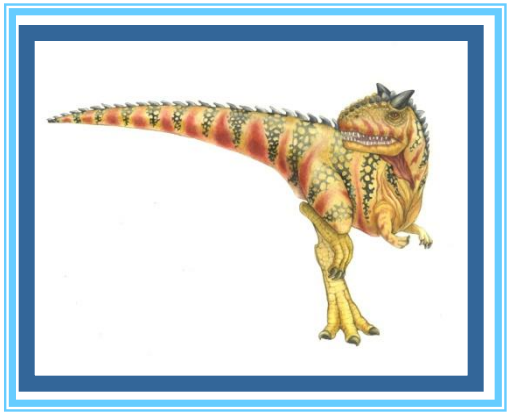
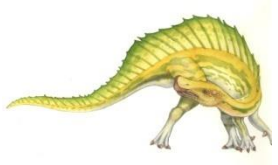
Linux 系统程序设计



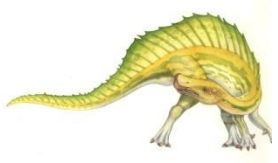
内容

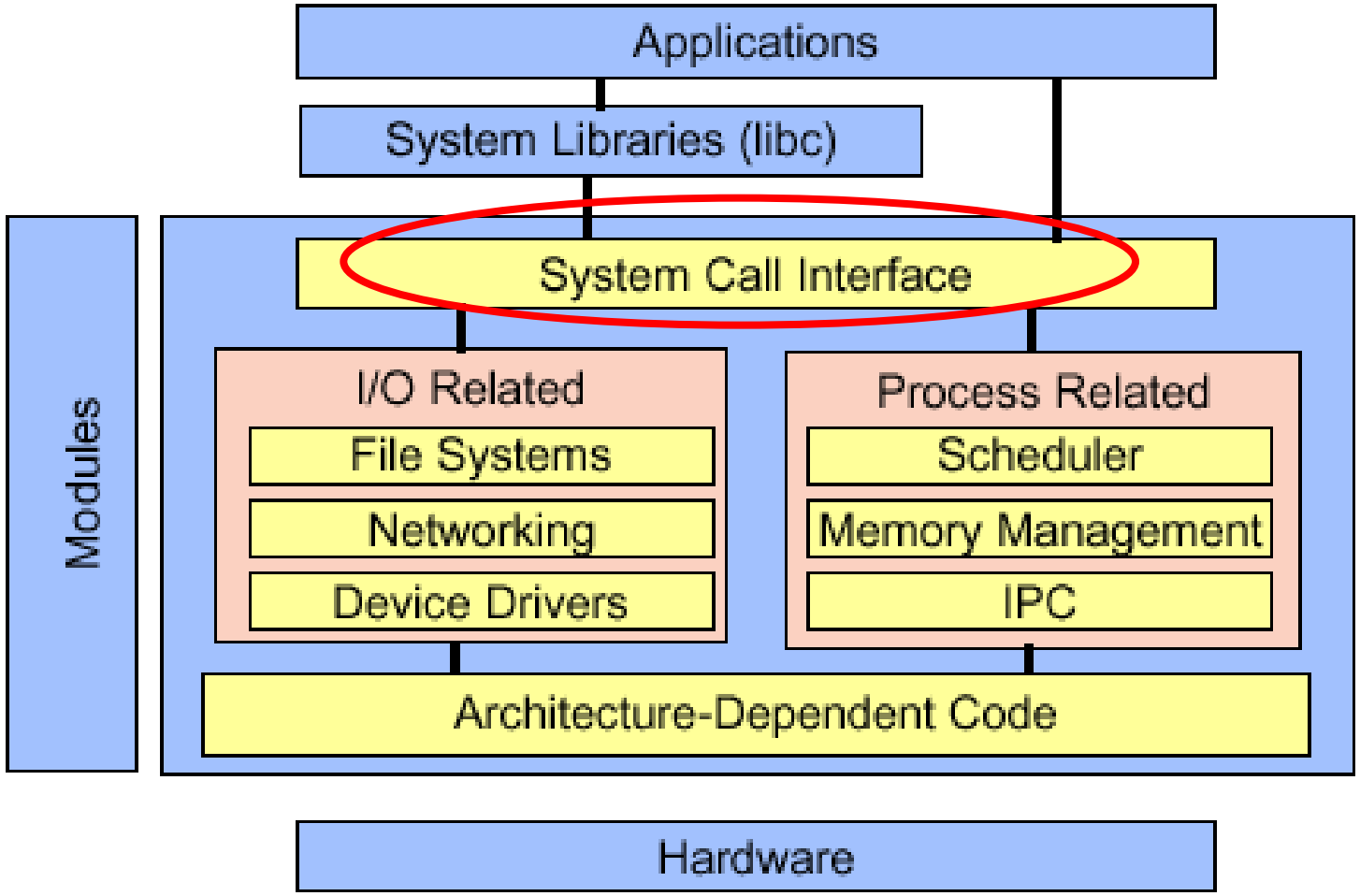
- 概述
- Linux环境下函数的使用
- 文件和目录操作
- 进程控制
- 进程通信
- 线程

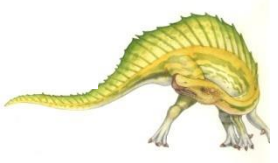


参考资料：

- 《Linux程序设计》
- 《UNIX环境高级编程》
  - 第3章 文件I/O
  - 第4章 文件和目录
  - 第8章 进程控制

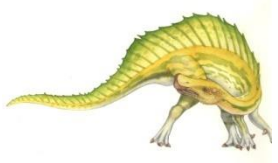



Applications
System Libraries (libc)
System Call Interface
Modules
I/O Related
File Systems
Networking
Device Drivers
Process Related
Scheduler
Memory Management
IPC
Architecture-Dependent Code
Hardware



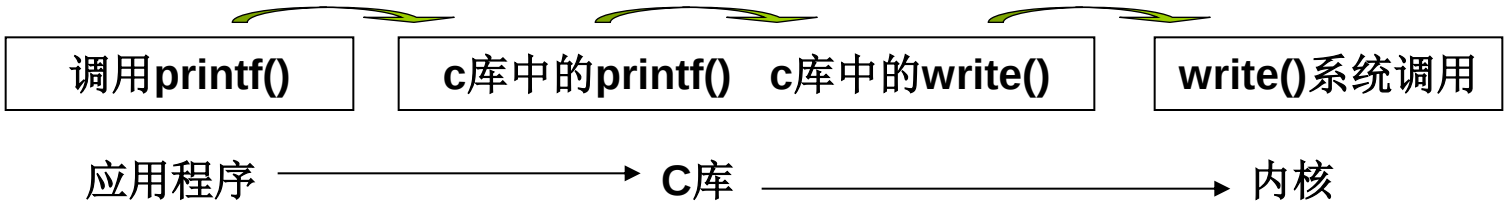
系统调用和C函数库

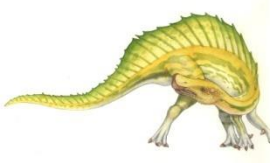
- **系统调用：**在用户进程和硬件设备之间添加了一个中间层。应用程序调用操作系统提供的功能模块（函数）。
  - 对文件和设备进行访问和控制需要使用系统调用实现
  - Linux程序员通过API函数使用系统调用
- **C函数库**
  - 依赖于系统调用
  - 一般来说，标准函数库建立在系统调用的上层，提供的功能比系统调用强，使用也比较方便。
  - 例如打开文件，标准I/O库为fopen函数 。系统调用为open



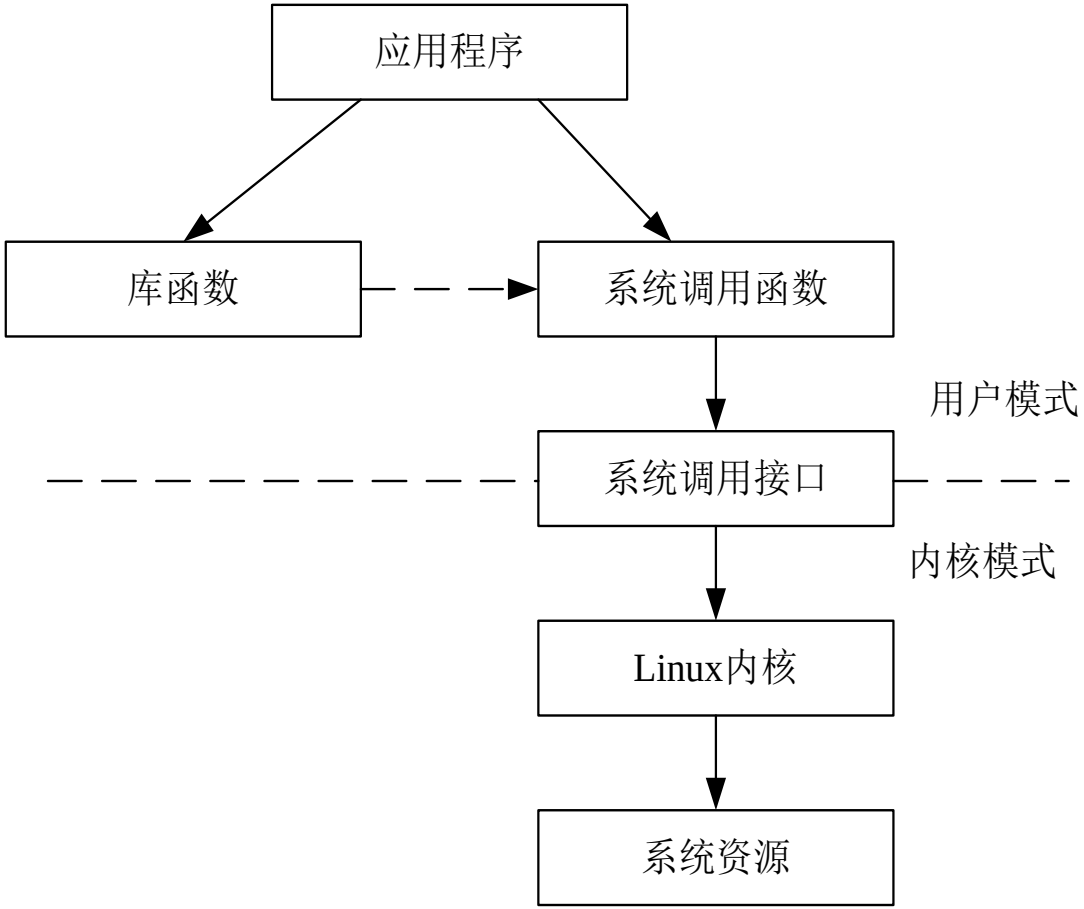
系统调用、API和C库

- Linux的应用编程接口（API）遵循 POSIX标准（可移植操作系统接口（Portable Operating System Interface ）
- Linux的系统调用作为c库的一部分提供。c库中实现了Linux的主要API，包括标准c库函数和系统调用。
- 应用编程接口(API)其实是一组函数定义，这些函数说明了如何获得一个给定的服务；而系统调用是通过软中断向内核发出一个明确的请求，每个系统调用对应一个封装例程（wrapper routine，唯一目的就是发布系统调用）。一些API应用了封装例程。
- 系统调用的实现是在内核完成的，而用户态的函数是在函数库中实现的





库函数和系统调用



C库函数

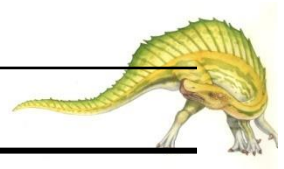
API函数

系统调用（函数）

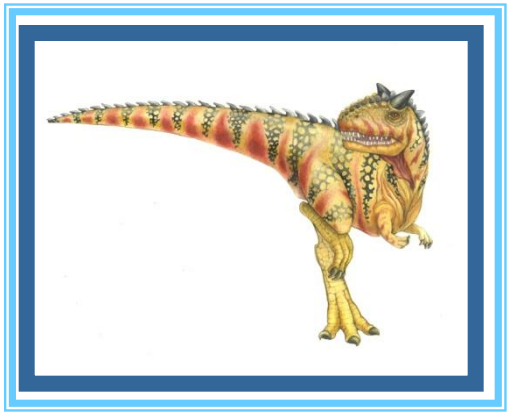
系统调用实现

扩展名

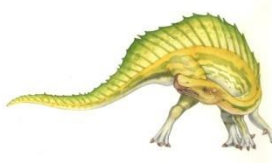
.c	C source code which must be preprocessed
.i	C source code which should not be preprocessed
.cc .cp .cpp .CPP. c++ .C .cxx	C++ source code which must be preprocessed
.ii	C++ source code which should not be preprocessed
.h	C or C++ header file to be turned into a precompiled header
.H .hh	C++ header file to be turned into a precompiled header
.s	Assembler code
.S	Assembler code which must be preprocessed
.o	Object file
.a	Static library file (archive file)
.so	Dynamic library file (shared object)



Linux环境下库函数的使用



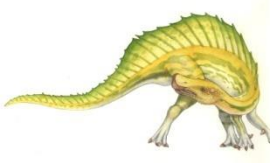
- Linux环境下的有各种C库函数，如：
  - 数学函数、字符操作函数、系统时间与日期函数、内存分配函数、排序与查找等。



例1

- 例　从键盘读入一行字符，测试读入字符是否为大写字符。
  - isupper()库函数判断字符是否为大写英文字母

```
/*isupper.c*/
#include "stdio.h"
#include <ctype.h>
main()
{
 char c;
 while((c=getchar())!='\n')
    if(isupper(c))
       printf("%c is an uppercase character\n",c);
}
```



system函数

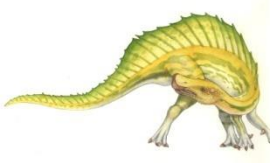
- C标准库中的system函数提供了一种调用其它程序的简单方法。
  - 利用system函数调用程序结果与从shell中执行这个程序基本相似。
  - 使用system函数比使用系统调用exec函数族更方便；然而，system函数使用shell调用命令，它受到系统shell自身的功能特性和安全缺陷的限制。

所需头文件	#include<stdlib.h>
函数功能	在进程中开始另一个进程
函数原型	int system(const char *string);
函数传入值	系统变量
函数返回值	执行成功则返回执行shell命令后的返回值，调用/bin/sh失败则返回127，其他失败原因则返回－1，参数string为空（NULL），则返回非零值
备注	system()调用fork()产生子进程，子进程调用/bin/sh –c string 来执行参数string字符串所代表的命令，此命令执行完后随即返回原调用的进程。如果调用成功，返回shell命令后的返回值也可能是127，因此，最好能检查errno来确定

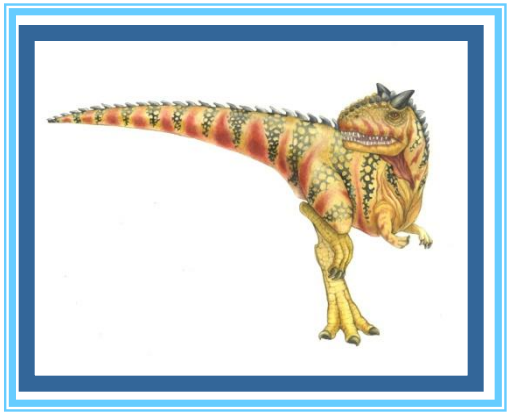
例2

- 例：设计一个程序，要求列出当前目录下的文件信息，以及系统“/home”和“/dev/lp0”的文件信息。

```
/*system.c*/
#include<stdio.h>          /*文件预处理，包含标准输入输出库*/
#include<stdlib.h>         /*文件预处理，包含system函数库*/
int main () {              /*C程序的主函数，开始入口*/
    int newret;
    printf("列出当前目录下的文件信息：\n");
    newret=system("ls -l");          /*调用ls程序，列出当前目录下的文件信息*/
    printf("列出“/home”的文件信息：\n");
    newret=system(“ls /home -l”);          /*列出/home子目录下的文件信息*/
    printf("列出“/dev/ lp0”的文件信息：\n");
    newret=system("ls /dev/lp0 -l");          /*列出“/dev/ lp0”的文件信息*/
    return 0;
}
```



文件操作(系统调用)



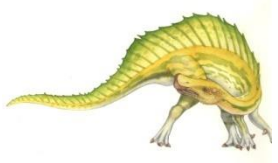
文件I/O

- 文件 I/O
  - open/creat, close, read, write, lseek
  - dup/dup2
  - fcntl
  - ioctl

文件描述符

- 对于内核而言，所有打开文件都由文件描述符引用。文件描述符是一个非负整数。
  - int fd;
    - 0　标准输入。
    - 1　标准输出。
    - 2　标准错误。

- 文件操作步骤:
  - open/creat -> read/write -> [lseek] -> close

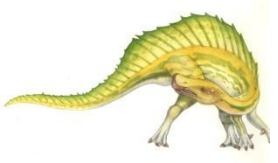


open系统调用

- 在使用文件前需要打开文件，使用系统调用open：

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
int open(const char *path, int oflags); //只用于打开
int open(const char *path, int oflags, mode_t mode);//若文件不存在，用于创建文件
```

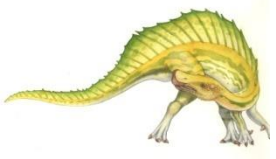
  - 准备打开的文件或设备的名字作为参数path传递给函数，oflags参数用来定义打开文件所采取的动作。



open系统调用

- oflags参数是通过主要文件访问模式与其他可选模式的结合来指定的。open调用可以下表中所示的文件访问模式之一。

模　式	说　明
O_RDONLY	以只读方式打开
O_WRONLY	以只写方式打开
O_RDWR	以读写方式打开



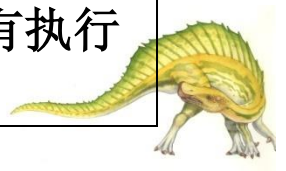
open系统调用

- open调用还可以在oflags参数中包括下列可选模式的组合（用“按位或”操作）：
  - O_APPEND：把写入数据追加在文件的末尾。
  - O_TRUNC：把文件长度设置为零，丢弃已有的内容。
  - O_CREAT：如果需要，就按参数mode中给出的访问模式创建文件。
  - O_EXCL：与O_CREAT一起使用，确保调用者创建出文件。open是一个原子操作，也就是说，它只执行一个函数调用。使用这个可选模式可以防止两个程序同时创建一个文件。如果文件已经存在，open调用将失败。
  - O_NONBLOCK：对设备read/write时，不阻塞。

访问权限的初始值

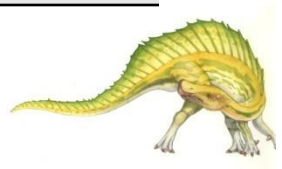
- 当使用带有O_CREAT标志的open来创建文件时，我们必须使用有三个参数格式的open调用。第三个参数mode是几个标志按位OR后得到的，这些标志在头文件sys/stat.h中定义，它们是：
  - S_IRUSR：读权限，文件属主（所有者）。
  - S_IWUSR：写权限，文件属主。
  - S_IXUSR：执行权限，文件属主。
  - S_IRGRP：读权限，文件所属组。
  - S_IWGRP：写权限，文件所属组。
  - S_IXGRP：执行权限，文件所属组。
  - S_IROTH：读权限，其他用户。
  - S_IWOTH：写权限，其他用户。
  - S_IXOTH：执行权限，其他用户。

例：open (“myfile”, O_CREAT, S_IRUSR|S_IXOTH);
新建一个名为myfile的文件，文件属主拥有读权限，其他用户拥有执行权限 。



creat 创建文件

所需头文件	#include<sys/types.h> #include<sys/stat.h> #include<fcntl.h>
函数功能	创建文件
函数原型	int creat(const char * pathname, mode_t mode);
函数传入值	建立文件的访问路径，用来设置新增文件的权限 参数mode的取值和说明请参考节的文件权限部分
函数返回值	正确返回0，若有错误发生则会返回－1

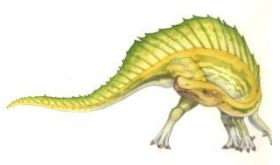


close 关闭文件

所需头文件	#include<unistd.h>
函数功能	关闭文件
函数原型	int close(int fd);
函数传入值	整型
函数返回值	若文件顺利关闭则返回0，发生错误时返回－1
备注	虽然在进程结束时，系统会自动关闭已打开的文件，但仍建议自行关闭文件，并确实检查返回值

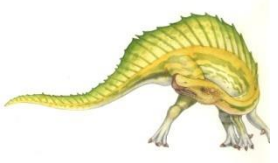
read 读文件

所需头文件	#include<unistd.h>
函数功能	读取文件
函数原型	ssize_t read(int fd,void * buf ,size_t count);
函数传入值	fd：文件描述符 buf：读取内容的内存地址 count：读取的字节数
函数返回值	有错误发生则会返回－1
备注	若返回的字节数比要求读取的字节数少，则有可能读到了文件尾、从管道(pipe)或终端机读取



write 写文件

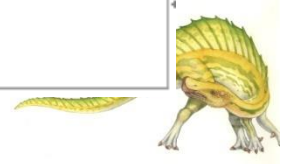
所需头文件	#include<unistd.h>
函数功能	写入文件
函数原型	ssize_t write (int fd,const void * buf,size_t count);
函数传入值	fd：文件描述符 buf：写入内容的内存地址 count：写入的字节数
函数返回值	有错误发生则会返回－1
备注	返回值。如果顺利write()会返回实际写入的字节数



lseek 定位

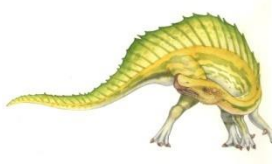
- lseek函数用于在指定的文件描述符中将文件指针定位到相应的位置。

所需头文件	#include <unistd.h> #include <sys/types.h>	
函数原型	off_t lseek(int fd, off_t offset, int whence)	
函数传入值	fd: 文件描述符	
	offset: 偏移量，每一读写操作所需要移动的距离，单位是字节，可正可负（向前移，向后移）	
	whence: 当前位置的基点	SEEK_SET: 当前位置为文件的开头，新位置为偏移量的大小
		SEEK_CUR: 当前位置为文件指针的位置，新位置为当前位置加上偏移量
		SEEK_END: 当前位置为文件的结尾，新位置为文件的大小加上偏移量的大小
函数返回值	成功: 文件的当前位移 -1: 出错	



fcntl函数

- 用fcntl函数改变一个已打开的文件的属性，可以重新设置读、写、追加、非阻塞等标志，
- 使用函数fcntl通过F_GETFL、F_SETFL可以分别用于读取、设置文件的属性，更改的文件标志有O_APPEND，O_ASYNC， O_DIRECT， O_NOATIME 和 O_NONBLOCK。

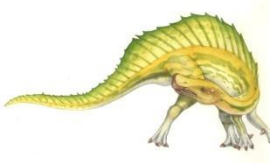


fcntl文件控制

```
/* Do the file control operation described by CMD on FD. */
extern int fcntl (int __fd, int __cmd, ...);
```

此函数第一个参数 fd 为欲修改属性的文件描述符。第二个参数 cmd 为相应操作，常用的命令如下：

```
//come from /usr/include/bit/fcntl.h
/* Values for the second argument to `fcntl'.  */
#define F_DUPFD     0       /* Duplicate file descriptor. */     //复制文件描述符
#define F_GETFD     1       /* Get file descriptor flags.  */    //获得文件描述符标志
#define F_SETFD     2       /* Set file descriptor flags.  */    //设置文件描述符标志
#define F_GETFL     3       /* Get file status flags.  */        //获取文件状态
#define F_SETFL     4       /* Set file status flags.  */        //设置文件状态
```



fcntl例

- 1、获取文件的flags，即open函数的第二个参数:

```
flags = fcntl(fd,F_GETFL,0);
```

- 2、设置文件的flags:

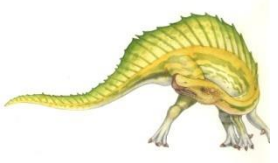
```
fcntl(fd,F_SETFL,flags);
```

- 3、增加文件的某个flags，比如文件是阻塞的，想设置成非阻塞:

```
flags = fcntl(fd,F_GETFL,0);
flags |= O_NONBLOCK;
fcntl(fd,F_SETFL,flags);
```

- 4、取消文件的某个flags，比如文件是非阻塞的，想设置成为阻塞:

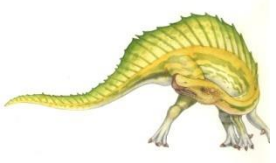
```
flags = fcntl(fd,F_GETFL,0);
flags &= ~O_NONBLOCK;
fcntl(fd,F_SETFL,flags);
```



例：一个简单的例子

- cat open_demo.c

```
/* a rudimentary example program */
#include <fcntl.h>
#include <stdio.h>
main()
{
    int fd, nread;
    char buf[1024];

    /*open file “data” for reading */
    fd = open(“data”, O_RDONLY);

    /* read in the data */
    nread = read(fd, buf, 1024);
     printf(buf);
    /* close the file */
    close(fd);
}
```



例：一个文件拷贝程序

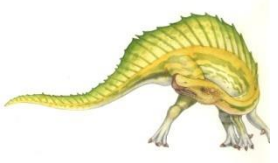
- 利用open、read和write函数，实现逐个字符地把一个文件拷贝到另外一个文件。

cat filecp.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
int main()
{
    char c;
    int in, out;
    /*以只读方式打开file.in文件*/
    in = open("file.in", O_RDONLY);
    /*以读写方式打开或创建文件file.out，文件主对file.out文件有读写权限*/
    out =open("file.out",O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
    while(read(in,&c,1) == 1)
        write(out,&c,1);
    exit(0);
}
```

在/usr/include，使用grep命令查找.h文件

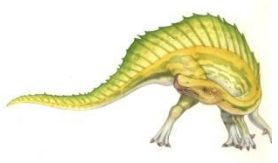
I/O效率低



fstat、stat和lstat系统调用

- fstat系统调用返回与打开的文件描述符相关的文件的状态信息，该信息将会写到stat结构中，stat的地址以参数形式传递给fstat。
- stat和lstat返回的是通过文件名查到的状态信息。它们的结果基本一致，但当文件是一个符号链接时，lstat返回的是该符号链接本身的信息，而stat返回的是该链接指向的文件的信息。
- 下面是它的语法：

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
int fstat(int fildes, struct stat *buf);
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

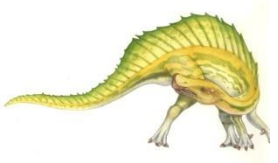


struct stat

```
struct stat {
    mode_t st_mode;  /*file type & mode*/
    ino_t     st_ino;     /*inode number (serial number)*/
    dev_t    st_dev;   /*device number (file system)*/
    nlink_t   st_nlink;  /*link count*/
    uid_t     st_uid;     /*user ID of owner*/
    gid_t     st_gid;     /*group ID of owner*/
    off_t     st_size;    /*size of file, in bytes*/
    time_t   st_atime; /*time of last access*/
    time_t   st_mtime; /*time of last modification*/
    time_t   st_ctime;  /*time of lat file status change*/
    long      st_blksize;     /*Optimal block size for I/O*/
    long      st_blocks;    /*number 512-byte blocks allocated*/
};
```

stat结构

stat部分成员	说　明
st_mode	文件权限和文件类型信息
st_ino	与该文件关联的inode
st_dev	保存文件的设备
st_uid	文件属主的UID号
st_gid	文件属主的GID号
st_atime	文件上一次被访问的时间
st_ctime	文件的权限、属主、组或内容上一次被改变的时间
st_mtime	文件的内容上一次被修改的时间
st_nlink	该文件上硬链接的个数

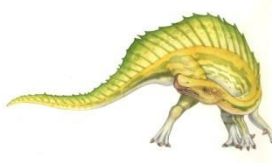


stat结构

- stat结构中返回的st_mode标志还有一些与之关联的宏，它们定义在头文件sys/stat.h中。这些宏包括对访问权限、文件类型标志以及一些用于帮助测试特定类型和权限的掩码的定义。
- 文件类型标志包括:

/usr/include/bits/stat.h

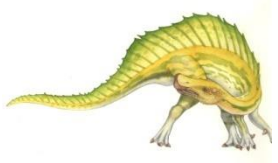
```
#define    __S_IFMT    0170000 /* These bits determine file type.
   */

/* File types.  */
#define    __S_IFDIR    0040000    /* Directory.  */
#define    __S_IFCHR    0020000    /* Character device.  */
#define    __S_IFBLK    0060000    /* Block device.  */
#define    __S_IFREG    0100000    /* Regular file.  */
#define    __S_IFIFO    0010000    /* FIFO.  */
#define    __S_IFLNK    0120000    /* Symbolic link.  */
#define    __S_IFSOCK   0140000    /* Socket.  */
```



stat结构

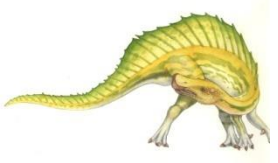
- st_mode访问权限标志与的open系统调用中的内容是一样的。
  - S_IRUSR：读权限，文件属主。
  - S_IWUSR：写权限，文件属主。
  - S_IXUSR：执行权限，文件属主。
  - S_IRGRP：读权限，文件所属组。
  - S_IWGRP：写权限，文件所属组。
  - S_IXGRP：执行权限，文件所属组。
  - S_IROTH：读权限，其他用户。
  - S_IWOTH：写权限，其他用户。
  - S_IXOTH：执行权限，其他用户。

  /usr/include/sys/stat.h



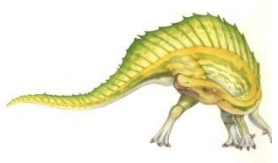
stat结构

- 其他模式标志包括：
  - S_ISUID：文件设置了SUID位。
  - S_ISGID：文件设置了SGID位。
- 用于解释st_mode标志的掩码还包括：
  - S_IFMT：文件类型。
  - S_IRWXU：属主的读/写/执行权限。
  - S_IRWXG：属组的读/写/执行权限。
  - S_IRWXO：其他用户的读/写/执行权限。



文件类型宏操作

- 文件类型的宏定义。它们只是对经过掩码处理的模式标志和相应的设备类型标志进行比较。它们包括：
  - S_ISBLK：测试是否是特殊的块设备文件。
  - S_ISCHR：测试是否是特殊的字符设备文件。
  - S_ISDIR：测试是否是目录。
  - S_ISFIFO：测试是否是FIFO设备。
  - S_ISREG：测试是否是普通文件。
  - S_ISLNK：测试是否是符号链接。



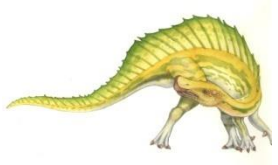
文件类型宏操作

```
/* 文件类型测试宏*/
#define   __S_ISTYPE(mode, mask)   (((mode) & __S_IFMT) == (mask))

#define   S_ISDIR(mode)      __S_ISTYPE((mode), __S_IFDIR)      //为目录
#define   S_ISCHR(mode)      __S_ISTYPE((mode), __S_IFCHR)      //为字符设备
#define   S_ISBLK(mode)      __S_ISTYPE((mode), __S_IFBLK)      //为块设备
#define   S_ISREG(mode)      __S_ISTYPE((mode), __S_IFREG)      //为常规文件
#ifdef __S_IFIFO
```

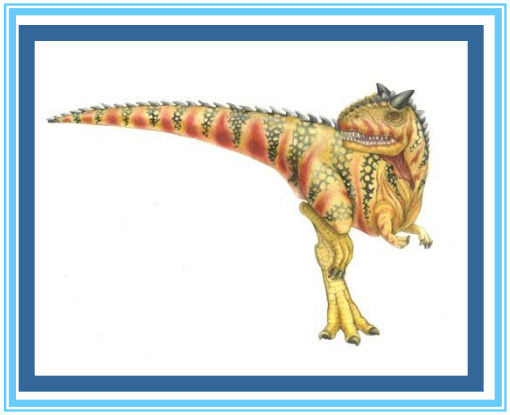
```
# define S_ISFIFO(mode)      __S_ISTYPE((mode), __S_IFIFO)      //为管道文件
#endif
#ifdef __S_IFLNK
# define S_ISLNK(mode)       __S_ISTYPE((mode), __S_IFLNK)      //为连接文件
#endif
```

- 例：根据命令行参数给定的文件名，判断其文件类型。

  filetype.c



目录操作



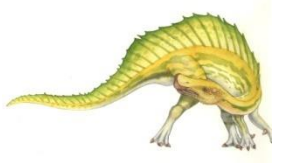
目录操作

- 创建目录系统调用：mkdir
  - mkdir系统调用的作用是创建目录，它相当于mkdir程序。mkdir调用将把参数path作为新建目录的名字。目录的权限由参数mode设定，mode的含义将按open系统调用的O_CREAT选项中的有关定义设置，当然，它还要服从umask的设置情况。

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
```

- 删除目录：rmdir
  - rmdir系统调用的作用是删除目录，但只有在目录为空时才行。rmdir程序就是用这个系统调用来完成工作的。

```
#include <unistd.h>
int rmdir(const char *path);
```



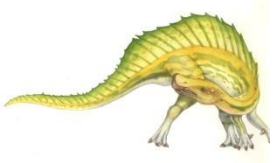
改变当前目录和得到当前目录：chdir、getcwd

- 切换目录 **chdir**

```
#include <unistd.h>
int chdir(const char *path);
```

- 程序可以通过调用**getcwd**函数来确定自己的当前工作目录。

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```



opendir

- opendir函数

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

  - opendir函数的作用是打开一个目录并建立一个目录流。如果成功，它返回一个指向DIR结构的指针，该指针用于读取目录数据项。

readdir

- **readdir函数**

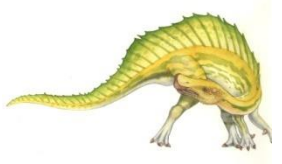
```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

  - readdir函数将返回一个指针，指针指向的结构里保存着目录流drip中下一个目录项的有关资料。后续的readdir调用将返回后续的目录项。如果发生错误或者到达目录尾，readdir将返回NULL。
  - dirent结构中包含的目录数据项内容包括以下部分：
    - ino_t d_ino——文件的inode节点号。
    - char d_name[]——文件的名字。

例：A Directory-Scanning Program

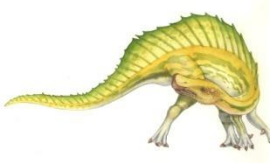
- printdir.c程序实现一个简单的目录列表功能，采用递归遍历各级子目录。

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <stdlib.h>
void printdir(char *dir, int depth)
{
      DIR *dp;
      struct dirent *entry;
      struct stat statbuf;
      //检查指定目录是否存
      if((dp = opendir(dir)) == NULL) {
         fprintf(stderr,"cannot open directory: %s\n", dir);
        return;
       }
```



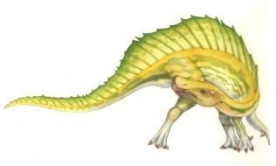
A Directory-Scanning Program

```
chdir(dir);//进入指定目录
while((entry = readdir(dp)) != NULL) {
    lstat(entry->d_name,&statbuf); /*根据文件名获得文件stat结构*/
    if(S_ISDIR(statbuf.st_mode)) {
       /*检查该数据项是否是一个目录 */
         if(strcmp(“.”,entry->d_name) == 0 ||strcmp(“..”,entry-
       >d_name) == 0)  continue;    /*检查是否是“.”和“..”目录 */
         printf(“%*s%s/\n”,depth,”“,entry->d_name);
    /*如果该数据项是一个目录，我们就需要对它进行递归遍 历 */
         printdir(entry->d_name,depth+4);
       }
       else printf(“%*s%s\n”,depth,”“,entry->d_name);
    }
    //回到目录树的上一级
    chdir(“..”);
    closedir(dp);
}
```



A Directory-Scanning Program

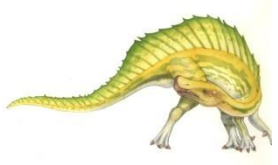
```
int main()
    {
       printf("Directory scan of /home:\n");
       printdir("/home",0);
       printf("done.\n");
       exit(0);
    }
```



程序说明

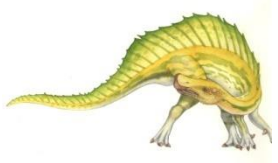
- 绝大部分操作都是在printdir函数里完成的
- 在用opendir函数检查完指定目录是否存在后，printdir调用chdir进入指定目录。如果readdir函数返回 的数据项不为空，程序就检查该数据项是否是一个目录。如果不是，程序就根据depth的值缩进打印文件数据项的内容。
- 如果该数据项是一个目录，需要对它进行递归遍历。在跳过.和..数据项（它们分别代表当前目录和上一级目录）后，printdir函数调用自己并再次进入一个同样的处理过程。
- 一旦while循环完成，chdir(“..”)调用把它带回到目录树的上一级，从而可以继续列出上级目录中的清单。
- 调用closedir(dp)关闭目录以确保打开的目录流数目不超出其需要。

Process Control（进程控制）



进程控制

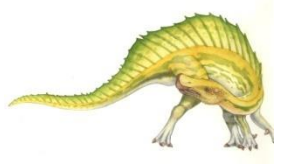
- 创建新进程
- 执行程序
- 进程终止



进程标识

- 每个进程都有一个非负整型的唯一进程ID。因为进程ID标识符总是唯一的，常将其用做其他标识符的一部分以保证其唯一性。下列函数返回这些标识符。

```
#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
```

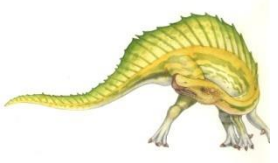


fork函数

- **fork**:创建一个新子进程

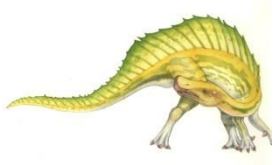
```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

  - 返回值：
    - 子进程中为0，
    - 父进程中为子进程I D
    - 出错为-1
  - 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新子进程的进程ID。



例

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void  main(){
    int pid;
    printf("Hello world!\n");
    pid=fork();     /* 创建进程*/
    if (pid==0)    {
            //子进程执行代码
      printf("child process PID %d\n",getpid());}
else  {
          //父进程执行代码
       sleep(35);
       printf("parent process PID %d\n",getpid());
} }
```

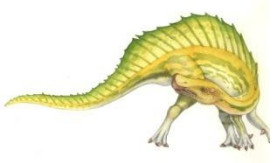


fork

- fork通常格式：

```
if ( (pid=fork()) < 0) {
    /* error handling */
} else if (pid == 0) {
    /* child */
} else {
    /* parent */
};
```

- 例：fork.c，fork2.c



exec函数

- 用fork函数创建子进程后，子进程往往要调用一种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。
- 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。

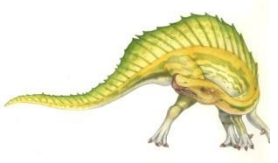
exec函数

- 有六种不同的exec函数可供使用，它们常常被统称为exec函数。

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0,  ... /* (char *)0 */ );
int execv(const char *pathname, char *const argv []);
int execle(const char *pathname, const char *arg0, ... /* (char *)0,
    char *const envp[] */ );
int execve(const char *pathname, char *const   argv[], char *const
    envp []);

int execlp(const char *filename, const char *arg0,   ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv []);
```

表8-4　六个exec函数之间的区别

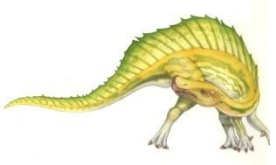
函　数	*pathname*	*filename*	参 数 表	*argv*[]	environ	*envp*[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(字母表示)		p	l	v		e



例

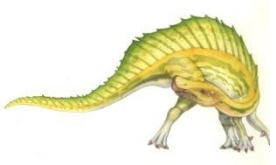
```
# include <stdio .h>
#include <unistd.h>
main ( ) {
    int PID = fork ();
    if ( PID > 0 ) {
      wait ( NULL );
      execlp("sleep","sleep","10",NULL);
        /* execute sleep 10 => sleep 10 sec. */
      printf ( "hello child process\n " );}
     else {
       printf ( "hello parent process\n " );
}}
```

程序运行结果：
hello parent process

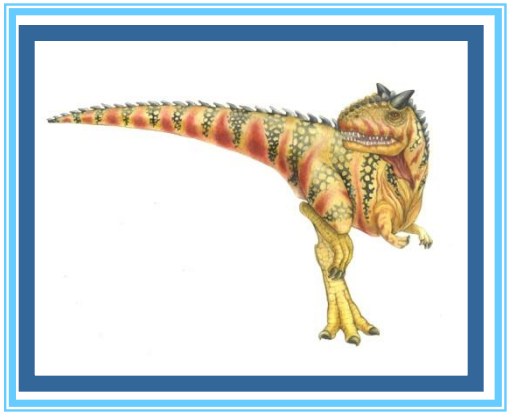


exec函数

- 例：
  - fork1.c
  - exec.c , exec1.c



进程通信



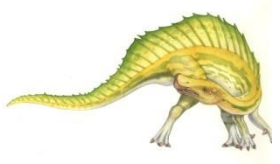
Linux进程通信

内容：

- 概述
- System V IPC相关的系统调用
- System V IPC信号量
- System V IPC共享内存

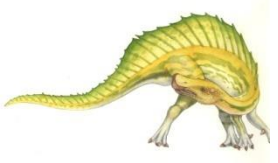
Linux进程通信方法

- Linux实现进程间通信(IPC Inter Process Communication)方法有：
  - System V IPC机制:
    - 信号量;
    - 消息队列;
    - 共享内存。
  - 管道（pipe）、命名管道
  - 套接字（socket）
  - 信号(signal)



System V IPC相关的系统调用

- 用信号量对进程要访问的临界资源进行保护:
  - semget(): 创建或打开信号量;
  - semop() : 增加或减少一个信号量的值;
  - semctl(): 对信号量进行控制。
- 用消息队列在进程间以异步方式发送消息:
  - msgget(): 创建新消息队列或打开现有消息队列;
  - msgsnd(): 发送消息;
  - msgrcv(): 接收消息;
  - msgctl(): 对消息队列进行控制，如删除消息队列。
- 共享内存区域进行进程之间交换数据:
  - shmget(): 创建或打开共享内存区。
  - shmat (): 连接共享内存区。
  - shmdt (): 拆除共享内存区连接
  - shmctl(): 共享内存储区控制



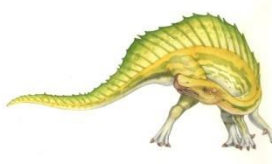
IPC资源共有的特性

- 信号量、消息队列、共享内存统称IPC资源
- 创建IPC资源XXXget()、控制IPC资源XXXctl()（其中：XXX代表msg，shm，sem三者之一）。
- XXXget()函数
  - 两个共同参数：key和XXXflag。
  - key既可由ftok()函数产生，也可以是IPC_PRIVATE常量，key值是IPC资源的外部表示。
  - XXXflag包括读写权限，还可以包含IPC_CREATE和IPC_EXCL标志位。它们组合的效果如下：
    - 指定key为IPC_PRIVATE,保证创建一个惟一的IPC资源。
    - 设置XXXflag参数的IPC_CREATE标志位，但不设置IPC_EXCL。如果相应key的IPC资源不存在，则创建一个IPC资源，否则返回已存在的IPC资源。
    - XXXflag参数的IPC_CREATE和IPC_EXCL同时设置。如果相应key的IPC资源不存在，则创建一个IPC资源。否则返回一个错误信息。

- XXXctl()：均提供IPC_SET,IPC_STAT和IPC_RMID命令。前两者用来设置或得到IPC资源的状态信息，IPC_RMID用来释放IPC资源。

共同的操作模式:

- 都是先通过XXXget()创建一个IPC资源，返回值是该IPC资源ID。在以后的操作中，均以IPC资源ID为参数,以对相应的IPC资源进行操作。
- 别的进程可以通过XXXget()取得已有的IPC资源ID（权限允许的话）并对其操作，从而使进程间通信成为可能。



消息队列

- **消息队列：**就是一个消息的链表，是一系列保存在内核中的消息的列表。用户进程可以向消息队列添加消息，也可以从消息队列读取消息。

消息队列的常用函数

函 数	功 能
fotk	由文件路径和工程ID生成标准key
msgget	创建或打开消息队列
msgsnd	添加消息
msgrcv	读取消息
msgctl	控制消息队列

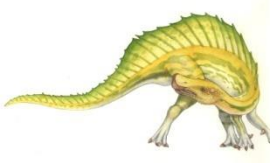
- 例： 设计一个程序，要求用函数msgget创建消息队列，从键盘输入的字符串添加到消息队列，然后应用函数msgrcv读取队列中的消息并在计算机屏幕上输出。
- 分析：程序先调用msgget函数创建、打开消息队列，接着调用msgsnd函数，把输入的字符串添加到消息队列中，然后调用msgrcv函数，读取消息队列中的消息并打印输出，最后调用msgctl函数，删除系统内核中的消息队列。

```
//msgfork.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <unistd.h>
struct msgmbuf  {                  /*结构体，定义消息的结构*
    long msg_type;                 /*消息类型*/
    char msg_text[512]; };         /*消息内容*/
int main(){
int qid,len;
key_t key;
struct msgmbuf msg;
```

```
if((key=ftok(".",'a'))==-1) {   /*调用ftok函数，产生标准的key*/
      perror("产生标准key出错");
     exit(1);}

/*调用msgget函数，创建、打开消息队列*/
if((qid=msgget(key,IPC_CREAT|0666))==-1) {
     perror("创建消息队列出错");
     exit(1);}
printf("创建、打开的队列号是：%d\n",qid);  /*打印输出队列号*/
```

```
int pid=fork();
if (pid>0){
    printf(" 我是父进程PID=：%d 发送消息\n",getpid())
     puts("请输入要加入队列的消息：");
     /*键盘输入的消息存入变量msg_text*/
     if((fgets((&msg)->msg_text,512,stdin))==NULL) {
            puts("没有消息");
            exit(1);}
     msg.msg_type=getpid();
     len=strlen(msg.msg_text);
     /*调用msgsnd函数，添加消息到消息队列*/
     if((msgsnd(qid,&msg,len,0))<0) {
            perror("添加消息出错");
            exit(1);}
}
```

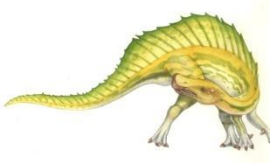
```
else{
    printf("我是子进程PID=：%d 接收消息\n",getpid())
        /*调用msgrcv函数，从消息队列读取消息*/
    if((msgrcv(qid,&msg,512,0,0))<0) {
        perror("读取消息出错");
        exit(1);}
        /*打印输出消息内容*/
    printf("读取的消息是：%s\n",(&msg)->msg_text);
        /*调用msgctl函数，删除系统中的消息队列*/
    if((msgctl(qid,IPC_RMID,NULL))<0){
        perror("删除消息队列出错");
        exit(1);}
    }
exit (0);}
```



- 分成两个独立的程序：msgsnd.c,msgrcv.c。分别编译和运行

```
[root@localhost syspro]# ./msgsnd
创建、打开的队列号是：131072
 我发送消息进程PID=：3784 发送消息
请输入要加入队列的消息：
sajhsdaj
[root@localhost syspro]#
```

```
[root@localhost syspro]# ./msgrcv
创建、打开的队列号是：131072
我是接收消息进程PID=：3798 接收消息
读取的消息是：sajhsdaj
```



管道通信

UNIX系统在操作系统的发展中最重要贡献之一是该系统首创了管道（pipes）。

- 管道是指用于连接一个读进程和一个写进程，以实现它们之间通信的共享文件，又称为pipe文件。向管道（共享文件）提供输入的发送进程（即写进程），而接收管道输出的接收进程（即读进程）可从管道中接收数据。管道通信是基于文件系统形式的一种通信方式。
- 管道分为无名管道和有名管道两种类型。无名管道是一个只存在于打开文件机构中的一个临时文件，从结构上没有文件路径名，不占用文件目录项。无名管道利用系统调用pipe(filedes)创建。

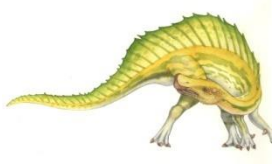
管道

- pipe系统调用的语法格式是：

  int filedes [2];

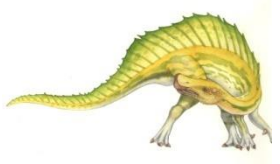
  int pipe (filedes);

- 内核创建一条管道完成下述工作：
  - 分配一个索引结点inode
  - 在系统打开文件表中创建一个读管道文件对象和一个写管道文件对象
  - 在创建管道的进程文件描述表中分配二表项，filedes[0]和filedes[1]分别指向系统打开文件表的读和写管道文件对象
  - 系统调用所涉及的数据结构如下图所示。



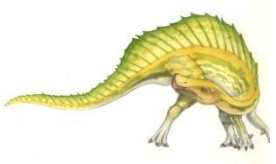
管道通信程序

```
#include <stdio.h>
#include <stdlib.h>
char parent[]={“A message from parent .”};
char child[]={“A message from child .”};
main()
{int chan1[2],chan2[2];
 char buf[100];
 if (pipe(chan1)==-1 || pipe(chan2)==-1) errexit(“pipe”);
 if (fork()>0)       /*父进程运行这段代码*/
 {close(chan1[0]);    close(chan2[1]);
   write(chan1[1],parent,sizeof parent);
   close(chan1[1]);
   read(chan2[0],buf,100);
```

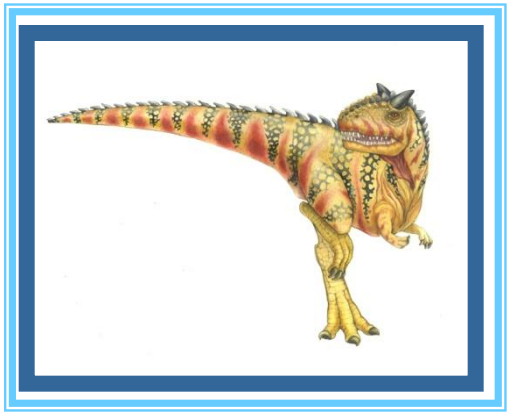


管道通信程序

```
    printf("parent process : %s \n",buf);
    close(chan2[0]);
   }
   else  /*子进程运行这段代码*/
   {close(chan1[1]); close(chan2[0]);
     read(chan1[0],buf,100);
     printf("child process : %s \n",buf);
     write(chan2[1],child,sizeof child);
     close(chan2[1]); close(chan1[0]);
    }
}
```

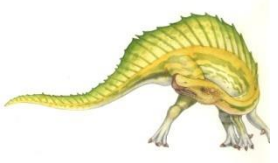


Linux线程

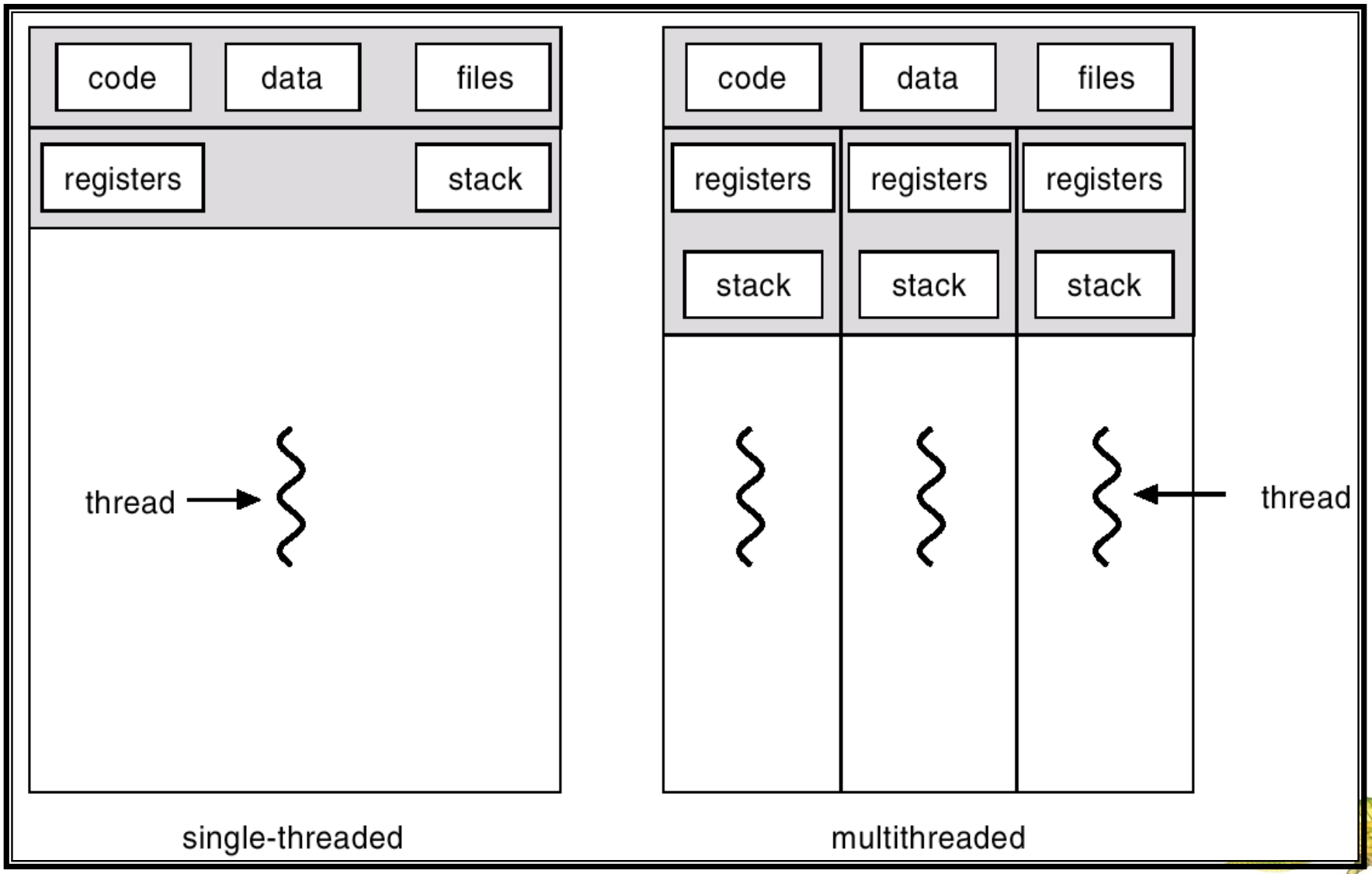


线程概念

- 进程概念的两个特征
  - 拥有资源的单位Unit of Resource ownership
    - 进程被分派保存进程映像的续地址空间
  - 分派的单位Unit of Dispatching
    - 进程是由一个或多个程序的一次执行
    - 可能会与其他进程交替执行
- 线程（轻型进程*lightweight process*），包括
  - 一个线程标识符a thread ID
  - 程序计数器program counter
  - 寄存器集register set
  - 栈空间stack space



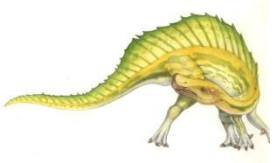
Single and Multithreaded Processes



Linux线程

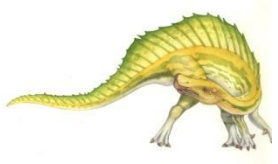
- Linux 2.6内核支持clone()系统调用创建线程。

- Pthread是POSIX标准为线程创建和同步定义的API



POSIX线程库

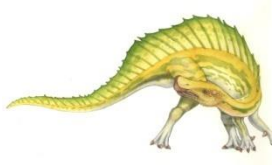
- POSIX线程库(Pthread):
  - pthread_create()：创建线程函数；
  - pthread_exit()：主动退出线程；
  - pthread_join()：用于将当前线程挂起来等待线程的结束。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源就被收回。
  - pthread_cancel()：终止另一个线程的执行。



线程创建

所需头文件	#include< pthread.h >
函数功能	创建一个线程
函数原型	**int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void *(*start_routine)(void*), void * arg);**
函数参数	**第1个参数：产生线程的标识符** **第2个参数：所产生线程的属性，通常设为NULL** **第3个参数：新的线程所执行的函数代码** **第4个参数：新的线程函数的参数**
函数返回值	执行成功则返回0，失败返回-1，而错误号保存在全局变量errno中。

例：pthread_create(&tid,&attr,runner,argv[1]);

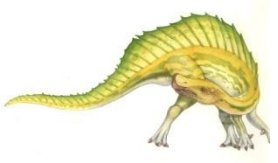


线程退出 pthread_exit

所需头文件	#include< pthread.h >
函数功能	终止一个线程
函数原型	**void pthread_exit(void *value_ptr);**
函数传入值	value_ptr：用户定义的指针，用来存储被等待线程的返回值。
函数返回值	执行成功则返回0，失败返回-1，而错误号保存在全局变量errno中。
备注	主线程中使用pthread_exit只会使主线程自身退出，产生的子线程继续执行；用return则所有线程退出。子线程或主线程使用exit，则整个线程全部终止。

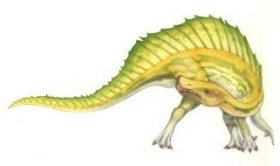
等待线程结束

所需头文件	#include< pthread.h >
函数功能	等待线程结束
函数原型	**int pthread_join(pthread_t thread, void **value_ptr);**
函数传入值	thread：等待线程的标识符 value_ptr：用户定义的指针，用来存储被等待线程的返回值。
函数返回值	执行成功则返回0，失败返回错误号。
备注	



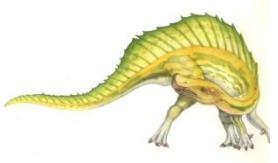
创建线程例

```
void * thread_fun1(void *arg){
    printf("thread %d  return\n",arg);
     ......
}
int main(void){
      pthread_t   tid1, tid2;
     pthread_create(&tid1, NULL, thread_fun, (void *)1); /*创建第1个线程*/
     pthread_create(&tid2, NULL, thread_fun, (void *)2);  /*创建第2个线程*/
     ......
     pthread_join(tid1, NULL);       /*等待第1个线程结束*/
      pthread_join(tid2, NULL);      /*等待第2个线程结束*/
     ......
}
```

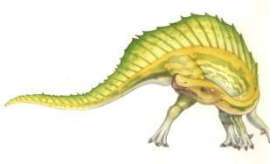


例：设计使用pthread线程库创建2个新线程，在父进程（也可以称为主线程）和新线程中分别显示进程id和线程id，并观察线程id数值。

```
//thread.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>  /*pthread_create()函数的头文件*/
#include <unistd.h>
pthread_t ntid1,ntid2;
void printids(const char *s) {/*各线程共享的函数*/
        pid_t     pid;
        pthread_t  tid;
        pid = getpid();
        tid = pthread_self();
        printf("%s  pid=  %u  tid= %u （0x%x）\n", s, (unsigned int)pid,
    (unsigned int)tid, (unsigned int)tid);
}
 void *thread_fun(void *arg) {/*新线程执行代码*/
        printids(arg);
        return NULL;
}
```



```
int main(void){
        int err;
                /*下列函数创建线程*/
        err = pthread_create(&ntid, NULL, thread_fun, "我是新线程1: ");
        if (err != 0) {fprintf(stderr, "创建线程失败: %s\n", strerror(err));
        exit(1);  }
        err = pthread_create(&ntid, NULL, thread_fun, "我是新线程2: ");
        if (err != 0) {printf(stderr, "创建线程失败: %s\n", strerror(err));
        exit(1);}
        printids("我是父进程:");
        pthread_join(ntid1, NULL);          /*等待第1个线程结束*/
        pthread_join(ntid2, NULL);          /*等待第2个线程结束*/
        return 0;
}
```



编译pthread线程方法

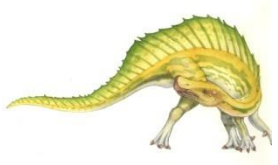
- 编译pthread线程的程序：

  gcc –lpthread -o thread thread.c

- Ubuntu 12.04后

  gcc -o thread thread.c –lpthread

```
[root@localhost syspro]# gcc -lpthread -o thread thread.c
[root@localhost syspro]# ./thread
我是主线程：  pid= 4274 tid= 3086489264 (0xb7f816b0)
我是新线程1：  pid= 4274 tid= 3086486432 (0xb7f80ba0)
我是新线程2：  pid= 4274 tid= 3075996576 (0xb757fba0)
```



Q&A