**深 圳 大 学**

**实 验 报 告**

**课程名称： 操作系统(1500110002)**

**实验序号： 实验1**

**实验名称： 设计简单的Shell程序**

**指导教师： 周明洋**

**班级： 高性能特色班 姓名： 何泽锋 学号： 2022150221**

**实验时间： 2025年 4 月 10 日——2025年5月5日**

**实验报告提交时间： 2025年 5 月 8 日**

# 一、实验目的

综合利用进程控制的相关知识，结合对shell功能的和进程间通信手段的认知，编写简易shell程序，加深操作系统的进程控制和shell接口的认识。

# 二、实验环境

1.硬件环境：PC机

2.软件环境：Linux系统或Windows 7中文版。

# 三、实验内容

1.全面实践进程控制、进程间通信的手段；

2.编写简易shell程序。

# 四、实验要求

1. 操作部分：

|  |
| --- |
| （1）学习 BlackBoard中的“综合1预备1-进程控制”，完成材料中的全部操作，并截屏记录（实验中的全部操作都需要截屏记录，并配有必要的文字说明或结果的解读）。（30分）  （2）学习 BlackBoard中的“综合1预备2-进程间通信与同步”，完成材料中的全部操作，并截屏记录（实验中的全部操作都需要截屏记录，并配有必要的文字说明或结果的解读）。（40分） |

以上操作部分的记录请放到报告的末尾。

2. 编程部分：

|  |
| --- |
| （1）尝试自行设计一个C语言小程序，完成最基本的shell角色：给出命令行提示符、能够逐次接受命令；对于命令分成三种，内部命令（例如help命令、exit命令等）、外部命令（常见的ls、cp等，以及其他磁盘上的可执行程序HelloWrold等）以及无效命令（不是上述三种命令）。（20分）  （2）参考“综合1预备2-进程间通信与同步”中的4.1.1小节内容将上述shell进行扩展，使得你编写的shell程序具有支持管道的功能，也就是说你的shell中输入“dir || more”能够执行dir命令并将其输出通过管道将其输入传送给more作为标准输入。（10分）  （3）可以将1/2直接合并完成。  （4）设计标准的参考。  ①提示符最低标准是固定字符串。提升标准是使用含当前路径的信息为提示符。  ②接受命令的最低标准是一次接受一个命令就推出shell程序，提升标准是在shell内部循环读取和执行命令。  ③如果实现输出重定向或输入重定向可以最多加10分，加分上限满分100。 |

# 五、实验步骤

**1.编程部分**

（1）实现最基本Shell功能

|  |
| --- |
| 功能一：命令行提示符（含当前路径）     * 代码解释：使用 getcwd() 获取当前工作目录，打印格式为 [当前路径] mysh>，若获取失败则打印默认提示符 mysh> * 运行结果：可以看到输出了当前目录以及提示符     功能二：循环读取命令     * 代码解释：在main函数中使用while(1)不断读取用户输入，使用fgets()读取一行命令，并检测是否为空命令 * 运行结果：可以看到可以连续输入命令     功能三：内部命令处理  实现了三个内部命令：help、exit、cd       * 代码解释：help函数直接打印帮助信息；exit指令会直接调用exit()结束程序，cd指令会调用chdir()更改当前目录。main函数会根据输入进行判断并调用对应的内部命令 * 运行结果：     功能四：外部命令     * 代码解释：父进程使用fork()创建子进程，子进程中使用execvp()执行外部命令，父进程使用waitpid()等待子进程结束，父进程进行对应的错误处理例如命令不存在、执行失败等。 * 运行结果：     功能五：无效命令     * 代码解释：在解析命令失败时（如命令语法错误、参数为空）返回错误信息 * 运行结果： |

（2）支持管道功能

|  |
| --- |
| ① parse\_command()将命令按“ | ”分割成多个子命令    ② execute\_piped\_commands() 使用 pipe() 创建多个管道，每个命令在单独子进程中执行，通过 dup2() 重定向标准输入输出，最终通过 wait() 等待所有子进程完成。    运行结果： |

（3）支持输入/输出重定向

|  |
| --- |
| 主循环中检测是否有“<”或“>”，若有，则记录文件名，并将该位置设为NULL，在执行命令前打开文件并使用dup2()替换标准输入/输出，支持追加写入和覆盖写入    运行结果： |

**2.进程控制**

2.1进程控制

（1）创建HelloWorld测试文件

|  |
| --- |
| 代码部分如下所示：    通过“gcc -o HelloWorld HelloWorld.c”命令编译为可执行文件，然后通过“./HelloWorld”运行测试文件，得到如下输出结果：    通过file命令检查输出的可执行文件，输出结果如下所示，表明这是x86-64平台上ELF格式的可执行文件    ELF文件包含许多信息，操作系统可以根据这些信息创建进程影响，将代码拷贝到新进程虚拟空间的低地址，将可执行文件拷贝到较高的位置，再高端位置创建用户堆栈。 |

（2）进程的基本概念

|  |
| --- |
| 编写HelloWorld-getchar.c文件，代码如下所示，编写完成后使用gcc进行编译    将HelloWorld-getchar文件运行三次（在三个终端），然后在第四个终端用“ps -A | grep HelloWorld”命令查看进程，结果如下图所示，可以看到此时有3个HelloWorld-getchar进程，进程号分别为4200、4307、4324，进程分别运行在伪终端pts/3、pts/4、pts/5。其中进程编号指的是教材中的PCB，在Linux称为PID    使用“ps -aux | grep HelloWorld”查看完整进程信息，可以看到进程的运行情况，当前为S+，表示阻塞状态。可以看到当前输出了4个与HelloWorld的进程，这是因为当前使用ps查看的端口也与其有关 |

（3）进程间组织关系

|  |
| --- |
| Linux 系统中所有进程通过进程控制块 PCB形成多种组织关系。  ①根据进程创建过程可以有**亲缘关系**，通过进程间的父子关系组织成一个进程树；  ②根据用户登录活动有**会话、进程组**等关系；  ③根据调度策略和进程状态则会归入到**不同队列**中，将实时进程组织成位图队列，而将普通进程组织成红黑树，进程阻塞时还会被挂入事件的等待队列中。  **·亲缘关系**  亲缘关系包括：父子关系、兄弟关系，可以用如下的图进行表示    通过pstree命令查看全系统的进程树结构。    查看前面运行的三个HelloWorld-getchar进程    除此之外，进程的父子关系还可以通过后运行“top | more&”和“ps j |more”命令显示，改过程将在下方演示  **·会话、进程组、线程组及控制终端**  “会话/进程组/线程组”几个概念呈现层级关系，Linux 系统中可以有多个**会话（session）**，每个会话可以包含多个**进程组（process group）**，每个进程组可以包含多个进程，每个进程构成一个**线程组**——一个线程组由一个进程内的一个或**多个线程组成**  通过“top |more &”创建 top 进程和 more 进程，并且在后台运行。然后通过“ps j |more”命令创建 ps 进程和第二个 more 进程，其中 ps 的参数 j 表示用任务格式来显示进程。可以看到此会话的所有进程的 SID 都为3855（会话领头进程 bash 的 PID）。 |

（4）进程控制命令

|  |
| --- |
| **·创建进程的命令**  编辑一个 shell 脚本，用于创建进程，含义为第一行用于指出该脚本需要用/usr/bin/bash 来解释执行，第二行 shell 内部命令 echo 用来显示，第三行 HelloWorld 是外部命令    用“chmod a+x shell-script”命令为它增加执行权限，再用 ls –l 检查是否有“x”执行权限    在终端输入文件名即可运行脚本程序    **·撤销进程的命令**  命令行上撤销进程主要通过kill 命令来完成。由于 kill 命令的输入需要进程号，通常先用ps1命令查看到进程的进程号 PID，然后再用“kill -PID”命令杀死 PID 指定进程  举例来说: 运行得 HelloWorld-getchar，先通过“ps -ef |grep HelloWorld-getchar”查到进程号为7153，然后通过“kill 7153”即可以杀死这个进程。如下图所示展示了“查询——kill——查询”的过程，可以看到进程被杀死 |

2.2创建进程与撤销

（1）fork()创建子进程

|  |
| --- |
| ·fork函数介绍：fork 函数被成功调用后将安装父进程的样子复制出一个完全相同的子进程。父进程 fork()函数结束时的返回值是子进程的 PID 编号。新创建的子进程则也是从 fork()返回处开始执行，但不同的是它获得的返回值是 0。也就是说父子进程执行上几乎相同，唯一区别是子进程中返回 0 值而父进程中返回子进程 PID。如果 fork()出错则只有父进程获得返回值-1 而子进程未能创建。子进程是父进程的副本，它将获得父进程数据空间、堆、栈等资源的相同副本。但父子进程间不共享这些存储空间，它们之间共享的存储空间只有代码段。  ·创建代码验证fork()：  此处代码的意思是创建一个子进程，如果返回-1说明创建失败，如果创建成功，父进程进入else部分，子进程进入else if部分，分别输出对应语句。    编译运行结果如下，此时两个进程都为退出，getchar会等待终端的字符输入。    然后用 ps 命令和 pstree 命令查看其父子信息，如下图为ps命令查询的结果，可以看到fork-demo运行了两个进程，进程号分别是7552和 7553，其中 7552 进程是 7553 的父进程。    调用“pstree 7552”指令，可以看到fork-demo创建了一个fork-demo    修改fork-demo函数，修改部分如下图所示，简单分析运行结果，首先与原始的程序一致，父进程P0会创建子进程P1，然后分别输出对应语句，此时子进程P1创建子进程P2，父进程P0创建子进程P3（P3、P4顺序不一定一致，取决于P0、P1运行速度），新的两个进程会从fork语句开始运行，进入getchar等待字符输入，因此在后台能看到4个进程，进程树的深度应该为3    首先使用ps指令查看，与分析结果一致，一共4个进程，其中父进程P0为8004，P1为哦8005，P3、P4分别为8006、8007，PGID代表是同一个进程组    使用pstree进一步查看进程创建关系，与分析一致，进程树深度为3 |

（2）孤儿进程和僵尸进程

|  |
| --- |
| ·孤儿进程  定义：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被 init 进程（进程号为 1，不一定为1，取决于操作系统）所收养，并由 init 进程对它们完成状态收集工作  ·创建孤儿进程验证代码：  ①首先运行原始的fork\_demo此时创建了两个线程，在另一个终端使用“ps j”命令查看，如图所示存在两个进程，其中8468为父进程，8469为子进程    ②在运行进程的终端中输入一次回车，然后再查看“ps j”，可以看到此时只剩下8469进程，其PPID（父进程PID）变为1477，该PID又操作系统分配，被init进程收养。    ③再按一次回车，此时可以看到子进程页运行结束，在ps中已经看不到该进程 |

|  |
| --- |
| ·僵尸进程  定义：一个子进程在其父进程还没有调用 wait()或 waitpid()的情况下退出。这个子进程就是僵尸进程。  ·创建僵尸进程运行代码    运行后再另一个终端使用ps查看进程状态，可以看到子进程处于“Z+”状态（表明是僵尸状态，<defunct>），此时父进程状态为“S+”，表示处于阻塞（睡眠）状态。    当父进程sleep结束，将执行 wait()处理字进程的遗留数据对象，子进程将从僵尸状态转为完全消失状态。 |

（3）exec 函数族

|  |
| --- |
| exec函数可以用于创建进程用于执行不同的任务，不同于fork，子进程与父进程是相同的。  ·编写测试代码：    execve()函数的第一个参数就是想要变成的“新”进程影像（可执行文件），第二个参数是命令行参数，第三个变量是环境变量。这里可以看到字进程通过 execve()将自己变身为“/usr/bin/ls”，因此不再执行与父进程里的代码——即后面的“printf("this printf()will not be exec,because …”是没有机会得到执行的。真是的运行结果如所示，可以看到字进程变成了 ls。 |

（4）kill()撤销进程

|  |
| --- |
| 通过 kill 命令通过发送信号来撤销进程，在程序中可以调用kill()函数。函数原型为“int kill(pid\_t pid, int sig);”，第一个参数是目标进程的 PID，第二个参数是发送的信号。另有 raise()函数用于自己发送信号，相当于 kill 给自己发信号 kill(getpid(), SIGNAL)  ·编写程序验证：    如图所示，编写了一个程序并调用kill()函数结束自身，也可以用raise()来替代 |

2.3.创建pthread线程

（1）进程与线程

|  |
| --- |
| 如下图所示，表示了多进程、多线程并发的示意图，注意其中黄色的内核空间是相同、共用的（虽然每个进程都绘制了一个内核空间）    每个进程或线程在内核空间中都会有自己的进程控制块 task\_struct，但是只有进程才会有独立的内存空间。创建进程和线程的开销并不相同。无论是进程还是线程，都必须有进程控制块 PCB（struct task\_struct），但是线程间共享进程空间因此不需要独立的内存描述符（struct mm\_struct）以及所辖的内存区域描述符等，也不需要独立的文件描述符或其他资源。 |

（2）创建方法

|  |
| --- |
| 使用 pthread 库来创建线程，代码如下所示：    使用“gcc pthread-demo.c -o pthread-demo -lpthread”完成编译并输出 pthread-demo 可执行文件。运行 pthread-demo 将看到两个线程都在执行，各自输出一行信息，等待 10 秒后两个线程都将自动结束    再次运行，并使用另一个终端观察线程运行情况，输入“ps -aLf”查看信息，可以看到NLWP列显示2，表示统一进程的线程数。 |

2.4.进程/线程资源开销

|  |
| --- |
| 更直观的对比进程和线程的开销，编写两个程序 fork-100-demo.c和pthread-100-demo.c，分别创建100个进程和100个线程。然后用Linux的/proc/slabinfo中给出的信息来简单比较各自在进程控制块和内存描述符上的资源开销。  ·fork-100-demo.c    ·pthread-100-demo.c    分别运行两个程序，比较在进程控制块中的开销  ·进程：  在运行程序前使用“cat /proc/slabinfo |grep task\_struct”命令开销    运行后再次查看，对比前后控制块数量，可以看到运行前为860，运行后为993，多了133个数据对象    结束程序后，数量减少，回到866个    ·线程  同理在运行前中后分别检测pthtead程序的线程资源数量，得到如下图，可以看到也是增加了约100个，含有一定的开销 |

2.5内存描述符开销

|  |
| --- |
| 分别运行两个程序，用“cat /proc/slabinfo |grep mm\_struct”命令来检查运行前后的mm\_struct数量  ·进程  可以看到mm\_struct的数量增加了    ·线程  运行前后没有变化，这是因为线程是用共享进程空间，并没有增加一个 mm\_struct    用“cat /proc/slabinfo |grep vm\_area\_struct”观察vm\_area\_struct数量变化  ·进程  观察 fork-100-demo，可以发现新增大约 15xx 左右的使用量    ·线程  发现只需要新增大约 1xx 左右的使用量    从实验结果可以看出，线程的开销确实比进程要小 |

**3.进程间通信与同步**

3.1进程间通信

传统概念的进程通信包含：管道、消息队列、共享内存等机制

3.1.1管道

|  |
| --- |
| **无名管道**  举例如下：  ①调用两次 fork()分别创建两个子进程（后面会替换成 ls 和 more 两个进  程），此时两个子进程继承了 shell 的所打开的文件资源（含文件描述符 3 和 4），最后 shell 进程用 close()关闭文件并释放描述符 3 和 4    ②子进程 2 通过 dup2(3,0)将文件描述符 3 拷贝到文件描述符 0（标准输入文件），  然后用 close()关闭文件 3、4 并释放相应的描述符，最后执行 execve()系统调用来执行 more 程序，于是 more 从标准输入（对应文件描述符 0）读取的数据实际上就是管道的读端提供的数据——即 ls 写入的数据。具体细节如图 4-2 所示，此时管道起到连接两个进程输入输出的作用。    代码测试：  （1）pipe-demo.c文件，pipe()将通过两个文件描述符（整数）来指代管道缓冲区的读端和写端    （2）运行其表明父进程发送了消息到管道，自进程成功接受到了“Message from parent”。 |

|  |
| --- |
| 命名管道  无名管道有一个主要缺点，只能通过父子进程之间（及其后代）使用文件描述符的继承来访问，无法在任意的进程之间使用。命名管道（named pipe）或者叫 FIFO 则突破这个限制。可以说 FIFO 就是无名管道的升级版——有可访问的磁盘索引节点，即 FIFO 文件将出现在目录树中  （1）创建命名管道    （2）使用cat os-exp-fifo尝试读入数据，此时管道没有任何数据。进入阻塞状态    （3）在另一个终端中，使用“echo Hello, Named PIPE! > os-exp-fifo”往管道内写入数据，观察终端，可以看到通过cat读取到了数据，实现了字符串的回显 |

3.1.2 System V IPC

|  |
| --- |
| System V IPC的对象：  ①信号量，用来管理对共享资源的访问  ②共享内存，用来高效地实现进程间的数据共享  ③消息队列，用来实现进程间数据的传递。  （1）查看系统中System V IPC对象，使用“ipcs”命令，可以看到此时含有三段共享内存    （2）显示消息队列的限制条件，使用“ipcs -ql”，展示如下，可以看到消息的最大长度为8192，队列的默认最大值为16384 |

|  |
| --- |
| 消息队列  （1）编写msgtool.c文件    （2）执行“msgtool s 1 Hello,my\_msg\_queue!”以发送类型为 1 的消息，可以看到此时发送成功。    再开启另一个终端用“ipcs -q”查看新创建的消息队列，ID是0x6d05233a，含有20字节的信息。    再执行“msgtool r 1”，读取类型为1的消息    使用“ipcs -q”查看此时的消息队列，可以看到已经为空 |

|  |
| --- |
| 共享内存  （1）创建测试代码    （2）运行输出，输出结果表明新创建的共享内存的 ID 为32787，长度为 4096 字节，当前还没有进程将他映射到自己的进程空间（nattch 列为 0）    （3）创建另一个进程，通过影射该共享内存而使用该共享内存，代码如下：    （4）运行程序，注意需要输入前面创建的共享内存ID，    （5）使用“ps -a”命令获取shmatt-write-de的进程PID，可以看到当前为18586    （6）使用cat指令观察该进程空间，如下图所示    （7）在运行程序的终端输入回车，此时将解除共享内存映射，使用“ipcs - m”指令可以看到共享内存区没有使用，nattch列为0    （8）此时尝试用另一个程序取读取共享内存，其代码如下所示：    （9）虽然创建该共享内存的进程已经结束了，可是 shmatt-read-demo 映射 ID 为的共享内存后，仍读出了原来写入的字符串 |

3.2进程间信号同步

3.2.1 System V IPC

该部分上方已经介绍

3.2.2 POSXI信号量

|  |
| --- |
| 有名信号量  有名信号量由于可以通过标识来访问，因此可以同时用于进程间同步和线程间同步。  （1）创建验证代码，使用“gcc psem-named-open.c –o psem-named-open –lpthread”命令进行编译    （2）运行psem-named-open。如果没有输入作为标识的文件名字符串，则给出体系要求用户输入；如果输入一个文件名字符串，正常情况将完成创建过程    （3）创建另一份代码来验证P/V操作中的V操作，即对信号量进行减1操作，可能引发阻塞，代码如下    （4）编译运行，输入前面创建信号量时使用的文件名标识HelloWorld.c，此时打印出当前信号量值为 0    （5）再次运行出现阻塞    （6）使用ps命令查看当前运行的程序，可以发现进程处于S（阻塞状态）    （7）创建进程进行P操作，使得前面的进程从阻塞状态唤醒并执行。代码如下：    （8）编译运行，可以看到信号量的值为1，原来阻塞的的 psem-named-post-demo 被唤醒并执行完毕      （9）创建撤销信号量的程序 |

|  |
| --- |
| 无名控制量  无名信号量适用于线程间通信，如果无名信号量要用于进程间同步，信号量要放在共享内存中，无名信号量和有名信号量的区别主要在创建上，无名信号量使用 sem\_init()创建 |

|  |
| --- |
| 互斥量  （1）先用一个代码来展示多线程并发且没有用互斥量保护共享变量的情形    （2）编译运行，可以看到因为没有互斥量的控制，共享变量的操作结果是不确定的，因此运行结果也不一致    （3）修改代码，添加互斥量    （4）再次编译运行，可以看到多次运行结果一致 |

# 六、心得体会

**1.操作部分**

在本次实验的操作部分中，深入学习并实践了进程控制与进程间通信的基本概念和技术。通过完成“综合1预备1-进程控制”和“综合1预备2-进程间通信与同步”的全部操作，我对操作系统中的关键机制有了更深刻的理解。

（1）进程控制

· 创建HelloWorld测试文件：通过编译和运行简单的C程序，我熟悉了从编写源代码到生成可执行文件的全过程，并理解了ELF格式可执行文件的基本结构。

* 进程的基本概念：通过多次运行相同的程序，并使用ps命令查看进程状态，我掌握了如何查看进程信息及理解不同状态（如阻塞状态）的意义。
* 进程间组织关系：利用pstree命令，我观察到了系统中进程树的结构，加深了对父子进程关系的理解。同时，我还了解了会话、进程组等高级概念。

· 进程控制命令：通过编写shell脚本和撤销进程的命令，我学会了如何管理进程的生命周期，包括创建、查看、终止进程等操作。

（2）进程间通信

· 管道：通过无名管道和命名管道的示例，我理解了进程间如何通过管道进行数据传递，以及它们各自的适用场景。

* System V IPC：学习了信号量、共享内存和消息队列的使用方法，这让我明白了不同的进程间通信机制各有优劣，适用于不同的应用场景。
* POSIX信号量：通过有名信号量和无名信号量的实例，我掌握了信号量在进程同步中的作用。

**2.编程部分**

| **功能要求** | **是否完成** |
| --- | --- |
| 提供命令行提示符（含当前路径） | √ |
| 支持循环读取命令 | √ |
| 内部命令（help, exit, cd） | √ |
| 外部命令执行 | √ |
| 无效命令处理 | √ |
| 支持管道功能 | √ |
| 支持输入/输出重定向 | √ |

|  |
| --- |
| 指导教师批阅意见：  成绩评定：  指导教师签字：  2025年 月 日 |
| 备注： |

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后15工作日内。