



A 卷

2019–2020 学年第 2 学期
(2020 春季)

《System Programming》
期末考试卷

学号: 18373722

班级: 182111

姓名: 朱英豪

2020 年 06 月 24 日



《System Programming》

期末考试卷

注意事项：1、可用中文、图示回答。

2、简答题请标出标号，分点（条）回答；突出要点，简明扼要，条理清晰。

题号	1	2	3	4	5	总分
成绩						

题目：

Totally 5 Parts.....(100 points)

1. 简答题 (28 points). 答案直接写在题目下面。请先想好要点，再简明扼要，分小点罗列陈述。

(1) (16 points) 请回答有关 Linux 文件的几个问题。

(a) (5 points) 请列举出至少五种 Linux 下的文件类型，并分别解释这些类型文件的用途。

1. 普通文件：以字节为单位的数据流。包括二进制文件、文本文件、可执行文件等。文本文件和二进制文件对 Linux 来说没有区别，对普通文件的解释由处理该文件的关联程序进行。

2. 目录文件：目录文件中可以包含其他类型的文件，也可以包含目录。

3. 块设备文件：存储数据以供系统存取的接口设备，简单而言就是硬盘。(blog)

4. 字符设备文件，串行端口的接口设备，例如键盘、鼠标等等。(blog)

5. 特殊文件：特殊文件中较为常见的是符号链接文件，符号连接文件实际就是软链接文件。当访问软链接文件时，系统会从它的数据块中获取源文件的路径，再到这个路径中访问源文件。

(b) (5 points) 硬链接文件与软链接文件都能实现类似“快捷方式”的效果，但是它们使用起来还是有不同之处。对比分析这两种文件在使用时的异同点。

相同点：

1. 软硬链接都可以实现对于原文件的链接功能
2. 创建链接文件的命令均以 ln 开头：ln [选项] 源文件 目标文件。

不同点：

1. 选项缺省时，系统会创建一个硬链接文件；若搭配-s 选项，则会创建一个软链接文件。
2. 软链接文件就是一个新文件，执行 ln -s 时，目标文件会获取一个独享的 inode
3. 创建硬链接文件时，系统并不会去查找 inode 表，而是在硬链接文件上级目录的 dentry 中添加一条记录。
4. 创建硬链接文件时，源路径中的对象不能是一个目录，因为硬链接文件与源文件的 inode 相同；软链接的 inode 与源文件不同，不受此限制。
5. 硬链接文件可以在同一文件系统的不同目录中，但不能跨文件系统；而软链接文件与源文件的 inode 不同，因此软链接文件可以跨文件系统。

(c) (6 points) 举例阐述输入输出重定向的作用及使用方法。

作用：

Shell 默认可接受用户输入中断的命令并在执行后将错误信息和输出结果打印到终端，但在实际应用中，并非任何情境下我们都希望 Shell 执行这项默认操作。此时我们便需要重定向——即使用用户指定的文件而非默认资源（键盘、显示器）来获取或接受信息。

输入重定向：将标准输入指向某个文件，从而实现从文件中获取输入

输出重定向：指定某个文件作为标准输出设备来存储文件信息。

使用用法：

1. 输入重定向：格式为 命令<文件名，比如 sort<sp.txt，把 sp.txt 文件中的内容作为 sort 的输入。
2. 输出重定向：格式为 命令>文件名，比如 cat /etc/passwd > ps.log，cat 会输出 /etc/passwd 中内容，但此时并不会输出到屏幕上，而是输出到 ps.log 中

(2) (12 points) 对比 call 调用过程与外中断处理过程，以及栈在两个过程中的作用。

两者都要保护断点，跳至子程序或中断服务程序、保护现场、子程序或中断处理、恢复现场、恢复断点，两者都可以嵌套。

但是二者有区别的：call 过程的时间是已知的，可以查看代码知道跳转时间，而外中断是随机发生的。

子程序完全为主程序服务的，两者属于主从关系，而中断服务程序与主程序两者一般是无关的，两者是平行关系。

主程序调用子程序过程完全属于软件处理过程，不需要专门的硬件电路，而中断处理系统是一个软、硬件结合系统，需要专门的硬件电路才能完成中断处理的过程。

call：转移指令，分为近转移和远转移。

外中断：分为可屏蔽中断和不可屏蔽中断。

1. 可屏蔽中断，则先判断 IF，如果是 1 则响应，获取中断类型码 n，否则不响应；
2. 不可屏蔽中断，则中断类型码为 2，不需要再获取中断类型码。

然后标志寄存器入栈，

设置 IF=0, TF=0, CS、IP 入栈，

设置 CS、IP 为中断向量表中对应地址值，

从该处转去执行中断处理程序，

最后将 CS、IP 以及标志寄存器依次弹出还原。

栈在此保存了标志寄存器的值以及原来的程序 CS、IP 的值，相比于 call 多保存了标志寄存器的 IF 以及 TF，IF 的值决定是否需要响应可屏蔽中断，TF 的值决定是否有单步中断发生。

总结：二者在处理时，均用来保护断点及现场。Call 调用时，一般只存储 CS、IP，记录跳转前的指令位置，过程结束后弹出原来存储的值；外中断处理过程时，还会存储标志寄存器。

2. 程序阅读题(30 points)。

(1) (6 points) 请写出以下代码的执行结果，并简要说明各输出项的理由。

```
#!/bin/bash
```

```
var="hello"  
echo '$var'  
echo "$var"  
echo $var
```

执行结果：

```
$var  
hello  
hello
```

原因：

1. 单引号中的变量会被解释为简单的字符串
2. 双引号和直接引用会返回变量的值，而非变量的地址

(2) (12 points) 请首先阅读以下程序

```
#include<stdlib.h>  
#include<stdio.h>  
int main() {  
    int status;  
    pid_t pid;  
  
    printf("Hello\n");  
    pid = fork();  
    printf("%d\n", !pid);  
    if(pid != 0) {  
        if(waitpid(-1, &status, 0) > 0) {  
            if(WIFEXITED(status) != 0)  
                printf("%d\n", WEXITSTATUS(status));  
        }  
    }  
    printf("Bye\n");  
    exit(2);  
}
```

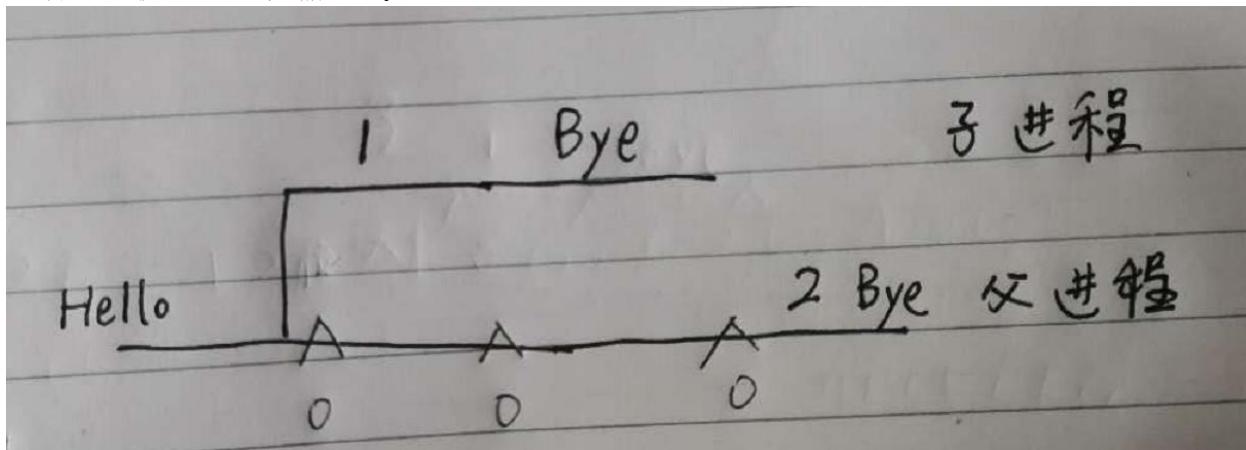
(a) 请简要分析一下此程序可能会有哪些输出。

程序先输出 Hello

子进程、父进程分别输出!pid 为 1 和 0

父进程输出 0 后，被挂起，等待子进程的结束，

子进程不进入 if 循环，打印 Bye 后，以 2 退出码退出，
此时父进程结束挂起状态，WEXITSTATUS(status)输出退出码 2，
接着向下执行，再次输出 Bye。



输出情况 1	输出情况 2	输出情况 3	输出情况 4
Hello	Hello	Hello	Hello
0	1	1	0
1	0	Bye	Bye
Bye	Bye	0	(子进程创建失败)
2	2	2	
Bye	Bye	Bye	

(b) 若此程序的输出为：

```
Hello
0
1
Bye
2
Bye
```

请描述这种情形下此程序的执行流程。

程序先输出 Hello
Fork 后，父进程先输出 0，子进程再输出 1
父进程被挂起，等待子进程的结束，
子进程不进入 if 循环，打印 Bye 后，以 2 退出码退出，
此时父进程结束挂起状态，WEXITSTATUS(status)输出退出码 2，
接着向下执行，再次输出 Bye。

(3) (12 points) 阅读下面的汇编代码片段，回答问题：

```
init:  
    mov ax,data  
    mov ds,ax  
  
    mov ax,0  
    mov es,ax  
  
    push es:[9 * 4]  
    pop ds:[0]  
    push es:[9 * 4 + 2]  
    pop ds:[2]  
  
    cli  
    mov word ptr es:[9 * 4],offset int9  
    mov es:[9 * 4 + 2],cs  
    sti  
  
    mov ax,0b800h  
    mov es,ax  
  
    ret
```

```
restore:  
    mov ax, 0  
    mov es, ax  
  
    mov ax, data  
    mov ds, ax  
  
    push ds:[0]  
    pop es:[9 * 4]  
    push ds:[2]  
    pop es:[9 * 4 + 2]  
  
    ret
```

(a) 根据代码解释 init 子程序做了哪些事情。

1. 将原来的 9 号中断例程地址存储到 data 数据段中
2. 将中断向量表中的 9 号例程的对应地址设置为 int9 的地址
3. 将 es 段地址指向屏幕缓存

(b) 根据代码解释 restore 子程序做了哪些事情。

1. 将中断向量表还原
2. 将地址指向存储在 data 数据段中的原来的 9 号中断例程的地址

(c) 指令 cli 和 sti 的作用分别是什么？为什么要在这里使用这两个指令？

cli 指令的作用是禁止中断发生，在 cli 起效之后，所有外部中断都被屏蔽，这样可以保证当前运行的代码不被打断，起到保护代码运行的作用。

sti 指令的作用是允许中断发生，在 sti 起效之后，所有外部中断都被恢复，这样可以打破被保护代码的运行，允许硬件中断转而处理中断的作用

使用这两个指令的目的：

在修改中断向量表的过程中起到保护作用。

如果在修改中断向量表的过程中发生了中断，则会导致程序跳转到错误的地址，造成系统异常。

3. (12 points) 根据程序功能，阅读程序，并在程序后的相应字母标号后填写适当的汇编语句，使整个程序能够完成指定的功能。对填写每条指令，要说明填写的理由。

```
; 名称: sub256
; 功能: 两个 256 位数据做减法, 这里的 256 是二进制位, 不是十进制位
; 参数: es:di 指向存储第一个数的内存单元, es:si 指向存储第二个数的内存单元, 采用"小端法"存放
; 结果: 运算结果存储在第一个数的存储空间中

sub256:
    _____(a)_____ ; 保存用到的寄存器
    push cx
    push si
    push di

    sub ax,ax ; CF = 0

    _____(b)_____ ; 循环 256/16=16 次
s:    mov ax,es:[di] ; 被减数
    sbb ax,es:[si] ; 减数
    _____(c)_____ ; 结果
    inc di ; 移动到下一个字
    _____(d)_____ ; inc 指令不影响 PSW
    _____(e)_____
    inc si
    loop s

    _____(f)_____ ; 恢复寄存器的值
    _____(g)_____
    _____(h)_____
    pop ax
    ret
```

- (a) 汇编语句: push ax； 理由是: 程序中使用到了 ax 寄存器，且结尾处 pop 了 ax。
- (b) 汇编语句: mov cx,16； 理由是: 循环次数为 16，故设置 cx 为 16
- (c) 汇编语句: mov es:[di],ax； 理由是: 保存运算结果到第一个数的存储空间相应的地址。
- (d) 汇编语句: inc di； 理由是: 每次运算 16 位，单次 inc 只会增加 8 位，因此需要移动两个字。

(e) 汇编语句: inc si ; 理由是: 每次运算 16 位, 单次 inc 只会增加 8 位, 因此需要移动两个字。

(f) 汇编语句: pop di ; 理由是: 最后一个压栈的是 di, 按照压栈顺序的逆序恢复寄存器。

(g) 汇编语句: pop si ; 理由是: 倒二压栈的是 si, 按照压栈顺序的逆序恢复寄存器。

(h) 汇编语句: pop cx; 理由是: 倒三压栈的是 cx, 按照压栈顺序的逆序恢复寄存器。

4. (14 points) 分析程序可能的输出情况, 并给出你的理由。

(1) 程序如下:

```

1  #include <stdio.h>
2  #include <signal.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <sys/types.h>
6
7  void catchSignal(int signalNumber) {
8      printf("catch!\n");
9      fflush(stdout);
10     signal(signalNumber, catchSignal);
11 }
12
13 > int main() {
14     pid_t pid;
15     pid = fork();
16     if(pid == 0) {
17         printf("send SIGQUIT\n");
18         fflush(stdout);
19         kill(getppid(), SIGQUIT);
20     } else if (pid > 0) {
21         signal(SIGQUIT, catchSignal);
22         wait(0);
23     }
24     return 0;
25 }
```

请写出上面程序可能的输出情况, 并说明理由:

输出情况
send SIGQUIT
catch!

fork 创建两个进程:

在子进程中, 输出 send SIGQUIT, 清空缓冲区后向父进程发送 SIGQUIT 的信号。

在父进程中, wait(0)等待所有进程结束后才能继续进行, 接受到子进程发出的信号后, 通过 catchSignal 函数处理。catchSignal 函数随后输出 catch!, 并设置自己为信号的处理函数。因为后续不再有信号传递过来, 因此不会再调用 catchSignal 函数, 程序结束。

(2) 程序如下：

```

1 #include <stdio.h>
2 #include <signal.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <sys/types.h>
7
8 void catchSignal(int signalNumber) {
9     printf("catch!\n");
10    fflush(stdout);
11    signal(signalNumber, catchSignal);
12 }
13
14 int main() {
15     int fd[2];
16     while (pipe(fd) == -1);
17     pid_t pid;
18     pid = fork();
19     if(pid == 0) {
20         close(fd[1]);
21         char buf[4] = {0};
22         read(fd[0], buf, sizeof(buf));
23         printf("send SIGBUS\n");
24         fflush(stdout);
25         kill(getppid(), SIGQUIT);
26         close(fd[0]);
27     } else if (pid > 0) {
28         close(fd[0]);
29         signal(SIGQUIT, catchSignal);
30         fflush(stdout);
31         char* ok = "ok";
32         write(fd[1], ok, strlen(ok) + 1);
33         close(fd[1]);
34         wait(0);
35     }
36 }
37 }
```

请写出上面程序可能的输出情况，并说明理由：

输出情况
send SIGBUS
catch!

理由：

fork 创建两个进程。

Pipe 默认阻塞，子进程 read 时，管道中为空，故阻塞。

当父进程写入 ok，子进程读到信号。

因为父进程此时 wait 所有进程，故子进程先结束，其输出 send SIGBUS 后，发送 SIGQUIT 给父进程，父进程调用 catchSignal 函数时，再输出 catch!。

5. (16 points) 下面这个程序呈现的是一个典型的生产者和消费者问题。若干个生产者和消费者线程共同使用同一个有限容量数组作为缓冲区，完成了生产和消费大量产品的任务。

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <stdlib.h>
4
5 /**
6  * 存放产物的容器的容量
7 */
8 #define SIZE 3
9
10 /**
11  * 所生产的数字的最大值
12  * 大于该值后，所有线程将停止生产和消费，准备退出
13 */
14 #define MAX 100
15
16 /**
17  * 程序中使用的生产者线程的个数
18 */
19 #define PRODUCE_THREAD_NUM 5
20
21 /**
22  * 程序中使用的消费者线程的个数
23 */
24 #define CONSUME_THREAD_NUM 5
25
26 /**
27  * 用来盛放生产产物的数据结构
28 */
29 struct Container {
30     // 数组，存放具体的产物
31     int *data;
32
33     // 下一次存放产物所要使用的下标
34     int producePos;
35
36     // 下一次取产物所要使用的下标
37     int consumePos;
38
39     // 容器中现有的产品数量
40     int productNumber;
41
42     // 容器的容量
43     int size;
44 } container;
```

```
47  /** =====
48  * 从这里开始若干个函数都是为操作container中的变量准备的
49  * ===== */
50
51  /**
52  * 本函数负责增长数组的下标,
53  * 正常情况下, 直接把下标+1;
54  * 如果已到达数组末尾, 那么把下标置零
55  *
56  * @param size container的最大容量
57  * @param i    需要增长的下标
58  * @return     增长后的下标
59  */
60  int indexGrow(int size, int i) {
61      return ++i >= size ? 0 : i;
62  }
63
64  /**
65  * 向容器中添加一个产品 (即一个整数)
66  *
67  * @param item 产品的数值
68  */
69  void put(int item) {
70      container.data[container.producePos] = item;
71      container.producePos = indexGrow(container.size, container.producePos);
72      container.productNumber++;
73      printf("produce: %d\n", item);
74      fflush(stdout);
75  }
76
77  /**
78  * 从容器中取出一个产品
79  *
80  * @return 返回该产品的数值 (即一个整数)
81  */
82  int take() {
83      int result = container.data[container.consumePos];
84      container.consumePos = indexGrow(container.size, container.consumePos);
85      container.productNumber--;
86      printf("consume: %d\n", result);
87      fflush(stdout);
88      return result;
89  }
90
91  /**
92  * 判断此时消费者线程能不能继续消费
93  *
94  * @return 如果能, 返回真; 否则返回假
95  */
96  int canConsume() {
97      return container.productNumber >= 1;
98  }
99
100 /**
101 * 判断此时生产者线程能不能继续生产
102 *
103 * @return 如果能, 返回真; 否则返回假
104 */
105 int canProduce() {
106     return container.productNumber < container.size;
107 }
108
109 /**
110 * 关于container的内容到此结束
111 * =====
```

```
113 	/**  
114 	* 下面两个数值分别记录当前是否要停止生产、停止消费  
115 	* 如果值为0，那么不应该停止；否则，应该停止  
116 	*/  
117 	int produceStop, consumeStop;  
118  
119 	/**  
120 	* 当前产品的数值  
121 	*/  
122 	int mCount;  
123  
124 	pthread_mutex_t mutex;  
125  
126 	pthread_cond_t canConsumeCond, canProduceCond;  
127  
128 	/**  
129 	* 执行初始化操作，如初始化容器，初始化锁和条件变量等  
130 	*/  
131 	void init() {  
132     container.size = SIZE;  
133     container.data = (int *) malloc( size: container.size * sizeof(int));  
134     container.consumePos = container.producePos = container.productNumber = 0;  
135  
136     pthread_mutex_init(&mutex, 0);  
137     pthread_cond_init(&canConsumeCond, 0);  
138     pthread_cond_init(&canProduceCond, 0);  
139     produceStop = consumeStop = 0;  
140     mCount = 0;  
141 }  
142  
143 void *product() {  
144     while (1) {  
145         pthread_mutex_lock(&mutex);  
146         while (!canProduce() && !produceStop) {  
147             pthread_cond_wait(&canProduceCond, &mutex);  
148         }  
149         if (produceStop) {  
150             pthread_mutex_unlock(&mutex);  
151             break;  
152         }  
153         int item = ++mCount;  
154         put(item);  
155         if (item >= MAX) {  
156             produceStop = 1;  
157         }  
158         pthread_cond_signal(&canConsumeCond);  
159         pthread_mutex_unlock(&mutex);  
160     }  
161     return NULL;  
162 }  
163  
164 void *consume() {  
165     while (1) {  
166         pthread_mutex_lock(&mutex);  
167         while (!canConsume() && !consumeStop) {  
168             pthread_cond_wait(&canConsumeCond, &mutex);  
169         }  
170         if (consumeStop) {  
171             pthread_mutex_unlock(&mutex);  
172             break;  
173         }  
174         int item = take();  
175         if (item >= MAX) {  
176             consumeStop = 1;  
177         }  
178         pthread_cond_signal(&canConsumeCond);  
179         pthread_mutex_unlock(&mutex);  
180     }  
181     return NULL;  
182 }
```

```

184     void closeResources() {
185         pthread_mutex_destroy(&mutex);
186         pthread_cond_destroy(&canConsumeCond);
187         pthread_cond_destroy(&canProduceCond);
188     }
189
190
191 > int main() {
192     init();
193     pthread_t produceThread[PRODUCE_THREAD_NUM];
194     pthread_t consumeThread[CONSUME_THREAD_NUM];
195     for (int i = 0; i < PRODUCE_THREAD_NUM; i++) {
196         pthread_create(produceThread + i, 0, product, 0);
197     }
198     for (int i = 0; i < CONSUME_THREAD_NUM; i++) {
199         pthread_create(consumeThread + i, 0, consume, 0);
200     }
201
202     for (int i = 0; i < PRODUCE_THREAD_NUM; i++) {
203         pthread_join(produceThread[i], 0);
204     }
205     for (int i = 0; i < CONSUME_THREAD_NUM; i++) {
206         pthread_join(consumeThread[i], 0);
207     }
208     closeResources();
209     return 0;
210 }
```

(1) 请根据代码详细解释 143 行开始的 product() 函数的功能。并说明 146 行的代码中的“while”可以改为“if”吗，给出你的理由。

product()函数的功能：

1. 加锁
2. 判断能否生产并且是否已停止生产，若不能生产且没有停止，则进入 while 循环，阻塞等待条件变量并解锁。
3. 若不满足条件：若停止生产，则解锁
4. 若容量未满，则继续生产
5. 判断 item 是否 ≥ 100 ，若 ≥ 100 ，则停止生产
6. 唤醒等待在条件变量上的一个线程并解锁

不可以改为“if”。

原因：

1. 使用 pthread_cond_wait() 会使线程阻塞，因为线程在能否生产的条件下等待，当条件满足后仍需重新绑定锁。
2. 如果使用 if，不会再次检查是否满足条件，导致线程可能被错误唤醒。

(2) 请根据代码详细解释 164 行开始的 consume() 函数的功能。并说明 167 行的代码中的“while”可以改为“if”吗，给出你的理由。

consume()函数的功能：

1. 加锁
2. 判断是否不能消费并且没有停止消费，若不能消费且没有停止消费，则进入 while 循环，阻塞等待条件变量并解锁
3. 若不满足条件：若已停止消费，则解锁
4. 若仍有物品可以消费，则继续消费
5. 判断 item 是否 ≥ 100 ，若 ≥ 100 ，则停止消费
6. 唤醒等待在条件变量上的一个线程并解锁。

不可以改为“if”。

原因：

若改成 if，则只会判断一次，若此时产品容量为 0，消费者线程此时满足 if 条件，即产品剩余量为 0，且产品消费总数未到 100。

此时虽然线程释放了共享锁，但仍会继续执行消费产品，导致线程同步异常。

(3) 程序最终可以正常退出吗，如果不能，将会产生什么问题。请指出是哪一行代码导致了该问题，并给出修改方案。

不能正常退出。

生产线程都被挂起，使得其无法被激活生产，造成卡死。

第 178 行改成：pthread_cond_signal(&canProduceCond);