|  |  |  |  |
| --- | --- | --- | --- |
| **《操作系统原理》实验报告** | | | |
| 实验名称 | 多进程编程 | 实验序号 | 2 |
| 实验日期 | 2023/3/31 | 实验人 |  |
| 一、实验题目  设计一个 C 程序作为一个 shell 接口，它接受用户命令，然后在单独的进程中执行每个命令。shell 界面给用户一个提示，然后输入下一个命令。  实现 shell 接口的一种技术是让父进程首先读取用户在命令行中输入的内容，然后创建一个单独的子进程来执行该命令。除非另有说明，否则父进程在继续之前等待子进程退出。但是，UNIX shell 通常还允许子进程在后台或同时运行。为此，我们在命令末尾添加一个与号 (&)，父进程和子进程将同时运行。单独的子进程是使用 fork() 系统调用创建的，用户的命令是使用系统调用 exec() 系列之一执行的。  用户在 osh> 提示符下输入命令 ps -ael，存储在 args 数组中的值是：  args[0] = "ps"  args[1] = "-ael"  args[2] = NULL  这个 args 数组将被传递给 execvp() 函数，它具有以下原型：  execvp (char \*command, char \*params[]) ;  这里，command 表示要执行的命令，params 存储此命令的参数。对于这个项目，execvp() 函数应该被调用为 execvp(args [0], args)。请务必检查用户是否包含 & 以确定父进程是否要等待子进程退出。  下一个要求是修改 shell 接口程序，使其提供history功能，允许用户访问最近输入的命令。通过使用该功能，用户最多可以访问 10 个命令。命令会从1开始连续编号，超过10会继续编号。例如，如果用户输入了35条命令，则最近的10条命令将编号为26到35。  程序还应管理基本的错误处理。当用户输入 !! 执行历史记录中的最新命令。当用户输入单 ! 后跟整数 N，执行历史记录中的第 N 个命令。该程序还应该管理基本的错误处理。如果历史中没有命令，输入 ! ！应该会产生一条消息“No commands in history”。如果没有与用单个 ! 输入的数字对应的命令，程序应该输出“No such command in history”。 | | | |
| 二、相关原理与知识   1. **Linux进程相关的基础知识**   在Linux操作系统中，进程在内存里有三部分的数据：“数据段”、”堆栈段”和“代码段”。 代码段即存放了程序代码的数据，假如机器中有数个进程运行相同的一个程序，那么它们就可以使用同一个代码段。堆栈段用来存放子程序的返回地址、子程序的参数以及程序的局部变量。数据段用来存放程序的全局变量，常数以及动态数据分配的数据空间（比如用malloc之类的函数取得的空间）。  在Linux系统中，使用fork()函数产生子进程的系统调用，它执行一次返回两个值：对于子进程，返回值为0；对于父进程，返回值为大于0的子进程pid，因此对于程序只要判断fork函数的返回值就可以判断自己是处于父进程还是子进程当中。调用fork()函数时，系统首先让新的进程与旧的进程使用同一个代码段，因为它们的程序还是相同的；对于数据段和堆栈段，系统则复制一份给新的进程。这样，父进程的所有数据都可以留给子进程，但是，子进程一旦开始运行，虽然它继承了父进程的一切数据，但实际上数据却已经分开，相互之间不再有影响了，也就是说，它们之间不再共享任何数据了。  通常，在系统调用fork()之后，一个进程会使用系统调用exec()，以用新程序来取代进程的内存空间。系统调用exec(将二进制文件装入内存(消除了原来包含系统调用exec0的程序的内存映射)，并开始执行。采用这种方式，两个进程能相互通信，并能按各自的方法执行。父进程能创建更多的子进程，或者如果在子进程运行时没有什么可做，那么它采用系统调用wait()把自己移出就绪队列来等待子进程的终止。子进程的pid值为0,而父进程的pid值大于0。子进程通过系统调用execlp() ( execlp()是系统调用exec()的一种版本)，用UNIX命令来覆盖其地址空间。父进程通过系统调用wait()来等待子进程的完成。当子进程完成时( 通过显示或隐式调用exit )，父进程会从wait()调用处开始继续，并调用系统调用exit()以表示结束。   1. **Linux下的shell框架与命令行解析**   通常而言，Linux下的一个shell可以化简为如下形式：  while(1)  {  typePrompt();  readCommand();    int pid = fork();    if(pid < 0)  {  puts("Unable to fork the child, inner error.");  }  else if(pid == 0) *// the child thread*  {  execve(command); *//execve the command*  }  else *// the parent thread*  {  wait(NULL); *//waiting for the child to exit*  }  }  **即shell中运行父进程，等待我们输入指令，之后fork()产生子进程，通过execvp()函数来执行输入。**我们的代码即参考这个思路，对上述代码进行扩充。  对于命令行的执行，用户在osh>提示符处输入命令，这些值存储在args数组上。args数组将被传递给execvp()函数，该函数具有原型:  Execvp (char \*command, char \*params[]);  在这里，command表示要执行的命令，params存储命令参数。对于这个项目， | | | |
| execvp()函数应该被execvp(args[0]，args)调用。 | | | |
| 三、实验过程  我们针对上述代码做出改进，以使其符合我们的要求。   1. **常量、变量定义部分**   窗体顶端  窗体底端   |  | | --- | | #define MAX\_LINE 80  #define MAX\_HISTORY 10  char input[MAX\_LINE];  char \*args[MAX\_LINE / 2 + 1];  int background;  char history[MAX\_HISTORY][MAX\_LINE];  int history\_count = 0;  窗体底端 |   其中MAX\_LINE为一行命令的最大字符数（包括空格），此处设为80；MAX\_HISTORY即记录的历史记录的最大个数，此处即为10.  input字符串用来记录用户输入的命令；  \*\*args即将input通过空格分离后的命令的各个部分，其中第二维大小为MAX\_LINE/2 + 1的解释为，当用户输入的命令达到最坏情况，即命令每个部分都只有1个字母然后用空格分隔，最终args中接收到的命令个数即为MAX\_LINE/2 + 1；  background记录用户命令是否让父子进程同时进行，若同时进行即为1，父进程等待子进程完成则为0；  \*\*history中记录具体命令的历史  history\_count用来记录当前历史记录的个数，当history\_count > MAX\_HISTORY时，记录历史记录时会对最前面的历史记录进行清除。   1. typePrompt()  |  | | --- | | printf("osh>"); |   即输出用户输入提示符“osh > ”。   1. readCommand()  |  | | --- | | background = 0;  fflush(stdout);  fgets(input, MAX\_LINE, stdin);    if (strcmp(input, "exit**\n**") == 0) {  **break**;  }    if (strcmp(input, "**\n**") == 0) {  continue;  }    if (strcmp(input, "history**\n**") == 0) {  if (history\_count <= MAX\_HISTORY)  for (int i = 0; i < history\_count; i++) {  printf("%d %s", i + 1, history[i]);  }  else  for(int i = 0; i < MAX\_HISTORY; i++){  printf("%d %s", history\_count - MAX\_HISTORY + i + 1, history[i]);  }    continue;  }    if (strcmp(input, "!!**\n**") == 0) {  if (history\_count == 0) {  printf("No commands in history**\n**");  continue;  } else {  strcpy(input, history[history\_count - 1]);  }  } else if (input[0] == '!') {  int index = atoi(input + 1);  if (index > history\_count || index < history\_count - MAX\_HISTORY + 1) {  printf("No such command in history**\n**");  continue;  } else {  strcpy(input, history[index - (history\_count - MAX\_HISTORY + 1)]);  }  } else {  add\_history(input, history, &history\_count);  }    parse\_input(input, args, &background);    if (strcmp(args[0], "cd") == 0) {  if (args[1] == NULL) {  fprintf(stderr, "osh: expected argument to **\"**cd**\"\n**");  } else {  if (chdir(args[1]) != 0) {  perror("osh");  }  }  continue;  } |   对于指令读取部分，主要步骤与思路如下：  首先对于要新输入的指令，初始化background为0，以判断父子进程是否并行；调用fflush函数清空缓冲区，确保“osh > ”立刻被打印在shell中，这对交互式进程十分重要，有助于用户了解程序运行情况。  随后调用fgets函数整行读取用户输入的指令，最长不超过MAX\_LINE，并保存在input中。针对execvp()函数直接执行无效果的几个函数，我们特别在此板块进行了定义，以求程序的完善，比如常见的“exit”、“cd”、“history”命令。  如程序所示，对于input，如果为空，则直接continue程序重新输入；如果输入exit直接break程序；如果输入history命令则遍历history数组，输出其中存储的历史记录。  对于cd命令，我选择使用chdir()函数实现路径跳转，具体即当args[0]读取到cd命令后，直接调用chdir(args[1])即可。  对于题目所要求的，“输入 !! 执行历史记录中的最新命令。当用户输入单 !！后跟整数 N，执行历史记录中的第 N 个命令”，代码中也均有所体现并针对异常状况做了处理：若当前无命令，打印“No such command in history”； 若N为10条之前的命令或还未输入的命令，或N缺失，打印“No such command in history”。  值得一提的是，对于所有非history类的代码（history、!!、!N），剩余输入的所有命令都将被使用add\_history()函数记录，该函数的具体定义如下：   |  | | --- | | void add\_history(char \*input, char history[][MAX\_LINE], int \*history\_count) {  if (\*history\_count < MAX\_HISTORY) {  strcpy(history[\*history\_count], input);  (\*history\_count)++;  } else {  for (int i = 1; i < MAX\_HISTORY; i++) {  strcpy(history[i - 1], history[i]);  }  strcpy(history[MAX\_HISTORY - 1], input);  (\*history\_count)++;  }  } |   当当前记录命令数量小于MAX\_HISTORY时，直接写入history数组；当当前记录命令等于MAX\_HISTORY时，删除第一条命令并将新命令加入history数组。  对于其他可以直接使用execvp()函数调用的linux命令，代码在检测到输入的指令并非所特殊指定的“exit”、“history”、“cd”、“!!”、“!N”后，会将首先将输入分割，并正确写入\*args数组中，其中输入input分割的使用parse\_input()函数，其具体定义如下：   |  | | --- | | void parse\_input(char \*input, char \*\*args, int \*background) {  char \*token = strtok(input, " **\n**");  int i = 0;  while (token != NULL) {  if (strcmp(token, "&") == 0) {  \*background = 1;  } else {  args[i++] = token;  }  token = strtok(NULL, " **\n**");  }  args[i] = NULL;  } |   输入input分割部分，使用strtok函数，以“ ”和“\n”为界对input进行分割并将分割结果写入\*args数组中。需要特别注意的是，**检测父子进程是否同时进行也在这一步**，即当指令结尾检测到“&”时，说明父子进程同时进行，此时将background置为1，在进程运行的时候使用。   1. fork()函数生成子进程   具体代码如下：   |  | | --- | | pid\_t pid = fork();    if (pid == 0) {  if (execvp(args[0], args) == -1) {  perror("execvp");  exit(EXIT\_FAILURE);  }  } else if (pid > 0) {  if (!background) {  waitpid(pid, NULL, 0);  }  } else {  perror("fork");  exit(EXIT\_FAILURE);  } |   即首先生成父子进程的进程号，之后对于子进程，直接调用execvp()函数执行输入的命令；对于父进程，根据background值决定是否等待子进程完成。 | | | |
| 四、实验结果与分析   1. 执行命令 | | | |
| 1. history命令   （1）历史记录低于10条    （2）历史记录多于10条     1. !!命令   （1）无历史记录    （2）有历史记录     1. !N命令   （1）N在最近十条命令内    （2）N不在最近十条命令内    （3）N缺失     1. cd 命令     如图，此处执行了从EXP-2 -> 操作系统上机 -> 文档 -> 操作系统上机 -> EXP-2   1. exit命令 | | | |
| 五、问题总结   1. 获取命令时用了gets()函数，gets()函数是个危险函数，在Linux虚拟机终端编译不通过。   解决方式：使用更为安全的fgets函数   1. 调用execvp(args[0],args)时出现错误   查阅资料，发现原因为：将i个命令参数存入args后，args[i]未置为NULL  解决方式：将i个命令参数存入args后，把args[i]置为NULL | | | |
| 六、源代码  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <sys/types.h>  #include <sys/wait.h>  #define MAX\_LINE 80  #define MAX\_HISTORY 10  void parse\_input(char \*input, char \*\*args, int \*background) {  char \*token = strtok(input, " \n");  int i = 0;  while (token != NULL) {  if (strcmp(token, "&") == 0) {  \*background = 1;  } else {  args[i++] = token;  }  token = strtok(NULL, " \n");  }  args[i] = NULL;  }  void add\_history(char \*input, char history[][MAX\_LINE], int \*history\_count) {  if (\*history\_count < MAX\_HISTORY) {  strcpy(history[\*history\_count], input);  (\*history\_count)++;  } else {  for (int i = 1; i < MAX\_HISTORY; i++) {  strcpy(history[i - 1], history[i]);  }  strcpy(history[MAX\_HISTORY - 1], input);  (\*history\_count)++;  }  }  int main() {  char input[MAX\_LINE];  char \*args[MAX\_LINE / 2 + 1];  int background;  char history[MAX\_HISTORY][MAX\_LINE];  int history\_count = 0;  while (1) {  background = 0;  printf("osh > ");  fflush(stdout);  fgets(input, MAX\_LINE, stdin);  if (strcmp(input, "exit\n") == 0) {  break;  }  if (strcmp(input, "\n") == 0) {  continue;  }  if (strcmp(input, "history\n") == 0) {  if (history\_count <= MAX\_HISTORY)  for (int i = 0; i < history\_count; i++) {  printf("%d %s", i + 1, history[i]);  }  else  for(int i = 0; i < MAX\_HISTORY; i++){  printf("%d %s", history\_count - MAX\_HISTORY + i + 1, history[i]);  }    continue;  }  if (strcmp(input, "!!\n") == 0) {  if (history\_count == 0) {  printf("No commands in history\n");  continue;  } else {  strcpy(input, history[history\_count - 1]);  }  } else if (input[0] == '!') {  int index = atoi(input + 1);  if (index > history\_count || index < history\_count - MAX\_HISTORY + 1) {  printf("No such command in history\n");  continue;  } else {  strcpy(input, history[index - (history\_count - MAX\_HISTORY + 1)]);  }  } else {  add\_history(input, history, &history\_count);  }  parse\_input(input, args, &background);  if (strcmp(args[0], "cd") == 0) {  if (args[1] == NULL) {  fprintf(stderr, "osh: expected argument to \"cd\"\n");  } else {  if (chdir(args[1]) != 0) {  perror("osh");  }  }  continue;  }  pid\_t pid = fork();  if (pid == 0) {  if (execvp(args[0], args) == -1) {  perror("execvp");  exit(EXIT\_FAILURE);  }  } else if (pid > 0) {  if (!background) {  waitpid(pid, NULL, 0);  }  } else {  perror("fork");  exit(EXIT\_FAILURE);  }  }  return 0;  } | | | |