AMBOW
EDUCATION
GROUP

安博教育

第二章 进程及进程间通讯

内容提要

进程

IPC 概述

管道

System V IPC

POSIX IPC

进程概述

进程简介

进程和线程是调度的基本单位：

进程和线程的管理是操作系统中的核心部分。

线程描述进程内的执行，负责执行包含在进程的地址空间中的代码。

20 世纪 60 年代，进程 (process) 一词首先在麻省理工学院的 MULTICS 和 IBM 的 CTSS/360 系统中被引入。

进程概述

进程的三个重要特性

独立性　　进程是系统中独立存在的实体，它可以拥有自己独立的资源，比如文件和设备描述符等。未经进程本身允许，其他进程不能访问到这些资源。

动态性　　程序只是一个静态的指令集合，而进程是一 个正在系统中活动的指令集合。在进程中加入了时间的概念。进程具有自己的生命周期和各种不同的状态。

并发性　并发性由独立性和动态性衍生而来。若干个进程可以在单处理机状态上并发执行。

并发与并行:

并行 指在同一时刻内，有多条指令在多个处理机上同时执行。

并发 指在同一时刻内只能有一条指令执行，但多个进程的指令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果。

进程组成

作为申请系统资源的基本单位，进程必须有一个对应的物理内存空间，要对其进行高效的管理，首先要用数据结构对空间进行描述。

进程编号

进程以进程号 PID(process ID) 作为标识。任何对进程的操作都要有相应的 PID 号。

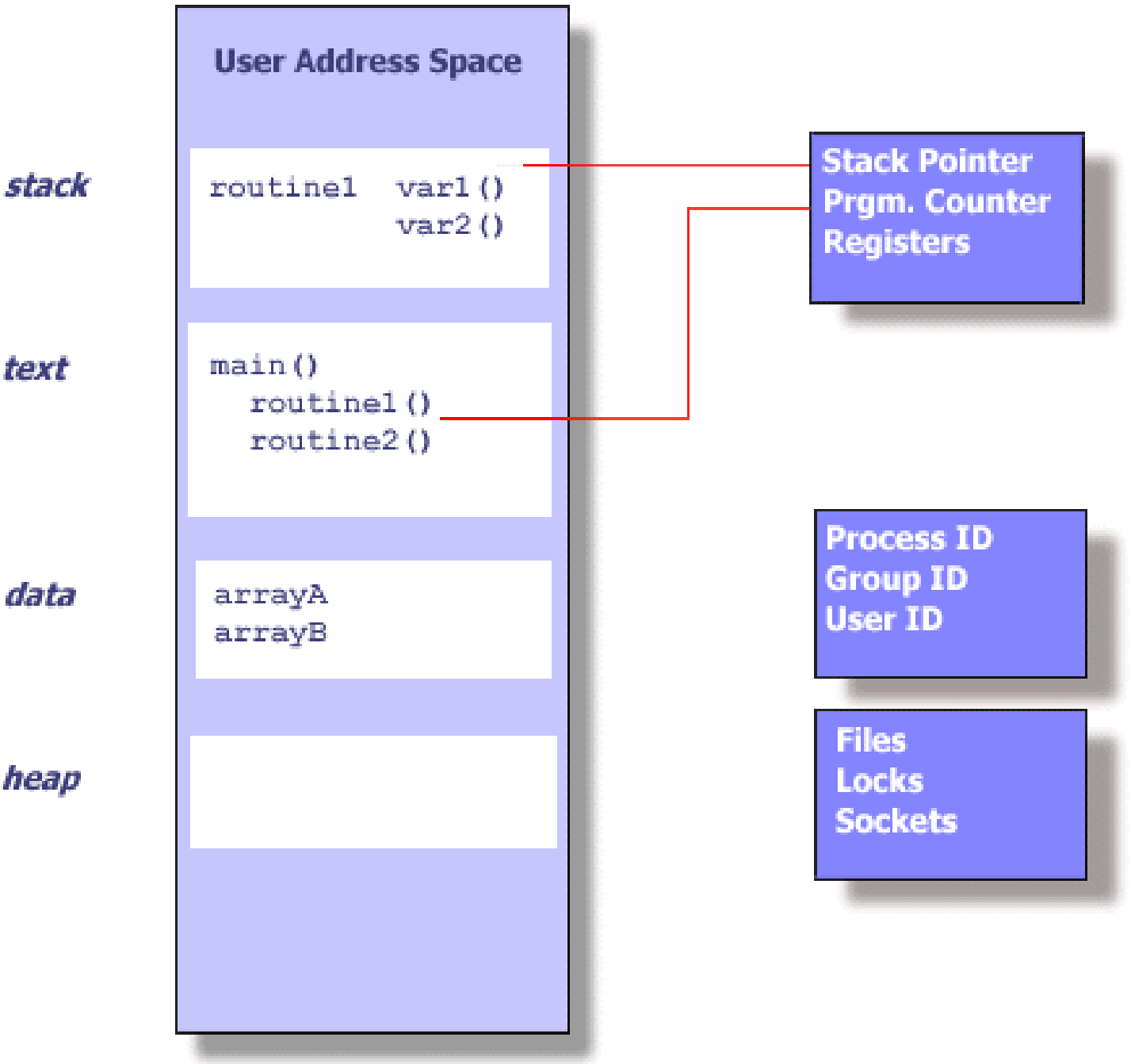
每个进程都属于一个用户，进程要配备其所属的用户编号 UID 。

每个进程都属于一个用户组，所以进程还要配备其归属的用户组编号 GID 。

GNU/Linux 进程有两种基本形式，分别是内核线程 (由 kernel_thread 创建) 和用户进程 (由 fork, clone 创建) 。

进程组成

作为申请系统
效的管理，首
进程编号
进程以进程号
每个进程都属
每个进程都属
GNU/Linux
户进程（由 fo



进行高

PID 号。

建）和用

进程组成

进程上下文

运行进程的环境称为进程上下文 (context)

进程的上下文组成

进程控制块 PCB 包括进程的编号、状态、优先级以及正文段和数据段中数据分布的大概情况。

正文段 (text segment) 存放该进程的可执行代码。

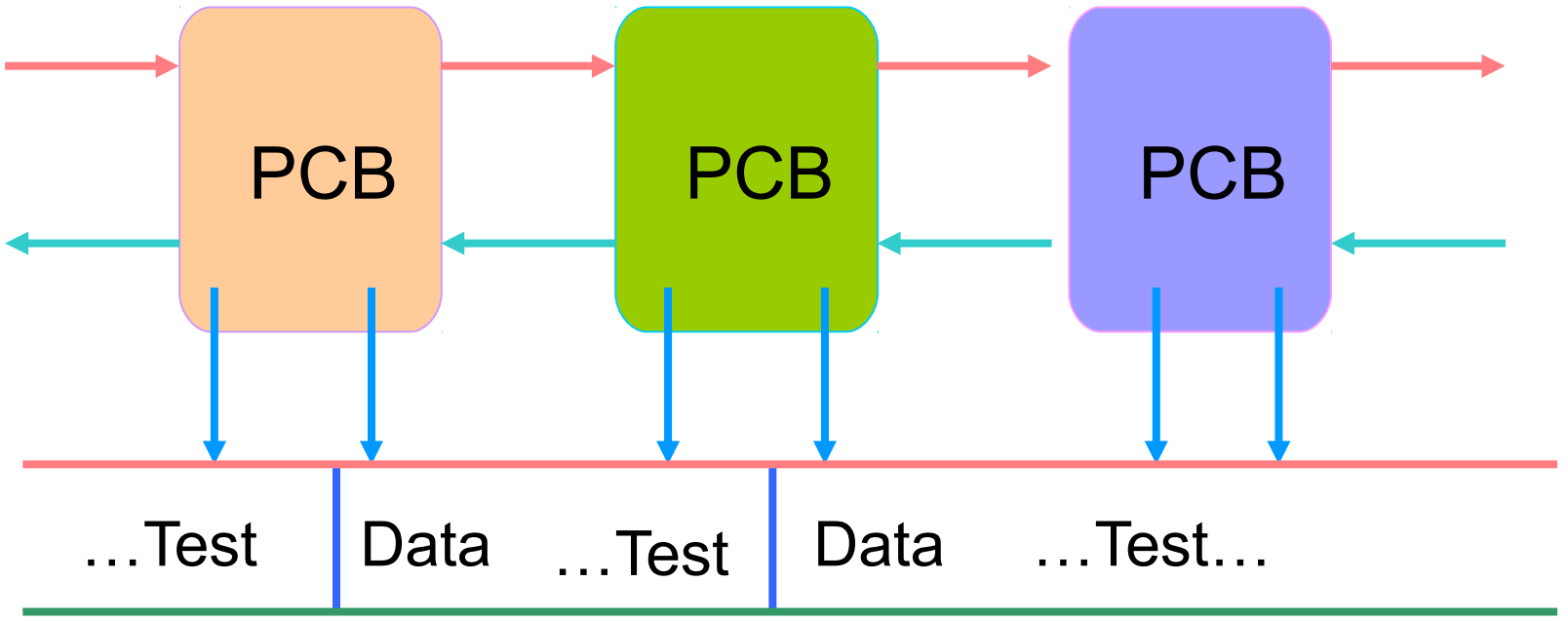
数据段 (data segment) 存放进程静态产生的数据结构

用户堆栈 (stack)

进程组成

进程表

进程表（process table）将系统中所有的 PCB 块联系起来。



进程组成

访问进程

Linux 中的 PCB 块又称为 task struct 结构，Linux 根据系统物理内存的大小限制已打开进程的总数目。系统每次访问一个进程时，内核根据 PID 在进程表中查找相应的进程 PCB 块（具体查找过程通过一个 PID 的 hash 表实现），再通过 PCB 块找到其对应的代码段与数据段，并进行操作。

C 语言程序

入口

crt0.o

主函数 int main(char *argc, char * argv[])

出口

正常出口：从 main 返回

调用 exit 退出 (库函数调用 <stdlib.h>)

调用 _exit 退出 (系统调用 , 立即中止进程 <unistd.h>)

异常出口：调用 abort 退出

信号中止

退出处理

很多时候我们需要在程序退出的时候做一些诸如释放资源的操作，但程序退出的方式有很多种，比如 main() 函数运行结束、在程序的某个地方用 exit() 结束程序、用户通过 Ctrl+C 或 Ctrl+break 操作来终止程序等等，因此需要有一种与程序退出方式无关的方法来进行程序退出时的必要处理。

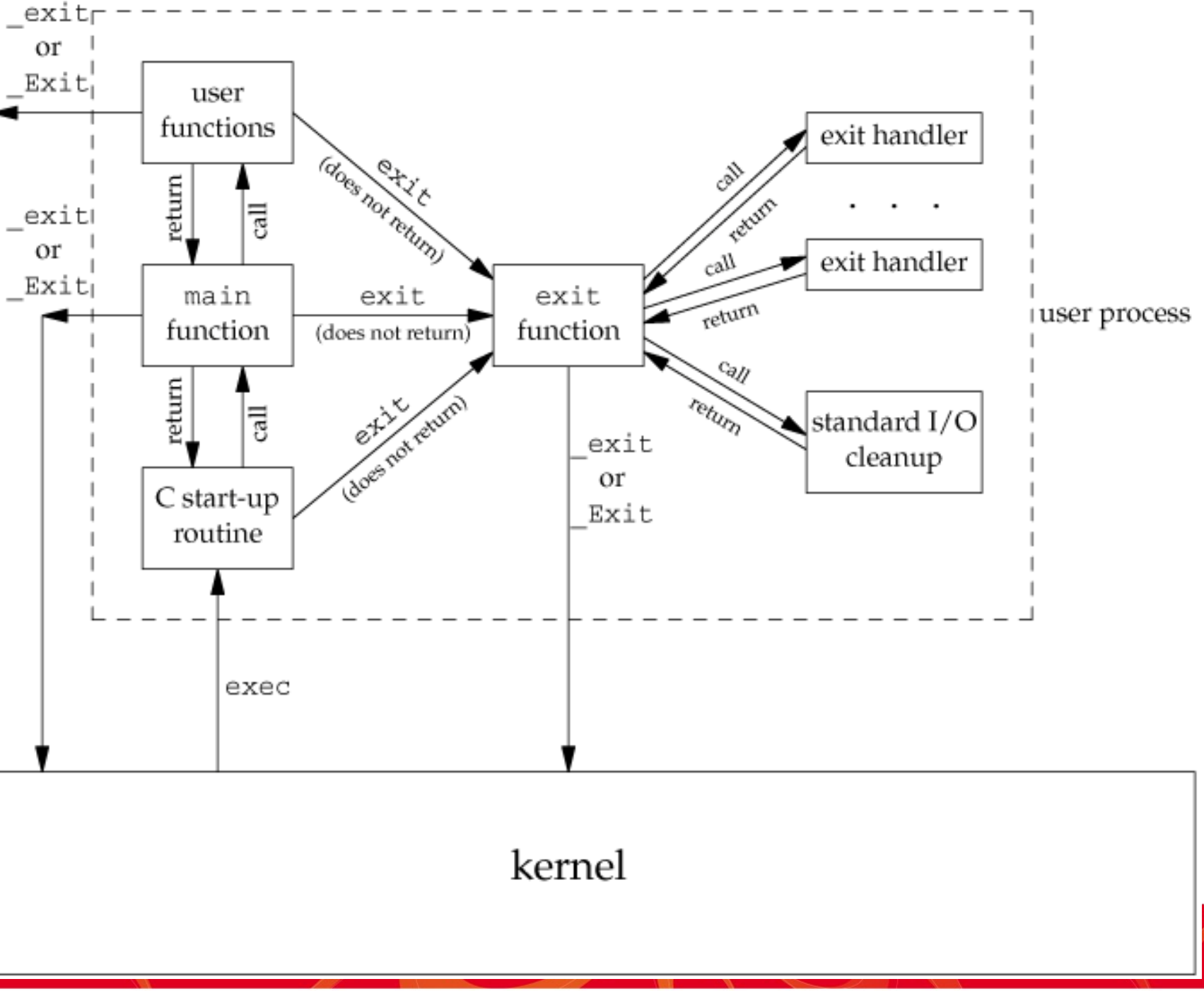
正常终止

用 atexit() 函数来注册程序正常终止时要被调用的函数。

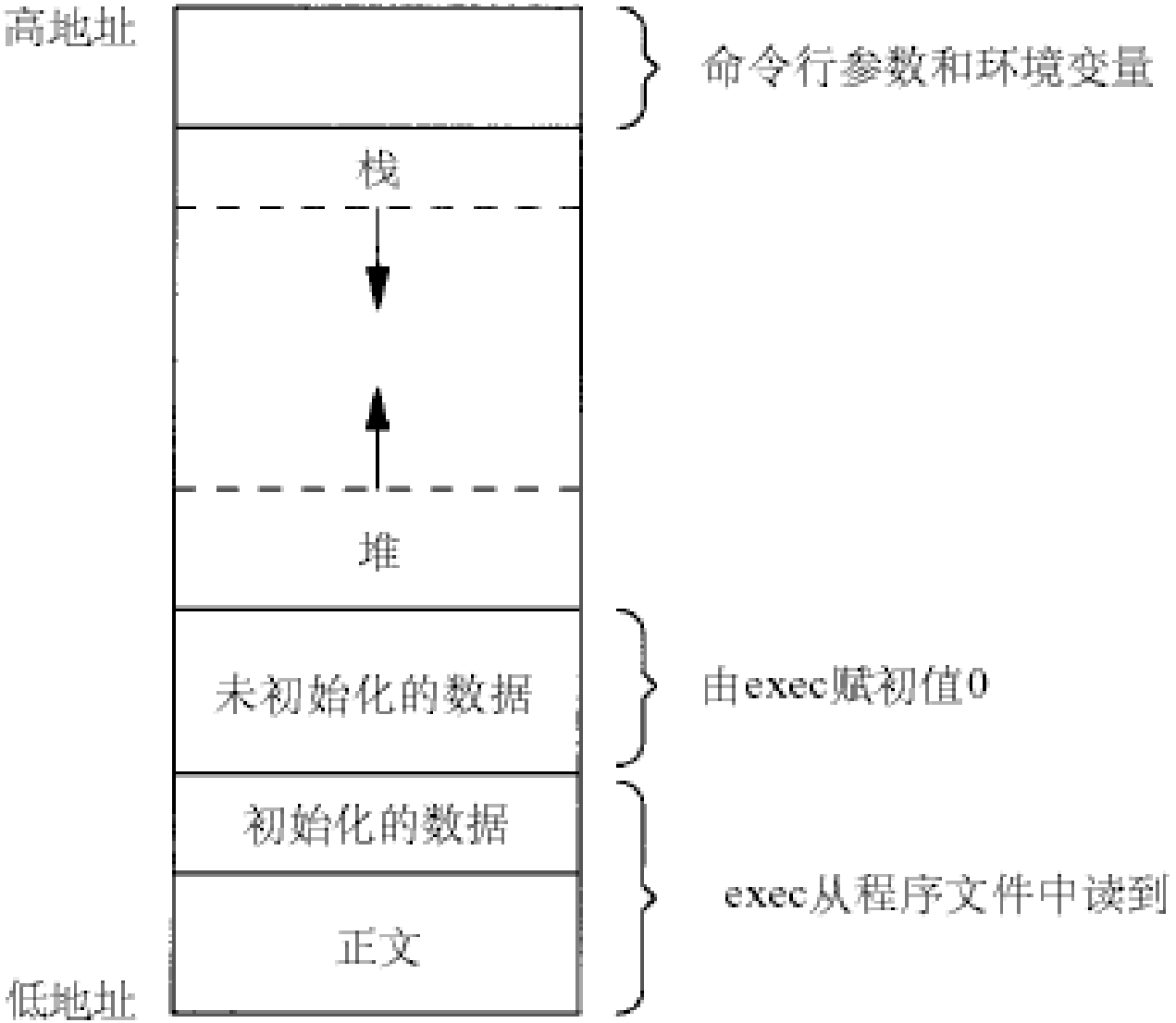
int atexit(void (*)(void));　　// 推荐使用

int on_exit(void (* function)(int, void*),void *arg); // 非标准，某些系统不支持，不推荐使用

C 语言程序


_exit
or
_Exit
user
functions
exit
(does not return)
exit handler
call
return
. . .
exit handler
_exit
or
_Exit
main
function
exit
(does not return)
exit
function
user process
return
call
call
return
return
call
exit
(does not return)
standard I/O
cleanup
_exit
or
_Exit
C start-up
routine
exec
kernel

C 语言程序



典型的存储器安排

进程标识

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);                    返回：调用进程的进程ID

pid_t getppid(void);                   返回：调用进程的父进程ID

uid_t getuid(void);                    返回：调用进程的实际用户ID

uid_t geteuid(void);                   返回：调用进程的有效用户ID

gid_t getgid(void);                    返回：调用进程的实际组ID

gid_t getegid(void);                   返回：调用进程的有效组ID
```

进程控制

系统调用 fork()

父子进程间同步

exec 函数簇

信号

fork

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

功能：创建一个新的进程。

返回：子进程中为 0 ，父进程中为子进程 ID ，出错为 -1

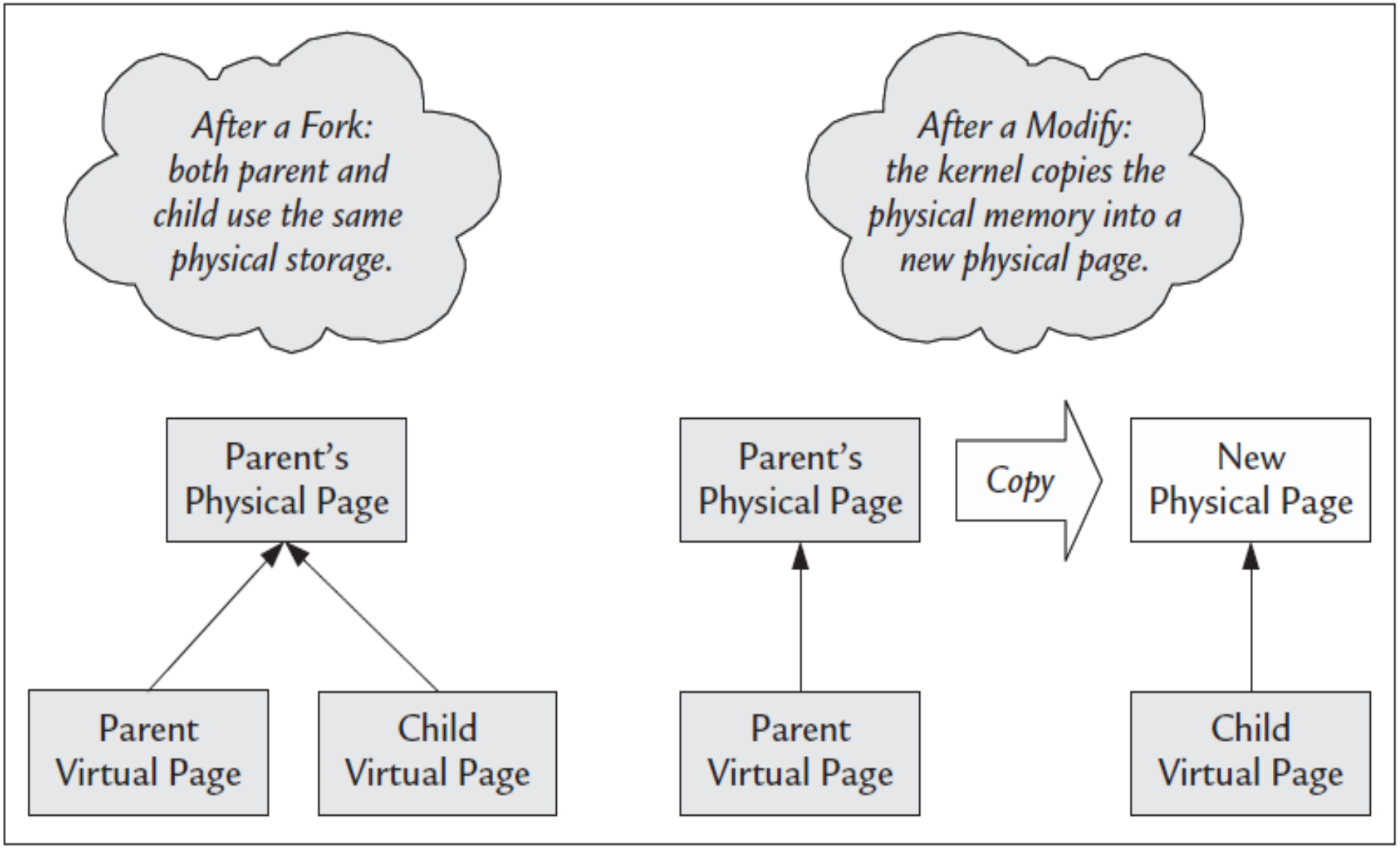
说明：

由 fork 创建的新进程被称为子进程（ child process ）。

该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是 0 ，而父进程的返回值则是子进程的进程 ID 。

一般来说，在 fork 之后是父进程先执行还是子进程先执行是不确定的。这取决于内核所使用的调度算法。

fork



Copy-on-Write Flag Triggers a Page Fault When Data Is Modified

fork

fork 后父子进程的异同

不同点：

fork 的返回值；

进程 ID 、不同的父进程 ID ；

父进程设置的锁，子进程不继承；

子进程的未决告警被清除；

子进程的未决信号集设置为空集。

相同点：

使用 fork 函数得到的子进程从父进程处继承了整个进程的地址空间，包括：进程上下文 (一般不包含执行代码)、进程堆栈、内存信息、打开的文件描述符、信号控制设置、进程优先级、进程组号、当前工作目录、根目录、资源限制、控制终端等。

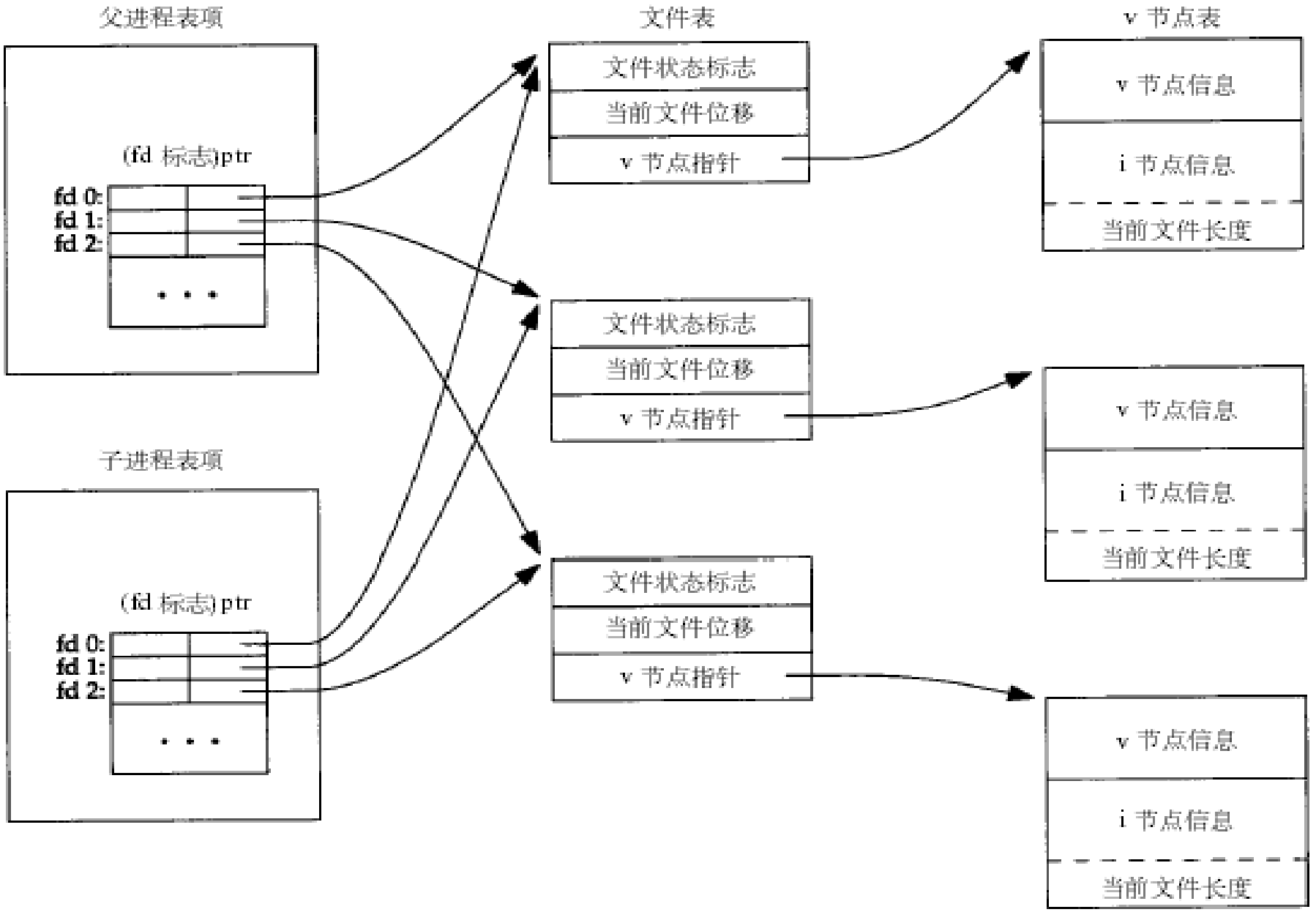
fork

练习：

调用 fork 创建进程，创建进程前分别用 write 和 printf 往 stdout 输出信息，程序中操作全局变量，打印父子进程号等信息。

运行时直接运行和运行并将结果定向到管道文件比较区别。

fork



fork之后父、子进程之间对打开文件的共享

vfork & clone

vfork

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork(void);
```

clone

```
#include <sched.h>
int clone(int (*fn)(void *), void
*child_stack, int flag, void *arg);
```

vfork

pid_t vfork(void);

vfork 系统调用不同于 fork ，用 vfork 创建的子进程共享地址空间，也就是说子进程完全运行在父进程的地址空间上，子进程对虚拟地址空间任何数据的修改同样为父进程所见。但是用 vfork 创建子进程后，父进程会被阻塞直到子进程调用 exec 或 exit 。这样的好处是在子进程被创建后仅仅是为了调用 exec 执行另一个程序时，因为它就不会对父进程的地址空间有任何引用，所以对地址空间的复制是多余的 ，通过 vfork 可以减少不必要的开销。

clone

int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);

clone 可以让你有选择性的继承父进程的资源，你可以选择象 vfork 一样和父进程共享一个虚存空间，你也可以不和父进程共享，你甚至可以选择创造出来的进程和父进程不再是父子关系，而是兄弟关系。

返回：成功返回子进程 ID ，失败返回 -1 。

参数

fn 是函数指针，指向创建进程要执行的程序

child_stack 为要创建的进程分配系统堆栈空间（在 linux 下系统堆栈空间是 2 页面， 8KB 的内存，其中在这块内存中，低地址上放入了进程控制块 task_struct 的值）

flags 标志用来描述需要从父进程继承那些资源

arg 是传给子进程的参数

父子进程简单的同步

如果父进程没有在等待子进程退出，而子进程又退出了，这个子进程就会成为僵尸 (zombie) 进程 (既不是活的，也不是死的)，僵尸进程浪费了系统资源。

如果父进程先于子进程退出，已经在运行的子进程会被视为继承自 init 进程。

避免僵尸进程的方法是父进程调用 wait 函数等待子进程结束；或是让父进程忽略子进程产生的退出信号。

wait 和 waitpid

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int * status) ;
pid_t waitpid(pid_t pid, int * status, int options) ;
```

功能：等待进程。若成功则返回子进程 ID 号，若出错则返回 -1 。

参数 status ： 用于存放进程结束状态。

pid ： 要等待的进程 ID 。

pid == -1 等待任一子进程。于是在这一功能方面 waitpid 与 wait 等效。

pid > 0 等待其进程 ID 与 PID 相等的子进程。

pid == 0 等待其组 ID 等于调用进程的组 ID 的任一子进程。

pid < -1 等待其组 ID 等于 PID 的绝对值的任一子进程。

options ： 设置等待方式。

0 ： 不设置。

WNOHANG ： 如果没有任何已经结束的进程则马上返回，不等待。

WUNTRACED ： 如果子进程进入暂停状态则马上返回。

wait 和 waitpid

调用 wait 或 waitpid 的进程可能会：

阻塞（如果其所有子进程都还在运行）。

带子进程的终止状态立即返回（如果一个子进程已终止，正等待父进程存取其终止状态）。

出错立即返回（如果它没有任何子进程）。

exec 函数簇

在用 vfork 函数创建子进程后，子进程往往要调用一个 exec 函数以执行另一个程序。

当进程调用一种 exec 函数时，该进程完全由新程序代换，而新程序则从其 main 函数开始执行。因为调用 exec 并不创建新进程，所以前后的进程 ID 并未改变。exec 只是用另一个新程序替换了当前进程的正文、数据、堆和栈段。

exec 函数簇

```
#include <unistd.h>
int execl(const char * pathname, const char * arg 0, ... /* (char *) 0 */);
int execv(const char * pathname, char *const a rgv [] );
int execle(const char * pathname, const char * a rg 0, .../* (char *)0, char *const e n v p [] */);
int execve(const char * pathname char *const a rgv [], char *const envp [] );
int execlp(const char * filename, const char * a rg 0, ... /* (char *) 0 */);
int execvp(const char * filename, char *const a rgv [] );
```

功能：实现代码替换

返回值：若出错则为 - 1 ，若成功替换新代码。

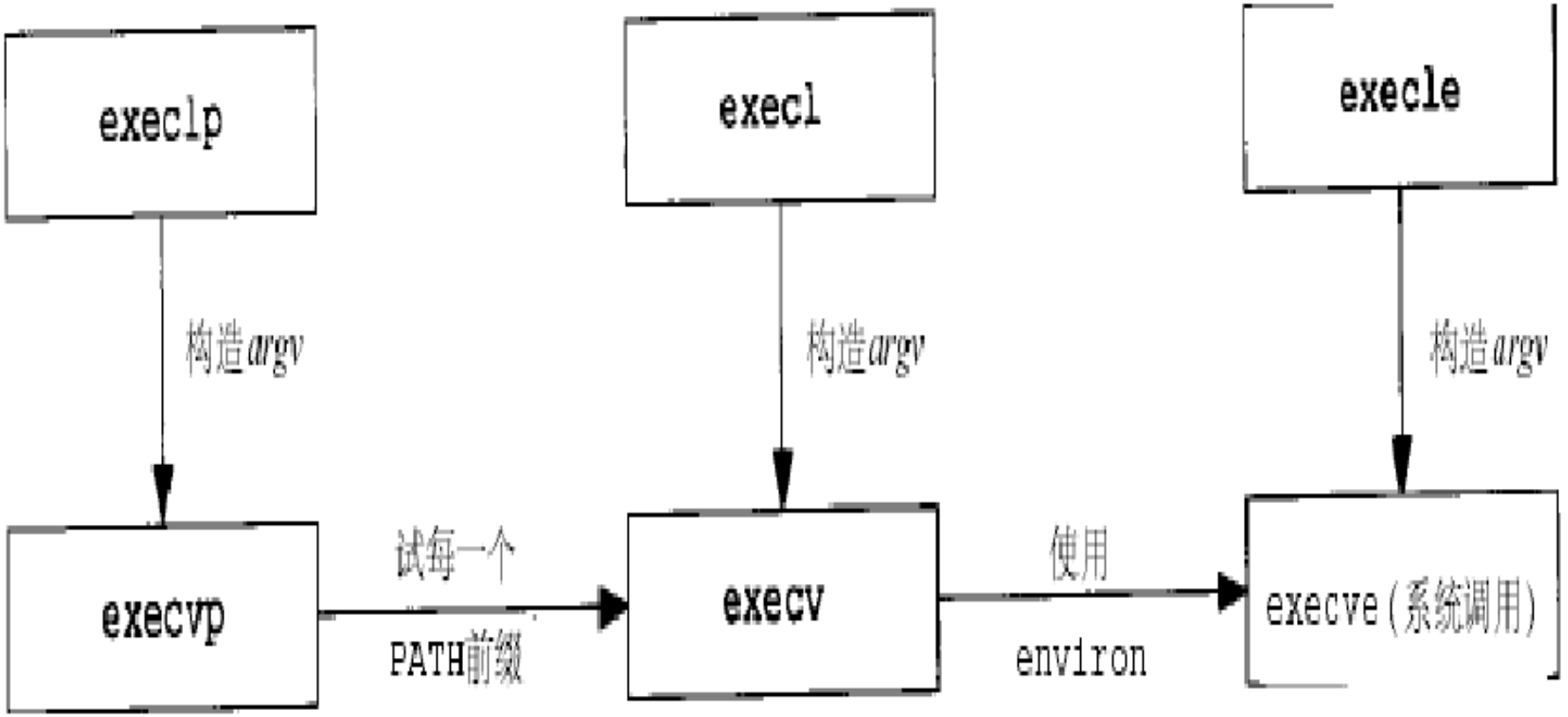
- E: 指可以传递环境变量表
- L: 单独的参数传递，最后要有一个 NULL
- V: 传一个指针数组名
- P: 按照环境变量来查找

exec 函数簇

六个exec函数之间的区别

函　数	*pathname*	*filename*	参 数 表	*argv*[]	environ	*envp*[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(字母表示)		p	l	v		e

exec 函数簇



六个exec函数之间的关系

exec 函数簇

应用举例

```
char *ps_argv[]={ “ps” ,” -ax” , NULL};
char *ps_envp[]={ “PATH=/bin:/usr/bin” ,” TERM=console” , NULL}
execl( “/bin/ps” , “ps” , “-ax” , NULL);
execv( “/bin/ps” , ps_argv);
execle( “/bin/ps” , “ps” , “-ax” , NULL, ps_envp);
execve( “/bin/ps” , ps_argv, ps_envp);
execlp( “ps” , “ps” , “-ax” , NULL);
execvp( “ps” , ps_argv);
```

信号

信号时 GNU/Linux 中进程的回调符号，可以为某个进程注册为在某事件发生时接收信号，或是在某个默认操作退出时忽略信号。

信号是软件中断。信号 (signal) 机制是 Unix 系统中最为古老的进程之间的通信机制。它用于在一个或多个进程之间传递异步信号。

产生信号：

当用户按某些终端键时，可以产生信号。例如：在终端上按 Ctrl-C 键通常产生中断信号（SIGINT）。这是停止一个已失去控制程序的方法。

硬件异常产生信号：除数为 0、无效的存储访问等等。这些条件通常由硬件检测到，并将其通知内核。然后内核为该条件发生时正在运行的进程产生适当的信号。例如，对执行一个无效存储访问的进程产生一个 SIGSEGV 。

Linux 中的信号

```
[root@baozong code]# 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
```

Linux 中的信号

名称	说明
SIGABRT	进程异常终止（调用 abort 函数产生此信号）
SIGALRM	超时（ alarm ）
SIGFPE	算术运算异常（除以 0 ， 浮点溢出等）
SIGHUP	连接断开
SIGILL	非法硬件指令
SIGINT	终端终端符 (Clt-C)
SIGKILL	终止（不能被捕捉或忽略）
SIGPIPE	向没有读进程的管道写数据
SIGQUIT	终端退出符 (Clt-\)
SIGTERM	终止（由 kill 命令发出的系统默认终止信号）
SIGUSR1	用户定义信号
SIGUSR2	用户定义信号

Linux 中的信号

名称	说明
SIGSEGV	无效存储访问（段违例）
SIGCHLD	子进程停止或退出
SIGCONT	使暂停进程继续
SIGSTOP	停止（不能被捕捉或忽略）
SIGTSTP	终端挂起符 (Clt-Z)
SIGTTIN	后台进程请求从控制终端读
SIGTTOUT	后台进程请求向控制终端写

信号处理

可以要求系统在某个信号出现时按照下列三种方式中的一种进行操作。

(1) **忽略此信号**。大多数信号都可使用这种方式进行处理，但有两种信号却决不能被忽略。它们是： SIGKILL 和 SIGSTOP 。这两种信号不能被忽略的原因是：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），则进程的行为是未定义的。

(2) **捕捉信号**。为了做到这一点要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。如果捕捉到 SIGCHLD 信号，则表示子进程已经终止，所以此信号的捕捉函数可以调用 waitpid 以取得该子进程的进程 ID 以及它的终止状态。

(3) **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。

信号处理函数

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

功能：为信号号为 signum 的信号安装处理函数

返回：成功返回之前的处理句柄，失败返回 SIG_ERR

参数： handler 为用户指定的处理函数，或者是

SIG_DEF 默认处理函数

SIG_IGN 忽略

信号处理函数

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

功能：发送信号到某进程

返回：成功为 0 ，失败为 -1

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

功能：发送信号到当前进程

返回：成功为 0 ，失败为 -1

信号处理函数

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

功能：在 secends 秒后发送 SIGALARM 信号。若 secends 为 0 ，则取消已设置闹铃。

返回：返回之前闹钟的剩余秒数，如果之前未设闹钟则返回 0 。

#include <unistd.h>

int pause(void);

功能：挂起进程直到接收到一个信号

返回： pause 返回 -1 ，并将 errno 设置为 EINTR

系统命令

ps 命名

提供活动在当前系统上的当前进程的快照

top 命令

实时列出指定 CPU 上进程的活动

kill 命令

向一个进程发出信号

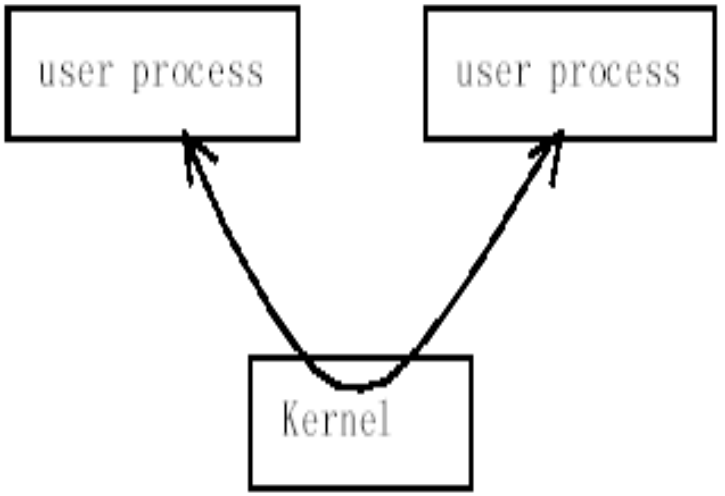
进程间通信概述

进程是相互独立的，进程间的通信需要专门的机制。

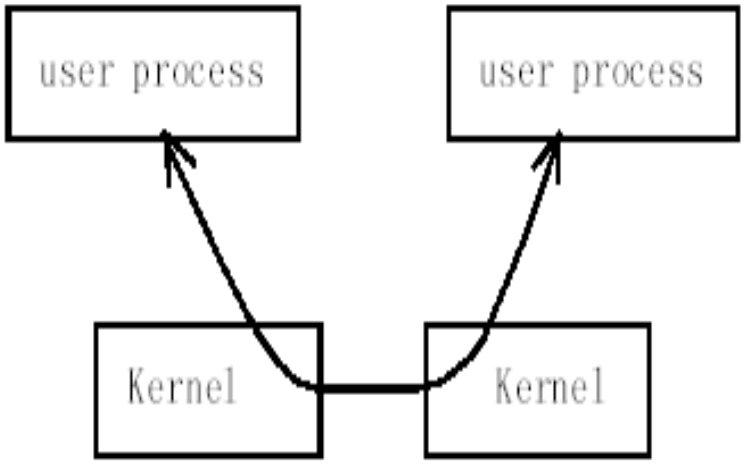
进程之间的通信可以经由文件系统，但实际使用较为复杂（例如，需要锁机制）。

UNIX IPC (InterProcess Communication) 机制是各种进程通信方式的统称。

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。



IPC on one host



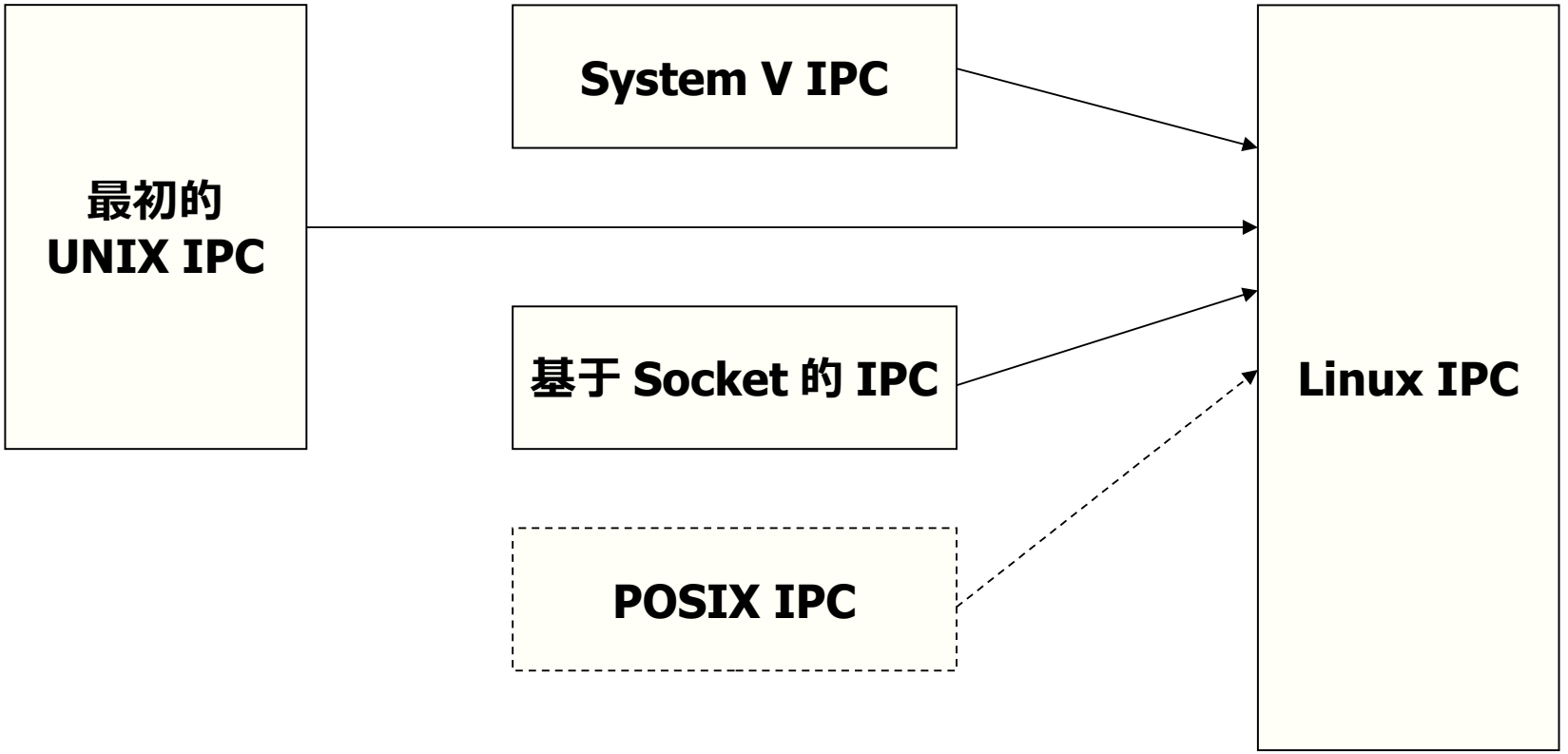
IPC on different hosts:
Network Programming

进程间通信概述

对于 UNIX 的发展，贝尔实验室和 BSD 在进程间通信方面的侧重点有所不同：

- 贝尔实验室对 Unix 早期的进程间通信手段进行了系统的改进和扩充，形成了“ System V IPC”，通信进程局限在单个计算机内；
- BSD 则主要考虑跨计算机的进程间通信，形成了基于套接口（ socket ）的进程间通信机制。

进程间通信概述

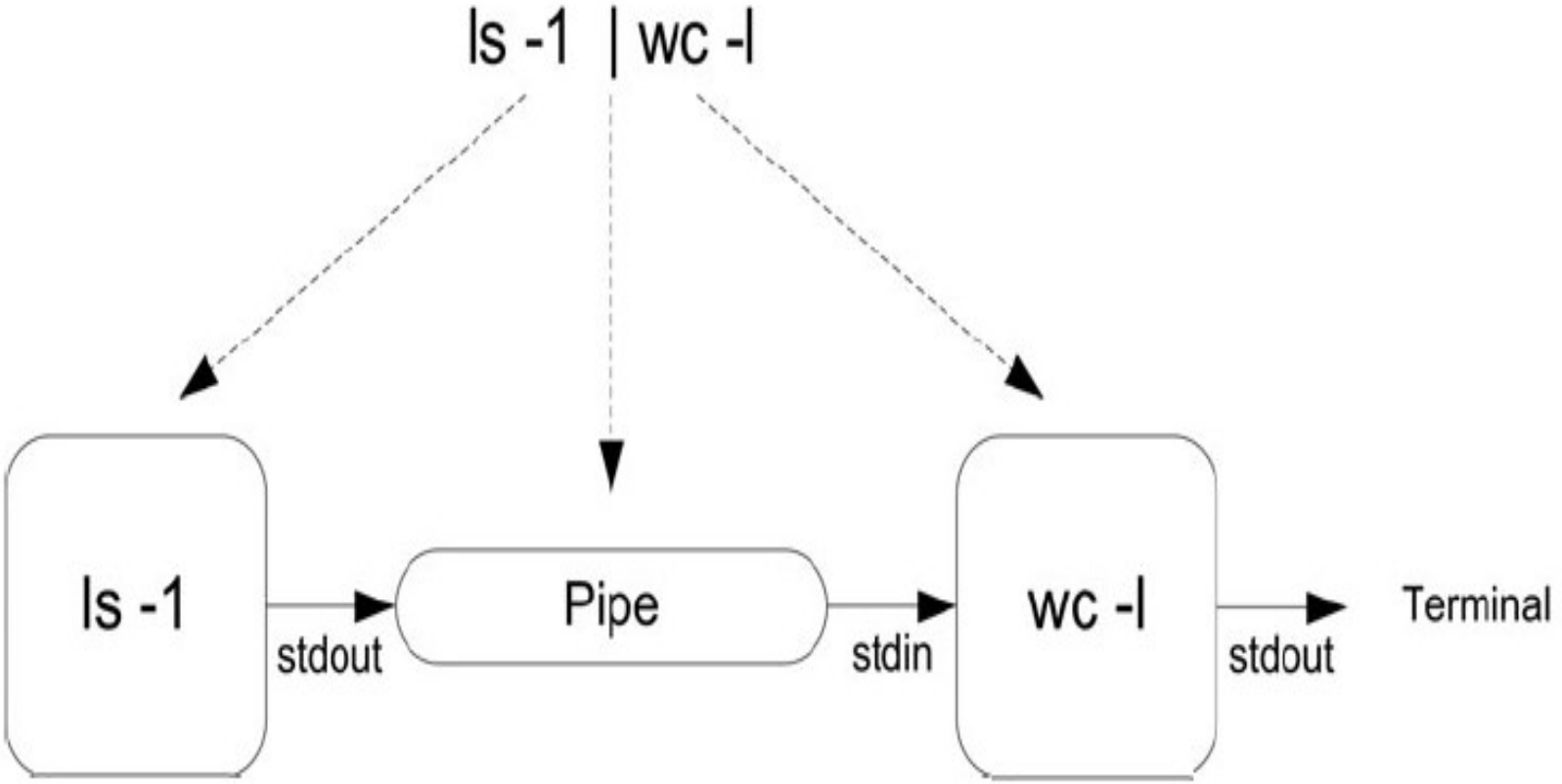


- 最初的 Unix IPC：信号、管道、 FIFO；
- System V IPC： 消息队列、旗语(信号量)、共享内存区；
- POSIX IPC： 消息队列、旗语(信号量)、共享内存区。

管道编程

管道模型是一个古老但仍然有效的进程间通信机制。它提供半双工管道的功能，提供先入先出（FIFO）队列以供进程间的通信。

命令行管道： ls -l | wc -l



管道编程

管道是单向的、先进先出的、无结构的、固定大小的字节流，它把一个进程的标准输出和另一个进程的标准输入连接在一起。写进程在管道的尾端写入数据，读进程在管道的首端读出数据。数据读出后将从管道中移走，其它读进程都不能再读到这些数据。管道提供了简单的流控制机制。进程试图读空管道时，在有数据写入管道前，进程将一直阻塞。同样，管道已经满时，进程再试图写管道，在其它进程从管道中移走数据之前，写进程将一直阻塞。

管道通信

管道有一些固有的局限性：

- 因为读数据的同时也将数据从管道移去，因此，管道不能用来对多个接收者广播数据。
- 管道中的数据被当作字节流，因此无法识别信息的边界。
- 如果一个管道有多个读进程，那么写进程不能发送数据到指定的读进程。同样，如果有多个写进程，那么没有办法判断是它们中那一个发送的数据。

匿名管道

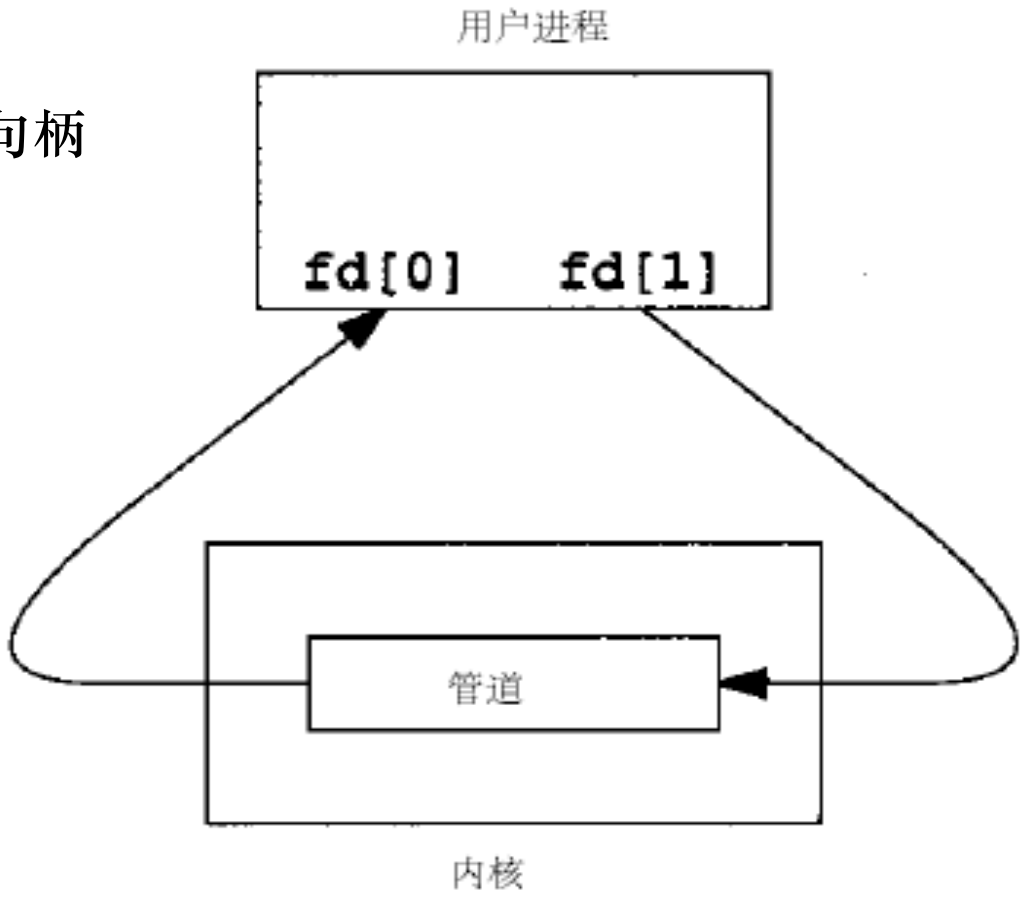
匿名管道提供了一个进程与其相同先祖子进程通信的方法。

```
#include <unistd.h>
int pipe(int filedes[2]);
```

成功返回 0 ，失败返回 -1

filedes[0]: 为读句柄 , filedes[1]: 为写句柄

注：管道只不过是一对文件描述符，所以所有能操作文件描述符的函数都能用于管道，如 read, write, fnctl, select 等



匿名管道

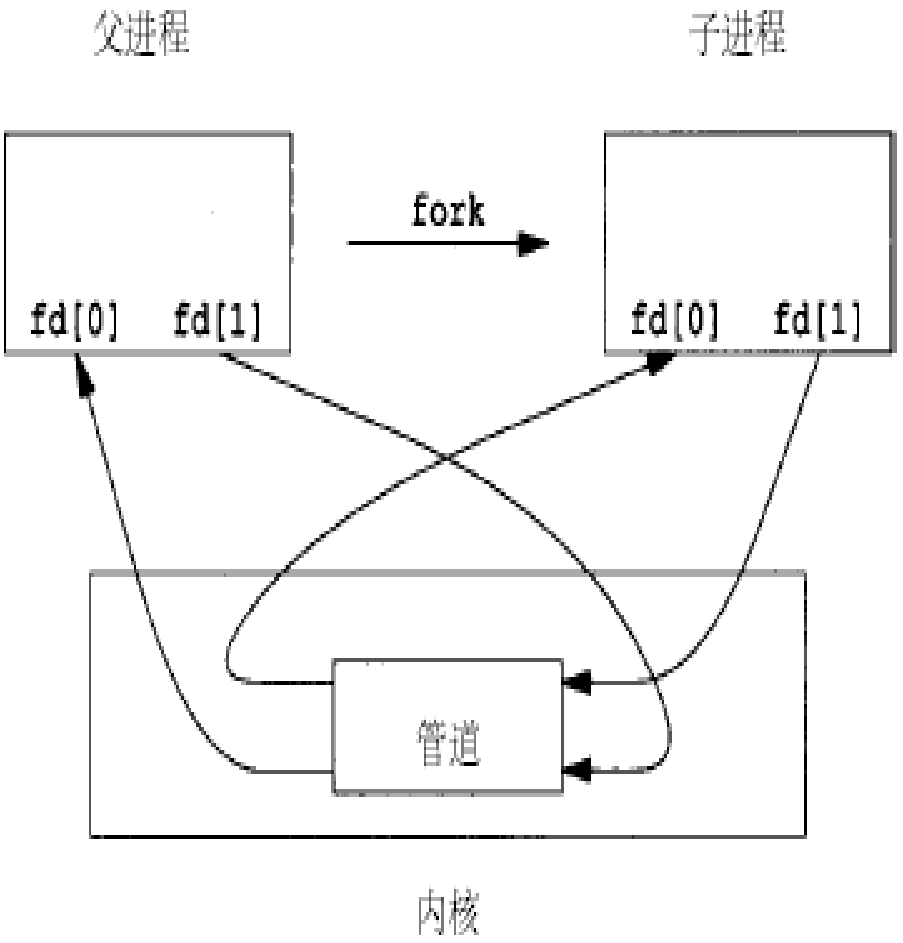


图14-2 fork之后的半双工管道

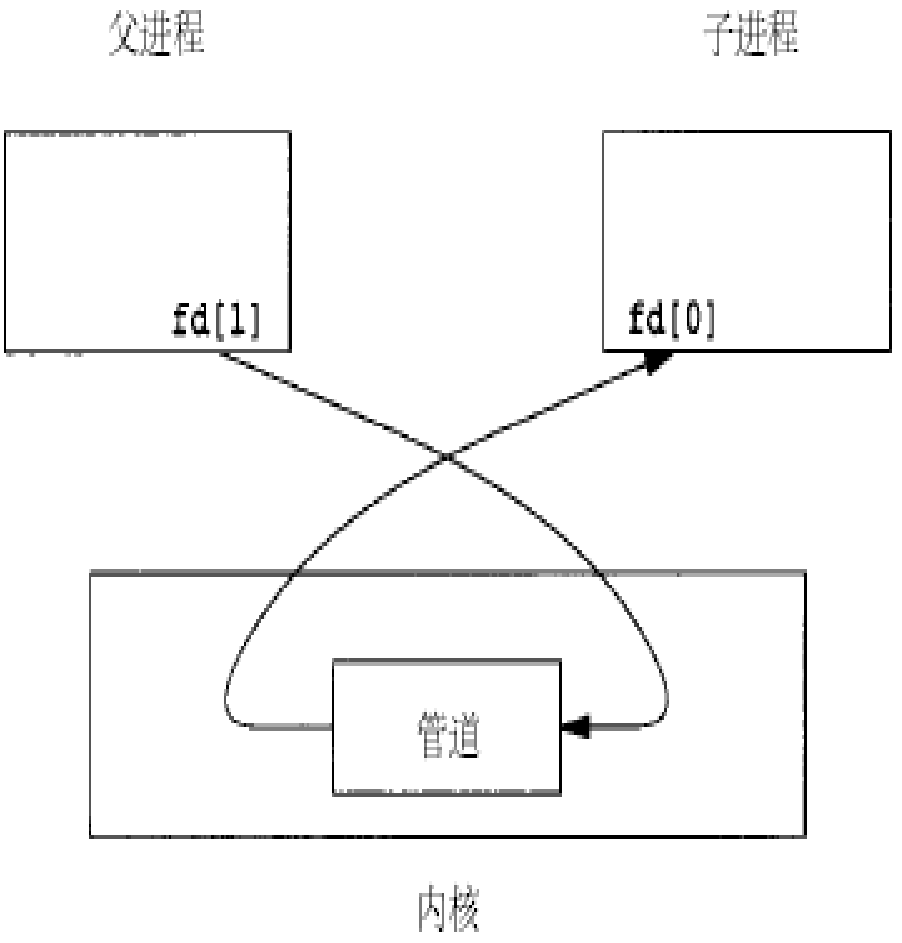


图14-3 从父进程到子进程的管道

匿名管道

写管道时，常数 PIPE_BUF 规定了内核中管道缓存器的大小

管道的一端关闭时：

写端关闭，读该管道在所有数据都被读取后， read 返回 0 ，表示达到了文件结束

读端关闭，写该管道产生信号 SIGPIPE

匿名管道

#include <unistd.h>

int dup(int oldfd);

int dup2(int oldfd, int targetfd);

函数 dup 和 dup2 提供了复制文件描述符的功能，常用语 stdin, stdout, stderr 的重定向。

匿名管道

#include <unistd.h>

in

in

函

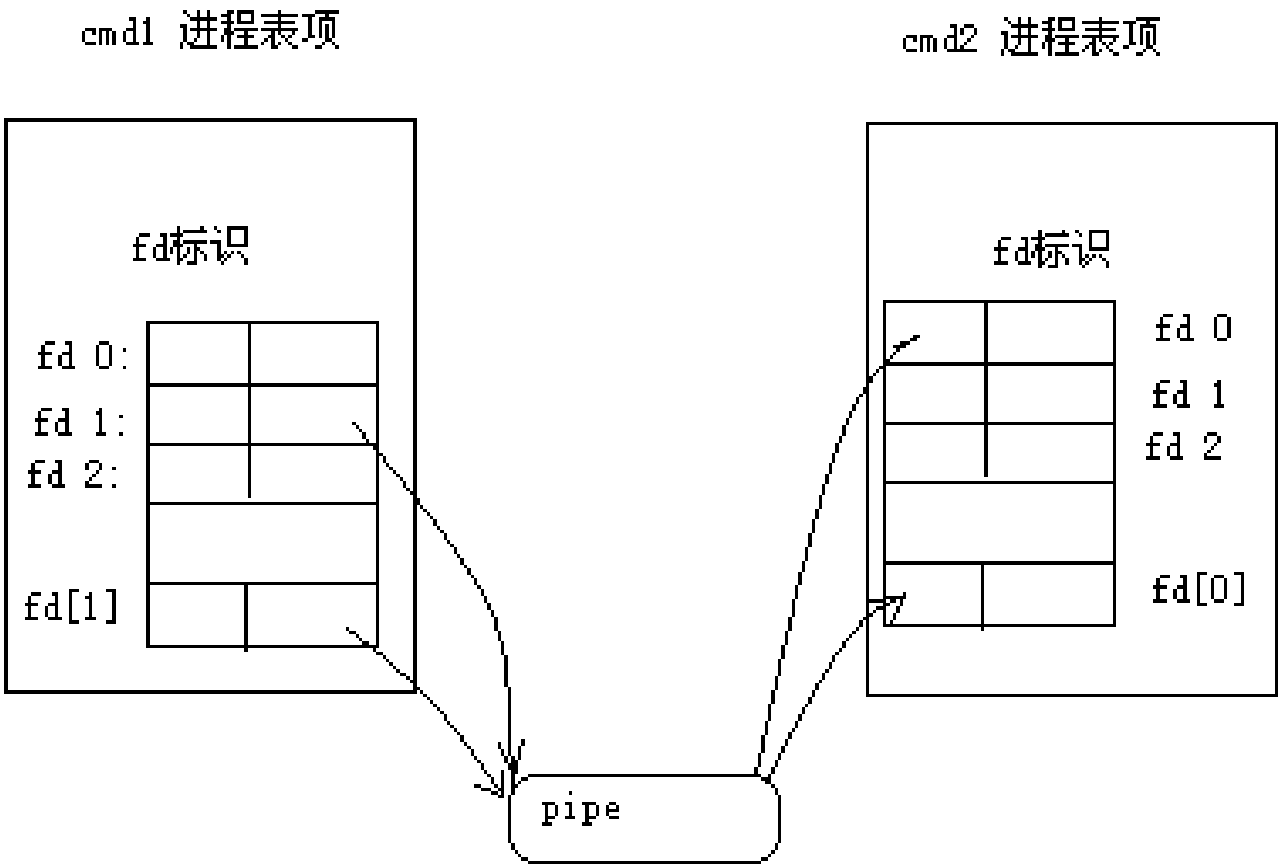
```
    int fd1, fd2;
    ...
fd2 = dup( fd1 );
```

```
 int oldfd;
oldfd = open("app_log", (O_RDWR | O_CREATE), 0644 );
dup2( oldfd, 1 );
close( oldfd );
```

练习

程序完成 ls -l | wc -l 功能

pipe, fork, close, dup2, execlp



命名管道

#include <sys/types.h>

#include <sys/stat.h>

int mkfifo(const char * pathname, mode_t mode);

函数 mkfifo 用于在文件系统中创建一个文件，提供 FIFO 功能。因为其在文件系统中可见，因此可以用于任何进程之间通信。成功返回 0 ，失败返回 -1 。

pathname 须为文件系统中不存在的文件， mode 可设为如 S_ISFIFO|0666

命名管道

FIFO 的同步与读写

打开 FIFO 时的同步 （如： open ）

一般情况下（没有说明 O_NONBLOCK ），只读打开要阻塞到某个其它进程为写打开此 FIFO ；类似的，为写打开一个 FIFO 要阻塞到某个其它进程为读而打开它。

如果指定了 O_NONBLOCK ，则只读打开立即返回；只写打开也立即返回，但如果没有进程已经为读而打开此 FIFO ，那么 open 将出错返回 -1 ， errno 置为 ENXIO 。

读写 FIFO 时的同步（如： read ， write ）

同 pipe

命名管道

命令行命名管道

mkfifo cmd_fifo

cat cmd_fifo　　//tty1

echo hi > cmd_fifo　　//tty2

应用举例： C/S 模式应用 client.c, server.c

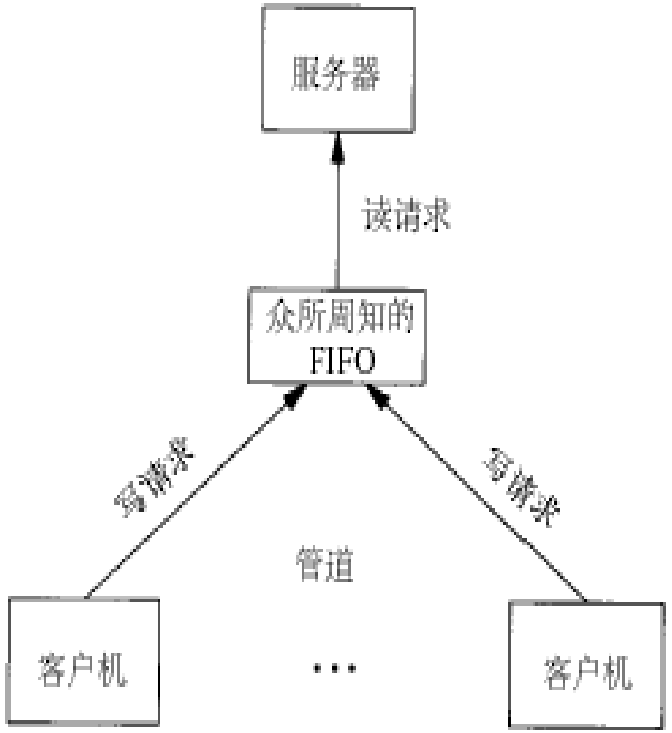


图14-11　客户机用FIFO向服务器发送请求

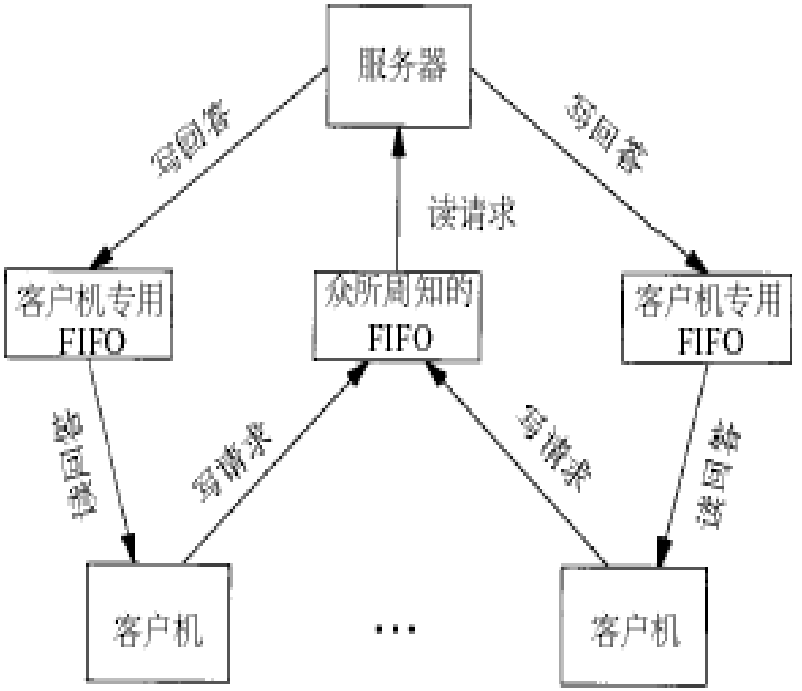


图14-12　客户机-服务器用FIFO进行通信

System V IPC

IPC objects

消息队列 (message queue)

旗语 (信号量)(semaphore set)

共享内存 (shared memory)

shell 命令

ipcs, ipcrm

System V IPC 的共同特征

标识符与关键字

引用 IPC 对象：标识符

创建 IPC 对象时指定关键字 (key_t key;)

key 的选择；预定义常数 IPC_PRIVATE ； ftok 函数

内核将关键字转换成标识符

许可权结构

和文件类比

struct ipc_perm

System V IPC 的共同特征

创建 System V IPC 时访问授权

S_IRUSR	用户读
S_IWUSR	用户写
S_IRGRP	组读
S_IWGRP	组写
S_IROTH	其它人读
S_IWOTH	其它人写

创建 System V IPC 时指令

IPC_CREAT	key 不存在则创建
IPC_EXCL	key 存在则失败
IPC_NOWAIT	请求的 IPC 须等待则出错

System V IPC 的共同特征

#include <sys/ipc.h>

key_t ftok(char * fname, int id)

系统建立 IPC 通信必须指定一个 ID 值。通常情况下，该 id 值通过 ftok 函数得到。

fname 为指定的文件名 (该文件必须是存在而且可以访问的), id 是子序号，虽然为 int ，但是只有 8 个比特被使用 (0-255) 。

当成功执行的时候，一个 key_t 值将会被返回，否则返回 -1 。

在一般的 UNIX 实现中，是将文件的索引节点号取出，前面加上子序号得到 key_t 的返回值。如指定文件的索引节点号为 65538 ，换算成 16 进制为 0x010002 ，而你指定的 ID 值为 38 ，换算成 16 进制为 0x26 ，则最后的 key_t 返回值为 0x26010002 。

查询文件索引节点号的方法是： ls -i

System V IPC 的共同特征

设置 System V IPC 时的控制命令

IPC_RMID	删除标识符指示的 IPC 量
IPC_SET	设置选项
IPC_STAT	读取选项

ipc_perm 结构

```
struct ipc_perm
{
  uid_t  uid;          /* Owner's user ID. */
  gid_t  gid;          /* Owner's group ID. */
  uid_t  cuid;         /* Creator's user ID. */
  gid_t  cgid;         /* Creator's group ID. */
  mode_t mode;         /* Read/write permission. */
  key_t  key;
  unsigned short seq;
};
```

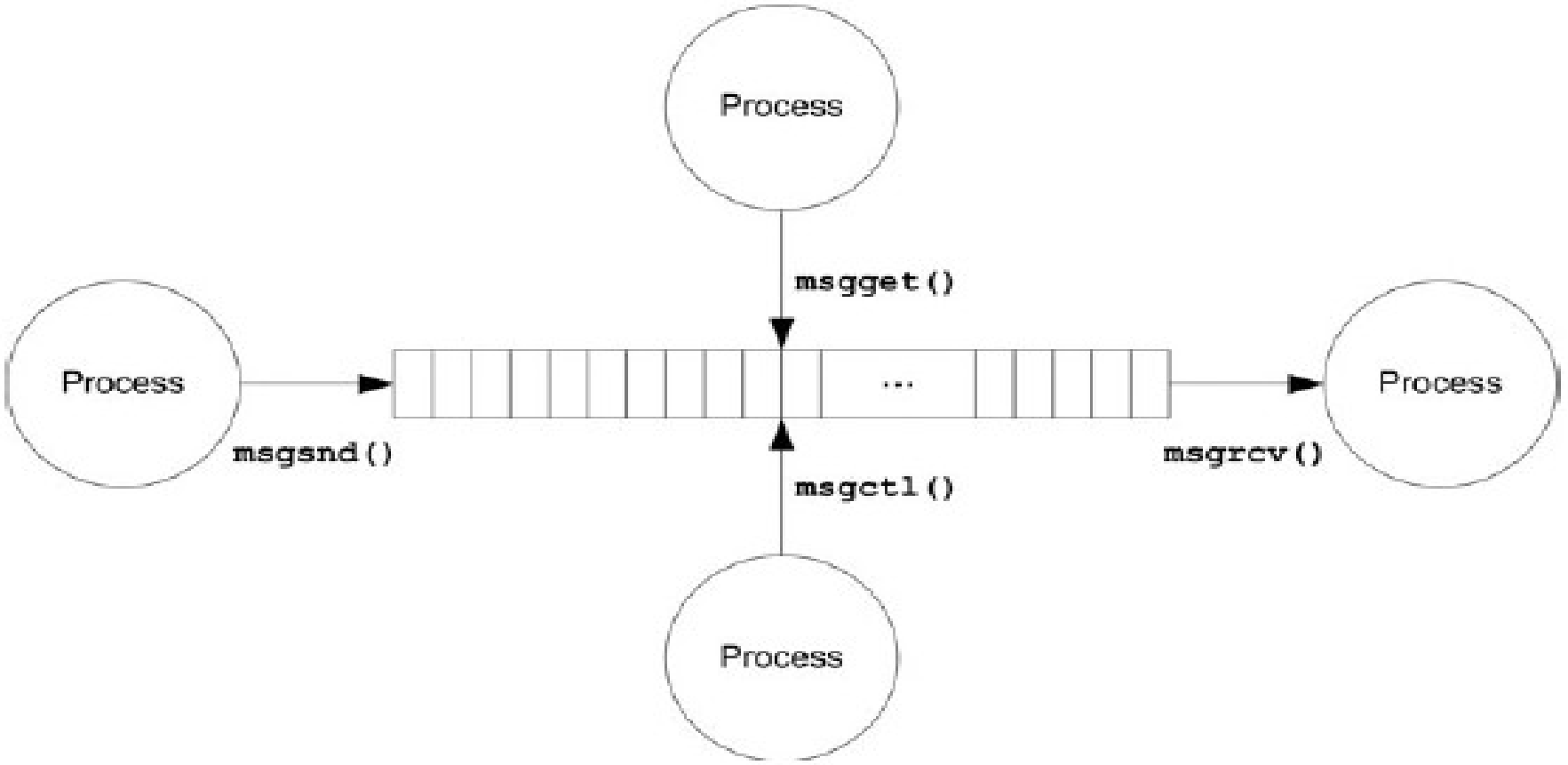
System V IPC 函数概览

功能	消息队列	信号量	共享内存
分配一个 IPC 对象，获得对 IPC 的访问	msgget	semget	shmget
IPC 操作：发送 / 接收消息，信号量操作，连接 / 释放共享内存	msgsnd/ msgrcv	semop	shmat/ shmdt
IPC 控制：获得 / 修改状态信息，取消 IPC	msgctl	semctl	shmctl

消息队列 IPC

消息队列就是消息的一个链表，它允许一个或多个进程向它写消息，一个或多个进程从中读消息。这些消息存在于内核中，由“队列 ID”来标识。

消息队列的实现包括创建和打开队列、添加消息、读取消息和控制消息队列这四种操作



消息队列 IPC

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
 int msgget(key_t key, int msgflg);
```

功能：创建一个新的消息队列，或者希望获取一个已经存在的消息队列 (msgflg = 0) 。

返回值：成功返回消息队列标识符；失败返回 -1 。

参数：

key ： 消息队列键值 / 名字。

msgflg ： 指令和访问权限标志，包括：

IPC_CREAT 如果内核中没有此队列，则创建它。

IPC_EXCL 当和 IPC_CREAT 一起使用时，如果队列已经存在，则失败。

打开权限类似 open(), 相当于文件的访问权限。

消息队列 IPC

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd ( int msgid, struct msgbuf *msgp, int msgsz, int msgflg );
```

功能：往队列中发送一条消息 。

返回值：成功返回 0 ，错误返回 -1 。

参数：

msgid ： 是消息队列标识符 , 它是由系统调用 msgget 返回的 .

msgp ： 是指向消息缓冲区的指针 .

msgsz ： 中包含的是消息的字节大小 , 但不包括消息类型的长度 (4 个字节).

msgflg ： 可以设置为 0(此时为忽略此参数), 或者使用 IPC_NOWAIT.

如果消息队列已满 , 那么此消息则不会写入到消息队列中 , 控制将返回到调用进程中 . 如果没有指明 , 调用进程将会挂起 , 直到消息可以写入到队列中。

消息队列 IPC

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv(int msgid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg);
```

功能：读取消息，从消息队列中取走消息。

返回值：成功返回 0 ，错误返回 -1 。

参数：

msgid ： 用来指定将要读取消息的队列 .

msgp ： 代表要存储消息的消息缓冲区的地址 .

msgsz ： 是消息缓冲区的长度 , 不包括 mtype 的长度 , 它可以按照如下的方法计算 :

msgsz = sizeof(struct mymsgbuf)-sizeof(long);

mtype ： 是要从消息队列中读取的消息的类型 . 如果此参数的值为 0, 那么队列中最长时间的一条消息将返回 , 而不论其类型是什么 .

msgflg ： 没有消息可接收时，如果该参数中 IPC_NOWAIT 被设置，那么立刻返回 -1 ；IPC_NOWAIT 被清除则挂起。

消息队列 IPC

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl ( int msgid, int cmd, struct msqid_ds *buf );
```

功能：对消息队列的操作。

返回值：成功返回 0，错误返回 -1 。

参数：

msgid: 消息队列 ID.

cmd: IPC_RMID 从系统内核中移走消息队列，IPC_STAT 读取消息队列当前设置，IPC_SET 修改消息队列的设置。

buf: 队列状态数据结构。

旗语同步

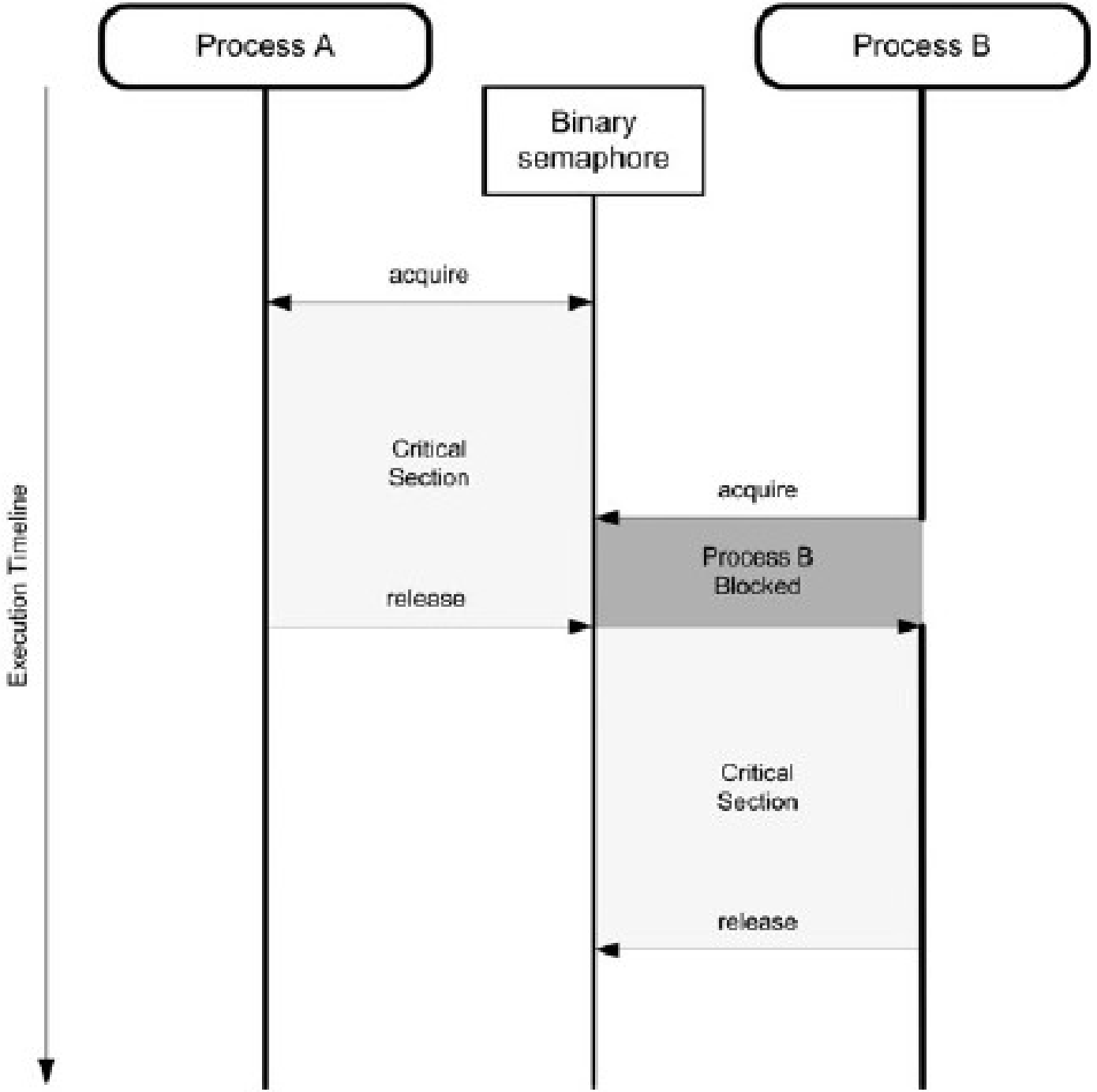
旗语是一个受保护的变量，它可以提供对两个以上进程共享资源限制访问的方法。旗语允许两个操作，称为获取与释放。获取操作允许进程取得旗语，如果旗语已被其它进程获取，则阻塞直到旗语可用为止，进程使用完旗语后需释放旗语，以使其它进程可获取它。释放旗语会自动唤醒下一个等待获取旗语的进程。

旗语由 Edsger Dijkstra 为 T.H.E. 操作系统引入的。最初被定义为 P 和 V 。 P 是荷兰语 proberen(尝试), V 是 verhogen(增加) 。

旗语同步

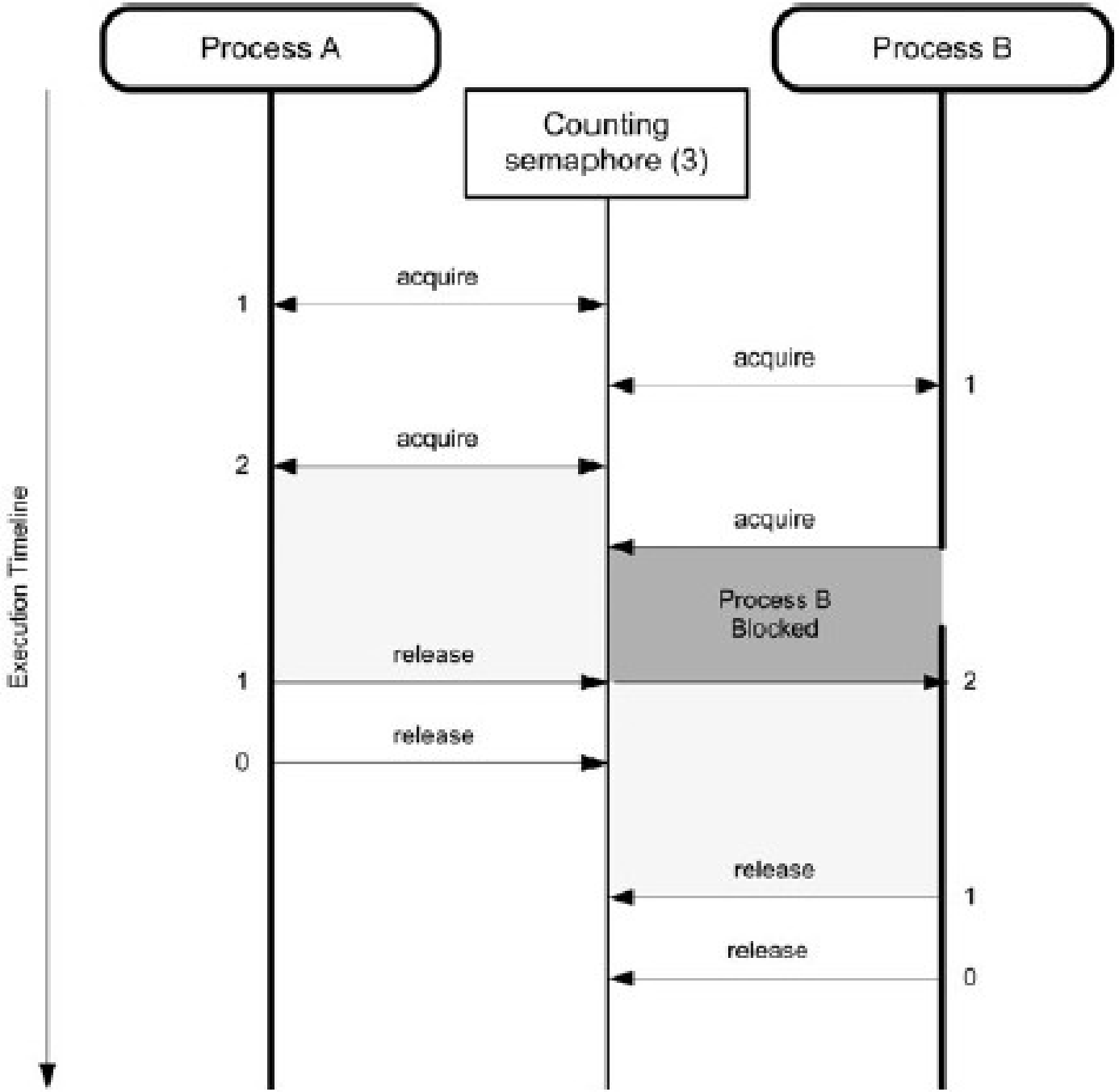
旗语分为两种基本类型。第一种是二进制旗语，二进制旗语代表单个资源；第二种是计数旗语，用来代表数量大于一的共享资源。

旗语[illegible]



Simple binary semaphore example with two processes.

旗语同



Counting semaphore example with two processes.

旗语同步

#include<sys/sem.h>

int semget(key_t key, int nsems, int semflg);

功能：创建一个新的旗语或取得一个已有的旗语。

返回值：成功返回旗语标识符，失败返回 -1 。

参数：

key ： 系统范围内的唯一标识符，其它进程访问旗语的依据。

IPC_PRIVATE 告诉 semget 自己生成标识符，所以其它进程无法访问该旗语。使用 ftok() 函数获得唯一标识。

nsems ： 旗语个数，为 1 时表示单个旗语，大于 1 表示旗语数组，用来获取已有旗语时设为 0

semflg ： 指令和访问权限标志，

IPC_CREAT 创建新的旗语， IPC_CREAT|IPC_EXCL 如果旗语存在则失败， errno 设为 EEXIST 。

semflg=0 用来获取一个已有的旗语。

打开权限类似 open(), 相当于文件的访问权限。

旗语同步

#include<sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);

功能：提供获取和释放旗语或旗语数组的方法。

返回值：成功返回 0 ，否则 -1 。

参数：

semid ： 旗语描述符。

sops ： 旗语结构 (数组) 指针，包含了具体操作。

struct sembuf{

unsigned short sem_num; // 对应旗语数组中的旗语， 0 对应第一个旗语

short sem_op; // 旗语的当前值记录相应资源目前可用数目；
sem_op>0 进程要释放 sem_op 数目的共享资源； sem_op=0 用于对共享资源是否已用完的测试 (调用进程将调用 sleep() ，直到信号量的值为 0) ；
sem_op<0 进程要申请 -sem_op 个共享资源

short sem_flg; //IPC_NOWAIT ： 无旗语可用则失败， SEM_UNDO ： 在进程结束时，相应的操作将被取消，如果设置了该标志位，在进程没有释放共享资源就退出时，内核将代为释放。

};

nsops ： 要进行旗语操作的数量 (旗语结构 (数组) 元素个数) 。

旗语同步

#include<sys/sem.h>

int semctl(int semid, int semnum, int cmd, …);

功能：控制旗语信息。

返回值：成功返回 0(或要获取的值)，否则 -1 。

参数：

semid ： 旗语标识符。

semnum ： 旗语编号。

cmd ： 要进行的操作。

第四个参数，是 union semun 的实例，具体值依赖 cmd 。

```
union semun{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
}arg;
```

旗语同步

cmd 命令 (设第四个参数为 arg):

IPC_STAT　　获取旗语信息，信息由 arg.buf 返回；

IPC_SET　　设置旗语信息，待设置信息保存在 arg.buf 中；

IPC_RMID　　删除旗语或旗语数组 ;

GETALL 返回所有旗语的值，结果保存在 arg.array 中，参数 sennum 被忽略；

GETNCNT　　返回等待 semnum 所代表旗语的值增加的进程数，相当于目前有多少进程在等待 semnum 代表的旗语所代表的共享资源；

GETPID　　返回最后一个对 semnum 所代表旗语执行 semop 操作的进程 ID ；

GETVAL 返回 semnum 所代表旗语的值；

GETZCNT　　返回等待 semnum 所代表旗语的值变成 0 的进程数；

SETALL　　通过 arg.array 更新所有旗语的值；同时，更新与本旗语集相关的 semid_ds 结构的 sem_ctime 成员；

SETVAL　　设置 semnum 所代表旗语的值为 arg.val ；

共享内存编程

共享内存区域是被多个进程共享的一部分物理内存。如果多个进程都把该内存区域映射到自己的虚拟地址空间，则这些进程就都可以直接访问该共享内存区域，从而可以通过该区域进行通信。

共享内存是进程间共享数据的一种最快的方法，一个进程向共享内存区域写入了数据，共享这个内存区域的所有进程就可以立刻看到其中的内容。

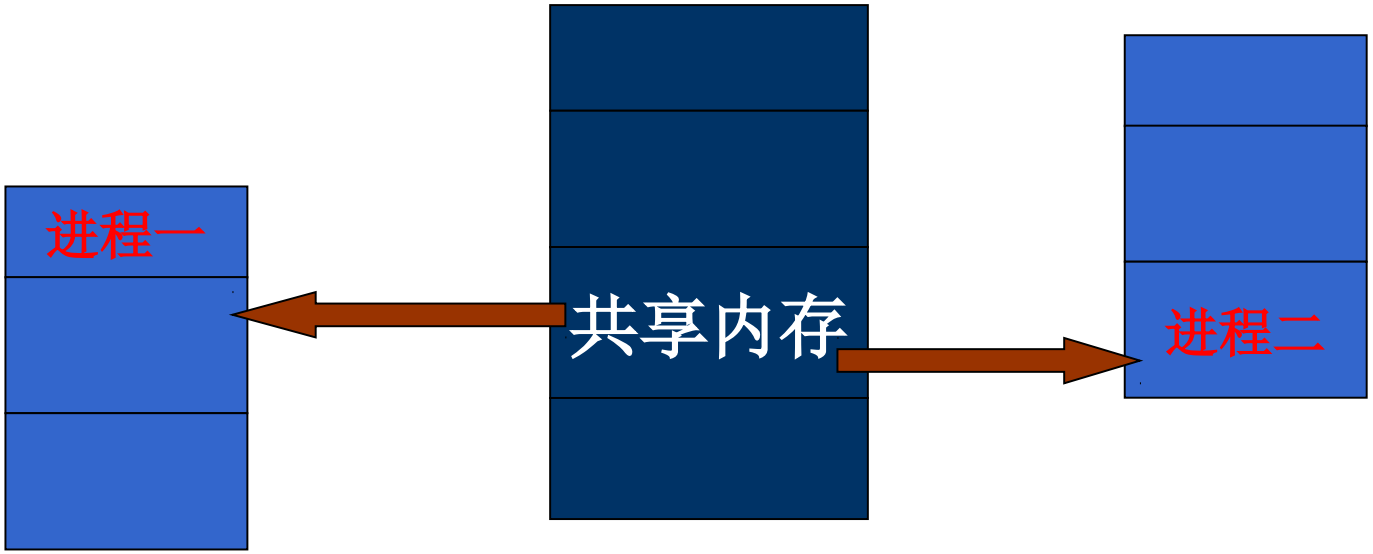
不提供任何同步功能

共享内存编程

共享内存区域是被多个进程共享的一部分物理内存。如果多个进程都把该内存区域映射到自己的虚拟地址空间，则这些进程就都可以直接访问该共享内存区域，从而可以通过该区域进行通信。

共享内存是进程间共享数据的一种最快的方法，一个进程向共享内存区域写入了数据，共享这个内存区域的所有进程就可以立刻看到其中的内容。

不提供任何同步功能



共享内存编程

#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);

功能：创建共享内存或获取一个已经存在的共享内存 (shmflg=0) 。

返回值：成功返回共享内存标识符，失败返回 -1 。

参数：

key: 标识共享内存的键值。 (IPC_PRIVATE, ftok())

size: 要建立共享内存的长度 (打开已有共享内存时置 0) 。
(n*PAGE_SIZE)

shmflg: 指令和访问权限标志。

IPC_CREAT 如果共享内存不存在，则创建一个共享内存，否则打开操作。

IPC_EXCL 只有在共享内存不存在的时候，新的共享内存才建立，否则就产生错误。

打开权限类似 open(), 相当于文件的访问权限。

共享内存编程

#include <sys/shm.h>

int *shmat(int shmid, const void *shmaddr, int shmflg);

功能：允许进程访问一块共享内存（共享内存刚创建时不能使用）。

返回值：成功返回共享内存的起始地址，失败返回 -1 。

参数：

shmid ： 共享内存的标识符。

shmaddr ： 共享内存的起始地址

shmflag ： 本进程对该内存的操作模式。如果是 SHM_RDONLY 的话，就是只读模式。

共享内存编程

#include <sys/shm.h>

int shmdt(const void *shmaddr);

功能：释放共享内存。

返回值：成功时返回 0 。失败时返回 -1 。

参数：

shmaddr 是共享内存的起始地址。

共享内存编程

#include<sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

功能：共享内存控制函数。

返回值：成功返回 0 ，失败返回 -1 。

参数：

shmid ： 共享内存的 ID 。

cmd ： 允许的操作，最常用的包括：

IPC_RMID 删除共享内存段。

buf ： 保存内存模式状态和访问权限的数据结构，通常为 0 。

POSIX IPC

Interface	Message queues	Semaphores	Shared memory
Header file	`<mqueue.h>`	`<semaphore.h>`	`<sys/mman.h>`
Object handle	*mqd_t*	*sem_t **	*int* (file descriptor)
Create/open	*mq_open()*	*sem_open()*	*shm_open()* + *mmap()*
Close	*mq_close()*	*sem_close()*	*munmap()*
Unlink	*mq_unlink()*	*sem_unlink()*	*shm_unlink()*
Perform IPC	*mq_send()*, *mq_receive()*	*sem_post()*, *sem_wait()*, *sem_getvalue()*	operate on locations in shared region
Miscellaneous operations	*mq_setattr()*—set attributes *mq_getattr()*—get attributes *mq_notify()*—request notification	*sem_init()*—initialize unnamed semaphore *sem_destroy()*—destroy unnamed semaphore	(none)

xxx_open

POSIX IPC 量的打开或创建由 xxx_open 函数完成，使用类似于文件打开操作 (open)

例:

```
fd = shm_open("/mymem", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
```

xxx_close

消息队列使用 mq_close 关闭打开的消息队列。

旗语使用 sem_close 关闭打开的旗语。

共享内存使用 munmap 关闭共享内存。

xxx_unlink

xxx_unlink 用来删除该 IPC 量。删除 IPC 只发生在没有进程使用该量时，即对消息队列和旗语，所有打开过该 IPC 量的进程都用 xxx_close 关闭了该 IPC 量，对共享内存，所有映射的内存都使用 munmap 之后。

POSIX IPC 编译

POSIX IPC 需使用实时库 librt，所以编译时需为 gcc 添加 -lrt 选项。

消息队列

打开或创建消息队列

```
#include <fcntl.h>              /* Defines O_* constants */
#include <sys/stat.h>           /* Defines mode constants */
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */);
```

Returns a message queue descriptor on success, or *(mqd_t)* −1 on error

Flag	Description
O_CREAT	Create queue if it doesn't already exist
O_EXCL	With O_CREAT, create queue exclusively
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing
O_NONBLOCK	Open in nonblocking mode

消息队列

关闭消息队列

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

Returns 0 on success, or −1 on error

删除消息队列

```
#include <mqueue.h>

int mq_unlink(const char *name);
```

Returns 0 on success, or −1 on error

消息队列

获取消息队列属性

```
struct mq_attr {
    long mq_flags;       /* Message queue description flags: 0 or
                            O_NONBLOCK [mq_getattr(), mq_setattr()] */
    long mq_maxmsg;      /* Maximum number of messages on queue
                            [mq_open(), mq_getattr()] */
    long mq_msgsize;     /* Maximum message size (in bytes)
                            [mq_open(), mq_getattr()] */
    long mq_curmsgs;     /* Number of messages currently in queue
                            [mq_getattr()] */
};
```

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
```

Returns 0 on success, or −1 on error

设置消息队列属性

```
#include <mqueue.h>

int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr,
               struct mq_attr *oldattr);
```

Returns 0 on success, or −1 on error

消息队列

发送消息

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
            unsigned int msg_prio);
```

Returns 0 on success, or −1 on error

接收消息

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio);
```

Returns number of bytes in received message on success, or −1 on error

POSIX 旗语

创建或打开命名旗语

```
#include <fcntl.h>              /* Defines O_* constants */
#include <sys/stat.h>           /* Defines mode constants */
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...
                /* mode_t mode, unsigned int value */ );
```

Returns pointer to semaphore on success, or SEM_FAILED on error

POSIX 旗语

关闭命名旗语

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Returns 0 on success, or −1 on error

删除命名旗语

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

Returns 0 on success, or −1 on error

POSIX 旗语

命名旗语操作：

```
#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);

int sem_post(sem_t *sem);

int sem_getvalue(sem_t *sem, int *sval);
```

Returns 0 on success, or −1 on error

POSIX 旗语

初始化匿名旗语

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Returns 0 on success, or −1 on error

删除匿名旗语

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Returns 0 on success, or −1 on error

POSIX 共享内存

创建或打开共享内存

```
#include <fcntl.h>              /* Defines O_* constants */
#include <sys/stat.h>           /* Defines mode constants */
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
```

Returns file descriptor on success, or −1 on error

新创建的共享内存长度为 0，其后可使用 ftruncate() 扩张其长度，然后再调用 mmap()

Flag	Description
O_CREAT	Create object if it doesn't already exist
O_EXCL	With O_CREAT, create object exclusively
O_RDONLY	Open for read-only access
O_RDWR	Open for read-write access
O_TRUNC	Truncate object to zero length

POSIX 共享内存

删除共享内存

```
#include <sys/mman.h>

int shm_unlink(const char *name);
```

Returns 0 on success, or −1 on error