进程控制与信号
Process Control & Signal Basics

课 程 名 ： 计算机系统
主 讲 人 ： 孟文龙

本课内容

- **进程控制Processes Control**
- Shell

示例：僵尸进程回收

```c
void forks()
{
    if (Fork() == 0) {                  /* 子进程直接退出 */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {                            /* 父进程无限循环 */
        printf("Running Parent, PID = %d\n", getpid());
        while (1);
    }
}
```

forks.c

```
openEuler> ./forks
Running Parent, PID = 95300
Terminating Child, PID = 95302
openEuler> ps
  PID TTY          TIME CMD
95168 pts/0    00:00:00 bash
95300 pts/0    00:00:46 forks
95302 pts/0    00:00:00 forks <defunct>
95352 pts/0    00:00:00 ps
openEuler> kill 95300

openEuler> ps
  PID TTY          TIME CMD
95168 pts/0    00:00:00 bash
95472 pts/0    00:00:00 ps
```

- ps 命令显示的子进程标记为“defunct”，即僵尸进程
- 杀死父进程，从而让 init 回收子进程

示例：子进程不终止

```
void forks()
{
    if (Fork() == 0) {          /* 子进程无限循环 */
        printf("Running Child, PID = %d\n", getpid());
        while (1);
    } else {                    /* 父进程直接退出 */
        printf("Terminating Parent, PID = %d\n", getpid());
        exit(0);
    }
}
```

forks.c

```
openEuler> ./forks
Terminating Parent, PID = 96016
Running Child, PID = 96017
openEuler> ps
  PID TTY          TIME CMD
95168 pts/0    00:00:00 bash
96017 pts/0    00:00:20 forks
96045 pts/0    00:00:00 ps

openEuler> kill 96017
openEuler> ps
  PID TTY          TIME CMD
95168 pts/0    00:00:00 bash
96094 pts/0    00:00:00 ps
```

- 父进程终止，但子进程仍处于活动状态
- 必须明确地杀死子进程，否则将会一直执行

与子进程同步：wait/waitpid

- 父进程通过 wait 系统调用回收子进程
- pid_t wait(int *statusp)
  - 挂起当前进程的执行，直到它的某一子进程终止
  - 返回已终止的子进程的 pid
  - 若 statusp != NULL，则向该指针指向的整型变量中存入一个表示①终止原因和②退出状态的值
    - 用 wait.h 头文件中定义的宏来检查

      WIFEXITED, WEXITSTATUS

      WIFSIGNALED, WTERMSIG

      WIFSTOPPED, WSTOPSIG

      WIFCONTINUED

与子进程同步：wait 示例 1

```
void fork9()
{
    int child_status;

    if (Fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has"
               "terminated\n");
    }
    printf("Bye\n");
}
```

forks.c

可行的输出：

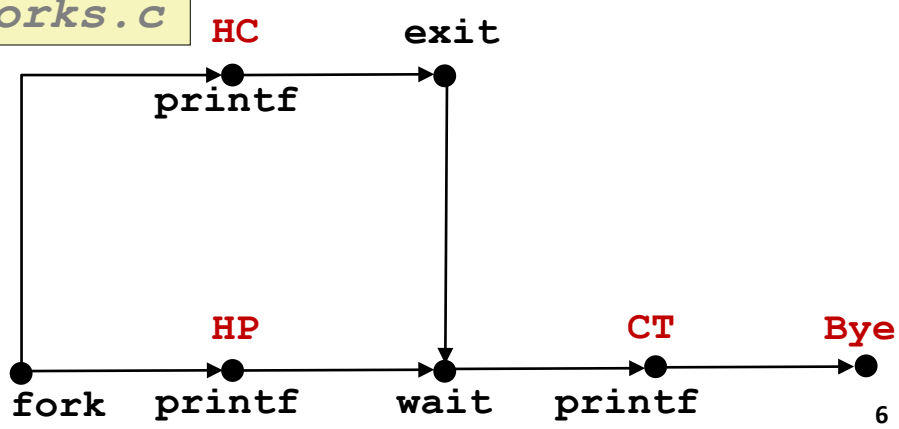
HC
HP
CT
Bye

可行的输出：

HP
HC
CT
Bye

不可行的输出：

HP
CT
Bye
HC



与子进程同步：wait 示例 2

- 子进程完成结束的顺序是任意的（没有固定的顺序）
- 可用宏 WIFEXITED 和 WEXITSTATUS 获取进程的退出状态

forks.c

```
void fork10()
{
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = Fork()) == 0)
            exit(100+i);                    /* 子进程以状态 100+i 退出 */
    for (i = 0; i < N; i++) {               /* 父进程逐个回收子进程 */
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status)) //判断子进程是否正常退出
            printf("Child %d terminated with exit status %d.\n",
                wpid, WEXITSTATUS(child_status));//如果正常退出，则获取
                                                 退出状态码即exit(100+i)里的100+i
        else
            printf("Child %d terminated abnormally.\n", wpid);
    }
}
```

waitpid: 等待特定进程

- pid_t waitpid(pid_t pid, int *statusp, int options)
  - 挂起当前进程，直到指定进程终止

控制等待行为

forks.c

```
void fork11()
{
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = Fork()) == 0) {
            exit(100+i);                        // 子进程
        }
    for (i = N-1; i >= 0; i--) {                // 父进程有序回收子进程
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d.\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally.\n", wpid);
    }
}
```

waitpid：等待特定进程（续）

- pid_t waitpid(pid_t pid, int *statusp, int options)
  - 参数 pid 的含义
    - 为正，表示等待进程 ID 等于该 pid 的子进程
    - 为 -1，表示等待集由本进程的所有子进程组成
    - 小于 -1，表示等待集由进程组 ID 为 -pid 的所有子进程组成
  - 参数 options 的含义
    - 0：挂起当前进程，直到等待集合中的任一子进程终止
    - **WNOHANG**：若等待集合中没有已终止的子进程，则立即返回 0，从而父进程可继续其它工作
    - **WUNTRACED**：挂起当前进程，直到等待集合中的任一子进程终止或停止
    - **WCONTINUED**：挂起当前进程，直到等待集合中的任一子进程终止，或一个停止的子进程因收到 **SIGCONT** 信号而恢复运行

waitpid：等待特定进程（续）

WNOHANG：若等待集合中没有已终止的子进程，则立即返回 0，从而父进程可继续其它工作

```
pid_t ret = waitpid(-1, &status, WNOHANG);
if (ret == 0) {
    // 没有子进程退出，父进程可以继续其它工作
}
```

waitpid：等待特定进程（续）

特性	wait	waitpid
等待对象	任意一个子进程	可指定某一个或一类子进程
灵活性	低	高
非阻塞支持	否	支持（通过 options 参数）
推荐使用场景	子进程很少，管理简单	需要精确控制或管理多个子进程时更适合

进程休眠

- `unsigned int sleep(unsigned int secs)`
  - 时间到则返回 0
  - 若休眠过程被信号中断，则返回剩余未休眠的时间
- `unsigned int alarm(unsigned int secs)`
  - `secs` 秒后向本进程发送 **SIGALRM** 信号
  - 返回此前的其它 `alarm` 调用的剩余秒数，重新计时
- `int pause(void)`
  - 令本进程休眠，直到收到一个信号
  - 返回值为 `-1`
- 与 `wait` 不同，`sleep`、`pause` 不能回收子进程
- 三者均在 `unistd.h` 中声明

execve：加载并运行程序

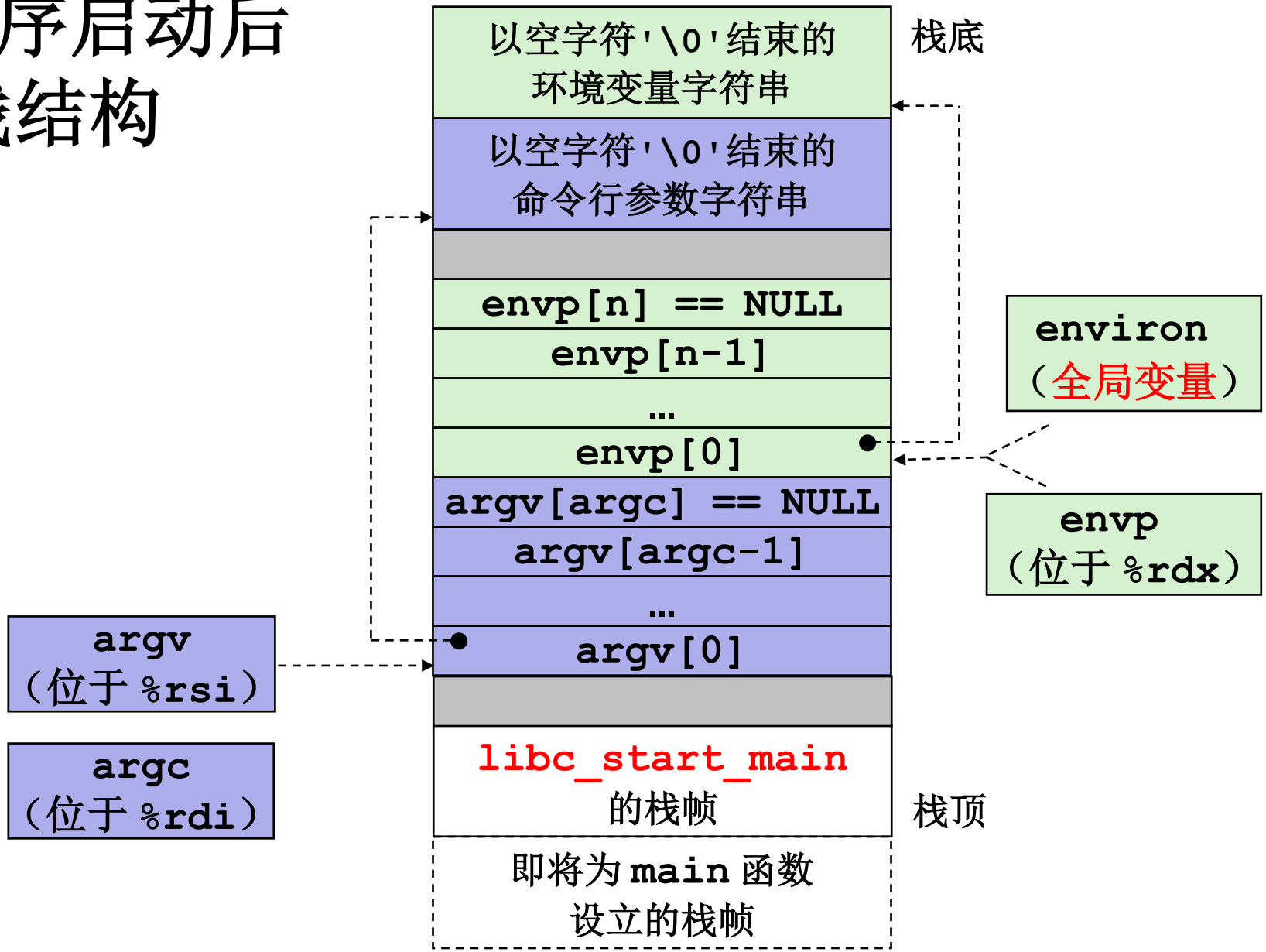
- `int execve(char *filename, char *argv[], char *envp[])`
- 位于 unistd.h 头文件
- 在当前进程中载入并运行程序
  - filename：可执行文件
    - 目标文件或脚本（用 #! 指明解释器，如：#!/bin/bash）
  - argv：参数列表，按照惯例，argv[0] == filename
  - envp：环境变量列表
    - 形如“*name=value*”的字符串（如：USER=openEuler）

```
char *argv[] = {"/bin/ls", "-l", NULL};
char *envp[] = {"USER=openEuler", NULL};
execve("/bin/ls", argv, envp);
```

execve：加载并运行程序（续）

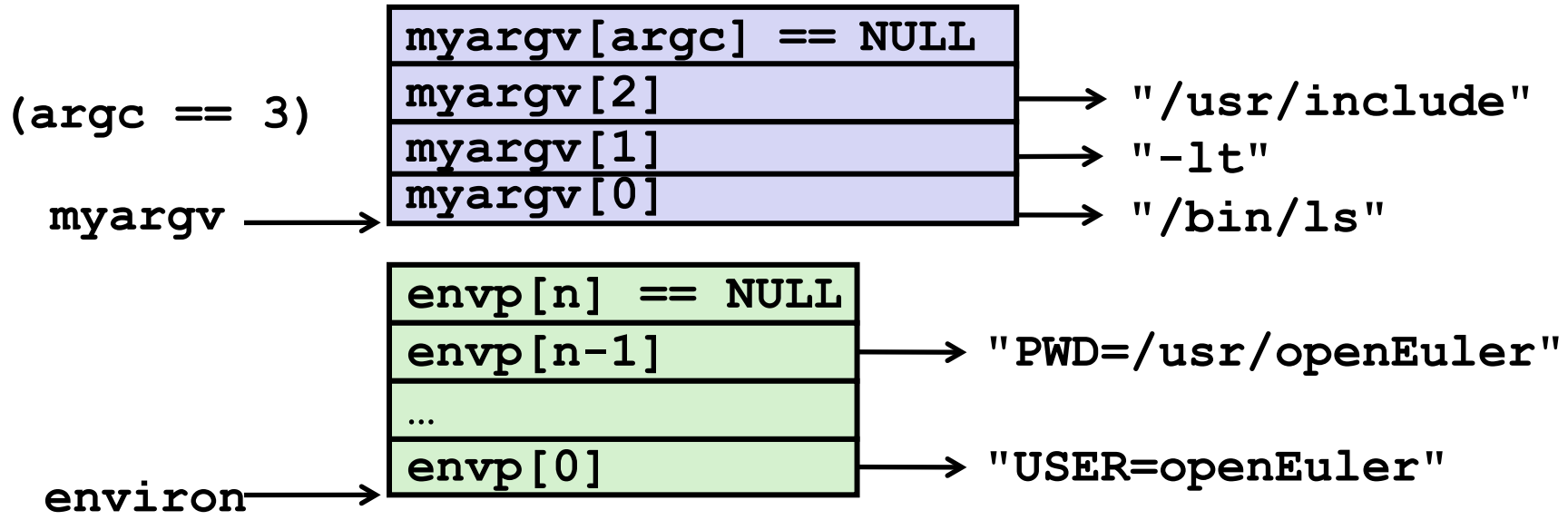
- `int execve(char *filename, char *argv[], char *envp[])`
- 删除进程现有的虚拟内存段，创建一组新段（栈与堆初始化为 0）
- 建立起虚拟地址空间中的段与可执行目标文件中的节之间的映像关系（根据程序头表）
- 新的代码与数据段被初始化为可执行文件的内容，然后转至入口点，即 `_start()` 首地址（根据 ELF 头）
- 除了一些头部信息外，并未实际读文件，直到发生缺页
- 覆盖当前进程的代码、数据、栈
- 调用 1 次返回 0 次
  - 除非有错误，如指定的文件不存在

新程序启动后的栈结构


以空字符'\0'结束的
环境变量字符串
栈底
以空字符'\0'结束的
命令行参数字符串
envp[n] == NULL
envp[n-1]
…
envp[0]
argv[argc] == NULL
argv[argc-1]
…
argv[0]
libc_start_main
的栈帧
栈顶
即将为 main 函数
设立的栈帧
environ
（全局变量）
envp
（位于 %rdx）
argv
（位于 %rsi）
argc
（位于 %rdi）

execve 示例

- 在子进程中利用当前环境执行：/bin/ls -lt /usr/include

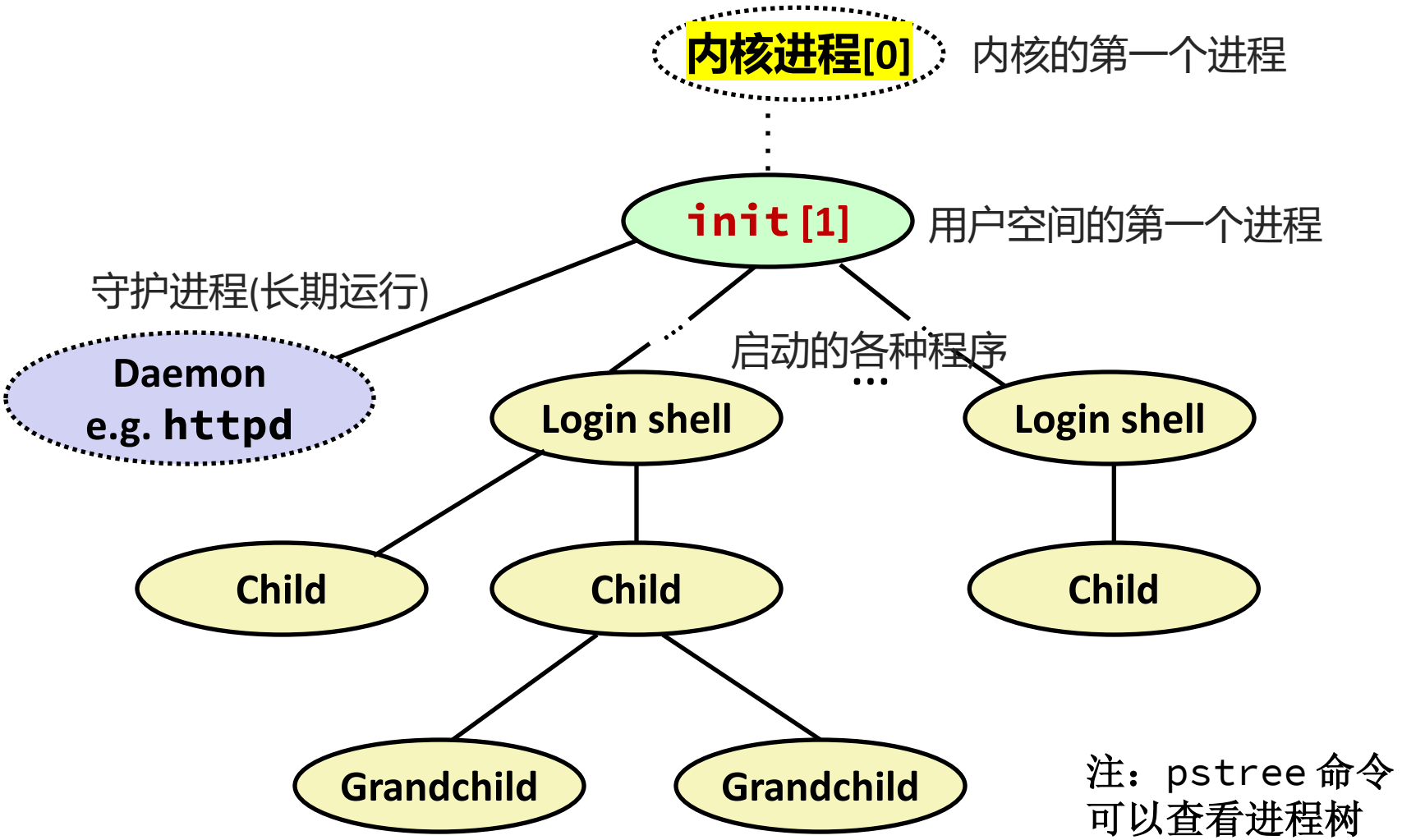



```
if ((pid = Fork()) == 0) {    /* 子进程执行程序 */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

本课内容

- 进程控制Processes Control
- **Shell**

Linux 进程体系


内核进程[0]
内核的第一个进程
init [1]
用户空间的第一个进程
守护进程(长期运行)
Daemon
e.g. httpd
启动的各种程序
Login shell
Login shell
Child
Child
Child
Grandchild
Grandchild
注：pstree 命令
可以查看进程树

shell 程序

- shell 是一个交互型应用程序，代表用户运行其它程序
  - sh　　最早的 shell（Stephen Bourne，贝尔实验室）
  - csh/tcsh　　BSD Unix C shell
  - bash　　“Bourne-Again” Shell，缺省的 Linux shell

```
int main()
{
    char cmdline[MAXLINE];        // 命令行

    while (1) {
        /* 读 */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))          // Ctrl-D
            exit(0);

        /* 求值 */
        eval(cmdline);
    }
}
```

shellex.c

shell 执行一系列读/求值步骤：

读步骤读取用户的命令行

求值步骤解析命令，代表用户运行程序

简单 shell 例程的问题

- shell 能够正确地等待并回收前台作业
- 但是后台作业可能会：
  - 终止后成为僵死进程
  - shell 一般不会终止,该进程一直得不到回收
  - 导致内存泄漏，终将使得再无内存可用

	前台作业	后台作业
运行方式	占用终端，需等待结束	不占用终端，立即返回
I/O	直接与终端交互	通常不与终端交互
管理	直接用快捷键控制	需用 `jobs`、`fg`、`bg` 管理

怎么办？

- 解决办法：异常控制流 ECF
  - 后台进程完成时，内核将中断正常处理程序，给我们提醒
  - 在 Unix 中，这种提醒机制称作信号
  - Windows 下称作消息