SYSTEM CALLS

fork()

is used to create processes.

- do not take any arguments and returns a process ID (mostly an integer value).
- creates a new process (child process) that runs concurrently with the parent process (the process that makes the fork() call).

Syntax:

pid_t fork(void);

returns the following values:

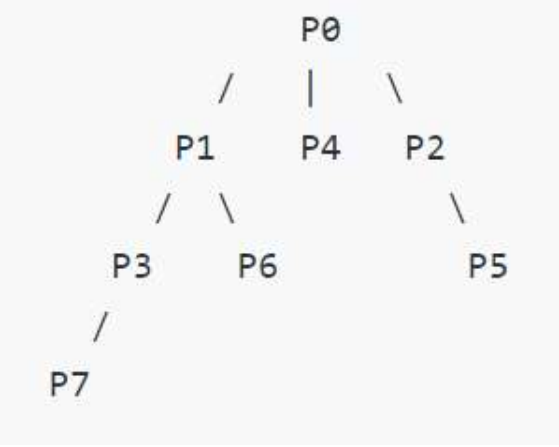
- **Negative value -** it represents the creation of the child process was unsuccessful.
- **Zero -** it represents a new child process is created.
- **Positive value -** The process ID of the child process to the parent. The returned process ID is type pid_t defined in sys/types.h. Usually, the process ID is an integer.

The following header files are included when we use fork() in C.

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

- fork() is defined in <unistd.h> header file, so we need to include this header file to use a fork.
- Type pid_t is defined in <sys/types.h> and process ID's are of pid_t type.

```
fork ();

fork ();

fork ();
```



Total Number of Processes = 2^n,

where n is number of fork system calls.

So here n = 3, $2^3 = 8$

Eg :

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
   // child process because return value zero
   if (fork() == 0)
      printf("Hello from Child!\n");
   // parent process because return value non-zero.
   else
      printf("Hello from Parent!\n");
}
int main()
{
   forkexample();
   return 0;
}
```

Output:

1.

Hello from Child!

Hello from Parent!

(or)

2.

Hello from Parent!

Hello from Child!

NOTE : Since the child and parent processes run concurrently, these processes reside in different memory spaces. i.e., it does not mean they are identical. OS allocate different data and states for these two processes, and the control flow of these processes can be different.
Eg:

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

void forkexample()
{
   int x = 1;
   if (fork() == 0)
      printf("Child has x = %d\n", ++x);
```

```
    else
        printf("Parent has x = %d\n", --x);
}
int main()
{
    forkexample();
    return 0;
}
```

Output:

Parent has x = 0

Child has x = 2

(or)

Child has x = 2

Parent has x = 0

Here, global variable change in one process does not affected two other processes because data/state of two processes are different. And also parent and child run simultaneously so two outputs are possible.

wait()

You can control the execution of child processes by calling wait() in the parent. wait() forces the parent to suspend execution until the child is finished.

- wait() returns the process ID of a child process that finished.
- If the child finishes before the parent gets around to calling wait(), then when wait() is called by the parent, it will return immediately with the child's process ID.

prototype for the wait() system call is:

int wait(status)

int *status;

status : pointer to an integer where the UNIX system stores the value returned by the child process.

wait() fails if any of the following conditions hold:

- The process has no children to wait for.
- status points to an invalid address.

***NOTE** : When wait() returns they also define **exit status** (which tells our, a process why terminated) via pointer, If status are not **NULL**.*
If any process has no child process then wait() returns immediately "-1".

The format of the information returned by wait() is as follows:

- If the process ended by calling the exit() system call, the second lowest byte of status is set to the argument given to exit() and the lowest byte of status is set to zeroes.
- If the process ended because of a signal, the second lowest byte of status is set to zeroes and the lowest byte of status contains the signal number that ended the process.

```
int main()
{
   pid_t cpid;
   if (fork()== 0)
      exit(0);         /* terminate child */
   else
      cpid = wait(NULL); /* reaping parent */
   printf("Parent pid = %d\n", getpid());
   printf("Child pid = %d\n", cpid);

   return 0;
}
```

Output:

Parent pid = 12345678

Child pid = 89546848

if more than one child processes are terminated, then wait() reaps any arbitrarily child process but if we want to reap any specific child process, we use ***waitpid()*** function.

pid_t waitpid (child_pid, &status, options);

Options Parameter

- If 0 means no option parent has to wait for terminates child.
- If **WNOHANG** --- parent does not wait if child does not terminate just check and return waitpid().(not block parent process)
- If child_pid is **-1** then means any ***arbitrarily child***, here waitpid() work same as wait() work.

Return value of waitpid()

- pid of child, if child has exited
- 0, if using WNOHANG and child hasn't exited.

exit()

it ends a process and returns a value to its parent.

void exit(status)

int status;

where status is an integer between 0 and 255. This number is returned to the parent via wait() as the exit status of the process.

- a status value of 0 or EXIT_SUCCESS indicates success, and any other value or the constant EXIT_FAILURE is used to indicate an error.
- exit() is actually not a system routine; it is a library routine that call the system routine _exit().
- exit() cleans up the standard I/O streams before calling _exit(), so any output that has been buffered but not yet actually written out is flushed.
- Calling _exit() instead of exit() will bypass this cleanup procedure.
- exit() does not return.

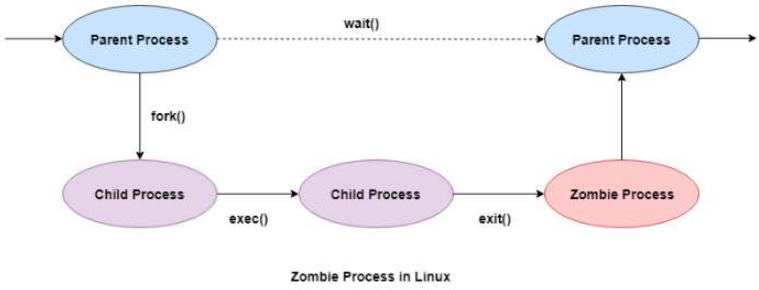
```
//demonstrates exit() returning a status to wait().

int main()

{

unsigned int status;

if ( fork () == 0 ) { /* == 0 means in child */

scanf ("%d", &status);

exit (status);

}

else { /* != 0 means in parent */

wait (&status);

printf("child exit status = %d\n", status > 8);

}

}
```

Note: since wait() returns the exit status multiplied by 256 (contained in the upper 8 bits), the status value is shifted right 8 bits (divided by 256) to obtain the correct value.

Zombie and Orphan Processes

Zombie Process:

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.



Zombie Process in Linux

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        pid_t child_pid = fork();
        if (child_pid > 0)
                sleep(50);
        else
                exit(0);
        return 0;
}
```

Orphan Process:

- A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.
- orphan process is soon adopted by init process, once its parent process dies.

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

```
#include <unistd.h>
int main()
{
        int pid = fork();
        if (pid > 0)
                printf("in parent process");
        else if (pid == 0)
        {
                sleep(30);
                printf("in child process");
        }
        return 0;
}
```

getppid() and getpid()

Both getppid() and getpid() are inbuilt functions defined in **unistd.h** library.

getppid() :

- returns the process ID of the parent of the calling process.
- this function returns a value of 1 which is the process id for **init** process.

Syntax:
pid_t getppid(void);
It never throws any error therefore is always successful.

getpid() :
returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames.
Syntax:
pid_t getpid(void);
It never throws any error therefore is always successful.

```
#include <iostream>
#include <unistd.h>
using namespace std;
int main()
{
   int pid = fork();
   if (pid == 0)
      cout << "\nCurrent process id of Process : "
         << getpid() << endl;
```

```
    return 0;
}
```

Output (Will be different on different systems):

Current process id of Process : 4195

getuid(): function returns the real user ID of the calling process. The real user ID identifies the person who is logged in.

```
uid_t getuid(void);
```

getsid() :

```
#include <unistd.h>

pid_t getsid(pid_t pid);
```

obtains the process group ID of the process that is the session leader of the process specified by pid. If pid is (pid_t) 0, it specifies the calling process.

Upon successful completion, getsid() returns the process group ID of the session leader of the specified process. Otherwise, it returns (pid_t)−1 and sets errno to indicate the error.

access ()

access command is used to check whether the calling program has access to a specified file. It can be used to check whether a file exists or not. The check is done using the calling process's real UID and GID.

```
int access(const char *pathname, int mode);
```

the first argument → path to the *directory/file*
second argument → flags *R_OK, W_OK, X_OK or F_OK*.

- **F_OK flag** : Used to check for existence of file.
- **R_OK flag** : Used to check for read permission bit.
- **W_OK flag** : Used to check for write permission bit.
- **X_OK flag** : Used to check for execute permission bit.

Note: If access() cannot access the file, it will return -1 or else it will be 0.

Check for all permission bits (read, write, execute)

```c
#include<stdio.h>
#include<unistd.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
extern int errno;

int main(int argc, const char *argv[]){
   int fd = access("sample.txt", (R_OK | W_OK) & X_OK);
   if(fd == -1){
      printf("Error Number : %d\n", errno);
      perror("Error Description:");
   }
   else{
      printf("No error\n");
   }
   return 0;
}
```

Check for all permission bits (read, write, execute) to demonstrate how the code functions, when we get an error.

```c
#include<stdio.h>
#include<unistd.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
extern int errno;
int main(int argc, const char *argv[]){
   int fd = access("sample.txt", R_OK & W_OK & X_OK);
   if(fd == -1){
      printf("Error Number : %d\n", errno);
      perror("Error Description:");
   }
   else{
      printf("No error\n");
   }
   return 0;
}
```

Here, fd = -1 and we get the error message for the reason of failure of the calling process.

alarm()

Every process has an alarm clock stored in its system-data segment. When thealarm goes off, signal SIGALRM is sent to

the calling process. A child inherits its parent's alarm clock value, but the actual clock isn't shared. The alarm clock remains set across an exec.

The prototype for alarm() is:

unsigned int alarm(seconds)

unsigned int seconds;

where seconds defines the time after which the UNIX system sends the SIGALRM signal to the calling process.

Each successive call to alarm() nullifies the previous call .

Returns → the number of seconds until that alarm would have gone off. If seconds has the value 0, the alarm is canceled.

alarm() has no error conditions.

The following is an example program that demonstrates the use of the signal()
and alarm() system calls:

```
/* timesup.c */
#include <stdio.h>
#include <sys/signal.h>
#define EVER ;;
void main();
int times_up();
void main()
{
signal (SIGALRM, times_up);    /* go to the times_up function */
/* when the alarm goes off. */
alarm (10);                    /* set the alarm for 10 seconds */
for (EVER)                     /* endless loop. */
; /* hope the alarm works. */
}
int times_up(sig)
int sig; /* value of signal */
{
printf("Caught signal #< %d >n", sig);
printf("Time's up! I'm outta here!!\n");
exit(sig); /* return the signal number */
}
```

Close()

Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

Syntax in C language

```
#include <fcntl.h>
int close(int fd);
```

Parameter:

- **fd :**file descriptor

Return:

- **0** on success.

- **-1** on error.

How it works in the OS

- Destroy file table entry referenced by element fd of file descriptor table – As long as no other process is pointing to it!
- Set element fd of file descriptor table to **NULL**

- C

```c
// C program to illustrate close system Call
#include<stdio.h>
#include <fcntl.h>
int main()
{
   int fd1 = open("foo.txt", O_RDONLY);
   if (fd1 < 0)
   {
      perror("c1");
      exit(1);
   }
   printf("opened the fd = % d\n", fd1);

   // Using close system Call
   if (close(fd1) < 0)
   {
      perror("c1");
      exit(1);
   }
   printf("closed the fd.\n");
}
```

Output:

opened the fd = 3

closed the fd.

Read()

From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.

Syntax in C language

size_t read (int fd, void* buf, size_t cnt);

Parameters:

- **fd:** file descriptor
- **buf:** buffer to read data from
- **cnt:** length of buffer

Returns: How many bytes were actually read

- return Number of bytes read on success
- return 0 on reaching end of file

- return -1 on error
- return -1 on signal interrupt

Important points

- **buf** needs to point to a valid memory location with length not smaller than the specified size because of overflow.
- **fd** should be a valid file descriptor returned from open() to perform read operation because if fd is NULL then read should generate error.
- **cnt** is the requested number of bytes read, while the return value is the actual number of bytes read. Also, some times read system call should read less bytes than cnt.

```
// C program to illustrate
// read system Call
#include<stdio.h>
#include <fcntl.h>
int main()
{
int fd, sz;
char *c = (char *) calloc(100, sizeof(char));

fd = open("foo.txt", O_RDONLY);
if (fd < 0) { perror("r1"); exit(1); }

sz = read(fd, c, 10);
printf("called read(% d, c, 10). returned that"
      " %d bytes were read.\n", fd, sz);
c[sz] = '\0';
printf("Those bytes are as follows: % s\n", c);
}
```

Output:

called read(3, c, 10). returned that 10 bytes were read.

Those bytes are as follows: 0 0 0 foo.

Write()

Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

```
#include <fcntl.h>

size_t write (int fd, void* buf, size_t cnt);
```

Parameters:

- **fd:** file descriptor
- **buf:** buffer to write data to
- **cnt:** length of buffer

Returns: How many bytes were actually written

- return Number of bytes written on success

- return 0 on reaching end of file
- return -1 on error
- return -1 on signal interrupt

Important points

- The file needs to be opened for write operations
- **buf** needs to be at least as long as specified by cnt because if buf size less than the cnt then buf will lead to the overflow condition.
- **cnt** is the requested number of bytes to write, while the return value is the actual number of bytes written. This happens when **fd** have a less number of bytes to write than cnt.
- If write() is interrupted by a signal, the effect is one of the following:
  -If write() has not written any data yet, it returns -1 and sets errno to EINTR.
  -If write() has successfully written some data, it returns the number of bytes it wrote before it was interrupted.

```
// C program to illustrate
// write system Call
#include<stdio.h>
#include <fcntl.h>
main()
{
int sz;
int fd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd < 0)
{
   perror("r1");
   exit(1);
}
sz = write(fd, "hello geeks\n", strlen("hello geeks\n"));
printf("called write(% d, \"hello geeks\\n\", %d)."
   " It returned %d\n", fd, strlen("hello geeks\n"), sz);
close(fd);
}
```

Output:

called write(3, "hello geeks\n", 12). it returned 11

sleep()

```
unsigned int sleep(unsigned int seconds);
```

it causes the calling thread to sleep either until the number of real-time seconds specified in *seconds* have elapsed or until a signal arrives which is not ignored.

RETURN VALUE → Zero if the requested time has elapsed, or the number of seconds left to sleep, if the call was interrupted by a signal handler.

dup()

it creates a copy of a file descriptor.

- It uses the lowest-numbered unused descriptor for the new descriptor.
- If the copy is successfully created, then the original and copy file descriptors may be used interchangeably.
- They both refer to the same open file description and thus share file offset and file status flags.

Syntax:
int dup(int oldfd);
oldfd: old file descriptor whose copy is to be created.

```
// CPP program to illustrate dup()
#include<stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
   // open() returns a file descriptor file_desc to a
   // the file "dup.txt" here"
   int file_desc = open("dup.txt", O_WRONLY | O_APPEND);
   if(file_desc < 0)
      printf("Error opening the file\n");

   // dup() will create the copy of file_desc as the copy_desc
   // then both can be used interchangeably.

   int copy_desc = dup(file_desc);

   // write() will write the given string into the file
   // referred by the file descriptors
   write(copy_desc,"This will be output to the file named dup.txt\n", 46);
   write(file_desc,"This will also be output to the file named dup.txt\n", 51);
   return 0;
}
```

Explanation: The open() returns a file descriptor file_desc to the file named “dup.txt”. file_desc can be used to do some file operation with file “dup.txt”. After using the dup() system call, a copy of file_desc is created copy_desc. This copy can also be used to do some file operation with the same file “dup.txt”. After two write operations one with file_desc and another with copy_desc, same file is edited i.e. “dup.txt”.

Nice()

```
#include <unistd.h>
int nice(int incr);
```

- it allows a process to change its priority. The invoking process must be in a

scheduling class that supports the nice().

- It adds the value of incr to the nice value of the calling process. A process's nice value is a non-negative number for which a greater positive value results in lower CPU priority.

- A maximum nice value of (2 * NZERO) −1 and a minimum nice value of 0 are imposed by the system. NZERO is defined in <limits.h> with a default value of 20. Requests for values above or below these limits result in the nice value being set to the corresponding limit. A nice value of 40 is treated as 39.

- Calling the nice() function has no effect on the priority of processes or threads with policy SCHED_FIFO or SCHED_RR.

- Only a process with the {PRIV_PROC_PRIOCNTL} privilege can lower the nice val

Return : Upon successful completion →the new nice value minus NZERO. Otherwise, −1 is returned, the process's nice value is not changed, and errno is set to indicate the error.

The nice() function will fail if:

EINVAL

The nice() function is called by a process in a scheduling class other than time-sharing or fixed-priority.

EPERM

The incr argument is negative or greater than 40 and the {PRIV_PROC_PRIOCNTL} privilege is not asserted in the effective set of the calling process.

link()

```
#include <unistd.h>
int link(const char *path1, const char *path2);
```

creates a new link (directory entry) for the existing file and increments its link count by one.

path1→ points to a path name naming an existing file.

path2 → points to a pathname naming the new directory entry to be created.

- Upon successful completion, link() marks for update the st_ctime field of the file. Also, the st_ctime and st_mtime fields of the directory that contains the new entry are marked for update.

- If link() fails, no link is created and the link count of the file remains unchanged.

- Upon successful completion, 0 is returned. Otherwise, −1 is returned, no link is created, and errno is set to indicate the error.

Chdir()

```
#include <unistd.h>
int chdir(const char *path);
```

The chdir()cause a directory pointed to by path to become the current working directory.

The starting point for path searches for path names not beginning with / (slash).
The path argument points to the path name of a directory.

For a directory to become the current directory, a process must have execute (search) access to the directory.

Return Values

Upon successful completion, 0 is returned. Otherwise, −1 is returned, the current working directory is unchanged, and errno is set to indicate the error.

Unlink()

```
#include <unistd.h>
int unlink(const char *path);
```

it deletes a name from the filesystem.

3 cases :

- If that name was the last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse.
- If the name was the last link to a file but any processes still have the file open the file will remain in existence until the last file descriptor referring to it is closed.
- If the name referred to a symbolic link the link is removed. If the name referred to a socket, fifo or device the name for it is removed but processes which have the object open may continue to use it.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.