操作系统 作业 2

张远航 2015K8009929045

2017 年 9 月 20 日

一、 请研究一下 Linux `gettimeofday()` 系统调用的用法，并测量一下 `gettimeofday` 的执行时间。

我们做一百万次 `gettimeofday` 系统调用，比较开始和结束时的 `usec` （微秒值），然后求平均（观察到程序一般运行不到 1 秒）。一般来说，总运行时间在 0.02 ～ 0.04 秒左右，故平均一次调用耗时约 0.03 微秒。用到的程序：[1]

```
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>

int main() {
        struct timeval tv, tv_start, tv_end;
        struct timezone tz;
        int i;

        gettimeofday(&tv_start, &tz);
        for (i = 0; i < 1000000; ++i)
                gettimeofday(&tv, &tz);
        gettimeofday(&tv_end, &tz);
        printf("%.5f\n", (tv_end.tv_usec-tv_start.tv_usec)/1000000.0);

        return 0;
}
```

系统环境：

```
Linux caszhang 4.8.0-42-generic #45~16.04.1-Ubuntu SMP Thu Mar 9 14:10:58 UTC
2017 x86_64 x86_64 x86_64 GNU/Linux
```

二、 1. 在 xv6 中，PCB 信息利用 `proc` 结构体存储，父进程和子进程间按树结构组织。各进程的结构体统一保存在进程表 `ptable` 内的数组 `struct proc proc[NPROC]` 中（`proc.c, 10:13`）。`proc` 结构体定义如下（`proc.h, 38:52`）：

```
struct proc {
        uint sz;                     // 进程的内存空间（以字节计）
        pde_t* pgdir;                // 页表
        char *kstack;                // 进程在内核中的栈底
```

[1] 循环分支跳转的时间可大致忽略；想法来自Stack Overflow

```
        enum procstate state;        // 进程的状态
        volatile int pid;            // 进程ID
        struct proc *parent;         // 父进程
        struct trapframe *tf;        // 当前系统调用的陷阱帧
        struct context *context;     // swtch()至此处可切换到该进程的上下文
        void *chan;                  // 非零值代表在chan上睡眠
        int killed;                  // 非零值代表进程已被杀死
        struct file *ofile[NOFILE];  // 当前打开的文件
        struct inode *cwd;           // 当前目录
        char name[16];               // 进程名（调试用）
};
```

2. fork() 首先调用 allocproc() 创建一个新的子进程，然后开始复制父进程的 PCB。先将父进程的页表用 copyuvm 复制到子进程的 PCB 中；假设该步正常，接下来便复制父进程的内存大小 sz ，在 parent 记录父进程的地址；复制父进程陷阱帧并清空 %eax 寄存器，从而对于子进程， fork 返回 0；最后复制打开文件列表、进程名。最后，子进程被设为可运行（RUNNABLE）状态。至此，子进程创建完毕，且收到 fork 返回值零。

   接下来，子进程中执行 exec("foo") 语句，用 foo 覆盖掉当前子进程。 exec 操作会将进程 PCB 中的页表、内存空间以及陷阱帧中的 %eip 、 %esp 寄存器都替换为 foo 的相关信息。

   前面子进程创建完的同时，在父进程中， fork 返回值为子进程 PID（不为零），故进入 else 段调用 wait() 开始等待子进程执行结束。 wait 函数会不断扫描进程表中的各 PCB，在这个例子中，父进程有唯一的子进程。如果找到的这个进程未进入“僵尸”状态，便调用 sleep 函数进一步等待。 sleep 函数中，进程 PCB 的 chan 和 state 信息会被改变，进程进入睡眠状态。

   当子进程最终进入僵尸状态，主进程 wait 函数会把子进程的栈和内存释放，清零 PCB 中记录的栈底指针、PID、父进程地址、名称和杀死信号，并将进程状态设为未使用（UNUSED）。

三、 一共生成八个进程，进程关系图如下：