Linux 进程创建及进程间通信

一、 进程创建相关的系统调用

1.fork（ ）

创建一个新的子进程。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。

系统调用格式： pid=fork( )

参数定义：int fork( )

fork( )返回值意义如下：

0：在子进程中，pid 变量保存的 fork( )返回值为 0，表示当前进程是子进程。

>0：在父进程中，pid 变量保存的 fork( )返回值为子进程的 id 值（进程唯一标识符）。

-1：创建失败。

如果 fork( )调用成功，它向父进程返回子进程的 PID，并向子进程返回 0，即 fork( )被调用了一次，但返回了两次。此时 OS 在内存中建立一个新进程，所建的新进程是调用 fork( )父进程（parent process）的副本，称为子进程（child process）。子进程继承了父进程的许多特性，并具有与父进程完全相同的用户级上下文。父进程与子进程并发执行。

内核为 fork( )完成以下操作：

（1）为新进程分配一个进程表项和进程标识符

进入 fork( )后，内核检查系统是否有足够的资源来建立一个新进程。若资源不足，则 fork( )系统调用失败；否则，内核为新进程分配一进程表项和唯一的进程标识符。

（2）检查同时运行的进程数目

超过预先规定的最大数目时，fork( )系统调用失败。

（3）拷贝进程表项中的数据

将父进程的当前目录和所有已打开的数据拷贝到子进程表项中，并置进程的状态为“创建”状态。

（4）子进程继承父进程的所有文件

对父进程当前目录和所有已打开的文件表项中的引用计数加 1。

（5）为子进程创建进程上、下文

进程创建结束，设子进程状态为“内存中就绪”并返回子进程的标识符。

（6）子进程执行

虽然父进程与子进程程序完全相同，但每个进程都有自己的程序计数器 PC(注意子进程的 PC 开始位置)，然后根据 pid 变量保存的 fork(　)返回值的不同，执行了不同的分支语句。

2.getpid()

取得目前进程的识别码（进程 ID），许多程序利用取到的此值来建立临时文件，以避免临时文件相同带来的问题。

系统调用格式：　　int getpid()

例如：#include<unistd.h>

　　main()

　{printf(“pid=%d\n”,getpid());}

3.getppid

取得目前进程的父进程识别码。

系统调用格式：　　int getppid()

例如：#include<unistd.h>

　　main()

　{printf(“My parent’s pid=%d\n”,getppid());}

二、 进程创建程序实例

进程创建程序示例：

```
#include<stdio.h>
    main()
    { int   p1;
      while((p1=fork())= = -1);
      if(p1= =0)
          printf("This is a child process.");          /*在子进程中*/
      else                                             /*在父进程中*/
          {
            printf("This is a parent process.");
          }
     }
```

三、进程同步控制所涉及的系统调用

在 Unix/Linux 中 fork( )是一个非常有用的系统调用，但在 Unix/Linux 中建立进程除了 fork( )之外，也可用与 fork( ) 配合使用的 exec( )。

1. exec( ) 系列

系统调用 exec( )系列，也可用于新程序的运行。fork( )只是将父进程的用户级上下文拷贝到新进程中，而 exec( )系列可以将一个可执行的二进制文件覆盖在新进程的用户级上下文的存储空间上，以更改新进程的用户级上下文。exec( )系列中的系统调用都完成相同的功能，它们把一个新程序装入内存，来改变调用进程的执行代码，从而形成新进程。如果 exec( )调用成功，调用进程将被覆盖，然后从新程序的入口开始执行，这样就产生了一个新进程，新进程的进程标识符 id 与调用进程相同。exec( )没有建立一个与调用进程并发的子进程，而是用新进程取代了原来进程。所以 exec( )调用成功后，没有任何数据返回，这与 fork( )不同。

在 Linux 中，并不存在 exec( )函数，exec 指的是一组函数。exec( )系列系统调用在 UNIX 系统库 unistd.h 中，共有 execl、execlp、execle、execv、execvp 和 execve 六个，其基本功能相同，只是以不同的方式来给出参数。其中，只有 execve 是真正意义上的系统调用，其它都是在此基础上经过包装的库函数。

一种是直接给出参数的指针，如：

```
int    execl(path,arg0[,arg1,...argn],0);
char    *path,*arg0,*arg1,...,*argn;
```

另一种是给出指向参数表的指针，如：

```
int execv(path,argv);
char *path,*argv[ ];
```

具体使用可参考有关书。

2. exec( )和 fork( )联合使用

系统调用 exec 和 fork( )联合使用能为程序开发提供有力支持。用 fork( )建立子进程，然后在子进程中使用 exec( )，这样就实现了父进程与一个与它完全不同子进程的并发执行。

一般，wait、exec 联合使用的模型为：

```
int status;
    .....
if (fork( )= =0)
{
    .....;
    execl(...);
    .....;
}
 wait(&status);
```

3.wait（ ）

wait( )函数在/usr/include/sys/wait.h 文件中定义。等待子进程运行结束。如果子进程没有完成，父进程一直等待。wait( )将调用进程挂起，直至其子进程因暂停或终止而发来软中断信号为止。如果在调用 wait（）时子进程已经结束，则 wait（）会立即返回子进程结束状态值。子进程的结束状态值会由参数 status 返回，而子进程的进程 ID 也会一块儿返回。如果不在意结束状态值，则参数 status 可以设成 NULL。子进程的结束状态值请参考后面“四、软中断通信函数”中的 waitpid（）。

系统调用格式：

int　wait(status)

int　*status;

其中，status 是用户空间的地址。它的低 8 位反应子进程状态，为 0 表示子进程正常结束，非 0 则表示出现了各种各样的问题；高 8 位则带回了 exit( )的返回值。exit( )返回值由系统给出。

需包含的头文件为#include <unistd.h>

内核对 wait( )作以下处理：

（1）首先查找调用进程是否有子进程，若无，则返回出错码；

（2）若找到一个处于“僵死状态”的子进程，则将子进程的执行时间加到父进程的执行时间上，并释放子进程的进程表项；

（3）若未找到处于“僵死状态”的子进程，则调用进程便在可被中断的优先级上睡眠，等待其子进程发来软中断信号时被唤醒。

4.exit（ ）

终止进程的执行。

系统调用格式：

void exit(status)

int status;

其中，status 是返回给父进程的一个整数，以备查看。

需包含的头文件为#include <unistd.h>

为了及时回收进程所占用的资源并减少父进程的干预，Unix/Linux 利用 exit( )来实现进程的自我终止，通常父进程在创建子进程时，应在进程的末尾安排一条 exit( )，使子进程自我终止。exit(0)表示进程正常终止，exit(1)表示进程运行有错，异常终止。

如果调用进程在执行 exit( )时，其父进程正在等待它的终止，则父进程可立即得到其返回的整数。内核须为 exit( )完成以下操作：

（1）关闭软中断

（2）回收资源

（3）写记帐信息

（4）置进程为“僵死状态”

5. sleep（ ）

使进程挂起指定的时间，直至指定时间（由 seconds 表示）用完或者收到信号。

系统调用格式：

unsigned int sleep(unsigned int seconds);

需包含的头文件为#include <unistd.h>

四、进程互斥所涉及的系统调用

lockf(files,function,size)

作用：对指定文件的指定区域（由 size 指示）进行加锁或解锁，以实现进程的同步与互斥。

本函数的头文件为

#include <unistd.h>

参数定义：

int　lockf(files,function,size)

int　files,function;

long　size;

其中：files 是文件描述符；function 是锁定和解锁：1 表示锁定，0 表示解锁。size 是锁定或解锁的字节数，为 0，表示从文件的当前位置到文件尾。

五、Linux 进程间通信

进程间通信就是在不同进程之间传播或交换信息，进程间通信的目的在于实现数据传输、数据共享、事件通知、资源共享和进程控制等。Linux 进程间通信（IPC）主要由早期 Unix 进程间通信、基于 System V 进程间通信、基于 Socket 进程间通信和 POSIX 进程间通信几部分发展而来。Unix 进程间通信的方式包括管道、FIFO、信号；System V 进程间通信方式包括 System V 消息队列、System V 信号灯、System V 共享主存等，通信进程局限在单个计算机内；基于 Socket 进程间通信则越过了该界限，能实现多机间进程的通信；POSIX 进程间通信包括 POSIX 消息队列、POSIX 信号灯、POSIX 共享主存。

现在 Linux 使用的进程间通信方式主要有以下几种。

1.管道（pipe）和有名管道（named pipe）

在 Linux 系统中，管道是一种很重要的通信方式，它将一个程序的输出直接连接到一个程序的输入。从本质上说，管道也是一种文件，但它又和一般的文件有所不同，管道具有以下特点：

- 管道是半双工的，数据只能单向流动；需要双方通信时，需要建立起两个管道。
- 管道只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）。
- 管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，不属于某种文件系统，而是单独构成一种文件系统，并且只存在于主存中。

- 进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

注意：从管道读数据是一次性操作，数据一旦被读，它就从管道中被抛弃，释放空间以便写更多的数据。

2.信号（signal）

信号是比较复杂的通信方式，类似于 Windows 下的消息，用于通知接受进程有某种事件发生。除了用于进程间通信外，进程还可以发送信号给进程本身。Linux 除了支持 Unix 早期信号语义函数 signal 外，还支持语义符合 Posix.1 标准的信号函数 sigaction。

信号机制是 Unix 系统中最为古老的进程之间的通信机制，用于在一个或多个进程间传递异步信号。信号是对中断机制的一种模拟，故在早期的 Unix 版本中又把它称为软中断。很多条件可以产生一个信号，如用户按某些终端键、无效的存储访问、管道的读进程终止后有进程写此管道、进程所设置的时间超时等。

当某个信号出现时，系统按照下列三种方式中的一种进行处理。

- 忽略信号

大多数信号都可使用这种方式进行处理，但有两种信号绝不能被忽略：SIGKILL 和 SIGSTOP，它们向超级用户提供一种使进程终止或停止的可靠方法。另外，如果忽略某些由硬件异常产生的信号（例如非法存储访问或除以 0），则进程的行为是未定义的。

- 捕捉信号

通知内核在某种信号发生时调用一个用户函数。在用户函数中，可执行用户希望对这种事件进行的处理。如捕捉到 SIGCHLD 信号，则表示子进程已经终止，可以调用 waitpid（）以取得该子进程的进程 ID 以及它的终止状态。

- 执行系统默认动作

对大多数信号的系统默认动作是终止该进程。每一个信号都有一个默认动作，它是当进程没有给这个信号指定处理程序时，内核对信号的处理。有 5 种默认的动作：异常终止（abort）、退出（exit）、忽略（ignore）、停止（stop）和继续（continue）。

3. 消息（message）队列

消息队列也叫报文队列，是消息的链接表。目前主要有两种类型的消息队列：POSIX 消息队列和系统 V 消息队列，系统 V 消息队列目前被大量使用。考虑到程序的可移植性，新开发的应用程序应尽量使用 POSIX 消息队列。

消息队列用于运行同一台机器上的进程间通信，和管道很相似，有足够权限的进程可以向消息队列中发送某种类型的消息，被赋予读权限的进程可以从消息队列中读取某种类型的消息。但消息主存可以根据需要自行定义，从而使消息队列克服了信号承载信息量少、管道只能承载无格式字节流以及缓冲区大小受限等缺点，在实际编程中应用较广。可以用流管道或套接口的方式取代它。

4. 共享主存

共享主存是在系统内核分配的一块缓冲区，多个进程都可以访问该缓冲区。采用共享主存通信的一个显而易见的好处是效率高，因为进程可以直接读写主存，而不需要复制任何数据，避免了在内核空间与用户空间的切换。对于像管道和消息队列等通信方式，则需要在内核空间和用户空间进行四次数据复制，而共享主存只需要复制两次数据：从输入文件到共享主存区和从共享主存区到输出文件。实际上，进程之间在共享主存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享主存区域；而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享主存中，并没有写回文件。共享主存中的内容往往是在解除映射时才写回文件的。因此，采用共享主存的通信方式效率是非常高的。

共享主存类似于 Windows 环境编程中的主存影像文件，在 Linux 系统中针

对其它通信机制运行效率较低而设计，多用于存储应用程序的配置信息。这个机制的不利方面是其同步和协议都不受程序员控制，必须确保将句柄传递给子进程和线程，因此往往与其它通信机制，如信号量结合使用。

5. 信号量

信号量及信号量上的操作是一种解决同步、互斥问题的较通用的方法，并在很多操作系统中得以实现，Linux 改进并实现了这种机制。信号量也称为信号灯，用来协调不同进程间的数据对象，最主要的应用就是共享主存方式的进程间通信。本质上，信号量是一个计数器，用来记录对某个资源（如共享主存）的存取状况。

维护信号量状态的是 Linux 内核操作系统而不是用户进程。信号量主要提供对进程间共享资源访问控制的手段，用来保护共享资源。除了用于访问控制外，还可以用于进程间及同一进程不同线程间的进程同步。信号量有以下两种类型：

- 二值信号量

最简单的信号量形式，它的值只能取 0 或 1，类似于互斥锁，但二者关注的内容不同。信号量强调共享资源，只要共享资源可用，其它进程同样可以修改信号量的值；互斥更强调进程，占用资源的进程使用完资源后，必须由进程本身来解锁。

- 计算信号量。

信号量的值可以取任意非负值（受内核本身的约束）。

6. 套接口（socket）

套接口也称为套接字，可用于不同机器之间的进程通信。一个套接口可以看作进程间通信的端点（endpoint），每个套接口的名字都是唯一的，其它进程可以发现、连接并且与之通信。通信域用来说明套接口通信用的协议，不同的通信域有不同的通信协议以及套接口的地址结构等，因此，创建一个套接口时，要指明它的通信域。比较常见的是 Unix 域套接口（采用套接口机制实现单机内的进程

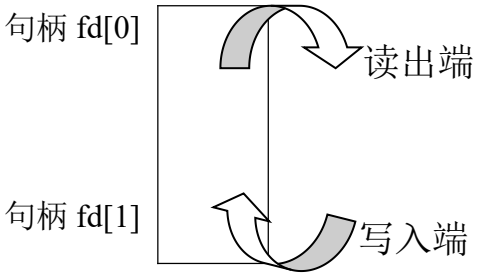
间通信）及网际通信域。

套接口起初由 Unix 系统的 BSD 分支开发，由于其功能强大、接口友好，现在已经移植到大多数操作系统中，成为事实上的网络编程接口。套接口是网络编程中一个非常重要的概念，Linux 以文件的形式实现套接口，与套接口相应的文件属于 sockfs 特殊文件系统，创建一个套接口就是在 sockfs 中创建一个特殊文件，并建立起为实现套接口功能的相关数据结构。换句话说，对每一个新创建的 BSD 套接口，Linux 内核都将在 sockfs 特殊文件系统中创建一个新的 inode。

六、管道

1. 什么是管道

管道是一种遵循先进先出原则的数据结构，先进入管道的数据，也能先从管道中读出。所谓管道，是指能够连接一个写进程和一个读进程的、并允许它们以生产者—消费者方式进行通信的一个共享文件，又称为 pipe 文件。由写进程从管道的写入端（句柄 1）将数据写入管道，而读进程则从管道的读出端（句柄 0）读出数据。数据一旦读取后，就会在管道中自动删除。



管道通信是一种常用的单向进程间通信机制，它以管道数据结构作为内部数据存储方式，以文件系统作为数据存储媒体，将一个写进程和一个读进程连接在一起，实现两个进程间的通信。管道作为一临界资源，使用过程中父子进程之间除了需要写同步以外，在对管道进行读写操作时还需要互斥进入。

2. 管道的类型

Linux 系统中有两种管道，分别是无名管道和有名管道，在这里只介绍无名管道。管道中的数据格式是字符流。

无名管道为创建管道的进程及其子孙提供传递消息的信道，逻辑上它是由操

作系统在主存中创建的临时文件实现的，被看作管道文件，物理上则由文件系统的高速缓冲区构成。由于涉及文件系统，无名管道文件描述符只能供创建管道的进程及其子孙进程共享使用，因为子进程能够继承父进程打开的所有文件，所以能够继承父进程所创建的无名管道文件。因此，进程间使用管道的通信方式仅限于同一家族的进程之间使用。

创建一个无名管道的函数是 int pipe(int fd[2]),它同时分配两个文件描述符，一个用于读，另一个用于写，它们是管道的两端。从一端写入的内容，可以从另一端读出。显然，这对于同一个进程来说，没有任何用处。但在创建新进程时，子进程继承父进程的所有文件描述符，子进程同时也拥有 pipe（）所创建的两个文件描述符。因此，如果父进程写入管道的一端，子进程可以从管道的另一端读出。反之亦然，这样就实现了父子进程、兄弟进程之间的通信。

管道两端可分别用描述字 fd[0]和 fd[1]描述。fd[0]只能用于读，称为管道读端；fd[1]只能用于写，称为管道写端。一般文件的 I/O 函数都可用于管道，如 close()、read()、write()等。

从管道中读取数据的规则为：

- 如果管道的写端不存在，则认为已经读到了数据的末尾，读函数返回的读出字节数为 0。
- 当管道的写端存在时，如果请求的字节数大于 PIPE_BUF，则返回管道中现有的数据字节数；如果请求的字节数目不大于 PIPE_BUF，若管道中数据量小于请求的数据量，则返回管道中现有的数据字节数，否则返回请求的字节数（管道中数据量不小于请求的数据量）。

向管道中写入数据的规则为：

向管道中写入数据时，Linux 将不保证写入数据的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读走管道缓冲区中的数据，那么写操作将一直阻塞。要注意，只有在管道的读端存在时，向管道中写

入数据才有意义。否则，向管道中写入数据的进程将收到内核传来的 SIFPIP 信号，应用程序可以处理该信号，也可以忽略该信号（默认动作是应用程序终止）。

七、管道通信函数

1.pipe( )

作用：

建立一无名管道。调用成功时，返回 0；错误时返回-1。

系统调用格式：

pipe(filedes)

参数定义

int　pipe(filedes);

int　filedes[2];

其中，filedes[1]是写入端，filedes[0]是读出端。

该函数使用头文件如下：

#include <unistd.h>

#inlcude <signal.h>

#include <stdio.h>

2.read( )

系统调用格式：

read(fd,buf,nbyte)

功能：从 fd 所指示的文件中读出 nbyte 个字节的数据，并将它们送至由指针 buf 所指示的缓冲区中。如该文件被加锁，等待，直到锁打开为止。

参数定义：

int　read(fd,buf,nbyte);

int　fd;

char *buf;

unsigned　nbyte;

3.write( )

系统调用格式

write(fd,buf,nbyte)

功能：把 nbyte 个字节的数据，从 buf 所指向的缓冲区写到由 fd 所指向的文件中。如文件加锁，暂停写入，直至开锁。

参数定义同 read( )。

注意：管道为一临界资源，使用过程中父子进程之间除了需要读写同步以外，在对管道进行读写操作时还需要互斥进入，可以使用对文件上锁和开锁的系统调用 lockf(files,function,size)。