System Calls - Syntax

fork()

- The fork system call is used for creating a new process in Linux, and Unix systems, which is called the ***child process***, which runs concurrently with the process that makes the fork() call (parent process).
- After a new child process is created, both processes will execute the next instruction following the fork() system call.
- The child process uses the same pc(program counter), same CPU registers, and same open files which are used in the parent process.
- It takes no parameters and returns an integer value.

- Below are different values returned by fork().
- ***Negative Value***: The creation of a child process was unsuccessful.
- ***Zero***: Returned to the newly created child process.
- ***Positive value***: Returned to parent or caller. The value contains the process ID of the newly created child process.

```c
#include <stdio.h>
#include <sys/types.h>;
#include <unistd.h>;
int main()
{

    // make two process which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

Output

Hello world

Hello world

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("Hello\n");
    return 0;
}
```

Output:-

Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello

The number of times ‘hello’ is printed is equal to the number of processes created. Total Number of Processes = 2^n, where n is the number of fork system calls. So here n = 3, $2^3 = 8$

exec() system call

- The exec family of functions replaces the current running process with a new process.
- It can be used to run a C program by using another C program.
- It comes under the header file **unistd.h.**
- There are many members in the exec family

- **execvp** : Using this command, the created child process does not have to run the same program as the parent process does.
- The **exec** type system calls allow a process to run any program files, which include a binary executable or a shell script .

```
int execvp (const *file, char *const argv[]);
```

- **file:** points to the file name associated with the file being executed.
  argv: is a null terminated array of character pointers.

We will have two .C files , **program1.c** and **program2.c**

And we will replace the **program1.c** with **program2.c** by calling execvp() function in program1.c .

//program2.c

```
#include<stdio.h>
#include<unistd.h>
int main()
{   int i;
    printf("I am program2.c called by execvp() ");
    printf("\n");
    return 0;
}
```

Compile the program and create an executable file program2 using

```
gcc program2.c -o program2
```

//program1.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{   //A null terminated array of character pointers
     char *args[]={"./program2",NULL};
     execvp(args[0],args);
/*All statements are ignored after execvp() call as this whole
process(program1.c) is replaced by another process (program2.c)
*/
printf("Ending-----");
return 0;
 }
```

Now create an executable file using gcc as

gcc program1.c –o program1

Now run it as

./program1

Output of the program

I am program2.c called by execvp()

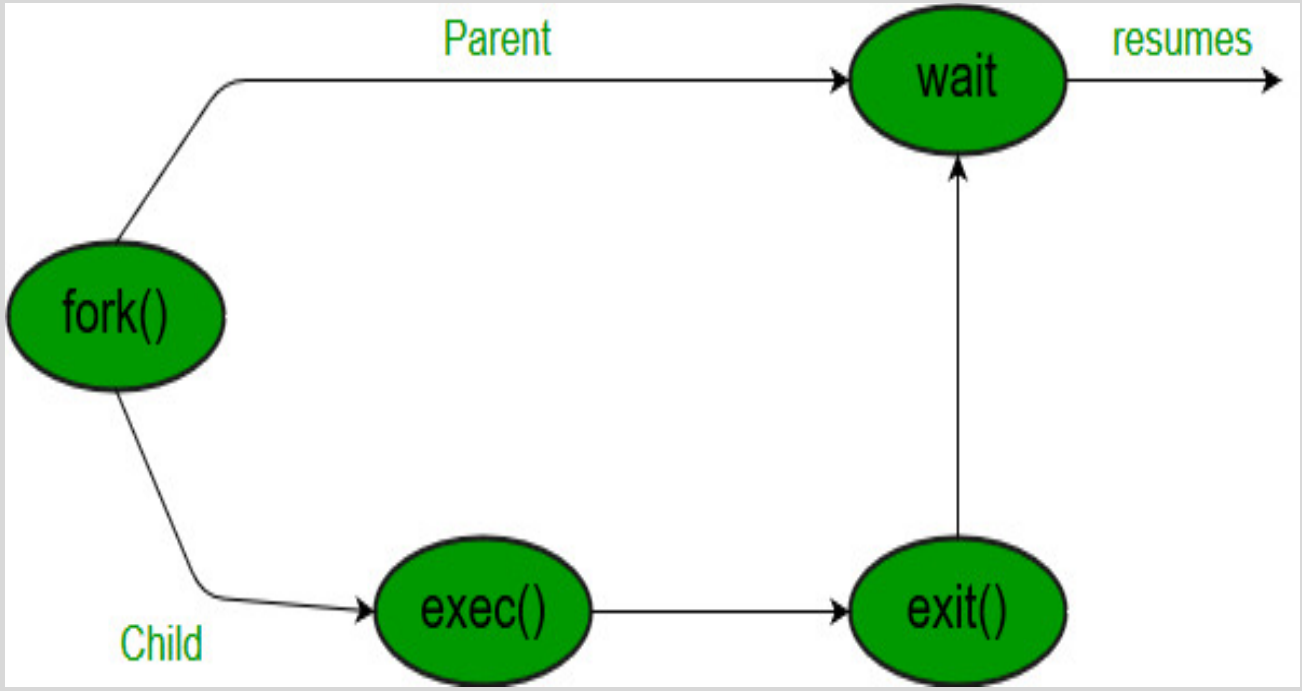
getpid() and getppid()

```
// C Code to demonstrate getpid() and getppid()
#include<stdio.h>
#include <unistd.h>
int main()
{int pid;
pid = fork();
if (pid == 0)
{    printf( "\n Process id : %d “ , getpid() );
     printf("\n Process id : %d “ , getppid());
}
return 0;
}
```

```
#include<stdio.h>
#include<stdlib.h>
 #include<unistd.h>
 #include<sys/types.h>
 int main(int argc, char **argv)
 { pid_t pid;
   pid = fork();
   if(pid==0)
    { printf("It is the child process and pid is %d\n",getpid());
      exit(0); }
   else if(pid > 0)
    { printf("It is the parent process and pid is %d\n",getpid()); }
    else
    { printf("Error while forking\n");
    exit(EXIT_FAILURE); }
    return 0; }
```

wait() system call

- A call to wait() blocks the calling process until one of its child processes exits or a signal is received.
- After child process terminates, parent ***continues*** its execution after wait system call instruction.



Syntax of wait()

```
// take one argument status
// returns the process ID of dead child.
pid_t wait(int *stat_loc);
```

wait takes argument NULL. There are also other means of arguments to wait, which is out of scope here.

```
// C program to demonstrate working of wait()
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{    pid_t cpid;
     if (fork()== 0)
          exit(0);        /* terminate child */
     else
          cpid = wait(NULL); /* waiting parent */
     printf("Parent pid = %d\n", getpid());
     printf("Child pid = %d\n", cpid);
     return 0;
 }
```

RSET
RAJAGIRI SCHOOL OF ENGINEERING & TECHNOLOGY (AUTONOMOUS)

open() system call

Syntax

int open (path, flag)

Example:-

int fd1;

fd1= open(“test.txt”, O_RDONLY)

O_RDONLY stands for Read Only mode.

Return value

Returns the file descriptor , -1 upon failure

Close() system call

Closes a file

Syntax

int close(fp)

It takes the file pointer as input, return 0 on success, returns -1 if error.

// C program to illustrate open , close system Call

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
int main()
{     int fd1 = open(“test.txt", O_RDONLY);
      if (fd1 < 0) {
            printf(“Error\n “);
            exit(1); }
      printf("opened the fd = % d\n", fd1);
      // Using close system Call
      if (close(fd1) < 0) {
            printf(“Error\n”);
            exit(1);
      }
      printf("closed the fd.\n");
}
```

stat() system call

- Returns information regarding a file

```
struct stat
 {
  dev_t st_dev;          /* ID of device containing file */
 ino_t st_ino;           /* inode number */
mode_t st_mode;          /* protection */
nlink_t st_nlink;        /* number of hard links */
uid_t st_uid;            /* user ID of owner */
gid_t st_gid;            /* group ID of owner */
dev_t st_rdev;           /* device ID (if special file) */
off_t st_size;           /* total size, in bytes */
blksize_t st_blksize;    /* blocksize for filesystem I/O */
blkcnt_t st_blocks;      /* number of blocks allocated */
time_t st_atime;         /* time of last access */
time_t st_mtime;         /* time of last modification */
time_t st_ctime;         /* time of last status change */ };
```

```
#include<sys/stat.h>
#include<time.h>
#include<stdint.h>
int main()
{
  struct stat sfile;
  stat("even.c", &sfile);

  printf("Inode Number = %ld\n", sfile.st_ino);
  printf("Size = %ld\n",   .....
```

The dirent structure is defined as:

```
#include<sys/dirent.h>
  struct dirent
  {  long d_ino; /* inode number */
     off_t d_off; /* offset to the next dirent */
     unsigned short d_reclen; /* length of this record */
    unsigned char d_type; /* type of file */
     char d_name[256];  /* filename*/    };
```

d_ino ….. inode number of a file

d_offset ….. offset of that directory entry in the actual file system directory

d_name ….. name of the file or directory

d_reclen record length of this entry

d_type ….. type of file(text file, block file, directory etc)

readdir()

- readdir() is a system call used to read into a directory.
- It returns a pointer to the dirent structure.

/* Unix program to read all regular files from a directory */

```
/printfile.c*/
#include<stdio.h>
#include<dirent.h>
#include<stdlib.h>
void main()
{  DIR *dirp;
struct dirent *dp;
if((dirp=opendir("/home/kumar"))==NULL)  /* trying to open kumar directory*/
 {printf("\n cannot open");
exit(1); }
```

```
for(dp=readdir(dirp); dp!=NULL; dp=readdir(dirp))
{
 if(dp->d_type==DT_REG)   /* printing only if file is a regular file */
printf("%s\n",dp->d_name);
}
closedir(dirp);
}
```


thanks