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!

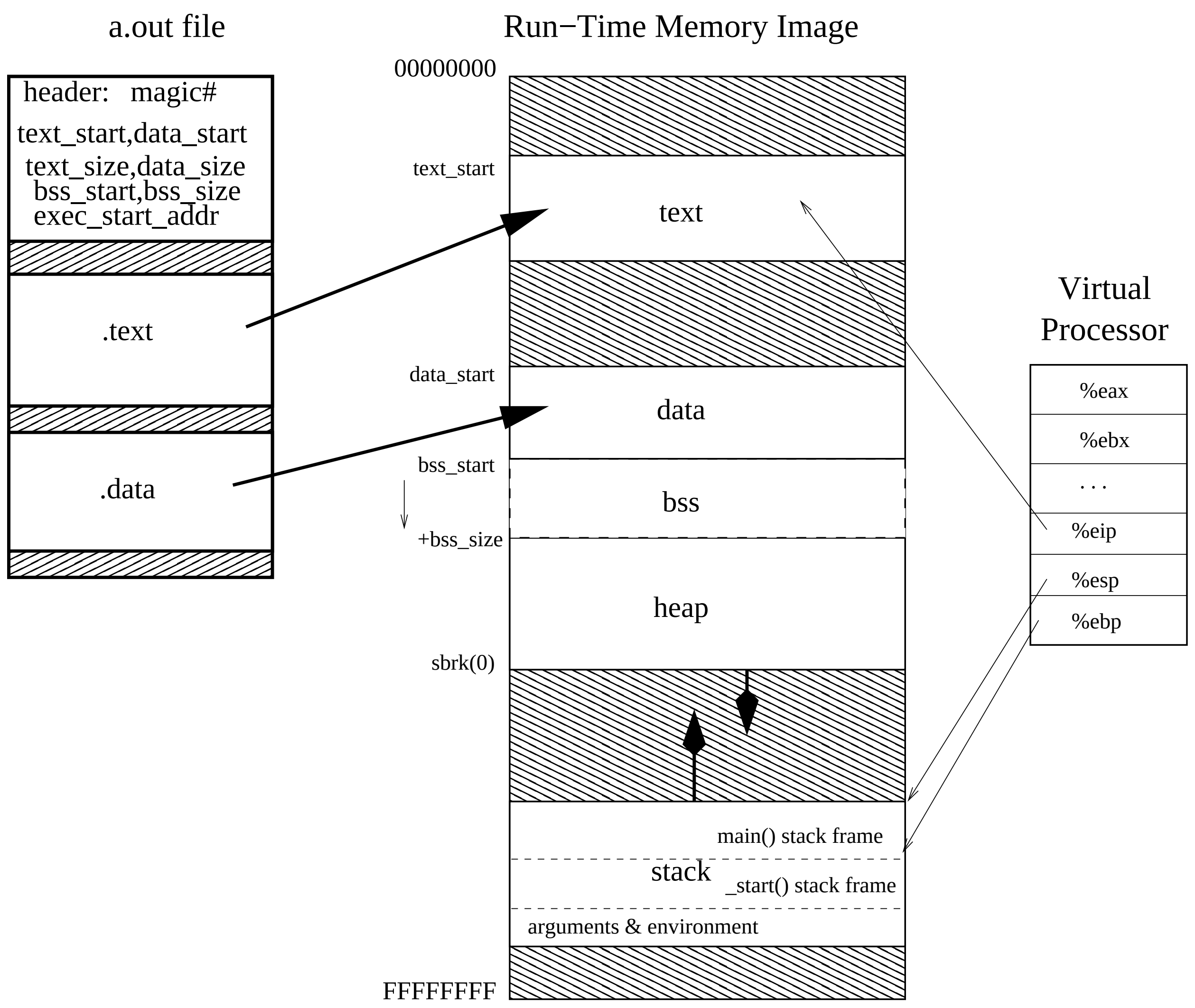
Processes are identified by an integer **Process ID (pid)**. All processes have a parent which caused their creation, and thus the collection of processes at any instant forms an ancestry tree. The pid of the current running process can be retrieved with the `getpid` system call, while the `getppid` 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. [on many modern versions of Linux, this pid #1 runs a different program known as `systemd`. However, this still performs the same basic functions as the classic `init` program]

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 (using the `sbrk` system call), 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 must 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 these 3 vital pieces of information into kernel memory, discards the entire virtual address space of the process as it currently exists. Conceptually, the kernel loads the executable image into the process's virtual memory beginning at some specific absolute virtual address. The executable file, or `a.out`, contains:
• The virtual address and size of the text and data regions.
• The virtual address and initial size of the bss region.
• The images of the text and data regions
• 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. This is covered below under "Process Termination"

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[]);// Actual syscall
 /* Below are all library functions which then call execve */
 int execv (char *path, char *argv[]);
 int execvp (char *file, char *argv[]);
 int execvpe (char *file, char *argv[], char *envp[]);
 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).

In the example below, we use the `execvp` wrapper function:

```
main(int argc, char **argv)
{
        argv[0]="dog";
        execvp("cat",argv);
        perror("We reached this point, an error must have happened in exec");
}
```

This program invokes `cat` (allowing the library function execvp to search the PATH environment variable for the unqualified name `cat`) and passes along all of the arguments. However, argv[0] is given the value "dog". In the event that cat encounters an error, it (like most programs) uses argv[0] to report its own name, with humorous results in this case (go ahead, try the example).

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. This is often called the "Applications Binary Interface" (ABI). 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 so-called "Magic Number" in the a.out header. The standards bodies which set the format of a.out files (typically the Extensible Linking Format or ELF) assign a specific magic number to each possible architecture.

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). (*Note: in most versions of Linux, nested script interpreters are permitted up to 4 in a row*)

The interpreter program will be executed with argv[0] set to `/path/to/interpreter`. If the optional argument `arg` is present in the #! line, it will be inserted as the next argument (argv[1]). This is sometimes used to modify the behavior of the interpreter program, such as to turn on debugging.

The name of the script file becomes argv[1] (argv[2] if the optional `arg` was specified in the #! line), which allows the interpreter to open this file and begin to interpret (execute) it. Interpreter programs have a syntax which is compatible with this. For example, the man page for `sh` shows:

```
sh [-optional_flags] commands_file [arg1 [arg2 ..... ]]
```

Note that if you use a "p" variant such as execlp and specify a relative filename, the name of the script file as seen in argv[1] (argv[2] if the optional `arg` was present) nonetheless appears to be fully-qualified. This is because the wrapper function such as execlp has searched the PATH environment variable and replaced the unqualified name of the script file with the actual qualified pathname as determined by the PATH.

Then the entire argv vector suppied in the original exec system call, not including argv[0], is appended to the argv seen by `interpreter`. `sh` like most interpreter programs will open `commands_file` and read commands from it, rather than reading commands from standard input. arg1, arg2, etc, if supplied, are available as shell variables $1, $2, etc. The first line of the script file, beginning with #!, does not adversely affect the script, because the # makes the rest of the line a comment in shell script language (and most other interpreted languages). With the above-described manipulations of the argv vector, scripts work as expected, and a script file can be freely substituted for a native binary file.

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`. New applications should generally avoid this and instead always use an explicit #! line.

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 path 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.

The Environment Variables are a mechanism for passing along configuration and status information from one program to the next. When a session begins (e.g. via the `login` command) the environment variables are initialized as needed for that session. `HOME` gets set to the home directory, etc. Environment variables are inherited by commands spawned off from the shell, or the environment can be modified as needed for each command.

In shell syntax, the following:
```
PATH=./:$PATH
```
adds the current directory to the PATH environment variable. Then it will no longer be necessary to prefix your program with ./ for example `./a.out`. However, this represents a security exposure since, if you happen to be in an unfamiliar directory owned by someone else, and you type ls, they could have planted a "trojan" ls program which now is running with your privileges. Therefore, ./ is not normally part of PATH.

By using the `export` shell command, a change to an environment variable is made visible to commands that are then spawned. Otherwise, the change is specific to that shell. Much more information about environment variables, PATH, and UNIX shell programming in general is widely available from online tutorials & the supplementary textbook.

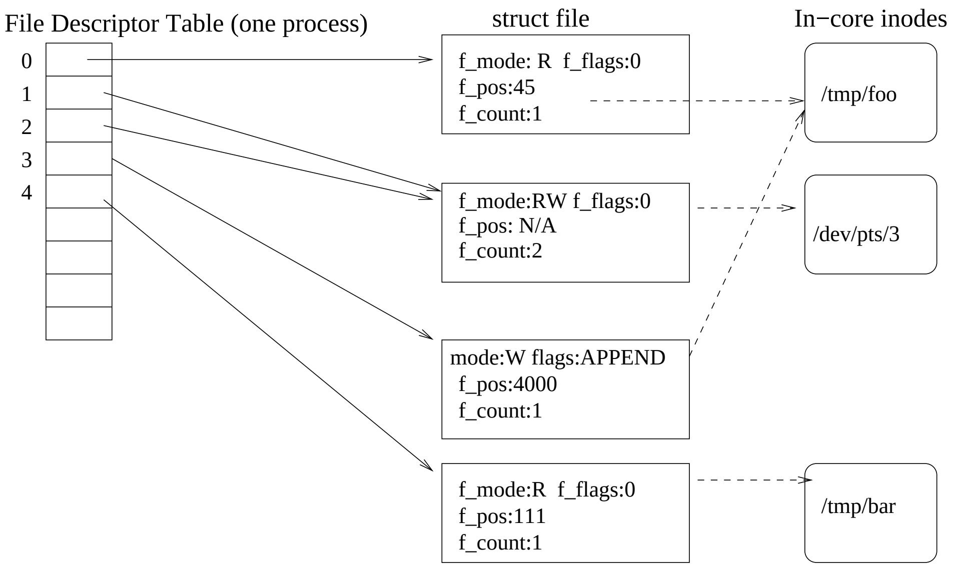
The open file table and file descriptors

We're now going to take a deeper dive on how the kernel handles file descriptors internally. There are two data structures involved. First, each process has its own **file descriptor table** which is indexed by file descriptor number. This table (array) contains pointers to another kernel data structure known (in the Linux kernel) as `struct file`. One `struct file` exists for each instance of opening a file, and these can be pointed to by multiple file descriptors, including potentially file descriptors from different processes. Lastly, each struct file contains a pointer (it is actually via another structure which is not illustrated to simplify the diagram) to a `struct inode` in the kernel, allowing access to the file's metadata and data.

`struct file` contains many fields. Right now, we are concerned with the following:
• `f_mode`: The basic manner in which the file was opened (RDONLY, WRONLY, RDWR).
• `f_flags`: The remainder of the second argument to the open system call which need to be remembered after the open. For example `O_APPEND`. In contrast, flags such as O_TRUNC or O_CREAT are only important at the time of the open and then are discarded. There are many possible flags. Some of them can be altered after the open with the `fcntl` system call.
• `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.

The `f_pos` field maintains a *cursor* into the file. It is initialized to 0 when the file is first opened. A read or write system call always 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, to perform *random access* to the file.

However, when the file has the `O_APPEND` flag, all writes automatically begin at the current size of the file, i.e. all writes will append to the file. After the append, `f_pos` contains the new size of the file. Because the setting of `f_pos` to the current size of the file and the write happen atomically, it is guaranteed that a file descriptor opened with `O_APPEND` can never overwrite previously written data. This is very useful for things such as audit logs. Of course (as long as one has the proper file permissions) one could always open the file a second time without the APPEND flag.



The diagram above is a schematic representation of these data structures, all of which reside in kernel memory. We see that standard input has been directed (see below) to the file /tmp/foo, which has also been opened separately in O_WRONLY|O_APPEND mode. File descriptors 1 and 2 continue to point to the terminal device (note `f_count==2`). Finally, file descriptor 4 is an opening of the file /tmp/bar.

The act of `open`ing a file creates both a new file descriptor and new `struct file` and links everything together.

Dup and I/O redirection

However, just as we can have hard links in the filesystem, we can have multiple file descriptors which link to exactly the same struct file. 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 then **exactly equivalent** to the original one. When the dup takes place, the `f_count` field of the `struct file` in question is incremented.

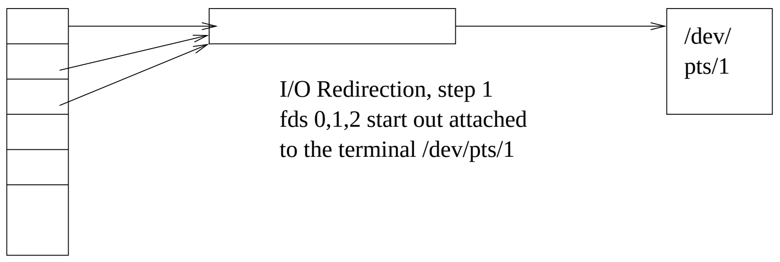
dup comes in two flavors: original `dup`, which picks a file descriptor for you (as usual, the lowest available fd number is chosen), or `dup2` which allows you to pick the new file descriptor number, which is first `close`d if already open.

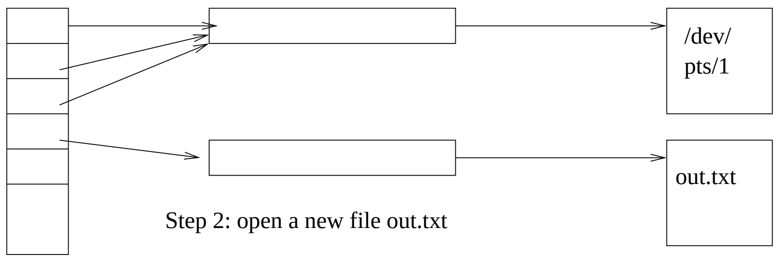
A `close` on an open file descriptor NULLS out that exact file descriptor table entry and thus removes one active reference to the corresponding `struct file`. `f_count` is decremented. When the number of references falls to 0, the `struct file` itself is destroyed. That deletes one particular instance 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.

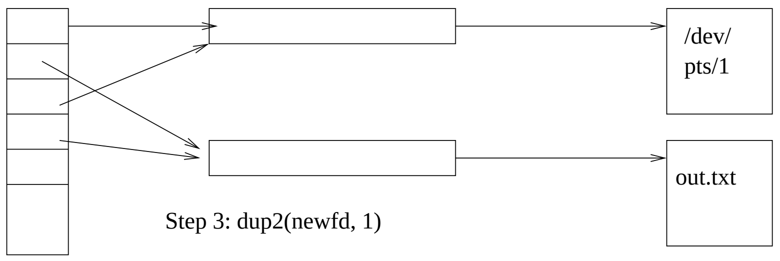
The most frequent application of dup is to redirect standard input, standard output or standard error:

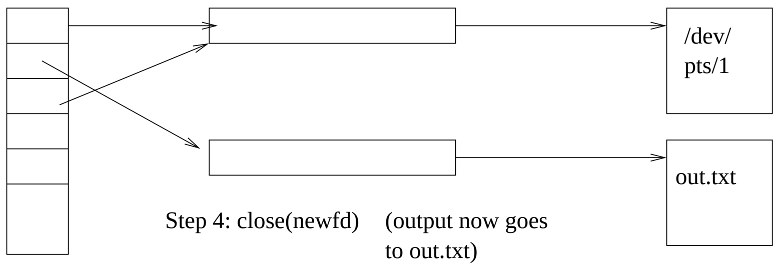
```
char *outname="out.txt";
if ((fd=open(outname,O_CREAT|O_APPEND|O_WRONLY,0666))<0)
{
          fprintf(stderr,"Can’t open log file %s",outname);
          perror("");
          return -1;
}
if (dup2(fd,1)<0) {
          //It would be very odd for the dup2 system call to fail in this case
          perror("WOW Can’t dup2 to stdout");
          return -1;
}
close(fd);
if (execlp("/usr/local/bin/nextprog","nextprog","arg1","arg2",NULL)<0)
{
          //OTOH, execlp failing is not as shocking
          perror("Whoops, can’t exec /usr/local/bin/nextprog!");
          return -1;
}
```

In this example, nextprog is invoked with stdout redirected to a log file whose name is contained in the char * variable outname. Note the close(fd). After the dup2 call, both file descriptor fd AND file descriptor 1 (standard output) point to the newly-opened file `logfnm`. The close gets rid of this extra reference to this file. See below under "Expected file descriptor environment" Below is an illustration of I/O redirection of file descriptor 1:









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.

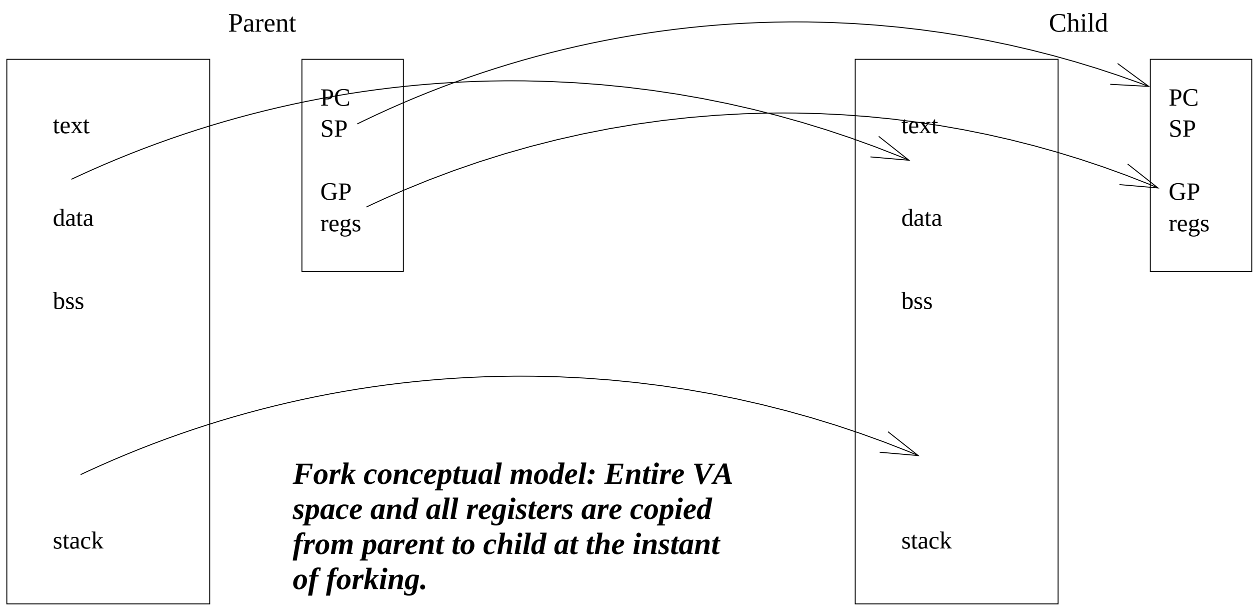
The duplication of the parent process also applies to the various properties or state variables that the kernel associates with each process. For example, the current working directory, user and group ids, and open file descriptor relationships, are all copied.

```
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("pid==%d i==%d\n",pid,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.

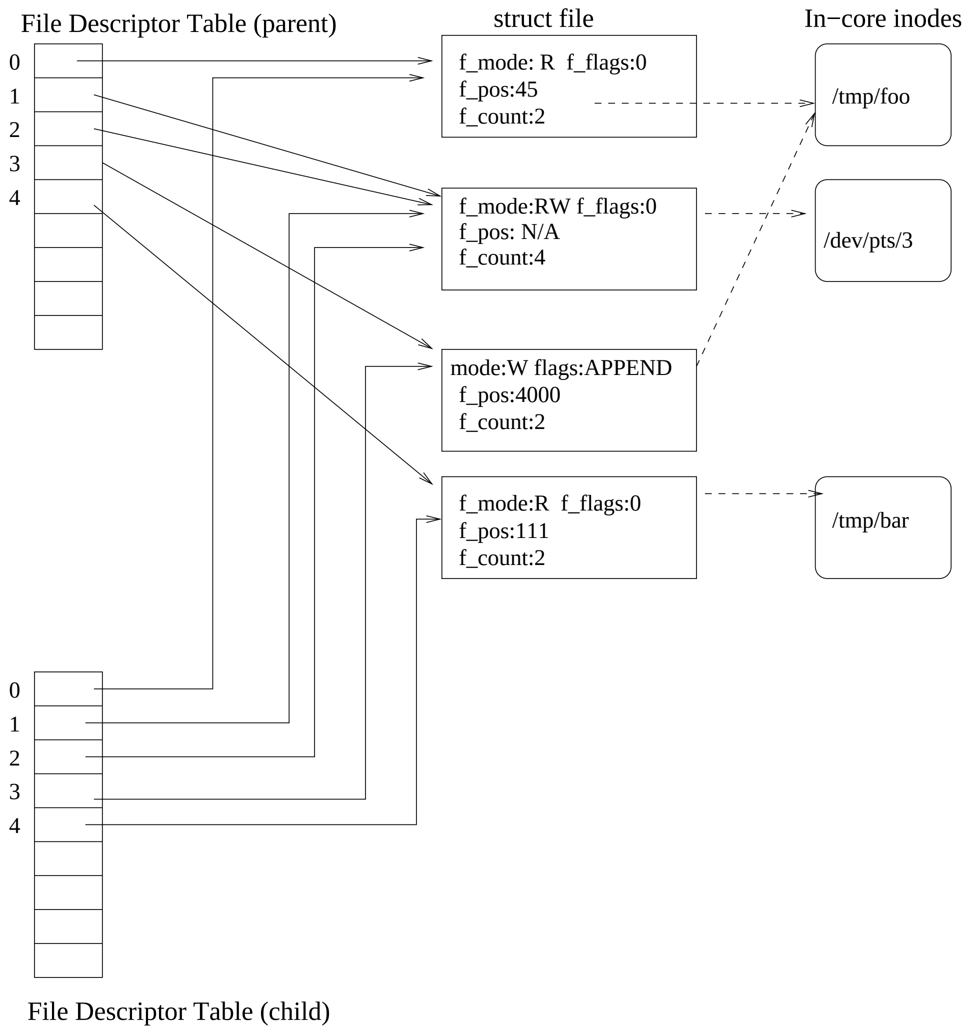
Note that is **indeterminate** whether, after the fork system call completes, the parent runs first, the child runs first, or (on a multiprocessor system) both processes run simultaneously. Programmers should not assume any particular case. Nor should any assumption be made based on examining pid numbers 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.



Fork conceptual model: Entire VA space and all registers are copied from parent to child at the instant of forking.

fork and the file descriptor table

A fork makes an exact copy of the parent's file descriptor table, resulting in an additional reference to each file table entry. The `f_count` fields of each affected `struct file` are incremented. In effect, fork performs a `dup` of the file descriptor table to the child. This sharing of `struct file`s 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`). However, any opens or closes performed in the child are not visible in the parent and vice-versa, since after the initial copying, these are two independent file descriptor tables.



Above is the same example from earlier, but after the process has forked.

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 attempting to read/write to/from the standard descriptors.

On Linux systems, the typical flow of process creation which leads to a user's interactive shell creates one instance of opening the terminal associated with the session, i.e. one struct file. The open modes are O_RDWR. That is then dup'd so that file descriptors 0,1,2 all point to it. As a result, one finds that a write to 0, or a read from 1 or 2, works just fine. This is not "portable" behavior and applications should not be coded to depend on it. In addition, when there is I/O redirection, the < shell operator always uses O_RDONLY mode and the > or 2> operators use O_WRONLY.

Troubleshooting file descriptor tables

A helpful tool on Linux is the /proc filesystem. If we look at `/proc/###/fd` where ### is the pid, we will see the open file descriptor table of that process, e.g.:

```
$ ls -l /proc/606648/fd
total 0
lrwx------. 1 hak users 64 Sep 23 20:44 0 -> /dev/pts/0
lrwx------. 1 hak users 64 Sep 23 20:44 1 -> /dev/pts/0
lrwx------. 1 hak users 64 Sep 23 20:44 2 -> /dev/pts/0
l-wx------. 1 hak users 64 Sep 23 20:44 3 -> /tmp/xyz.txt
```

The entries in this directory are given names which are the file descriptor numbers (in decimal) of each open file descriptor and each appears to be a symlink pointing to the actual path associated with the open file. (There are times where there is no path, such as pipes, which we cover in Unit 4). Above we see a process where the usual 3 fds are associated with a terminal and it has apparently opened the file /tmp/xyz.txt

Then if we read the file (e.g. with cat) `/proc/###/fdinfo/##` where ### is the pid and ## is the file descriptor number, it will provide additional information about the struct file referenced by that file descriptor:

```
$ cat /proc/606648/fdinfo/0
pos:    0
flags:  02002
mnt_id: 27
```

```
ino:    3

$ cat /proc/606648/fdinfo/3
pos:    0
flags:  0100001
mnt_id: 100
ino:    1310731
```

The `flags`: information contains the open flags, and other flags which can be set by the fcntl system call, or by the kernel itself for various reasons. In the first example, the terminal was opened with O_RDWR and the O_APPEND flag was turned on by the kernel (because the terminal is not seekable). The xyz.txt file was opened O_WRONLY and we also see the flag O_LARGEFILE which allows the file to be more than 2GB. This flag is turned on by default even when the user didn't include it in the open flags explicitly.

Process termination / The Truth about main

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 termation. 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. However, so many programmers failed to do this, leading to unpredictable return codes, that the C standard was actually modified so that the `main` function (but only that function) has an implicit `return 0;` if there is no explicit return with a specified value. In the author's opinion, this is an unnecessary perversion of the standard, but the author doesn't get to vote on this.

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 an attribute 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.

The function `exit(int rc)` which you have all been using is not actually a system call. As we can see from the pseudo-code above, it is simply a function provided by the standard C library. The "real" exit system call is `_exit(int rc)`, and forces the **immediate** termination of the process. In contrast, the library function `exit()` 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 unit 4.

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.

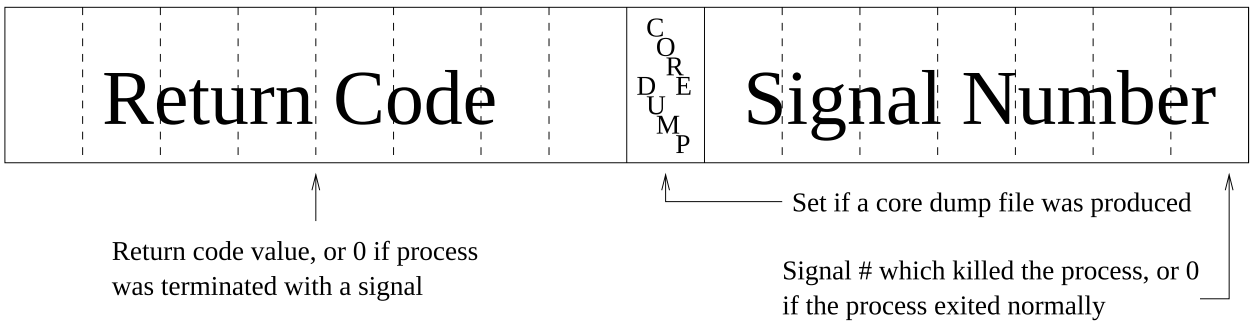
Regardless of the termination reason, when a process terminates, all file descriptors are closed by the kernel as if `close` had been called on them. All resources used by the process (such as memory) are freed (unless they are also being shared by other extant processes). Other state information (such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

Exit Status Code

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 voluntarily 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:


Return Code
CORE DUMP
Signal Number
Set if a core dump file was produced
Return code value, or 0 if process was terminated with a signal
Signal # which killed the process, or 0 if the process exited normally

```
#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 %d0,
                                        WTERMSIG(status));
                         else
                                  fprintf(stderr,"exited with nz return val %d0,
                                          WEXITSTATUS(status));
                         return -1;
                 }
          }
          else
                    fprintf(stderr,"exited normally0);
}
```

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

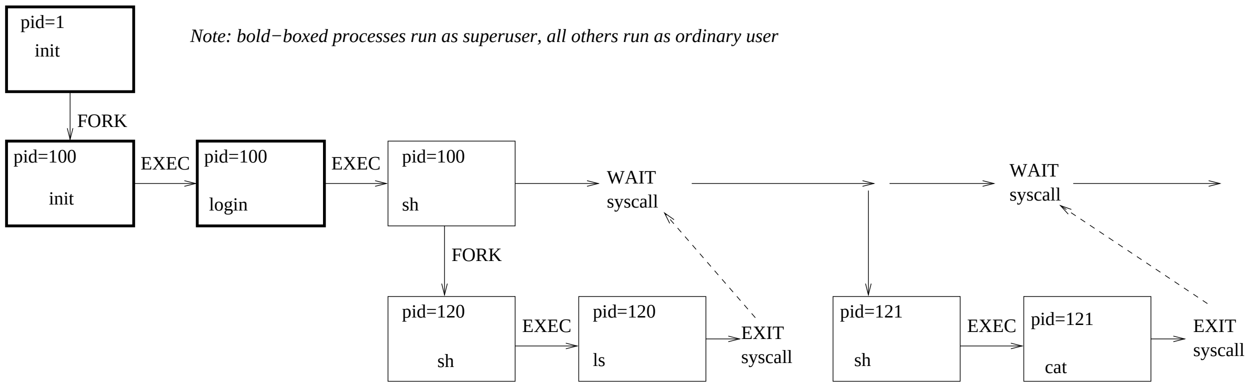
```
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

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);
          }
}
```

The `struct rusage` keeps track of resource consumption for a process. Among the stats 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`.

Note: typically the command line session starts via a much more convoluted process, e.g. through the GUI, or through the sshd daemon process

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 which then 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 online tutorials or the supplementary text on 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
• 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

Within kernel memory, there is a global variable called `current` which the kernel maintains as a pointer to the task_struct of the currently running process. (This is a simplification, and we'll have more to say in units 7 and 8)

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}

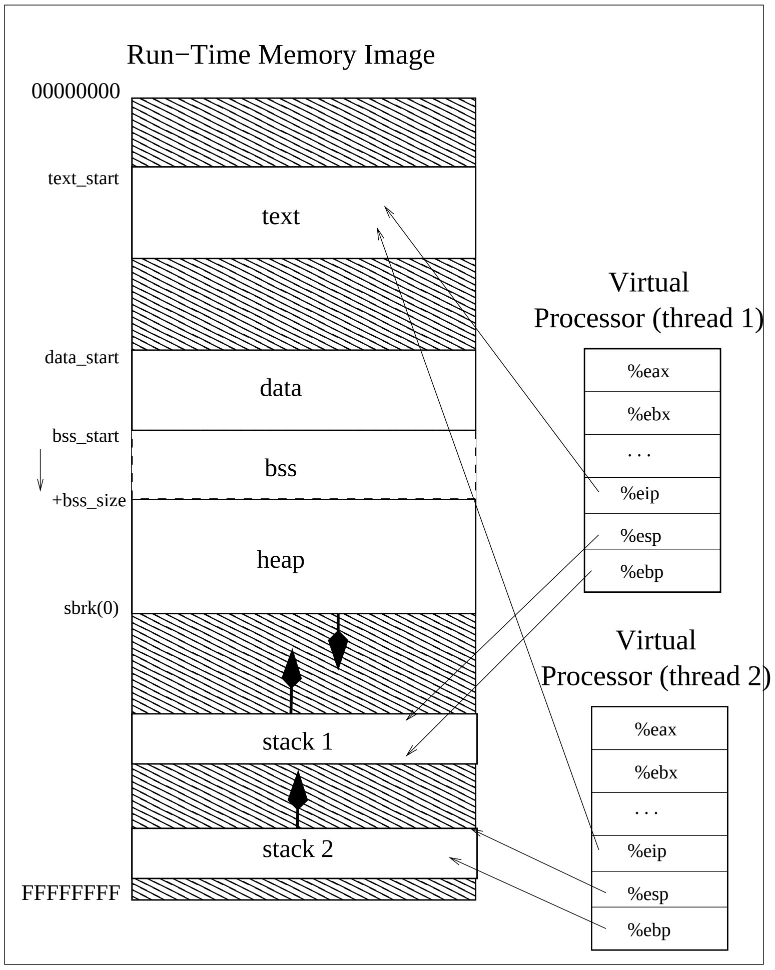
In a typical multi-threaded program, `clone` is called (via library wrapper functions) with all the `CLONE_XX` flags set, and therefore almost everything is shared between the parent and the new thread.

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" because any subsidary structures are also copied. If the flag is 1, the child simply points to the same sub-structure, which is a "shallow copy".

In a multi-threaded program, all threads exist in the same virtual address space. It is essential however that each thread have its own distinct stack region. Otherwise, function calls and local variables would interfere with each other! The `child_stack` parameter gives 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`.



The illustration above depicts a multi-threaded program in which there are two tasks (threads) executing in a single process. Note that each thread is its own "virtual processor" with independent registers, and that each thread has its own stack.

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`