UNIT-III (PROCESS & SIGNALS)

1Q) Define a Process in UNIX or POSIX system?

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

2Q) List the ways for Termination of a Process?

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

3Q) Explain about the system calls which will be used for Normal termination of a process?

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.

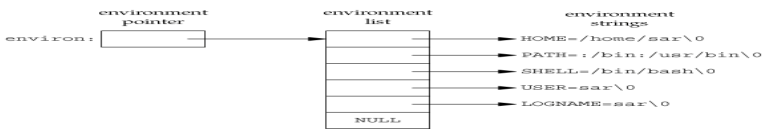
- any of these functions is called without an exit status.
- main does a return without a return value

4Q) Explain the 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`: Generally any environmental variable is of the form: ***name=value.***

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

Figure: Environment consisting of five C character strings



5Q) Explain the Memory Layout of a C Program in Memory?

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.

6Q) Explain about Process Environment- 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

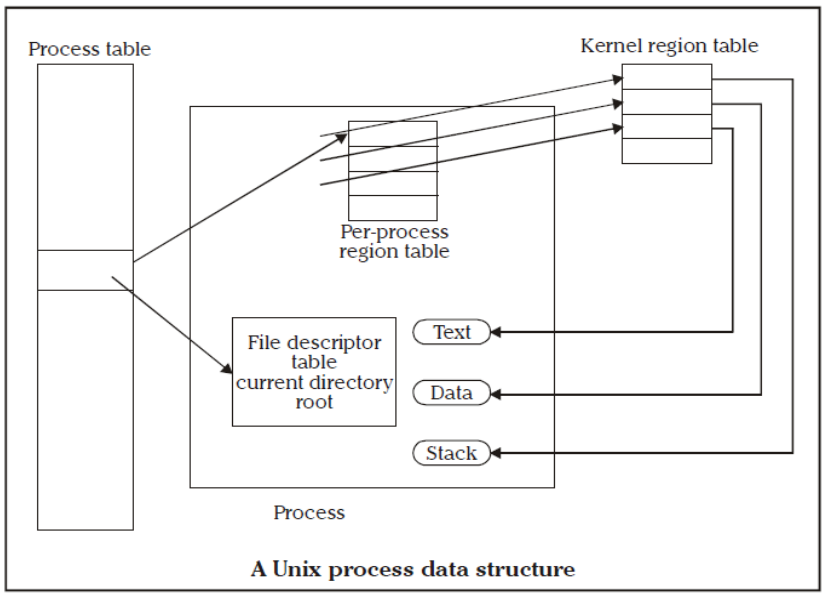
7Q) Explain the Kernel support for Process?

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.

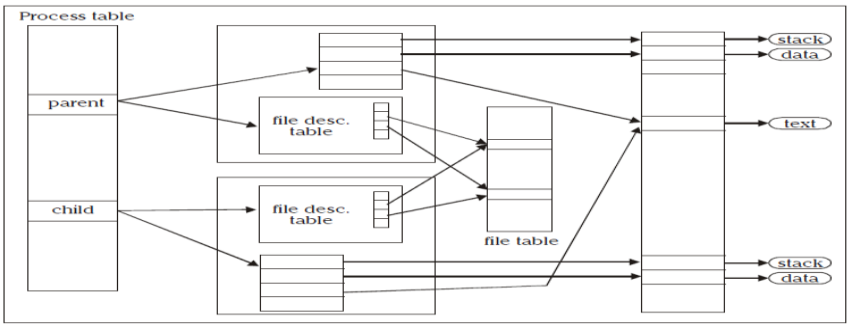


Figure: Parent & child relationship after fork

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.

8Q) Explain about fork () System call?

An existing process can create a new one by calling the `fork` function.

```
#include <unistd.h>

pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, 1 on error.

- The new process created by `fork` is called the child process.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- The reason `fork` returns 0 to the child is that a process can have only a single parent, and the child can always call `getppid` to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)
- Both the child and the parent continue executing with the instruction that follows the call to `fork`.
- The child is a copy of the parent.
- For example, the child gets a copy of the parent's data space, heap, and stack.
- Note that this is a copy for the child; the parent and the child do not share these portions of memory.
- The parent and the child share the text segment .

9Q) List the reasons for a fork () to fail?

The two main reasons for `fork` to fail are

(a) if too many processes are already in the system, which usually means that something else is wrong, or

(b) if the total number of processes for this real user ID exceeds the system's limit.

10Q) Explain the Uses of a fork?

There are two uses for `fork`:

- ❖ 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.
- ❖ 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`.

11Q) Explain about vfork ()?

- ✓ The function `vfork` has the same calling sequence and same return values as `fork`.

- ✓ 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.

12Q) Explain about wait and waitpid () System calls?

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the `SIGCHLD` signal to the parent. Because the termination of a child is an asynchronous event - it can happen at any time while the parent is running - this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler.

A process that calls `wait` or `waitpid` can:

- ✓ Block, if all of its children are still running
- ✓ Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
- ✓ Return immediately with an error, if it doesn't have any child processes.

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

pid_t wait(int *statloc);

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

Both return: process ID if OK, 0 (see later), or 1 on error. The differences between these two functions are as follows.

- The `wait` function can block the caller until a child process terminates, whereas `waitpid` has an option that prevents it from blocking.
- The `waitpid` function doesn't wait for the child that terminates first; it has a 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 caller until a child terminates. If the caller blocks and has multiple children, `wait` returns when one terminates.

For both functions, the argument statloc is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.

13Q) List the features of waitpid () that aren't provided by wait ()?

The `waitpid` function provides three features that aren't provided by the `wait` function they are:

- ✓ The `waitpid` function lets us wait for one particular process, whereas the `wait` function returns the status of any terminated child. We'll return to this feature when we discuss the `popen` function.
- ✓ The `waitpid` function provides a nonblocking version of `wait`. There are times when we want to fetch a child's status, but we don't want to block.
- ✓ The `waitpid` function provides support for job control with the `WUNTRACED` and `WCONTINUED` options.

14Q) Write about waitid () system call?

The `waitid` function is similar to `waitpid`, but provides extra flexibility.

```
#include <sys/wait.h>
Int waited(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Returns: 0 if OK, -1 on error

The *idtype* constants for waited are as follows:

P_PID	Wait for a particular process: id contains the process ID of the child to wait for.
P_PGID	Wait for any child process in a particular process group: id contains the process group ID of the children to wait for.
P_ALL	Wait for any child process: id is ignored.

The options argument is a bitwise OR of the flags as shown below: these flags indicate which state changes the caller is interested in.

WCONTINUED	Wait for a process that has previously stopped and has been continued, and whose status has not yet been reported.
WEXITED	Wait for processes that have exited.
WNOHANG	Return immediately instead of blocking if there is no child exit status available.
WNOWAIT	Don't destroy the child exit status. The child's exit status can be retrieved by a subsequent call to `wait`, `waitid`, or `waitpid`.
WSTOPPED	Wait for a process that has stopped and whose status has not yet been reported.

15Q) explain about wait3 () and wait4 () system calls?

The only feature provided by these two functions that isn't provided by the `wait`, `waitid`, and `waitpid` functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.

The prototypes of these functions are:

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

#include <sys/wait.h>

#include <sys/time.h>

#include <sys/resource.h>

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

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

Both return: process ID if OK,-1 on error

The resource information includes such statistics as the amount of user CPU time, the amount of system CPU time, number of page faults, number of signals received etc. the resource information is available only for terminated child process not for the process that were stopped due to job control.

16Q) write short notes on exec function family?

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 an `exec`, because a new process is not created; `exec` merely replaces the current process - its text, data, heap, and stack segments - with a brand new program from disk.

There are 6 exec functions:

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0,... /*
(char *)0 */ ); int execv(const char *pathname, char
*const argv []);

int execle(const char *pathname, const char *arg0,... /*(char *)0, char

*const envp */ );

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

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

- The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument. When a filename argument is specified
  - If filename contains a slash, it is taken as a pathname.
  - Otherwise, the executable file is searched for in the directories specified by the `PATH` environment variable.
- The next difference concerns the passing of the argument list (`l` stands for list and `v` stands for vector). The functions `execl`, `execlp`, and `execle` require each of the command-line arguments to the new program to be specified as separate arguments. For the other three functions (`execv`, `execvp`, and `execve`), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.
- The final difference is the passing of the environment list to the new program. The two functions whose names end in an `e` (`execle` and `execve`) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the `environ` variable in the calling process to copy the existing environment for the new program.

Function	pathname	filename	Arg list	argv[]	environ	envp[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(letter in name)		p	l	v		e

The above table shows the differences among the 6 exec functions.

17Q) Describe how to change userids and group ids in Linux Environment?

When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access. Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

```
#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. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

- If the process has superuser privileges, the `setuid` function sets the real user ID, effective user ID, and saved set-user-ID to uid.
- 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.
- If neither of these two conditions is true, `errno` is set to `EPERM`, and 1 is returned.

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

- Only a superuser process can change the real user ID. Normally, the real user ID is set by the `login`(1) program when we log in and never changes. Because `login` is a superuser process, it sets all three user IDs when it calls `setuid`.
- The effective user ID is set by the `exec` functions only if the set-user-ID bit is set for the 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 can't set the effective user ID to any random value.
- The saved set-user-ID is copied from the effective user ID by `exec`. If the file's set-user-ID bit is set, this copy is saved after `exec` stores the effective user ID from the file's user ID.

ID	exec		setuid(uid)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged

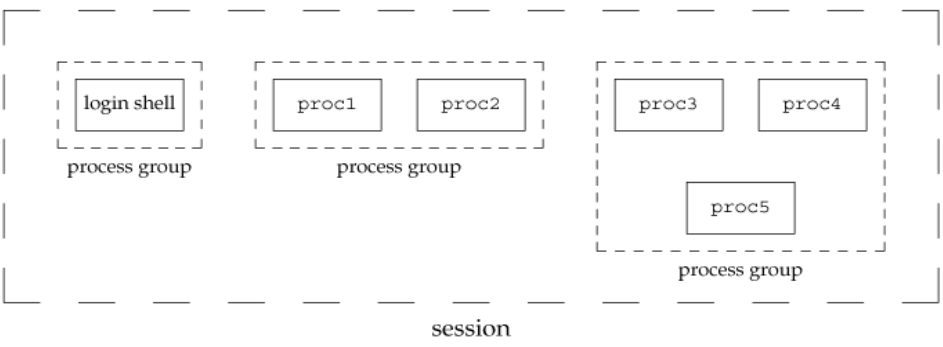
				user
real user ID	unchanged	unchanged	set to uid	unchanged
effective user ID	unchanged	set from user ID of program file	set to uid	set to uid
saved set-user	copied from effective user ID	copied from effective user ID	set to uid	unchanged

The above figure summarizes the various ways these three user IDs can be changed

18Q) Explain about process Session?

A session is a collection of one or more process groups. For example, we could have the arrangement shown in Figure Here we have three process groups in a single session.

Arrangement of processes into process groups and sessions



A process establishes a new session by calling the `setsid` function.

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

Returns: process group ID if OK, 1 on error

If the calling process is not a process group leader, this function creates a new session. Three things happen.

- The process becomes the session leader of this new session. (A session leader is the process that creates a session.) The process is the only process in this new session.
- The process becomes the process group leader of a new process group. The new process group ID is the process ID of the calling process.
- The process has no controlling terminal. If the process had a controlling terminal before calling `setsid`, that association is broken.

This function returns an error if the caller is already a process group leader. The `getsid` function returns the process group ID of a process's session leader. The `getsid` function is included as an XSI extension in the Single UNIX Specification.

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

```
pid_t getsid(pid_t pid);
```

Returns: session leader's process group ID if OK, 1 on error

If pid is 0, `getsid` returns the process group ID of the calling process's session leader.

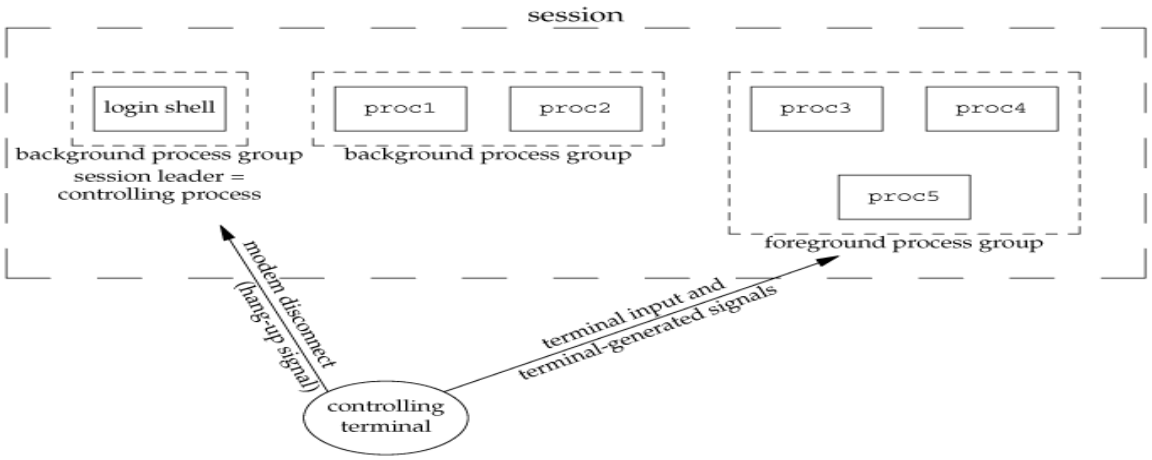
19Q) Explain about Process Control Terminal?

Sessions and process groups have a few other characteristics.

- A session can have a single controlling terminal. This is usually the terminal device (in the case of a terminal login) or pseudo-terminal device (in the case of a network login) on which we log in.
- The session leader that establishes the connection to the controlling terminal is called the controlling process.
- The process groups within a session can be divided into a single foreground process group and one or more background process groups.
- If a session has a controlling terminal, it has a single foreground process group, and all other process groups in the session are background process groups.
- Whenever we type the terminal's interrupt key (often DELETE or Control-C), this causes the interrupt signal be sent to all processes in the foreground process group.
- Whenever we type the terminal's quit key (often Control-backslash), this causes the quit signal to be sent to all processes in the foreground process group.
- If a modem (or network) disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (the session leader).

These characteristics are shown in Figure below

Process groups and sessions showing controlling terminal



20Q) Define a signal? List the different types of signals /description /and when they get triggered in Linux Environment?

Signals are software interrupts. Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

Different signals:

Name	Description	Default action
SIGABRT	abnormal termination (abort)	terminate+core
SIGALRM	timer expired (alarm)	terminate
SIGBUS	hardware fault	terminate+core
SIGCANCEL	threads library internal use	ignore
SIGCHLD	change in status of child	ignore
SIGCONT	continue stopped process	continue/ignore
SIGEMT	hardware fault	terminate+core
SIGFPE	arithmetic exception	terminate+core
SIGFREEZE	checkpoint freeze	ignore
SIGHUP	hangup	terminate
SIGILL	illegal instruction	terminate+core
SIGINFO	status request from keyboard	ignore
SIGINT	terminal interrupt character	terminate
SIGIO	asynchronous I/O	terminate/ignore
SIGIOT	hardware fault	terminate+core
SIGKILL	termination	terminate
SIGLWP	threads library internal use	ignore
SIGPIPE	write to pipe with no readers	terminate
SIGPOLL	pollable event (poll)	terminate
SIGPROF	profiling time alarm (setitimer)	terminate
SIGPWR	power fail/restart	terminate/ignore
SIGQUIT	terminal quit character	terminate+core
SIGSEGV	invalid memory reference	terminate+core
SIGSTKFLT	coprocessor stack fault	terminate
SIGSTOP	stop	stop process
SIGSYS	invalid system call	terminate+core
SIGTERM	termination	terminate
SIGTHAW	checkpoint thaw	ignore
SIGTRAP	hardware fault	terminate+core

SIGTSTP	terminal stop character	stop process
SIGTTIN	background read from control tty	stop process

SIGTTOU	background write to control tty	stop process
SIGURG	urgent condition (sockets)	ignore
SIGUSR1	user-defined signal	terminate
SIGUSR2	user-defined signal	terminate
SIGVTALRM	virtual time alarm (`setitimer`)	terminate
SIGWAITING	threads library internal use	ignore
SIGWINCH	terminal window size change	ignore
SIGXCPU	CPU limit exceeded (`setrlimit`)	terminate+core/ignore
SIGXFSZ	file size limit exceeded (`setrlimit`)	terminate+core/ignore
SIGXRES	resource control exceeded	Ignore

When a signal is sent to a process, it is pending on the process to handle it. The process can react to pending signals in one of three ways:

- Accept the **default action** of the signal, which for most signals will terminate the process.
- **Ignore** the signal. The signal will be discarded and it has no effect whatsoever on the recipient process.
- Invoke a **user-defined** function. The function is known as a signal handler routine and the signal is said to be ***caught*** when this function is called.

21 Q) Explain the Kernel support of signals?

- When a signal is generated for a process, the kernel will set the corresponding signal flag in the process table slot of the recipient process.
- If the recipient process is asleep, the kernel will awaken the process by scheduling it.
- When the recipient process runs, the kernel will check the process U-area that contains an array of signal handling specifications.
- If array entry contains a zero value, the process will accept the default action of the signal.
- If array entry contains a 1 value, the process will ignore the signal and kernel will discard it.
- If array entry contains any other value, it is used as the function pointer for a user-defined signal handler routine.

22 Q) Explain about signal () system call?

The function prototype of the signal API is: `#include <signal.h>`

```
void (*signal(int sig_no, void (*handler)(int)))(int);
```

The formal argument of the API are: sig_no is a signal identifier like SIGINT or SIGTERM. The handler argument is the function pointer of a user-defined signal handler function.

The following example attempts to catch the SIGTERM signal, ignores the SIGINT signal, and accepts the default action of the SIGSEGV signal. The pause API suspends the calling process until it is interrupted by a signal and the corresponding signal handler does a return:

```
#include<iostream.h>

#include<signal.h>

/*signal
handler
function*/
void
catch_sig(in
t sig_num)

{

      signal (sig_num,catch_sig);

      cout<<"catch_sig:"<<sig_num<<endl;

}


/*main function*/ int
main()

{

      signal(SIGTERM,catch_s
      ig);
      signal(SIGINT,SIG_IGN
      );
      signal(SIGSEGV,SIG_DF
      L);

      pause( );               /*wait for a signal interruption*/

}
```

The SIG_IGN specifies a signal is to be ignored, which means that if the signal is generated to the process, it will be discarded without any interruption of the process.The SIG_DFL specifies to accept the default action of a signal.

23 Q) Explain about the Signal Mask?

A process initially inherits the parent's signal mask when it is created, but any pending signals for the parent process are not passed on. A process may query or set its signal mask via the sigprocmask API:

```
#include <signal.h>

int sigprocmask(int cmd, const sigset_t *new_mask, sigset_t *old_mask);
```

Returns: 0 if OK, 1 on error

The new_mask argument defines a set of signals to be set or reset in a calling process signal mask, and the cmd argument specifies how the new_mask value is to be used by the API. The possible values of cmd and the corresponding use of the new_mask value are:

SIG_SETMASK Overrides the calling process signal mask with the value specified in the new_mask argument.

SIG_BLOCK Adds the signals specified in the new_mask argument to the calling process signal mask.

SIG_UNBLOCK Removes the signals specified in the new_mask argument from the calling process signal mask.

- ✓ If the actual argument to new_mask argument is a NULL pointer, the cmd argument will be ignored, and the current process signal mask will not be altered.
- ✓ If the actual argument to old_mask is a NULL pointer, no previous signal mask will be returned.
- ✓ The sigset_t contains a collection of bit flags.

The BSD UNIX and POSIX.1 define a set of API known as sigsetops functions:

```
#include<signal.h>

int sigemptyset (sigset_t* sigmask);
int sigaddset(sigset_t* sigmask, const int sig_num);
int sigdelset(sigset_t* sigmask, const
int sig_num);
int sigfillset(sigset_t* sigmask);

int sigismember (const sigset_t* sigmask, const int sig_num);
```

The sigemptyset API clears all signal flags in the sigmask argument.

- ▶ The sigaddset API sets the flag corresponding to the signal_num signal in the sigmask argument.
- ▶ The sigdelset API clears the flag corresponding to the signal_num signal in the sigmask argument.
- ▶ The sigfillset API sets all the signal flags in the sigmask argument.

[all the above functions return 0 if OK, -1 on error]

- ▶ The sigismember API returns 1 if flag is set, 0 if not set and -1 if the call fails.

The following example checks whether the SIGINT signal is present in a process signal mask and adds it to the mask if it is not there.

```
#include<stdio.h>

#include<signal.h> int main()

{

        sigset_t      sigmask;

        sigemptyset(&sigmask);            /*initialise set*/


        if(sigprocmask(0,0,&sigmask)==-1)        /*get current signal mask*/

        {

               perror("sigprocmask"); exit(1);

        }

        else sigaddset(&sigmask,SIGINT); /*set SIGINT flag*/


        sigdelset(&sigmask, SIGSEGV);
                                                 /*cle
        ar SIGSEGV flag*/
        if(sigprocmask(SIG_SETMASK,&sigmask,0)==-1)

               perror("sigprocmask");

}
```

A process can query which signals are pending for it via the sigpending API:

```
#include<signal.h>

int sigpending(sigset_t* sigmask);
```

Returns 0 if OK, -1 if fails.

The sigpending API can be useful to find out whether one or more signals are pending for a process and to set up special signal handling methods for these signals before the process calls the sigprocmask API to unblock them.

The following example reports to the console whether the SIGTERM signal is pending for the process:

```
#include<iostream.h>

#include<stdio.h>

#include<signal.h> int main()

{

        sigset_t
```

```
        sigmask; sigemptyset(&igmask);
        if(sigpending(&sigmask)==-1)
              perror("sigpending");
        else cout << "SIGTERM signal is:"
              << (sigismember(&sigmask,SIGTERM) ? "Set" : "No Set")  <<
              endl;
}
```

In addition to the above, UNIX also supports following APIs for signal mask manipulation:

```
#include<signal.h>


int sighold(int signal_num);
int sigrelse(int signal_num);
int sigignore(int signal_num);
```

24 Q) Explain About SIGACTION API?

The sigaction API blocks the signal it is catching allowing a process to specify additional signals to be blocked when the API is handling a signal.

The sigaction API prototype is:

```
#include<signal.h>

int sigaction(int signal_num, struct sigaction* action, struct
sigaction* old_action);
```

Returns: 0 if OK, 1 on error

The struct sigaction data type is defined in the <signal.h> header as:

```
struct sigaction

{

      void          (*sa_handler)(int);

      sigset_t      sa_mask;

      int           sa_flag;

}
```

The following program illustrates the uses of sigaction:

```
#include<iostream.h>

#include<stdio.h>

#include<unistd.h>

#include<signal.h>


void callme(int sig_num)
```

```
{
        cout<<"catch signal:"<<sig_num<<endl;
}
int main(int argc, char* argv[])
{
        sigset_t sigmask;
        struct sigaction
        action,old_action
        ;
        sigemptyset(&sigm
        ask);
        if(sigaddset(&sigmask,SIGTERM)==-1 ||
                sigprocmask(SIG_SETMASK,&sigmask,0)==-1) perror("set
                signal mask");
        sigemptyset(&actio
        n.sa_mask);
        sigaddset(&action.
        sa_mask,SIGSEGV);
        action.sa_handler=
        callme;
        action.sa_flags=0;
        if(sigaction(SIGINT,&action,
                &old_action)==-1)
                perror("sigaction");
        pause(); cout<<argv[0]<<"exists\n"; return 0;
}
```

25Q) Explain about Kill () system call?

A process can send a signal to a related process via the kill API. This is a simple means of inter-process communication or control. The function prototype of the API is:

```
#include<signal.h>
int kill(pid_t pid, int signal_num);
```

Returns: 0 on success, -1 on failure.

The signal_num argument is the integer value of a signal to be sent to one or more processes designated by pid. The possible values of pid and its use by the kill API are:

pid > 0 The signal is sent to the process whose process ID is pid.

pid == 0 The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.

pid < 0 The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal.

pid == 1 The signal is sent to all processes on the system for which the sender has permission to send the signal.

26Q) write about alarm () system call?

The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds. The function prototype of the API is:

```
#include<signal.h>

Unsigned int alarm(unsigned int time_interval);
```

Returns: 0 or number of seconds until

previously set alarm The alarm API can

be used to implement the sleep API:

27Q) Explain about a Daemon Process and its characteristics?

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down.

DAEMON CHARACTERISTICS

The characteristics of daemons are:

- Daemons run in background.
- Daemons have super-user privilege.
- Daemons don't have controlling terminal.
- Daemons are session and group leaders.

28Q) Explain about Zombie Process?

Zombie process. On UNIX and Unix-like computer operating systems, a zombie process or defunct process is a process that has completed execution (via the exit system call) but still has an entry in the process table: it is a process in the "Terminated state".

29 Q) Explain about Orphan Process?

An **orphan process** is a computer **process** whose parent **process** has finished or terminated, though it remains running itself. In a Unix-like operating system any **orphaned process** will be immediately adopted by the special init system **process**.

****30 Q) Distinguish Zombie and Orphan Processes?****