|  |
| --- |
| 深圳大学 |
| 操作系统之编程观察  （Linux平台）  (V 1.00) |
| 罗秋明 著 |

目录

[第1章 Linux安装与访问 1](#_Toc486237049)

[1.1. 安装Linux 1](#_Toc486237050)

[1.1.1. 下载Centos7 1](#_Toc486237051)

[1.1.2. Centos7安装 2](#_Toc486237052)

[1.2. 虚拟机安装Linux 7](#_Toc486237053)

[1.2.1. VirtualBox安装 7](#_Toc486237054)

[1.2.2. 虚拟机配置 10](#_Toc486237055)

[1.2.3. 虚拟机安装Linux 12](#_Toc486237056)

[1.3. ssh远程终端访问 13](#_Toc486237057)

[1.4. 初次接触Linux 14](#_Toc486237058)

[1.4.1. 简单操作 15](#_Toc486237059)

[1.4.2. 运行HelloWorld程序 18](#_Toc486237060)

[1.4.3. 操作系统接口 21](#_Toc486237061)

[1.5. 阅读注意事项 21](#_Toc486237062)

[1.6. 小结 22](#_Toc486237063)

[第2章 进程控制 23](#_Toc486237064)

[2.1. 进程基本概念 23](#_Toc486237065)

[2.1.1. 进程实体 24](#_Toc486237066)

[2.1.2. 进程间组织关系 25](#_Toc486237067)

[2.1.3. 进程控制命令 29](#_Toc486237068)

[2.2. 创建与撤销进程 30](#_Toc486237069)

[2.2.1. fork()创建子进程 30](#_Toc486237070)

[2.2.2. 孤儿进程和僵尸进程 32](#_Toc486237071)

[2.2.3. exec函数族 33](#_Toc486237072)

[2.2.4. 通过kill()撤销进程 35](#_Toc486237073)

[2.2.5. 创建守护进程 35](#_Toc486237074)

[2.3. 创建pthread线程 38](#_Toc486237075)

[2.3.1. 进程与线程 39](#_Toc486237076)

[2.3.2. 创建方法 40](#_Toc486237077)

[2.4. 进程/线程资源开销 41](#_Toc486237078)

[2.4.1. PCB开销 41](#_Toc486237079)

[2.4.2. 内存描述符开销 43](#_Toc486237080)

[2.5. 小结 44](#_Toc486237081)

[2.6. 练习 44](#_Toc486237082)

[第3章 进程调度 46](#_Toc486237083)

[3.1. 调度与均衡 46](#_Toc486237084)

[3.1.1. 调度与均衡框架 49](#_Toc486237085)

[3.1.2. 全系统的调度统计 50](#_Toc486237086)

[3.2. 进程状态及其转变 52](#_Toc486237087)

[3.2.1. 进程状态 52](#_Toc486237088)

[3.2.2. 状态转换 56](#_Toc486237089)

[3.2.3. 进程的调度统计 58](#_Toc486237090)

[3.3. 进程的调度 61](#_Toc486237091)

[3.3.1. 普通进程的 CFS调度 61](#_Toc486237092)

[3.3.2. 实时进程调度 65](#_Toc486237093)

[3.4. 进程迁移与负载均衡 72](#_Toc486237094)

[3.4.1. CFS进程的负载均衡 72](#_Toc486237095)

[3.4.2. 实时进程的负载均衡 77](#_Toc486237096)

[3.5. 小结 79](#_Toc486237097)

[3.6. 练习 79](#_Toc486237098)

[第4章 进程间通信与同步 80](#_Toc486237099)

[4.1. 进程间通信 80](#_Toc486237100)

[4.1.1. 管道 80](#_Toc486237101)

[4.1.2. System V IPC 83](#_Toc486237102)

[4.2. 进程间同步 92](#_Toc486237103)

[4.2.1. System V IPC信号量集 93](#_Toc486237104)

[4.2.2. POSIX信号量 101](#_Toc486237105)

[4.3. 小结 107](#_Toc486237106)

[4.4. 练习 107](#_Toc486237107)

[第5章 内存管理 108](#_Toc486237108)

[5.1. 虚存空间管理 108](#_Toc486237109)

[5.1.1. 进程映像 108](#_Toc486237110)

[5.1.2. 堆区 112](#_Toc486237111)

[5.1.3. 文件映射区 116](#_Toc486237112)

[5.1.4. 栈区 118](#_Toc486237113)

[5.1.5. 访问任意进程的虚存 122](#_Toc486237114)

[5.1.6. 虚存使用的物理页帧 124](#_Toc486237115)

[5.2. 分页机制与页表 126](#_Toc486237116)

[5.2.1. 分页机制 126](#_Toc486237117)

[5.2.2. 进程页表 130](#_Toc486237118)

[5.3. 物理内存组织管理 134](#_Toc486237119)

[5.3.1. 页帧、节点、内存域 134](#_Toc486237120)

[5.3.2. 空闲页帧管理——Buddy系统 140](#_Toc486237121)

[5.3.3. 物理内存分配与回收 142](#_Toc486237122)

[5.3.4. 内存回收 150](#_Toc486237123)

[5.4. 小结 160](#_Toc486237124)

[5.5. 练习 160](#_Toc486237125)

[第6章 综合——新进程创建到运行 161](#_Toc486237126)

[6.1. shell读入命令 161](#_Toc486237127)

[6.1.1. 用户空间与内核空间 161](#_Toc486237128)

[6.1.2. 读入命令 162](#_Toc486237129)

[6.2. 创建进程 167](#_Toc486237130)

[6.2.1. fork()拷贝进程 168](#_Toc486237131)

[6.2.2. 替换进程映像 168](#_Toc486237132)

[6.2.3. 开始运行新进程 169](#_Toc486237133)

[6.2.4. 进程映像与缺页 170](#_Toc486237134)

[6.3. 小结 170](#_Toc486237135)

[6.4. 练习 171](#_Toc486237136)

[第7章 VFS文件系统 172](#_Toc486237137)

[7.1. VFS 172](#_Toc486237138)

[7.1.1. VFS对象 173](#_Toc486237139)

[7.1.2. 文件系统类型 174](#_Toc486237140)

[7.2. 文件基本操作 175](#_Toc486237141)

[7.2.1. 命令行基本操作 175](#_Toc486237142)

[7.2.2. 编程接口 177](#_Toc486237143)

[7.3. 目录结构 180](#_Toc486237144)

[7.3.1. 树形结构 180](#_Toc486237145)

[7.3.2. 软/硬链接 181](#_Toc486237146)

[7.3.3. 文件系统创建与挂载 184](#_Toc486237147)

[7.4. 页缓存 188](#_Toc486237148)

[7.4.1. 页缓存基本概念 188](#_Toc486237149)

[7.4.2. 页缓存动态变化 190](#_Toc486237150)

[7.5. 非文件功能 191](#_Toc486237151)

[7.5.1. 交换 191](#_Toc486237152)

[7.5.2. 设备接口 195](#_Toc486237153)

[7.5.3. proc文件系统 199](#_Toc486237154)

[7.6. 小结 200](#_Toc486237155)

[7.7. 练习 201](#_Toc486237156)

[第8章 EXT2文件系统 202](#_Toc486237157)

[8.1. EXT2磁盘数据的组织 202](#_Toc486237158)

[8.1.1. 整体布局 202](#_Toc486237159)

[8.1.2. 超级块 203](#_Toc486237160)

[8.1.3. 块组描述符 205](#_Toc486237161)

[8.1.4. 索引节点 205](#_Toc486237162)

[8.1.5. 目录结构 209](#_Toc486237163)

[8.2. EXT2文件系统的创建 210](#_Toc486237164)

[8.2.1. 分配磁盘空间 210](#_Toc486237165)

[8.2.2. 创建环回设备 211](#_Toc486237166)

[8.2.3. 创建EXT2文件系统 211](#_Toc486237167)

[8.2.4. 挂载文件系统 211](#_Toc486237168)

[8.3. 查看EXT2磁盘数据 212](#_Toc486237169)

[8.3.1. 布局信息 212](#_Toc486237170)

[8.3.2. 块组描述符 215](#_Toc486237171)

[8.3.3. 索引节点与文件内容 216](#_Toc486237172)

[8.3.4. 目录结构 219](#_Toc486237173)

[8.4. 小结 223](#_Toc486237174)

[8.5. 练习 223](#_Toc486237175)

[附录（vi编辑命令） 224](#_Toc486237176)

# Linux安装与访问

* 1. 安装Linux
     1. 下载Centos7
     2. Centos7安装
  2. 虚拟机安装Linux
     1. VirtualBox安装
     2. 虚拟机配置
     3. 虚拟机安装Linux
  3. ssh远程终端访问
  4. 初次接触Linux
     1. 简单操作
     2. 运行HelloWorld程序

代码 1‑1 HelloWorld.c

#include <stdio.h>

int main()

{

printf(“HelloWorld!\n”);

return 0;

}

代码 1‑2 HelloWorld-getchar.c

#include <stdio.h>

int main()

{

printf(“HelloWorld!\n”);

getchar();

return 0;

}

* + 1. 操作系统接口
  1. 阅读注意事项
  2. 小结

# 进程控制

* 1. 进程基本概念
     1. 进程实体
     2. 进程间组织关系
     3. 进程控制命令

代码 2‑1 shell脚本示例（创建进程）

#!/usr/bin/bash

echo Running a shell script!

HelloWorld

* 1. 创建与撤销进程
     1. fork()创建子进程

代码 2‑2 fork()示例代码fork-demo.c

#include <stdio.h>

#include <unistd.h>

int main(int argc, char \*\* argv )

{

int pid = fork();

if(pid == -1 ) {

printf("error!\n");

} else if( pid ==0 ) {

printf("This is the child process!\n");

getchar();

} else {

printf("This is the parent process! child process id = %d\n", pid);

getchar();

wait(NULL);

}

return 0;

}

代码 2‑3 fork-twice-demo.c

#include <stdio.h>

#include <unistd.h>

int main(int argc, char \*\* argv )

{

int pid = fork();

if(pid == -1 ) {

printf("error!\n");

} else if( pid ==0 ) {

printf("This is the child process!\n");

fork();

getchar();

} else {

printf("This is the parent process! child process id = %d\n", pid);

fork();

getchar();

}

return 0;

}

* + 1. 孤儿进程和僵尸进程

代码 2‑4 zombie-demo.c

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

main()

{

pid\_t pid;

pid = fork();

if(pid < 0)

printf("error occurred!\n");

else if(pid == 0) {

printf("Hi father! I'm a ZOMBIE\n");

exit(0); //no one waits for this process.

}

else {

sleep(10);

wait(NULL); //the zombie process will be reaped now.

}

}

* + 1. exec函数族

代码 2‑5 fork-exec-demo.c

#include <stdio.h>

#include <unistd.h>

int main(int argc, char \*\* argv )

{

int pid = fork();

if(pid == -1 ) {

printf("error!\n");

} else if( pid ==0 ) {

printf("This is the child process!\n");

char \*argv[ ]={"ls", "-al", "/etc/passwd", NULL};

char \*envp[ ]={"PATH=/usr/bin", NULL};

execve("/usr/bin/ls", argv, envp);

printf("this printf()will not be executed,because it will be replaced by /usr/bin/ls code!\n");

} else {

printf("This is the parent process! child process id = %d\n", pid);

getchar();

}

return 0;

}

* + 1. 通过kill()撤销进程
    2. 创建守护进程

代码 2‑6 daemon-demo.c

#include <unistd.h>

#include <signal.h>

#include <fcntl.h>

#include <sys/syslog.h>

#include <sys/param.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int init\_daemon(void)

{

int pid;

int i;

// 1）忽略一些控制终端发来的信号,信号的内容请参考其他文献

signal(SIGTTOU,SIG\_IGN);

signal(SIGTTIN,SIG\_IGN);

signal(SIGTSTP,SIG\_IGN);

signal(SIGHUP ,SIG\_IGN);

// 2）在后台运行

if( pid=fork() ){ // 父进程

exit(0); //结束父进程，子进程继续

}else if(pid< 0){ // 出错

perror("fork");

exit(EXIT\_FAILURE);

}

// 3）脱离控制终端、登录会话和进程组

setsid();

// 4）禁止进程重新打开控制终端

if( pid=fork() ){ // 父进程

exit(0); // 结束第1代子进程，第2代子进程继续（第2代子进程不再是会话组长）

}else if(pid< 0){ // 出错

perror("fork");

exit(EXIT\_FAILURE);

}

// 5）关闭打开的文件描述符

// NOFILE 为文件描述符最大个数，不同系统有不同限制,<sys/param.h> 的宏定义

for(i=0; i< NOFILE; ++i){

close(i);

}

// 6）改变当前工作目录

chdir("/tmp");

// 7）重设文件创建掩模

umask(0);

// 8）处理 SIGCHLD 信号

signal(SIGCHLD,SIG\_IGN);

// 9) 输出日志

openlog("daemon-demo.log",LOG\_PID,0);

syslog(LOG\_INFO,"daemon-demo is running ...\n");

return 0;

}

int main(int argc, char \*argv[])

{

init\_daemon();

while(1);

return 0;

}

* 1. 创建pthread线程
     1. 进程与线程
     2. 创建方法

代码 2‑7 pthread-demo.c线程创建示例代码

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void thread(void)

{

printf("This is a pthread.\n");

sleep(10);

}

int main(void)

{

pthread\_t id;

int i,ret;

ret=pthread\_create(&id,NULL,(void \*) thread,NULL);

if(ret!=0){

printf ("Create pthread error!\n");

exit (1);

}

printf("This is the main process.\n");

pthread\_join(id,NULL);

return (0);

}

* 1. 进程/线程资源开销
     1. PCB开销

代码 2‑8 fork-100-demo.c代码

#include <stdio.h>

#include <unistd.h>

int main(int argc, char \*\* argv )

{

int i;

for(i=0;i<100;i++){

int pid = fork();

if(pid == -1 ) {

printf("error!\n");

} else if( pid ==0 ) {

printf("This is the child process!\n");

sleep(10);

return 0;

} else {

printf("This is the parent process! child process id = %d\n", pid);

}

}

sleep(10);

return 0;

}

代码 2‑9 pthread-100-demo.c代码

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void thread(void)

{

printf("This is a pthread.\n");

sleep(10);

}

int main(void)

{

pthread\_t id[100];

int i,ret;

for(i=0;i<100;i++){

ret=pthread\_create(&id[i],NULL,(void \*) thread,NULL);

if(ret!=0){

printf ("Create pthread error!\n");

exit (1);

}

}

printf("This is the main process.\n");

for(i=0;i<100;i++){

pthread\_join(id[i],NULL);

}

return (0);

}

* + 1. 内存描述符开销
  1. 小结
  2. 练习

# 进程调度

* 1. 调度与均衡
     1. 调度与均衡框架
     2. 全系统的调度统计
  2. 进程状态及其转变
     1. 进程状态

代码 3‑1 进程状态 (linux-3.13/include/linux/sched.h)

135 #define TASK\_RUNNING 0

136 #define TASK\_INTERRUPTIBLE 1

137 #define TASK\_UNINTERRUPTIBLE 2

138 #define \_\_TASK\_STOPPED 4

139 #define \_\_TASK\_TRACED 8

140 /\* in tsk->exit\_state \*/ 以下两个状态出现在task\_struct->exit\_state

141 #define EXIT\_ZOMBIE 16

142 #define EXIT\_DEAD 32

143 /\* in tsk->state again \*/

144 #define TASK\_DEAD 64

145 #define TASK\_WAKEKILL [[1]](#footnote-1) 128

146 #define TASK\_WAKING 256

147 #define TASK\_PARKED 512 在get\_task\_state()返回TASK\_INTERRUPTIBLE

148 #define TASK\_STATE\_MAX 1024

* + 1. 状态转换

代码 3‑2 HelloWorld-loop-getchar.c

#include <stdio.h>

int main()

{

long i,j,temp;

printf("HelloWorld!\n");

while(1)

{ for (i=0;i<1024\*1024\*1024;i++)

{ temp+=1;

}

printf("one iteration!\n");

getchar();

}

return 0;

}

* + 1. 进程的调度统计
  1. 进程的调度
     1. 普通进程的 CFS调度

代码 3‑3 Run-NICE.sh

./HelloWorld-loop &

nice -10 ./HelloWorld-loop &

* + 1. 实时进程调度

代码 3‑4 RT-process-demo.c

#include <sched.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

int main()

{

int rc,current\_scheduler\_policy;

struct sched\_param my\_params; //将要设置的调度参数

current\_scheduler\_policy=sched\_getscheduler(0);

printf("SCHED\_OTHER = %d SCHED\_FIFO =%d SCHED\_RR=%d \n", SCHED\_OTHER, SCHED\_FIFO, SCHED\_RR);

printf("the current scheduler = %d \n", current\_scheduler\_policy);

printf("press any key to change the current scheduler and priority to SCHED\_RR\n");

getchar(); //checkpoint 1

my\_params.sched\_priority=sched\_get\_priority\_max(SCHED\_RR); // 最高的RR实时优先级

rc=sched\_setscheduler(0,SCHED\_RR,&my\_params); //设置为RR实时进程

if(rc<0)

{

perror("sched\_setscheduler to SCHED\_RR error");

exit(0);

}

current\_scheduler\_policy=sched\_getscheduler(0);

printf("the current scheduler = %d \n", current\_scheduler\_policy);

printf("press any key to change the current scheduler and priority to SCHED\_FIFO\n");

getchar(); //checkpoint 2

my\_params.sched\_priority=sched\_get\_priority\_min(SCHED\_FIFO); // 最低FIFO实时优先级

rc=sched\_setscheduler(0,SCHED\_FIFO,&my\_params); //设置为FIFO实时进程

if(rc<0)

{

perror("sched\_setscheduler to SCHED\_FIFO error");

exit(0);

}

current\_scheduler\_policy=sched\_getscheduler(0);

printf("the current scheduler = %d \n", current\_scheduler\_policy);

printf("press any key to ange the current scheduler and priority to SCHED\_OTHER(CFS) \n");

getchar(); //checkpoint 3

rc=sched\_setscheduler(0,SCHED\_OTHER,&my\_params); //设置为普通进程[[2]](#footnote-2)

if(rc<0)

{ perror("sched\_setscheduler to SCHED\_OTHER error");

exit(0);

}

current\_scheduler\_policy=sched\_getscheduler(0);

printf("the current scheduler = %d \n", current\_scheduler\_policy);

printf("press any key to exit\n");

getchar(); //checkpoint 4

return 0;

}

代码 3‑5 RR-FIFO.sh脚本

[lqm@localhost lqm]# cat RR-FIFO.sh

RR-FIFO-sched 2 90&

RR-FIFO-sched 2 90&

sleep 5s

RR-FIFO-sched 1 95&

[lqm@localhost lqm]#

代码 3‑6 RR-FIFO-sched.c

#include <sched.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

int main(int argc,char \*\* argv)

{

long i,j,temp;

char cmd\_str[100];

int rc,current\_scheduler\_policy;

struct sched\_param my\_params; //将要设置的调度参数

if(argc!=3)

{ printf("usage:RR-FIFO-sched sched\_class priority \nsched\_class: 0 for CFS; 1 for FIFO; 2 for RR\n");

exit(1);

}

my\_params.sched\_priority=atoi(argv[2]);

rc=sched\_setscheduler(0,atoi(argv[1]),&my\_params);

if(rc<0)

{

perror("sched\_setscheduler error\n");

exit(0);

}

current\_scheduler\_policy=sched\_getscheduler(0);

printf("the PID:%d current scheduler = %d \n", getpid(),current\_scheduler\_policy);

for(i=0;i<1024\*1024\*1024;i++)

for(j=0;j<10;j++)

temp++;

sprintf(cmd\_str,"cat /proc/%d/sched > ./sched-%d ; date >> ./sched-%d", getpid(), getpid(), getpid());

system(cmd\_str); //记录各个进程的/proc/PID/sched以及时间信息

return 0;

}

* 1. 进程迁移与负载均衡
     1. CFS进程的负载均衡

代码 3‑7 HelloWorld-loop-nice+5.c

#include <stdio.h>

int main()

{

int k;

printf("HelloWorld!\n");

nice(5);

while(1);

k=k+1;

return 0;

}

* + 1. 实时进程的负载均衡

代码 3‑8 RT-Balance1.sh

root@localhost lqm]# cat RT-Balance1.sh

RR-FIFO-sched 1 10&

RR-FIFO-sched 1 20&

RR-FIFO-sched 1 30&

RR-FIFO-sched 1 40&

* 1. 小结
  2. 练习

# 进程间通信与同步

* 1. 进程间通信
     1. 管道

代码 4‑1 pipe-demo.c代码

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

int main()

{

pid\_t pid = 0;

int fds[2];

char buf[128];

int nwr = 0;

pipe(fds); //在fork()前执行

pid = fork();

if(pid < 0)

{

printf("Fork error!\n");

return -1;

}else if(pid == 0)

{

printf("This is child process, pid = %d\n", getpid());

printf("Child:waiting for message...\n");

close(fds[1]); //关闭写端fds[1]

nwr = read(fds[0], buf, sizeof(buf)); //从读端fds[0]读入数据

printf("Child:received\"%s\"\n", buf);

}else{

printf("This is parent process, pid = %d\n", getpid());

printf("Parent:sending message...\n");

close(fds[0]); //关闭写端fds[0]

strcpy(buf, "Message from parent！");

nwr = write(fds[1], buf, sizeof(buf)); //往写端fds[1]写出数据

printf("Parent:send %d bytes to child.\n", nwr);

}

return 0;

}

* + 1. System V IPC

代码 4‑2 msgtool.c代码

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#define MAX\_SEND\_SIZE 80

struct mymsgbuf { //消息的结构体

long mtype; //消息类型

char mtext[MAX\_SEND\_SIZE]; //消息内容

};

void send\_message(int qid, struct mymsgbuf \*qbuf, long type, char \*text);

void read\_message(int qid, struct mymsgbuf \*qbuf, long type);

void remove\_queue(int qid);

void change\_queue\_mode(int qid, char \*mode);

void usage(void);

int main(int argc, char \*argv[])

{

key\_t key;

int msgqueue\_id;

struct mymsgbuf qbuf;

if(argc == 1)

usage();

/\* Create unique key via call to ftok() \*/

key = ftok(".", 'm');

/\* Open the queue - create if necessary \*/

if((msgqueue\_id = msgget(key, IPC\_CREAT|0660)) == -1)

{

perror("msgget");

exit(1);

}

switch(tolower(argv[1][0]))

{

case 's': send\_message(msgqueue\_id, (struct mymsgbuf \*)&qbuf,atol(argv[2]), argv[3]);

break;

case 'r': read\_message(msgqueue\_id, &qbuf, atol(argv[2]));

break;

case 'd': remove\_queue(msgqueue\_id);

break;

case 'm': change\_queue\_mode(msgqueue\_id, argv[2]);

break;

default: usage();

}

return(0);

}

void send\_message(int qid, struct mymsgbuf \*qbuf, long type, char \*text)

{

/\* Send a message to the queue \*/

printf("Sending a message \n");

qbuf->mtype = type; //填写消息的类型

strcpy(qbuf->mtext, text); //填写消息内容

if((msgsnd(qid, (struct msgbuf \*)qbuf,

strlen(qbuf->mtext)+1, 0)) ==-1)

{

perror("msgsnd");

exit(1);

}

return;

}

void read\_message(int qid, struct mymsgbuf \*qbuf, long type)

{

/\* Read a message from the queue \*/

printf("Reading a message \n");

qbuf->mtype = type;

msgrcv(qid, (struct msgbuf \*)qbuf, MAX\_SEND\_SIZE, type, 0);

printf("Type: %ld Text: %s\n", qbuf->mtype, qbuf->mtext);

return;

}

void remove\_queue(int qid)

{

/\* Remove the queue \*/

msgctl(qid, IPC\_RMID, 0);

return;

}

void change\_queue\_mode(int qid, char \*mode)

{

struct msqid\_ds myqueue\_ds;

/\* Get current info \*/

msgctl(qid, IPC\_STAT, &myqueue\_ds);

/\* Convert and load the mode \*/

sscanf(mode, "%ho", &myqueue\_ds.msg\_perm.mode);

/\* Update the mode \*/

msgctl(qid, IPC\_SET, &myqueue\_ds);

return;

}

void usage(void)

{

fprintf(stderr, "msgtool - A utility for tinkering with msg queues\n");

fprintf(stderr, "nUSAGE: msgtool (s)end \n");

fprintf(stderr, " msgtool (r)ecv \n");

fprintf(stderr, " msgtool (d)elete\n");

fprintf(stderr, " msgtool (m)ode \n");

exit(1);

}

代码 4‑3 shmget-demo.c代码

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdlib.h>

#include <stdio.h>

#define BUFSZ 4096

int main ( void )

{

int shm\_id;

shm\_id=shmget(IPC\_PRIVATE, BUFSZ, 0666 ) ; //创建共享内存

if (shm\_id < 0 ) {

perror( "shmget fail!\n" ) ;

exit ( 1 );

}

printf ( "Successfully created segment : %d \n", shm\_id ) ;

system( "ipcs -m"); //执行ipcs –m命令，显示系统的共享内存信息

return 0;

}

代码 4‑4 shmatt-write-demo.c代码

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main ( int argc, char \*argv[] )

{

int shm\_id ;

char \* shm\_buf;

if ( argc != 2 ){

printf ( "USAGE: atshm <identifier>" );

exit (1 );

}

shm\_id = atoi(argv[1]);

if ( (shm\_buf = shmat( shm\_id, 0, 0)) < (char \*) 0 ){ //映射共享内存到进程空间

perror ( "shmat fail!\n" );

exit (1);

}

printf ( " segment attached at %p\n", shm\_buf );

system("ipcs -m"); //显示共享内存信息

strcpy(shm\_buf,"Hello shared memory!\n");

getchar();

if ( (shmdt(shm\_buf)) < 0 ) { //解除共享内存的映射

perror ( "shmdt");

exit(1);

}

printf ( "segment detached \n" );

system ( "ipcs -m " );

getchar();

exit ( 0 );

}

代码 4‑5 shmatt-read-demo.c代码

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main ( int argc, char \*argv[] )

{

int shm\_id ;

char \* shm\_buf;

if ( argc != 2 ){

printf ( "USAGE: atshm <identifier>" );

exit (1 );

}

shm\_id = atoi(argv[1]);

if ( (shm\_buf = shmat( shm\_id, 0, 0)) < (char \*) 0 ){

perror ( "shmat fail!\n" );

exit (1);

}

printf ( " segment attached at %p\n", shm\_buf );

system("ipcs -m");

printf("The string in SHM is :%s\n",shm\_buf); //将共享内存区的内容打印出来

getchar();

if ( (shmdt(shm\_buf)) < 0 ) {

perror ( "shmdt");

exit(1);

}

printf ( "segment detached \n" );

system ( "ipcs -m " );

getchar();

exit ( 0 );

}

* 1. 进程间同步
     1. System V IPC信号量集

代码 4‑6 struct semid\_ds

struct semid\_ds {

struct ipc\_perm sem\_perm; // 信号量集的操作许可权限

struct sem \*sem\_base; // 某个信号量sem结构数组的指针

ushort sem\_nsems; // sem\_base 数组的个数=信号量个数

time\_t sem\_otime; //最后一次成功修改信号量数组的时间

time\_t sem\_ctime; //成功创建时间

};

代码 4‑7 struct struct sem

struct sem {

int semval; /\* current value \*/

int sempid; /\* pid of last operation \*/

struct list\_head sem\_pending; /\* pending single-sop operations \*/

};

代码 4‑8 sys-V-semget-demo.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/sem.h>

#include <sys/ipc.h>

#define SEM\_R 0400 //用户（属主）读

#define SEM\_A 0200 //用户（属主）写

#define SVSEM\_MODE (SEM\_R | SEM\_A | SEM\_R>>3 | SEM\_R>>6)

int main(int argc,char \*argv[])

{

int c,oflag,semid,nsems;

oflag = SVSEM\_MODE | IPC\_CREAT; //设置创建模式

while((c = getopt(argc,argv,"e"))!= -1) //根据参数选项e决定是否IPC\_EXCL模式

{

switch(c)

{

case 'e':

oflag |= IPC\_EXCL;

break;

}

}

if (optind != argc -2) //判断命令行参数个数是否合法

{

printf("usage: semcreate [-e] <pathname> <nsems>");

exit(0);

}

nsems = atoi(argv[optind+1]); //获取信号量集合中的信号量个数

semid = semget(ftok(argv[optind],0),nsems,oflag); //创建信号量集

exit(0);

}

代码 4‑9 union semun

union semun {

int val; /\* value for SETVAL \*/

struct semid\_ds \_\_user \*buf; /\* buffer for IPC\_STAT & IPC\_SET \*/

unsigned short \_\_user \*array; /\* array for GETALL & SETALL \*/

struct seminfo \_\_user \*\_\_buf; /\* buffer for IPC\_INFO \*/

void \_\_user \*\_\_pad;

};

代码 4‑10 sys-V-sem-getval-demo.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/sem.h>

#include <sys/ipc.h>

union semun

{

int val;

struct semid\_ds \*buf;

unsigned short \*array;

};

int main(int argc,char \*argv[])

{

int semid,nsems,i;

struct semid\_ds seminfo;

unsigned short \*ptr;

union semun arg;

if(argc != 2)

{

printf("usage: semgetvalues <pathname>");

exit(0);

}

semid = semget(ftok(argv[1],0),0,0); //打开已经存在的信号量

arg.buf = &seminfo;

semctl(semid,0,IPC\_STAT,arg); //获取信号量集的数据，保存到semid\_ds结构

nsems = arg.buf->sem\_nsems; //获取信号量的数目

ptr = calloc(nsems,sizeof(unsigned short));

arg.array = ptr;

semctl(semid,0,GETALL,arg); //获取各个信号量的值

for(i=0;i<nsems;i++)

printf("semval[%d] = %d\n",i,ptr[i]);

exit(0);

}

代码 4‑11 sys-V-sem-setval-demo.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/sem.h>

#include <sys/ipc.h>

union semun //定义信号量操作联合体结构

{

int val;

struct semid\_ds \*buf;

unsigned short \*array;

};

int main(int argc,char \*argv[])

{

int semid,nsems,i;

struct semid\_ds seminfo;

unsigned short \*ptr;

union semun arg;

if(argc < 2)

{

printf("usage: semsetvalues <pathname>[values ...]");

exit(0);

}

semid = semget(ftok(argv[1],0),0,0); //打开已经存在的信号量集合

arg.buf = &seminfo;

semctl(semid,0,IPC\_STAT,arg); //获取信号量集的相关信息

nsems = arg.buf->sem\_nsems; //信号量的个数

if(argc != nsems + 2 )

{

printf("%s semaphores in set,%d values specified",nsems,argc-2);

exit(0);

}

//分配信号量

ptr = calloc(nsems,sizeof(unsigned short));

arg.array = ptr;

for(i=0;i<nsems;i++) //设置各个信号量的值

ptr[i] = atoi(argv[i+2]);

semctl(semid,0,SETALL,arg); //通过arg设置信号量集合

exit(0);

}

代码 4‑12 struct sembuf

struct sembuf {

unsigned short sem\_num; /\* semaphore index in array \*/

short sem\_op; /\* semaphore operation \*/

short sem\_flg; /\* operation flags \*/

};

代码 4‑13 sys-V-sem-p&v-demo.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/sem.h>

#include <sys/ipc.h>

int main(int argc,char \*argv[])

{

int c,i,flag,semid,nops;

struct sembuf \*ptr;

flag = 0;

while( ( c = getopt(argc,argv,"nu")) != -1) //根据命令行参数设置操作模式

{

switch(c)

{

case 'n':

flag |= IPC\_NOWAIT; //非阻塞

break;

case 'u':

flag |= SEM\_UNDO; //不可恢复

break;

}

}

if(argc - optind < 2)

{

printf("usage: semops [-n] [-u] <pathname> operation...");

exit(0);

}

if((semid = semget(ftok(argv[optind],0),0,0)) == -1) //打开一个已经存在的信号量集合

{

perror("semget() error");

exit(-1);

}

optind++; //指向当前第一个信号量的位置

nops = argc - optind; //信号量个数

ptr = calloc(nops,sizeof(struct sembuf));

for(i=0;i<nops;++i)

{

ptr[i].sem\_num = i; //选择被操作的信号量

ptr[i].sem\_op = atoi(argv[optind+i]); //设置信号量的值

ptr[i].sem\_flg = flag; //设置操作模式

}

if(semop(semid,ptr,nops) == -1) //对信号量执行P/V操作

{

perror("semop() error");

exit(-1);

}

exit(0);

}

代码 4‑14 sys-V-sem-delete-demo.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/sem.h>

#include <sys/ipc.h>

union semun

{

int val;

struct semid\_ds \*buf;

unsigned short \*array;

};

int main(int argc,char \*argv[])

{

int semid,nsems,i;

struct semid\_ds seminfo;

unsigned short \*ptr;

union semun arg;

if(argc != 2)

{

printf("usage: delete\_sems <pathname>");

exit(0);

}

semid = semget(ftok(argv[1],0),0,0); //打开已经存在的信号量

arg.buf = &seminfo;

int ret = semctl(semid,0,IPC\_RMID,NULL); //删除信号量集

if(ret == -1)

{

printf("delete sem error\n");

exit(1);

}

exit(0);

}

* + 1. POSIX信号量

代码 4‑15 psem-named-open-demo.c代码

#include <semaphore.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

int main(int argc,char \*\*argv)

{

sem\_t \*sem;

if(argc!=2)

{

printf("please input a file name to act as the ID of the sem!\n");

exit(1);

}

sem=sem\_open(argv[1],O\_CREAT,0644,1); //创建一个命名的POSIX信号量

exit(0);

}

代码 4‑16 psem-named-wait-demo.c

#include <semaphore.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

int main(int argc,char \*\*argv)

{

sem\_t \*sem;

int val;

if(argc!=2)

{

printf("please input a file name!\n");

exit(1);

}

sem=sem\_open(argv[1],0); 获取信号量对象

sem\_wait(sem); 执行P操作（-1操作）

sem\_getvalue(sem,&val); 获得出当前信号量的值

printf("pid %ld has semaphore,value=%d\n",(long)getpid(),val);

return 0;

}

代码 4‑17 psem-named-post-demo.c

#include <semaphore.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

int main(int argc,char \*\*argv)

{

sem\_t \*sem;

int val;

if(argc!=2)

{

printf("please input a file name!\n");

exit(1);

}

sem=sem\_open(argv[1],0);

sem\_post(sem); //对信号量执行P操作（增1）

sem\_getvalue(sem,&val);

printf("value=%d\n", val);

exit(0);

}

代码 4‑18 psem-named-unlink-demo.c代码

#include <semaphore.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

int main(int argc,char \*\*argv)

{

if(argc!=2)

{

printf("please input a file name to act as the ID of the sem!\n");

exit(1);

}

sem\_unlink(argv[1]); //撤销指定的信号量

exit(0);

}

代码 4‑19 no-mutex-demo.c代码

#include <pthread.h>

#include <unistd.h>

#include <stdio.h>

#include <malloc.h>

#define thread\_num 16

#define MB 1024 \* 1024

int \*array;

int length; //array length

int count;

int t; //number of thread

void \*count3s\_thread(void\* id);

int main()

{

int i;

int tid[thread\_num];

pthread\_t threads[thread\_num];

length = 64 \* MB;

array = malloc(length\*4); //256MB

for (i = 0; i < length; i++) //initial array

array[i] = i % 2 ? 4 : 3; //偶数i对应数值3

for (t = 0; t < thread\_num; t++) //循环创建16个线程

{

count = 0;

tid[t]=t;

int err = pthread\_create(&(threads[t]), NULL, count3s\_thread,&(tid[t]) );

if (err)

{

printf("create thread error!\n");

return 0;

}

}

for (t = 1; t < thread\_num; t++)

pthread\_join(threads[t], NULL); //等待前面创建的计数线程结束

printf("Total count= %d \n",count);

return 0;

}

void \*count3s\_thread(void\* id) //计数线程执行的函数

{

/\*compute portion of the array that this thread should work on\*/

int length\_per\_thread = length / thread\_num; //length of every thread

int start = \*(int \*)id \* length\_per\_thread; //every thread start position

int i;

for (i = start; i < start + length\_per\_thread; i++)

{

if (array[i] == 3)

{

count++; //计数，为加入互斥保护

}

}

}

代码 4‑20 no-mutex-demo.c代码

#include <pthread.h>

#include <unistd.h>

#include <stdio.h>

#include <malloc.h>

#define thread\_num 16

#define MB 1024 \* 1024

int \*array;

int length; //array length

int count;

int t; //number of thread

void \*count3s\_thread(void\* id);

mutex m; //增加一个互斥量

int main()

{

pthread\_mutex\_init(&m,NULL); //初始化互斥量

……

}

void \*count3s\_thread(void\* id)

{

/\*compute portion of the array that this thread should work on\*/

int length\_per\_thread = length / thread\_num; //length of every thread

int start = \*(int \*)id \* length\_per\_thread; //every thread start position

int i;

for (i = start; i < start + length\_per\_thread; i++)

{

if (array[i] == 3)

{

pthread\_mutex\_lock(&m); //进入临界区

count++;

pthread\_mutex\_unlock(&m); //退出临界区

}

}

}

* 1. 小结
  2. 练习

# 内存管理

* 1. 虚存空间管理
     1. 进程映像
     2. 堆区

代码 5‑1 Mem-malloc.c代码

#include <stdio.h>

#include <malloc.h>

void \*bufs[5];

int main()

{

printf("Before malloc()s.\n");

getchar();

bufs[0]=malloc(1024\*8);

bufs[1]=malloc(1024\*8);

bufs[2]=malloc(1024\*8);

bufs[3]=malloc(1024\*8);

bufs[4]=malloc(1024\*8);

printf("After 5 malloc()s.\n");

getchar();

free(bufs[1]);

free(bufs[3]);

printf("After 2 free()s.\n");

getchar();

return 0;

}

* + 1. 文件映射区

代码 5‑2 mmap-demo.c

#include <fcntl.h>

#include <sys/mman.h>

#include <stdio.h>

#include <string.h>

int main()

{

int fd;

void \*start\_addr;

struct stat sb;

char str1[]="Modification in the memory---------";

fd = open("./file-mapped.txt", O\_RDWR); //打开./file-mapped.text

fstat(fd, &sb); // 取得文件大小

start\_addr = mmap(NULL, sb.st\_size, PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, 0);

if(start\_addr == MAP\_FAILED) // 判断是否映射成功

return;

printf("Mapped area stared by addr:%llx:\n%s\n", start\_addr,start\_addr);

getchar();

strcpy(start\_addr,str1); //修改共享内存区

printf("Write string into the mapped area!\n");

getchar();

munmap(start\_addr, sb.st\_size); //解除映射

close(fd);

return 0;

}

* + 1. 栈区

代码 5‑3 stack-demo.c

#include<stdio.h>

void up\_and\_down(int);

int main(void)

{

printf("Initial state, please check /proc/PID/maps!\n");

getchar();

up\_and\_down(1);

return 0;

}

void up\_and\_down(int n)

{

int var\_in\_stack\_frame[64\*1024]; //局部数组，占用堆栈空间

printf("Level %d:n location %p\n",n,&n);

getchar();

if(n<3)

up\_and\_down(n+1);

printf("Level %d:n location %p\n",n,&n);

getchar();

return;

}

代码 5‑4 pthread-stack-demo.c线程创建示例代码

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void thread(void)

{

int thread\_var=0xff;

printf("This is a pthread. thread\_var addr:%llx\n",&thread\_var);

sleep(100);

}

int main(void)

{

pthread\_t id1,id2;

int i,ret;

getchar();

ret=pthread\_create(&id1,NULL,(void \*) thread,NULL);

if(ret!=0){

printf ("Create pthread error!\n");

exit (1);

}

ret=pthread\_create(&id2,NULL,(void \*) thread,NULL);

if(ret!=0){

printf ("Create pthread error!\n");

exit (1);

}

printf("This is the main process. i addr:%llx\n",&i);

pthread\_join(id1,NULL);

pthread\_join(id2,NULL);

return (0);

}

* + 1. 访问任意进程的虚存

代码 5‑5 show-var-addr.c代码

#include <stdio.h>

int main()

{

char Mesg[]="HelloWorld!\n";

printf("Mesg's addr:%llx\n",Mesg);

getchar();

return 0;

}

代码 5‑6 read-proc-mem.c代码

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/mman.h>

#include<fcntl.h>

#include<string.h>

#include<sys/ptrace.h>

int main(int argc,char \*\*argv)

{

char filename[200];

char buf[200];

bzero(filename,200);

bzero(buf,200);

sprintf(filename,"/proc/%s/mem",argv[1]); //根据命令参数确定/proc/PID/mem文件名

int fd=open(filename,O\_RDONLY); //打开/proc/PID/mem

int pid;

sscanf(argv[1],"%llx",&pid);

ptrace(PTRACE\_ATTACH,pid,0,0); //ptrace跟踪目标进程

long offset;

sscanf(argv[2],"%llx",&offset);

off\_t r=lseek64(fd,offset,SEEK\_SET); //其中的定位使用64位定位方式

if(r==-1)printf("lseek error:%m\n");

printf("位置:%x\n",r);

ssize\_t size=read(fd, buf, 200); //读入目标进程的/proc/PID/mem内容

if(size==-1) printf("read error:%m\n");

printf("Data at offset%llx:%s\n",offset,buf); //输出目标进程指定地址上的内容

close(fd);

return 0;

}

* + 1. 虚存使用的物理页帧
  1. 分页机制与页表
     1. 分页机制

代码 5‑7 Linux/include/asm-generic/page.h

39 typedef struct {

40 unsigned long pte;

41 } pte\_t;

42 typedef struct {

43 unsigned long pmd[16];

44 } pmd\_t;

45 typedef struct {

46 unsigned long pgd;

47 } pgd\_t;

48 typedef struct {

49 unsigned long pgprot;

50 } pgprot\_t;

51 typedef struct page \*pgtable\_t;

* + 1. 进程页表

代码 5‑8 show-virt-addr.c代码

#include <stdio.h>

void main()

{

char \*str = "Hello world!\n";

printf("%s @ %p\n", str, str);

getchar();

return;

}

* 1. 物理内存组织管理
     1. 页帧、节点、内存域
     2. 空闲页帧管理——Buddy系统

代码 5‑9 free\_arae (linux-3.13/[include/](http://lxr.linux.no/linux+v2.6.24/include/)[linux/](http://lxr.linux.no/linux+v2.6.24/include/linux/)[mmzone.h](http://lxr.linux.no/linux+v2.6.24/include/linux/mmzone.h))

83 struct free\_area {

84 struct list\_head free\_list[MIGRATE\_TYPES];

85 unsigned long nr\_free;

86 };

* + 1. 物理内存分配与回收

代码 5‑10 process-pages-demo.c代码

#include <stdio.h>

#include <sys/mman.h>

#include <stdlib.h>

#include <fcntl.h>

#define MB 1024\*1024

#define KB 1024

int main()

{

int i,j;

char temp;

int fd= -1;

char \* filemap\_buf;

char \* anon\_buf;

printf("-----\n");

getchar(); // checkpoint 1 initial state

if((fd=open("/home/lqm/tmpfile",O\_RDWR))<0)

printf("open tmpfile err!\n");

if((filemap\_buf=(char \*)mmap(NULL,8\*MB,PROT\_READ|PROT\_WRITE,MAP\_SHARED,fd,0))==MAP\_FAILED)

printf("mmap fail!\n");

anon\_buf=(char \*)malloc(16\*MB);

printf("allocated/mmapped\n");

printf("filemapped @ %p \n",filemap\_buf);

printf("anon @ %p \n",anon\_buf);

getchar(); // checkpoint 2 allocation

for (i=0;i<4\*MB;i+=4096)

temp=filemap\_buf[i];

for(i=0;i<8\*MB;i+=4096)

temp=\*(anon\_buf+i);

printf("read finish\n ");

getchar(); // checkpoint 3 read finish

for(i=0;i<2\*MB;i+=4096)

filemap\_buf[i]=temp;

for(i=0;i<4\*MB;i+=4096)

anon\_buf[i]=0xaa;

printf("write finish \n");

getchar(); // checkpoint 4 write finish/make the pageframes dirty

munmap(filemap\_buf,8\*MB);

free(anon\_buf);

printf("all done \n");

getchar(); // checkpoint 5

return 0;

}

* + 1. 内存回收

代码 5‑11 reclaim-swap-writeback-demo.c代码

#include <stdio.h>

#include <sys/mman.h>

#include <stdlib.h>

#include <fcntl.h>

#define MB 1024\*1024

#define KB 1024

int main()

{

int i,pid;

char temp;

int fd= -1;

char \*filemap\_buf;

char command\_smaps[100];

char command\_status[100];

char command\_meminfo[100];

char command\_maps[100];

char \* anon\_buf;

pid=getpid();

sprintf(command\_smaps,"cat /proc/%d/smaps > smaps-check",pid);

sprintf(command\_status,"cat /proc/%d/status > status-check",pid);

sprintf(command\_meminfo,"cat /proc/meminfo >meminfo-check");

sprintf(command\_maps,"cat /proc/%d/maps >maps-check",pid);

if((fd=open("/home/lqm/tmpfile",O\_RDWR))<0)

printf("open tmpfile err!\n");

if((filemap\_buf=(char \*) mmap(NULL, 8\*MB, PROT\_READ|PROT\_WRITE, MAP\_SHARED,fd,0)) == MAP\_FAILED )

printf("mmap fail!\n");

mlock(filemap\_buf,8\*KB);

anon\_buf=(char \*)malloc(16\*MB);

mlock(anon\_buf,16\*KB);

printf("allocated/mmapped\n");

printf("filemapped @ %p \n",filemap\_buf);

printf("anon @ %p \n",anon\_buf);

system(command\_maps);

system(command\_smaps);

system(command\_status);

system(command\_meminfo);

getchar(); // checkpoint 1 allocation

for(i=0;i<4\*MB;i+=4096)

filemap\_buf[i]=temp;

for(i=0;i<8\*MB;i+=4096)

anon\_buf[i]=0xaa;

printf("write finish \n");

system(command\_smaps);

system(command\_status);

system(command\_meminfo);

getchar(); // checkpoint 2 write finish/make the pageframes dirty

//run pages-blackhole-demo.c

printf("please run pages-blackhole-demo\n");

getchar(); // checkpoint 3 be swapped out

system(command\_smaps);

system(command\_status);

system(command\_meminfo);

getchar();

printf("compete for the page frames\n");

while(1) //compete for the page frames, by writing

{ for(i=0;i<1\*MB;i+=4096)

filemap\_buf[i]=temp;

for(i=0;i<2\*MB;i+=4096)

anon\_buf[i]=0xaa;

}

getchar(); // checkpoint 4 will never get to this point

munmap(filemap\_buf,8\*MB);

free(anon\_buf);

return 0;

}

代码 5‑12 pages-blackhole-demo.c代码

#include <stdio.h>

#include <sys/mman.h>

#include <stdlib.h>

#include <fcntl.h>

#define MB 1024\*1024

int main()

{

int i,j;

char temp;

int fd= -1;

char \* filemap\_buf;

char \* anon\_buf;

anon\_buf=(char \*)malloc(2000\*MB);

while(1)

{ for(i=0;i<2000\*MB;i+=4096)

anon\_buf[i]=0xaa;

sleep(1);

}

return 0;

}

* 1. 小结
  2. 练习

# 综合——新进程创建到运行

* 1. shell读入命令
     1. 用户空间与内核空间
     2. 读入命令
  2. 创建进程
     1. fork()拷贝进程
     2. 替换进程映像
     3. 开始运行新进程
     4. 进程映像与缺页
  3. 小结
  4. 练习

# VFS文件系统

* 1. VFS
     1. VFS对象
     2. 文件系统类型
  2. 文件基本操作
     1. 命令行基本操作
     2. 编程接口

代码 7‑1 file-op-demo.c

[root@localhost lqm]# cat file-op-demo.c

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main()

{

char buf\_byte[1];

int i;

int fd1,fd2,fd3;

fd1=open("file1.txt",O\_WRONLY|O\_CREAT);

for(i=0;i<4096;i++)

{ buf\_byte[0]= (i % 10)+'0';

write(fd1,buf\_byte,1);

}

printf("fd of file1.txt is %d \n",fd1);

getchar(); //check point 1

close(fd1);

getchar(); //check point 2

fd2=open("file2.txt",O\_WRONLY|O\_CREAT);

fd1=open("file1.txt",O\_RDONLY);

lseek(fd1,1026,SEEK\_SET );

for(i=0;i<10;i++)

{ read(fd1,buf\_byte,1);

write(fd2,buf\_byte,1);

}

printf("fd of file1.txt is %d. fd of file2.txt is %d",fd1,fd2);

getchar(); //check point 3

dup2(fd2,8);

write(8,"Hello!",6);

fd3=open("file2.txt",O\_WRONLY);

write(fd3,"open-again",10);

getchar(); //check point 4

return 0;

}

* 1. 目录结构
     1. 树形结构
     2. 软/硬链接
     3. 文件系统创建与挂载
  2. 页缓存
     1. 页缓存基本概念
     2. 页缓存动态变化

代码 ‑2 page-cache-demo.c

#include <stdio.h>

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main()

{

long i;

int fd;

char dat\_buf[1];

fd=open("data-2G.dat",O\_RDONLY);

for(i=0;i<1024\*1024\*2;i+1024)

read(fd,dat\_buf,1);

return 0;

}

* 1. 非文件功能
     1. 交换

代码 7‑3 swap\_info[] (linux-3.13/mm/swapfile.c)

struct swap\_info\_struct \*swap\_info[MAX\_SWAPFILES];

代码 7‑4swap\_extent (linux-3.13/include/linux/swap.h)

136 struct swap\_extent {

137 struct list\_head list; 按地址顺序构成链表

138 pgoff\_t start\_page; 本子区的第一个页槽

139 pgoff\_t nr\_pages; 本子区的页槽数量

140 sector\_t start\_block; 本子区在磁盘上的起始块号

141 };

* + 1. 设备接口
    2. proc文件系统
  1. 小结
  2. 练习

# EXT2文件系统

* 1. EXT2磁盘数据的组织
     1. 整体布局

* + 1. 超级块

代码 8‑1 ext2\_super\_block (linux-3.13/fs/ext2/ext2.h)

410 struct ext2\_super\_block {

411 \_\_le32 s\_inodes\_count; /\* Inodes count \*/ 索引节点总数

412 \_\_le32 s\_blocks\_count; /\* Blocks count \*/ 盘块的总数

413 \_\_le32 s\_r\_blocks\_count; /\* Reserved blocks count \*/ 保留的盘块数

414 \_\_le32 s\_free\_blocks\_count; /\* Free blocks count \*/ 空闲盘块数

415 \_\_le32 s\_free\_inodes\_count; /\* Free inodes count \*/ 空闲索引节点数

416 \_\_le32 s\_first\_data\_block; /\* First Data Block \*/ 第一个数据盘块号

417 \_\_le32 s\_log\_block\_size; /\* Block size \*/ 盘块大小（以2的幂次表示）

418 \_\_le32 s\_log\_frag\_size; /\* Fragment size \*/ 片的大小

419 \_\_le32 s\_blocks\_per\_group; /\* # Blocks per group \*/ 块组内部的盘块数量

420 \_\_le32 s\_frags\_per\_group; /\* # Fragments per group \*/块组中的片数

421 \_\_le32 s\_inodes\_per\_group; /\* # Inodes per group \*/ 块组内的索引节点数

422 \_\_le32 s\_mtime; /\* Mount time \*/ 最后一次安装挂载时间

423 \_\_le32 s\_wtime; /\* Write time \*/ 最后一次写操作的时间

424 \_\_le16 s\_mnt\_count; /\* Mount count \*/ 安装挂载次数（检查后清零）

425 \_\_le16 s\_max\_mnt\_count; /\* Maximal mount count \*/最大挂载次数，超过则触发文件系统检查

426 \_\_le16 s\_magic; /\* Magic signature \*/ 文件系统魔数（标志）

427 \_\_le16 s\_state; /\* File system state \*/ 文件系统状态

428 \_\_le16 s\_errors; /\* Behaviour when detecting errors \*/出错时的行为

429 \_\_le16 s\_minor\_rev\_level; /\* minor revision level \*/ 文件系统的次版本号

430 \_\_le32 s\_lastcheck; /\* time of last check \*/上一次文件系统检查时间

431 \_\_le32 s\_checkinterval; /\* max. time between checks \*/ 两次检查之间的间隔

432 \_\_le32 s\_creator\_os; /\* OS \*/ 创建该文件系统的操作系统

433 \_\_le32 s\_rev\_level; /\* Revision level \*/ 版本号

434 \_\_le16 s\_def\_resuid; /\* Default uid for reserved blocks \*/保留块的缺省UID

435 \_\_le16 s\_def\_resgid; /\* Default gid for reserved blocks \*/保留块的缺省用户组ID

436 /\*

437 \* These fields are for EXT2\_DYNAMIC\_REV superblocks only.

438 \*

439 \* Note: the difference between the compatible feature set and

440 \* the incompatible feature set is that if there is a bit set

441 \* in the incompatible feature set that the kernel doesn't

442 \* know about, it should refuse to mount the filesystem.

443 \*

444 \* e2fsck's requirements are more strict; if it doesn't know

445 \* about a feature in either the compatible or incompatible

446 \* feature set, it must abort and not try to meddle with

447 \* things it doesn't understand...

448 \*/

449 \_\_le32 s\_first\_ino; /\* First non-reserved inode \*/第一个非保留的索引节点号

450 \_\_le16 s\_inode\_size; /\* size of inode structure \*/ 索引节点大小

451 \_\_le16 s\_block\_group\_nr; /\* block group # of this superblock \*/ 本超级块的块组号

452 \_\_le32 s\_feature\_compat; /\* compatible feature set \*/具有兼容特点的位图

453 \_\_le32 s\_feature\_incompat; /\* incompatible feature set \*/具有非兼容特点的位图

454 \_\_le32 s\_feature\_ro\_compat; /\* readonly-compatible feature set \*/只读特点的位图

455 \_\_u8 s\_uuid[16]; /\* 128-bit uuid for volume \*/128位的文件系统标识符

456 char s\_volume\_name[16]; /\* volume name \*/ 卷名

457 char s\_last\_mounted[64]; /\* directory where last mounted \*/ 最后一个安装点路径名

458 \_\_le32 s\_algorithm\_usage\_bitmap; /\* For compression \*/ 用于压缩

459 /\*

460 \* Performance hints. Directory preallocation should only

461 \* happen if the EXT2\_COMPAT\_PREALLOC flag is on.

462 \*/

463 \_\_u8 s\_prealloc\_blocks; /\* Nr of blocks to try to preallocate\*/预分配的盘块数

464 \_\_u8 s\_prealloc\_dir\_blocks; /\* Nr to preallocate for dirs \*/ 目录的预分配盘块数

465 \_\_u16 s\_padding1; （填充位，无具体功能）

466 /\*

467 \* Journaling support valid if EXT3\_FEATURE\_COMPAT\_HAS\_JOURNAL set.

468 \*/ 以下是EXT3日志相关的

469 \_\_u8 s\_journal\_uuid[16]; /\* uuid of journal superblock \*/

470 \_\_u32 s\_journal\_inum; /\* inode number of journal file \*/

471 \_\_u32 s\_journal\_dev; /\* device number of journal file \*/

472 \_\_u32 s\_last\_orphan; /\* start of list of inodes to delete \*/

473 \_\_u32 s\_hash\_seed[4]; /\* HTREE hash seed \*/

474 \_\_u8 s\_def\_hash\_version; /\* Default hash version to use \*/

475 \_\_u8 s\_reserved\_char\_pad;

476 \_\_u16 s\_reserved\_word\_pad;

477 \_\_le32 s\_default\_mount\_opts;

478 \_\_le32 s\_first\_meta\_bg; /\* First metablock block group \*/

479 \_\_u32 s\_reserved[190]; /\* Padding to the end of the block \*/

480 };

* + 1. 块组描述符

代码 8‑2 ext2\_group\_desc (linux-3.13/fs/ext2/ext2.h)

194 struct ext2\_group\_desc

195 {

196 \_\_le32 bg\_block\_bitmap; /\* Blocks bitmap block \*/数据盘块位图所在的盘块号

197 \_\_le32 bg\_inode\_bitmap; /\* Inodes bitmap block \*/索引节点位图所在的盘块号

198 \_\_le32 bg\_inode\_table; /\* Inodes table block \*/索引节点表的起点所在的盘块号

199 \_\_le16 bg\_free\_blocks\_count; /\* Free blocks count \*/ 空闲盘块数

200 \_\_le16 bg\_free\_inodes\_count; /\* Free inodes count \*/ 空闲索引节点数

201 \_\_le16 bg\_used\_dirs\_count; /\* Directories count \*/ 在用目录个数

202 \_\_le16 bg\_pad; 字节对齐的填充

203 \_\_le32 bg\_reserved[3]; 保留（使得结构体整体占用32字节）

204 };

* + 1. 索引节点

代码 8‑3 ext2\_inode (Linux-3.13/fs/ext2/ext2.h)

297 struct ext2\_inode {

298 \_\_le16 i\_mode; /\* File mode \*/ 文件类型和访问权限

299 \_\_le16 i\_uid; /\* Low 16 bits of Owner Uid \*/ 所有者的UID（低16位）

300 \_\_le32 i\_size; /\* Size in bytes \*/ 文件长度（字节）

301 \_\_le32 i\_atime; /\* Access time \*/ 访问时间戳

302 \_\_le32 i\_ctime; /\* Creation time \*/ 创建时间戳

303 \_\_le32 i\_mtime; /\* Modification time \*/ 修改时间戳

304 \_\_le32 i\_dtime; /\* Deletion Time \*/ 删除时间戳

305 \_\_le16 i\_gid; /\* Low 16 bits of Group Id \*/ 所有者GID（低16位）

306 \_\_le16 i\_links\_count; /\* Links count \*/ 硬链接计数

307 \_\_le32 i\_blocks; /\* Blocks count \*/ 文件长度（block计数）

308 \_\_le32 i\_flags; /\* File flags \*/ 文件标志

309 union {

310 struct {

311 \_\_le32 l\_i\_reserved1;

312 } linux1; Linux中特定的信息之1

313 struct {

314 \_\_le32 h\_i\_translator;

315 } hurd1;

316 struct {

317 \_\_le32 m\_i\_reserved1;

318 } masix1;

319 } osd1; /\* OS dependent 1 \*/

320 \_\_le32 **i\_block[EXT2\_N\_BLOCKS]**;/\* Pointers to blocks \*/ 数据盘块指针数组

321 \_\_le32 i\_generation; /\* File version (for NFS) \*/

322 \_\_le32 i\_file\_acl; /\* File ACL \*/ 文件访问控制列表

323 \_\_le32 i\_dir\_acl; /\* Directory ACL \*/ 目录访问控制列表

324 \_\_le32 i\_faddr; /\* Fragment address \*/

325 union {

326 struct {

327 \_\_u8 l\_i\_frag; /\* Fragment number \*/

328 \_\_u8 l\_i\_fsize; /\* Fragment size \*/

329 \_\_u16 i\_pad1;

330 \_\_le16 l\_i\_uid\_high; /\* these 2 fields \*/

331 \_\_le16 l\_i\_gid\_high; /\* were reserved2[0] \*/

332 \_\_u32 l\_i\_reserved2;

333 } linux2; Linux中特定的信息之2

334 struct {

335 \_\_u8 h\_i\_frag; /\* Fragment number \*/

336 \_\_u8 h\_i\_fsize; /\* Fragment size \*/

337 \_\_le16 h\_i\_mode\_high;

338 \_\_le16 h\_i\_uid\_high;

339 \_\_le16 h\_i\_gid\_high;

340 \_\_le32 h\_i\_author;

341 } hurd2;

342 struct {

343 \_\_u8 m\_i\_frag; /\* Fragment number \*/

344 \_\_u8 m\_i\_fsize; /\* Fragment size \*/

345 \_\_u16 m\_pad1;

346 \_\_u32 m\_i\_reserved2[2];

347 } masix2;

348 } osd2; /\* OS dependent 2 \*/

349 };

* + 1. 目录结构

代码 8‑4 ext2\_dir\_entry和ext2\_dir\_entry\_2 (linux-3.13 /fs/ext2/ext2.h)

578 struct ext2\_dir\_entry {

579 \_\_le32 inode; /\* Inode number \*/ 目录项对应的索引节点

580 \_\_le16 rec\_len; /\* Directory entry length \*/ 目录项长度

581 \_\_le16 name\_len; /\* Name length \*/ 名字长度

582 char name[]; /\* File name, up to EXT2\_NAME\_LEN \*/

583 };

…

591 struct ext2\_dir\_entry\_2 {

592 \_\_le32 inode; /\* Inode number \*/

593 \_\_le16 rec\_len; /\* Directory entry length \*/

594 \_\_u8 name\_len; /\* Name length \*/

595 \_\_u8 file\_type; 文件类型（见下面603行）

596 char name[]; /\* File name, up to EXT2\_NAME\_LEN \*/

597 };

…

603 enum { ext2\_dir\_entry\_2中的file\_type枚举类型

604 EXT2\_FT\_UNKNOWN = 0,

605 EXT2\_FT\_REG\_FILE = 1, 普通文件

606 EXT2\_FT\_DIR = 2, 目录文件

607 EXT2\_FT\_CHRDEV = 3, 字符设备文件

608 EXT2\_FT\_BLKDEV = 4, 块设备文件

609 EXT2\_FT\_FIFO = 5, 命名管道FIFO

610 EXT2\_FT\_SOCK = 6, SOCK

611 EXT2\_FT\_SYMLINK = 7, 符号链接

612 EXT2\_FT\_MAX

613 };

* 1. EXT2文件系统的创建
     1. 分配磁盘空间
     2. 创建环回设备
     3. 创建EXT2文件系统
     4. 挂载文件系统
  2. 查看EXT2磁盘数据
     1. 布局信息
     2. 块组描述符
     3. 索引节点与文件内容
     4. 目录结构
  3. 小结
  4. 练习

1. 这个不是基本状态，表示即使在TASK\_UNINTERRUPTIBLE状态下也可以响应致命信号（终止进程） [↑](#footnote-ref-1)
2. sched\_setscheduler()对SCHED\_OTHER调度类不能设置优先级。 [↑](#footnote-ref-2)