MODULE – 3
UNIX Processes

4.1 Introduction:

A Process is a program under execution in a UNIX or POSIX system

4.1.1 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, and 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

4.1.2 Process Termination:

There are eight ways for a process to terminate.

Normal termination occurs in five ways:

- Return from main
- Calling exit
- Calling _exit or _Exit
- Return of the last thread from its start routine
- Calling pthread_exit from the last thread

Abnormal termination occurs in three ways:

- Calling abort
- Receipt of a signal
- Response of the last thread to a cancellation request

Exit Functions

Three functions terminate a program normally: _exit and _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);
```

```
void _Exit(int status);

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

All three exit functions expect a single integer argument, called the exit status. Returning an integer value from the main function is equivalent to calling exit with the same value. Thus **exit(0);** is the same as **return(0);** from the main function.

In the following situations the exit status of the process is undefined.

- Any of these functions is called without an exit status
- Main does a return without a return value
- Main "falls off the end", i.e if the exit status of the process is undefined.

Example: **$ cc hello.c**
$ ./a.out hello, world
$ echo
$? // print the exit status
13

4.1.3 Command-Line Arguments:

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

Example: Echo all command-line arguments to standard output

```
#include "apue.h"
int main(int argc, char *argv[])
{
int i;
for (i = 0; i < argc; i++) /* echo all command-line args */
printf("argv[%d]: %s\n", i, argv[i]);
exit(0);
}
```

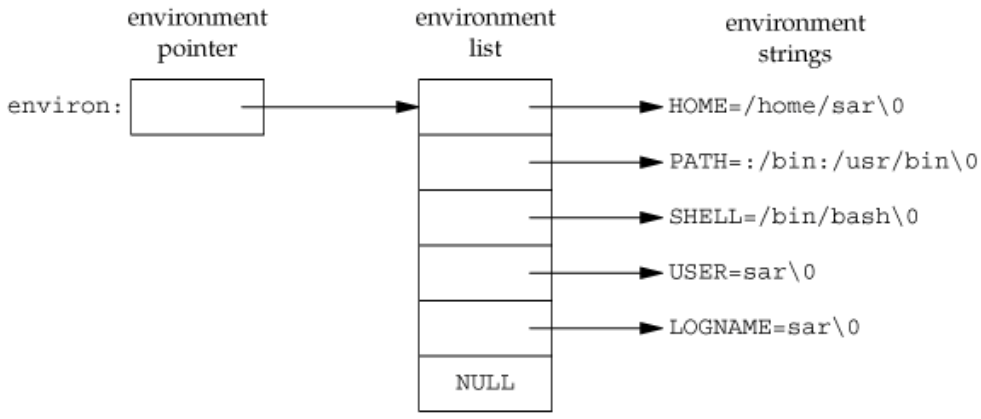
Output:
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo

4.1.4 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;

Figure: Environment consisting of five C character strings



4.2 Memory Layout of a C Program:

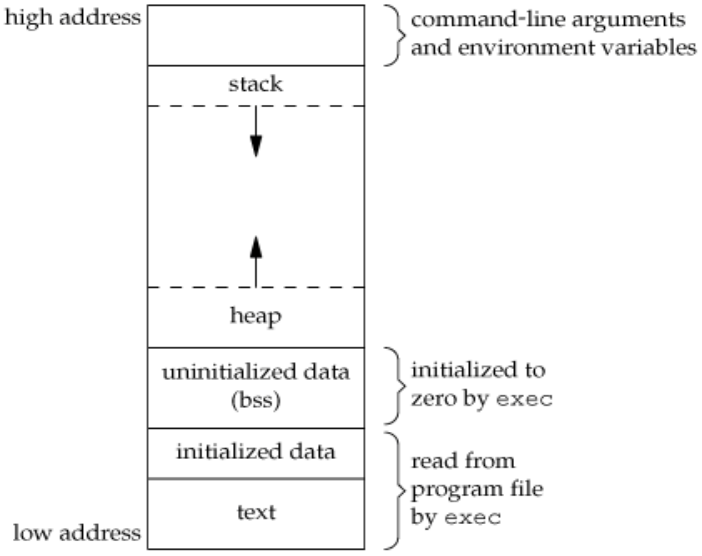
Historically, a C program has been composed of the following pieces:

- **Text segment**, the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.
- **Initialized data segment**, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration

  int maxcount = 99; appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.
- **Uninitialized data segment**, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration

  long sum[1000]; appearing outside any function causes this variable to be stored in the uninitialized data segment.

- **Stack**, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.
- **Heap**, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.



4.2.1 Shared Libraries:

Nowadays most UNIX systems 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 re-link, edit every program that uses the library. With cc compiler we can use the option –g to indicate that we are using shared library.

4.2.2 Memory Allocation:

ISO C specifies three functions for memory allocation:

- malloc, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
- calloc, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
- realloc, which increases or decreases the size of a previously allocated area. When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

  #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 return: non-null pointer if OK, NULL on error **void free(void *ptr);** The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any data object. Because the three alloc functions return a generic void * pointer, if we #include <stdlib.h> (to obtain the function prototypes), we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type. The function free causes the space pointed to by ptr to be deallocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three alloc functions.

The realloc function lets us increase or decrease the size of a previously allocated area. For example, if we allocate room for 512 elements in an array that we fill in at runtime but find that we need room for more than 512 elements, we can call realloc. If there is room beyond the end of the existing region for the requested space, then realloc doesn't have to move anything; it simply allocates the additional area at the end and returns the same pointer that we passed it. But if there isn't room at the end of the existing region, realloc allocates another area that is large enough, copies the existing 512-element array to the new area, frees the old area, and returns the pointer to the new area.

The allocation routines are usually implemented with the sbrk(2) system call. Although sbrk can expand or contract the memory of a process, most versions of malloc and free never decrease their memory size. The space that we free is available for a later allocation, but the freed space is not usually returned to the kernel; that space is kept in the malloc pool.

It is important to realize that most implementations allocate a little more space than is requested and use the additional space for record keeping the size of the allocated block, a pointer to the next allocated block, and the like. This means

that writing past the end of an allocated area could overwrite this record-keeping information in a later block. These types of errors are often catastrophic, but difficult to find, because the error may not show up until much later. Also, it is possible to overwrite this record keeping by writing before the start of the allocated area. Because memory allocation errors are difficult to track down, some systems provide versions of these functions that do additional error checking every time one of the three alloc functions or free is called. These versions of the functions are often specified by including a special library for the link editor. There are also publicly available sources that you can compile with special flags to enable additional runtime checking.

Alternate Memory Allocators

Many replacements for malloc and free are available.

- **libmalloc:** SVR4-based systems, such as Solaris, include the libmalloc library, which provides a set of interfaces matching the ISO C memory allocation functions. The libmalloc library includes mallopt, a function that allows a process to set certain variables that control the operation of the storage allocator. A function called mallinfo is also available to provide statistics on the memory allocator.
- **vmalloc:** Vo describes a memory allocator that allows processes to allocate memory using different techniques for different regions of memory. In addition to the functions specific to vmalloc, the library also provides emulations of the ISO C memory allocation functions.
- **quick-fit:** Historically, the standard malloc algorithm used either a best-fit or a first-fit memory allocation strategy. Quick-fit is faster than either, but tends to use more memory. Free implementations of malloc and free based on quick-fit are readily available from several FTP sites.
- **alloca Function:** The function alloca has the same calling sequence as malloc; however, instead of allocating memory from the heap, the memory is allocated from the stack frame of the current function. The advantage is that we don't have to free the space; it goes away automatically when the function returns. The alloca function increases the size of the stack frame. The disadvantage is that some systems can't support alloca, if it's impossible to increase the size of the stack frame after the function has been called.

4.2.3 Environment Variables:

The environment strings are usually of the form: ***name=value.*** The UNIX kernel never looks at these strings; their interpretation is up to the various applications. The shells, for example, use numerous environment variables. Some, such as HOME and USER, are set automatically at login, and others are for us to set. We normally set environment variables in a shell start-up file to control the shell's actions. The functions that we can use to set and fetch values

from the variables are setenv, putenv, and getenv functions. The prototype of these functions are

#include <stdlib.h>

char *getenv(const char *name);

Returns: pointer to value associated with name, NULL if not found.

Note that this function returns a pointer to the value of a **name=value** string. We should always use getenv to fetch a specific value from the environment, instead of accessing environ directly. In addition to fetching the value of an environment variable, sometimes we may want to set an environment variable. We may want to change the value of an existing variable or add a new variable to the environment. The prototypes of these functions are

#include <stdlib.h>

int putenv(char *_str_);
int setenv(const char *_name_, const char *_value_, int _rewrite_);
int unsetenv(const char *_name_);

All return: 0 if OK, nonzero on error.

- 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.
- 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.
- The **unsetenv** function removes any definition of name. It is not an error if such a definition does not exist.

Note the difference between **putenv** and **setenv**. Whereas **setenv** must allocate memory to create the **name=value** string from its arguments, **putenv** is free to place the string passed to it directly into the environment.

Environment variables defined in the Single UNIX Specification

Variable	**Description**
COLUMNS	terminal width
DATEMSK	getdate(3) template file pathname
HOME	home directory
LANG	name of locale
LC_ALL	name of locale
LC_COLLATE	name of locale for collation
LC_CTYPE	name of locale for character classification

LC_MESSAGES	name of locale for messages
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
LINES	terminal height
LOGNAME	login name
MSGVERB	fmtmsg(3) message components to process
NLSPATH	sequence of templates for message catalogs
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

NOTE:

- If we're modifying an existing name:
  a) If the size of the new value is less than or equal to the size of the existing value, we can just copy the new string over the old string.
  b) If the size of the new value is larger than the old one, however, we must malloc to obtain room for the new string, copy the new string to this area, and then replace the old pointer in the environment list for name with the pointer to this allocated area.
- If we're adding a new name, it's more complicated. First, we have to call malloc to allocate room for the name=value string and copy the string to this area.
  a) Then, if it's the first time we've added a new name, we have to call malloc to obtain room for a new list of pointers. We copy the old environment list to this new area and store a pointer to the name=value string at the end of this list of pointers. We also store a null pointer at the end of this list, of course. Finally, we set environ to point to this new list of pointers.
  b) If this isn't the first time we've added new strings to the environment list, then we know that we've already allocated room for the list on the heap, so we just call realloc to allocate room for one more pointer. The pointer to the new name=value string is stored at the end of the list (on top of the previous null pointer), followed by a null pointer.

4.3 setjmp and longjmp Functions:

In C, we can't goto a label that's in another function. Instead, we must use the setjmp and longjmp functions to perform this type of branching. As we'll see, these two functions are useful for handling error conditions that occur in a deeply nested function call.

#include <setjmp.h>

int setjmp(jmp_buf env);

Returns: 0 if called directly, nonzero if returning from a call to longjmp

void longjmp(jmp_buf env, int val);

The setjmp function records or marks a location in a program code so that later when the longjmp function is called from some other function, the execution continues from the location onwards. The env variable(the first argument) records the necessary information needed to continue execution. The env is of the jmp_buf defined in <setjmp.h> file, it contains the task.

Example of setjmp and longjmp

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD 5

jmp_buf jmpbuffer;

int main(void)
{
    char line[MAXLINE];
    if (setjmp(jmpbuffer) != 0)
    printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
    do_line(line);
    exit(0);
}
…
void cmd_add(void)
{
    int token;
    token = get_token();
    if (token < 0) /* an error has occurred */
    longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

- The setjmp function always returns '0' on its success when it is called directly in a process (for the first time).
- The longjmp function is called to transfer a program flow to a location that was stored in the env argument.
- The program code marked by the env must be in a function that is among the callers of the current function.

- When the process is jumping to the target function, all the stack space used in the current function and its callers, upto the target function are discarded by the longjmp function.
- The process resumes execution by re-executing the setjmp statement in the target function that is marked by env. The return value of setjmp function is the value(val), as specified in the longjmp function call.
- The 'val' should be nonzero, so that it can be used to indicate where and why the longjmp function was invoked and process can do error handling accordingly.

Note: The values of ***automatic*** and ***register*** variables are indeterminate when the longjmp is called but static and global variable are unaltered. The variables that we don't want to roll back after longjmp are declared with keyword 'volatile'.

4.4 getrlimit, setrlimit Functions:

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

```
#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

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 process can change its soft limit to a value less than or equal to its hard limit.
- 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.
- Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant RLIM_INFINITY

RLIMIT_AS	The maximum size in bytes of a process's total available memory

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(2)
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_SBSIZE	The maximum size in bytes of socket buffers that a user can consume at any given time
RLIMIT_STACK	The maximum size in bytes of the stack
RLIMIT_VMEM	This is a synonym for RLIMIT_AS

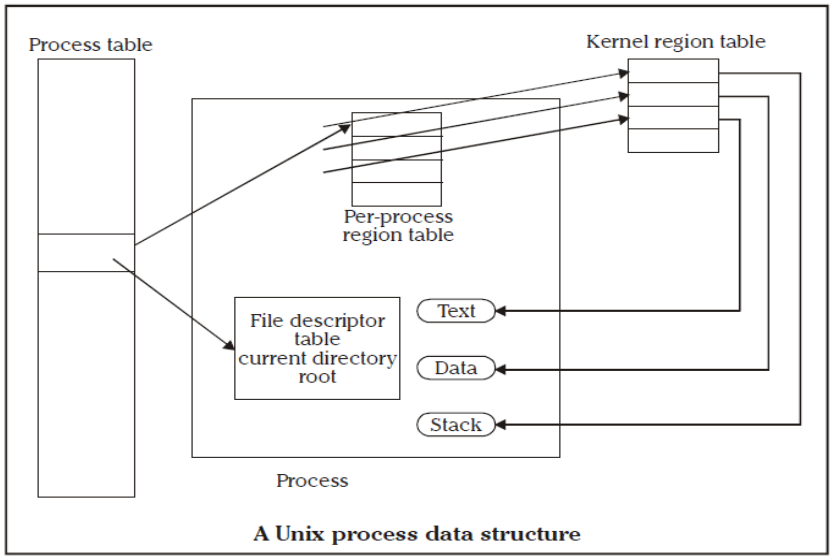
4.5 UNIX Kernel Support for Processes:

The data structure and execution of processes are dependent on operating system implementation.

A UNIX process consists minimally of a text segment, a data segment and a stack segment. A segment is an area of memory that is managed by the system as a unit.

- A text segment consists of the program text in machine executable instruction code format.
- The data segment contains static and global variables and their corresponding data.
- A stack segment contains runtime variables and the return addresses of all active functions for a process. UNIX kernel has a process table that keeps track of all active process present in the system. Some of these processes belongs to the kernel and are called as "system process". Every entry in the process table contains pointers to the text, data and the stack segments and also to U-area of a process. U-area of a process is an extension of the process table entry and contains other process

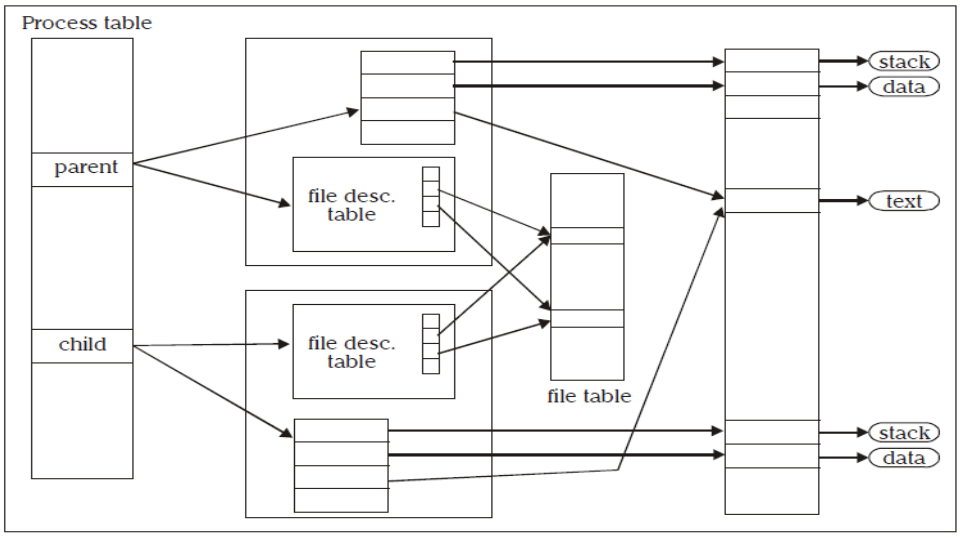
specific data such as the file descriptor table, current root and working directory inode numbers and set of system imposed process limits.



A Unix process data structure

All processes in UNIX system expect the process that is created by the system boot code, are created by the fork system call. After the fork system call, once the child process is created, both the parent and child processes resumes execution. When a process is created by fork, it contains duplicated copies of the text, data and stack segments of its parent as shown in the Figure below. Also it has a file descriptor table, which contains reference to the same opened files as the parent, such that they both share the same file pointer to each opened files.

Parent & child relationship after fork



The process will be assigned with attributes, which are either inherited from its parent or will be 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 that parent process.
- **An effective user identification number (eUID)**: this allows the process to access and create files with the same privileges as the program file owner.
- **An effective group identification number (eGID)**: this allows the process to access and create files with the same privileges as the group to which the program file belongs.
- **Saved set-UID and saved set-GID**: these are the assigned eUID and eGID of the process respectively.
- **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 to a root directory.
- **Signal handling**: the signal handling settings.
- **Signal mask**: a signal mask that specifies which signals are to be blocked.
- **Unmask**: 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 PID.
- **Pending signals**: the set of signals that are pending delivery to the parent 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 chid process.

fork and ***exec*** are commonly used together to spawn a sub-process to execute a different program. The advantages of this method are:

- A process can create multiple processes to execute multiple programs concurrently.
- Because each child process executes in its own virtual address space, the parent process is not affected by the execution status of its child process.