操作系统课程设计实验报告

实验名称	进程控制		
学号	1120141831	姓名	朴泉宇

一、实验目的

在 Windows 以及 Linux 平台下实现多进程实现，掌握多进程的基本实现方法以及运行过程，为接下来的进程之间的信号量等实验打下基础。

二、实验内容

1. 分别编写 Windows 下的多进程代码、Linux 下的多进程代码
2. 在 Windows 下编译运行程序，得到相应结果
3. 在 Linux 下编译运行程序，得到相应结果
4. 对比 Windows 下以及 Linux 下的代码以及运行的逻辑区别

三、实验环境及配置方法

1. 利用 Visual Studio 编写 Windows 下的代码，并编译运行
2. 利用 Notepad++编写 Linux 下的代码，在 Ubuntu 中调用 g++编译，运行

四、实验方法和实验步骤（程序设计与实现）

1. 编写 Windows 下的多进程代码 - ChildProcess

ChildProcess 只完成输出特定字符串以及等待 3 秒的功能。

```
1.  #include <stdio.h>
2.  #include <Windows.h>
3.
4.  int main()
5.  {
6.      printf("Hi, my name is Piao Quanyu.\n");
7.      Sleep(3000);
8.
9.      return 0;
10. }
```

2. 编写 Windows 下的多进程代码 - ParentProcess

(1) 进行多进程的初始化操作。

若传入的参数有误（例如没有在 cmd 中正确调用子进程，体现于 argc），则报错并结束。

```
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    memset(&si, 0, sizeof(si));
    si.cb = sizeof(si);
    memset(&pi, 0, sizeof(pi));

    if (argc != 2)
    {
        printf("Usage: %s [cmdline]\n", argv[0]);
        return;
    }
```

(2) 创建子进程。

子进程调用正确，则开始进行 CreateProcess()，如果生成子进程出错，则结束程序；若成功调用子进程，则父进程打出“CreateProcess Success!”的字符串，并调用 GetSystemTime()获得当前时间并输出。

```
if (!CreateProcess
    (
        NULL,
        argv[1],
        NULL,
        NULL,
        FALSE,
        0,
        NULL,
        NULL,
        &si,
        &pi
    )
)
{
    printf("CreateProcess failed (%d).\n", GetLastError());
    return;
}
```

```
else
{
    printf("CreateProcess Success!\n");
    GetSystemTime(&startTime);
    printf("ChildProcess started at %04d.%02d.%02d %s %02d:%02d:%02d.%04d UTC.\n", startTime.wYear, startTime.wMonth, startTime.wDay, arrDayofWeek[startTime.wDayOfWeek], startTime.wHour, startTime.wMinute, startTime.wSecond, startTime.wMilliseconds);
}
```

其中，时间变量声明为：

```
    SYSTEMTIME startTime, endTime;
```

星期的格式化输出数组为：

```
/*Day of Week Format*/
TCHAR arrDayofWeek[7][4] =
{
    TCHAR("Sun"),
    TCHAR("Mon"),
    TCHAR("Tue"),
    TCHAR("Wed"),
    TCHAR("Thu"),
    TCHAR("Fri"),
    TCHAR("Sat"),
};
```

(3) 等待子进程结束并打印结束时间和子进程所用时间。

调用 WaitForSingleObject()函数来等待子进程的结束，之后再次调用 GetSystemTime()来获得子进程结束后的时间，并且计算子进程使用时间，将二者都输出。

```
    WaitForSingleObject(pi.hProcess, INFINITE);
    GetSystemTime(&endTime);
    printf("ChildProcess ended at %04d.%02d.%02d %s %02d:%02d:%02d.%04d UTC.\n", endTime.wYear, endTime.wMonth, endTime.wDay, arrDayofWeek[endTime.wDayOfWeek], endTime.wHour, endTime.wMinute, endTime.wSecond, endTime.wMilliseconds);

    int milliseconds, seconds, minutes;
    milliseconds = endTime.wMilliseconds - startTime.wMilliseconds;
    seconds = endTime.wSecond - startTime.wSecond;
```

```
    minutes = endTime.wMinute - startTime.wMinute;
    if (milliseconds < 0)
    {
        milliseconds = milliseconds + 1000;
        seconds--;
    }
    if (seconds < 0)
    {
        seconds = seconds + 60;
    }
    if (minutes < 0)
    {
        minutes = minutes + 60;
    }
    printf("ChildProcess used %ds%dms.\n", seconds, milliseconds);
```

（4）关闭子进程句柄

```
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
```

3. 编写 Linux 下的多进程代码 - ChildProcess

与 Windows 下的代码类似，只在头文件以及 sleep()函数上有所区别。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    printf("Hi, my name is Piao Quanyu.\n");
    sleep(3);

    return 0;
}
```

4. 编写 Linux 下的多进程代码 - ParentProcess

(1) 时间变量声明

```
1.      struct timeval startTime;
2.      struct timeval endTime;
3.      int seconds, milliseconds;  //Used for printing usedTime
```

(2) 子进程相关变量的声明

pid_t pid：记录 fork()产生的子进程的 pid，用于接下来父子程序的判断。

char *exec_argv[4]：记录调用的子进程参数，用于 execv()时的传参。

int i：用于记录父进程调用子进程参数时的循环过程。

```
1.      pid_t pid;
2.      char *exec_argv[4];
3.      int i;
```

(3) 检查子进程调用是否正确（检查 argc）

```
1.      if(argc != 2)
2.      {
3.          printf("argc Error!\n");
4.          printf("CreateProcess failed.\n");
5.          return 0;
6.      }
```

(4) fork()出子进程，进入 if 部分，并调用 execv()切换到 ChildProcess

```
1.      pid = fork();
2.
3.      if(pid==0)  //ChildProcess goes to here
4.      {
5.          for(i = 0; i<argc; i++)
6.          {
7.              exec_argv[i] = argv[i+1];   //argv[0]:ParentProcess, argv[1]:ChildProc
    ess argv[2]:NULL
8.          }   //exec_argv[0]:ChildProcess, exec_argv[1]:NULL
9.
10.         printf("CreateProcess Success!\n");
11.         execv(argv[1], exec_argv);  //execv("ChildProcess", exec_argv);
12.     }
```

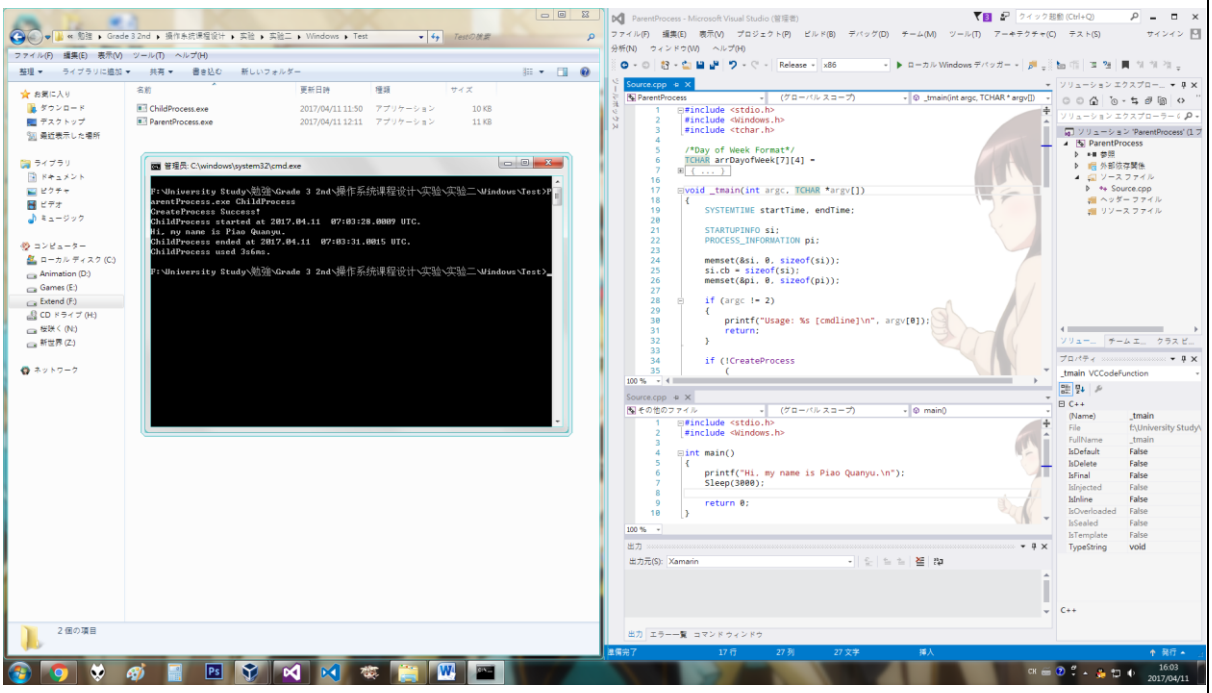
(5) 此时，父进程进入 else 部分

调用 asctime()以及 gmtime()输出子进程开始运行的时间。调用 wait()等待子进程结束，打出子进程结束的时间。计算输出子进程所用时间。

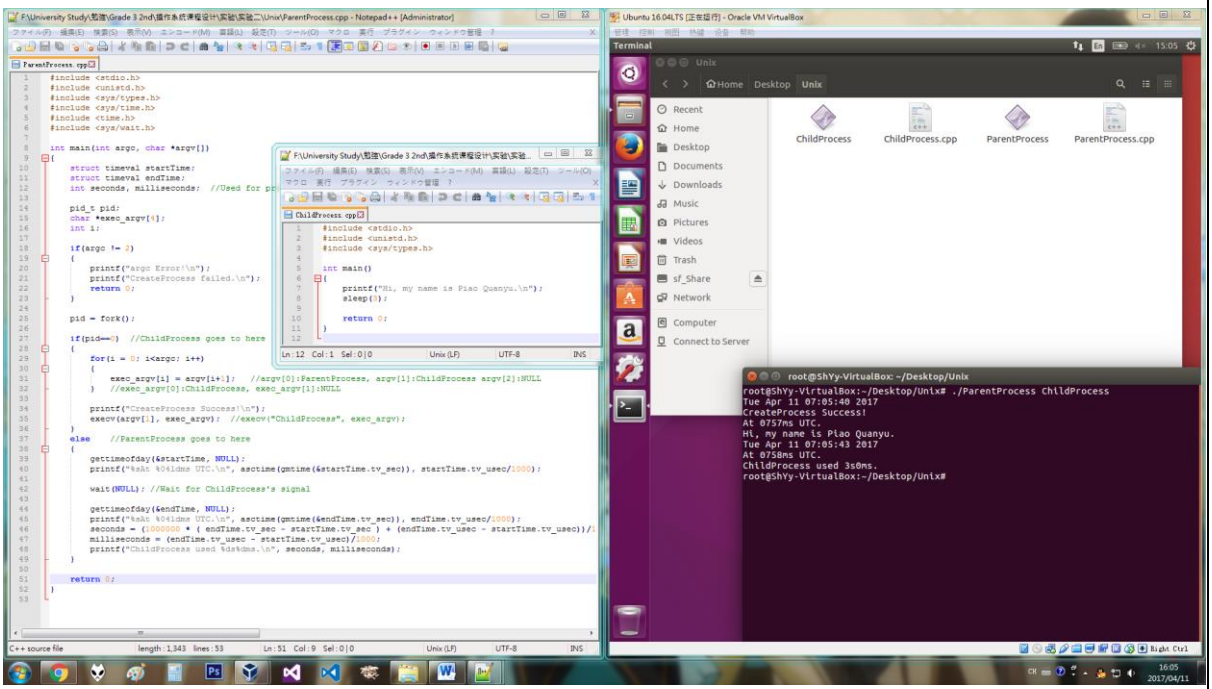
```
1.      else    //ParentProcess goes to here
2.      {
3.          gettimeofday(&startTime, NULL);
4.          printf("%sAt %04ldms UTC.\n", asctime(gmtime(&startTime.tv_sec)), startTim
    e.tv_usec/1000);
5.
6.          wait(NULL); //Wait for ChildProcess's signal
7.
8.          gettimeofday(&endTime, NULL);
9.          printf("%sAt %04ldms UTC.\n", asctime(gmtime(&endTime.tv_sec)), endTime.tv
    _usec/1000);
10.         seconds = (1000000 * ( endTime.tv_sec - startTime.tv_sec ) + (endTime.tv_u
    sec - startTime.tv_usec))/1000000;
11.         milliseconds = (endTime.tv_usec - startTime.tv_usec)/1000;
12.         printf("ChildProcess used %ds%dms.\n", seconds, milliseconds);
13.     }
```

五、 实验结果和分析

1. Windows 下的实验结果



2. Linux 下的实验结果



3. 对比分析

Windows 下的 CreateProcess()子进程调用，没有代码、数据、堆栈等的共用，是执行一个全新的程序，并且通过 WaitForSingleObject()进行父子进程之间的同步。

Linux 下的 fork()，如其函数名的意思，就是分支，父子进程到 fork()函数之前的代码、数据、堆栈都是共用的，但 fork()之后的程序则分开运行。父进程继续运行，而子进程可以调用 execv()替换掉 fork()来的进程，而只保留 pid，并运行另一个已经写好的程序（ChildProcess）。父进程则调用 wait()来等待 fork()出的子进程结束，完成同步。

六、 讨论、心得

这次实验实现了 Windows 和 Linux 平台下的子进程的创建、运行以及父子进程的同步，为接下来更复杂的多进程编码打下基础。

多进程编码和以前编过的单进程不同，模块之间的耦合度更低，编码复杂度更低，但相应的，性能方面不及单进程多线程的编码方式，可谓各有优势。

通过这次实验，我对操作系统的线程、进程有了进一步了解。在接下来的实验学习中力求对操作系统有更深刻的理解。