Chapter 7.3

Process System Calls

Processes and Programs

A process is a running instance of a program. There may be many processes of a given program.

Linux allows for several programs to manage processes.

- `ps` – Shows all running processes.
- `top` – Monitors resource usage of processes.
- `kill` – Sends a signal to a process to terminate.

`Kill` Example

Run the following in a terminal:

```
ps -eaf | more
```

Now run in second shell:

```
ps -eaf | grep more
```

Now execute in second shell:

```
kill ####
```

where `####` is the Processes PID.

Top Example

Run the following in a terminal:

```
top
```

Run the following in another terminal:

```
gcc top.c -lm
a.out
```

Process Shell Commands

Several commands help alter how a process behaves while running.

- `&` – Run program in background so that `stdin` in "disconnected."
- `CTRL-Z` – Suspend process currently connected to `stdin`.
- `bg` – Resume suspended process in background. (`stdin` still disconnected.)
- `fg` – Resume suspended process in foreground. (`stdin` connected.)

& Example

Run the following in a terminal:

```
gcc top2.c -lm
a.out &
```

Run the following in another terminal:

```
top
```

Process System Calls – `fork()`

The system call `fork()` is used to "spawn" one process ("child") from another ("parent").

Consider the following code from `fork1.c`

```
printf("About to 'fork`...\n");
i = fork();
printf("Fork returned %d\n", i);
while (1);
```

```
fork1 &
ps -eaf
```

Process System Calls – `fork()`

What is the difference in the following code?

Consider `fork2.c`

```
printf("About to 'fork`...\n");
i = fork();
printf("Fork returned %d\n", i);
if (i != 0) while (1);
```

```
fork2 &
ps -eaf
```

Process System Calls – `fork()`

What will this program `fork3.c` do?

```
i = fork();
for (j=0; j<10; j++)
{ if (i == 0) /* child process */
  { printf("Why?");
    sleep(2);
  }
  else            /* parent process */
  { printf("Because I said so.\n");
    sleep(1);
  }
}
```

Process System Calls – `getpid()`

We can use `getpid()` to find out the process id of a task. What will this program do?

getpid1.c getpid2.c

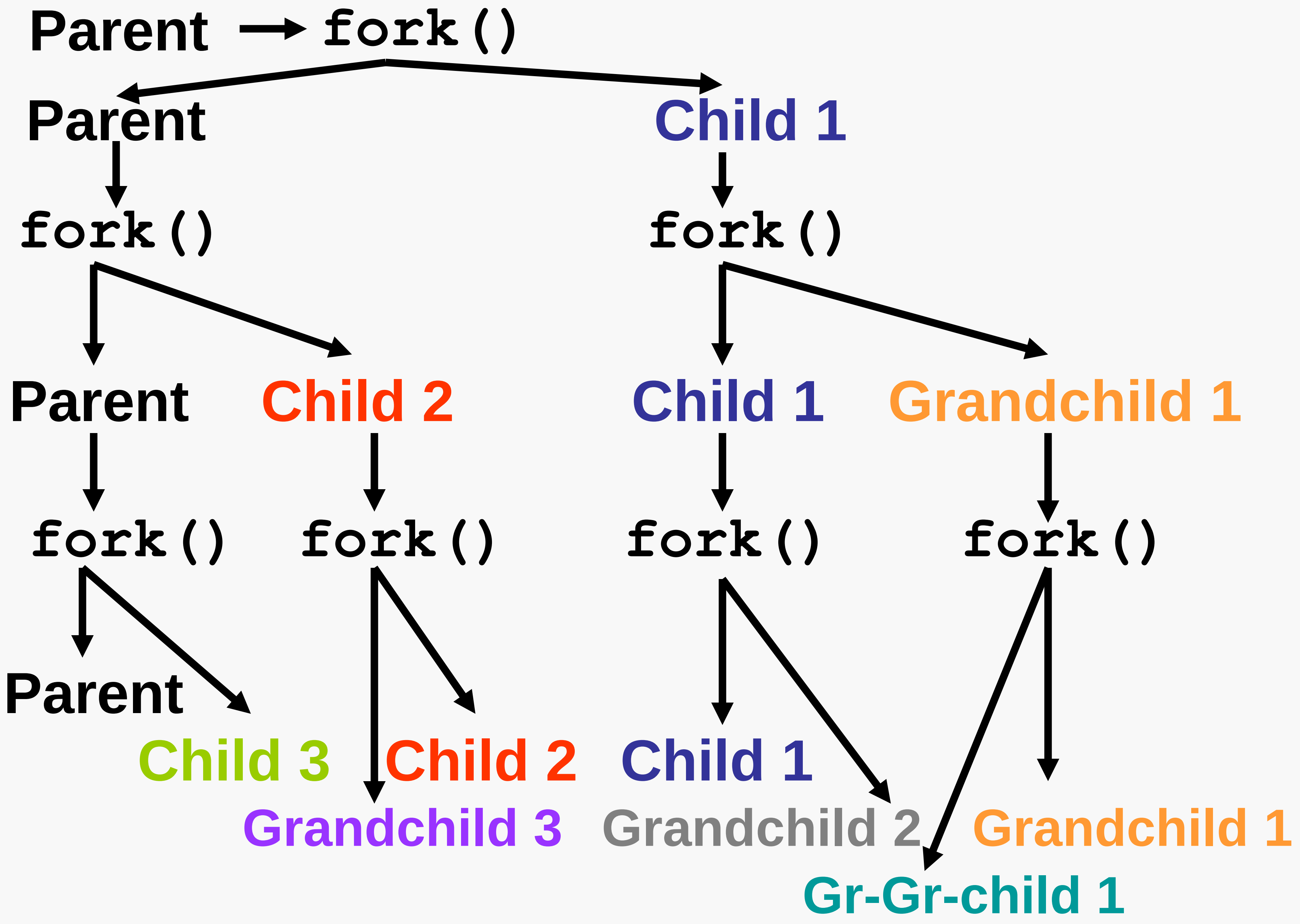
```
int pid;

pid = getpid();
printf("Begin: Pid is %d\n", pid);
fork();
fork();
fork();
pid = getpid();
printf("End:   Pid is %d\n", pid);
```

Process System Calls – fork()

Maybe the addition of the following lines in program getpid3.c will help clarify the function.

```
pid = getpid();
printf("Begin: Pid is %d\n", pid);
fork(); printf(“%d”, getpid());
fork(); printf(“%d”, getpid());
fork();
printf("End:Pid is %d\n", getpid());
```


Parent
fork()
Parent
Child 1
fork()
fork()
Parent
Child 2
Child 1
Grandchild 1
fork()
fork()
fork()
fork()
Parent
Child 3
Child 2
Child 1
Grandchild 3
Grandchild 2
Grandchild 1
Gr-Gr-child 1

Process System Calls – `fork()`

What’s the problem with the following program `forktiming.c`?

```
switch (i=fork())
{ case 0: // Child
    for (j=0; j<TIRED; j++)
    { if (!IceCream)
      { printf("I won't eat spinach!!!\n");
        sleep(1);
      }
    }
    break;
    default: // Parent
      printf("Here's your Ice Cream.\n");
}
```

Process System Calls – `wait()`

The function `wait()` can be used to wait for a child process to finish.

wait1.c

```
void DoWhatParentsDo(int pid)
{ int wait_ret;

  wait_ret = wait(NULL);
  printf("Waited for %d. Return = %d\n",
                                   pid, wait_ret);
  printf("Here's your Ice Cream.\n");
}
```

Process System Calls – wait()

What if `wait()` *is called with a non-*`NULL` *argument?*

`wait2.c`

```
wait_ret = wait(&child_status);
printf("Waited for %d. Ret. val. = %d\n",
                           pid, wait_ret);
exit_byte = child_status >> 8;
/* bbbb bbbb ---- ---- */
signal_byte = child_status & 0x7F;
/* ---- ---- 0bbb bbbb */
core_bit = child_status & 0x80;
/* ---- ---- b--- --- */
```

Process System Calls – `wait()`

How do we wait for more than one process to finish?

`wait3.c`

```
if (fork() != 0)
 { if (fork() != 0)
   { if (fork() != 0)
     { wait_ret = wait(NULL);
       printf("Waited for %d to finish.\n", wait_ret);

       wait_ret = wait(NULL);
       printf("Waited for %d to finish.\n", wait_ret);

       wait_ret = wait(NULL);
       printf("Waited for %d to finish.\n", wait_ret);
     }
     else DoThirdChild();
   }
   else DoSecondChild();
 }
 else DoFirstChild();
```

Process System Calls – `kill()`

Consider one more example of `fork()` using PID of child.

`kill.c`

```
sleep(rand() % 20);
printf("You lose sucker...\n");
kill(pid, 1);
wait_ret = wait(&child_status);
```

System Calls – `system()`

The C library contains a function called `system()` which functions as a `fork()` and `wait()`.

system1.c

```
printf(“Making a 'system` call... \n");
strcpy(text,"ls -l");
system(text);
printf("\nDid it work?\n");
sleep(4);
printf("Indeed it did.\n");
```

System Calls – `system()` 2

Now consider one more example: `system2.c` calling `sysdemo.c`.

```
printf("\nSystem2 Running...\n");
j = rand() % 5;
for (i=0; i<j; i++)
{ k = rand() % 10 + 2000;
  sprintf(text, "sysdemo %d", k);
  printf("About to call '%s'...\n\n", text);
  system(text);
  printf("\nSysdemo called and finished.\n");
}
printf("\nSystem2 Finished.\n");
```

System Calls – execvp()

The C library also contains a function called execvp() which functions similarly to system() except that it does not spawn a new process but simply switches completely to another process and does not finish running any more code from the calling process.

execvp1.c

```
printf("About to call 'execvp'");
i = execvp(program, arg);
printf("Finished calling %s\n", program);
```