Linux进程间通信

Version 1.0

西安電子科技大學

需要掌握的要点

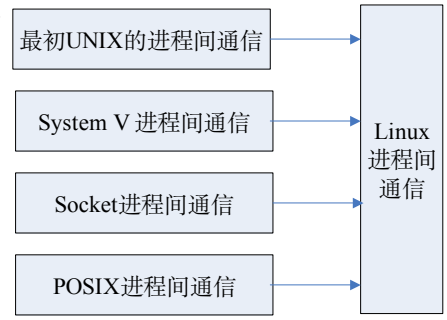
- IPC的概念和分类
- 匿名管道
- 命名管道（FIFO）
- 信号
- 信号量
- 共享内存
- 消息队列

西安電子科技大學

Linux下进程间通信概述

- **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**
- **集合System V IPC(贝尔实验室)和socket的进程间通信机制(BSD)的优势**

- Unix进程间通信（IPC）方式包括管道、FIFO以及信号。
- System V进程间通信（IPC）包括System V消息队列、System V信号量以及System V共享内存区。
- Posix 进程间通信（IPC）包括Posix消息队列、Posix信号量以及Posix共享内存区。



Linux下进程间通信概述

- **进程间通信的难点：**
  - 独立的进程地址空间；
  - 异步工作
- **常用的进程间通信机制**
  - 管道（Pipe）及有名管道（named pipe）：进程间建立字节流队列。
  - 消息队列（Messge Queue）：进程间建立消息队列。
  - 信号（Signal）：信号是在软件层次上对中断机制的一种模拟。一个进程可以向另一个进程发送信号，就像外设触发了CPU的中断。
  - 共享内存（Shared memory）：将同一块物理内存映射到两个不同进程的地址空间中。
  - 信号量（Semaphore）：主要作为不同进程之间的同步和互斥手段。
  - 套接字（Socket）：这是一种更为一般的进程间通信机制，它可用于网络中不同机器上的两个进程之间的进程间通信，应用非常广泛。

无名管道和有名管道

- 管道主要包括两种：无名管道和有名管道。
  - 无名管道特点
    - 只能用于具有亲缘关系的进程之间的通信（父子进程或者兄弟进程）
    - 半双工的通信模式，具有固定的读端和写端。
    - 管道也可以看成是一种内存中的特殊文件，对于它的读写也可以使用普通的read()、write()等函数。
  - 有名管道特点
    - 它可以使互不相关的两个进程实现彼此通信。
    - 对应一个实际的文件，并且在文件系统中是可见的。在建立了管道之后，两个进程就可以把它当作普通文件一样进行读写操作，使用非常方便。
    - 严格地遵循先进先出规则，对管道的读总是从开始处返回数据，对它们的写则把数据添加到末尾，因此不支持如lseek()等文件定位操作。

无名管道

- 关于无名管道的系统调用
  - 管道创建和关闭
    - 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fds[0]和fds[1]，其中fds[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。
    - 管道关闭时只需将这两个文件描述符关闭即可，可使用普通的close()函数逐个关闭各个文件描述符。

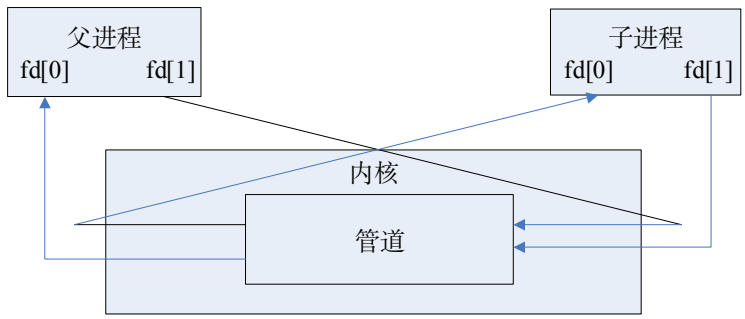
无名管道

- 无名管道系统调用
  - pipe()
    - 创建管道可以通过调用pipe()来实现.
  - pipe()语法：

所需头文件	#include <unistd.h>
函数原型	int pipe(int fd[2])
函数传入值	fd[2]：管道的两个文件描述符，之后就可以直接操作这两个文件描述符
函数返回值	成功：0
	出错：-1

无名管道

- 无名管道系统调用
  - 父子进程管道的文件描述符对应关系



无名管道

- **无名管道系统调用**
  - 管道读写注意点：
    - 只有在管道的读端存在时，向管道写入数据才有意义。否则，向管道写入数据的进程将收到内核传来的SIGPIPE信号（通常为Broken pipe错误）。
    - 向管道写入数据时，Linux将不保证写入的原子性，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。如果读进程不读取管道缓冲区中的数据，那么写操作将会一直阻塞。
    - 如果管道缓冲区是空的，那么读进程将会一直阻塞。

标准流管道

- **标准流管道函数说明**
  - 与Linux的文件操作中有基于文件流的标准I/O操作一样，管道的操作也支持基于文件流的模式。这种基于文件流的管道主要是用来创建一个连接到另一个进程的管道，这里的“另一个进程”也就是一个可以进行一定操作的可执行文件。
  - 标准流管道就将一系列的创建过程合并到一个函数popen()中完成。它所完成的工作有以下几步。
    - 创建一个管道。
    - fork()一个子进程。
    - 在父子进程中关闭不需要的文件描述符。
    - 执行exec函数族调用。
    - 执行函数中所指定的命令。

标准流管道

- 标准流管道
  - popen函数格式：

所需头文件	#include <stdio.h>	
函数原型	FILE *popen(const char *command, const char *type)	
函数传入值	command：指向的是一个以 null 结束符结尾的字符串，这个字符串包含一个 shell 命令，并被送到/bin/sh 以-c 参数执行，即由 shell 来执行	
	type：	“r”：文件指针连接到 command 的标准输出，即该命令的结果产生输出 “w”：文件指针连接到 command 的标准输入，即该命令的结果产生输入
函数返回值	成功：文件流指针	
	出错：–1	

  - pclose函数格式：

所需头文件	#include <stdio.h>
函数原型	int pclose(FILE *stream)
函数传入值	stream：要关闭的文件流
函数返回值	成功：返回由 popen()所执行的进程的退出码
	出错：–1

有名管道

- 有名管道(FIFO)
  - 有名管道可以实现任意量个进程之间的通信，并不限制两个进程间有亲缘关系；
  - 有名管道作为一种特殊的文件存放在文件系统中，使用完毕仍然存在，除非对其进行删除。
  - 有名管道也只能用于单向传输。

有名管道

- **有名管道(FIFO)**
  - 通过mkfifo()创建有名管道,可以指定管道的路径和打开的模式。
  - mkfifo()函数语法:

所需头文件	#include <sys/types.h> #include <sys/state.h>	
函数原型	int mkfifo(const char *filename,mode_t mode)	
函数传入值	filename：要创建的管道	
函数传入值	mode：	O_RDONLY：读管道
		O_WRONLY：写管道
		O_RDWR：读写管道
		O_NONBLOCK：非阻塞
		O_CREAT：如果该文件不存在，那么就创建一个新的文件，并用第三个参数为其设置权限
		O_EXCL：如果使用 O_CREAT 时文件存在，那么可返回错误消息。这一参数可测试文件是否存在
函数返回值	成功：0	
	出错：-1	

管道通信

- **有名管道(FIFO)**
  - FIFO相关出错信息

EACCESS	参数 filename 所指定的目录路径无可执行的权限
EEXIST	参数 filename 所指定的文件已存在
ENAMETOOLONG	参数 filename 的路径名称太长
ENOENT	参数 filename 包含的目录不存在
ENOSPC	文件系统的剩余空间不足
ENOTDIR	参数 filename 路径中的目录存在但却非真正的目录
EROFS	参数 filename 指定的文件存在于只读文件系统内

管道通信

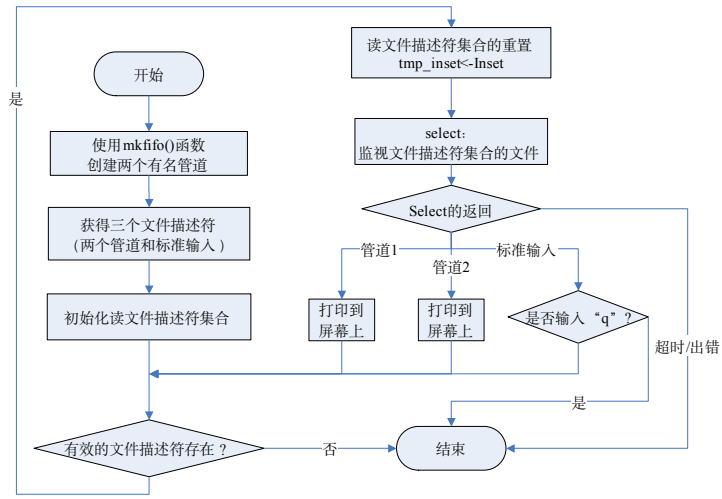
- **有名管道(FIFO)**
  - FIFO读写
    - 对于读进程
      - 若该管道是阻塞打开，且当前FIFO内没有数据，则对读进程而言将一直阻塞到有数据写入。
      - 若该管道是非阻塞打开，则不论FIFO内是否有数据，读进程都会立即执行读操作。即如果FIFO内没有数据，则读函数将立刻返回0。
    - 对于写进程
      - 若该管道是阻塞打开，则写操作将一直阻塞到数据可以被写入。
      - 若该管道是非阻塞打开而不能写入全部数据，则读操作进行部分写入或者调用失败。

管道通信实验

- **实验目的**
  - 通过编写有名管道多路通信实验，读者可进一步掌握管道的创建、读写等操作，同时，也复习使用select()函数实现管道的通信。
- **实验内容**
  - 在多路复用模型实验中，用到有名管道（使用mknod命令创建）和多路复用（使用poll()函数）。以下实验在功能上跟这个实验完全相同，这里只是用管道函数创建有名管道（并不是在控制台下输入命令），而且使用select()函数替代poll()函数实现多路复用（使用select()函数是出于以演示为目的）。

管道通信实验

- 实验步骤



Linux中的信号机制

- ✔ Linux信号机制是在应用软件层次上对中断机制的一种模拟，是一种异步通信方式
- ✔ 每个进程都有一个自己私有的信号处理函数映射表，当该进程成收到一个信号时，对应的信号处理函数被触发执行。
- ✔ 一个进程可以向另外一个进程发送信号，也可以向自己发送信号；操作系统内核也可以向一个进程发送信号，以通知某些硬件事件。
- ✔ 信号处理函数映射表中共有64个表项。前32个信号，编号为1～31，有预定义的含义和处理函数；后32个作为扩充。
- ✔ 前32个是不可靠信号(非实时的)，后32个为可靠信号(实时信号)。不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号丢失，而后者不会。

回调函数例一：Linux中的信号机制

$ kill -l

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	34) SIGRTMIN	35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3
38) SIGRTMIN+4	39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12	47) SIGRTMIN+13
48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14	51) SIGRTMAX-13	52) SIGRTMAX-12
53) SIGRTMAX-11	54) SIGRTMAX-10	55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7
58) SIGRTMAX-6	59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX			

信号通信

- 信号概述
  - 信号缺省操作

信 号 名	含 义	默 认 操 作
SIGHUP	该信号在用户终端连接（正常或非正常）结束时发出，通常是在终端的控制进程结束时，通知同一会话内的各个进程与控制终端不再关联	终止
SIGINT	该信号在用户键入 INTR 字符（通常是 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	该信号和 SIGINT 类似，但由 QUIT 字符（通常是 Ctrl-\）来控制	终止
SIGILL	该信号在一个进程企图执行一条非法指令时（可执行文件本身出现错误，或者试图执行数据段、堆栈溢出时）发出	终止
SIGFPE	该信号在发生致命的算术运算错误时发出。这里不仅包括浮点运算错误，还包括溢出及除数为 0 等其他所有的算术的错误	终止
SIGKILL	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
SIGALRM	该信号当一个定时器到时的时候发出	终止
SIGSTOP	该信号用于暂停一个进程，且不能被阻塞、处理或忽略	暂停进程
SIGTSTP	该信号用于交互停止进程，用户可键入 SUSP 字符时（通常是 Ctrl+Z）发出这个信号	停止进程
SIGCHLD	子进程改变状态时，父进程会收到这个信号	忽略

回调函数例一：Linux中的信号机制

✔ 如何向一个进程发送信号?

所需头文件	#include <signal.h> #include <sys/types.h>	
函数原型	int kill(pid_t pid, int sig)	
函数传入值	pid：	正数：要发送信号的进程号
		0：信号被发送到所有和当前进程在同一个进程组的进程
		−1：信号发给所有的进程表中的进程（除了进程号最大的进程外）
		<-1：信号发送给进程组号为-pid 的每一个进程
	sig：信号	
函数返回值	成功：0	
	出错：−1	

回调函数例一：Linux中的信号机制

✔ 如何设置信号关联动作?

所需头文件	#include <signal.h>	
函数原型	typedef void (*sighandler_t)(int); sighandler_t signal(int signum, sighandler_t handler);	
函数传入值	signum：指定信号代码	
	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
函数返回值	成功：以前的信号处理配置	
	出错：−1	

信号通信

- **信号发送和捕捉**
  - 信号发送：kill()和raise()

    kill()函数同读者熟知的kill系统命令一样，可以发送信号给进程或进程组,它不仅可以中止进程（实际上发出SIGKILL信号），也可以向进程发送其他信号。

    kill()函数语法：

所需头文件	#include <signal.h> #include <sys/types.h>	
函数原型	int kill(pid_t pid, int sig)	
函数传入值	pid：	正数：要发送信号的进程号
		0：信号被发送到所有和当前进程在同一个进程组的进程
		-1：信号发给所有的进程表中的进程（除了进程号最大的进程外）
		<-1：信号发送给进程组号为-pid 的每一个进程
	sig：信号	
函数返回值	成功：0	
	出错：-1	

信号通信

- **信号发送和捕捉**
  - 信号捕捉：alarm()、pause()
    - alarm()也称为闹钟函数，它可以在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SIGALARM信号 。

所需头文件	#include <unistd.h>
函数原型	unsigned int alarm(unsigned int seconds)
函数传入值	seconds：指定秒数，系统经过 seconds 秒之后向该进程发送 SIGALRM 信号
函数返回值	成功：如果调用此 alarm()前，进程中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
	出错：-1

    - pause()函数是用于将调用进程挂起直至捕捉到信号为止（该信号没有被阻塞）。这个函数很常用，通常可以用于判断信号是否已到。

所需头文件	#include <unistd.h>
函数原型	int pause(void)
函数返回值	-1，并且把 error 值设为 EINTR

信号通信

- **信号发送和捕捉**
  - 信号的处理：signal、sigaction
    - 使用signal()处理信号时，只需要指出要处理的信号和处理函数即可。它主要是用于前32种非实时信号的处理，不支持信号传递信息，但是由于使用简单、易于理解，因此也受到很多程序员的欢迎。
    - signal()函数的语法：

所需头文件	#include <signal.h>	
函数原型	typedef void (*sighandler_t)(int); sighandler_t signal(int signum, sighandler_t handler);	
函数传入值	signum：指定信号代码	
	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
函数返回值	成功：以前的信号处理配置	
	出错：-1	

信号通信

- **信号发送和捕捉**
  - 信号的处理：signal、sigaction
    - sigaction() 函数相对于signal()更加健壮，推荐使用这个

所需头文件	#include <signal.h>
函数原型	int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
函数传入值	signum：信号代码，可以为除 SIGKILL 及 SIGSTOP 外的任何一个特定有效的信号
	act：指向结构 sigaction 的一个实例的指针，指定对特定信号的处理
	oldact：保存原来对相应信号的处理
函数返回值	成功：0
	出错：-1

信号通信

- 信号发送和捕捉
  - 信号的处理：signal、sigaction
    - sigaction()函数中第2个和第3个参数用到的sigaction结构

```
struct sigaction
{
    void (*sa_handler)(int signo);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restore)(void);   //Linux中不支持
}
```

sa_handler是一个函数指针，指定信号处理函数，这里除可以是用户自定义的处理函数外，还可以为SIG_DFL（采用缺省的处理方式）或SIG_IGN（忽略信号）。它的处理函数只有一个参数，即信号值。

sa_mask是一个信号集，它可以指定在信号处理程序执行过程中哪些信号应当被屏蔽，在调用信号捕获函数之前，该信号集要加入到信号的信号屏蔽字中.

sa_flags中包含了许多标志位，是对信号进行处理的各个选择项

信号集函数

- 使用信号集函数组处理信号时涉及一系列的函数，这些函数按照调用的先后次序可分为以下几大功能模块：创建信号集合、注册信号处理函数以及检测信号。
- 其中，创建信号集合主要用于处理用户感兴趣的一些信号，其函数包括以下几个。
- sigemptyset()：将信号集合初始化为空。
- sigfillset()：将信号集合初始化为包含所有已定义的信号的集合。
- sigaddset()：将指定信号加入到信号集合中去。
- sigdelset()：将指定信号从信号集合中删去。
- sigismember()：查询指定信号是否在信号集合之中。
- Sigprocmask()：检测或改变信号阻塞状态
- sigpending()：信号是否为未决信号

信号通信

- 信号发送和捕捉
  - 总之，在处理信号时，一般遵循如图6.13所示的操作流程。

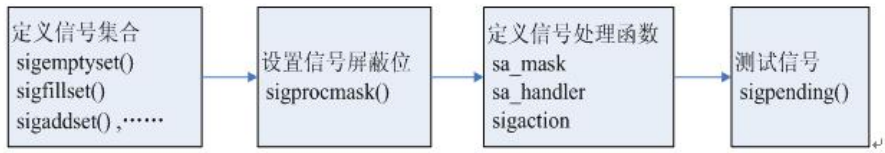


图 6.13　一般的信号操作处理流程

信号量（semaphore）

- 信号量概述
  - 信号量是用来保护共享资源的一种机制，使得共享资源在一个时刻只有一个进程（线程）访问。
  - 信号量对应于某一种资源，取一个非负的整型值。信号量值指的是当前可用的该资源的数量，若它等于0则意味着目前没有可用的资源。
  - 针对信号量的两种操作：
    - P操作：如果有可用的资源（信号量值>0），则占用一个资源（给信号量值减去一，进入临界区代码）;如果没有可用的资源（信号量值等于0），则进入到该信号量的等等队列并阻塞休眠，直到系统将其唤醒
    - V操作：如果在该信号量的等待队列中有进程在等待资源，则唤醒一个阻塞进程。如果没有进程等待它，则释放一个资源（给信号量值加一）。

信号量

- **信号量的使用方法：**

(1) 定义信号量S
(2) INIT_VAL(S); /* 对信号量S进行初始化 */
(3) 非临界区;
(4) P(S); /* 进行P操作 */
(5) 进入临界区（使用资源R）; /* 只有有限个（通常只有一个）进程被允许进入该区*/
(6) V(S); /* 进行V操作 */
(7) 非临界区;

信号量

- **信号量编程 (SYSTEM V)**
  - 使用信号量的步骤：
    - 第一步：创建信号量或获得在系统已存在的信号量，此时需要调用semget()函数。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。
    - 第二步：初始化信号量，此时使用semctl()函数的SETVAL操作。当使用二维信号量时，通常将信号量初始化为1。
    - 第三步：进行信号量的PV操作，此时调用semop()函数。这一步是实现进程之间的同步和互斥的核心工作部分。
    - 第四步：如果不需要信号量，则从系统中删除它，此时使用semclt()函数的IPC_RMID操作。此时需要注意，在程序中不应该出现对已经被删除的信号量的操作。

信号量

- 创建信号量
  - semget()函数语法:

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semget(key_t key, int nsems, int semflg)
函数传入值	key：信号量的键值，多个进程可以通过它访问同一个信号量，其中有个特殊值IPC_PRIVATE。它用于创建当前进程的私有信号量。
	nsems：需要创建的信号量数目，通常取值为 1。
	semflg：同 open()函数的权限位，也可以用八进制表示法，其中使用 IPC_CREAT 标志创建新的信号量，即使该信号量已经存在（具有同一个键值的信号量已在系统中存在），也不会出错。如果同时使用IPC_EXCL 标志可以创建一个新的唯一的信号量，此时如果该信号量已经存在，该函数会返回出错。
函数返回值	成功：信号量标识符，在信号量的其他函数中都会使用该值。
	出错：-1

信号量

- 信号量编程
  - semctl()函数语法:

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semctl(int semid, int semnum, int cmd, union semun arg)
函数传入值	semid：semget()函数返回的信号量标识符。
	semnum：信号量编号，当使用信号量集时才会被用到。通常取值为 0，就是使用单个信号量（也是第一个信号量）。
	cmd：指定对信号量的各种操作，当使用单个信号量（而不是信号量集）时，常用的操作有以下几种： IPC_STAT：获得该信号量（或者信号量集合）的 semid_ds 结构，并存放在由第四个参数 arg 结构变量的 buf 域指向的 semid_ds 结构中。semid_ds 是在系统中描述信号量的数据结构。 IPC_SETVAL：将信号量值设置为 arg 的 val 值。 IPC_GETVAL：返回信号量的当前值。 IPC_RMID：从系统中，删除信号量（或者信号量集）

	arg：是 union semnn 结构，可能在某些系统中不给出该结构的定义，此时必须由程序员自己定义。
	union semun
	{
	int val;
	struct semid_ds *buf;
	unsigned short *array;
	}

函数返回值	成功：根据 cmd 值的不同而返回不同的值。 IPC_STAT、IPC_SETVAL、IPC_RMID：返回 0 IPC_GETVAL：返回信号量的当前值
	出错：-1

信号量

- 信号量编程
  - semop()函数语法:

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semop(int semid, struct sembuf *sops, size_t nsops)
函数传入值	semid：semget()函数返回的信号量标识符。

	sops：指向信号量操作数组，一个数组包括以下成员：
	struct sembuf
	{
	short sem_num; /* 信号量编号，使用单个信号量时，通常取值为 0 */
	short sem_op;
	/* 信号量操作：取值为-1 则表示 P 操作，取值为+1 则表示 V 操作*/
	short sem_flg;
	/* 通常设置为 SEM_UNDO。这样在进程没释放信号量而退出时，系统自动释放该进程中未释放的信号量 */
	}

	nsops：操作数组 sops 中的操作个数（元素数目），通常取值为 1（一个操作）
函数返回值	成功：信号量标识符，在信号量的其他函数中都会使用该值。
	出错：-1

- 例子
  - Init_sem()
  - Del_sem()
  - Sem_p()
  - Sem_v()

西安電子科技大學

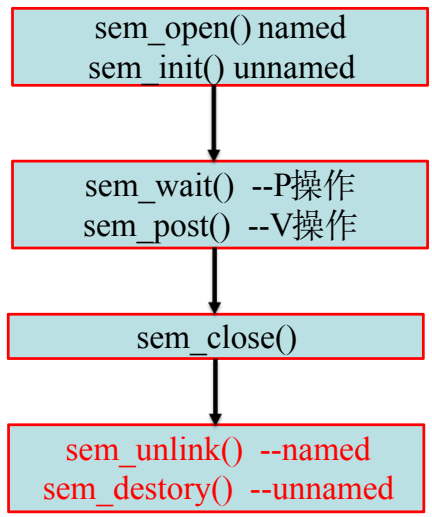
信号量

- **信号量编程 (POSIX)**
  - 使用信号量的步骤：
    - 第一步：创建并初始化信号量或获得在系统已存在的信号量，此时需要调用sem_open()函数(named)或sem_init()函数(unnamed)。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。
    - 第二步：进行信号量的PV操作，调用sem_wait()函数进行P操作，调用sem_post()函数进行V操作。这一步是实现进程之间的同步和互斥的核心工作部分。
    - 第四步：如果不需要信号量，则从系统中删除它，此时使用sem_unlink()函数(named)或sem_destory()函数(unnamed) 。
- **特别注意 在编译时要加上 –pthread 选项，否则会出现对sem_xxx 的未定义引用。**
- **https://blog.csdn.net/sicofield/article/details/10897091**

信号量

- **信号量编程流程(POSIX)**

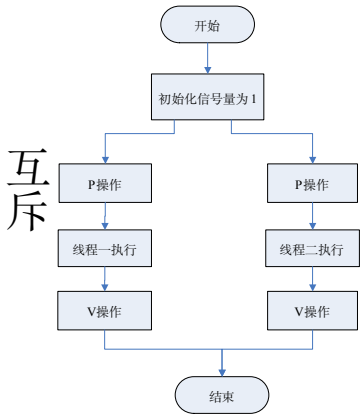
例子



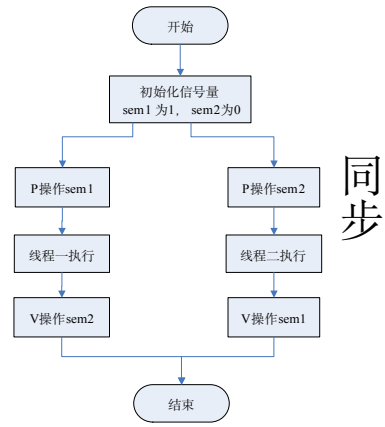
进程间的同步和互斥

- 信号量进程程控制
  - PV原子操作主要用于进程或线程间的同步和互斥这两种典型情况。

互斥



同步



信号量

- 信号量编程 (POSIX)
- sem_open - initialize and open a named semaphore

```
#include <fcntl.h>          /* For O_* constants */
#include <sys/stat.h>    /* For mode constants */
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

信号量

- 信号量编程 (POSIX)
- sem_init - initialize an unnamed semaphore

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- sem_close - close a named semaphore

```
#include <semaphore.h>
int sem_close(sem_t *sem);
```


信号量

- 信号量编程 (POSIX)
- sem_wait, sem_timedwait, sem_trywait - lock a semaphore

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

信号量

- 信号量编程 (POSIX)
- sem_post - unlock a semaphore

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```

- sem_getvalue - get the value of a semaphore

```
#include <semaphore.h>
int sem_getvalue(sem_t *sem, int *sval);
```


信号量

- 信号量编程 (POSIX)
- sem_destroy - destroy an unnamed semaphore

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

- sem_unlink - remove a named semaphore

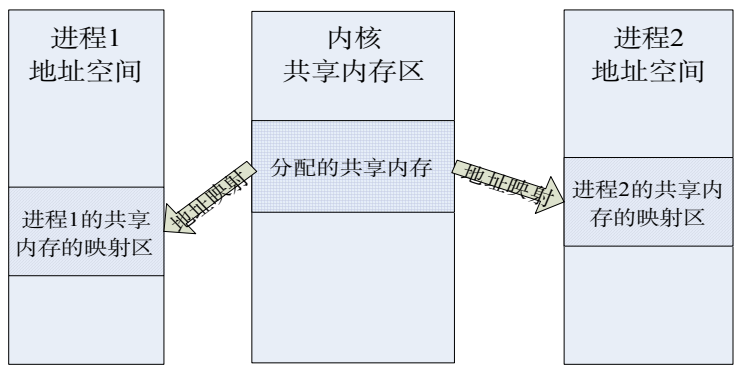
```
#include <semaphore.h>
int sem_unlink(const char *name);
```

共享内存

- 共享内存是一种最为高效的进程间通信方式，进程可以直接读写内存，而不需要任何数据的拷贝
- 为了在多个进程间交换信息，内核专门留出了一块内存区，可以由需要访问的进程将其映射到自己的私有地址空间
- 进程就可以直接读写这一内存区而不需要进行数据的拷贝，从而大大提高的效率。
- 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

- 共享内存原理示意图



共享内存

- **共享内存实现的步骤(SYSTEM V):**
  - 创建共享内存，这里用到的函数是shmget，也就是从内存中获得一段共享内存区域
  - 映射共享内存，也就是把这段创建的共享内存映射到具体的进程空间中去，这里使用的函数是shmat
  - 到这里，就可以使用这段共享内存了，也就是可以使用不带缓冲的I/O读写命令对其进行操作
  - 撤销映射的操作，其函数为shmdt
  - 删除共享内存用shmctl函数。 (shmctl(shmid,IPC_RMID,0))

共享内存

- **shmget函数语法:**

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int shmget(key_t key, int size, int shmflg)
函数传入值	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，其中有个特殊值IPC_PRIVATE。它用于创建当前进程的私有共享内存。
	size：共享内存区大小
	shmflg：同 open()函数的权限位，也可以用八进制表示法
函数返回值	成功：共享内存段标识符
	出错：-1

共享内存

- **shmat函数语法：**

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	char *shmat(int shmid, const void *shmaddr, int shmflg)	
函数传入值	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 则表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）	
	shmflg	SHM_RDONLY：共享内存只读
		默认 0：共享内存可读写
函数返回值	成功：被映射的段地址	
	出错：–1	

共享内存

- **shmdt函数语法：**

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int shmdt(const void *shmaddr)
函数传入值	shmaddr：被映射的共享内存段地址
函数返回值	成功：0
	出错：–1

共享内存

- **共享内存实现的步骤(POSIX):**
  - 创建共享内存对象，这里用到的函数是shm_open,
  - 分配共享内存的大小，这里用的的函数是ftruncate
  - 映射共享内存，也就是把这段创建的共享内存映射到具体的进程空间中去，这里使用的函数是mmap
  - 到这里，就可以使用这段共享内存了，也就是可以使用不带缓冲的I/O读写命令对其进行操作
  - 撤销映射的操作，这里使用的函数为munmap
  - 关闭共享内存对象，这里使用的函数为close
  - 删除共享内存用shm_unlink函数。
- **例子**

消息队列

- **消息队列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。从这点上看，消息队列具有一定的FIFO特性，但是它可以实现消息的随机查询，比FIFO具有更大的优势。同时，这些消息又是存在于内核中的，由“队列ID”来标识。**

消息队列(SYSTEM V)

- 消息队列的实现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这四种操作
- 创建或打开消息队列使用的函数是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
- 添加消息使用的函数是msgsnd函数，它把消息添加到已打开的消息队列末尾
- 读取消息使用的函数是msgrcv，它把消息从消息队列中取走，与FIFO不同的是，这里可以指定取走某一条消息
- 控制消息队列使用的函数是msgctl，它可以完成多项功能。

消息队列(SYSTEM V)

- msgget函数语法：

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int msgget(key_t key, int msgflg)
函数传入值	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，其中有个特殊值IPC_PRIVATE。它用于创建当前进程的私有消息队列。
	msgflg：权限标志位
函数返回值	成功：消息队列 ID
	出错：−1

消息队列(SYSTEM V)

- **msgsnd函数语法：**

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)	
函数传入值	msqid：消息队列的队列 ID	

	msgp：指向消息结构的指针。该消息结构 msgbuf 通常为：
	struct msgbuf
	{
	long mtype; /* 消息类型，该结构必须从这个域开始 */
	char mtext[1]; /* 消息正文 */
	}	

	msgsz：消息正文的字节数（不包括消息类型指针变量）	
	msgflg：	IPC_NOWAIT 若消息无法立即发送（比如：当前消息队列已满），函数会立即返回。
		0：msgsnd 调用阻塞直到发送成功为止。
函数返回值	成功：0	
	出错：−1	

消息队列(SYSTEM V)

- **msgrcv函数语法：**

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgrcv(int msgid, void *msgp, size_t msgsz, long int msgtyp, int msgflg)	
函数传入值	msqid：消息队列的队列 ID	
	msgp：消息缓冲区，同于 msgsnd()函数的 msgp	
	msgsz：消息正文的字节数（不包括消息类型指针变量）	
	msgtyp：	0：接收消息队列中第一个消息
		大于 0：接收消息队列中第一个类型为 msgtyp 的消息
		小于 0：接收消息队列中第一个类型值不小于 msgtyp 绝对值且类型值又最小的消息
	msgflg：	MSG_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 若在消息队列中并没有相应类型的消息可以接收，则函数立即返回
		0：msgsnd()调用阻塞直到接收一条相应类型的消息为止
函数返回值	成功：0	
	出错：−1	

消息队列(SYSTEM V)

- **msgctl函数语法：**

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgctl (int msgqid, int cmd, struct msqid_ds *buf)	
函数传入值	msqid：消息队列的队列 ID	
	cmd： 命令参数	IPC_STAT：读取消息队列的数据结构 msqid_ds，并将其存储在 buf 指定的地址中
		IPC_SET：设置消息队列的数据结构 msqid_ds 中的 ipc_perm 域（IPC 操作权限描述结构）值。这个值取自 buf 参数
		IPC_RMID：从系统内核中删除消息队列
	buf：描述消息队列的 msgqid_ds 结构类型变量	
函数返回值	成功：0	
	出错：−1	

消息队列(POSIX)

- **Message queues are created and opened usingmq_open();**
- **Messages are transferred to and from a queue usingmq_send() and mq_receive().**
- **When a process has finished using the queue, it closes it using mq_close(),**
- **when the queue is no longer required, it can be deleted using mq_unlink().**
- **Queue attributes can be retrieved and modified usingmq_getattr() and mq_setattr().**
- **A process can request asynchronous notification of the arrival of a message on a previously empty queue usingmq_notify().**
- **相关函数**
  - **mq_close(), mq_getattr(), mq_notify(), mq_open(), mq_receive(), mq_send(), mq_setattr(), mq_timedreceive(), mq_timedsend(), mq_unlink()**
- **例子**