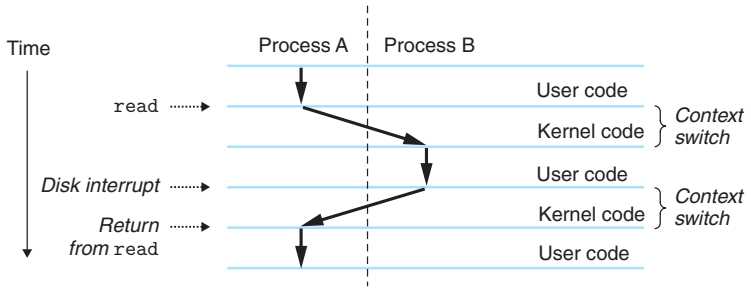

Figure 8.14
Anatomy of a process context switch.

processor after the disk controller has finished transferring the data from disk to memory.

The disk will take a relatively long time to fetch the data (on the order of tens of milliseconds), so instead of waiting and doing nothing in the interim, the kernel performs a context switch from process A to B. Note that, before the switch, the kernel is executing instructions in user mode on behalf of process A (i.e., there is no separate kernel process). During the first part of the switch, the kernel is executing instructions in kernel mode on behalf of process A. Then at some point it begins executing instructions (still in kernel mode) on behalf of process B. And after the switch, the kernel is executing instructions in user mode on behalf of process B.

Process B then runs for a while in user mode until the disk sends an interrupt to signal that data have been transferred from disk to memory. The kernel decides that process B has run long enough and performs a context switch from process B to A, returning control in process A to the instruction immediately following the `read` system call. Process A continues to run until the next exception occurs, and so on.

8.3 System Call Error Handling

When Unix system-level functions encounter an error, they typically return -1 and set the global integer variable `errno` to indicate what went wrong. Programmers should *always* check for errors, but unfortunately, many skip error checking because it bloats the code and makes it harder to read. For example, here is how we might check for errors when we call the Linux `fork` function:

```
1       if ((pid = fork()) < 0) {
2           fprintf(stderr, "fork error: %s\n", strerror(errno));
3           exit(0);
4       }
```

The `strerror` function returns a text string that describes the error associated with a particular value of `errno`. We can simplify this code somewhat by defining the following *error-reporting function:*

```
1  void unix_error(char *msg) /* Unix-style error */
2  {
3      fprintf(stderr, "%s: %s\n", msg, strerror(errno));
4      exit(0);
5  }
```

Given this function, our call to `fork` reduces from four lines to two lines:

```
1      if ((pid = fork()) < 0)
2          unix_error("fork error");
```

We can simplify our code even further by using *error-handling wrappers*, as pioneered by Stevens in [110]. For a given base function `foo`, we define a wrapper function `Foo` with identical arguments but with the first letter of the name capitalized. The wrapper calls the base function, checks for errors, and terminates if there are any problems. For example, here is the error-handling wrapper for the `fork` function:

```
1  pid_t Fork(void)
2  {
3      pid_t pid;
4
5      if ((pid = fork()) < 0)
6          unix_error("Fork error");
7      return pid;
8  }
```

Given this wrapper, our call to `fork` shrinks to a single compact line:

```
1      pid = Fork();
```

We will use error-handling wrappers throughout the remainder of this book. They allow us to keep our code examples concise without giving you the mistaken impression that it is permissible to ignore error checking. Note that when we discuss system-level functions in the text, we will always refer to them by their lowercase base names, rather than by their uppercase wrapper names.

See Appendix A for a discussion of Unix error handling and the error-handling wrappers used throughout this book. The wrappers are defined in a file called `csapp.c`, and their prototypes are defined in a header file called `csapp.h`. These are available online from the CS:APP Web site.

8.4 Process Control

Unix provides a number of system calls for manipulating processes from C programs. This section describes the important functions and gives examples of how they are used.