COMP 8567

Advanced Systems Programming

Process Control

Outline

- Unix Processes
- Creating a new process: **fork()**
- Terminating a process: **exit()**
- Waiting for a process: **wait()**
- Waiting for a specific process: **waitpid()**
- Orphan and Zombie Processes
- **exec()**
- Changing Directory : **chdir()**
- Summary

Unix Processes

- Unix is a **multiuser** and **multitasking** operating system
  - **Multiple users** can run their programs concurrently and share hardware resources
  - An **user can run multiple programs** and tasks concurrently
- It appears that the execution is done in parallel, however in reality the OS switches between multiple users and processes rapidly (back and forth) in an **interleaved** manner
- **Unix Processes:**
  - An executing program is a process.
  - ex1.c (program)-> ex1 (executable)->./ex1 (process)
- **Every process in Unix has the following :**
  - A unique process ID (PID)
  - Some code : instructions that are being executed
  - Some data : variables
  - A stack : a form of memory where it is possible to push and pop variables/data
  - An environment : CPU registers' contents, tables of open files

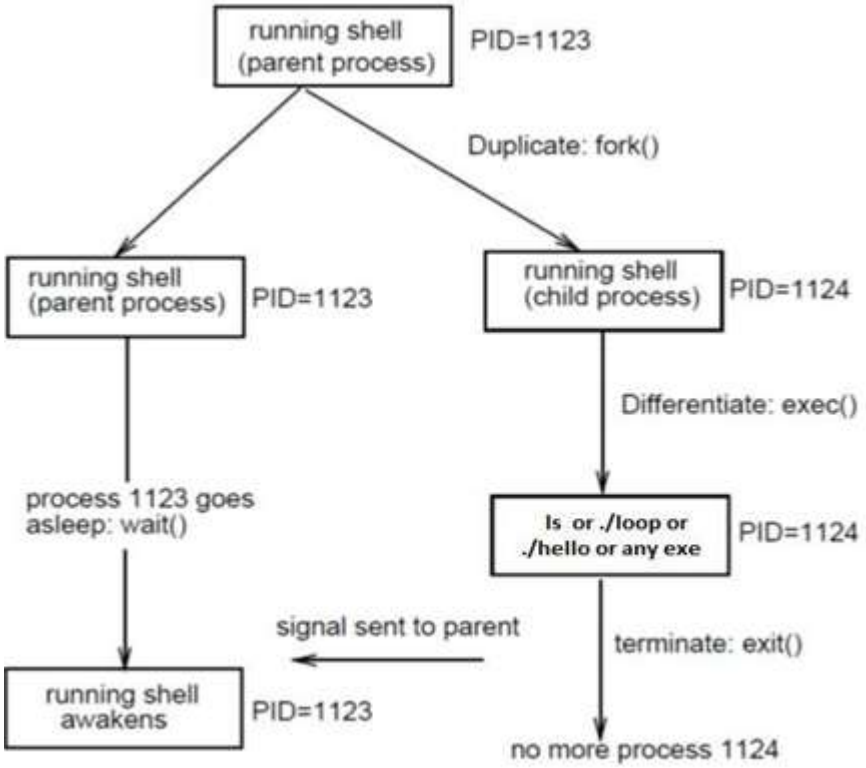
Unix Processes..

- Unix starts as a single process, called **init.** The PID of init is 1.
- The only way to create a new process in Unix, is to **duplicate** an existing one.
- the process **init** is the **ancestor** of all subsequent processes.
- Process **init** never dies

The **creation or spawning** of new processes is done with two **system calls** :

- **fork() :** duplicates the caller process
- **exec() :** replaces the caller process by a new one.

Ex: Running any executable from the shell (ls , ./loop etc)



Creating a New Process: fork()

- Synopsis: **pid_t fork(void)**
- when successful, the fork() system call :
  - creates a copy of the caller (parent) process
  - returns the **PID** of the newly created child process to the parent
- returns 0 to the new process (the child).
- If not successful, fork() returns -1.
- fork() is a strange system call : called by a single process but **returns twice**, to two different processes (parent and child)
- **Very important:** After the fork() system call is invoked in the parent and the child process is created, both the parent and the child process run **concurrently** in the system.
- Execution resumes **at the line immediately after the fork() statement** in both the parent and the child process.

Parent Process

```
int main(void)
{
int i=fork(); // returns pid of child (>0) on
success
if(i==0)
{
printf("\n\nCHILD PROCESS\n");
else if (i<0)
{
printf("\n\nERROR\n");
}
else
{
printf("\n\nPARENT PROCESS\n");
}
}
```

Duplicate child process is created. Both parent and child execute **concurrently**

Child Process

```
int main(void)
{
int i=fork(); //returns 0 on success
if(i==0)
{
printf("\n\nCHILD PROCESS\n");
else if (i<0)
{
printf("\n\nERROR\n");
}
else
{
printf("\n\nPARENT PROCESS\n");
}
}
```

f1.c
fork9loop.c

- After forking, the child process will have :
  - its own unique PID
  - a different PPID (PID of its parent process)
  - its own **copy** of the parent's data segment and **file descriptors**

fork() is primarily used in two situations :

- A process wants to execute another program (Ex: Bash is a process that wants to run ./welcome)
- A process has a main task and when necessary, creates a child to handle a subtask (servers/sockets)

```
main() {

int a;
a=fork();
a=fork();
printf("\nProcess id = %d, Parent id= %d", getpid(), getppid());

}
```

```
main() {

int a;
a=fork();
a=fork();
printf("\nProcess id = %d, Parent id= %d", getpid(), getppid());

}
```

```
main() {

int a;
a=fork();
a=fork();
printf("\nProcess id = %d, Parent id= %d", getpid(), getppid());

}
```

Note: ***n*** consecutive fork() calls in a process creates ***(2 ^{n} – 1)*** ***child/descdendent*** processes

After the fork() system call creates a child process, the execution **resumes from the line immediately after fork()** in both the parent and the child process

```
main() {

int a;
a=fork();
a=fork();
printf("\nProcess id = %d, Parent id= %d", getpid(), getppid());

}
```

```
#include <stdio.h>
#include <stdlib.h>
//f44.c

main()
{

int a;
a=fork();
a=fork();
printf("\nProcess id = %d, Parent id= %d", getpid(), getppid());

}
```

```
// fork33.c

 include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
int pid;
pid = fork();
pid = fork();
pid = fork();

//seven new processes are created, in addition to the parent process.

if(pid==0)
{
for(;;);
}
else if(pid<0)
{
printf("Error Forking");
}
else
{
for(;;);
}

}
```

Terminating a Process: exit()

Synopsis: **void exit(int status);**

This call terminates a process and does not return a value

The **status** value is available to the parent process through the wait() system call.

When invoked by a process, the exit() system call :

- closes all the process's file descriptors
- frees the memory used by its code, data and stack
- sends a **SIGCHLD** signal to its parent (remember, every process has a parent(barring init) and waits for the parent to accept its return code.
  - **SIGCHLD:** A signal that indicates a process started by the current process has terminated
  - Note: SIGCHILD is a signal and **not a return value**

Waiting for a process: wait() // f2.c

- Synopsis: **pid_t wait(int *status);**
- This call allows a parent process to wait for one of its children to terminate and to accept its child's termination code.
- When called, wait() can:
  - Block (suspend) the caller process, until one of its children terminates
  - or return the PID of the child process, If a child has terminated and is waiting for its termination to be accepted, or
  - return immediately with an error(-1) if it does not have any child process.
- When successful, **wait()** returns the PID of the terminating child process.

//forkexit1.c

- Some bit-manipulation macros have been defined to deal with the value in the variable **status**(you need to include < sys/wait.h >)
- **WIFEXITED(status)** : returns true for normal child termination.
- **WEXITSTATUS(status)** : used only when WIFEXITED(status) is true, it returns the exit status as an integer(0-255).
- **WIFSIGNALED(status)** : true for abnormal child termination
- **WTERMSIG(status)** : used only when WIFSIGNALED(status) is true, it returns the signal number that caused the abnormal death of the child process.

Waiting for a specific process: waitpid()

Synopsis:

Pid_t waitpid(pid_t pid, int *status, int options);

This call allows a parent process to wait for a specific child to terminate and to accept its child's termination code.

Note: wait(&status) is equivalent to waitpid(-1, &status, 0)

//pid is always a positive integer

Orphan and Zombie Processes //op.c

- A process that terminates does not actually leave the system before the parent process accepts its return.
- There are 2 interesting situations :

  Parent exits(**for example, the parent has been killed prematurely**) while its children are still alive- **The children become orphans.**

  Parent is not in a position to accept the exit and the termination code of the child process (parent is in an infinite loop)- **The children become zombies**
- Because some process must accept their return codes, the kernel simply changes their **PPID to 1** (init process) in the absence of a parent process
- Orphan processes are systematically adopted by the process init (PID of init is 1). and init accepts all its children returns.

Zombie Processes //zomex.c

When parent processes is not able to accept the termination code of their child processes, the children become zombies and remain in the system's process table waiting for the acceptance of their return. However, they loose their resources (data, code, stack...).

Because the system's process table has a fixed-size, too many zombie processes can require the intervention of the system administrator.

```
//zomex.c
int main(int argc, char *argv[]){
int pid;
pid = fork();
if(pid==0){
//Child
printf("child process, pid=%d\n", getpid());
exit(0);
}
else if(pid<0){
printf("Error Forking");
}
else{
//Parent
while(1)
sleep(5);
}}
```

Use the following command to periodically kill all the **forked processes** by the user in your system. Otherwise, the system performance will be negatively affected.

$ killall –u uername

$ ps –u //obtain the list of user processes

PATH VARIABLE

$ echo $PATH

$ export PATH= $PATH:/home/pranga

$ export PATH=$PATH:~ //both are the same

$ export PATH=$PATH:~/chapter5

$ echo $PATH

exec()

The exec() family of system calls allows a process to **replace its current code, data and stack with those of another program.**

- **int execl(const char *path, [const char *argi,]+ NULL)** // whole pathname of the executable is required
- Same as: int execl(const char *path, arg(0),arg(1),arg(2),……arg(n),NULL);
  - Example: execl(“/bin/ls”, “/bin/ls”, “-1”,NULL); //executes ls -1
- **int execlp(const char *path, [const char *argi,]+ NULL)** //whole pathname of the executable not required if it is added to the PATH variable
- Same as: int execlp(const char *path, arg(0),arg(1),arg(2),……arg(n),NULL);
  - Example: execlp(“ls”, ”ls”, “-1”, NULL);
- **int execv(const char *path, const char *argv[])** // whole pathname of the executable required
- **int execvp(const char *path, const char *argv[])** // whole pathname of the executable not required if it is added to the PATH variable
  - All the arguments along with the NULL terminator are first stored in argv[] , the argument vector

Cont…

- where i = 0; : : : ; n and + means one or more times.
- The difference between these 4 system calls has to do with syntax.
- execl() and execv() require the **whole pathname of the executable** program to be supplied.
- execlp() and execvp() use the variable **$PATH** to find the program.
- exec*() never returns when it is successful. It returns -1 if it is not successful

```
int main(){    //forkexec.c
   int p;
   p = fork();
   if(p==-1)   {
      printf("There is an error while calling fork()");
   }
   if(p==0) {
      printf("\nWe are in the child process\n");
      printf("\nThe child process is now being replaced by the executable ls -1 \n");
      int k=execlp(“ls”, “ls”, “-1”, NULL);
      exit(0); }
   else
   {
     int k=wait();
      printf("\nWe are in the parent process\n");
      exit(0);   }
   return 0;
}
```

execl() //execl.c

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

int main(void) {
  char *programName = "/bin/ls";
  char *arg1 = "-1";
 int k=execl(programName, programName, arg1, NULL);
 //int execl(const char *path, [const char *argi,]+ NULL)
// arg0 must be the name of the program
  return 0;
}
```

execlp() //execlp.c

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

int main(void) {
  char *programName = “ls";
  char *arg1 = "-1";
 int k=execlp(programName, programName, arg1, NULL);
 //int execlp(const char *path, [const char *argi,]+ NULL)
// arg0 must be the name of the program
  return 0;
}
```

execv() //execv.c

```
#include <unistd.h>

int main(void) {
  char *programName = "/bin/ls";
  char *args[] = {programName, "-1", "/home/pranga/chapter5", NULL};
  int k=execv(programName, args);
 //int execv(const char *path, const char *argv[])
 //argv[0] must be the name of the program
  return 0;
}
```

execvp() //execvp.c

```
#include <unistd.h>

int main(void) {
  char *programName = "ls";
  char *args[] = {programName, "-1", "/home/pranga/chapter5", NULL};
  //Display the contents of  "/home/pranga/chapter5“
  int k=execvp(programName, args);
 //int execvp(const char *path, const char *argv[])
 //argv[0] must be the name of the program
  return 0;
}
```

For both execl() and execlp() arg0 must be the name of the program.

For both execv() and execvp() arg[0] must be the name of the program.

chdir() //chdirfork.c

A child process inherits its current working directory from its parent.

Each process can change its working directory using chdir().

Synopsis : int chdir(const char * pathName);

chdir() returns 0 if successful -1 otherwise.

It fails if the specified path name does not exist or if the process does not have execute permission from the directory.

Summary

- Unix Processes
- Creating a new process: fork()
- Terminating a process: exit()
- Waiting for a process: wait()
- Waiting for a specific process: waitpid()
- Orphan and Zombie Processes
- exec()
- Changing Directory : chdir()
- Summary

THANK YOU