UNIX Process Creation/Termination

From Glass/Ables, “UNIX for Programmers and Users”

Name	Function
fork	duplicates a process
getpid	obtains a process' ID number
getppid	obtains a parent process' ID number
exit	terminates a process
wait	waits for a child process
exec	replaces the code, data, and stack of a process

FIGURE 13.32

UNIX process-oriented system calls.

NAME

fork - create a child process

SYNOPSIS

```
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent, except for the following points:

* The child has its own unique process ID, and this PID does not match the ID of any existing process group (**setpgid**(2)).

* The child's parent process ID is the same as the parent's process ID.

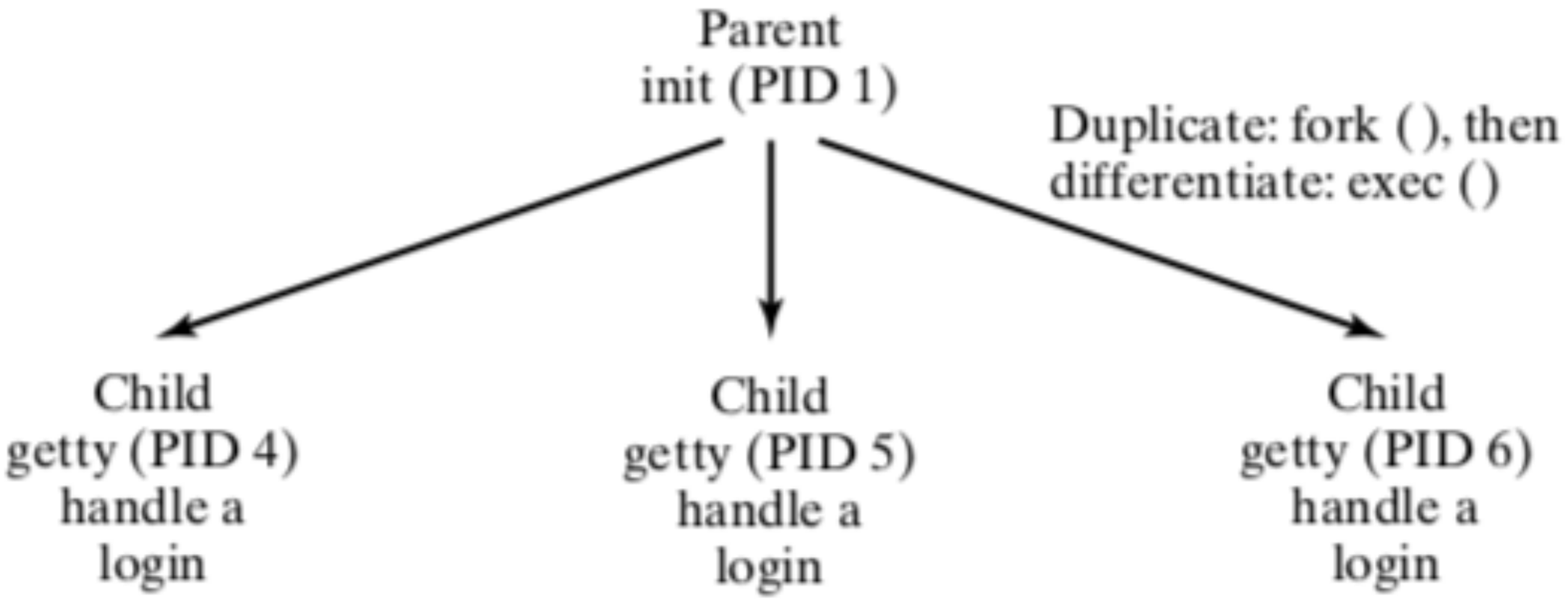

FIGURE 13.30

The initial process hierarchy.

```
#include <stdio.h>
main ()
{
 int pid, status, childPid;
 printf ("I'm the parent process and my PID is %d\n", getpid ());
 pid = fork (); /* Duplicate */
 if (pid != 0) /* Branch based on return value from fork () */
   {
     printf ("I'm the parent process with PID %d and PPID %d\n",
              getpid (), getppid ());
     childPid = wait (&status); /* Wait for a child to terminate. */
     printf ("A child with PID %d terminated with exit code %d\n",
             childPid, status >> 8);
   }
 else
   {
     printf ("I'm the child process with PID %d and PPID %d\n",
              getpid (), getppid ());
     exit (42); /* Exit with a silly number */
   }
  printf ("PID %d terminates\n", getpid () );
}
$ mywait                              ...run the program.
I'm the parent process and my PID is 13464
I'm the child process with PID 13465 and PPID 13464
I'm the parent process with PID 13464 and PPID 13409
A child with PID 13465 terminated with exit code 42
PID 13465 terminates
$ _
```

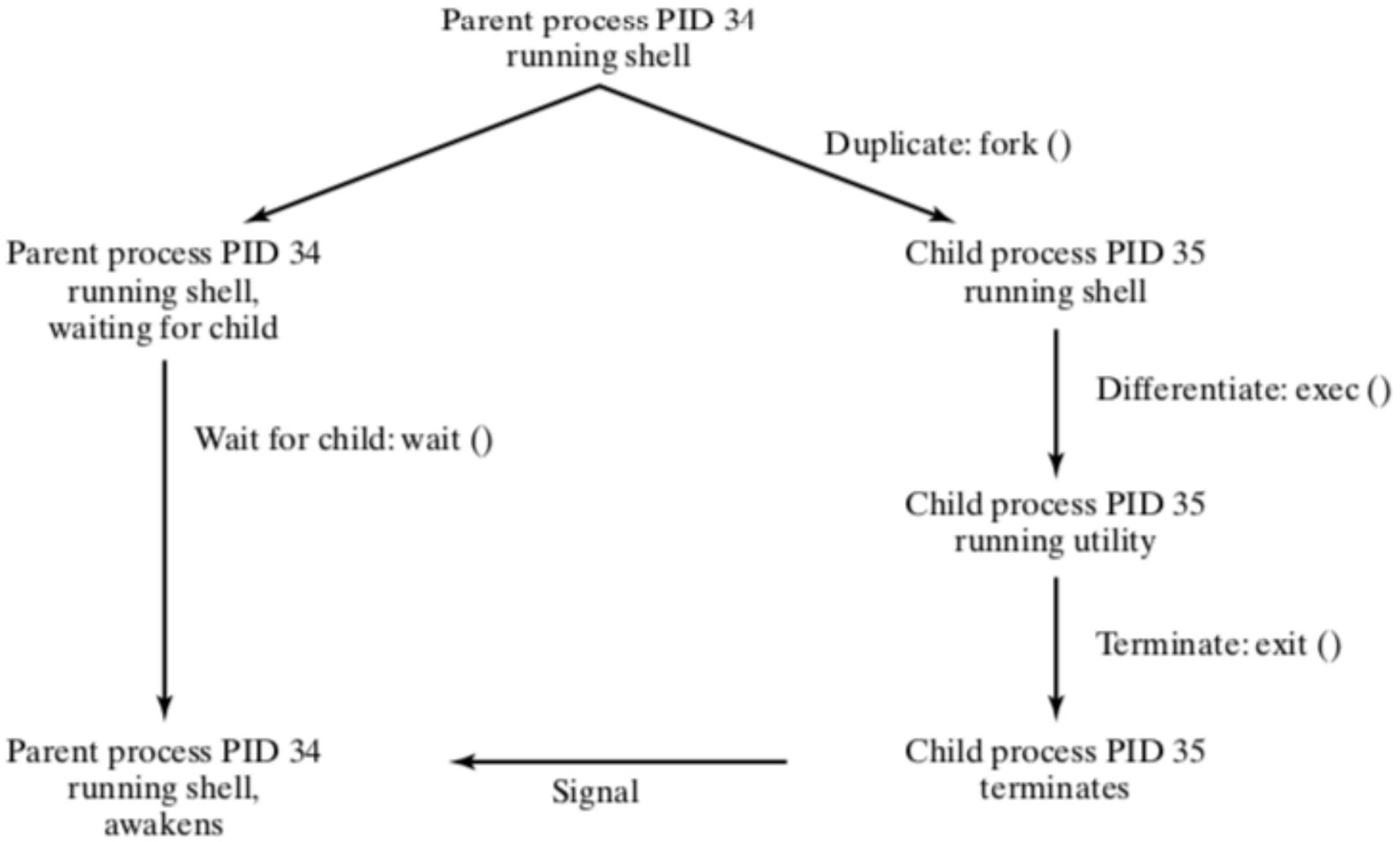

FIGURE 13.31

How a shell runs a utility.

Library Routine: int **execl** (const char* *path*, const char* *arg0*, const char* *arg1*, …, const char* *argn*, NULL)

int **execv** (const char* *path*, const char* *argv*[])

int **execlp** (const char* *path*, const char* *arg0*, const char* *arg1*,…, const char* *argn*, NULL)

int **execvp** (const char* *path*, const char* *argv*[])

The exec () family of library routines replaces the calling process' code, data, and stack from the executable file whose pathname is stored in *path*.

execl () is identical to execlp (), and execv () is identical to execvp (), except that execl () and execv () require the absolute or relative pathname of the executable file to be supplied, whereas execlp () and execvp () use the $PATH environment variable to find *path*.

If the executable file is not found, the system call returns –1; otherwise, the calling process replaces its code, data, and stack from the executable file and starts to execute the new code. A successful exec () never returns.

execl () and execlp () invoke the executable file with the string arguments pointed to by *arg1*..*argn*. *arg0* must be the name of the executable file itself, and the list of arguments must be terminated with a null.

execv () and execvp () invoke the executable file with the string arguments pointed to by *argv*[1]..*argv*[n], where *argv*[n+1] is NULL. argv[0] must be the name of the executable file itself.

FIGURE 13.38

Description of the execl (), execv (), execlp (), and execvp () library routines.

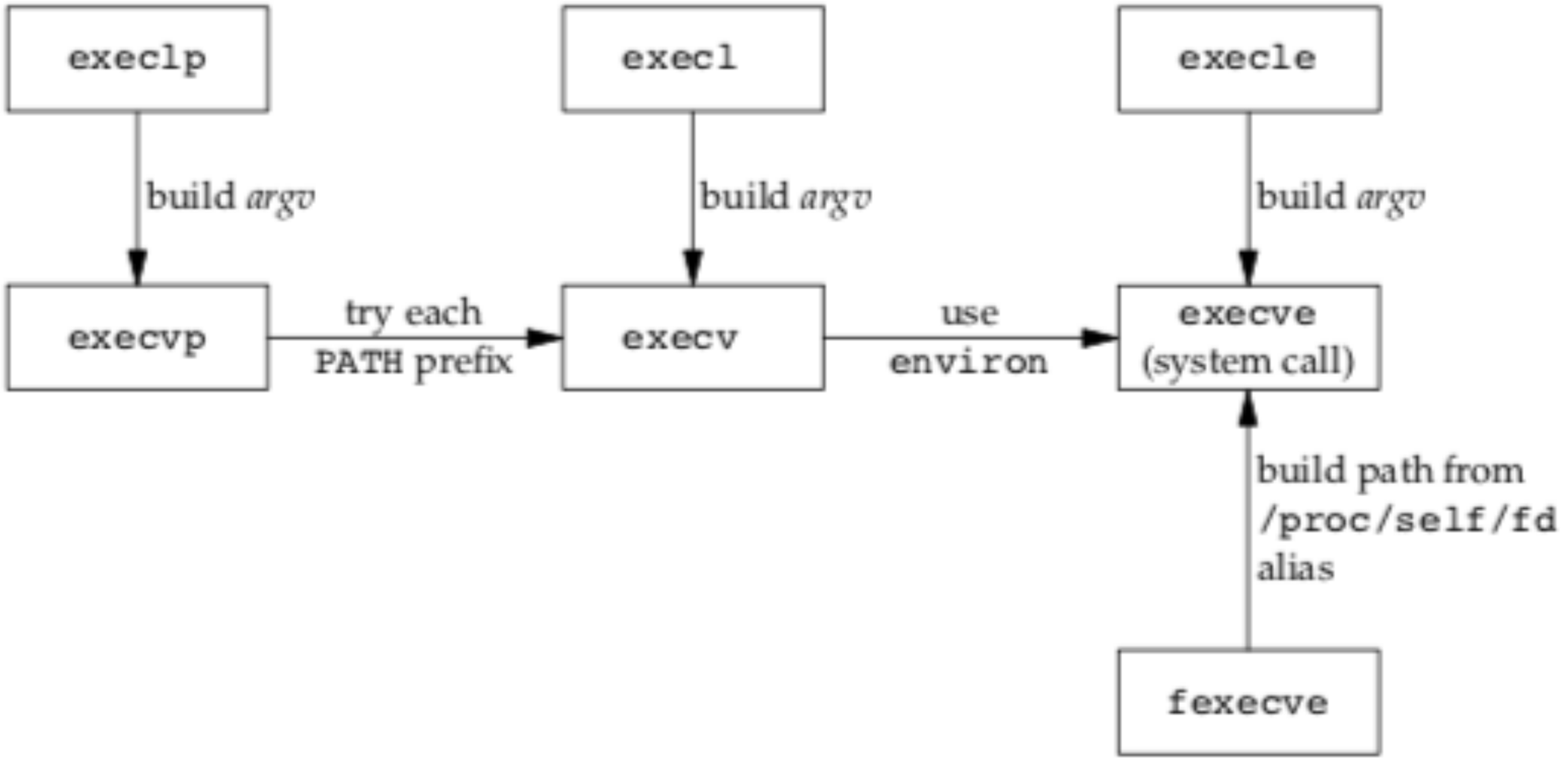

Figure 8.15 Relationship of the seven `exec` functions