**进程间通信**

**内容概要**

* 进程间通信概念
* 管道通信
* 消息队列
* 共享内存
* 信号量

**进程间通信的概念**

* 数据传输：一个进程需要将它的数据发送给另一个进行，发送的数据量在一个字节到几兆字节之间。
* 共享数据：多个进程想要操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到。
* 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
* 资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供锁和同步机制。
* 进程控制：有些进程希望完全控制另一个进程的执行（如debug进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

**进程间通信的变革**

Linux进程间的通信（IPC）由以下几部分发展而来

* 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POXIS进程间通信

进程间通信的方式包括：管道、FIFO和信号。

System V进程间通信的方式包括：system V消息队列、System V信号灯、system V共享内存。

POXIS进程间通信包括：POXIS消息队列、POXIS信号灯、POXIS共享内存。

**现代的进程间通信方式**

* 管道（pipe）和命名管道（FIFO）
* 信号（signal）
* 消息队列
* 共享内存
* 信号量
* 套接字（socket）

**管道通信**

* 管道是针对于本地计算机的两个进程之间的通信而设计的通信方法，管道建立以后，实际获得两个文件描述符：一个用于读取而另外一个用于写入。
* 最常见的IPC机制，通过pipe系统调用。
* 管道的单工的，数据只能向一个方向流动，需要双向通信时，需要建立起两个管道。
* 数据的读出和写入：一个进程向管道中写入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。

**管道分类**

匿名管道

* 在关系进程中（父进程和子进程、兄弟进程之间）
* 由pipe系统调用，管道由父进程建立。
* 管道位于内核空间，其实是一块缓存。

命名管道（FIFO）

* 两个没有任何关系的进程之间可通过命名管道进行数据传输。本质是内核中的一块缓存，另在文件系统中以一个特殊的设备文件（管道文件）存在。
* 通过系统调用mkfifo创建。

**管道创建**

#include <unistd.h>

int pipe(int fd[2]);

返回：成功返回0，错误返回-1

两个文件描述符组

fd[0]：pipe的读端

fd[1]：pipe的写端

**管道案例**

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

int main()

{

int pipe\_fd[2];

if (pipe(pipe\_fd) < 0)

{

printf("pipe create error\n");

return -1;

}

else

printf("pipe create success\n");

close(pipe\_fd[0]);

close(pipe\_fd[1]);

}

**管道读写**

管道用于不同进程之间的通信。实际上，通常先创建一个管道，再通过fork函数创建一个子进程。

关闭父进程fd[0]和子进程fd[1]。

#include <unistd.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

/\*\*

\* 父进程通过管道传输两个数据给子进程

\* 由子进程负责从管道中读取并输出

\*/

int main()

{

int fd[2];

/\* 创建管道 \*/

if (pipe(fd) < 0) {

perror("pipe error");

exit(1);

}

pid\_t pid;

if ((pid = fork()) < 0) {

perror("fork error");

exit(1);

} else if (pid > 0) { /\* 父进程 \*/

close(fd[0]); /\* 父进程用来写入数据 \*/

int start = 1, end = 100;

if (write(fd[1], &start, sizeof(int))

!= sizeof(int)) {

perror("write error");

exit(1);

}

if (write(fd[1], &end, sizeof(int))

!= sizeof(int)) {

perror("write error");

exit(1);

}

close(fd[1]);

} else { /\* 子进程 \*/

close(fd[1]); /\* 子进程用来读取数据 \*/

int start, end;

if (read(fd[0], &start, sizeof(int)) < 0) {

perror("read error");

exit(1);

}

if (read(fd[0], &end, sizeof(int)) < 0) {

perror("read error");

exit(1);

}

close(fd[1]);

printf("child process read start: %d, end: %d\n", start, end);

}

exit(0);

}