进程和线程
1.并行和并发的区别
2.同步和异步的区别
3.信号：基本概念
3.1 信号到达后，进程的行为
3.2 信号类型和默认行为
3.3 改变信号处置：signal()
3.4 发送信号kill()
3.5 检查进程的存在
3.6 发送信号的其他方式：`raise()`和`killpg()`
3.4 显示信号描述
3.5 信号集
3.6 信号掩码（阻塞信号传递）
3.7 改变信号处置：sigaction()
3.8 等待信号：pause()
4.进程
4.0 进程的内存布局
4.1 进程和虚拟内存管理
4.2 进程的创建
4.2.1 父子进程之间的文件共享
4.2.2 fork()的内存含义
4.2.3 fork()之后的竞争以及使用同步信号规避竞争
5.线程
5.1 **线程控制相关函数**
6. 调度算法
6.1 进程调度算法
6.1.2 先来先服务调度算法
6.1.3 最短作业优先调度算法
6.1.4 高相应比优先调度算法
6.1.5 时间片轮转调度算法
6.1.6 最高优先级调度算法
6.1.7 多级反馈队列调度算法

进程和线程

1.并行和并发的区别

并发：在操作系统中，是指一个时间段中有几个程序都处于已经启动运行到运行完毕之间，而且这几个程序都是在同一个处理机上运行。其中两种并发关系分别是***同步和互斥***。（并发是指同一时刻只能有一条指令执行，但多个进程指令被快速轮换执行，使得在宏观上有多个进程被同时执行的效果--宏观上并行，针对单核处理器）

- 互斥：进程间相互排斥的使用临界资源的现象，就叫互斥，就是两个进程不可以同时去调用同一个内存地址的资源。
- 同步(synchronous)：进程之间的关系不是相互排斥临界资源的关系，而是相互依赖的关系。进一步的说明：就是前一个进程的输出作为后一个进程的输入，当第一个进程没有输出时第二个进程必须等待。具有同步关系的一组并发进程相互发送的信息称为消息或事件。（***彼此有依赖关系的调用不应该同时发生，而同步就是阻止那些"同时发生"的事情***）

- 其中并发又有伪并发和真并发，伪并发是指单核处理器的并发，真并发是指多核处理器的并发。

2.同步和异步的区别

- 同步(synchronous)：进程之间的关系不是相互排斥临界资源的关系，而是相互依赖的关系。进一步的说明：就是前一个进程的输出作为后一个进程的输入，当第一个进程没有输出时第二个进程必须等待。具有同步关系的一组并发进程相互发送的信息称为消息或事件。
- 异步(asynchronous)：异步和同步是相对的，同步就是顺序执行，执行完一个再执行下一个，需要等待、协调运行。异步就是彼此独立,在等待某事件的过程中继续做自己的事，不需要等待这一事件完成后再工作。线程就是实现异步的一个方式。异步是让调用方法的主线程不需要同步等待另一线程的完成，从而可以让主线程干其它的事情。

3.信号：基本概念

信号是事件发生时对进程的通知机制，有时也成为软件中断。

一个进程（具有合适权限的）能够向另一进程发送信号，信号的这一用法可以用作为一种同步技术，甚至是进程间通信的原始形式。进程可以向自身发送信号。

注意，在下面我们说过，进程是很呆板的，进程所能做的很多事情都是依靠内核来做的，进程发送信号也是如此，发送进程的诸多信号，通常都是源于内核。引发内核为进程产生信号的各类事件如下：

- 硬件发生异常，硬件检测到一个错误条件并通知内核，随机由内核发送相关的信号给对应的进程，比如说被0除；
- 用户键入能够产生信号的终端特殊字符。其中包括中断字符（Ctrl + C）等等；
- 发生软件事件；

针对每一个信号，都定义了一个唯一的整数，从1顺序展开。`<signal.h>`一SIGXXXX形式的符号名字对这些整数做了定义，一般不同操作系统的实际编号都是不一样的，所以说我们一般使用信号的符号名字。

比如说，当用户键入中断字符的时候，将传递给进程 `SIGINT` *信号。*

信号分为两大类，第一组用于内核向进程通知事件，构成所谓的标准信号。Linux中标准信号的编号范围为1~31。

另一组信号由实时信号构成，（应该是进程之间实现同步的信号）。

3.1 信号到达后，进程的行为

信号到达之后，进程视具体信号执行如下默认操作之一：

- 忽略信号，内核将信号丢弃，信号对进程没有产生任何影响；
- 终止（杀死）进程，指的是进程异常终止，而不是进程因为调用`exit()`而发生的正常终止；
- 产生核心转储文件，同时进程终止：核心转储文件就是保存进程终止的时候的上下文环境，我们可以利用`gdb`来将该文件加载到调试器中以检查进程终止时的状态；
- 停止进程，暂停进程的执行；
- 于之前暂停后再度恢复进程的执行；

除了上述所讲，进程根据特定的信号采取的默认行为之外，程序也可以改变信号到达时的相应行为，就好比是QT上的信号与槽函数，我们可以自定义槽函数。在系统中，自定义槽函数称为对信号的处置设置，程序可以将对信号的处置设置为：

- 采取默认行为；
- 忽略信号，就是我们可以手动去忽略某些信号；

- 执行信号处理程序，就是执行我们自定义的一些槽函数。

信号处理器程序是由程序员编写的函数，用于为响应传递来的信号而执行适当的任务。比如说，shell为SIGINT信号（由中断字符Ctrl + C产生）提供了信号处理器程序，令其停止当前正在执行的工作，并将***控制返回到shell的主输入循环***；

需要注意的是我们无法将信号处理设置为终止进程或者转储核心，但是我们可以为信号安装一个处理器程序，并于其中调用 `exit()` ***函数或者*** `abort()` ***。*** `abort()` ***函数为进程产生*** `SIGABRT` ***信号，该信号将引发进程转储核心文件并终止。***

3.2 信号类型和默认行为

`SIGABRT`
当进程调用 abort()函数（21.2.2 节）时，系统向进程发送该信号。默认情况下，该信号会终止进程，并产生核心转储文件。这实现了调用 abort()的预期目标，产生核心转储文件用于调试。

`SIGALRM`
经调用 `alarm()` 或 `setitimer()` 而设置的实时定时器一旦到期，内核将产生该信号。实时定时器是根据挂钟时间进行计时的（即人类对逝去时间的概念）

`SIGBUS`
产生该信号（总线错误，bus error）即表示发生了某种内存访问错误。如 49.4.3 节所述，当使用由 `mmap()` 所创建的内存映射时，如果试图访问的地址超出了底层内存映射文件的结尾，那么将产生该错误。

`SIGCHLD`
当父进程的某一子进程终止（或者因为调用了 exit()，或者因为被信号杀死）时，（内核）将向父进程发送该信号。当父进程的某一子进程因收到信号而停止或恢复时，也可能会向父进程发送该信号。

`SIGCLD`
与 SIGCHLD 信号同义。

`SIGCONT`
将该信号发送给已停止的进程，进程将会恢复运行（即在之后某个时间点重新获得调度）。
当接收信号的进程当前不处于停止状态时，默认情况下将忽略该信号。进程可以捕获该信号，以便在恢复运行时可以执行某些操作。

`SIGINT`
当用户键入终端中断字符（通常为 Control-C）时，终端驱动程序将发送该信号给前台进程组。该信号的默认行为是终止进程。

`SIGKILL`
此信号为“必杀（sure kill）”信号，处理器程序无法将其阻塞、忽略或者捕获，故而“一击必杀”，总能终止进程。

3.3 改变信号处置：signal()

UNIX系统提供了两种方法来改变信号处置：`signal()` 和 `sigaction()` 。`signal()` 时设置信号处置的原始API，所提供的接口比 `sigaction()` 简单。最重要的是，`sigaction()` 函数在不同Unix中实现间没有差异，即具备可移植性，所以说 `sigacton()` 是建立信号处理器的首选API。

```c
#include<signal.h>
void (*signal(int sig,void (*handler)(int)))(int);
//该函数的返回值为一个函数指针
//参数：
//sig，为特殊情况信息，即希望修改处置的信号编号
//handler，为特殊情况下将要调用的函数的地址值，即函数指针，标识信号抵达时所调用的函数的地址，该函数没有返回值，并接收一个整形参数

void handler(int sig)
{
    /*Code for the handler*/
}
```

`signal()`的返回值是之前的信号处置。像handler参数一样，这是一枚指针，指向的是带有一个整型参数且无返回值的函数。利用这一点，我们可以暂时为信号建立一个处理器函数，接着将信号处置重置为其本来面目：

```c
void (*oldHandler)(int);//声明一个函数指针变量oldHandler，用来存储之前的信号处置函数

oldHandler = signal(SIGINT,newHandler);
if(oldHandler == SIG_ERR)
    errExit("signal");
/*Do something else here.During this time,if SIGINT is delivered,newHandler will be used to handle the signal*/

if (signal(SIGINT,oldHandler) == SIG_ERR)
    errExit("signal");
```

从上面可以看出，我们如果使用`signal()`的话，将无法在不改变信号处置的同时，还能获得当前的信号处置，要想做到这一点，就得需要使用`sigaction()`。

我们可以针对信号处理器函数指针做如下类型定义，将有助于理解`signal()`的原型：

```c
typedef void (*sighandler_t)(int);

//接着我们可以利用sighandler_t来将函数原型改写为
sighandler_t signal(int sig,sighandler_t handler);
```

在为`signal()`指定handler参数的时候，可以使用如下值来代替函数地址:

`SIGDFL`

将信号处置重置为默认值，可以将之前调用`signal()`调用所改变的信号处置还原；

`SIG_IGN`

忽略该信号，如果信号专为此进程而生，那么内核会默默将其丢弃，进程从未知道产生过该信号；

如果`signal()`成功将返回之前的信号处置，有可能是先前安装的处理器函数地址，也可能是常量SIG_DFL和SIG_IGN之一，如果调用失败，`signal()`将会返回SIG_ERR。

示例1：

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

```c
#include<signal.h>

void timeout(int sig)
{
    if(sig==SIGALRM)
        puts("Time out!");
    alarm(2);
    //本行代码的目的是为了每隔2秒重复产生SIGALRM信号，在程序的一开始的时候会调用alarm函数，接着会在信号处理器中继续调     //用alarm函数，这样的话，程序会继续产生SIGALRM信号，接着会继续在信号处理其中调用，最终结束程序

}
void keycontrol(int sig)
{
    if(sig==SIGINT)
        puts("CTRL + C pressed");
}
//该函数的目的是为了测试signal()函数的基本功能
int main(int argc,char *argv[])
{
    int i;
    signal(SIGALRM,timeout);
    signal(SIGINT,keycontrol);
    alarm(2);

    for(i = 0;i<3;i++)
    {
        puts("wait...");
        sleep(100);
    }
        return 0;
}
```

程序的执行结果：

```
kirito@LAPTOP-Q1OL0END:~/Codes/signal_handler$ ./a.out
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

在实际的执行过程中，我们可以看到程序执行时间并未达到5分钟，而是很快就结束，具体原因如下：

发生信号时将唤醒由于调用sleep函数而进入阻塞状态的进程。

调用函数的主体的确是操作系统，但是进程处于睡眠状态的时候，无法调用函数。因此，产生信号的时候，为了调用信号处理器，将唤醒由于sleep函数而进入阻塞状态的进程。

而且进程一旦被唤醒，就不会再进入睡眠状态。即使还未到sleep函数中规定的时间也是如此。

（这个过程如果从内核角度来看的话，就是该进程在执行的过程中，先是睡眠，接着内核收到了alarm的信息，接着内核会向该进程发送信号，然后内核会先将进程唤醒，***接着是内核代表进程来调用处理器程序***，当处理器返回时，主程序会在处理器打断的位置恢复执行）

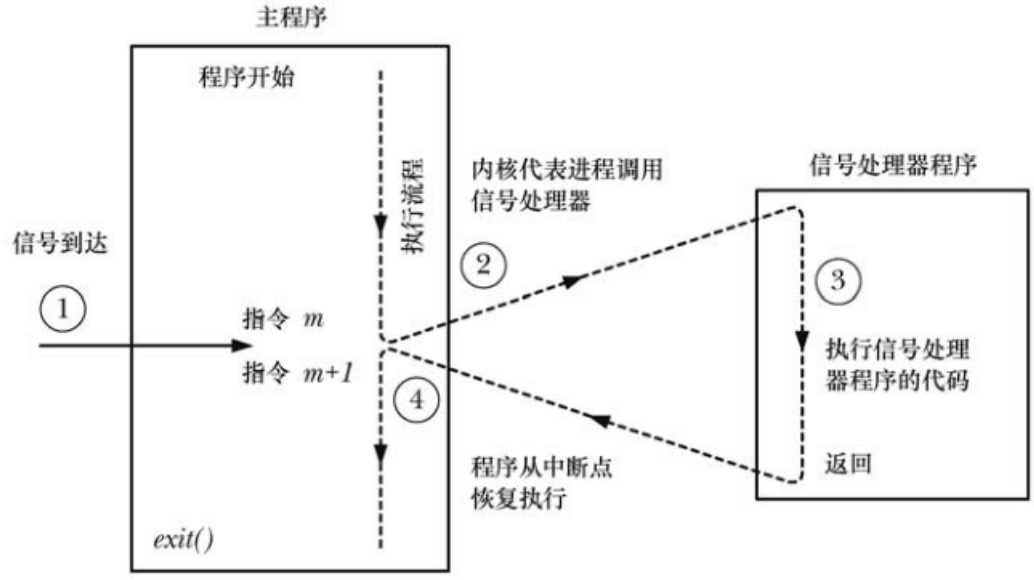


图 20-1：信号到达并执行处理器程序

内核在调用信号处理程序的时候，会将引发调用的信号编号作为一个整型参数传递给处理器函数，即处理器函数中的sig参数，如果我们使用处理器仅仅只捕获一种信号，那么该参数几乎没用；但是，如果我们安装相同的处理器来捕捉不同类型的信号，那么就可以利用该参数来判定引发对处理器调用的是何种信号，接着需要进行什么操作。

3.4 发送信号kill()

我们一般使用kill作为进程向另一进程发送信号的术语，原因在于早期的UNIX实现中大多数信号的默认行为是终止进程。

```
#include<signal.h>
int kill(pid_t pid,int sig);

//return 0 on success,or -1 on error
```

下图是手册中的对于函数功能的描述：

```
DESCRIPTION
       The kill() system call can be used to send any signal to any process group or process.

       If pid is positive, then signal sig is sent to the process with the ID specified by pid.

       If pid equals 0, then sig is sent to every process in the process group of the calling process.

       If  pid  equals -1, then sig is sent to every process for which the calling process has permission
       to send signals, except for process 1 (init), but see below.

       If pid is less than -1, then sig is sent to every process in the process group whose ID is -pid.

       If sig is 0, then no signal is sent, but existence and permission checks are still performed; this
       can be used to check for the existence of a process ID or process group ID that the caller is per-
       mitted to signal.

       For a process to have permission to send a signal, it must either be privileged (under Linux: have
       the  CAP_KILL  capability  in  the user namespace of the target process), or the real or effective
       user ID of the sending process must equal the real or saved set-user-ID of the target process.  In
       the  case of SIGCONT, it suffices when the sending and receiving processes belong to the same ses-
       sion.  (Historically, the rules were different; see NOTES.)
```

- 如果`pid`大于0，即该进程是positive的，该信号就会发送给`pid`指定的进程；
- 如果`pid`等于0，该信号就会发送给与调用进程同一个组的所有进程；
- 如果`pid`等于-1，信号将会发送给每一个调用进程有权限发送信号的进程，就是调用进程可以向谁发信号就把该信号发送给谁，除了`init`进程；
- 如果`pid`小于-1的话，该信号会发送到指定组中的所有进程，组的进程号是`-pid`，该种操作在shell作业控制中有特殊用途；

- 如果信号值`sig`的值为零的话，不会有任何信号发送出去，但是依旧会检查相应的进程是否存在以及检查一下是否有权限向对应的进程发送信号。这可以用来检查那些进程有权限发送信号的进程ID或者进程组ID的存在性。

下图是手册中的对于该函数返回值的描述：

```
RETURN VALUE
       On success (at least one signal was sent), zero is returned.  On error, -1 is returned, and  errno
       is set appropriately.

ERRORS
       EINVAL An invalid signal was specified.

       EPERM  The  calling  process does not have permission to send the signal to any of the target pro-
              cesses.

       ESRCH  The target process or process group does not exist.  Note that an existing process might be
              a zombie, a process that has terminated execution, but has not yet been wait(2)ed for.
```

3.5 检查进程的存在

验证进程的存在并不能保证特定的程序的运行，因为内核会随着进程的生灭循环使用进程ID。而一段时间之后，同一进程ID所指恐怕是另一进程了。此外特定进程ID可能存在，但是一个僵尸进程。

3.6 发送信号的其他方式：`raise()`和`killpg()`

`raise()`函数用来向自身发送信号：

```
#include<signal.h>
int raise(int sig);

//return 0 on success,or nonzero on error
//调用函数出错的话，唯一可能发生的错误是EINVAL，即sig无效
```

调用`raise()`相当于对`kill()`的如下调用：

```
kill(getpid(),sig);
```

支持线程的话，会将`raise()`实现为：

```
pthread_kill(pthread_self(),sig);
```

`killpg()`函数用来向某一个进程组的所有成员发送一个信号：

```
#include<signal.h>

int killpg(pid_t pgrp,int sig);
//return 0 on success,or -1 on error
```

3.4 显示信号描述

即我们可以通过从指定的函数获取指定信号的相关描述：

每一个信号都有一串与之相关的可打印说明，这些描述为于数组`sys_siglist`中。

```c
#include<string.h>
char *strsignal(int sig);
//returns pointer to signal description string
```

3.5 信号集

许多信号相关的系统调用都需要能表示一组不同的信号。多个信号可以使用一种称之为信号集的数据结构来表示，其系统类型为`sigset_t`。有相关的函数来操作信号集：

```c
#include<signal.h>

int sigemptyset(sigset_t *set);
//初始化一个未包含任何成员的信号集
int sigfillset(sigset_t *set);
//初始化一个包含所有信号的信号集
//Both return 0 on success,or -1 on error
```

注意必须使用`sigemptyset()`或者`sigfillset()`来初始化信号集。

```c
#include<signal.h>

int sigaddset(sigset_t *set,int sig);
//向集合中添加单个信号
int sigdelset(sigset_t *set,int sig);
//向集合中删除单个信号
```

3.6 信号掩码（阻塞信号传递）

内核会为每一个进程维护一个信号掩码，（线程也有这个属性），即一组信号，并将阻塞其针对该进程的传递。相当于排一个队，一个一个来给相应的进程发送信号。

当进程接收了一个该进程正在阻塞的信号的话，那么会将该信号添加到进程的信号集中。现在相当于有两个队伍，一个是内核为进程维护的信号掩码，阻塞外界发送给进程的信号；另一个是进程自己维护的信号集，该集合中都是进程接收的信号集合，正在排队一个一个等待进程来处理。

我们可以使用`sigpending()`来确定进程中处于等待状态的是哪一些信号。

3.7 改变信号处置：sigaction()

`signal()`在UNIX系列的不同操作系统中可能存在区别，但是`sigaction()`完全相同，具有可移植性，所以说我们一般都是使用`sigaction()`。

```c
#include<signal.h>

int sigaction(int sig,const struct sigaction *act,struct sigaction *oldact);
//return 0 on success,or -1 on error
```

sig参数标识想要获取或者改变的信号编号，就是捕获的信号，该参数可以是除去SIGKILL和SIGSTOP之外的任何信号；

act参数是一个指针，指向描述信号新处置的数据结构。如果对信号的现有处置有兴趣，可以将该参数指定为NULL。oldact参数是指向同一结构类型的指针，用来返回之前信号处置相关信息。可以设置为NULL，什么也不获得。

sigaction的结构体：

```
struct sigaction{
    void (*sa_handler)(int);//函数指针，指向处理函数
    sigset_t sa_mask;//信号集，进程自己维护的一个关于信号的结构体

    int sa_flags;//控制处理函数调用的信号
    void (*sa_restorer)(void);//不供应用程序使用
};
```

3.8 等待信号：pause()

使得进程暂停执行，直至信号到达为止。

4.进程

> ***题外话：***
>
> ----------进程的呆板----------
>
> 对于进程自身来说，许多的时间的发生都是无法去预计的，进程自己是很呆板的，进程不知道自己的内存空间是驻留在内存中还是保留在交换空间中（磁盘空间中的保留区域），进程也不知道自己访问的文件是什么，仅仅是通过名称引用文件而已。
>
> 进程的运行方式堪称为“与世隔绝”，进程之间不能直接通信，进程本身是无法创建新进程的，哪怕自己“自行了断”都不行，进程也不能与计算机外接的输入输出的设备直接通信。
>
> ----------内核的强大智能----------
>
> 内核是运行系统的中枢所在，对于系统无所不知，为系统上所有进程的运行提供便利：
>
> - 由哪一个进程来接掌CPU的使用，即内核对于进程的调度，是内核说了算；
> - 在内核维护的数据结构中，包含了所有正在运行的进程有关的信息，并且会随着进程的行为去更新这些数据结构；
> - 每一个进程的虚拟内存与计算机物理内存以及磁盘交换区之间的映射关系也在内核维护的数据结构之中；
> - ***进程之间所有的通信都是要通过内核所提供的通信机制来完成***；
> - 响应进程发出的请求，内核会创建新的进程，终结现有进程，最后由内核（设备驱动程序）来执行与输入/输出设备之间的所有的直接通信；
>
> 这些措辞：
>
> “某一个进程创建另一个进程”、“某一个进程可以创建管道”、“某一个进程将数据写进文件”、“调用exit()以终止某进程”等等
>
> ***这些都是由内核来居中“调停”，上面的说法本质是，“某一个进程可以请求内核创建一个进程”。***

进程是由内核定义的抽象的实体，并为该实体分配用以执行程序的各项系统资源，***从内核角度来看的话，进程是由用户内存空间和一系列内核数据结构组成的，其中用户内存包含了程序代码以及代码所使用的变量，而内核数据结构则用于维护进程状态信息。***

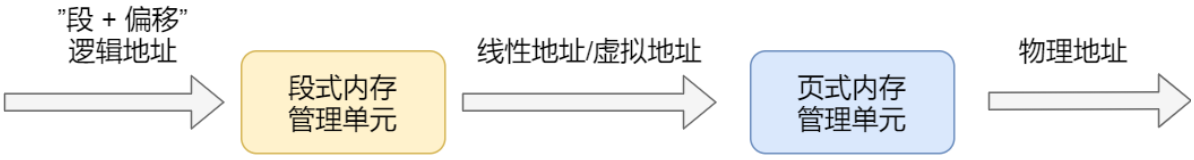
4.0 进程的内存布局

进程一般划分为以下几个部分（也称为段）：

- 文本段：程序的指令，文本段具有只读属性，以防止进程通过错误指针意外修改自身指令。因为多个进程可以同时运行同一个程序，所以又将文本段设置为可共享，***这样一份程序代码的拷贝可以映射到所有这些进程的虚拟地址空间中***；
- 数据：程序使用的静态变量和全局变量，该数据段又分为初始化数据段和未初始化数据段，未初始化数据段包含了未进行显式初始化的全局变量和静态变量，程序启动之前，系统将本段的所有内存初始化为0；
- 堆：程序可以从该区域动态分配额外的内存；
- 栈：栈是一个动态增长和收缩的段，由栈帧组成，系统会为每个当前调用的函数分配一个栈帧。栈帧中存储了函数的局部变量、实参和返回值。

4.1 进程和虚拟内存管理

Linux内存布局主要采用的是页式内存管理，用该种方法来管理虚拟内存和物理内存之间的映射关系。



由于早期的intel处理器采用的是段式内存管理，后来进行升级，将段式内存管理和页式内存管理结合起来，演化出了上图。

由于intel处理器的发展历程所致，毕竟CPU的硬件结构是这样，Linux内核只好服从intel的选择，但是“上有政策，下有对策”，若惹不起就躲着走。Linux系统的每一段都是从0地址开始的，意思是所有的段的起始地址是一样的，***这就导致了Linux系统中的地址空间都是线性地址即虚拟地址。***

为了这样内存管理模式，内核需要为每一个进程维护一张页表。通常情况下，由于可能存在大段的虚拟地址空间并未投入使用，故而也没有必要为其维护相应的页表条目。

内核可以为进程分配和释放页（和页表条目）。所以进程的有效虚拟地址范围在其生命周期可以发生变化。

- 由于栈向下增长超出之前曾达到的位置；
- 当在堆中分配和释放内存时。通过`malloc`等函数来提高program break的位置
- 当调用`shmat()`连接System V共享内存区时，或者调用`shmdt()`脱离共享内存区时；
- 当调用`mmap()`创建内存映射时，或者当调用`munmap()`接触内存映射的时候；

> 虚拟内存的实现需要硬件中分页内存管理单元PMMU的支持。PMMU 把要访问的 每个虚拟内存地址转换成相应的物理内存地址，当特定虚拟内存地址所对应的页没有驻留 于 RAM 中时，将以页面错误通知内核。

虚拟内存管理将进程的虚拟地址空间和RAM物理地址空间隔离开来，有很多优点：

- 进程与进程、进程与内核相互隔离，因为每一个进程都有属于自己的一个页表映射到不同的物理地址。每一个进程的页表条目指向RAM或者交换区中截然不同的物理页面的集合；
- 适当情况下，***两个或者更多的进程能够共享内存，这是由于内核可以使不同进程的页表条目指向相同的RAM页。***相关情景如下：

-a 执行同一程序的多个进程，可以共享一份（只读的）程序代码副本。当多个程序执行相同的程序文件，会隐式地实现这一类型的共享；

-b 进程可以使用`shmget()`和`mmap()`系统调用显式地请求与其他进程共享内存区。这么做的是出于进程间通信的目的；

- 便于实现内存保护机制，对页表赋予各种权限，接着赋予进程响应的权限。允许每一个进程堆内存采取不同的保护措施；
- 程序员和编译器、链接器之类的工具无需关注程序在RAM中的物理布局；
- 一个进程所占的内存（虚拟内存大小）能够超出RAM容量；

还有就是，每一个进程使用的RAM减少了，RAM(内存，主存)中同时可以容纳的进程数量就增多了，这就增大了如下事件的概率：在任意时刻，CPU都可执行至少一个进程，提高了CPU的利用率。

下面图示勾勒了shell执行一条命令的所执行的步骤：shell读取命令，进行各种处理，随之创建子进程以执行该命令，如此循环不已。

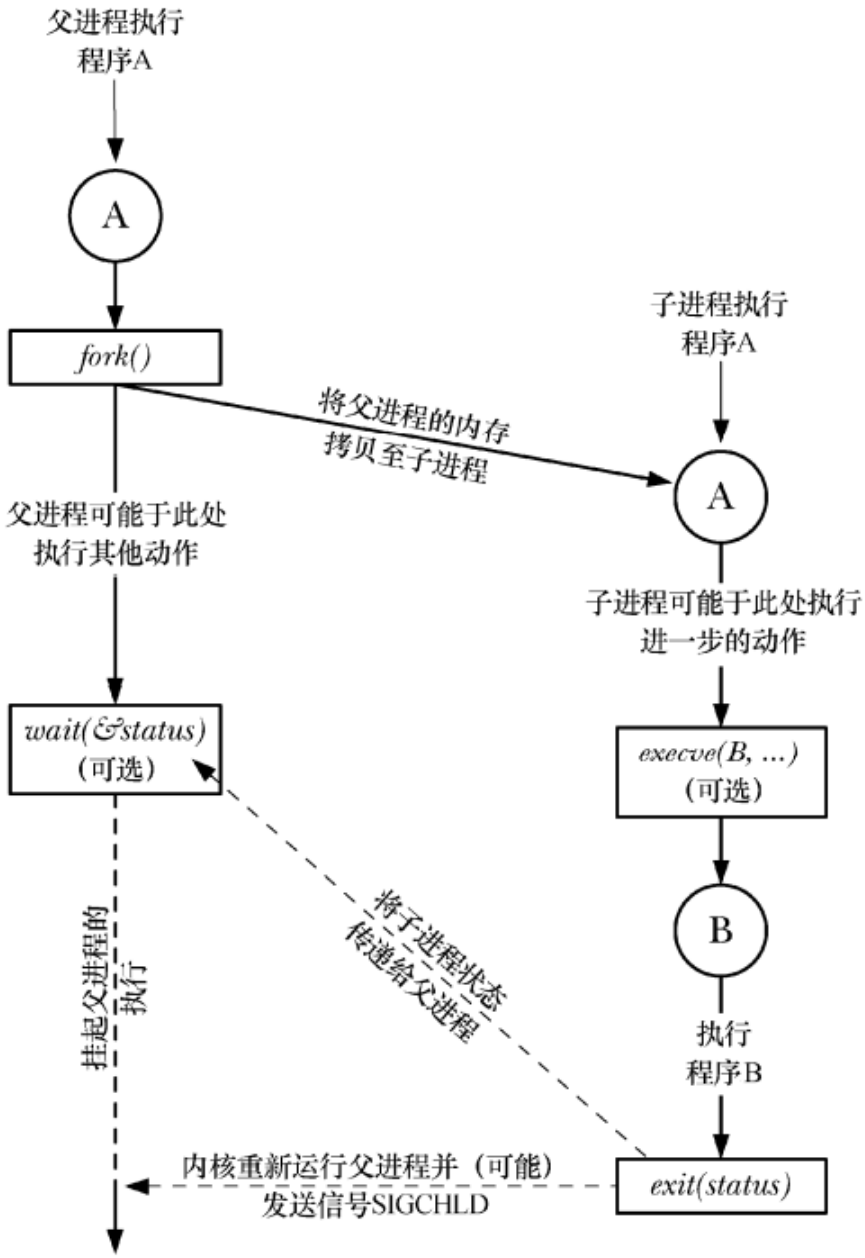


图 24-1：概述函数 fork()、exit()、wait()和 execve()的协同使用

注意上图中，`execve()`的调用并非必须，有的时候让子进程继续执行与父进程相同的程序反而会有妙用。最终，两种情况殊途同归：都是要通过调用`exit()`（或者接收一个信号）来终止子进程，而父进程可以调用`wait()`来获取其终止状态。

同样，对`wait()`的调用也属于可选项，父进程完全可以对子进程不闻不问，继续我行我素。不过一般都是要使用`wait()`的，每每在SIGCHLD信号的处理程序种使用。当子程序终止的时候，内核会为其父进程产生此类信号（默认的处理是忽略SIGCHLD信号）

exec函数说明:

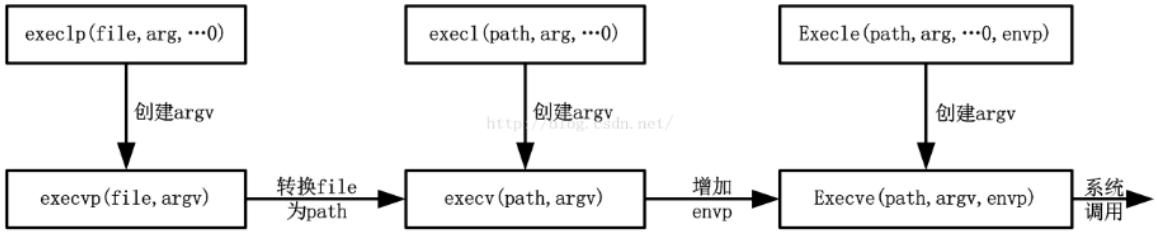
```
#include <unistd.h>
extern char **environ;

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

```
l - execl(), execlp(), execle()
       The const char *arg and subsequent ellipses can be thought of as
arg0, arg1, ..., argn.  Together        they describe a list of one or more
pointers to null-terminated  strings  that
       represent  the  argument  list available to the executed program.
The first argument, by              convention, should point to the
filename associated with the file being executed.  The
       list of arguments must be terminated by a null pointer, and, since
these are variadic functions,        this pointer must be cast (char *)
NULL.

       By contrast with the 'l' functions, the 'v' functions (below) specify
the command-line arguments        of the executed program as a vector.

v - execv(), execvp(), execvpe()
       The char *const argv[] argument is an array of pointers to null-
terminated strings that represent        the argument list available to the
new program.  The first argument,  by  con-
       vention, should point to the filename associated with the file being
executed.  The array of           pointers must be terminated by a null
pointer.
```

六个exec函数的区别在于：

- 待执行程序文件是由文件名字还是由文件路径名来确定的；
- 新程序的参数是一个一个列出还是由一个指针数组来引用的；
- 把调用进程的环境传递给新程序还是给新程序指定新的环境；

这些函数只有在出错的时候才会返回到调用者（返回-1的时候为失败）；否则，控制将传递给新程序的起始点，通常也就是我们fork出来的子程序。



4.2 进程的创建

系统是调用`fork()`创建一个新的进程（child）,对于其父亲的翻版

```
#include<unistd.h>
pid_t fork(void);
//In parent :returns process ID of child on success,or -1 on error;
//In successfully created child :always returns 0
```

调用`fork()`之后将会存在两个进程，且每一个进程都会从`fork()`的返回处继续执行。

两个进程将执行相同的程序文本段，但却各自拥有不同的栈段、数据段以及堆段拷贝。子进程的栈、数据以及栈段开始时是对父进程内存响应部分的完全复制。执行`fork()`之后，每一个进程均可修改各自的栈数据、以及堆段中的变量，且不影响其他进程。

子进程可以调用`getpid()`以获取自身的进程ID，调用`getppid()`以获取父进程ID。

调用`fork()`之后，系统将率先"垂青"于哪一个进程，是无法确定的，在设计拙劣的程序中，这种不确定性可能会导致所谓"竞争条件（`race conndition`）"的错误。

4.2.1 父子进程之间的文件共享

执行`fork()`时，子进程会获得父进程所有的文件描述符的副本，这意味着父进程和子进程对应的各种描述符均指向相同的打开文件句柄。并且一个打开文件的这些属性因之而在父子进程间实现了共享。

父子进程间共享打开文件属性的妙用屡见不鲜，比如说如果父进程和子进程需要同时写入同一个文件的时候，使用文件偏移量可以保证二者不会覆盖彼此的输入内容，但是这并不能阻止父子进程的输出混杂到一块，我们想要规避这一种现象，需要进行进程间同步。比如说父进程可以使用系统调用`wait()`来停止运行等待子进程退出。

如果不需要这种对文件描述符的共享的话，那么我们在设计应用程序的时候就应该在fork调用之后注意：一，令父子进程使用不同的文件描述符；二，各自立即关闭不再使用的描述符，以免造成冲突。

比如说管道通信中，由于普通的管道通信是半双工的，所以说在执行fork系统调用之后，我们需要将父子进程相应不需要的端口给关闭：

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
    int p[2];
    pipe(p);    //create pipe,p[0] is read port,p[1] is write port
```

```
    if(fork()==0){  //child process
        close(0);   //close stansard input
        dup(p[0]);  //duplicate pipe read port to standard input
        close(p[0]);    //close original pipe
        close(p[1]);
        execl("/usr/bin/sort","sort",NULL);
    }else{      //parent process
        close(1);   //close stansard output
        dup(p[1]);  //duplicate pipe write port to standard output
        close(p[0]);    //close original pipe
        close(p[1]);
        execl("/bin/ls","ls",NULL);
    }
    return 0;
}
```

4.2.2 fork()的内存含义

由于fork调用之后，从概念上讲，子进程会对父进程程序段、数据段、堆段以及栈段创建拷贝。在早期的UNIX实现中，此类复制确实是原汁原味：将父进程内存拷贝至交换空间，以此创建新进程映像，而在父进程保持自身内存的同时，将换出映像置为子进程。这样难免有点浪费。所以大部分的UNIX实现采用两种技术：

- 内核将每一个进程的代码段标记为只读，从而使得进程无法修改自身代码，这样的话，父子进程就可以共享统一代码段。系统调用fork()在为子进程创建代码段的时候，子进程页表项指向与父进程相同的物理内存页帧；
- 对于父进程数据段、堆段和栈段的各页，内核采用写时复制技术来处理，本质上就是当进程使用到该部分数据的时候，系统会复制该部分数据，接着系统会将新的页面拷贝分配给所需要的进程，同时还会对子进程的相关页表项作适当的调整，

4.2.3 fork()之后的竞争以及使用同步信号规避竞争

在调用`fork()`之后，创建出子进程。在内核对其进行调度的时候，会产生问题，谁先谁后的问题，有的了Linux中采用fork()之后先调度父进程，有的却先调度子进程。如果我们想要实现我们去自定义某一种特定的执行顺序的时候，我们就需要采用某种同步技术。比如说，信号量、文件锁、以及进层间经由管道的信息发送。

5.线程

每一个进程都可以执行多个线程。可以将线程想象成共享同一虚拟内存以很多其他属性的进程，每一个进程都会执行相同的程序代码，共享同一个数据区域和堆，可是，每一个线程都拥有属于自己的栈，用来装载本地变量和函数调用链接信息。

创建线程如果以内核的角度来看的话：

- 进程向内核发起请求，内核创建一个缩小版本的进程，缩小版本的进程的虚拟内存地址都是一样的；
- 接着内核给每一个进程单独分配一个栈，载入响应的函数调用链接一类的信息；
- 进程创建线程完毕；

线程的主要优点在于线程之间的数据共享(通过全局变量)更为容易，而且就某一些算法来说的话，使用多线程来实现比用多进程来实现更为的自然，因为算法肯定是用一个程序来进行说明的，不能说使用很多的程序来对其进行说明。

什么是线程?

- LWP：light weight process 轻量级的进程，本质仍是进程(在Linux环境下)
- 进程：独立地址空间，拥有PCB
- 线程：也有PCB，但没有独立的地址空间(共享)
- 区别：在于是否共享地址空间。 独居(进程)；合租(线程)。
- Linux下： 线程：最小的执行单位，调度的基本单位。
- ***进程：最小分配资源单位，可看成是只有一个线程的进程。***

Linux内核线程实现原理

类Unix系统中，早期是没有“线程”概念的，80年代才引入，借助进程机制实现出了线程的概念。因此在这类系统中，进程和线程关系密切。

- 轻量级进程(light-weight process)，也有PCB，创建线程使用的底层函数和进程一样，都是clone。
- 从内核里看进程和线程是一样的，都有各自不同的PCB，但是PCB中指向内存资源的三级页表是相同的。
- 进程可以蜕变成线程
- 线程可看做寄存器和栈的集合
- 在linux下，线程最是小的执行单位；进程是最小的分配资源单位
- 察看LWP号：ps –eLf pid 查看指定线程的lwp号。

三级映射：进程PCB --> 页目录(可看成数组，首地址位于PCB中) --> 页表 --> 物理页面 --> 内存单元--参考：《Linux内核源代码情景分析》 ----毛德操

- 对于进程来说，相同的地址(同一个虚拟地址)在不同的进程中，反复使用而不冲突。原因是他们虽虚拟址一样，但，页目录、页表、物理页面各不相同。相同的虚拟址，映射到不同的物理页面内存单元，最终访问不同的物理页面。
- 但！线程不同！两个线程具有各自独立的PCB，但共享同一个页目录，也就共享同一个页表和物理页面。所以两个PCB共享一个地址空间。
- 实际上，无论是创建进程的fork，还是创建线程的`pthread_create`，底层实现都是调用同一个内核函数clone。
- 如果复制对方的地址空间，那么就产出一个“进程”；如果共享对方的地址空间，就产生一个“线程”。
- 因此：Linux内核是不区分进程和线程的。只在用户层面上进行区分。所以，线程所有操作函数`pthread_*`是库函数，而非系统调用。

线程共享资源

- 1.文件描述符表
- 2.每种信号的处理方式
- 3.当前工作目录
- 4.用户ID和组ID
- 5.内存地址空间 (.text/.data/.bss/heap/共享库)

线程非共享资源

- 1.线程id
- 2.处理器现场和栈指针(内核栈)
- 3.独立的栈空间(用户空间栈)

- 4.errno变量
- 5.信号屏蔽字
- 6.调度优先级

线程优、缺点

- 优点： 1. 提高程序并发性 2. 开销小 3. **数据通信、共享数据方便**
- 缺点： 1. 库函数，不稳定 2. 调试、编写困难、gdb不支持 3. 对信号支持不好
- **优点相对突出，缺点均不是硬伤。Linux下由于实现方法导致进程、线程差别不是很大。**

5.1 线程控制相关函数

`pthread_self`**函数 获取线程ID。其作用对应进程中** `getpid()` **函数。**

- `pthread_t pthread_self(void)`; 返回值：成功：0； 失败：无!
- 线程ID：`pthread_t`类型，本质：在Linux下为无符号整数(%lu)，其他系统中可能是结构体实现
- 线程ID是进程内部，识别标志。(两个进程间，线程ID允许相同)
- 注意：不应使用全局变量 `pthread_t tid`，在子线程中通过`pthread_create`传出参数来获取线程ID，而应使用`pthread_self`。

`pthread_create`**函数 创建一个新线程。 其作用，对应进程中fork() 函数。**

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)`
- 返回值：成功：0； 失败：错误号 -----Linux环境下，所有线程特点，失败均直接返回错误号。

参数:

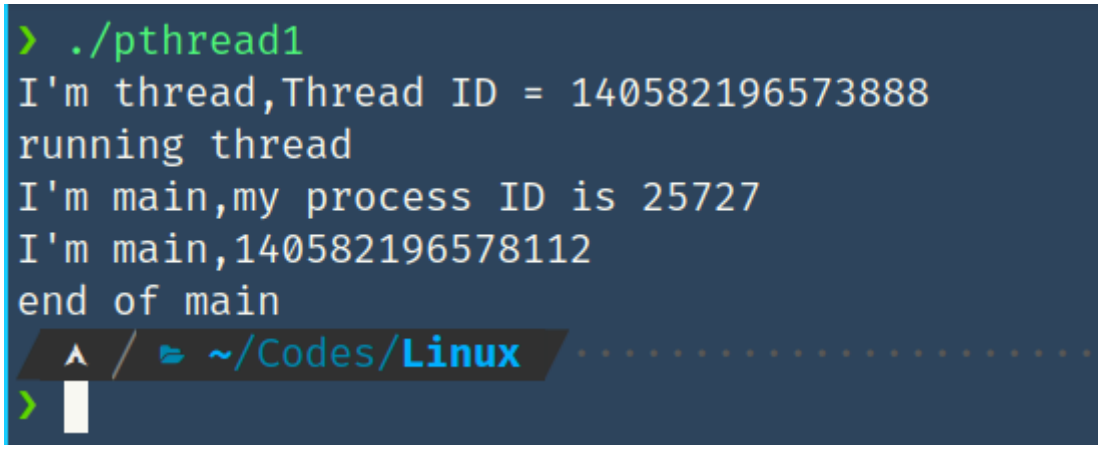
- `pthread_t`：当前Linux中可理解为：`typedef unsigned long int pthread_t`;
- 参数1：传出参数，保存系统为我们分配好的线程ID
- 参数2：通常传NULL，表示使用线程默认属性。若想使用具体属性也可以修改该参数。
- 参数3：函数指针，指向线程主函数(线程体)，该函数运行结束，则线程结束。
- 参数4：线程主函数执行期间所使用的参数，如要传多个参数, 可以用结构封装。

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void *fun(void *arg)
{
    printf("I'm thread, Thread ID = %lu\n", pthread_self());
    return NULL;
}

int main(void)
{
    pthread_t tid;

    pthread_create(&tid, NULL, fun, NULL);
    sleep(1);       // 在多线程环境中，父线程终止，全部子线程被迫终止
    printf("I am main, my pid = %d\n", getpid());

    return 0;
}
```

```c
#inlcude<unistd.h>
unsigned int sleep(unsigned int seconds);
//参数：线程挂起时间
//返回值：进程/线程挂起到参数所指定的时间则返回0，若有信号中断的话则返回剩余时间
//作用：线程告诉操作系统，在second秒的时间内，自身不需要调度（直接睡觉了），不要给自己分配占用CPU的时间，使得CPU更容易易主
```

```c
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
// a simple example for thread
void * thread_main(void *);

int main(int argc,char *argv[])
{
    pthread_t t_id;
    int thread_param = 5;

    if(pthread_create(&t_id,NULL,thread_main,(void *)&thread_param)!=0)
    {
        puts("pthread_create() error");
        return -1;
    }
    sleep(2);
        printf("I'm main,my process ID is %lu\n",getpid());
        printf("I'm main,%lu\n",pthread_self());
    puts("end of main");
    return 0;
}

void * thread_main(void *arg)
{
    int cnt = *((int *)arg);
        printf("I'm thread,Thread ID = %lu\n",pthread_self());
    for(int i = 0;i < cnt;i++)
    {
        sleep(1);
        puts("running thread");

    }
    return NULL;
}

```

执行结果：



子线程只输出一次，主线程中间挂起2秒，接着子线程在循环中挂起2秒，刚好挂起第2秒的时候，主线程结束，子线程也跟着结束执行。

线程与共享 线程间共享全局数据

- 线程默认共享数据段、代码段等地址空间，经常使用的是全局变量。而进程不共享全局变量，只能借助`mmap`。

`pthread_exit`函数 将单个线程退出

- `void pthread_exit(void *retval)`;参数：`retval`表示线程退出状态，通常传`NULL`

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void *tfn(void *arg)
{
    int i;
    i = *((int *)arg); //强转。
    if (i == 2)
        pthread_exit(NULL);
    sleep(i);    //通过i来区别每个线程

    printf("I'm %dth thread, Thread_ID = %lu\n", i+1, pthread_self());
    return NULL;
}
int main(int argc, char *argv[])
{
    int n , i;
    pthread_t tid;

    if (argc == 2)
        n = atoi(argv[1]);

    for (i = 0; i < n; i++)
    {
        pthread_create(&tid, NULL, tfn, (void *)i);  //将i转换为指针，在tfn中再强转回整形。
    }

```

```
30      sleep(n);
31      printf("I am main, I'm a thread!\n" "main_thread_ID = %lu\n",
    pthread_self());
32
33      return 0;
34  }
```

线程中禁止使用exit函数，会导致进程内所有的线程全部退出。

- 在不添加sleep控制指定线程退出的情况下，`pthread_create`在循环的过程中，几乎在瞬间创建了5个进程，但是只有第一个线程或者其他线程有机会输出，这取决于内核的调度，**这也解释了，如果我们在创建了一个线程之后，如果不停顿一下，会导致错误输出的问题。**这个时候如果有线程执行`exit`的话，将会将整个进程退出，那么肯定所有的线程就会退出。
- 多线程中，应尽量少用，或者不使用`exit`函数，取而代之使用`pthread_exit`函数，将单个线程退出。
- 另外，`pthread_exit`或者`return`返回的指针所指向的内存单元必须是全局的或者是使用`malloc`分配的，不能在线程函数的栈上分配，因为当其他线程得到这个返回指针的时候，线程函数已经退出了，那么所指向的内存单元如果是在栈上的话，自然就会被释放掉。

`pthread_join`函数，阻塞等待线程退出，并且可以获取线程退出时的状态，对应进程中`waitpid()`函数。

- `int pthread_join(pthread_t thread,void **retval)`，执行成功的话函数返回0，否则返回错误号。
- 参数1，thread：线程ID；retval：存储线程结束状态；

`pthread_cancel`函数，杀死(取消)线程，对应进程中过的kill函数。

- 线程的取消并不是实时的，而是有一定的延时。并且需要等待线程到达某一个取消点（检查点，***存档***）。

`pthread_datach`函数，实现线程分离

- 线程分离状态：指定该状态，线程主动与主控线程断开关系。线程结束后，其退出状态不由其他线程获取，而是自己自动释放。网络、多线程服务器经常使用。因为不能说因为某一个线程而耽误主控线程。
- 线程被分离之后，就不能使用`pthread_join`等待其终止状态；
- 进程中如果有该机制的话，就不会产生僵尸进程。僵尸进程的产生主要是由于进程死后，大部分资源被释放，但是一些子进程无法自己释放，导致内核认为该进程仍存在

6. 调度算法

```
❯ ./pthread1
I'm thread,Thread ID = 140582196573888
running thread
I'm main,my process ID is 25727
I'm main,140582196578112
end of main
 ~/Codes/Linux
❯
```

6.1 进程调度算法

进程调度算法也称 CPU 调度算法，毕竟进程是由 CPU 调度的。

当 CPU 空闲时，操作系统就选择内存中的某个「就绪状态」的进程，并给其分配 CPU。

什么时候会发生 CPU 调度呢？通常有以下情况：

1. ***当进程从运行状态转到等待状态***；
2. 当进程从运行状态转到就绪状态；
3. 当进程从等待状态转到就绪状态；
4. ***当进程从运行状态转到终止状态***；

其中发生在 1 和 4 两种情况下的调度称为「非抢占式调度」，2 和 3 两种情况下发生的调度称为「抢占式调度」。

非抢占式的意思就是，当进程正在运行时，它就会一直运行，直到该进程完成或发生某个事件而被阻塞时，才会把 CPU 让给其他进程。

而抢占式调度，顾名思义就是进程正在运行的时，可以被打断，使其把 CPU 让给其他进程。***那抢占的原则一般有三种，分别是时间片原则、优先权原则、短作业优先原则。***

你可能会好奇为什么第 3 种情况也会发生 CPU 调度呢？假设有一个进程是处于等待状态的，**但是它的优先级比较高**，因所以说在调度的时候就会优先考虑它，如果该进程等待的事件发生了，它就会转到就绪状态，一旦它转到就绪状态，如果我们的调度算法是以优先级来进行调度的，那么它就会立马抢占正在运行的进程，所以这个时候就会发生 CPU 调度。

那第 2 种状态通常是时间片到的情况，因为时间片到了就会发生中断，于是就会抢占正在运行的进程，从而占用 CPU。

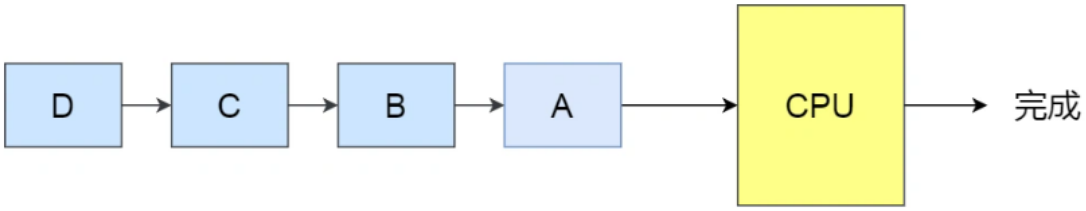
调度算法影响的是等待时间（进程在就绪队列中等待调度的时间总和），而不能影响进程真在使用 CPU 的时间和 I/O 时间。我们选择调度算法，最关键的问题就在于我们怎么才可以使得各个进程等待时间总和达到最小值；

接下来说说常见的调度算法：

- 先来先服务调度算法（First Come First Severd,FCFS）
- 最短作业优先调度算法（Shortest Job First，SJF）
- 高响应比优先调度算法（Highest Response Ratio Next，HRRN）
- 时间片轮转调度算法（Round Robin，RR）
- 最高优先级调度算法（Highest Priority First，HPF）
- 多级反馈队列调度算法（Multilevel Feedback Queue）

6.1.2 先来先服务调度算法

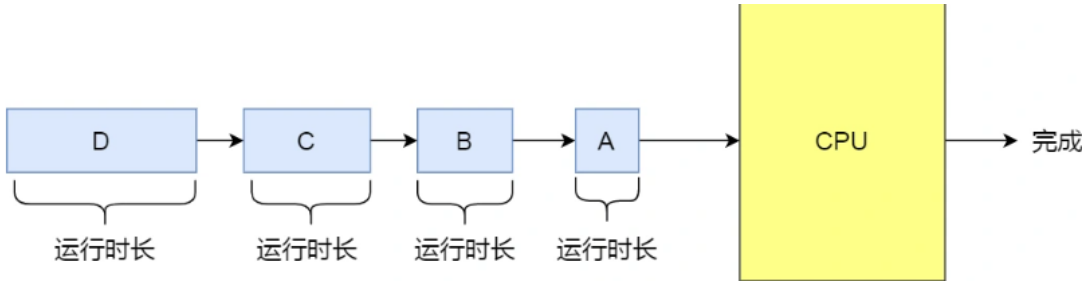
最简单的调度算法，就是先来先得，这属于非抢占式算法。



*问题所在：*当一个比较大的程序运行的时候，后面的那些短的进程的等待时间就会非常的漫长，这十分不利于短作业。

6.1.3 最短作业优先调度算法

哪一个进程短，则先执行哪一个，这有助于提高系统的吞吐量。



*问题：*对于长的进程来讲的话明显不友好。

6.1.4 高相应比优先调度算法

前面的「先来先服务调度算法」和「最短作业优先调度算法」都没有很好的权衡短作业和长作业。

那么，**高响应比优先 （*Highest Response Ratio Next, HRRN*）调度算法**主要是权衡了短作业和长作业。

每次进行进程调度时，先计算「响应比优先级」，然后把「响应比优先级」最高的进程投入运行，「响应比优先级」的计算公式：

$$优先权 = \frac{等待时间 + 要求服务时间}{要求服务时间}$$

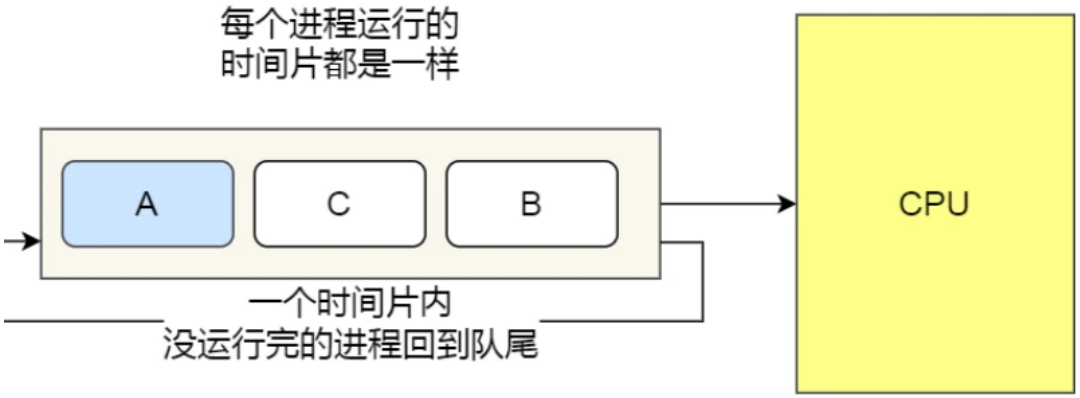
从上面的公式，可以发现：

- 如果两个进程的「等待时间」相同时，「要求的服务时间」越短，「响应比」就越高，这样短作业的进程容易被选中运行；
- 如果两个进程「要求的服务时间」相同时，「等待时间」越长，「响应比」就越高，这就兼顾到了长作业进程，因为进程的响应比可以随时间等待的增加而提高，当其等待时间足够长时，其响应比便可以升到很高，从而获得运行的机会；

6.1.5 时间片轮转调度算法

最古老、最简单、最公平、使用最广的算法就是时间片轮转调度算法。

意思就是给每一个进程都分配指定的时间片，当该时间片结束的时候，将该没有运行完的进程放在调度的进程队列末尾:



注意:如果该进程在时间片内阻塞的话或者已经结束的话,CPU会立即进行切换.

另外时间片的长度也是一个很关键的问题:

- 如果时间片设置的很短,就会导致一直在进行上下文切换,这就导致本末倒置,极大的降低了CPU的效率;
- 如果时间片设置的很长的话,也有可能引起对短作业进程的响应时间很长;
- 通常时间片设置为20ms~50ms.

6.1.6 最高优先级调度算法

上面的时间片轮转算法已经很公平了,使得所有的进程都是一样的地位,谁也不偏袒谁,大家的运行时间都是一样的.

这显然不显示,因为我们有很多系统级别的程序需要优先调度,所以说我们引进最高优先级调度算法.

优先级又分为两种:

- 静态优先级,就是在创建进程的时候,就已经确定了该进程的优先级,然后整个运行时间优先级都不会变化.
- 动态优先级,根据进程的动态变化设置其优先级,比如说***如果进程运行时间增加,则降低其优先级,如果进程等待时间增加,则升高其优先级.***

该算法也有两种处理优先级高的方法，非抢占式和抢占式：

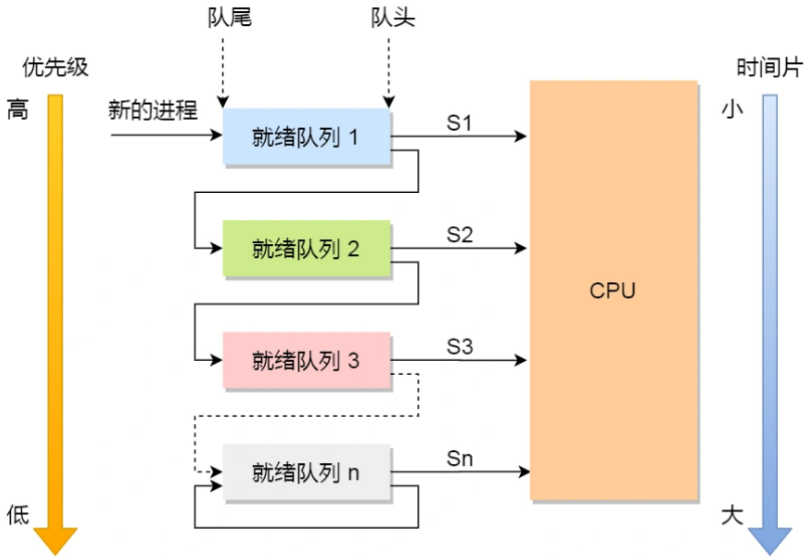
- 非抢占式：当就绪队列中出现优先级高的进程，运行完当前进程，再选择优先级高的进程，这个就是就算你有限度很高也不能去抢别人进程的CPU。
- 抢占式：当就绪队列中出现优先级高的进程，当前进程挂起，调度优先级高的进程运行。

6.1.7 多级反馈队列调度算法

该算法是时间片轮转算法和最高优先级算法的结合。

顾名思义：

- 「多级」表示有多个队列，每个队列优先级从高到低，同时优先级越高时间片越短。
- 「反馈」表示如果有新的进程加入优先级高的队列时，立刻停止当前正在运行的进程，转而去运行优先级高的队列；



（时间片：S1 < S2 < S3）

来看看，它是如何工作的：

- 设置了多个队列，赋予每个队列不同的优先级，每个**队列优先级从高到低**，同时**优先级越高时间片越短**；
- 新的进程会被放入到第一级队列的末尾，按先来先服务的原则排队等待被调度，如果在第一级队列规定的时间片没运行完成，则将其转入到第二级队列的末尾，以此类推，直至完成；
- 当较高优先级的队列为空，才调度较低优先级的队列中的进程运行。如果进程运行时，有新进程进入较高优先级的队列，则停止当前运行的进程并将其移入到原队列末尾，接着让较高优先级的进程运行；

可以发现，对于短作业可能可以在第一级队列很快被处理完。对于长作业，如果在第一级队列处理不完，可以移入下次队列等待被执行，虽然等待的时间变长了，但是运行时间也会更长了，所以该算法很好的**兼顾了长短作业，同时有较好的响应时间。**