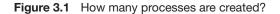


Processes

## **Exercises**

- **3.12** Describe the actions taken by a kernel to context-switch between processes.
- 3.13 Construct a process tree similar to Figure 3.7. To obtain process information for the UNIX or Linux system, use the command ps -ael. Use the command man ps to get more information about the ps command. The task manager on Windows systems does not provide the parent process ID, but the *process monitor* tool, available from technet.microsoft.com, provides a process-tree tool.
- **3.14** Explain the role of the init (or systemd) process on UNIX and Linux systems in regard to process termination.
- **3.15** Including the initial parent process, how many processes are created by the program shown in Figure 3.32?

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    int i;
    for (i = 0; i < 4; i++)
        fork();
    return 0;
}</pre>
```

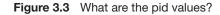


## 12 Chapter 3 Processes

- **3.16** Explain the circumstances under which the line of code marked printf("LINE J") in Figure 3.33 will be reached.
- **3.17** Using the program in Figure 3.34, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)
- **3.18** Give an example of a situation in which ordinary pipes are more suitable than named pipes and an example of a situation in which named pipes are more suitable than ordinary pipes.
- **3.19** Consider the RPC mechanism. Describe the undesirable consequences that could arise from not enforcing either the "at most once" or "exactly once" semantic. Describe possible uses for a mechanism that has neither of these guarantees.
- **3.20** Using the program shown in Figure 3.35, explain what the output will be at lines X and Y.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
pid_t pid;
   /* fork a child process */
   pid = fork();
   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
      printf("LINE J");
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }
   return 0;
}
```

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
ł
pid_t pid, pid1;
   /* fork a child process */
   pid = fork();
   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      pid1 = getpid();
      printf("child: pid = %d",pid); /* A */
      printf("child: pid1 = %d",pid1); /* B */
   }
   else { /* parent process */
      pid1 = getpid();
      printf("parent: pid = %d",pid); /* C */
      printf("parent: pid1 = %d",pid1); /* D */
      wait(NULL);
   }
   return 0;
}
```



- **3.21** What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer level.
  - a. Synchronous and asynchronous communication
  - b. Automatic and explicit buffering
  - c. Send by copy and send by reference
  - d. Fixed-sized and variable-sized messages

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#define SIZE 5
int nums[SIZE] = {0,1,2,3,4};
int main()
int i;
pid_t pid;
  pid = fork();
  if (pid == 0) {
     for (i = 0; i < SIZE; i++) {
       nums[i] *= -i;
       printf("CHILD: %d ",nums[i]); /* LINE X */
     }
  }
  else if (pid > 0) {
     wait(NULL);
     for (i = 0; i < SIZE; i++)</pre>
       printf("PARENT: %d ",nums[i]); /* LINE Y */
  }
  return 0;
}
```

Figure 3.4 What output will be at Line X and Line Y?