Introduction to System Programming

System Call for Process Management

Minal Shah

Outline

- fork()
- waitpid
- exec family

Fork system Call

- The fork system call is used to create a new processes.
- The newly created process is the child process.
- The process which calls fork and creates a new process is the parent process.
- The child and parent processes are executed concurrently.
- The child and parent processes reside on different memory spaces. These memory spaces have same content and whatever operation is performed by one process will not affect the other process.

Fork system Call

- When the child processes is created; now both the processes will have the same Program Counter (PC), so both of these processes will point to the same next instruction.
- The files opened by the parent process will be the same for child process.
- The child process is exactly the same as its parent but there is difference in the **processes ID's**:
  - The process ID of the child process is a unique process ID which is different from the ID's of all other existing processes.
  - The Parent process ID will be the same as that of the process ID of child's parent.

Properties of Child Process Inherited from the parent

- process credentials (real/effective/saved UIDs and GIDs)
- environment
- stack
- memory
- open file descriptors (note that the underlying file positions are shared between the parent and child, which can be confusing)
- close-on-exec flags
- signal handling settings
- nice value
- scheduler class
- process group ID
- session ID
- current working directory
- root directory
- file mode creation mask (umask)
- resource limits
- controlling terminal

Properties Unique to the Child

- process ID
- different parent process ID
- Own copy of file descriptors and directory streams.
- process, text, data and other memory locks are NOT inherited.
- process times, in the tms struct
- resource utilizations are set to 0
- pending signals initialized to the empty set
- timers created by timer_create not inherited
- asynchronous input or output operations not inherited

Fork system Call in c

- There are no arguments in fork() and the return type of fork() is integer.
- You have to include the following header files when fork() is used:
  - **#include <sys/types.h>**
    #include <unistd.h>
  - When working with fork(), <sys/types.h> can be used for type pid_t for processes ID's as pid_t is defined in <sys/types.h>.
  - The header file <unistd.h> is where fork() is defined so you have to include it to your program to use fork().
  - The return type is defined in <sys/types.h> and fork() call is defined in <unistd.h>. Therefore, you need to include both in your program to use fork() system call.

Fork system Call in c

- Syntax of fork()
  - pid_t fork(void);
  - The return type is pid_t.
  - When the child process is successfully created, the PID of the child process is returned in the parent process and 0 will be returned to the child process itself.
  - If there is any error then -1 is returned to the parent process and the child process is not created.
  - No arguments are passed to fork().

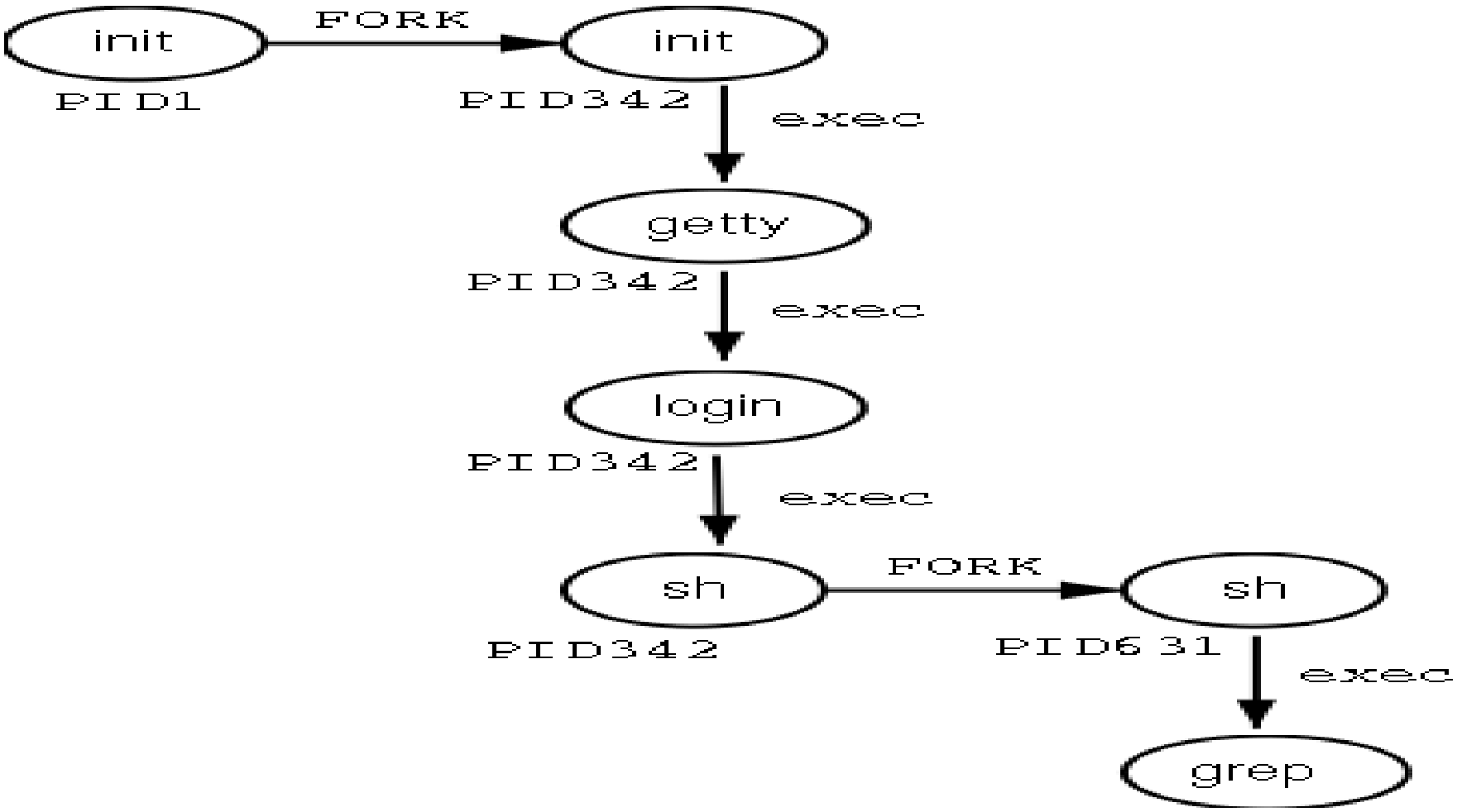
Fork System Call in c Example

```
/* LetsFork/fork1.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc,char *argv[])
{
    printf("Before Fork, My PID: [%d]\n",getpid());
    fork();
    printf("After Fork, My PID: [%d]\n",getpid());
    return 0;
}
```

Here is the sample output:

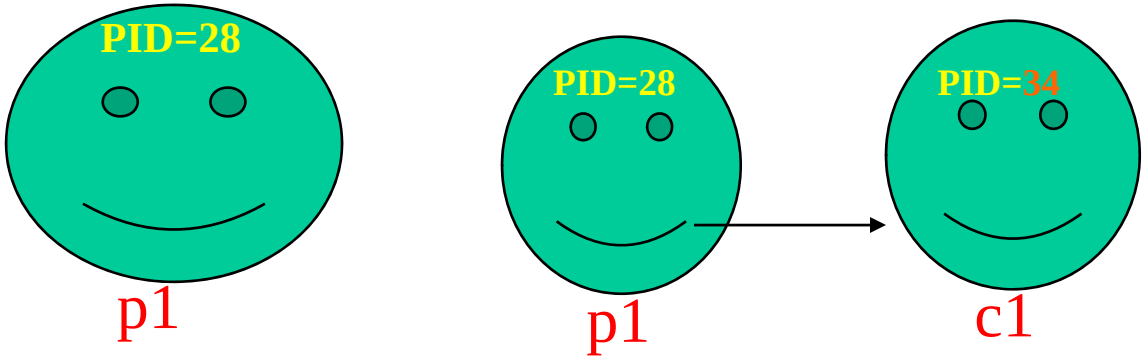
```
Before Fork, My PID: [8265]
After Fork, My PID: [8265]
After Fork, My PID: [8266]
```

Login process



The `fork()` System Call

- The fork() is one of the those system calls, which is called once, but returns twice!
- After fork() both the parent and the child are executing the same program.
- On error, fork() returns -1
- Remember, after fork() the execution order is not guaranteed.



Consider a piece of program

```
...
pid_t pid = fork();
printf("PID: %d\n", pid);
...
```

The parent will print:
`PID: 34`
And the child will **always** print:
`PID: 0`

/* PROGRAM FOR WRITING TO A FILE IN CHILD PROCESS AND PARENT WOULD READ THAT FILE */

```
#include<fcntl.h>
#include<stdio.h>
#include<sys/types.h>
#include<wait.h>
void main()
{   int fp, pid;
    char chr = 'A';
    pid = fork();
    if(pid == 0)
    {
            fp = open("f1.txt",O_WRONLY | O_CREAT,0666);
            printf("In child chr is %c\n",chr);
            chr = 'B';
            write(fp,&chr,1);
            printf("In child chr is after change %c\n",chr);
            printf("Child exiting\n");
            close(fp);    }
```

/* PROGRAM FOR WRITING TO A FILE IN CHILD PROCESS AND PARENT WOULD READ THAT FILE */

```
    else
    {
            wait((int *)0);
            fp=open("f1.txt",O_RDONLY);
            read(fp,&chr,1);
            printf("Chr after parent reads %c\n",chr);
            close(fp);
    }
}
```

Output

```
In child chr is A
In child chr is after change B
Child exiting
Chr after parent reads B
```

/* PROGRAM FOR ACCESSING FILE GLOBALLY */

```
#include<fcntl.h>
#include<stdio.h>
#include<sys/types.h>
#include<wait.h>
void main()
{   int fp, pid;
    char buff[11];
    fp = open("m1.txt",O_RDONLY);
    pid = fork();
    if(pid == 0)
    {
            printf("Child Begins %d\n",getpid());
            read(fp,buff,10);
            buff[10]='\0';
            printf("Child Reads :");
            puts(buff);
```

/* PROGRAM FOR ACCESSING FILE GLOBALLY */

```
      printf("Child exiting\n");
      }
    else
    {       wait((int *)0);
            read(fp,buff,10);
            buff[10]='\0';
            printf("Parent Reads :");
            puts(buff);
            printf("Parent exiting\n");
            close(fp);
    }
}

// content of m1.txt
ARISE AWAKE AND STOP NOT TILL THE GOAL IS REACHED.
```

/* PROGRAM FOR ACCESSING FILE GLOBALLY */

// content of m1.txt
ARISE AWAKE AND STOP NOT TILL THE GOAL IS REACHED.

Output (if wait is used)

Child Begins 4194
Child Reads :ARISE AWAK
Child exiting
Parent Reads :E AND STOP
Parent exiting

Output (if wait is not used)

Parent Reads :ARISE AWAK
Parent exiting
Child Begins 4172
Child Reads :E AND STOP
Child exiting

/* PROGRAM FOR CHECKING WHETHER THE HANDLE IS REALLY SHARED */

```
#include<fcntl.h>
#include<stdio.h>
#include<sys/types.h>
#include<wait.h>
void main()
{   int fp, pid;
    char buff[11];
    fp = open("m1.txt",O_RDONLY);
    pid = fork();
    if(pid == 0)
    {       printf("Child Begins File handle is %d\n",lseek(fp,01,1));
            read(fp,buff,10);
            buff[10]='\0';
            printf("File handle is now at %d in Child process \n",lseek(fp,01,1));
     }
```

/* PROGRAM FOR CHECKING WHETHER THE HANDLE IS REALLY SHARED */

```
    else
    {
        wait((int *)0);
        printf("File handle in Parent process is %d\n",lseek(fp,01,1));
        close(fp);
    }
}
```

Output(without wait)

File handle in Parent process is 1
Child Begins File handle is 2
File handle is now at 13 in Child process

Output(with wait)

Child Begins File handle is 1
File handle is now at 12 in Child process
File handle in Parent process is 13

Exec family

- The exec() call replaces a current process' image with a new one (i.e. loads a new program within current process).
- The new image is either regular executable binary file or a shell script.
- By exec() we usually refer to a family of calls:
  - int execl(char *path, char *arg, ...);
  - int execv(char *path, char *argv[]);
  - int execle(char *path, char *arg, ..., char *envp[]);
  - int execve(char *path, char *argv[], char *envp[]);
  - int execlp(char *file, char *arg, ...);
  - int execvp(char *file, char *argv[]);
- Here's what l, v, e, and p mean:
  - l means an argument list,
  - v means an argument vector,
  - e means an environment vector, and
  - p means a search path.
- **#include <unistd.h>**

Exec family

- 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.

Exec family

- 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. .

Exec family

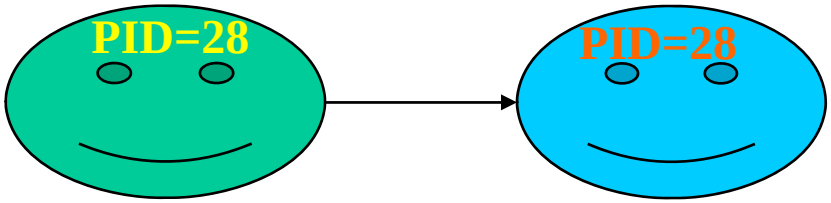
- execl(): The first member, which is execl, takes the full pathname of the script or command as its first argument, the arguments to be passed for that script or command as its subsequent arguments, and at last, it also accepts a null pointer which marks the end of arguments.
- execlp( ) Used after the fork() system call by one of the two processes to replace the process‟ memory space with a new program.
  - It loads a binary file into memory destroying the memory image of the program containing the execlp system call and starts its execution.
  - The child process over lays its address space with the UNIX command /bin/ls using the execlp system call.

Exec family

- execv(): There is only a minor difference between execl and execv.
- execvp():The exec family of functions replaces the current process image with a new process image.
  - The functions execvp will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash (/) character.
  - The search path is the path specified in the environment by the PATH variable. If this variable isn't specified, the default path ``/bin:/usr/bin:'' is used .

The exec( ) System Call

- Upon success, exec() **never** returns to the caller.
- If it does return, it means the call failed.
- Typical reasons are: non-existent file (bad path) or bad permissions.
- Arguments passed via exec() appear in the argv[] of the main() function.



Legend:

The exec( )System Call Example

- **example.c**

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
    {
       printf("PID of example.c = %d\n", getpid());
       char *args[] = {"Hello", "C", "Programming", NULL};
       execv("./hello", args);
       printf("Back to example.c");
       return 0;
    }
```

The exec( )System Call Example

- **hello.c**

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
        printf("We are in Hello.c\n");
        printf("PID of hello.c = %d\n", getpid());
        return 0;
}
```

OUTPUT:

- PID of example.c = 4733
  We are in Hello.c
  PID of hello.c = 4733

//Demonstrate execl() (1)

```
//e1.c
#include<stdio.h>
#include<unistd.h>
void main()
{
    printf("Before exec my ID is %d\n",getpid());
printf("My parent ID is %d\n",getppid());
printf("exec starts\n");
execl("/home/student/e2","e2",(char *)0); //specify the exact path of e2
    file
printf("This will not print\n");
}
```

Compile the file as

```
$gcc –o e1 e1.c
```

//Demonstrate execl() (2)

```
//e2.c
#include<stdio.h>
#include<unistd.h>
void main()
{
    printf("After the exec my process ID is %d\n",getpid());
    printf("My parent ID is %d\n",getppid());
    printf("exec ends\n");
}
```

Compile the file as

```
$gcc –o e2 e2.c
```

To run this

```
$ ./e1
```

//Terminal and file buffering (1)

```
//e6.c
#include<stdio.h>
#include<unistd.h>
void main()
{
    printf("The cat jumped over the wall");
     execl("/home/student/e7","e7",(char *)0);
        //specify the exact path of e7 file
}
```

Compile the file as

$gcc –o e6 e6.c

//Terminal and file buffering (2)

```
//e7.c
#include<stdio.h>
#include<unistd.h>
void main()
{
    printf("The clock went dong");
}
```

Compile the file as

```
$gcc –o e7 e7.c
```

To run this

```
$ ./e6
 //only the clock line is printed
```

//Terminal and file buffering (1)

```
//Now change the following code
//e6.c
#include<stdio.h>
#include<unistd.h>
void main()
{
    printf("The cat jumped over the wall\n");
   execl("/home/student/e7","e7",(char *)0);
   //specify the exact path of e7 file
}
```

Compile the file as

```
$gcc –o e6 e6.c
```

//Terminal and file buffering (2)

//don’t change e7.c

To run this

$ ./e6

Output would

The cat jumped over the wall

The clock went dong

//demonstrate execv() (1)

```
//e8.c
#include<stdio.h>
#include<unistd.h>
void main()
{
    char *temp[3];
    temp[0]=”ls”;
    temp[1]=”-l”;
    temp[2]=(char *)0;
     execv(“/bin/ls”,temp);
     printf(“this will not be printed\n”);
}
```

Compile the file as $gcc –o e8 e8.c

Run as $./e8

//demonstrate execvp ()(1)

```
//e9.c
#include<stdio.h>
#include<unistd.h>
void main()
{
    char *temp[4];
    temp[0]=”e10”;  //or temp[0]=”/home/student/e10”
    temp[1]=”hello”;
    temp[2]=”world”;
    temp[3]=(char *)0;
    printf(“I am parent my pid is %d\n”,getpid());
     execvp(temp[0],temp);
     printf(“this will not be printed\n”);
}
```

Compile the file as
$gcc –o e9 e9.c

//demonstrate execvp ()(2)

```
//e10.c
#include<stdio.h>
#include<unistd.h>
void main(argc,argv)
int argc;
char *argv[]
{
    printf("I am child after execl my pid is %d\n",getpid());
printf("child is %s and its arguments are %s
    %s\n",argv[0],argv[1],argv[2]);
printf("exec ends");
}
```

//demonstrate execvp ()(3)

Compile the file as
$gcc –o e10 e10.c

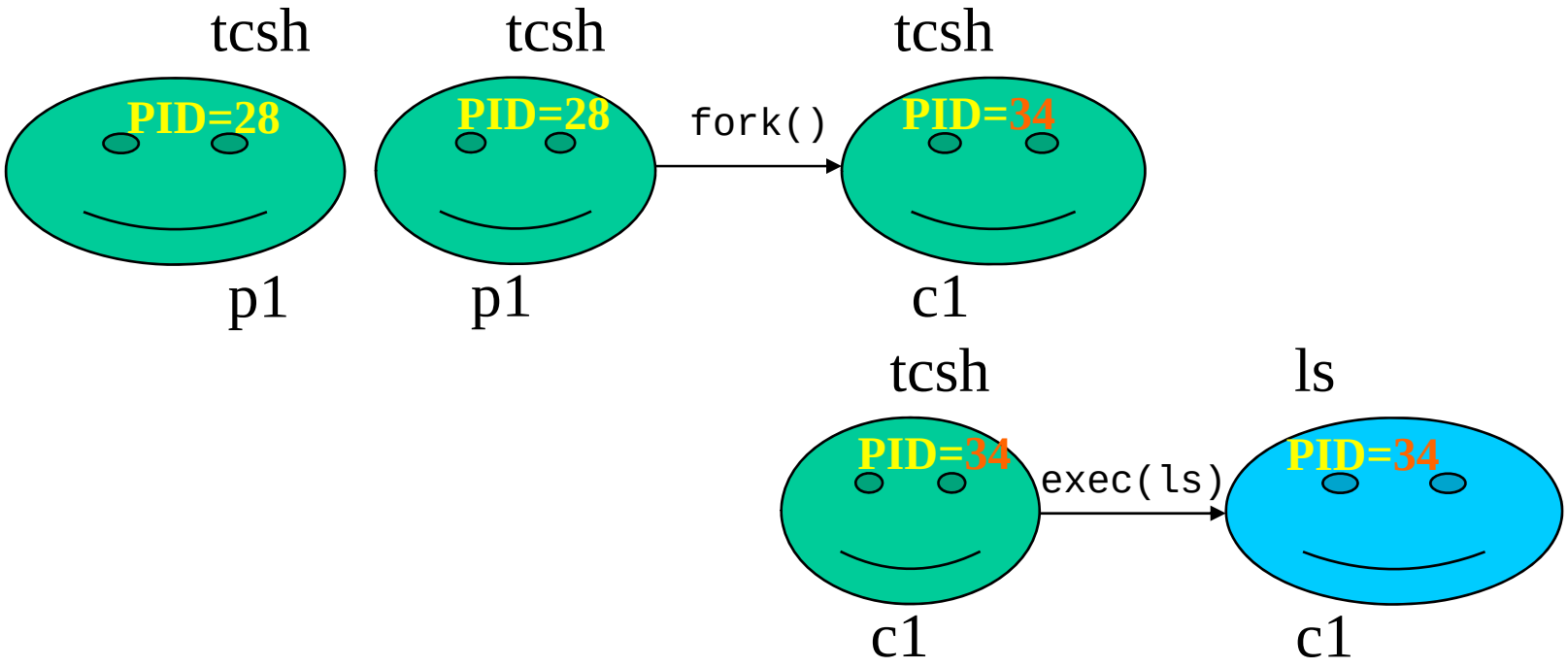
$./e9

At what cost, fork()?

- Simple implementation of fork():
  - allocate memory for the child process
  - copy parent's memory and CPU registers to child's
  - *Expensive* !!
  - In 99% of the time, we call exec() after calling fork()
  - the memory copying during fork() operation is useless
  - the child process will likely close the open files & connections
  - overhead is therefore high

`fork()` and `exec()` Combined

- Often after doing `fork()` we want to load a new program into the child. *E.g.*: a shell.



Combining fork() and exec() system calls

example.c

```
#include <stdio.h>
    #include <sys/types.h>
    #include <unistd.h>
    int main(int argc, char *argv[])
    {
        printf("PID of example.c = %d\n", getpid());
        pid_t p;
        p = fork();
        if(p==-1)
        {
            printf("There is an error while calling fork()");
        }
```

Combining fork() and exec() system calls

- **example.c (2)**

```
if(p==0)
{
    printf("We are in the child process\n");
    printf("Calling hello.c from child process\n");
    char *args[] = {"Hello", "C", "Programming", NULL};
    execv("./hello", args);
    }
    else
    {
        printf("We are in the parent process");
    }
    return 0;
}
```

Combining fork() and exec() system calls

- **hello.c:**

```
#include <stdio.h>
#include <unistd.h>
    #include <stdlib.h>
    int main(int argc, char *argv[])
    {
        printf("We are in Hello.c\n");
        printf("PID of hello.c = %d\n", getpid());
        return 0;
    }
```

OUTPUT:

```
PID of example.c = 4790
We are in Parent Process
We are in Child Process
Calling hello.c from child process
We are in hello.c
PID of hello.c = 4791
```

The System `wait()` Call

- Forces the parent to suspend execution, i.e. wait for its children or a specific child to die (*terminate* is more appropriate terminology, but a bit less common).

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- The wait() causes the parent to wait for any child process.
- The waitpid() waits for the child with specific PID.
- The status, if not NULL, stores exit information of the child, which can be analyzed by the parent using the W*() macros.
- The return value is:
  - PID of the exited process, if no error
  - (-1) if an error has happened

Wait or not wait ?

- There are ***two attitude*** that can be taken by the parents towards its child.
  - It may **wait** for the child to die so that it can spawn the next process. The **death** is informed by the **kernel** to the parent. The shell process wait for command to die before it returns the prompt to take up **next command**.
  - It may **not wait** for the child to die at all and may continue to spawn other processes.(the **init** process work this way) because the **init** is the parent of several processes. Shell can also create a process **without waiting** for it to die

The wait() System Call

- A child program returns a value to the parent, so the parent must arrange to receive that value
- The wait() system call serves this purpose
  - it puts the parent to sleep waiting for a child's result
  - when a child calls exit(), the OS unblocks the parent and returns the value passed by exit() as a result of the wait call (along with the pid of the child)
  - if there are no children alive, wait() returns immediately
  - also, if there are zombies waiting for their parents, wait() returns one of the values immediately (and deallocates the zombie)

The waitpid() Function

- The **waitpid()** function allows a parent to wait for a particular child to terminate
  - It also allows a parent process to check whether a child has terminated without blocking

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status_location, int options);
```

- The function takes three paremeters: a pid, a pointer to the location for returning a status, and a flag specifying options
- There are several variations on the **pid** parameter and the resulting actions of the **waitpid()** function
  - pid == -1 waits for any child
  - pid > 0 waits for the specific child whose process ID is pid

The waitpid() Function

- pid == -1 waits for any child
- pid > 0 waits for the specific child whose process ID is pid
- pid == 0 waits for any child in the same process group as the caller
- pid < -1 waits for any child in the process group noted by the absolute value of pid

- The **options** parameter is the bitwise inclusive OR of one or more flags
  - WNOHANG option causes the function to return even if the status of a child is not immediately available
  - It returns 0 to report that there are possibly unwaited-for children but that their status is not available

waitpid() helps to obtain status information about one of its child process

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
    pid_t pid;
    int status, n;
    if ((pid =fork()) < 0)
    {
        perror("Fork error");
        exit(1);
    }
    else if(pid == 0)
    {
```

waitpid() helps to obtain status information about one of its child process

```
        printf("child pid = %d\n",getpid());
        sleep(5);
        exit(2);
    }
    else
    if ((n = waitpid(pid,NULL,0)) < 0)
    {
        perror("waitpid error");
        exit(3);
    }
    printf("n = %d\n",n);
    exit(0);
}
```

The `exit()` System Call

```
#include <stdlib.h>

void exit(int status);
```

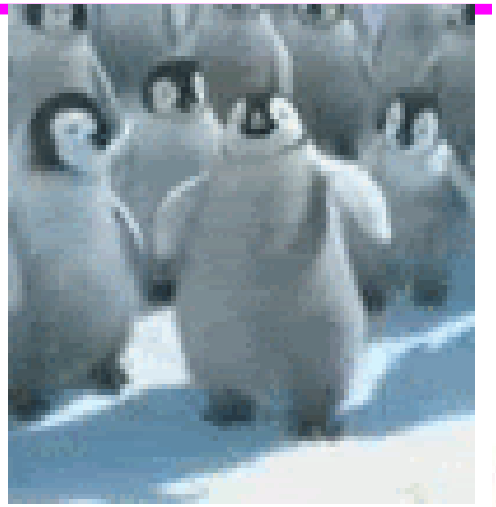
- This call gracefully terminates process execution. Gracefully means it does clean up and release of resources, and puts the process into the zombie state.
- By calling wait(), the parent cleans up all its zombie children.
- exit() specifies a return value from the program, which a parent process might want to examine as well as status of the dead process.
- _exit() call is another possibility of quick death without cleanup.

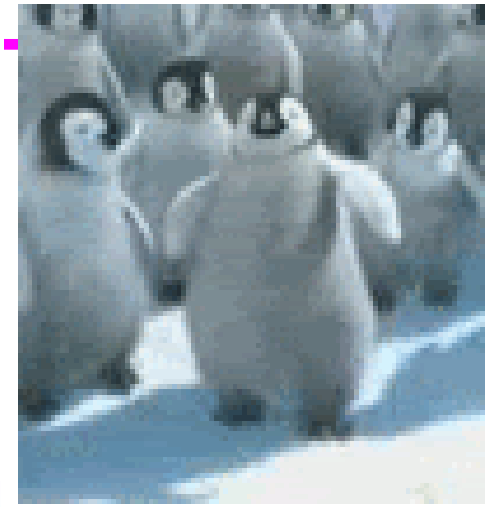
Orderly Termination: exit()

- After the program finishes execution, it calls *exit*()
- This system call:
  - takes the "result" of the program as an argument
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the zombie/defunct state
    - If not, it deallocates all data structures, the process is dead
  - cleans up all waiting zombies
- Process termination is the ultimate garbage collection (resource reclamation).

Process Lifecycle

- exit() performs following operations.
  * Flushes unwritten buffered data.
  * Closes all open files.
  * Removes temporary files.
  * Returns an integer exit status to the operating system.
- The mystery behind exit() is that it takes only integer args in the range 0 – 255 . Out of range exit values can result in unexpected exit codes. An exit value greater than 255 returns an exit code modulo 256.
  For example, exit 9999 gives an exit code of 15 i.e. (9999 % 256 = 15).





Minal Shah