**进程间通信(IPC)**

Linux下的进程通信手段基本上是从UNIX平台上的进程通信手段继承而来的。对UNIX发展做出重大贡献的两大主力AT&T的贝尔实验室和BSD(加州大学伯克利分校的伯克利软件发布中心)在进程间通信的侧重点有所不同。前者对UNIX早期的进程间通信手段进行了系统的改进和扩展，形成了“System V IPC”对象，其通信进程主要局限在单个计算机内；后者则跳过该限制，形成了基于套接字(Socket)的进程间通信机制。而Linux则把二者的优势都继承了下来。

常用的进程间通信方式包括以下七种：

1. 传统：无名管道(pipe)，有名管道(fifo)，信号(signal)；
2. System V IPC：共享内存(share memory)，消息队列(message queue)，信号量(semaphore)；
3. BSD：套接字(Socket)。

进程间通信方式比较：

1. pipe：具有亲缘关系的进程间，单工，数据在内存中；
2. fifo：可用于任意进程间，双工，有文件名，数据在内存中；
3. signal：唯一的异步通信方式；
4. share memory：效率最高(可直接访问内存)，需要同步和互斥机制；
5. message queue：常用于CS模式中，按消息类型访问；
6. semaphore：配合共享内存使用，用以实现同步和互斥。
7. **管道通信**

管道是Linux中进程间通信的一种方式，它把一个程序的输出直接连接到另一个程序的输入。Linux的管道主要包括两种：pipe和fifo。

无名管道：

1. 它只能用于具有亲缘关系的进程间通信(父子或兄弟)；
2. 它是一个半双工的通信模式，具有固定的读端和写端；
3. 管道可以看成是一种特殊的文件，对于它的读写要使用文件IO(read()和write())。管道不是普通的文件，它不属于任何文件系统，并且只存在于内存中。
4. 管道是基于文件描述符的通信方式，当一个管道建立时，它会创建两个文件描述符fd[0]和fd[1]，其中fd[0]固定用于读管道，而fd[1]固定用于写管道，这样就构成了一个半双工的通道。

无名管道的读写需要注意以下几点：

1. 当写端存在时：若管道中有数据，则读端取走；若管道中无数据，则读端会阻塞直至管道中有数据为止。当写端不存在时：若管道中有数据，则读端取走；若管道中无数据，则读端read返回值为0。
2. 当读端存在时：若管道未满：则写端写数据；若管道满了(最大容量为65536个字节)，则写端阻塞。当读端不存在(指的是读写进程中的fd[0]都关闭)时，写端被信号(13 SIGPIPE)杀死。(只有在读端存在时，向管道写入数据才有意义)
3. 为了保证严谨性，在读进程中，我们要关闭fd[1]；在写进程中，我们要关闭fd[0]。

pipe()：创建无名管道

|  |  |
| --- | --- |
| 头文件 | #include <unistd.h> |
| 原型 | int pipe(int pipefd[2]); |
| 参数 | fd是包含两个文件描述符的数组 |
| 返回值 | 成功：0 |
| 失败：-1(设置errno) |

注意：

1. 创建无名管道成功之后，就打开了两个文件描述符，我们可以通过fd[0]和fd[1]来进行无名管道的读(read)、写(write)和关闭(close)操作；
2. 我们可以利用ls /proc/pid/fd –l指令来查看一个进程所打开的文件描述符的情况。

有名管道：

无名管道只能用于具有亲缘关系的进程之间，这就限制了无名管道的使用范围。有名管道可以使互不相关的两个进程互相通信。有名管道可以通过路径名来指出，并且在文件系统中可见。有名管道遵循先进先出的原则。和无名管道一样，对有名管道的读写也只能使用文件IO，且不支持lseek()操作。

有名管道的读端和写端的特点和无名管道基本一致。

mkfifo()：创建有名管道

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <sys/types.h>  #include <sys/stat.h> | |
| 原型 | int mkfifo(const char \*pathname, mode\_t mode); | |
| 参数 | pathname | 要创建的管道 |
| mode | 指定创建管道的访问权限，一般用八进制表示(0666) |
| 返回值 | 成功 | 0 |
| 失败 | -1(并设置errno) |

要注意进行出错判断：

|  |  |
| --- | --- |
| errno=EEXIST | 要创建的有名管道pathname存在 |
| errno=ENOSPC | 文件系统的剩余空间不足 |
| errno=EROFS | pathname指定的文件存在于只读文件系统中 |
| errno=ENOENT | pathname包含的目录不存在 |
| errno=ENAMETOOLONG | pathname的路径名称太长 |
| errno=EACCESS | 用户对pathname指定的目录无可执行权限  (在目录中新建或删除文件，须对该目录具有写权限) |

在创建完有名管道之后，可以通过open()指定有名管道的打开方式，进而利用read()和write()进行读写操作。在前面无名管道的读写操作中，我们知道read()和write()会自动阻塞，这里同样适用。特别地，open()函数也会阻塞，即通信双方的进程必须都运行了，open()函数才会成功，若只运行一端，则open()函数会阻塞。

open()、read()、write()函数都是自动阻塞机制的。

1. **信号通信**

信号是在软件层次上对中断机制的一种模拟。在原理上，一个进程接收到一个信号与处理器收到一个中断请求可以说是一样的。

信号是异步的，一个进程不必做任何操作来等待信号的到达，事实上，进程也不知道信号什么时候到达。

信号可以直接进行用户空间进程和内核进程之间的交互，内核进程也可以利用它来通知用户空间进程发生了什么系统事件。

它可以在任何时候发给某一进程，而无需知道该进程的状态。

1. 如果该进程当前未处于执行状态，则该信号就由内核保存起来，直到该进程恢复运行再传递给它为止；
2. 如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传递给进程。

信号是进程间通信机制中唯一的异步通信机制。

信号事件的发生有两个来源：

1. 硬件来源：按下键盘按钮或其他硬件故障；
2. 软件来源：发送信号的系统函数kill()、alarm()、raise()等；非法操作。

用户进程对信号的响应方式：

1. 忽略信号：对信号不做任何处理，但有两个信号不能忽略：SIGKILL和SIGSTOP；
2. 捕捉信号：定义信号处理函数，当信号发生时，执行相应的操作；
3. 执行默认操作：Linux对每种信号都规定了默认操作。

常见信号的含义及其默认操作：

|  |  |  |
| --- | --- | --- |
| 信号名 | 含义 | 默认操作 |
| SIGINT(2) | 该信号在用户终端输入INTR字符(ctrl+c)时发出，终端驱动程序发送此信号并送到前台进程中的每一个进程 | 终止 |
| SIGQUIT(3) | 类似于SIGINT，但由QUIT字符控制(ctrl+\) | 终止 |
| SIGILL(4) | 该信号在一个进程试图执行非法指令时发出(段错误) | 终止 |
| SIGFPE(8) | 该信号在发生致命算数运算错误时发出(除数为0) | 终止 |
| SIGKILL(9) | 该信号用来立即结束程序的运行，并且不能被阻塞、处理和忽略 | 终止 |
| SIGUSR1(10)  SIGUSR2(12) | 常用于进程间的通信，signal(SIGUSR1, handler);  void hander(int sign); | 终止 |
| SIGALRM(14) | 该信号当一个定时器到时时发出 | 终止 |
| SIGCHLD(17) | 子进程改变状态时，父进程会受到这个信号 | 忽略 |
| SIGCONT(18) | 让一个处于T状态的进程恢复运行，但运行于后台 | 恢复进程 |
| SIGSTOP(19) | 暂停一个进程，且不能被阻塞、处理和忽略 | 暂停进程 |
| SIGTSTP(20) | 该信号用于交互停止进程，用户在输入SUSP字符时发出(ctrl+z) | 停止进程 |

信号的处理包括信号的发送、捕获及处理，它们有各自相对应的函数：

1. 信号的发送kill()和raise()：

|  |  |
| --- | --- |
| 头文件 | #include <sys/types.h> #include <signal.h> |
| 原型 | int kill(pid\_t pid, int sig); |
| 参数 | pid：要发送信号的进程号 |
| sig：要发送的信号 |
| 返回值 | 成功：0 |
| 失败：-1 |

实际上，kill系统命令只是kill()函数的一个用户接口。

|  |  |
| --- | --- |
| 头文件 | #include <sys/types.h> #include <signal.h> |
| 原型 | int raise(int sig); |
| 参数 | sig：要发送的信号 |
| 返回值 | 成功：0 |
| 失败：-1 |

注意：raise()函数只能向调用进程自身发信号。

1. 信号的捕捉alarm()和pause()：

alarm()也称为闹钟函数，它可以在一个进程中设置一个定时器，当定时器指定的时间到时，它就向进程发送SIGALRM信号。要注意的是，一个进程只能有一个闹钟时间，如果在调用alarm()之前已将设置过闹钟时间，则任何以前的闹钟时间都被新值所代替。

|  |  |
| --- | --- |
| 头文件 | #include <unistd.h> |
| 原型 | unsigned int alarm(unsigned int seconds); |
| 参数 | seconds：定时秒数 |
| 返回值 | 成功：如果调用此alarm()之前已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回0。 |
| 失败：-1 |

pause()函数用于将进程挂起(S)直至捕捉到信号为止。

|  |  |
| --- | --- |
| 头文件 | #include <unistd.h> |
| 原型 | int pause(void); |
| 返回值 | -1；errno被置为EINTR。 |

1. 信号的处理signal()：

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <signal.h> | |
| 原型 | typedef void (\* sighandler\_t) (int);  sighandler\_t signal(int signum, sighandler\_t handler); | |
| 参数 | signum | 指定要处理的信号代码eg：SIGUSR1 |
| handler | SIG\_IGN：忽略该信号 |
| SIG\_DFL：采用系统默认方式处理信号 |
| 自定义的信号处理函数指针 |
| 返回值 | 成功 | 之前的信号处理方式handler |
| 失败 | SIG\_ERR |

1. **共享内存**

可以说，共享内存是一种最为高效的进程间通信方式，因为进程可以直接读写内存不需要任何数据的复制。

为了在多个进程间交换信息，内核专门留出了一块内存区，这段内存区可以由需要访问的进程将其映射到自己的私有地址空间。因此，进程就可以直接读写这一内存区而不需要进行数据的复制，从而大大提高了效率。

当然，由于多个进程共享一段内存，因此也需要依靠某种同步机制，如互斥锁和信号量等。

共享内存的使用包括如下步骤：

1. 创建/打开共享内存shmget()；
2. 映射共享内存shmat()；
3. 读写共享内存，例如：fgets()&printf()；
4. 撤销映射shmdt()；
5. 检查共享内存属性并删除共享内存对象shmctl()。

shmget()：

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/shm.h> | |
| 原型 | int shmget(key\_t key, int size, int shmflag); | |
| 参数 | key | 共享内存的键值，多个进程可以通过它访问同一个共享内存，其中有一个特殊值IPC\_PRIVATE，用于创建当前进程的私有共享内存 |
| size | 共享内存区大小 |
| shmflag | 同open()函数的权限位，也可以用八进制表示 |
| 返回值 | 成功 | 共享内存段标示符shmid |
| 失败 | -1 |

shmat()：

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/shm.h> | |
| 原型 | void \*shmat(int shmid, const void \*shmaddr, int shmflag); | |
| 参数 | shmid | 要映射的共享内存区标识符 |
| shmaddr | 将共享内存映射到指定地址(若为0，标示系统自动分配地址并把该段共享内存映射到调用进程的地址空间) |
| shmflag | SHM\_RDONLY：共享内存只读；  默认0：共享内存可读写 |
| 返回值 | 成功 | 被映射的段地址 |
| 失败 | -1 |

shmdt()：

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/shm.h> | |
| 原型 | int shmdt(const void \* shmaddr); | |
| 参数 | shmaddr | 被映射的共享内存段地址 |
| 返回值 | 成功 | 0 |
| 失败 | -1 |

注意：

在创建共享内存时，我们常使用shmget(key, 1024, IPC\_CREAT | IPC\_EXCL | 0666)的形式，表示若不存在，就创建；若存在，就报错errno=EEXIST。从而，在不同进程通信时，我们就可以区分出先后。

一般地，为了保证进程间的同步通信，我们常将共享内存和信号通信或信号灯集结合起来使用。

在用信号通信进行同步时，通信双方必须知道彼此的pid方能发送信号，所以在这种情况下，我们就需要将进程pid和信息封装成结构体。

先运行的一方：

1. 写自己的pid；
2. 唤醒对方kill()；
3. pause()；
4. 读取对方的pid并保存。

后运行的一方：

1. pause()；
2. 读取对方的pid并保存；
3. 写自己的pid；
4. 唤醒对方kill()。

shmctl()：

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/shm.h> | |
| 原型 | int shmctl(int shmid, int cmd, struct shmid\_ds \*buf); | |
| 参数 | shmid | 共享内存区标识符 |
| cmd | IPC\_STAT：获取对象属性  IPC\_SET：设置对象属性  IPC\_RMID：删除对象 |
| buf | 指定cmd为IPC\_STAT或IPC\_SET时，用以保存/设置属性 |
| 返回值 | 成功 | 0 |
| 失败 | -1(设置errno) |

注意：

1. struct shmid\_ds定义如下：

struct shmid\_ds {

……

shmatt\_t shm\_nattch; /\*No. of current attaches\*/

}

1. 我们常常先利用IPC\_STAT命令获取共享内存的映射数，若映射数为0，则删除共享内存。
2. 消息队列

消息队列(message queue)是功能最强大的一种进程间通信方式。消息队列，顾名思义，就是一些消息的列表，用户可以在消息队列中添加消息和读取消息等。消息队列由消息队列id来唯一标识(msgid)。消息队列和普通的队列一样，遵循FIFO的特性，消息队列的强大之处在于，它可以根据类型来收发消息。

消息发送端进程和消息接收端进程不需要额外实现进程间的同步。

消息队列的操作包括创建或打开消息队列、添加消息、读取消息和控制消息队列4种。

1. 创建或打开消息队列使用msgget()，这里创建的消息队列的数量会受到系统消息队列数量的限制；
2. 添加消息使用msgsnd()，它把消息添加到已打开的消息队列的末尾；
3. 读取消息使用msgrcv()，它把消息从队列中取走，与FIFO不同的是，它可以取走指定类型的消息；
4. 控制消息队列使用msgctl()，它可以获取消息队列的属性，还可以用于删除消息队列。

msgget()：

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h> | |
| 原型 | int msgget(key\_t key, int msgflag); | |
| 参数 | key：消息队列的键值，多个进程可以通过它访问同一个消息队列，其中有一个特殊值IPC\_PRIVATE，用于创建当前进程的私有消息队列 | |
| shmflag：同open()函数的权限位，也可以用八进制表示 | |
| 返回值 | 成功 | 消息队列标识符magid |
| 失败 | -1(errno) |

msgsnd()：

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h> | |
| 原型 | int msgsnd(int msgid, const void \*msgp, size\_t msgsz, int msgflag); | |
| 参数 | msqid | 消息队列标识符 |
| msgq | 指向消息结构的指针。该消息结构需要调用者定义如下：  struct msgbuf {  long mtype; /\*消息类型，该结构必须以此开始\*/  char mtext[1]; /\*消息正文\*/  } |
| msgsz | 要发送的消息正文的字节数 |
| msgflag | 0：msgsnd调用阻塞直到消息发送完成为止 |
| IPC\_NOWAIT：若消息无法立即发送(如队列已满)，则函数会立即返回 |
| 返回值 | 成功 | 0 |
| 失败 | -1(设置errno) |

msgrcv()：

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h> | |
| 原型 | ssize\_t msgrcv(int msgid, void \*msgp; size\_t msgsz; long msgtype; int msgflag); | |
| 参数 | msqid | 消息队列标识符 |
| msgq | 消息缓冲区，同msgsnd()的msgp |
| msgsz | 要接收的消息正文的字节数 |
| msgtype | 0：接收队列中第一个消息；  >0：接收消息队列中第一个类型为msgtype的消息；  <0：接收消息队列中第一个类型值不小于msgtype绝对值且类型值最小的消息。 |
| msgflag | 0：msgrcv调用阻塞直到接收到一条指定类型的消息为止 |
| IPC\_NOERROR：若返回的消息比msgsz字节多，则消息就会截短到msgsz字节，且不通知消息发送进程。 |
| IPC\_NOWAIT：若消息队列中没有指定来型的消息可以接收，则函数会立即返回 |
| 返回值 | 成功 | 接收到的消息正文的长度 |
| 失败 | -1(设置errno) |

msgctl()：

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h> | |
| 原型 | int msgctl(int msgid, int cmd, struct msqid\_ds \*buf); | |
| 参数 | msqid | 消息队列标识符 |
| cmd | IPC\_STAT：获取对象属性  IPC\_SET：设置对象属性  IPC\_RMID：删除对象 |
| buf | 指定cmd为IPC\_STAT或IPC\_SET时，用以保存/设置属性 |
| 返回值 | 成功 | 0 |
| 失败 | -1(设置errno) |

1. 信号量

信号灯(semaphore)，也叫信号量。它是不同进程间或一个给定进程内部不同线程间同步的机制。

信号灯的种类：

1. posix有名信号灯；
2. posix无名信号灯(基于内存的信号灯)；
3. system V信号灯(IPC对象)。

system V 的信号灯是一个或者多个信号灯的集合。其中的每一个都是单独的计数信号灯。而posix信号灯指的是单个计数信号灯。计数信号灯：值在0到n之间。用来统计资源，其值代表可用资源数。

在Linux系统中，使用信号量通常分为以下几个步骤：

1. 创建信号量semget()。不同进程通过使用同一个信号量键值来获得同一个信号量；
2. 初始化信号量semctl()：使用semctl()函数的SETVAL操作；
3. PV操作：调用semop()函数；
4. 删除信号量：使用semctl()函数的IPC\_RMID操作。

semget()：

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h> | |
| 原型 | int semget(key\_t key, int nsems, int semflag); | |
| 参数 | key：和信号灯集关联的key值 | |
| nsems：信号灯集中包含的信号灯的数目 | |
| semflag：常为，IPC\_CREAT | IPC\_EXCL | 0666 | |
| 返回值 | 成功 | 信号灯集id(semid) |
| 失败 | -1(errno) |

semctl()：

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <sys/types.h> #include <sys/ipc.h>  #include <sys/sem.h> | |
| 原型 | int semctl(int semid, int semnum, int cmd, [union semun arg]); | |
| 参数 | semid | 信号灯集id |
| semnum | 要修改的信号灯编号 |
| cmd | GETVAL：获取信号灯的值  SETVAL：初始化信号灯  IPC\_RMID：删除信号灯集 |
| buf | 当cmd为IPC\_RMID时，有3个参数；其余情况有四个参数，且第4个参数是一个共用体类型的变量，用户必须自己定义该共用体：  union semun {  int val; /\*value for SETVAL\*/  };(用作初始化时) |
| 返回值 | 成功 | 0 |
| 失败 | -1(设置errno) |

semop()：

|  |  |  |
| --- | --- | --- |
| 头文件 | #include <sys/types.h> #include <sys/ipc.h>  #include <sys/sem.h> | |
| 原型 | int semop(int semid, struct sembuf \*sops, unsigned nsops); | |
| 参数 | semid | 信号灯集id |
| sops | 指向信号灯集操作结构体的结构体指针；  struct sembuf {  unsigned short sem\_num; /\*要操作的信号灯编号\*/  short sem\_op;/\*1----V操作；-1----P操作\*/  short sem\_flg;/\*0(阻塞), IPC\_NOWAIT(不等), SEM\_UNDO(撤销)\*/  } |
| nsops | 要操作的信号灯的个数 |
| 返回值 | 成功 | 0 |
| 失败 | -1(设置errno) |