中国科学技术大学软件学院
SCHOOL OF SOFTWARE ENGINEERING OF USTC

Program & Process

based on Linux3.2

孟宁

可执行文件->进程的执行上下文

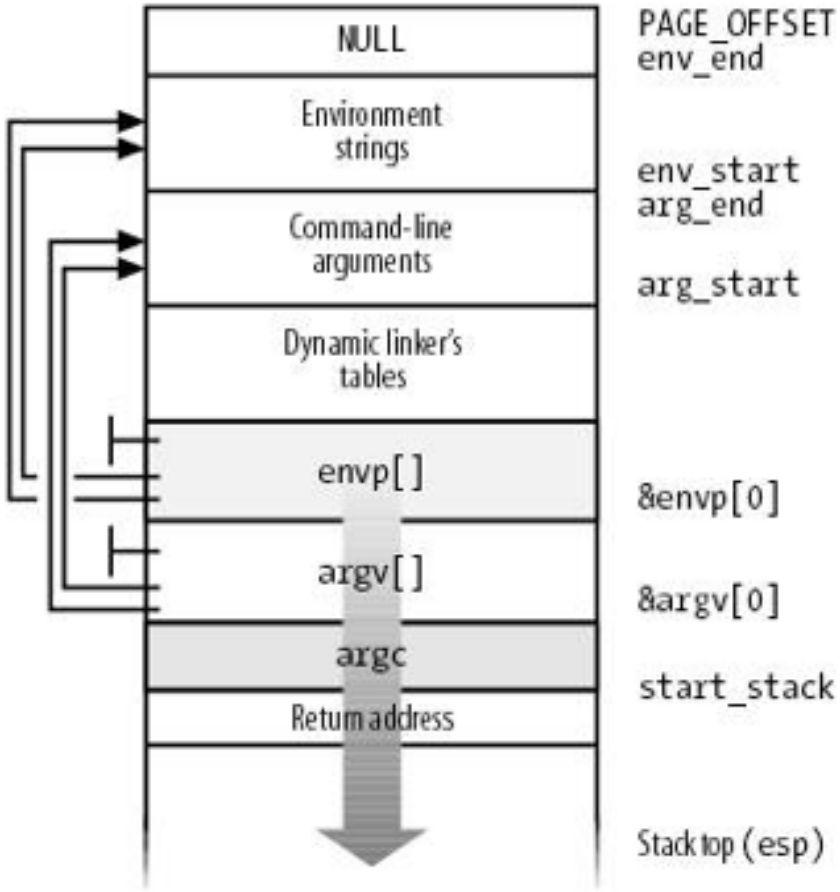
- 命令行参数、环境变量等
  - Shell提示符下输入
  - 从shell继承而来，用户可修改
- 可执行文件
  - 可执行文件的格式
  - 程序以可执行文件的形式存放在磁盘上
- 库
  - 可供很多程序使用的一些例程的集合
  - 静态库　vs　共享库
- 可执行文件的加载与进程的实现

命令行参数和shell环境

- 用户使用shell来执行某个程序时，可以指定命令行参数
  - 例如：$ ls -l /usr/bin 列出/usr/bin下的目录信息
- Shell本身不限制命令行参数的个数，命令行参数的个数受限于命令自身
  - 例如，int main(int argc, char *argv[])
  - 又如， int main(int argc, char *argv[], char *envp[])

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char * argv[])
{
        int pid;
        /* fork another process */
        pid = fork();
        if (pid<0)
        {
                /* error occurred */
                fprintf(stderr,"Fork Failed!");
                exit(-1);
        }
        else if (pid==0)
        {
                /*              child process           */
                execlp("/bin/ls","ls",NULL);
        }
        else
        {
                /*              parent process          */
                /* parent will wait for the child to complete*/
                wait(NULL);
                printf("Child Complete!");
                exit(0);
        }
}
```

命令行参数和环境串都放在用户态堆栈中



程序段

- 在逻辑上，Unix程序的线性地址空间被划分为各种段（segment）
  - 正文段，text
  - 数据段，data
  - 堆栈段等
- 在mm_struct中都有对应的字段/include/linux/mm_types.h

```
00198:      unsigned long start_code, end_code, start_data, end_data;
00199:      unsigned long start_brk, brk, start_stack;
00200:      unsigned long arg_start, arg_end, env_start, env_end;
```

可执行文件

- 可执行文件是一个普通的文件，它描述了如何初始化一个新的进程上下文
  - Fork + execve
- Linux标准的可执行格式
  - ELF：Executable and Linking Format
- 旧版的可执行文件格式
  - a.out：Assembler OUT put format
- 其他
  - MS-DOS的exe文件
  - UNIX BSD的COFF文件

可执行文件格式的内核处理代码

- include\linux\binfmts.h

```
struct linux_binfmt {
        struct list_head lh;
        struct module *module;
        int (*load_binary)(struct linux_binprm *, struct  pt_regs * regs);
        int (*load_shlib)(struct file *);
        int (*core_dump)(struct coredump_params *cprm);
        unsigned long min_coredump;        /* minimal dump size */
};
int register_binfmt(struct linux_binfmt *fmt);
```

- fs/binfmt_elf.c

```
static struct linux_binfmt elf_format = {
        .module         = THIS_MODULE,
        .load_binary    = load_elf_binary,
        .load_shlib     = load_elf_library,
        .core_dump      = elf_core_dump,
        .min_coredump   = ELF_EXEC_PAGESIZE,
};
```

库

- 源文件è目标文件è可执行文件
- 最小的程序也会利用到C库
  例如：void main(void) {}
  - 要为main的执行建立执行上下文
  - 在进程结束时，杀死进程（在main的最后插入exit()）
- 其他库
  - libm，包含浮点操作的基本函数
  - libX11，所有X11窗口系统图形接口的基本底层函数

静态链接 vs 动态链接

- 静态库（libxxx.a）
  - Gcc的-static选项指明使用静态库
- 动态链接：共享库(libxxx.so)

共享库和文件的映射

- 参阅/proc/1/maps了解init进程的线性区

```
ubuntu@ubuntu:~$ sudo cat /proc/1/maps
[sudo] password for ubuntu:
00110000-0011a000 r-xp 00000000 08:01 789210     /lib/tls/i686/cmov/libnss_files-2.11.1.so
...
00253000-003a6000 r-xp 00000000 08:01 789193     /lib/tls/i686/cmov/libc-2.11.1.so
...
004fd000-004ff000 rw-p 00000000 00:00 0
00567000-00581000 r-xp 00000000 08:01 659752     /sbin/init
00581000-00582000 r--p 00019000 08:01 659752     /sbin/init
00582000-00583000 rw-p 0001a000 08:01 659752     /sbin/init
00969000-0096a000 r-xp 00000000 00:00 0          [vdso]
00973000-0097a000 r-xp 00000000 08:01 655559     /lib/libnih-dbus.so.1.0.0
...
00e19000-00e1a000 rw-p 00015000 08:01 789219     /lib/tls/i686/cmov/libpthread-2.11.1.so
00e1a000-00e1c000 rw-p 00000000 00:00 0
00f5a000-00f75000 r-xp 00000000 08:01 655478     /lib/ld-2.11.1.so
...
20685000-206e6000 rw-p 00000000 00:00 0          [heap]
b77f0000-b77f3000 rw-p 00000000 00:00 0
b7801000-b7803000 rw-p 00000000 00:00 0
bfa47000-bfa5c000 rw-p 00000000 00:00 0          [stack]
```

可执行文件的加载

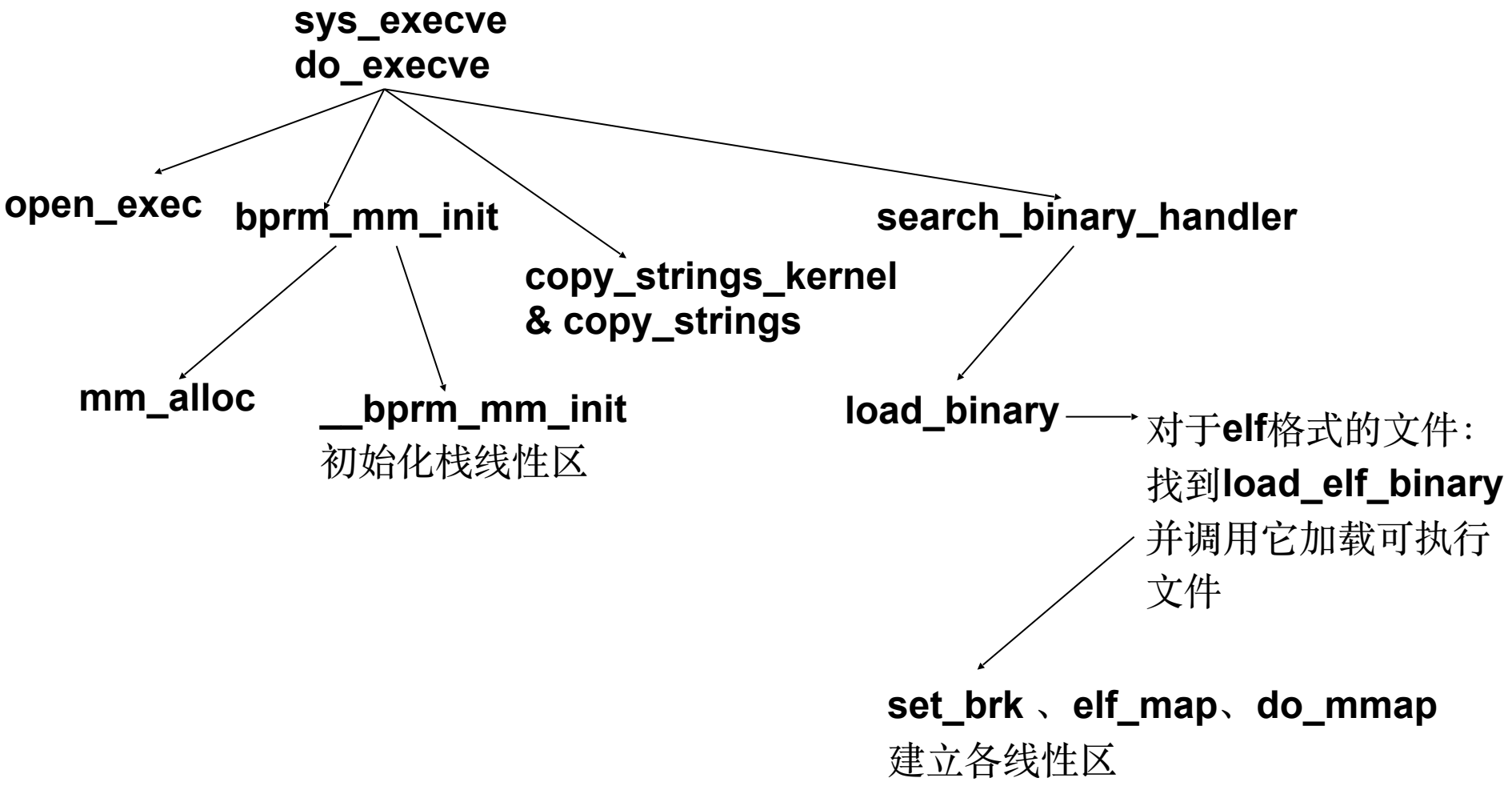
```
#include <unistd.h>

extern char **environ;

int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ..., char * const envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

- 用一个指定的可执行文件所描述的上下文代替进程的上下文
- è系统调用：execve
  - ➢ sys_execve

可执行文件的加载

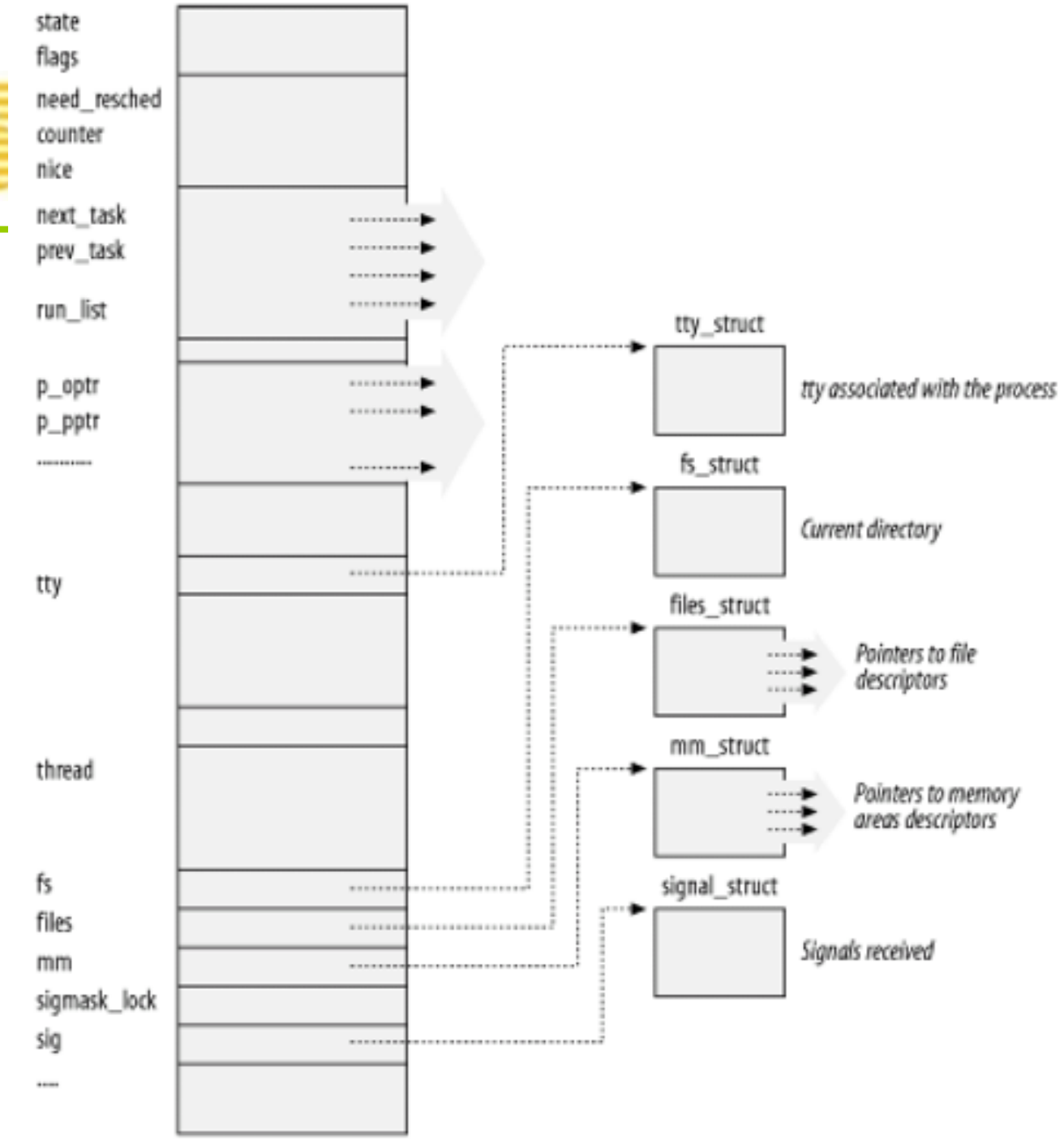

sys_execve
do_execve
open_exec
bprm_mm_init
search_binary_handler
copy_strings_kernel
& copy_strings
mm_alloc
__bprm_mm_init
初始化栈线性区
load_binary
对于elf格式的文件：
找到load_elf_binary
并调用它加载可执行
文件
set_brk 、elf_map、do_mmap
建立各线性区

进程、轻量级进程、线程和内核线程

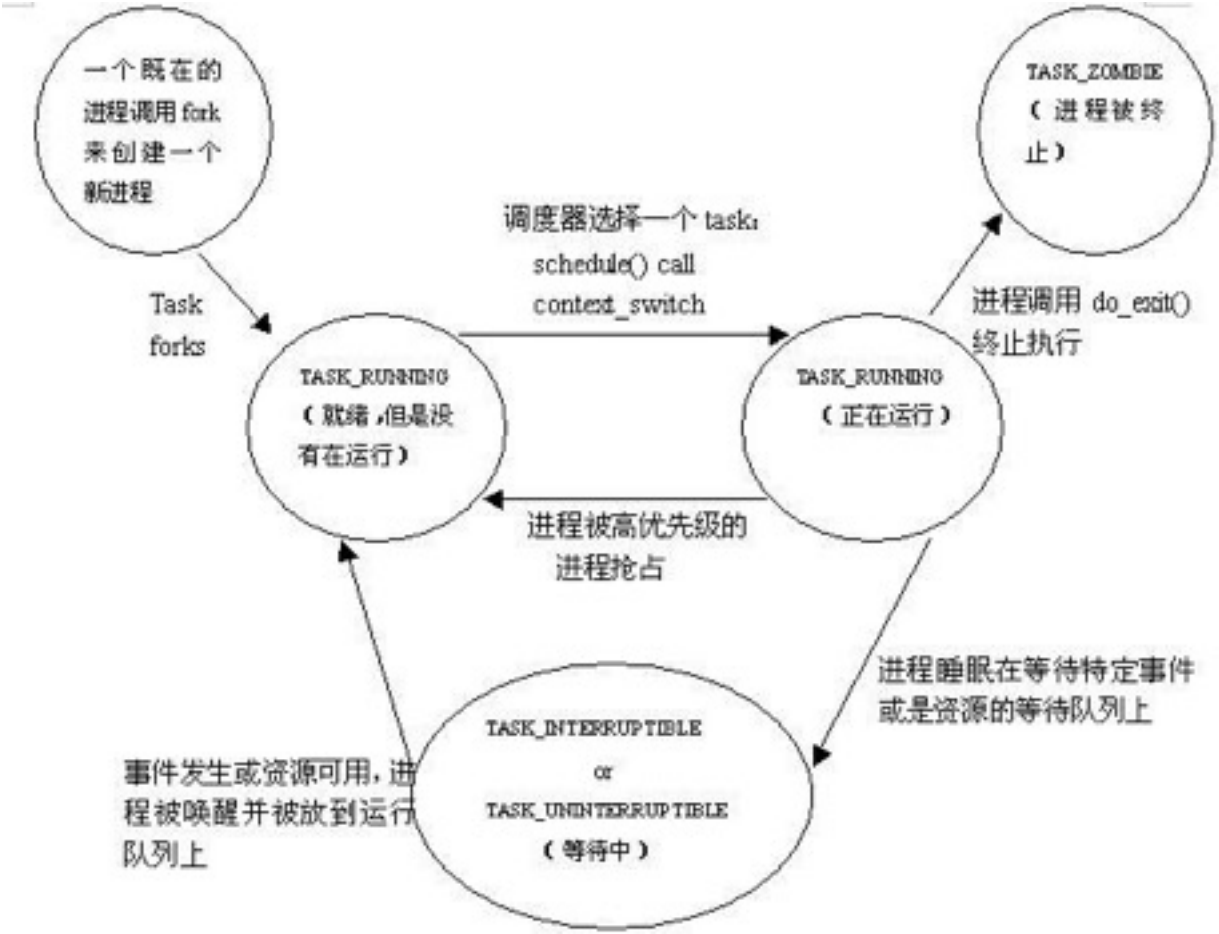
- CPU（单CPU系统）是一条指令一条指令执行的；
- 进程是执行上下文、定义一个执行流，有特别权限的进程管理其他进程，也就是分配CPU执行时间；
- 轻量级进程是为了支持多线程（一个进程中有多个执行流）而使多个进程共享一些资源，从操作系统的角度看没有线程，全部通过进程来分配资源并进行调度管理，但从多线程应用开发者的角度看，它似乎在同一个进程中创建多个执行流(线程)；
- 而内核线程并不是线程，而是一个特殊一点的进程，仅工作在内核态，一般是一些服务程序为了避免内核态和用户态切换的开销，而普通进程一般根据需要通过系统调用在内核态和用户态之间反复切换。

进程描述符

- 为了管理进程，内核必须对每个进程进行清晰的描述。
- 进程描述符提供了内核所需了解的进程信息
  - 源码include/linux/sched.h定义line:1220 struct task_struct
  - 数据结构很庞大
    - 基本信息
    - 管理信息
    - 控制信息



Linux进程状态转换图



Linux2.6进程的状态

- 进程描述符struct task_struct中的state
- 与状态相关的一些宏
  - include/linux/sched.h line:183

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED                  4
#define __TASK_TRACED           8
/* in tsk->exit_state */
#define EXIT_ZOMBIE             16
#define EXIT_DEAD               32
/* in tsk->state again */
#define TASK_DEAD               64
#define TASK_WAKEKILL           128
#define TASK_WAKING             256
#define TASK_STATE_MAX          512
```

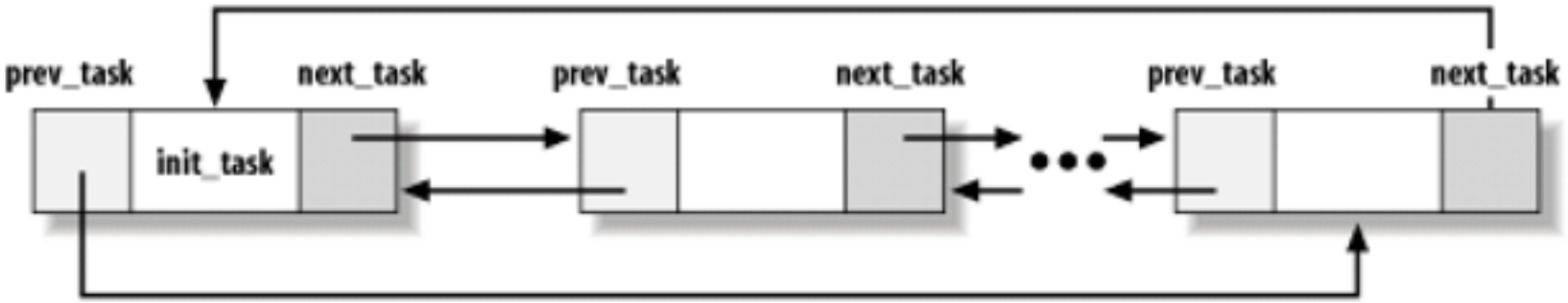
标识一个进程

- 使用进程描述符地址
  - 进程和进程描述符之间有非常严格的一一对应关系使得用32位进程描述符地址标识进程非常方便
- 使用PID (Process ID，PID)
  - 每个进程的PID都存放在进程描述符struct task_struct的pid域中
  - Pid最大值，参见kernel/pid.c int pid_max
  - 顺序使用&&循环使用

进程链表

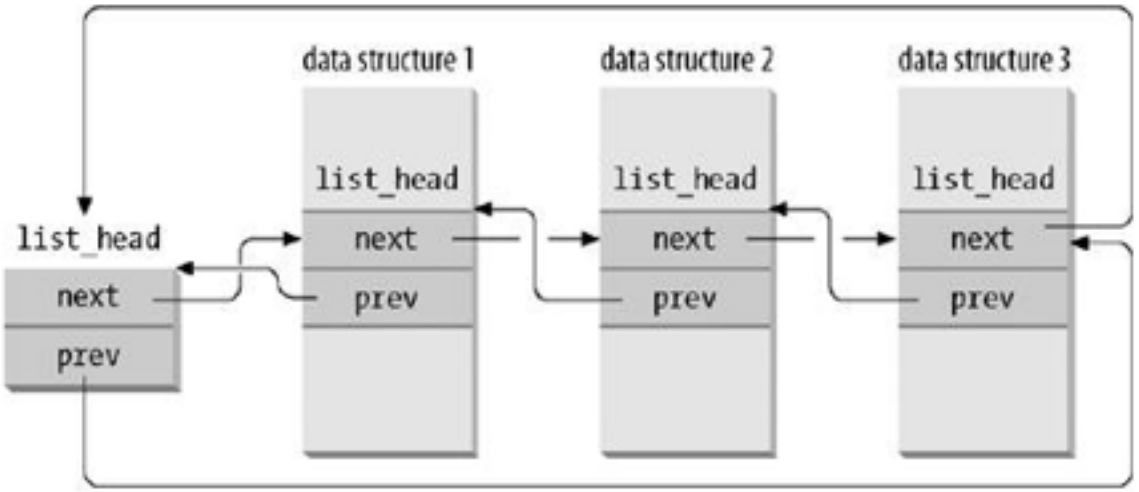
- 为了对给定类型的进程(比如所有在可运行状态下的进程)进行有效的搜索，内核维护了几个进程链表
- 所有进程链表struct list_head tasks;

Figure 3-3. The process list

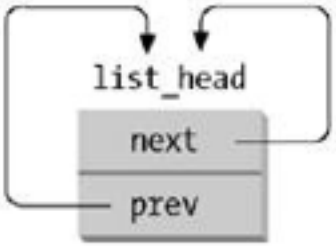


进程链表的操作

- 参见include/linux/list.h



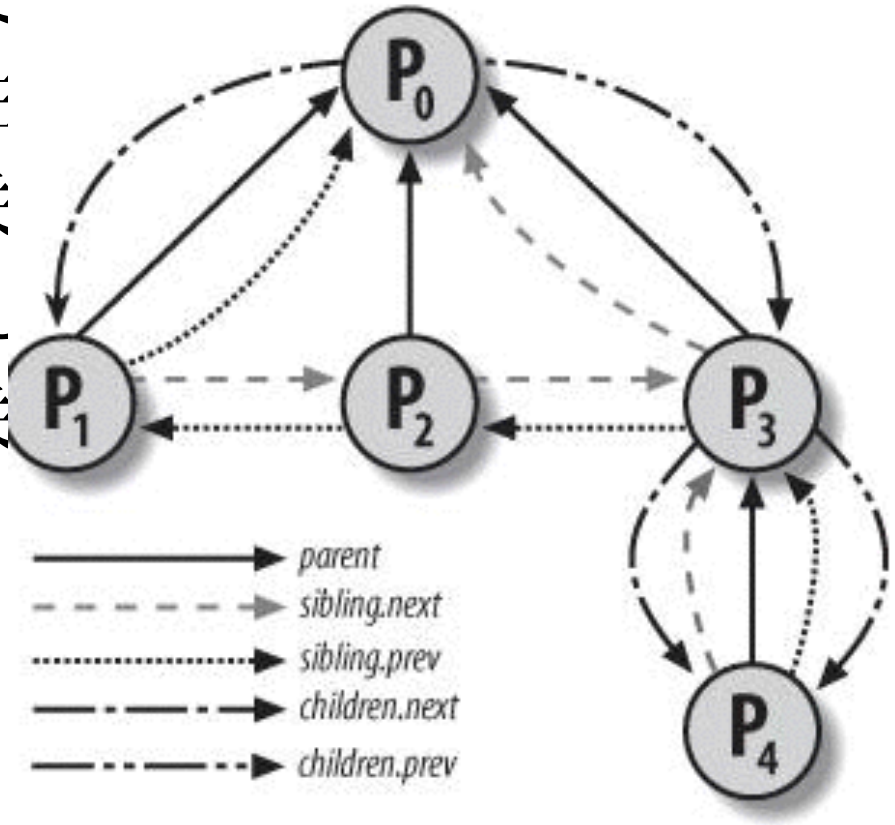
(a) a doubly linked listed with three elements



(b) an empty doubly linked list

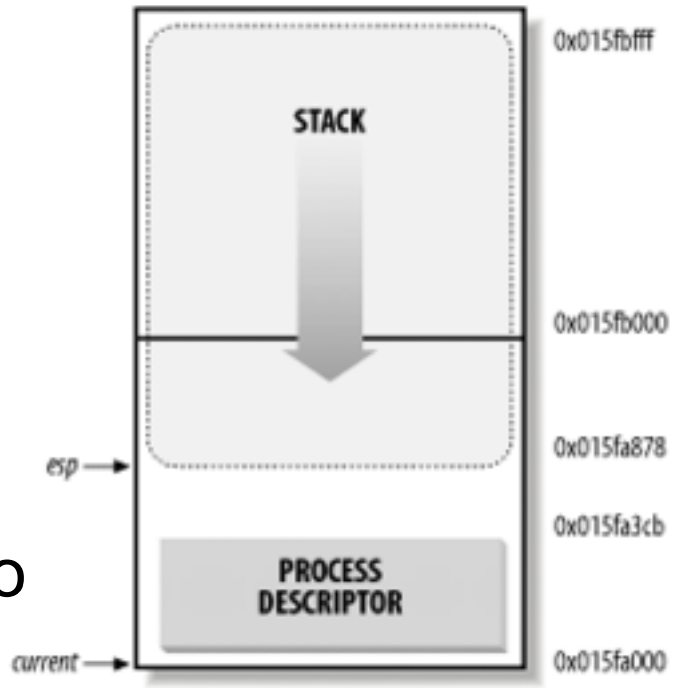
进程之间的亲属关系

- 程序创建的进程具有父子关系，在编程时往往需要引用这样的父子关系。进程描述符中有几个域用来表示这样的关系



进程的内核堆栈

- Linux为每个进程分配一个8KB大小的内存区域，用于存放该进程两个不同的数据结构：
  - Thread_info
  - 进程的内核堆栈
- 进程处于内核态时使用，不同于用户态堆栈
- 内核控制路径所用的堆栈很少，因此对栈和Thread_info来说，8KB足够了



- thread_union(include/linux/sched.h)

```
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

- thread_info由体系结构相关部分定义阅读arch/x86/include/asm/thread_info.h

不论你多么富有，多么有权势，当生命结束之时，所有的一切都只能留在世界上，唯有灵魂跟着你走下一段旅程。人生不是一场物质的盛宴，而是一次灵魂的修炼，使它在谢幕之时比开幕之初更为高尚。
——稻盛和夫

谢谢大家！

参考资料：
《深入理解Linux内核》第三版