**深 圳 大 学 实 验 报 告**

**课 程 名 称： 操作系统**

**实验项目名称： 实验一 并发程序设计**

**学 院： 计算机与软件学院**

**专 业： 计算机科学与技术**

**指 导 教 师： 阮元**

**报告人： 刘睿辰 学号： 2018152051 班级： 数计班**

**实 验 时 间： 2021.3.10-2021.4.3**

**实验报告提交时间： 2021.4.3**

**教务处制**

|  |
| --- |
| **一、实验目的与要求：**  通过进程的创建、撤销和运行加深对进程概念和进程并发执行的理解，明确进程与程序之间的区别。 |
| **二、方法、步骤：**   1. 掌握在在linux中编程编译运行的方法，试验你的第一个helloworld程序； 2. 学习预备材料和后面的阅读例程，理解函数fork()、execl()、exit()、getpid()和waitpid()的功能和用法； 3. 编写hello-loop.c程序（在helloworld例程基础上加一个死循环）。使用gcc hello-loop.c –o helloworld生成可执行文件hello-loop。并在同一个目录下，通过命令“./hello-loop”执行之。使用top和ps命令查看该进程，记录进程号以及进程状态； 4. 使用kill命令终止hello-loop进程； 5. 使用fork()创建子进程，形成以下父子关系：     通过检查进程的pid和ppid证明你成功创建相应的父子关系并用pstree验证其关系。注意在执行前后，分别检查/proc/slabinfo中进程控制块PCB的数量是否能反映出你创建的进程数量变化？   1. 编写代码实现孤儿进程，用pstree查看孤儿进程如何从原父进程位置转变为init进程所收养的过程；编写代码创建僵尸进程，用ps j查看其运行状态； 2. 编写一个代码，使得进程循环处于以下状态5秒钟运行5秒钟阻塞（例如可以使用sleep( )），并使用top或ps命令检测其运行和阻塞两种状态，并截图记录。 |
| **三、实验过程及内容：**   1. 在linux环境下运行helloworld程序，检测linux环境是否正常。   这里面我们需要知道linux环境下C代码的编译。通过计算机体系结构相关课程的学习，我们了解到我们需要首先建立helloworld.c文件，在这份文件中编写C代码然后进行编译。编译指令为    这里面我们通过该指令形成可执行文件helloworld，然后通过以下指令就可以执行。    所以我们首先在文本编辑器里编辑如图1所示的代码，然后gcc编译，最后运行结果如图2所示，linux环境正常。  图2. helloworld程序编译运行结果  图1. helloworld样例程序   1. 理解函数fork()、execl()、exit()、getpid()和waitpid()的功能和用法。 2. fork函数   函数定义：pid\_t pfid = fork();  函数解释：pid\_t类型是一个进程号类型，在头文件<sys/types.h>中定义。fork()函数会产生一个新的子进程，其子进程会复制父进程的数据与堆栈空间，并继承父进程的用户代码，组代码，环境变量、已打开的文件代码、工作目录和资源限制等。关于这个函数的返回值，如果fork()成功则在父进程会返回新建立的子进程代码(PID)，而在新建立的子进程中则返回0。如果fork 失败则直接返回-1，失败原因存于errno中。该过程如图3所示。    图3. fork()函数原理图   1. waitpid函数   函数定义：pid\_t waitpid(pid\_t pid,int \* status,int options);  函数调用：waitpid(pid, NULL, 0);  函数解释：waitpid()会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用waitpid()时子进程已经结束，则wait()会立即返回子进程结束状态值。子进程的结束状态值会由参数status返回，而子进程的进程识别码也会一快返回。如果不在意结束状态值，则参数status可以设成NULL。参数pid为欲等待的子进程识别码，其他数值意义如下  pid<-1 等待进程组识别码为pid绝对值的任何子进程；  pid=-1等待任何子进程，相当于wait()；  pid=0 等待进程组识别码与目前进程相同的任何子进程；  pid>0 等待任何子进程识别码为pid的子进程；  关于返回值，如果执行成功则返回子进程识别码(PID)，如果有错误发生则返回-1。失败原因存于errno。   1. getpid（取得进程识别码）   函数定义：pid\_t getpid(void)；  函数解释：getpid()用来取得目前进程的进程识别码，许多程序利用取到的此值来建立临时文件，以避免临时文件相同带来的问题。该函数返回目前进程的进程识别码。  这里面再介绍一个getppid函数，这个函数返回目前这个进程的父进程的进程号。有时候为了确定两个进程之间的关系可以这个函数和getpid结合使用。   1. exit（正常结束进程）   函数定义：void exit(int status)；  函数解释：exit()用来正常终结目前进程的执行，并把参数status返回给父进程，而进程所有的缓冲区数据会自动写回并关闭未关闭的文件。  这里，如果表示异常退出，则表示正常退出。   1. execl函数（执行文件）   函数定义：int execl(const char \* path, const char \* arg, ...)；  函数解释：execl()用来执行参数path字符串所代表的文件路径，接下来的参数代表执行该文件时传递过去的argv(0)、argv[1]……，最后一个参数必须用空指针(NULL)作结束。  关于返回值，如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno中。  调用ls命令范例： execl("/bin/ls", "/bin/ls", "-l" , "/etc", NULL)。   1. 编写hello\_loop.c程序（在helloworld例程基础上加一个死循环）。使用gcc hello\_loop.c –o helloworld生成可执行文件hello\_loop。并在同一个目录下，通过命令“./hello\_loop”执行之。使用top和ps命令查看该进程，记录进程号以及进程状态。   我们编写hello\_loop.c程序，如图4所示。  图4. hello\_loop.c程序代码  这是一个死循环，然后我们在linux命令行编译之后运行，发现程序不断输出helloworld这个语句。这时符合我们的设计的，但是我们需要看一下进程号以及进程状态。首先，我们用ps aux指令来看一下进程状态，如图5所示，可以发现进程号为3946，状态为R+，这个代表运行中。  图5. ps aux指令查看进程状态  然后我们再用top指令查看进程实时状态，动态地查看系统运维状态。简单来说，top指令就是linux系统的“任务管理器”。top指令的运行结果如图6所示。  图6. top指令查看进程状态  从图6中我们看到，hello\_loop进程的PID是3946，CPU占用率为26.2%。CPU占用率事实上是动态变化的。   1. 使用kill命令终止hello\_loop进程。如图7所示，我们输入进程号，然后杀死，结果如图8所示，进程已经被杀死。   图7. kill指令杀死进程    图8. kill指令杀死进程结果   1. 使用fork()创建子进程，形成不同的父子关系。    1. 构造一个父进程拥有100个子进程的关系（tree.c）。   一个父进程拥有100个子进程的进程关系树如图9所示。    图9. 进程关系1  根据进程树我们知道，我们需要进行100次循环，然后每次在进程号为0，也就是子进程的位置加输出信息，输出当前子进程的PID以及该子进程的父进程的PID。根据我们的设计，这100个子进程的父进程的PID都应该是相等的。代码如图10所示。  图10. 让父进程拥有100个子进程的代码  此外，当循环结束的时候我们添加一个死循环while(1)，这样保持进程不终止，我们有充裕的时间来查看进程树。  我们运行程序，得到结果图如图11所示。  图11. tree.c代码运行结果  显然，结果显示父进程PID都是相等的。为了检查数目，我们需要将进程树打印出来。我们使用pstree指令，外加参数-p可以输出进程号，然后输入父进程的PID。从图11中我们很明显看出父进程的PID就是5389。所以我们打印进程树的指令就是    我们打印出的进程树如图12所示。    图12. tree.c所创建的进程树（1）  我们发现进程树是正序排列的，那我们直接去看最下面的部分，如图13所示。    图13. tree.c所创建的进程树（2）  我们可以看出从5390到5489，一共100个子树，这也就创建成功了。  下面我们来看一下进程控制块PCB的数量的变化。在没运行程序的时候，我们查看PCB的数量如图14所示，为711个。当运行了该程序之后，我们发现进程控制块的数量变成了810个，增加了100个。  图14. tree.c导致的PCB数量的变化   * 1. 构造一个链条式的结构（list.c）   一个链条式的结构如图15所示。  在这种结构中，每个父进程都只有一个子进程。    图15. 链条式结构  当父进程生成了一个子进程之后，父进程不能再产生任何子进程，所以需要马上跳出循环。新产生的子进程则可以输出父子信息，这样方便我们确定从属关系。  注意，在循环的最后我们要有sleep语句，否则将会产生孤儿进程，这在后面会加以说明。  代码如图16所示。  图16. 链条式结构代码  代码运行结果如图17所示。  图17. list.c运行结果（1）  从图17中的运行结果中我们可以看出，第n条语句的父进程就是第n-1条语句的进程号，这也就显示出这是一个线性的关系。  我们观察到最后的语句，发现是3164，这个和第一个子进程3065，发现正好是100个子进程，如图18所示。  图18. list.c运行结果（2）  然后我们用pstree指令打印出进程树，如图19所示，显示这是一个线性的结构。  图19. list.c创建的进程树  然后我们检测一下PCB数量的变化。如图20所示，  图20. list.c导致的PCB数量变化  可以看到，原来的PCB数量为712，程序运行之后PCB数量为810，增加了98个。   * 1. 构造二叉树结构（tree2.c）   建立二叉树结构如图21所示。    图21. tree2.c生成的二叉树结构  我们通过分析二叉树的结构，我们发现如果要建立5层二叉树，则应该循环5次，每一次要经历两次fork()操作。但是我们不能连着进行两次fork()操作，我们来看一下原因。  如图22所示，如果两次fork()操作连续的话，第一次fork()会在第一个节点生成两个子树。第二次fork()操作会在两个节点分别生成子树。    图22. 连续两次fork()的结果  如果按照如图22所示的情况，一次循环两个节点分别形成子树，那么第二次循环之后，原来生成子树的2号节点将会有三个子树（第二次循环也会生成两个子树），这样的话就不符合设计要求。所以我们需要加一个判断条件，也就是仅仅当pid1不为0的时候才能添加子树。而当某节点的pid1和pid2都不为0的时候，此时不能再生成子树，所以需要退出循环。  我们的代码如图23所示。    图23. 生成进程二叉树的代码  接下来我们运行代码，如图24所示，我们观察到根节点应该是4323。然后我们从这个节点开始打印进程树，如图25所示。  图24. 生成进程二叉树的代码运行结果    图25. tree2.c所创建的进程树  图25右侧显示不全，我们以4324为根节点再次打印，结果如图26所示，这样我们就确定这就是一个有5层子进程的二叉树。  图26. tree2.c所创建的进程树（部分）  我们再来看一下PCB数量的变化。如图27所示，增加了60个PCB。我们知道，一共有5层子树，那么子进程数量为    原来有PCB数量为705，程序运行之后为765，增加了60个，这是符合预期的。    图27. tree2.c所导致的PCB数量变化   1. 编写代码实现孤儿进程以及僵尸进程，并查看进程状态。（orphan.c以及zombie.c）   如果一个正常运行的子进程，父进程退出了但是子进程还在，该进程此刻是孤儿进程，被init收养；一个正常运行的子进程，如果此刻子进程退出，父进程没有及时调用wait或waitpid收回子进程的系统资源，该进程就是僵尸进程，如果系统收回了，就是正常退出。  为了构造孤儿进程，我们就要让父进程退出而其子进程在其后退出。所以我们的设计是，让子进程睡眠时间长于父进程，这样就实现父进程先于子进程退出。代码如图28所示。  图28. orphan.c代码  我们运行代码，调用pstree指令查看进程树，在如图29所示的位置找到了orphan.c运行的位置。父进程的PID为5070，由它产生的子进程的PID为5071。  图29. orphan.c运行出来的进程树  时隔一段时间之后，再看orphan的位置。现在父进程已经退出，只剩下子进程（PID为5071），它被init(systemd)所收养，如图30所示。  现在我们再看僵尸进程。  当父进程没有退出而子进程正常退出之后，父进程没有及时调用wait或waitpid收回子进程的系统资源，该进程就是僵尸进程。为了构建僵尸进程，我们让子进程正常退出，而父进程执行时间长于子进程却不会收回子进程的资源。代码如图31所示。    图30. 子进程(PID=5071)被init(systemd)收养  图31. 构造僵尸进程  运行该代码，如图32所示，我们得到了这样的运行结果。父进程的PID为5327，其子进程的PID为5328，然后我们用ps j指令查看状态，如图33所示，子进程(PID=5328)已经成为了僵尸进程。  图33. 僵尸进程结果  图32. zombie.c运行结果   1. 编写代码，使进程运行5秒钟然后休眠5秒，然后查看进程状态。（cons.c）   有了前面的基础，这个程序就相对容易了。我们知道sleep函数中的参数可以控制程序休眠一段时间，单位为秒（linux环境下，windows依然是毫秒）。但是如何让程序运行5秒呢？  在C语言中，在标准C语言中<time.h>头文件中宏定义的一个常数CLOCKS\_PER\_SEC，用于将clock()函数的结果转化为以秒为单位的量。而我们知道在C语言中，clock()返回该程序从启动到函数调用占用CPU的时间，单位为毫秒。那我们就可以设置一个开始时间，然后不断用clock()截取时间，直到时间差为5秒。至于单位转化的问题，用clock()/CLOCKS\_PER\_SEC即可完成转化。  代码如图34所示。  图34. 交替进程代码  运行代码，然后我们使用ps指令查看一下运行时间。我们一般使用ps -eo指令，然后输入进程号，使其输出启动时间以及运行时间，这里仅仅代表了程序运行的时间，不包括程序休眠的时间。结果如图35所示，结果表明程序确实运行了5秒钟。  图35. 交替进程代码运行时间  至于休眠时间，我们已经在系统函数sleep中进行了规定。  下面我们用top函数来实时观测该进程的状态。运行程序，然后在另一个窗口运行top指令，可以观察到如图36和如图37两种进程状态。这就证明程序先运行然后休眠的过程。  图37. 交替进程—CPU占用率为0，程序处于sleep状态  图36. 交替进程—CPU占用率不为0，程序处于running状态    如图36所示，显示的是进程处于running状态；如图37所示，程序处于sleep状态，而且TIME+参数代表CPU时间总计，4.99秒的running也符合我们的设计。 |
| **四、实验结论：**  本次实验通过熟悉fork()函数、sleep()函数、waitpid()函数等等，实现了进程的创建、不同形状进程树的构造、进程的杀死、孤儿进程、僵尸进程以及交替进程的构造。结果图在第三部分中都已经显示。 |
| **五、实验体会：**  本次实验第一次通过fork函数构造进程。在分析进程树的时候，由于父进程会构造子进程，新构建的子进程也会构造子进程，所以有时候想建立设计形态的进程树并非易事，需要仔细思考之后才能创建成功。  关于孤儿进程以及僵尸进程，我们要养成良好的编程习惯，否则生成孤儿进程以及僵尸进程之后将会浪费系统资源。 |
| **实验附件**   1. helloworld.c为环境测试代码； 2. hello\_loop.c为循环代码； 3. tree.c为创建一个父进程拥有100个子进程的代码； 4. list.c为创建一个100层子进程且每个父进程只有一个子进程的线性结构的代码； 5. tree2.c为创建二叉树结构进程树的代码； 6. orphan.c为创建孤儿进程的代码； 7. zombie.c为创建僵尸进程的代码； 8. cons.c为创建交替进程的代码。 |
| 指导教师批阅意见：  成绩评定：  指导教师签字：  年 月 日 |
| 备注： |