpid`. More detail can be found in the man pages.

A parent process that does not perform a wait to pick up its zombie children will cause the system process table to become cluttered with a lot of <zombie> processes. (There is a way around this which will be mentioned in conjunction with the `SIGCHLD` signal in Unit #4) If a parent exits before the child, then who will collect the zombie status? The answer is the `init` process, which becomes the parent of any orphaned process.

Typical fork/exec flow cycle



When the system is first booted, there is only one user-level process, which is known as `init` and has pid of 1. `init` is responsible for the user-level initialization of the system, starting the user interface, starting system services, etc. In the above extremely simplified view, `init` has spawned (by fork and exec) a process which listens on a login terminal (e.g. one of the virtual consoles on Linux). This program, `login`, accepts the user name and password, verifies the credentials, and then execs itself into a command-line shell. The default shell is `/bin/sh`.

Typically, the shell receives a command as a line of text, parses it, and forks and execs the command so it runs in a new process. Unless the command is followed by the `&` symbol,

it runs in the *foreground* and the shell waits for the child process to exit. It collects the exit status (via one of the wait system call variants above) and then accepts the next command. One can view the exit status of the last command through the shell variable `$?`, e.g.

```
$ ls -foobargument
ls: invalid option -- e
Try 'ls --help' for more information.

$ echo $?
1
```

Background processes

If one invokes `command &` from the shell prompt, a new process is forked by the shell and execs command, but the shell does NOT wait around for child process completion. It instead issues a command prompt and executes the next command while the first command also runs. The first command is then said to be a "background process". There are some complications: what happens if the child process wants to read from standard input? It would be "competing" for characters with the shell itself, and/or with subsequent commands. We can't really explore this topic further without understanding signals and the `tty` layer, so the interested reader could consult some online tutorials on using job control and background processes in UNIX.

The task_struct

As we will discuss further in Unit #5, the kernel executes in one big shared virtual address space (whereas user-mode processes are contained in distinct VA spaces). This allows the kernel to create data structures and use pointers without worrying about which address space they are part of. The kernel maintains information about each process in kernel memory.

In the Linux kernel, a `struct task_struct` is allocated for each process (to be precise, it is allocated for each schedulable task, which equates to each thread in the case multi-threaded programs). This is a fairly large structure and contains either directly, or indirectly through other structures pointed at, just about everything one would ever want to know about a process. Some examples of the process state maintained via the task_struct:

• Relationships with other processes: parent pid, list of children, list of siblings, lists of process and session groups.
• Credentials: uid, primary gid, list of gids that we are a member of, effective uid and gid when executing setuid or setgid programs, etc.
• Open file descriptor table (a lot more about that later this unit)
• Resource usage counters: accumulated user and system cpu time, memory usage, I/O

usage, etc.
• Process address space layout (lots more about this in Unit 5)
• Current working directory
• Currently executing program, command-line arguments

There is a global variable called `current` which the kernel maintains as a point to the task_struct of the currently running process. On a multi-CPU machine, each CPU has its own sense of `current`. [In fact, this global variable is implemented as a macro which accesses the per-cpu private data area in kernel memory]. As an example:

```
/* Extremely simplified overview of struct task_struct */
/* For the purposes of illustrating fork/clone.  Missing a lot of stuff */
/* From /usr/include/linux/sched.h */
struct task_struct {
          struct list_head tasks;               // Part of linked list of all tasks
          struct mm_struct *mm;                 // Virtual memory layout
          int exit_code,exit_signal;   // exit status
          pid_t pid;                            // what gettid returns
          pid_t tgid;                           // what getpid returns
          struct task_struct *parent;  // who's your daddy?
          struct list_head children;   // linked list of child tasks
          struct list_head sibling;    // Part of linked list of siblings
          struct list_head thread_group;        // Part of ll of threads in thread grp
          cputime_t utime, stime;               // accumulated user/sys CPU time
          u64 start_time;                       // when this task was spawned
          struct cred *cred;
          char comm[TASK_COMM_LEN];    // Pathname of current executable
          struct fs_struct *fs;                 // cwd, umask, etc.
          struct files_struct *files;  // file descriptor table
          struct signal_struct *signal;         // Wait till unit #4!!
          struct sighand_struct *sighand;       // Lots more signal stuff elided
};

struct task_struct *current_pointers[MAXCPU];   // Simplified
#define current (current_pointers[my_cpuid])    // Simplified

/********** (Simplified) Example of using the current pointer *******/
int sys_getpid()                        // getpid system call
{
          return current->tgid;
}

int sys_getuid()
{
          return current->cred->uid;
}
/*****************************************************************/
```

Multi-threaded program / clone system call

All modern UNIX systems support multi-threaded processes. Our definition of **multi-threaded** shall be: a process in which two or more independent, schedulable threads of control co-exist within a shared address space. The POSIX standard, which governs compatibility issues among UNIX variants, says that in a multi-threaded program, all threads share the same pid, because they are, after all, part of the same process. The `gettid` system call is defined to return a unique integer for each thread within a multi-threaded process.

Different UNIX variants (e.g. Linux, BSD, Solaris) have different ways of making a multi-threaded program. We will look at the Linux approach, in which a system call `clone` is defined:

```
int clone( int (*start_fn)(void *), void *child_stack, int flags, void *arg)
```

The clone system call is like fork, except all of the things that fork "copies" for the child process are now allowed to be specified piecemeal. The `flags` argument is a bitwise combination of flags specifying this behavior. For example, `CLONE_VM|CLONE_FILES` means that parent and child will forever *share* the address space and the file descriptor table. Therefore if the child thread opens a file and stores the file descriptor in a global variable, the parent thread can reference that same global variable and can use that same file descriptor. `fork` then becomes equivalent to `clone` where the `flags` are set to 0 : all aspects of the parent are copied to the child but then become independent for fork. In fact, the Linux kernel implements fork and clone as the same system call.

{Aside: the name `CLONE_XX` has an inverted sense. It would have perhaps been better to call the flags `SHARE_XX` since When such a flag is set, the corresponding data structure is shared rather than copied (cloned). Alas, this is how Linux named it}

If we look at the `struct task_struct`, this is implemented by having each of these things that can be shared vs. copied tracked via a struct which is pointed to by the main task_struct. If the corresponding bitwise flag is 0 (e.g CLONE_FILES), then a new sub-structure is allocated and copied from the parent's, and the child points to the new sub-structure. This is known in data structures theory as a "deep copy". If the flag is 1, the child simply points to the same sub-structure, which is a "shallow copy".

For multiple threads to work correctly in a shared address space, there must be independent stacks for each. Otherwise, function calls and local variables would interfere with each other! The `child_stack` parameter give the address of a new memory region (see Unit 5) that the parent has created for the child's stack. Unlike fork, execution of the child thread begins not at the next line of code after the clone system call, but by calling (*start_fn)(arg); When this function returns, the child thread dies, and

the return value of the function becomes the exit code of the thread, much like the return value of main in a conventional single-threaded program.

To further confuse things, the `clone` that you see is really a library wrapper function. The real, underlying `sys_clone` system call works more like fork. Moreover, most applications programmers use additional wrapper libraries to do multi-threaded programming. The most common library is the POSIX Threads (pthreads), with functions such as `pthread_create` to make a new thread.

Yet another area of confusion will be in reading kernel source code. Because of the way Linux historically approached multi-threading, within the kernel, each thread is associated with a unique `task_struct` and a unique `pid`. The kernel uses the term "thread group ID" (`current->tgid`) as an identifier for a collection of threads as typically found in a multi-threaded program. To comply with POSIX, the `getpid` system call actually returns the `current->tgid`, and the `gettid` system call returns `current->pid`.

Programs which are multi-threaded are much harder to debug. However, a great many applications are well-suited to the thread paradigm. These include server applications (e.g. web and email service) and programs which present a graphical user interface. That's about all we'll say about multi-threaded programming for now. The interested reader is referred to the `man 2 clone` and `man 7 pthreads`

Preemptive and Cooperative Multitasking

We have seen how the kernel uses time-slicing to provide an illusion: that a single processor is actually running multiple simultaneous tasks. On a multiprocessor system, the illusion is that the total number of tasks can exceed the number of physical processors.

Simple operating systems use **cooperative multitasking**. A task which is currently running on the processor voluntarily gives up the processor and allows another task to run.

Most operating systems designed for general multiuser use recognize that tasks can't be trusted to relinquish the processor. **Pre-emptive multitasking** means that a currently running task can be forcefully suspended and a context switch made to another, presumably ''better'' task to run.

The trigger for preemption may be based on priority, i.e. a higher-priority task has just become ready to run. It may also be based on time-slicing, in which each task is given a certain amount of time and then another task is given the CPU. This requires a **Periodic Interval Timer** interrupt. Or, it the trigger could be a blend of priority and time. In the Linux kernel, it is the `scheduler` subsystem which makes the decision as to which task

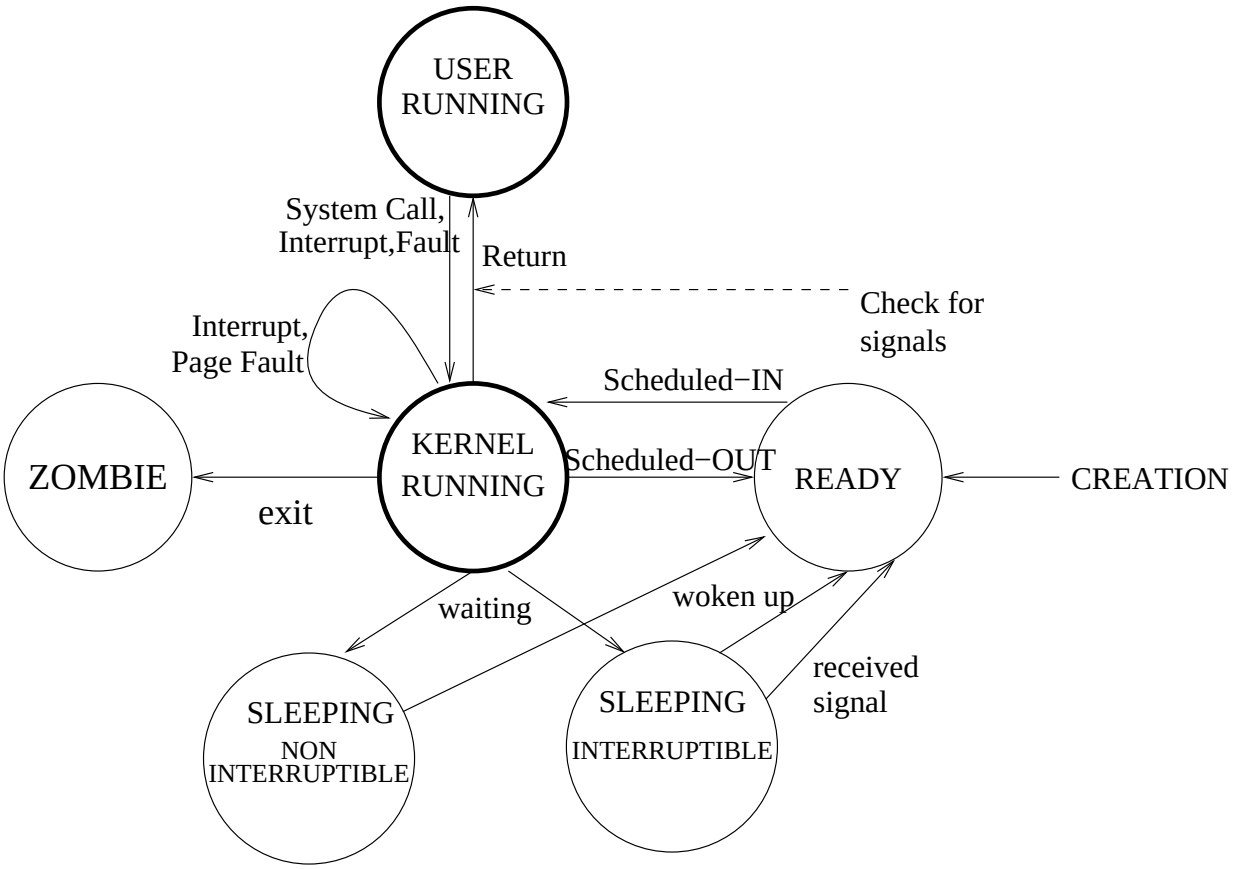
to run and when.

In a future unit, we will see how the kernel actually implements a task switch. (For our purposes, a "task" is the same as a thread, and in a single-threaded process, is the same as a process). Let us, for now, look at the basic concepts.

Process/task states

Universal among multitasking operating systems is the notion of task **states**.

A simple model of task states is presented below. The state names chosen are generalized, and not necessarily the names used in any particular UNIX kernel. We are also ignoring certain subtleties, such as job control and process tracing.



The state diagram depicts the task state transitions, from the perspective of the task.

USER RUNNING: The task currently has a CPU and is running a user program.

KERNEL RUNNING: The task is being executed in kernel mode, as the result of system call, fault or (asynchronous) interrupt.

READY: The task is ready to run, but does not currently have a CPU. A READY task is found on a run queue.

SLEEPING: The task is awaiting some event and is therefore not runnable. A task in the SLEEPING state has been temporarily suspended. For example, if a read system call has been performed to read the next character from the tty, the task must sleep until the character actually arrives. A sleeping task will never be scheduled until it is woken up. A sleeping task is always in kernel mode, and got there synchronously, via a system call or fault handler. Interrupt (asynchronous) handlers can never cause sleeping.

We may further distinguish between INTERRUPTIBLE and NON-INTERRUPTIBLE sleeps. This decision is made when the task puts itself to sleep. INTERRUPTIBLE sleeps will be terminated if a signal arrives (e.g. Control-C from the terminal), whereas in a NON-INTERRUPTIBLE sleep delivery of the signal will be deferred. Signals will be defined in Unit #4.

ZOMBIE: A task which has terminated but whose statistics have not yet been claimed by the parent. Once the zombie status has been claimed by the parent through the wait family of system calls, the task no longer exists.

Task Scheduling and Fairness

Under most circumstances, time spent executing on a CPU is a scarce resource for which tasks compete. This makes the **scheduler** part of the kernel an interesting problem that is often studied in OS research. Scheduling algorithms have varied widely from OS to OS, and even from one version to another. We can generalize a few broad principles:

• Tasks tend to be either **compute-bound** or **I/O-bound**. The former spend most of their time computing and thus have a heavy appetite for user-mode CPU time. The latter spend most of their time waiting for I/O. These classical definitions are often stressed by media applications, e.g. a streaming video server which is both I/O bound with network traffic and compute-bound with compression and decompression algorithms. Compute time among CPU-bound processes should be fairly distributed so that jobs complete in a reasonable time. Note that a given task may change its nature, e.g. a process such as Matlab which spends most of its time waiting for user input (I/O bound) but then has bursts of high CPU demand when it calculates results.

• Each task has an "importance", or **static priority**, which can be configured directly or indirectly by the system administrator to allow a task to receive a larger or smaller share of CPU time.

• The scheduler should allocate CPU time "fairly". Tasks that are at the same static priority level should, over a long sample period, receive approximately the same amount of CPU time. Tasks at different static priority levels should receive proportional amounts of CPU time.

• Tasks should appear as responsive as possible to interactive events. E.g. when a key is pressed or the mouse is moved, the application should respond quickly.

• The scheduler system itself should have a low overhead. The context switch is not a good time to be executing complicated, long-winded algorithms.

The Context Switch

A **Context Switch** is when one task is replaced by another on a given CPU. We will cover the mechanics in later units. At this stage, it is important to understand when a context switch happens:

• When the current task enters a non-READY state, e.g. because it makes a blocking system call.

• When the current task voluntarily yields the CPU, e.g. through the `sched_yield` system call.

• When the scheduler subsystem of the kernel decides that the current task's turn is done, and it is time for another task to have the CPU. This is known as **pre-emption**. In order to this to happen, the kernel must be in control, i.e. pre-emption only takes place when the task is in the KERNEL_RUNNING state. As the state diagram above depicts, this transition happens during a system call, fault or interrupt.

The Clock Tick

Regarding the last point, one specific interrupt is of great importance to scheduling. It is the **Periodic Interval Timer** or **'tick'** interrupt which arrives at a given frequency (every ms is typical). This gives the kernel the opportunity to pre-empt the task and give another task a chance to run, even if the current task does not make a system call or incur a fault and there is no other hardware interrupt activity on the system. The tick interrupt is also the timebase for the system time-of-day, things with timeout values such as the `alarm` system call and network protocols, etc.

When the clock tick interrupt arrives and control re-enters the kernel, the interrupt routine is able to determine if control came from user mode or kernel mode. It then "charges" a tick against the appropriate resource usage counter. In pseudo-code:

```
/* Make-believe kernel code*.
clock_intr_handler()
{
          if (came_from_user_mode)
                    current->utime++;  // See above for meaning of current
          else
                    current->stime++;
          clock_ticker++;              /* Increment global monotonic counter of ticks*/
          time_subsystem_tick();       /* Update time of day, run timeout
                                                callbacks, etc. */
          scheduler_tick();  /* Possibly switch tasks */
          /* Returns from interrupt */
}
```

UNIX Static Priority Model

Historically, the static priorities in UNIX were represented by so-called `nice values`, ranging from -19 to +20, with a default value of 0. Positive "nice" values give a task *poorer* static priority, i.e. they make it "nicer" to other, competing tasks with average (0) nice value. Classically, any task can increment its nice value, using the `nice` system call or command, but only a task running as the superuser (uid==0) can decrement the nice value and give itself "better" static priority. (In modern UNIX kernels, the privilege of giving oneself negative nice values is more fine-grained and can be assigned directly, without the process having to be the all-powerful, uid 0 superuser.)

Inside the kernel, other numbers may be used to represent priority, and they may have an entirely different interpretation from the traditional nice values. To maintain compatibility with POSIX standards, all kernels translate their internal priority number to the traditional -19..+20 nice value.

Static vs Dynamic Priority

To complicate matters, Linux uses the terms static and dynamic priority in a manner that differs from most authors on operating systems.

Traditional Definitions

• Dynamic priority: a relative value that fluctuates depending on what the task has done recently, and is consulted by the scheduler at each scheduling decision point (the "tick")
• Static priority: a fixed value that is established for the task and only changes by means of a system call. The nice value is an example of the traditional definition of static priorty. Note that the static priority influences the value of dynamic priority.

Linux Kernel Definitions

• Dynamic priority: the nice value, or other scheduling parameter that can be tuned.
• Static priority: one of 100 levels from 0 to 99. A ready task with higher static priority always runs before a task with lower priority. See the section on real-time scheduling.

The Quantum

A term used in the operating systems field regarding scheduling is **quantum** or "**time-slice**". This is the amount of time that a CPU-bound task runs before being pre-empted. Scheduling algorithms vary regarding their assignment of quantum. Some algorithms have an entirely fixed quantum, others a completely variable number, and others some intermediate solution (e.g. earlier Linux kernel schedulers used a variable quantum which was computed only when the task is scheduled in. Current kernels effectively re-compute the quantum at every tick)

Multi-processor systems and run queues

On a single-processor system, the list of all tasks which are READY to run is known as **the run queue**. It is not really a queue in the FIFO sense, because a task with better static priority can "jump the line" and get scheduled sooner than a poorer priority task that waits longer. So it is more of a rude queue with tasks cutting the line.

On a multi-processor system, there is one "run queue" for each processor. Typically, when a task becomes READY to run, the kernel decides which processor has the least workload and puts the task on that CPU's run queue. Thereafter, as long as the task remains READY, it typically remains on that CPU. This is because the cache for that CPU would still be "warm" with respect to the instructions and data that the task recently accessed. Under some circumstances, tasks can be migrated to a different CPU if loads get severely out of balance.

Normal Interactive Process Scheduling in Linux / CFS

The scheduling algorithm used for "normal" tasks by the Linux kernel is known as the Completely Fair Scheduler (CFS). It aims to satisfy the principles set forth above. The CFS algorithm attempts to provide "ideal latency" to all CPU-bound processes. Latency is defined as how much time elapses from when a task is pre-empted to when it gets the CPU again.

As a practical matter, the latency has a lower bound, because otherwise the system would spend most of its time in task switches, instead of doing useful work. Let us call P the latency period, and let us say this tunable value has been set to 10ms. If there are N=2 runnable tasks of equal static priority, each could run for 5ms and this would satisfy that the latency period, P, that each tasks sees should be 10mS. However, as N grows, this would imply smaller and smaller time slices, and eventually the overhead of scheduling and context switching will become prohibitively high. Another tunable parameter, G, is the scheduler granularity, the minimum time slice that tasks could have. If P/N<G, then P is capped at G. By default, in the Linux kernel, P is 5ms and G is 1mS, so if there are more than 5 runnable tasks (per CPU) then the latency period gets capped at 1ms. Since the clock tick is almost always 1ms, having a time slice of <1ms isn't possible anyway.

Weighted timeslice

The "nice" value, under the Linux CFS scheduler, is a process scheduling "weight". There are 39 nice steps (-20 to +19). Each nice step represents a 10% relative difference in CPU allocation. (This is a purely Linux interpretation of nice values -- other operating systems may have very different policies). The table below converts nice values into the weights, represented by the capital letter W:

```
/* nice -20 */ 86.6807,  70.0732,  55.1592,  45.1885,  35.4404
/* nice -15 */ 28.4707,  22.7090,  18.2666,  14.5986,  11.6367
/* nice -10 */ 9.3242,   7.4414,   5.9570,   4.7891,   3.8145
/* nice -5  */ 3.0479,   2.4424,   1.9443,   1.5488,   1.2471
/* nice +0  */ 1.0000,   0.8008,   0.6396,   0.5137,   0.4131
/* nice +5  */ 0.3271,   0.2656,   0.2100,   0.1680,   0.1338
/* nice +10 */ 0.1074,   0.0850,   0.0684,   0.0547,   0.0439
/* nice +15 */ 0.0352,   0.0283,   0.0225,   0.0176,   0.0146
```

Let's say process A has a nice value of 0 and process B has a nice value of 1, and these are the only two runnable processes. Then the weights (rounded off) are WA=1.00, and WB=0.8. We define the "load weight" LW as the sum of the weights of all runnable tasks. In this case, LW=1.800.

Next we define the CPU share for any task, w*n*, as w*n*=W*n*/LW. For A this is 1.00/1.800=55.5% and for B = 0.800/1.800 = 44.5%. Thus we see that the difference between two tasks separated by one nice level is approximately 10%. This formula is logarithmic, with each step in the table above being a multiplier of 1.25 relative to the next step. We see that if A had a nice value of -20 and B +19, A would get 99.98% of the CPU and B would get just 0.02%.

Under this weighted model, the perfect timeslice for a given task would be s*n*=P*w*n. Let us say P=10ms. A would have a weighted timeslice of 5.55msec, and B would get 4.45msec. Together they consume 10mS, the desired latency period.

This algorithm extends trivially to any number of runnable processes, and insures that the targeted weighted timeslice of any given process is its "fair share" of the available CPU, considering all of the other runnable processes and their weights. In actual kernel implementation, the weights are coded as integers, rather than the floating-point numbers used above, because the kernel avoids the use of the floating point registers.

Virtual Runtime

Under the CFS scheduler, the figure of merit when comparing runnable processes for scheduling is unfortunately called *virtual runtime*, or vruntime. Like many things in the Linux kernel, this is poorly named, and perhaps would better be called the "weighted actual runtime share".

The idealized allocation of timeslice presented in the previous section can not be realized in practice because 1) pre-emption only happens during a scheduler tick, which has a granularity of (typically) 1 msec, and 2) while a task is running, other tasks awaken, changing the weighted load.

At every scheduler tick, the ideal timeslice for the currently running process is recomputed. If the process has now been on the CPU for longer than that timeslice (subject to rounding to the nearest clock tick) then it is a candidate for re-scheduling. Re-

scheduling can also occur when the system load changes (because processes go to sleep or wake up, or because nice values are changed).

The vruntime of the running process (`current`) is updated: vruntime+= T/w, where T is the amount of time elapsed since the last time the load was examined (typically since the last clock tick) and w is the relative weight of the process (Wn/LW). The higher the relative weight, the less vruntime will be "charged" to the current process. The vruntime is cumulative for the life of the process.

Therefore, the lower the vruntime of a process, the greater is its relative merit for being scheduled now. The process with the lowest vruntime of all the runnable proceses is the one which should be on the CPU.

The CFS scheduler maintains a data structure (it is implemented as a red-black binary tree with caching of the lowest element) to keep all of the runnable tasks in order by vruntime. Retrieval of the "next best" task to run is therefore constant time. When the scheduler considers re-scheduling, it examines the tree for the task with the lowest vruntime. If the currently running task happens to be the lowest, then nothing happens, otherwise a task switch takes place and a new task gets the CPU.

```
/* Vastly simplified pseudo-code presentation of actual Linux kernel code */
scheduler_tick()
{
          /* Assume CLOCK_PERIOD is 1ms */
          current->vruntime += CLOCK_PERIOD / current->w;
          ideal_timeslice = P * current->w;
          if (clock_ticker - current->scheduled_in_ticktime  > ideal_timeslice)
          {
                    t = find_best_runnable_task();
                    if (t->vruntime < current->vruntime)
                              context_switch-to(t);
          }
          /* At this line of code, current is the next task to run,   *
           * which may be the same task as before or may be a new one */
          current->scheduled_in_ticktime = clock_ticker;
}
```

When examined in fine-grain detail, the actual timeslices of tasks will not match their "ideal" computed value. But on the average over a longer period of time, the selection of the next task to run based on lowest vruntime will result in an equitable distribution of CPU time which is approximately equal to that which would have been obtained if it were possible to implement the ideal weighted timeslice. This is depicted below, with tasks A and B as previously described having nice values of 0 and 1. The ideal timeslices

of 5.55 and 4.45 msec must be rounded up to 6 and 5, because it is impossible to give a task a timeslice which is a fraction of the tick time (we round up, rather than to the nearest integer, because a time slice of 0 is impossible).

When A is running, at every scheduler tick, it is assessed 1 msec / 0.555 = 1.8 units of vruntime, but when B is running, it gets charged for 1 msec / 0.445 = 2.25 units of vruntime. Looking at the skeleton code for `scheduler_tick()` above, the kernel will not pre-empt the task until it has run out its ideal timeslice (this is a bit of a fib but accept it for now). If no other tasks become ready, A will continue to get 6ms slices and B 5ms. After A has run for its 6 ticks, it has been charged 6 / 0.555 = 10.8 units, and after B runs for its 5 ticks, it is charged 5 / 0.445 = 11.2 units. The two tasks will alternate running for their respective timeslices, but each time A is being cheated of 0.55 msec and B is getting 0.55 msec bonus time. Eventually, the vruntime catches up with B and it skips one turn. For the purposes of the example timeline below, let us assume that the vruntimes of A and B start out at 1000.

Current	VRA	VRB	Action
n/a	1000	1000	A runs first
A	1010.8	1000	B selected
B	1010.8	1011.2	A selected
A	1021.6	1011.2	B selected
B	1021.6	1022.4	A selected
A	1032.4	1022.4	B selected
B	1032.4	1033.6	A selected
A	1043.2	1033.6	B selected
B	1043.2	1044.8	A selected
...			
B	1270	1280	A selected
A	1280.8	1280	B selected
B	1280.8	1291.2	A selected
A	1291.6	1291.2	B selected
B	1291.6	1302.4	A selected
A	1302.4	1302.4	A selected
A	1313.2	1302.4	B selected
B	1313.2	1324.8	A selected
etc.			

A and B alternate 6ms and 5ms time slices. After 27 volleys, A’s vruntime winds up being equal to B’s after running for its 6ms. Since B is not "better" then A gets another turn to run. A has run for 28 turns at 6ms each or 168ms of total actual CPU time, while B had 27 turns of 5 ms each or 135ms. In this 303ms elapsed period, A’s CPU share is 55.5% and B’s is 44.5%, just as it should be.

The `vruntime` value meets the traditional definition of dynamic priority, even if the Linux kernel doesn't use that term to describe it.

Scheduler interactions with fork

In the CFS scheduler, upon a fork, the child process inherits the vruntime (and nice value or weight) of the parent at the time of the fork. This somewhat mitigates the cheating that could otherwise occur if child processes were given a 0 initial vruntime.

Interactive performance & pre-emption

The vruntime approach naturally favors a process that has woken up after a long sleep, because its vruntime has not been incremented. When a process W is awoken, its vruntime is compared to the current process C. If VR(W)>VR(C), then the current process is still "better" than the awoken, and it will not be pre-empted. This can happen if the current task has a much better nice value than the awoken task.

The actual algorithm in the kernel is somewhat more sophisticated, because of the need to consider the last CPU that a task ran on before sleeping or getting pre-empted. Depending on how the load is balanced, and the length of time elapsed, it may be better to let a task wait a little longer in order to get back on its last CPU.

Group Scheduling

In a sense, it might be considered unfair that under the traditional UNIX scheduling model, a user who has 20 processes running is getting a bigger share of the overall CPU time than a user with just 1 process. The only cap to this is the per-user process limit. Modern Linux kernels allow for "group-based" scheduling. Tasks can be placed into groups (not related to the "group" as in gid), either automatically by uid, or manually by the system administrator. The percentage of CPU time allocated to each group can then be tuned by the administrator. Under this model, each user can be restricted to a maximum amount of the overall CPU time, and that user's tasks compete among themselves to divide up that share.

Real-Time and Quasi-Real-Time Scheduling

The scheduler in a general-purpose operating system is designed to provide fairness and interactive responsiveness. This is not necessarily appropriate for embedded systems which are controlling physical systems. Consider an absurdly hypothetical system which runs a nuclear power plant. Task A operates the control rods and task B updates power production records for billing purposes. If Task A becomes ready to run because some

action needs to be taken with the control rods, it would not be a great design if it had to wait until Task B completed its billing computations.

In a real-time scheduling system, tasks are given fixed, static priorities by the system administrator. A high-priority task, if ready to run, always pre-empts a low-priority task. In fact, the lower priority task will not be able to run at all until all higher priority tasks are asleep. If multiple tasks at the same priority level are ready to run, some real-time schedulers say that the first task to become ready runs first and continues to run until it sleeps or voluntarily yields. Linux calls this `SCHED_FIFO` scheduling policy. Another approach, which Linux calls `SCHED_RR`, says that each ready real-time task in a given priority level will run for a fixed **quantum** or **time-slice**, after which the next ready task at that priority level will be allowed to run, etc. Eventually, the CPU will get back to the first task. This is known as **round-robin** scheduling. However, the pre-emption by higher priority tasks continues to take place.

The Linux kernel uses the convention that static priorities run from 0 to 99, with 99 being the best. Non-realtime tasks have static priority of 0, while realtime tasks have a non-zero static priority. Therefore, if *any* realtime task is ready to run, the non-realtime task has to wait. Only privileged users can create real-time tasks, otherwise it would be trivial to monopolize CPU resources.

The Linux kernel and most other general-purpose kernels are not truly real-time kernels, because they can not guarantee a specific minimum response time between an event (e.g. an interrupt arrives) and the scheduling of the real-time task. A term that is often used is "quasi" real-time or "nearly real time." There are kernels which are designed from the ground-up to be truly real-time. Such kernels are seen only in embedded systems and would not be very suitable for general-purpose computing. Often a real-time, embedded system will use one kernel for the real-time stuff, and host a general-purpose kernel such as Linux for administration or user interface. However, Linux has been making steady progress. Recent kernels include support for deadline-based scheduling rather than simplistic static priority scheduling for realtime tasks.

I/O Scheduling and Fairness

Traditional UNIX systems only concerned themselves with CPU time allocation fairness. But in terms of user and administrator perceptions, the issue of disk I/O and network fairness is just as important. Think about an I/O bound system where user 123 has spawned off a bunch of processes that are making a lot of disk I/O requests. These processes have ordinary nice value of 0 giving them "average" priority. Now uid 0 has a system maintenance process with nice value of -20 (best priority). The nice value would only affect CPU timeslice allocation, but the scarce resource here is disk I/O bandwidth.

Modern Linux kernels solve this by using similar algorithms for determining I/O scheduling and network traffic scheduling, which is calls Completely Fair Queueing. Processes have an I/O priority which is analogous to the cpu scheduling `nice` value.

When a given disk device has multiple pending requests, the I/O scheduler takes into account the I/O priority associated with the task that originated the I/O request and tries to apportion the I/O bandwidth accordingly, just as the CFS scheduler attempts to apportion CPU time.

Signals

Within the UNIX virtual computer (process) model, signals serve a role analogous to interrupts in a physical computer. They (can) the cause execution of the user-mode program to be interrupted and transfer to a pre-established handler, just like vectored hardware interrupts. They can also be masked and unmasked.

Signals may be viewed as taking place in two stages: **sending** (aka **generating** or **posting**) a signal, and then **delivery**, i.e. the action in the recipient process which results. Signals may be sent explicitly by one user process to one or more processes (including itself) via the `kill` system call. The kernel may also send different signals to a process as a result of errors or events, in which case, like hardware interrupts, signals can be said to be **synchronous** or **asynchronous**, if, respectively, the cause of the signal can or can not be associated with a particular instruction or operation that the process is currently executing. Here are some common sync and async signals:
• Illegal memory access: SIGSEGV or SIGBUS (sync)
• Illegal instruction opcode: SIGILL (sync)
• Hangup on controlling terminal session: SIGHUP (async)
• Keyboard interrupt (typically ^C key) from controlling terminal: SIGINT (async)
• Asynchronous I/O notification: SIGIO (async)
• Death of a child: SIGCHLD (async)

kill system call

A process can send any other process, including itself, any signal by using the `kill` system call:

```
int kill(pid_t pid,int sig)
```

The user id of the killer must match the recipient, i.e. you can't willy-nilly kill other folks' processes. However, the super-user (uid==0) may send any process a signal. Special values can be used for `pid` to do nifty things:
• pid== -1: Kill all processes with user id equal to killer.
• pid== -n: (n!=1) Kill all processes in process group n.
• pid==0: Send to all processes in the process group

The name "kill" is a bit of a misnomer. Depending on a number of things that we will discuss momentarily, sending a signal to a process will not necessarily cause it to die. ["process groups" are beyond the scope of this course]

The `sig` number is best specified using a constant from the system header files, as discussed below. A 0 value for `sig` doesn't actually send the signal, but checks whether it would have been possible to do so. It can be used to probe the existence of a process.

Pre-defined UNIX signal numbers and behaviors

UNIX defines dozens of signals, some system-dependent, which represent different conditions. Symbolic names, such as `SIGHUP`, are found by #includ'ing `<sys/signal.h>`. The assignment of signal numbers is kernel-specific. Although certain signal numbers such as 1, 2, 3, 9 and 15 are unlikely ever to change, good programmatic practice is to use the symbolic names at all times. The following signal definitions are correct for most Linux kernels running on the x86 architecture:

```
Sig#      Name              Def Act           Desc
1         SIGHUP            Terminate Hangup on controlling tty
2         SIGINT            Terminate Keyboard interrupt
3         SIGQUIT           Coredump  Keyboard abort
4         SIGILL            Coredump  Illegal Instruction
5         SIGTRAP           Coredump  Tracing/breakpoint trap
6         SIGABRT           Coredump  Abort (raised by abort(3))
7         SIGBUS            Coredump  Bus Error (bad memory access)
8         SIGFPE            Coredump  Floating Point/Math Exception
9         SIGKILL           Terminate Non-blockable kill
10        SIGUSR1           Terminate User definable
11        SIGSEGV           Coredump  Segmentation Violation (non-exist addr)
12        SIGUSR2           Terminate User definable
13        SIGPIPE           Terminate Write to a broken pipe
14        SIGALRM           Terminate Alarm clock (see alarm(2))
15        SIGTERM           Terminate Normal terminate request
16        SIGSTKFLT         Terminate Co-processor stack fault (UNUSED)
17        SIGCHLD           Ignore    Child exit/status change
18        SIGCONT           Resume    Continue stopped process
19        SIGSTOP           Stop      Stop process (unblockable)
20        SIGTSTP           Stop      Keyboard stop
21        SIGTTIN           Stop      Stopped waiting for tty input
22        SIGTTOU           Stop      Stopped waiting for tty output
23        SIGURG            Ignore    Urgent data on socket
24        SIGXCPU           Coredump  Exceeded CPU time limit
25        SIGXFSZ           Coredump  Exceeded file size limit
26        SIGVTALRM         Terminate Virtual alarm clock (see setitimer(2))
27        SIGPROF           Terminate Profiling timer
28        SIGWINCH          Ignore    Tty window size change
29        SIGIO             Terminate Signal-driven I/O (aka SIGPOLL)
30        SIGPWR            Ignore    Power going down
31        SIGSYS            Coredump  Bad system call #
```

To learn more about signals on your Linux platform, `man 7 signal`

Pending signals and signal masks

The kernel maintains two bitfields concerning signals for each process. Conceptually, these are arrays of boolean (1-bit) values. There is an element of this array for each valid signal number (signal 0 is never a valid signal, so this array conceptually starts at index

1). A bit in the *pending signals* array is set when a particular signal is posted to that process. Note that since this is a boolean array, not an integer array, the kernel does not "count" *how many* pending signals there are of a given signal number, just the presence or absence of such.

The other such (conceptual) array represents the *signal mask*, or the list of signal numbers which are currently blocked. A 1 bit means that signal is blocked (masked) while 0 means it can go through. When a signal is posted to a process and that signal is currently masked, the signal is not forgotten, because the corresponding bit in the pending signals array is set first. The pending signal can not be delivered when it is blocked, but if and when that the signal number is un-blocked, that signal will be delivered. We will discuss the mechanisms for changing the signal mask shortly.

When is the signal delivered?

Referring to the process state diagram shown in the previous Unit #3, *delivery* of a signal only happens when the process is in the KERNEL RUNNING state and is about to transition back to USER RUNNING. At this point, we see the note "Check for signals." If there are any bits set in the process' pending signals bitmap which are not being blocked, then the pending signal is acted upon. That is the moment of delivery (we'll see later what happens if multiple signals are pending). After delivery, the pending bit is cleared.

When a signal is generated in another process, that process is obviously in the KERNEL RUNNING state (since it is executing the `kill` system call). However, the target process is not necessarily in a RUNNING (user or kernel) state. Therefore, delivery of the signal does not necessarily happen right after generation. It is indeterminate when this happens.

On a uniprocessor system, the generating process is on the lone CPU and the target process must therefore be in a non-RUNNING state (unless a process is sending a signal to itself). Therefore delivery can not take place until the scheduler decides that it is the target process's turn to be scheduled-in.

However, on a multi-CPU system, the target process might be in a RUNNING state on a different CPU than the sender. Rather than wait for potentially a full clock tick cycle for the kernel to regain control of the target process and deliver the signal, under these circumstances the kernel, running on behalf of the sender process on say CPU #1, causes a hardware **inter-processor interrupt (IPI)** to be sent to the target, say on CPU #2. This forces CPU #2 to run the interrupt handler, which gives control back to the kernel.

When the signal is being generated by the kernel in response to a fault (e.g. bad memory access), that is a case of the process sending a signal to itself. The kernel posts the appropriate signal (e.g. SIGSEGV) to the currently running process, and the signal is

delivered almost immediately, as soon as control tries to return to USER RUNNING.

Blocked signals are not delivered at all while the signal is still blocked, and then are delivered at some indeterminate time after the signal becomes unblocked.

Signals and Sleeping

When a process is in the SLEEPING-INTERRUPTIBLE state mentioned in Unit #3, the arrival of a signal, which is not currently being blocked, will cause the target process to wake up from the sleep. It will transition to the READY state and then, when it is scheduled, to KERNEL RUNNING and then try to return to USER RUNNING, at which point the kernel acting on behalf of the process will deliver the signal. However, if the target process is in SLEEPING-UNINTERRUPTIBLE, then the arrival of a signal does not have an immediate effect. The signal is remembered but the target continues to sleep. This means that there is no user-level way to wake up a process in an uninterruptible sleep, except when the event for which it is waiting is satisfied.

As we'll see in Unit #8, the decision of whether a sleep event is interruptible by a signal is made on a case-by-base basis in the kernel code. Generally, things that are expected to be "long" waits, such as waiting for keyboard or network I/O, are interruptible while "short" waits such as waiting for disk I/O are un-interruptible. Unfortunately the Linux documentation is extremely poor at describing this and often one has to read the kernel source code to figure out the correct answer.

Signal delivery & dispositions

Upon signal delivery, the pending signal bit for that signal number is cleared. There are several basic outcomes or "dispositions". As we will see shortly, the process can establish the desired outcomes on a per-signal-number basis. The possible dispositions are: terminate, terminate w/ core dump, stop, resume, ignore, handle. Let's look at each one in detail:

Process Termination by delivery of signal

If the disposition of a particular signal is "terminate" and that signal is delivered, the receiving process immediately exits, because the kernel effectively makes the `_exit` system call on behalf of the process. This means that, just as if `_exit` had been called voluntarily by the program:

- All open file descriptors are closed
- All memory in use by the process is released
- The reference count to the current directory of the process is decremented
- Other resources that we haven't covered, such as locks, are released
- An `rusage` structure is left behind along with the exit status, to be picked up by a

parent process using the `wait` family of system calls.

Note that stdio buffers are *not* flushed here, because the code to flush them doesn't run -- the process instantly exits. This is why debugging output is usually sent to `stderr`, the pre-defined `FILE *` stdio stream which has buffering set to unbuffered.

Termination by signal and termination by `_exit` system call are mutually exclusive, and they are the only two ways that a process can die. We saw in the last unit that these two mutually exclusive causes of death are encoded in the 16-bit exit status word that the `wait` system call retrieves.

Termination with core dump

In some cases determined by signal number (see table above), termination is accompanied by the generation of a **coredump** file. This file is named `core` and is created in the current working directory of the process, as if the process itself had opened this file for creation/writing and wrote to it. If the process does not have permission to create this file, it is not created. A resource limit (see the `ulimit` command or the `setrlimit` system call) can be set to limit the maximum size of the core file. Be careful: many popular Linux "distros" set the default core ulimit to 0, which means core files are not generated. The core dump file contains the contents of all relevant memory sections ("core" is an old term for main memory) at the time of termination, as well as the registers. This allows a debugger (such as gdb or adb) to reconstruct the call stack and the contents of all local and global variables, and determine where in the code the offense occurred. The format of the core dump file is architecture-specific.

Ignore the signal

When a signal is posted to a process which is IGNoring that signal, *and* that signal is not currently blocked, then the signal will be discarded. However, if it is masked it is remembered in the pending signals array (because the disposition could be changed later and then the signal un-blocked). Certain signals (SIGKILL and SIGSTOP) can't be ignored.

Stop and Resume

This is used in conjunction with **job control**, which allows a running process or processes, attached to a user's terminal session, to be frozen, unfrozen, detached or re-attached to that terminal. Stopping the process is not the same as terminating it. Stop is the default action for certain signal numbers (see table above) which the kernel has assigned for job control. **Resume** is the default action for `SIGCONT`, which is the only signal which can produce this action.

Handle the Signal

If the disposition has been set to a valid function address using the `signal` or `sigaction` system calls documented below, then the signal handler is invoked (see below).

Establishing signal disposition

Classically, the `signal` system call provides the means for controlling signal disposition, although it can not block or unblock signals.

```
void (*signal(int signum, void (*handler)(int)))(int);
```

This rather complicated prototype indicates that, for the signal `signum`, the handler will be set to the function `handler` (which takes an `int` argument). `signal` will return the previous value of the handler, which can be stashed away if it is desired to restore it later. On failure, `signal` returns the value `SIG_ERR`.

In addition to a bona-fide signal handler function address, `handler` can be:
• SIG_IGN: Ignore the signal or
• SIG_DFL: Reset to the default handling for this signal, as given in the table above. Note that certain signals have a default disposition which is to Ignore (SIGCHLD, SIGURG, SIGWINCH, SIGPWR).

These are numbers chosen so that they can't possibly be the valid address of a function.

The signal handler

A signal handler is an ordinary function. When a signal is delivered and the disposition of that signal number is to invoke a handler, it appears as if the handler function had been called from the instruction within the program that was executing when the signal was delivered. This is analogous to an interrupt service routine in responding to a hardware interrupt.

The signal handler function is called with a single integer parameter, the signal number that is being handled (using the `sigaction` interface, additional parameters can be passed. This is beyond the scope of this unit). The same handler function can be specified for multiple signals and this parameter can be used to disambiguate.

In most versions of UNIX, including Linux, the default behavior is that when signal *n* is delivered to a process and handled, *n* is temporarily added to the blocked signals mask upon entry to the handler, at the same time that the pending signal bit for *n* is cleared. This prevents the signal handler from being called re-entrantly. (If you don't want this behavior, see discussion of the `SA_NODEFER` flag). Upon return from the handler, the original blocked signal mask is restored. If the signal *n* was received while the handler's

was executing, then the handler will be immediately re-invoked (but this is not re-entrant).

The signal handler function has a return type of void. When the signal handler function returns, execution resumes at the point of interruption. For synchronous signals, execution resumes by re-trying the instruction that caused the fault. If the cause of the fault has not been corrected, then the program can get stuck in an endless loop of fault, handler, fault, handler, etc.

If the signal handler does not return, it either exits the program, or takes a "long jump" :

setjmp/longjmp

Consider a command-line mathematical program. The main loop executes commands through a function `do_command`. Eventually, one of these commands leads, via a long chain of function calls, to:

```
long_compute()
{
          /* a very long calculation */
}
```

At some point during `long_compute()`, the user gets impatient and hits ^C. We wouldn't want to abort the entire program and potentially lose unsaved data. We'd like to return to the command prompt. We would also like to avoid cluttering the code with frequent checks of some sort of global "abort" flag. Thankfully, C has a user-level mechanism known as "setjmp/longjmp". Another name for this is **non-local goto**.

```
#include <setjmp.h>

jmp_buf int_jb;              /*typedef'd in setjmp.h, really a int[] */

void int_handler(int sn)
{
          longjmp(int_jb,1); /* No need for & since it is an array */
          /* NOTE: there is a flaw in this code */
          /* See text for discussion of siglongjmp and other solutions */
}

main()
{
 char cmdbuf[BUFSIZ];
          (void)signal(SIGINT,int_handler);
          for(;;)
          {
                    printf("Command> ");
                    if (!fgets(cmdbuf,sizeof cmdbuf,stdin)) break;
                    if (setjmp(int_jb)!=0)
                    {
                              printf("\nInterrupted\n");
```

```
                        continue;
                }
                do_command(cmdbuf);
        }
}
```

`setjmp`, from the program's standpoint, appears to be a function that is called once, and returns one or more times. `setjmp` stores context information, such as the program counter, stack pointer and register contents, into the `jmp_buf`, which is simply a `typedef` for an array of a sufficient size to hold this information. The program counter location stored is the address of the instruction following the call to setjmp. setjmp returns 0 when it is called (for the first time.)

A call to `longjmp` uses the context information stored in the supplied `jmp_buf` to return control to the setjmp point. The second argument will be the perceived return value from setjmp. It can not be 0, and if 0 is supplied the argument will silently be changed to 1. `longjmp` restores all of the saved register values, except the one used to indicate the return value from a function, which it sets to the supplied value.

The jump buffer must be visible both to setjmp and to longjmp, which means it must be a global variable, not a local variable. Because longjmp rolls back the stack frame to its state when setjmp was called, longjmp must be called from a function descended from the one in which setjmp was called. Otherwise, the stack frame may have already been overwritten by other function calls and sheer terror and chaos will reign.

In effect, longjmp provides the ability to `goto` across function boundaries. As is the case with ordinary `goto`, the use of `setjmp/longjmp` is not (necessarily) an evil thing! Frequently it is the cleanest way to bail out of deep code.

Note that setjmp/longjmp are entirely a user-level mechanism. They are not system calls.

Signal Mask Restoration After Longjmp

When a signal handler takes a longjmp, the restoration of the blocked signals mask does not take place. In the example above, subsequent SIGINT will not have any effect, because the blocked signal mask will have SIGINT set. There are several possible solutions:

• We could use `sigaction` to establish the signal disposition and specify `SA_NODEFER` in the `sa_flags`. Then the handler will be invoked without masking SIGINT. If a second SIGINT arrives between the invocation of the handler and the longjmp, the handler will be called re-entrantly. In this particular example, no harm will come from that (the `longjmp` operation is idempotent) so that's a valid solution.

• We could use the `sigsetjmp` library function instead. This function also saves the state of the process blocked signals mask inside the context buffer (which is now of type `sigjmp_buf`). The corresponding `siglongjmp` library function restores the mask

just before completing the longjmp. *Caveat*: The restoration of the mask and the jump are not *atomic*. There is a brief *race condition* window where the signal is unmasked but the longjmp hasn't happened yet. Therefore, if another instance of SIGINT had arrived while the handler was running, the signal handler may wind up getting invoked re-entrantly. Again, in the particular example above, this does not cause a problem.

• After we have arrived back at the setjmp location from the handler, we could call `sigprocmask` to unmask SIGINT. This may result in a nearly immediate re-invocation of the handler, but this is not a re-entrant invocation, because the first instance of the handler has already jumped out.

Masking/Blocking signals

We will now explore how to manipulate the process's blocked signals mask. The block/unblock mechanism is most useful in protecting **critical regions** of code, where data structures are in an inconsistent state. For example, consider a routine to transfer a balance between accounts:

```
transfer_balance(a,b,n)
struct account *a,*b;
{
          a->balance-=n;
          b->balance+=n;
}
```

Every 24 hours, we want to prepare a summary report:

```
main()
{
          /*...*/
          signal(SIGALRM,alarm_handler);
          set_alarm();
          /*...*/
}

alarm_handler()
{
 struct account *p;
          for (p=first_account;p;p=p->next)
                    printf("Account %s   Balance %d\n",a->name,a->balance);
          set_alarm();
}

set_alarm()
{
          alarm(24*3600);
}
```

This uses the `alarm` system call to set an alarm which, at the specified number of seconds in the future, will cause delivery of SIGALRM to the process.

Consider what happens when the alarm signal arrives between the two statements of `transfer_balance`. The data structure is in an inconsistent state and the summary report is incorrect. We will examine these issues of concurrency and synchronization in greater detail in Unit 6. For now, the best fix is to block the alarm signal during the critical region:

```
transfer_balance(a,b,n)
struct account *a,*b;
{
 sigset_t set;
          sigemptyset(&set);
          sigaddset(&set,SIGALRM);
          sigprocmask(SIG_BLOCK,&set,NULL);
          a->balance-=n;
          b->balance+=n;
          sigprocmask(SIG_UNBLOCK,&set,NULL);
}
```

Each process has a bit field which represents which signals are blocked. The `sigprocmask` system call can be used to set bits in that mask, remove bits, set the entire mask at once, or query the current value of the mask. It uses a data type `sigset_t` to abstract the issue of how many bits are in that vector, and what data type has enough bits to hold it. For example, on the Linux kernel, where there are 64 signal numbers and when running on an architecture with 32 bit ints, it is defined in the system header files as:

```
typedef struct {
        unsigned long sig[_NSIG_WORDS];                  //this works out to [2]
} sigset_t;
```

If the SIGALRM signal arrives while it is blocked, it will stay in the pending signal set until the second `sigprocmask` system call removes the mask. Shortly thereafter the signal will be delivered and the handler will be invoked.

The sigaction interface

In the early years of UNIX, there were some inconsistencies in how the signal(2) system call functioned, in particular some UNIX versions reset the signal disposition of the signal in question upon entry to the handler. Because of the potential confusion, a new system call was introduced among all UNIX variants in the 1990s. The `sigaction` system call uses a more complicated interface, but can accomplish a lot more in a single call.

```
int sigaction(int sig,const struct sigaction *act, struct sigaction *oldact)

struct sigaction {

          void      (*sa_handler)(int);
          sigset_t sa_mask;
```

```
        int       sa_flags;
        /* Some more stuff...pretty complicated, read man page */
};
```

As with `signal`, `sa_handler` can be set to SIG_DFL to reset to default action, or SIG_IGN to ignore the signal, or the address of the signal handler function. `sa_flags` is a bitwise flags word that specifies a potpourri of options, including:

• SA_RESETHAND: Restore signal action to SIG_DFL when entering handler, aka unreliable signal semantics.

• SA_NODEFER: Do not automatically block signal while being handled. Aka SA_NOMASK

• SA_SIGINFO: Call the signal handler with 3 arguments. The first argument is the signal number. The second is a pointer to a `siginfo_t` structure which contains a lot of useful (and potentially architecture-specific) information about the cause of the signal. The third argument is a pointer to a `ucontext_t` structure which represents the context at the point in the program at which the signal was received. This argument is primarily for use in multi-threaded programs.

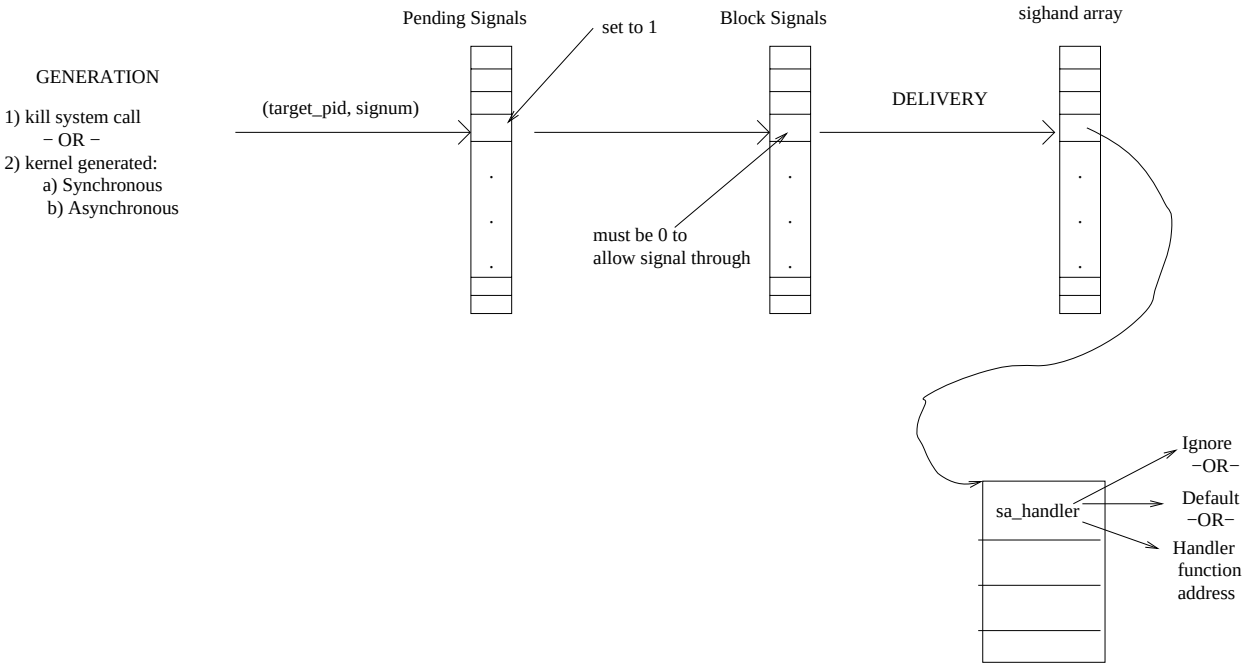
• SA_RESTART: Increases the likelihood that system calls interrupted by this signal and handled will restart when the handler returns. See discussion below under "Interrupted System Calls"

Make sure to set sa_flags=0 if you don't want any flags. Otherwise the field may contain random garbage that will be mis-interpreted as various SA_XXX flags!

`sa_mask` describes which signals to block upon entry to the handler, in addition to the signal being handled itself, which is blocked unless SA_NODEFER has been specified. Be sure to set this correctly with `sigemptyset`, `sigaddset`, etc. otherwise extraneous signals may be masked during your handler's invocation.

On modern releases of Linux, the `signal` system call is just a wrapper which translates to a `sigaction` system call, with the sa_flags set to 0 and the sa_mask set to empty. `man 2 signal` documents this in more detail.

The diagram below provides a pictorial overview of how signals work:



Real-time signals

Signals were originally conceived as a way to terminate or control processes, hence the rather severe name `kill` for the system call to send a signal. Signals have been used as a primitive inter-process communications (IPC) mechanism, e.g. a daemon process which responds to a SIGHUP and re-reads its configuration files. However, the fact that traditional signals don't queue and carry no additional data makes them of limited use for IPC. In a later unit, we will see better ways for processes to send messages to each other.

On some more modern versions of UNIX, including Linux, **real-time** signals are supported. These are numbered `SIGRTMIN` to `SIGRTMAX` and have no pre-defined meanings. Under the current Linux kernel, signals 32 through 63 are real-time signals. However signals 32-34 are used by the threads library, and should thus be avoided by other code. Therefore SIGRTMIN is defined as 35. Your mileage may vary, so always use the macros and avoid hard-coded signal numbers.

Unlike standard signals, real-time signals queue. If a process is sent, e.g. 5 instances of signal #35 while that signal is being blocked, upon un-blocking the signal, it will be delivered to the process 5 separate times. Furthermore, the real time signals are delivered in the order in which they were sent.

The `sigqueue` system call is similar to `kill`, but allows the sender of the signal to attach a small, opaque chunk of data:

```
int sigqueue(pid_t pid, int sig, union sigval value)
```

```
union sigval {
          int sival_int;
          void *sival_ptr;
}
```

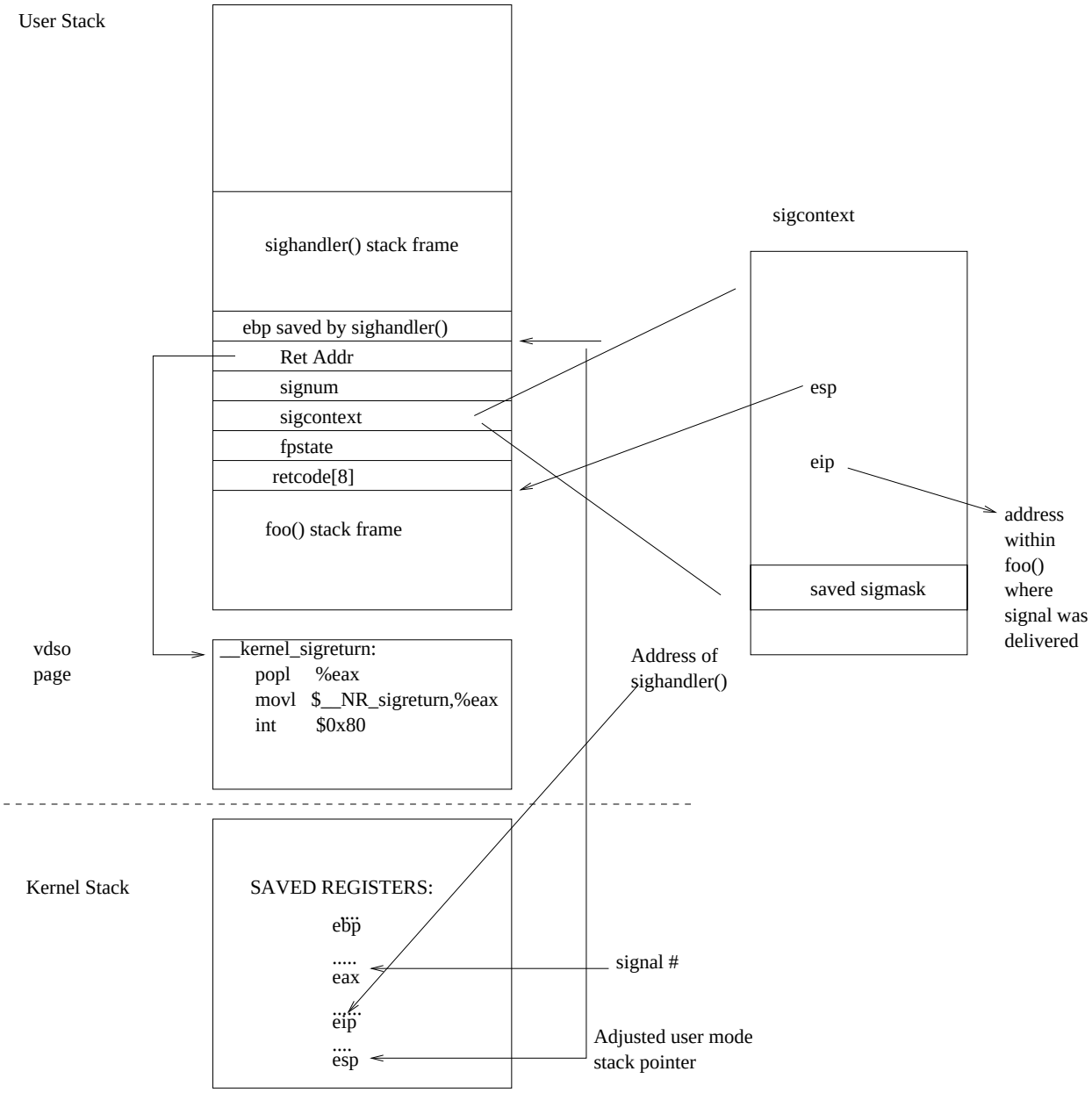
The union will be the larger of the int size or the pointer size for that machine. The contents are passed along to the signal handler by the kernel and are opaque (the kernel does not interpret the union value). `sigqueue` could also be used to send a traditional signal (signal number < 32), but such a signal still won't queue. The opaque data will be delivered, but if multiple instances of the traditional (not real-time) signal occur, newer instances will overwrite the earlier data.

Invocation of and return from signal handlers -- kernel interactions

The kernel controls the invocation of a signal handler. When a signal is delivered to a process and the disposition is set to a handler, the kernel reaches into the user-mode stack and creates a new stack frame which makes it appear that the handler was called from whatever point in the program that we happened to be in when the signal arrived.

However, there is another trick: In order to implement certain semantics, such as restoring the original blocked signals mask upon exit from the handler, the kernel needs to place a "hook" in the stack. The return code in this artificial stack frame is not the program counter location of the point where the signal was received. Instead, it is a pointer to a small region of memory that the kernel has created (once, when the program was started) and populated with a few bytes of executable code which cause the `sigreturn` system call to be invoked. This is not a system call that one would generally invoke explicitly from within program code!

When the kernel set up the stack frame, it stashed the real return address, along with a copy of all the registers, the old signal mask, and a bunch of other stuff, all inside of that stack frame. The `sigreturn` syscall looks back at the user-mode stack and fetches these saved values, restors the signal mask and performs any other needed restoration actions, then returns back to user mode, with the program counter *now* set to the point at which execution should resume in the main program, and the stack pointer accordingly restored to its correct place.



Interrupted System Calls

Certain system calls are designated as "long" calls. Generally, these are calls that may block for an indefinite period of time, for example, reading from the terminal or network connection. When an asynchronous signal arrives while the process is blocked in a "long" system call (of course a synchronous signal can not arrive at this point since the process is not executing instructions), and that signal is handled with a signal handler function, the system call is said to have been interrupted. Assuming the signal handler returns (it doesn't longjmp out or exit the process), the system call will return a failure

with errno set to EINTR (Interrupted System Call).

(To be clear, the system call returns but the handler is invoked first. I.e. if we look at the stack trace, the handler appears to have been called from the point of just returning from the system call library function. Then when the handler returns, we resume from that point and the code following system call return executes. This section about "interrupted system calls" applies only to system calls that are interrupted by a signal before completing, and the signal is handled, and the signal handler returns.)

Often, this is exactly what one wants. However, if it is necessary to make the signal handling perfectly transparent, the system call must be "restarted" when the signal handler returns and the process must go back to sleep until the system call finishes. To specify that behavior, use the SA_RESTART flag with `sigaction`. This flag defaults to OFF.

Restart means that the system call is effectively made a second time with identical parameters. This doesn't always make sense, specifically if the previous, interrupted system call had some non-idempotent effect, such as reading or writing bytes from a file descriptor. Therefore, in certain situations, the Linux kernel will not restart an interrupted system call, even though SA_RESTART is on, and will give the EINTR error, or will give a non-error return value from the system call reflecting the partial work that the system call did before being interrupted. In some cases, the kernel must go through great internal work to make a system call transparently restartable, although this is not visible to the user.

Then there are a few rare situations where the kernel will *always* restart a given interrupted system call, regardless of the SA_RESTART flag.

The moral of the story: When coding a program that uses signal handlers, most system call errors have to be checked and special logic used so that `EINTR` is not treated as a fatal error, if there is a possibility that a signal is received and handled while that system call is sleeping.

SIGCHLD

The `SIGCHLD` signal is somewhat special. By default, whenever a process terminates, a `SIGCHLD` signal is sent to its parent. The default disposition (SIG_DFL) of `SIGCHLD` is to ignore the signal, but this is not quite the same thing as explicitly ignoring the signal with `sigaction` or `signal`. The distinction has to do with compatibility with different varieties of UNIX, some of which (including Linux) maintain that when `SIGCHLD` is explicitly ignored (disposition is set to SIG_IGN), it tells the kernel that the parent has no interest in the child, and thus a zombie will not be created.

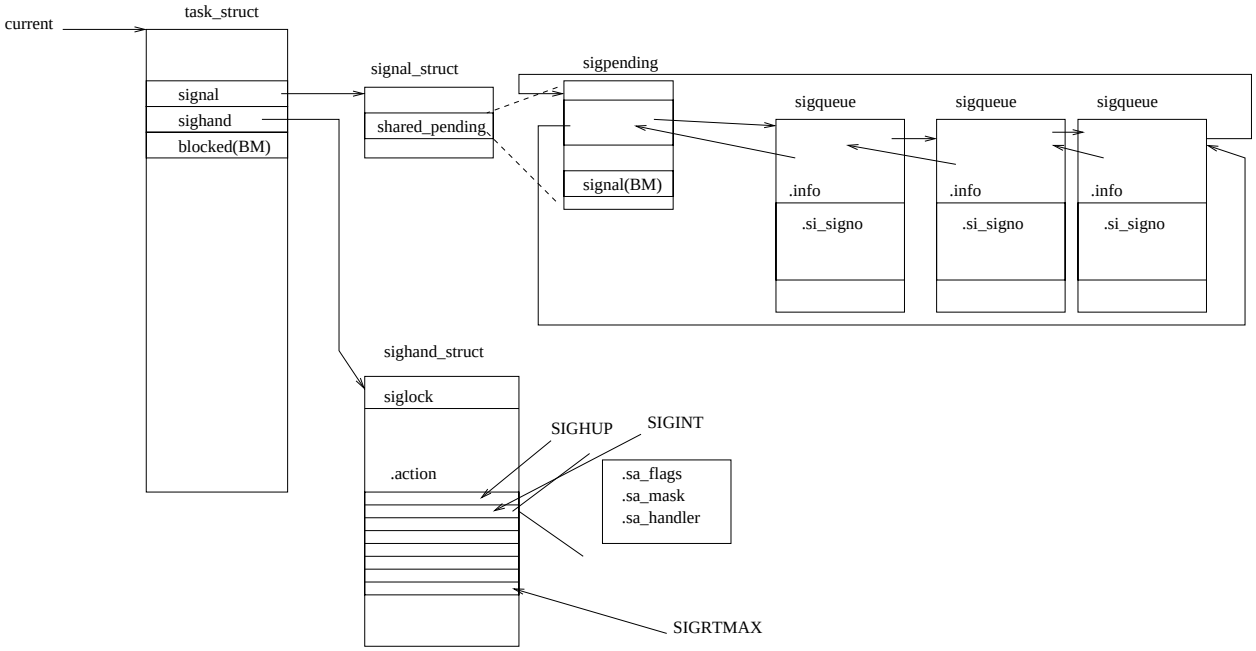
Signal interactions with fork and exec

During a successful `exec` system call:
• The kernel clears the set of pending signals. The set of blocked signals remains the same.
• Any signals which were set to go to a signal handler are changed to `SIG_IGN`, because in the new executable, the address of the signal handler(s) would obviously be invalid.
• If the SIGCHLD signal was being explicitly ignored, its disposition is reset to the default (which is to implicitly ignore it).

After a `fork`, the set of pending signals for the child process is initialized to be empty. The parent is not affected. The signal disposition in the child and set of blocked signals is exactly the same as the parent.

Visualizing kernel data structures related to signals

There are three members of the `struct task_struct` which keep track of signals on a per-process basis. They are illustrated in the figure below.



The `sighand_struct` is essentially an array of structures very similar to the `struct sigaction` used in the system call of the same name. This system call simply installs the provided `sigaction` structure into the kernel's table for the specified process and signal number, subject of course to proper permissions and error checks.

The `signal_struct` (note the awful naming conventions) keeps track of pending signals. We are glossing over the details of "shared" signals which can be delivered to any thread in a multi-threaded program and "private" signals. The `shared_pending`

member is embedded in the `signal_struct` and anchors a doubly-linked list of pending signals, each of which is represented by a `struct sigqueue`. There is also a bitmask (unfortunately named `signal`) which allows the kernel to quickly determine which signals are pending by signal number.

When a signal is posted to a process, the kernel adds it to this "queue" of pending signals. Because signals 1 through 31 do not queue or "count", if there is already a pending signal of the same number, the new signal is discarded. However, real-time signals (32 through 63) do add to the queue under all circumstances.

When the kernel checks to see if any signals need to be delivered to the process (this happens whenever returning to user mode) it examines the bitmask in signal number order, and picks the lowest-numbered pending signal. It runs through the list of pending signals and removes the one with the matching signal number, then delivers that signal. At the same time, the corresponding bit in the pending signals array is cleared if that is the last instance of that signal number in the pending signals queue. For non-real-time signals, that is always the case since the queue can not contain multiple instances of the same signal number.

During a `clone` system call, if the `CLONE_SIGHAND` flag is set, the parent and child process share the same `sighand_struct` in kernel memory. This flag is normally set when threads are created within a multi-threaded program, therefore any thread can call `sigaction` and affect the signal disposition for the entire program. Of course, during `fork`, all clone flags are 0, and the parent and child have separate signal handler tables after the fork (the child’s starting out as an exact copy of the parent’s). There are additional complications regarding signal handling in multi-threaded programs which we are not going to cover in this course.

Pipes

The UNIX **pipe** provides the fundamental means of interprocess communication, in accordance with the UNIX philosophy of providing small, flexible tools that can be combined to build up larger solutions.

A pipe is a uni-directional FIFO. It is created with the `pipe` system call:

```
main()
{
 int fds[2];
          if (pipe(fds)<0)
          {
                    perror("can't create pipe");
                    return -1;
          }
```

Two new file descriptor numbers are allocated and returned via the two-element array `fds`. `fds[0]` is the read side of the pipe, `fds[1]` is the write side.

Although a pipe is accessed like a file, using file descriptors, there is no path in the filesystem which refers to the pipe. This means that processes which communicate via a pipe must be descended from a common process which created the pipe and passed along the file descriptors via the I/O redirection methods covered in Unit 3. This is a common application when spawning a pipeline of commands from a UNIX shell, in which all of the command processes are descended from the same shell process. In subsequent units, we will explore other pipe-like mechanisms that allow arbitrary processes to communicate, including processes on different host systems.

Pipe properties

• The pipe is uni-directional. An attempt to read from the write side, or write to the read side, will be greeted with EBADF. On some systems (e.g. Solaris), the pipe is bidirectional, however, this behavior should not be relied upon.
• The pipe is a FIFO. The order of the bytes written to the write side of the pipe will be strictly preserved when read from the read side. Data will never be lost, modified, duplicated or transferred out of sequence. However, if multiple processes are attempting to read from the same pipe, the distribution of data to each will be unpredictable.
• Message boundaries are not preserved by the pipe. If one process performs, e.g., 4 small writes to the pipe, and later a read is performed, all of the data will be returned as one large chunk. In order to perform record-oriented, as opposed to stream-oriented I/O, some sort of application-level record marking protocol needs to be employed (e.g. records delimited with newlines).

Flow Control

• When reading from a pipe, the `read` system call will block until there are some data available. It will then return any and all available data (up to limit specified by the caller

as the third argument to read), and will not wait until an entire buffer full of data are available. [We are ignoring non-blocking I/O options for now.]
• Each pipe is a FIFO of a specific capacity, which is system-dependent and possibly controllable by the system administrator. A typical FIFO size is 64K. If there is insufficient room in the FIFO to handle the entire write request, the write will block until there is room. This provides the "back pressure" against the writing process to create "flow control" and throttle a fast writer piping to a slow reader. (Again, we are ignoring non-blocking I/O options at this time. Non-blocking behavior can be specified for pipes using the `fcntl` system call. This is beyond the scope of this unit.)

Atomicity, Interrupted read/write, Short Writes

If the write request size is 4K or less, the write will be handled **atomically**, and, in the event that there are multiple processes writing to the same pipe, these atomic writes will be preserved and not interleaved. However, if the write request size exceeds 4K, the write will be broken up into smaller chunks of 4K, and could potentially be interleaved with other 4K chunks from other writers. However, pipes are rarely used in this fashion.

Unless non-blocking I/O is being used, a write system call to a pipe blocks until the entire requested amount has made it into the pipe. For example, let's say the pipe is completely full, and a process does `write(pipe_wrfd,buf,65536)` then after some period of time, the reader drains off 4096 bytes. Under this scenario, the write system call does **not** return 4096. The process remains sleeping. When the 65536th byte has been written into the pipe, then write returns, with a return value of 65536. If a programmer had written bad code that doesn't correctly handle "short writes" there is no harm here, just as short writes won't generally happen when writing to ordinary files on disk.

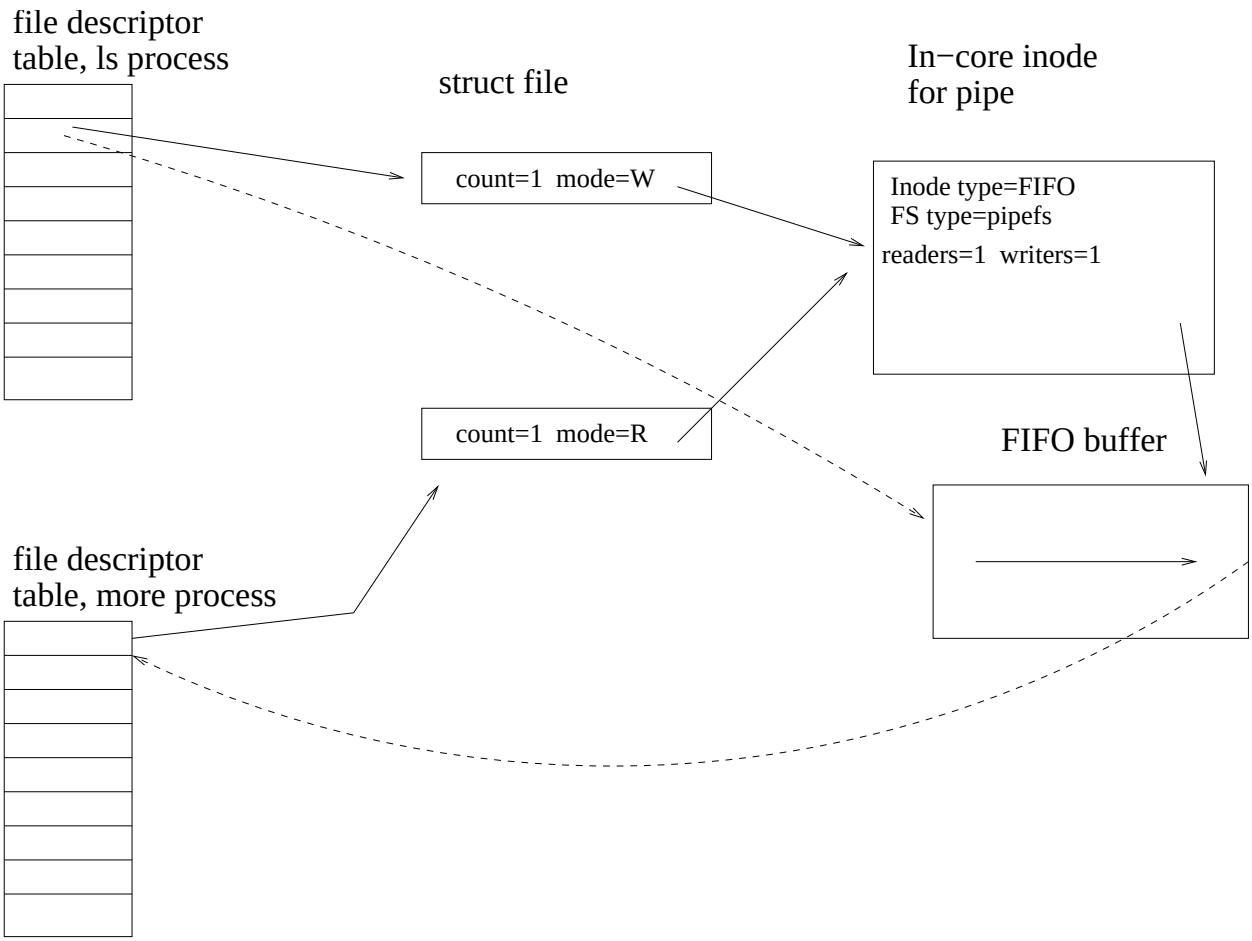
However, things get much uglier if signal handlers are involved. Let us repeat this scenario. The reader drains 4096 bytes and this allows the writer to move 4096 bytes into the pipe. At this point, a signal arrives. Since waiting for pipe read/write is considered a "long" sleep, it is interruptible, and the signal wakes up the writer. What is the poor kernel to do? Since 4096 of the 65536 bytes have been written into the pipe, the kernel can't ignore that fact and mislead the caller of the write system call by returning a 0 or -1, both of which would indicate that nothing was written. So, there is no choice but to return 4096. Here is a case where a bad programmer could get a rude lesson in partial writes!

But what if no bytes had made it into the pipe prior to the signal arrival? Now the Linux kernel has options: (1) if the SA_RESTART flag is set in the signal handling table for the signal number that was received, upon return from the user-level signal handler, the kernel will restart the write system call, with the identical parameters (i.e. with the same buffer address and the same write request count). This is fine since nothing actually happened during the first write system call. The process will return back to sleep until the write completes. (2) if SA_RESTART is not set, the kernel causes the write system

call to return EINTR. After the signal handler returns, execution of the program resumes at the point of returning from the write system call, the return value is -1, and errno is EINTR.

Connection Control

• When the write side of the pipe closes (there are no more open file descriptors in any processes referring to it), this generates an EOF condition. Once any pending data have been read out of the FIFO, all subsequent reads will return 0.
• When the read side of the pipe closes, this condition is known as a **broken pipe**. Any pending data in the FIFO are discarded. An attempt to write to a broken pipe (including a previous write which is sleeping waiting on a full pipe) will result in the failure of the write system call with EPIPE. A signal (SIGPIPE) will also be delivered to the writer process. The default action of SIGPIPE is to terminate the process. This ensures that a program which is poorly written and does not error check the `write` system call doesn't continue to try to push data into the broken pipe forever.
• In order to preserve this behavior with respect to EOF and SIGPIPE/EPIPE, it is important to close any extraneous file descriptors that might be hanging on to the read or write side of a pipe, especially during dups and forks.
• An interesting "feature" that occurs with broken pipes is that if the pipe is partially filled and then the reader side is closed, this will **not** generate an error on `close` on the write side. This would appear to be a design flaw, in that data loss would be undetectable from the writing process's view. However, the presumption is that a parent process invoked both the reader and the writer and connected them with a pipeline. The abnormal exit of the reader process would be detected by the parent (through the `wait` system call) which would then report the failure of the pipeline as a whole.
• The figure below illustrates a writer process (ls) connected to a reader process (more) via a pipe.



Named Pipes

Pipes created with the `pipe` system call do not have any associated pathname in the filesystem namespace. This means that the only access is through the file descriptors; it is not possible to use `open` to gain access. Thus, the only way two processes can communicate using a pipe is if they share a common ancestor.

A **named pipe** is identical to a regular pipe, but it is accessed through a node in the filesystem having type `S_IFIFO`. A named pipe is sometimes called a FIFO. To create this special node one can use the `mknod` system call:

```
if (mknod("/tmp/namedpipe",S_IFIFO|0600,0)<0)
{
        perror("unable to create named pipe /tmp/namedpipe");
}

$ ./a.out
$ ls -l /tmp/namedpipe
```

```
prw-------    1 hak      root            0 Sep 25 23:12 /tmp/namedpipe
```

In the example above, the leading "p" in the ls output identifies the created inode as a named pipe (IFIFO). Note that the permissions mode is bitwise-OR'd.

The FIFO inode can also be created from the command prompt using the `mknod` or `mkfifo` command.

Semantics of named pipes are slightly different from anonymous pipes. Since the named pipe is opened with `open`, it can be opened read-only, write-only or read/write (i.e. `O_RDONLY`, `O_WRONLY`, `O_RDWR`). There can be more than one opening of the pipe too (i.e. multiple `struct file` in the kernel). When a named pipe is opened read-only, the open system call blocks until there is at least one instance of the same pipe being opened for writing (or read/write). Likewise, a write-only open blocks waiting for readers. A read/write open always succeeds immediately.

`read` from a named pipe returns whatever data are currently in the FIFO, or blocks if there are no data, just like anonymous pipes. If there are no writers, read returns 0 (after returning any pending data). But unlike an anonymous pipe, this EOF-like condition is not permanent. If another writer appears, future reads may return additional data. In the meantime, all reads would return 0. Because this condition results in a "spin loop," it is better practice to close the read side of the named pipe upon an EOF condition, and then re-open it. The open will then block until another writer comes along.

Likewise, a write to a named pipe with no readers results in either a `SIGPIPE` or `EPIPE`, just like anonymous pipes. Furthermore, the same kind of non-blocking options can be applied.

Named pipes can be used as a means of inter-process communication among unrelated processes on the same local system. They can be used as a command/control channel to connect with "daemon" processes that are running in the background, disconnected from any controlling terminal.

Virtual Memory

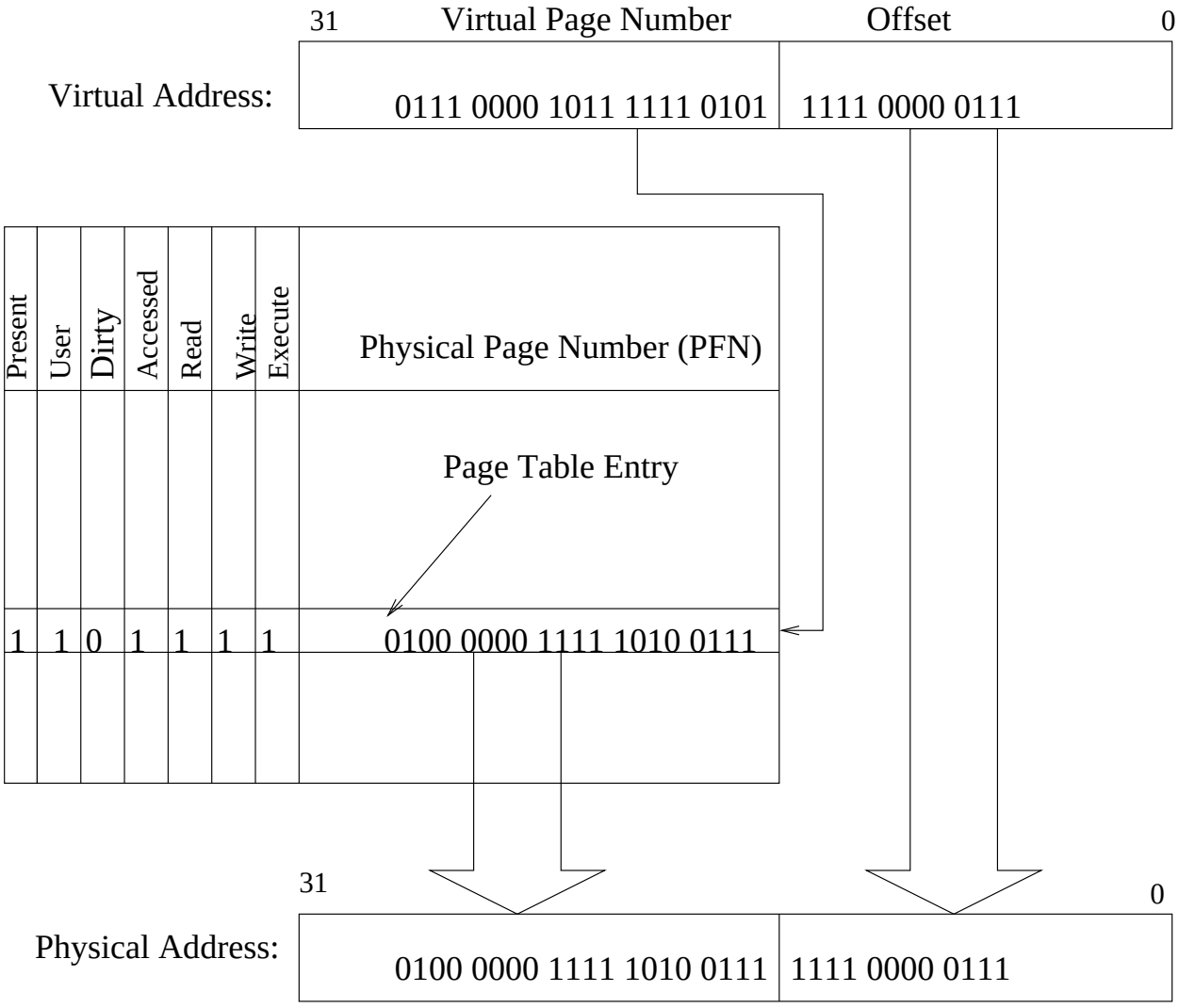
We have stated that a UNIX process is a virtual computer in which a thread of execution (virtual processor) runs within a private virtual address space. In this unit, we will begin to explore the mechanisms by which this is implemented, both from a generic hardware and a generic kernel standpoint.

All addresses used in a program are **virtual addresses**. Before being used to address physical memory or memory-mapped I/O, virtual addresses are **translated** to **physical addresses**. This translation is performed **by hardware** within the processor known as the **Memory Management Unit (MMU)**.

Since the operating system kernel has sole access to the MMU's lookup tables, it has the ability to completely control the view of memory a given process can see.

Address translation is performed with the granularity of a **page**. An address is divided into two parts: The **page number**, comprising the most significant bits, and the **offset**, comprising the least significant bits. Address translation operates on the **virtual page number**, replacing it with the **physical page number**. The offset is passed through unchanged. Typical page sizes are 2K, 4K and 8K. On the X86 32-bit and 64-bit architectures a page size of 4K is used.

Another name for a physical page of memory is a **page frame**, because we can think of physical pages as empty frames into which the actual meaningful content is placed from time to time. We'll see that a given page frame will hold many different virtual page images over time.



The virtual page number (conceptually) indexes the following per-page data:

- **Present bit**: If set, this virtual page is resident (maps to a physical page frame) and the physical page number field is valid. If the Present bit is clear, there is no physical page frame allocated to this virtual address. When there is no translation (page table entry has clear Present bit), it results in a **Page Fault**. As we will see, this is not necessarily a bad thing.

- The **Physical Page Number** (also known as the Page Frame number or PFN) which is translated for this virtual page. If the Present bit is clear, this field is invalid.

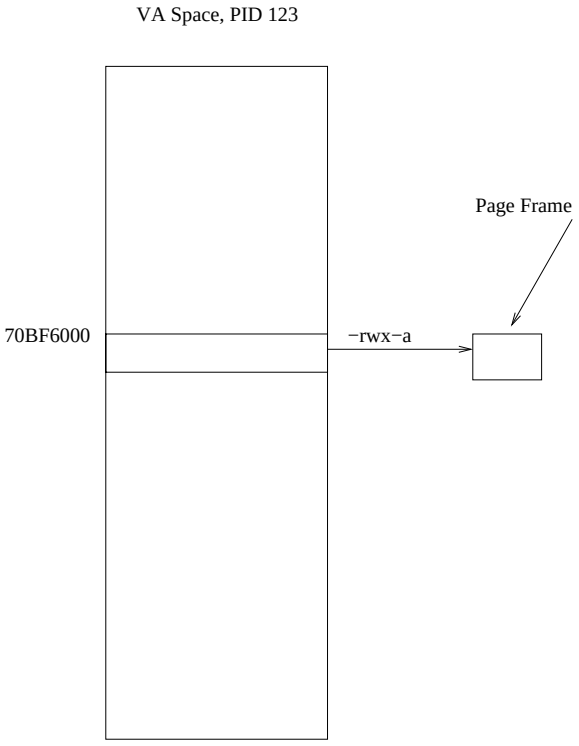
- **Protection bits**: Determines what type of accesses are allowed to this page: read, write, execute. An attempt to perform a disallowed access results in a **protection fault**. (Not all hardware maintains all three of these protection bits. In particular,

the x86 architecture prior to Pentium-4 has only two levels of access control: readonly and readwrite, and does not distinguish between protection faults and page faults.)

- **Dirty bit**: Set by hardware when a page is written to via this PTE. Cleared by the kernel when the page has been synced to backing storage.

- **Accessed bit**: Set by hardware when a page is accessed (read, write or execute) via this PTE. Cleared by the kernel while aging (scanning) page frames for re-use.

- **User/Supervisor Bit**: When set, this page can be accessed while the processor is in user mode. If clear, the processor must be in Supervisor (aka "kernel") mode to access the page. This allows kernel memory to be mapped into the virtual address space of a process and be visible only when the process enters the kernel. The Linux X86 architecture uses the User/Supervisor Bit approach, but other architectures switch to a different set of page tables upon entry to the kernel.



Multi-Level Page Tables

Conceptually, as depicted above, the virtual page number is an index into an array of

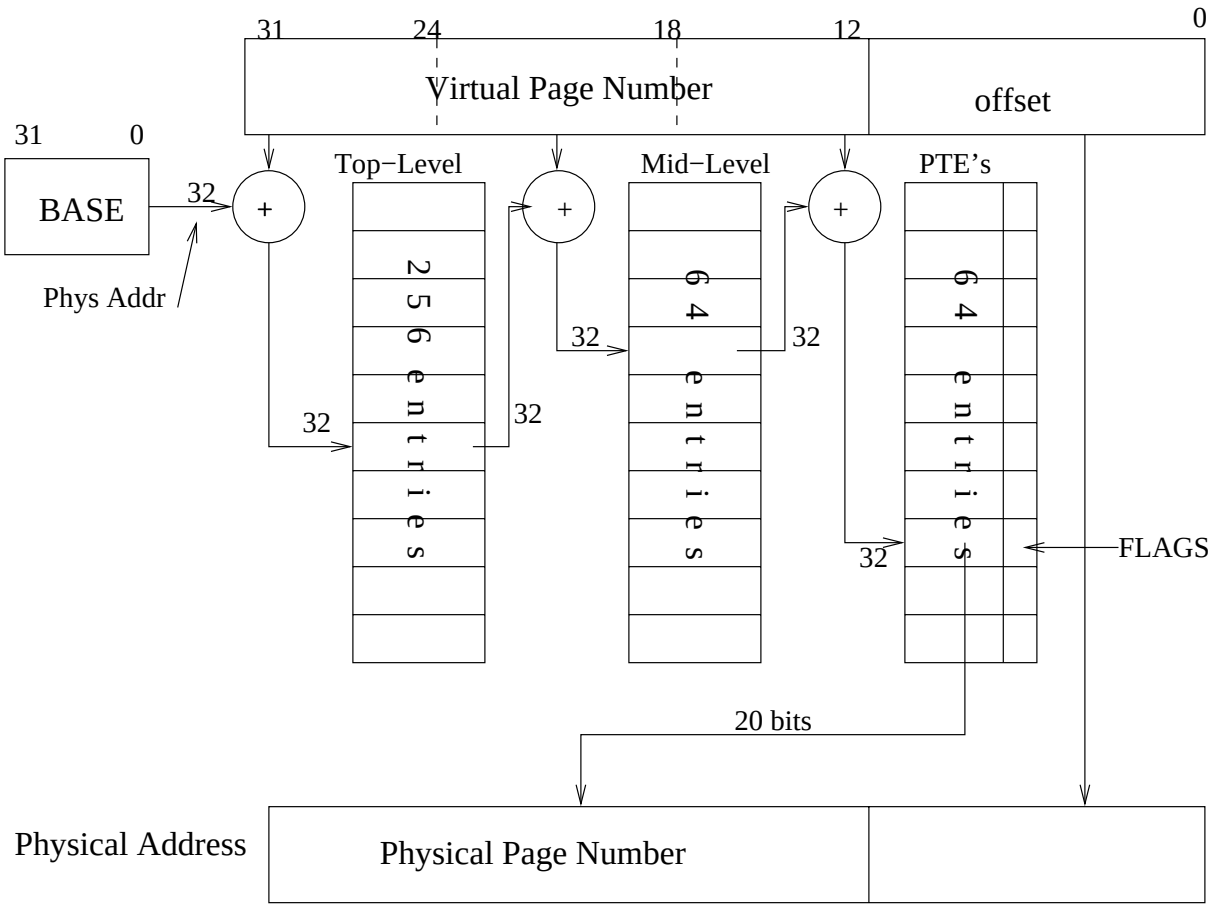
Page Table Entries (PTEs), located at a specific physical address. There are several problems which prevent this from being a realistic implementation.

Let us consider a 32-bit architecture with 4K pages, therefore the page numbers are 20 bits long and there are 2^20 possible page numbers. If each PTE is 32 bits long, then each page table would consume 4MB. Although this might seem like a small amount of memory, remember that each process on the system has its own page table. A system with 100 processes would therefore be wasting a large amount of memory on page tables.

The problem is even more severe on 64-bit machines which have a larger virtual address space. Even if only 48 bits of the address space are recognized, with a 4K page, there would be 2^36 PTEs (each PTE must be 64 bits to handle the longer addresses) consuming 2^39, or 512GB, of physical memory per process!

The problem of large page tables is addressed with **multi-level** page tables. The physical address of the top-level page table is given in the CPU/MMU "BASE" register (register %cr3 on X86). The most significant bits of the virtual address index into this top-level table. The result is not the page table entry, but is the physical address of the corresponding table at the next lower level of the hierarchy. That physical address, plus the bits from the next most significant part of the virtual address, locates the entry in the next page table, etc. At the last step, the actual PTE (page table entry) is fetched, which contains the PFN and the flags.

On modern computer architectures, 2-, 3- or 4-level paging structures are the norm. The illustration below shows 3 levels, and is meant to be schematic, not any particular processor type. 2-level or 3-level structures are typical for 32 bit machines, while 64 bit virtual addresses mandate 4-level structures to achieve any kind of memory efficiency.



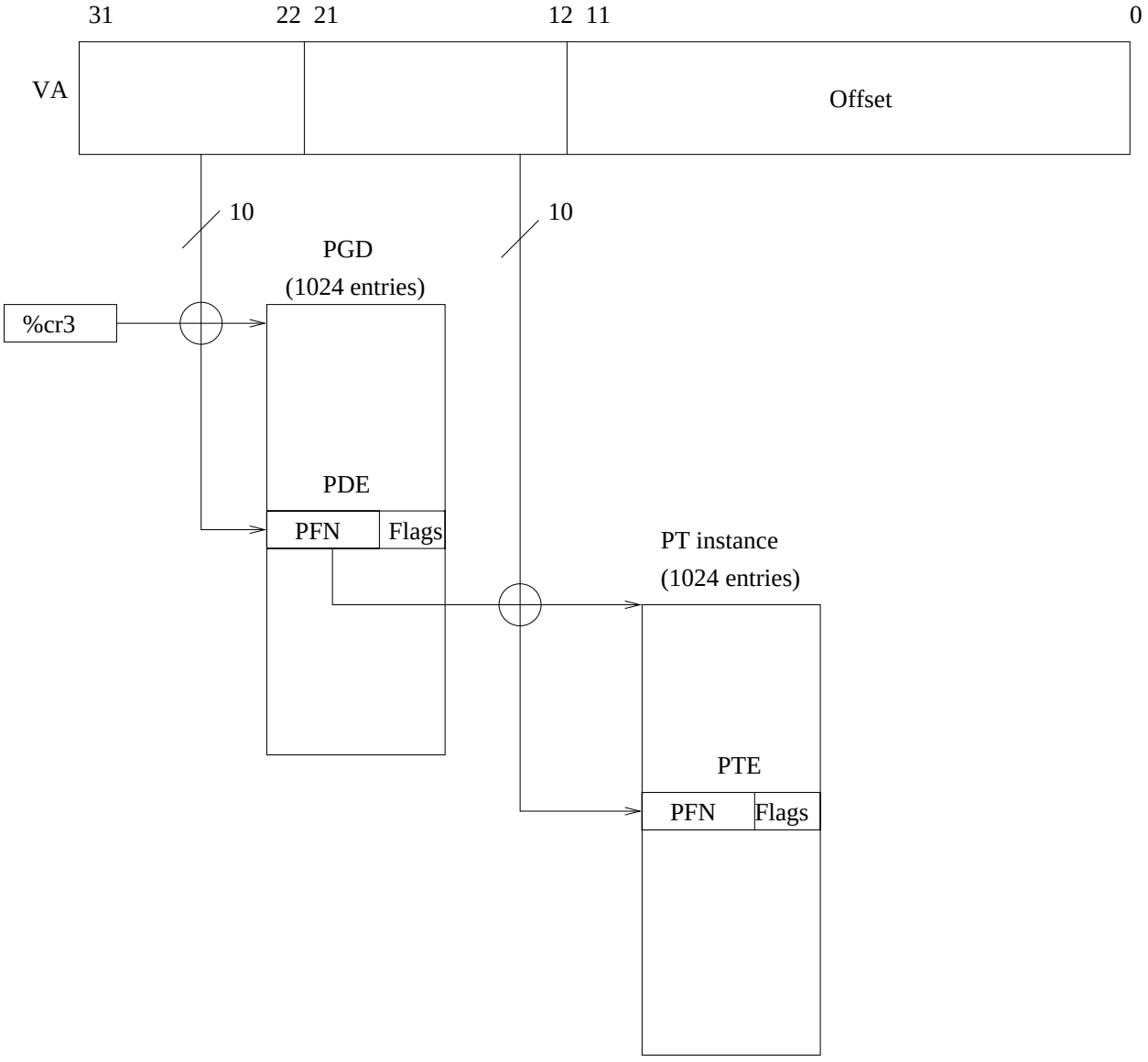
Page tables are in effect a data structure, kept in physical memory. The layout of this data structure is mandated by the processor architecture. The kernel has been compiled with architecture-specific code that understands the layout of the page tables and can manipulate and manage them.

X86-32, 2-level page structure

On the X86 32 bit system, the page table is 2-level with the page number split 10/10 bits. The %cr3 special control register in the CPU (which must be running in Supervisor mode to change it) gives the 32-bit physical memory address of the start of the top-level table. The Linux kernel calls this the Page Global Directory (PGD). The PGD is an array of 1024 entries, each 32 bits long, and each entry (known as a PDE: Page Directory Entry) is in the same format as a Page Table Entry (PTE). However, the Page Frame Number (PFN) field of the PDE is a pointer to the next level of the 2-level structure. Each PGD entry represents 2^22 (4MB) of virtual address space.

We see that the PGD consumes 4096 bytes, or one physical page. The next level, the Page Table (PT), contains 1024 PTE (Page Table Entries) each 32 bits (4 bytes) long, thus each PT also fits within a 4K page. The virtual address space and the physical address

space are both 32 bits, or 4GB. There is a mode called Physical Address Extension (PAE), which we will not cover here, that makes physical addresses 64 bits so more than 4GB of physical RAM can be accessed. However, in PAE mode, virtual addresses are still limited to 32 bits. PAE is rarely used since most modern computers would operate in 64 bit mode.



X86-64, 4-level page structure

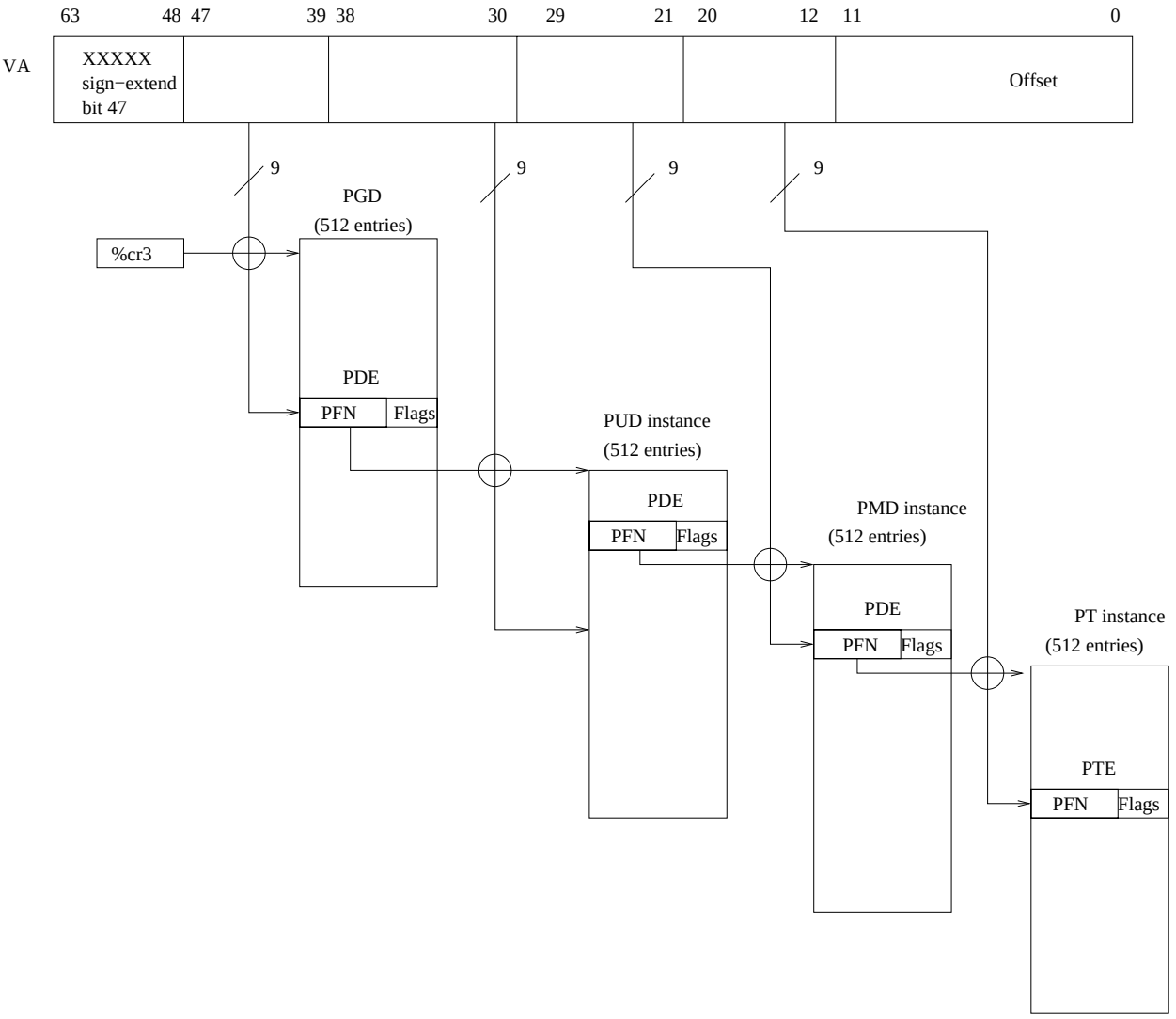
Under the X86 64 bit architecture, registers are 64 bits wide. Virtual addresses however are actually only 48 bits wide, and are sign-extended into the remaining 16 bits. This makes addressing somewhat confusing. E.g. valid addresses jump from 0x0000 7FFF

FFFF FFFF to 0xFFFF 8000 0000 0000 (spaces inserted for clarity). Therefore, a given virtual address space is limited to "only" 256TB. This point at which the addresses become "negative" is the dividing line in the Linux x86-64 kernel between user and kernel virtual memory. As with the 32-bit model, the kernel takes the upper portion. Now however the user process has 2^47=128TB of virtual address space and the kernel also has 128TB. The Intel X86-64 architecture supports up to 52 bits (a 40-bit PFN) of physical addressing which is 4PB, although contemporary implementations use a much smaller physical address bus size.

Page size continues to be 4K to retain compatibility with older code. Therefore we have 48-12 = 36 bits remaining as a virtual page number. This is split evenly into four levels (9/9/9/9). Each PTE must be 64 bits wide since 32 bits would not be enough room to encode the 40-bit PFN. Therefore within one 4K table we get 2^9=512 entries.

The %cr3 register (now 64 bits wide) points to the PGD, as with the 32-bit architecture. Entries in the PGD give the PFNs pointing to instances of the next level, which is called Page Upper Directory (PUD). The PUD in turn contains PFN pointers to instances of Page Middle Directories (PMD), which finally contain the pointers to the Page Tables which contain the PTEs.

Each entry of the PGD in 64 bit mode now references 2^39=512 GB of address space. Each PUD references 2^30=1GB, each PMD 2^21=2MB and finally each PTE 2^12=4KB.



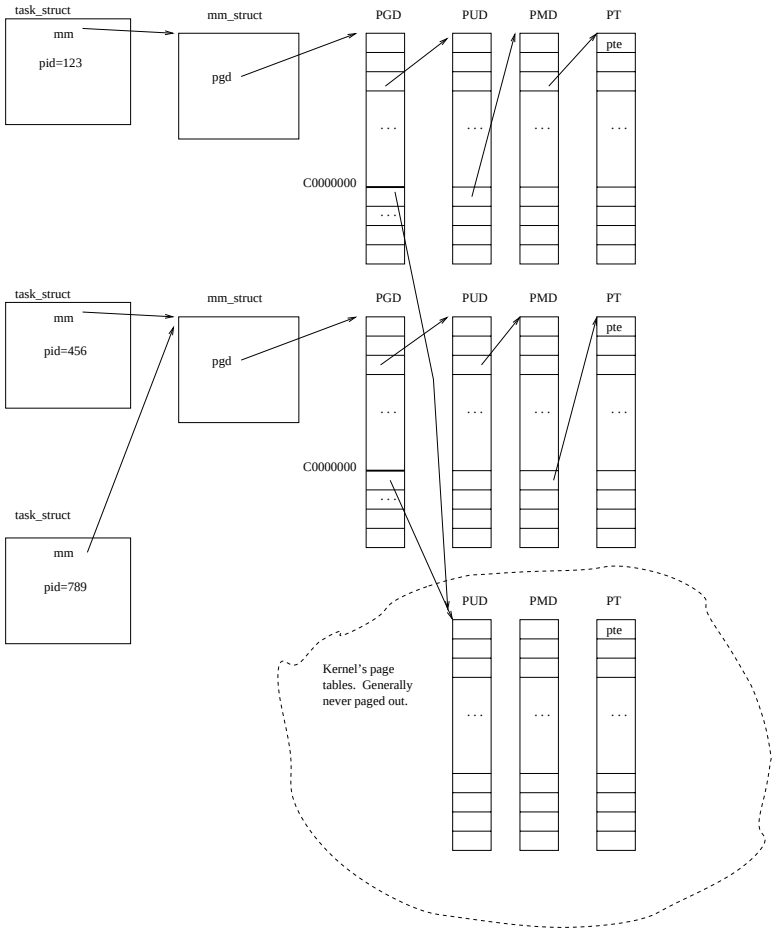
Sparse and Shared Multi-Level Page Tables

It is possible for entries in Page Directory tables (PGD/PUD/PM) to contain "NULL pointers". This means that the entire contiguous range of virtual addresses corresponding to that entry is not mapped. This "NULL" pointer is implemented by having the Present bit of the entry set to 0. Whenever such a NULL pointer is encountered during the traversal of the multi-level page table structure, it causes a Page Fault, the same as if the Present bit of the PTE were 0.

The Linux kernel uses this advantageously in performing "just in time" page table allocations. All processes get allocated an individual PGD. However, the lower-level tables hanging off the PGD entries are not allocated until the corresponding areas of virtual memory are created (or grown) AND a page fault happens (see more under Page Fault Resolution) Areas of virtual memory which are not used (these can be quite large,

especially with a 64-bit system) do not consume additional resources. This is analogous to sparse file allocation (recall from unit 2 that when one lseeks beyond the current end of file and writes, this creates a "hole" in the file for which actual disk storage may not be allocated)

It is also possible for a given Page Table or PUD or PMD to be pointed at by more than one PGD. The Linux kernel uses this trick so it can create just one instance of the multi-level page table structure for the shared kernel memory, and allow all processes to share that (see more under Linux Address Space).



Translations Cache & Context Switch

Page table entries reside in main memory, and thus consume two important resources: The memory needed to store the page tables and the time required to perform translations. We have seen how multi-level page tables address the former concern. The latter is extremely significant too, since a translation must be performed on every memory access. If the MMU worked literally as described thus far, for a 2-level page table, 2

additional memory accesses would be required for every access attempted, i.e. memory performance would be reduced to one third. For a 4-level table, it would be reduced to one fifth! Therefore, a cache, often called the **Address Translation Cache (ATC)** or **Translation Lookaside Buffer (TLB)**, is used to speed translations.

Like any cache, the ATC is a content-addressable memory. A **key** is presented, and either the stored **value** is accessed (a "**hit**"), or the key is not found in the cache, which is known as a **cache miss**. Recall that the address translation process takes the page number portion of a virtual address and returns a page table entry. Therefore, the key in this case is the virtual address being translated (or more correctly, the page number portion of the VA), and the value accessed is the PTE.

For every virtual memory access, the ATC is searched to see if a valid, cached PTE exists for the referenced virtual page number. If so, the translation is performed using the cached page table entry. Otherwise, hardware traverses the page table structure and loads the correct translation, keeping it cached for possible future re-use. The cache is of finite (and typically rather small) size, so the insertion of a new cached translation into the ATC generally eradicates an older cache entry to make room.

A **context switch** is when the kernel changes from one process to another which implies a change in both the task and the address space. (in contrast, a **task switch**, at least in the Linux kernel, could be a change from one thread to another while staying in the same address space). To perform a context switch on X86, the kernel simply modifies the %cr3 register with the physical address of the PGD representing the new process. However, this creates a problem with the TLB, since the cached virtual address mappings are no longer valid in the new virtual address space.

On some architectures (e.g. SPARC, MIPS and X86-64), a **context number** or **Address Space ID** is used in conjunction with the ATC/TLB to disambiguate the virtual addresses of multiple processes' translations which might be resident in the cache at once. Each process is assigned a unique context number. A CPU special register contains the current context number and it is that number, combined with the virtual page number, which keys the associative array. To perform a context switch, both the page table base address register and the context number register are changed at the same time.

While the newer 64-bit X86 processors have support for a context number (it is 12 bits), neither the older 64-bit processors nor the 32-bit processors have this feature. Lacking context numbers, when the page table BASE register (%cr3) is changed, meaning a switch to a different address space, the CPU hardware invalidates (flushes) all TLB entries, except those where the cached PTE has a special G (Global) flag set. The Linux kernel creates the PTEs for the shared kernel virtual address space so they all have the G flag. Therefore, when making a context switch, while the user-mode PTEs get flushed, all of the kernel-mode PTEs can remain in the TLB. This is a big performance boost.

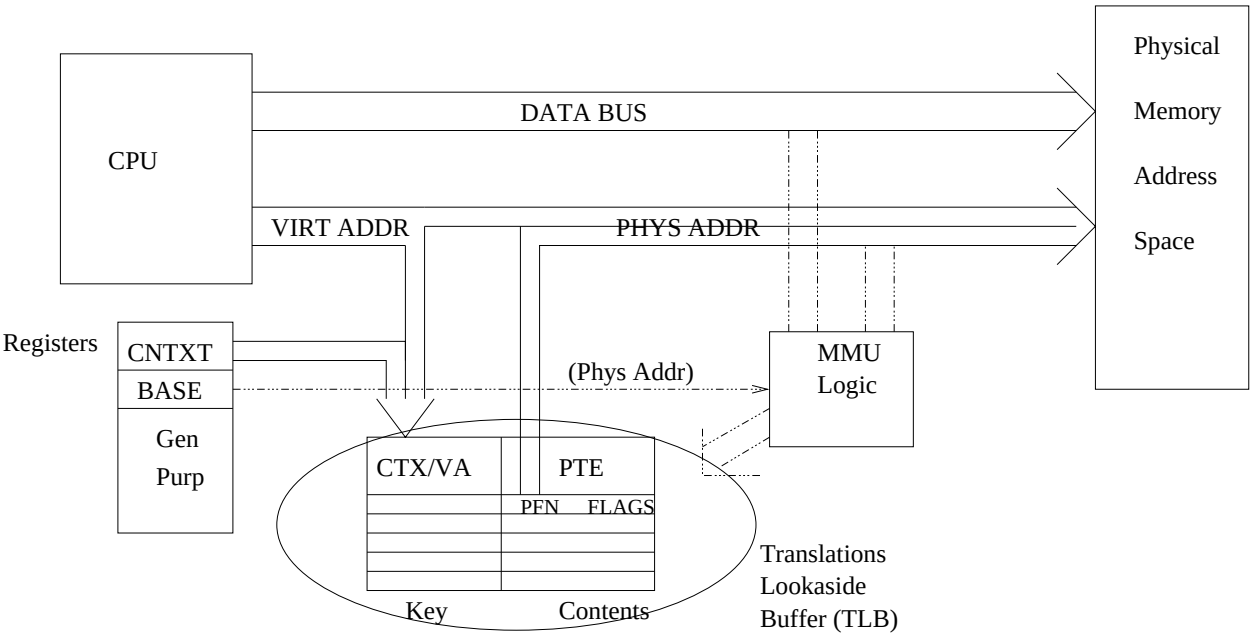
Another complication arises when the kernel has to change the PTE mapping for a given virtual page, e.g. an area of memory is being un-mapped, or changed from readonly to

readwrite. If the PTE is cached in the TLB it must be flushed. On older versions of the X86 processors, it was necessary to flush the entire TLB, but modern versions allow specific entries to be flushed individually with a special (supervisor mode) instruction called `INVLPG`.

Even more fun with TLB flushing comes on multiprocessor systems and especially multi-threaded programs, because each processor maintains its own TLB. It could be that two or more physical processors are executing threads in the same address space (process). If the address space is changed in a way that would invalidate TLB entries, the other processors need to be alerted so they can flush their corresponding entries.

Because the TLB is small, performance is greatly improved when memory accesses are clustered so that, other than the first access, they "hit" the TLB cache. The kernel tries to optimize its internal data structures layout so as to keep frequently accessed data together within the same page. Thus the choice of memory address allocation within the kernel can have a profound impact on performance.

The figure below depicts the flow of address on a generic system with an MMU and a TLB, having the context number feature.



The UNIX user-level memory model

We have seen that the virtual address space of a process consists of a number of regions. These include:

- text region: holds the executable code
- data region: initialized global variables

• bss region: uninitialized (0-filled) global variables including dynamically-allocated variables. The ending address of this region is known as the **break address**, and can be queried or set with the `brk` system call. The break address may turn out to not fall on a page boundary, in which case the bss region really extends to the next page boundary beyond the break address. The standard C library function `malloc` ultimately uses `brk` to request additional bss memory from the kernel. Another name for the dynamically allocated part of the bss region is the `heap`. Typically, the static part of the bss region (corresponding to the declared variables) is the first part (lower memory addresses), and the heap is contiguous with it.
• stack region: function call stack and local variables. With multi-threaded processes, each thread has its own stack. Stack regions have the unusual property that they automatically grow (in the direction in which stacks usually grow, towards low memory addresses on most architectures) without an explicit system call.
• dynamically-linked libraries code
• initialized global variables for dynamically-linked libraries
• uninitialized global variables for dynamically-linked libraries
• shared memory being used for inter-process communication
• memory-mapped files

We will now see that from the kernel's standpoint, these memory region names have no particular significance. The kernel views a virtual address space as a list of regions, each of which has associated with it:
• The starting virtual address of the region (always aligned to a page boundary)
• The length of the region (a multiple of the page size), or equivalently the ending virtual address of the region.
• Bitwise flags which describe properties of the region such as the protections, or the fact that the region automatically grows towards low memory
• Backing store: If the memory region is backed by a node in the filesystem, then this mapping is described by device#/inode# and offset. Otherwise, the region is **anonymous**, and has no corresponding file.

Tracking the Address Space in the Linux Kernel

As depicted in the diagram later in this unit, each `task_struct` in the kernel contains a pointer to a `struct mm_struct` which represents an entire virtual address space (think of "mm" as "memory map"). One of the many elements of `mm_struct` is `pgd`, the physical address of the PGD. There is also the head of a linked list of memory regions, which are kept in ascending order of starting virtual address, and each of which are described by a `struct vm_area_struct`:

```
 struct mm_struct {
         struct vm_area_struct *mmap; /* Head of linked list of regions */
         void *pgd;                   /* Addr of PGD */
         struct list_head mmlist;     /* Linkage to all mm_structs in system */
         int _file_rss,_anon_rss;     /* Current Resident Set Sizes */
         int total_vm;                /* Total VM size, in pages */
         int hiwater_rss;             /* High water mark of RSS usage */
         int hiwater_vm;              /* High water mark of total VM size */
 /* There's plenty more to this structure that we're skipping */
};

 struct vm_area_struct {
         struct mm_struct *vm_mm;              /* Pointer back to address space */
         struct vm_area_struct *vm_next;       /* Linked list pointer */
         void *vm_start;                       /* Start (virt) address of this region */
         void *vm_end;                         /* End address + 1 */
         unsigned long vm_flags;               /* VM_XXX flags */
         pgprot_t vm_page_prot;                /* PTE flags (see text about COW ) */
         struct file *vm_file;                 /* Mapped file, or NULL */
         unsigned long vm_pgoff;               /* Offset (#pages) into mapped file */
         /* And there is more to this structure to be seen later */
};

/* Explanation of vm_flags (bitwise combinations )
VM_READ         Read permission
VM_WRITE        Write permission
VM_EXEC         Execute permission
VM_SHARED       Region may be shared among multiple processes
VM_GROWSDOWN    Region automatically grows towards lower addresses
VM_EXECUTABLE   Region maps to an executable file
VM_LOCKED       Region should be locked in memory and never swapped out
VM_IO           Special region for memory-mapped I/O
VM_DONTCOPY     Discard this region when forking instead of copying
VM_DONTEXPAND   Disallow expansion of region with mremap
VM_DENYWRITE    Disallow over-write of active text pages
*/
```

Establishing a new region with mmap

```
void *mmap(void *addr, size_t len, int prot, int flags,
                   int file_descriptor, off_t offset)
```

The mmap(2) system call causes the specified file which has already been opened with `file_descriptor` to be mapped into the process's address space, creating a new region. It can also be used to create a new anonymous memory region; these two choices are mutually exclusive. The first address of the mapped region will correspond to the specified `offset` in the file, which must be a multiple of the page size, for the specified length `len`. The virtual address at which to map the segment can be specified explicitly as `addr`, but usually it is left to the kernel to choose by passing NULL.

`prot` is a bitwise flag combination which can include the bits PROT_READ, PROT_WRITE or PROT_EXEC.

`flags` is a bitwise flag which can contain, among others:
• MAP_SHARED: Write operations to mapped region change the contents of the file.
• MAP_PRIVATE: Mutually exclusive with MAP_SHARED: Write operations to the mapped region do not change the contents of the file.
• MAP_DENYWRITE: Setting it will prevent write access to the mapped filed using the traditional write system call. See discussion at end of this unit. However, the Linux kernel currently prevents programs from setting this flag with the mmap system call, although the flag is used internally by the kernel to protect executable files.
• MAP_GROWSDOWN: The region should have the property that it automagically grows towards low memory, i.e. it is intended to be a stack region.
• MAP_ANONYMOUS: A new, anonymous region is being requested. The file_descriptor and offset parameters are ignored.

Please read the man pages for mmap to obtain more complete information.

mmap within the kernel

The effect of (a successful) mmap call inside the kernel is to allocate a new `vm_area_struct` and to insert it into the singly-linked list of such memory regions that is hanging off the `mm_struct`. The `vm_flags` field is initialized based on the `flags` and `prot` arguments to the system call. E.g. PROT_READ maps to VM_READ, MAP_GROWSDOWN maps to VM_GROWSDOWN. If MAP_ANONYMOUS was specified, the `vm_file` is set to NULL, otherwise it points to the `struct file` (in kernel memory) that the file descriptor was referencing. Because the reference count (`f_count`) is incremented, the process may close the file descriptor after mmap, and the kernel "hangs on" to the open file instance. `vm_pgoff` is set to the `offset` argument, except the latter is specified in bytes, and is divided by 4096 (on X86 architecture) to arrive at vm_pgoff in pages. It is therefore an error to specify an offset which is not page-aligned.

Shared vs Private Mappings

When MAP_SHARED is specified (equivalent to VM_SHARED in kernel), writing to the memory region is immediately and transparently visible to this or any other process through the traditional system calls such as `read`. Likewise, any changes to the file, e.g. through `write`, are immediately visible through the mapped region.

When the mapping is MAP_PRIVATE, Copy on Write (see below) happens. Upon the first memory write access to a page in a MAP_PRIVATE region, the association between that virtual page and the file is broken. Thus writes to a MAP_PRIVATE region do NOT

cause the associated file to be modified. It is unspecified if writes to the file (e.g. via the write system call) are visible in the memory region, before the association has been broken by a write to the memory region. On the Linux kernel, they are (but see discussion of MAP_DENYWRITE).

Pages in a MAP_PRIVATE area, once they have been written to, become anonymous pages, because the association of that particular virtual page with that particular area of the mapped file has been broken. Therefore, if they need to be paged-out, they are sent to swap; they can not be paged-out to the original file.

File size not multiple of page size, file truncation, atime/mtime

The mapped region may extend beyond the current end of the file. This could happen either because the `len` parameter specified is larger than the file, or because after the mapping has been established, the file size is reduced (e.g. through the `truncate` system call). If the process attempts to access memory which corresponds to just beyond the current end of file, but not beyond a page size boundary, it will see bytes with a value of 0. However, when the process tries to go further out, beyond the page boundary, even though the memory address is within the bounds of the mapped region, the kernel will be unable to satisfy the page fault, because the backing store does not exist in the filesystem. Under this condition, the kernel must deliver a **SIGBUS** to the process.

Read and write access via a file-mapped region will set the inode atime and mtime fields appropriately, but not necessarily instantaneously, for reasons which should become clear later in this unit when we look at the PFRA.

Remapping, Unmapping

The `munmap` system call destroys a memory region and any associated mapping to a file. The `mremap` system call is used to make an existing memory region larger or smaller, or to move it to another virtual address. `mprotect` can change the protections (R/W/X) associated with a region (but those new protections must be consistent with how the file descriptor was open in the case of file-mapped regions). The reader is encouraged to read the man pages for these system calls carefully. How these system calls manipulate the `vm_area_structs` should be obvious.

Exec and virtual memory

During the exec system call, a new address space is initialized and the existing address space of a process is freed. The page tables themselves are freed and any physical pages that the process had mapped are placed on the free list. It is as if the process had called `munmap` for each of its regions.

The basis of the initialization of the new address space is the filename passed to the exec

system call. This refers to an executable file in one of the binary executable formats which that operating system and processor type can execute. The first few bytes of the file determines the executable format, and are known as the **magic number**. We have seen that the a.out file contains the text region image, the data region initial image (variable initializers), and the initial requested size of the bss region.

In Unit 3, a highly conceptual model was presented of how a new program is loaded into memory during exec. Now that we understand mappings between files and virtual memory, we can take a more refined look and see that executable "loading" is really the establishment of a number of mmap regions.

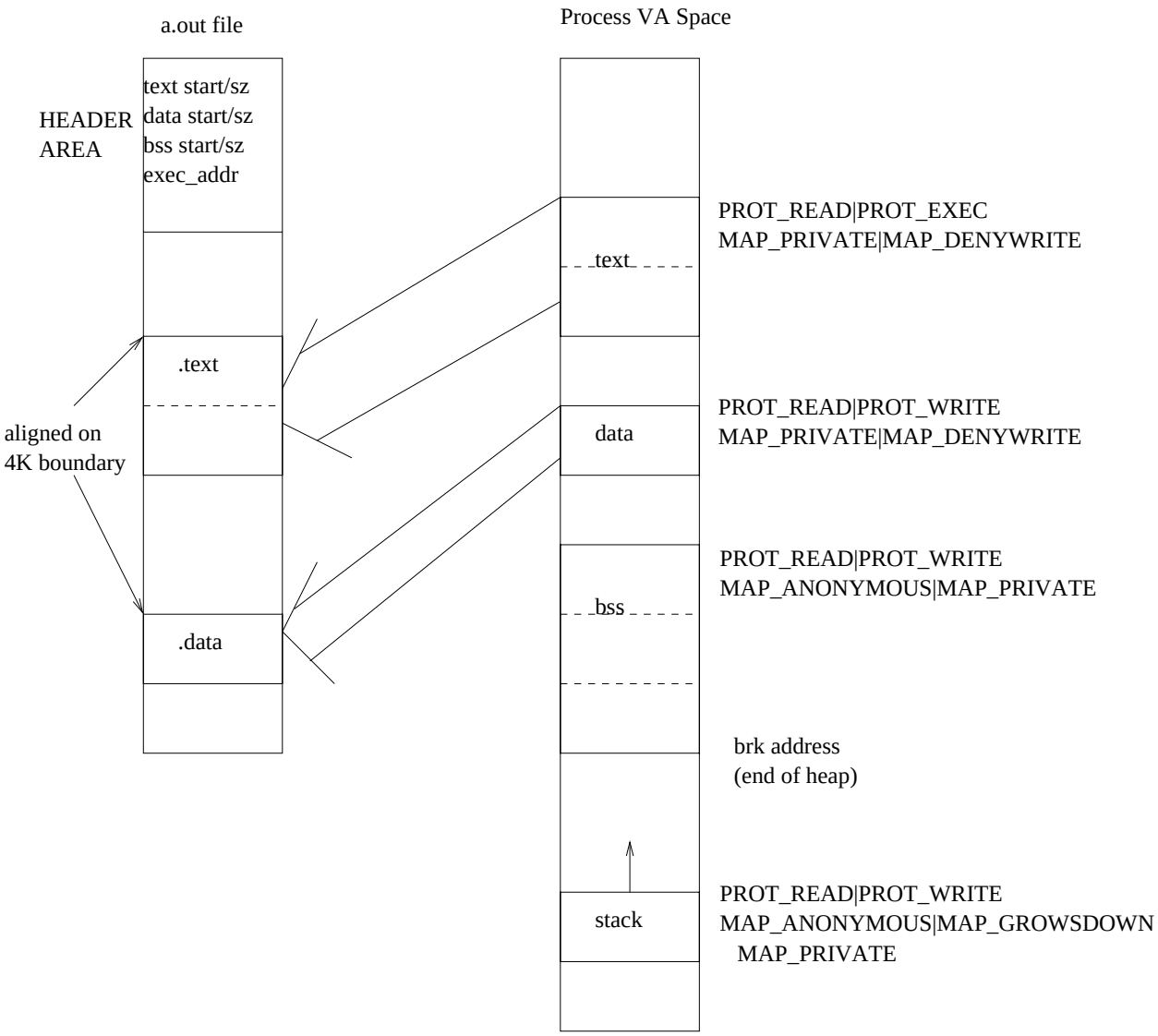
• The text region is created as if the process had used mmap to create a mapping to the area in the a.out file holding the text image. The protections of the text region are PROT_READ|PROT_EXEC. The flags are MAP_PRIVATE|MAP_DENYWRITE. Note that since page frames are never allocated to virtual address spaces until an actual access occurs, the very first thing which the new program does, upon attempting to fetch its initial opcode from memory and execute it, is to cause a page fault! The program is demand-paged in from the a.out file as needed.

• The data region is created as if by mmap with MAP_PRIVATE|MAP_DENYWRITE. The mapping is to the area of the a.out file which contains the .data section (image of initializer values). The protections are PROT_READ|PROT_WRITE. Because the mapping is MAP_PRIVATE, the program "sees" the correct initializers when they are first accessed, but writes to those variables don’t cause the unpleasant consequence of modifying the a.out file.

• The bss region is created, with the size requested in the a.out header, as a MAP_ANONYMOUS|MAP_PRIVATE mapping, with PROT_READ|PROT_WRITE protections. As it is accessed, the kernel will allocate page frames, zero-filling them first. As a MAP_ANONYMOUS region, the bss will be backed by swap space.

• The stack region is created with a small initial size and with the MAP_ANONYMOUS|MAP_PRIVATE|MAP_GROWSDOWN properties. The protections are PROT_READ|PROT_WRITE. The stack region will be grown as it is accessed, as described previously, with new page frames being zero-filled. The stack is also backed by swap space.

In the example below, a new program has been set up by exec with a text region of 2 pages, data region 1 page long, and an initial bss region of 3 pages.



Why the MAP_DENYWRITE flag?

Both text and data regions are mapped with the MAP_DENYWRITE flag set. The reason for this is the following situation: Let's say a program is running from a particular a.out file, and then someone comes along and re-writes that executable (e.g. it is being recompiled or reinstalled). However, the running program is still mapped to the file, so what gets paged-in would be a mixture of the original contents of the file and the newer contents. Clearly this won't work...it amounts to a corruption of the program. The DENYWRITE flag will prevent any kind of write access to the executable file, including truncation. It will fail with the error `ETXTBUSY`. However, one can unlink or rename the executing a.out file, and install a new version with the same name. Since the mapping is based on inode number, not pathname, the currently running process (or processes) will continue to use the old executable, while any new exec's would reference the new

executable (since they evaluate a pathname, as if by **open**).

Those responsible for the development of the Linux kernel removed MAP_DENYWRITE from the list of flags that can be specified by the mmap system call (the flag is accepted but silently ignored). The reasoning went like this: User A which has read-only access to a file F could establish a read-only mmap mapping with DENYWRITE, even though they do not have write permission on F. This can create a denial-of-service attack, in that another user B, who does have write permission, would now be prevented from writing to the file. When the kernel sets up a new address space during exec, it is using internal versions of mmap, and is thus not actually using the flag MAP_DENYWRITE, but is specifying VM_DENYWRITE directly.

fork & clone interactions with Virtual Memory

We have seen that the `fork` system call is really a special case of the `clone` system call (on Linux) with the `clone_flags` set to all-zero. When the `CLONE_VM` clone flag is present, the child task will share all aspects of the virtual address space with the parent. This is accomplished within the kernel by having the `mm` field of the `task_struct` of the child simply be copied from that of the parent, so both tasks are sharing the `mm_struct`. This also means that there is one PGD for parent and child. Any system call affecting the virtual memory regions (`vm_area_structs`) in the child has the same effect on the child and vice-versa.

However, in `fork`, the `CLONE_VM` is 0. This implies a a deep copy of the `struct mm_struct` in the parent to form the memory map of the child. This means that each `vm_area_struct` is copied, forming a new linked list attached to the new `mm_struct` of the child. The page tables are also deep-copied. The child gets a new PGD which is a copy of the parent's, and any PUD, PMD and PTs are also recursively deep-copied. Each PTE in the parent is thus duplicated to the child, and if the page in question is Present, the reference count of the corresponding struct page is also incremented.

Regions that are writable and MAP_PRIVATE, including the data, bss and stack regions of any process, are set up for **copy-on-write**. This is described further in the section with that name.

The kernel also, when making copies of the page tables for the child, clears the PTE A(ccessed) bits in the child PTE's only. The parent is unaffected. This is because that virtual page in the child has not actually been accessed yet. The child is about to come to life and none of its virtual pages has been accessed!

The Page Frame Pool

Now that we have made an initial exploration of how the kernel manages the virtual

address space of processes, let's take a look from the standpoint of physical memory. We'll then put it together and discuss the mechanisms for paging-in and paging-out.

A computer system has a certain amount of physical RAM. Some of that is taken up statically by the kernel, i.e. kernel text, data and bss, and system control structures that must exist at a certain physical address, such as the Interrupt Descriptor Table. There are additional demands for physical memory:
• Dynamically allocated kernel data structures, e.g. the task_struct, kernel-mode stacks, in-core inodes and directory entries, pending signals, loadable kernel modules, and shared kernel/hardware data structures such as page tables.
• User process address space. We've seen that this can be broken down into anonymous regions, which are not persistent, and file-mapped regions where resident pages correspond to specific portions of specific, persistent files.
• I/O buffers. In some cases the I/O buffer corresponds to a portion of a file, but in other cases (e.g. network packet buffers) it does not.
• Filesystem block caches: Filesystems sometimes need to cache disk blocks which are not the actual contents of a given file, e.g. directory entries.

The demand for page frames ebbs and flows throughout the life of the system. At certain times, there may be abundant page frames which are empty and uncommitted, and therefore available for immediate allocation. At other times, the page frame pool may be depleted, causing pressure for page frames to be freed up to satisfy new demand. The kernel, through the Page Frame Reclamation Algorithm (PFRA), always tries to keep a reasonable number of free page frames, so that an allocation request does not fail or encounter excessive delay. We'll study the PFRA shortly.

We will now consider how page frames are allocated by kernel routines, and how those page frames may be subdivided for efficient allocation of smaller objects.

Sidebar: Addressing Issues on X86-32

Recall that the kernel, in a 32-bit X86 environment, uses the last 1GB of virtual address space for itself, leaving the first 3GB to user-mode. Also recall that all kernel virtual addresses are shared, i.e. there is not a separate kernel virtual address space for each process or component of the kernel. Therefore, it would seem to be impossible for the kernel to have more than 1GB of internal data structures.

This is addressed through an interesting strategy. Linux further subdivides both physical and kernel virtual space so that the first 896MB of physical page frames map **directly** to the first 896MB of the last GB of virtual address space. This greatly simplifies kernel code as a physical page number can be used directly to compute the virtual address and vice versa. The last 128MB of the kernel's virtual address space is used to establish **temporary mappings** to physical pages which are above 896MB. The kernel calls these two areas of physical memory **zones**, the former being the NORMAL zone and the latter

the HIGHMEM zone. This klugery is not required for 64 bit architectures, where the kernel has ample virtual address space to make a direct mapping to all physical pages.

When allocating memory for internal data structures, the kernel uses the pool of pages in the NORMAL zone. Since user processes are not confined to the shared 1GB virtual address space, their pages always get pulled from the HIGHMEM zone first, and then, with reluctance, from the NORMAL zone if there is insufficient free memory in the HIGHMEM zone.

This kluge is not needed in 64-bit mode, since there is ample virtual address space.

Page Descriptor Table

The kernel maintains a single, contiguous array of **page descriptor** structures, each of which represents one page frame:

```
struct page {
          unsigned long flags;                              /* bitwise flags */
          atomic_t _count;                        /* usage counter */
          atomic_t _mapcount;                     /* how many PTE mappings to this page */
          /* This is a GCC compiler extension known as a blind union */
          union {
                    unsigned long private;                  // buddy order, et al
                    struct address_space *mapping;          // reverse mapping
          };
          pgoff_t index;                          /* offset of page within mapping*/
          struct list_head lru;                   /* free/active lists */
          void *virtual;                          /* kernel virt address */
};
```

The page descriptor table, called `mem_map` (another term often used in UNIX kernels other than Linux is "core map") consumes a small amount of memory overhead. Specifically, on a 32-bit system, each `struct page` is 32 bytes long, so the overhead is 32/4096 or 0.7%. The page frame number is implicit by the index of the page descriptor within the `mem_map` array. For any given page, if it falls within the low-memory ("NORMAL") zone, the kernel can easily figure out the direct-mapped virtual address. Otherwise, if the page is currently mapped into kernel memory space, the `virtual` field says where.

A given page frame is either mapped (there is at least one PTE in somebody's virtual memory area, including the kernel's, that points to it) or it is free. To save space, some fields of the `struct page` are overloaded with mutually exclusive values. For example, the blind union `private / mapping` contains either information about the free page or information about the reverse mapping if it is mapped. The `_mapcount` field will be 0 if the page is free and >0 if the page is mapped somewhere. The confusingly named `_count` field is a temporary usage counter that is incremented when certain memory management routines are doing something with that page, and is used as a form of locking to prevent pages from disappearing while kernel code is playing with

them. The `flags` field contains bitwise flags of the form `PG_xxxx` which encode various status values. For example, the `PG_locked` flag is set when an I/O operation is in progress on that page. The usage of the other fields in the `struct page` will be discussed during the rest of this unit.

Address space release (munmap)

When a `vm_area` is destroyed via the munmap operation, either via the system call or the internal version which is called during exec or exit, the kernel visits the page tables which are spanned by that region. For any PRESENT PTE, the PFN gets us to the struct page. The `mapcount` is decremented. If this was the last mapping to the page and the count is now 0, the page is moved to the free list. This process is recursive. If an entire page table worth of PTEs have been freed up, the page table itself can be returned to the page frame pool. Finally when the entire process address space is free, the PGD is freed.

If any PTEs indicate that this was an anonymous page that had been swapped out, the kernel updates its *swap map*, decrementing the reference count on that swap slot. When the reference count is 0, the swap slot is free to be re-used.

Upon process termination, the kernel calls an internal version of munmap for each vm_area_struct in that process's address space. Therefore it is never necessary for user code to explicitly free memory prior to program exit!

Page Allocator -- Buddy System

The basic kernel page frame allocator is based on asking for one or more contiguous pages. Although most of the time kernel routines are asking for just one page, sometimes the kernel needs a rather large buffer which spans many pages, e.g. when setting up a Direct Memory Access transfer to/from an I/O device.

In general, memory allocation routines (e.g. `malloc()` in user mode) are faced with a tradeoff between space efficiency and time efficiency. If there is a request for say 7 contiguous "units" of memory, it is desirable in terms of space efficiency to satisfy that request with exactly 7 units. However, as memory is allocated and freed, **fragmentation** arises. There may be 3 units free, then 1 occupied unit, then 4 more free units. Even though there are 7 free units, they are not contiguous and thus do not satisfy the request.

In some allocation settings, e.g. on-disk filesystems, it is possible to de-fragment the system by swapping storage locations so as to group clumps of free units together. This is not possible in the C programming environment because once memory has been allocated, its raw address has been passed back as a pointer and there is no way to track down all references (including offset references) to that memory region, and thus no way to change the address of the region.

Given that fragmentation is unavoidable, the next best thing in terms of space efficiency is, given a request for N contiguous units, satisfy it with the smallest available contiguous

region of size M units, M>=N. Now this leads to a time efficiency issue: how long will it take to search the list of available memory units to find this "best fit". In the case of the kernel, because memory allocation is a frequent and critical operation, it is desirable that this time be a constant one, not one which grows as the number of distinct free memory units increases because of fragmentation. Therefore, the Linux kernel makes a compromise, and adopts the well-known "buddy system" algorithm to give fairly decent space efficiency and lookup time which, although not strictly constant-time, has a small and fixed upper bound.

The "buddy system" algorithm maintains, for each of the memory allocation zones (e.g. NORMAL and HIGHMEM) an array of 11 pointers. The [0] element of this array points to a circular doubly-linked list of page descriptors for page frames which are free but not contiguous with other free pages. The [1] element points to such a list for groups of 2 contiguous pages. The [2] element does the same for groups of 4 contiguous pages, and so on up to the [10] element which collects groups of 1024 contiguous pages, i.e. 4MB. These lists are chained through the `lru` field of the page descriptor (which has other uses when the page frame is not free) and furthermore the `private` field of the first page descriptor in a contiguous chunk holds the order of that chunk. Subsequent page descriptors in the chunk are not part of the list but their existence is inferred by their position in the `mem_map`.

When a routine in the kernel wants say one page frame, the buddy system tries to take that from the [0] order list of free page frames. If there are none, it grabs a chunk of two page frames from the [1] list and splits it, handing one back to the caller and moving the other to the [0] list. If the [1] list is empty, this process continues moving up the ladder, e.g. taking a clump of 4 contiguous frames, returning 1 to the caller, and splitting the remainder into a single free frame and a pair of frames.

When a page frame or contiguous group of frames is de-allocated by the kernel to be placed back in the free pool, the "buddy system" checks to see if the freed page(s) is/are a contiguous "buddy" of another free page. If so, they are merged to form a higher-order contiguous region. This process can continue up to order 10, i.e. coalescing into a chunk of 1,024 free pages.

Note that the buddy system trades CPU time efficiency for absolute memory efficiency, by introducing an additional restriction. A group of pages on the, e.g. [2] list, i.e. a group of 4 contiguous pages, will always start with a page number divisible by 4. If say page frame numbers 1024-1027 inclusive are free, these can be used to satisfy a request for 4 contiguous pages. However, if 1025-1028 inclusive are free, this does not satisfy the request, because the start of the free area, 1025, is not a multiple of 4. Instead, the buddy system views this as a page of order 0 (#1025), two pages of order 1 (1026-1027) and another page (1028) of order 0.

Given a page frame number `a`, its order-`N` buddy with page frame number `b` therefore satisfies the expression:

`b==a ^ (1<<N)`
Any page frame number `a` of order-`N` has a parent of order `N+1` which lives at:
`p=a & ~ (1<<N)`

These simple bit-arithmetic relationships allow the deallocation and conglomeration algorithm to be performed very quickly, without extensive data structures manipulation and resulting memory accesses. The penalty is increased fragmentation.

Slab Allocator

The kernel subdivides page frames using a system called the **Slab Allocator** for small objects (e.g. allocating a new `task_struct`). This system is fairly complicated and will not be fully described here. The kernel grabs page frames and uses them as slabs, storing an internal descriptor at the beginning of the slab to track how the page frame or frames is being subdivided. Because the kernel is a tightly-controlled piece of code and the type of data structures which will need to be allocated and released frequently is understood in advance, the slab allocator can be optimized so that it can quickly allocate and free objects of a given type.

When the kernel requires space for a new object of a certain type, it consults the slab associated with that type. The slab descriptor bitmap quickly determines if there is a free slot open, and if so where. If there are no free slots, the slab allocator goes back to the page frame allocator and asks for an additional page (not necessarily contiguous) which is subdivided into slots.

Likewise, de-allocation of an object allocated through the slab allocator means simply noting in the descriptor that the slot is free. If all slots in the page frame are free, the page frame itself is released to the free pool.

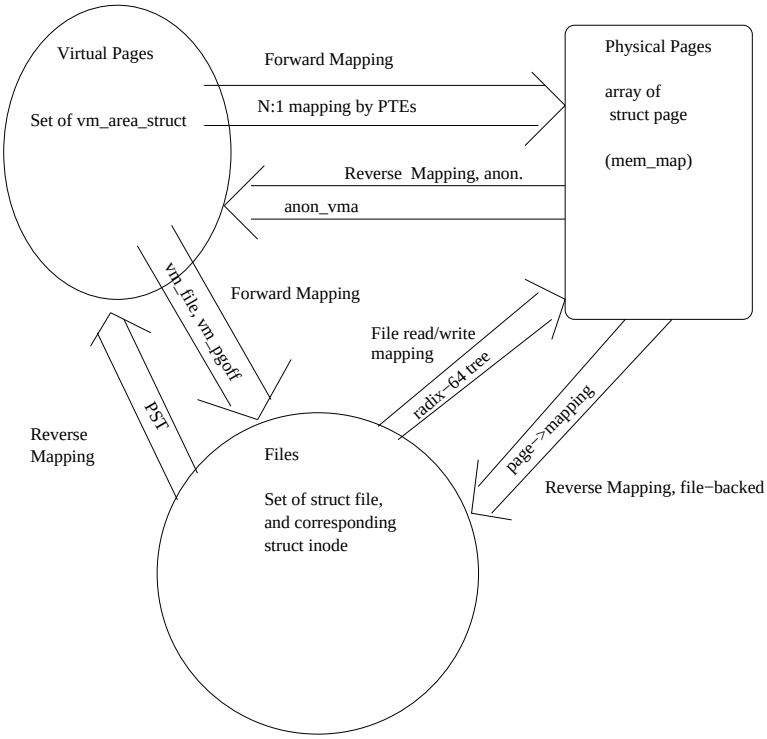
The slab allocator is both space efficient (the overhead is just 1 bit in the bitmap per object, plus a few bytes for the overall slab header) and time efficient (no convoluted data structures). This is important because small data structures (such as `struct file`) are frequently being allocated and released by the kernel.

The Trinity of Memory and Addressing

We can look at randomly-addressable things from three standpoints:
• Virtual pages: The set of all virtual memory regions in all address spaces along with their associated properties
• Physical pages (page frames)
• The file and disk I/O system. Each file is randomly-addressable and portions of the file may be cached in physical memory. In addition, a file or a raw disk partition may be used as **swap space** to hold anonymous virtual pages when they have to be paged-out.

There are relationships in both directions between each of these three views, which are managed by various kernel data structures. The diagram near the end of this unit shows all of these together and can be quite intimidating. The relationships, or mappings, are as follows:

• **Forward Mapping**: Given a virtual page (i.e. a specific virtual memory area and offset within that area), what page frame holds it or, if no page frame holds it, how do we find the canonical contents of that page frame? The page table data structure, which ultimately delivers us to the PTE, determines the page frame. This is an N:1 mapping because a given page frame can be shared among multiple virtual memory regions (e.g. MAP_SHARED mmap). If the PTE is empty, the vm_file and vm_pgoff fields of the vm_area_struct point us in the right direction for finding the corresponding chunk of the file which backs up this virtual memory region.
• **Reverse Mapping**: For a given page frame, find all PTEs that map to it. Different data structures are used for reversed mapping depending on whether the given page is anonymous or file-mapped.
• **File read/write (I/O) Mapping**: When a given page-sized chunk of a file is wanted for a read or write system call, determine if that chunk is already cached in memory and if so, in what page frame?



These mappings form a circle, and thus it is difficult to present this material which has circularities in our comprehension prerequisites. We must jump into it at some point, so we will start with Forward Mapping!

Page Fault Handling -- Forward Mapping

The page fault exception is used by the CPU to indicate an access to an invalid or untranslated virtual address. However, this is usually not a real problem. The kernel enjoys handling page faults, and uses page faults to implement a number of features, one of which is **Demand Paging**: page frames are not actually allocated, and I/O operations (if applicable) are not actually initiated, until a process establishes the need to do so by attempting to access the corresponding virtual page. This is a significant speed and memory usage improvement. Without demand paging, programs would take a long time to load and start, because the entire program code (including all those libraries) and data would have to be read in from disk. With Demand Paging, only the code and data which is actually used consumes resources.

The kernel's page fault handler knows:
- The identity of the faulting process (`current` global variable)
- The attempted virtual address.
- The type of access (read/write).
- Whether the virtual page in question had a valid PTE
- The registers at the time of fault (including user/supervisor flag).

The page fault handler can have one of two general outcomes: Either the page fault is resolved, resulting in the establishment of a valid PTE (which has the appropriate R/W/X permissions for the access being attempted) pointing to a valid page frame with the correct data in it, or the page fault can not be resolved, resulting in the delivery of a SIGSEGV, SIGBUS or SIGKILL to the process. Let's take a look at the page fault handler code:

```
/* Please note: All of this code is fictional, but based on a true story.
 * It is a heavily simplified and cleansed condensation of actual Linux
 * kernel source code for page fault handling.  Many of the variable and
 * function names have been changed to protect the innocent.  The interested
 * reader is invited to begin at /usr/src/linux/mm/memory.c           */

/* handle_pagefault_usermode() is the fictitious name of the fault handler
 * which is invoked when the MMU gives a page fault and the kernel determines
 * that the fault happened while in user mode.  Because this is a synchronous
 * entry into the kernel, the current variable is that of the process which
 * caused the page fault.  Upon return from this function, we will return to
 * user mode and either the faulted instruction will be re-started or a
 * signal handler will be invoked                                      */
handle_pagefault_usermode(void *addr, int access_type)
{
 struct vm_area_struct *vma,*growsdown;
 void *new_addr;


        if (addr >= KERNEL_MEMORY_START)       // Attempt to access kmem from user
                force_sig(current,SIGSEGV);
        /* Note: the force_sig function posts the requested signal to the
         * specified task.  If that signal is currently being ignored or
         * blocked, the setting is changed to SIG_DFL which usually terminates.
         * We can't ignore or defer the signal because the current instruction
         * has faulted and therefore continued execution makes no sense */

        growsdown=NULL;
        /* Is the fault address inside of a defined region? */
        for(vma=current->mm->mmap;vma;vma=vma->vm_next)
        {
                if (addr>=vm->vm_start && addr<vm->vm_end) break;
                if (vm->vm_flags & VM_GROWSDOWN &&
                          vm->vm_start-addr < STACK_FUZZINESS &&
                          regs->esp <= addr + STACK_FUZZINESS)
                                  growsdown=vma;
        }
        if (!vma && growsdown)
        {
                new_addr= addr&~(PAGE_SIZE-1);         // round down to page boundary
                vma=growsdown;
                /* There are additional checks, e.g. will the newly grown
                 * stack exceed resource limits, or collide with another area */
                vma->vm_start=new_addr;
        }
```

```
          if (!vma)          // No valid region found
                  return force_sig(current,SIGSEGV);

          /* Inside of a region, may have been a protections fault */
          if (!protections_ok(vm->vm_flags,access_type))
                  return force_sig(current,SIGSEGV);
          /* We have now proved that the user's access is valid,
           *       let's resolve the page fault */
 try_again:
          vp_addr = addr & ~(PAGE_SIZE-1);   // Start of virtual page w/ fault
          switch (handle_mm_fault(current->mm,vma,vp_addr,access_type))
          {
            case VM_FAULT_OOM:
                  if (current->pid == 1) {     // Oh no, can't kill init!
                          yield();             // Let PFRA work
                          goto try_again;
                  }
                  printk("VM: Killing process %d0,current->pid);
                  do_exit(SIGKILL);
                  /*NOTREACHED*/
            case VM_FAULT_SIGBUS:
                  force_sig(current,SIGBUS);
                  break;
            case VM_FAULT_MAJOR:
                  current->maj_flt++;
                  break;
            case VM_FAULT_MINOR:
                  current->min_flt++;
                  break;
          }
}

handle_mm_fault(struct mm_struct *mm,struct vm_area_struct *vma,
                                      void *addr,int access_type)
{
 pte_t **pgd,**pud,**pmd,**pt;
 pte_t *ptep,pte;
          pgd=mm->pgd;
          /* PGD is always allocated.  Make sure PUD/PMD/PT are allocated */
          /* corresponding to the faulted address.  The xxx_alloc functions */
          /* return a pointer to the next table, possibly allocating a new */
          /* page frame for the table and installing a pointer to it in the */
          /* enclosing table.  Physical addresses are mapped by the kernel */
          /* to virtual addresses for this temporary purpose; this mechanism */
          /* is very arch-specific and beyond the scope of this class */

          if (!(pud=pud_alloc(mm,pgd,addr))) return VM_FAULT_OOM;
          if (!(pmd=pmd_alloc(mm,pud,addr))) return VM_FAULT_OOM;
          if (!(pt=pt_alloc(mm,pmd,addr))) return VM_FAULT_OOM;
          /* Compute offset of PTE within Page Table */
          /* On X86-32, PAGE_SHIFT is 12 bits and PAGE_MASK is 0x000003FF */
          /* On X86-64, PAGE_MASK becomes 0x000001FF */
          ptep= &pt[((unsigned long)addr >> PAGE_SHIFT) & PAGE_MASK];
```

```
        pte= *ptep;

        if (!pte_present(pte))                  // PRESENT bit is 0
        {
                if (pte == 0)                   // First time access or file-backed
                {
                        if (vma->vm_file)  // File-mapped
                                return pagein_from_file(mm,vma,addr,ptep,access_type);
                        else                            // Anonymous
                                return pagein_anon(mm,vma,addr,ptep,access_type);
                }
                /* PTE encodes swap loc for paged-out anon page */
                return pagein_from_swap(mm,vma,addr,pte,ptep,access_type);
        }
        /* PTE is present, must be a COW protection fault */
        if (write_attempt(access_type))
        {
                pfn_t pfn;
                pfn=((unsigned long)pte >> PFN_SHIFT) & PFN_MASK;
                /* This next condition is common after fork-exec */
                if (mem_map[pfn]._mapcount==1)
                {
                        /* Some code elided to check if MAP_PRIVATE page
                         *       that had been marked down for COW but had
                         *       only a single mapping, needs to adjust
                         *       the radix-64 tree to invalidate           */

                        *ptep|=PTE_WRITE;
                        return VM_FAULT_MINOR;
                }
                void *new_pf;
                /* Allocate a new page frame and provide a valid temporary */
                /* virtual address so we can access it */
                if (!(new_pf=pageframe_alloc(1))) return VM_FAULT_OOM;
                memcpy(map_pf_to_va(new_pf),addr,PAGE_SIZE);
                *ptep|=PTE_WRITE;
                return VM_FAULT_MINOR;
        }
}

/* This function is only called for first-time access to an anon virt page */
pagein_anon(struct mm_struct *mm,struct vm_area_struct *vma,void *addr,
                pte_t *pte_p,int access_type)
{
  void *new_pf;
  pfn_t pfn;                    // 32 bit or 64 bit
  unsigned pte_bits;
        pte_bits=vma->vm_page_prot | PTE_PRESENT;
        if (read_attempt(access_type))
        {
                /* For first-time read, we can grab a shared COW 0 page */
                new_pf=global_shared_zeropage_paddr;
                if (!new_pf) return VM_FAULT_OOM;
```

```
                pte_bits &= ~PTE_WRITE;      // Not writable -- COW
        }
        else
        {
                new_pf=pageframe_alloc(1);
                /* map_pf_to_va provides a (temporary) virt addr */
                bzero(map_pf_to_va(new_pf),PAGE_SIZE);
        }
        pfn=(unsigned long)new_pf>>PAGE_SHIFT;
        mem_map[pfn]._mapcount++;    // Update core map
        /* Fill in the PTE.  PFN_SHIFT is the bit pos of the PFN in the PTE */
        *ptep= (pfn<<PFN_SHIFT) | pte_bits;
        update_rss(ANON);  // Update rss, highwater, etc. stats.
        return VM_FAULT_MINOR;
}

pagein_from_file(struct mm_struct *mm,struct vm_area_struct *vma,void *addr,
                pte_t *pte_p,int access_type)
{
 void *new_pf;
 struct page *new_pf;
 pfn_t pfn;
 unsigned long pte_bits;
 int rc;
        /* Determine offset within file, in units of pages */
        va_offset=(unsigned long)(addr-vma->vm_start) >> PAGE_SHIFT;
        file_offset=va_offset+vma->vm_pgoff;
        /* See if that page is already resident in memory */
        rc=VM_FAULT_MINOR;
        if (!(new_pf=file_address_space_search(vma->vm_file,file_offset)))
        {
                /* No, need to allocate a fresh page frame */
                if (!(new_pf=pageframe_alloc(1))) return VM_FAULT_OOM;
                pfn=(unsigned long)new_pf >> PAGE_SHIFT;
                new_pf= &mem_map[pfn];
                /* Ask the filesystem module to initiate a disk read
                 * operation, depositing the result in physical page new_pf.
                 * This puts the task into a non-interruptible SLEEPING state
                 * until the I/O operation completes (or fails          */
                if (!do_file_readpage(vma->vm_file,file_offset,new_pf))
                {
                        pageframe_free(new_pf);
                        return VM_FAULT_SIGBUS;
                }
                rc=VM_FAULT_MAJOR;
        }
        pfn= new_pf-mem_map;         // PFN is implicit by index within array
        new_pf->_mapcount++;         // Here comes new PTE
        pte_bits=vma->vm_page_prot|PTE_PRESENT;
        *ptep=pfn<<PFN_SHIFT|pte_bits;
        update_rss(FILE);
        return rc;
}
```

```
pagein_from_swap(struct mm_struct *mm,struct vm_area_struct *vma,void *addr,
                  pte_t pte,pte_t *pte_p,int access_type)
{
 void *new_pf;
 pfn_t pfn;
 int pte_bits;
 unsigned long swap_area,swap_slot;
          swap_area=(pte&SWAP_AREA_MASK)>>SWAP_AREA_SHIFT;
          swap_slot=(pte&SWAP_SLOT_MASK)>>SWAP_SLOT_SHIFT;
          if (!(new_pf=pageframe_alloc(1))) return VM_FAULT_OOM;
          /* This operation results in a non-interruptible SLEEP */
          if (!read_from_swap(swap_area,swap_slot,new_pf))
          {
                    pageframe_free(new_pf);
                    return VM_FAULT_SIGBUS;
          }
          /* Some more complexity elided to deal with a SHARED, ANON
           * page that had more than one mapping and had been swapped out */
          mark_swap_slot_free(swap_area,swap_slot);
          pfn=(unsigned long)new_pf >> PFN_SHIFT;
          pte_bits=vma->vm_page_prot|PTE_PRESENT;
          *ptep=pfn<<PFN_SHIFT|pte_bits;
          mem_map[pfn]._mapcount++;
          update_rss(ANON);
          return VM_FAULT_MAJOR;
}
```

Prior to the code seen above, the kernel examines the faulting address and the user/supervisor mode bit. If the faulting address is inside the kernel's reserved virtual address range AND we were in user mode, then a SIGSEGV is sent to the process, no questions asked. If we were in supervisor mode, then the page fault happened inside kernel code. The kernel tries to avoid any situations where this happens, as the code for handling this is messy. We won't be considering kernel-mode page faults in the Linux kernel any further.

Now that we have determined that it was a user-mode program that got us into the page fault handler, and that the access was not to the kernel reserved address space, we have to see if this was a legitimate virtual memory access. This is done by examining the virtual address space description contained in the list of `vm_area_structs` to see if the faulting address is inside of an existing valid virtual memory region. If it is, we then need to determine if there has been a protection violation (access attempt mode doesn't match PTE mode). There are two exceptions to this: stacks and copy-on-write.

Stacks

When the stack grows e.g. through a push instruction, the stack pointer is decremented first and then the write takes place. If the stack pointer is now lower than the current start

of the stack region, a fault will be incurred. If the faulted address is "close" to the stack pointer (an allowance is made for certain X86 instructions which access memory on the stack first, then adjust the stack pointer) and if the memory region just above the faulted address has the GROWSDOWN property, then the stack region is expanded by one page toward low memory addresses ("down") as if by the `mremap` system call. There are checks performed to make sure this doesn't exceed the stack size ulimit resource limit, or cause the stack to collide with another region. This automagical stack growth is quite essential to transparent program execution.

This adjustment of the `vm_start` address does not by itself resolve the page fault. The logic flow now continues, the same as a page fault in any other valid memory region.

Copy-On-Write (COW)

There are several cases where the kernel must logically make a copy of a page. One common example is during `fork()`. Another is when a file is mapped MAP_PRIVATE, a page is initally faulted in for reading from the file, and then it is written to, requiring that the association with that part of the file be broken.

If the kernel actually allocated new page frames and copied, byte-for-byte, the entire contents of each page frame of the region being copied, that would be a terrible performance penalty. Instead, the kernel continues to point both virtual mappings to the same page frame, but marks down the PTE entries in question to disallow write access. In the case of a fork, the W bit is turned off in both the original parent PTE and the new child PTE. The `vm_page_prot` field of the `vm_area_struct` is modified so that any new PTE mappings in the shared memory regions (which would otherwise have the PTE_WRITE bit turned on because they have VM_WRITE) caused by a page fault are installed with the W bit off, which will allow the page frame to be shared until somebody tries to write to it.

Likewise in the case of a MAP_PRIVATE file mapping, the `vm_page_prot` of the associated region does not have the PTE_WRITE bit on. Pages which are faulted in by reading are installed with read-only PTEs. If the process later tries to write via the memory mapping, this causes a page fault which alerts the kernel to break the mapping for that page and make a private copy. Both of these COW examples have the same implementation and effect even though the purposes differ.

Regions that have the VM_SHARED property (i.e. MAP_SHARED memory mapped regions) are **not** copy-on-write since by definition they are shared after a fork. The PTEs are not modified and the parent and child will continue to map to the same page frames for all pages in this shared region.

As long as none of the processes with the C-O-W mapping attempts to write, there is no need to do anything further. They will each see the correct data which are unchanged from the point of forking, in the first example case, or from the original contents of the

file, in the second case. As soon as any process tries to write, it will incur a page/protection fault. The kernel recognizes this COW situation because the faulting address is valid, the virtual memory area flags say we should be allowed to write to this area, AND there was a PTE PRESENT for that address.

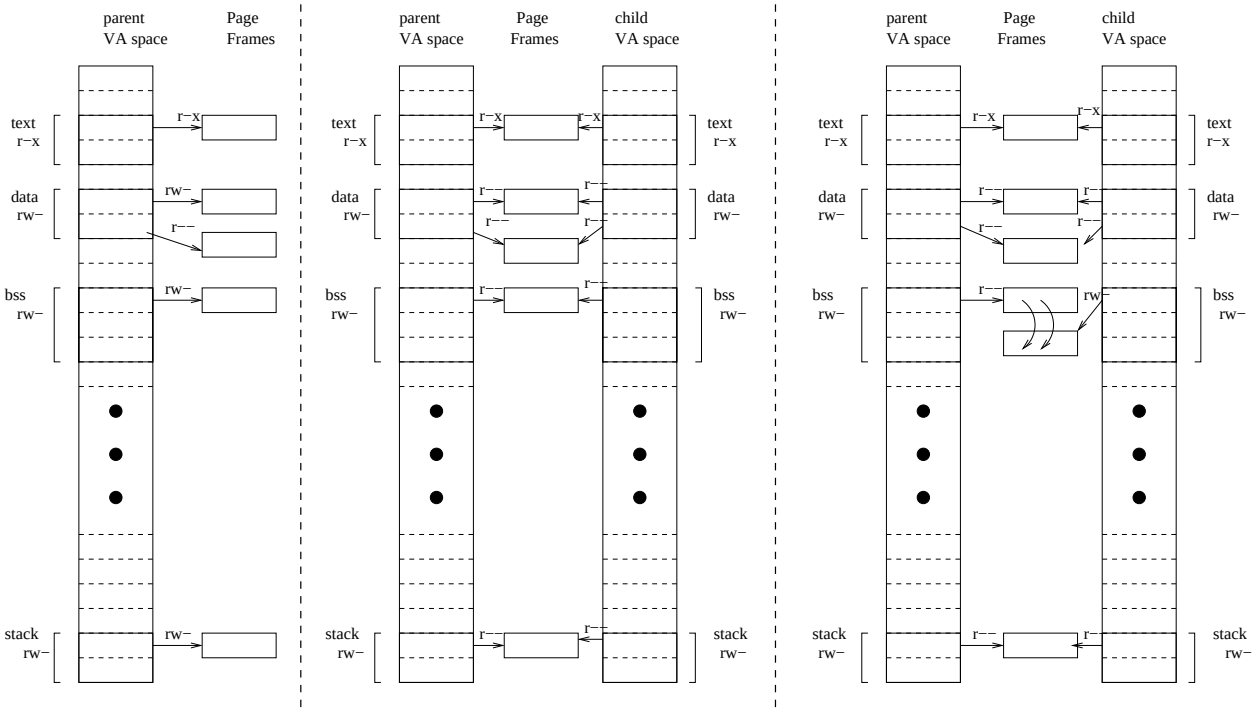
To resolve the COW fault, the kernel grabs a new free page frame, copies to contents of the existing (shared) page frame to the new frame, and then updates the PTE of the faulting process to point to the new frame, turning on the PTE Write Enable bit at the same time. The other process(es) sharing that mapping continue to see it read-only.

One important optimization on the above happens when a process forks and then either the parent or the child (but usually the child in the common UNIX programming idiom) execs. Then the shared memory region is unmapped from one of the processes. As we have seen, the kernel maintains a core map data structure giving the state of each page frame, and can track how many references there are to any given page frame. So the unmapping results in the reference count going back down to 1. When the kernel sees this condition, it does not have to allocate a new page frame and memcpy. It merely changes the PTE to turn the write enable bit back on.

Note that the MAP_PRIVATE mapping of a file creates a hybrid region. Within that region, some virtual pages may be mapped in read-only, and still correspond to the original file, while others that have been dirtied are effectively anonymous pages. They will have to be written back to *swap* which we will define shortly, if their page frames get reclaimed. We'll see how the `mapping` field of the page descriptor is used to distinguish, on a page frame by page frame basis, whether a given page frame is still file-backed or is anonymous.

There is another example of copy-on-write: if we have an anonymous region and a new page is demand-paged via a read mode page fault. We need to see a page full of 0 bytes. But if we are only reading it, why do we need a distinct page frame full of 0s for each such "virgin" virtual anonymous page? We don't! The kernel keeps a single page frame full of 0s and shares it, using the C-O-W mechanism to split the sharing when anyone tries to write to it.

In the illustration below, before the fork, the parent process has demand-paged one page of text, one page of bss, one page of stack, and two pages of data. It wrote to the first data page but the second page has only be read, so its PTE is still r--. After the fork (second panel), all PTE W bits are off. In the third panel, the child attempts to write to the bss page. This causes copy-on-write. Note that the parent's PTE is not affected.



Invalid Addresses and SIGSEGV

If, however, the faulted address is outside of all regions and is not associated with stack growth, or if the attempted access is a violation of the regions protections, a signal `SIGSEGV` is forcibly posted to the current process. "Forcibly" means the kernel doesn't care if the process has this signal number blocked or ignored: The user program attempted to access an illegal virtual memory address, and there is no point in letting it continue on its current path. The address will not magically become valid, and therefore the faulted instruction can never be satisfied and would loop forever. The signal then terminates the process.

If the signal has a handler, but the signal is currently being blocked, the signal is unblocked, the handler is reset to SIG_DFL, and once again the signal kills the process. The reasoning is that a handler should not be invoked if the corresponding signal is being blocked.

It is, however, permissible for the signal to be *handled* as that will force execution to jump to another, presumably less flawed part of the program. This technique is often used in complex, modular applications (such as the open-source image editor *The Gimp*) to isolate a flawed module and prevent it from crashing the entire program and possibly leading to a loss of data.

Benign Page Faults / Paging-in

Page faults which are within an existing region and which are not the result of protection violations are therefore not the result of any program error, and need to be resolved by establishing a valid forward mapping PTE that is pointing to a valid physical page frame which contains the correct contents corresponding to the virtual page. This page fault resolution mechanism can have one of four mutually exclusive outcomes:

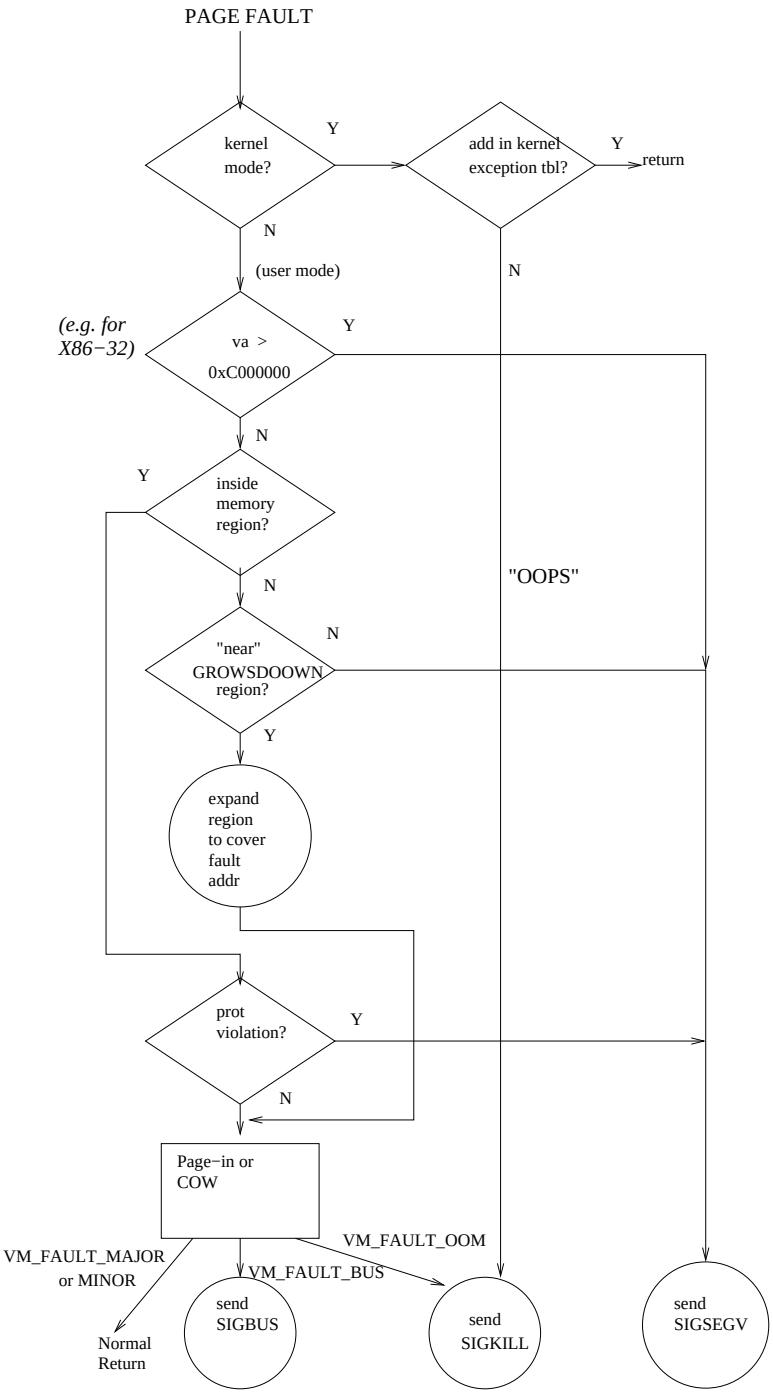
• `Minor fault`: The page fault is resolved quickly without requiring a disk I/O operation or putting the process to sleep. E.g. demand-paging new bss pages (just allocate a page, zero fill and return), paging-in file-mapped pages which are still cached in memory, or breaking up a Copy-on-Write shared page frame. Minor faults are good.

• `Major fault`: It was necessary to sleep pending an I/O operation to satisfy this fault. Major faults are not necessarily evil (e.g. demand paging an executable file from disk) but we would like to minimize their number.

• `Bus Error`: The kernel was unable to page-in the page because of an error other than being out of memory. E.g. the page is mapped to a disk file but an I/O error occurred trying to access the disk, or the mapping is no longer valid because the file has been truncated. A SIGBUS will be delivered to the process.

• `Out of Memory`: The kernel ran out of memory to allocate a free page frame or an internal data structure. This is very bad. Under ordinary circumstances, the kernel tries to prevent this from happening by maintaining a pool of free memory. If this does happen, the kernel has no choice but to terminate the current process. See also discussion below on "over-commit".

The Linux page fault handling algorithm is summarized by the following simplified flowchart:



On-demand creation of Page Tables

Recall that the structure of the page table is multi-level and sparse. Only the top-level Page Global Directory is pre-allocated. The kernel uses **lazy allocation** in creating the

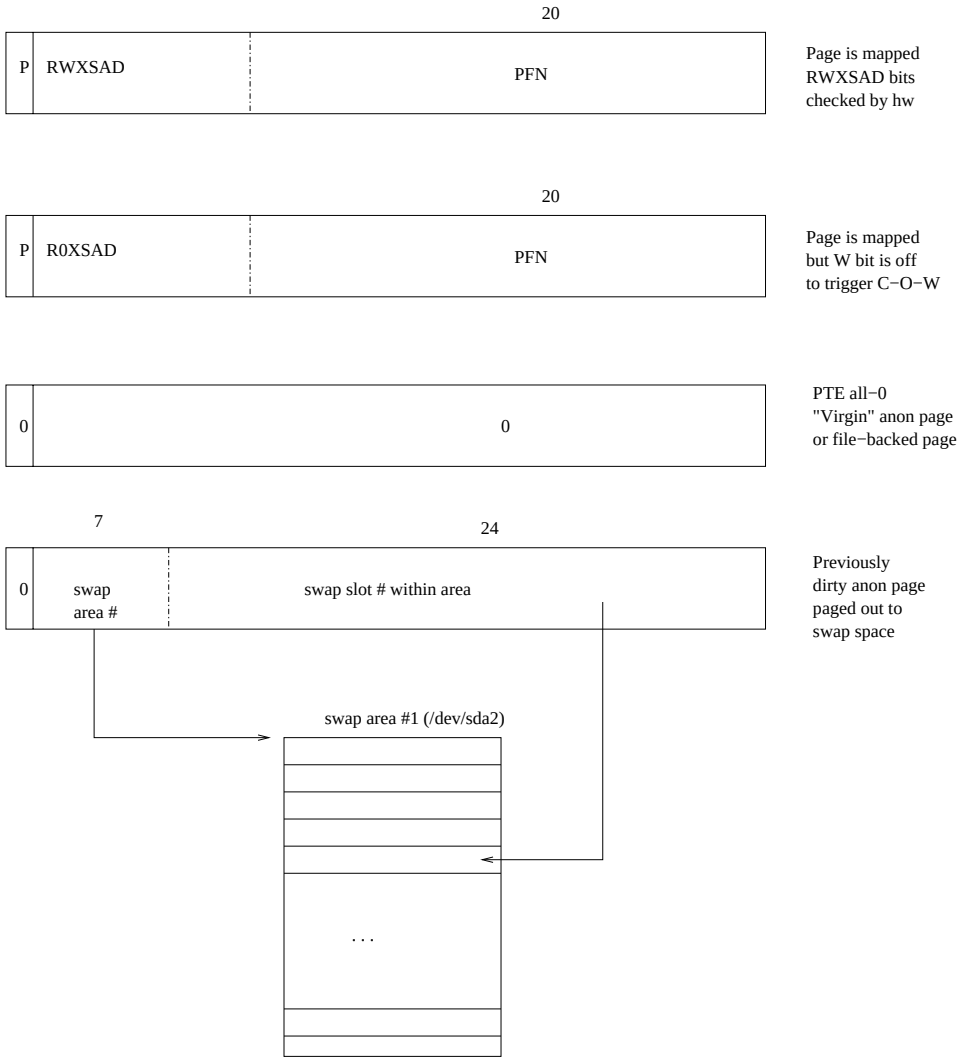
page upper directories, page middle directories, and page tables. Therefore, at this time of page fault resolution, the kernel must (recursively) allocate the directories/tables mapping the faulted virtual memory address, if they are not already allocated. When new page tables or page directories are allocated, they are set to 0, meaning none of the entries are viewed by the hardware as PRESENT yet. The kernel then installs new Page Directory or Page Table Entries as needed.

Paging-in from where?

To complete page fault resolution, the kernel must locate the canonical copy of the data corresponding to the virtual page which is faulted. In brief, this can fall into one of the following cases:
• The virtual page is file-mapped.
• The virtual page is anonymous and not previously written to.
• The virtual page is anonymous and previously dirtied, and swapped out.
• The virtual page is already present in physical memory, and this is a copy-on-write.

The kernel can determine which case applies by looking at the bits of the PTE corresponding to the faulted address, and by looking at the attributes of the corresponding `vm_area_struct`.
• The PFN field is all zero and the Present bit is 0, and the `vm_area_struct` indicates that this region is mapped to a file. The corresponding page-sized chunk of the file might already be cached in a page frame because of recent read/write sytsem call activity, or it may require the allocation of a new page frame and the initiation of a disk I/O operation. See "File Mappings"
• The PFN field and Present bit are all 0 and the `vm_area_struct` tells us that this is an anonymous region. The PTE being entirely 0 tells us that this virtual page is "virgin": it has not previously been written to and swapped out. This situation is always resolvable as a minor fault: The kernel finds a page frame from the free pool and maps it to the faulted virtual page. If the faulted access was a write, the kernel 0-fills the page. However, for a read access, the kernel optimizes by mapping to a single, shared physical page containing all 0, and setting up a COW situation by making the PTE protection bits read-only. At a later time, if the process writes to this page, things will be straightened out by Copy-On-Write. This behavior of **zero-filling** new anonymous pages is very important for security. Otherwise, sensitive information from another process could be viewed when the page frame is recycled.
• The PFN field of the PTE is non-zero, but the Present bit is 0 (and the Dirty bit is also 0). The kernel deliberately sets up this odd situation to indicate an anonymous page which has been `swapped out`. The page frame number does not correspond to a real address, but is used as an identifier to help the swap subsystem find the image of the page in the swap area. We will need to initiate an I/O operation to swap-in the page image to a new page frame. This will be discussed further under "Swap Space".

• The Present bit is 1. At this point in the code, we know that it must be a hardware protection violation which tripped the page fault, and we have previously checked that the attempted access is valid for this region. Therefore we need to resolve the copy-on-write as discussed above.



Note that in the above diagram, the order of the bits within the PFN has been abstracted to make a clearer presention. If you dare to look at the X86 documentation, you'd find it differs.

Page Frame Reclamation / Paging-Out

We can think of the pool of page frames as a cache for actively-used data which are otherwise stored on disk (in a file or in swap space). Linux uses a strategy of maintaining a pool of free page frames proactively. Consider what would happen if this strategy were

not in place: We call the total number of page frames mapped to a process the **Resident Set Size (RSS)**. As processes request virtual memory and are allocated page frames, their RSS would grow, and the pool of free frames would shrink and shrink. Eventually, there will not be any frames available to satisfy a new request. This could block the process for a long time while a "victim" page is selected to be written back to backing store and re-assigned.

Instead, the Linux kernel uses a kernel task (*a kernel task aka kernel thread is like a user-mode process, but it never leaves kernel mode. It can make system calls, using a slightly modified, internalized syntax. Kernel tasks are scheduled by the scheduler, can go to sleep and wakeup, etc. They do not have their own mm_struct but "borrow" one from a user process, which is OK since they only access the shared kernel memory area*) called `kswapd` to periodically scan the `mem_map`, looking for possible victims from which to steal back pages and thus reduce the total RSS. There are two parameters which can be tuned by the system administrator establishing low-water and high-water marks for free pages. `kswapd` works hard to keep the free page count above low-water, and goes to sleep when it reaches the high-water mark. To be more precise, each memory zone (e.g. NORMAL, HIGHMEM) has its own high and low water marks, but we will ignore this detail.

The Page Frame Reclamation Algorithm (PFRA) maintains two lists of non-free page frames: active and inactive. These lists are chained via the `lru` field of the page descriptor. Periodically, the PFRA scans the page descriptors on the active list with the desire to find hypoactive pages and move them to the inactive list. It uses the reverse mapping (see below) in the page descriptor to find the PTEs mapping to this physical page (when the page is shared there will be multiple PTE pointing to the same physical page). Each PTE is examined to see if the ACCESSED bit is set, then the ACCESSED bit is cleared. The PFRA also keeps a second bitwise flag, `PG_referenced`, in the `flags` field of the page descriptor, which stores the previous value of the ACCESSED PTE flag. (when there are multiple PTEs mapping the same page, PG_referenced is the OR of the ACCESSED PTE bits).

When an page which had `PG_referenced` set to 0 is visited again and the ACCESSED PTE flag(s) is (are all) still clear, it means the page has not been used in two successive scan passes. The PFRA will then move the page to the tail of the *inactive list*.

After the PFRA finishes hunting for active pages to move to the inactive list, it starts trying to pull victims off the inactive list, starting at the head of the list. I.e. it starts with the pages which have been sitting on the inactive list the longest. This is known as the **Least Recently Used (LRU)** ordering. The theory is that pages which have remained inactive for a long time will probably continue to be inactive.

For each inactive page scanned, the ACCESSED bit(s) is (are) checked again. Note that this bit is only cleared when the PFRA has visited the page from the *active* list, therefore if there has been any access whatsoever between the time that the page was moved to

inactive and the time it was visited on the inactive list, this PTE bit will be set, and the page is just moved back to the active list. Otherwise, the page will be reclaimed. If the PTE indicates that the page is dirty, a write-back to backing store is begun. This operation might take a while, so `kswapd` continues to look at additional pages that are not dirty, until it has reclaimed the desired number of pages. Pages which are not dirty can be reclaimed by simply immediately unmapping the PTE(s). Pages which were in the process of being written back are unmapped when that writeback operation concludes.

Pages that the PFRA has reclaimed are then placed on the free list. Since they are placed at the tail of the free list (for a given buddy order) and the page allocator pulls from the front, this further enhances the notion of Least Recently Used. Pages which are on the free list and haven't been re-used but still contain valid images can be pulled back off so that a page fault can be resolved as a minor fault.

The PFRA is adaptive and if it feels that it isn't reclaiming enough memory, it increases the rate at which it scans active pages, thus making the activity timeout window shorter and increasing the pool of potential inactive pages. This is a delicate balancing act because the PFRA/`kswapd` consumes CPU resources in doing all of this scanning. Excessive scanning wastes CPU time, but insufficient reclamation can cause memory starvation and severe performance penalties.

Like scheduling, memory allocation and reclamation algorithms are the subject of extensive research and development.

Reverse Mapping

For any given page frame, there may be multiple mappings for it. The kernel must keep track of all of them and prevent inconsistent views. For example, a given page may appear as part of the virtual address space of one or more processes, correspond to a particular offset in a particular memory-mapped file, and furthermore correspond to a particular offset within the hard disk storing that file. The PFRA, when visiting page frames, must be able to find all existing mappings (PTEs) of that page so that it can locate and possibly modify the PTEs. To visit the PTEs, the PFRA must determine the `vm_area_struct` and offset within that region (there could be multiple regions).

Either a page is mapped to a file, or it is anonymous. These two conditions are mutually exclusive. The `mapping` field of the page descriptor contains a pointer to one of two objects which will help with the reverse mapping. If the least significant bit is 1, the page is anonymous and `mapping &~01` points to a `struct anon_vma`. If the lsb is 0, the page maps to a file and `mapping` points to a `struct address_space` object for the file. This use of the least significant bit is a programming trick: because kernel pointers are always aligned on 4-byte boundaries, the least significant bit can be wasted for this purpose.

Anonymous Mappings

The `anon_vma` is simply the head of a doubly-linked circular list of memory region descriptors (`vm_area_struct`), chaining through the `anon_vma_node` pointers in the latter. Furthermore, the `vm_area_struct` contains a pointer (`anon_vma`) back to the `anon_vma` list head. More often than not, the list contains just one memory region descriptor. However, when a region becomes shared (e.g. after a fork) this list contains each region in each process which refers to it. This will also be the case for MAP_SHARED regions that were inherited via fork. Therefore, each `anon_vma` structure represents one anonymous virtual memory region (which might be shared among multiple processes).

A process data region or other MAP_PRIVATE file mappings might find itself on the `anon_vma` list and also part of a file mapping, because some pages in the region still correspond to the file, but some have been written to making them anonymous. Within this region, mapped-in pages are either "virgin" and still correspond to the original file, or they have been dirtied and are now backed by swap space. This distinction is captured via the least significant bit of the `mapping` field of each page descriptor as described above.

The PFRA, after getting the `anon_vma` list start from the page descriptor, can walk the linked list anchored by the `struct anon_vma` and visit each memory region (`vm_area_struct`) which maps that page, which gives the starting virtual address of the that region. The `index` field of the page descriptor gives the offset, in pages, of that particular page within the region. The `pgd` field of the `struct mm_struct`, which in turn is accessible from the vm_area_struct, gives the address of the page global directory for the process. It is now a trivial matter to walk through the page table structure and find the PTE.

Swap Space and anonymous page-outs

When a dirty anonymous page is selected for reclamation, it must be saved someplace before reclaiming the page. Since we can't save it to a file, it must be saved to **swap space**. The kernel maintains one or more swap areas, which are either raw disk partitions, or files, in either case prepared with the system utility command `mkswap`, and then made available as part of swap space with the `swapon` command.

Anonymous pages aren't meant to be persistent, and so swap space is just a pool of disk space where we can stash them. When the system reboots, that part of disk is not preserved. There is nothing within a swap area which would tell one the identities of the page images. That information is kept by the kernel in RAM and simply discarded when the system halts. The kernel maintains an array of swap area descriptors, each of which includes a free map. Each slot of a free map gives the reference count of the

corresponding page-sized slot of the corresponding swap area.

A particular page frame containing an anonymous page may be mapped by multiple PTEs in one or more processes. Therefore, when that page frame is swapped out, ALL of the PTE must be marked as not present. A free slot in some swap area is chosen (the algorithm to do so keeps things such as disk I/O performance in mind). Free slots are indicated by a swap map entry of 0. Once the slot is allocated, the swap map entry will reflect the number of mappings that had been pointing at that shared page frame.

Now the Linux kernel uses a slightly dirty trick. The swap area number (a 7-bit integer) combined with the slot number within the area form a unique identifier of the swap slot. 24 bits are allowed for the slot number, meaning the maximum size of one swap area is 2^24*2^12=2^36 or 64 GB. The total amount of swap space on the system is limited to 2^(36+7) = 8 TB.

Given this information, the page fault handler can, when the page is once again referenced, find the corresponding swap slot on disk and read it in. This swap identifier is stored in each PTE which had mapped the swapped-out page. There are 31 bits in the identifier and a PTE is 32 bits. The Linux kernel packs the bits into the PTE in such a way that the Present flag is always 0, and the remaining 31 bits are scattered between the PFN and the Flags fields. It isn't a valid PTE but since the Present bit is OFF, the hardware will never look at it or touch it. (On X86-64, there are 63 bits of the PTE to play with for encoding a swap area and swap slot....that's a LOT of swap space!)

A page that was previously swapped out and is now faulted back in from swap space gets flagged as DIRTY by the kernel, even if this page fault is just for reading. This is because the underlying virtual page image is in fact dirty. By virtue of the fact that it was in swap space, it is no longer the virgin (all-0) anonymous page. Also recall that the Linux kernel uses copy-on-write to handle read-only page faults for virgin anonymous pages.

File Mappings

For files which are mapped into memory, it is important to be able to locate quickly the page frame which currently contains a particular (page-aligned) offset in the file, and/or to determine that no such page is currently resident. Consider the `read` system call, the semantics of which must work correctly and transparently with memory-mapped files. `read` amounts to finding the page frame which holds that part of the file (if there is no such frame, ask the filesystem to bring it in from disk and sleep until that happens), and then copying the requested portion of that page into the user buffer.

Likewise, when faulting-in a memory-mapped file, it could be that the same (page-aligned) area of the file has recently been accessed from the same or different process using the read or write system call, or via an mmap'ing, in which case the fault can be resolved a minor fault by pointing the PTE at the existing page frame which contains that

image.

An `address_space` structure is used to provide this quick lookup. One of these is allocated for every open inode. It has many fields, but among them are two types of trees. One is a flexible radix-64 tree which, given an offset (in page-sized units) into the file, finds the struct page corresponding to that part of the file, or NULL if there is none such. Each node of the radix-64 tree has 64 pointers. If the file size is 256K or less (64*4096) then the entire file's address space is handled by a single-level tree. Otherwise, the tree grows. A two-level tree can handle (64*64*4096) or 16MB of file addressing, etc. up to 6 levels.

```
struct address_space {                  /* /usr/src/linux/include/linux/fs.h */
        struct inode            *host;          /* owner: inode, block_device */
        struct radix_tree_root  page_tree;      /* radix tree of all pages */
        spinlock_t              tree_lock;      /* and lock protecting it */
        unsigned int            i_mmap_writable;/* count VM_SHARED mappings */
        struct prio_tree_root   i_mmap;         /* tree of private and shared mappings */
        spinlock_t              i_mmap_lock;    /* protect tree, count, list */
        unsigned long           nrpages;        /* number of total pages */
          /* some other fields elided for simplicity */
};
```

PST & Reverse Mapping for file-mapped regions

While the radix tree is great for going from a known offset in the file to a page descriptor, the PFRA is trying to go a different way: find all PTEs which refer to this particular offset within the file. The `address_space` pointer of the `struct page` identifies the file, and the `index` field gives the offset in pages within the file. A simple approach would be to link all regions mapping to the file into a linked list, as is done for the `anon_vma` structure. However, while sharing of an anonymous region is usually limited to a small number of processes, memory-mapped files, especially those containing a common executable, are shared by many processes. There can also potentially be multiple mappings to different parts of one file (consider an a.out file where the .text and .data section are mapped separately).

Linux uses a Priority Search Tree for *each file*. It is accessed via the `address_space` structure's `i_mmap` pointer, and can quickly answer the following question: "given a particular offset within the file, tell me all memory regions which contain it." I.e. the PST sorts the mapping regions by their starting offset. The PFRA examines all memory region descriptors (`vm_area_struct`) which map to the page. Each descriptor contains the starting virtual address of the region, and the corresponding offset within the file (this is in the `vm_pgoff` field). Now the offset of the page within the region can be readily calculated, and from there the PTE can be found.

Paging-out file-mapped page frames

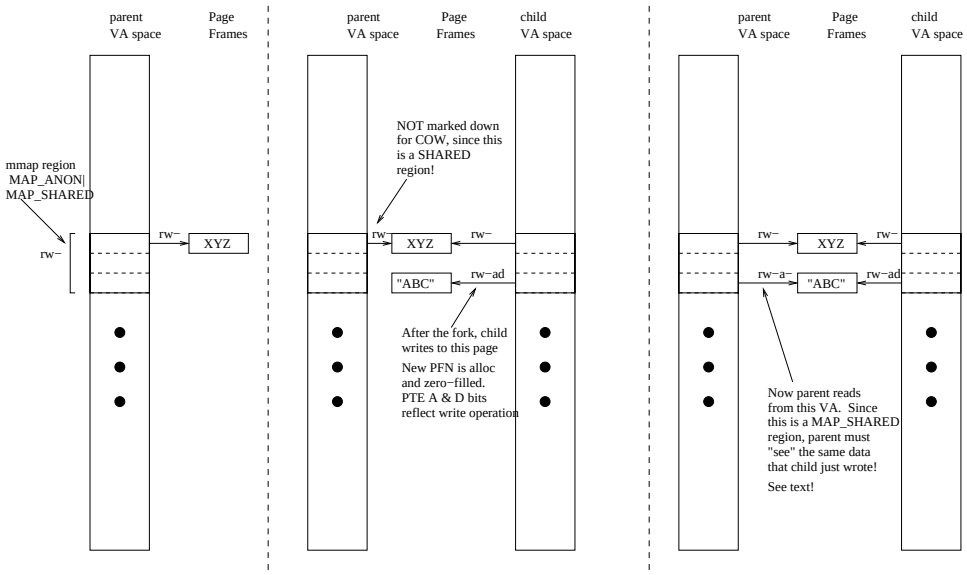
A file-mapped page which needs to be paged out and which is dirty must be part of a MAP_SHARED file-mapped region (if it were MAP_PRIVATE, the dirty pages would be paged-out to swap). We know from the `mapping` and the `index` fields of the `struct page` the file and the offset within that file where the page needs to go. So paging-out a file-mapped dirty page involves asking the underlying filesystem module to write the page to that offset within the file on the disk.

A special case: MAP_ANONYMOUS|MAP_SHARED

When an mmap region is established with MAP_ANONYMOUS | MAP_SHARED, it is clearly anonymous: no part of any file corresponds to it. But in order for it to work correctly when shared among multiple address spaces, the Linux kernel must resort to a trick. Consider what would happen if such a region is created and then is being shared by two processes after a fork. Process A demand-pages in a page of this region for the first time. Since it is anonymous, the kernel satisfies this as a minor fault with a zero-filled page frame. Let us further suppose that A writes to this page. At some later time, process B also tries to access, for the first time, the same virtual memory address that A did. In order for MAP_SHARED to work correctly, B must **not** get a zero-filled page frame! Instead it must get the same page frame that A has.

To implement this, the Linux kernel treats this region as if it were mapped to a file, but not a file that is on disk or has a pathname in the ordinary sense. The kernel creates a one-time inode that is associated with the `/dev/zero` special file, which is a character device that appears to supply endless 0 bytes. The inode is disconnected from pathname space so there is no way another process could ever access it. Mapping to an instance of `/dev/zero` satisifies the idea that new pages of this MAP_ANON|MAP_SHARED region should be 0-filled, but also preserves the sense that each such mapping is distinct. Now in the above example, when B incurs its page fault, this will be treated by the kernel as a forward-mapping file-mapped case. The kernel will consult this isolated inode and look at its `address_space` structure, from which the page frame that already contains the correct image of the region (currently mapped in A) will be found. It is then a simple minor page fault to hook up this PFN to B.

However, `/dev/zero` is not a real file. When page frames are associated with it, they are marked as anonymous, despite their file-mapped apparent origin. If a page frame associated with this MAP_ANON|MAP_SHARED region gets reclaimed, this will get written back to *swap space*, just like an anonymous page. Some special magic is then needed when/if the page is faulted back in to make sure the `address_space` correctly reflects its existence.



Resource Exhaustion

In an ideal world, systems would always have sufficient resources to handle the jobs which are thrown at them. But that world is not a terribly interesting one in terms of operating systems design. In reality, memory resource exhaustion is often a problem. We can consider two distinct problem areas: Insufficient physical memory, and insufficient swap space.

Thrashing

When memory becomes very scarce, the system can spend much of its time freeing up pages by swapping-out, followed by swapping-in when a process which has had its pages stolen gets scheduled and tries to run. This is an age-old problem known as **thrashing**. Older versions of Linux tried to ameliorate it by using what Linux called a "swap token". A process which holds the swap token is given a temporary reprieve by the PFRA. The swap token is given to a process based on heuristic rules...basically a process which has recently been victimized by the PFRA, has good priority and looks like it will do something useful if allowed to run.

This isn’t necessarily a perfect solution. To contrast, the Solaris kernel takes a different approach. When memory runs low, a kernel thread called the "swapper" picks entire processes to victimize and steals all of the process’ pages. A process thus swapped-out is basically put to sleep and therefore doesn’t get scheduled. When things get calmer, the process is released from the swapper’s grasp and is allowed to run and fault its pages back in.

More recent versions of Linux use a sophisticated framework known as Control Groups (cgroup) to provide memory management policies. This allows groups of processes to be considered together in terms of their memory usage, and to determine which groups have priority. When memory is low, low-priority cgroups would be targeted first for page reclaims.

Memory over-commit and the out-of-memory killer

Just like an airline which over-books seats, it is possible for the kernel to hand out more virtual memory than it has physical memory and swap to cover. If it happens that all of the memory is demanded at once and there are insufficient free swap slots to page-out to, the kernel will have no choice but to kill processes in order to regain memory. There is an Out-Of-Memory killer which is triggered when the system becomes desperately low of memory. It picks large processes which haven't been running too long.

This might seem like a design flaw. After all, the programmer successfully allocated memory using malloc (which in turn used `brk` or `mmap`), so why has the kernel gone back on its promise and why should a process be terminated because of something which is not its fault?

On the other hand, the most conservative approach, which guarantees that one will never run out of memory, is to reserve a swap space slot for each virtual anonymous page allocated. Then there will always be a place to page-out any resident page and we can never get into this trouble. The problem with this approach is that processes will start getting memory allocation failures long before the system has reached a high probability of running out of memory, unless the administrator has configured a very large amount of swap space. The reason: most virtual anonymous pages are never actually paged-out. Consider a large process which forks and then execs. Most of the virtual address space of the child will be discarded.

Older Linux kernels took the first approach. Solaris kernels tend towards to more conservative approach. Modern (2.6+) Linux kernels allow the administrator to tune the memory over-commit policy.

```
The following is verbatim from \fCDocumentation/vm/overcommit-accounting\fP:

The Linux kernel supports the following overcommit handling modes

0       -       Heuristic overcommit handling. Obvious overcommits of
                address space are refused. Used for a typical system. It
                ensures a seriously wild allocation fails while allowing
                overcommit to reduce swap usage.  root is allowed to
                allocate slighly more memory in this mode. This is the
                default.

1       -       Always overcommit. Appropriate for some scientific
                applications.
```

```
2       -       Don't overcommit. The total address space commit
                for the system is not permitted to exceed swap + a
                configurable percentage (default is 50) of physical RAM.
                Depending on the percentage you use, in most situations
                this means a process will not be killed while accessing
                pages but will receive errors on memory allocation as
                appropriate.

The overcommit policy is set via the sysctl 'vm.overcommit_memory'.

The overcommit percentage is set via 'vm.overcommit_ratio'.

The current overcommit limit and amount committed are viewable in
/proc/meminfo as CommitLimit and Committed_AS respectively.

Gotchas
-------

The C language stack growth does an implicit mremap. If you want absolute
guarantees and run close to the edge you MUST mmap your stack for the
largest size you think you will need. For typical stack usage this does
not matter much but it's a corner case if you really really care

In mode 2 the MAP_NORESERVE flag is ignored.


How It Works
------------

The overcommit is based on the following rules

For a file backed map
        SHARED or READ-only     -       0 cost (the file is the map not swap)
        PRIVATE WRITABLE        -       size of mapping per instance

For an anonymous or /dev/zero map
        SHARED                  -       size of mapping
        PRIVATE READ-only       -       0 cost (but of little use)
        PRIVATE WRITABLE        -       size of mapping per instance

Additional accounting
        Pages made writable copies by mmap
        shmfs memory drawn from the same pool
```

The /proc interface

Under the Linux kernel, we can get a lot of internal information about the memory subsystem via the `/proc` interface. Here are some of the fun things:

• `/proc/pid/maps` : a human-readable listing of all the virtual memory regions (vm_area_struct). Each line represents one region and contains: the start and end virtual

address, some of the VM_flags (READ,WRITE,EXEC and SHARED vs PRIVATE), and if the region is file-backed, the offset, device and inode number as well as the pathname of the file.

• `/proc/pid/smaps` : same info as above, but in an expanded format which shows all of the VM_XXX flags, and a count of how many PTEs are mapped in, and how many are currently dirty, plus a bunch more interesting stuff. `/proc/pid/numa_maps`: similar to `maps`, one line per region, summarizes number of PTE mappings and how many are dirty. Has some other stuff regarding NUMA (non-uniform memory architecture) that is way beyond this course!

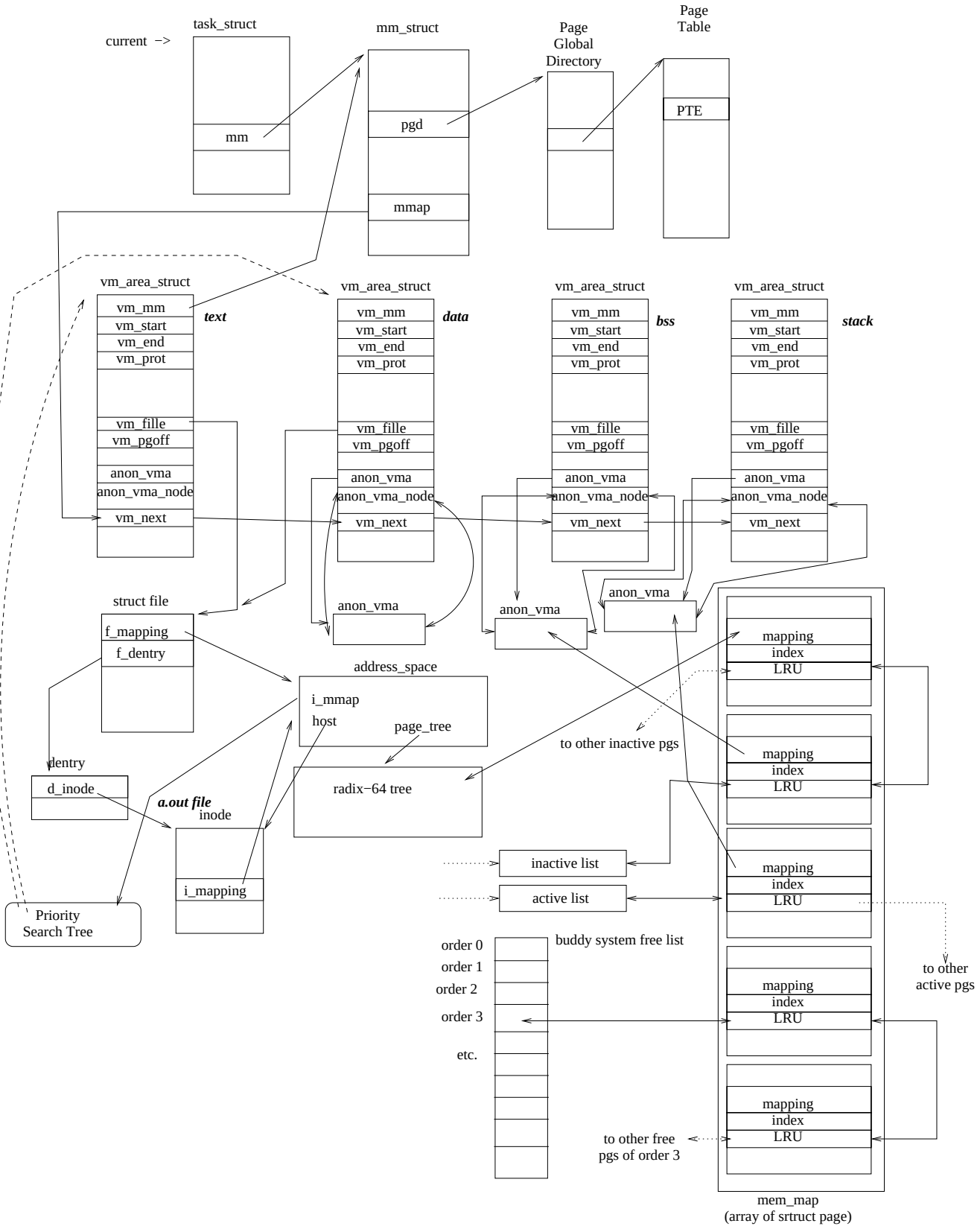
• `/proc/pid/pagemap`: This is a binary file consisting of a series of 64-bit entries, each of which provides information about the corresponding PTE in the virtual address space. Not all of the PTE is exposed unfortunately, but we can see the Page Frame Number part of the PTE and a few other properties.

• `/proc/kpageflags`: This binary file is global (not per-PID) because it is indexed by PFN. It contains a series of 64-bit entries, in PFN order, which encode the PG_XXX page frame flags.

• `/proc/kpagecount`: A similar file which contains the `mapcount` field of the struct page.

A summary of data structures

On the next page is a very complicated diagram showing the relationships between the various kernel data structures that keep track of virtual memory. It does not represent every possible situation, but shows us one process with the usual 4 memory regions.


task_struct
current ->
mm
mm_struct
pgd
mmap
Page Global Directory
Page Table
PTE
vm_area_struct
vm_mm
vm_start
vm_end
vm_prot
vm_fille
vm_pgoff
anon_vma
anon_vma_node
vm_next
text
data
bss
stack
struct file
f_mapping
f_dentry
anon_vma
address_space
i_mmap
host
page_tree
radix−64 tree
dentry
d_inode
a.out file
inode
i_mapping
Priority Search Tree
to other inactive pgs
inactive list
active list
buddy system free list
order 0
order 1
order 2
order 3
etc.
mapping
index
LRU
to other active pgs
to other free pgs of order 3
mem_map
(array of srtruct page)

Appendix: How an executable file is created

In the original UNIX kernel, each program was entirely **statically-linked**, i.e. all of the code required to execute the program, and in particular all of the libraries which the program requires, are put together at compile-time into one large monolithic executable file. This practice began to change during the 1990s, and today almost all programs in almost all UNIX-derived systems are *dynamically linked*. It is still possible to compile a program which is statically linked, but this is exceptional. We will first discuss how statically linked programs are linked together, and then delve into dynamic linking.

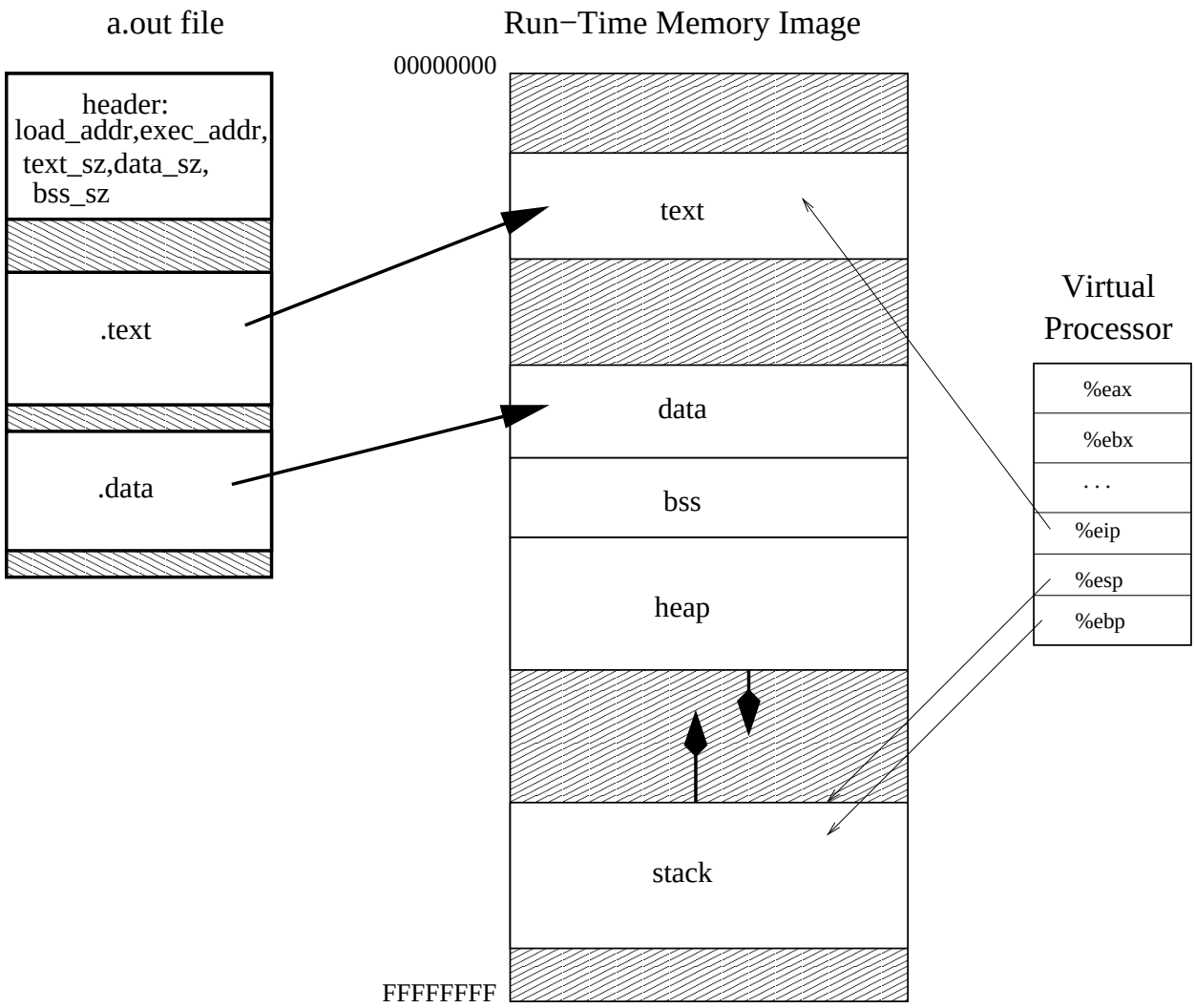
Compiling a C program into an executable file is a multi-stage process involving several tools. When one simply runs:

```
cc test.c
```

The `cc` command transparently executes the tools, detailed below, resulting in an executable file called `a.out` (if there are no fatal errors in compilation). The "a" in "a.out" stands for "Absolute", meaning that all symbolic references have been resolved. An a.out file contains pure machine code that can be directly executed by the processor. The executable file ("a.out") contains everything that the operating system needs to create a program's initial memory configuration and begin execution. The header of the a.out identifies it to the operating system as an executable file and specifies the processor architecture on which the machine code instructions will run. A program compiled for an x86 architecture can not be executed, for example, on a SPARC processor. The header also gives the size of the text, data and bss memory requirements, as explained below.

The `a.out` file includes the literal bytes of executable code which will be loaded into the process's `text` region. These are in a contiguous part of the `a.out` file. The header gives the offset of the beginning of the `.text` section of the `a.out`, and also the **load address**, i.e. the virtual address at which the compilation system expects this text to be loaded.

Likewise, there is a `.data` section in the `a.out` which contains an image of what the `data` region of the process will look like at program startup.


a.out file
Run−Time Memory Image
00000000
header:
load_addr,exec_addr,
text_sz,data_sz,
bss_sz
.text
.data
text
data
bss
heap
stack
FFFFFFFF
Virtual
Processor
%eax
%ebx
. . .
%eip
%esp
%ebp

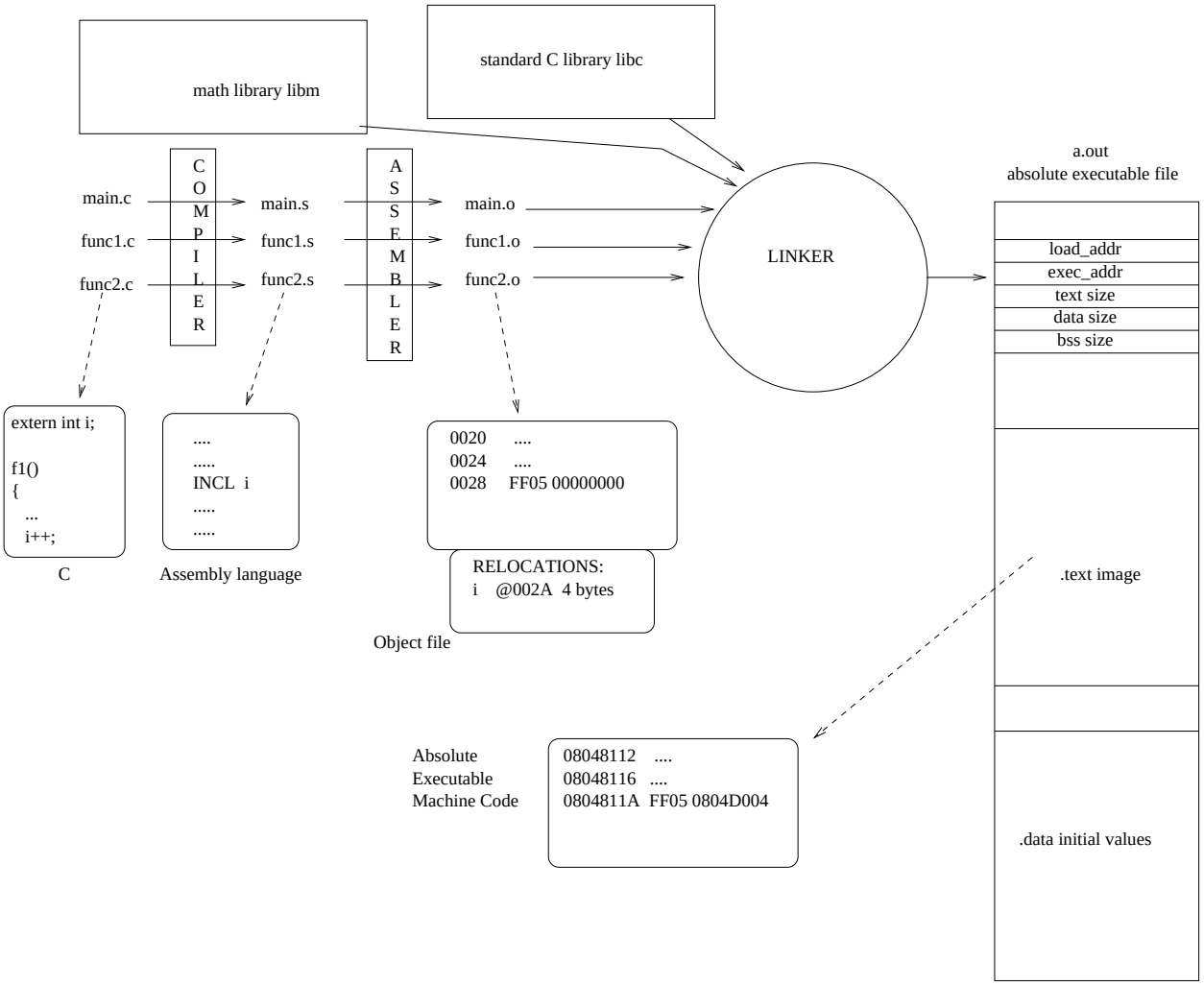
The **bss** section contains all un-initialized global variables. The C language specification states that all such variables, lacking an explicit initializer, must be initialized by the operating system or C run-time environment to 0.

```
int j=2;
int k;
main()
{
 int l;
          /*...*/
}
```

In the example above, `j` has an explicit initializer, and will be in the data segment. The value of the initializer will be found in the .data section of the a.out. `k` is uninitialized. Therefore it will reside in the bss segment of memory and will have an initial value of 0. The ISO C standard says that "If an object that has static storage duration is not initialized explicitly, it is initialized impliclitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a NULL pointer constant". This is satisfied by filling every byte of the bss region with 0, and is performed by the operating system. The variable `l` has automatic storage scope, and therefore will be found on the stack (or, if the compiler is set for heavy optimization and a pointer to `l` is never taken, it may live in a register). Automatic variables are not 0-initialized.

Behind cc-losed doors

The `cc` command is often and erroneously called "the compiler." In fact, it is a wrapper program which invokes a series of tools. Each tool transforms a specific kind of input file to a specific kind of output file. It is possible to instruct `cc` to interrupt the process at any stage, and to access and manipulate these intermediate files as needed.



The compiler proper takes C code, passing it through the macro pre-processor, and compiles it into symbolic assembly language. By using the `-S` option to `cc`, we can halt the process before the assembler and linker are invoked, and view the intermediate assembly language file, which will have a `.s` extension.

The assembler takes symbolic assembly language files (`.s`) and converts them into relocatable object files (which have a `.o` extension). An object file is similar to an executable ("a.out") file in that it contains machine code, however symbols which can not be resolved (as detailed below) are left dangling. Therefore, it is not ready to run until these dangling links are settled, possibly by linking with additional object files. By giving `cc` the `-c` option, the linker will not be run and the `.o` files will be left for subsequent examination or linking. It is also possible to invoke the assembler directly with the `as` command.

Although some compilers do not separate the assembler, it is useful to divorce it from the compiler and make it a separate utility. In this way, a common assembler program can be

written and employed by compilers for different languages. For example, the GNU compilers for C, C++ and FORTRAN are all different programs, yet they all call on the `as` program for assembly. It is also useful to be able to introduce `.s` files that have been written by hand for certain performance optimizations or platform-specific features.

The linker takes multiple object (`.o`) files (or libraries of object files, with `.a` extensions, as discussed below), resolves symbol references, and creates an absolute executable file. If unresolved symbols remain at this point, the linker will not be able to create an absolute executable and an error will result.

Like the assembler, the linker is a separate tool which can be called directly as `ld`. The reason for this rather un-mnemonic two-letter name is that historically, the linker has also been called the "loader", because it was responsible for creating the "load deck" of punch cards that could then be directly run by the computer. Besides, `ln` was already taken as a command name.

When compiling C programs, it is best to allow the `cc` wrapper to invoke `ld` rather than trying to do it manually as there are a number of complex platform-specific options needed to actually create a run-able program.

The `.o` files need not have come from C source code. It is possible to distribute binary object files or libraries which can be linked against user-supplied code. It is also possible to create cross-language executables, where part of the code is written directly in low-level assembly language, or in another high-level language (with all due caution about mixed run-time environment expectations).

Separate Compilation

Very simple C programs can be contained in a single `.c` file. As the size of the project increases, it becomes extremely inefficient to do this. The tasks of pre-processing, compiling and assembling are CPU-intensive. With a single file, any change, no matter how small, requires that the entire file be recompiled.

The program can be split up into a number of smaller `.c` files. There are varying opinions on how finely to divide the program and on what basis to select which parts go into which file. Some guidelines:

1) Groups of functions which form a coherent subsystem are good candidates for including in their own file. In C++ terms, this would be a class. Some subsystems are so complex that they need to be further subdivided.

2) Whenever a single `.c` file is on the order of 1000 lines of code, it is probably getting

too big. On the other hand, having lots of files each containing one or two 10-line functions is also wasteful.

3) Code which is being incorporated from another source, or which is being worked on by a different programmer, should have its own file, or perhaps be structured as a library, to make it easier to deal with changes.

As an example, let's say our program has been divided into files `f1.c`, `f2.c` and `f3.c`. To compile this, we could run: `cc f1.c f2.c f3.c`, which would perform the entire pre-processing, compiling and assembling process on each of the three files, then invoke the linker to create the `a.out`. However, suppose we make a change to `f1.c` only. Why do all that work again for the other two files, which have not changed?

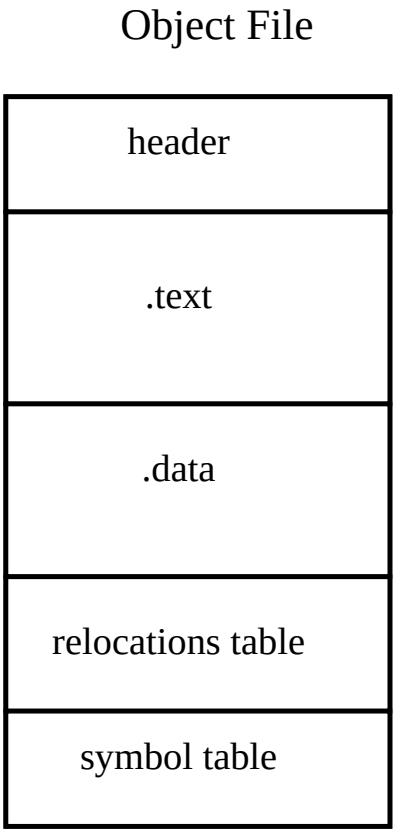
Instead, we can first run: `cc -c f1.c f2.c f3.c`, which will pre-process, compile and assemble each of the files, creating `f1.o`, `f2.o` and `f3.o`. Then we run: `cc f1.o f2.o f3.o`. The `cc` wrapper sees that we are giving it `.o` files and passes them directly to the linker. Now, if we change `f1.c`, we run `cc -c f1.c` to re-compile it alone, and then run `cc f1.o f2.o f3.o` again to link the program. This process is automated by the tool `make`, which uses dependency rules and file modification times to determine which compilation phases to invoke and on which files.

The linker vs the compiler

The job of the linker is to take one or more relocatable object files, resolve symbol references, and create an absolute executable file. The linker does not know about C's type algebra, nor should it, since the linker is independent of any particular high-level language.

Unfortunately for the student, the C language also deals in symbols, e.g. variable and function names, and this issue is sometimes difficult to separate from the symbol processing which happens at link time. Here are some things that never appear in an object file and do not concern the linker:
• C language type specifiers (such as "p is a pointer to a pointer to a function returning int and taking two int arguments")
• C language structure, union and enum definitions and typedef names.
• `goto` labels
• Internal labels that may be generated by the compiler for loops, switch statements, if statements, etc.
• automatic (local) variables. An important exception is a local variable that is declared with the `static` storage class. This variable has local scope, i.e. its name is not visible outside of the curly-braced block in which it is declared, but it lives in the same

neighborhood as global variables.



What's in a .o?

Let's see what is actually in an object file and how it gets there:

```
/* f1.c */
extern int i;

f()
{
          i=2;
}
```

```
/* f2.c */
int i;
main()
{
          i=1;
          f();
}
```

In `f1.c`, the `extern` storage class for variable *i* tells the compiler that this variable is external to that `.c` file. Therefore, the compiler does not complain when it does not see a declaration that variable. Instead, it knows that `i` is a global variable, and should be

accessed by using an absolute addressing mode. Neither the compiler nor the assembler knows what that actual address will be. That is not decided until the linker puts all the object files together and assigns addresses to symbols. Therefore, the assembler must leave a "place-holder" in the object file. Likewise, in f2.o, there will be a reference to the symbol `f`.

Let us examine the assembly language files produced (using an older gcc under Linux on an x86 system):

```
**** f1.s
         .text
         .globl f
        .type    f,@function
f:
        pushl %ebp          *These instructions set up
        movl %esp,%ebp              *the stack frame pointer
        movl $2,i
        leave                       *Restore frame pointer
        ret
.Lfe1:
        .size    f,.Lfe1-f

**** f2.s
         .text
         .globl main
        .type    main,@function
main:
        pushl %ebp          *These instructions set up
        movl %esp,%ebp              *the stack frame pointer
         movl $1,i
        call f
        leave                       *Restore frame pointer
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .comm   i,4,4
```

The assembler directive `.text` tells the assembler that it is assembling opcodes to go in the .text section of the object file. The `.globl` directive will cause the assembler to export the associated symbol as a defining instance. `.type` is used to pass along information into the object file as to the type of the symbol. Please note that it has nothing to do with the C language notion of type. Symbol types may either be functions or variables. The linker is able to catch gross errors such as if `f` were defined as a variable in `f1.c` instead of a function. The `.size` directive calculates the size of the function by subtracting the value of the symbol representing the first instruction of the function (e.g. `main`) from a placeholder assembler symbol (e.g. `.Lfe1`) representing the end of the function. Note that the CALL to function `f` is done symbolically, as is the assignment into global variable `i`.

The resulting object file is similar to an a.out file, however all addresses are relative. In addition, the object file will have a section known as the **symbol table** containing an entry for every symbol that is either defined or referenced in the file, and another section called the **relocations table**, described below.

Now, let us view the .text section of `f1.o` (the listing below was re-formatted from the output of objdump -d):

```
Offset Opcodes                        Disassembly
0000   55                          pushl %ebp
0001   89E5                        movl %esp,%ebp
0003   C7050000000002000000        movl $2,0
000D   C9                          leave
000E   C3                          ret
```

First, note that the offset of the first instruction is 0. Obviously, this can not be a valid memory address. All offsets in object files are relative to the object file. It isn't until the linker kicks in that symbols gain absolute, usable addresses.

Next note that in the instruction beginning at offset 0003, the constant 2 is moved to memory address 0. We know from examining the corresponding assembly language input file that this is the instruction that moves 2 into variable `i`. The assembler has left a place-holder of 00000000 in the object file where the linker will have to fill in the actual 32-bit address of symbol `i` once that is known. C705 is the x86 machine language opcode for the MOVL instruction where the source addressing mode is Immediate and the destination mode is Absolute. The next 4 bytes are the destination address and the final 4 bytes of the instruction are the source operand (which is in Intel-style, or "little-endian" byte order).

Here is the dump of f2.o:

```
Offset Opcodes                        Disassembly
0000   55                          pushl %ebp
0001   89E5                        movl %esp,%ebp
0003   C7050000000001000000        movl $1,0
000D   E8FCFFFFFF                  call 000E
0012   C9                          leave
0013   C3                          ret
```

Note that the placeholder in the CALL instruction is FFFFFFFC, but the disassembler decodes that as 000E. This is because the CALL instruction uses a Program Counter Relative addressing mode. The address to which execution jumps is the operand in the instruction added to the value of the Program Counter register *corresponding to the byte beyond the last byte of the CALL instruction*. In two's complement, FFFFFFFC is -4, therefore the CALL appears to be to the instruction at offset 000E. Of course, all of this is meaningless since it is just a placeholder that will be overwritten by the linker. Nonetheless, it reminds us that there are different **Relocation Types** depending on the

addressing mode being used.

The symbol and relocation tables for the two object files will look something like this:

```
f1.o:
                    SYMBOL TABLE
NAME    TYPE          VALUE
f       func          0 in .text
i       variable      [reference]
                    RELOCATIONS TABLE
SYMBOL=i   OFFSET=0005  LENGTH=4  RELTYPE=ABSOLUTE



f2.o:
                    SYMBOL TABLE
NAME    TYPE          VALUE
f       function      [reference]
i       variable      8 in .bss
main    function      0 in .text
                    RELOCATIONS TABLE
SYMBOL=i   OFFSET=0005  RELTYPE=ABSOLUTE
SYMBOL=f   OFFSET=000E  RELTYPE=PCRELATIVE
```

What ld does

The first task of `ld` is to take inventory of all of the `.o` files being presented to it. It loads in the symbol tables from all of the object files to create a unified symbol table for the entire program. There is only one global namespace for all linker symbols. This might be considered a deficiency. Let's say we have a module `foo.c` that defines a function called `calculate`. If we attempt to incorporate that module into a program written by someone else, they may have also made a function called `calculate`. It is, after all, a common name.

If there is more than one defining instance of a symbol, i.e. if a symbol is *multiply-defined*, this is generally a fatal error. Consider what would happen if a programmer accidentally included two versions of function `f` above in two different `.c` files. When `f` is called somewhere in the program, which version should be called?

To ameliorate this problem of flat global linker namespace, a convention exists that one should prepend to one's global variable and function names a reasonably unique prefix. Therefore, we might call our function `foo_calculate`. This is less likely to conflict with another name. It isn't a perfect solution, but it works fairly well in reality.

In languages like C++, which takes name overloading to an extreme, the identified names used in the source code are **name-mangled** to be non-conflicting when placed into the global linker namespace.

Global variable and function names that are intended to remain private to the `.c` file in which they are declared should be protected with the `static` storage class (see below). `static` symbols still require the assistance of the linker to be relocated. However, the use of `static` causes the compiler and the assembler to flag that symbol as a LOCAL symbol. The linker will then enter the symbol into a private namespace just for the corresponding object file, and the symbol will never conflict with symbols from other object files.

There are rare cases where it is useful to deliberately redefine a symbol. For instance, we may need to change how a piece of code, available only in library or object file form, calls another function. This all falls under the heading of "wild and crazy ld tricks" and will not be discussed further. Just remember that any duplicate symbol definitions are generally wrong.

There is one antiquated exception which `ld` still honors: the so-called COMMON block, which is a hangover from FORTRAN. If there are multiple defining instances of a variable in the bss section (i.e. a global variable without an initializer), `ld` will look the other way. This is because there really isn't a fatal conflict. As long as the definitions all give the same size and all agree that it is a bss symbol, there's no worry about which definition is the correct one. All are equivalent. So, if one accidentally declares a global uninitialized variable twice, the program will compile fine. It is sloppy, however.

Frequently symbols have a defining instance, but they are never actually referenced. For example, the programmer may write a function, but never call it, or declare a global variable, yet never use it in an expression. The linker doesn't care about this.

However, it cares deeply about the opposite case: a symbol that is referenced (by appearing in a relocations table) but which has no defining instance. Such a situation makes it impossible for the linker to complete its task of creating an absolute executable file with no dangling references. Therefore, it will stop with a fatal error, reporting the undefined symbol and the object file or files in which it is referenced.

Once all of the symbol definitions and references are resolved, `ld` will assemble the `a.out` file by concatenating all of the text sections of all of the object files, forming the single `.text` section of the `a.out`. In doing so, the text sections are relocated. An instruction which was at offset 10 in a particular object file's text section may now be at offset 1034 in the `a.out`.

Furthermore, `ld` has a concept of the `load address` of the program, i.e. the memory address at which the first byte of the `.text` section of the `a.out` will be loaded. This knowledge is part of configuring `ld` for a particular operating system and processor type, and is not normally something that the programmer needs to worry about. The load

address is also placed in the `a.out` file so the operating system is sure to load the program at the address for which it was linked. Knowing the size of each object file's `.text` section, `ld` can calculate the absolute address that it will occupy in the final image, and can thus complete the symbol table, assigning absolute values to each text symbol.

A similar process is undertaken for the `data` and `bss` sections. `ld` assigns an absolute address to each `data` or `bss` symbol, based on the configured starting address of the `data` and `bss` memory regions. In the case of `data` symbols, the initializers contained in the individual object files are concatenated, forming the `.data` section of the `a.out`, which will thus contain the byte-for-byte image of what that section will look like when the program starts. For `bss` symbols, there are obviously no initializers. `ld` keeps track of the total number of `bss` bytes needed, and places that information into the `a.out` header so the operating system can allocate the memory when the program is loaded. During this process, each `data` or `bss` symbol is assigned an absolute address in the symbol table.

Now `ld` concatenates the various object files, processing the relocation records, replacing each "placeholder" with the actual, absolute address that the associated symbol is now known to have. In the case of Program-Counter-Relative relocation types, the proper offset is calculated with respect to the absolute address of the placeholder in question.

Libraries

When you write a C program, you expect certain functions, such as `printf`, to be available. These are supplied in the form of a **library**. A library is basically a collection of `.o` files, organized together under a common wrapper format called a `.a` file. It is similar to a "tar" or "zip" archive (although no compression is provided).

Convention is that a library *ZZZ* is contained in the library file: `libZZZ.a`. Using the `-l` option to `cc` tells the `cc` program to ask the linker to link with the specified library. For example: `cc myprog.c -lm` asks for the system library file `libm.a` (the math library) to be additionally linked. By default, `cc` always links the standard C library `libc.a`. These libraries are located in a system directory, typically `/usr/lib`.

Although a library embodies a collection of object files, the behavior when linking to a library is slightly different than if one just linked in all of the object files individually. In the latter case, the `.text` and `.data` sections of each object file would wind up in the `a.out`, regardless of whether or not anything in a particular object file was actually used. With a library, `ld` builds a symbol table of all the object files in the library, and then only selects those object files that are actually needed for inclusion into the executable.

Dynamic Linking

When dynamic (shared) libraries are used, there are two parts to the linkage process. At compile time, `ld` links against the dynamic library (which has a `.so` extension in UNIX) for the purpose of learning which symbols are defined by it. However, none of the code or data initializers from the dynamic library are actually included in the a.out file. Instead, `ld` records which dynamic libraries were linked against.

At execution time, the second phase takes place before the `main` gets invoked. A small helper program called `ld.so` (typically, actual names vary depending on operating system build) is loaded into the process address space by the kernel and executed. The intervention of `ld.so` is made possible by the binary interpreter feature of the ELF (Extensible Linker Format) a.out format. Like the #! method for shell script (and other languages) interpreters, a flag in the a.out header informs the kernel that it needs to actually load `ld.so`, which is passed the absolute pathname of the `a.out` file as an argument. `ld.so` then does a series of mmap calls to establish the text, data and bss regions for the conventional (statically linked) part of the program, based on the a.out header information. It then locates each of the required dynamic libraries (also information from the header) and for each one, mmaps that library's text, data and bss (if needed) regions.

When using `strace` on Linux to trace system calls and we are tracing the exec of a dynamically linked program, we'll see the execve of that program. We won't see `ld.so` getting loaded because that happens as an internal kernel function, not a system call. Then next we'll see the various opens and mmaps.

Some additional magic is required to link the static and the dynamic portions of the executable. This aspect is extremely specific to the operating system and processor architecture. In general, for functions which are defined in a shared library, dummy stubs are provided in what is known as a **Procedure Linkage Table**. These stubs contain jumps to addresses which are filled in by the dynamic linker. This allows the static code to make function calls without being dynamic-aware. Likewise, a table known as the **Global Offset Table** provides transparent access to any global variables which a shared library exports. The dynamic libraries themselves need to be compiled in such as way that they are capable of being loaded at any virtual address. This is done with the `-fpic` (position independent code) flag to the compiler. The `ld.so` dynamic linker fills in the entries in the PLT and GOT based on the actual addresses that got assigned at run-time. The program is now completed linked and ready to execute, so `ld.so` jumps to the `_start` address contained in the a.out header. Since ld.so is designed to work in concert with the C library, this start function (the code of which is located in the text region associated with `libc.so`) does an munmap to destroy the text region that had contained the `ld.so` code.

There is a slight performance penalty when using dynamic libraries. In additional to the delay in executing the program imposed by needing to load and link the libraries, calls to functions in the dynamic library require an additional branch, and accesses to global symbols from within the shared library must use a position-independent addressing method, which generally requires an indirect addressing mode.

One down-side of dynamic linking is that a missing dynamic library (".so" files in UNIX terms, or "DLL"s in Windows-speak) will prevent a program from running, even if it was complete and correct at compile time. However, this disadvantage is more than outweighed by the reduction in executable size and the convenience of being able to correct errors in system libraries on the fly, without having to locate and recompile each and every executable based on the faulty library.

The program `ldd` can be used to debug dynamic library resolution problems. This example shows the dynamic libraries that are needed for `ls` on a 32-bit Linux system:

```
$ ldd /bin/ls
        linux-gate.so.1 =>  (0xb77c4000)
        librt.so.1 => /lib/librt.so.1 (0xb77b6000)
        libacl.so.1 => /lib/libacl.so.1 (0xb77ad000)
        libc.so.6 => /lib/libc.so.6 (0xb765b000)
        libpthread.so.0 => /lib/libpthread.so.0 (0xb7642000)
        /lib/ld-linux.so.2 (0xb77c5000)
        libattr.so.1 => /lib/libattr.so.1 (0xb763b000)
```

The numbers in parentheses are the addresses at which each of the dynamic libraries would load if the program were actually being executed.

Under certain circumstances, such as security-sensitive applications, it may be appropriate to use static linkage. This ensures that the program will always be able to run, and will always run with exactly the expected code.

Synchronization among Concurrent tasks

In this unit we will consider a very important set of problems in Operating Systems design: synchronization between multiple *overlapping* threads of execution which share data structures in memory. We will see that the issues presented exist in a variety of contexts, although the specifics of how those issues can be solved differs. Specifically, in the UNIX environment, multiple threads of execution exist:

--User level--
• Within a single, conventional (single-threaded) UNIX process, a signal which causes invocation of a handler function can be considered to be a new thread of execution, because the main program code can not control the point at which the signal handler will be invoked. Although the main program's thread is suspended during the execution of the handler, the execution timelines of the handler and the main program overlap and thus can introduce issues of synchronization if both the main program and the handler need to access the same data structure.
• Within a single UNIX process when using a multi-threaded programming environment, e.g. the Posix Threads (pthreads) library. We have not yet formally considered threads programming in this course.
• Among UNIX processes which share a memory region (e.g. through the `mmap` system call). Here the threads of execution are the individual processes. In both this case and the case above, the threads will execute either virtually at the same time (if there is only a single real CPU), or truly in parallel (if there are multiple CPUs or cores)

--Kernel level--
• We will soon see that the Linux kernel is like a giant, multi-threaded program. If there is a single processor, the situation is analogous to a single-threaded UNIX process with handled signals, in that the only synchronization concerns are between what the kernel is doing synchronously on behalf of the process, and what an interrupt handler is doing asynchronously.
• When the Linux kernel is run on a multi-processor system, it is analogous to a multi-threaded UNIX process. There are synchronization issues among these multiple threads of execution, each on an independent processor, as well as with interrupt handlers.

In this unit, we will use the term "thread" and "task" interchangeably and discuss synchronization problems broadly, applying them to all of the above categories.

There are three primary problems in synchronization:
• Maintaining the integrity of data structures when multiple tasks need to update them
• Preventing an erroneous or inconsistent view of data as it is being updated
• Coordinating events between tasks

Shared Update Problem: A Simple Race Condition

Consider an imaginary banking system in which each teller is handled by a separate task. All tasks share a common in-memory data structure which represents the state of all the depositors' accounts. The routine *deposit* updates an account balance:

```
struct account {
          int balance;
          char *name;
          /* etc */
};

deposit(ap,amount)
struct account *ap;
int amount;
{
        ap->balance += amount;
}
```

In this example, the account structure is in memory shared among multiple tasks. On the other hand, the local variable *amount* is private to a particular instance of *deposit*; it is in private memory which is not accessed by any other task (either as an automatic variable maintained on the task's stack or, if the compiler so chooses, held in a register which is part of the task's context).

Now, consider the assembly language instructions implementing the above function (as seen using GCC on 32-bit X86 architecture):

```
deposit:
        pushl   %ebp                            #save ebp register
        movl    %esp, %ebp                      #ebp becomes frame pointer
        movl    8(%ebp), %eax                   #move &ap.balance to eax
        movl    12(%ebp), %ecx                  #move amount to ecx
1       movl    (%eax), %edx                    #FETCH current val to edx
2       addl    %ecx, %edx                      #ADD amount
3       movl    %edx, (%eax)                    #STORE result
        popl    %ebp                            #restore ebp
        ret                                     #return from fn call
```

Note the three instructions with labels 1, 2 and 3, which execute the FETCH, ADD and STORE operations respectively. Abstractly, any computation operation involving memory operands must be performed in these three distinct steps.

Consider an example case where *deposit* is executed simultaneously in two tasks, with *amount*=100, and *balance*=100 before execution begins. The following schedule illustrates proper execution:

Balance	Task A	Task B
100	BEFORE	BEFORE
100	1:fetch balance (=100)	
100	2:add (=200)	
100	3:write balance (200)	
200		1:fetch balance (=200)
200		2:add (=300)
200		3:write balance (300)
300	AFTER	AFTER

There are only two possible sequences of execution which yield the correct answer (300->balance): A1A2A3B1B2B3 or B1B2B3A1A2A3. There are several sequences which yield the wrong answer, e.g.:

Balance	Task A	Task B
100	BEFORE	BEFORE
100	1:fetch balance (=100)	
100	2:add (=200)	
100		1:fetch balance (=100)
100		2:add (=200)
100		3:write (200)
200	3:write (200)	
200	AFTER	AFTER

In this example, between statement 1, which reads the previous value, and statement 3, which writes the new value back, the other task could interlope.

The crux of the problem is that during this window of opportunity, there are multiple copies of the data (the balance). Which one is the "right" one??

Particularly tricky is that the correct operation in this case depends on the essentially random sequence of task scheduling. This is known as a **race condition**. The code could work flawlessly for years on a particular operating system or piece of hardware, and then fail suddenly when the system is updated, the hardware is upgraded, or even when the system load changes resulting in a shift in scheduling order.

Atomicity

The fundamental problem is that the update of the balance is not performed atomically. An operation is deemed to be **atomic** if and only if it can only either run to completion without being interrupted or, if it fails to run, has no effect. Because the update above is not performed atomically, simultaneous or overlapping update requests can interfere with each other.

If we could make the ADD atomic, then there would never be a time where multiple and inconsistent copies of the data would be visible.

(Although this example is somewhat contrived because many processors provide an atomic add instruction, let us continue with it as it is structurally equivalent to more complicated, real-world problems.)

Critical Regions and Mutual Exclusion

In the example above, we can identify the time between and including the execution of step 1 and the completion of step 3 as a **critical region**. Data consistency is vulnerable during a critical region because a private version of a shared object exists or because a shared data structure is in an intermediate state.

Is it necessary to enforce **Mutual Exclusion** in order to guarantee that at most one task is in the critical region at any given time. There are two ways to view mutual exclusion:
• The code-oriented view: mutual exclusion is applied to the critical region between and including steps 1 and 3 of the `deposit` function.
• The data-oriented view: we say that access to a particular `struct account` must be mutually exclusive.

Implementing MUTEX by Disabling Interrupts or Signals

On a uniprocessor system, mutually exclusive access to the critical region can be effected

by preventing any other task from taking control of the cpu during that time. When considering a single-threaded conventional UNIX process, this can be done by blocking all signals upon entry to the critical region, then restoring the previous signal mask on exit:

```
deposit(ap,amount)
struct account *ap;
int amount;
{
 sigset_t oldmask,newmask;
          sigset_fill(&newmask);                 /* Set of all possible signals */
          sigprocmask(SIG_BLOCK,&newmask,&oldmask);       /* Block */
        ap->balance += amount;
          sigprocmask(SIG_SETMASK,&oldmask,NULL);                   /* Restore */
}
```

Likewise, if this example were running within the kernel on a uni-processor machine, analogous protection can be obtained by masking all interrupts on entry to the critical region, then restoring the mask on exit.

However, on a multi-processor system, disabling interrupts only affects the local processor. It would still be possible for `deposit` to be running simultaneously on another processor. At the user-level, blocking signals would not be effective in a multi-threaded environment, or where multiple independent processes share a memory region, because there are other threads of control besides signal handlers.

Spin Lock Mutex

A spin lock is a simple form of mutual exclusion which does not rely on blocking interrupts or signals. The two essential elements of a spin lock are:
• The lock itself, which is simply a variable that has two possible states: locked and unlocked.
• The lock contention mechanism: a tight retry loop.

A task desiring to enter a critical region or access a mutex-protected data structure checks the lock variable. If it is unlocked, it sets the variable to locked and proceeds. If it finds the variable in the locked state, the task executes a tight loop, re-examining the variable until the variable indicates unlocked. On exit from the critical region, the task sets the variable to the unlocked state.

Spin locks only make sense if:
• The critical region is brief so the total number of retries is expected to be small.
• The tasks can not get "stuck" in the critical region thus resulting in an endless spin lock loop.

• The tasks in contention are able to execute in parallel. It would make no sense for a kernel interrupt handler to use a spin lock on single-processor machine, because if the interrupt were to arrive while the kernel was already inside the critical region, the interrupt handler would loop forever waiting for an event which can never come because it requires that the handler return and execution in the main thread resume. We'll see that where spin locks are used inside the Linux kernel, they are protected by a conditional compilation.

As a first-cut approach, we could do the following:

```
struct account {
          int spinlock;                /* 0 is unlocked, NZ is locked */
          int balance;
          char *name;
          /* etc */
};

deposit(ap,amount)
struct account *ap;
int amount;
{
          /* BEGIN CRITICAL REGION */
          while (ap->spinlock!=0)
                    ;
          ap->spinlock=1;
         ap->balance += amount;
          ap->spinlock=0;
          /* END CRITICAL REGION */
}
```

Unfortunately, we have fallen into the same atomicity trap as we had before! Namely, task A could find ap->spinlock clear, but before it has a chance to lock it by setting it to 1, task B could come along and also find 0 in that memory location.

Atomic Instructions

What is required is an instruction which can **atomically test and set.** Such an instruction has the following schematic operation:

```
TAS(int *addr):
          mask interrupts  (this CPU)
          lock memory bus  (all CPUs)
          invalidate other CPU's local caches on addr
          if ((R= *addr)==0)
          {
                    *addr=1
          }
          release memory bus
          unmask interrupts
          return R
```

On a multi-processor system, access to the shared memory pool is coordinated by a **memory arbiter**. When an atomic instruction such as TAS is executed, the local processor locks the memory bus, which instructs the arbiter to prevent any other processor from accessing memory during this critical region. The use of write-back caching introduces additional hardware-level concerns. The hardware must provide a way for this CPU to broadcast a message to all other CPUs causing them to invalidate any cached copy of the memory containing the lock which they might have in their local cache(s).

Processors may have other atomic instructions. For example, in the X86 architecture, many opcodes can be made atomic by prefixing them with the LOCK byte (0xF0). For some of the synchronization methods shown in this unit, these additional atomic instructions may streamline implementation. However, with just one atomic instruction, such as TAS, all other mechanisms can be built up from it.

As an aside, on processors that completely lack any kind of atomic instruction, it is still possible to achieve mutual exclusion through a method known as *Peterson's Algorithm*. This method quickly becomes very awkward when more than two tasks need to be synchronized. This method is now largely of theoretical and historical interest only, since almost all modern processors have atomic instructions (and those that do not typically do not run software that has synchronization issues).

Given that we have an atomic test-and-set operation, our example code can be protected with it:

```
/* DATA-ORIENTED approach */

deposit(ap,amount)
struct account *ap;
int amount;
{
          /* BEGIN CRITICAL REGION */
          while (TAS(&ap->spinlock)!=0)
                    ;
          /* When we get here, we are the only task that could be
           * accessing this struct account */
         ap->balance += amount;
          ap->spinlock=0;
          /* END CRITICAL REGION */
}
```

Code-based or Data-based?

Mutual exclusion could be viewed, as discussed above, as applied to a particular data structure, or applied to a particular region of code. Because the spinlock in the example above is embedded in the `account` data structure, we would tend to think of this as data-oriented locking. Here is the same example, re-written as code-oriented:

```
/* CODE-ORIENTED approach */
```

```
deposit(ap,amount)
struct account *ap;
int amount;
{
  static int deposit_spinlock=0;       /* Protects deposit crit region */
          /* BEGIN CRITICAL REGION */
          while (TAS(&deposit_spinlock)!=0)
                    ;
          /* When we get here, we are the only task holding the deposit_spinlock */
         ap->balance += amount;
          deposit_spinlock=0;
          /* END CRITICAL REGION */
}
```

Unfortunately programmers are often not clear enough about their locking paradigms, and this can cause subtle yet fatal errors when other programmers attempt to write code within the same framework (such as within the kernel). A comment is always in order when locking is used, explaining very clearly what the locking paradigm is and what code and/or data are being locked!

Mutex locking with blocking/yielding

In a multi-processor system, spin locks are reasonably efficient because the task holding the lock will soon release it and only a few cycles will be wasted. However, on a uni-processor system, if task A comes along and finds that the lock is currently held by task B, it means that task B is currently pre-empted, waiting for its turn at the lone CPU. Therefore, things would actually move along faster if task A **blocked** and relinquished the CPU to another runnable task (perhaps task B).

We can also consider, on a multi-processor system, task A holding a mutex for a relatively long time (say a complex data structure change). Task B, running on another process, finds the mutex locked. With spin locks, task B wastes CPU time that could otherwise be allocated to another task C which is ready to run.

Let us define `YIELD` as a function which suspends execution of the current task (and potentially allows another task to execute). When this function returns, the current task has been scheduled again, but this does not imply that the lock is now free. We can define three mutual exclusion locking primitives:

```
mutex_lock(struct mutex *lp)
{
          while (TAS(lp->lock))
                    YIELD(lp);          //includes arranging for wakeup
          return 1;
}

mutex_trylock(struct mutex *lp)
{
          return !TAS(lp->lock);
}
```

```
mutex_unlock(struct mutex *lp)
{
          lp->lock=0;
          WAKEUP(lp);         // wake up any tasks block on mutex *lp
}
```

As we will see in later units, forms of these mutex lock primitives are found both in the kernel code and in the *POSIX Threads* library which provides support for multi-threaded user-level programming. In both cases, the mutex lock is a considerably more complicated data structure as it must store the identifiers of any threads which have blocked on trying to acquire the mutex. When the thread holding the mutex executes `mutex_unlock`, these blocked threads must be notified of this event and run again. This is an example of the **sleep and wakeup** problem. We will have to defer discussion of the mechanics of sleeping and waking up until a subsequent unit.

The deadlock problem

Deadlock occurs when task A is waiting for task B, and vice-versa, or, more generally, there exists any cycle in the wait-for graph depicting the state of all the tasks. Consider the example of tasks A and B which "simultaneously" attempt operations OA and OB, respectively, both requiring locks on resources R1 and R2:

```
OA()
{
         mutex_lock(&R1);
         mutex_lock(&R2);
         /*whatever*/
         mutex_unlock(&R2);
         mutex_unlock(&R1);
}

OB()
{
         mutex_lock(&R2);
         mutex_lock(&R1);
         /*whatever*/
         mutex_unlock(&R1);
         mutex_unlock(&R2);
}
```

Consider this schedule (recall that lock and unlock are atomic primitives)

A	B
lock R1	
	lock R2
	wait for R1
wait for R2	

Deadlock Avoidance

One approach to deadlocks is **deadlock avoidance**. Deadlock can be avoided if locks are **strictly ordered** and all tasks acquire locks in this same order. If any lock can not be obtained, the task must release ALL held locks and try again:

```
obtain_locks()
{
 retry:
        for(i=0;i<N;i++)
        {
                if (!mutex_trylock(&mut[i]))
                {
                        for(j=i;--j>=0;)
                                mutex_unlock(&mut[j]);
                        goto retry;
                }
        }

}
```

Deadlock Detection

Deadlock avoidance isn't always possible for the general case. If the code participating in the potential deadlock isn't written by the same person, or cooperating parties, then it is impossible to enforce ordering. Since the Linux kernel is controlled by one entity, deadlock avoidance is practiced within it for kernel code. But for general-purpose user-level applications, deadlock detection is required:

When an underlying operating system kernel is available to provide the locking primitives, it is possible to perform **deadlock detection**. The operating system can maintain a data structure representing the wait-for dependencies between tasks, and can traverse the graph whenever a lock is requested to see if it would result in deadlock. If so, all tasks involved in the deadlock can be signaled. Deadlock detection is very typically seen in database applications, and in the file-level locking primitives provided by UNIX system calls such as `fcntl`.

Live Lock

A term often used as a corrollary to deadlock is "livelock". When two or more tasks are in conflict with each other such that their states continue to change but eventually return to a previous state without any possibility of ever terminating, that is considered livelock. Another way of defining livelock is an endless loop involving two or more inter-dependent tasks.

Livelock is a complex problem that often perplexes kernel designers and can come in many forms. Resource starvation often triggers livelock. The problem of "thrashing"

seen in Unit 5 is a form of livelock, although it may not meet the strict formal definition, in that two or more memory-starved processes are stuck in a loop of stealing each others' pages.

With respect to mutex locking, there is a formal method to prevent livelock which is similar to the formal method for preventing deadlock:
a) grab locks in a strict order which is consistent for all tasks involved
(b) upon finding a mutex already locked, release all previously acquired locks in the exact reverse of the acquisition order and do not attempt to re-acquire any locks until this is done
(c) the locking primitive must be defined so that when multiple tasks attempt to obtain the lock at the same time, exactly one succeeds. Obviously if more than one gets through, that is a serious error. But if the primitive is defective in that all conflicting tasks fail and nobody gets the lock, that situation can lead to livelock.

Given these requirements, the deadlock avoidance example immediately above is safe against livelock, because we have defined mutex_trylock in such a way that condition (c) is satisfied. In practical applications, livelock and deadlock can creep in because one of these three conditions is violated and that is not immediately obvious from code inspection. We'll see an example soon in the "inconsistent views" section.

```
/* Example of potential livelock */
OA()
{
          for(;;)
          {
                    if (!mutex_trylock(&R1)) continue;
                    if (!mutex_trylock(&R2))
                    {
                              mutex_unlock(&R1);
                              continue;
                    }
          }
         /*whatever*/
         mutex_unlock(&R1);
         mutex_unlock(&R2);
}

OB()
{
          for(;;)
          {
          /* Note that we aren't trying the locks in a consistent order */
          /* This potentially leads to livelock */
                    if (!mutex_trylock(&R2)) continue;
                    if (!mutex_trylock(&R1))
                    {
                              mutex_unlock(&R2);
                              continue;
                    }
```

```
          }
        /*whatever*/
        mutex_unlock(&R2);
        mutex_unlock(&R1);
}
```

Fine vs Coarse Grain Locking, Permissive Locking

If the only object of synchronization was to prevent data corruption, we would always use the biggest, most conservative possible lock. However, another concern is performance. In many real-world cases, a strict mutex lock is too restrictive, in that it prevents one task from getting anything done whatsoever while another task holds a mutex that it needs. We would like to have additional synchronization mechanisms which can be applied at a finer level, and potentially allow more productive simultaneity. In general, we'd like to lock the least amount possible without compromising integrity. Our first example of improving locking granularity will be reader/writer locking.

Read/write locking

When an object is protected by a mutex, this will prevent two tasks from reading the object simultaneously. Since no harm is done by this, it would be desirable to allow it. However, when a task wants to write to the object, that should lock out both other writers and readers, since the object may be in an inconsistent state during the write action.

With read/write locking, any number of tasks may hold a **permissive-** aka **read-only lock** but only one task may hold the **exclusive lock** which allows writing, and only if there are no permissive locks currently held. One possible implementation building on spin locks and the YIELD/WAKEUP mechanism is:

```
struct rwlock {
          int spinlock;      // Protects next two fields
          int readers;
          int writers;
          /* And more stuff for managing blocking/waking */
}

rw_rdlock(struct rwlock *lock)
{
          for(;;)
          {
                    while (TAS(&lock->spinlock)!=0)
                              ;
                    if (lock->writers==0) break;
                    lock->spinlock=0;
                    YIELD();
          }  //spinlock is held on loop exit
```

```
          lock->readers++;
          lock->spinlock=0;
}

rw_wrlock(struct rwlock *lock)
{
          for(;;)
          {
                    while (TAS(&lock->spinlock)!=0)
                              ;
                    if (lock->writers==0 && lock->readers==0) break;
                    lock->spinlock=0;
                    YIELD();
          }  //spinlock is held on loop exit
          lock->writers++;
          lock->spinlock=0;
}


rw_unlock(struct rwlock *lock)
{
          while (TAS(&lock->spinlock))
                    ;
          if (lock->readers>0) lock->readers--;
          lock->writers=0;
          WAKEUP ANY TASKS WAITING ON lock
          lock->spinlock=0;
}
```

The Linux kernel often has "clever" implementations of synchronization mechanisms based on other atomic instructions which are available, especially on the X86 processor family which is its primary target. Within the Linux (32-bit) kernel, reader/write locks are implemented as a 32 bit value. The least significant 24 bits are used as a counter holding the two's-complement of the number of read locks currently held. Bit 24 is the boolean flag and is 1 when all locks are released. Therefore, the value 0x01000000 represents that the read/write lock is unlocked. When the write lock is held, the value is 0x00000000. If, e.g. two tasks hold read locks, the value would be 0x00FFFFFE. (Bits 31-25 are used for bitwise flags by the kernel)

To obtain a read lock, the lock word is atomically decremented. (The X86 architecture has an atomic ADD instruction. Other architectures that lack this must emulate it by using a spin lock to protect a non-atomic add) If this resulted in a zero or negative value, then the lock word must have previously been 0 (write lock already held), therefore the task atomically increments the lock word to undo the previous operation, then waits until the lock word becomes non-zero, then attempts to obtain the lock again.

To obtain a write lock, the value 0x01000000 is atomically subtracted from the lock word. If the result is 0, then it is known that the previous value must have been 0x01000000, i.e. no readers and writers. Furthermore, the current value is now 0, and thus the write lock has been obtained. If the result is non-zero, the task undoes the

operation by atomically adding 0x01000000 and then tries again.

Example of reader/writer locks

In our previous bank account system, we can refine the locking so that write locks are used to make a deposit, but read locks are used to look up a balance.

```
deposit(account,amount)
struct account *account;
{
        rw_wrlock(&account->rwlock);
          account->balance+=amount;
          rw_unlock(&account->rwlock);
        return 1;
}

int viewbalance(struct account *account)
{
 int bal;
          rw_rdlock(&account->rwlock);
          bal=account->balance;
          rw_unlock(&account->rwlock);
          return bal;
}
```

The problem of inconsistent views

The following example illustrates the use of read/write locks, but also introduces a subtle problem:

```
transfer(from,to,amount)        /* Task A /
struct account *from,*to;
{
        if (from==to) return -1;        /*Nonsense operation*/
          for(;;) /* Deadlock avoidance, possible livelock, see text */
          {
                    rw_wrlock(&from->rwlock);
                    if (rw_try_wrlock(&to->rwlock)) break;
                    rw_unlock(&from->rwlock);
          }
        if (from->balance < amount)
        {
                rw_unlock(&to->rwlock);
                rw_unlock(&from->rwlock);
                return -1;
        }
        from->balance-=amount;
        to->balance+=amount;
        rw_unlock(&to->rwlock);
        rw_unlock(&from->rwlock);
        return 1;
}

totalize()              /* Task B */
{
```

```
 int i,tot;
        tot=0;
        for(i=0;i<n_acct;i++)
        {
                rw_rdlock(&accounts[i].rwlock);
                tot+=accounts[i].balance;
                rw_unlock(&accounts[i].rwlock);
        }
        return tot;
}
```

Task B runs periodically to provide a report of the sum total of all accounts in the system. Task A runs from time-to-time to transfer an amount from one account to the other. Clearly task B is read-only with respect to the `balance` field, while task A modifies it. Although the example above protects against any one account having the wrong balance as a result of two instances of task A overlapping, by acquiring writer locks on both accounts before changing them, it does not correctly protect task B against the problem of **inconsistent views**.

Aside: The use of rw_try_wrlock is to address a potential deadlock. Consider what might happen if the blocking rw_wrlock were used on both the to and from account. Numerous deadlocks would be possible. E.g. task A transfers from 100 to 200, B from 200 to 300, and C from 300 to 100. Each task grabs the "from" lock and now A waits for B which waits for C which waits for A: DEADLOCK!

However, the solution presented is not provably correct against LIVELOCK. Consider A transfers from 100 to 200 and B from 200 to 100 and these operations commence simultaneously. A locks 100, B locks 200, A finds 200 locked, B finds 100 locked, A releases 100, B releases 200. A locks 100, B locks 200, and so on....While this is an unlikely situation, to protect against it, we'd either have to introduce a "tie-breaker", e.g. after failing to obtain the second lock, the task waits a random amount of time or an amount of time based on its unique PID; or we could strictly order the locks by checking the memory addresses of the from and to variables, and always locking the one with the lower address first, regardless of whether that is the from or the to.

Suppose task B has just visited account 100 and incremented the grand total by the balance in account 100. Task B releases the reader lock on account 100. Then task A comes along to transfer an amount from account 100 to account 105. Task A will find both accounts unlocked because task B is done looking at #100 and hasn't gotten to #105 yet. The result is that the amount being transferred will be counted twice by task B.

The reason for the failure is that when task B runs, it in fact requires a reader lock on the entire data structure, not just the individual elements of it. Here is a conservative approach to this problem:

```
struct rwlock totalize_lock; // Assume initialized to unlocked

transfer(from,to,amount)        /* Task A /
struct account *from,*to;
```

```
{
        if (from==to) return -1;        /*Nonsense operation*/
          rw_wrlock(&totalize_lock);
          for(;;)
          {
                    rw_wrlock(&from->rwlock);
                    if (rw_try_wrlock(&to->rwlock)) break;
                    rw_unlock(&from->rwlock);
          }
        if (from->balance < amount)
        {
                rw_unlock(&to->rwlock);
                rw_unlock(&from->rwlock);
                return -1;
        }
        from->balance-=amount;
        to->balance+=amount;
        rw_unlock(&to->rwlock);
        rw_unlock(&from->rwlock);
          rw_unlock(&totalize_lock);
        return 1;
}

totalize()              /* Task B */
{
 int i,tot;
          rw_rdlock(&totalize_lock);
        tot=0;
        for(i=0;i<n_acct;i++)
        {
                rw_rdlock(&accounts[i].rwlock);
                tot+=accounts[i].balance;
                rw_unlock(&accounts[i].rwlock);
        }
          rw_unlock(&totalize_lock);
        return tot;
}
```

Unfortunately while this approach prevents the inconsistent view problem, it suffers from terrible performance. While the totalize task is running, all updates to the accounts are locked out. Since the totalizer is a potentially long-running task, this could result in unacceptable delays.

On database systems this problem of inconsistent views is addressed with a Multi-Version Concurrency Control (MVCC) approach, in which each reader is given a private and consistent view of the data until all write activity has completed. While this approach is very interesting, it is completely unsuitable for the synchronization problems found within a kernel, because it relies on a single serializer process (the DBMS daemon process) to arbitrate access and create this consistent view. The kernel does not have the benefit of this external arbiter.

Seqlocks

A possible solution is **optimistic synchronization**, which makes the assumption that most of the time there will be no conflict, and therefore the cost of fixing things when a conflict does occur is more than outweighed by the savings in avoiding excessive locking.

We will use a type of optimistic synchronization which is used in certain places in the Linux kernel, where it is given the name **seqlock**. A seqlock consists of a plain mutex lock and an integer sequence number field, which is initialized to 0. The lock is similar to reader/writer locks, in that different semantics are given to readers vs. writers.

A writer obtains a seqlock by first obtaining the spinlock mutex, and then atomically incrementing the sequence counter by one. To release the seqlock, the counter is incremented once again, and then the mutex is released.

Therefore, if the sequence counter is odd, then the lock is held by a writer. A reader trying to obtain a read lock first examines the sequence counter (without any spin lock or other protection), saving a copy to a local register. If the value read was odd, then the reader waits. If the value is even, the reader is able to enter its critical region and access the data. (This even/odd waiting in and of itself does not solve the entire problem. It merely avoids having the reader make an obviously wasteful incursion into the critical region when it knows that the writer is already there.) At the conclusion of the reader's work, it examines the sequence counter again. If the number has changed, then a writer has interloped, and the reader must try the entire process all over again. This is the other essential part of optimistic synchronization: the ability to reliably detect when the optimism was unfounded, and the ability to harmlessly repeat the operation with hope for a better outcome the next time.

In the previous example, the reader is task B which is calculating the total balance, while task A is a writer. Let us re-write this using seqlocks.

```
/* Seqlock schematic definition */
write_seqlock(struct seqlock *s)
{
          spin_lock(&s->lock);
          atomic_add(&s->count,1);
}

write_sequnlock(struct seqlock *s)
{
          atomic_add(&s->count,1);
          spin_unlock(&s->lock);
}

read_seqbegin(struct seqlock *s)
{
  int a;
          while ((a=s->count)%2)
                    YIELD();
```

```
          return a;
}

read_seqretry(struct seqlock *s,int orig)
{
          return s->count != orig;
}

seqlock_t seqlock;           // Global lock for whole system

transfer(from,to,amount)        /* Task A /
struct account *from,*to;
{
        if (from==to) return -1;        /*Nonsense operation*/
          write_seqlock(&seqlock);
          for(;;) /* Deadlock avoidance */
          {
                    rw_wrlock(&from->rwlock);
                    if (rw_try_wrlock(&to->rwlock)) break;
                    rw_unlock(&from->rwlock);
          }
        if (from->balance < amount)
        {
                rw_unlock(&to->rwlock);
                rw_unlock(&from->rwlock);
                return -1;
        }
        from->balance-=amount;
        to->balance+=amount;
        rw_unlock(&to->rwlock);
        rw_unlock(&from->rwlock);
          write_sequnlock(&seqlock);
        return 1;
}

totalize()              /* Task B */
{
 int i,tot;
 int seq;
        tot=0;
          do {
                    seq=read_seqbegin(&seqlock);          //read first seq#
                    for(i=0;i<n_acct;i++)
                    {
                              rw_rdlock(&accounts[i].rwlock);
                              tot+=accounts[i].balance;
                              rw_unlock(&accounts[i].rwlock);
                    }

          } while (read_seqretry(&seqlock,seq));          //compare to current seq#
        return tot;
}
```

Seqlocks are useful in many places in the Linux kernel. One straightforward example is the `ps` command, which in turn relies on the kernel's `/proc` pseudo-filesystem to return a directory listing in which each node represents a currently running process. This is a reader type of task -- in no way does the process listing modify anything. On the other hand, the fork, exit and wait system calls do change the list of processes, either inserting a new process or removing a dead process. These writer tasks perform the write_seqlock primitives on a seqlock which protects the entire process list, once before they modify it, and again when the modification is complete. The reader task checks the seqlock, assembles a local copy of the list of pids, then checks it again. If the seq# has changed, then the list is not valid, and the reader repeats the attempt and walks the list again. Eventually, unless the system is absurdly busy with fork/exit activity, the reader will assemble the complete pid list without disturbance. This is a far better alternative than having the ps command effectively freeze out any forks or exits.

Task Coordination

Our third major area of synchronization problems concerns coordination among tasks that are dependent on each other to proceed. The classic example is the producer and consumer connected via a FIFO. The consumer can not proceed until the producer puts an element into the FIFO. But if the producer puts elements in faster than they can be consumed, the producer has to wait for the consumer to drain the FIFO. There are several primitives that can be used for task coordination. We will look at two: semaphores and condition variables.

Semaphores

Semaphores are a classic inter-task synchronization mechanism first introduced by Dijkstra in 1965. They are most useful in coordinating producer/consumer tasks or wherever access to a fixed number of resources must be controlled.

A semaphore is a non-negative integer. The integer represents a count of the number of "resources" available and is initialized with that number. When the initial value is 1, the semaphore acts as a mutex lock.

We will define three **atomic** operations:

> "P", which stands for "proberen te verlagen", Dutch for "try to decrease". The P operation checks to see if the semaphore is positive. If it is not, the caller blocks. Otherwise, the semaphore is decremented by 1. This check and decrement is atomic. The P operation is also sometimes called "wait" or "down" or "dec".
>
> "V", which stands for "verhogen", Dutch for "increase". The V operation increments the semaphore by 1. If the semaphore value is now positive, any

sleeping tasks are awakened. V is also called "inc" , "up" or "post".

"TRY", which attempts the P operation, but if the semaphore is not positive, returns immediately with a failure code.

In addition to the counter, a semaphore must contain some way of recording which tasks are blocking on it. The actual blocking and unblocking are not part of the semaphore per se, but must be provided by the operating system.

```
/* A simplistic coordination example using semaphores to manage
 * consumption of a "scarce resource"
*/
struct semaphore okmem;

initialize()
{
          sem_init(&okmem,1024);                /* We have 1024 "slots" */
}

make_something()
{
          sem_wait(&okmem);
          /* allocate the thing */
          return (thething);
}

destroy_something()
{
          sem_inc(&okmem);
}
```

Condition Variables

Condition variables are used to suspend tasks which need to wait for a specific condition (aka event) to occur before proceeding. When another task generates the event, it wakes up any tasks which are sleeping on the condition variable. An underlying operating system must provide sleep and wakeup primitives.

The condition variable facilities used in the example below are found in the POSIX threads library and are available under Linux, Solaris and many other operating systems. The Solaris kernel relies heavily on condition variables. The Linux kernel uses a very similar facility called "wait queues", which will be discussed in a subsequent unit, and semaphores, which are discussed above.

Condition Variable Operations

A condition variable is always used in conjunction with a mutex lock. The following condition variable operations are provided:

• `pthread_cond_wait`: Block the calling task on the condition variable (first argument), which represents a specific condition or event the task wishes to wait for. A mutex is supplied as the second argument. The identity of the task calling `pthread_cond_wait` is stored in the condition variable data structure, the supplied mutex is released and the calling task is put to sleep atomically. The task sleeps until it is awakened by some other task calling `pthread_cond_broadcast` or `pthread_cond_signal`. Before returning control to the caller, `pthread_cond_wait` re-acquires the mutex. Schematically, we can express this primitive as:

```
pthread_cond_wait(pthread_cond_t *cp,pthread_mutex_t *mut)
{
        ATOMICALLY {
                note within *cp that we are interested
                pthread_mutex_unlock(mut)
                go to sleep
        }
        /* We have been awakened */
        pthread_mutex_lock(mut);
}
```

• `pthread_cond_signal`: Release at most one task blocked on the supplied condition variable. If more than one task is blocked on the condition variable, one is picked (the POSIX definition does not define which one but many implementations pick the longest-waiting.) If no tasks are blocked, this call has no effect, and is not "remembered". Should be called under the protection of the mutex for the condition variable, as discussed below.

• `pthread_cond_broadcast`: Release all tasks blocked on the supplied condition variable. If there are no blocked tasks, this call has no effect and is not "remembered". Should be called under the protection of the mutex for the condition variable, as discussed below.

Producer/Consumer problem using condition variables

```
volatile struct fifo {
          char buf[BSIZE];
          int next_write,next_read;
          int item_count;
          pthread_cond_t full,empty;   /* Condition variables */
          pthread_mutex_t mutex;                /* Common mutex for fifo */
} fifo;

initialize(struct fifo *fifo)
{
          fifo->mutex=PTHREAD_MUTEX_INITIALIZER;
          fifo->full=PTHREAD_COND_INITIALIZER;
          fifo->empty=PTHREAD_COND_INITIALIZER;
          fifo->next_write=0;
          fifo->next_read=0;
          fifo->item_count=0;
}

producer(struct fifo *fifo,char c)
{
          pthread_mutex_lock(&fifo->mutex);
          while (fifo->item_count>=BSIZE)                  /* Full */
                    pthread_cond_wait(&fifo->full,&fifo->mutex);
          fifo->buf[fifo->next_write++]=c;
          fifo->next_write%=BSIZE;
          fifo->item_count++;
          pthread_cond_signal(&fifo->empty);
          pthread_mutex_unlock(&fifo->mutex);
}


consumer(struct fifo *fifo)
{
 char c;
          pthread_mutex_lock(&fifo->mutex);
          while (fifo->item_count<=0)            /* Empty */
                    pthread_cond_wait(&fifo->empty,&fifo->mutex);
          c=fifo->buf[fifo->next_read++];
          fifo->next_read%=BSIZE;
          fifo->item_count--;
          pthread_cond_signal(&fifo->full);
          pthread_mutex_unlock(&fifo->mutex);
          return c;
}
```

When the condition variable is a simple counting expression, such as the FIFO example above, then condition variables and semaphores are virtually equivalent. But condition variables can also control more complicated boolean expressions which do not obey this count-up/count-down paradigm.

Some Golden Rules of Condition Variables

• **The condition variable is not the condition.** The condition is some boolean expression which is tested separately.
• **The condition must be re-tested.** Return from `pthread_cond_wait` does not guarantee that the condition is now true, because it is **not atomic** with respect to the condition: between the time that the task wakes up and the time that it re-acquires the mutex within `pthread_cond_wait`, another task might come along, acquire the mutex, and perform an operation which causes the condition to become false again. In the producer/consumer example, we see that the empty or full condition is tested with a while loop.
• **The mutex must be held while the condition is being tested**. The condition test itself is not necessarily atomic (e.g. the boolean expression is made up of && or || clauses). Furthermore, testing the condition and calling `pthread_cond_wait` are not inhererently atomic. They must be made atomic through the use of the mutex. Consider the FIFO example above without a mutex. Between the time that, e.g., the consumer finds the fifo empty in the while() test and the time it puts itself to sleep with pthread_cond_wait a producer could come along, write a character (rendering the empty condition false) and call pthread_cond_signal. Since the consumer has not reached cond_wait yet, the signal is lost and the consumer may sleep forever. This is known as the **lost wakeup problem**. By protecting access with a mutex, the test and the sleep effectively become atomic, preventing a lost wakeup.
• **Use** `pthread_cond_signal` , only when at most one awakened thread would be able to proceed. Otherwise use `pthread_cond_broadcast` to make sure all threads which could possibly proceed are woken up.

• **Hold the mutex when** `pthread_cond_signal` **or** `cond_broadcast` **is called**. If not, another thread may be testing the condition just as the wakeup arrives and would therefore miss the wakeup and potentially wait forever.

What is the Kernel?

In this unit, we will begin an in-depth exploration of the Linux kernel. The first and most obvious question is: just what exactly is the "kernel"?!

The Linux kernel is an unusual type of program. Most programs to which we are accustomed have one point of entry, run for some time, and then terminate. The kernel has one initial point of entry which is used at boot-time, and then multiple points of controlled re-entry. The termination of the kernel amounts to handing the CPU back to the firmware which either reboots or turns off the machine.

The kernel is compiled and linked just like an ordinary C program, in that it consists of many individual object (`.o`) files which are compiled from source code (the Linux kernel is written exclusively in C and assembly). Within the kernel is the equivalent of a `text` section, containing the executable code, as well as `data` section with variable initializers. Like an ordinary C program, when the kernel starts it sets aside space for uninitialized `bss` variables, and can dynamically allocate memory for other data structures. Also, like a user-level C program, the kernel can dynamically load and unload executable modules.

Unlike the C environment that is documented in the C language standards documents, the kernel does not have the standard library. There is no printf, no malloc, no fopen, etc. because these standard library functions would require an operating system to provide the services. There is a subset of library functions which is linked in the kernel such as strcpy. The kernel is also coded to avoid all use of the floating point registers, so functions such as pow() and sqrt() would never be seen in kernel code, and indeed the keywords `float` and `double` would generally be absent too.

At this point, it might be instructive to look at how the Linux kernel source code is arranged. The kernel can be compiled for a variety of target architectures. From the top-level directory, we see the following directories, each of which contains architecture-independent code. Thus the code below is only in C, not assembly. There is one top-level directory called `arch`, under which is one entry for each supported architecture, e.g. `arch/x86`, and below each of those is another set of subdirectories with similar structure to the architecture-independent set. Here is a description of these directories:

- `block`: A fairly small set of utility routines pertaining to block-level access to hard disks.
- `crypto`: Encryption functions.
- `drivers`: A vast set of routines for interfacing with I/O devices, including arch-neutral stuff such as the SCSI messaging layer, and support for specific peripherals, motherboards, etc.
- `fs`: Routines to support the file system, such as open, read, write system calls. Also

subdirectories for each supported file system type module.
• `include`: Header (.h) files
• `init`: Code which is executed when the system first boots.
• `ipc`: System V and POSIX IPC mechanisms
• `kernel`: A considerable amount of code including all of the synchronization, task control, signals, halting and restarting the system, etc.
• `lib`: A number of utility routines needed by different subsystems, including synchronization primitives such as spin locks and semaphores.
• `mm`: The virtual memory subsystem.
• `net`: Networking protocols
• `security`: Some extra stuff for "security-hardened" versions of the kernel.
• `sound`: Sound drivers and support

(there are a few other subdirectories which have to do with the kernel build and install process or other features but are beyond the scope of this course.)

Boot-up

The following describes how the linux kernel under X86-32 bit architecture is booted:
• The linux kernel is built from source and often included as a binary distribution. A valid kernel must be present to support the particular architecture of the machine that is being booted, and the particular device drivers that are needed at boot time. The kernel source code as described above is first compiled into an a.out file. However, a.out files mean nothing to the firmware, so additional preparation is necessary. The *image* of the text and data sections is pulled out of the compiled kernel. Typically, to save space on smaller bootable media, the kernel image is compressed. A bootable kernel consists of a small, uncompressed portion of bootable code followed by the compressed text/data image. This bootable kernel (typically called `vmlinuz` where the "z" stands for compressed) is installed on the bootable medium (hard disk, CD/DVD ROM, USB stick).
• To boot the system, the firmware ("BIOS") causes the kernel image to be read in from the boot device (e.g. the hard disk) into a specific place in physical memory. The firmware then transfers control to the starting address of the kernel.
• BIOS runs in supervisor mode, with address translation turned off (so all addresses used in code are physical) and with interrupts disabled. Hand-off to the kernel image is made in this condition. Thus the first part of the kernel's initialization routines have been compiled and linked with a specific address in mind which corresponds to the load address. The next task is to de-compress the image which is in one place in physical memory and write it to another location. For the (32-bit) x86 architecture, this is physical address `0x00100000`. A jump is then made to that physical address.
• At around this time, the kernel turns on virtual address translation. On the x86 32-bit architecture, the Linux kernel uses virtual addresses below 0xC000000 exclusively for user-level processes, and reserves the 0xC000000-0xFFFFFFFF range exclusively for

kernel memory. We'll see how this is exploited to accomplish fast access to kernel data structures. **Note: All parts of the kernel see the same virtual address mappings.** To get the kernel running on virtual, rather than physical addresses, page tables are created which map 0xC00000000 to physical address 0x00100000, and so on for higher addresses, until the entire static part of the kernel (text, data, bss) has been mapped. The static bss pages are explicitly zero-filled at this time. A temporary stack is set up to allow the kernel initialization code to run.

• The MMU is turned on and thereafter the kernel runs on virtual addresses.

• The physical page frames that are not part of the static part of the kernel become the page frame pool. The kernel is able to dynamically allocate memory for itself out of this pool, in addition to satisfying user-level page fault demands.

• The interrupt vector tables are appropriately initialized (see "Interrupt Handling in Hardware" below).

• The kernel then calls initialization routines for its various subsystems, e.g. virtual memory, file systems, scheduling. They allocate dynamic kernel memory as needed. An inventory of hardware devices is taken and the initialization routines for each device are called.

• Interrupts are now enabled.

• Now the kernel is able to provide its services, but has no user-mode processes to which they can be provided yet. The device which contains the root filesystem is determined from information passed during boot loading, and the root filesystem is mounted.

• The kernel creates process #1 and causes that process to exec/map a specific binary program (`/sbin/init`), running as the super-user (uid and gid = 0). This "init" program must reside on the root filesystem, and is the bridge between kernel initialization and user-mode initialization. It remains running for the life of the system.

• pid 1 is marked as ready to run and the kernel relinquishes control to the process scheduler which immediately selects pid 1 to run (since there is nothing else to do!) Control now exits kernel mode and the init process begins to execute at user level. Of course control immediately re-enters the kernel as init page-faults in its first page of executable text.

• First, `init` runs a series of programs and shell scripts which complete the initialization of the system. This includes locating and mounting other filesystem volumes as needed. Then, init spawns other user-mode programs which ultimately provide access to the machine for the end-user.

Note: on more recent Linux systems, the straightforward, traditional /sbin/init is replaced by a more unwieldy /bin/systemd. Either way, it is pid #1 which is responsible for user-level system startup

When booting up a multiprocessor machine, the BIOS starts the machine in single-processor mode. Fairly late in the kernel's initialization, the other processors are enabled. They are initially given the *idle task* to run. Once additional processes other than pid 1

are runnable, the other CPUs will start to pick up their loads.

Re-entering the kernel

We have seen that there is one initial entrypoint to the kernel, through the boot process. Thereafter, the kernel relinquishes control to user-mode processes. The kernel is entered again only through the interrupt mechanism.

We can broadly divide re-entry into two categories:
• **Synchronous**: When the interupt is raised because of the execution of a specific instruction, that is said to be synchronous. The Linux kernel calls these synchronous events **Exceptions**. Other similar terms are "Fault" and "Trap". Exceptions are said to occur in the context of a particular running process. The kernel associates a unique kernel-mode stack for each process (or to be precise, for each user-level thread), and so we can think of the kernel as containing one kernel-mode thread of control for each user-level thread, which gets activated when an exception is raised. Thus the handling of the exception in the kernel is in effect a controlled extension of the user-level thread into the kernel.
• **Asynchronous**: These events are not correlated with a specific instruction, but come from hardware devices. The Linux kernel calls these "Interrupts", which is somewhat ambiguous because Exceptions are also a form of interrupt. Asynchronous interrupts are handled on whatever kernel-mode stack happens to be active at the time.

In most operating systems literature, synchronous entries to the kernel are said to be "*top half*," and asynchronous are said to be in the "*bottom half*". Unfortunately the Linux kernel uses the term "bottom half" in an inconsistent manner, so we will avoid top/bottom half terminology to reduce confusion.

Exception Types

• Many exceptions are the result of program error. These are also called **faults**. A fault is defined as a condition which prevents the current opcode from being executed. The default behavior of the kernel for most faults is to post a signal to the offending process. Examples of such faults include: integer divide by 0 and illegal instruction.
• A **Page Fault** exception occurs when the hardware was unable to perform a memory access. On the x86-32 architecture, this same fault type is used both for when a valid PTE can not be found and for a protection violation, the distinction between the two being passed to the fault handler by hardware in a specific register. Page Faults are quite normal and not necessarily a sign of program error, as discussed in Unit 5.
• The kernel uses some exceptions to be clever and efficient about things. For example, the kernel usually does not bother with floating point registers. It turns the floating point unit off, and then if the process tries to use a floating-point operation, it raises an exception and the kernel knows that it needs to worry about it for that process. So these

exceptions are like a Page Fault, in that the kernel's handler must determine whether the exception is benign or the result of an errant program.
• When faults are resolved successfully by the kernel, control returns to user level and the previously faulted instruction is re-tried, and this time should succeed.
• A **System Call** is a particularly important type of exception, and we'll spend some time looking at system calls in detail.

Interrupt Types

• Hardware devices will raise interrupts to indicate that they have entered a ready or non-ready state, that data are available, or that a data transfer operation has completed. Each hardware device is assigned an Interrupt Request (IRQ) number (some devices may share IRQs).
• The kernel programs a chip on the motherboard to deliver a periodic "heartbeat" interrupt, e.g. every millisecond. This **Periodic Interval Timer (PIT)** is very important to the scheduler. Other time-based events in the kernel (e.g. network protocol retransmission timers) are derived from this clock source. (*Note: on modern Linux kernels supporting multi-CPU systems, this is complicated by the fact that different CPUs might be running at different clock rates. A more sophisticated mechanism is used, but the end result is the same: each CPU gets a periodic interrupt to drive the scheduling subsystem*)
• On a multi-processor system, one processor may execute a series of instructions which cause a hardware interrupt to be posted to the other processors. This is known as an **Inter-processor Interrupt (IPI)**. The Linux kernel uses this in several specific cases: (a) To shut down the system (b) to request that the other processors re-examine their task scheduling (covered in next unit) (c) to purge stale TLB entries.
• Hardware will deliver interrupts to notify the kernel of important system events, such as a hardware failure, battery power failure, etc. If continued system operation is possible, the kernel will ultimately translate these into user-mode events. Otherwise the kernel will print out an informative message and halt the system.

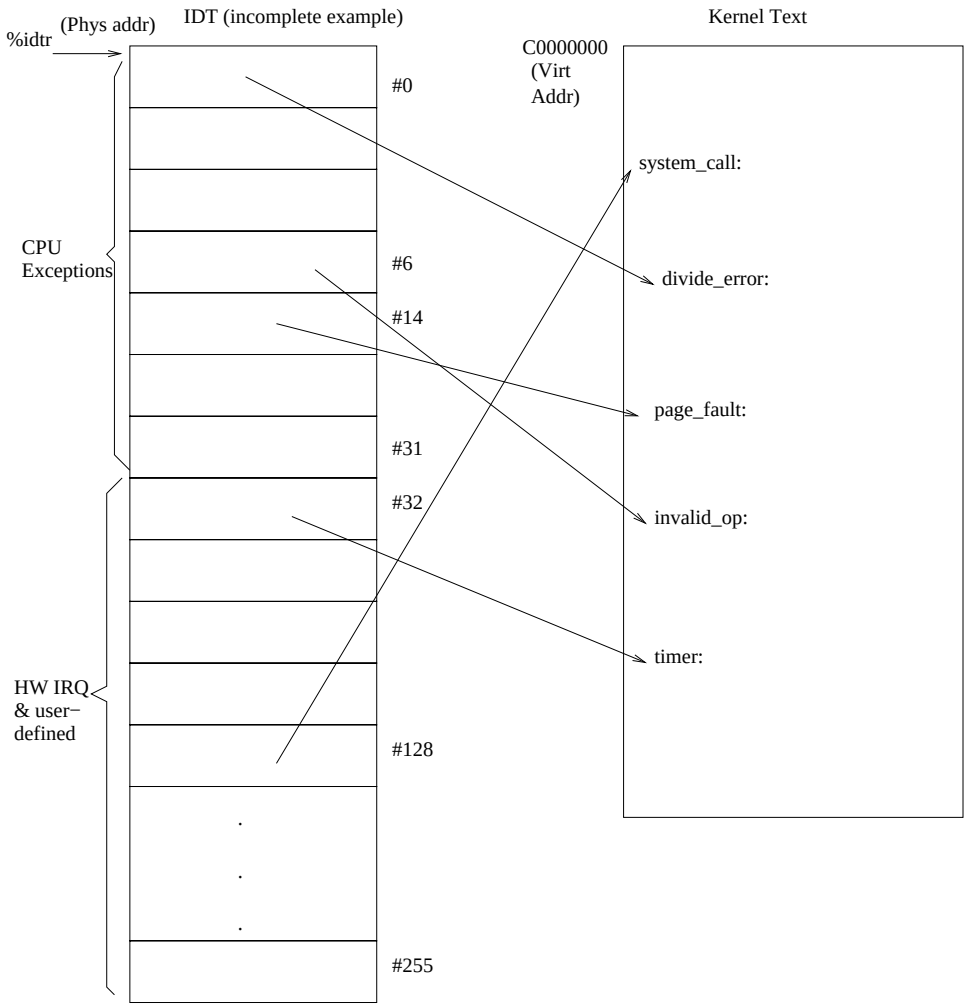
Important Shared Processor/Kernel Data Structures

The behavior of the CPU is hard-wired (or programmed in its microcode) and thus immutable to any software code including the kernel. The kernel is compiled for a specific architecture and knows about data structures that the processor expects to see in memory, which control how the processor reacts to certain events. Because the processor itself must respond to virtual memory lookup failures (page faults) these data structures are typically defined at physical, rather than virtual memory addresses. On the X86-32 bit architecture, these data structures are:
• The **Interrupt Descriptor Table (IDT)** is an array of 256 interrupt descriptors. The physical address of this array is set via the special `idtr` register, which can only be

accessed through special privileged instructions LIDT and SIDT. Although the structure of the IDT is baroque because of legacy X86 segmented memory model issues, it is basically a table of program counter (%eip) addresses, which are the handler addresses (virtual addresses) for each of the 256 possible interrupt or exception/fault vectors. Note that the X86 hardware doesn't distinguish between interrupts and exceptions (faults) with respect to the IDT. The kernel has pre-initialized all IDT entries to point to valid kernel functions before allowing user-level code to begin execution after system boot. The IDT is generally not modified after this.

• The **Task State Segment (TSS)** is a small data structure which resides inside a larger data structure known as the Global Descriptor Table (GDT). The Linux kernel does not utilize all of the functionality that is available with the TSS, but uses it simply to control the stack pointer address that will be used during interrupt/exception handling. The special `tr` register controls the location of the TSS and is generally set once. On multi-processor systems, there is a TSS for each processor.

• The page table, which has been described in unit #5. The `cr3` register controls the physical address of the highest-level page table (the Page Global Directory).



Interrupt and Exception handling in hardware

On the 32-bit X86 architecture, the following steps are taken by the processor in response to an interrupt or exception:

• Each interrupt or exception has an associated **vector** between 0 and 255. This is used to index the IDT, from which the handler address is fetched. The handler address is the virtual address of the first opcode of the associated handler function within the kernel. This function is generally written in assembly language. The kernel has made sure that all 256 entries have valid handler addresses. In the X86 architecture, vectors 0-31 are reserved for processor-generated exceptions, while 32-255 are user-defined and typically used for I/O devices, with a straightforward mapping between IRQ numbers and IDT vectors. Note that the system call vector 128 (decimal) falls within this range.

• When the processor is currently running in User mode, it fetches the TSS. Within the TSS is the new value (virtual address) of the stack pointer. This will be the kernel-mode stack, as described later. The kernel maintains a separate kernel stack for each task (this

includes each thread in a multi-threaded program), and makes sure the correct stack address was placed within the TSS before performing a context switch. If however, the processor was already in Supervisor mode at the time of the exception/interrupt, then the stack pointer is already within a valid kernel stack, and the TSS is not used by the processor.

• The processor now transitions to Supervisor mode (if not already in it), and the %esp register is pointing within the kernel-mode stack for the task.

• The processor pushes onto the kernel-mode stack the value of the old stack pointer register `%esp`, the flags/status register `%eflags`, and the program counter register `%eip`. The `%cs` and `%ss` registers (which have to do with how code and stack memory are accessed) are also saved. This set of 5 registers is critical because it is what the hardware relies on to determine where the stack is, and where the next instruction is.

• For certain types of exceptions, the processor pushes an error code onto the stack. This is used, e.g., to distinguish between a page translation fault and a page protection fault.

• The program counter location of the handler, which was fetched from the IDT, is now loaded into the `eip` register, effecting a jump to that entrypoint.

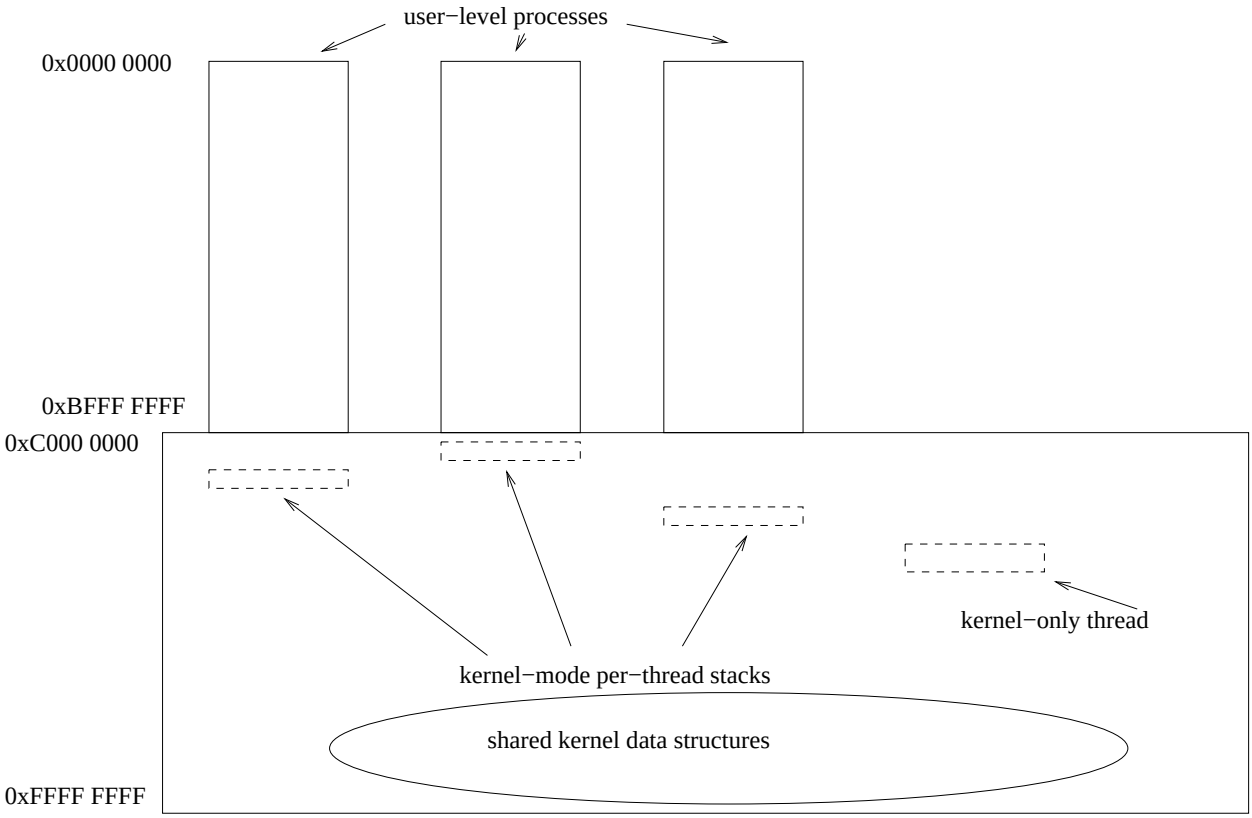
• Recall that the kernel controls the page tables, and establishes them for each process. The kernel virtual address space, 0xC0000000-0xFFFFFFFF, is always present in these page tables, but the User/Supervisor flags in the Page Table Entries are set so user processes can not directly access the kernel memory (this would be really bad....however user processes running as root can use a pseudo-device `/dev/kmem` to access kernel memory as if it were a file). This page table arrangement means that as soon as control enters the kernel, the shared kernel address space is immediately available.

Any interrupt or exception handler ultimately terminates by executing the special `iret` instruction, which:

• Pops the 5 registers from the stack which had been saved by hardware.

• Resets the privilege level to the previous value (because the eflags register has been restored from the stack)

• Resumes execution with the stack pointer now set back to the original stack. Interrupts occur between instructions, and so execution resumes at the next instruction. For exception returns, if the instruction caused a fault (e.g. Page Fault) then that instruction is re-started, otherwise the next instruction is executed.



Kernel control flow paths & pre-emption

Now that we've seen the types of kernel entrypoints, lets continue further to talk about how control flows within the kernel and back out again.

At any given moment, a CPU is either:
• Executing a user-level thread (process)
• Executing a kernel thread. This is identical to a user-level thread except that the virtual address space is entirely within the kernel.
• Handling an exception
• Handling an interrupt
• Temporarily halted because there is nothing else to do at the moment. We call this the "Idle" state.

While handling an exception or interrupt, it is possible that another exception or interrupt will arise. Thus the flow of control in the kernel is multiply re-entrant, and nested. E.g. while handling a system call (exception), a keyboard interrupt is received. The exception handler is suspened (by hardware) and the keyboard interrupt handler begins to run. While servicing that, a disk interrupt is received and handled. When the disk handler

finishes, the keyboard handler resumes and finishes, then the system call is allowed to continue.

To simplify kernel programming and synchronization issues, the Linux kernel is carefully coded so that kernel code never produces exceptions, with one caveat: During the handling of a system call (exception), a Page Fault (exception) may be raised. Thus we can say that exceptions will never arrive during interrupt handling, and will never arrive during exception handling other than a Page Fault during a System Call. An interrupt may occur at any time, because it is asynchronous.

Pre-emption means a potentially non-voluntary context switch between one thread (task) to another. Pre-emption of a user-mode task by another user-mode task is straightforward and generally happens as control is returning from the PIT (Timer/Clock) interrupt handler back to userland. A voluntary context switch may also happen because a system call encounters a blocking condition and therefore explicitly relinquishes the processor.

With regard to pre-emption of a task which is already executing in kernel code, this introduces complexity in the kernel's coding, especially with regard to locking. The traditional default configuration of Linux kernel is such that pre-emption does not take place in kernel code -- the pre-emption only occurs as the task is about to return to user mode from a system call, fault or interrupt.

When kernel pre-emption is enabled that changes the kernel code and therefore this is not a mode that can be toggled on the fly or even selected at system startup. The kernel must be compiled with pre-emption on or off. Here is an example illustrating the difference:
• Control has entered the kernel synchronously, e.g a system call or page fault, from task "A"
• While handling the system call or fault, an interrupt is received
• The handling of the interrupt causes another task "B" with better scheduling priority to become READY (e.g. disk I/O has completed).
• With pre-emption OFF, the interrupt handler completes and control resumes in the kernel code, handling the system call in the context of task "A". Once the system call finishes, upon return to user mode, pre-emption takes place (see below under "Deferred Return from System Call").
• With pre-emption ON, upon return from the interrupt handler to the kernel system call handler, the NEED_RESCHED flag (see below under "Deferred Return from System call") is noticed and a context switch takes place. Execution of task "A" is suspended in the kernel at the point in the system call handler where the interrupt happened to arrive (it could be any arbitrary point). Task "B" gets the CPU. At some later time, task "A" gets the CPU again, and the system call resumes.

The advantage of full kernel pre-emption is latency: a high priority task which wakes up need not wait for the system call to complete before it gets the CPU. The disadvantage is the added code complexity and locking, which can also introduce a slight performance penalty because it burns up CPU cycles. A compromise between these two extremes is

"voluntary pre-emption" where the kernel is coded with specific pre-emption points in system calls or fault handlers where it might take a "long time" (but not indefinite) to do something. At these points the NEED_RESCHED flag is explicitly tested and a context switch is voluntarily made if needed.

Making a System Call

We are now ready to discuss one particular kernel control path: the system call. We'll illustrate a fairly simple one: `getuid()`. A user-level program calls `getuid()` as an ordinary C function. This function is provided by the standard C library which is linked with all C programs. The `getuid()` function is written partially in C and partially in assembly language. This provides the "glue" between the user-level domain of the C program, and the kernel's system call API.

The 32-bit X86 API

Argument passing between user-level functions in C (X86-32) is via the stack. However, during a system call, the processor will be switching to a different, kernel stack. The user-level program obviously can not write to the kernel stack. Conversely, although the kernel *can* access the user-level stack memory, it would rather not, since that might create a Page Fault. The solution is to pass arguments to system calls in registers. The convention used on the x86-32 architecture is that the 32-bit arguments are passed such that the first argument is in the `ebx` register. The second is placed in the `ecx` register. The third through sixth are placed in registers `edx, esi, edi, ebp` respectively. If the number of arguments to the system call exceeds 6 (rare), then all of the arguments are placed on the user-level stack and the kernel receives just the address of that argument block.

Each available system call is assigned a specific number by the kernel. A given system call number also has a specification for what arguments are expected. The standard C library must therefore be compiled, at least in part, with an eye towards the exact architecture on which the program will be run. As the kernel evolves towards higher version numbers and new system calls are added, backwards-compatability with older code must be maintained. System call numbers are not re-used. If it is necessary to change the semantics of a system call, a new call is defined with a new number, and the kernel provides both versions. In the Linux 2.6.15 X86-32 version kernel, there were 294 defined system call numbers, by version 2.6.23 that number had grown to 325, and by 2.6.32 it had reached 338. There is no end in sight with 377 system calls in 4.9.34!

The system call number is passed to the kernel in the `eax` register. Now the user-level `getuid` function is ready to actually make the system call. There are no arguments to this particular system call, so the system call # corresponding to getuid (199 decimal) is put in eax and then a special instruction is used. There are two ways to make a system

call: using the `INT $0x80` software interrupt exception instruction, or using the `SYSENTER` instruction. We'll follow the former example. The reader is referred to the book *Understanding the Linux Kernel* for more information on the SYSENTER method.

The `INT $0x80` instruction causes an exception to be raised with vector code 128. The hardware then vectors to the kernel entrypoint in the IDT for vector 128, which the kernel had previously initialized to point to the (symbolic) kernel virtual address `system_call`. The code below is a simplified version of /usr/src/linux/arch/x86/kernel/entry.S:

```
system_call:
          pushl     %eax                          #contains system call #
          SAVE_ALL                                #macro to push important
                                                  #registers on the stack
          movl      $0xFFFFE000,%ebp              #mask SP to get to
          andl      %esp,%ebp                     #thread_info
          testw     $_TIF_SYSCALL_TRACE,TI_FLAGS(%ebp)     #test thread_info.flags
          jnz       syscall_trace_entry           #for syscall tracing on
          cmpl      $nr_syscalls, %eax            #bounds check
          jae       syscall_badsys
          call      *sys_call_table(0,%eax,4)     #indirect addressing
          movl      %eax,PT_EAX(%esp)             #poke return code into EAX slot
syscall_exit:
          cli                                     #temporarily mask interrupts
          movl      TI_flags(%ebp),%ecx           #get flags field of thread_info
 #ALLWORK_MASK includes all TIF_XXX thread info flags that indicate more
 #work might be needed before returning to user space
          testw     $_TIF_ALLWORK_MASK,%cx        #see if any flags are set
          jne       syscall_exit_work             #if so more work before exit
restore_all:
          RESTORE_REGS                            #pop registers from stack
          addl      $4,%esp                       #discard original eax
          iret                                    #return from interrupt
syscall_badsys:
          movl      $-ENOSYS,PT_EAX(%esp)         #poke error return code
          jmp       syscall_exit                  #simplified
syscall_exit_work:                                #simplified
          testb   $_TIF_NEED_RESCHED,%cl          #flags already in ecx
          jz        work_notifysig                #if clear, must be signal pending
work_resched:
          call      schedule                      #otherwise, task switch
          cli                                     #avoid missing an interrupt
          movl      TI_flags(%ebp),%ecx           #check flags again
          andl      $_TIF_WORK_MASK,%ecx
          jz        restore_all                   #OK to return to userland
          testb     $_TIF_NEED_RESCHED,%cl
          jnz       work_resched                  #still need rescheduling
work_notifysig:
          #we won't be covering this code in depth
```

The `SAVE_ALL` macros pushes all of the registers that the kernel is likely to clobber. In conjunction with the hardware pushes and the instruction `pushl %eax` just above, the kernel stack now looks like this:

```
0x00(%esp) - ebx            general-purpose registers,
0x04(%esp) - ecx            saved by
0x08(%esp) - edx            SAVE_ALL
0x0C(%esp) - esi            "
0x10(%esp) - edi            "
0x14(%esp) - ebp            "
0x18(%esp) - eax            " (will be syscall return value)
0x1C(%esp) - ds                      "
0x20(%esp) - es                      "
0x24(%esp) - fs                      "
0x28(%esp) - orig_eax       orig system_call number (used for syscall restart)
0x2C(%esp) - eip            program counter in user land, saved by hw
0x30(%esp) - cs             code segment register, saved by hw
0x34(%esp) - eflags         flags register, saved by hw
0x38(%esp) - oldesp         user-land stack pointer, saved by hw
0x3C(%esp) - oldss          user-land stack segment reg, saved by hw
```

The next instruction places a mask into register `ebp` and applies that mask to the stack pointer `esp`. Normally, in C programs, the ebp register has a very important function as the local frame pointer. However, we are still in an assembly language entrypoint, and thus ebp is available as a scratch register. The purpose of this masking operation deserves a considerable detour:

Kernel-mode stack and the thread_info structure

Recall that the kernel allocates an individual stack area for each user-level thread of control. On X86-32, the kernel stack is only two pages (8K). On X86-64 (as of Linux 4.X kernels) the kernel stack is either 4 or 8 pages (16K or 32K). For simplicity, we will assume an 8K stack. While this seems rather small, bear in mind that every bit of code in the kernel is carefully controlled. Large amounts of local variable space are discouraged, recursive programming is never used, and the depth of function call nesting rarely gets obnoxious. Therefore, the kernel programmers can rely on the fact that this 8K stack will not overflow.

Now, to compound this trickery, the kernel sticks a small data structure called `struct thread_info` at the limit of the stack, i.e. at the lowest memory address. The kernel stack pointer value stored in the TSS memory area by the kernel for each process/thread is the highest address of the allocated stack area. On entry to the kernel through an exception or interrupt, the kernel stack is empty, and the kernel stack pointer is thus furthest away from this `thread_info` data structure, and as kernel functions are called, the stack pointer gets closer to it, but should never be in any danger of over-writing it.

This arrangement of memory addresses means that on entry to the kernel, a simple masking operation of the stack pointer `%esp` yields the beginning of the

`thread__info` structure. That address is kept in the `%ebp` register for a while. Also, at any point in the kernel, the inline function `current_thread_info()` performs that same masking operation. Let's look into the `thread_info` struct (/usr/src/linux/arch/x86/include/asm/thread_info.h):

```
/* On recent Linux kernels */
struct thread_info {
        unsigned long           flags;          /* low level flags */
};

/* On older Linux kernels */
struct thread_info {
        struct task_struct      *task;          /* main task structure */
        struct exec_domain      *exec_domain;   /* execution domain */
        unsigned long           flags;          /* low level flags */
        unsigned long           status;         /* thread-synchronous flags */
        __u32                   cpu;            /* current CPU */
        int                     preempt_count;  /* 0 => preemptable */
        mm_segment_t            addr_limit;     /* highest valid VA */
          void                        *sysenter_return;  /* don't worry */
        struct restart_block    restart_block;  /* for syscall restart */
 #ifdef CONFIG_X86_32        /* Don't worry about these next two obscure lines */
        unsigned long           previous_esp;
        __u8                    supervisor_stack[0];
 #endif
          int                         uaccess_err;       /* err from accessing user mem*/
};
```

There isn't much room on the kernel stack, so `thread_info` is pretty small. What is kept in there is only what is needed by the assembly language entry/exit routines. The kernel needs to keep a lot more information about a process, so it allocates another data structure called `struct task_struct` which we have seen in other units.

On older Linux kernels, a pointer to the `task_struct` was kept in `thread_info` along with a lot of other things. All of this has been evicted to other places (e.g. the `current` pseudo-variable in the per-CPU private memory area is the place to get the `task_struct` pointer) leaving just the `flags` word in `thread_info`.

Per-CPU variable area

On recent Linux kernels, many kernel variables that pertain only to a given CPU are contained in a special area of memory called the per-CPU variable storage area (alternatively sometimes called the this_cpu area). The most extensively used per-cpu variable is `current` which is actually a macro which accesses a particular variable slot in the per-CPU area. The slot stores a pointer to the `task_struct` for the task currently running on this CPU. Although it is really a macro, it appears like `current` is declared as if `struct task_struct *current;` and is used that way throughout kernel source code.

Now, back to our system call, already in progress

Having computed the address of the thread_info structure, the next line of assembly examines a bitwise flags word. There are quite a few `TIF_XXX` flags defined. Of interest here is a tracing hook: if the `TIF_SYSCALL_TRACE` flag is set, then the thread making the system call is being traced (e.g. through the `strace` command) and the kernel diverts to an alternate entry which will record the parameters of the system call and pass them back as an event to the tracer. We won't go down that road, however.

The next two lines are very important, as they illustrate data validation. Recall that the user-level process is completely untrusted as far as the kernel is concerned. If the system call number passed by the user is greater than the highest system call number (or negative...think about two's complement) then the `syscall_badsys` code is jumped to. This places an error code into the slot in the stack where the user's `eax` register was saved. Upon exit from the system call handler, this value will be popped into %eax.

System Call Return Value

On the X86 architecture, when a function returns an int (or other 32-bit value such as a pointer), that value is returned in the `eax` register. All return values from kernel system calls are in fact signed integers. A negative value is used to indicate an error, and that value is -error_number.

Therefore, an invalid system call will return the value -ENOSYS via the %eax register. Now, an unfortunate history lesson crops up. The UNIX API, for reasons that may be lost to time, specifies that when system calls fail, they should set the global variable errno, and return -1. So the user-level glue function does this (pseudocode):

```
int generic_system_call(arguments...)
{
          put arguments into registers
          put system call # into %eax
          INT $0x80
          if (%eax<0)
          {
                    errno= -%eax;
                    return -1;
          }
          return %eax;
}
```

Kernel system call hand-off to C

However, we know in this case that a valid system call number was used. We have been tracing out assembly language code, but most of the kernel is written in C. The instruction

```
          call    *sys_call_table(0,%eax,4)
```

uses the X86 indexed register offset indirect addressing mode as follows: The %eax register (containing the system call #) is multipled by 4 (the sizeof a pointer), and that offset is added to the base address of a table of function pointers `sys_call_table`. The result of that addition is used to fetch 4 bytes from memory, and _that_ address is the one which is called as a subroutine. It is *as if*:

```
  /* Declare and initialize array of function pointers */
 /* The system call names and positions are purely an example */
  void (*sys_call_table[])()={sys_open,sys_read,sys_close,....};
   /* Hand-off to syscall handler, pseudocode */
          (*sys_call_table[%eax])(args);
```

When the kernel is compiled, the sys_call_table is filled in with the name (i.e. the virtual address) of each C function which implements each system call. It is the convention that a system call which is known as XXX to the user is implemented by a kernel function called sys_XXX.

```
asmlinkage long sys_getuid(void)
{
        return current->cred->uid;     /*cred is the credentials info */
}
```

This system call is coded purely in C, and is in fact found in the architecture-neutral portion of the kernel source code (at /usr/src/linux/kernel/timer.c). The only unusual thing is the compiler directive `asmlinkage`, which indicates that this function is being called directly from assembly language, and thus the arguments are passed in the registers. (to be precise, `asmlinkage` is a macro which expands out to other, more confusing, gcc-specific compiler directives). If you look in the source code you won't find the exact code above, in which some of the macros have been expanded out for better readability.

Note that the uid is a property of the currently running process, and thus is fetched from the `task_struct` structure via the `current` pointer.

Let's take a look at another system call (`time`) which passes an argument:

```
asmlinkage long sys_time(time_t __user * tloc)
{
        time_t i;
        struct timespec tv;

        getnstimeofday(&tv);           //Kernel keeps time in ns resolution
        i = tv.tv_sec;                          //tv_nsec ignored, therefore rounds DOWN

        if (tloc) {
                if (put_user(i,tloc))
                        i = -EFAULT;
        }
        return i;
}
```

Recall that `time` returns the time as an int, but also accepts a pointer to an int. That

argument comes in to the system call as `tloc` in the code above, and if supplied, the kernel takes the value and writes it into the user's memory using the kernel function `put_user`. If the user supplied an invalid memory address, `put_user` will catch that and return a non-zero value, which will cause the system call to fail with EFAULT.

Returning from a system call

Once the system call specific handler routine `sys_XXX` returns, the system call return value is in the `%eax` register (because that's where C function return values are placed by the compiler). Looking back at the `system_call` assembly language routine, we see this return value is written to the kernel stack in the location where the `eax` register will be popped when returning back to user mode.

Now at label `syscall_exit` interrupts are temporarily masked (on multi-processor systems, this applies to the local processor only). The reason for this is to protect the next few testing and branching instructions as a *critical region*. Recall that the `thread_info` structure address is in `%ebp`. The next line of assembly code fetches the bitwise `flags` into register `%ecx` and tests to see if any flags are set which would indicate that, instead of returning directly to user mode, some other action might be required. Let's say for a moment that those flags are clear. Then the code at `restore_all` uses a macro `RESTORE_REGS` to pop all of the registers which had been saved by `SAVE_REGS`. The system call return value is now in `%eax` (regardless of how control reached `restore_all`) and the extra stacked copy of `eax` which contained the system call number is simply discarded.

Now the `iret` instruction is executed. This causes the hardware to restore the `eip, cs, eflags, esp` and `ss` registers. The result is that execution resumes in the user process, with the stack pointer back on the user's stack. All of the registers are exactly as they were when the user process executed the `INT $0x80` instruction, with the exception of the `eax` register which now holds the system call return value. The restoration of the `eflags` register means the privilege level is returned to user mode, and the interrupt mask is restored to the normal value (which when running in user mode is to allow interrupts).

Deferred return from system call, interrupt or fault handler

There are two major reasons why the CPU would not return directly back to the user-mode program upon completion of a system call, an interrupt handler, or a fault handler: (1) another task "may" be "better" to run, as determined by the scheduler. (2) a deliverable signal is pending for our process.

Pre-emptive context switch

The scheduler system may from time to time determine that the current task is not necessarily the "best" task to be running on our CPU. This could happen from within the tick interrupt handler (because the current task has used up its timeslice) or when, during our system call, another task has woken up and it has better priority. The Linux kernel is not fully pre-emptive. Pre-emptive context switches only happen at the moment that control is about to return to user mode. (of course, if kernel code in a system call encounters a blocking condition, such as reading from an empty pipe, that causes an immediate context switch). The key is the bitwise flag `TIF_NEED_RESCHED` which is part of the `thread_info` structure on the kernel stack.

Referring to the `entry.S` code, if the `TIF_NEED_RESCHED` flag is set, then at `work_resched` the kernel scheduler function `schedule` is called. We'll see that this function, if it picks another process to run, causes a *context switch*, and the original process appears to be frozen at the instant of having called `schedule` from `work_resched`. At some later time, the original process is selected to run again, and execution resumes at that frozen point. Control returns from `schedule`, and then the next few lines are identical to those we have already seen: they check to see if anything else has come up, or whether it is OK to return to user mode.

Signal Delivery

It could also be that while the task was in kernel mode, a signal became pending for that task. Perhaps the system call changed the signal mask and a previously received signal became un-blocked, or perhaps the system call itself caused a signal to be raised, or perhaps a signal from another process just happened to come along while we were in the system call. Signal delivery, as discussed in Unit #4, only happens at the moment that control was about to return from kernel mode back to user mode.

`work_notifysig` is called when the threadinfo flags indicate that a non-blocked signal is pending for the process. The kernel then has some work to do to deliver the signal to the process. As we've seen in Unit #4, that might mean terminating the process (the kernel calls `do_exit` on behalf of the process) or invoking the signal handler. In the latter case, the kernel has to modify the user-mode stack to make it appear that the handler was called from the point in the user-mode program (%eip value) where control had entered the kernel, and then change the user-mode registers saved on the kernel stack to cause control to return to user mode at the %eip location of the signal handler, rather than the point where it left off!

The orig_eax stack slot is always the original system call number that got us into the kernel. The eax slot will contain the syscall return value after the system call has run. orig_eax is used in signal handling for "restarted system calls" but we won't be looking at that code.

The X86-64 API

Although the examples herein are for the older 32-bit X86 API, also known as i386, we should examine how things change when using the 64-bit API, known as X86-64.

At user level, the first 6 arguments to a function are passed in registers, not the stack. The argument slots are registers %rdi,%rsi,%rdx,%rcx,%r8,%r9. The X86-64 API does not use the INT 0x80 instruction, but a new instruction called `SYSCALL`, which is somewhat faster because it avoids stack-write memory accesses. It performs the following steps **in hardware:**

- Save the return address (%rip register) in register %rcx (overwriting its value)
- Save the current value of the flags register (%rflags) in %r11 (overwriting it too)
- Make some adjustments to the %cs and %ss registers to allow kernel code to execute properly.
- Set the processor to privileged mode
- Load the value of a special register (MSR_CSTAR) into %rip. This privileged register has been pre-loaded by the kernel to point to the system call entrypoint.

Note that in this 64-bit API, the hardware does not switch stacks nor write anything to the stack when performing a system call.

Because the %rcx register is clobbered by the SYSCALL instruction, the kernel's system call convention specifies that the arguments to the system call are passed in registers: %rdi,%rsi,%rdx,%r10,%r8,%r9. The system call number is passed in %rax. Therefore, the user-level "glue" code takes the 4th argument in %rcx and moves it into %r10, and adds the system call number in %rax. The kernel's system call assembly-language entry code will put the 4th argument back into %rcx from %r10 prior to dispatching to kernel C functions via the system call table.

We are however still on the user-mode stack. Upon entry via any kernel entrypoint, the kernel uses the privileged `SWAPGS` instruction to interchange the user-mode value of the %gs register with a "hidden" %gs register that the kernel has pre-configured to point to the per-CPU area for this CPU. By using an obscure addressing mode of the X86 known as "segment override" the kernel can then easily access this memory. One of the per-CPU variables is the kernel-mode stack pointer. This value is now loaded by the kernel into %rsp and the kernel saves the user-mode registers on the kernel stack. Prior to return to user mode, the kernel makes sure that the kernel stack pointer is stored in the per-cpu scratchpad area, executes the SWAPGS instruction again to save the hidden gs register, and restores the user-mode %rip and %rflags values (which were saved on the kernel stack) into %rcx and %r11 respectively. The `SYSRET` instruction is then used to reverse the effects of `SYSCALL` and return control to user mode.

If the above description of X86-64 caused pain and/or confusion, do not panic. We will conduct the rest of our examples in 32 bits.

Summary of 32 and 64 bit X86 Syscall APIs

	32	64
Opcode	INT $0x80	SYSCALL
syscall#	%eax	%rax
arg1	%ebx	%rdi
arg2	%ecx	%rsi
arg3	%edx	%rdx
arg4	%esi	%r10
arg5	%edi	%r8
arg6	%ebp	%r9
retval	%eax	%rax

Appendix - X86 Architecture and Assembly Language

The following material comes from ECE466 -- Compilers and is intended to assist with understand the assembly language portion of the kernel.

X86 refers broadly to a family of Intel (and compatible) microprocessors manufactured in the last 20 years or so. It is also called the X86 architecture by Intel. The first 32-bit X86 processor was the 80386. X86-64 is a 64-bit extension to X86. Intel's is a CISC architecture which is a direct linear descendent of the very first microprocessor, the 4004 (a 4-bit product).

There are many who find the X86 architecture to be a dinosaur, and a badly designed one at that, which should have long ago become extinct. However, IBM's choice of it for its first personal computer sealed its fate as the most popular processor architecture.

The X86-64 architecture extends the 32-bit X86 to use 64-bit registers, while retaining backwards compatibility with 32-bit X86 code.

Below is a summary of the X86/X86-64 architecture The reader is detoured to the official reference manuals for full details.

Intel vs UNIX assembly syntax

The Intel documentation uses the Intel standard assembly language syntax, but the UNIX assembler `as` follows a different convention (which is consistent across different processors). In the UNIX syntax, an identifier is unambiguously an assembler symbol. To reference a register, its name is prefixed with a percent sign, e.g. %eax. To use a symbol as an immediate value, the dollar sign is used as a prefix. Otherwise the symbol means the contents of that address. Register indirect addressing modes are indicated by

brackets or parentheses. UNIX assembly instructions are opcode src1,src2,dst for 3-address instructions or opcode src,dst for 2-address. (Note that Intel syntax is dst,src). We will use the UNIX syntax in these notes. Another name for this is the "AT&T" syntax, after the original authors of UNIX

In the UNIX/AT&T syntax, where a particular opcode can be performed at different precisions, that opcode receives a letter suffix: b,w,l,q for 8,16,32 and 64-bit operations respectively.

Assembly language files are denoted with a .s or .S suffix. The latter is used in the Linux kernel for files which are hand-crafted in assembly, as opposed to .s files which were generated by the compiler. Assembly language is line-break sensitive. Each line is an instruction (not including whitespace, assembler directives and comments) and each instruction must be contained on that line. The line has 2 parts: opcode operands. These fields are separated by whitespace. The label is a symbolic name given to that instruction or memory location. It is specified as `label:` and can either appear at the beginning of the line, or on a line by itself, in which case the label is attached to the next line. By stylistic convention, the opcode is never at the beginning of the line. There is least one whitespace character (typically a tab) so that the opcode is not mistaken as a label. The operands field contains 0, 1 or more operands delimited by commas, and following the addressing mode syntax explained below.

Assembler Directives / Pseudo Opcodes

Assembler directives are pseudo-opcodes that do not correspond to actual opcodes that the processor executes, but cause the assembler to modify its operation, or to emit special code or data. These include `.text` and `.data` to switch between the text and data sections of the a.out file, `.byte` to emit a single byte, `.long` to emit a 4-byte value, and `.string` to emit a nul-terminated string. This is not an exhaustive list and the reader is referred to the documentation for `as`.

X86 Register Model

When referring to X86 registers, their size is implied by a prefix. For example, there is a 32-bit register called EAX. The least significant 16 bits of that register are called AX. It is possible to refer to the least significant byte as AL and the next most significant byte as AH. In the 64-bit X86-64 instruction set, the 64-bit version of EAX would be called RAX. We will consider the 32-bit model first.

The register model of X86 is convoluted and archaic, making efficient register allocation and instruction selection a challenge. The following general-purpose registers are typically used for holding temporary values, general integer computation, etc.

- %eax: The "accumulator". Many instructions use %eax as an implied operand.
- %ebx: The "base register" (not to be confused with %ebp).

• %ecx: The "counter register".
• %edx: The "data register".
• %esi: Source register for string operations
• %edi: Destination register for string operations

The following special registers are used for control flow:
• %eip: The "instruction pointer", aka the Program Counter. At the time of instruction execution, %eip contains the address of the next instruction to be fetched. A branch instruction modifies %eip and causes the next instruction to be fetched from that new address.
• %esp: The stack pointer.
• %ebp: Typically used in the C / assembly language convention for the stack frame "base pointer". Aka the "frame pointer".
• %eflags: The flags register. It contains the condition code flags (carry, parity, BCD adjust, zero, signed, overflow) as well as a number of flags and control bits which can only be modified when running in Kernel (Supervisor) mode, such as the interrupt enable flag and the user/supervisor privilege level.

Segmentation, Special-Purpose and Additional Registers

The X86 addressing scheme is based on an obsolete concept known as "segment/offset" addressing. In all modern operating systems, program addresses are linear, and the segmentation is basically ignored. The register model contains the registers %cs, %ds, %ss which are initialized by the kernel and should not be touched. They are what enable code, data and stack accesses to work. Additional segment registers %es, %fs and %gs are general-purpose and, because a linear addressing model is being used, could be employed as general-purpose scratch registers, subject to some restrictions as to which registers may appear in which instructions. However, both the %fs and %gs registers are used by the kernel and the standard library, and should be avoided.

We have seen additional registers such as %cr3 and %tr. These are generally only accessible when running in kernel mode, and require special opcodes to read or write. Additional registers are present for various floating point operations. The kernel does not use these at all, but is required to save and restore them when context switching between user-level tasks, if they are being used by those tasks.

On X86-64, there are additional general-purpose registers `%r8 - %r15`.

Addressing Modes

There are a number of addressing modes which are used to specify where to find or put the operands of an instruction:
• Register Direct: Specify the register name with a % prefix, e.g. %eax.

• Immediate: The immediate value must be prefixed with the dollar sign, e.g. $1
• Memory Absolute: The absolute address of the operand is specified without a prefix qualifier. E.g. `movl $1,y` moves the immediate value 1 into the memory address which is associated with the linker symbol y.
• Base-index (Register Indirect with offset): The X86 has a handy mode for accessing elements of an array. The syntax is `disp(%base,%index,scale)` . The address of the operand is computed as `addr=base+index*scale+disp`. The base and index may be any of the general-purpose registers (eax, ebx, ecx, edx, ebp, dsi, edi, esp (not allowed as the index)). The displacement is a 32-bit absolute address. The scale factor may be 1, 2, 4 or 8. Some of these parameters may be omitted, forming simpler addressing modes. E.g. in `movl $1, (%eax)` the eax register contains a pointer to a memory location, into which the immediate value 1 is moved.

X86 is generally a 2-address architecture, meaning that one of the operands is both a source and a destination. There are many combinations of src/dst addressing modes including some odd restrictions. Generally speaking, most opcodes allow register/register, register/immediate, register/memory or immediate/memory combinations. Memory/memory is generally not allowed.

Function Calling Convention

We will discuss what the Intel documentation calls the CDECL convention for procedure calling, as that is what is used in the C/UNIX world. Other calling conventions do exist. In the X86-32 architecture, all arguments to a function are pushed on the stack, and the return value is returned in the %eax register. If the return value is 64 bits (long long), it is returned in the register pair %edx:%eax, with the %edx being the most significant 32 bits.

Recall that %esp is the stack pointer, and the stack grows towards low memory. The PUSH instruction predecrements the stack pointer, then writes the value to (%esp). Likewise, POP reads from (%esp) and then postincrements %esp. Arguments in C are pushed to the stack in right-to-left order. Therefore, just before issuing the CALL instruction, the leftmost argument is on the top of the stack. This convention allows variadic functions to work properly. The callee does not need to know in advance (at compile time) the exact number of arguments which will be pushed. It is able to retrieve the arguments left-to-right by positive offsets from %esp.

The CALL instruction pushes the value of %eip, thus on entry to a function (%esp) contains the address of the instruction to which control should return (i.e. the instruction after the CALL). The first thing any function does is set up its local stack frame. Let's look at an example:

```
f1()
{
        f2(2);
}
```

```
f2(int b)
{
int a;
        a++;
        b--;
        return 1;
}

f1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp                !one arg slot + one padding slot
        movl    $2, (%esp)              !put arg onto stack
        call    f2
        leave
        ret

f2:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp               !extra space for alignment
        incl    -4(%ebp)                !access local var a
        decl    8(%ebp)                 !access param b
        movl    $1,%eax                 !return value
        leave
        ret
```

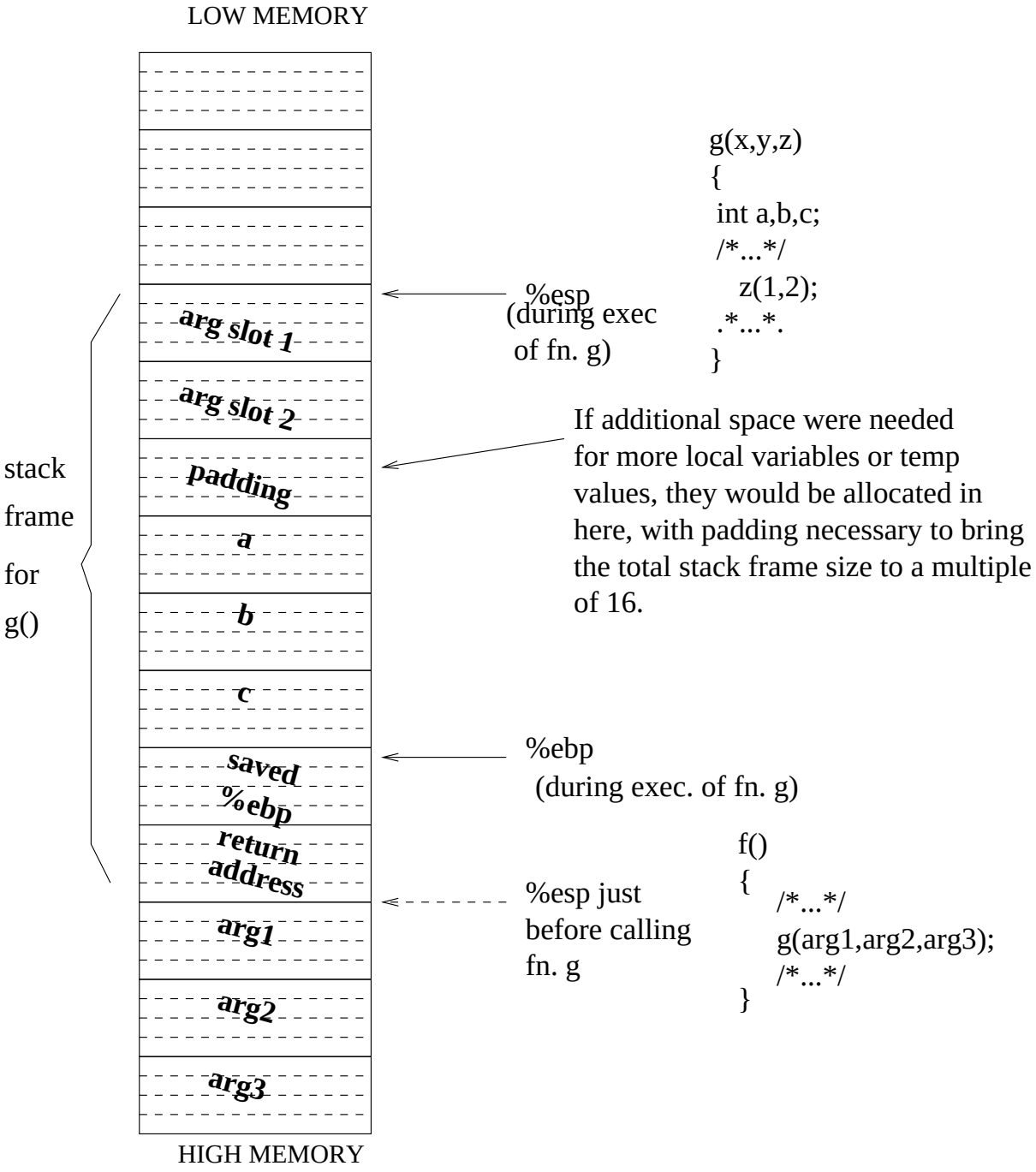
The %ebp register is the frame pointer, and will be used to access both local variables and parameters. Its value must be preserved so the first action is to save it on the stack. Then the stack pointer is decremented to create room for local variables. In our example, function g has one local variable which takes up 4 bytes. The %ebp contains the value of the stack pointer after saving the old %ebp. Therefore 4(%ebp) is the return address, (%ebp) is the saved %ebp, and the first parameter is 8(%ebp). Parameters will be at positive offsets from %ebp and local variables will be at negative offsets. Generally speaking, the local variables mentioned first in a function will have the lowest memory address (i.e. highest negative offset from %ebp), but that behavior is not guaranteed.

When a function call is made, arguments can be pushed on the stack in right-to-left order, using the `pushl` instruction. After the `CALL` instruction, an `addl $X,%esp` would be needed to adjust the stack pointer and reverse the effects of the previous pushes. Alternatively, one could determine during code generation which function call (within the function being generated) has the highest number of arguments. The number of bytes thus required for passing arguments can be added to the total local stack frame size, as if these "argument slots" were hidden local variables. Then the arguments can be passed via `movl OFFSET(%esp)`, in any order desired, and there is no need to adjust the stack pointer after the call. This is the approach that gcc takes.

Upon leaving a function, the LEAVE instruction is used, which performs two operations: %ebp is moved into %esp, thus restoring the stack pointer to its value just after the base

pointer save on entry, then %ebp is popped from the stack. Now everything is restored, and the RET instruction pops the return address from the stack and resumes execution in the caller.

If the compiler chose to use any registers which are callee-saves, we would see pushes of those registers on entry and corresponding pops on exit.



X86-64 Function Calling

Under the 64 bit architecture, the first 6 integer arguments are passed in registers, rather than on the stack. Arguments are placed in left-to-right order in registers %rdi, %rsi, %rdx, %rcx, %r8, %r9. If there are additional arguments, they are put on the stack right-to-left, i.e. with the right-most argument at the highest memory address, just like X86-32. If structs are passed as arguments, they are always placed on the stack. The integer return value is in the %rax register.

This hybrid register/memory argument passing model introduces some complexity with variadic functions, aka `<stdarg.h>`. GCC implements `stdarg` as a compiler built-in.

X86-64 Global Variables

There is an odd limitation in the X86-64 instruction set: the absolute addressing mode is not supported for 64-bit addresses. To access a memory operand, a register indirect addressing mode must be used.

```
extern int i;

f()
{
          i=2;
}

f:
        pushq   %rbp                            #Prologue, save base pointer
        movq    %rsp, %rbp              #Set new base pointer
          subq      $32, %rsp           #Create stack frame
        movl    $2, i(%rip)             #Program Counter Relative mode
        leave
        ret
```

There will be a 32-bit "hole" in the `movl` opcode which will be a program counter relative relocation type (similar to the example of the CALL opcode earlier in this unit). At link time, when the address of symbol i has been resolved, this hole will be filled with the i's address, minus the address of the hole itself.

This introduces a limitation that code and data must fall within the same contiguous 2GB memory region at run time, which the X64-64 spec calls a "medium" memory model. To use a "large" memory model where code and data may be anyplace within the 64-bit address space, different opcodes are used:

```
          movabsq   $i, %rax            #Move 64 bit immediate value to rax
        movl    $2, (%rax)              #Register indirect
```

Caller/Callee saves

It is the case for any architecture and operating system that there is a function calling "convention" which specifies how arguments are passed and returned, and how registers may be used. This convention dictates which of the registers are expected to survive a function call, and which ones may be used as "scratch" registers, and are therefore expected to be volatile across function calls. Another way of saying this is there are caller-saved registers (the scratch registers.. if the caller wants to keep a value in there through a function call it must explicitly save it) and callee-saved registers (if a function wants to use one of these registers it must explicitly save it on entry and restore it before returning).

In the X86-32 architecture under UNIX, the %eax,%ecx,and %edx registers are scratch registers (caller-saves). You will find that the compiler tends to put short-lived values in these registers. Of course the %eflags register is also expected to be modified by a function call. The %ebx,%edi,%esi and %es [CAUTION: this is a 16-bit register] registers are callee-saved. The compiler may use these for longer-lived values (such as local variables which are assigned to a register for all or part of the function to improve speed). However, if one of these registers is used by the compiler, it must emit code to push it on the stack on entry, and pop it on return.

On X86-64, the caller-save (scratch) registers are %rax,%rcx,%rdx,%rsi,%rdi, and %r8-%r11, while the callee-save (long-term) registers are %rbx, %r12-%r15. Note that %rsi and %rdi are caller-save on 64 bit, whereas they were callee-save on 32-bit. This is because they are used for argument passing on 64 bit.

X86 General-Purpose Register Summary

-32 reg	Saved by	Notes	-64 reg	Saved by	Notes
%eax	Caller	fn retval	%rax	Caller	fn retval
%ebx	CallEE		%rbx	CallEE	
%ecx	Caller		%rcx	Caller	arg #4
%edx	Caller	longlong ret	%rdx	Caller	arg #3
%edi	CallEE		%rdi	Caller	arg #1
%esi	CallEE		%rsi	Caller	arg #2
%es	CallEE		N/A		
			%r8	Caller	arg #5
			%r9	Caller	arg #6
			%r10	Caller	
			%r11	Caller	
			%r12	CallEE	
			%r13	CallEE	
			%r14	CallEE	
			%r15	CallEE	

Device Drivers

Device drivers are a collection of functions grouped into a module within the kernel. They can be statically linked into the kernel and available at boot time, or dynamically loaded as needed.

The purpose of device drivers is to provide a universal interface to hardware devices, isolating the details of manipulating the hardware within the device driver code. Some device drivers correspond to actual hardware, others are "pseudo-devices", such as the pseudo-tty driver which provides remote command-line access through programs such as telnet or ssh. Other notable pseudo-devices include `/dev/zero`, which is an infinite source of 0 bytes and a sink for data, and `/dev/null`, which is used as a sink for data or a source of immediate EOF.

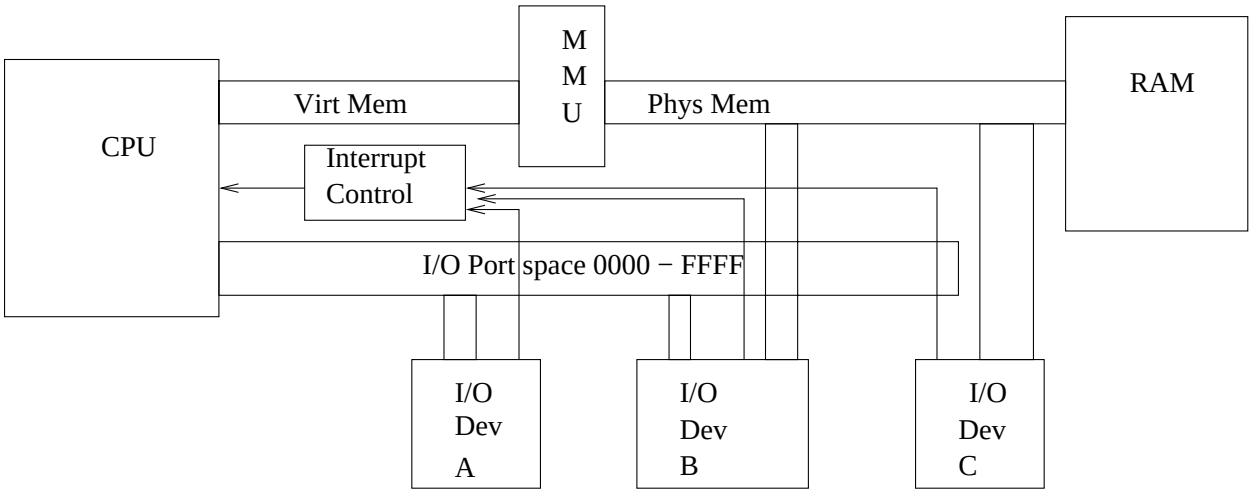
Within the Linux kernel, a device driver can be thought of as having four facets:
• Routines called when the driver is loaded, which register the driver with the kernel and probe for hardware (if applicable). Similar routines exist for unloading a driver (if it is not in use).
• Routines which are called directly, or indirectly, from a synchronous context, i.e. a system call or fault handler. E.g. during a read system call, control eventually reaches the device driver's read routine, which initiates the I/O operation, and then puts the task to sleep pending completion.
• Routines which are called in an interrupt context. This is often called the "bottom" of the driver. Interrupt handler routines must be lightweight. They must not consume excessive CPU time, or perform any operation which might block indefinitely.
• Routines which are called in a deferred, or "soft" interrupt context. When a device driver must perform a complicated or potentially blocking operation in response to an interrupt, it uses this callback mechanism to avoid encumbering the interrupt response path.

I/O Addressing and Device Registers

Access to I/O devices is an extremely architecture-dependent issue. On the X86 architecture, there is a separate 16-bit address space called **I/O Ports** which is used for I/O device control and status registers. This address space is not the same as memory, and must be accessed using special IN and OUT instructions in X86 assembly. In addition, many I/O devices on the X86 PC architecture have mappings to physical addresses. The kernel uses an area of its virtual address space to create mappings to these special I/O memory areas. The physical addresses containing I/O devices are "holes" in the pool of available page frame addresses.

Because I/O ports are not memory addresses, they are not subject to the MMU. I/O ports generally can only be accessed from kernel mode. It is possible to associate an I/O port

permissions mask with a process which allows that process to access specific I/O ports. This is sometimes done to accelerate graphics performance for user-mode programs. Setting I/O permissions requires root (superuser) access. Most other architectures do not have the equivalent of I/O ports and all hardware must be accessed via memory.



While space here does not permit a full discourse on I/O device architecture, in general we can say that any given I/O device has one or more `registers` which are accessed either via I/O port number addresses and/or physical memory addresses. The selection of which port numbers or physical addresses are used is part of the auto-configuration process that is built in to I/O protocols such as PCI. I/O device registers have nothing to do with CPU registers. In general, an I/O device has the following types of registers:

• Configuration: Writing to a configuration register changes the basic operation of the device, selects modes of operation, determines what other port or physical addresses the device might use, determines what interrupt vectors the device might use, etc.

• Control: Writing to a control register starts or stops a device operation. E.g. on the obsolete floppy disk I/O controller, the control register can be used to start and stop the spinning of the diskette media.

• Status: Reading from a status register gives information such as whether data are ready to be transferred, if the device is functioning properly, if media are inserted, etc.

• Data transfer: On "polled" I/O devices (see discussion below), data transfer can be accomplished by reading/writing the data transfer register(s)

Often, a physical register at a given I/O port address contains a combination of these functions. E.g. a port number might be a Configuration & Control register during an OUT operation, and a Status register during IN.

Buses and device driver layering

A long time ago, most I/O devices were "dumb" and relied on the CPU and the device driver to operate them. A disk device interface might literally have bits for turning the

spindle motor on/off, stepping the head to find the track, and moving the data bit-by-bit from the head. In modern computers, the devices are much smarter. Very often, there are additional layers which are known as "buses" to isolate the CPU and operating system from the low-level details. On these buses, specific "protocols" are followed, similar to network protocols, which define the state diagram of the bus, the types of transactions and messages which can take place, error handling, etc.

Historically, buses can be categorized as serial or parallel. In the latter case, the I/O bus resembles a memory bus, with control, address, clock and data lines. At one time, parallel I/O buses predominated, with the most prolific being the Small Computer Systems Interface (SCSI) which used buses of 50, 68 or 80 wires.

In modern computer architecture, buses are physically implemented with point-to-point serial interfaces (much like modern Ethernet ports). With rising data transfer rates, it becomes impossible to manage the "slew" between the data lines of a parallel bus, and it becomes impractical to have multiple devices tapped into the same bus wire (other than power) because of capacitive loading and transmission line type effects. Thus we see mostly SATA/SAS (essentially a serialized version of SCSI) for disk drives and Universal Serial Bus (USB) for just about everything else. Although these are "serial" bus ports, to improve performance, the port may have multiple "lanes", each with its own clock and data wires. This does not make it parallel because electrically, each lane is independent.

A given instance of a bus can logically have multiple target devices attached to it, even if physically there are some electronics in between to resolve issues at the physical level. E.g. a USB "Hub" extends a USB bus and allows additional devices to be logically part of that bus. A multiplexer (or "extender") can allow multiple SATA or SAS hard drives to be accessed via a single SATA/SAS physical port.

From the standpoint of the kernel, there is a hierarchy of layers that must be navigated to communicate with a target I/O device:

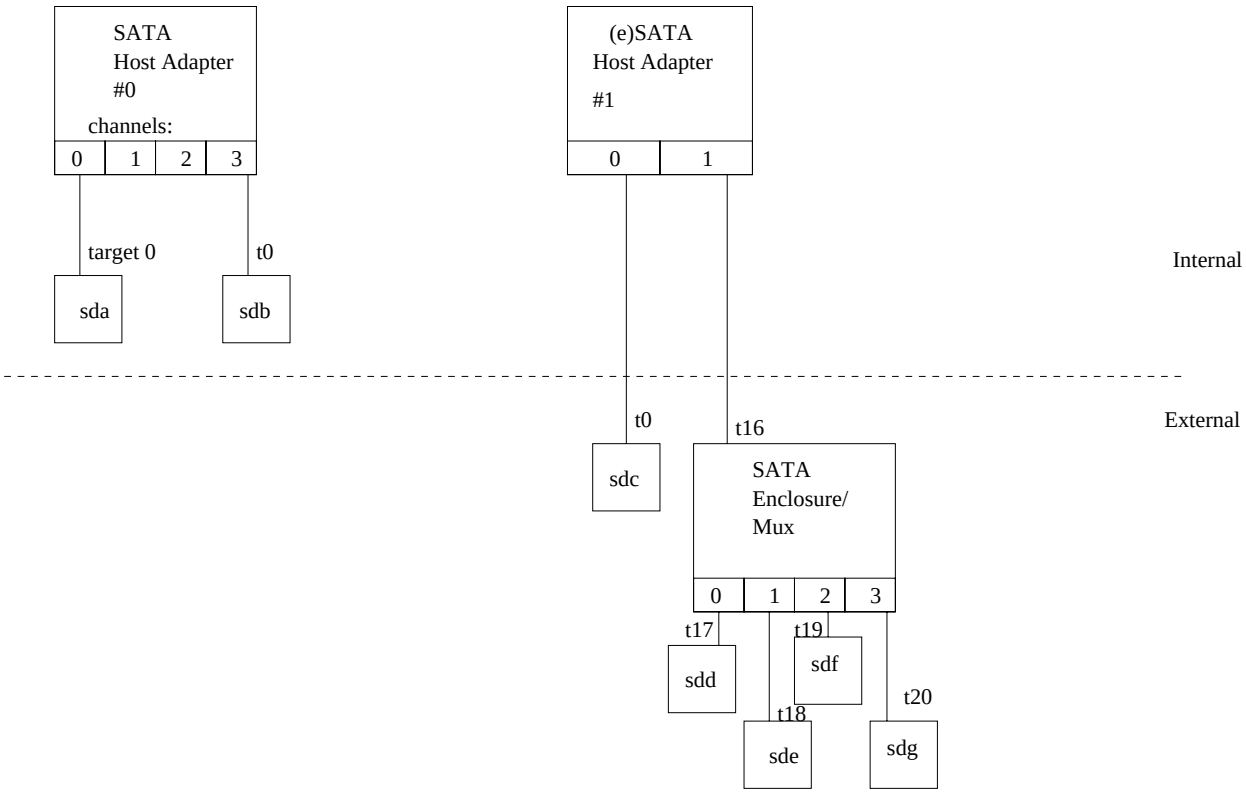
• The Host Controller is the physical chipset which is directly accessed by the kernel using IN/OUT instructions and/or memory-mapped I/O. It provides one or more physical bus ports which allow for bus-level communication with other devices.

• The Bus layer is a collection of kernel modules that provide the protocol support for a given bus. E.g. we want to transfer data to a given device at a given bus address, one of these modules prepares the necessary USB or SAS/SATA message blocks and submits it to the correct host controller module. In theory, the bus layer modules are independent of the host controller drivers. It shouldn’t matter which specific chipset we have on our computer, because the protocol messages are all the same

• Bus addressing layer: A given controller may have multiple physical ports, each of which forms an independent bus. The Linux kernel calls these "channels". On each channel, physical devices connect to the bus with a specific bus address known as the target address (which must be unique to that given bus). Finally, each target may have sub-addressing.

To further complicate matters, many mass storage devices are now connected via USB. Although from a bus perspective these target devices are USB devices, they speak the same protocol known historically as SCSI (which is also spoken by SATA and SAS hard drives). The Linux kernel will then instantiate a virtual SCSI "bus" to represent these USB devices as if they were connected via SCSI, SAS or SATA, so that it can use the existing SCSI disk layer to access them.

The `/sys/bus` directory within the `/sys` pseudo-filesystem allows one to visualize the physical devices that the kernel "sees" on each bus. Other utilities, such as `lsusb`, present this in a more user-friendly output format. Reading `/proc/scsi/scsi` gives a list of the SCSI (including SATA, SAS and USB-connected) storage devices that the kernel sees. One can also write to this file (as superuser) to effect certain control operations, such as forcing the kernel to re-scan for a new disk.



Data Transfer Modes, Direct Memory Access

Transferring data to and from I/O devices can be done in what is called **polled mode**, in which the CPU transfers one small chunk of data directly using the device control registers, then waits for an interrupt to indicate that the device is ready for the next chunk. Such an approach is not efficient for devices which need to transfer bulk data with any appreciable speed.

Direct Memory Access means that I/O devices can read and write physical memory without the intervention of the CPU. A DMA transfer is initiated by the kernel. First one or more page frames are allocated for the transfer (or they may already be allocated...e.g. a mapped file). The kernel pokes the hardware device control registers to inform it of the buffer address, then initiates the operation. Data transfer happens and the CPU is not involved in this. The CPU receives an interrupt when the transfer has been completed. Some I/O devices work one request at a time, and must be given contiguous page frames. Others can establish a complex work queue of different transfers to different pages.

DMA can be said to be synchronous or asynchronous. The above describes synchronous DMA. In asynchronous mode, the device decides when to initiate a transfer. The kernel pre-allocates a pool of page frames for the device's use, and receives interrupts whenever a transfer has happened. This asynchronous mode is the way network interface cards have to work, since the kernel can not predict in advance when a network packet will arrive.

Page frames which are being used for DMA transfer are locked by the kernel by setting flags in the page descriptor data structure. Another term for this is **pinning** the page to the transfer. Pinning the page prevents it from being reclaimed or another I/O transfer being initiated to or from the same page, until the first transfer is done.

On the X86 architecture, the DMA controller speaks directly to the memory arbiter/controller with physical addresses. In some other computer architectures, I/O devices are given their own virtual address space with an IO MMU.

Major and Minor numbers

Device drivers are identified within the kernel by major and minor numbers. By default, they are each 8 bits long and together form a 16-bit **device number**, which is of type `dev_t`. (On some modern Linux kernels, the device number address space is larger) A unique major number is assigned to each device driver in the system, while the minor numbers are assigned by the device driver to refer to the devices it manages. For example, all of the `/dev/hda*` devices, which refer to (legacy) ATA hard drives, are managed by one device driver with major number 3. Unique minor numbers refer to each partition of each disk in the system.

There are two distinct namespaces for major device numbers, `character` and `block`. Block devices are those in which filesystem volumes usually live. They are generally addressed in terms of large chunks of contiguous data ("blocks") and are random-access. Character devices tend to work one character at a time, and are not random-access. Examples of character devices include "COM", mouse and keyboard ports.

On some UNIX operating systems other than Linux, e.g. Solaris, block devices also have a character device interface, which is the "raw" (unbuffered) mode. In Linux, this is not

the case -- unbuffered access is done by using the `O_DIRECT` flag when opening the block special file.

Allocation of major device numbers is determined by the maintainers of kernel code. Most important devices have pre-allocated numbers. Device driver developers can use some of the unassigned numbers.

The actual major and minor device numbers are of little concern to the end-user or programmer. Devices are accessed through the regular filesystem namespace, by means of character special and block special inodes. These, which were mentioned in Unit 2, have inode type fields of `S_IFCHR` or `S_IFBLK`, respectively.

Block Devices and the Buffer Cache

There are two ways of looking at block devices. On the one hand, they are an array of sectors. At the same time, a given sector may contain a portion of a file, or may contain a filesystem data structure such as an inode, but not both. Further complicating this, block device files can be accessed via mmap in addition to read/write system calls.

Within the Linux kernel, another interface exists known as the **Buffer Cache**. It is synchronized with the page cache, but is addressed in terms of device numbers and block (sector) numbers. Filesystem modules tend to use the buffer cache interface to read inodes, superblocks, freelist bitmaps, indirect blocks and other metadata and control structures.

Buffers are controlled by `struct buffer_head` structures. When a page frame holds data which are also accessible through the buffer cache interface, the `private` field of the page descriptor is used as a pointer to the buffer head representing the first block within the page. Other blocks in the same page are chained through their buffer head structures through the `b_this_page` field of the buffer head. these areas.

During normal file read/write activity, whole pages at a time are read/written. The filesystem layer translates this 4096-byte chunk which is contiguous with respect to the file into a series of buffer cache requests representing the disk blocks which comprise that chunk. It could be (and indeed it often is the case with EXT2/3/4 filesystems using a 4K allocation unit) that these are also contiguous disk blocks, but that is not always the case. When the I/O operation completes, the list of sectors comprising that page is recorded, as described in the paragraph above. At the same time, that page has to be correlated to the address space of the block device special file:

The inode associated with a particular block device has an `address_space` with a radix-64 tree. Given a device number and offset, we can find if there is a page frame which contains the image of that part of the raw disk. However, some complexity is introduced when the same piece of data is accessed both through a regular file and through direct use of the block device file. The filesystem does not necessarily use a

block size (minimum allocation size) which is the same as the page size. Since files are not necessarily allocated to contiguous disk blocks, a page of memory representing the contents from the file's perspective may contain the images of multiple, non-contiguous blocks from the block device's standpoint.

So, given a device number and block number, the kernel can translate that block number into the nearest page-aligned offset within the block device, and then consult the radix-64 tree associated with the block device's inode. If there is a page descriptor attached to the corresponding node of that tree, then that page *might* contain the cached copy of the block in question. The kernel needs to walk through the buffer head list associated with that page and see if there are any matches.

A block device file being like any other file, it can be mmap'd. There are some very subtle consistency problems which can arise when a block device file is mmap'd and so is a file which lives on that block device! Since blocks are smaller than pages and not allocated contiguously, a page frame can either hold the valid image of a page-aligned contiguous portion of the file, or the valid image of a page-aligned, contiguous portion of the block device, but not both at the same time. Fortunately this is not a common situation. It is generally considered bad form to be working on the block device directly at the same time it is mounted.

Direct and Raw I/O

A major advantage of memory-mapped files is the saving of an additional memory copy operation. When using e.g. the write system call, the data must be copied once from the user address space to a buffer page in kernel space, then copied again during DMA. Linux also supports **direct I/O** in which the buffer address supplied to read or write system calls is used directly by the kernel as the DMA buffer. Finally there is a "raw" character special device which can be bound to a block device and used to issue I/O requests directly to the disk. These latter two modes bypass the page cache and are useful to database applications which effectively implement their own proprietary filesystems to store the database tables, and therefore are not concerned with caching or the UNIX filesystem model.

Block I/O & Disk Strategy

All I/O operations with block devices are handled by the device driver using `buffer_head` structures, because the block device works at the granularity of a block, not a page. To be more correct, device I/O happens in terms of 512 byte **sectors**, and blocks always contain contiguous sectors. An I/O request is encapsulated in another data structure called `struct bio`. Each `bio` refers to one or more `buffer_head` structures and contains a direction flag (either READ or WRITE).

Linux assumes that disks are addressed in terms of sequential sector numbers. Of course, actual physical disks are addressed in terms of three variables: cylinder, head and sector. Older disk technology, such as floppy disks, required the disk I/O commands to be in these CHS terms, and if that is the case, the device driver will translate. All modern hard disks use Logical Block Addressing and perform the translation to the internal geometry within the hard disk's on-board controller. This translation is done such that "S" is least significant and "C" is most signficant. Therefore, the effect of presenting a series of I/O requests to the disk with the logical block number increasing by one each time is to step the head assembly from the innermost to the outermost track. Note that since recording density in terms of Mbits/square inch tends to be a limiting factor, there will be more sectors per track on the outer parts of the platter than the inner.

Moving the head assembly is the slowest operation for a hard drive. For example, a specification for a particular mid-range hard drive, 250GB capacity, 7200 RPM, gives a data transfer rate of 70MB/sec. Average random-access seek time is 9msec, while average single-track seek time is 1msec. The average rotational latency (because of the need to wait for the disk head to line up with the sector in question) is approximately one half of the rotational period, which for the drive in question yields 4.16 msec.

Therefore it will take approximately 7 microseconds to read the data from one 512-byte sector, but up to 600 times longer to read two non-adjacent sectors on the same track, and over a thousand times longer to read two sectors which are on different tracks. It is apparently advantageous to transfer contiguous sectors and to minimize head seeking. The classic operating systems approach to this problem is known as the **"Elevator Algorithm"** and resembles the manner in which a traditional (i.e. non-"smart") elevator services requests. All disk I/O requests for a given device are placed into a sorted list. The device driver has a notion of the current request and the direction, either towards higher logical sector numbers or towards lower. The driver services all of the requests in one direction, then works in the other direction until all requests are done, then changes direction again, etc.

Unfortunately, this algorithm does not exhibit any queing "fairness", in that much older requests may get passed up in favor of much newer requests. Linux allows the elevator algorithm to be tuned by the administrator to balance fairness against performance, and calls this "Completely Fair Queueing".

To some extent, the on-board disk controller itself performs the same function. With modern hard drive interfaces such as SATA-II and SCSI, a mode of operation called **Tagged Queuing** is used. Requests are given to the hard disk along with a small integer tag to identify them. The hard disk's on-board controller buffers these requests and satisfies them in the order which it thinks is optimal. As each request completes, the disk sends a message back to the controller board in the computer. This message contains the tag number of the completed request.

The advent of Solid State Disks has complicated matters. There is no "seek time"

because there is nothing mechanically moving within. Read performance is basically independent of block address ordering, but because writes are performed in larger chunks, and because the SSD internal controller tends to translate the block numbers presented by the host into internal block numbers to accomplish "wear levelling", it can be difficult for the kernel to make good scheduling decisions that optimize performance. This continues to be an active area of OS research.

Partitions, Concatentations and Logical Devices

More often than not, the raw disk drive is larger than the largest volume that we'd like to store on it, or, for security, performance or administration reasons, we'd like to have our overall filesystem consist of several independent volumes, but we have fewer physical disks than the number of volumes we'd like to create. This is where partitioning comes into play.

When a hard disk is used with partitions, the first sector contains a `partition table` which contains a list of partitions. Each partition is described by its starting sector offset (not including the partition table), its length, an identifier of the type of filesystem which is intended to be created on it, and flags such as whether the partition should be writable or mountable.

Historically, each operating system had its own partition table format (as well as its own volume format) and that made reading disks from a "foreign" system challenging. Linux supports several partition table formats. The most common are MBR/FDISK (MBR==Master Boot Record and FDISK was the name of the DOS command to partition the disk), which is the same format used by MSDOS and Windows, and EFI/GPT (Extensible Firmware Interface / Globally Unique Identifier Partition Table) which is supported by many operating systems and hardware platforms. Because the older MBR/FDISK format is limited to disks of 2TB and under, it is generally necessary to use GPT for larger disks.

When a physical disk drive is "discovered" by a Linux kernel, either during the initial scan of the buses during boot, or when a new device appears on a bus (e.g. a USB storage device is plugged in), the kernel first fetches sector 0 of the disk. By looking at the first few bytes, it can take an educated guess if the partition table is FDISK, GPT or some other supported type. It then loads the partition table (which may span several sectors) and in addition to creating a device inode such as `/dev/sdh` to represent the entire physical disk, also creates nodes `/dev/sdh1` for the first partition, `/dev/sdh2` for the second, etc. Any accesses to the partitions are adjusted by adding the appropriate sector offset to arrive at the raw disk block number.

It can also happen that we need to make a single filesystem (volume) which is larger than any single disk that we have. Linux has a tool known as LVM (Logical Volume Manager) to create virtual disk drives which are formed by first concatenating multiple physical

disks (or partitions). That concatentation can then be re-partitioned as needed to create logical volumes. In some cases, a logical volume might also have a partition table placed on it.

Mirroring

Because hard disks are prone to data loss or complete failure (this includes Solid State Disks too), many system administrators configure mirrored drives. There are several ways to approach this. Systems which are intended for server use often have a hardware card which performs the mirroring, or it can be done by the kernel using the metadisk (`md`) driver and related tools.

We haven't the time here to explore all of the mirroring configurations. The reader is invited to explore "RAID" (Redundant Array of Inexpensive Disks) "levels" such as RAID-1 and RAID-5.

When a physical disk is mirrored, we use the virtual disk interface, e.g. `/dev/md1`. A write to that virtual disk results in write requests being issued to two (or more) underlying physical disks (or partitions) so there are always two very recent copies of the data. A read can be satisfied from either disk and thus read performance is approximately doubled.

In the event that one disk fails, the other disk continues to handle all read and write requests, so there is no interruption to service. An alert is raised and the system administrator replaces the defective disk. The kernel is informed of this replacement and begins a background kernel-mode task to "re-sync" the mirror. Each block of the good disk is read, in sequence, and written to the replacement disk. This could take hours or even days depending on the size of the disk and the amount of ordinary read/write traffic load.

A modern trend has been to move away from mirroring and concatenation at the disk level and instead have filesystems which natively support being composed of multiple disks. One of the most popular is ZFS (Zettabyte Filesystem) which originated with the Solaris kernel but is now available for other UNIX variants including Linux. One advantage of having the filesystem manage the disks is that it knows what is on them. If a disk needs to be replaced on a large volume which is not that full, ZFS is more efficient because it only has to re-sync the data that are actually there, vs. the entire disk.

Creation of the /dev inodes

Traditionally, the inodes under /dev were created statically, or manually by the system administrator. On Linux systems, the device drivers which are part of the kernel at boot time each execute a self-probe to see if any devices are present which they handle.

System utilities which vary from implementation to implementation determine the results of these probes and create the appropriate /dev entries without user intervention. A similar process is performed when device drivers are dynamically loaded into the kernel after boot. These `/dev` names do not necessarily correspond with physical/bus addressing, but follow some naming convention which is specific to each driver. E.g. the `sd` driver simply names the hard drives a,b,c etc. as it discovers them.

One can use the mknod command or system call to create an inode which is a block device or character device, specifying the major and minor device number in doing so. Let us say we execute the command `mknod /tmp/disk b 8 1`. Since major device 8 is the `sd` disk driver, this is equivalent to the system-created `/dev/sda1` node. While it may be useful to create device nodes by hand, it can also be dangerous. E.g. let us say we have a setuid-root utility which allows non-root users to mount USB memory sticks at a certain place in the filesystem such as `/mnt/sticks`. A malicious user could create the above block device inode on this USB stick and give 666 permissions to it. Then, when the removable volume is mounted, this would allow the non-root user to open the raw disk partition for reading and writing! To guard against this, the `nodev` mount option is used which tells the kernel to ignore all block and character device inodes on that mounted filesystem. (A similar `nosuid` option ignores the setuid and setgid bits on executables for security reasons).

E.g. Opening and Reading a block device special file

In traditional UNIX systems, these device special inodes are created statically in a directory called `/dev/`. The `mknod` command, which requires root access, creates device nodes. Let us say we execute the command:

```
more /dev/sda1
```

This will page through the raw contents of the first partition of the first hard drive (not a particularly useful operation though.) Within the kernel, calling `open` with the pathname `/dev/sda1` performs the following steps:

• Allocate a free file descriptor number from `current->files->fdtable[]`
• Allocate a `struct file` and point fdtable[fd] to it.
• Lookup the supplied path name, resulting in a `dentry`. Point `file->f_dentry` to it.
• Get the `inode` corresponding to `dentry->inode`. (If it is not already in the inode cache, read it in from disk). Check the inode permissions to see if the open is allowed (generally only root is allowed to open raw disk devices). At this point, the inode methods table is still the one associated with the parent filesystem (e.g. ext2fs).
• Seeing that the inode type is a device special file, the inode is modified so that it appears to be in one of two dummy filesystems, `bdev` or `cdev`, for block or character special nodes respectively. The generic open routine for either of these dummy filesystem types looks at the major device number to find the device driver descriptor. The device driver's

open method is called. In our case, this is a block device. Generally, block devices are already "open" because they were probed during system startup, so there is nothing much to do here, other than to note within the private section of the inode that our target is SCSI disk "a", partition #1.
• The system call returns with the file descriptor. When we later perform a read system call on this file descriptor:
• the `read` method of the inode is invoked. This points to the generic read method for block devices.
• The "buffer cache" is consulted to see if we already have an image of this particular range of disk sectors in memory.
• Assuming that we do not, an I/O request is formed and submitted to the block I/O subsystem, which dispatches the request to the correct driver. A page frame is allocated and "pinned" for DMA transfer. The `address_space` data structure for the block device inode is updated to reflect that this page frame contains (or will contain) the image of those blocks within the disk.
• The driver is `sd`, the generic SCSI disk driver. `sd` knows that SCSI disk "a" is, let us say, target #1 on channel #0 of SCSI host controller #0. `sd` creates a SCSI READ REQUEST message and invokes the host controller driver for controller #0 to deliver that message on channel 0 to target 1. The `sd` driver also translates the offset, which is relative to partition #1, into the Logical Block Address for the disk drive. The read system call goes to sleep.
• The drive receives the request. When it has read the requested sector or group of sectors, it responds with a DATA TRANSFER message. The host controller receives this message and does a DMA operation to transfer the data into the DMA buffer page frame. Once the data transfer is complete, the controller raises an interrupt. Note that the CPU is not involved with the actual data transfer.
• The host controller driver handles the interrupt. It determines which disk I/O request just completed, and wakes up the read tasks that was sleeping on that request.
• The read system call wakes up and copies the data into the user-supplied buffer.

Interrupts

Interrupt handlers are effectively stealing CPU cycles from whatever task happens to be running at the time. Recall that interrupts can be handled either when the processor is executing user code, or when it is executing kernel code. Interrupts may interrupt the kernel in either a synchronous (fault/system call) context, or in an asynchronous one (i.e. executing another interrupt handler).

Because interrupt handlers are barging in uninvited, it is only polite that they consume as few CPU cycles as possible before executing the IRET instruction. Interrupt handlers should not perform extensive computation or data structures manipulations, and they

must absolutely, positively never block.

If an interrupt handler wants to acquire a spin lock, it must be a spin lock which only interrupt handlers grab. Conversely, a spin lock which is needed by both synchronous and asynchronous kernel code must also be protected by disabling interrupts. Otherwise, a synchronous kernel routine could acquire a spin lock and then be interrupted by an interrupt handler. The handler would stall trying to obtain the spin lock, which of course would never be released because the kernel control path holding the lock has been interrupted.

Because interrupt handlers "steal" CPU time, they bypass the scheduler's policy attempts. If device drivers are coded poorly, and have bulky interrupt handlers, then heavy I/O will make the system feel unresponsive, and scheduler tuning efforts, such as using the `nice` command or system call, will not be effective, because interrupts are not scheduled, they just happen. We'll see how Linux approaches this problem with deferrable functions.

Interrupt Registration

When a device driver first initializes its hardware, the driver code registers with the kernel the interrupt vector(s) that the hardware is expected to use. When an interrupt arrives with a particular vector code, the generic kernel interrupt handler checks each registered device driver interrupt service callback routine. Each driver must check to see if its hardware caused the interrupt (e.g. by looking at a bitwise flag in the device's control register). Handling of a particular vector is serialized, e.g. while handling IRQ3, another instance of IRQ3 can not intervene. In almost all cases, the IRQ line that a particular device uses can be configured, and the kernel attempts to place each device or device group (handled by the same driver) on a unique IRQ. The pseudofile `/proc/interrupts` will list the active IRQs on your Linux system.

Interrupt Handler Synchronization

In general there are three phases to handling an interrupt (e.g. disk I/O complete).

- **Critical actions:** These must be performed with interrupts on the local processor disabled (the `CLI` instruction on X86), otherwise race conditions with severe consequences could occur. Critical actions include updating the kernel data structures which track pending interrupts, saving registers on the stack, and acknowledging the interrupt by writing to the appropriate motherboard interrupt control register. On the X86 architecture, further interrupts are already blocked when an interrupt handler is invoked.
- **Non-critical**: Some actions may be performed very quickly by the interrupt service routine with no blocking problems. After all of the critical actions take place, local interrupts can be enabled again (the `STI` instruction on X86) and these non-critical tasks could be interrupted by another handler Such non-critical tasks include: checking if the particular hardware device actually caused the interrupt, acknowledging the interrupt for

that particular device, and in some cases reading data.
• **Deferrable**: Actions which might block (e.g. shared data structure which may be locked from a system call) or which might take a long time (e.g. copying a large data buffer) are not executed during the reign of the interrupt handler. Instead, the interrupt handler notes that there is still work to be done, and allows the handler to return. At some later time, this deferred work is performed.

Deferrable Functions, tasklets and work queues

The Linux kernel has several mechanisms by which work can be deferred The basis for all of them is a **Soft IRQ**. There are a maximum of 32 soft IRQs in the (X86-32 Linux) kernel, controlled by a bitwise flag word. This is checked at periodic points within the kernel, including when returning from an interrupt. When a particular softIRQ has been activated by having its bit turned on in a device driver interrupt handler, the associated function gets called at the time that softIRQs are being checked. The softIRQ flag is cleared when the soft handler is called, but that handler may determine that the activation condition still exists and may thus turn itself back on.

One of the main ideas of deferrable functions is that asynchronous kernel code should not be able to steal too much CPU time from user processes or synchronous code. To address this, if softIRQs are called for more than 10 times in a row while returning from a hard IRQ, the kernel stops looking at them, allows the hard IRQ to return, and wakes up a kernel task called `ksoftirqd`. This task also runs any active softIRQs, ad nauseam, but because it is a task it is in a synchronous context and is subject to the whims of the scheduler. It thus competes fairly for CPU time with other tasks.

The softIRQ mechanism is used extensively within the kernel to deal with timeout-related events, in conjunction with the kernel's timebase system which allows callback functions to be established which will happen a certain number of ticks later. An example would be retranmission of network data in the TCP protocol. When the TCP segment is transmitted, a timer callback is registered, then cancelled when the ACK arrives. If there is no ACK, the timer is called back, and this activates a softIRQ which ultimately leads back to the TCP protocol routines and effects the retransmission. Note that the TCP routine is not called directly from the timer interrupt context.

The softIRQ mechanism is also used to implement **tasklets**. These allow device drivers, network protocol drivers or other kernel code to dynamically register a deferred function and activate it on demand. If there are any tasklets activated, this causes the softIRQ allocated to the tasklet subsystem to be active. Tasklets are serialized. Let's say a device driver has registered a tasklet to copy data out of a device buffer. It does not have to worry that this tasklet might be called simultaneously on two different processors.

Neither softIRQs nor tasklets may execute blocking operations (e.g. sem_dec), because they potentially execute in an interrupt context, on somebody else's time. Work queues

are used when the kernel needs to perform a job which may potentially block or take a really long time. A kernel task is allocated to each work queue.

Character special devices

Unlike block devices, character devices are generally not random-access and not meant to store files. Examples of character devices include the keyboard, mouse, sound card, serial ports. As with block devices, when a process opens a character device special file, it is intercepted and the inode associated with the open file is one from a dummy filesystem, in this case `cdev`. Thereafter, read and write operations are handled by the device driver, which is free to impose its own semantics which may be different from ordinary file I/O (e.g. when reading from a terminal, by default read blocks until a newline is received).

Unlike block device files, read and write of character devices does not involve the page cache. It is not possible to `mmap` a character file. The methods which a character device driver exposes are: open, read, write, release (last close), ioctl.

The last method, ioctl, is a catch-all interface for any and all control operations on the device, such as setting the speed. An example of the use of `ioctl` will be given below.

The TTY

The term "tty" has its roots with the original serial printer-terminal of yesteryear, the TeleTYpe:

While the thirst for graphically-oriented user interfaces has made old news out of text-based systems, a lot of important stuff still gets done over ttys (although generally not the hard-copy style depicted above). It is certainly the most efficient way of connecting to a

computer.

The role of the UNIX tty driver

Under most operating systems, the tty device driver provides command-line-oriented functionality: As characters are received, they are accumulated in a line buffer. Characters are echoed back to the "screen" (this is known as full-duplex mode). Local editing, in the form of backspace,word backspace and line redo is provided. When the end-of-line character (CR or LF) is received, then and only then is the accumulated line released to the caller of read(). This functionality is known as **line discipline**, and this specific mode of operation is known as **canonical mode**.

Below the line discipline layer is the actual hardware side of the driver. For real tty ports, this is an interface to a UART chip which implements one or more RS232 ports.

ioctl

The ioctl system call is used to get and set parameters of I/O devices, and is one of the most un-orthogonal and overloaded aspects of the UNIX filesystem interface!

```
int ioctl(fd,request,arg)
int fd,request;
char *arg;
```

fd refers to an open device special file. The *request* code is an integer which is unique at least among all the ioctl requests supported by a device driver. *arg* typically points to a structure containing arguments for the ioctl operation, although it can also just point to an int, and in some cases (antiquated usage), it isn't a pointer at all, but rather an int!

An include file, often found in `<sys/>`, defines the request codes and structures for a particular device driver or class of drivers.

To effect ioctl over ttys in UNIX, there are many ways of doing the same thing, because many different tty handling schemes have merged and diverged over the years. We'll look at the System V/Solaris termio approach, which is supported under most versions of UNIX. For more information, read the man pages `termios` and `tty_ioctl`

```
#include <termios.h>

/* struct termios, as defined in termio.h, shown below */

struct termios {
        unsigned short  c_iflag;        /* input modes */
        unsigned short  c_oflag;        /* output modes */
        unsigned short  c_cflag;        /* control modes */
        unsigned short  c_lflag;        /* line discipline modes */
        char    c_line;                 /* line discipline */
        unsigned char   c_cc[NCCS];     /* control chars */
};
```

The current terminal settings are retrieved with TCGETS and set with TCSETS:

```
struct termios t;
        if (ioctl(fd,TCGETS,&t)<0)
        {
                perror("TCGETS failed");
                return -1;
        }
        t.c_cc[VERASE]='\177';          /* Set erase char to DEL */
        if (ioctl(fd,TCSETS,&t)<0)
        {
                perror("TCSETS failed");
                return -1;
        }
```

There are many flag bits and options which will not be covered here. Some are archaic, such as the specification of how much delay should be added between the output of a Carriage Return character (ASCII code 0x0D) and more printable characters (the purpose being to allow simple mechanical printing devices time to move the print head back to the start of the line). Others are specific to RS232 hardware, such as baud rate, parity and stop bits.

Let's just look at canonical input processing. If the ICANON flag is set in *c_lflag*, then **canonical mode** input processing is set. In this mode, the device driver performs local line editing (allowing backspace, kill, etc.). When the end-of-line character is received, the line is made available. This means that read will block until the entire line is ready, then return that line. Therefore, read on a tty in canonical mode will always (unless the typist hits a lot of keys before hitting return) return fewer than the requested number of characters. Note that the newline character **is** returned by read. When an end-of-file character is received (^D by default) as the first character of a line, read returns 0.

The ECHO flag bit in *c_lflag* determines whether received characters are echoed back (full-duplex operation). This bit is turned off, for example, by the getpasswd() library routine. While in canonical mode, the *c_cc* array contains the character codes associated with various functions, such as VERASE, VWERASE, VKILL, VEOF.

When the ISIG bit is set in *c_lflag*, receipt of c_cc[VINTR] will generate a SIGINT, and receipt of c_cc[VQUIT] will generate a SIGQUIT, to be delivered to the process group which is currently attached to the terminal (see below under Job Control). A SIGINT is also generated when BRKINT is set in **c_iflags** and an RS232 line break is received.

When the ICANON bit in *c_lflag* is clear, the terminal is in **raw mode**. The characters in the *c_cc* array no longer represent VERASE, VKILL, etc. Instead, just two slots are used: VMIN and VTIME. Characters are not assembled into lines, but rather are made available to read based on the settings of c_cc[VTIME] and c_cc[VMIN] as follows:

TIME==0, MIN==0: read returns immediately with whatever characters are available (possibly 0)

TIME==0, MIN>0: read blocks indefinitely, until at least MIN characters have been read.

TIME>0, MIN==0: read returns as soon as a character is available, or when the timer expires, whichever comes first. The timer starts when read is called, and is of duration TIME*100 msec.

TIME>0, MIN>0: read returns when MIN characters are available, or it has been more than TIME*100 msec since a character has been received.

In all cases, the count argument to read represents the maximum number of characters which the caller is prepared to take. The return value is the actual number of characters read.

The same tty settings can be queried and set from the command line using the `stty` command.

Job Control and /dev/tty

Most UNIX variants provide tty job control, which facilitates running multiple jobs from the same tty line.

Job control is performed by having the tty driver maintain a **terminal process group id**, i.e.: the process group id (pgid) of the process group (job) which is in the foreground and attached to the terminal. The pgid in turn is simply the process id of the process which happens to be the group leader. A process group, or "job", is a group of interconnected processes, typically a pipeline. Input from the tty is delivered to the foreground jobs, and output from the foreground job is delivered to the tty. Furthermore, each tty is associated with a **session id**. When a tty is disconnected, by default, `SIGHUP` is delivered to all processes in that session.

The shell manipulates the terminal process group id and session id by performing `ioctl`s on the tty device, and manipulates the pgid and sid values of each of its processes with the `setpgid` and `setsid` system calls.

The special pseudo-device `"/dev/tty"` is always associated with the controlling tty of the process. It is useful if it is necessary to receive input from the tty, for example, to read a password, and stdin has been redirected.

When the user hits the suspend key (c_cc[VSUSP], normally ^Z), a SIGSTOP is sent to all the processes in the foreground process group. At this point the shell, which was blocked in the wait system call, receives notification of STOPPED processes. The shell performs an ioctl to change the terminal process group id of the controlling tty back to the shell, allowing the user to type more commands. The previously running job is still stopped. If the user types "bg", the job is sent a SIGCONT and it resumes, in the background, allowing the shell to accept the next command. If the user types "fg", the job is brought back to the foreground by resetting the tty pgid to the pgid of the job, and the shell goes back to sleep pending the completion or stopping of the fg job.

When a backgrounded job (pgid != tty pgid) attempts to read or write from the tty, it is hit with SIGTTOU or SIGTTIN, since that job is not the one attached to the tty. The default result of these signals is to STOP the process(es). The job must be brought to the foreground by the shell before it can continue.

Daemons

Programs which are not intended to interact directly with a user via the terminal but instead are servers are known as `daemons`. Conventionally they are given names ending in

lowercase ’d’, e.g. `httpd` is the server daemon that handles HTTP requests. Most often daemons interact via network connections but there are others that are purely on the local system.

To become a daemon, the process typically forks once and allows the parent process to exit(0). The shell that invoked them thus gets an immediate return. Then the daemon must disassociate itself from the user environment. It closes all file descriptors, typically redirecting 0, 1 and 2 to a logging file, and establishes itself with a new session id and process group id, which prevents it from receiving signals relating to job control actions or tty events. It is also a good idea to `cwd` to a different directory, unique to the daemon.

Daemon processes typically exist forever. By convention, they respond to SIGTERM with a graceful shutdown. SIGHUP (there can be no "hangup" since the daemon is not connected to a tty) is conventionally used to signal a daemon to re-read its configuration files. Some daemons also respond to more complex control directives via a named pipe or network socket.

Pseudo-ttys

Ironically, pseudo-ttys are now the most common form of ttys, and real ttys are relegated to console access lines, modems, etc.

A pseudo tty is actually a pair of pseudo-devices: the slave and the master (or controller) device. The slave appears to be a tty device. ioctls which work on real ttys work on the slave device (except of course hardware-specific parameters such as baud rate). The master takes the place of the hardware-specific portion of a real tty driver.

Pseudo-ttys are commonly used in for remote network login and terminal windows under a graphical user interface such as X-Windows.

The master/controller device is opened by a process providing the "back-end" of the emulated tty, e.g. the sshd daemon. Any characters written to the master become available on the slave side as if they were received over a serial line. Similarly, any characters written to the slave side are read on the master side.

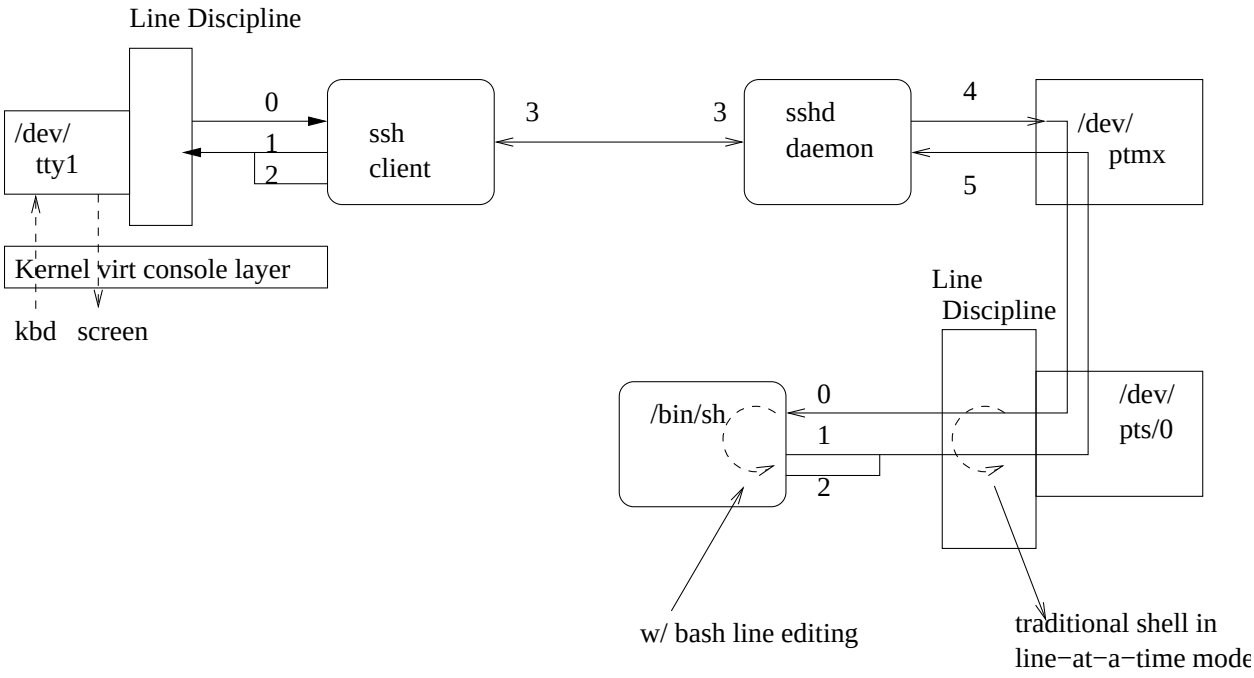
Another use of virtual ttys is the virtual console switching provided by Linux. A group of devices, `/dev/tty0`, `/dev/tty1`, etc. forms the set of virtual consoles. By pressing the Ctrl-Alt-Fn key sequence, the user can change to which virtual console the keyboard and screen are currently attached.

In the illustration below, a user on one system is using virtual screen /dev/tty1 and running the ssh client program. It has established a TCP/IP network connection to a remote system using the SSH protocol. On the remote system, the sshd daemon has handled the SSH connection. It has opened the pseudo-terminal master device `/dev/ptmx`, and has issued an ioctl to cause a pseudo terminal slave, `/dev/pts/0`, to be created.

Keystrokes entered on the left system become readable characters from `/dev/tty1`. This tty device is in non-canonical mode so that as soon as a single character is received, it is available on the file descriptor. Characters are read by the ssh client on its file descriptor 0. These characters are packaged into the SSH protocol and transmitted via the network socket on fd 3. On the right-hand system, the sshd daemon receives the SSH protocol messages and writes the characters into its fd 4. File descriptors 4 and 5, connecting to the master side of the pseudo-tty system, are connected by the driver to the slave side.

If the terminal is in canonical mode, then the line discipline module associated with `/dev/pts/0` performs local echo and line edit functions, and no data are readable on fd 0 of the shell until an entire line has been accumulated. With shells such as `bash` that perform their own line editing, the terminal is in non-canonical mode and the shell reads each character as it is received, and performs echo.

Likewise, any characters echoed back by the line discipline module, or written to file descriptors 1 or 2 of the shell, or to any child processes spawned by the shell, get passed back through the pseudo-terminal pair and are read on fd 5 by sshd, packaged into an SSH message, sent across the network socket, unpacked by the ssh client, and written to fd 1, where the virtual console tty driver displays them on the screen.



Interprocess Communication (IPC)

A process is of little use if it is unable to communicate with other processes (and users). There are many forms of IPC in UNIX. We have already seen how pipes (and "named pipes") can be used to connect processes to form a pipeline. UNIX also provides the **System V IPC** mechanisms of shared memory, message queues and IPC semaphores. These are all somewhat dated and cumbersome, although they are still sometimes used in new code. A newer message queues implementation was standardized by POSIX and is available on recent UNIX versions; it is even less popular than SystemV IPC.

All of the above IPC methods are limited to processes executing on the same machine. Under UNIX, the **socket layer** provides access to network protocols which can connect processes on different machines. Through socket programming, the same API can be used regardless of whether the target process is local or remote.

The socket layer can be used with a wide variety of networking protocols. In this unit, we will consider the application of sockets to the internet. It is important to note that the sockets API can be used to access a number of networking protocols, and that the internet protocols described in this unit are not defined by the sockets API, and could be accessed by other interfaces. However, this particular combination of sockets and the "TCP/IP" internet protocol suite is very popular and basis for many applications and services.

The material in this unit is intended as an introduction to a complex topic that is at the fringes of Operating Systems vs Computer Networks. The reader should consult other authorities, such as Steven's *Advanced Programming in the UNIX Environment* for a full presentation on TCP/IP and sockets programming in UNIX.

What is the Internet

The roots of the **Internet** trace back to about 1969, although the protocols that form the basis of today's internet did not emerge until the early 1980s.

At its simplest, the internet is a loose collection of private networks, sharing some common protocols and policy conventions. This allows computers and other devices connected to these networks to exchange data with each other and to participate in distributed applications. This same technology can be used both on the global internet and within private networks in so-called "intra-net" applications.

The basic service provided by the Internet is known as Internet Protocol, or IP. IP service is roughly analogous to the Postal Service. Messages, known as **IP packets** are addressed to their destination by means of a unique 32 bit integer, known as the **IP address**. Each packet bears both the destination and the source ("return") address. (*Note: the 32-bit address applies to IPV4. The IPV6 protocol uses a more cumbersome*

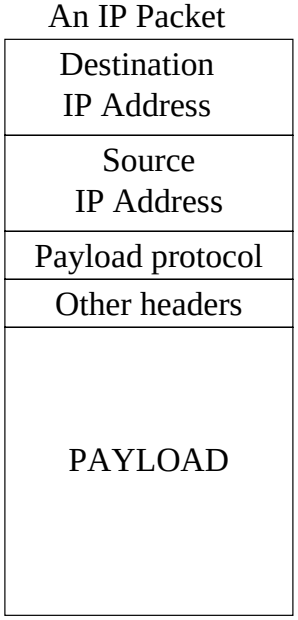
128-bit address which will not be covered in these notes)

The IP address identifies hosts within the Internet. As with buildings having multiple street entrances and thus multiple Postal addresses, it is possible for one host to have multiple IP addresses, for example, if it is connected to multiple networks.

IP addresses are represented in **dotted decimal notation**, consisting of 4 decimal numbers representing the four bytes of the 32 bit address, delimited by periods, e.g., `192.168.4.5` All numeric values in the Internet protocols are in **network byte order**, which is defined to be big-endian.

The allocation of IP addresses and their manipulation is beyond the scope of this course. Consider the IP address to be just a unique identifier. There is, however, one address which is reserved to be an alias for the local host: `127.0.0.1`.

Each IP packet is self-contained and is stateless, i.e. there is no dependency on other IP packets sent earlier or later. The destination address contains all of the information necessary to deliver the message. That task is given to dedicated devices known as **routers**.



Message boundaries are preserved. IP does not, as a rule, deliver a partial packet. Either the entire packet is delivered, or it is lost. A message is never split up and delivered as a series of smaller messages (although this splitting may be performed internally, the split messages are never delivered as such), nor are multiple messages ever coalesced and delivered as one.

Like the Postal Service, IP provides "best effort, unreliable" delivery:
• Packets may be lost in transit, in fact, that is a common occurrence.

• The latency incurred in delivering a packet is unpredictable and unguaranteed.
• Packets may be delivered out of order.
• Packets may be delivered in damaged condition (the message has been corrupted), however this is not that common an occurence since most link-layers guard against line errors.

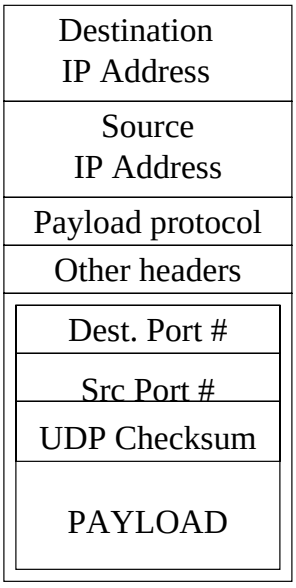
UDP and port numbers

IP addresses refer to specific hosts on the Internet. Since there are typically multiple processes utilizing services on the same machine, **port numbers** are used to disambiguate and create an array of, as it were, mailboxes within a single machine in which processes can receive messages.

The **User Datagram Protocol** is a simple wrapper around IP. It provides a source and destination port number (16 bit integer) which, combined with the source and destination IP address found in the IP packet, identifies both ends of the message transmission.

UDP provides checksumming for error detection, but many implementations ignore it, so it should not be assumed that UDP detects errors. UDP does not provide any other "value-added" services above IP.

A UDP Message
carried within IP packet



When an IP message arrives at a host, the protocol field is checked. A UDP message appears within the payload field of an IP packet, with a protocol of UDP. The IP payload is extracted from the IP packet and handed up to the UDP module which, in turn, examines the destination port number and finds a process that is waiting for messages on

that port number. The payload portion of the UDP message is extracted and becomes the message read by the user-level program.

TCP and reliable transport

The **Transmission Control Protocol (TCP)** utilizes IP to transport its payload, and provides a service akin to a pair of UNIX pipes between hostA:portA and hostB:portB.

• TCP guarantees that data will never be lost (without notification). Since it provides a stream service, it is also vital that it guarantee that data will not be delivered out of order.

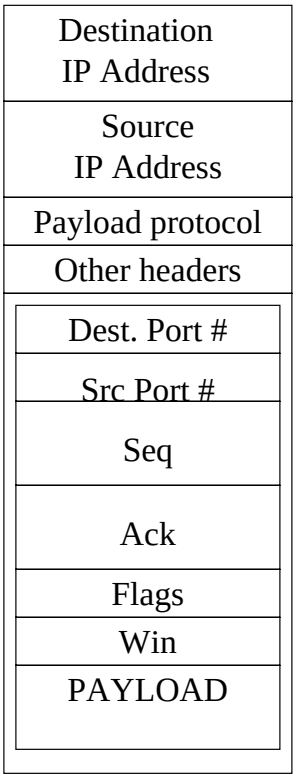
TCP is a connection-oriented service which utilizes 16 bit port numbers, just as UDP does, to identify individual "mailboxes" on hosts. The UDP and TCP port number spaces, however, are independent.

• Before data can be sent over TCP, a connection has to be made between hostA:portA, which requests the connection, and hostB:portB, which is ready to accept the connection. Either side can shut down the connection in either the read, write or both directions, just as is the case with a UNIX pipe, resulting in an "end-of-file" or "broken-pipe" indication on the remote end, as appropriate.

• Unlike the message-oriented service provided by UDP, TCP is a byte stream service. There is no concept of a distinguishable "message", again, just like a UNIX pipe.

• TCP provides flow control which limits the total amount of data that can be queued against a receiver, just as is the case with a UNIX pipe. This prevents a fast writer from overrunning a slow reader.

A TCP Segment
carried within IP packet



• TCP provides reliability by buffering written data locally until an acknowledgement message arrives from the remote end. TCP bundles pending data into a **tcp segment** which is carried within an IP packet. After sending data, TCP sets a timer which is reset when the data are acknowledged. If the timer goes off, the data are presumed lost in transit and are re-sent.
• In each message, the SEQ and ACK fields are modulo-2^32 numbers which provide a way of identifying each byte in each of the directional streams, while WIN indicates the maximum amount of additional data which the host is willing to accept on the receive side. If the stream carries more than 2^32 bytes in its lifetime, the sequence numbers simply roll over.
• The Sequence Number corresponding to the first byte sent over the connection after connection establishment is known as the Initial Sequence Number. It is chosen randomly for reasons of performance and security. The sequence numbers in the transmit and receive direction are completely independent of each other. The ACK field contains the last sequence number received from the other direction, plus one (or in other words, the next expected sequence number).
• When the last byte of data has been sent over the connection in a particular direction, the TCP protocol assigns a sequence number to represent the end-of-data mark. This is known as a **FIN**. The fact that FIN has a sequence number means that not only are the

data themselves reliably retransmitted, but also the end-of-data mark, so that when that FIN has been acknowledged, the sender has assurance that all of the data sent have been received, and that the receiver also knows that no more data are coming.
• Further details of the functioning of the TCP protocol are beyond the scope of this course.

The role of the operating system

On a multitasking operating system, the kernel is responsible for running the protocol code necessary to implement IP, TCP and UDP (at the least), and provides a facility whereby individual processes can utilize network services and associate themselves with particular ports.

Several standard APIs have existed for accessing network services. The most popular one has been the "sockets" model introduced with the 4BSD flavor of UNIX in the early 1980s. Many non-UNIX operating systems use a variant of this model, e.g. the "winsock" API under the MS Windows product family.

The first step in sending or receiving data across the network is to obtain a **socket**. A socket is just a file descriptor, created by the `socket` system call.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(domain,type,protocol)
```

domain specifies an address family to be used for this socket. For our applications, we will use `AF_INET`, which specifies TCP/IP, using IP version 4 addressing. sockets is a generic API which can support a wide range of protocols and address families. Another common address family is AF_UNIX, or **UNIX-Domain sockets**, in which the socket addresses appear to be non-persistent names in the filesystem. UNIX-domain sockets are restricted to intra-host communications, however, and will not be covered further. `AF_INET6` sockets use IP version 6 addressing, which is an extension of IP version 4. Most applications in the internet today are still based on version 4 addressing.

type specifies the type of protocol desired. In normal sockets usage, *protocol* is left as 0 and the operating system chooses the protocol based on *domain* and *type*. Within the AF_INET domain, only 2 types are supported:

`SOCK_DGRAM` implies the use of UDP.

`SOCK_STREAM` implies the use of TCP.

`socket` returns the file descriptor of the new socket, or -1 on error.

Under the sockets API, an address is specified with a `struct sockaddr`:

```
struct sockaddr {
        u_short sa_family;              /* address family */
        char    sa_data[14];            /* bare minimum address buffer */
};
```

In practice, a struct sockaddr is never used. Rather, the address structure for the particular family (e.g. `struct sockaddr_in`) is used. Since each such structure has the address family as the first 2 bytes, this works and the kernel is able to decode the address further.

Under TCP/IP, an address consists of a **network address**, which identifies a particular host, and a **port number** which resolves individual sockets on the same host. This is embodied in `struct sockaddr_in`:

```
#include <netinet/in.h>               /* Defines sockaddr_in */
#include <arpa/inet.h>                /* Historical, include it! */

struct in_addr {
          unsigned long  s_addr;                          /* Network byte order */
};

struct sockaddr_in {
          short sin_family;           /* Always AF_INET */
          unsigned short sin_port;    /* Network byte order */
          struct in_addr sin_addr;    /* A struct, historical! */
          char sin_zero[8];           /* Useless padding */
};
```

Note the seemingly pointless use of a struct to hold the network address. See <netinet/in.h> for the historical reasons behind this. Also note that s_addr and sin_port are in network byte order (big endian), not native (host) byte order! The sockets interface is generic, and although we, as trained students of TCP/IP, know that IP addresses and port numbers can be interpreted as integers, from the standpoint of the sockets API, they are "opaque data" which must be passed through to the appropriate protocol driver in the kernel without interpretation or modification.

Thus to read or write these network byte order fields in a portable fashion requires the use of the byteorder macros `htons, htonl, ntohs` and `ntohl` which are either no-ops, or byte swappers, depending on the native byte order of the machine on which the program is running.

All sockets must be bound to a source address before data are transmitted from them. To explicitly bind a specific port number, use the `bind` system call:

```
int bind(int s,struct sockaddr *name,int namelen)
```

0 is returned on success, -1 on error. Possible errors include:

> `EACCES`: attempt to access bind a privileged address and caller is not root (Note: UNIX TCP/IP, privileged addresses are port numbers <1024. Traditionally, the capability of binding a privileged port is reserved to the superuser (uid==0),

however in modern kernels, finer-grained control is possible and non-root processes can bind these ports).

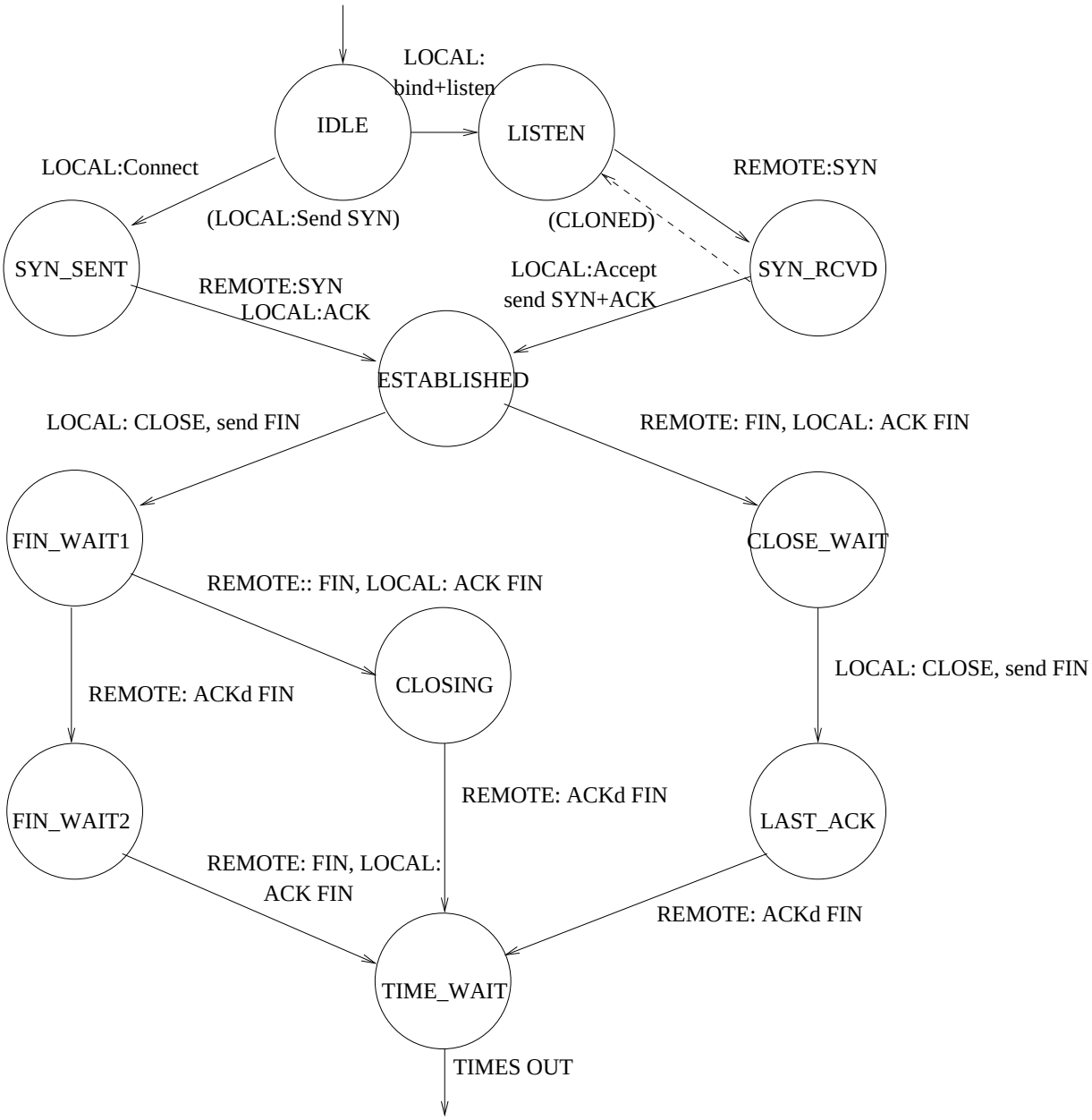
`EADDRINUSE`: The requested address is already bound to another socket somewhere on the machine.

The use of `bind` is often skipped because the sockets API specifies an automatic binding to the next available port number under certain operations (connect, send). Specifying a *sin_port* of 0 and *s_addr* of INADDR_ANY when calling `bind` also has this effect.

Note that `bind` takes a `struct sockaddr *`, which is generic. However, in sockets programming, one would typically be working with a specific structure, e.g. `struct sockaddr_in`. Therefore a type cast is needed to avoid compiler warnings.

At this point, semantics diverge between SOCK_DGRAM and SOCK_STREAM. For SOCK_STREAM in AF_INET (TCP), a connection must be established before data can be transferred. SOCK_DGRAM (UDP) is a stateless, connectionless protocol and does not require connection before data transfer.

The state diagram below is derived from the TCP protocol, with extra information along the edges to describe actions taken by either the local or the remote process. The TCP state of all sockets on the system can be queried with the `netstat` command.



There are two sides to opening a SOCK_STREAM(TCP) connection. First, the PASSIVE opener binds a specific port number and issues a *listen*, enabling that port to receive connection requests. Then ACTIVE opener issues a *connect* specifying a remote address and port number.

```
#include <sys/types.h>
#include <sys/socket.h>

int  connect(int  sockfd, struct sockaddr *serv_addr, int
addrlen );
```

This causes the TCP protocol to query the remote system (*sin_addr*) to see if anyone is listening on the specified *sin_port*. If not, `connect` comes back with ECONNREFUSED. Otherwise, the connection proceeds. `connect` blocks until the connection is either established, refused or times out.

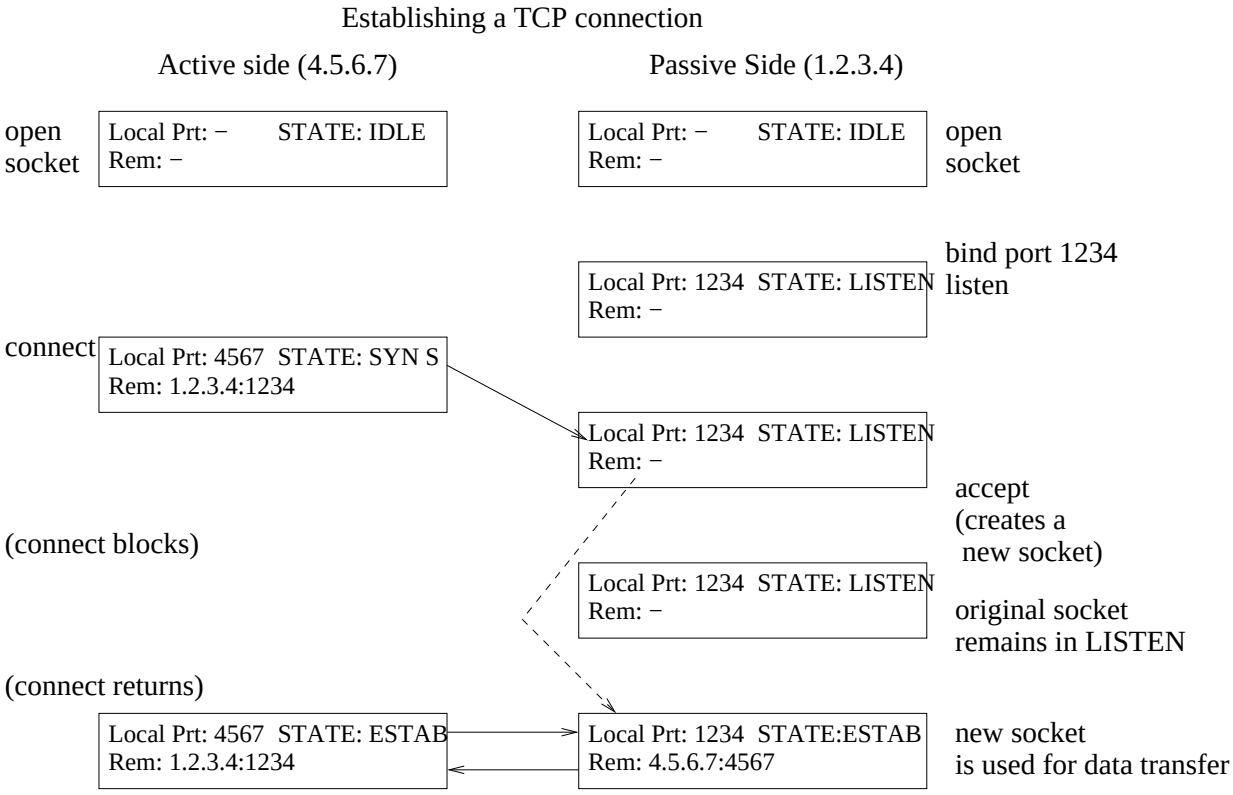
`connect` can be called for SOCK_DGRAM sockets as a convenience function. It causes the kernel to remember the destination address, but does not result in any actual protocol action.

On the PASSIVE side, the listening process uses `accept` to receive and accept the incoming connection.

```
int accept(int listener_fd,struct sockaddr *addr,int *addrlen)
```

addrlen is a **value-result** parameter. It must be set before calling `accept` to the maximum allowable size to return in *addr*. When `accept` returns, it contains the actual size used. For TCP/IP, the address is fixed-length, so this is a superfluous feature.

`accept` blocks until a connection arrives. `accept` creates a **new socket** to reference the incoming connection, and returns its file descriptor. The remote address is returned in *addr*. The original socket, *listener_fd*, continues to exist in the LISTEN state, where more connection requests can be accepted. The returned socket advances to the ESTABLISHED state, where data can be transferred between the hosts. If multiple connection requests arrive rapidly, the cloned sockets reside in the SYN_RCVD state until, one by one, they are accept'd by the process. The operating system may impose a limit on these "backlogged" nascent connections. This limit can be set (within a range) by the second parameter in the `listen` system call.



Determining at what address to listen is no small task. Certain services are generally assigned to specific port numbers, which are controlled by internet standards bodies. These are the so-called "well-known ports," such as the SMTP mail protocol on port 25, and the SSH remote login service on port 22.

This raises the issue of uniqueness. In TCP, a connection is identified by the pair {local port, local address}/{remote port, remote address}. This allows the local port number to be overloaded for servers. For example, at any time there may be multiple sockets open with a local port number of 23 (TELNET), but each is connected to a different remote address/port.

Let's look at an example of binding, listening and accepting a connection on the PASSIVE side:

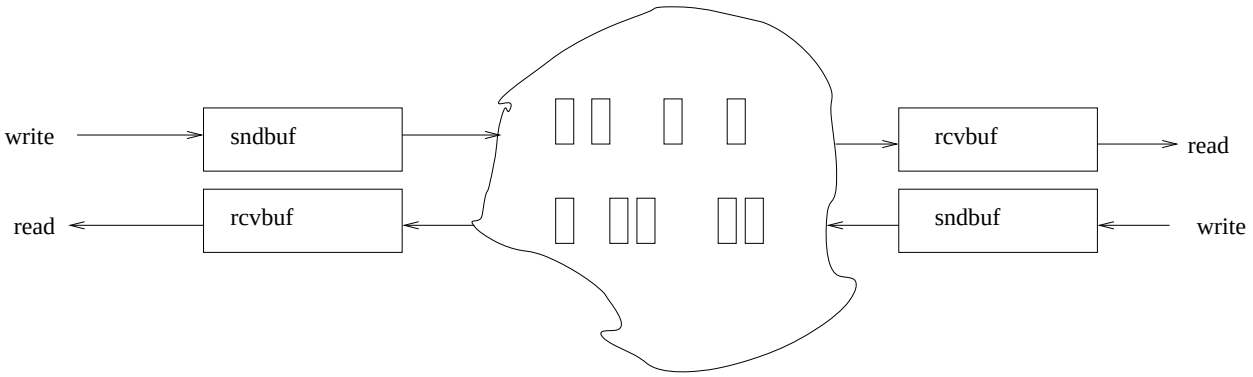
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

f()
{
struct sockaddr_in sin;
int s,s2,len;
          if ((s=socket(AF_INET,SOCK_STREAM,0))<0)
          {
                    perror("socket");
                    return -1;
          }
          sin.sin_family=AF_INET;
          sin.sin_port=htons(25);               /* The mail port */
          sin.sin_addr.s_addr=INADDR_ANY;
          if (bind(s,(struct sockaddr *)&sin,sizeof sin)<0)
          {
                    perror("bind");
                    close(s);
                    return -1;
          }
          if (listen(s,128)<0)
          {
                    perror("listen");
                              return -1;
          }
          len=sizeof sin;
          if ((s2=accept(s,(struct sockaddr *)&sin,&len))<0)
          {
                    perror("accept");
                    return -1;
          }
          return s2;
}
```

Sockets and read/write

Sockets are a form of file, and through the abstraction layer of the virtual file system code within the kernel, many of the same system calls which work for regular files on the disk also work for sockets. Under the Linux kernel, an in-core inode exists for each open socket. It belongs to file system type `sockfs` (in much the same way that pipes belong to `pipefs`). The inode type is `S_IFSOCK`.

We can think of a TCP connection as a pair of pipes connecting the two endpoints. In particular, once a connection has reached the TCP ESTABLISHED state, the two directions are independent of each other.

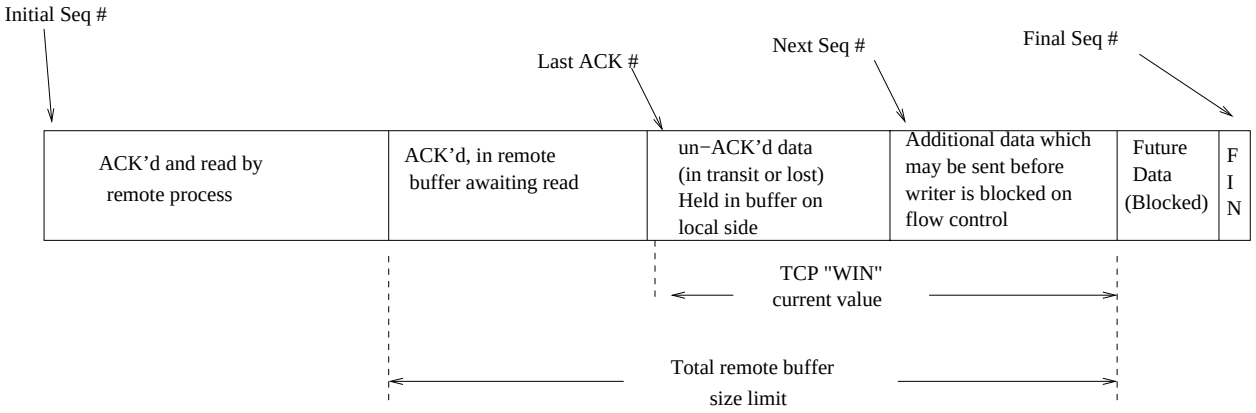


However unlike pipes which are contained in a single host, data involved in a TCP connection can be in one of three places:
• Buffered by the kernel on the sending side
• In transit over the network
• Buffered by the kernel on the receiving side

The reliability of TCP means that on the sending side, copies of the data must be retained in the send buffer until those bytes have been received at the far end (as proved by the receipt of ACK messages). Likewise at the receiving end, the kernel must buffer the data until the user-level program reads it.

Data transfer over a TCP connection can be accomplished with the normal read and write systems calls. Since TCP does not preserve message boundaries, a read will simply return all currently queued data sitting in the receive buffer, ignoring any the boundaries with which the data were originally sent. Reading from a TCP connection is akin to reading from a pipe. The read system call will block until the far end writes some data, and then will return with whatever data are available. When the remote side shuts down the connection in that direction, read returns an EOF (after all queued data have been read).

We can look at the lifetime of each half of a TCP connection as being a stream of data identified by sequence numbers. We can further categorize parts of that stream as depicted below:



The send buffer and receive buffer are of finite size, typically less than 64K. Likewise, TCP messages that are in transit are taking up buffer resources in the network, so the protocol desires to limit the total amount of data in transit (again typically under 64K). When the sender is too fast, either because the receiver is slow, or the network is slow, or both, flow control engages to keep things in balance.

The remote side will advertise a WINdow with each ACK, relative to the highest sequence number it has ACK'd, such that its receive buffer will not be overflowed. On the local side, when a process attempts to write data which would exceed the window, it blocks until the remote side moves the window. If network throughput is slow, the data which are "in transit" will grow until the they fill the window, also engaging flow control.

It is possible for "short writes" to occur when writing to a TCP socket if the number of bytes which the process is attempting to write exceeds the window size. The kernel may write whatever bytes fit, and return a short write count, or it may block the process, depending on the kernel version.

Sending and receiving UDP messages

For UDP, *sendto* and *recvfrom* are used for data transfer. Since UDP is a connectionless protocol, the destination address and port number must be specified with each transmitted message:

```
int sendto(int sockfd,
          char *msg,
          int msglen,
          int flags,
          struct sockaddr *to,
          int tolen)
```

flags is normally left 0. If a UDP socket has been *connect*ed, the destination address for all messages sent on that socket is set. *send* can then be used, ommitting the address:

```
send(sockfd,msg,msglen,flags)
```

To receive a message:

```
int recvfrom(int sockfd,
          char *buf,
          int buflen,
          int flags,
          struct sockaddr *from,
          int *fromlen)
```

`recvfrom` will block until a message is available. The next message is placed into *buf*. If the message overflows *buflen*, excess bytes are discarded for UDP sockets. *fromlen* is value-result parameter which is set to the length of *from*, and is filled in with the actual length on return, just like `accept`. The address placed in *from* is the remote address associated with the recieved message.

If a UDP socket is `connect`ed, then only messages from the connected remote address/port# will be received, and `recv` can be used instead:

```
recv(sockfd,buf,buflen,flags)
```

Closing a connection

UDP is a connectionless protocol (nevermind the fakery of `connect` on a UDP socket), so there is no notion of establishing and tearing down a connection. The `close` system call on a UDP socket simply releases the socket and file descriptor back into the resource pool, and discards any pending, unread messages.

With TCP, on the other hand, the connection is a pair of one-way pipes. Each direction can be shut down independently.

`shutdown` closes one or both directions of a connected socket:

```
int shutdown(socket,how)
```

If *how* is 0, the receive direction is shut down. If *how* is 1, the transmit direction is shut

down. If *how* is 2, both directions are closed.

When the transmit side is shut down, the remote end will receive an end-of-file indication when it has read all remaining data. Any data that have been sent but not yet received by the remote end remain queued until read.

When the receiving side is shut down, any pending unread data are discarded. An attempt by the remote system to write to the socket which has either been shutdown in the write direction locally, or in the read direction at the far end, will result in EPIPE or delivery of SIGPIPE, just as is the case with local pipes. Be aware, however, that some Linux kernel versions will not generate a SIGPIPE but instead will fail write with an ECONNRESET if the TCP connection times out with data pending in the buffer and additional writers are attempted.

When a TCP socket is closed, either with an explicit `close` system call or at process exit, both directions are also closed down, as if `shutdown(s,2)` had been invoked. Any data which are queued to be transmitted at the network layer are **not** discarded. The TCP connection continues to exist even after the close. However, if those queued data are ultimately not deliverable, that error status would be lost. An option can be set (SO_LINGER) using the `setsockopt` system call which will force the `close` to block until all queued data are successfully transmitted to the far end. For this reason, it is important to check the return value from close.

Using the DNS

Because 32 bit integers are not terribly mnemonic host names, the Internet provides a distributed database called the Domain Name System (DNS) to translate between numeric IP addresses and hierarchical, textual names, such as `cooper.edu`. DNS requests and responses are via UDP port 53. Applications use a library to isolate themselves from the details of the DNS protocol.

This mapping is potentially many-to-many. While the case of a single IP address equating to a single hostname is the most common, there are frequent cases where a single name has multiple IP addresses, or a single IP address has multiple names.

UNIX provides standard library routines to query the Domain Name System. `gethostbyname` takes a domain name and returns a list of IP addresses: `gethostbyaddr` performs the reverse operation, taking an IP adddress and returning a list of names. Both are not re-entrant safe, in that they return a pointer to static data (and furthermore the underlying structure contains further pointers to static data).

```
#include <netdb.h>
extern int h_errno;
```

```
struct hostent *gethostbyname(const char *name);

struct hostent *gethostbyaddr(const char *addr,
          int len, int type);

struct hostent {
      char    *h_name;        /* official name of host */
      char    **h_aliases;    /* alias list */
      int     h_addrtype;     /* host address type */
      int     h_length;       /* length of address */
      char    **h_addr_list;  /* list of addresses */
}
#define h_addr  h_addr_list[0]  /* for backward compatibility */
```

For simplicity, `h_name` and `h_addr` can be used by applications which don't care about the possible existence of multiple names or addresses.

In addition, some utility functions are available to convert between dotted decimal notation (e.g. 10.1.2.3) and an unsigned long:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);

unsigned long int inet_addr(const char *cp);

char *inet_ntoa(struct in_addr in);
```

`inet_addr` parses a dotted-decimal string and returns an unsigned long (32 bit) address, already in network byte order. If the string is not parseable, the all-ones value (-1) is returned. This is 255.255.255.255 in dotted quad notation, which is never a valid host address. However, it does represent the broadcast address which is used in certain system applications and thus *inet_addr* is **unclean**. *inet_aton* is preferred although some older UNIX variants do not support it. It returns zero on failure and non-zero on success. Here's an example of looking up a name:

```
f(nm)
char *nm;
{
 struct hostent *he;
 struct sockaddr_in sin;
       if ((sin.sin_addr.s_addr=inet_addr(nm))== -1)
        {
                if (!(he=gethostbyname(nm)))
                {
                        fprintf(stderr,"Unknown host:%s",nm);
                             herror(" ");
                        return -1;
                }
                memcpy(&sin.sin_addr.s_addr,he->h_addr_list[0],
                             sizeof sin.sin_addr.s_addr);
        }
```

The Domain Name System has its own set of error codes, which are distinct from system call error codes. These codes can distinguish, for example, between a negative confirmation (requested name or address definitely doesn't exist) and a system failure (no answer could be determined at this time). `herror` prints a symbolic representation of the error code of the last failed DNS operation.

Recent UNIX distributions have a new interface called `getaddrinfo`, and declare that functions such as `gethostbyname` are obsolescent. Programmers should beware that sometimes using the fancy new function makes it difficult to get your code to work on older systems.

Handling multiple sockets with select

Read and write operations on sockets may block, which causes problems for programs which need to simultaneously communicate on multiple sockets. There are two approaches to solving this: using a multi-threaded program, and using non-blocking I/O in conjunction with the `select` system call. We will not discuss threads in this unit.

To put a socket (or any other file descriptor) into non-blocking mode, use `fcntl`:

```
s=socket(....)
/* connect, etc. */
if (fcntl(s,F_SETFL,O_NONBLOCK)<0) perror("fcntl");
n=read(s,buf,sizeof buf);
if (n<0)
{
        if (errno==EWOULDBLOCK) /*it's ok */
        else perror("socket read");
}
```

Now we can use `read`, `write`,`recvfrom` and `sendto` without fear of our program becoming "stuck" in a blocking operation. The system call will just fail temporarily with an error of `EWOULDBLOCK`. (In some UNIX variants, `EAGAIN` is the error code). We can try again. However, this is only half the battle. If we were to write a program which just keeps on banging on the operation in a tight loop when it encounters `EWOULDBLOCK`, this would be what is known as a "spin loop" or "busy-wait" loop. It would mean that the program is consuming vast amounts of user and cpu time to accomplish nothing. What we need is a way of blocking and keeping an eye on multiple file descriptors at once.

The `select` system call can be used to monitor any number of file descriptors to see when reading or writing operations would succeed (at least partially) without blocking. The syntax is somewhat awkward:

```
/* Read man page on select for some discussion of new vs old
   ways of getting the right header files.  */
#include <sys/types.h>
#include <unistd.h>
#include <sys/select.h>


main()
{
 int s1,s2,maxfd;
 fd_set readfds,writefds;
        /* open sockets s1 and s2, connect, etc. */
        if (s1>s2) maxfd=s1; else maxfd=s2;
        if (fcntl(s1,F_SETFL,O_NONBLOCK)<0) {perror("fcntl s1");return -1;}
        if (fcntl(s2,F_SETFL,O_NONBLOCK)<0) {perror("fcntl s2");return -1;}
        for(;;)
        {
                FD_ZERO(&readfds);
```

```
                FD_ZERO(&writefds);
                FD_SET(s1,&readfds);
                FD_SET(s2,&writefds);
                if (select(maxfd+1,&readfds,&writefds,NULL,NULL)<0)
                {
                        perror("select error");
                        return -1;
                }
                if (FD_ISSET(s1,&readfs))
                {
                        while ((n=read(s1,...)))>0  // should read something
                                /*...*/
                }
                if (FD_ISSET(s2,&writefds))
                {
                        while ((n=write(s2,...))>0) // should write at least 1 byte
                                /*...*/
                }
        }
```

The `select` system call operates on bit vectors which represent a set of file descriptors. The mechanics of manipulating these bit vectors are best left to the macros such as `FD_SET` which are provided by the include files. The second, third and fourth parameters to `select` are pointers to these bit vectors. They are each file descriptor sets that we are "interested in". The second argument represents file descriptors for which we want to monitor read-ready events, the next is a similar set for write-ready events, and the last is described as "exception events". That's how the man page describes the sets, but to be more precise, at least with respect to Linux kernel:

• A "read-ready" event is any of the following: at least one byte of data available to be read; there has been an end-of-file condition (including remote end of socket connection has closed); there has been an error posted against the file descriptor.

• A "write-ready" event is any of the following: At least one byte of data may be written without blocking; there has been an error posted against the file descriptor.

• The "exception" event is only used for an obscure sockets programming feature called "out-of-band" or "urgent" data delivery, and will not be discussed further here.

The first argument to `select` is the highest file descriptor number set in **any** of the bit vectors being passed in, **plus one**. Another way of looking at it is: the maximum length of the bit vectors. This is a problematic part of the API. While one could just set this to some rather large number (there are various ways, some static, some run-time, of querying the maximum number of open files), doing so imposes a performance penalty in the kernel, which must examine each of the passed vectors to the limit that the first argument gives. Many programmers write utilities to front-end select and manage the bit vectors so as to keep track of the highest file descriptor number.

The fifth argument to `select` is a timeout. It is represented by a pointer to a `struct timeval`, which gives both seconds and microseconds. If the argument is `NULL`, then there is no timeout and select blocks until one or more of the events we are interested in

arrives. The `struct timeval` may be set with both elements zero, in which case `select` returns immediately.

Upon return from select, the bit vectors which were passed as pointers are modified to reflect which file descriptors had the corresponding events pending. The return value from select is the total number of bits which are set in all three vectors. Most programmers do not make use of this number except to detect error (-1 return), and instead, knowing which file descriptors were registered going in, query the result vector and act accordingly. Since the vectors are over-written by the kernel, it is often useful to make a backup copy.

poll system call

`select` is a potentially tricky interface to use, and many programmers prefer to use threads. However, multi-threaded programming has its own sets of pitfalls and gotchas too. There is another system call `poll` which performs essentially the same task as `select`, but with a different interface. In many ways it is a superior interface, but because historically `poll` was not available on many UNIX variants, most programmers gravitate towards `select`. The example below should be self-explanatory with the assistance of the man page for poll(2).

```
struct pollfd pollarry[3];
         pollarry[0].fd=fd_readsocket;
         pollarry[0].events=POLLIN;
         pollarry[1].fd=fd_writesocket;
         pollarry[1].events=POLLOUT;
         pollarry[2].fd=fd_controlsocket;
         pollarry[2].events=POLLIN|POLLOUT;
         if (poll(&pollfd,3,-1)<0)
                  perror("poll");
         else
         {
                  if (pollarry[0].revents&POLLIN)
                           do_read(pollarry[0].fd);
                  if (pollarry[1].revents & (POLLERR|POLLHUP))
                           do_error(pollarry[1].fd);
                  //etc.
         }
```