进程基础

一口Linux

无线传感器网项目实战

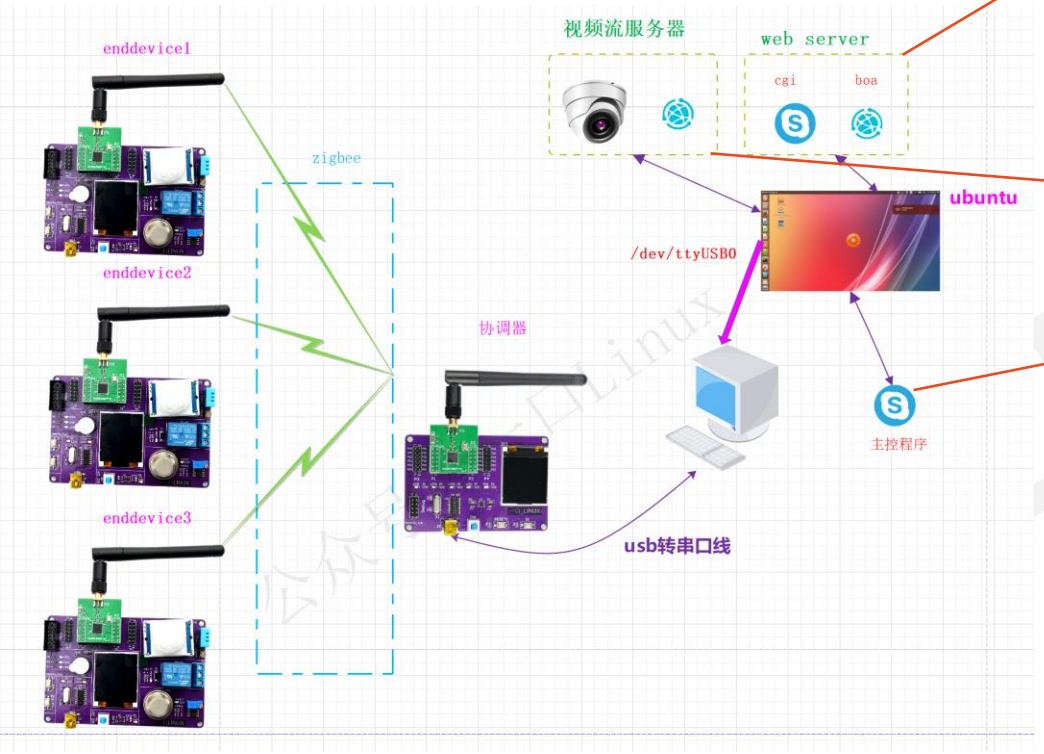
公众号:一口Linux

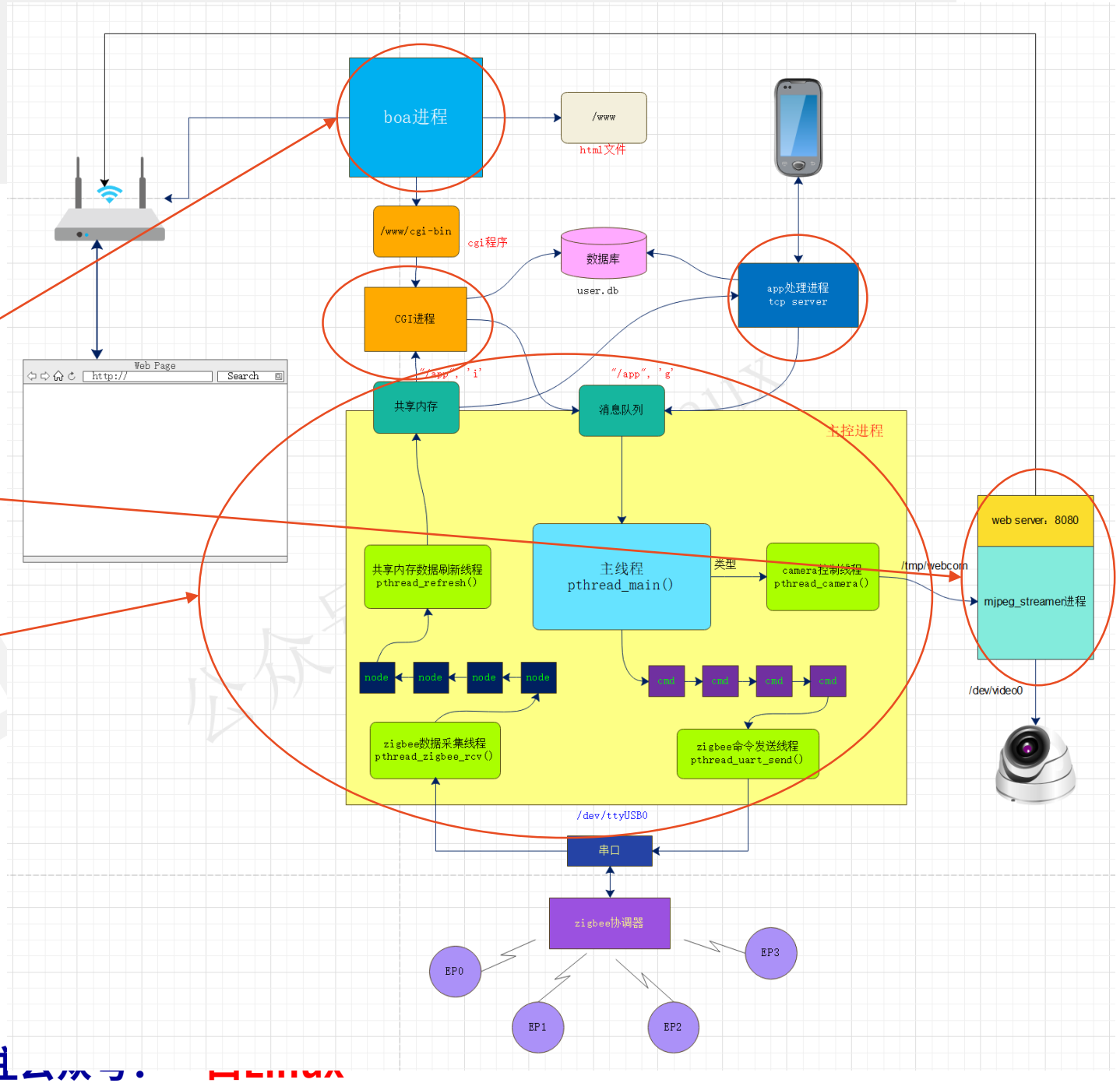
彭老师个人微信号

01

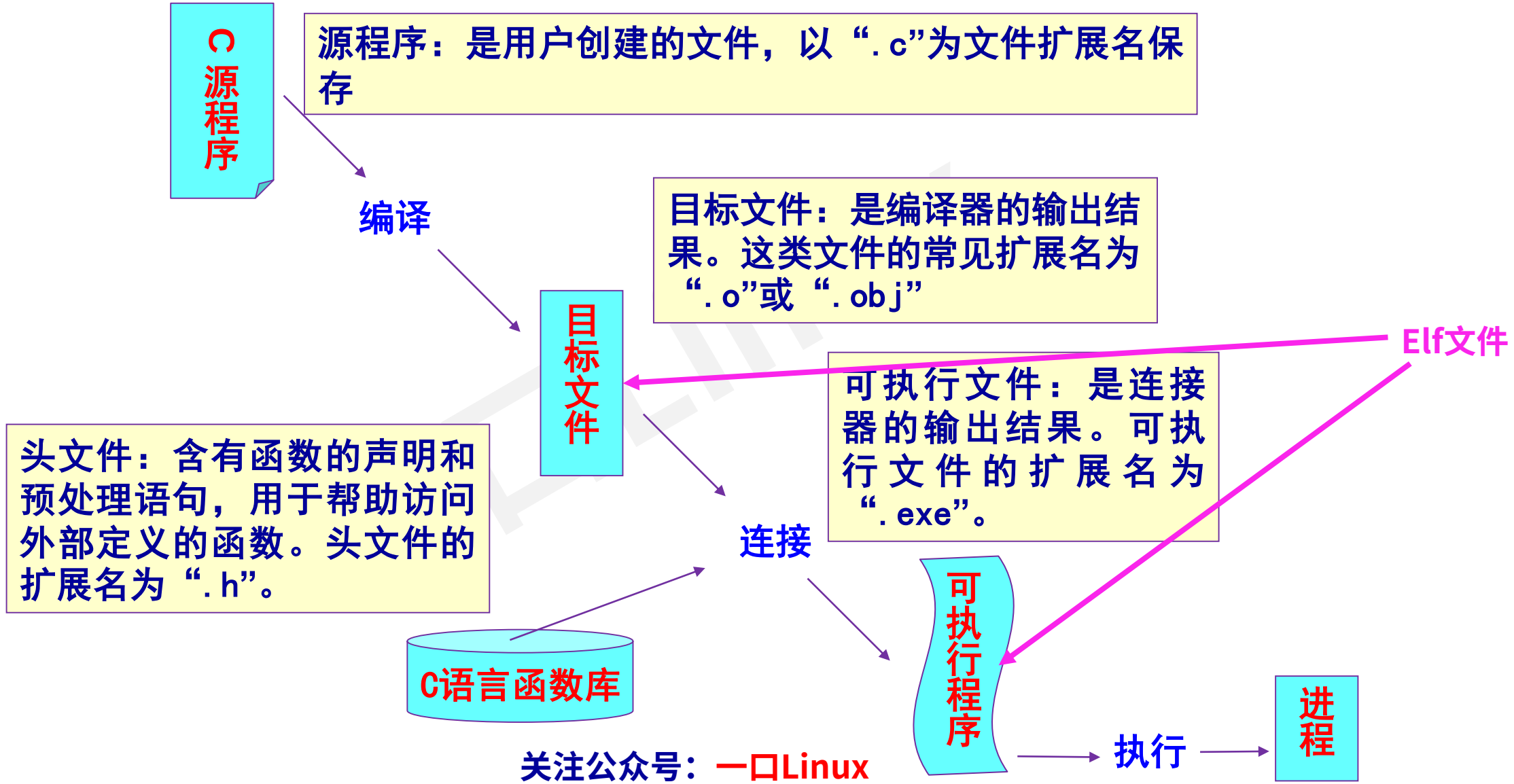
程序、进程概念

项目中的进程





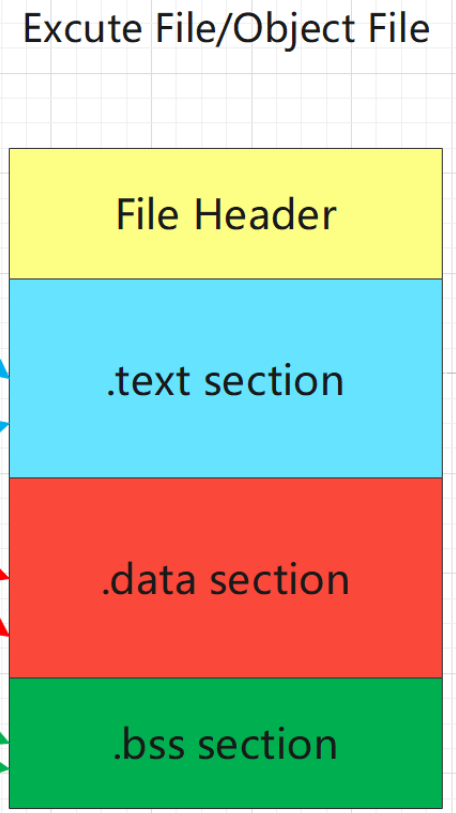
C源码如何变成进程


C源程序
源程序：是用户创建的文件，以“.c”为文件扩展名保存
编译
目标文件：是编译器的输出结果。这类文件的常见扩展名为“.o”或“.obj”
目标文件
Elf文件
可执行文件：是连接器的输出结果。可执行文件的扩展名为“.exe”。
头文件：含有函数的声明和预处理语句，用于帮助访问外部定义的函数。头文件的扩展名为“.h”。
连接
C语言函数库
可执行程序
执行
进程
关注公众号：一口Linux

C源码到可执行文件/目标文件

```c
int global_init_var = 99;
int global_uninit_var;

void func(int i)
{
    printf("%d\n",i);
}

int main(void)
{
    static int static_var1 = 85;
    static int static_var2;

    int a = 1;
    int b;
    func(static_var1 + static_var2 +a + b);
    return 0;
}
```

堆栈中

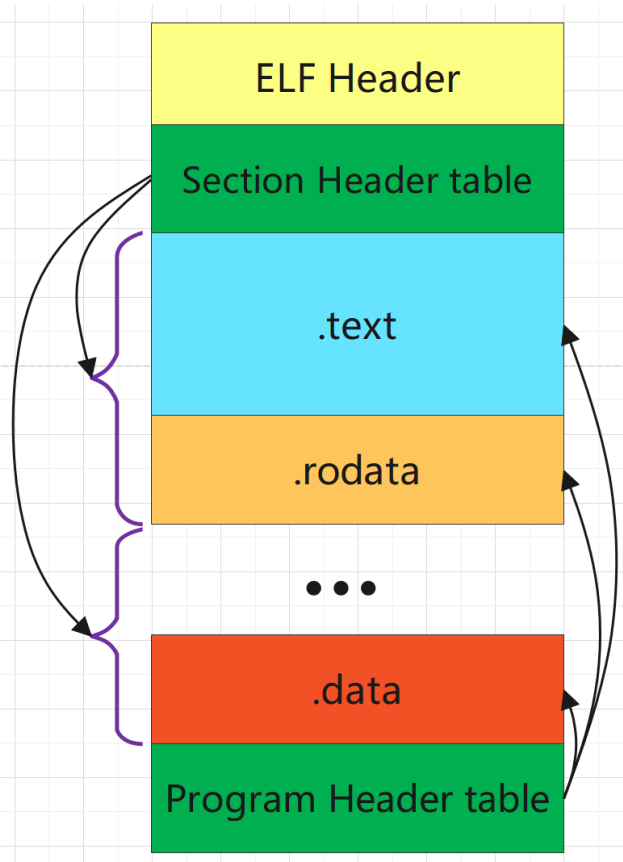


ELF文件

查看命令 readelf

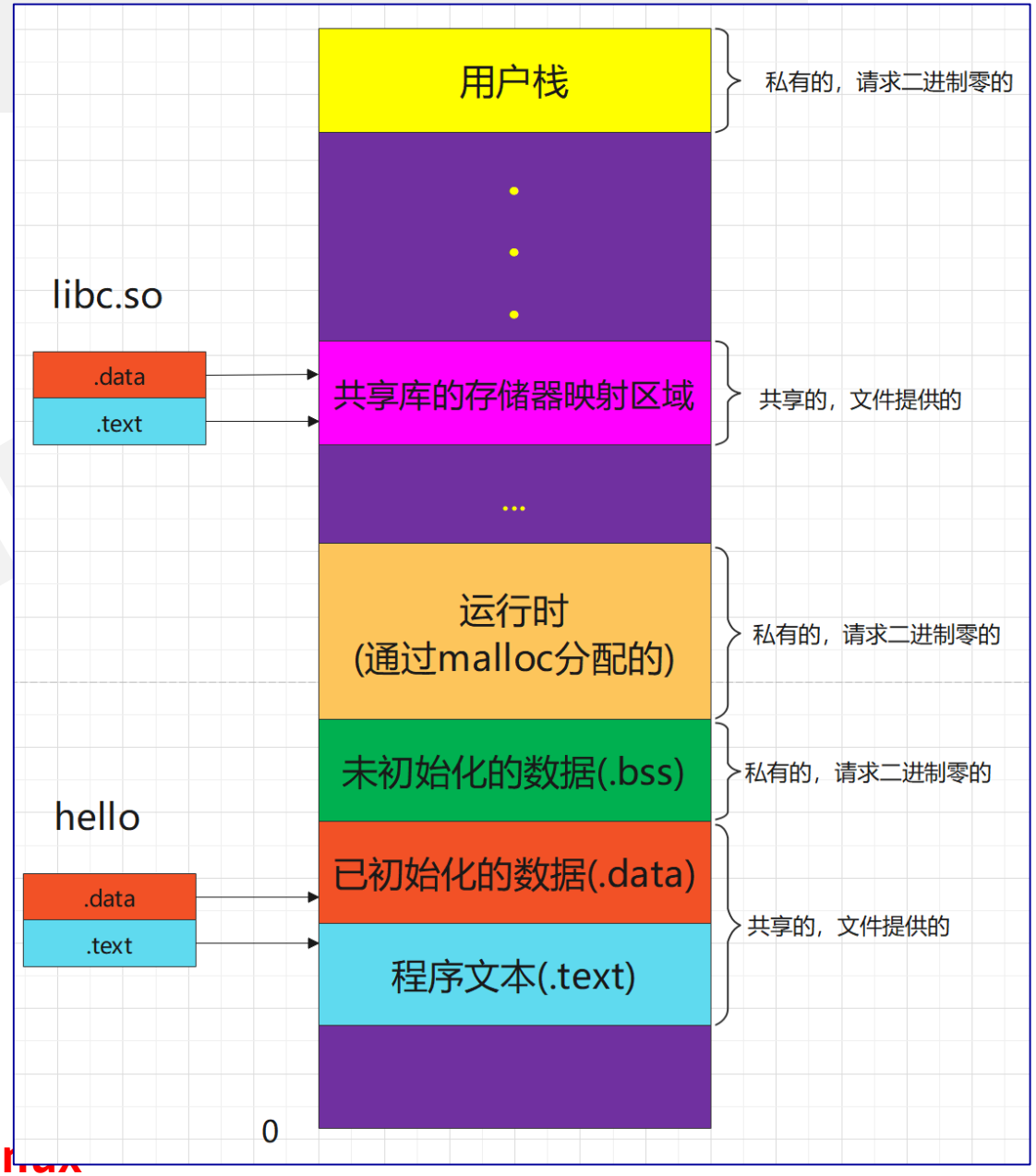
- ELF：Executable and Linkable Format.
- Linux系统上的可执行文件，.o文件，共享库，coredump文件都是ELF格式。
  - ELF header在文件开始处描述了整个文件的组织，Section提供了目标文件的各项信息（如指令、数据、符号表、重定位信息等）
  - section header table包含每一个section的入口，给出名字、大小等信息
  - Program header table指出怎样创建进程映像，含有每个program header的入口
- elf中常见的段有如下几种：

代码段 text 存放函数指令
数据段 data 存放已初始化的全局变量和静态变量
只读数据段 rodata 存放只读常量或const关键字标识的全局变量
bss段 bss 存放未初始化的全局变量和静态变量，这些变量由于未初始化，所以没有必要在elf中为其分配空间。bss段的长度总为0
调试信息段 debug 存放调试信息
行号表 line 存放编译器代码行号和指令的duiing关系
字符串表 strtab 存储elf中存储的各种字符串
符号表 symtab elf中到处的符号，用于链接



ELF到进程

- 进程是一个程序被OS加载到RAM中执行后的一个完整的执行环境。
  - 正文段：
    - 存放被执行的机器指令。
  - 用户数据段：
    - 存放的是全局变量、常数以及动态数据分配的数据空间(如malloc函数取得的空间)等。
  - 堆栈段：
    - 存放的是函数的返回地址、函数的参数以及程序中的局部变量。



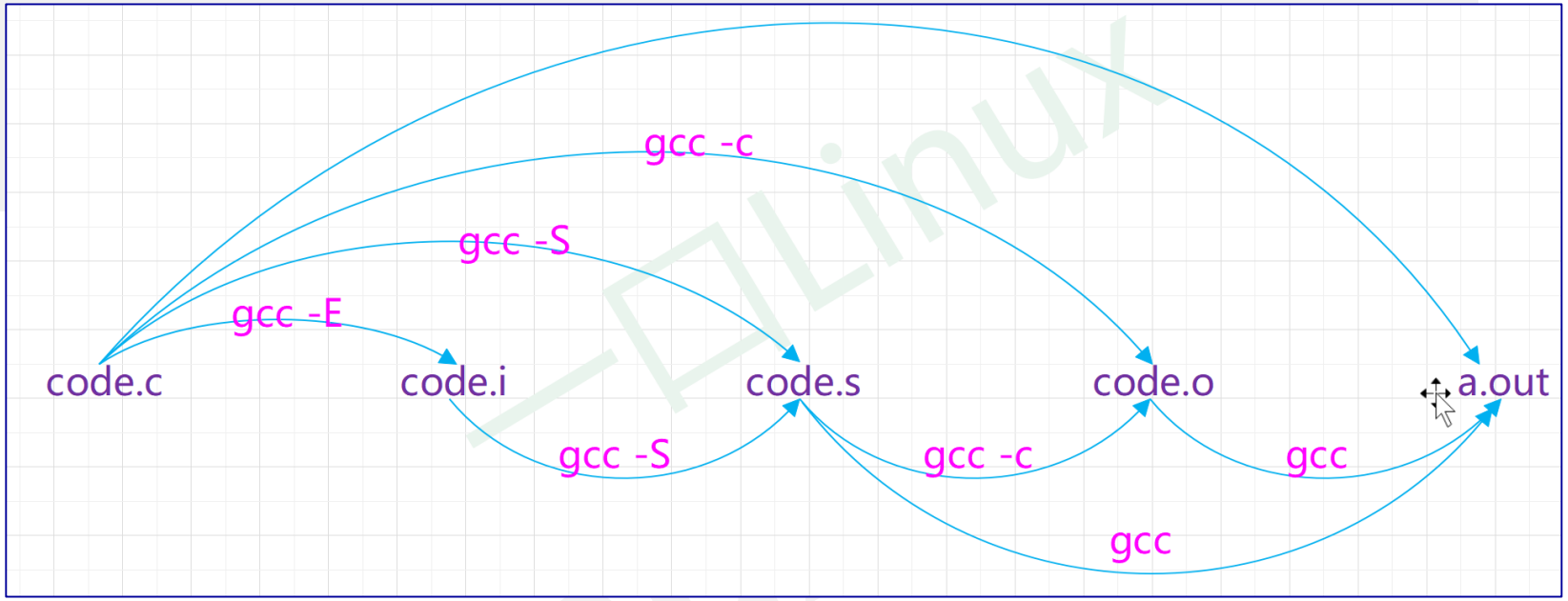
进程和程序

- 1. 进程是一个独立的可调度的任务
  - 进程是一个抽象实体。当系统在执行某个程序时，分配和释放的各种资源
- 2. 进程是一个程序的一次执行的过程
- 3. 进程和程序的区别
  - 程序是静态的，它是一些保存在磁盘上的指令的有序集合，没有任何执行的概念
  - 进程是一个动态的概念，它是程序执行的过程，包括创建、调度和消亡
- 4. 进程是程序执行和资源管理的最小单位 lwp

比如，linux的vi编辑器，它就是一段在linux下用于文本编辑的工具，那么它是一个程序，而我们在linux终端中，可以分别开启两个vi编辑器的进程。

编译命令

stallman



选项	作用
-V	显示制作GCC工具自身时的配置命令； 同时显示编译器驱动程序、预处理器、编译器的版本号
-E	预处理后即停止，不进行编译。预处理后的代码送往标准输出。GCC忽略任何不需要预处理的输入文件
-c	编译、汇编到目标代码，不进行链接。缺省情况下，GCC通过用`.o'替换源文件名的后缀`.c'， `.i'， `.s'等，产生OBJ文件名。可以使用-o 选项选择其他名字。GCC忽略-c 选项后面任何无法识别的输入文件。
-o	输出到指定文件。如果没有指定，则输出到a.out。无论是预处理、编译、汇编还是连接，这个选项都可以使用。
-S	编译后即停止，不进行汇编。对于每个输入的非汇编语言文件，输出结果是汇编语言文件。缺省情况下， GCC 通过用`.s'替换源文件名后缀`.c'， `.i'等等，产生汇编文件名。可以使用-o 选项选择其他名字。 GCC 忽略任何不需要汇编的输入文件

02

进程操作、5状态

启动进程

- 1. 手工启动
  - 由用户输入命令直接启动进程 ls ./a.out
  - 前台运行和后台运行 ./run &

- 2. 调度启动
  - 系统根据用户事先的设定自行启动进程
  - at
    - 在指定时刻执行相关进程，
  - crontab
    - 周期性执行相关进程

5.linux命令at.pdf

点击

进程相关的几个命令

- ps
  - 列出系统中当前运行的那些进程。
- top
  - 实时显示系统中各个进程的资源占用状况，类似于Windows的任务管理器。
- kill
  - 向Linux系统的内核发送一个系统操作信号和某个程序的进程标识号，然后系统内核就可以对进程标识号指定的进程进行操作。
- nice/renice
  - 优先级操作
- bg
  - 将一个在后台暂停的命令，变成继续执行。
- fg
  - 将后台中的命令调至前台继续运行。

ps

实例

```
peng@ubuntu:~$ ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 04:36 ?        00:00:01 /sbin/init auto noprompt
root           2       0  0 04:36 ?        00:00:00 [kthreadd]
root           4       2  0 04:36 ?        00:00:00 [kworker/0:0H]
root           6       2  0 04:36 ?        00:00:00 [mm_percpu_wq]
root           7       2  0 04:36 ?        00:00:00 [ksoftirqd/0]
root           8       2  0 04:36 ?        00:00:00 [rcu_sched]
root           9       2  0 04:36 ?        00:00:00 [rcu_bh]
root          10       2  0 04:36 ?        00:00:00 [migration/0]
root          11       2  0 04:36 ?        00:00:00 [watchdog/0]
root          12       2  0 04:36 ?        00:00:00 [cpuhp/0]
```

参数

命令参数	说明
-e	显示所有进程.
-f	全格式。
-h	不显示标题。
-l	长格式。
-w	宽输出。
-a	显示终端上的所有进程，包括其他用户的进程。
-r	只显示正在运行的进程。
-u	以用户为主的格式来显示程序状况。
-x	显示所有程序，不以终端机来区分。

命令参数	说明
UID:	程序被该 UID 所拥有，指的是用户ID
PID:	就是这个程序的 ID
PPID：	PID的上级父进程的ID
C：	CPU使用的资源百分比
STIME：	系统启动时间
TTY:	登入者的终端机位置
TIME：	使用掉的 CPU时间。
CMD:	所下达的指令为何

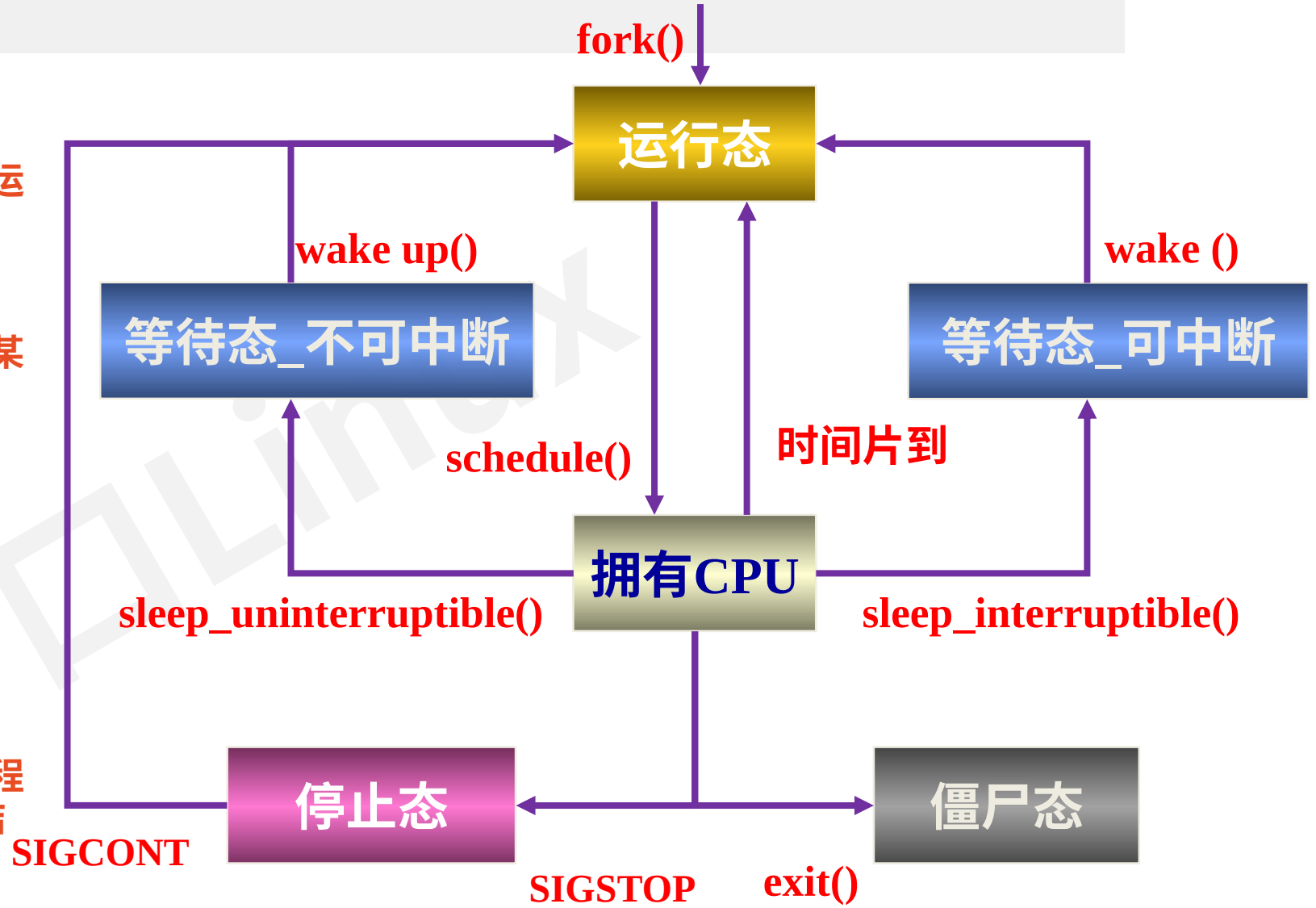
kill

- 本项目只需要知道
  - Kill -9 *pid*　向进程pid发送信号 9:SIGKILL

```
peng@ubuntu:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
```

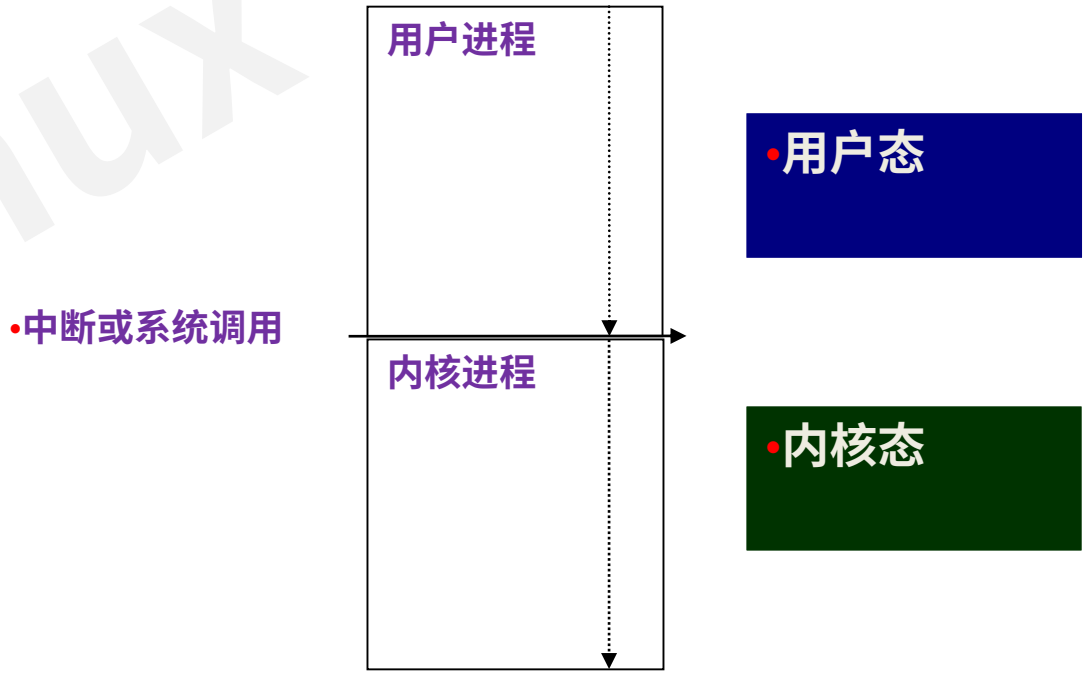
进程运行状态

- 运行态：
  - 此时进程或者正在运行，或者准备运行。
- 等待态：
  - 此时进程在等待一个事件的发生或某种系统资源。
    - ❖ 可中断
    - ❖ 不可中断
- 停止态：
  - 此时进程被中止。
- 死亡态：
  - 这是一个已终止的进程，但还在进程向量数组中占有一个task_struct结构。



进程的模式

- 进程的执行模式分为用户模式和内核模式
- Cpu的模式，用户态下只能访问用户空间
  - 内核模式的代码可以**无限制地访问**所有处理器指令集以及全部内存和I/O空间。
  - 用户模式的进程要享有此特权，它必须通过**系统调用**向设备驱动程序或其他内核模式的代码发出请求。



task_struct

- 内核会创建一个叫进程描述符task_struct的数据结构对象来管理该对象 <linux/sched.h>

state	进程的运行状态。参考sched.h中进程状态的宏定义。运行态，等待态，停止态，死亡态
rt_priority;	表示此进程的运行优先级:
struct mm_struct * mm;	该结构体记录了进程内存使用的相关情况，虚拟地址的概念可以提一下
pid_t pid;	进程号,是进程的唯一标识
pid_t tgid;	线程组id号，
struct task_struct * group_leader;	这个是主线程的进程描述符。线程之所以用进程描叙符表示，因为linux并没有单独实现线程的相关结构体，只用一个进程来代替线程，然后对其做一些特殊的处理。
struct list_head th read_group;	这个是该进程所有线程的链表
struct fs_struct *fs;	它包含此进程当前工作目录和根目录、
struct files_struct * files;	打开的文件相关信息结构体。f_mode字段描述该文件是以什么模式创建的:只读、读写、还是只写。f_pos保存文件中下一个读或写将发生的位置

03

Linux进程系统调用

进程系统调用

- fork函数的机制
- exec函数形式的区别
- exit和_exit的区别
- wait和waitpid

本项目主控是单进程、多线程，此处知识点请参考：
《4.知识点相关资料\4.主控\7.[粉丝问答6]子进程进程的父进程关系.pdf》

《4.知识点相关资料\4.主控\5.Linux进程基础.pdf》

进程创建：fork()

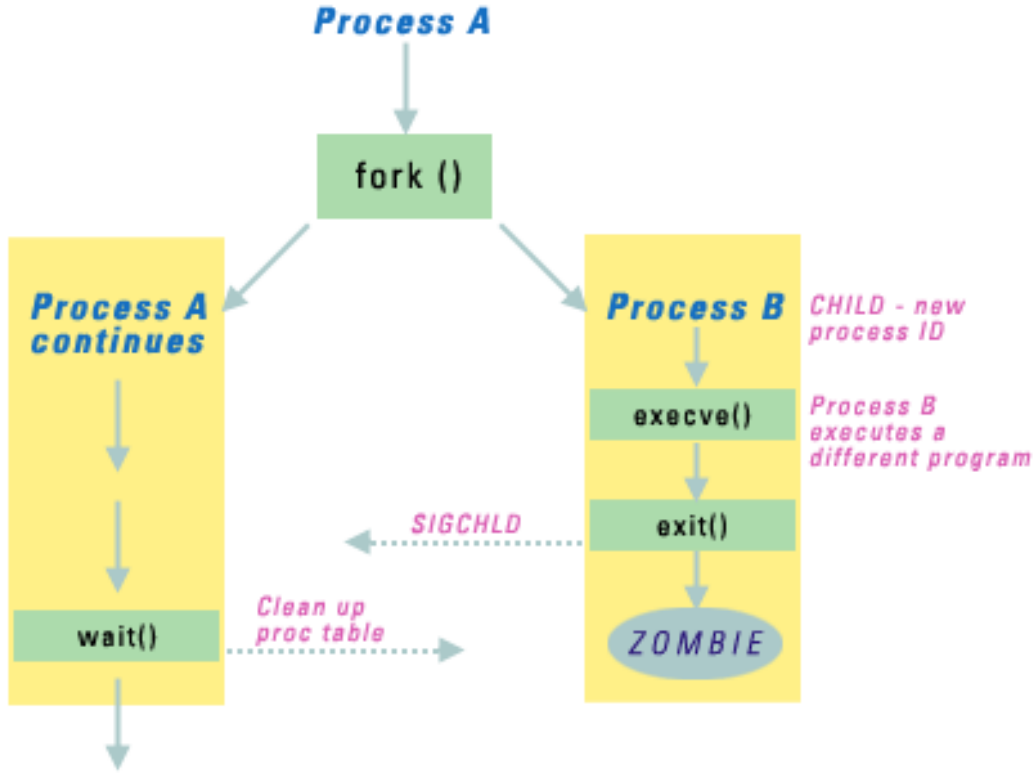
所需头文件	#include <sys/types.h> // 提供类型pid_t的定义 #include <unistd.h>
函数原型	pid_t fork(void);
函数返回值	0：子进程
	子进程PID(大于0的整数)：父进程
	-1：出错

实例

```c
int main()
{
    pid_t pid;

    if ((pid = fork()) == -1) {
        perror("fork");
        return  -1;
    } else if (pid == 0) {
        /* this is child process */
        printf("The return value is %d  In child process!!  My PID is %d,  My PPID is %d\n",
        pid,getpid(), getppid());

    } else {
        /* this is parent */
        printf("The return value is %d In parent process!!  My PID is %d,  My PPID is %d\n",
        pid,getpid(), getppid());
    }
    return  0;
}
```

fork用法

- 1.僵尸进程
  - 1. 父进程 Process A 创建子进程 Process B，当子进程退出时会给父进程发送信号 SIGCHLD；
  - 2. 如果父进程没有调用 wait 等待子进程结束，退出状态丢失，转换成僵死状态，子 进程会变成一个僵尸进程
- 2. 孤儿进程
  - 如果父进程退出，并且没有调用 wait 函数，它的子进程就变成孤儿进程，会被一个特殊进程继承，这就是 init 进程，init 进程会自动清理所有它继承的僵尸进程。



  - 【16.04做了修改由upstart继承
  - ubuntu14 由init继承】

fork函数

- 使用fork函数得到的子进程从父进程的继承了整个进程的地址空间，包括：
  - 进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设置、进程优先级、进程组号、当前工作目录、根目录、资源限制、控制终端等。
- 子进程与父进程的区别在于：
  - 1、父进程设置的锁，子进程不继承（因为如果是排它锁，被继承的话，矛盾了）
  - 2、各自的进程ID和父进程ID不同
  - 3、子进程的未决告警被清除；
  - 4、子进程的未决信号集设置为空集。

vfork函数

- 由于fork完整地拷贝了父进程的整个地址空间，因此执行速度是比较慢的。
- 为了提高效率，Unix系统设计者创建了vfork。
  - vfork也创建新进程，但不产生父进程的副本。
  - 它通过允许父子进程可访问相同物理内存从而伪装了对进程地址空间的真实拷贝，当子进程需要改变内存中数据时才拷贝父进程。
- 这就是著名的“写操作时拷贝”(copy-on-write)技术

exec函数族

- exec函数族提供了一种在进程中启动另一个程序执行的方法。
- 它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的**数据段、代码段和堆栈段**。
- 在执行完之后，原调用进程的内容除了进程号外，其他全部都被替换了。
- 可执行文件既可以是二进制文件，也可以是任何Linux下可执行的脚本文件。

exec函数族-何时使用？

- 当进程认为自己不能再为系统和用户做出任何贡献了时就可以调用exec函数，让自己执行新的程序

- 如果某个进程想同时执行另一个程序，它就可以调用fork函数创建子进程，然后在子进程中调用任何一个exec函数。这样看起来就好像通过执行应用程序而产生了一个新进程一样

exec函数族语法

所需头文件	#include <unistd.h>
函数原型	int execl(const char *path, const char *arg, ...);
	int execv(const char *path, char *const argv[]);
	int execle(const char *path, const char *arg, ..., char *const envp[]);
	int execve(const char *path, char *const argv[], char *const envp[]);
	int execlp(const char *file, const char *arg, ...);
	int execvp(const char *file, char *const argv[]);
函数返回值	-1：出错

exec函数族使用区别

- 可执行文件查找方式 echo $PATH
  - 表中的前四个函数的查找方式都是指定完整的文件目录路径，
  - 而最后两个函数(以p结尾的函数)可以只给出文件名，系统会自动从环境变量“$PATH”所包含的路径中进行查找。
- 参数表传递方式
  - 两种方式：
    - 逐个列举或是将所有参数通过指针数组传递
  - 以函数名的第五位字母来区分，
    - 字母为“l”(list)的表示逐个列举的方式；
    - 字母为“v”(vertor)的表示将所有参数构造成指针数组传递，其语法为char *const argv[]
- 环境变量的使用
  - exec函数族可以默认使用系统的环境变量，也可以传入指定的环境变量。
  - 这里，以“e”(Enviromen)结尾的两个函数execle、execve就可以在envp[]中传递当前进程所使用的环境变量

实例

- p
  - 而最后两个函数(以p结尾的函数)可以只给出文件名，系统会自动从环境变量“$PATH”所包含的路径中进行查找。
- l
  - 字母为“l”(list) 表示参数逐个列举的方式；
- v
  - 字母为“v”(vertor)的表示将所有参数构造成指针数组传递，其语法为char *const argv[]

exec.c

```c
int main()
{
    if (execlp("ps", "ps", "-ef", NULL) < 0)
    {
        perror("execlp error!");
    }
    return 0;
}
```

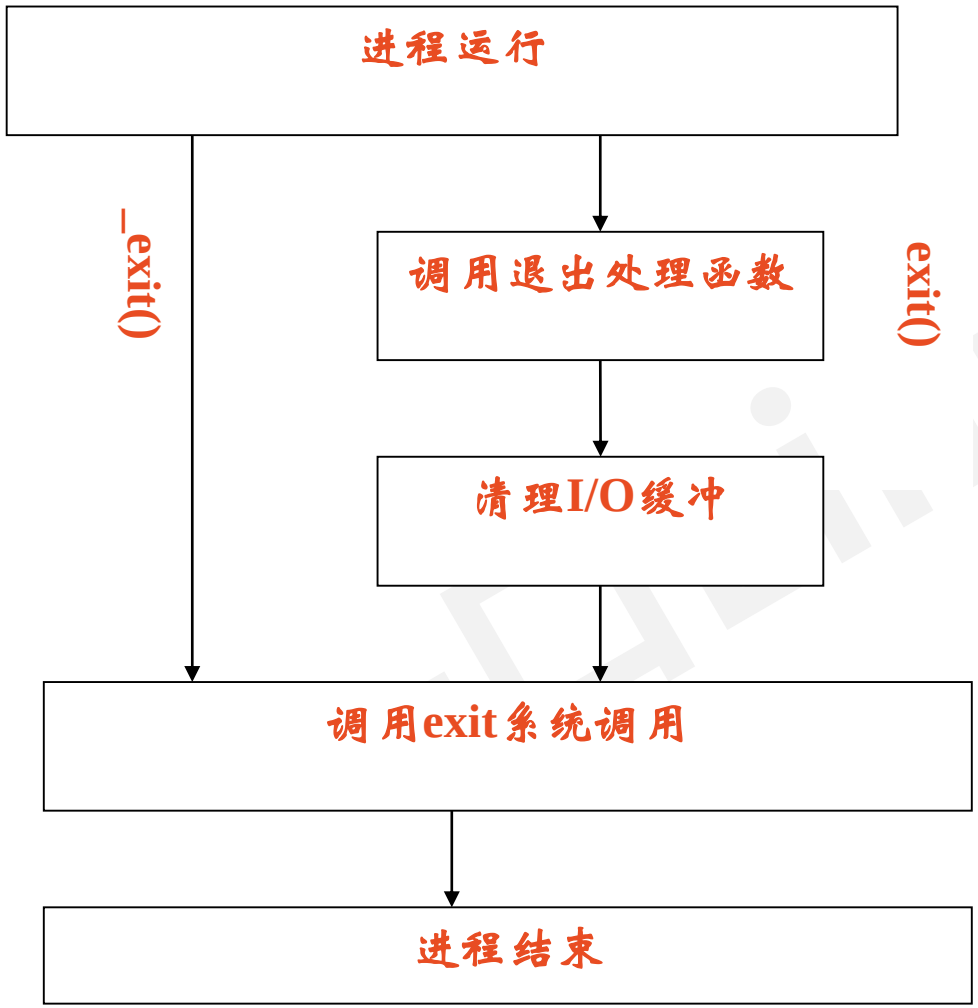
execv.c

```c
int main(int argc,char **argv)
{
    pid_t pid;

    printf("PID = %d\n",getpid());

    pid=fork();
    if(pid==0)
    {
        //子进程
        execvp("ls",argv);    // ./r -ef
//      execv("/bin/ls",argv);  ./run -l
//      execl("/bin/ls","ls","-l","/",NULL);
        //指令路径 指令 参数目录空
//      execlp("ls","ls","-al","/",NULL);
        sleep(10);
    }else if(pid!=-1)
    {
        //父进程
        printf("\nParrent porcess,PID = %d\n",getpid());
    }else
    {
        printf("error fork() child proess!");
    }
    return 0 ;
} « end main »
```

exit和_exit

所需头文件	exit：#include <stdlib.h>
	_exit：#include <unistd.h>
函数原型	exit：void exit(int status);
	_exit：void _exit(int status);
函数传入值	status是一个整型的参数，可以利用这个参数传递进程结束时的状态。通常0表示正常结束；其他的数值表示出现了错误，进程非正常结束。在实际编程时，可以用wait系统调用接收子进程的返回值，进行相应的处理。

exit和_exit



_exit和exit的区别

- _exit()：
  - 直接使进程终止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构；
- exit()
  - 在这些基础上作了一些包装，在执行退出之前加了若干道工序。
  - exit()函数在调用exit系统调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件，就是图中的"清理I/O缓冲"一项。

exit和_exit

```
int main()
{
   printf("Using exit...\n");
   printf("This is the end");
   exit(0);
}
```

```
int main()
{
  printf("Using _exit...\n");
  printf("This is the end");
  _exit(0);
}
```

有什么区别?

wait和waitpid

- wait函数
  - 调用该函数使进程阻塞，直到任一个子进程结束或者是该进程接收到了一个信号为止。如果该进程没有子进程或者其子进程已经结束，wait函数会立即返回。

- waitpid函数
  - 功能和wait函数类似。可以指定等待某个子进程结束以及等待的方式（阻塞或非阻塞）

wait

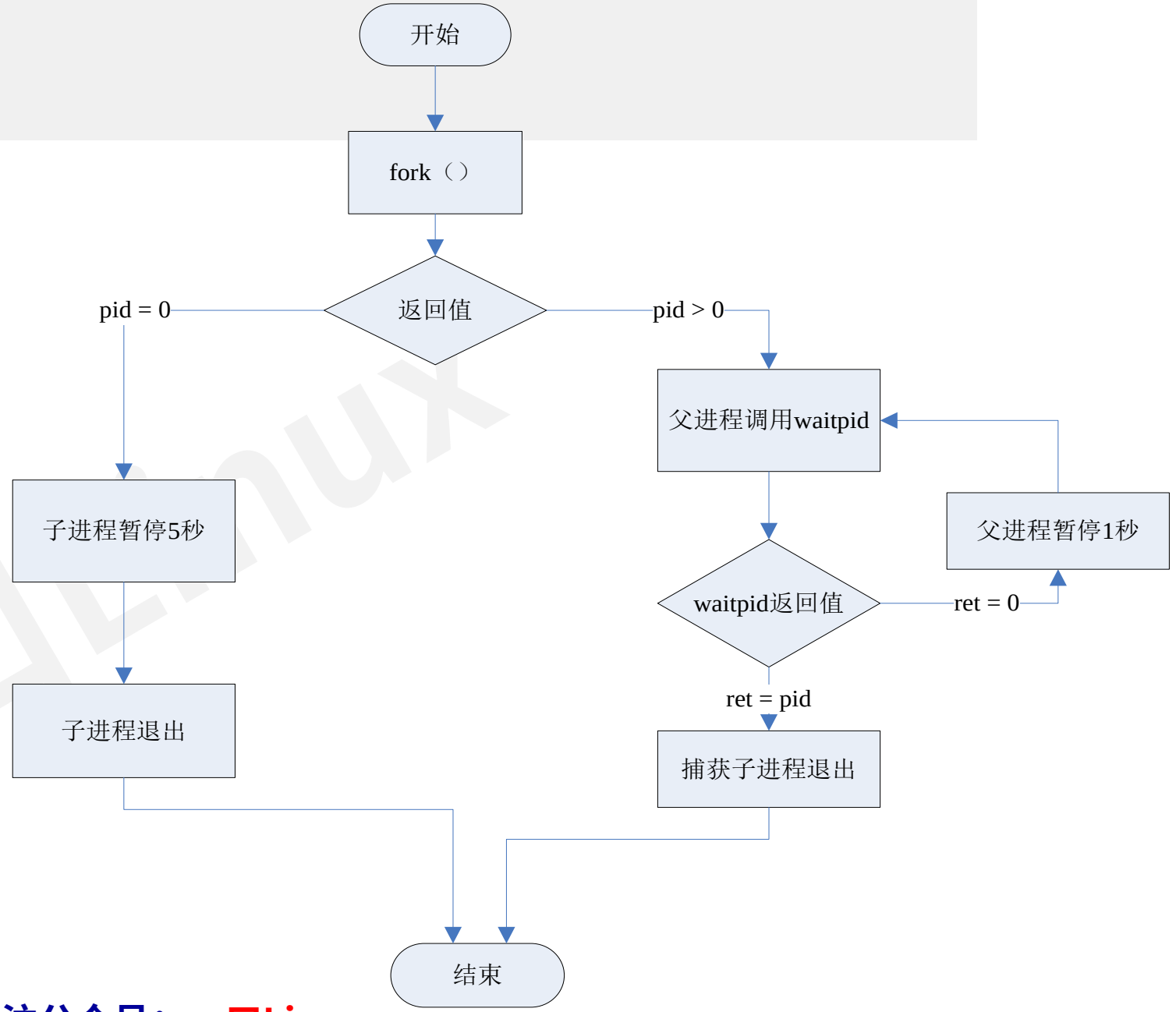
所需头文件	#include <sys/types.h> #include <sys/wait.h>
函数原型	pid_t wait(int *status)
函数参数	status是一个整型指针，指向的对象用来保存子进程退出时的状态。 • status若为空，表示忽略子进程退出时的状态 • status若不为空，表示保存子进程退出时的状态 另外，子进程的结束状态可由Linux中一些特定的宏来测定。
函数返回值	成功：子进程的进程号 失败：-1

waitpid

所需头文件	#include <sys/types.h> #include <sys/wait.h>	
函数原型	pid_t waitpid(pid_t pid, int *status, int options)	
函数参数	pid	pid>0：只等待进程ID等于pid的子进程，不管已经有其他子进程运行结束退出了，只要指定的子进程还没有结束，waitpid就会一直等下去。
		pid= -1：等待任何一个子进程退出，此时和wait作用一样。
		pid=0：等待其组ID等于调用进程的组ID的任一子进程。
		pid<-1：等待其组ID等于pid的绝对值的任一子进程。
	status	同wait
	options	WNOHANG：若由pid指定的子进程并不立即可用，则waitpid不阻塞，此时返回值为0
		WUNTRACED：若某实现支持作业控制，则由pid指定的任一子进程状态已暂停，且其状态自暂停以来还未报告过，则返回其状态。
		0：同wait，阻塞父进程，等待子进程退出。
函数返回值	正常：结束的子进程的进程号	
	使用选项WNOHANG且没有子进程结束时：0	
	调用出错：-1	

举例

- wait_z.c
- waitpid.c



守护进程

- 请参考以下文章
  - 《搞懂进程组、会话、控制终端关系，才能明白守护进程干嘛的？》
  - 物联网实训项目所有资料\4.知识点相关资料\4.主控\6.守护进程.pdf

更多嵌入式Linux知识
请关注一口君的公众号：一口Linux