innovate achieve lead

Process File System

BITS Pilani
Pilani Campus

Computer Science and Information Systems Department, BITS, Pilani

Today's Agenda

- Unix System Calls
  - Wait process
  - Zombie process
  - Orphan process
  - vfork system call

Wait() System Call

- The **wait**() system call suspends execution of the current process until one of its children terminates or a signal is received. At that moment, the caller resumes its execution.
- One of the main purposes of **wait()** is to wait for completion of child processes.
- If any process has more than one child processes, then after calling wait(), parent process has to be in wait state if no child terminates.
- In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then terminated the child remains in a "zombie" state

Wait() System Call

Returns:

- If any process has no child process then wait() returns immediately “-1”.
- If the parent process has a child that has terminated, that child's PID is returned and it is removed from the process table.
- If only one child process is terminated, then return a wait() returns PID of the terminated child process.
- If the parent process has a child that is not terminated, it (the parent) is suspended till it receives a signal. The signal is received as soon as a child dies.

Fork() and wait() system call

```
int main(void)
{
int pid;
int status;
printf("Hello World!\n");
pid = fork();
if(pid == -1) /* check for error in fork */ {
  perror("fork failed");
  exit(1);
  }
if(pid == 0)
  printf("I am the child process. %d\n",getpid());
else {
  wait(&status); /* parent waits for child to finish */
  printf("Child Process with pid = %d completed with a
  status %d\n",pid,status);
  printf("I am the parent process.%d\n",getpid());
  }
return 0;}
```

output

```
Hello World!

I am the child process. 1928

Child Process with pid = 1928

completed with a status 7424

I am the parent process.1927
```

Multiple forks and wait system call

```
main(){
pid_t whichone, first, second ;  int howmany, status ;
if((first = fork()) == 0) /* Parent spawns 1st child */ {
printf("I am the first child, & my ID is %d\n", getpid());
sleep(10);  exit(0); }
else if(first == -1) {
perror("1st fork: something went wrong\n") ;  exit(1);  }
else if((second = fork()) == 0) /* Parent spawns 2nd child */ {
printf("I am the second child, & my ID is %d\n", getpid( ));
sleep(15);  exit(0);  }
else if (second == -1){
perror("2nd fork: something went wrong\n") ;   exit(1);  }
printf("This is parent\n");
howmany = 0;
while(howmany < 2) {/* Wait Twice */
       whichone = wait(&status);  howmany++;
       if(whichone == first)
               printf("First child exited\ncorrectly");
       else
              printf("Second child exited\ncorrectly");}}
```

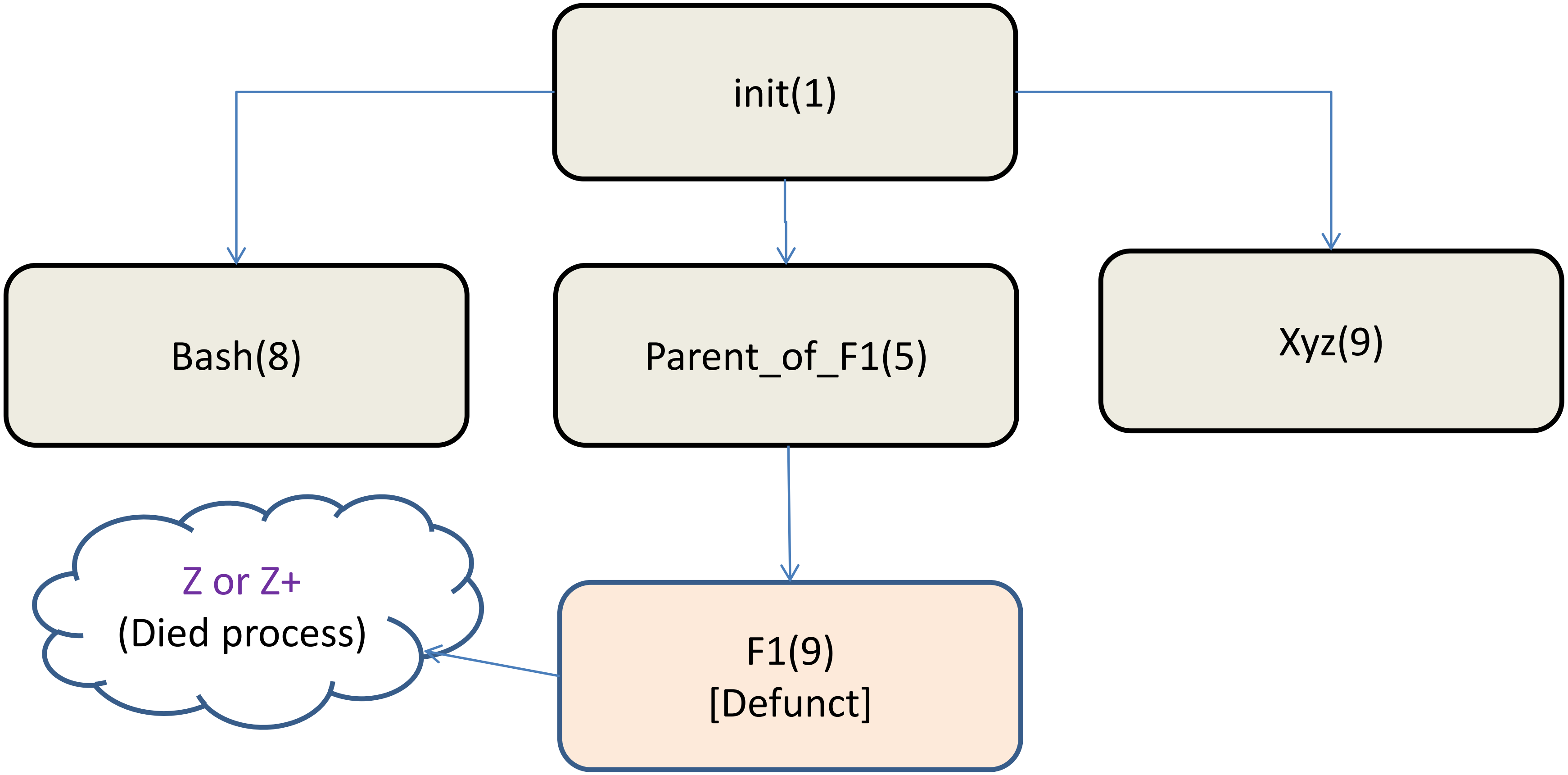
output

```
This is parent
I am the first child, & my ID is 1704
I am the second child, & my ID is 1705
First child exited
correctly
Second child exited
correctly
```

Zombie Process

- On Unix and Unix-like computer operating systems, a zombie process or defunct process is a process that has completed it’s execution but still has an entry in the process table.
- This entry is still needed to allow the parent process to read its child’s exit status.
- You cannot kill zombie process directly because it is already dead.
- To kill the zombie process, you should kill the parent process. However if the parent process is init (i.e., 1), then only thing you can do is reboot.
- It takes very tiny memory(description info) but will take process ID which is limited. So, it is better not to have zombie process.

Zombie Process



You can’t kill a zombie process(F1) because it’s already dead – like an actual zombie.

Zombie Process

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // Fork returns process id   // in parent process
    pid_t child_pid = fork();
    // Parent process
    if (child_pid > 0)
        sleep(50);
    // Child process
    else
        exit(0);
      return 0;
 }
```

Zombie Process

```
main() {
int pid ;
pid = fork(); /* Duplicate. Child and parent continue from here */

if ( pid != 0){
      printf("Its a Parent Process with pid=%d, goes to
      sleep\n",getpid());
      /* pid is non-zero, so I must be the parent */
      while (1)
      /* Never terminate and never execute a wait ( ) */
      sleep (5); /* stop executing for 10 seconds */
      }

else{
      printf("Child Process with pid = %d\n",getpid());
      /* pid is zero, so I must be the child */
      exit (1) ; /* exit with any number */
      }
}
```

Getting Rid of Zombie Processes

- Send the signal with the **kill** command, replacing *pid* in the command below with the parent process's PID:

  $ kill defunct-pid

- Still zombie process(defunct) present

  $ kill -9 defunct-pid(Force kill)

- Still zombie process(defunct) present

  $ kill parent-id-of-defunct-pid

- Still zombie process(defunct) present

If you still find defunct process eating up RAM then last and final solution is to reboot your machine.

Output

```
$ ./a.out
$ ps              obtain process status
PID   TT          STAT     TIME     COMMAND
5187  p0           Z       0:00      <exiting> the zombie child process
5149  p0           S       0:01      -csh (csh) the shell
5186   p0          S       0:00      a.out the parent process
5188  p0           R       0:00      ps

$ kill 5186       kill the parent process


$ ps    notice that the zombie is gone now
PID   TT          STAT     TIME     COMMAND
5149  p0          S        0:01      -csh (csh)
5189  p0          R        0:00     ps
```

Orphan Process

- An orphan process is a computer process whose **parent process has finished or terminated, though it remains running itself.**
- parent process ID is changed to init which is 1.
- It is possible to find the orphan process and kill by ***kill -9 pid*** command.
- It still **takes resources**, and having too many orphan process will overload init process. It is better not to have many orphan process.

Orphan Process

STAGE A

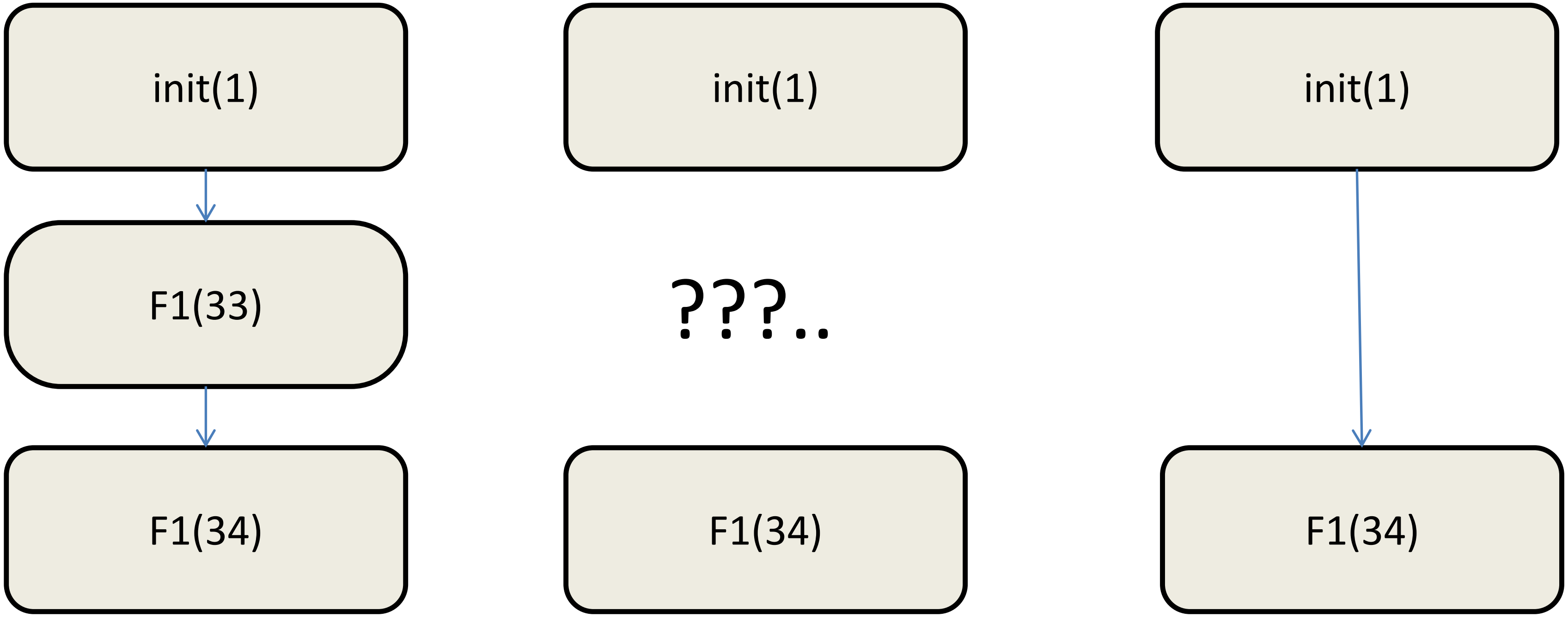
F1 (PID 33) creates child Process F1(PID 34)

STAGE B

F1 (PID 33) exits and child Process has no parent so its Orphan Process

STAGE C

F1(PID 34) has no longer parent, it is “adapted” by init(1) process, Which now becomes its parent process



init(1)

F1(33)

F1(34)

init(1)

???..

F1(34)

init(1)

F1(34)

Orphan Process

```
int main()
{
    /* Create a child process */
    int pid = fork();
      if (pid > 0)
        printf("in parent process");
        /* Note that pid is 0 in child process &
           negative if fork() fails */
    else if (pid == 0)
    {
        sleep(30);
        printf("in child process");
    }
      return 0; }
```

Orphan Processes

- When a parent dies before its child, the child is automatically adopted by the original “init” process whose PID is 1.

Orphan Process

```
main()
{
   int pid ;
   printf("I'am the original process with PID %d and PPID %d.\n",
   getpid(), getppid()) ;
   pid = fork ( ) ; /* Duplicate. Child and parent continue from here*/

   if ( pid != 0 ) /* pid is non-zero,so I must be the parent*/
   {
       printf("I'am the parent with PID %d and PPID %d.\n",
              getpid(), getppid()) ;
       printf("My child's PID is %d\n", pid ) ;
   }

   else /* pid is zero, so I must be the child */
   {
       sleep(4); /* make sure that the parent terminates first */
       printf("I'm the child with PID %d and PPID %d.\n",
              getpid(), getppid()) ;
   }
   printf ("PID %d terminates.\n", getpid()) ;
}
```

```
main() { int pid ;
printf("I'am the original process with PID %d and PPID
%d.\n", getpid(), getppid()) ;
pid = fork ( ) ; /* Duplicate. Child and parent continue
from here */
if ( pid != 0 ) /* pid is non-zero,so I must be the
   parent*/
{
printf("I'am the parent with PID %d and PPID
      %d.\n",getpid(), getppid());
printf("My child's PID is %d\n", pid );
}
else /* pid is zero, so I must be the child */
{ sleep(4); /* make sure that the parent terminates first
   */
Printf("After 10 Seconds")
printf("I'm the child with PID %d and PID .\n",getpid(),
      getppid()) ; }
printf ("PID %d terminates.\n", getpid()) ;}
```

Output

```
I'am the original process with PID 2219 and PPID 1754.

I'am the parent with PID 2219 and PPID 1754.

My child's PID is 2220

PID 2219 terminates.

After 10 seconds

I'm the child with PID 2220 and PPID 1.

PID 2220 terminates.
```

Fork Bomb

- Fork Bomb is a program which harms a system by making it run out of memory.
- It forks processes infinitely to fill memory.
- The fork bomb is a form of denial-of-service (DoS) attack against a Linux based system.
- Once a successful fork bomb has been activated in a system it may not be possible to resume normal operation without rebooting the system as the only solution to a fork bomb is to destroy all instances of it.

Fork Bomb

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

int main()
{
    while(1)
       fork();
    return 0;
}
```

Fork Bomb

fork() bomb script as below.

```
:(){ :|: & };:
```

- **Step by Step Explanation of the script:**
- **:()** means you are defining a function called :
- **{:|: &}** means run the function : and send its output to the : function again and run that in the background.
  - **:** – load another copy of the ‘:’ function into memory
  - **|** – and pipe its output to
  - **:** – another copy of ‘:’ function, which has to be loaded into memory
  - Therefore, **‘:|:’** simply gets two copies of ‘:’ loaded whenever ‘:’ is called
  - **&** – disown the functions, if the first ‘:’ is killed, all of the functions that it has started should NOT be auto-killed
  - **}** – end of what to do when we say ‘:’
- **;** Command Separator
- **:** runs the function first time

Vfork()

- System call create a new process.
- create a child process and block parent process.

Vfork() system call

```
int main() {
pid_t pid = vfork(); //creating the child process
if (pid == 0)        //if this is a child process
  {
   printf("Child process started\n");
   }
else                 //parent process execution
  {
   printf("Now I’m coming back to parent process\n");
  }
printf("finished process”);
return 0;
}
```

Output

```
Child process started

finished process

Now I’m coming back to parent process

finished process
```

Run with fork system call

```
int main() {
pid_t pid = fork();   //creating the child process
if (pid == 0)         //if this is a child process
   {
    printf("Child process started\n");
    }
else                 //parent process execution
   {
    printf("Now I’m coming back to parent process\n");
   }
printf("finished process”);
return 0;
}
```

Output

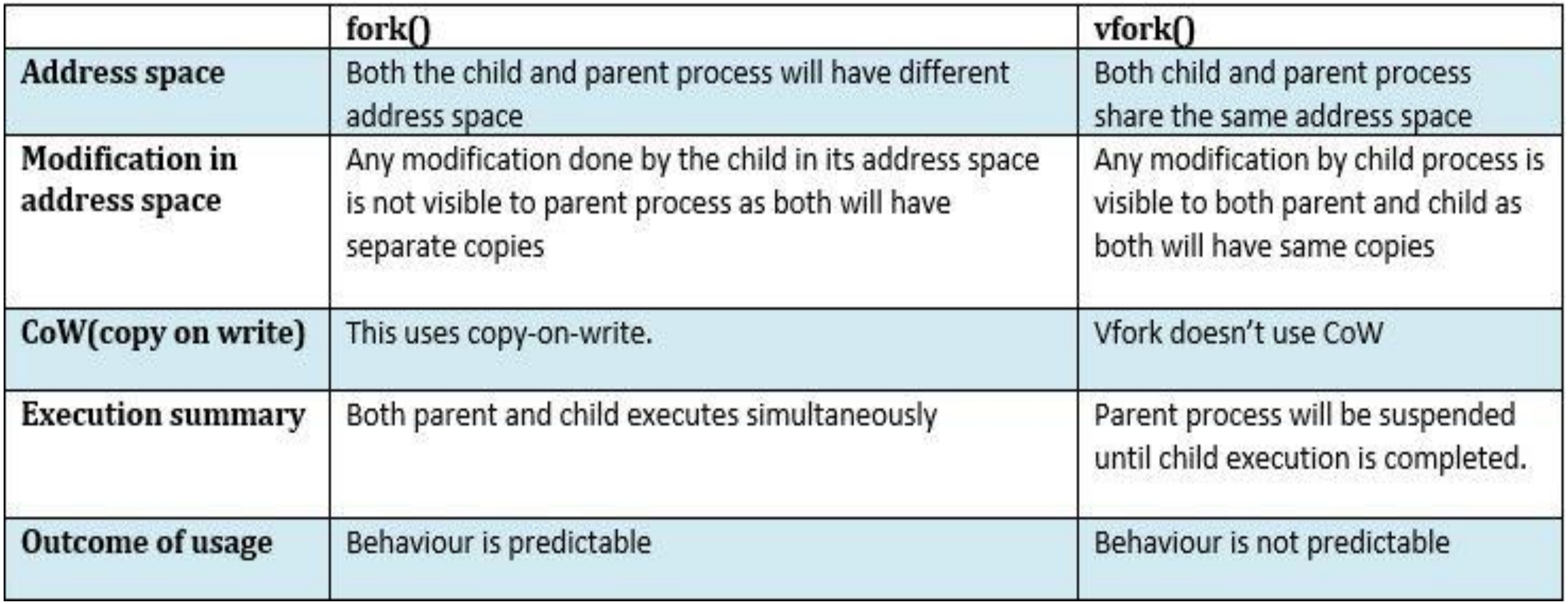
```
Now I’m coming back to parent process

finished process

Child process started

finished process
```

fork v/s vfork



	fork()	vfork()
Address space	Both the child and parent process will have different address space	Both child and parent process share the same address space
Modification in address space	Any modification done by the child in its address space is not visible to parent process as both will have separate copies	Any modification by child process is visible to both parent and child as both will have same copies
CoW(copy on write)	This uses copy-on-write.	Vfork doesn't use CoW
Execution summary	Both parent and child executes simultaneously	Parent process will be suspended until child execution is completed.
Outcome of usage	Behaviour is predictable	Behaviour is not predictable

Any Queries?