操作系统

实验报告

实验名称：实验二 进程间通信和命令解释器

姓名：陈亚楠

学号：16340041

实验名称：**进程间通信和命令解释器**

一、实验目的：

1.进程间共享内存实验，初步了解进程间通讯；

2. 实现简单的 Shell 命令解释器，了解程序运行。

二、实验要求：

Linux 下程序编译和调试实验

三、实验过程：

1.进程间共享内存实验：

Fibonacci 序列是一组数: 0,1,1,2,3,5,8,..., 通常它可以表示为:$fib_0 = 0$，$fib_1 = 1$，$fib_n = fib_{n-1} + fib_{n-2}$。使用系统调用 fork()编写一个 C 程序，它在其子程序中生成 Fibonacci 序列。这个程序采用 POSIX 共享内存方法来构建。

实验过程：

（1）首先创建共享内存段的数据结构，此数据结构包括两项：①长度为 MAX_SEQUENCE 的固定长度数组，它保存 Fibonacci 的值; ②子进程生成的序列的大小 sequence_size,其中 sequence_size ≤ MAX_SEQUENCE：

```
#define MAX_SEQUENCE 10
typedef struct {
    int fib_sequence[MAX_SEQUENCE];
    int sequence_size;
} shared_data;
```

（2）父进程接受命令行上传递的参数，执行错误检查以保证参数不大于 MAX_SEQUENCE：

```
// check
printf("input a sequence_size no more than 10:\n");
int n = 0;
scanf("%d", &n);
while(n > MAX_SEQUENCE) {
    printf("error sequence_size, input again:\n");
    scanf("%d", &n);
}
```

（3）父进程创建一个大小为 shared_data 的共享内存段：

```
// allocate a shared memory segment
segment_id = shmget(IPC_PRIVATE, 80, S_IRUSR | S_IWUSR);
if(segment_id < 0) {
    printf("shmget error: %s\n", strerror(errno));
    return -1;
}
```

（4）父进程将共享内存段附加到地址空间 shared_memory：

```
// attach the shared memory segment
shared_memory = (shared_data*)shmat(segment_id, NULL, 0);
if(shared_memory == (void*)-1) {
    printf("shmat error: %s\n", strerror(errno));
    return -1;
}
```

（5）父进程将命令行参数值赋予 shared_data：

```
// assignment
shared_memory -> sequence_size = n;
```

（6）父进程创建子进程，并调用系统调用 wait()等待子进程结束：

```c
// fork
if((pid = fork()) < 0) {
    printf("fork error: %s\n", strerror(errno));
    return -1;
} else if(pid == 0) {
    printf("child process:\n");
    shared_memory -> fib_sequence[0] = 0;
    shared_memory -> fib_sequence[1] = 1;
    for(int i = 2; i < shared_memory -> sequence_size; i++) {
        shared_memory -> fib_sequence[i] =
            shared_memory -> fib_sequence[i - 1] + shared_memory -> fib_sequence[i - 2];
    }
} else if(pid > 0) {
    wait(NULL);
    printf("parent process:\n");
    for (int i = 0; i < shared_memory -> sequence_size; i++)
    {
        printf("%d ", shared_memory -> fib_sequence[i]);
    }
    printf("\n");
}
```

（7）父进程输出共享内存段中 Fibonacci 序列的值：

```
chen@ChenYanan:~/桌面$ gcc fib_posix.c  -o fib_posix
chen@ChenYanan:~/桌面$ ./fib_posix
input a sequence_size no more than 10:
5
child process:
parent process:
0 1 1 2 3
chen@ChenYanan:~/桌面$ ./fib_posix
input a sequence_size no more than 10:
11
error sequence_size, input again:
6
child process:
parent process:
0 1 1 2 3 5
```

（8）父进程释放并删除共享内存段：

```c
// now detach the shared memory segment
if(shmdt(shared_memory) < 0) {
    printf("shmdt error: %s\n", strerror(errno));
    return -1;
}

// now remove the shared memory segment
if(shmctl(segment_id, IPC_RMID, NULL) < 0) {
    printf("shmctl error: %s\n", strerror(errno));
    return -1;
}
```

在本实验中，由于子进程是父进程的一个副本，共享内存区域也将被附加到子进程的地址空间。然后，子进程将会把 Fibonacci 序列写入共享内存并在

最后释放此区域。

二. 项目：UNIX Shell 和历史特点

此项目由修改一个 C 程序组成，它作为接收用户命令并在单独的进程执行每个命令的 Shell 接口。Shell 接口在下一个命令进入之后为用户提供了提示符。实现 Shell 接口的一种技术是父进程首先读用户命令行的输入，然后创建一个独立的子进程来完成这个命令。UNIX Shell 通过在命令的最后使用"&"符号允许子进程在后台运行（或并发地运行）。用系统调用 fork()来创建独立的子进程，通过使用 exec()族中的一种系统调用来执行用户命令。

1. 简单 Shell

该项目给出了一种基本的命令行 Shell 操作的 C 程序。此程序包括两个函数: main()和 setup()。setup() 函数读取用户的下一条命令(最多 80 个字符)，然后将之分析为独立的标记，这些标记被用来填充命令的参数向量（如果将要在后台运行命令，它将以"&"结尾，setup() 将会更新参数 background，以使 main() 函数相应地执行)。当用户按快捷键 Ctrl+D 后，setup()调用 exit()，此程序被终止。

Main()函数打印提示符 COMMAND-> ,然后调用 setup()，它等待用户输入命令。用户输入命令的内容被装入一个 args 数组。

这个项目由两部分组成：（1）创建子进程,，并在子进程中执行命令；（2）修改 Shell 以允许一个历史特性。

setup()函数的实现细节：

```c
/* setup()用于读入下一行输入的命令，并将它分成没有空格的命令和参数存于数组args[]中，
 * 用NULL作为数组结束的标志
 */
void setup(char inputBuffer[], char *args[], int *background) {
    int length = 0, /* 命令的字符数目 */
        i = 0,      /* 循环变量 */
        start = 0,  /* 命令的第一个字符位置 */
        ct = 0;     /* 下一个参数存入args[]的位置 */

    /* 读入命令行字符，存入inputBuffer */
    length = read(STDIN_FILENO, inputBuffer, MAX_LINE);

    start = -1;
    if (length == 0) {
        exit(0); /* 输入ctrl+d，结束shell程序 */
    }
    if (length < 0) {
        perror("error reading the command");
        exit(-1); /* 出错时用错误码-1结束shell */
    }
    /* 检查inputBuffer中的每一个字符 */
    for (i = 0; i < length; i++) {
        switch (inputBuffer[i]) {
            case ' ':
            case '\t': /* 字符为分割参数的空格或制表符(tab)'\t'*/
                if (start != -1) {
                    args[ct] = &inputBuffer[start];
                    ct++;
                }
                inputBuffer[i] = '\0'; /* 设置 C string 的结束符 */
                start = -1;
                break;
            case '\n': /* 命令行结束 */
                if (start != -1) {
                    args[ct] = &inputBuffer[start];
                    ct++;
                }
                inputBuffer[i] = '\0';
                args[ct] = NULL; /* 命令及参数结束 */
                break;
            default: /* 其他字符 */
                if (start == -1) {
                    start = i;
                }
                if (inputBuffer[i] == '&') {
                    *background = 1; /* 置命令在后台运行 */
                    inputBuffer[i] = '\0';
                }
        }
    }
    args[ct] = NULL; /* 命令字符数 > 80 */
}
```

2.创建子进程

父进程从 setup()返回时要创建一个子进程，并执行用户的命令。对于该项

目, 需要保证检测 background 的值，以决定父进程是否需要。

3.创建历史特性

这一步要修改程序使其能提供一个历史特性来允许用户能访问 10 个最近输入的命令。

当用户按下 SIGINT 信号 Ctrl+C 键时，系统能列出这 10 个命令。UNIX 使用信号(signals)来通知进程发生了一个特定事件。信号可以是同步的，也可以是异步的，这取决于资源及发出信号事件的原因。一旦因一个特定事件发生而生成了一个信号（如被零除，非法内存访问，或用户按快捷键 Ctrl+C 等），该信号被传送到必须处理该信号的进程。接收信号的进程可按如下方法处理:

①忽略信号；

②使用错误信号处理器；

③提供一个单独的信号处理函数。

这里我们选择建立自己的信号处理函数 handle_SIGINT()来处理 SIGINT 信号，并在 main()函数中创建信号处理器：

创建信号处理器：

```
/* 创建信号处理器 */
struct sigaction handler;
handler.sa_handler = handle_SIGINT;
sigaction(SIGINT, &handler, NULL);

/* 生成输出消息 */
strcpy(command.buffer, "Caught Control C\n");
```

建立信号处理函数：

```c
/* 信号处理函数 */
void handle_SIGINT() {
    write(STDOUT_FILENO, command.buffer, strlen(command.buffer));
    int count = 0;
    //从命令列表里输出命令
    for(int j = command_pointer - 1; j > 0; j--) {
        write(STDOUT_FILENO, history_command[j].buffer, strlen(history_command[j].buffer));
        count++;
        if(count >= 10) {
            break;
        }
    }
    exit(0);
}
```

在这之前我们将 buffer 处理成结构体，并创建了一个命令列表：

```c
//  定义buffer结构体
#define BUFFER_SIZE 50
typedef struct {
    char buffer[BUFFER_SIZE];
} _command;
_command command;

//  用来存放命令的数组，  命令列表
#define HISTORY_COMMAND_SIZE 50
_command history_command[HISTORY_COMMAND_SIZE];
int command_pointer = 0;
```

在该项目中，当命令被检测为错误时，则应将一则出错消息传给用户，并且该命令不进入历史列表，也不应调用 execvp() 函数。因此，这一步处理的时候，我们修改 setup()函数返回值，并在命令没有错误后将其加入命令列表：

```c
int setup(char inputBuffer[], char *args[], int *background) {
    int length = 0, /* 命令的字符数目 */
        i = 0,      /* 循环变量 */
        start = 0,  /* 命令的第一个字符位置 */
        ct = 0;     /* 下一个参数存入args[]的位置 */

    /* 读入命令行字符，存入inputBuffer */
    length = read(STDIN_FILENO, inputBuffer, MAX_LINE);

    start = -1;
    if (length == 0) {
        return 0;; /* 输入ctrl+d，结束shell程序 */
    }
    if (length < 0) {
        perror("error reading the command");
        return -1; /* 出错时用错误码-1结束shell */
    }

    memcpy(history_command[command_pointer].buffer, inputBuffer, strlen(inputBuffer));
    command_pointer++;
```

除此之外，修改 main()函数，在当命令错误时，不应调用 execvp() 函数：

```c
   int flag = setup(inputBuffer, args, &background); /* 获取下一个输入的命令 */
   /* 这一步要做：…
   */
  pid_t pid = fork();
  if(pid < 0) {
      printf("fork error");
  } else if(pid == 0) {
      printf("in child process, this process id is %d\n", getpid());
      if(flag == -1 || flag == 0) {
          return 0;
      }
      execvp(args[0], args);
```

实验结果：

```
chen@ChenYanan:~/桌面$ ./shell
cal
COMMAND->in parent process, this process id is 3762
COMMAND->in child process, this process id is 3763
      四月 2018
日 一 二 三 四 五 六
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

date
cal
COMMAND->in parent process, this process id is 3762
COMMAND->in child process, this process id is 3764
      四月 2018
日 一 二 三 四 五 六
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

date
date
COMMAND->in parent process, this process id is 3762
COMMAND->in child process, this process id is 3765
2018年 04月 28日 星期六 23:31:21 CST
^CCaught Control C
date
date
cal
date
```