联航精英训练营

UNIGRESS ELITE TRAINING CAMP

Linux高级编程（三）

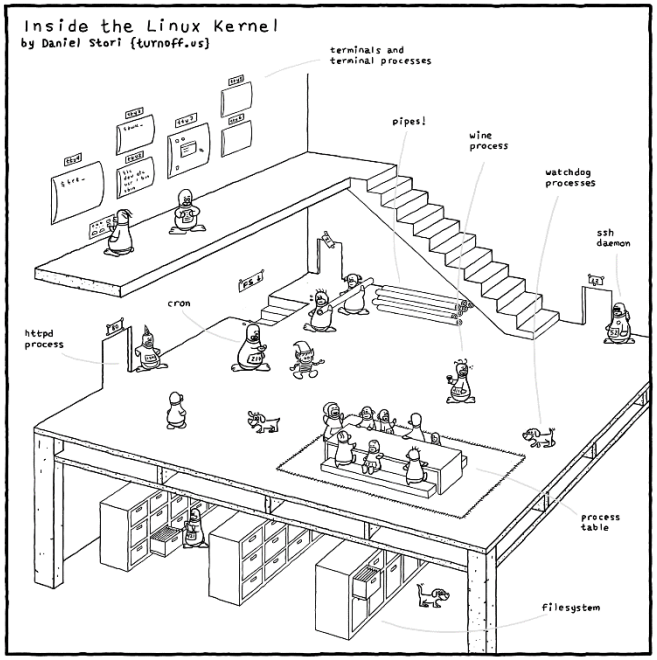
张勇涛

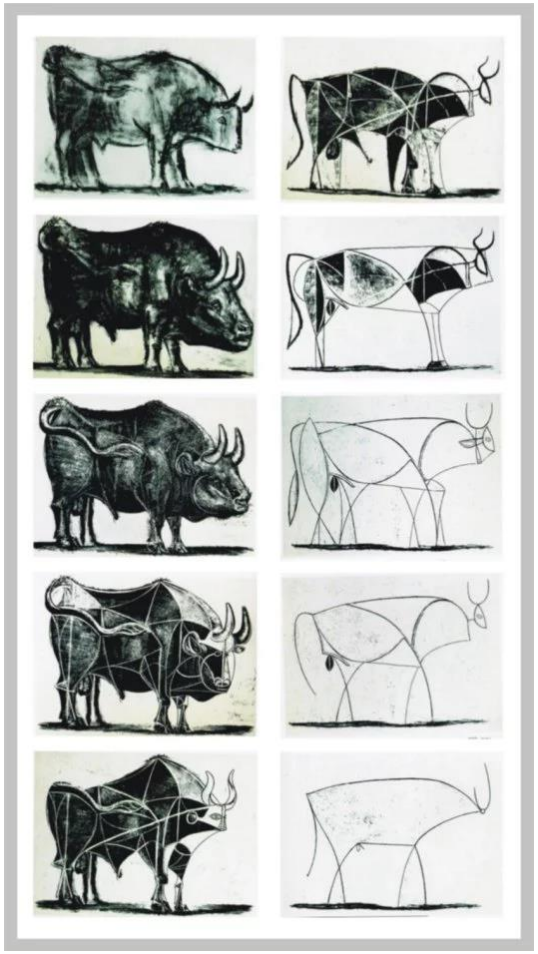
抽象

- 抽象来源于具体,但又超越具体

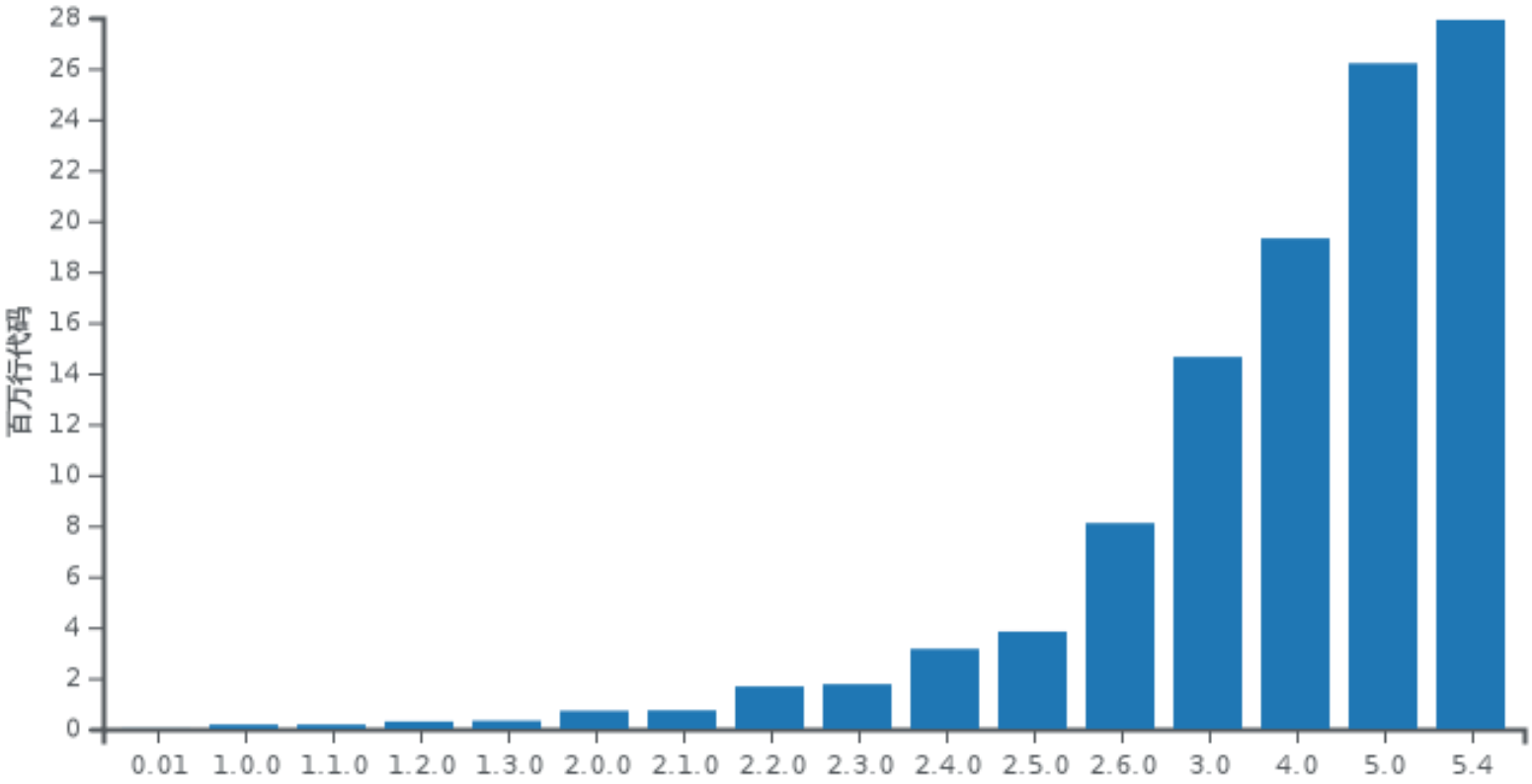
  抽象是从众多的事物中抽取出共同的、本质性的特征，而舍弃其非本质的特征。

- 操作系统与抽象





Linux内核代码量



内核态与用户态

- 内核态就是拥有多资源的状态,或者说访问资源多的状态,也称为特权态.

- 用户态就是非特权态.

- 内核态和用户态各有优势:

内核态：资源多, 但是安全性、可靠性要求高，维护较复杂

用户态：资源受限，但可靠性、安全性要求低，编写维护比较简单

态势的识别

- 所谓的用户态、内核态实际上是处理器的一种状态，而不是程序的状态。
- 通过设置状态字来设置用户态、内核态，以及其它状态

程序和进程

- **程序**（program）是存放在磁盘文件中的可执行文件。
- **进程**: 程序的执行实例被称为进程(process)。

  进程ID：每个linux进程都一定有一个唯一的数字标识符，称为进程ID（process ID）。进程ID总是一非负整数。

进程相关基本概念

- 进程的定义
  - 进程是一个独立的可调度的活动
  - 进程是一个抽象实体，当它执行某个任务时，将要分配和释放各种资源
  - 进程是可以并行执行的计算部分
- 进程是一个程序的一次执行的过程
- 进程是程序执行和资源管理的最小单位

进程与程序的区别

- 进程是动态的，程序是静态的：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，包括了动态创建、调度和消亡的整个过程。

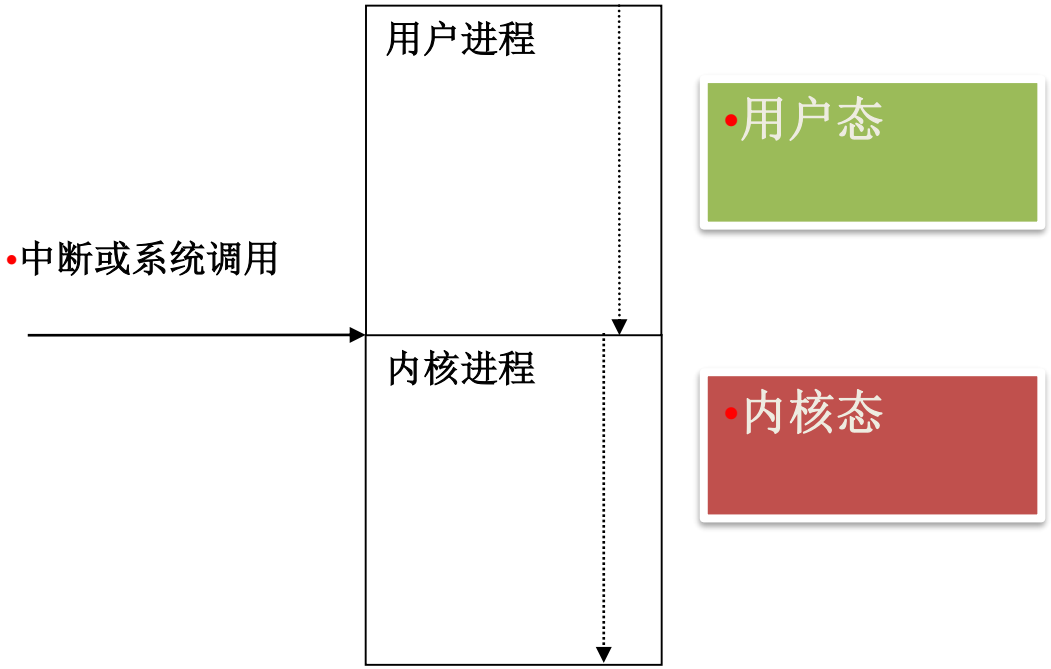
- 进程是暂时的，程序的永久的：进程是一个状态变化的过程，程序可长久保存。

- 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块（即进程状态信息）。

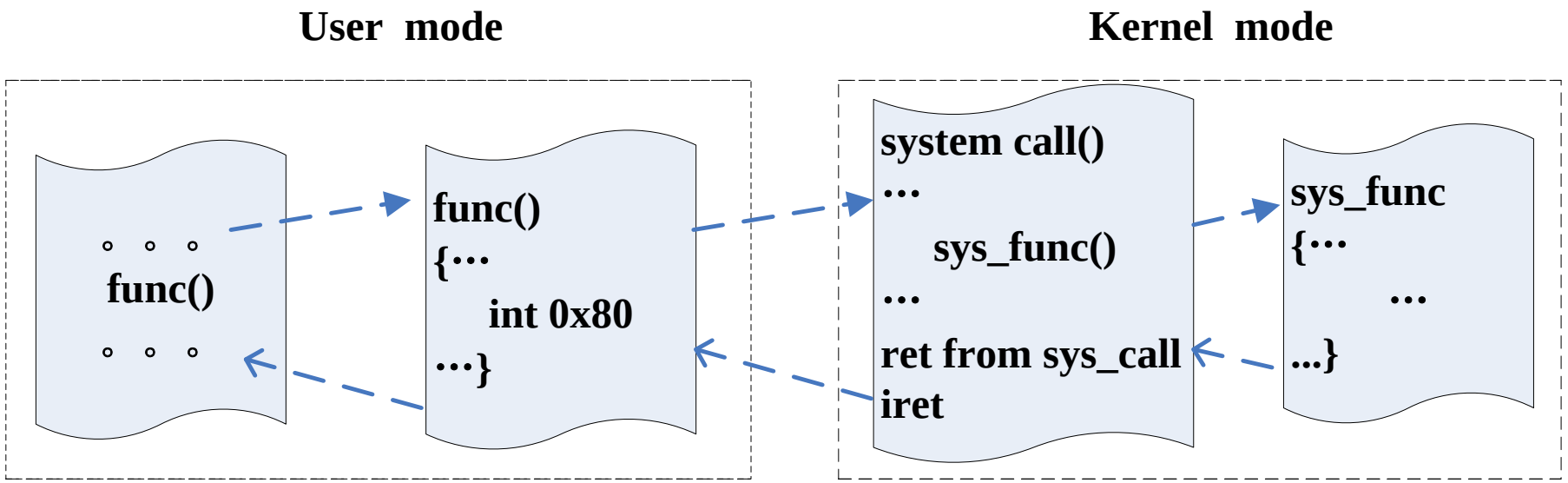
- 进程与程序的对应关系：通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包括多个程序。

Linux下的进程的模式和类型

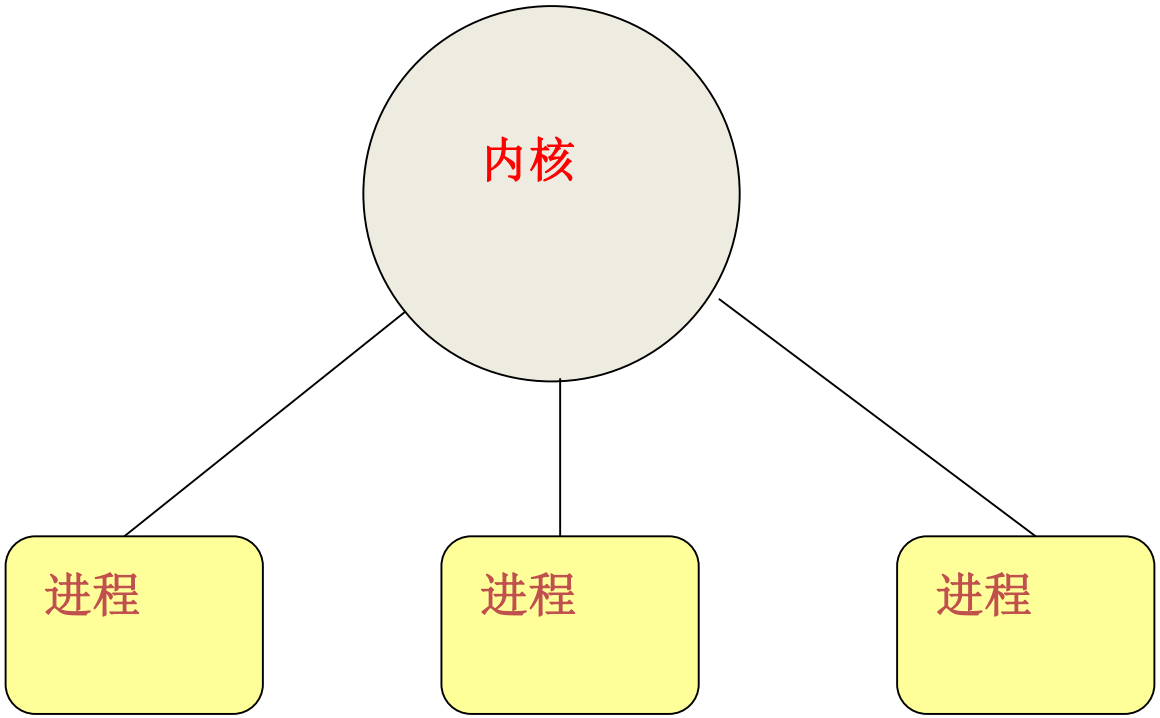
- 进程的执行模式划分为用户模式和内核模式



用户调用与内核调用关系



进程与内核的关系



进程的生命周期

- 创建
  - 每个进程都是由其父进程创建
  - 进程可以创建子进程，子进程又可以创建子进程的子进程
- 运行
  - 多个进程可以同时存在
  - 进程间可以通信
- 撤销
  - 进程可以被撤销，从而结束一个进程的运行。

进程的状态

- 就绪状态（Ready）：
- 阻塞状态（Blocked）：
- 运行状态（Running）：

进程的状态（运行态）

- 运行状态(Running)：

  占用处理器资源；处于此状态的进程的数目小于等于CPU的数目。

  - 在没有其他进程可以执行时（如所有进程都在阻塞状态），通常会自动执行系统的idle进程（相当于空操作）。

进程的状态（就绪态）

- 就绪状态(Ready)：

  进程已获得除处理器外的所需资源，等待分配处理机资源；只要分配CPU就可执行。

  - 可以按多个优先级来划分队列，如：时间片用完 - >低优，I/O完成 - >中优，页面调入完成 - >高优

进程的状态（堵塞态）

- 阻塞状态(Blocked)：

由于进程等待某种条件（如I/O操作或进程同步），在条件满足之前无法继续执行。该事件发生前即使把处理器分配给该进程，也无法运行。如：等待I/O操作的完成。

进程的状态

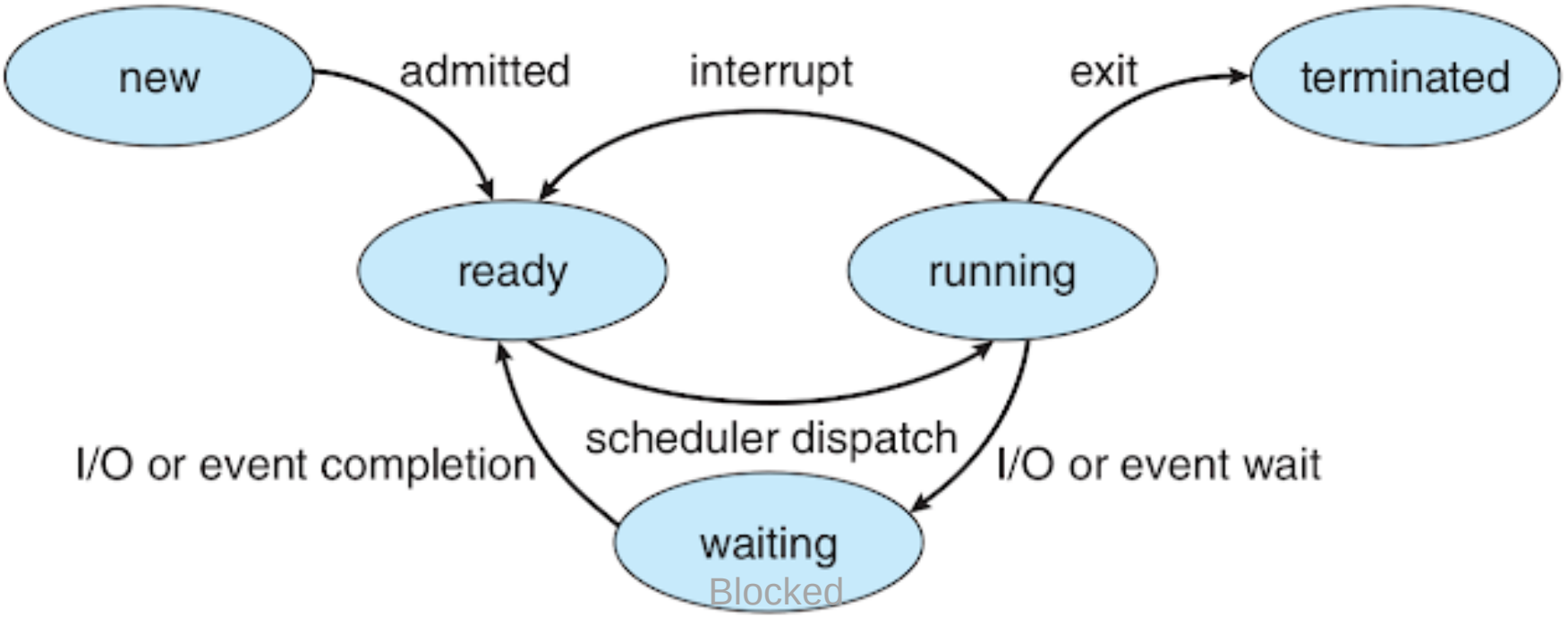
- 创建状态(New)：

  进程刚创建，但还不能运行(一种可能的原因是OS对并发进程数的限制)；如：分配和建立PCB表项（可能有数目限制）、建立资源表格（如打开文件表）并分配资源，加载程序并建立地址空间表。

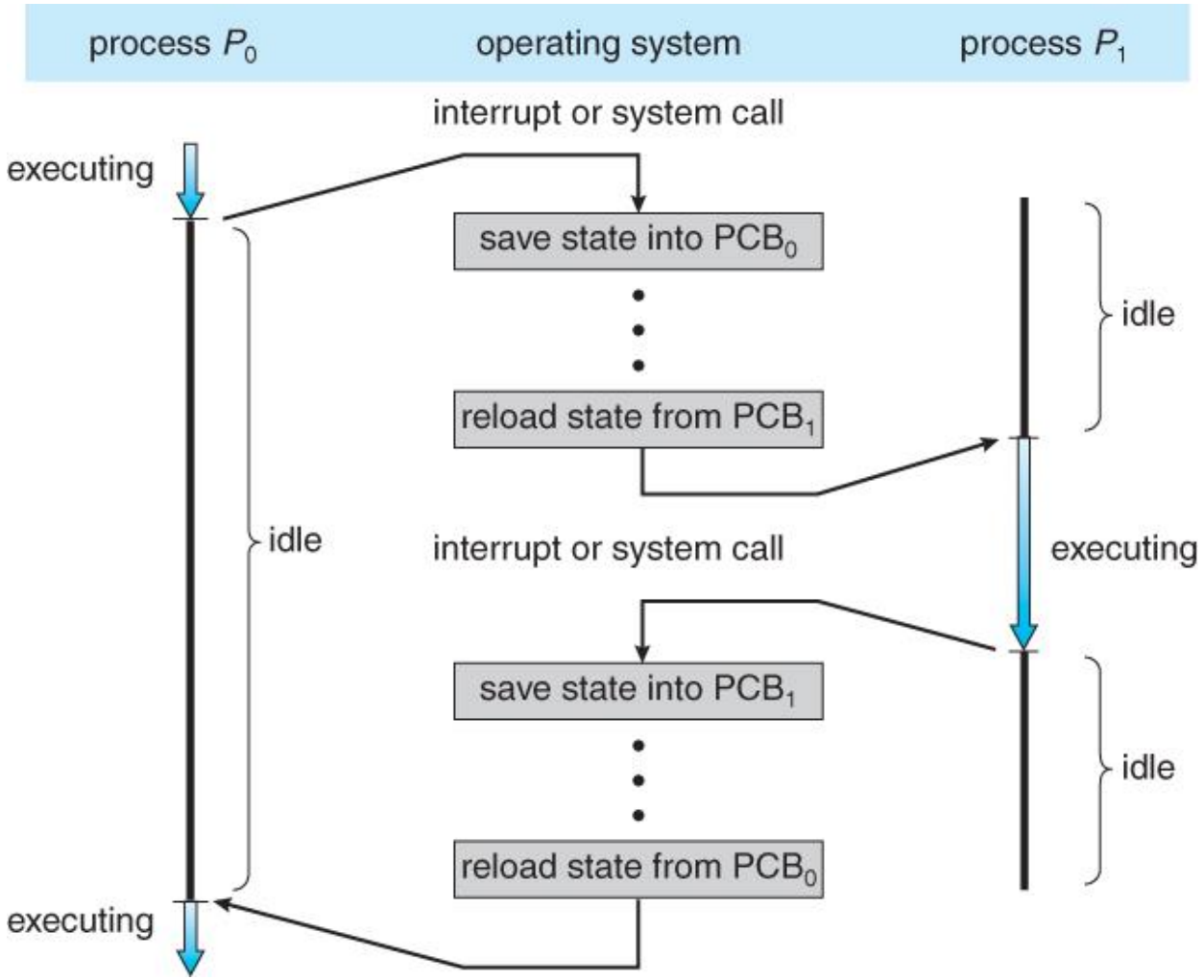
- 结束状态(Exit)：

  进程已结束运行，回收除PCB之外的其他资源，并让其他进程从PCB中收集有关信息（如记帐，将退出码exit code传递给父进程）。

进程状态转换图

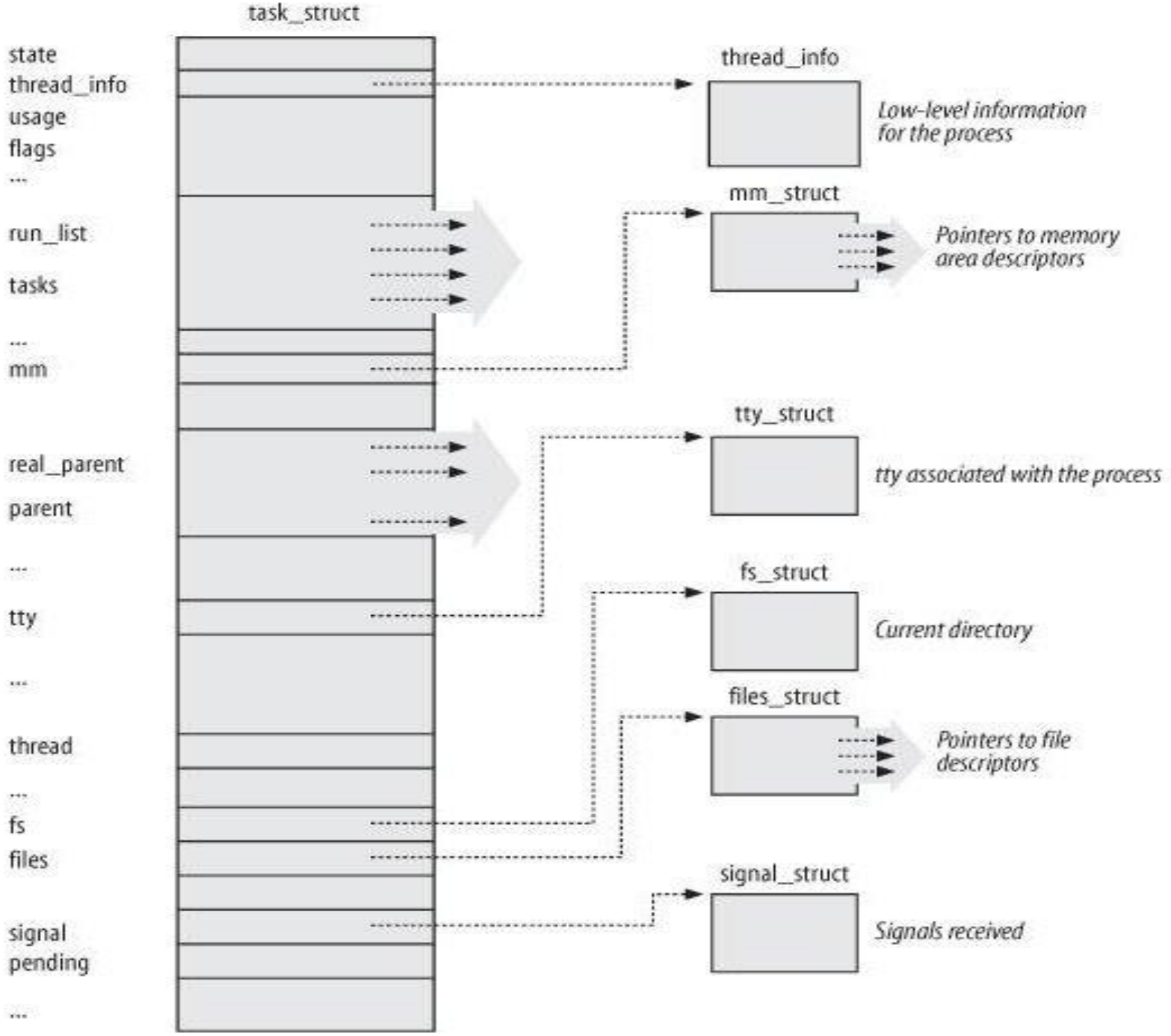


进程切换

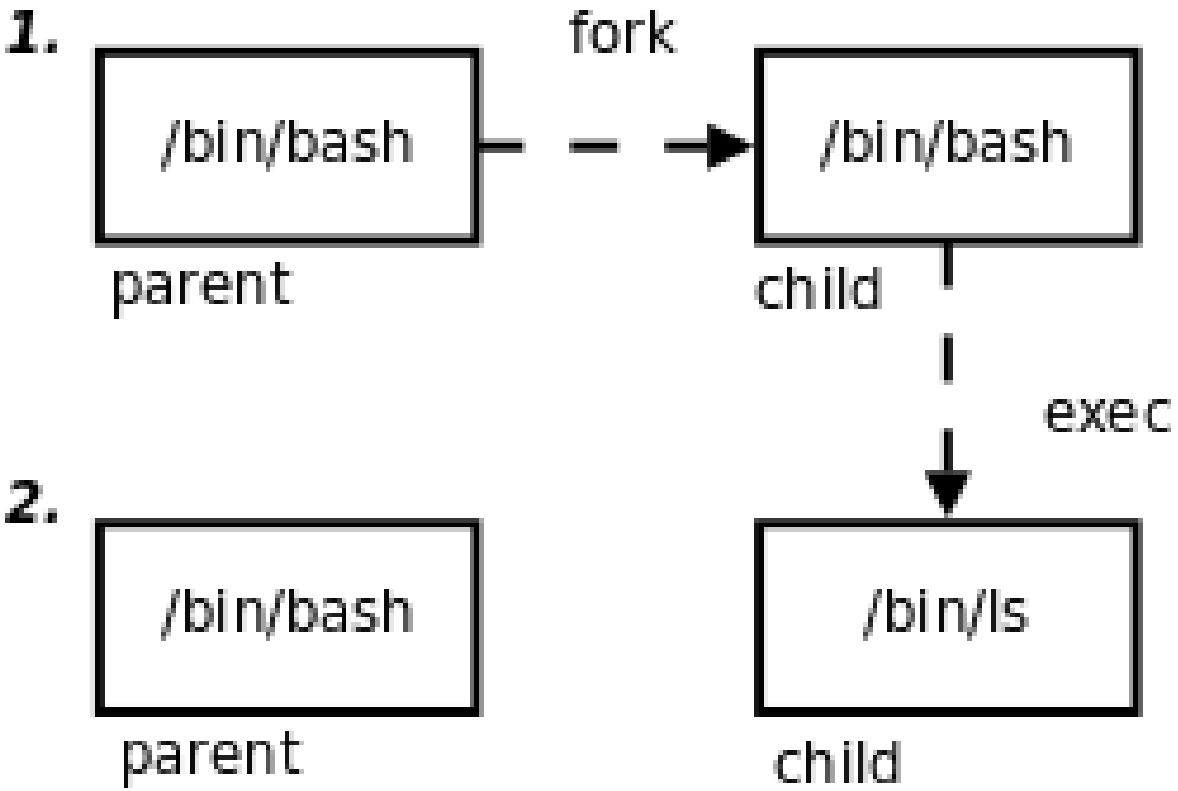


进程控制块（PCB）

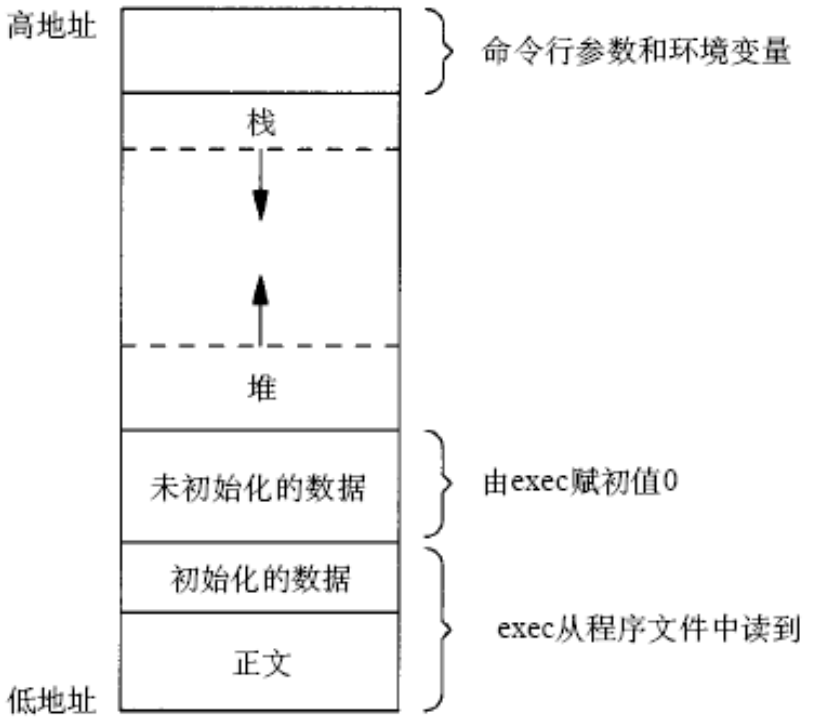
1. 进程id。系统中每个进程有唯一的id，在C语言中用pid_t类型表示，其实就是一个非负整数。
2. 进程的状态，有运行、挂起、停止、僵尸等状态。
3. 进程切换时需要保存和恢复的一些CPU寄存器。
4. 描述虚拟地址空间的信息。
5. 描述控制终端的信息。
6. 当前工作目录（Current Working Directory）。
7. umask掩码。
8. 文件描述符表，包含很多指向file结构体的指针。
9. 和信号相关的信息。
10. 用户id和组id。
11. 控制终端、Session和进程组。
12. 进程可以使用的资源上限（Resource Limit）。

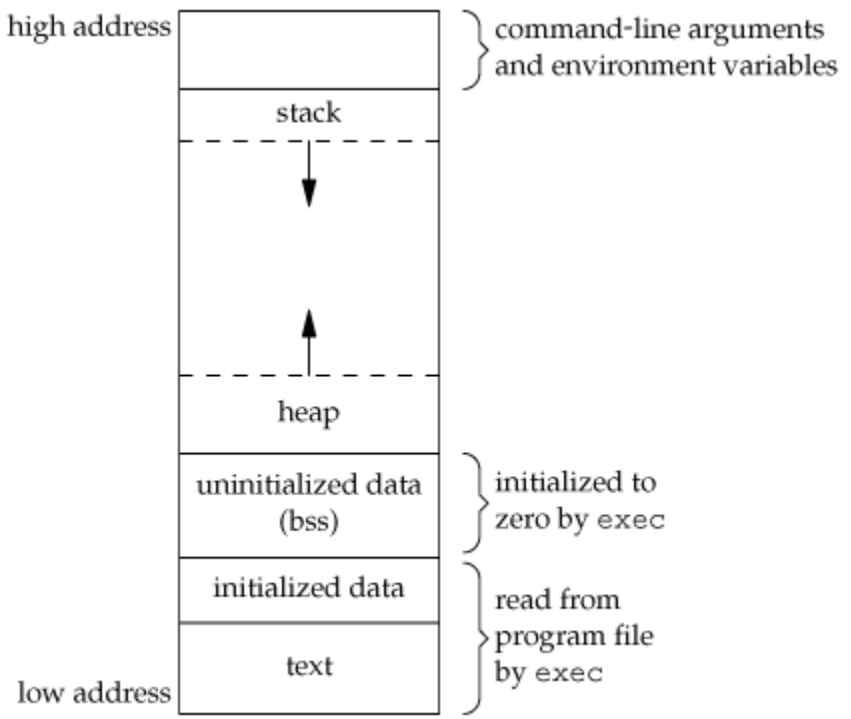

task_struct
state
thread_info
usage
flags
...
run_list
tasks
...
mm
real_parent
parent
...
tty
...
thread
...
fs
files
signal
pending
...
thread_info
Low-level information for the process
mm_struct
Pointers to memory area descriptors
tty_struct
tty associated with the process
fs_struct
Current directory
files_struct
Pointers to file descriptors
signal_struct
Signals received

fork和exec



进程中的内存分布





常用环境变量

- **PATH** 可执行文件的搜索路径。
- **SHELL** 当前Shell，它的值通常是/bin/bash。
- **TERM** 当前终端类型，在图形界面终端下它的值通常是xterm，终端类型决定了一些程序的输出显示方式，比如图形界面终端可以显示汉字，而字符终端一般不行。
- **LANG** 语言和locale，决定了字符编码以及时间、货币等信息的显示格式。
- **HOME** 当前用户主目录的路径，很多程序需要在主目录下保存配置文件，使得每个用户在运行该程序时都有自己的一套配置。

获取环境变量

- 用environ指针可以查看所有环境变量字符串，但是不够方便，如果给出name要在环境变量表中查找它对应的value，可以用getenv函数。

```
#include <stdlib.h>
char *getenv(const char *name);
```

getenv的返回值是指向value的指针，若未找到则为NULL。

修改环境变量

修改环境变量可以用以下函数：

```
#include <stdlib.h>

int setenv(const char *name, const char *value, int rewrite);
void unsetenv(const char *name);
```

setenv函数若成功则返回为0，若出错则返回非0。
setenv将环境变量name的值设置为value。

如果已存在环境变量name，那么:

若rewrite非0，则覆盖原来的定义；

若rewrite为0，则不覆盖原来的定义，也不返回错误。

unsetenv删除name的定义。即使name没有定义也不返回错误。

修改环境变量例子

```
/* env.c */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    printf("PATH=%s\n", getenv("PATH"));
    setenv("PATH", "hello", 1);
    printf("PATH=%s\n", getenv("PATH"));
    return 0;
}
```

进程控制

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ID(大于0的整数)：父进程
	-1：出错

fork出错的原因

- 系统中已经有了太多的进程
- 该实际用户I D的进程总数超过了系统限制
  - CHILD_MAX规定了每个实际用户I D在任一时刻可具有的最大进程数

fork的用法

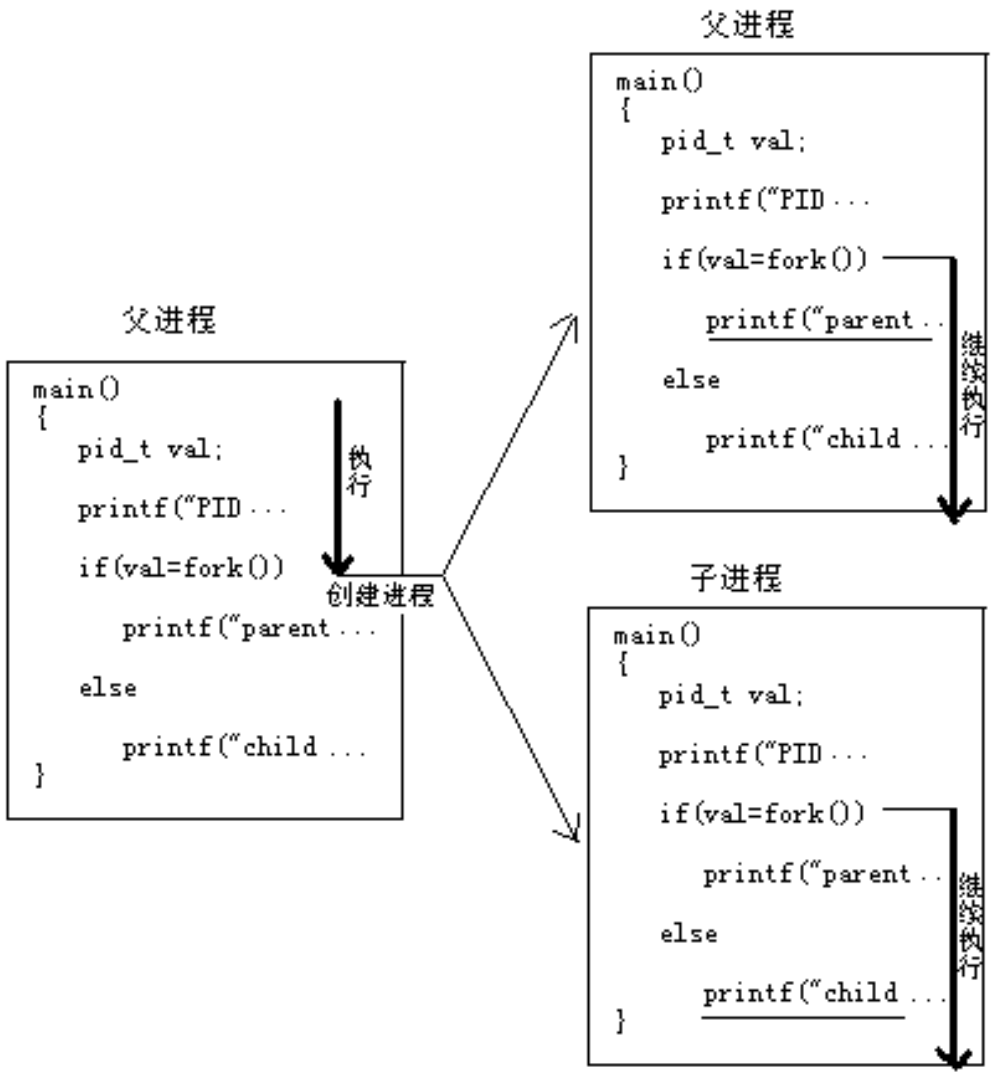
- 一个父进程希望复制自己，使父、子进程同时执行不同的代码段。
  - 网络服务器的经典代码

- 一个进程要执行一个不同的程序。
  - 这对shell是常见的情况。在这种情况下，子进程在从fork返回后立即调用exec函数。

fork.c

```
int main(void)
{
        pid_t pid;
        char *message;
        int n;
        pid = fork();
        if (pid < 0) {
                perror("fork failed");
                exit(1);
        }
        if (pid == 0) {
                message = "This is the child\n";
                n = 6;
        } else {
                message = "This is the parent\n";
                n = 3;
        }
        for(; n > 0; n--) {
                printf(message);
                sleep(1);
        }
        return 0;
}
```

fork运行分析



fork运行分析

Parent

```
int main()
{
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 6;
    } else {
        message = "This is the parent\n";
        n = 3;
    }
    for(; n > 0; n--) {
        printf(message);
        sleep(1);
    }
    return 0;
}
```

Child

```
int main()
{
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 6;
    } else {
        message = "This is the parent\n";
        n = 3;
    }
    for(; n > 0; n--) {
        printf(message);
        sleep(1);
    }
    return 0;
}
```

代码分析-1

- 父进程初始化。
- 父进程调用fork，这是一个系统调用，因此进入内核。
- 内核根据父进程复制出一个子进程，父进程和子进程的PCB信息相同，用户态代码和数据也相同。因此，子进程现在的状态看起来和父进程一样，做完了初始化，刚调用了fork进入内核，还没有从内核返回。
- 现在有两个一模一样的进程看起来都调用了fork进入内核等待从内核返回（实际上fork只调用了一次），此外系统中还有很多别的进程也等待从内核返回。是父进程先返回还是子进程先返回，还是这两个进程都等待，先去调度执行别的进程，这都不一定，取决于内核的调度算法。
- 如果某个时刻父进程被调度执行了，从内核返回后就从fork函数返回，保存在变量pid中的返回值是子进程的id，是一个大于0的整数，因此执下面的else分支，然后执行for循环，打印"This is the parent\n"三次之后终止。

代码分析-2

- 如果某个时刻子进程被调度执行了，从内核返回后就从fork函数返回，保存在变量pid中的返回值是0，因此执行下面的if (pid == 0)分支，然后执行for循环，打印"This is the child\n"六次之后终止。fork调用把父进程的数据复制一份给子进程，但此后二者互不影响，在这个例子中，fork调用之后父进程和子进程的变量message和n被赋予不同的值，互不影响。
- 父进程每打印一条消息就睡眠1秒，这时内核调度别的进程执行，在1秒这么长的间隙里（对于计算机来说1秒很长了）子进程很有可能被调度到。同样地，子进程每打印一条消息就睡眠1秒，在这1秒期间父进程也很有可能被调度到。所以程序运行的结果基本上是父子进程交替打印，但这也不是一定的，取决于系统中其它进程的运行情况和内核的调度算法，如果系统中其它进程非常繁忙则有可能观察到不同的结果。另外，也可以把sleep(1);去掉看程序的运行结果如何。
- 这个程序是在Shell下运行的，因此Shell进程是父进程的父进程。父进程运行时Shell进程处于等待状态，当父进程终止时Shell进程认为命令执行结束了，于是打印Shell提示符，而事实上子进程这时还没结束，所以子进程的消息打印到了Shell提示符后面。最后光标停在This is the child的下一行，这时用户仍然可以敲命令，即使命令不是紧跟在提示符后面，Shell也能正确读取。

fork小结

- fork函数的特点概括起来就是“调用一次，返回两次”，在父进程中调用一次，在父进程和子进程中各返回一次。从上图可以看出，一开始是一个控制流程，调用fork之后发生了分叉，变成两个控制流程，这也就是“fork”（分叉）这个名字的由来了。子进程中fork的返回值是0，而父进程中fork的返回值则是子进程的id（从根本上说fork是从内核返回的，内核自有办法让父进程和子进程返回不同的值），这样当fork函数返回后，程序员可以根据返回值的不同让父进程和子进程执行不同的代码。
- fork的返回值这样规定是有道理的。fork在子进程中返回0，子进程仍可以调用getpid函数得到自己的进程id，也可以调用getppid函数得到父进程的id。在父进程中用getpid可以得到自己的进程id，然而要想得到子进程的id，只有将fork的返回值记录下来，别无它法。
- fork的另一个特性是所有由父进程打开的描述符都被复制到子进程中。父、子进程中相同编号的文件描述符在内核中指向同一个file结构体，也就是说，file结构体的引用计数要增加。

fork的执行流程

- 检查可用的内核资源
- 取一个空闲的进程表项和唯一的PID号
- 检查用户没有过多的运行进程
- 将子进程的状态设为“创建”状态
- 将父进程的进程表项中的数据拷贝到子进程表项中
- 当前目录的索引节点和改变的根目录的引用计数加1
- 文件表中的打开文件的引用数加1

fork的执行流程(续)

- 在内存中作父进程上下文的拷贝
- 在子进程的系统级上下文中压入虚拟系统级上下文层;

- if（正在执行的进程是父进程）
  - 将子进程状态设为“就绪”状态
  - return 子进程的PID
- else
  - 初始化计时域
  - return 0

exec函数族

- fork函数用于创建一个子进程，该子进程几乎拷贝了父进程的全部内容。

- **exec函数族**就**提供了一个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，其他全部被新的进程替换了。

- 当进程调用一种exec函数时，该进程程序替换，从新程序的启动例程开始执行。调用exec并不创建的用户空间。代码和数据完全被新进程更新，所以调用exec前后该进程的id并未改变

- 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

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

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**
  - 表中的前四个函数的查找方式都是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，系统就会自动从环境变量“$PATH”所指出的路径中进行查找。

- **参数表传递方式**
  - 两种方式：逐个列举、将所有参数整体构造指针数组传递
  - 以函数名的第五位字母来区分的，字母为“l”(list)的表示逐个列举的方式，其语法为char *arg；字母为“v”(vector)的表示将所有参数整体构造指针数组传递，其语法为*const argv[]

- **环境变量的使用**
  - exec函数族可以默认系统的环境变量，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[]中指定当前进程所使用的环境变量

43

exec函数解释

- 这些函数如果调用成功则加载新的程序从启动代码开始执行，不再返回，如果调用出错则返回-1，所以exec函数只有出错的返回值而没有成功的返回值。
- 这些函数原型看起来很容易混，但只要掌握了规律就很好记。不带字母p（表示path）的exec函数第一个参数必须是程序的相对路径或绝对路径，例如"/bin/ls"或"./a.out"，而不能是"ls"或"a.out"。
- 对于带字母p的函数：

  如果参数中包含/，则将其视为路径名。

  否则视为不带路径的程序名，在PATH环境变量的目录列表中搜索这个程序。
- 带有字母l（表示list）的exec函数要求将新程序的每个命令行参数都当作一个参数传给它，命令行参数的个数是可变的，因此函数原型中有...，...中的最后一个可变参数应该是NULL，起sentinel的作用。
- 对于带有字母v（表示vector）的函数，则应该先构造一个指向各参数的指针数组，然后将该数组的首地址当作参数传给它，数组中的最后一个指针也应该是NULL，就像main函数的argv参数或者环境变量表一样。
- 对于以e（表示environment）结尾的exec函数，可以把一份新的环境变量表传给它，其他exec函数仍使用当前的环境变量表执行新程序。

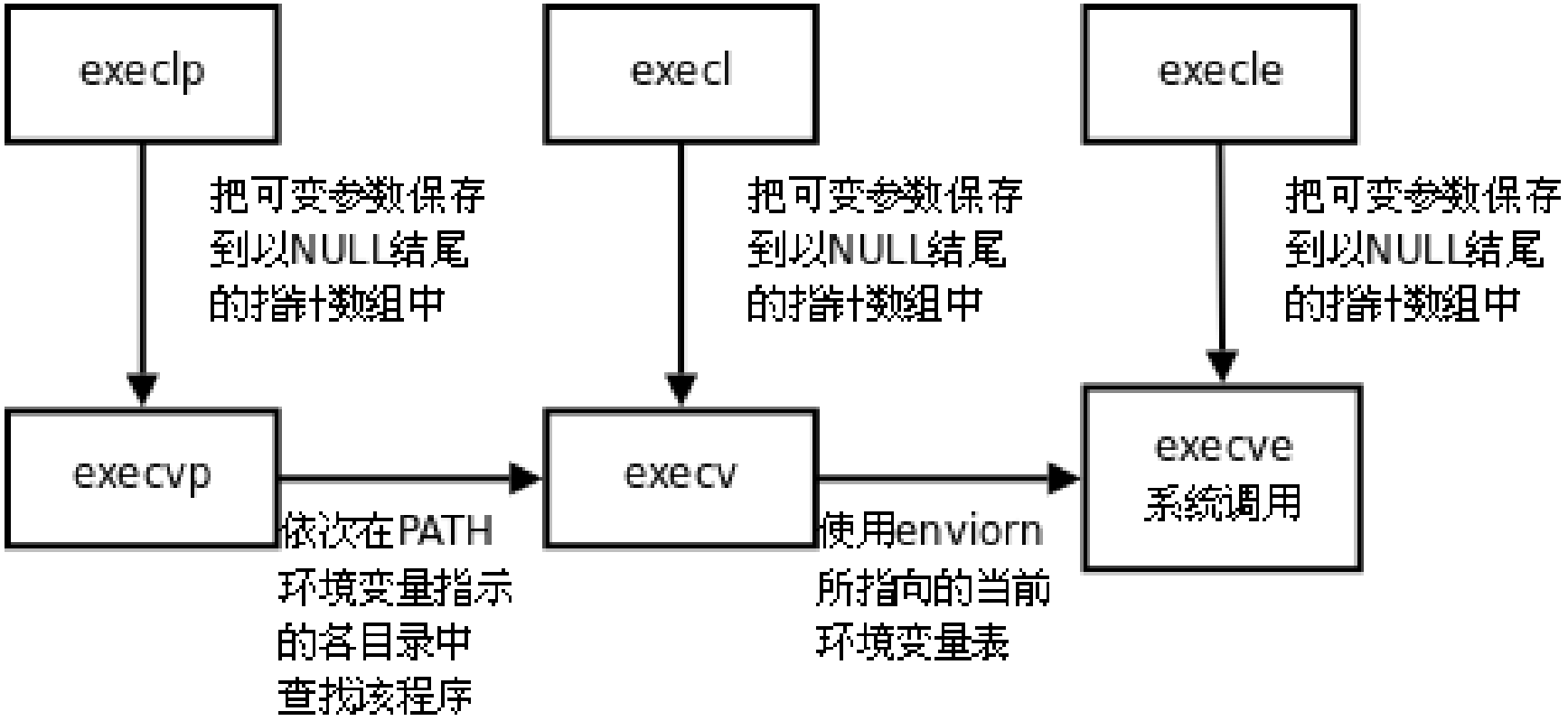
exec调用举例

- char *const ps_argv[] ={"ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL};
- char *const ps_envp[] ={"PATH=/bin:/usr/bin", "TERM=console", NULL};

- execl("/bin/ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);
- execv("/bin/ps", ps_argv);
- execle("/bin/ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL, ps_envp);
- execve("/bin/ps", ps_argv, ps_envp);
- execlp("ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);
- execvp("ps", ps_argv);

- 只有execve是真正的系统调用，其它五个函数最终都调用execve

调用分析



完整调用exec例子

```
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
    execlp("ps", "ps", "-A", NULL);
    perror("exec ps");

    exit(1);
}
```

于exec函数只有错误返回值，只要返回了一定是出错了，所以不需要判断它的返回值，直接在后面调用perror即可。注意在调用execlp时传了两个"ps"参数，第一个"ps"是程序名，execlp函数要在PATH环境变量中找到这个程序并执行它，而第二个"ps"是第一个命令行参数，execlp函数并不关心它的值，只是简单地把它传给ps程序，ps程序可以通过main函数的argv[0]取到这个参数。

system

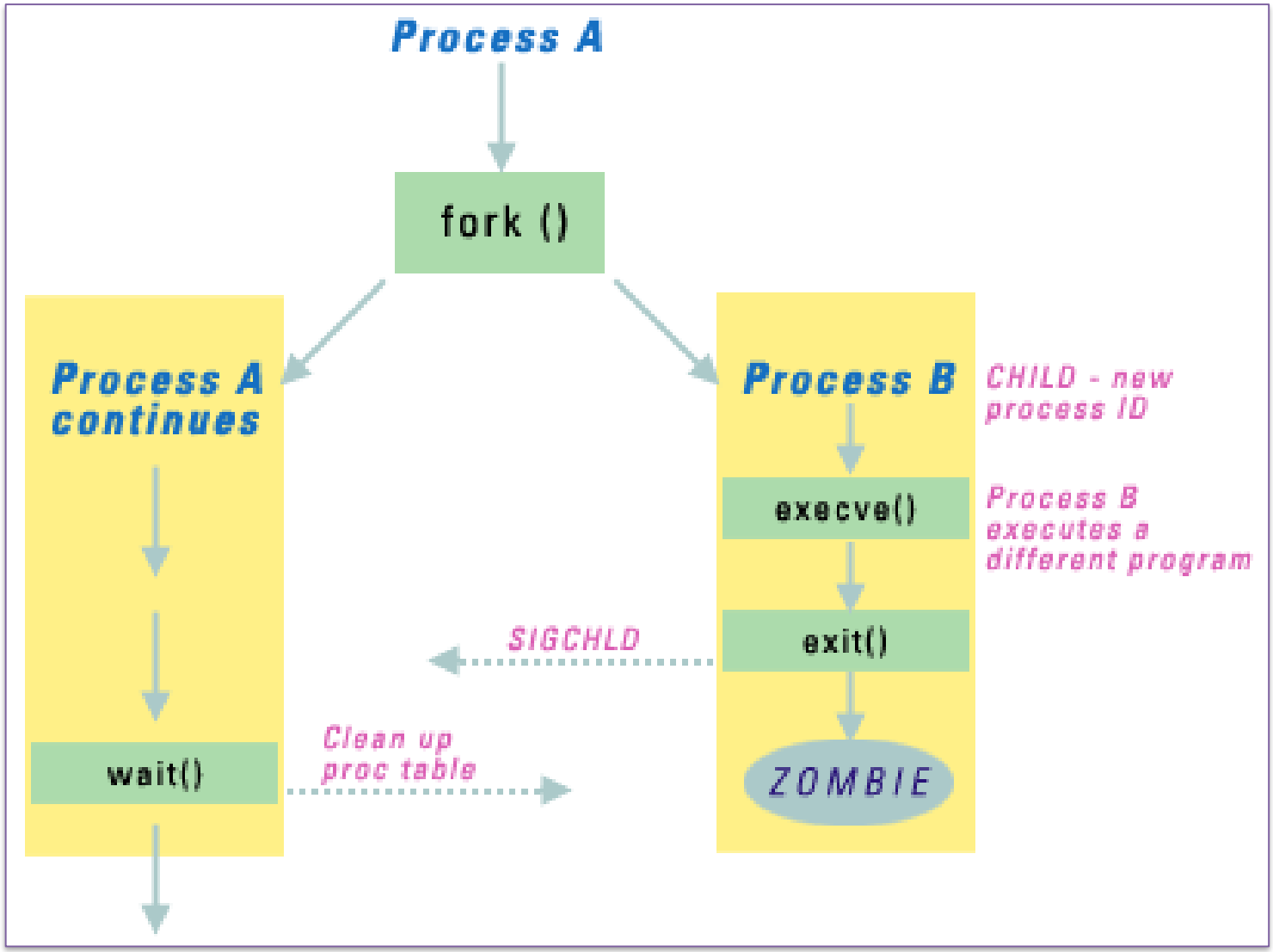
```
#include <stdlib.h>
int system (char *string);

#include<unistd.h>
#include <stdio.h>
int main()
{
        char cmd [] = { "/bin/ls –l" }
        system(cmd);
        return 0;
}
```

写时拷贝的概念

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。为了加快fork的执行速度，有些Unix系统设计者创建了vfork。vfork也创建新进程，但它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**

- 现在，很多嵌入式Linux系统的fork函数调用都采用vfork函数的实现方式，实际上uClinux所有的多进程管理都通过vfork来实现。

进程创建过程



进程终止

- 一个进程在终止时会关闭所有文件描述符，释放在用户空间分配的内存，但它的PCB还保留着，内核在其中保存了一些信息：如果是正常终止则保存着退出状态，如果是异常终止则保存着导致该进程终止的信号是哪个。这个进程的父进程可以调用wait或waitpid获取这些信息，然后彻底清除掉这个进程。我们知道一个进程的退出状态可以在Shell中用特殊变量$?查看，因为Shell是它的父进程，当它终止时Shell调用wait或waitpid得到它的退出状态同时彻底清除掉这个进程。

- 如果一个进程已经终止，但是它的父进程尚未调用wait或waitpid对它进行清理，这时的进程状态称为僵尸（Zombie）进程。任何进程在刚终止时都是僵尸进程，正常情况下，僵尸进程都立刻被父进程清理了，为了观察到僵尸进程，我们自己写一个不正常的程序，父进程fork出子进程，子进程终止，而父进程既不终止也不调用wait清理子进程。

进程消亡的原因

1. 寿终
2. 自杀
3. 他杀
4. 处决

进程的终止

- 正常终止：
  - 从main返回
  - 调用exit
  - 调用_exit

- 异常终止
  - 调用abort
  - 由一个信号终止

exit和_exit函数

exit和_exit函数用于正常终止一个程序

```
#include <stdlib.h>
void exit(int status) ;
```

exit则先执行一些清除处理.然后进入内核
清除操作包括调用执行各终止处理程序，关闭所有标准I/O流

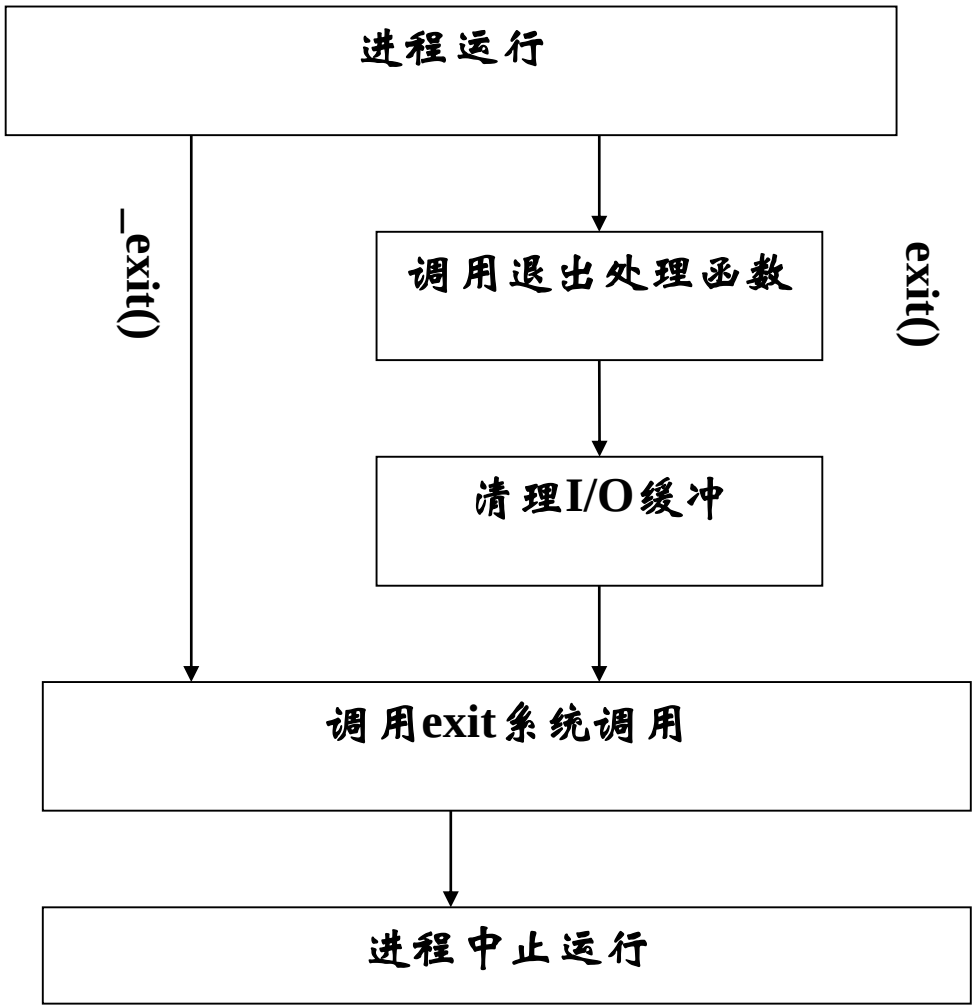
```
#include <unistd.h>
void _exit (int status) ;
```

_exit立即进入内核

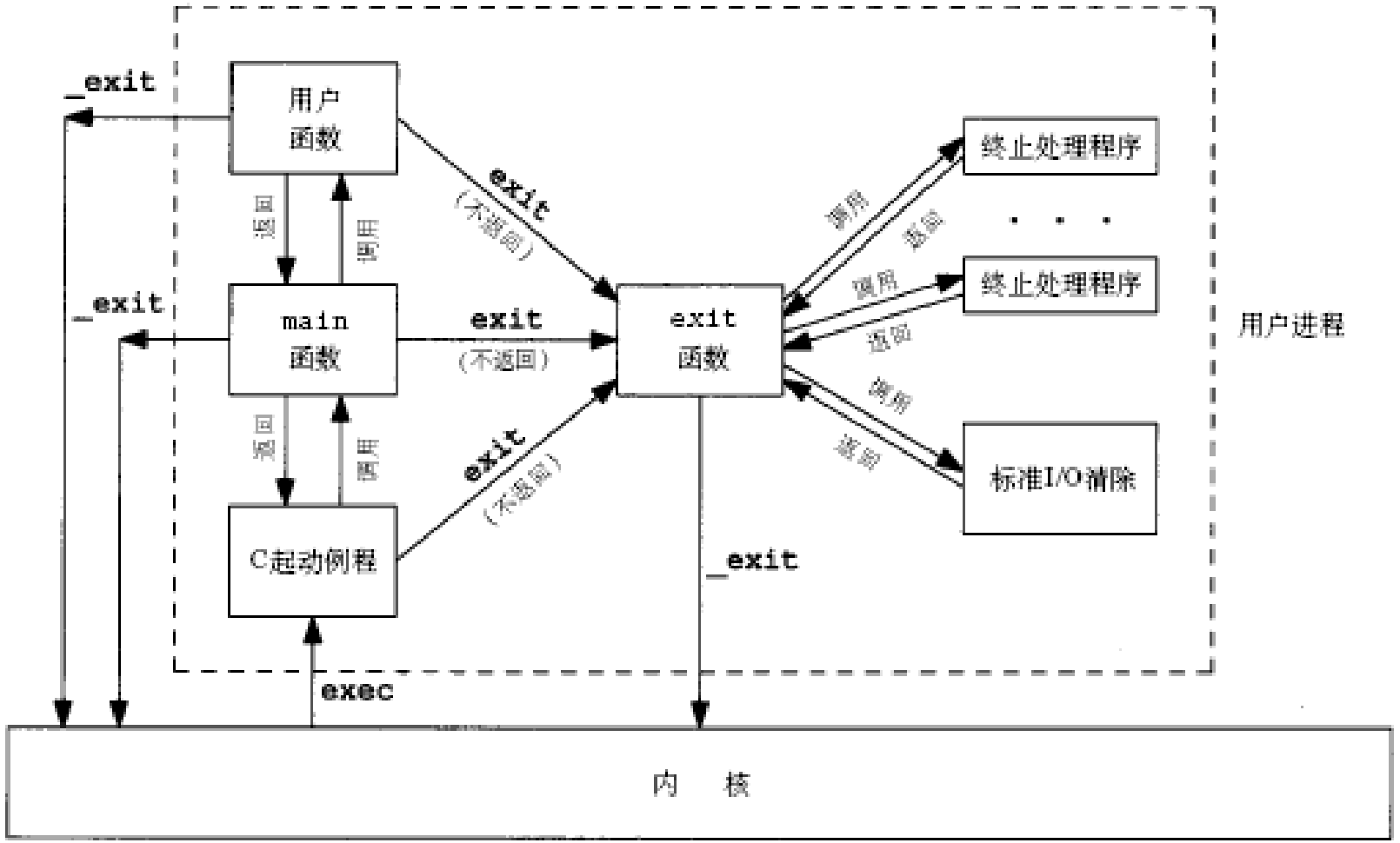
exit和_exit

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。一般来说，0表示没有意外的正常结束；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，从而针对不同的情况进行不同的处理。

exit和_exit



C程序是如何被启动终止的



_exit和exit的区别

- _exit()函数的作用最为简单：**直接使进程停止运行**，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()函数则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 举例

```
/* exit_test1.c */
#include<stdlib.h>
int main()
{
    printf("this process will exit!");
    exit(0);
    printf("never be displayed!");
}
```

在执行到exit(0)时，进程就已经终止了。

exit和_exit

```
int main()
{
    printf("Using exit...\n");
    printf("This is the content in buffer");
    exit(0);
}
[root@(none) 1]# ./exit_test2
Using exit...
This is the content in buffer
[root@(none) 1]#
```

```
int main()
{
    printf("Using _exit...\n");
    printf("This is the content in buffer");
    _exit(0);
}
[root@(none) 1]# ./_exit
Using _exit...
[root@(none) 1]#
```

atexit函数

- 按照ANSI C的规定，一个进程可以登记多至3 2个函数，这些函数将由exit自动调用。

```
#include <stdlib.h>
int atexit(void (* func) ( void ) ) ;
```

atexit的参数是一个函数地址，当调用此函数时无需向它传送任何参数，也不期望它返回一个值。

exit以登记这些函数的相反顺序调用它们。同一函数如若登记多次，则也被调用多次。

- 例子:atexit.c

僵尸进程与孤儿进程

```
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
        pid_t pid=fork();
        if(pid<0) {
                perror("fork");
                exit(1);
        }
        if(pid>0) { /* parent */
                while(1);
        }
        /* child */
        return 0;
}
```

```
./a.out &
$ ps u
```

如果一个父进程终止，而它的子进程还存在（这些子进程或者仍在运行，或者已经是僵尸进程了），则这些子进程的父进程改为init进程。

init是系统中的一个特殊进程，通常程序文件是/sbin/init，进程id是1，在系统启动时负责启动各种系统服务，之后就负责清理子进程，只要有子进程终止，init就会调用wait函数清理它。

僵尸进程是不能用kill命令清除掉的，因为kill命令只是用来终止进程的，而僵尸进程已经终止了。

wait和waitpid

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

若调用成功则返回清理掉的子进程id，若调用出错则返回-1。

- 父进程调用wait或waitpid时可能会：

1. 阻塞（如果它的所有子进程都还在运行）。
2. 带子进程的终止信息立即返回（如果一个子进程已终止，正等待父进程读取其终止信息）。
3. 出错立即返回（如果它没有任何子进程）。

wait和waitpid

- wait函数是用于**使父进程阻塞**，直到一个**子进程结束**或者是该进程接收到了一个指定的信号为止。如果该父进程没有子进程或者他的子进程已经结束，则wait就会立即返回。

- waitpid的作用和wait一样，但它并不一定等待第一个终止的子进程，它还有若干选项，如可提供一个非阻塞版本的wait功能，还能支持作业控制。实际上，**wait函数只是waitpid函数的一个特例**，Linux内部在实现wait函数时直接调用的就是waitpid函数 。

wait和waitpid

所需头文件	#include <sys/types.h> #include <sys/wait.h>
函数原型	pid_t wait(int *status)
函数传入值	这里的status是一个整型指针，是该子进程退出时的状态。 • status若为空，则代表任意状态结束的子进程 • status若不为空，则代表指定状态结束的子进程 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

wait和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)	
函数传入值	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：子进程的进程号	
	使用选项WNOHANG且没有子进程退出：0	
	调用出错：-1	

wait和waitpid的区别

- 这两个函数的区别是：

1. 如果父进程的所有子进程都还在运行，调用wait将使父进程阻塞，而调用waitpid时,如果在options参数中指定**WNOHANG**可以使父进程不阻塞而立即返回0。

2. wait等待第一个终止的子进程，而waitpid可以通过pid参数指定等待哪一个子进程。

- 可见，调用wait和waitpid不仅可以获得子进程的终止信息，还可以使父进程阻塞等待子进程终止，起到进程间同步的作用。如果参数status不是空指针，则子进程的终止信息通过这个参数传出，如果只是为了同步而不关心子进程的终止信息，可以将status参数指定为NULL。

waitpid例子

见waitpid.c

- 子进程的终止信息在一个int中包含了多个字段，用宏定义可以取出其中的每个字段：
- 如果子进程是正常终止的，WIFEXITED取出的字段值非零，WEXITSTATUS取出的字段值就是子进程的退出状态；
- 如果子进程是收到信号而异常终止的，WIFSIGNALED取出的字段值非零，WTERMSIG取出的字段值就是信号的编号。

和waitpid有关的宏

WIFEXITED（status）如果子进程正常结束则为非0 值。
WEXITSTATUS（status）取得子进程exit（）返回的结束代码，一般会先用WIFEXITED 来判断是否正常结束才能使用此宏。

WIFSIGNALED（status）如果子进程是因为信号而结束则此宏值为真
WTERMSIG（status）取得子进程因信号而中止的信号代码，一般会先用WIFSIGNALED 来判断后才使用此宏。

WIFSTOPPED（status）如果子进程处于暂停执行情况则此宏值为真。一般只有使用WUNTRACED 时才会有此情况。
WSTOPSIG（status） 取得引发子进程暂停的信号代码，一般会先用WIFSTOPPED 来判断后才使用此宏。

多进程程序开发

```
int main(void)
{
    pid_t pid;
    pid = fork();   //we should check the error
    if(pid == -1)
    {perror("fork error");exit;}
    else if(pid == 0)
    {
        --WAIT_PARENT  //sleep(1);
        ……                    //the child task
        --TELL_PARENT
        exit(0);
    }
     ……                    //the parent task
    --TELL_CHILD
    --WAIT_CHILD    // sleep(1);
    exit(0);
}
```

多进程程序开发

```
int main(void)
{
    pid_t pid1,pid2;
    if((pid1=fork())==0)
    {
        --WAIT_CHILD2  //   IPC
        ……                    //the child1 task
         --TELL_CHILD2
        exit(0);
    }else if((pid2=fork())==0)
    {       ……              //the child2 task
        --TELL_CHILD1     //
        --WAIT_CHILD1
        exit(0);
    }
    ……                    //the parent task
    --WAIT_ALL_CHILD    // waitpid
    exit(0);
}
```

Linux守护进程

- **守护进程**，也就是通常所说的**Daemon**进程，是Linux中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并且周期性的执行某种任务或等待处理某些发生的事件

- 守护进程常常在系统引导装入时启动，在系统关闭时终止

- Linux系统有很多守护进程，大多数服务都是用守护进程实现的

Linux守护进程

- 在Linux中，每一个系统与用户进行交流的界面称为终端，每一个从此终端开始运行的进程都会依附于这个终端，这个终端就称为这些进程的控制终端，当控制终端被关闭时，相应的进程都会被自动关闭。

- 守护进程能够突破这种限制，它从被执行开始运转，直到整个系统关闭才会退出。如果想让某个进程不因为用户或终端或其他的变化而受到影响，就必须把这个进程变成一个守护进程。

作业

- 分析shell的实现原理，实现一个最简单的shell，要求实现一个内部命令：cd，一个外部命令，如ls（不要求实现其代码，直接调用/bin/ls即可）。

专业铸就品质 梦想成就未来

The Specialty Casts Quality,the Dream Gains Future.