Practice Exercises

Q1: Using the program shown in Figure 3.30, explain what the output will be at LINE A.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int value = 5;
int main()
{
pid_t pid;
pid = fork();
if (pid == 0) { /* child process */
value += 15;
return 0;
}
else if (pid > 0) { /* parent process */
wait(NULL);
printf("PARENT: value = %d",value); /* LINE A */
return 0;
}
}
```

The value of line A will be (PARENT: value = 5). Because the child process will get a copy of the parent code and data without affecting the parent data. So, the child process only changes its copy.

Q2: Including the initial parent process, how many processes are created by the program shown in Figure 3.31?

```
#include <stdio.h>
#include <unistd.h>
int main()
{
/* fork a child process */
fork();
/* fork another child process */
fork();
/* and fork another */
fork();
return 0;
}
```

8 processes are created. (2*2*2 - 1) = 7 child processes.

Q3: Original versions of Apple's mobile iOS operating system provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.

a. The CPU scheduler must be aware of the different concurrent processes and must choose an appropriate algorithm that schedules the concurrent processes.

b. Concurrent processes may need to communicate with one another, and the operating system must therefore develop one or more methods for providing interprocess communication.

c. Because mobile devices often have limited memory, a process that manages memory poorly will have an overall negative impact on other concurrent processes. The operating system must therefore manage memory to support multiple concurrent processes.

Q4: Some computer systems provide multiple register sets. Describe what happens when a context switch occurs if the new context is already loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use?

a. The CPU current-register-set pointer will point to the register set of the new context, which takes a very small amount of time.

b. If the context is in memory, one of the contexts in a register set must be chosen and be moved to memory, and the new context must be loaded from memory into the set. This process takes a little more time.

Q5: When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process?

a. Stack

b. Heap

c. Shared memory segments

The Shared memory segments are only the shared things between the parent and the child processes. Stack and heap are copied from the parent process to the child process.

Q6: Consider the "exactly once"semantic with respect to the RPC mechanism. Does the algorithm for implementing this semantic execute correctly even if the ACK message sent back to the client is lost due to a network problem? Describe the sequence of messages, and discuss whether "exactly once" is still preserved.

Yes. Because the client will continually try to send data to the server until the ACK message appears to the client.

The general strategy is for the client to send the RPC to the server along with a timestamp. The client will also start a timeout clock. The client will then wait for one of two occurrences: (1) it will receive an ACK from the server indicating that the remote procedure was performed, or (2) it will time out. If the client times out, it assumes the server was unable to perform the remote procedure, so the client invokes the RPC a second time, sending a later timestamp. The client may not receive the ACK for one of two reasons: (1) the original RPC was never received by the server, or (2) the RPC was correctly received—and performed—by the server but the ACK was lost. In situation (1), the use of ACKs allows the server ultimately to receive and perform the RPC. In situation (2), the server will receive a duplicate RPC, and it will use the timestamp to identify it as a duplicate so as not to perform the RPC a second time. It is important to note that the server must send a second ACK back to the client to inform the client the RPC has been performed.

Q7: Assume that a distributed system is susceptible to server failure. What mechanisms would be required to guarantee the “exactly once” semantics for execution of RPCs?

The server should keep track in stable storage (such as a disk log) of information regarding what RPC operations were received, whether they were successfully performed, and the results associated with the operations. When a server crash takes place and an RPC message is received, the server can check whether the RPC has been previously performed and therefore guarantee “exactly once” semantics for the execution of RPCs.

Exercises

Q8: Describe the actions taken by a kernel to context-switch between processes.

The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state, and memory-management information. When a context switch occurs, the Kernel saves the context of the old process in its PCP and loads the saved context of the new process scheduled to run.

Q9: 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.

The tree will be so huge to include in this file. To construct this tree automatically instead of drawing it manually, you could use the command for linux “pstree”.

Q10: Explain the role of the init (or systemd) process on UNIX and Linux systems in regard to process termination.

The “init” or “systemd” processes are the first process initiated by the operating system and every other process is a child to either of these processes. If the process didn’t invoke wait() and instead terminated, thereby leaving its child processes as orphans. Traditional UNIX systems addressed this scenario by assigning the init process as the new parent to the orphan processes. The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan’s process identifier and process-table entry.

Q11: 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;
}
```

There are 16 processes including the parent process. Children processes = 2^4 - 1 = 15.

Q12: Explain the circumstances under which the line of code marked printf("LINE J") in Figure 3.33 will be reached.

```
#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;
}
```

The parent process must invoke fork correctly returning the pid of the new child process. The child process will copy the code and data of the parent then starts executing from the next instruction which has the turn. Checking for different values of the pid, so, when the child process returns 0 for the pid it will execute LINE J.

Q13: 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.)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
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;
}
```

child: pid = 0
child: pid1 = 2603
parent: pid = 2603
parent: pid1 = 2600

Q14: 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.

Example 1: Assume we have a process that counts characters in a file. An ordinary pipe can be used where the producer writes the file to the pipe and the consumer reads the files and counts the number of characters in the file.

Example 2: Consider the situation where several processes may write messages to a log. When processes wish to write a message to the log, they write it to the named pipe.

Q15: 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.

An important issue involves the semantics of a call. Whereas local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, as a result of common network error.

Q16: 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>
#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++)
printf("PARENT: %d ",nums[i]); /* LINE Y */
}
return 0;
}
```

CHILD: 0 CHILD: -1 CHILD: -4 CHILD: -9 CHILD: -16 PARENT: 0 PARENT: 1 PARENT: 2 PARENT: 3 PARENT: 4

Q17: 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:

A benefit of synchronous communication is that it allows a rendezvous between the sender and receiver. A disadvantage of a blocking send is that a rendezvous may not be required and the message could be delivered asynchronously. As a result, message-passing systems often provide both forms of synchronization.

b. Automatic and explicit buffering:

Automatic buffering provides a queue with indefinite length, thus ensuring the sender will never have to block while waiting to copy a message. There are no specifications on how automatic buffering will be provided; one scheme may reserve sufficiently large memory where much of the memory is wasted. Explicit buffering specifies how large the buffer is. In this situation, the sender may be blocked while waiting for available space in the queue. However, it is less likely that memory will be wasted with explicit buffering.

c. Send by copy and send by reference:

Send by copy does not allow the receiver to alter the state of the parameter; send by reference does allow it. A benefit of send by reference is that it allows the programmer to write a distributed version of a centralized application. Java's RMI provides both; however, passing a parameter by reference requires declaring the parameter as a remote object as well.

d. Fixed-sized and variable-sized messages

The implications of this are mostly related to buffering issues; with fixed-size messages, a buffer with a specific size can hold a known number of messages. The number of variable-sized messages that can be held by such a buffer is unknown. Consider how Windows 2000 handles this situation: with fixed-sized messages (anything < 256 bytes), the messages are copied from the address space of the sender to the address space of the receiving process. Larger messages (i.e. variable-sized messages) use shared memory to pass the message.