重庆理工大学/电气学院
CHONGQING UNIVERSITY OF TECHNOLOGY

嵌入式Linux系统开发教程

—基于ARM处理器通用平台（arm9-arm11-contexA系列）

丛超

2025年5月

1 进程控制概述

2 进程编程

01

进程控制概述

进程管理是操作系统中最为关键部分，它的设计与实现直接影响到系统的整体性能。由于进程在运行过程中，要使用许多计算机资源，如CPU，内存，文件等，通过进程管理，合理地分配系统资源，从而提高CPU的利用率。

为了协调多个进程对这些共享资源访问，操作系统要跟踪所有的进程的活动及他们对系统资源的使用情况，实施对进程和资源的动态管理。

6.1.1 进程定义

宏观上看：进程是计算机中的程序关于某数据集合的一次活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。

同时进程也是处于执行期的程序，但进程并不仅仅局限于一段可执行的代码段，通常还包括其他资源，如打开文件、挂起信号、内核内部数据、处理器状态、地址空间以及一个或者多个执行线程、用来存放全局变量的数据段。

进程是一个程序的一次执行过程，同时也是资源分配的最小单元。它和程序有本质区别：

口 程序是静态的

- 程序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

口 进程是动态的

- 它是程序执行的过程
- 包括动态创建、调度和消亡的整个过程。

总而言之进程是程序执行和管理的最小单元，因此，对系统而言，当用户在系统中输入命令执行一个程序时，他将起动一个进程。

程序本身不是进程。实际上完全可能存在两个不同的进程执行的是同一个程序，并且两个或两个以上并存的进程还可以共享许多如打开文件、地址空间之类的资源。

在Linux系统中，进程分为用户进程、守护进程、批处理进程三类。

6.1.2进程控制块

进程是Linux系统的基本调度和管理资源的单位，内核把进程存放在任务队列的双向循环链表中。链表中的每一项都是类型为task_struct结构.

在操作系统内，对每一个进程进行管理的数据结构称为进程控制模块(PCB)，主要描述当前进程状态和进程正在使用的资源。如下：

```
typedef struct task_struct{
        int pid;                    //进程ID 用来标识进程
        unsigned long state;    //进程状态 描述当前进程运行状态
        unsigned long count;    //进程时间片数
        unsigned long timer;    //进程休眠时间
        unsigned long priority;   //进程默认优先级 进程时间片数和优先级都属于进程调度信息
        unsigned long content[20]; //进程执行现场保存区，包含当前进程使用的操作寄存器、状态寄存器和栈指针寄存器等
}PCB;
```

tast_struct相对较大，在32位机上大约有1.7KB。进程描述符中包含的数据能完整的描述一个正在执行的程序，包括打开文件、进程地址空间、挂起的信号、进程状态以及其他更多信息。进程描述符及队列任务如下图所示。

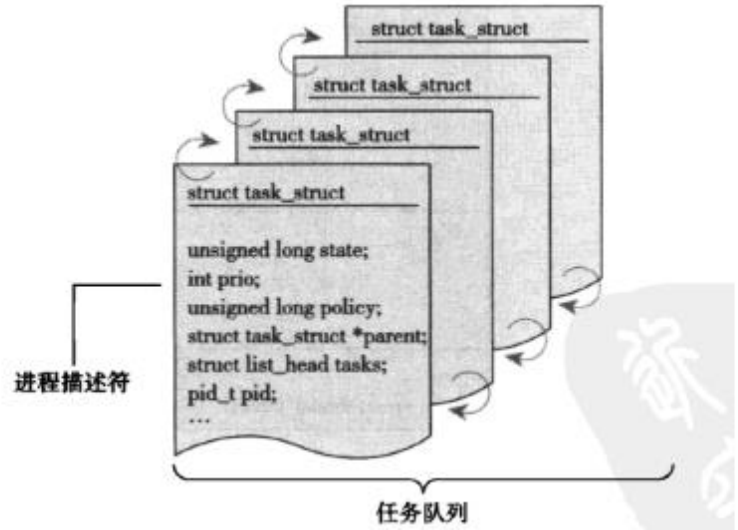


图6-1 进程描述符及任务队列

6.1.3分配进程描述符

Linux通过slab分配器分配task_struct结构，通过预先分配和重复使用task_struct，可以避免动态分配和释放所带来的资源消耗。

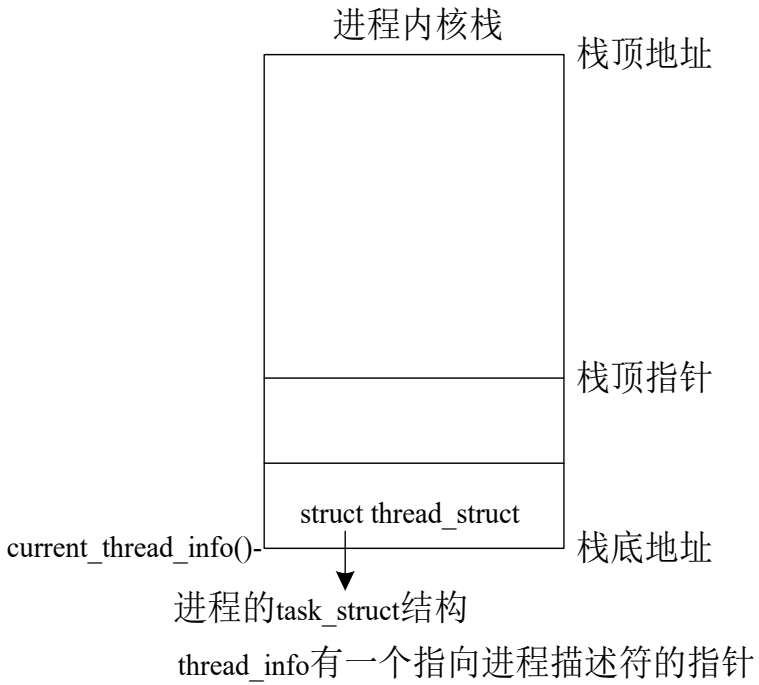


图6-2 进程描述符和内核栈

6.1.4进程的创建

内核通过一个唯一的进程标识值或PID来标识每个进程。一个进程要被执行，首先被创建，进程需要一定的系统资源，如CPU时间片、内存空间、操作文件、硬件设备等。进程创建包括以下操作：

（1）初始化当前进程PCB，分配有效进程ID，设置进程优先级和CPU时间片。

（2）为进程分配内存空间。

（3）加载任务到内存空间，将进程代码复制到内核空间。

（4）设置进程执行状态为就绪状态，将进程PCB放入到进程队列中。

6.1.5进程状态

系统中的每个进程都处于进程状态中的一种。

（1）TAST_RUNNING（运行）:系统中同一时刻可能有多个进程处于可执行状态，这些进程被放入可执行队列中，进程调度器的任务就是从可执行队列中分别选择一个进程在CPU上运行。

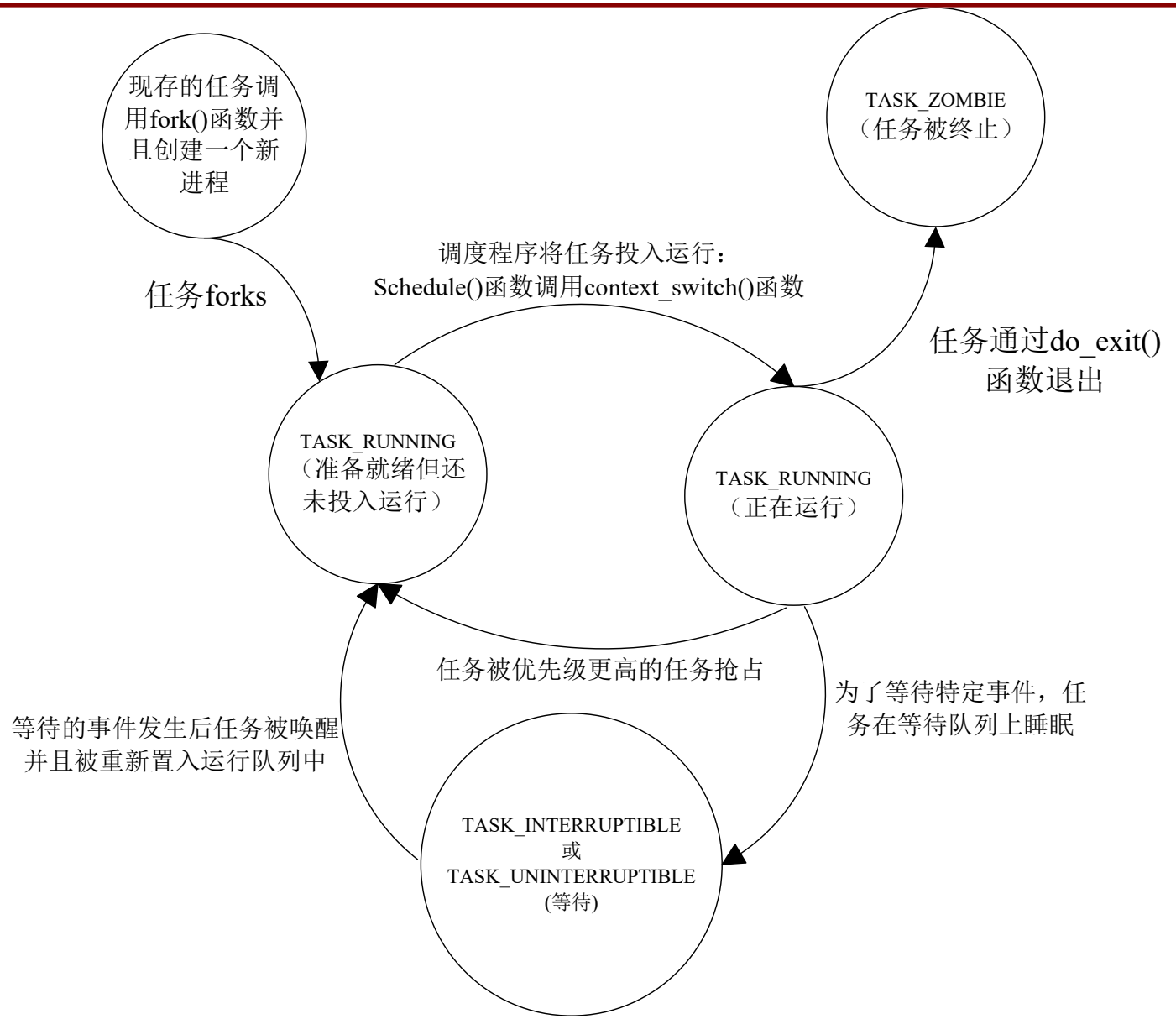
（2） TAST_INTERRUPTIBLE（可中断）：有些进程因为等待某事件的发生而被挂起，这些进程被放入等待队列中，当事件发生时，对应的等待队列中的一个或多个进程将被唤醒。进程中大多数处于可中断睡眠状态。

（3） TAST_UNINTERRUPTIBLE（不可中断）：有些进程处于睡眠状态，但是此刻进程不可中断，所以不响应异步信号。在进程对某些硬件进行操作时，需要使用不可中断睡眠状态对进程进行保护，以避免进程与设备交互的过程被打断。这种状态下的不可中断睡眠状态总是非常短的。

（4） TAST_ZOMBIE（僵死）：进程在退出的过程中，处于 TAST_DEAD状态，进程占有的所有资源全部被收回，只剩tast_struct这个空壳。之所以保留tast_struct，是因为tast_struct里面保留了进程的退出码及相关统计信息。

（5）TAST_STOPPED（停止）：该状态下，进程进程暂停，等待其他进程对其进行操作。

（6）TAST_DEAD（销毁状态）：进程在退出过程中，如果该进程是多线程程序中被deatch过的过程，或者父进程通过设置SIGCHLD信号的句柄为SIG_IGN。此时，进成将被处于退出状态，且退出过程不会僵死，该进程彻底释放。


现存的任务调用fork()函数并且创建一个新进程
任务forks
TASK_RUNNING
（准备就绪但还未投入运行）
调度程序将任务投入运行：
Schedule()函数调用context_switch()函数
TASK_RUNNING
（正在运行）
TASK_ZOMBIE
（任务被终止）
任务通过do_exit()函数退出
任务被优先级更高的任务抢占
为了等待特定事件，任务在等待队列上睡眠
等待的事件发生后任务被唤醒并且被重新置入运行队列中
TASK_INTERRUPTIBLE
或
TASK_UNINTERRUPTIBLE
(等待)

6.1.6进程调度

在操作系统中，进程需要按照一定的策略被调度执行，让每一个进程都能够取得CPU的执行权，来增加系统实时性和交互性。

在PCB中有一个成员——count时间片数，内核定期检查进程的运行状态，定时器定期产生中断信号，在定时器中断处理程序中对当前执行进程的时间片count进行递减，当时间片用尽时，进程被挂起，进行新进程调度，将CPU执行权交给新选进程。

时间片即CPU分配给各个程序的时间，每个进程被分配一个时间段，称为它的时间片，即该进程允许运行的时间，使各个程序从表面上看是同时进行的。

如果在时间片结束时进程还在运行，则CPU将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则CPU当即进行切换，而不会造成CPU资源浪费。

多任务操作系统是能并发地执行多个进程的操作系统，多任务操作系统可以分为抢占式多任务操作系统和非抢占式多任务操作系统，Linux系统提供了抢占式的多任务模式。

在该模式下，由调度程序来决定什么时候停止一个进程的运行以便其他进程能够得到执行的机会。

调度进程主要功能是对进程完成中断操作、改变优先级、查看进程状态等。

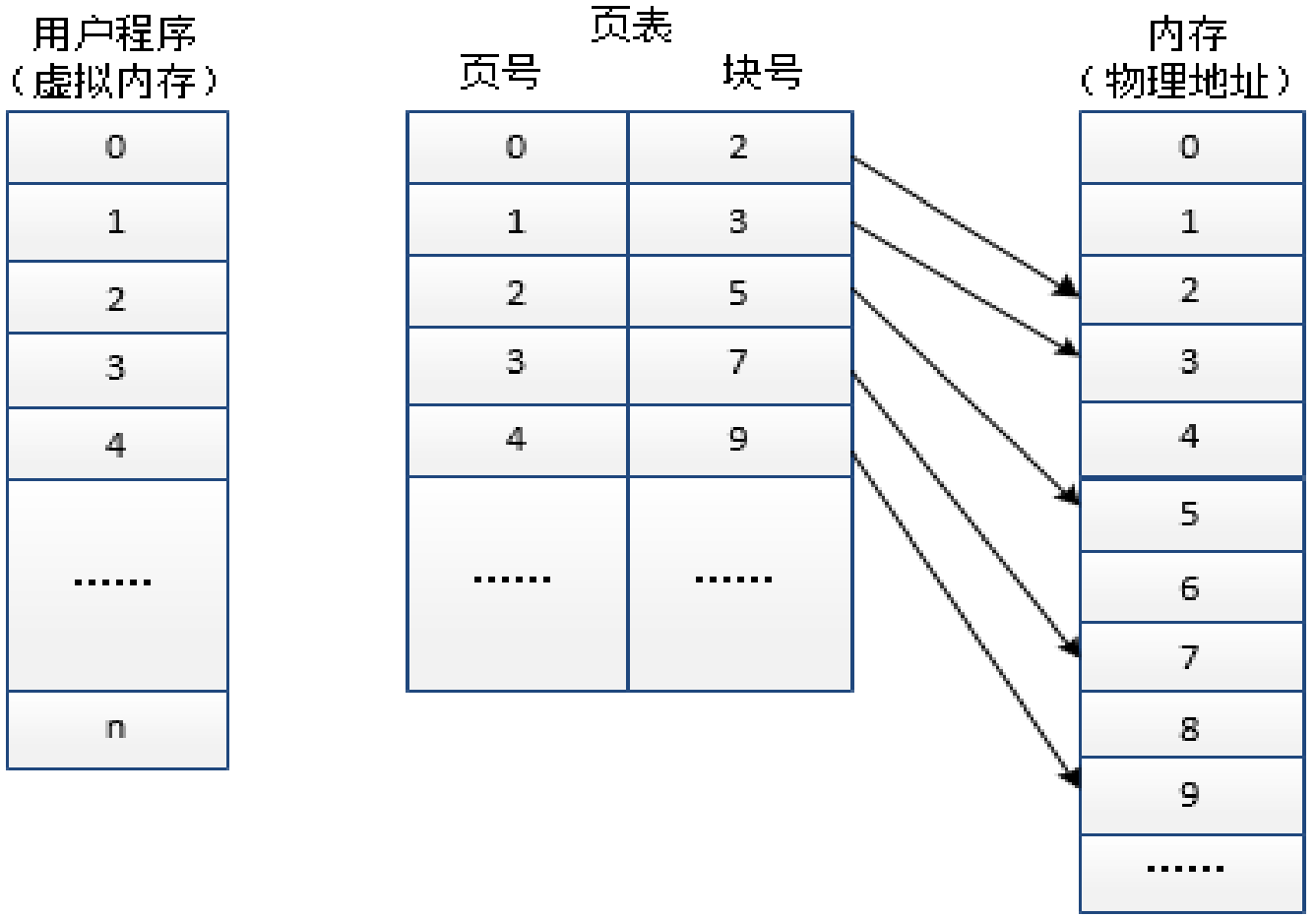
进程调度选出新进程后，CPU就要进行上下文切换，将新进程切换成当前执行进程，同时还要保存当前执行进程的执行现场，为下一次调度执行做准备。

6.1.7虚拟内存

虚拟内存是将系统硬盘空间和系统实际内存联合在一起供进程使用，给进程提供了一个比内存大得多的虚拟空间。

当程序运行时，系统会为程序开辟一段4G大小的虚拟内存，其中0~3G的虚拟地址用于存放程序的代码段、数据段等信息，当虚拟内存与物理内存相映射时，各个虚拟内存中地址相同的数据，会被MMU（Memory Management Unit，内存管理单元）映射到内存中不同的物理地址，为保证内核能根据进程中的虚拟地址在物理磁盘中找到进程中所需的数据，PCB应存储虚拟地址与物理地址的对应关系。

每个进程的pcb中都有一个指向页表的指针，进程、页表与内存之间的映射关系如图所示。



6.1.8文件锁

每个进程都可以打开文件，为防止多个进程同时操作一个文件，则由文件锁进行控制。文件锁的作用是阻止多个进程同时操作同一个文件。

(1)在进程中，关闭一个描述符时，则该进程通过这一描述符引用的文件上的任何一把锁都被释放。

(2)当一个进程终止时，它所建立的锁全部释放。

(3)由fork产生的子进程不继承父进程所设置的锁，子进程需要调用fcnt才能获得它自己的锁。

(4)当调用exec后，新程序可以继承原执行程序的文件锁。

02

进程编程

6.2.1启动进程

用于执行shell命令。

表头文件

#include <stdlib.h>

定义函数

int system(const char *command);

参数：command为linux指令。

返回值：返回-1表示错误；

返回0表示调用成功但没有出现子进程；

返回大于0 表示成功退出的子进程id。

Example6.2.1-1：system函数使用

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    pid_t result;
    printf("This is a system demo!\n\n");
    result = system("ls -l");
    return result;
}
```

2. exec系列函数调用

exec家族一共有六个函数，分别是execl()、execlp()、execle()、execv()、execvp()、execve()，它们包含在系统库unistd.h中，函数声明分别如下：

#include <unistd.h>

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

参数：

path和file表示要执行程序的包含路径的文件名；

arg为参数序列，中间用逗号分隔；

argv为参数列表；

envp为环境变量列表；

返回值：exec函数族的函数执行成功后不会返回，只有调用失败了，它们才会返回一个-1.

Example6.2.1-2:用execl函数调用指令

```
1. #include <unistd.h>
2. #include<stdio.h>
3. #include <errno.h>
4. #include<string.h>
5. int main(int argc,char **argv){
6. if (execl("/bin/ls","ls","-l",NULL) <0){
7. printf("err=%s\n",strerror(errno));
8. }
9. return 0;
10.}
```

Example6.2.1-3:用execvp函数调用指令

```
1.  #include <unistd.h>
2.  #include<stdio.h>
3.  #include <errno.h>
4.  #include<string.h>
5.  int main(int argc,char **argv){
6.  char *argls[]={ //同argv
7.  "ls",
8.  "-l",
9.  NULL
10. };
11. if (execvp("/bin/ls",argls) <0){
12. printf("err=%s\n",strerror(errno));
13. }
14. return 0;
15. }
```

3. fork分叉函数调用

表头文件

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

定义函数

（1）获取进程ID的函数如下：

```
pid_t getpid(void);  //获取当前进程ID
pid_t getppid(void); //获取父进程ID
```

（2）创建子进程的函数如下：

```
pid_t fork(void);
```

返回值：

失败返回-1。成功返回进程ID。

fork创建了一个新的进程，原进程称为父进程，新的进程为子进程。fork创建进程后，函数在子进程中返回0值，在父进程中返回子进程的PID。两个进程都有自己的数据段、BBS段、栈、堆等资源，父子进程间不共享这些存储空间。而代码段为父进程和子进程共享。父进程和子进程各自从fork函数后开始执行代码。

Example6.2.1-4:用fork创建子进程

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
int main(int argc,char **argv){
    char buf[256];
    pid_t pid=fork();
    if(pid==-1) return 0;
    else if (pid==0){
        strcpy(buf,"我是子进程");
        int i,a=5;
        for(i=0;i<5;i++) {
            printf("son ---- %d\n",i);
            sleep(1);
        }
    }
else {
        strcpy(buf,"我是父进程");
        int i,a=10;
        for(i=5;i<a;i++){
            printf("parent---- %d\n",i);
            sleep(3);
        }
    }
    printf("---- %s,my pid= %d\n",buf,getpid());
    return 0;
}
```

fork() 会创建一个新的子进程，返回两次：
父进程中返回子进程的 PID
子进程中返回 0
如果失败返回 -1

这是子进程执行的代码块。输出从 son ---- 0 到 son ---- 4，每秒一次，共持续 5 秒。buf 存储“我是子进程”。

这是父进程执行的代码块。输出从 parent ---- 5 到 parent ---- 9，每次间隔 3 秒，总共耗时 15 秒。buf 存储“我是父进程”。

另一个创建子进程的vfork函数如下：

表头文件

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

定义函数

```
pid_t vfork(void);
```

函数说明

vfork()会产生一个新的子进程，其子进程会复制父进程的数据与堆栈空间，并继承父进程的用户代码、组代码、环境变量、已打开的文件代码、工作目录和资源限制等。

返回值

如果vfork()成功则在父进程会返回新建立的子进程代码(PID)，而在新建立的子进程中则返回0。如果vfork 失败则直接返回-1，失败原因存于errno中。

vfork与fork主要区别：fork要拷贝父进程的数据段；而vfork则不需要完全拷贝父进程的数据段，子进程与父进程共享数据段。fork不对父子进程的执行次序进行任何限制；而在vfork调用中，子进程先运行，父进程挂起。

6.2.2 等待进程

在Linux中，当我们使用fork()函数启动一个子进程时，子进程就有了它自己的生命周期并将独立运行。

在某些时候，可能父进程希望知道一个子进程何时结束，或者想要知道子进程结束的状态，甚至是等待着子进程结束，那么我们可以通过在父进程中调用wait()或者waitpid()函数让父进程等待子进程的结束。

1.wait()函数

表头文件

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

函数声明

```
pid_t wait(int *status);
```

wait()函数在被调用的时候，系统将暂停父进程的执行，直到有信号来到或子进程结束，如果在调用wait()函数时子进程已经结束，则会立即返回子进程结束状态值。子进程的结束状态信息会由参数status返回，与此同时该函数会返子进程的PID，它通常是已经结束运行的子进程的PID。

wait()函数有几点需要注意的地方：

（1）wait()要与fork()配套出现，如果在使用fork()之前就调用wait()，wait()的返回值则为-1，正常情况下wait()的返回值为子进程的PID。

（2）参数status用来保存被收集进程退出时的一些状态，它是一个指向int类型的指针，但如果我们对这个子进程是如何死掉毫不在意，只想把这个僵尸进程消灭掉，（事实上绝大多数情况下，我们都会这样做），我们就可以设定这个参数为NULL。

2. waitpid()函数

函数声明

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

waitpid()函数的作用和wait()函数一样，但它并不一定要等待第一个终止的子进程，它还有其他选项，比如指定等待某个pid的子进程、提供一个非阻塞版本的wait()功能等。实际上wait()函数只是waitpid()函数的一个特例，在Linux内部实现 wait函数时直接调用的就是waitpid 函数。

函数声明

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

参数：

status，是进程的返回值，16位数，前8位是子进程返回的值，后8位用于系统占用的位。

pid用来指定要等待哪一个进程结束：

①pid>0时，只等待进程ID等于pid的子进程结束；

②pid=-1时，等待任何一个子进程退出，此时等同于wait的作用；

③pid=0时，等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，waitpid不会对它做任何理睬；

④pid<-1时，等待指定进程组中的任何子进程,这个进程组的ID等于pid的绝对值；

options 用于指定等待方式：

①WNOHANG表示不阻塞，即使没有子进程退出，它也会立即返回；

②WUNTRACED 表示如果子进程进入暂停，则马上返回；

③0表示阻塞，直到子进程结束。

返回值：正常返回的时候waitpid返回子进程ID；如果设置了选项WNOHANG，而调用中waitpid发现没有已退出的子进程，则返回0；如果调用中出错，则返回-1。

Example6.2.2-1:阻塞等待

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int A;
int main(int argc,char **argv){
    int i=0;
    pid_t pid=fork();
    if (pid==-1) return 0;//失败
    else if (pid==0){ //子进程
        printf("son pid =%d\n",getpid());
        A=10;
        while(1){
            printf("son ----- %d,%d\n",i,A)
            sleep(1);
            A++;
            i++;
            if (i==5) return 111;//exit(234);
        }
        printf("son -- over\n");
    }
    else if (pid>0){ //父进程
        printf("parent pid =%d\n",getpid());
        int a;
        wait(&a);
        printf("----a=%d\n",a>>8);
    }
}
```

全局变量 A。父子进程各自拥有独立的地址空间，因此 A 在两个进程中不共享。

wait(&a) 会阻塞，直到子进程退出。

Example6.2.2-2:非阻塞等待

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int A;
int main(int argc,char **argv){
    int i=0;
    pid_t pid=fork();
    if (pid==-1) return 0;//失败
    else if (pid==0){ //子进程
        printf("son pid =%d\n",getpid());
        A=10;
        while(1){
            printf("son ----- %d,%d\n",i,A);
            sleep(1);
            A++;
            i++;
            if (i==5)exit(234);
        }
        printf("son -- over\n");
    }
else if (pid>0){ //父进程
        printf("parent pid =%d\n",getpid());
        int a;
        pid_t tmpd;
        tmpd=waitpid(pid,&a,0);
        printf("----a=%d\n",a>>8);
    }
}
```

6.2.3进程终止

进程终止的方式有8种，前5种为正常终止，后3种为异常终止。

（1）主函数结束，进程终止。

结束主函数的方法为代码运行完毕，或调用return结束主函数。

（2）调用exit函数，进程终止。

void exit(int status) //退出进程

参数：进程结束后要返回给父进程的值。

该函数可以在程序的任何位置结束进程，退出之前先检查是否有文件被打开，如果有把文件打开，则把文件缓冲区数据写入文件，然后退出进程。

（3）调用_exit函数，终止进程。

void _exit(int status) //退出进程

参数：进程结束后要返回给父进程的值。

直接进入内核，不做任何检查，直接终止进程，可以用_exit或_Exit

Example6.2.3-2:退出进程

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc,char **argv){
    int i=0;
    pid_t pid=fork();
    if (pid==-1) return 0;//失败
    else if (pid==0){ //子进程
            printf("son pid =%d\n",getpid());
            while(1){
                    printf("son ----- %d\n",i);
                    sleep(1);
                    i++;
                    if (i==5) {
                            printf("我要退出子进程\n");
                            _exit(0);
                    }
            }
            printf("son -- over\n");
    }
    else if (pid>0){ //父进程
            printf("parent pid =%d\n",getpid());
            int a;
            wait(&a);
            printf("----a=%d\n",a>>8);
    }
}
```

（4）最后一个线程从启动例程返回

（5）最后一个线程调用pthread_exit，方法同（4）。

（6）最后一个线程对取消请求做出响应，方法同（4）。

（7）调用abort函数，异常终止进程。

```
#include <stdlib.h>
void abort(void); //异常终止进程
```

abort通常伴随core文件产生, 如果直接exit，不会产生core文件。

（8）接到一个信号并终止

如Ctrl+C发送SIGKILL信号给进程，则进行会强行关闭。raise函数则允许进程向自身发送信号。

Example6.2.3-4:：异常终止进程

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include<sys/wait.h>
int main(int argc,char **argv){
    int i=0;
    pid_t pid=fork();
    if (pid==-1) return 0;//失败
    else if (pid==0){ //子进程
        printf("son pid =%d\n",getpid());
        while(1){
            printf("son ----- %d\n",i);
            sleep(1);
            i++;
            if (i==5) {
                printf("我要异常终止子进程\n");
                abort();
            }
        }
        printf("son -- over\n");
    }
else if (pid>0){ //父进程
        printf("parent pid =%d\n",getpid());
        int a;
        wait(&a);
        printf("----a=%d\n",a>>8);
    }
}
```

Example6.2.3-5:发送SIGKILL信号终止进程

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include<sys/wait.h>
int main(int argc,char **argv){
    int i=0;
    pid_t pid=fork();
    if (pid==-1) return 0;//失败
    else if (pid==0){ //子进程
        printf("son pid =%d\n",getpid());
        while(1){
            printf("son ----- %d\n",i);
            sleep(1);
            i++;
            if (i==5) {
                printf("我要发送信号终止子进程\n");
                raise(SIGKILL);
            }
        }
        printf("son -- over\n");
    }
else if (pid>0){ //父进程
        printf("parent pid =%d\n",getpid());
        int a;
        wait(&a);
        printf("----a=%d\n",a>>8);
    }
}
```

作业4：

1. 编写写入数据程序，将以下内容你的姓名+学号写入到~/test.txt中。
   1) 你的姓名（拼音）
   2) 你的学号
   3) 完成作业的时间（当前系统时间，请考虑时区问题）

2. 编写一个进程程序，创建一个子进程，调用等待函数等待10秒，之后，将~/ test.txt中内容读出并打印到控制台。

重庆理工大学/电气学院

CHONGQING UNIVERSITY OF TECHNOLOGY

嵌入式Linux系统开发教程

—基于ARM处理器通用平台（arm9-arm11-contexA系列）