## 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.

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.

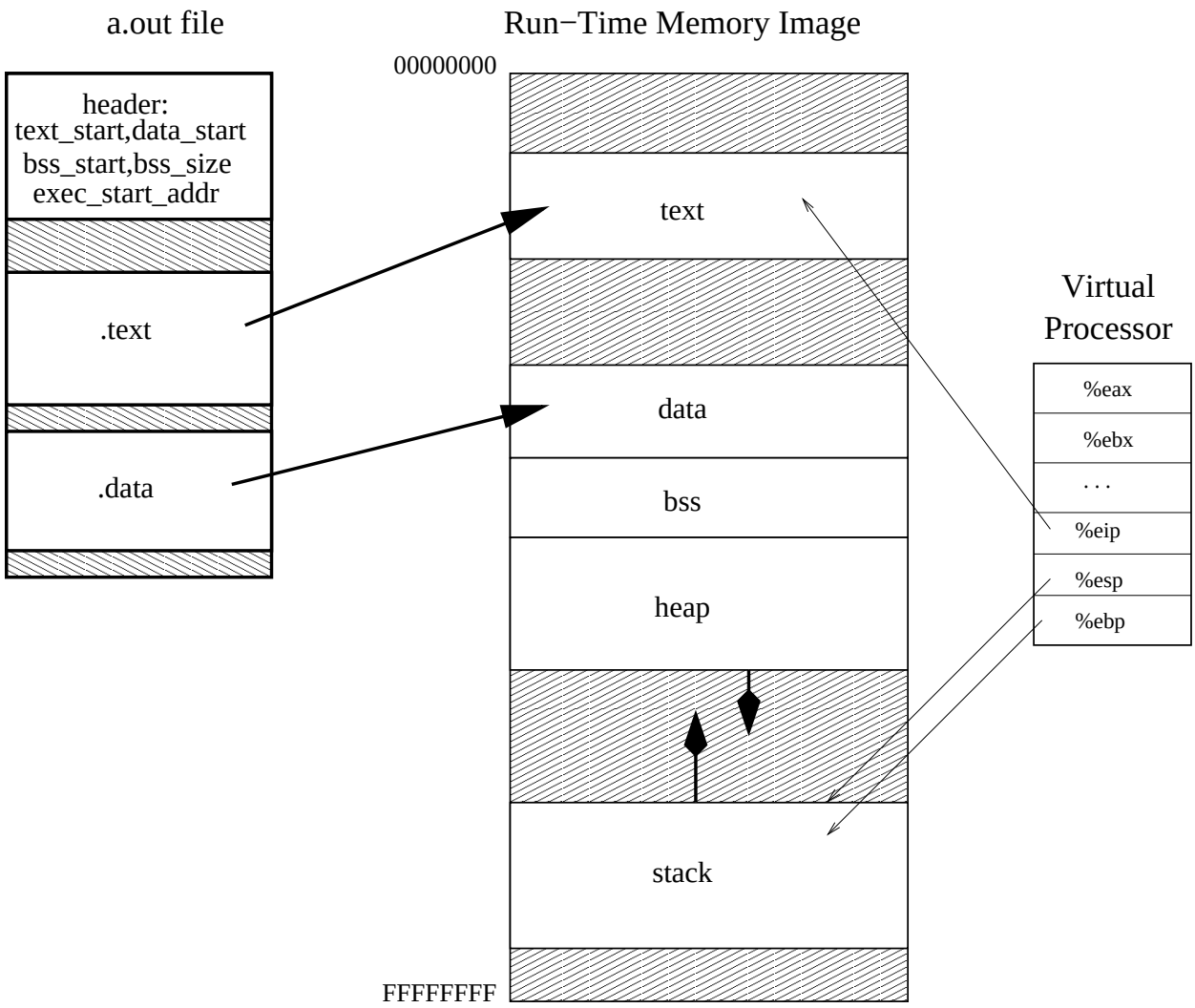
Process #1 is always at the root of the tree, and is always running a specialized system utility program called `init`. `init` is started by the kernel after bootstrap, and it in turn spawns off additional processes which provide services and user interfaces to the computer.

## Process State

The kernel maintains information about each process in kernel memory. Later we'll see the data structures used. Per-process information includes the userid (uid) and gid for access control purposes, the pid and ppid, the table of open file descriptors, resource usage counters (such as accumulated user/sys mode cpu time), and a host of other data.

## The Virtual Address Space of a Process

All UNIX processes have a virtual address space which consists of a number of **regions** 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 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 first 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 later 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[])`.

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, 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 environment variable `PATH` until an executable file with that name is found (this action is performed by the standard C library, not the kernel).

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.

If exec is successful, from the standpoint of the calling program, it appears never to return. On error, exec returns -1.

## 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 binary file (not another interpreter). It will be executed with argv[0] set to the "`interpreter`". 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 intepret (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`.

## 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, 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.

```
static 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 %\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.

## Clone

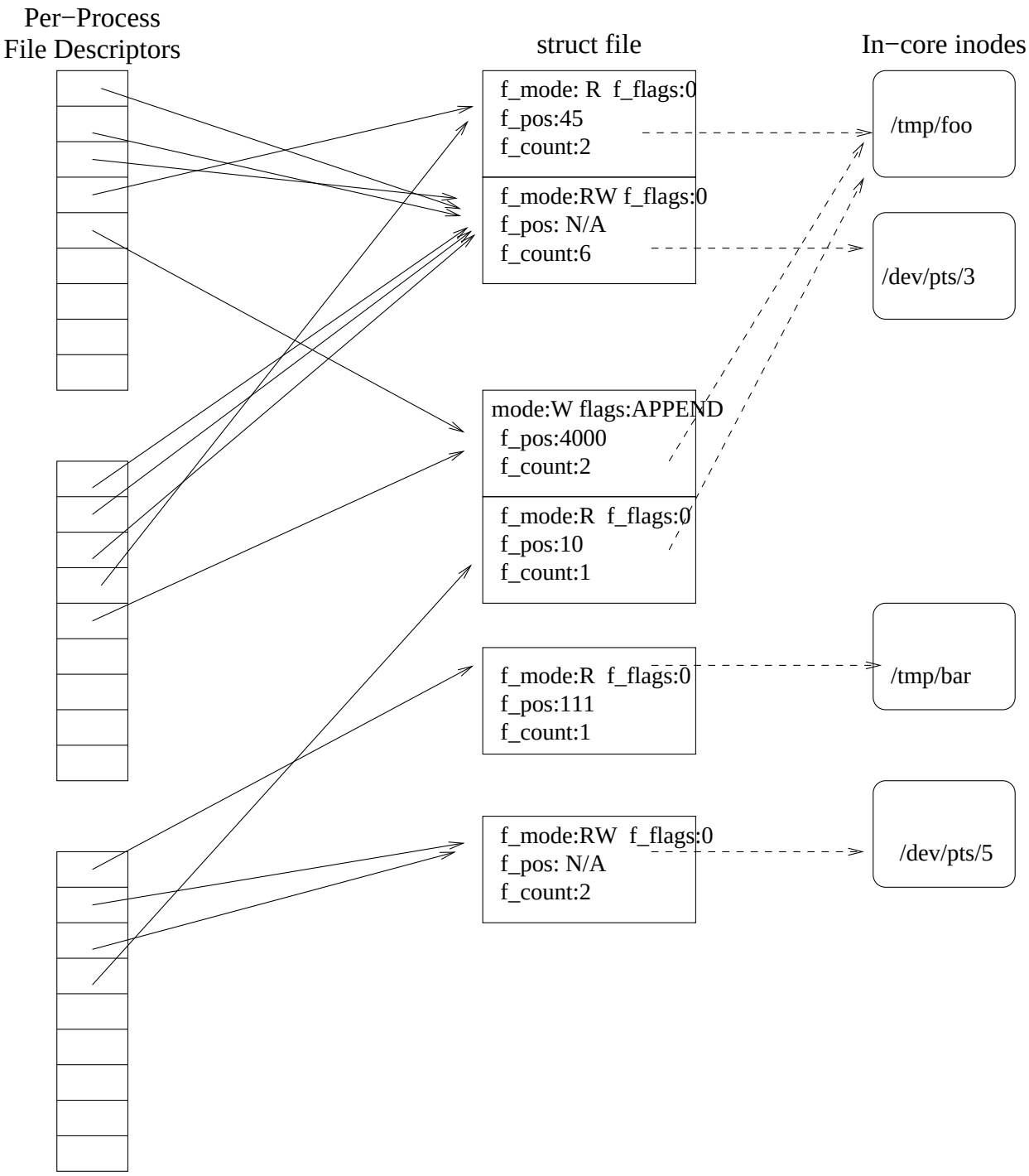
Under the Linux kernel, virtual processors and virtual address spaces can be separate issues. By using the `clone` system call rather than `fork`, a new virtual processor is created, but the many aspects of duplication which `fork` performs can be turned on or off piecemeal (e.g. the virtual address space, the relationship to open files, the current working directory). The new virtual processor is assigned a different pid.

By using `clone`, new **threads** of execution can be created which co-exist within the same address space. This is somewhat of a simplification, as it ignores complex issues such as managing the user-level call stacks of these different threads. Therefore, additional user-level library support is required to implement multi-threaded programming. The `clone` system call is rarely used directly, but instead, if the intent is multi-threaded programming, the threads library is used and it ultimately calls `clone`.

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.

## 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 files. 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 stderr:

```
if ((fd=open(logfnm,O_CREAT|O_APPEND|O_WRONLY,0644))<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 file`s 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);
        /* execute atexit callbacks */
        /* close and flush all stdio streams */
        /* other library cleanup */
        _exit(rc);
}
```

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, eventually calls the real exit system call `_exit(2)`, which causes the termination of the process, but first executes all of the cleanup routines. 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, unbuferred. In the event that the process is killed, it is beneficial to see all of the error messages leading up to that point.

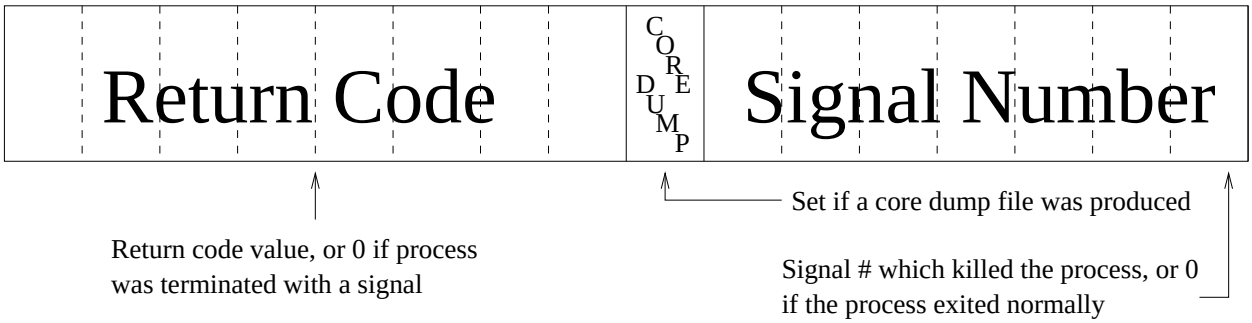
Regardless of the termination reason, when a process terminates, all file descriptors are closed by the kernel as if `close` had been called on them. All resources used by the process (such as memory) are freed (unless they are also being shared by other extant processes). Other state information (such as locks held by the process) is also adjusted.

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);
          }
}
```

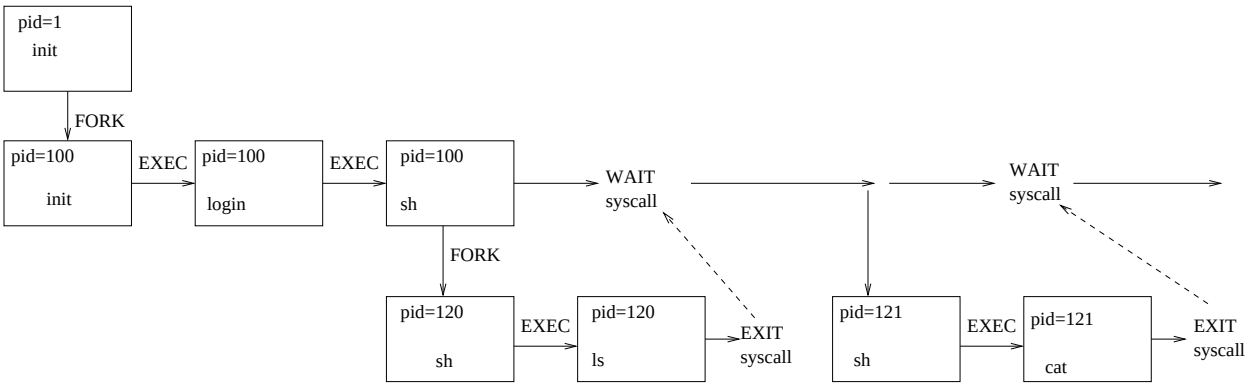
Among the resource usage information kept for each process is the total **user CPU time**

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. In later units we will see how better to manage this. 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 intialization 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 -foobarargument
ls: invalid option -- e
Try ‘ls --help’ for more information.

$ echo $?
1
```