Fork()与僵尸进程

Mar 27th, 2013 8:18 pm

使用fork()函数派生出多个子进程来并行执行程序的不同代码块，是一种常用的编程泛型。特别是在网络编程中，父进程初始化后派生出指定数量的子进程，共同监听网络端口并处理请求，从而达到扩容的目的。

但是，在使用fork()函数时若处理不当，很容易产生僵尸进程。根据UNIX系统的定义，僵尸进程是指子进程退出后，它的父进程没有“等待”该子进程，这样的子进程就会成为僵尸进程。何谓“等待”？僵尸进程的危害是什么？以及要如何避免？这就是本文将要阐述的内容。

fork()函数

下面这段C语言代码展示了fork()函数的使用方法：

```
1  // myfork.c
2
3  #include <unistd.h>
4  #include <stdio.h>
5
6  int main(int argc, char **argv) {
7      while (1) {
8          pid_t pid = fork();
9          if (pid > 0) {
10             // 主进程
11             sleep(5);
12         } else if (pid == 0) {
13             // 子进程
14             return 0;
15         } else {
16             fprintf(stderr, "fork error\n");
17             return 2;
18         }
19     }
20 }
```

调用fork()函数后，系统会将当前进程的绝大部分资源拷贝一份（其中的copy-on-write技术这里不详述），该函数的返回值有三种情况，分别是：

- 大于0，表示当前进程为父进程，返回值是子进程号；
- 等于0，表示当前进程是子进程；
- 小于0（确切地说是等于-1），表示fork()调用失败。

让我们编译执行这段程序，并查看进程表：

```
1 $ gcc myfork.c -o myfork && ./myfork
2 $ ps -ef | grep fork
3 vagrant  14860  2748  0 06:09 pts/0    00:00:00 ./myfork
4 vagrant  14861 14860  0 06:09 pts/0    00:00:00 [myfork] <defunct>
5 vagrant  14864 14860  0 06:09 pts/0    00:00:00 [myfork] <defunct>
6 vagrant  14877 14860  0 06:09 pts/0    00:00:00 [myfork] <defunct>
7 vagrant  14879  2784  0 06:09 pts/1    00:00:00 grep fork
```

可以看到子进程创建成功了，进程号也有对应关系。但是每个子进程后面都跟有“defunct”标识，即表示该进程是一个僵尸进程。

这段程序会每五秒创建一个新的子进程，如果不加以回收，那就会占满进程表，使得系统无法再创建进程。这也是僵尸进程最大的危害。

wait()函数

我们对上面这段程序稍加修改：

```
1 pid_t pid = fork();
2 if (pid > 0) {
3     // parent process
4     wait(NULL);
5     sleep(5);
6 } else ...
```

编译执行后会发现进程表中不再出现defunct进程了，即子进程已被完全回收。因此上文中的“等待”指的是主进程等待子进程结束，获取子进程的结束状态信息，这时内核才会回收子进程。

除了通过“等待”来回收子进程，主进程退出也会回收子进程。这是因为主进程退出后，init进程（PID=1）会接管这些僵尸进程，该进程一定会调用wait()函数（或其他类似函数），从而保证僵尸进程得以回收。

SIGCHLD信号

通常，父进程不会始终处于等待状态，它还需要执行其它代码，因此“等待”的工作会使用信号机制来完成。

在子进程终止时，内核会发送SIGCHLD信号给父进程，因此父进程可以添加信号处理函数，并在该函数中调用wait()函数，以防止僵尸进程的产生。

```
1  // myfork2.c
2
3  #include <unistd.h>
4  #include <stdio.h>
5  #include <signal.h>
6  #include <time.h>
7  #include <sys/wait.h>
8
9  void signal_handler(int signo) {
10     if (signo == SIGCHLD) {
11         pid_t pid;
12         while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
13             printf("SIGCHLD pid %d\n", pid);
14         }
15     }
16 }
17
18 void mysleep(int sec) {
19     time_t start = time(NULL), elapsed = 0;
20     while (elapsed < sec) {
21         sleep(sec - elapsed);
22         elapsed = time(NULL) - start;
```

```
23     }
24 }
25
26 int main(int argc, char **argv) {
27
28     signal(SIGCHLD, signal_handler);
29
30     while (1) {
31         pid_t pid = fork();
32         if (pid > 0) {
33             // parent process
34             mysleep(5);
35         } else if (pid == 0) {
36             // child process
37             printf("child pid %d\n", getpid());
38             return 0;
39         } else {
40             fprintf(stderr, "fork error\n");
41             return 2;
42         }
43     }
44 }
```

代码执行结果：

```
1 $ gcc myfork2.c -o myfork2 && ./myfork2
2 child pid 17422
3 SIGCHLD pid 17422
4 child pid 17423
5 SIGCHLD pid 17423
```

其中，signal()用于注册信号处理函数，该处理函数接收一个signo参数，用来标识信号的类型。

waitpid()的功能和wait()类似，但提供了额外的选项（wait(NULL)等价于waitpid(-1, NULL, 0)）。如，wait()函数是阻塞的，而waitpid()提供了WNOHANG选项，调用后会立刻返回，可根据返回值判断等待结果。

此外，我们在信号处理中使用了一个循环体，不断调用waitpid()，直到失败为止。那是因为在系统繁忙时，信号可能会被合并，即两个子进程结束只会发送一次SIGCHLD信号，如果只wait()一次，就会产生僵尸进程。

最后，由于默认的sleep()函数会在接收到信号时立即返回，因此为了方便演示，这里定义了mysleep()函数。

SIG_IGN

除了在SIGCHLD信号处理函数中调用wait()来避免产生僵尸进程，我们还可以选择忽略SIGCHLD信号，告知操作系统父进程不关心子进程的退出状态，可以直接清理。

```
1 signal(SIGCHLD, SIG_IGN);
```

但需要注意的是，在部分BSD系统中，这种做法仍会产生僵尸进程。因此更为通用的方法还是使用wait()函数。

Perl中的fork()函数

Perl语言提供了相应的内置函数来创建子进程：

```
1  #!/usr/bin/perl
2
3  sub REAPER {
4      my $pid;
5      while (($pid = waitpid(-1, WNOHANG)) > 0) {
6          print "SIGCHLD pid $pid\n";
7      }
8  }
9
10 $SIG{CHLD} = \&REAPER;
11
12 my $pid = fork();
13 if ($pid > 0) {
14     print "[Parent] child pid $pid\n";
15     sleep(10);
16 } elsif ($pid == 0) {
17     print "[Child] pid $$\n";
18     exit;
19 }
```

其思路和C语言基本是一致的。如果想要忽略SIGCHLD，可使用`$SIG{CHLD} = 'IGNORE';`，但还是要考虑BSD系统上的限制。

参考资料

- fork(2) - Linux man page
- waitpid(2) - Linux man page
- UNIX环境高级编程（英文版）（第2版）
- Linux多进程相关内容

Posted by Ji ZHANG Mar 27th, 2013 8:18 pm notes