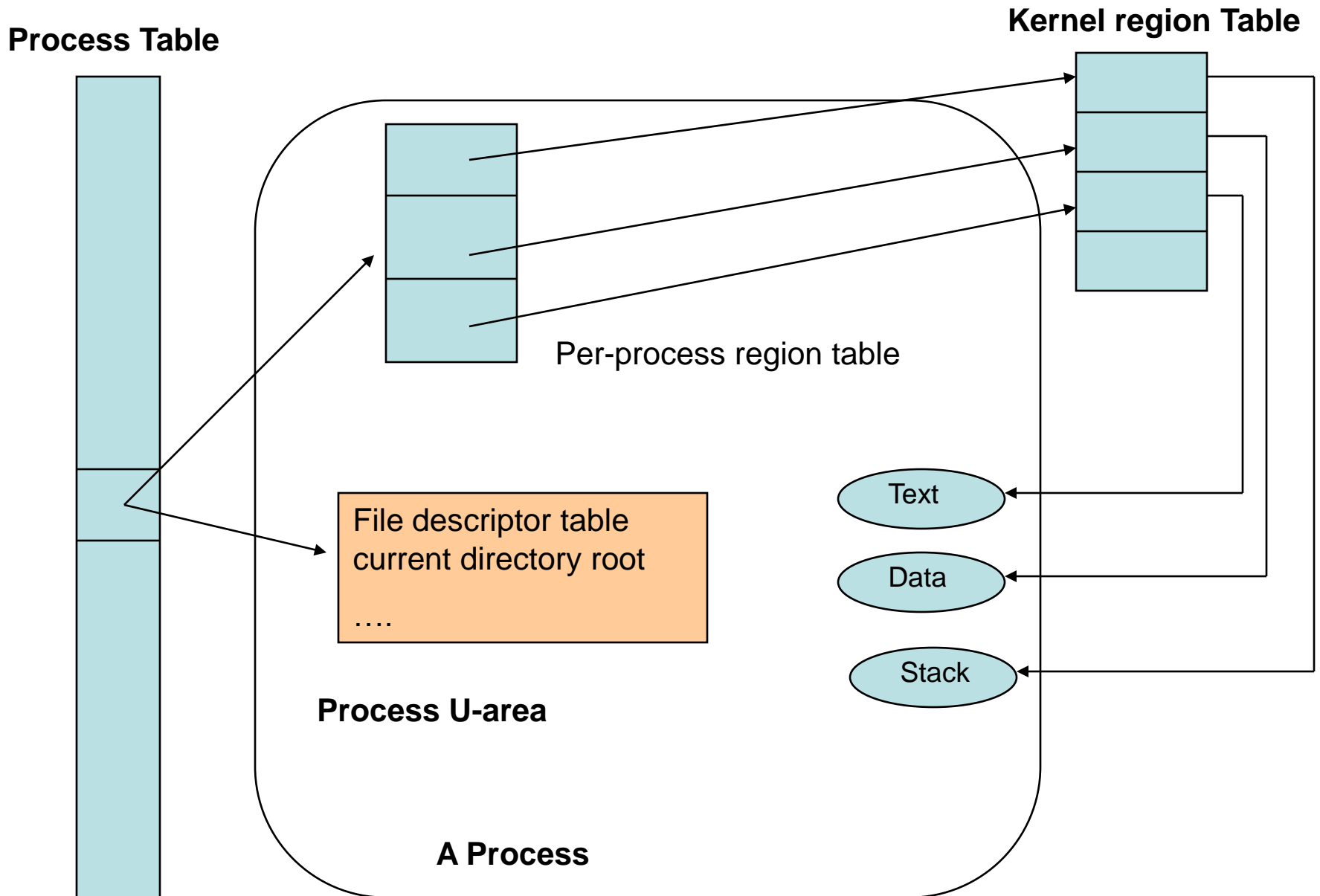


# UNIX Kernel Support for Processes

- **UNIX System V**
- UNIX process consists of
  1. Text segment
  2. Data segment
  3. Stack segment
- Segment is an area of memory that is managed by the system as a unit.
- **Process Table**
  - Keeps track of all active processes.
    1. System Processes
    2. User Processes
  - Contains pointers to segments and the U-area of a process.
- **U-area** is an extension of a Process table entry and contains other process-specific data
  1. File descriptor table
  2. current root
  3. working directory inode numbers
  4. Set of system imposed process resource limits

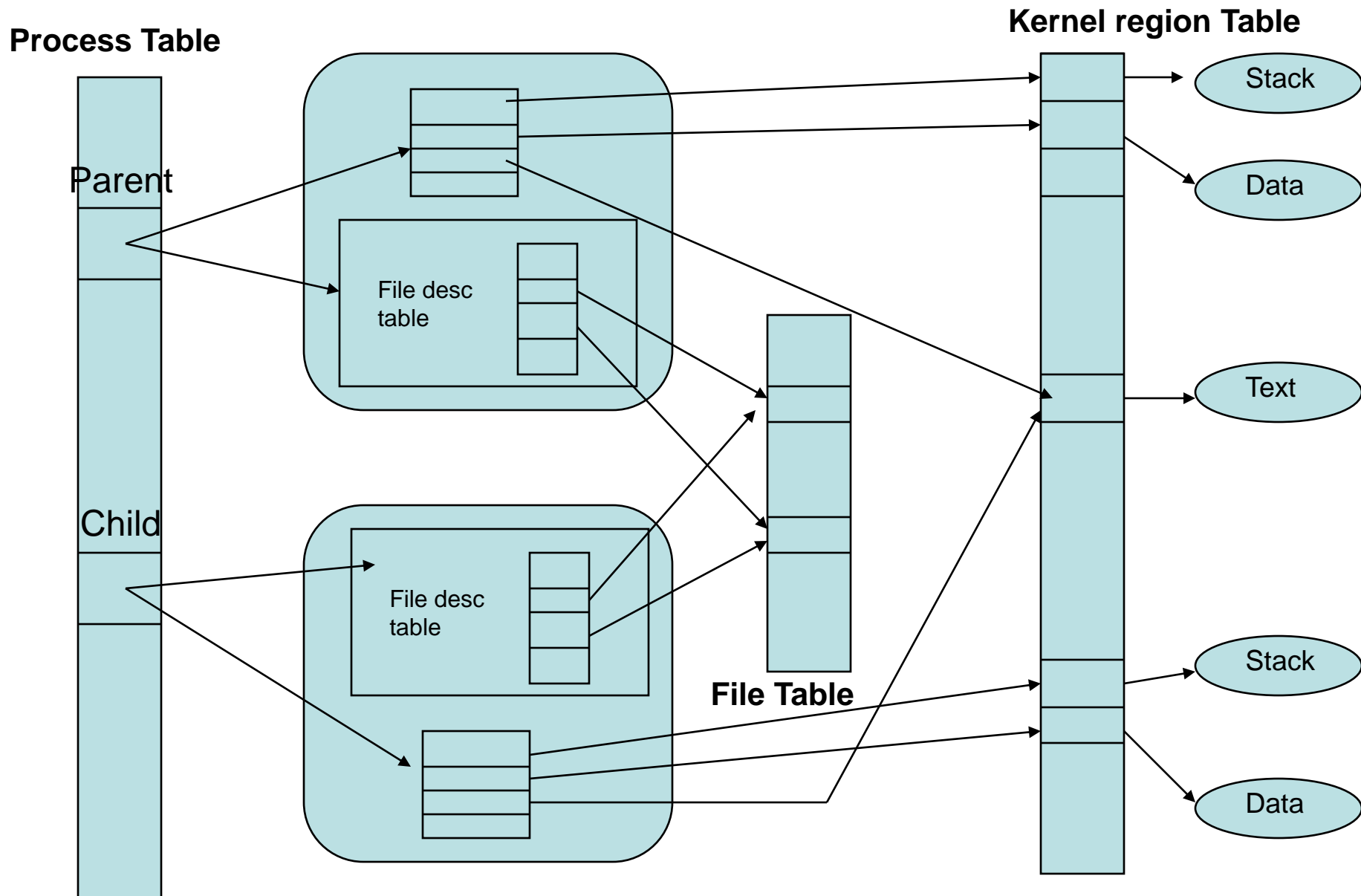
# A UNIX process data structure



# UNIX Kernel Support for Processes

- Very first process – 0 – created by the system boot code.
- All other processes – created via the **fork** system call.
- When a process is created by fork, it contains duplicated copies of the text, data and stack segments of the parent.
- It also has a File descriptor table such that both share the same file pointer.

# Data Structure of Parent and child after fork



# UNIX Kernel Support for Processes

**Attributes inherited from its parent or set by the kernel.**

1. Real User Identification ID (rUID)
2. Real group Identification number (rGID)
3. An effective user identification number (eUID)
4. An effective group Identification number (eGID)
5. Saved set-UID and set-GID
6. Process group identification number.
7. Supplementary group identification numbers
8. Current directory
9. Root directory
10. Signal Handling
11. Signal mask
12. Umask
13. Nice value
14. Controlling terminal

# UNIX Kernel Support for Processes

## Attributes different between the parent and the child process

- Process Identification number (PID)
- Parent process identification number (PPID)
- Pending signals
- Alarm clock time
- File locks

# UNIX Kernel Support for Processes

- `wait`, `waitpid` system calls – suspend till child finishes
- `Signal` or `sigaction` function to detect or ignore the child process termination.
- `_exit` system call
- `exec` system call – like changing jobs.
- `fork` and `exec` are used to spawn a sub process to execute a different program.
  - Multiple processes can execute multiple programs concurrently
  - Child executes in its own address space

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

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

```
pid_t wait(int * statloc)
```

```
pid_t waitpid(pid_t pid, int * statloc, int options)
```

- Child process - Zombie process
- Parent process – init process
- Waitpid:
  - If pid == -1
  - If pid > 0
  - If pid == 0
  - If pid < -1



```

int main (void)
{
    pid_t pid;
    if ( (pid = fork() < 0)
        err_sys("fork error");
    else if (pid == 0)                                /* first child */
    {
        if ( (pid = fork () < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit (0);                                /* parent from second fork == first child */

        /* We're the second child; our parent becomes init as soon as our real parent calls
        exit () in the statement above. Here's where we'd continue executing, knowing that
        when we're done, init will reap our status. */
        sleep(2);
        printf (" second child, parent pid = %d\n It, getppid () ) ;
        exit(0);
    }
    if (waitpid(pid, NULL, 0) != pid)                /* wait for first child */
        err_sys("waitpid error");

    /* We're the parent (the original process); we continue executing,
    knowing that we're not the parent of the second child. */
    exit (0) ;
}

```

# Wait3 and Wait4 functions

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

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

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

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

```
pid_t    wait3(int *statloc, int options, struct rusage *rusage);
```

```
pid_t    wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

Function	Pid	Options	Rusage	POSIX.1	SVR4	4.3+BSD
wait				●	●	●
waitpid	●	●		●	●	●
wait3		●	●		●	●
wait 4	●	●	●			●

# Race Conditions

- When multiple processes are trying to do something with shared data and final outcome depends on the order in which the processes are run.
- If a parent wants to wait for the child to terminate – call wait function
- If a child wants to wait for the parent to terminate –

*while (getppid() != 1)*

*sleep(1)*

```

#include "ourhdr.h"

TELL_WAIT () ;    /* set things up for TELL xxx & WAIT xxx */

if ( (pid = fork () < 0)
        err_sys("fork error");
else if (pid == 0) {          /* child */
        /* child does whatever is necessary... */
        TELL_PARENT(getppid());    /* tell parent we're done */
        WAIT_PARENT () ; /* and wait for parent */
        /* and the child continues on its way... */
        exit(0);
}

/* parent does whatever is necessary... */
TELL_CHILD (pid) ;          /* tell child we're done */
WAIT_CHILD () ;            /* and wait for child */

/* and the parent continues on its way... */
exit (0) ;

```

```

#include <sys/types. h>
#include "ourhdr.h"
static void charatotime (char *);

int main (void) {
    pid_t pid;
    if ( (pid = fork ()) < 0)
        err_sys ("fork error");
    else if (pid == 0) {
        charatotime ("output from child\n");
    } else {
        charatotime(."output from parent\n");
    }
    exit(0) ;
}

static void charatotime (char *str)
{
    char *ptr;
    int c;
    setbuf (stdout, NULL); /* set unbuffered */
    for (ptr = str; c = *ptr++; )
        putc(c, stdout);
}

```

# Output

\$ a.out

output from child

output from parent

\$ a.out

ooouttppuutt          ffrroomm cphairledn

t

\$ a.out

ooouttppuutt          ffrroomm pcahrieldt

\$ a.out

ooutput from parent

utput from child

```

int main (void)
{
    pid_t pid;
+    TELL_WAIT () ;
+
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {
+        WAIT_PARENT();      /* parent goes first */
        charatime("output from child\n");
    } else {
+        charatime("output from parent\n");
        TELL_CHILD(pid);
    }
    exit (0) ;
}

static void charatime(char *str) {

    char *ptr;
    int c;

    setbuf(stdout, NULL); /* set unbuffered */
    for (ptr = str; c = *ptr++;)
        putc(c, stdout);
}

```

# exec Functions

- Process is completely replaced.
- Process ID does not change
- Replaces the current program with a new program from the disk.

```
#include <unistd.h>
```

```
int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */);
```

```
int execv(const char *pathname, char *const argv[ ] );
```

```
int execlp(const char *pathname, const char *arg0, ... /* (char *) 0, char *const envp[ ] */);
```

```
int execve(const char *pathname, char *const argv[ ], char *const envp[ ] );
```

```
int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */);
```

```
int execvp(const char *filename, char *const argv[ ] );
```

All six return: -1 on error, no return on success



# Differences

Function	pathname	filename	arg list	argv[ ]	environ	emp[ ]
execl	●		●		●	
execlp		●	●		●	
execle	●		●			●
execv	●			●	●	
execvp		●		●	●	
execve	●			●		●
(Letter in name)		P	L	V		e

Limit on the total size of the argument list : ARG\_MAX

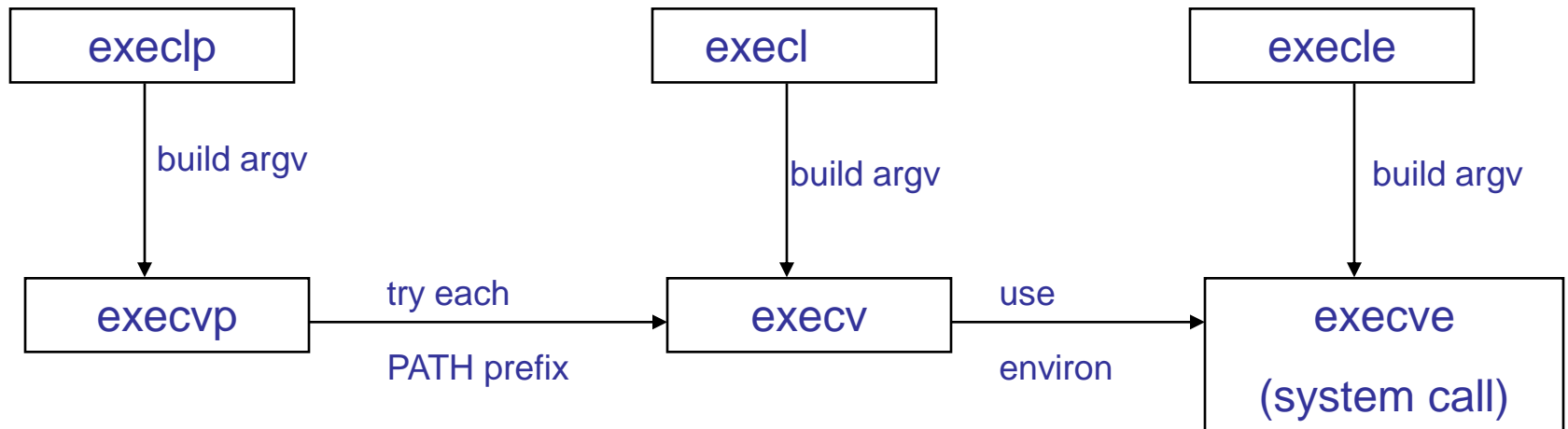
4096 on a POSIX.1 system.

Eg: `grep _POSIX_SOURCE /usr/include/*/*.h`

# Inherits

- Process ID and parent process ID
- real user Id and real group Id
- supplementary group Ids
- process group ID
- Session ID
- controlling terminal
- time left until alarm clock.
- current working directory
- root directory
- file mode creation mask
- file locks
- process signal mask.
- pending signals
- resource limits
- tms\_utime, tms\_stime, tms\_cutime, tms\_ustime

# Relationship



```

char *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
int main (void) {
    pid_t    pid;
    if ( (pid = fork () < 0)
        err_sys (" fork error");
    else if (pid == 0)
    {
        /* specify pathname, specify environment */
        if (execle (" /home/stevens/bin/echoall",
                    "echoall", "myarg1", "MY ARG2", (char *) 0,
                    env_init < 0)
            err_sys("execle error");
    }
    if (waitpid(pid, NULL, 0) < 0)
        err_sys ("wait error");
    if ( (pid = fork () < 0)
        err_sys ("fork error");
    else if (pid == 0) {
        /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *) 0) < 0)
            err_sys("execlp error");
    }
    exit (0) ;
}

```

```
#include "ourhdr.h"

int main(int argc, char *argv[]) {
    int i;
    char **ptr; extern char **environ;
    for (i = 0; i < argc; i++)          /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    for (ptr = environ; *ptr != 0; ptr++)
        printf("%s\n", *ptr);
    /* and all env strings */
    exit (0) ;
}
```

```
$ a.out
```

```
argv[0]: echoall
```

```
argv[1]: myarg1
```

```
argv[2]: MY ARG2
```

```
USER=unknown
```

```
PATH=/tmp
```

```
$ argv[0]: echoall
```

```
argv[1]: only 1 arg
```

```
USER=stevens
```

```
HOME=/home/stevens
```

```
LOGNAME=stevens
```

```
31 more lines.....
```

```
EDITOR=/usr/ucb/vi
```

```

int system(const char *cmdstring) {
    pid_t  pid;
    int    status;

    if (cmdstring == NULL)
        return(1);    /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {    /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);    /* execl error */
    } else {    /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}

```

Exit Code Number	Meaning	Example	Comments
1	catchall for general errors	let "var1 = 1/0"	miscellaneous errors, such as "divide by zero"
2	misuse of shell builtins, according to Bash documentation		Seldom seen, usually defaults to exit code 1
126	command invoked cannot execute		permission problem or command is not an executable
127	"command not found"		possible problem with \$PATH or a typo
128	invalid argument to <a href="#">exit</a>	exit 3.14159	<b>exit</b> takes only integer args in the range 0 - 255
128+n	fatal error signal "n"	<b>kill -9</b> \$PPIDof script	<b>\$?</b> returns 137 (128 + 9)
130	script terminated by Control-C		Control-C is fatal error signal 2, (130 = 128 + 2, see above)
255	exit status out of range	exit -1	<b>exit</b> takes only integer args in the range 0 - 255



# Process Times

```
#include <sys/times.h>
clock_t times (struct tms *buf);
```

```
Struct tms{
    clock_t tms_utime;           /*user CPU time */
    clock_t tms_stime;           /*system CPU time */
    clock_t tms_cutime ;         /*user CPU  time, terminated children */
    clock_t tms_cstime;          /* system CPU time, terminated children */
};
```

# Main program

```
#include <sys/times.h>
#include "ourhdr.h"

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

int main(int argc, char *argv[])
{
    int i;
    for (i = 1; i < argc; i++)
        do_cmd(argv[i]); /* once for each command-line arg */
    exit(0);
}
```

# Function to execute the command

```
static void do_cmd(char *cmd)      /* execute and time the "cmd" */
{
    struct tms tmsstart, tmsend;
    clock_t start, end;
    int status;

    fprintf(stderr, "\ncommand: %s\n", cmd);
    if ( (start = times (&tmsstart)) == -1)      /* starting values */
        err_sys("times error");

    if ( (status = system(cmd)) < 0)              /*execute command */
        err_sys ("systemO error");

    if ( (end = times(&tmsend)) == -1)            /* ending values */
        err_sys("times error");
    pr_times(end-start, &tmsstart, &tmsend);

    pr_exit(status);
}
```

# Function to calculate and print the time

```
static void pr_times (clock_t real, struct tms *tmsstart, struct tms *tmsend)
{
    static long clktck = 0;
    if (clktck == 0) /* fetch clock ticks per second first time */
    if ( (clktck = sysconf(_SC_CLK_TCK)) < 0)
        err_sys("sysconf error");
    fprintf(stderr, " real: %7.2f\n", real/ (double) clktck);
    fprintf(stderr, " user: %7.2f\n",
        (tmsend->tms_utime - tmsstart->tms_utime)/(double) clktck);
    fprintf(stderr, "sys:      %7.2f\n",
        (tmsend->tms_stime - tmsstart->tms_stime)/(double) clktck);
    fprintf(stderr, " child user: %7.2f\n",
        (tmsend->tms_cutime - tmsstart->tms_cutime)/(double) clktck);
    fprintf(stderr, " child sys: %7.2f\n",
        (tmsend->tms_cstime - tmsstart->tms_cstime) /(double) clktck);
}
```

# Output

```
$ a.out "sleep 5" "date"
```

```
command: sleep 5
```

```
real: 5.25
```

```
user: 0.00
```

```
sys: 0.00
```

```
child user:    0.02
```

```
child sys:     0.13
```

```
normal termination, exit status = a
```

```
command: date
```

```
Sun Aug 18 09:25:38 MST 1991
```

```
real: 0.27
```

```
user: 0.00
```

```
sys: 0.00
```

```
child user:    0.05
```

```
child sys:     0.10
```

```
normal termination, exit status = a
```

# Interpreter files

- Files that begin with  
    `#! pathname [optional-argument]`  
    *Eg: #! /bin/sh*
- The actual file that gets executed is the file specified by the *pathname*.
- *Pathname is usually absolute pathname.*
- *Some systems have a limit of 32 characters for the first line.*
- *Differentiate between*  
    *interpreter file: a text file that begins with a #!*  
    *interpreter: specified by pathname*

```
#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"

int main (void)
{
    pid_t pid;
    if ( (pid = fork 0 ) < 0)
        err_sys("fork error");
    else if (pid == 0) {          /* child */
        if (execl("/home/stevens/bin/testinterp",
            "testinterp", "myarg1", "MY ARG2", (char *) 0) < 0)
            err_sys("execl error");
        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
        exit(0);
    }
}
```

```
$ cat /home/stevens/bin/testinterp
#!/home/stevens/bin/echoarg foo
```

```
$ a.out
argv[0] : /home/stevens/bin/echoarg
argv[1] : foo
argv[2] : /home/stevens/bin/testinterp
argv[3] : myarg1
argv[4] : MY ARG2
```

# Example

- `awk -f myfile`

Interpreter file, `awkexample`:

```
#!/bin/awk -f
BEGIN (
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}
```

```
$ awkexample file1 FILENAME2 f3
ARGV[0] = /bin/awk
ARGV[1] = file1
ARGV[2] = FILENAME2
ARGV[3] = f3
```

When `/bin/awk` is executed, its command-line arguments are  
`/bin/awk -f /usr/local/bin/awkexample file1 FILENAME2 f3`



# Are interpreter files required?

1. They hide the fact that certain programs are scripts in some other language.

Eg: `awkexample optional-arguments`

Instead of

`awk -f awkexample optional-arguments`

2. They provide an efficiency gain.

```
awk 'BEGIN {  
    for (i = 0; i < ARGV; i++)  
        printf "ARGV[%d] = %s\n", i, ARGV[i]  
    exit  
' $*
```

3. They let us write shell scripts using shells other than `/bin/sh`.

Eg: `#!/bin/csh`

# Process Accounting

- When enabled, kernel writes an accounting record each time a process terminates.
- 32 bytes of binary data.

# Accounting Record

```
typedef u short comp_t; /* 3-bit base 8 exponent; 13-bit fraction */
```

```
struct acct{
```

```
    char ac_flag; /* flag */
```

```
    char ac_stat; /* termination status (signal & core flag only) */
```

```
        /*(not provided by BSD systems) */
```

```
    uid_t ac_uid; /* real user ID */
```

```
    gid_t ac_gid; /* real group ID */
```

```
    dev_t ac_tty; /* controlling terminal */
```

```
    time_t ac_btime; /* starting calendar time */
```

```
    comp_t ac_utime; /* user CPU time (clock ticks) */
```

```
    comp_t ac_stime; /* system CPU time (clock ticks) */
```

```
    comp_t ac_etime; /* elapsed time (clock ticks) */
```

```
    comp_t ac_mem; /* average memory usage */
```

```
    comp_t ac_io; /* bytes transferred (by read and write) */
```

```
    comp_t ac_rw; /* blocks read or written */
```

```
    char ac_comm[8]; /* command name: [8] for SVR4, [10] for 4.3+BSQ */
```

```
};
```

# ac\_Flag

- AFORK – process is the result of fork, but never called exec
- ASU – Process used super user privileges
- ACOMPAT – process used compatibility mode
- ACORE – process dumped core
- AXSIG – process was killed by a signal

```

int main (void)
{
    pid_t pid;
    if ( (pid = fork ()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(2);
        exit(2);
    }
    /* parent */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0)
    {
        sleep(4);
        abort ();
    }
    /* terminate with exit status 2 */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        execl (" /usr/bin/dd", "dd", "if=/boot", "of=/dev/null", NULL)
        exit(7);
    }
    /* first. child */

    if ( (pid = fork ()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(8);
        exit(0);
    }
    /* second child */

    sleep(6);
    kill(getpid(), SIGKILL);
    exit(6);
    /* shouldn't get here */
    /* third child */

    /* normal exit */

    /* fourth child */

    /* terminate with signal, no core dump */
    /* shouldn't get here */
}

```

# Procedure

1. Become a superuser and enable accounting, with the *accton* command.
2. Run the program
3. Become a superuser and turn accounting off.
4. Run the program to print the selected fields from the accounting file.

# Output

accton	e =	7,	chars =	64,	stat =	0:	S	
dd	e =	37,	chars =	221888,	stat =	0:		second child
a.out	e =	128,	chars =	0,	stat =	0:		parent
a.out	e =	274,	chars =	0,	stat =	134: F	D X	first child
a.out	e =	360,	chars =	0,	stat =	9: F	X	fourth child
a.out	e =	484,	chars =	0,	stat =	0: F		third child