UNIX PROCESSES

MAIN FUNCTION

- int main(int argc, char *argv[ ]);

argc – is the number of command line arguments

argv [ ] – is an array of pointers to the arguments

- A C program is started by a kernel, by calling Exec Function.
- A special start up routine is called before the main function is called
- This start up routine takes values from the kernel and sets things up so that the main function is called

Process termination

There are eight ways for a process to terminate.

- Normal termination
  - **Return from main**
  - **Calling exit**
  - **Calling _exit**
  - **Return of the last thread from its start routine**
  - **Calling pthread_exit from the last thread**
- **Abnormal termination**
  - Calling abort
  - Receipt of a signal
  - Response of the last thread to a cancellation request

exit and _exit functions

- _exit returns to kernel immediately
- exit performs certain cleanup processing and then returns to kernel
- PROTOTYPE

```
#include <stdlib.h>
 void _exit (int status)
 void exit (int status)
```

atexit function

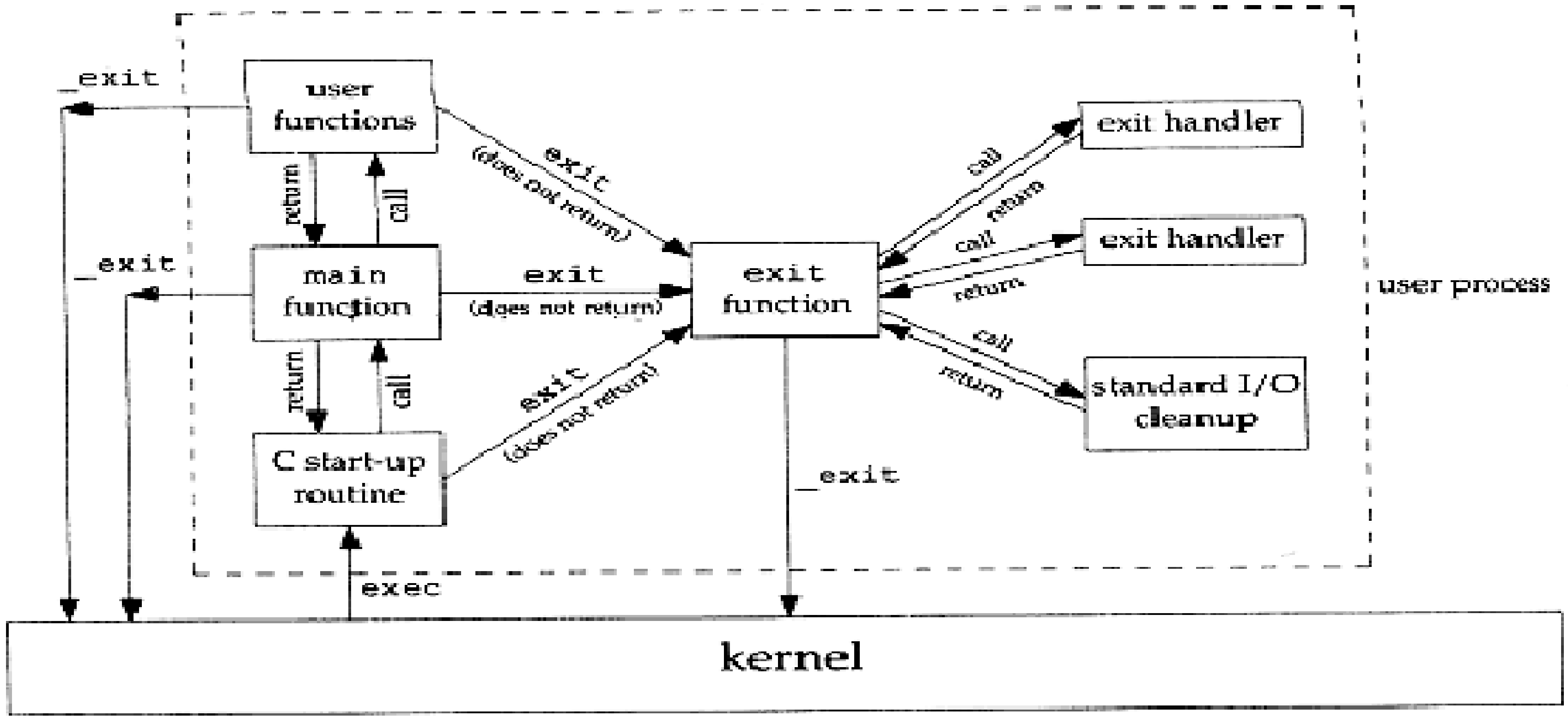
- With ANSI C a process can register up to 32 functions that are called by exit ---called exit handlers
- Exit handlers are registered by calling the atexit function

```
#include <stdlib.h>
int atexit (void (*fun) void));
```

- **Atexit function calls these functions in reverse order of their registration**
- **Each function is called as many times as it was registered**

```
void my_exit1(void);
void my_exit2(void);
 int main()
 {
   if (atexit(my_exit2) != 0)
        perror("can't register my_exit2");
   if (atexit(my_exit1) != 0)
        perror("can't register my_exit1");
   if (atexit(my_exit1) != 0)
        perror("can't register my_exit1");
   printf("main is done\n");
   return(0);
}
```

```
void  my_exit1(void)
{
   printf("first exit handler\n");
}
void my_exit2(void)
{
   printf("second exit handler\n");
}
```



How a C program is started and how it terminates.

Note that the only way a program is executed by the kernel is when one of the exec functions is called.

The only way a process voluntarily terminates is when _exit or _Exit is called, either explicitly or implicitly (by calling exit).

A process can also be involuntarily terminated by a signal (not shown in Figure).

Command-line arguments

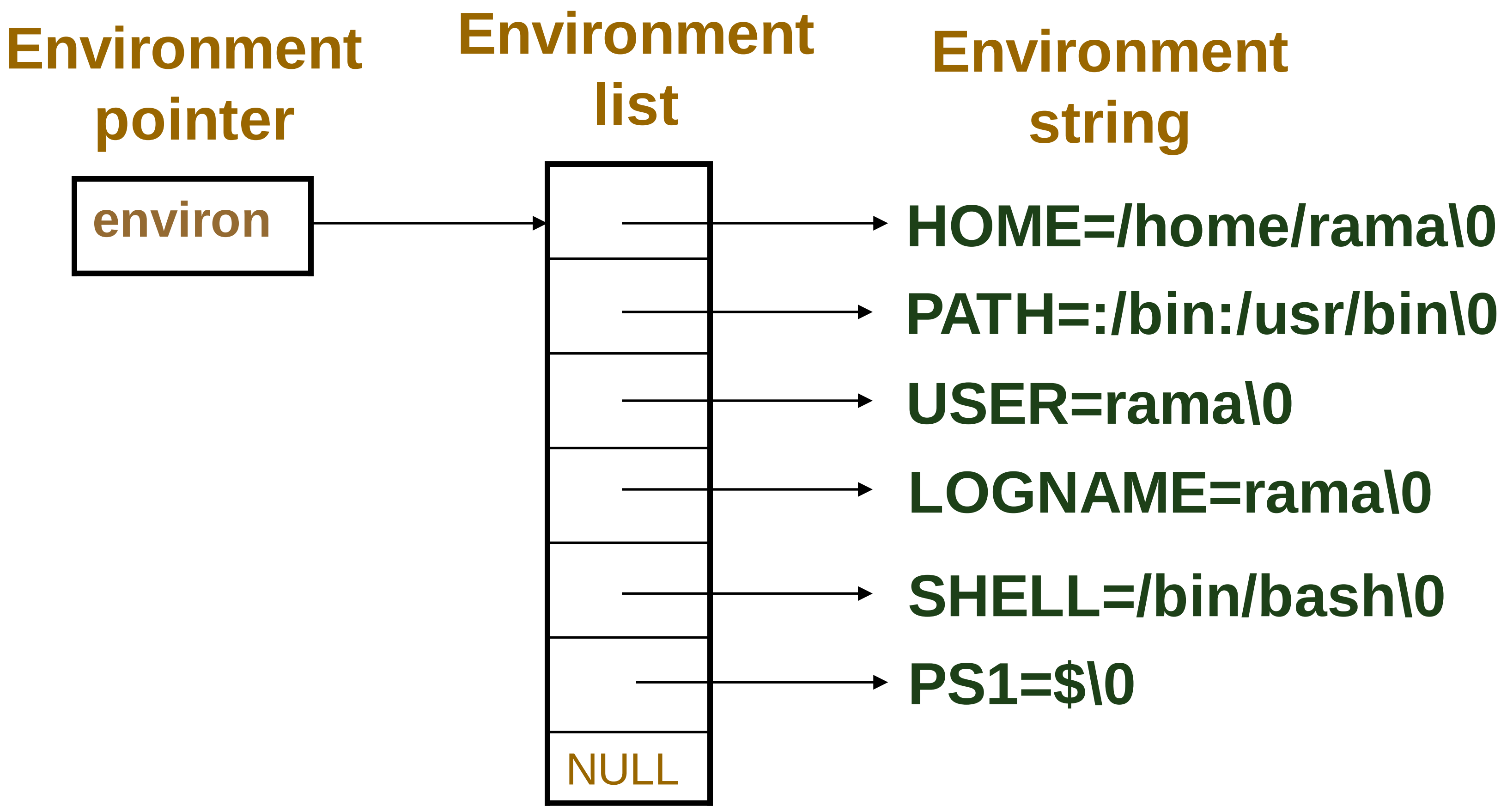
- /* program to echo command line arguments*/

```
int main (int argc, char* argv[ ])
 {
    for(int i=0;i<argc ;i++)
      {
         printf("argv[%d]:%s \n",i,argv[i]);
      }
 }
```

Environment list

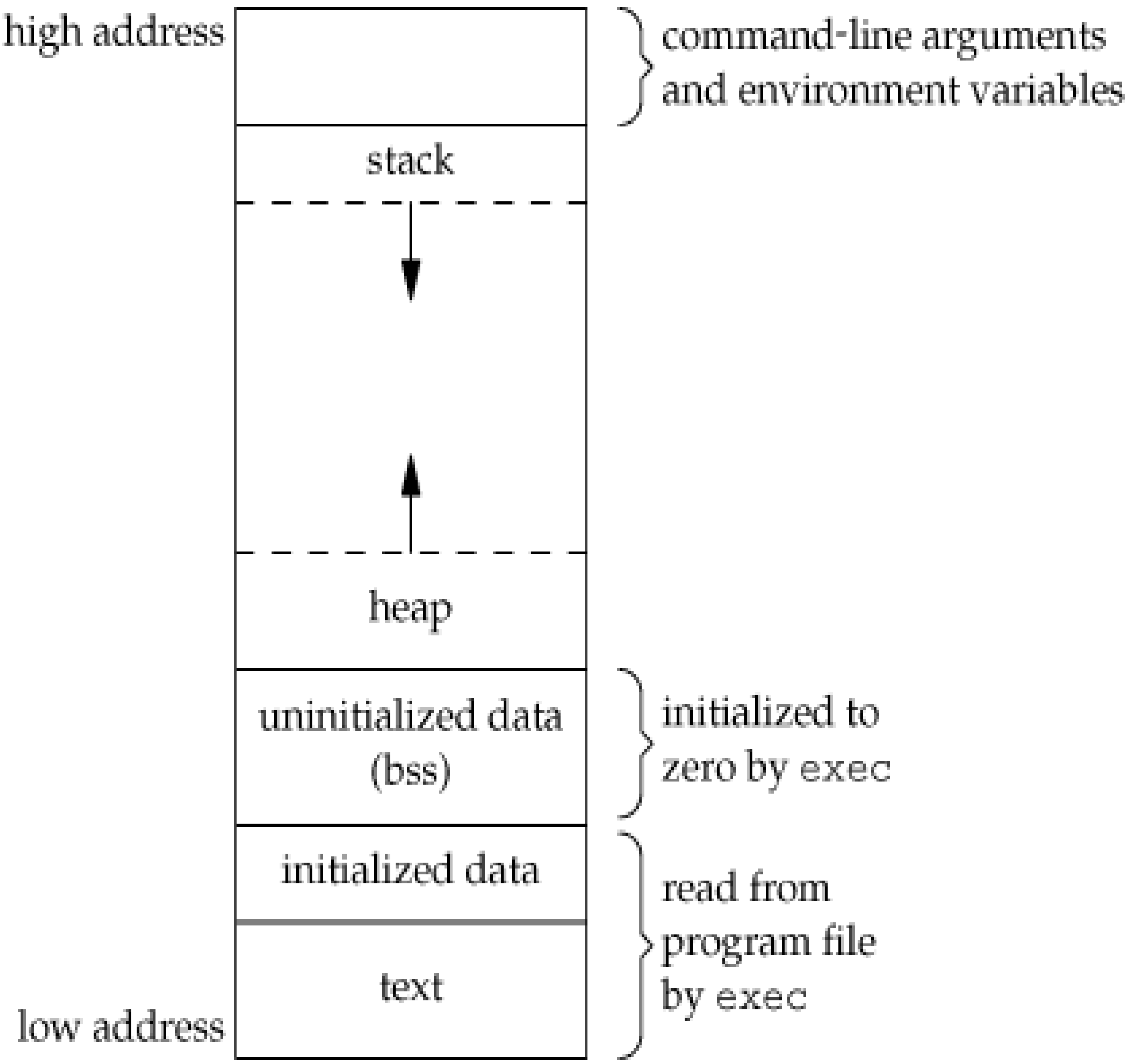
- **Environment list – is an array of character pointers, where each pointer contains the address of a null terminated C string.**
- **each string is of the form name=value;**
- **The address of array of pointers is contained in global variable environ.**

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


Environment pointer
Environment list
Environment string
environ
HOME=/home/rama\0
PATH=:/bin:/usr/bin\0
USER=rama\0
LOGNAME=rama\0
SHELL=/bin/bash\0
PS1=$\0
NULL

Memory layout of a C program

- **Text segment – sharable copy**
- **Initialized data segment – variables specifically initialized in the program**
- **Uninitialized data segment – "bss" segment. Data is initialized to arithmetic 0 or null**
- **Stack – return address and information about caller's environment**
- **Heap – dynamic memory allocation takes place on the heap**


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Memory allocation

- **malloc : allocates specified number of bytes of memory**
- **calloc : allocates specified number of objects of specified size**
- **realloc : changes size of previous allocated area**
- **free : Deallocates memory**

```
#include <stdlib.h>
void *malloc (size_t size);
void *calloc (size_t nobj, size_t size);
void *realloc (void *ptr, size_t newsize);
```

All three returns: Non-null pointer on success,
NULL on error

```
void free(void *ptr);
```

- realloc may increase or decrease the size of previously allocated area .
  If it decreases the size no problem occurs
  But if the size increases then:

1. Either there is enough space then the memory is reallocated and the same pointer is returned.
2. If there is no space then it allocates new area copies the contents of old area to new area frees the old area and returns pointer to the new area.

Alloca function

- **It is same as malloc but instead of allocating memory from heap, the memory allocated from the stack segment.**

- **The prototype for alloca is in stdlib.h.**

```
#include <stdlib.h>
void *alloca( size_t size);
```

Environment variables

- Are set automatically at login

- Environment strings are of the form

  name=value

```
#include <stdlib.h>
char  *getenv (const char *name);
int  putenv (const char *str);
int setenv (const char *name, const char *value,
                                          int rewrite);
void unsetenv (const char *name);
```

- getenv : Returns a value associated with name of the environment variable.
- putenv : takes a string of the form

  name=value , if it already exists then old value is removed.
- setenv : sets name to value.

  If name already exists then :
  - a) if rewrite is non-zero, then old definition is removed
  - b) if rewrite is zero old definition is retained
- unsetenv : removes any definition of name

setjmp and longjmp

- To transfer control from one function to another we make use of setjmp and longjmp functions.
- Non-Local goto statements.
- *setjmp* sets up for a nonlocal goto and *longjmp* performs a nonlocal goto.

```
#include <setjmp.h>
int setjmp (jmp_buf  buf);
void longjmp (jmp_buf  buf, int val);
```

- buf is of type jmp_buf , this data type is form of array that is capable of holding all information required to restore the status of the stack to the state when we call longjmp

- Val allows us to have more than one longjmp for one setjmp

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf       loc;
int fun()
{
      printf( "Enter fun, Now call longjmp....\n“);
      longjmp (loc, 5);
      printf( " End of Function ....\n“);
      return 0;
}
```

```
int main()
{
  int  res;
if ( (res=setjmp( loc )) != 0 )
  {
    printf(“After return from longjmp, res
                                        =  %d”,res);
    exit(0);
   }
  printf("Program continue after setting loc via
                                        setjmp...\n");
  fun();
  printf(“ End of Main\n";
  exit(0);
}
```

Output:

```
Program continue after setting loc via      setjmp...
Enter fun.. Now call longjmp....
After return from longjmp, res= 5
```

getrlimit and setrlimit

Every process has a set of resource limits, some of which can be:

1. Queried by the getrlimit()
2. Changed by setrlimit()

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

int getrlimit (int resource , struct   rlimit *rlptr);
int setrlimit (int resource , struct   rlimit *rlptr);
```

```
struct rlimit
 {
        rlim_t   rlim_cur;        /*soft limit*/
        rlim_t   rlim_max;       /*hard limit */
  }
```

1. Soft limit can be changed by any process to a value <= to its hard limit.
2. Any process can lower its hard limit to a value greater than or equal to its soft limit.
3. Only super user can raise hard limit.

- **RLIMIT_CORE – max size in bytes of a core file**
- **RLIMIT_CPU – max amount of CPU time in seconds**
- **RLIMIT_DATA – max size in bytes of data segment**
- **RLIMIT_FSIZE – max size in bytes of a file that can be created**
- **RLIMIT_NOFILE – max number of open files per process**
- **RLIMIT_NPROC – max number of child process**
- **RLIMIT_OFILE – same as RLIMIT_NOFILE**
- **RLIMIT_STACK – max size in bytes of the stack**

The program below prints out the current soft limit and hard limit for all the resource limits supported on the system.

```
#include  <sys/types.h>           #include  <sys/time.h>
#include  <sys/resource.h>        #include   <unistd.h>

#define   doit(name)        pr_limits(#name, name)
void pr_limits(char *name,  int resource)
{
        struct rlimit limit;
        if (getrlimit(resource, &limit)  == -1)
                perror("Error);
printf("% s  ", name);
        if (limit.rlim_cur == RLIM_INFINITY)
                printf("(infinite)  ");
        else
                printf("%ld  ", limit.rlim_cur);
        if (limit.rlim_max == RLIM_INFINITY)
                printf("(infinite)\n");
        else
                printf("%ld\n", limit.rlim_max);
}
```

```
int     main()
{
        doit(RLIMIT_DATA);
        doit(RLIMIT_NOFILE);
        exit(0);
}
```

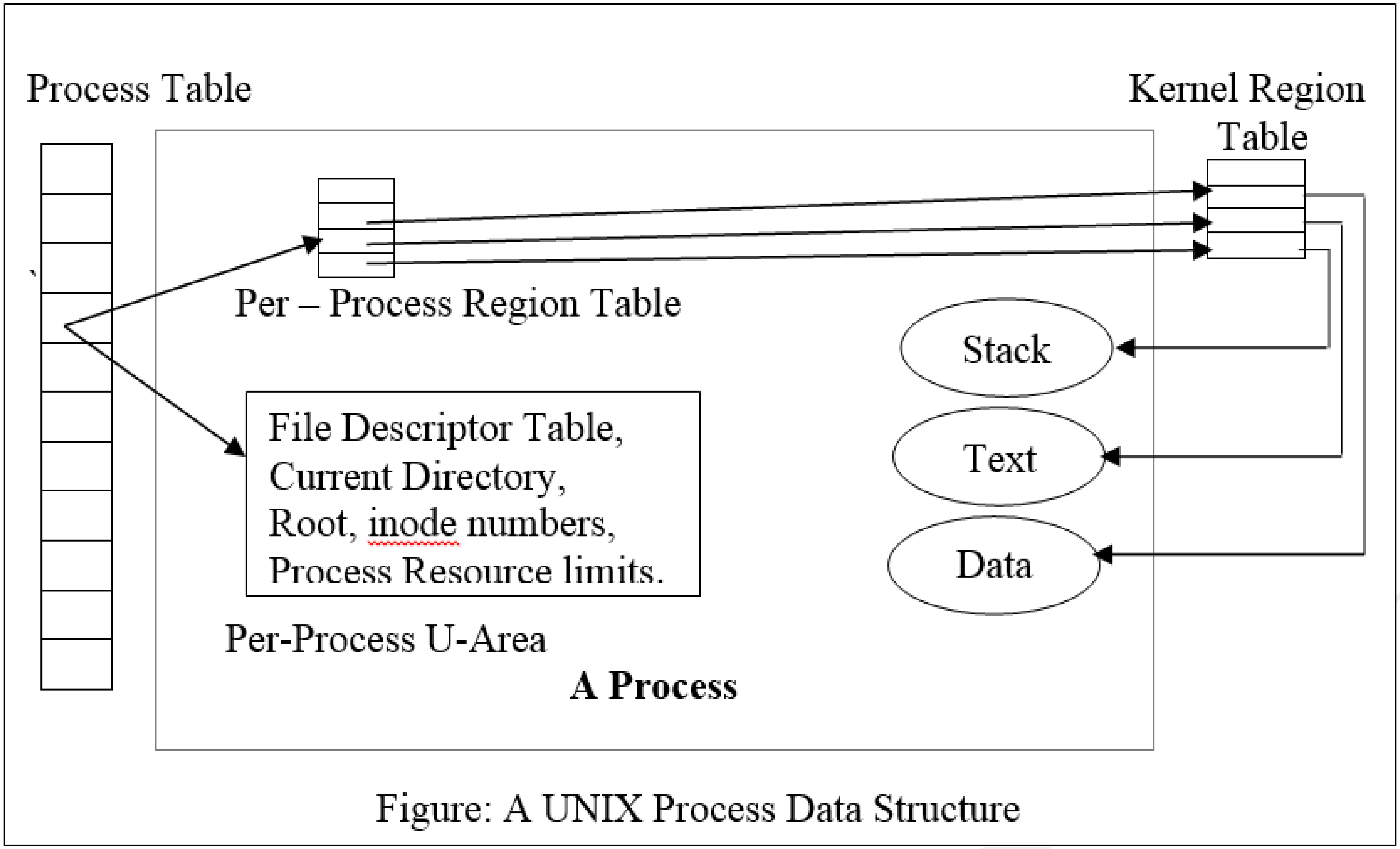
Output:

```
RLIMIT_DATA          (infinite)   (infinite)
RLIMIT_NOFILE           1024          1024
```

Process Control - Introduction

- A Process is a program under execution, For example a.out in UNIX or POSIX system.
- Processes are created with the fork system call. The child process created by fork is a copy of the original parent process, except that it has its own process ID.
- After forking a child process, both the parent and child processes continue to execute normally.

Kernel support for processes

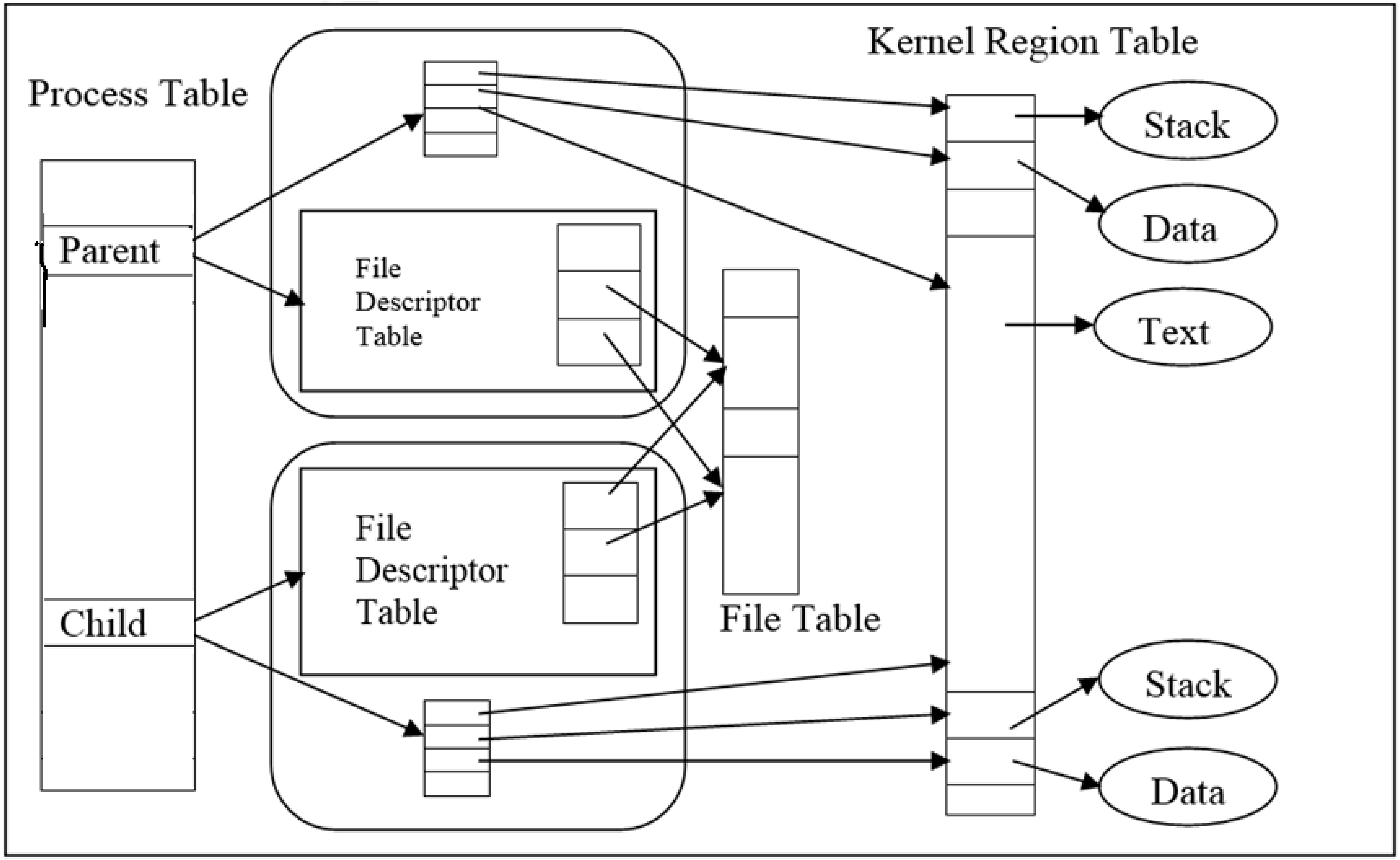


File Descriptor Table,
Current Directory,
Root, inode numbers,
Process Resource limits.

Figure: A UNIX Process Data Structure

- A process consists of
- **A text segment** – program text of a process in machine executable instruction code format
- **A data segment** – static and global variables in machine executable format
- **A stack segment** – function arguments, automatic variables and return addresses of all active functions of a process at any time
- **U-area** is an extension of Process table entry and contains process-specific data

After Forking – fork()



Besides open files the other properties inherited by child are

- Real user ID, group ID, effective user ID, effective group ID
- Supplementary group ID
- Process group ID
- Session ID
- Controlling terminal
- Saved set-user-ID and Saved set-group-ID
- Current working directory

- Root directory
- Signal handling
- Signal mask
- Umask
- Nice value
- The difference between the parent & child
- The process ID
- Parent process ID
- File locks
- Alarms clock time
- Pending signals

Process identifiers

- Every process has a unique process ID (PID)

  a non-negative integer.
- Special processes :

PID 0 :	Scheduler process also known as swapper.
PID 1:	init process, it never dies ,it’s a normal user process run with super user privilege.
PID 2:	Pagedaemon, responsible for supporting the paging of the virtual memory system.

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid (void);
pid_t getppid (void);
pid_t getpgrp(void);
uid_t getuid (void);
uid_t geteuid (void);
gid_t getgid (void);
gid_t getegid (void);
```

Program to display an attributes of a process

```
int  main ( )
{
        printf("\nProcess PID: %d ",getpid());
        printf("\nProcess PPID: %d",getppid());
        printf("\nProcess PGID: %d",getpgrp());
        printf("\nProcess real-UID: %d",getuid());
        printf("\nProcess real-GID: %d",getgid());
        printf("\nProcess effective-UID: %d",geteuid());
        printf("\nProcess effective-GID: %d", getegid());
        exit(0);
        }
```

Fork function

- **The only way a new process is created by UNIX kernel is when an existing process calls the fork function**

```
#include <sys/types.h>
#include <unistd.h>
 pid_t fork (void);
```

- The new process created by fork is called child process
- The function is called once but returns twice
- The return value in the child is 0
- The return value in parent is the process ID of the new child
- The child is a copy of parent
- Child gets a copy of parents text, data , heap and stack

The uses of fork

1. A process can duplicate itself so that parent and child can each execute different sections of code.
2. A process can execute a different program

- If process creation failed, fork returns -1.

The following reasons:

- EAGAIN : There aren't enough system resources to create another process, or too many processes running. This means exceeding the RLIMIT_NPROC resource limit.
- ENOMEM : The process requires more space than the system can supply.

vfork

- It is same as fork
- It is intended to create a new process when the purpose of new process is to exec a new program
- The child runs in the same address space as parent until it calls either exec or exit
- vfork guarantees that the child runs first ,

  until the child calls exec or exit

```
int     glob = 14;                              /* external variable in initialized data */

int main(void)
{
    int      var=15;                            /* automatic variable on the stack */
    pid_t     pid;
    if ((pid = fork()) ==-1)
      {
                        perror("fork");
              return 0;
      }
    if (pid == 0) {                              /* child */
                        glob+=2;                 /* modify variables */
                          var+=2;
      }
    if (pid > 0) {
                        sleep(2);               /* parent */
          }

     printf("\nglob= %d, var= %d\n", glob, var);
    return 0;
}
```

```
$ ./a.out

glob = 16, var = 17             child's variables were changed
glob = 14, var = 15             parent's copy was not changed
```

Zombie - Process

- A zombie process is a process that has terminated, but is still in the operating systems process table waiting for its parent process to retrieve its exit status.
- This is created when child terminates before the parent and parent would not able to fetch terminated status.
- The *ps* command prints the state of a zombie process as Z.

Write a program that creates a zombie and then call system to execute the PS Command to verify that the process is Zombie.

```
int main( )
{
            pid_t       pid;
            if ((pid = fork()) ==-1)
            {
                        perror("fork");   return 0;
            }
            if (pid == 0)           /* child */
                exit(0);
            if (pid > 0)            /* parent */
            {
                        sleep(4);
                        system("ps");
            }
            return(0);
}
```

Output:

```
  PID TTY          TIME     CMD
 3862 tty1     00:00:00 bash
 4122 tty1     00:00:00 a.out
 4123 tty1     00:00:00 a.out <defunct>
 4124 tty1     00:00:00 ps
```

Wait and waitpid functions

- **Used by parent process to wait for its child process to terminate or stop, and determine its status.**
- **These API's will deallocates process table slot of the child process.**

```
#include <sys/wait.h>
#include <sys/types.h>
pid_t wait (int *status)
pid_t waitpid (pid_t pid, int *status, int options);
```

The difference between above two functions are:

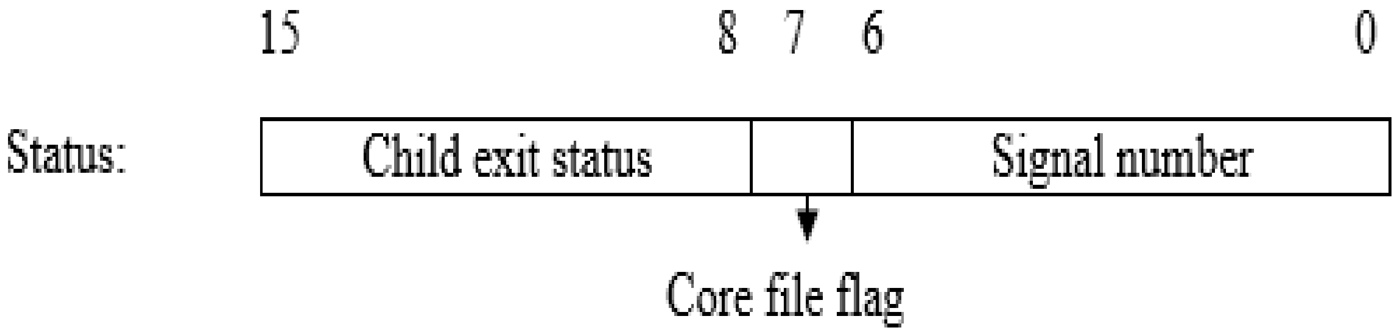
- Wait can block the parent process until a child process terminates, while waitpid as an option that prevents it from blocking.

- Waitpid doesnot wait for the first child to terminate - it has number of options that control which process it waits for.

- The interpretation of the *pid* argument for waitpid depends on its value:

PID Value	Meaning
pid ==-1	Waits for any child process., in this respect, waitpid equivalent to wait.
pid => 0	Waits for the child whose process ID equals to *pid*.
pid == 0	Waits for any child process in the same process group as the parent.
pid < -1	Waits for any child process whose process group ID is the absolute value of *pid*.

status



int WIFEXITED (int status) ; This macro returns a nonzero value if the child process terminated normally with exit or _exit.

int WEXITSTATUS (int status) : If WIFEXITED is true of status, this macro returns exit status value of the child process.

int WIFSIGNALED (int status) :This macro returns a nonzero value if the child process terminated because it received a signal that was not handled.

int **WTERMSIG** (*int status*)	:	If `WIFSIGNALED` is true of *status*, this macro returns the signal number of the signal that terminates the child process.
int **WCOREDUMP** (*int status*)	:	This macro returns a nonzero value if the child process terminated and produced a core dump.
int **WIFSTOPPED** (*int status*)	:	This macro returns a nonzero value if the child process is stopped.
int **WSTOPSIG** (*int status*)	:	If `WIFSTOPPED` is true of *status*, this macro returns the signal number of the signal that caused the child process to stop.

- The *options* argument is a bit mask. Its value should be the bitwise OR (that is, the `|' operator) of zero or more of the followings:

Constant	**Description**
WNOHANG	Indicate that the parent process shouldn't wait, if specified *pid* is not available immediately.
WUNTRACED	To request status information from stopped processes as well as processes that have terminated.

- The return value is normally the process ID of the child process whose status is reported. Or A value of -1 is returned in case of error. An error values may be:

- EINTR : The function was interrupted by a signal.

- ECHILD:There are no child processes to wait for, or the specified pid is not a child of the calling process.

```
#include <stdio.h>     #include <stdlib.h>     #include <unistd.h>
#include <sys/wait.h>
int   main(void)
{
        pid_t     pid;        int status;
        if ((pid = fork()) ==-1) {        perror("fork");              return 0;
        }
         if (pid == 0)
                exit(0);
        if (pid  > 0)
        {
                wait(&status);
                system("ps");
        }
  return(0);
}
```

Output:

```
 PID TTY          TIME CMD
3861 tty1     00:00:00 su
3862 tty1     00:00:00 bash
4146 tty1     00:00:00 a.out
4148 tty1     00:00:00 ps
```

Race condition

- **Race condition occurs when multiple process are trying to do something with shared data and final out come depends on the order in which the processes run.**

- If a parent process wants child to terminate, it must call one of the wait functions.
- Suppose if a child process wants to wait for its parent to terminate, The following form could be used:

```
while(getppid() !=1)
    sleep(1);
```

- this type of loop is called **polling**, it wastes CPU time, since the caller is woken up every second to test the condition.

Program with race condition

- We can avoid race condition and polling by:
  - Signals
  - Various forms of inter process Communication mechanisms
  - TELL_WAIT, TELL_PARENT, TELL_CHILD, WAIT_PARENT and WAIT_CHILD macros

Scenario of using above said macros is as follows:

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

  int main( )
        {
                  TELL_WAIT(); //set things up for TELL_xxx and WAIT_xxx

                          if ((pid = fork()) ==-1)
                       {
                                perror("fork");
                                return 0;
                       }
                  if (pid == 0)              //child
                  {
                             // Child does what ever is necessary
                             TELL_PARENT(getppid( ));  //tell parent we are done
                             WAIT_PARENT();              // wait for parent
                            // Child continuous on its way……..
                            exit(0);
                  }
                          // Parent does what ever is necessary
                          TELL_CHILD(pid);  //tell child we are done
                          WAIT_CHILD( ); // and wait for child
                          // and the Parent continuous on its way……..
                          exit(0);
       }
```

Program to demonstrate Race condition

```
void  fun(char  *str)
{
        int i=0;
        while(str[i] != '\0' )
        {
                putc(str[i], stdout);
                i++;
        }
}
int     main(void)
{
        pid_t   pid;
             if ((pid = fork()) ==-1)
                     {
                                perror("fork");
                                return 0;
                     }
        if (pid == 0)
                fun("output from child\n");
        if (pid > 0)
                fun("output from parent\n");
        exit(0);
}
```

Forms of output may be:

1. When child terminates before parent

    output from child
    output from parent

2. When parent terminates before child

    output from parent
    output from child

3. When both are executing simultaneously

    output output from child from parent   etc..

Modified form of above program to avoid race condition – Parent goes first

```
void  fun(char  *str)
{
        int i=0;
        while(str[i] != ’\0’ )
        {
                putc(str[i], stdout);
                i++;
        }
}
```

```
int main(void)
{
        pid_t   pid;

        TELL_WAIT();
        if ((pid = fork()) ==-1)
                {
                        perror("fork");
                        return 0;
                }
         if (pid == 0)
        {
                WAIT_PARENT();          /* parent goes first */
                fun("output from child\n");
        }
        if (pid >0)
        {
                fun("output from parent\n");
                TELL_CHILD(pid);
        }
        exit(0);
}
```

exec functions

- Exec replaces the calling process by a new program
- The new program has same process ID as the calling process
- No new program is created , exec just replaces the current process by a new program

There are six different versions of exec functions these are listed below:

1. int **execl** (char *path, char *arg0, ..., NULL);
2. int **execlp** (char *file, char *arg0, ..., NULL);
3. int **execle** (char *path, char *arg0, ..., NULL, char * envp[]);

4. int **execv** (char *path, char *argv[]);
5. int **execvp** (char *file, char *argv[]);
6. int **execve** (char *path, char *argv[], char *envp[]);

path	:	Path name of the called child process
argN	:	Argument pointer(s) passed as separate arguments
argv[N]	:	Argument pointer(s) passed as an array of pointers
env	:	Array of character pointers.

Relation between exec functions

$$\begin{tikzcd}
\text{execlp} \arrow[d, "\text{Build argv}"] & \text{execl} \arrow[d, "\text{Build argv}"] & \text{execle} \arrow[d, "\text{Build argv}"] \\
\text{execvp} \arrow[r, "\text{Try each}", "\text{PATH prefix}"'] & \text{execv} \arrow[r, "\text{use}", "\text{environ}"'] & \text{execve}
\end{tikzcd}$$

system function

- It helps us execute a command string within a program
- System is implemented by calling fork, exec and waidpid

```
#include <stdlib.h>
int system (const char  *cmdstring);
```

- **Return values of system function**

- **-1 – if either fork fails or waitpid returns an error other than EINTR**
- **127 -- If exec fails [as if shell has executed exit]**
- **termination status of shell -- if all three functions succeed**

```
#include  <sys/types.h>
#include  <sys/wait.h>
#include  <errno.h>
#include  <unistd.h>

int system(const char *cmdstring)
   /* version without signal handling */
{
   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);
}
```

```
/* calling system function*/
#include   <sys/types.h>
#include   <sys/wait.h>
#include   "ourhdr.h"
int main(void)
{
   int          status;

   if ( (status = system("date")) < 0)
        err_sys("system() error");
   pr_exit(status);
if ( (status = system("nosuchcommand")) < 0)
        err_sys("system() error");
   pr_exit(status);

   if ( (status = system("who; exit 44")) < 0)
        err_sys("system() error");
   pr_exit(status);

   exit(0);
}
```