Process Management

- Based on chapter 12.4 of text
- What is a process?
  - an unique instance of a running or runable program
    - code
    - data
    - stack
    - process ID

Process Management

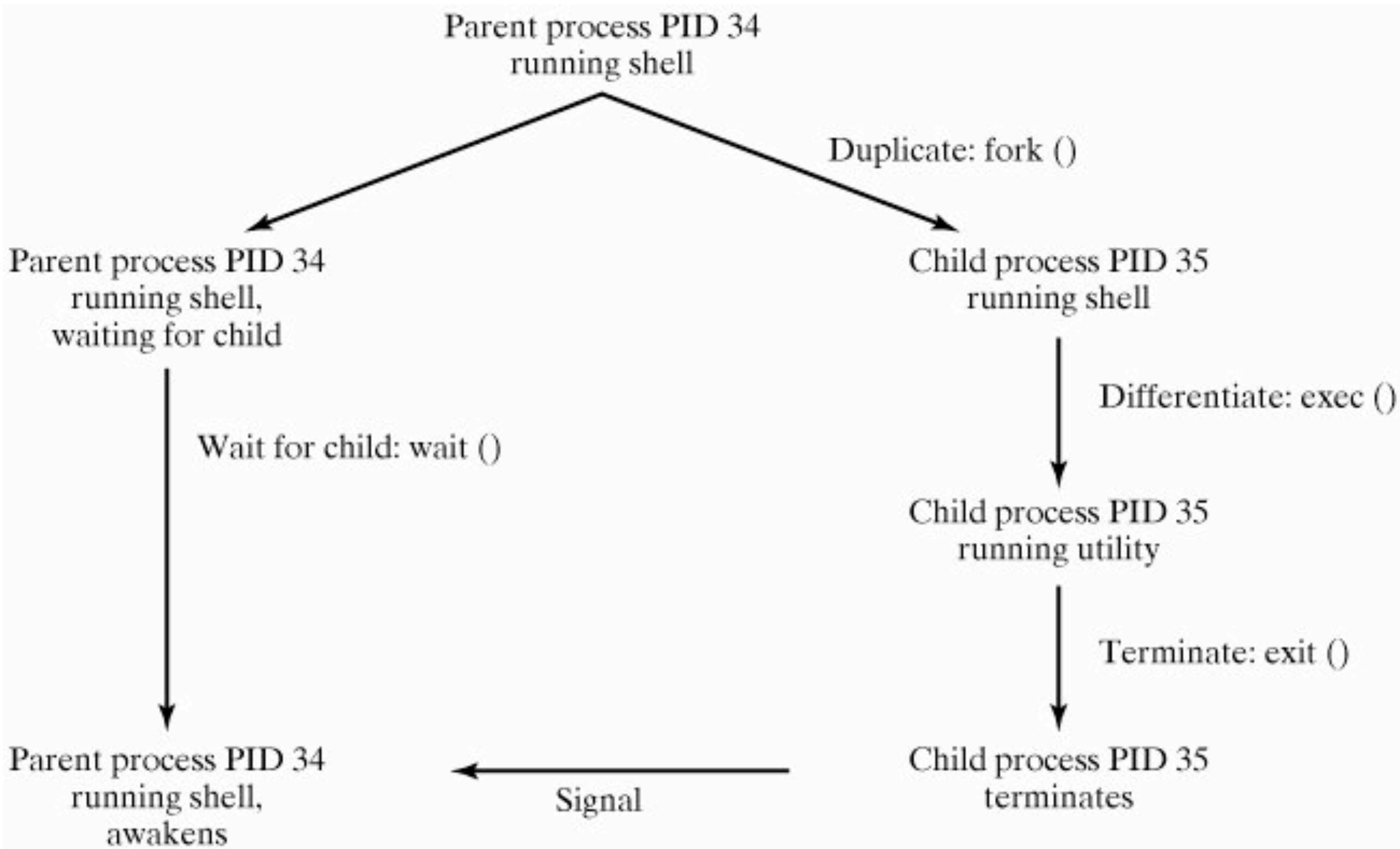
- Creating a new process
  - The only way to create a process is to duplicate an existing process
  - When Linux is started the *init* process is the only process (PID 1)
    - init is ancestors to all other processes

Process Management

- Process duplication
  - almost identical
    - same code, data, stack
  - except...
    - PID, PPID...
  - child can replace code with another executable
  - child termination is communicated back to parent

Process Management

- How a shell runs a utility (Fig 12-31)



Process Management

Figure 12-32. Linux process-oriented system calls.

Name	Function
fork	Duplicates a process.
getpid	Obtains a process's ID number.
getppid	Obtains a parent process's ID number.
exit	Terminates a process.
wait	Waits for a child process.
exec..	Replaces the code, data, and stack of a process.

Process Creation

- System Call: pid_t **fork** (void)
  - fork () causes a process to duplicate.
    - The child process is an almost-exact duplicate of the original parent process; it inherits a copy of its parent's code, data, stack, open file descriptors, and signal table.
    - However, the parent and child have different process ID numbers and parent process ID numbers.
- If fork () succeeds, it returns the PID of the child to the parent process, and returns 0 to the child process.
  - If it fails, it returns -1 to the parent process, and no child is created.

Process Creation

- Process ID
  - System Call: pid_t **getpid** (void)
  - System Call: pid_t **getppid** (void)
    - getpid () and getppid () return a process's ID and parent process's ID numbers, respectively.
    - They always succeed.
    - The parent process ID number of PID 1 is 1.

Process Creation

- fork()

```
#include <stdio.h>
main ()
{
 int pid;
 printf ("I'm 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'm the parent process 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 */
   {
     printf ("I'm the child process with PID %d and PPID %d.\n",
              getpid (), getppid ());
   }
 printf("PID %d terminates.\n",getpid ());/*Both procs execute this */
}
```

Process Creation

- now execute this code...

```
$ ./myfork                              ...run the program.
I'm the original process with PID 13292 and PPID
13273.
I'm the parent process with PID 13292 and PPID 13273.
My child's PID is 13293.
I'm the child process with PID 13293 and PPID 13292.
PID 13293 terminates.                   ...child terminates.
PID 13292 terminates.                   ...parent terminates.
$ _
```

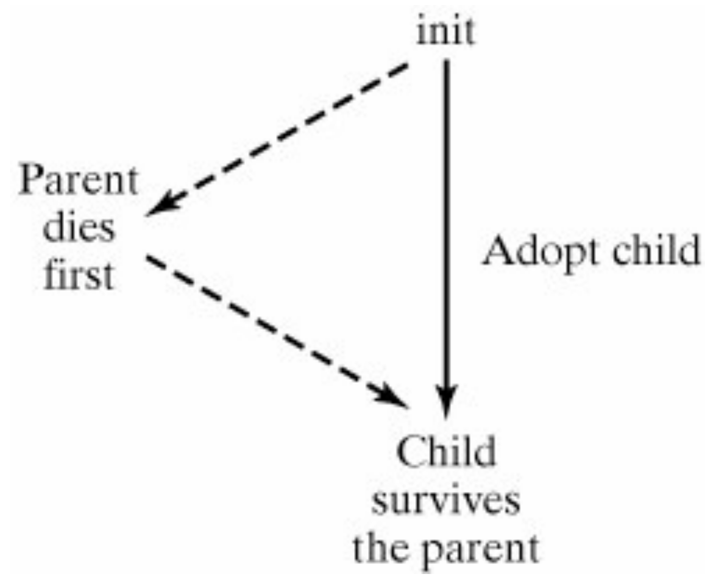
- **warning: it is dangerous for a parent to terminate without waiting for the death of its child. The only reason that the parent doesn't wait for its child in this example is because we haven't yet described the wait () system call!**

Orphan Process

- What if the parent dies before its child?
  - the child becomes an orphan
  - it is automatically adopted by the *init* process
    - recall init has PID 1

Orphan Process

- Insert some code, e.g., a sleep() command into the child to ensure that parent finished first
  - now look at the parent process IDs




```
$ ./orphan                    ...run the program.
I'm the original process with PID 13364 and PPID 13346.
I'm the parent process with PID 13364 and PPID 13346.
PID 13364 terminates.
I'm the child process with PID 13365 and PPID 1.
PID 13365 terminates.
$ _
```

Process Termination

- System Call: void **exit** (int status)
  - exit () closes all of a process's file descriptors, deallocates its code, data, and stack, and then terminates the process.
  - When a child process terminates, it sends its parent a SIGCHLD signal and waits for its termination code status to be accepted.
  - Only the lower eight bits of status are used, so values are limited to 0255.
  - The kernel ensures that all of a terminating process's children are orphaned and adopted by "init" by setting their PPID to 1.
  - The "init" process always accepts its children's termination codes. exit () never returns.

Process Termination

- **exit** () cont.
  - A parent accepts a child's termination code by executing wait (), which is described shortly.
  - A process that is waiting for its parent to accept its return code is called a zombie process
  - termination code ($status in C-shell, $? in other shells)

```
% cat myexit.c                  ...list the program.
#include <stdio.h>
main ()
{
 printf ("I'm going to exit with return code 42\n");
 exit (42);
}
% ./myexit                      ...run the program.
I'm going to exit with return code 42
% echo $status                  ...display the termination code.
42
```

Zombie Process

- What happens when parent does not accept return code?
  - What if parent terminates before child?
    - no problem, *init* adopt the orphan and always accepts the return code
  - What if parent is alive but never executes a wait()?
    - child's return code will never be accepted
    - child will remain zombie
    - A zombie process doesn't have any code, data, or stack, so it doesn't use up many system resources, but it does continue to inhabit the system's task list.

Zombie Process

- example of zombie creation

```
$ cat zombie.c                 ...list the program.
#include <stdio.h>
main ()
{
 int pid;
  pid = fork (); /* Duplicate */
 if (pid != 0) /* Branch based on return value from fork () */
    {
      while (1)  /* Never terminate, never execute a wait () */
        sleep (1000);
    }
 else
    {
      exit (42); /* Exit with a silly number */
    }
}
```

Zombie Process

- example of zombie creation

```
$ ./zombie &    ...execute the program in the background.
[1] 15896
$ ps                          ...obtain process status.
PID   TTY        TIME CMD
15870 pts2   00:00:00 bash                  ...the shell.
15896 pts2   00:00:00 zombie                ...the parent.
15897 pts2   00:00:00 zombie <defunct>      ...the zombie.
15898 pts2   00:00:00 ps
$ kill 15896                 ...kill the parent process.
[1] +   Terminated          ./zombie
$ ps                        ...notice the zombie is gone now.
PID   TTY        TIME CMD
15870 pts2   00:00:00 bash
15901 pts2   00:00:00 ps
$ _
```

- System Call: pid_t **wait** (int* status)
  - causes a process to suspend until one of its children terminates. A successful call to wait () returns the pid of the child that terminated and places a status code into *status* :
    - If the rightmost byte of status is zero, the leftmost byte contains the low eight bits of the value returned by the child's call to exit () or return ().
    - If the rightmost byte is nonzero, the rightmost seven bits are equal to the number of the signal that caused the child to terminate, and the remaining bit of the rightmost byte is set to 1 if the child produced a core dump.
    - If a process executes a wait () and has no children, wait () returns immediately with -1.
    - If a process executes a wait () and one or more of its children are already zombies, wait () returns immediately with the status of one of the zombies.

```
   $ cat mywait.c                              ...list the program.
#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
$ _
```

Process Management

- Library Function:
  - 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* arg 1,..., const char* argn, NULL)
  - int execvp (const char* path, const char* argv[ ])

Process Management

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

Process Management

- If the executable is not found, the system call returns -1; otherwise, the calling process replaces its code, data, and stack from the executable and starts to execute the new code.
  - A successful call to any of the exec system calls never returns.
- execl () and execlp () invoke the executable 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 null terminated.

Process Management

- execv () and execvp () invoke the executable 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.

Process Management

- Using the execw function

```
$ cat myexec.c                        ...list the program.
#include <stdio.h>
main ()
{
 printf ("I'm process %d and I'm about to exec an ls -l\n",getpid ());
 execl ("/bin/ls", "ls", "-l", NULL); /* Execute ls */
 printf ("This line should never be executed\n");
}


$ ./myexec                        ...run the program.
I'm process 13623 and I'm about to exec an ls -l
total 125
-rw-r--r--  1 glass    cs     277 Feb 15 00:47 myexec.c
-rwxr-xr-x  1 glass    cs   24576 Feb 15 00:48 myexec
$ _
```

Process Management

- Changing Priorities
  - process priority value
    - value between -20 and +19
    - small priority levels means the process will run faster
    - only super-user and kernel processes can have negative priority values
    - login shell has value 0

Process Management

- Being "nice"
  - child inherits parents priority value, but can change it
  - Library Function: int **nice** (int *delta*)
    - nice () adds delta to a process's current priority value. Only a super-user may specify a delta that leads to a negative priority value. Legal priority values lie between -20 and +19. If a delta is specified that takes a priority value beyond a limit, the priority value is truncated to the limit.
  - If nice () succeeds, it returns the new nice value; otherwise it returns -1. Note that this can cause problems, since a nice value of -1 is legal.

Process Management

- experiment using nice()

```
$ cat mynice.c                       ...list the source code.
#include <stdio.h>
main ()
{
 printf ("original priority\n");
 system ("ps -l"); /* Execute a ps */
 nice (0); /* Add 0 to my priority */
 printf ("running at priority 0\n");
 system ("ps -l"); /* Execute another ps */
 nice (10); /* Add 10 to my priority */
 printf ("running at priority 10\n");
 system ("ps -l"); /* Execute the last ps */
}
```

Process Management

- and run it...

```
$ mynice                         ...execute the program.
original priority
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY   CMD
0 S   500  1290  1288  0  76   0 -   552 rt_sig pts/4 ksh
0 S   500  1549  1290  0  76   0 -   583 wait4  pts/4 a.out
0 S   500  1550  1549  0  80   0 -   889 -      pts/4 ps
running at priority 0            ...adding 0 doesn't change it.
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY   CMD
0 S   500  1290  1288  0  76   0 -   552 rt_sig pts/4 ksh
0 S   500  1549  1290  0  75   0 -   583 wait4  pts/4 a.out
0 S   500  1551  1549  0  78   0 -   638 -      pts/4 ps
running at priority 10           ...adding 10 makes them run slower.
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY   CMD
0 S   500  1290  1288  0  76   0 -   552 rt_sig pts/4 ksh
0 S   500  1549  1290  0  90  10 -   583 wait4  pts/4 a.out
0 S   500  1552  1549  0  87  10 -   694        pts/4 ps
$ _
```

Process Management

- get real or effective ID of process: System Call:
  - uid_t **getuid** ()
  - uid_t **geteuid** ()
  - gid_t **getgid** ()
  - gid_t **getegid** ()
  - getuid () and geteuid () return the calling process's real and effective user ID, respectively. getgid () and getegid () return the calling process's real and effective group ID, respectively. The ID numbers correspond to the user and group IDs listed in the "/etc/passwd" and "/etc/group" files.
  - These calls always succeed.

Process Management

- set real or effective ID of process: System Call:
  - int setuid (uid_t id)
  - int seteuid (uid_t id)
  - int setgid (gid_t id)
  - int setegid (gid_t id)
    - seteuid () and setegid () set the calling process's effective user and group ID, respectively. setuid () and setgid () set the calling process's effective and real user and group ID, respectively, to the specified value.
    - These calls succeed only if executed by a super-user, or if id is the real or effective user (group) ID of the calling process. They return 0 if successful; otherwise, they return -1.

Process Management

- sample program: background processing

```
$ cat background.c                    ...list the program.
#include <stdio.h>
main (argc, argv)
int argc;
char* argv [];
{
 if (fork () == 0) /* Child */
   {
     execvp (argv[1], &argv[1]); /* Execute other program */
     fprintf (stderr, "Could not execute %s\n", argv[1]);
   }
}
$ background sleep 60   ...run the program.
$ ps                    ...confirm that it is in background.
PID   TTY          TIME CMD
10742 pts0     00:00:00 bash
10936 pts0     00:00:01 ksh
15669 pts0     00:00:00 csh
16073 pts0     00:00:00 sleep 60
16074 pts0     00:00:00 ps
$ _
```

Process Management

- Redirection
  - When a process forks, the child inherits a copy of its parent's file descriptors.
  - When a process execs, all non-close-on-exec file descriptors remain unaffected, including the standard input, output, and error channels.
  - The Linux shells use these two pieces of information to implement redirection.

Process Management

- For example, say you type the following command at a terminal:
  ls > ls.out
  - The parent shell forks and then waits for child shell to terminate.
  - The child shell opens the file "ls.out," creating it or truncating it as necessary.
  - The child shell then duplicates the file descriptor of "ls.out" to the standard output file descriptor, number 1, and then closes the original descriptor of "ls.out". All standard output is therefore redirected to "ls.out".
  - The child shell then exec's the ls utility. Since the file descriptors are inherited during an exec (), all of the standard output of ls goes to "ls.out".
  - When the child shell terminates, the parent resumes. The parent's file descriptors are unaffected by the child's actions, as each process maintains its own private descriptor table.

Process Management

- example

```
$ cat redirect.c                        ...list the program.
#include <stdio.h>
#include <fcntl.h>
main (argc, argv)
int argc;
char* argv [];
{
 int fd;
  /* Open file for redirection */
 fd = open (argv[1], O_CREAT | O_TRUNC | O_WRONLY, 0600);
 dup2 (fd, 1); /* Duplicate descriptor to standard output */
 close (fd); /* Close original descriptor to save descriptor space */
 execvp (argv[2], &argv[2]); /* Invoke program; will inherit stdout */
 perror ("main"); /* Should never execute */
}
$ redirect ls.out ls -lG      ...redirect "ls -lG" to "ls.out".
$ cat ls.out                            ...list the output file.
total 5
-rw-r-xr-x   1 glass         0 Feb 15 10:35 ls.out
-rw-r-xr-x   1 glass       449 Feb 15 10:35 redirect.c
-rwxr-xr-x   1 glass      3697 Feb 15 10:33 redirect
$ _
```