# 实验四：Linux环境多进程编程

郑海刚

HITSZ 实验与创新实践教育中心
Education Center of Experiments and Innovations, HITSZ

# 本讲概述

- 主要内容
  - 进程介绍
  - API：fork、exec

# 进程（process）

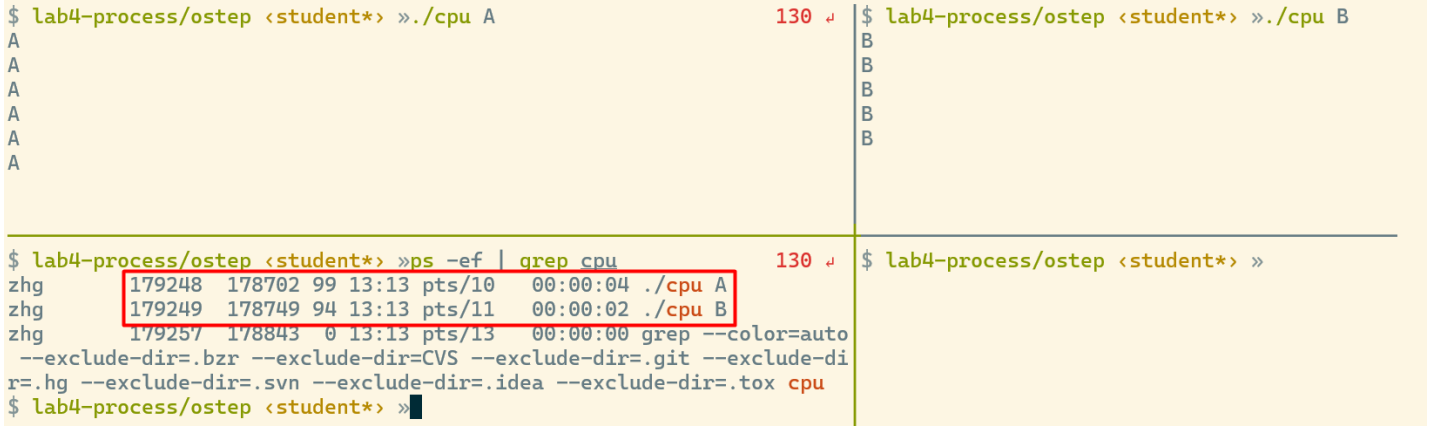
- process wikipedia定义：进程就是运行中的程序
  - 前面介绍的所有可执行文件运行，都会启动而且只启动一个进程
- lab4/ostep/cpu.c
  - 无限循环
    - 打印传入的字符
    - 自旋等待1秒

```c
int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];

    while (1) {
        printf("%s\n", str);
        Spin(1);
    }
    return 0;
}
```

# 进程：lab4/ostep/cpu.c

- 编译：gcc -o cpu cpu.c　　执行：./cpu A
- 每次执行都是一个进程，开两个终端分别执行：
  - 终端1：./cpu A　　终端2： ./cpu B
- 在第三个终端： ps -ef | grep cpu查看进程的信息



# argc & argv

- C标准中定义：程序的入口函数main可以不带参数或者带argc, argv
- argc，argv用于命令行参数解析：Arguments to main
  - argc是int型，参数的个数
  - argv是指针数组，参数的内容，包含可执行文件本身
  - ./cpu A
    - argc = 2
    - argv[0] = “./cpu”
    - argv[1] = “A”

# top工具： display Linux processes

- 查看整体运行情况

```
top - 16:24:43 up 3 days,  4:09,  0 users,  load average: 0.00, 0.00, 0.00
Tasks:  34 total,   1 running,  33 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   7636.9 total,   4504.4 free,    626.6 used,   2505.9 buff/cache
MiB Swap:   2048.0 total,   1950.4 free,     97.6 used.   6659.7 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  994 root      20   0    2188    348      0 S   0.3   0.0   0:54.83 init
    1 root      20   0    1840    848    792 S   0.0   0.0   0:00.27 init
  993 root      20   0    2188    348      0 S   0.0   0.0   0:00.00 init
  995 zhg       20   0   18404  11588   6492 S   0.0   0.1   1:03.68 zsh
 3998 root      20   0    2188    348      0 S   0.0   0.0   0:00.00 init
 3999 root      20   0    2188    348      0 S   0.0   0.0   0:00.56 init
 4000 zhg       20   0   17872  10864   6296 S   0.0   0.1   0:06.68 zsh
```

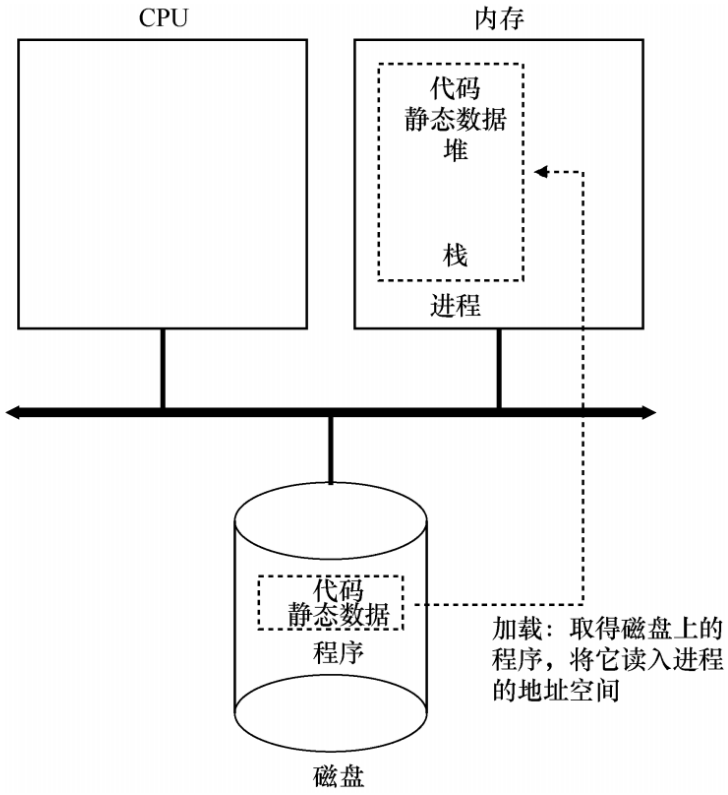
- 重要的进程信息
  - PID：进程ID号，从1开始编号，每个进程唯一
  - USER：所属的用户
  - %CPU %MEM：cpu和mem的利用率
  - COMMAND：创建该进程的命令

# 进程的发展过程

- 初期CPU和IO设备一样慢：程序写在打孔卡片上，交给操作管理员
- 单核时代
  - 批处理：将一批作业交给计算机，不需要人工切换，执行完一个作业再执行下一个作业，内存常驻只有一个作业
  - CPU的速度逐渐远高于IO设备，导致CPU出现空跑
  - 多道程序：多个作业常驻内存，作业等IO的时候切换到其他作业
  - 处理器同一时刻只能执行一条指令，多任务并发执行
- 多核时代
  - 进程能够同时运行，并行执行

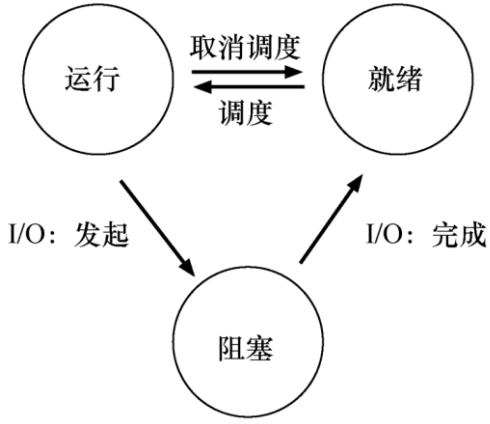
# 进程的创建

- 从程序到进程



# 进程的状态

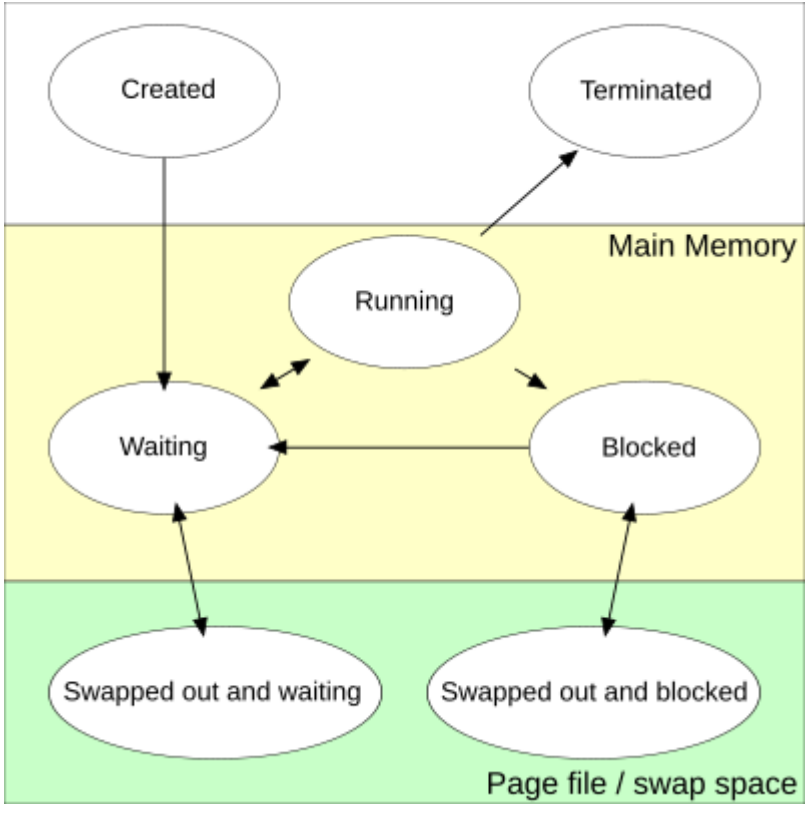
- 基本的状态：
  - 运行：CPU正在执行进程的指令
  - 就绪：可以被执行，但CPU正在执行其他进程
  - 阻塞：还不能被执行，比如文件读取时磁盘没返回




- lab2中的时间测量：
  - 进程不是一直在被CPU执行，所以有墙上时间、CPU时间
  - 只有运行态的算入CPU时间

# 进程的状态

- 实际更复杂一点：Process state
- top输出的S列 即状态



# 进程API：fork

- fork()系统调用，创建新的进程
- lab4/ostep/p1.c
  - 执行结果

```
$ lab4/ostep <master*> »./p1
hello world (pid:17622)
hello, I am parent of 17623 (pid:17622)
hello, I am child (pid:17623)
```

```c
int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
           rc, (int) getpid());
    }
    return 0;
}
```

  - 一次调用，两次返回，if两个分支都走到了
    - 两次返回，是在两个不同的进程各返回一次
    - 父进程fork返回值是子进程的pid，子进程返回值是0

# fork的特性

- fork： 叉子、分叉
- 新创建的进程是子进程，原来的进程为父进程
- 子进程“拷贝”了父进程的代码段、数据段
  - 代码段是只读的，实际是共享；数据段写时复制（Copy-on-write）
- 子进程不会从 main函数开始执行，而是从fork返回处开始执行
- man fork

# 父子进程独立的地址空间

- lab4/ostep/fork_var.c
  - 父子进程同一份代码执行不同分支
  - 能访问相同的变量，分别赋不同的值
  - 实际不同地址空间，数据是独立的

```
$ lab4-process/ostep »gcc fork_var.c
$ lab4-process/ostep »./a.out
hello world (pid:28111)

hello, I am parent of 28112 (pid:28111)
parent process(28111) first access:g_x=0, main_x=0
parent process(28111) second access:g_x=3, main_x=3
process(28111) last access:g_x=3, main_x=3

hello, I am child (pid:28112)
child process(28112) first access:g_x=0, main_x=0
child process(28112) second access:g_x=2, main_x=2
process(28112) last access:g_x=2, main_x=2
```

```c
int g_x;
int
main(int argc, char *argv[])
{
    int main_x=0;
    printf("hello world (pid:%d)\n\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        printf("child process(%d) first access:g_x=%d, main_x=%d\n",
                    (int) getpid(), g_x, main_x);
        g_x = 2;
        main_x = 2;
        printf("child process(%d) second access:g_x=%d, main_x=%d\n",
                    (int) getpid(), g_x, main_x);
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
           rc, (int) getpid());
        printf("parent process(%d) first access:g_x=%d, main_x=%d\n",
                    (int) getpid(), g_x, main_x);
        g_x = 3;
        main_x = 3;
        printf("parent process(%d) second access:g_x=%d, main_x=%d\n",
                    (int) getpid(), g_x, main_x);
    }
    printf("process(%d) last access:g_x=%d, main_x=%d\n",
                    (int) getpid(), g_x, main_x);
    printf("\n");
    return 0;

}
```

# 多进程加速矩阵乘法

- 不同的进程同时跑在不同的CPU上，并行执行，可以利用多核的性能
- 进程间的数据是隔离的，怎么把最终的结果即矩阵C汇总？
  - 需要进程间通信
- MPI框架会提供通信机制

# 进程API：exec、execvp

- 功能：子进程执行与父进程不同的程序
- lab4/ostep/p3.c
  - execvp用wc p3.c替换原有的代码
  - execvp不会返回，下一行的printf不会执行

```c
int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
           rc, wc, (int) getpid());
    }
    return 0;
}
```

# 系统调用（system call）

- 为什么fork、exec叫系统调用？用法上不就是普通的函数吗？
  - 系统调用最终会执行一条system call的指令
    - CPU进入到特权级
    - 执行linux内核中对应系统调用号的代码
- man syscall
  - 不同处理器架构system call的指令不相同
  - 不同系统调用的具体功能是在os内核实现的

| Arch/ABI | Instruction | System call # | Ret val | Ret val2 | Error | Notes |
|---|---|---|---|---|---|---|
| alpha | callsys | v0 | v0 | a4 | a3 | 1, 6 |
| arc | trap0 | r8 | r0 | - | - | |
| arm/OABI | swi NR | - | a1 | - | - | 2 |
| arm/EABI | swi 0x0 | r7 | r0 | r1 | - | |
| arm64 | svc #0 | x8 | x0 | x1 | - | |
| blackfin | excpt 0x0 | P0 | R0 | - | - | |
| i386 | int $0x80 | eax | eax | edx | - | |
| ia64 | break 0x100000 | r15 | r8 | r9 | r10 | 1, 6 |
| m68k | trap #0 | d0 | d0 | - | - | |
| microblaze | brki r14,8 | r12 | r3 | - | - | |
| mips | syscall | v0 | v0 | v1 | a3 | 1, 6 |
| nios2 | trap | r2 | r2 | - | r7 | |
| parisc | ble 0x100(%sr2, %r0) | r20 | r28 | - | - | |
| powerpc | sc | r0 | r3 | - | r0 | 1 |
| powerpc64 | sc | r0 | r3 | - | cr0.SO | 1 |
| riscv | ecall | a7 | a0 | a1 | - | |
| s390 | svc 0 | r1 | r2 | r3 | - | 3 |
| s390x | svc 0 | r1 | r2 | r3 | - | 3 |
| superh | trap #0x17 | r3 | r0 | r1 | - | 4, 6 |
| sparc/32 | t 0x10 | g1 | o0 | o1 | psr/csr | 1, 6 |
| sparc/64 | t 0x6d | g1 | o0 | o1 | psr/csr | 1, 6 |
| tile | swint1 | R10 | R00 | - | R01 | 1 |
| x86-64 | syscall | rax | rax | rdx | - | 5 |
| x32 | syscall | rax | rax | rdx | - | 5 |
| xtensa | syscall | a2 | a2 | - | - | |

# 课后阅读

- 《深入理解计算机系统》第8章 异常控制流
- 《操作系统导论》第26、27章