Creación de procesos en XINU

2.5 Distinction Between Sequential And Concurrent Programs

When a programmer creates a conventional (i.e., sequential) program, the programmer imagines a single processor executing the program step-by-step without interruption or interference. When writing code that will be executed concurrently, however, a programmer must take a different view and imagine multiple computations executing simultaneously. The code inside an operating system provides an excellent example of code that must accommodate concurrency. At any given instant, multiple processes may be executing. In the simplest case, each process executes application code that no other process is executing. However, an operating system designer must plan for a situation in which multiple processes have invoked a single operating system function, or even a case where multiple processes are executing the same instruction. To further complicate matters, the operating system may switch the processor among processes at any time; in a multitasking system, no guarantee can be made about the relative speed at which a given computation will proceed.

Designing code to operate correctly in a concurrent environment provides a tough intellectual challenge because a programmer must ensure that all processes perform the intended function correctly, no matter what operating system code they execute or in which order they execute. We will see how the notion of concurrent execution affects each line of code in an operating system.

To understand applications in a concurrent environment, consider the Xinu model. When it boots, Xinu creates a single concurrent process that starts running the main pro-

gram. The initial process can continue execution by itself, or it can create additional processes. When a new process is created, the original process continues to execute, and the new process executes concurrently. Either the original process or a new process can create additional processes that execute concurrently.

As an example, consider the code for a concurrent main program that creates two additional processes. Each of the two new processes sends characters over the console serial device: the first process sends the letter *A*, and the second sends the letter *B*. File *ex2.c* contains the source code, which consists of a main program and two functions, *sndA* and *sndB*.

```
/* ex2.c - main, sndA, sndB */

#include <xinu.h>

void    sndA(void), sndB(void);

/*------------------------------------------------------------------------
 * main  -  Example of creating processes in Xinu
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndA, 1024, 20, "process 1", 0) );
        resume( create(sndB, 1024, 20, "process 2", 0) );
}

/*------------------------------------------------------------------------
 * sndA  -  Repeatedly emit 'A' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndA(void)
{
        while( 1 )
                putc(CONSOLE, 'A');
}

/*------------------------------------------------------------------------
 * sndB  -  Repeatedly emit 'B' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndB(void)
{
        while( 1 )
                putc(CONSOLE, 'B');
}
```

In the code, the main program never calls either function directly. Instead, the main program calls two operating system functions, *create* and *resume*. Each call to *create* forms a new process that will begin executing instructions at the address specified by its first argument. In the example, the first call to *create* passes the address of function *sndA*, and the second call passes the address of function *sndB*.† Thus, the code creates a process to execute *sndA* and a process to execute *sndB*. *Create* establishes a process, leaves the process ready to execute but temporarily suspended, and returns an integer value that is known as a *process identifier* or *process ID*. The operating system uses the process ID to identify the newly created process; an application uses the process ID to reference the process. In the example, the main program passes the ID returned by *create* to *resume* as an argument. *Resume* starts (i.e., unsuspends) the process, allowing the process to begin execution. The distinction between normal function calls and process creation is:

> *A normal function call does not return until the called function completes. Process creation functions* create *and* resume *return to the caller immediately after starting a new process, which allows execution of both the existing process and the new process to proceed concurrently.*

In Xinu, all processes execute concurrently. That is, execution of a given process continues independent of other processes unless a programmer explicitly controls interactions among processes. In the example, the first new process executes code in function *sndA*, sending the letter *A* continuously, and the second executes code in function *sndB*, sending the letter *B* continuously. Because the processes execute concurrently, the output is a mixture of *A*s and *B*s.

What happens to the main program? Remember that in an operating system, each computation corresponds to a process. Therefore, we should ask, "What happens to the process executing the main program?" Because it has reached the end of the main program, the process executing the main program exits after the second call to *resume*. Its exit does not affect the newly created processes — they continue to send *A*s and *B*s. A later section describes process termination in more detail.

2.6 Multiple Processes Sharing A Single Piece Of Code

The example in file *ex2.c* shows each process executing a separate function. It is possible, however, for multiple processes to execute the same function. Arranging for processes to share code can be essential in an embedded system that has a small memory. To see an example of processes sharing code, consider the program in file *ex3.c*.

†Other arguments to *create* specify the stack space needed, a scheduling priority, a name for the process, the count of arguments passed to the process, and (when applicable) the argument values passed to the process; we will see details later.

```
/* ex3.c - main, sndch */

#include <xinu.h>

void    sndch(char);

/*------------------------------------------------------------------------
 * main  -  Example of 2 processes executing the same code concurrently
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndch, 1024, 20, "send A", 1, 'A') );
        resume( create(sndch, 1024, 20, "send B", 1, 'B') );
}

/*------------------------------------------------------------------------
 * sndch  -  Output a character on a serial device indefinitely
 *------------------------------------------------------------------------
 */
void    sndch(
          char  ch              /* The character to emit continuously   */
        )
{
        while ( 1 )
                putc(CONSOLE, ch);
}
```

As in the previous example, a single process begins executing the main program. The process calls *create* twice to start two new processes that each execute code from function *sndch*. The final two arguments in the call to *create* specify that *create* will pass one argument to the newly created process and a value for the argument. Thus, the first process receives the character *A* as an argument, and the second process receives character *B*.

Although they execute the same code, the two processes proceed concurrently without any effect on one another. In particular, each process has its own copy of arguments and local variables. Thus, one process emits *A*s, while the other process emits *B*s. The key point is:

> *A program consists of code executed by a single process of control. In contrast, concurrent processes are not uniquely associated with a piece of code; multiple processes can execute the same code simultaneously.*

The examples provide a hint of the difficulty involved in designing an operating system. Not only must each piece be designed to operate correctly by itself, the designer must also guarantee that multiple processes can execute a given piece of code concurrently without interfering with one another.

Although processes can share code and global variables, each process must have a private copy of local variables. To understand why, consider the chaos that would result if all processes shared every variable. If two processes tried to use a shared variable as the index of a *for* loop, for example, one process might change the value while another process was in the midst of executing the loop. To avoid such interference, the operating system creates an independent set of local variables for each process.

Function *create* also allocates an independent set of arguments for each process, as the example in file *ex3.c* demonstrates. In the call to *create*, the last two arguments specify a count of values that follow (1 in the example), and the value that the operating system passes to the newly created process. In the code, the first new process has character *A* as an argument, and the process begins execution with formal parameter *ch* set to *A*. The second new process begins with *ch* set to *B*. Thus, the output contains a mixture of both letters. The example points out a significant difference between the sequential and concurrent programming models.

Storage for local variables, function arguments, and a function call stack is associated with the process executing a function, not with the code for the function.

The important point is: an operating system must allocate additional storage for each process, even if the process shares the same code that another process or processes are using. As a consequence, the amount of memory available limits the number of processes that can be created.

2.7 Process Exit And Process Termination

The example in file *ex3.c* consists of a concurrent program with three processes: the initial process and the two processes that were started with the system call *create*. Recall that when it reaches the end of the code in the main program, the initial process ceases execution. We use the term *process exit* to describe the situation. Each process begins execution at the start of a function. A process can exit by reaching the end of the function or by executing a return statement in the function in which it starts. Once a process exits, it disappears from the system; there is simply one less computation in progress.

Process exit should not be confused with normal function call and return or with recursive function calls. Like a sequential program, each process has its own stack of function calls. Whenever it executes a call, an activation record for the called function is pushed onto the stack. Whenever it returns, a function's activation record is popped

off the stack. Process exit occurs only when the process pops the last activation record (the one that corresponds to the top-level function in which the process started) off its stack.

The system routine *kill* provides a mechanism to *terminate* a process without waiting for the process to exit. In a sense, *kill* is the inverse of *create* — *kill* takes a process ID as an argument, and removes the specified process immediately. A process can be killed at any time and at any level of function nesting. When terminated, the process ceases execution and local variables that have been allocated to the process disappear; in fact, the entire stack of functions for the process is removed.

A process can exit by killing itself as easily as it can kill another process. To do so, the process uses system call *getpid* to obtain its own process ID, and then uses *kill* to request termination:

```
kill( getpid() );
```

When used to terminate the current process, the call to *kill* never returns because the calling process exits.

plains conventions for type names.