磁盘取数据要用一段相对较长的时间(数量级为几十毫秒)，所以内核执行从进程A到进程B的上下文切换，而不是在这个间歇时间内等待，什么都不做。注意在切换之前，内核正代表进程A在用户模式下执行指令(即没有单独的内核进程)。在切换的第一部分中，内核代表进程A在内核模式下执行指令。然后在某一时刻，它开始代表进程B(仍然是内核模式下)执行指令。在切换之后，内核代表进程B在用户模式下执行指令。

随后，进程B在用户模式下运行一会儿，直到磁盘发出一个中断信号，表示数据已经从磁盘传送到了内存。内核判定进程B已经运行了足够长的时间，就执行一个从进程B到进程A的上下文切换，将控制返回给进程A中紧随在系统调用read之后的那条指令。进程A继续运行，直到下一次异常发生，依此类推。

8.3　系统调用错误处理

当Unix系统级函数遇到错误时，它们通常会返回−1，并设置全局整数变量errno来表示什么出错了。程序员应该总是检查错误，但是不幸的是，许多人都忽略了错误检查，因为它使代码变得臃肿，而且难以读懂。比如，下面是我们调用Unix fork函数时会如何检查错误：

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

strerror函数返回一个文本串，描述了和某个errno值相关联的错误。通过定义下面的*错误报告函数*，我们能够在某种程度上简化这个代码：

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

给定这个函数，我们对fork的调用从4行缩减到2行：

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

通过使用*错误处理包装函数*，我们可以更进一步地简化代码，Stevens在[110]中首先提出了这种方法。对于一个给定的基本函数foo，我们定义一个具有相同参数的包装函数Foo，但是第一个字母大写了。包装函数调用基本函数，检查错误，如果有任何问题就终止。比如，下面是fork函数的错误处理包装函数：

```
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   }
```

给定这个包装函数，我们对fork的调用就缩减为1行：