Introduction

Before looking at the process control primitives, we need to examine the environment of a single process.
We'll see how the main function is called when the program is executed.
How command-line arguments are passed to the new program.
What the typical memory layout looks like, how to allocate additional memory, how the process can use environment variables, and various ways for the process to terminate.
Additionally, we'll look at the longjmp and setjmp functions and their interaction with the stack.
Lastly we see the resource limits of a process.

main Function

A C program starts execution with a function called main.

The prototype for the main function is

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

Where :→argc is the number of command-line arguments,
→argv is an array of pointers to the arguments.

When a C program is executed by the kernel—by one of the exec functions a special start-up routine is called before the main function is called.
The executable program file specifies this routine as the starting address for the program; this is set up by the link editor when it is invoked by the C compiler.
This start-up routine takes values from the kernel—the command-line arguments and the environment—and sets things up so that the main function is called.

Process Termination

There are eight ways for a process to terminate. Normal termination occurs in five ways:

1. Return from main
2. Calling exit
3. Calling _exit or _Exit
4. Return of the last thread from its start routine
5. Calling pthread_exit from the last thread

Abnormal termination occurs in three ways:

1. Calling abort
2. Receipt of a signal
3. Response of the last thread to a cancellation request

The start-up routine is also written so that if the main function returns, the exit function is called. If the start-up routine were coded in C (it is often coded in assembler) the call to main could look like

exit(main(argc, argv));

Exit Functions

These functions terminates a program normally: _exit which return to the kernel immediately, and exit, which performs certain cleanup processing and then returns to the kernel.

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

#include <unistd.h>
void _exit(int status);
```

The reason for the different headers is that exit is specified by ISO C, whereas _exit is specified by POSIX.1.
Historically, the exit function has always performed a clean shutdown of the standard I/O library: the fclose function is called for all open streams. This causes all buffered output data to be flushed (written to the file) and executes exit handlers specified using atexit( ).

atexit Function

With ISO C, a process can register up to 32 functions that are automatically called by exit.
These are called exit handlers and are registered by calling the atexit function.

```
#include <stdlib.h>
int atexit(void (*func)(void));      Returns: 0 if OK, nonzero on error
```

This declaration says that we pass the address of a function as the argument to atexit. When this function is called, it is not passed any arguments and is not expected to return a value. The exit function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.

Calling abort function

You can abort your program using the `abort` function. The prototype for this function is in stdlib.h.

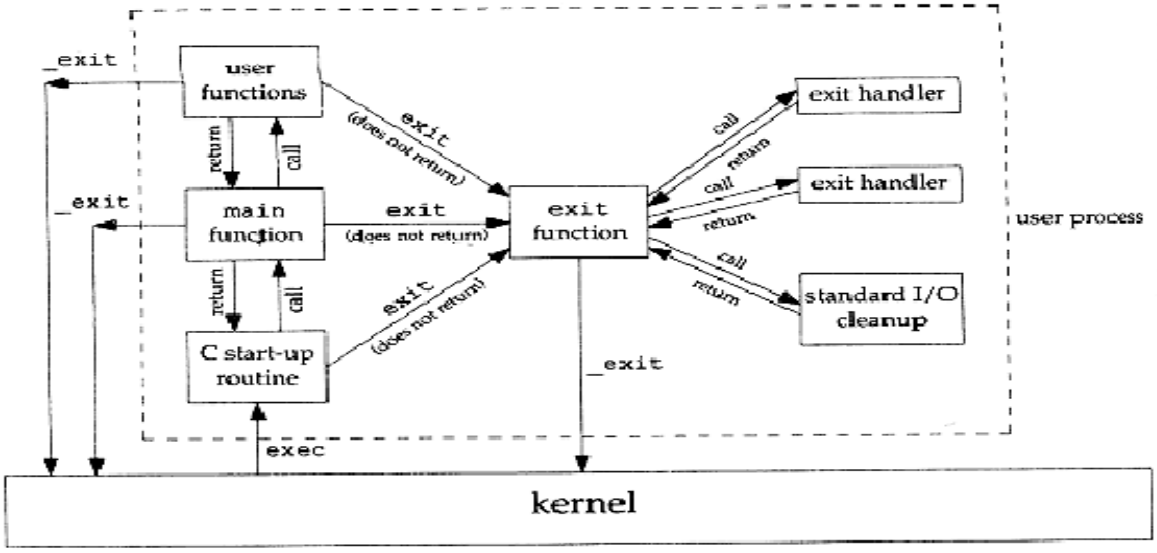
#include <stdlib.h>
void **abort** (*void*);

The `abort` function causes abnormal program termination. This does not execute cleanup functions registered with `atexit`. This function actually terminates the process by raising a `SIGABRT` signal, and your program can include a handler to intercept this signal.

Terminated by a signal

A *signal* is a software interrupt delivered to a process. Signals are generated by the following events. A program error such as dividing by zero, issuing an address outside the valid range and by pressing Ctrl ^z, or terminate it with Ctrl ^c.

Figure below summarizes how a C program is started and the various ways it can terminate.



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 is called, either explicitly or implicitly (by calling exit). A process can also be involuntarily terminated by a signal (not shown in Figure).

Example

The program in below demonstrates the use of the atexit function.

```
#include <stdio.h>
#include <stdlib.h>

void my_exit1(void)  {   printf("first exit handler\n");  }
void my_exit2(void)  {   printf("second exit handler\n"); }
int main(void)
{
    if (atexit(my_exit2) != 0)
            perror("atexit");
    if (atexit(my_exit1) != 0)
             perror("atexit");
    if (atexit(my_exit1) != 0 )
            perror("atexit");
    printf("main is done\n");
```

```
    return(0);
}
```

Executing this program it yields

```
$ ./a.out
main is done
first exit handler
first exit handle
second exit handler
```

An exit handler is called once for each time it is registered. In the program above, the first exit handler is registered twice, so it is called two times. Note that we don't call exit; instead, we return from main.

Command-Line Arguments

When a program is executed, the process that does the exec can pass command-line arguments to the new program.

This is part of the normal operation of the UNIX system shells.

Example

The program below echoes all its command-line arguments to standard output.

```
#include <stdio.h>
#include <stdlib.h>
 int main(int argc, char *argv[])
{
        int  i;
        for( i=0 ; i<argc ; i++)
                printf("argv[%d] : %s \n", i , argv[i] );
        exit(0);
}
```

Input: $./a.out yog msr it

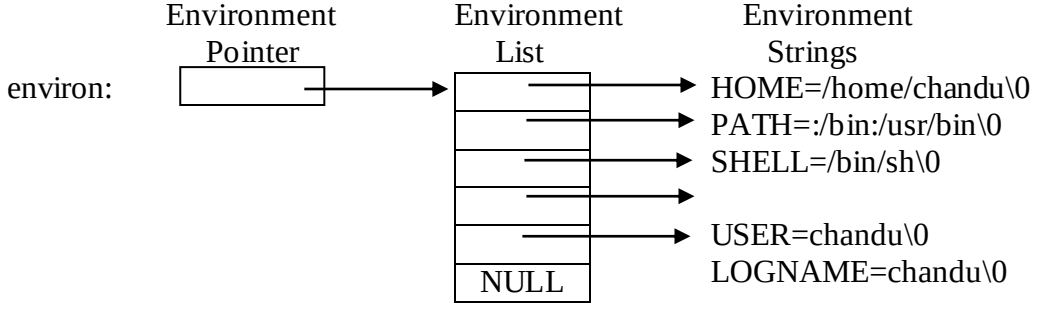
Output:
argv[0]: ./a.out
argv[1]: yog
argv[2]: msr
argv[3]: it

Environment List

Each program is also passed an environment list. Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string.
The address of the array of pointers is contained in the global variable environ:

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

For example, if the environment consisted of five strings, it could look like as shown in figure below.



By convention, the environment consists of name=value strings, as shown in figure above. Most predefined names are entirely uppercase, but this is only a convention.
Historically, most UNIX systems have provided a third argument to the main function that is the address of the environment list:

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

Memory Layout of a C Program

The C program normally composed of following segments. Segment is a portioned area of a memory.

- Text Segment: This segment contains the instructions that are executed by the CPU. This is the shared and read only segment, to prevent a program from accidentally modifying its instructions.

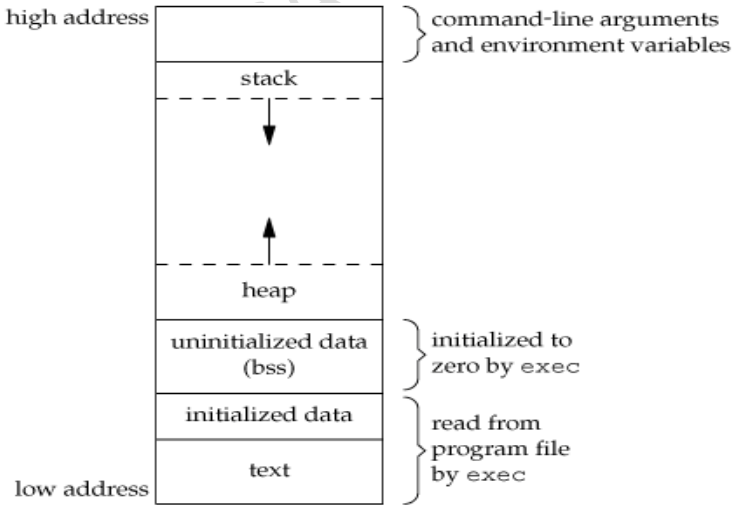
- Data Segment: Classified into two segments,

    - Initialized data segment: As the name itself represents it contains the variables with initialized data.
      For example: int size=100;
    - Un Initialized data segment: This segment can also be called as "bss (block started by symbol)" segment. Variables in this segment is initialized by the kernel to arithmetic 0 or NULL pointer before the program starts executing.
      For example: int size;

- Stack: Automatic variables are stored in this segment, along with information that is saved each time a function is called (such as the address of where to return to, and certain information about the caller's environment is saved on the stack). Hence stack is a storage for function arguments, automatic variables and return address of all active functions for a process at any time.

- Heap: Dynamic memory allocation takes place in this section only. Usually this segment located in between Uninitialized data and stack segment.

The figure below shows the typical arrangement of these segments.



The **size** command reports the sizes (in bytes) of the text, data, and bss segments. For example:

```
$ size /usr/bin/cc /bin/sh
   text    data     bss     dec     hex filename
  79606    1536     916   82058    1408a /usr/bin/cc
 619234   21120   18260  658614    a0cb6 /bin/sh
```

The fourth and fifth columns are the total of the three sizes, displayed in decimal and hexadecimal, respectively.

Shared Libraries

Most UNIX systems today support shared libraries. Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference.

This reduces the size of each executable file but may add some runtime overhead, either when the program is first executed or the first time each shared library function is called.

Another advantage of shared libraries is that library functions can be replaced with new versions without having to relink edit every program that uses the library. (This assumes that the number and type of arguments haven't changed.)

Different systems provide different ways for a program to say that it wants to use or not use the shared libraries.

Options for the **cc** and **ld** commands are typical. As an example of the size differences, the following executable file—the classic hello.c program—was first created without shared libraries:

```
$ cc -static hello1.c          prevent gcc from using shared libraries
$ size a.out
   text    data     bss     dec     hex filename
 375657    3780    3220  382657   5d6c1 a.out
```

If we compile this program to use shared libraries, the text and data sizes of the executable file are greatly decreased:

```
$ cc hello1.c                  gcc defaults to use shared libraries
$ size a.out
   text    data     bss     dec     hex filename
    872     256       4    1132     46c a.out
```

Memory Allocation

```
#include <stdlib.h>
#include <malloc.h>
void *malloc (size_t size);
void *calloc (size_t count, size_t size);
void *realloc (void *addr, size_t size);
void  free (void *addr);
```

malloc : Allocate a block of *size* bytes and returns a pointer to the first byte. The initial value of the memory is indeterminate.

calloc : Allocate a block of *count* * *size* bytes using `malloc`, and set its contents to zero. Here *count* is the number of blocks and *size* is the size (in bytes)of each block, and returns pointer to the first byte.

realloc : This function changes the size of a block of memory tht was previously allocated by `malloc` or by calloc to larger or smaller, possibly by copying it to a new location.

Here the *addr* is the pointer to the original block of memory. The new size in bytes specified by the size. There are several possible outcomes with realloc():

- If sufficient space exists to expand the memory pointed to by *addr*, the additional memory is allocated and the function returns *addr*.
- If sufficient space does not exists to expand the current block in its current location , a new block of the size for size is allocated., and existing data is copied from the old block to new block to the beginning of the new block. The old block is freed, and the function returns a pointer *addr* to the new block.
- If the *addr* argument is NULL, the function acts like malloc(), allocating a block of size bytes and returning a pointer to it.
- If the argument size is 0, the memory that *addr* points to is freed, and the function returns NULL.
- If the memory is insufficient for the reallocation(either expanding the old block or allocating a new one), the function returns NULL, and the original block is unchanged.

Free : Free a block previously allocated by malloc

All three return: non-null pointer if OK, NULL on error

alloca function

The function `alloca` supports a kind of half-dynamic allocation in which blocks are allocated dynamically but freed automatically.

It is same as malloc, but it allocates memory from stack region instead of heap region.

The prototype for `alloca` is in `stdlib.h`.

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

The return value of `alloca` is the address of a block of *size* bytes of memory, allocated in the stack frame of the calling function.

Advantages of `alloca`

Here are the reasons why `alloca` may be preferable to `malloc`:

- Using `alloca` wastes very little space and is very fast.
- Deallocates automatically without using free.

Disadvantages of `alloca`

- These are the disadvantages of `alloca` in comparison with `malloc`:
  1. If you try to allocate more memory than the machine can provide, you don't get a clear error message.
  2. Only few system support `alloca`, so it is less portable.

Environment Variables

1. The environment strings are usually of the form: ***name=value***
2. ISO C defines a function that we can use to fetch values from the environment, but this standard says that the contents of the environment are implementation defined.

```
#include <stdlib.h>
char *getenv(const char *name);
```

   Returns: pointer to value associated with name, NULL if not found
3. Table below lists the environment variables defined by the Single UNIX Specification.

Variable	Description
HOME	home directory
LC_ALL	name of locale
LC_CTYPE	name of locale for character classification
LC_MONETARY	name of locale for monetary editing
LC_NUMERIC	name of locale for numeric editing
LC_TIME	name of locale for date/time formatting
LOGNAME	login name
PATH	list of path prefixes to search for executable file
PWD	absolute pathname of current working directory
SHELL	name of user's preferred shell
TERM	terminal type
TMPDIR	pathname of directory for creating temporary files
TZ	time zone information

In addition to fetching the value of an environment variable, sometimes we may want to change the value of an existing variable or add a new variable to the environment list, Unix provides following functions:

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

All return: 0 if OK, nonzero on error

The operation of these three functions is as follows.

a. The putenv function takes a string of the form name=value and places it in the environment list. If name already exists, its old definition is first removed.
b. The setenv function sets name to value. If name already exists in the environment, then
   (a) if rewrite is nonzero, the existing definition for name is first removed;
   (b) if rewrite is 0, an existing definition for name is not removed, name is not set to the new value, and no error occurs.
c. The unsetenv function removes any definition of name. It is not an error if such a definition does not exist.

C program to prompt the user for the name of an environment variable and print its value if it is defined and a suitable message otherwise; and to repeat the process if user wants it.

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

int  main()
{
      int  choice;
         char envname[512];
      do
      {
            printf("Enter environment variable name::");
            scanf("%s",envname);
            printf("\n %s =  %s \n",envname, getenv(envname));
                  printf("\nenter 1 to continue and 0 to exit:");
            scanf("%d",&choice);
      }     while (choice != 0);
         return 0;
}
```



```
[yogish@cseise]$gcc 7b.c
[yogish@cseise]$ ./a.out
Enter environment variable name::PWD

PWD =  /root/6usp/6ise
enter 1 to continue and 0 to exit:1
Enter environment variable name::HOME

HOME =  /home/yogish
enter 1 to continue and 0 to exit:1
Enter environment variable name::LOGNAME

LOGNAME =  yogish
enter 1 to continue and 0 to exit:1
Enter environment variable name::asd
asd =  (null)

enter 1 to continue and 0 to exit:0
[yogish@cseise]$
```

setjmp and longjmp Functions

C goto statement only allow process to transfer process flow from one statement to another within the same function hence they called local goto statements. But using above said functions we can goto from one function to another,. Hence they are the non local goto statements. *setjmp* sets up for a nonlocal goto and *longjmp* performs a nonlocal goto.
Difference between local and nonlocal goto:

Local goto in C:

```
foo
{
        label:
                …
                …
                …
                goto label;
}
```

Non-locol goto in ANSI C:

```
foo1( )
{
        …
        setjmp( );  //Label
        …
}

foo2( )
{
        …
        longjmp( );  //goto
        …
}
```

setjmp and longjmp declared in the <setjmp.h>header file.

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

NOTE: setjmp must be called before longjmp. And program may consists of any number of longjmp but only one setjmp.

The setjmp function records a location where the future goto will (via the longjmp call) return.

The *jmp_buf* is defined in the <setjmp.h> header, and the *jmpb* argument records the locations in a program where the future longjmp call can return, each location must be recorded in a jmp_buf – typed variable and is set by a a *setjmp* call. setjmp returns 0 when it is initially called directlyin a process.

The longjmp function is called to transfer a program flow to a location that was stored in the *jmp_buf* setjmp is useful for dealing with errors and exceptions encountered in a low-level subroutine of a program.

Normally, setjmp and longjmp save and restore all the registers needed for coroutines, but the overlay manager needs to keep track of stack contents and assumes there is only one stack.

Return Value: setjmp returns 0 when it is initially called. If the return is from a call to longjmp, setjmp returns a non-zero value. longjmp does not return.

The following illustrates the use of *setjmp* and *longjmp*.

```
#include <stdio.h>      #include <setjmp.h>
static jmp_buf    loc;
int foo()
{
        printf("Enter foo. Now call longjmp....\n");
        longjmp (loc, 5);
        printf("Should never gets here....\n");
        return 0;
}
```

```
int main()
{
        int retcode;
        if ( (retcode=setjmp( loc )) != 0 )
        {
                printf("Get here from longjmp, Retcode :%d",retcode);
                return 0;
        }
        printf("Program continue after setting loc via setjmp...\n");
        foo();
        printf( "Should never get here ....\n");
        return 0;
}
```

Output: Program continue after setting loc via setjmp...
Enter foo. Now call longjmp....
Get here from longjmp, Retcode : 5

getrlimit and setrlimit Functions

Every process has a set of resource limits, some of which can be queried and changed by the getrlimit and setrlimit functions.

The prototypes of these functions are:

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

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

Both return: 0 if OK, nonzero on error

The resource limits for a process are normally established by process 0 when the system is initialized and then inherited by each successive process. Each implementation has its own way of tuning the various limits.

Each call to these two functions specifies a single resource and a pointer to the following structure:

```
struct rlimit {
        rlim_t  rlim_cur;  /* soft limit: current limit */
        rlim_t  rlim_max;  /* hard limit: maximum value for rlim_cur */
};
```

Three rules govern the changing of the resource limits.

a. A process can change its soft limit to a value less than or equal to its hard limit.
b. A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
c. Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant RLIM_INFINITY.

The resource argument takes on one of the following values.

RLIMIT_CORE	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
RLIMIT_CPU	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the SIGXCPU signal is sent to the process.
RLIMIT_DATA	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap.
RLIMIT_FSIZE	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the SIGXFSZ signal.

RLIMIT_LOCKS	The maximum number of file locks a process can hold.
RLIMIT_MEMLOCK	The maximum amount of memory in bytes that a process can lock into memory using mlock.
RLIMIT_NOFILE	The maximum number of open files per process. Changing this limit affects the value returned by the sysconf function for its _SC_OPEN_MAX argument.
RLIMIT_NPROC	The maximum number of child processes per real user ID. Changing this limit affects the value returned for _SC_CHILD_MAX by the sysconf function.
RLIMIT_RSS	Maximum resident set size (RSS) in bytes. If available physical memory is low, the kernel takes memory from processes that exceed their RSS.
RLIMIT_STACK	The maximum size in bytes of the stack.

The resource limits affect the calling process and are inherited by any of its children. This means that the setting of resource limits needs to be built into the shells to affect all our future processes.

Example

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

The following program prints the current limit and hard limits of all the resource limits.

```
#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) < 0)
                perror("getrlimit error for %s", name);
        printf("%-14s  ", name);
        if (limit.rlim_cur == RLIM_INFINITY)
                printf("(infinite)  ");
        else
                printf("%10ld  ", limit.rlim_cur);
        if (limit.rlim_max == RLIM_INFINITY)
                printf("(infinite)\n");
        else
                printf("%10ld\n", limit.rlim_max);
}

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

Output:

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

UNIX Kernel Support for processes.

The data structure and execution of processes are dependent on operating system implementation. As shown in the figure below, a UNIX process consists minimally of a text segment, data segment, and a stack segment.

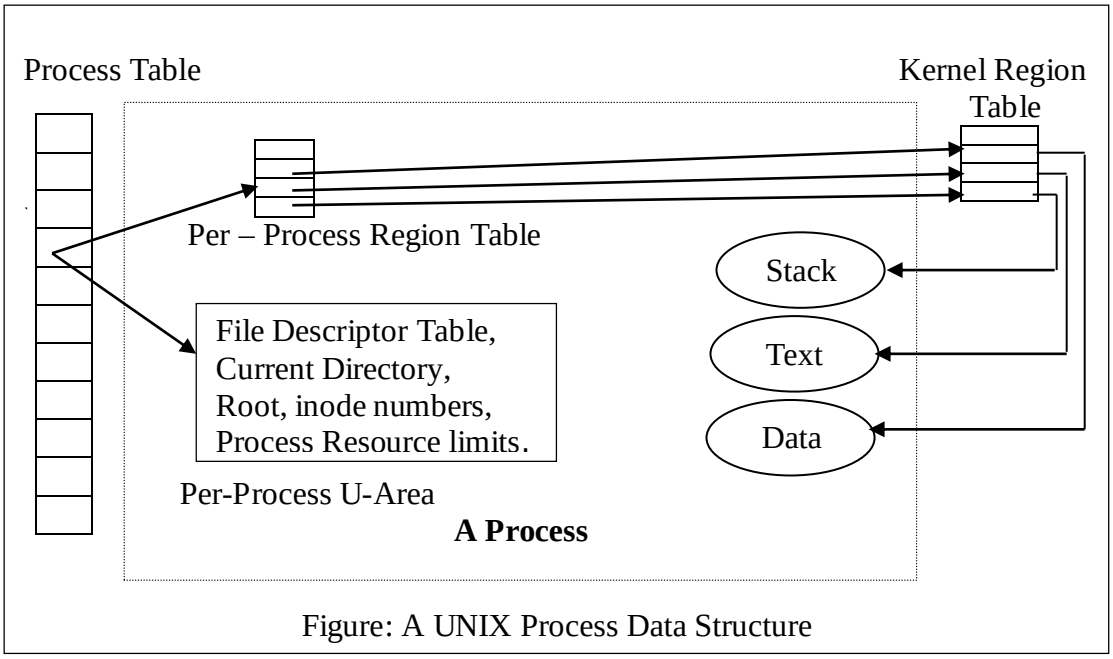
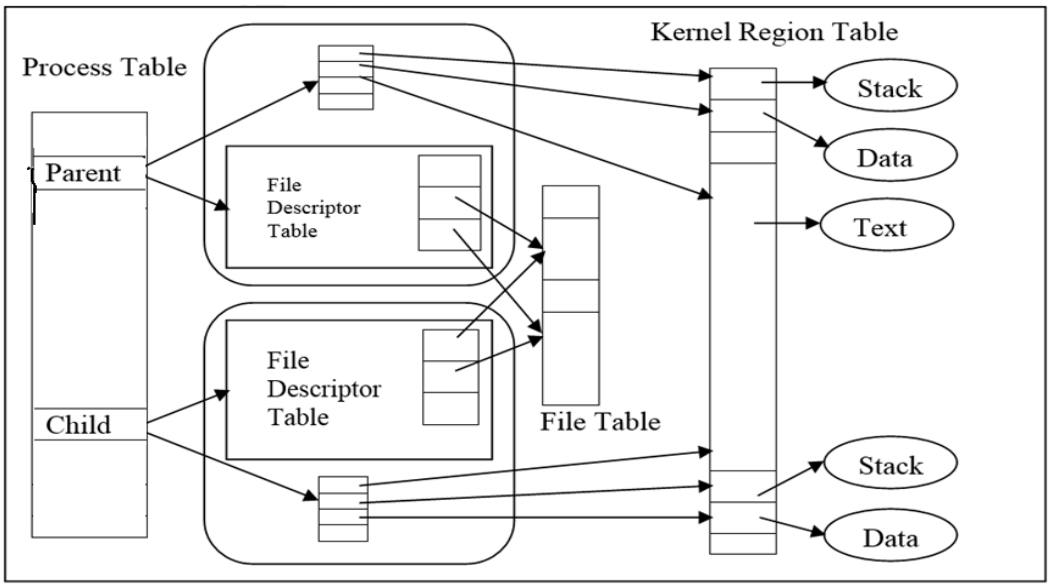


Figure: A UNIX Process Data Structure

A segment is an area of memory that is managed by the system as a unit. A text segment contains the program text of a process in machine—executable instruction code format. A data segment contains static and global variables and their corresponding data. A stack segment contains a run-time stack. A stack provides storage for function arguments, automatic variables, and return address of all active functions for a process at any time. A UNIX kernel has a process table that keeps track of all active processes. Some of the processes belong to the kernel, they are called *system processes*. The majority of processes are associated with the users who are logged in. Each entry in the process table contains pointers to the text, data, stack segments and the U-area of a process. The U-area is an extension of a process table entry and contains other process specific data, such as the file descriptor table, current root, and working directory inode numbers, and a set of system –imposed process resource limits, etc. All processes in UNIX system, except the first process (process 0) which is created by the system boot code, are created via the *fork* system call. After the fork system call both the parent and child processes resume execution at the return of the fork function

When a process is created by fork, it contains duplicate copies of the text, data, and stack segments of its parent. Also, it has an File descriptor table that contains references to the same opened files as its parent, such that they both share the same file pointer to each opened file.



Furthermore, the process is assigned the following attributes which are either inherited by from its parent or set by the kernel.

- ***A real user identification number (rUID):*** the user ID of a user who created the parent process.
- ***A real group identification number (rGID)*:** the group ID of a user who created the parent process.
- ***An effective user identification number (eUID)*:** this is normally the same as the real UID, except when the file that was executed to create the process has its set UID flag turned on, in that case the eUID will take on the UID of the file.
- ***An effective group identification number (eGID)*:** this is normally the same as the real GID, except when the file that was executed to create the process has its set UID flag turned on, in that case the eGID will take on the GID of the file.
- ***Saved set-UID and saved set-GID*:** these are the assigned eUID and eGID, respectively of the process.
- ***Process group identification number (PGID) and session identification number (SID)*:** these identify the process group and session of which the process is member.
- ***Supplementary group identification numbers*:** this is a set of additional group IDs for a user who created the process.
- ***Current Directory*:** this is the reference (inode number) to a working directory file.
- ***Root Directory*:** this is the reference (inode number) to a root directory file.
- ***Signal mask*:** a signal mask that specifies which signals are to be blocked.
- ***Umask*:** a file mode mask that is used in creation of files to specify which accession rights should be taken out.
- ***Nice value*:** the process scheduling priority value.
- ***Controlling terminal*:** the controlling terminal of the process.

In addition to the above attributes, the following attributes are different between the parent and child processes.

- ***Process identification number (PID)***: an integer identification number that is unique per process in an entire operating system.
- ***Parent process identification number (PPID)*:** the parent process ID.
- ***Pending signals*:** the set of signals that are pending delivery to the parent process. This is reset to none in the child process.
- ***Alarm clock time*:** the process alarm clock time is reset to zero in the child process.
- ***File locks*:** the set of file locks owned by the parent process is not inherited by the child process.

Process Control

Introduction

- A Process is a program under execution, For example a.out in UNIX or POSIX system. For example, a UNIX shell is a process that is created when a user logs on to a system.
- Processes are created with the `fork` system call (so the operation of creating a new process is sometimes called *forking* a process). 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. If you want your program to wait for a child process to finish executing before continuing, you must do this explicitly after the fork operation, by calling `wait` or `waitpid`. These functions give you limited information about why the child terminated—for example, its exit status code.
- A newly forked child process continues to execute the same program as its parent process, at the point where the `fork` call returns. You can use the return value from `fork` to tell whether the program is running in the parent process or the child.

Process Identification

Each process is named by a *process ID* number. A unique process ID is allocated to each process when it is created. The *lifetime* of a process ends when its termination is reported to its parent process; at that time, all of the process resources, including its process ID, are freed.

There are some processes:

- Process ID **0** : Is usually a scheduler process is often known as the swapper. It is part of the kernel hence it is known as a system process.
- Process ID **1** : Is usually an *init* process and it is invoked by the kernel at the end of bootstrap procedure. This process is responsible for bringing up a UNIX system after the kernel has been bootstrapped. *init* usually reads the system dependent initialization files such as /etc/rc* files and brings the system to a certain state such as multiuser. i*nit* process never dies. Furthermore he is becoming parent process for orphaned child process. But it is normally user process.
- Process ID **2**: Is the pagedaemon process. This process is responsible for supporting the paging of the virtual memory system. It is also system process.

The `pid_t` data type represents process IDs. You can get the process ID of a process by calling `getpid`. Addition to the *pid*, there are other identifiers for every process. The following functions returns these identifiers.

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid (void);     //returns the process ID of the current process
pid_t getppid (void);  // returns the process ID of the parent of the current process
pid_t grtpgrp(void);    // Returns process group id
uid_t  getuid (void);   // returns the real user ID of the process
gid_t  getgid (void);   //returns the real group ID of the process
uid_t  geteuid (void); // returns the effective user ID of the process
gid_t  getegid (void); // returns the effective group ID of the process
```

pid_t : is a premitive system data type is a signed integer type which is capable of representing a process ID.

uid_t: This is an integer data type used to represent user IDs. This is an alias for `unsigned int`.

gid_t: This is an integer data type used to represent group IDs. In the GNU library, this is an alias for `unsigned int`.

Program to display an attributes of a process

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
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());
}
```

Output: Process PID: 19601
Process PPID: 19561
Process PGID: 19601
Process real-UID: 500
Process real-GID: 500
Process effective-UID: 500
Process effective-GID: 500

Creating a Process

The `fork` function is the primitive for creating a process.

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

The fork function takes no arguments, and it returns a value of type pid_t. If the operation is successful. The result of the call may be one of the following:

- It returns a value of 0 in the child process and returns the child's process ID in the parent process.
- If process creation failed, `fork` returns a value of `-1` in the parent process. The following `errno` error conditions are defined for `fork`:
  - `EAGAIN` : There aren't enough system resources to create another process, or the use already has too many processes running. This means exceeding the `RLIMIT_NPROC` resource limit.
  - `ENOMEM` : The process requires more space than the system can supply.

There are two uses for fork:

1. When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time. This is common for network servers—the parent waits for a service request from a client. When the request arrives, the parent calls fork and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
2. When a process wants to execute a different program. This is common for shells. In this case, the child does an exec right after it returns from the fork.

The program below demonstrates the fork function, showing how changes to variables in a child process do not affect the value of the variables in the parent process.


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

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
```

In general, we never know whether the child starts executing before the parent or vice versa.

This depends on the scheduling algorithm used by the kernel. If it's required that the child and parent synchronize, some form of inter-process communication is required.
In the program shown above, we simply have the parent put itself to sleep for 2 seconds, to let the child execute. There is no guarantee that this is adequate.
Note the interaction of fork with the I/O functions in the program in the above program. The write function is not buffered. Because write is called before the fork, its data is written once to standard output.

`vfork` Function

The function `vfork` has the same calling sequence and same return values as `fork`. But the semantics of the two functions differ.
The `vfork` function is intended to create a new process when the purpose of the new process is to `exec` a new program.
The `vfork` function creates the new process, just like `fork`, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls `exec` (or `exit`) right after the `vfork`.
Instead, while the child is running and until it calls either `exec` or `exit`, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System.
Another difference between the two functions is that `vfork` guarantees that the child runs first, until the child calls `exec` or `exit`. When the child calls either of these functions, the parent resumes. (**This can lead to deadlock if the child depends on further actions of the parent before calling either of these two functions**.)

The program below is a modified version of the program from fork function. We've replaced the call to `fork` with `vfork` and removed the `write` to standard output. Also, we don't need to have the parent call `sleep`, as we're guaranteed that it is put to sleep by the kernel until the child calls either `exec` or `exit`.

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
int     glob = 14;                      /* external variable in initialized data */
int main(void)
{
   int      var=15;                     /* automatic variable on the stack */
   pid_t    pid;
   if ((pid = vfork()) ==-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
glob =16,  var =  17
```

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(l) Command to verify that the process is Zombie.

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

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


Output:

```
  PID TTY          TIME CMD
 3861 tty1     00:00:00 su
 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 funcions

The wait and waitpid API's are used by parent process to wait for its child process to terminate or stop, and determine its status and also these API's will deallocates process table slot of the child process, so that the slot can be reused by a new 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 does not wait for the first child to terminate- it has number of options that control which process it waits for.

If a child has already terminated and is a zombie, *wait* returns immediately with that child's status. Otherwise it blocks the parent until a child terminates. if the parent blocks and has multiple children, *wait* returns when one terminates

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.

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.

The status information from the child process is stored in the object that *status* points to, unless *status* is a null pointer.

Posix.1 specifies various macros to find out how the process terminated. These macros are defined in the header file <sys/wait.h> they are:

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.

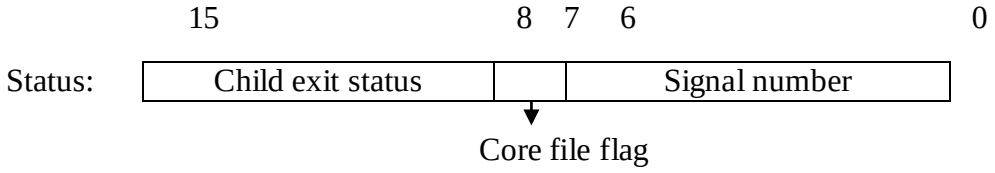
int **WTERMSIG** (*int status*) : If `WIFSIGNALED` is true of *status*, this macro returns the signal number of the signal that terminates the child process.

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.

Status bits are represented pictorially as follows:



Note: The simplified version of `waitpid`, and is used to wait until any one child process terminates.

The call: **wait (&status) ;** is exactly equivalent to: **waitpid (-1, &status, 0);**

Features of waitpid over wait function:

1. waitpid wait for perticular child process but *wait* waiting for anyone to terminate.
2. waitpid provides a nonblocking version of *wait*.
3. waitpid supports jobcontrol ie WUNTRACED.

wait3 and wait4 functions

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

pid_t wait3 (in *status, int options, struct rusage *usage);
pid_t wait4 (pid_t pid, int *status, int options, struct rusage *usage);
```


These are just like waitpid but these are having additional argument that allows the kernel to return a summary of resources used by the terminated process and all child processes**.**

The resource information includes the amount of use CPU time, the amount of system CPU time, number of page faults, number of signals received.

Write a program to avoid zombie using wait and then call system to execute the PS Command to verify that the process is Zombie or not.

```
#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)                  /* child */
                    exit(0);
```

```
                                        /* parent */
        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
```

Write a program to avoid zombie using waitpid(), and then call system to execute the PS Command to verify that the process is Zombie or not.

```
#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)                  /* child */
                  exit(0);

                                        /* parent */
        if (pid  > 0)
        {
                waitpid(pid,&status,0);
                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
```

The program shown below calls the prexit function, demonstrating the various values for the termination status.

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

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int status;
void prexit( )
{
   if (WIFEXITED(status))
          printf("normal termination, exit status = %d\n", WEXITSTATUS(status));
   if (WIFSIGNALED(status))
          printf("abnormal termination, signal number = %d%s\n",  WTERMSIG(status));
    if (WCOREDUMP(status))
          printf ("core file generated");
    If (WIFSTOPPED(status))
          printf("child stopped, signal number = %d\n", WSTOPSIG(status));
}

int main(void)
{
   pid_t   pid;

   if ((pid = fork()) ==-1)
          {
                 perror("fork");
                 return 0;
          }
   if (pid == 0)                                  /* child */
          exit(14);
    if(pid >0)
    {
          if (wait(&status) != pid)               /* wait for child */
          perror("wait");
          prexit( );                              /* and print its status */

      if ((pid = vfork()) ==-1)
          {
                 perror("fork");
                 return 0;
          }

      if (pid == 0)                               /* child */
          abort();                                /* generates SIGABRT */

       if(pid>0) {
                 if (wait(&status) != pid)        /* wait for child */
                        perror("wait");
                       prexit();                 /* and print its status */
          }
     return 0;
}
```

```
$ ./a.out
normal termination, exit status = 14
abnormal termination, signal number = 6
```

Race Condition

Race condition occurs when multiple processes are trying to do something on the shared data and the final outcome depends on the order in which the process run. We know that after forking both the process are running simultaniously so in general we can't predict which process runs first.

If a parent process wants child to terminate, it must call one of the wait functions.

Suppose if a child proces wants to wait for its parent to terminate, The folowing form could be used:

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

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

We can avoid race condition and polling by:

i. Signals
ii. Various forms of inter process mechanisms
iii. 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

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

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);
}
```


The above program prints two strings: One from the child and one from the parent. It contains a race condition because the output depends on the order in which the processes are run by the kernel.

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

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

```
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);
}
```

Output: output from parent
output from child

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

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

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

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

```
        TELL_WAIT();
                if ((pid = fork()) ==-1)
                    {
                            perror("fork");
                            return 0;
                    }

         if (pid == 0)
        {
                fun("output from child\n");
                TELL_PARENT(getppid( ));
        }
        if (pid >0)
        {
                WAIT_CHILD( );              /* Child goes first */
                fun("output from parent\n");
        }
        exit(0);
}
```

Output: output from child
output from parent

exec Functions

These functions are used to execute another program. When a process calls one of the exec functions, that process is completely replaced by the new program and the new program starts executing at its main function. The process ID does not change across the exec functions because a new process is not created. Exec replaces the current process's (its text, data, heap and stack segments) with a brand new program from the disk.

Note: Calling Exec is like a person changing jobs. After the change, the person still has the same name and personal identifications, but is now working on a different job than before.

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.

The differences between above six functions are:

1. Functions 1, 3, 4 and 6 takes a pathname as arguments but 2 , 5 will takes the file name as arguments.
2. Passing of the argument list (l - stands for list, v – stands for vector).
3. Passing of the environment list (e- environment list).

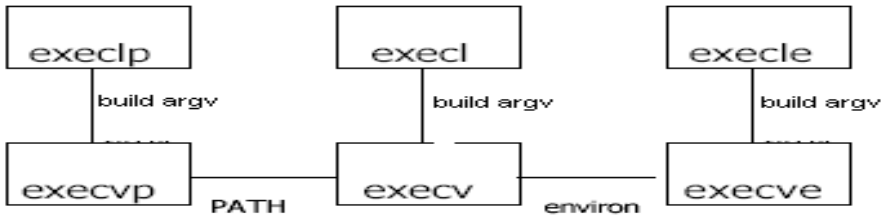
These functions normally don't return, since execution of a new program causes the currently executing program to go away completely. A value of -1 is returned in the event of a failure.

ENOEXEC : The specified file can't be executed because it isn't in the right format.
ENOMEM : Executing the specified file requires more storage than is available.

Executing a new process image completely changes the contents of memory, copying only the argument and environment strings to new locations. But many other attributes of the process are unchanged:

- The process ID and the parent process ID.
- Session and process group membership.
- Real user ID and group ID, and supplementary group IDs.
- Pending alarms.
- Current working directory and root directory.
- File mode creation mask.
- Process signal mask,
- Pending signals,
- Elapsed processor time associated with the process,
- If the set-user-ID and set-group-ID mode bits of the process image file are set, this affects the effective user ID and effective group ID (respectively) of the process.

Among six exec functions execve is a system call within the kernel. The other five are just library functions that eventually invoke this system call. The below picture shows the relation ship between the six different functions.



Relationship of the *exec()* functions.

Exec1.cpp

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

 main()
{
  cout<<"\n Exec1.cpp  ";                  //assume there is no Exec2 Executable file
if(( execl("/home/chandu/Exec2","exe2",NULL))!=0)
        cout<<"\nExec error..";
  cout<<"\nEnd of Exec1.cpp...\n";     //this will not print
  return 0;
}
```

Output: Exec1.cpp
Exec error..
End of Exec1.cpp

Exec2.cpp

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

main()
{
 cout<<"\n Exec2.cpp  ";
 cout<<"\n\nEnd of Exec2.cpp "<<endl<<endl<<endl;
 return 0;
}
```

Example 3:

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

 main()
{
 cout<<"\n Exec1.cpp  ";       //assume Exec2 Executable file is available
if(( execl("/home/chandu/Exec2","exe2",NULL))!=0)
          cout<<"\nExec error..";
 cout<<"\nEnd of Exec1.cpp...\n"; //this will not print
 return 0;
}
```

Output:

```
Exec2.cpp
End of Exec2.cpp
```


The system function

This function used for running a command. The prototype of the function is:

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

This function executes *cmd* as a shell command. It always uses the default shell `sh` to run the command.

The *system* function is implemented by using fork, exec, and waitpid APIs as shown below. The system() calls fork to create a child process, the child process in turn calls execl to execute the bourne shell program (/bin/sh) with –c and cmd as an arguments. The –c option instructs the bourne shell to interpret and execute the cmd arguments. After cmd is executed, the child process is terminated and the exit status of the bourne shell is passed to the parent process, parent retrieves the exit status using waitpid API..

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

```
#include <unistd.h>
int system(const char *cmd)
{
        pid_t   pid;
        int status;
        if (cmd == 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", cmd, (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);
}
```

Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.

1. If either the fork or waitpid fails, returns -1 other than EINTR.
2. If the exec fails, returns 127.
3. Otherwise, all three functions—fork, exec, and waitpid—succeed, and the return value from system is the termination status of the shell.

Example :

```
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
int status;
int system(const char *cmd)
{
        pid_t   pid;

        if (cmd == 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", cmd ,(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);
}

void exitstatusprint()
{
        if (WIFEXITED(status))
                printf("Normal exits:  %d", WEXITSTATUS(status));
        if (WIFSTOPPED(status))
                printf("Stopped by:   %d",WSTOPSIG(status));
        if (WIFSIGNALED(status))
                printf(" Signaled by: %d",WTERMSIG(status));
}

int  main(void)
{
        if ( (status = system("date; exit 0")) < 0)
                perror("system error");
        exitstatusprint();

        if ( (status = system("daaate")) < 0)
                perror("system error");
        exitstatusprint();

        if ( (status = system("who; exit 44")) < 0)
                perror("system error");
        exitstatusprint();

        exit(0);
}
```


Output:

```
Sun  Jan  12  10:20:10  IST  2006
Normal exits: 0                          //for date command

sh : daaate : not found
Normal exits: 1                          //for daaate command

chandu     tty01        Jan  12  10:35
 Normal exits: 44
```

Hence, System invokes the UNIX command interpreter file from inside an executing C program to execute a UNIX command, batch file, or other program named by the string "command".

To be located and executed, the program must be in the current directory or in one of the directories listed in the PATH string in the environment.

Interpreter Files

These files are text files that begin with a line of the form

#! *path name [optional-argument]*

The space between the exclamation point and the *pathname* is optional. The most common of these begin with the line

```
#!/bin/sh
```

The *pathname* is normally an absolute pathname, since no special operations are performed on it (Le., PATH is not used). The recognition of these files is done within the kernel as part of processing the exec system call. The actual file that gets execed by the kernel is not the interpreter file, but the file specified by the *pathname* on the first line of the interpreter file. Be sure to differentiate between the interpreter file (a text file that begins with #!) and the interpreter (specified by the *pathname* on the first line of the interpreter file).

Be aware that many systems have a limit of 32 characters for the first line of an interpreter file. This includes the #!, the *pathname*, the optional argument, and any spaces.

Users and Groups

Every user who can log in on the system is identified by a unique number called the *user ID*. Each process has an effective user ID which says which user's access permissions it has.

Users are classified into *groups* for access control purposes. Each process has one or more *group ID values* which say which groups the process can use for access to files.

The effective user and group IDs of a process collectively form its *persona*. This determines which files the process can access. Normally, a process inherits its persona from the parent process, but under special circumstances a process can change its persona and thus change its access permissions.

Changing User IDs and Group IDs

We can set the real user ID and effective user ID with the *setuid* function. Similarly, we can set the real group ID and the effective group ID with the *setgid* function.

```
#include <sys/types.h>
#include <unistd:h>
int setuid (uid_t uid) ;
int setgid(gid_t gid);
```

Both return: 0 if OK, -1 on error

There are rules for who can change the IDs.

1. If the process has superuser privileges, the *setuid* function sets the real user ID, effective user ID, and saved set-user-ID to *uid*.
2. If the process does not have superuser privileges, but *uid* equals either the real user ID or the saved set-user-ID, setuid sets only the effective user ID to *uid*. The real user ID and the saved set-user-ID are not changed.
3. If neither of these two conditions is true, errno is set to EPERM and an error is returned.

Above said rules are applicable for *setgid* function.

We can make a couple of statements about the three user IDs that the kernel maintains.

1. Only a superuser process can change the real user ID. Normally the real user ID is set by the *login* program when we log in and never changes. Since login is a superuser process, when it calls setuid it sets all three user IDs.
2. The effective user ID is set by the *exec* functions, only if the set-user-ID bit is set for other Program file. If the set-user-ID bit is not set, the *exec* functions leave the effective

user ID as its current value. We can call *setuid* at any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally, we cannot set the effective user ID to any random value.

3. The saved set-user-ID is copied from the effective user ID by exec. This copy is saved after *exec* stores the effective user ID from the file's user ID (if the file's set-user-ID bit is set).

seteuid and setegid Functions

This section describes the functions for altering the effective user ID and effective group IDs of a process.

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

int seteuid (uid_t neweuid);
int setegid (gid_t newgid);
```

This function sets the effective user ID of a process to *newuid*, provided that the process is allowed to change its effective user ID. A privileged process (effective user ID zero) can change its effective user ID to any legal value. An unprivileged process with a file user ID can change its effective user ID to its real user ID or to its file user ID. Otherwise, a process may not change its effective user ID at all.

The `seteuid` function returns a value of `0` to indicate successful completion, and a value of `-1` to indicate an error. The following `errno` error conditions are defined for this function:

`EINVAL` : The value of the *newuid* argument is invalid.
`EPERM` : The process may not change to the specified ID.

setreuid and setregid Functions

These functions are swapping the real user ID and The effective user ID, when invoked by the unprivileged user.

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

int setreuid (uid_t ruid, uid_t euid);
int setregid (gid_t rgid, gid_t egid);
```

This function sets the real user ID of the process to *ruid* and the effective user ID to *euid*. If *ruid* is `-1`, it means not to change the real user ID; likewise if *euid* is `-1`, it means not to change the effective user ID.

The `setreuid` function exists for compatibility with 4.3 BSD Unix, which does not support file IDs. You can use this function to swap the effective and real user IDs of the process

The return value is `0` on success and `-1` on failure. The following `errno` error conditions are defined for this function:

`EPERM` : The process does not have the appropriate privileges; you do not have permission to change to the specified ID.

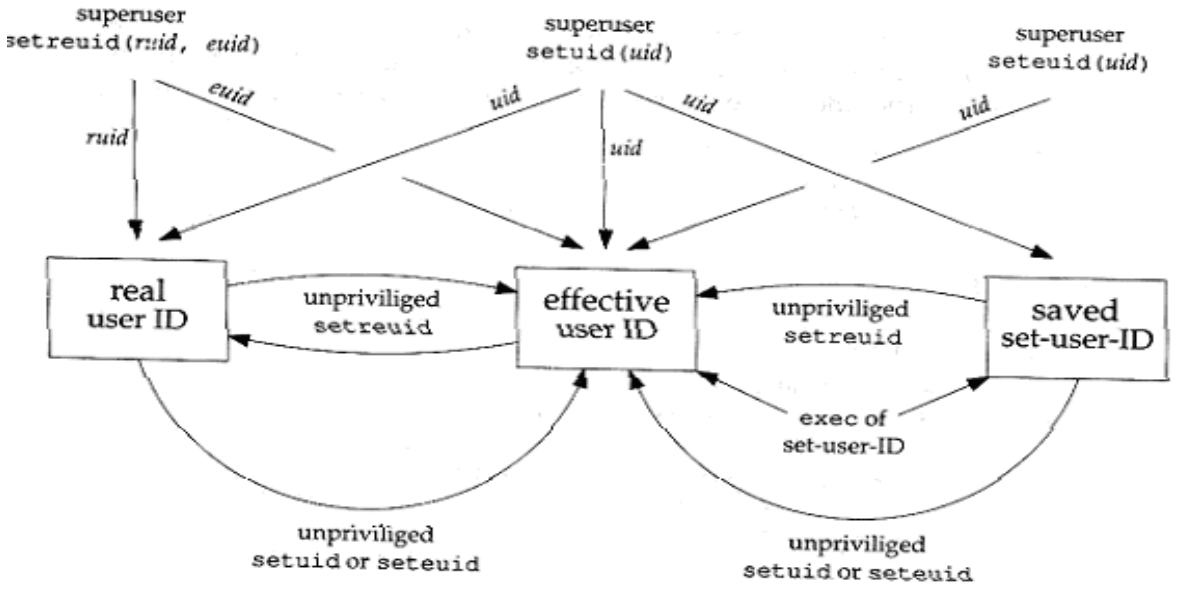


Figure : summary of all the functions that set the different user IDs

Process Accounting

Most Unix-like operating systems provide an option to do process accounting. When enabled the kernel writes an accounting record each time a process terminates. These accounting records are 32 bytes of binary data, which contains the name of the command, amount of CPU time used, the user ID , group user ID, starting time and so on.

The function *acct* (or by using *accton command*) enables and disables process accounting. When super user executes *accton* with a path name argument to enable the accounting. The pathname is usually /var/adm/pacct. Accounting is turned off by executing *accton* without any arguments.

The structure of the accounting records is defined in the header <sys/acct.h> and looks like

```
typedef  u_short comp_t;
struct  acct
{
        char     ac_flag;       /* flag  (see below mentioned table) */
        char      ac_stat;      /* termination status (signal & core flag only) */
        uid_t     ac_uid;       /* real user 1D */
        gid_t     ac_gid;       /* real group 1D */
        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 */
} ;
```

The *ac_flag* member records certain events during the execution of the process.

ac_flag	Description
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.

- The data required for the accounting record (CPU times, number of characters transferred, etc.) are all kept by the kernel in the process table and initialized whenever a new process is created (e.g., in the child after a fork).
- Each accounting record is written when the process terminates. This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started.
- The accounting records correspond to processes, not programs. A new record is initialized by the kernel for the child after a fork, not when a new program is exec-ed. Although exec does not create a new accounting record, the command name changes and the AFORK flag is cleared. This means that, if we have a chain of three programs (A execs B, then B execs C, and C exits), only a single accounting record is written. The command name in the record corresponds to program C but the CPU times, for example, are the sum for programs A, B, and C.

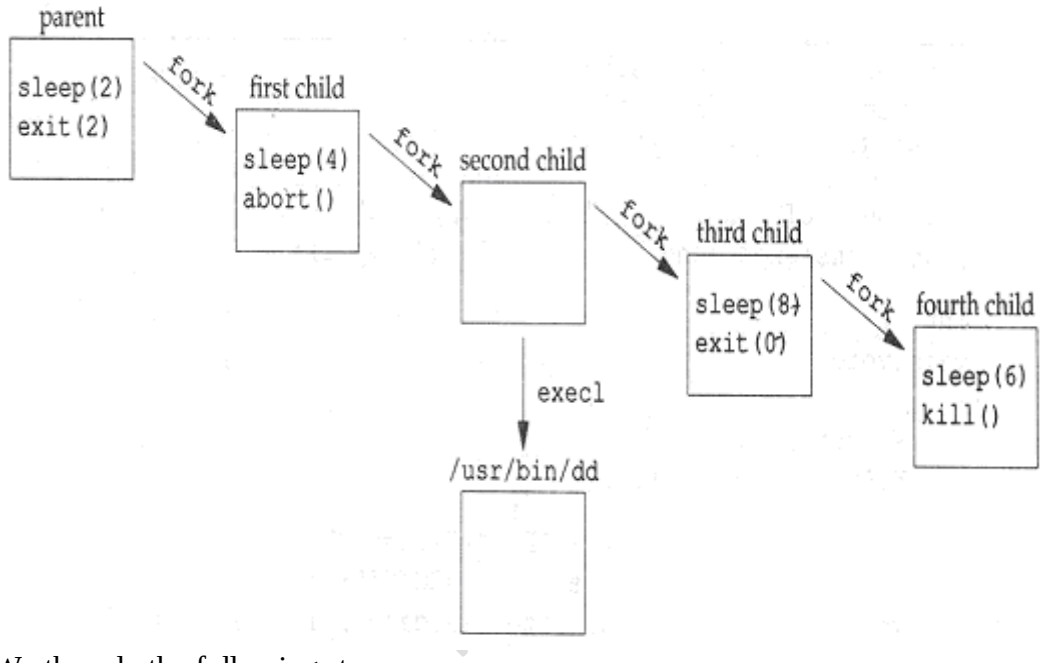
To have some accounting data to examine, we will run the below mentioned Program, which calls fork four times. Each child does something different and then terminates. A picture of what this program is doing is shown in below.

Program to generate accounting data: **accprogram1**

```
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
into main(void)
{
        pid_t   pid;

        if ( (pid = fork()) < 0)
                perror("fork error");
        else if (pid != 0)
        {                                       /* parent */
                sleep(2);
                exit(2);                        /* terminate with exit status 2 */
        }
                                                /* first child */
        if ( (pid = fork()) < 0)
                perror("fork error");

        else if (pid != 0)
                {
                        sleep(4);
                        abort();                /* terminate with core dump */
                }
                                                /* second child */
        if ( (pid = fork()) < 0)
                perror("fork error");

        else if (pid != 0)
        {
                execl("/usr/bin/dd", "dd", "if=/boot", "of=/dev/null", NULL);
                exit(7);                        /* shouldn't get here */
        }
                                                /* third child */
```

```
        if ( (pid = fork()) < 0)
                perror("fork error");
        else if (pid != 0)
        {
                sleep(8);
                exit(0);                                /* normal exit */
        }
                                                /* fourth child */
        sleep(6);
        kill(getpid(), SIGKILL);                /* terminate with signal, no core dump*/
         exit(6);                               /* shouldn't get here */
}
```

Process structure for accounting example is shown below:



We then do the following steps:

1. Become super user and enable accounting, with the *accton* command. Note that, when this command terminates, accounting should be on, therefore the first record in the accounting file should be from this command. .
2. Run Program accprogram1. This should append five records to the accounting file(one for the parent and one for each of the four children).
   A new process is not created by the execl in the second child. There is only a single accounting record for the second child.
3. Become superuser and turn accounting off. Since accounting is off when this *accton* command terminates, it should not appear in the accounting file.

Process Times

This is also called CPU time and measures the central processor resources used by a process. It is measured in clock ticks, which has 50, 60 or 100 ticks per second. The primitive system data type clock_t holds these time values. The POSIX defines the constant _SC_CLK_TCK to specify the number of ticks per second.

UNIX maintains three values for a process:

1. clock time : This is the amount of time the process takes to run.
2. user CPU time: Is the CPU time that is attributed to user instructions.
3. system CPU time: Is the CPU time that is attributed to the kernel, When executes on behalf of the process.

The sum of the user cpu time & system cpu time is often called the cpu time.
Any process can call the `times` function to obtain these values for itself and any terminated children.
The prototype of times function is as:

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

clock_t times(struct tms *buf);
```

Returns: elapsed wall clock time in clock ticks if OK, -1 on error

This function fills in the tms structure pointed to by 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 */
};
```

Example

The program below executes each command-line argument as a shell command string, timing the command and printing the values from the `tms` structure.


```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/times.h>

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

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

   setbuf(stdout, NULL);
   for (i = 1; i < argc; i++)
      do_cmd(argv[i]);    /* once for each command-line arg */
   exit(0);
}

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

    printf("\ncommand: %s\n", cmd);

    if ((start = times(&tmsstart)) == -1)    /* starting values */
       perror("times error");

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

    if ((end = times(&tmsend)) == -1)       /* ending values */
```

```
        perror("times error");
    pr_times(end-start, &tmsstart, &tmsend);
    prexit( );
}

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");
    printf(" real:  %7.2f\n", real / (double) clktck);
    printf(" user:  %7.2f\n",
      (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf(" sys:   %7.2f\n",
      (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf(" child user:   %7.2f\n",
      (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    printf(" child sys:    %7.2f\n",
      (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}
```

Most frequently asked questions

1. With a block diagram explain how a 'C' program is started and the various ways it can terminate.
2. What are the different ways for a process to terminate?. Explain: exit, _exit and atexit functions with their prototypes.
3. What is an environment variable? Give four of them and explain their use.
4. Explain memory layout of a 'C' program?
5. Explain with an example use of setjmp and longjmp functions.
6. Explain dynamic allocation techniques. Explain Advantages and disadvantages of alloca.
7. Explain with an example the setrlimit and getrlimit functions
8. Explain getenv, setenv, putenv and unsetenv functions.
9. Write a program that creates a zombie and then call system to execute the PS(l) Command to verify that the process is Zombie.
10. Explain different **exec** functions how they function differ from each other?
11. What is race condition explain with an example? Write a program to demonstrate Race condition.
12. Explain wait, waitpid, wait3 and wait4 functions with their prototypes and uses.
13. Write a program in 'C' to obtain process attributes.
14. Write a note on process accounting.
15. What is race condition explain with an example, How to avoid Race condition?