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!

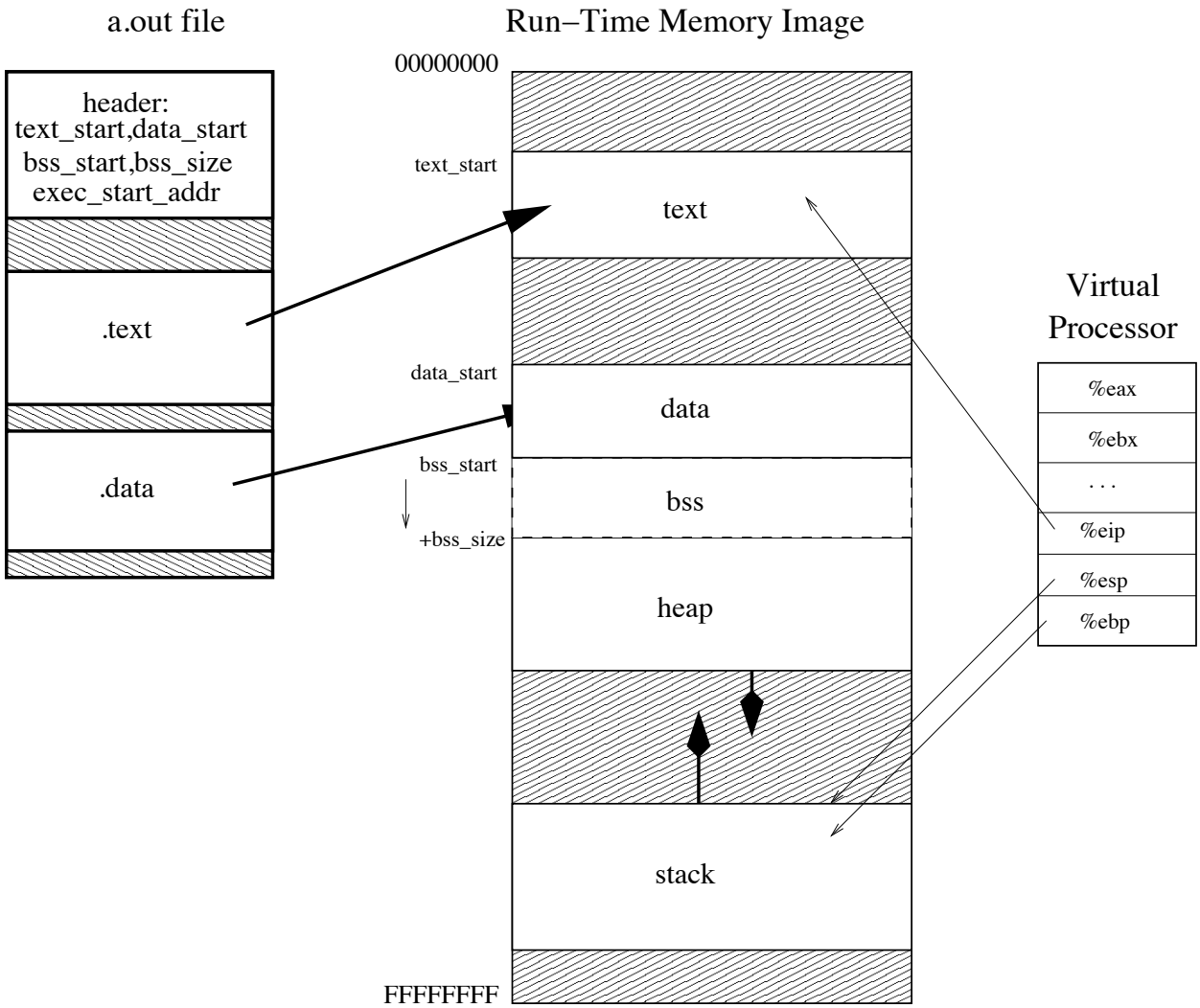
Processes are identified by an integer **Process ID (pid)**. All processes have a parent which caused their creation, and thus the collection of processes at any instant forms an ancestry tree. The pid of the current running process can be retrieved with the `getpid` system call, and the `getppid` system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

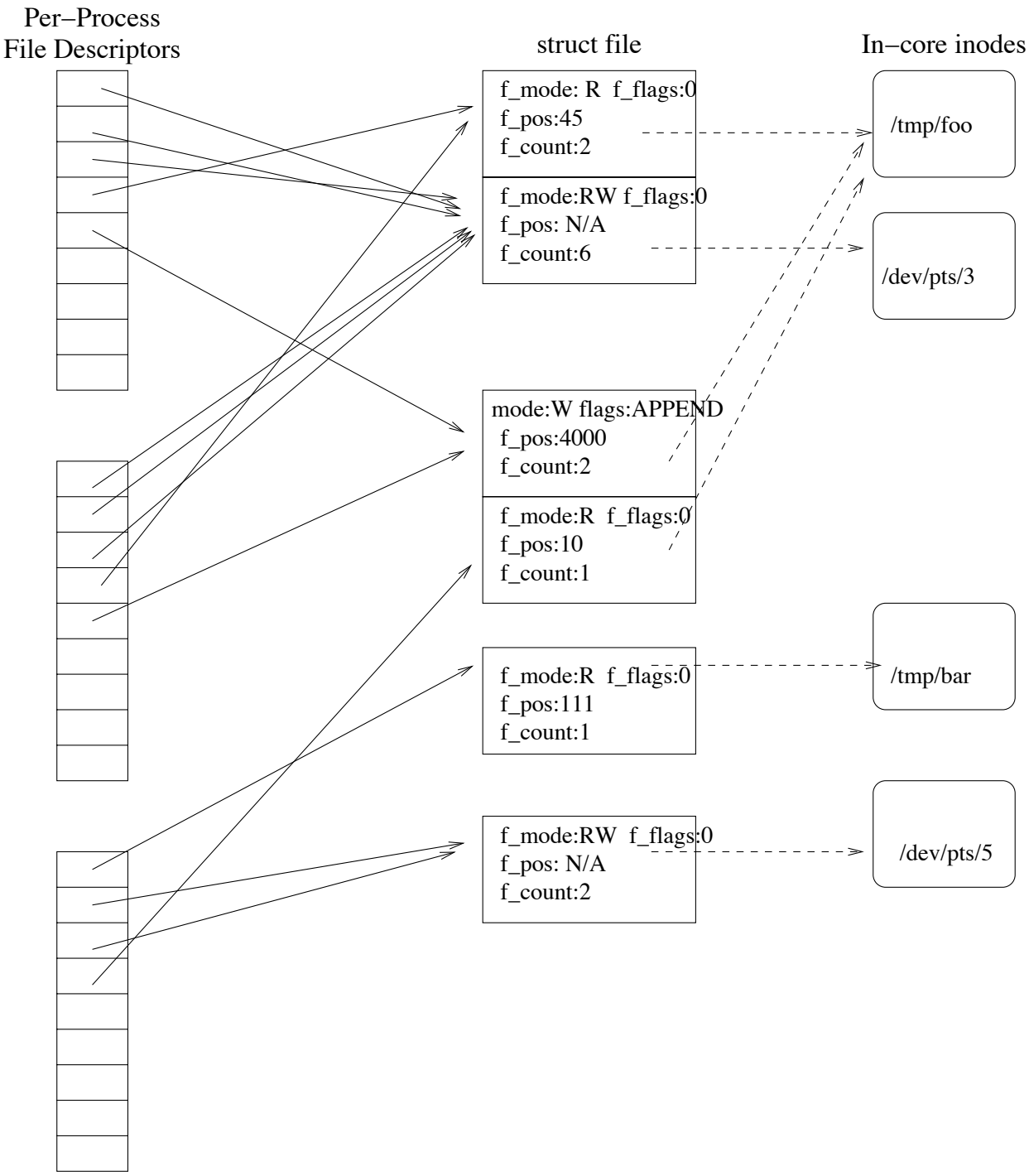
Although the child is an exact copy of the parent, it is nonetheless an independent entity

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.

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 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 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);
        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.

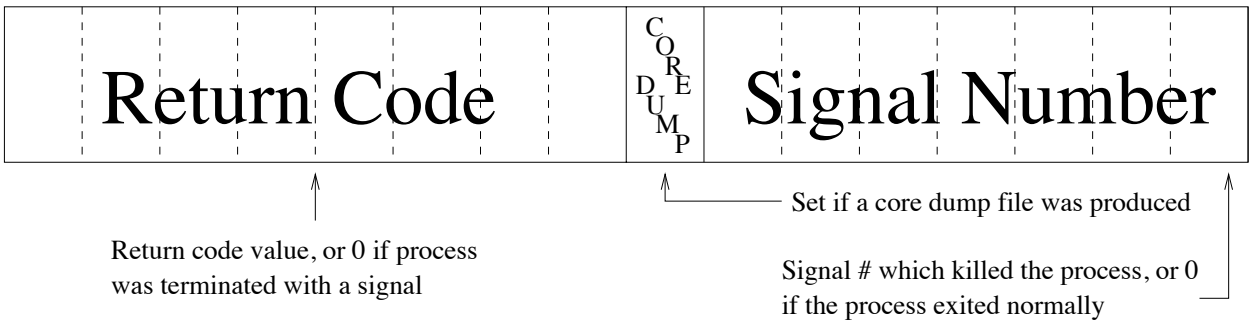
Regardless of the termination reason, when a process terminates, all file descriptors are closed by the kernel as if `close` had been called on them. All resources used by the process (such as memory) are freed (unless they are also being shared by other extant processes). Other state information (such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

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);
          }
}
```

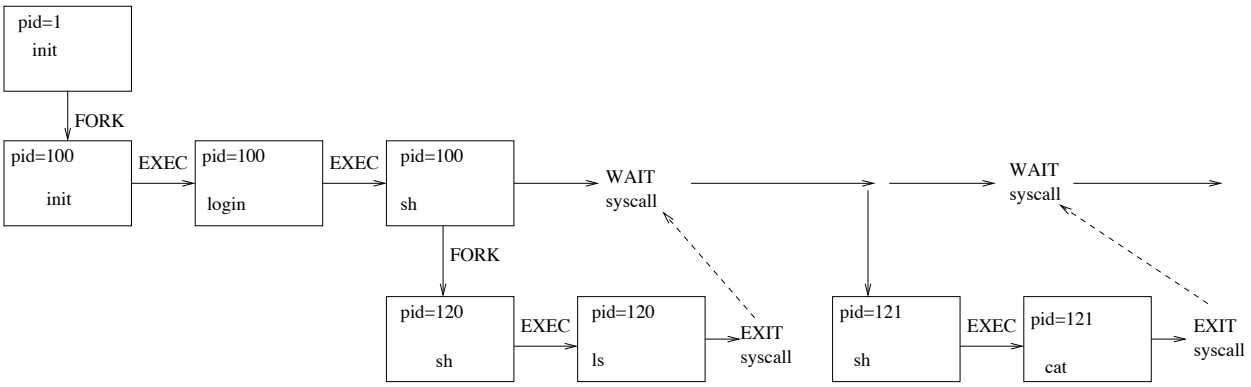
Among the resource usage information kept for each process is the total **user CPU time**

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.

Process State

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:

```
pid=current->pid;  //Get the PID of the current process ** See note below!
uid=current->credentials->uid;                  // Get the uid of the current process
cputime=current->utime;               // Get the accumulated user-mode CPU time
```

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.

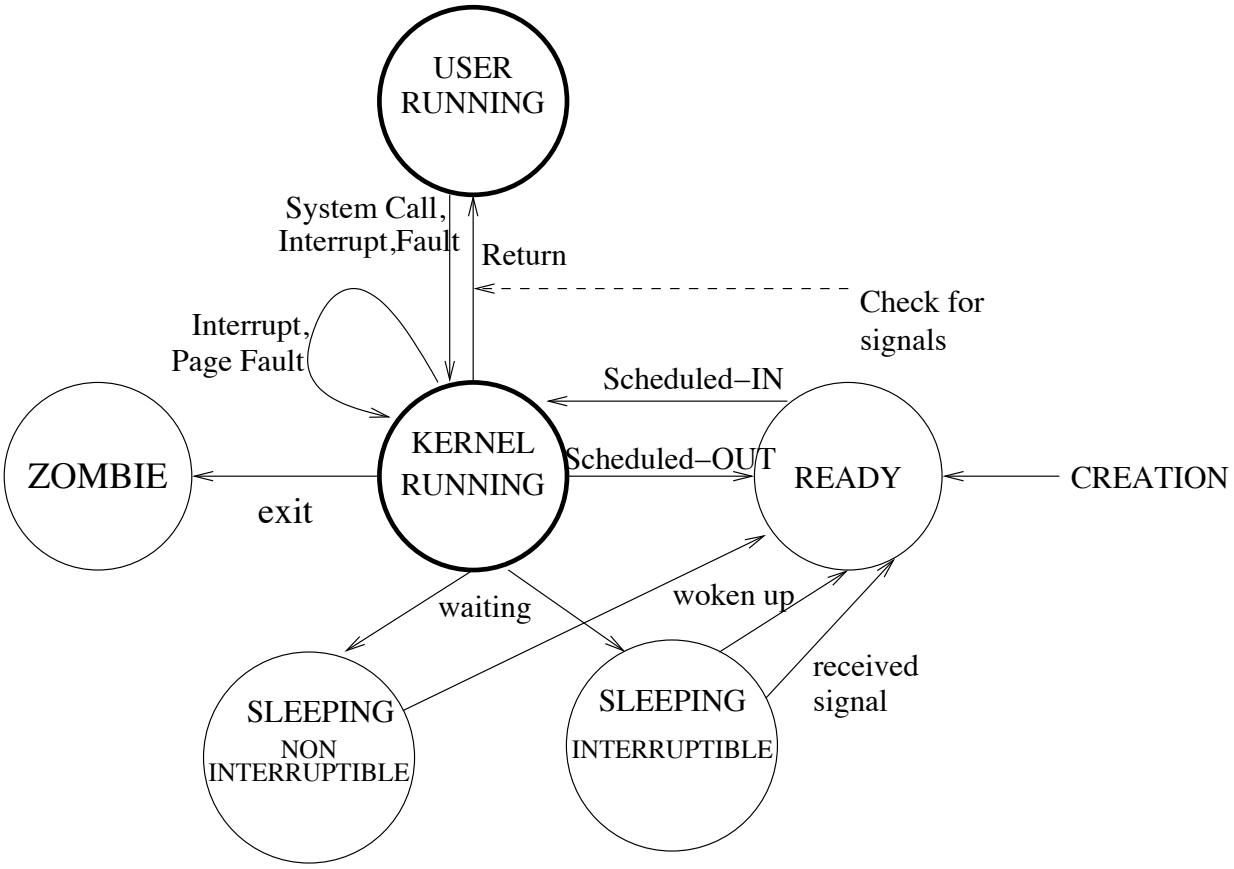
The trigger for preemption may be based on priority, i.e. a higher-priority task has just become ready to run. It may also be based on time-slicing, in which each task is given a certain amount of time and then another task is given the CPU. This requires a **Periodic Interval Timer** interrupt. Or, 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 static priority, and they may have an entirely different interpretation from the traditional nice values. E.g. some Linux kernels use a 0 to 99 scale, where 0 is the worst and 99 is the best. To maintain compatibility with POSIX standards, all kernels translate their internal priority number to the traditional -19..+20 nice value.

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 Linux Scheduling / CFS

The scheduling algorithm used for "normal" tasks by the Linux kernel (scheduling class `SCHED_NORMAL`) is known as the Completely Fair Scheduler (CFS). It aims to satisfy the principles set forth above. Because CPU speeds have increased greatly, but the tick frequency is still generally 1ms, we can afford a more complex algorithm that performs more calculations at every tick. 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 = 44.4%. 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, i.e. its dynamic priority. 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.

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 has the ability to designate certain tasks as having real-time priority. These tasks, if ready to run, will follow the rules above and will also pre-empt any general-purpose tasks. 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.