操作系统课程设计实验报告

**实验二 Windows与Linux进程控制**

指导教师：陆慧梅老师

班 级：07111507

学 号：1120151880

姓 名：廖汉龙

邮 箱：[liaohanlong@outlook.com](mailto:liaohanlong@outlook.com)

2018 年 3 月 12 日

一、实验目的

**1.了解Linux系统操作与命令**

**(1)掌握在Windows及Linux系统下的进程创建**

掌握在Windows系统及Linux系统下的进程创建过程:实现在Windows下使用CreateProcess（）函数创建进程、GetSystemtime（）获取进程运行时间（由于实验要求最后的程序运行时间精确到微秒，GetSystemtime()函数只能够精确到毫秒，所以，最后，实验采用了QueryPerformanceCounter()与函数QuetyPerformanceCouner()）；在Linux下使用fork()/vfork()/exec（）创建进程，gettimeofday（）获取进程时间。

**(2)实现子进程调用并获取运行时间**

在Windows及Linux系统下掌握调用子进程的方式，并使用相应函数获取进程运行时间。

**2.了解Linux 下基本的程序编译链接操作**

**(1)了解Linux下的程序编译原理**

**(2)了解Linux下的程序编译基本的操作**

通过实践，掌握在Linux 下的一些基本的编译程序的过程原理，并且能够熟练操作。

**二、实验内容**

设计并实现Unix的“time”命令。“mytime”命令通过命令行参数接受要运行的程序，创建一个独立的进程来运行该程序，并记录程序运行的时间。

在Windows下实现:

* 使用CreateProcess()来创建进程
* 使用WaitForSingleObject()在“mytime”命令和新创建的进程之间同步
* 调用GetSystemTime()来获取时间

在Linux下实现:

* 使用fork()/vfork /exec()来创建进程运行程序
* 使用wait()等待新创建的进程结束
* 调用gettimeofday()来获取时间

mytime的用法：  
$ mytime.exe program1

要求输出程序program1运行的时间

$ mytime.exe program2 t

t为时间参数，为program2的输入参数，控制program2的运行时间。最后输出program2的运行时间，应和t基本接近。

显示结果： \*\*小时\*\*分\*\*秒\*\*毫秒\*\*微秒

**三、实验环境及配置方法**

**1.Windows环境：**

操作系统：Windows 10 家庭中文版

编译环境：Visual Studio Code

**2.Linux环境**

虚拟机软件：VMware Workstation 12 Player

内存：2048MB

处理器个数：2

操作系统：Ubuntu 16.04 LTS 英文版

内核版本号：Linux 4.6.10

GCC版本号：5.3.1 20160413（Ubuntu 5.3.1-14ubuntu2）

Make版本号：GNU Make 4.1

编译软件：Visual Studio Code

**四、实验方法和实验步骤（程序设计与实现）**

**1.windows 系统下的设计与实现**

实验分为两个部分:

Mytime program1

Mytime program2 t

其中t是传入program2 的参数，用以控制program2的运行时间

**（1）创建进程：**

使用到的API为：

|  |
| --- |
| BOOL bCreateOK = CreateProcess(  NULL, //可执行的应用程序名称  argv[1], //指定创建一个子进标识，可以理解为子进程传入参数  NULL, //缺省的进程安全性  NULL, //缺省的线程安全性  FALSE, //不继承打开文件的句柄  CREATE\_NEW\_CONSOLE, //使用新的控制台  NULL, //新的环境  NULL, //当前目录  &si, //启动信息  &pi); //返回进程和线程信息 |

其中作为返回进程信息的结构变量pi,其主要的结构为：

|  |
| --- |
| PROCESS\_INFORMATION pi;  typedef struct \_PROCESS\_INFORMATION {  HANDLE hProcess; //进程句柄  HANDLE hThread; // 进程主线程句柄  DWORD dwProcessId; // 进程号  DWORD dwThreadId; // 进程主线程号  } PROCESS\_INFORMATION |

（2）子进程同步以及调用不用的子程序

同步子程序的函数调用使用API为：WaitForSingleObject(pi.hProcess, INFINITE);

此处API的第二个结构设置为INFINITE, 不限制得到回应的时间，如果设置特定时间，则会在到达时间后返回。

因为mytime程序需要实现分别调用一个参数和两个参数的功能，所以需要对传入的参数进程判定，并且实现将参数传入子程序。程序的传入参数argc 记录了参数个数，当参数的个数为3时，表明此时传入了一个参数t, t此时作为主程序的第三个参数（第一个参数是当前的程序的可执行文件的路径）,需要把t传入子进程。

在控制台的调用形式形如：

Mytime program2 t

所以只需要对此进程判断即可。

|  |
| --- |
| if(argc == 3) CreateArgv(argv[1], argv[2]); |

CreatArgv（）函数的定义为：

|  |
| --- |
| void CreateArgv(TCHAR t1[], TCHAR t2[]){  int i = 0;  while(t1[i] != '\0'){  i ++;  }  t1[i] = ' ';  i++;  int j = 0;  while(t2[j] != '\0'){  t1[i++] = t2[j++];  }  t1[i] = '\0';  } |

（3）记录子程序运行时间

由于实验要求最后的结果需要精确到微秒，但是实验给出的GetSystemTime()函数只能精确到毫秒，所以，此处采用了QueryPerformanceCounter()与函数QuetyPerformanceCouner()对时间的记录进程了处理。

在程序中需要定义：

|  |
| --- |
| LARGE\_INTEGER timeStart, timeEnd;  LARGE\_INTEGER frequency;  QueryPerformanceFrequency(&frequency); |

最后通过自定义函数printtime(&timeStart, &timeEnd)对时间进行进行运算与打印。

函数定义如下：

|  |
| --- |
| void printtime(LARGE\_INTEGER starttime, LARGE\_INTEGER endtime, double quadpart){  double elapsed = (endtime.QuadPart - starttime.QuadPart) / quadpart;  int h = elapsed / 3600;  int m = (elapsed - h \* 3600) / 60;  int s = elapsed - h \* 3600 - m \* 60;  int ms = (elapsed - h \* 3600 - m \* 60 - s) \* 1000;  int us = (elapsed - h \* 3600 - m \* 60 - s - ms / 1000) \* 1000;  printf("The child process' running time is %dh %dm %ds %dms %dus\n", h, m, s, ms, us);  } |

由于实验要求需要使用GetSystemTime()的方法进行系统时间的获取，所以，再次我也对这个方法进行一定的阐述：

SYSTEMTIME结构如下：

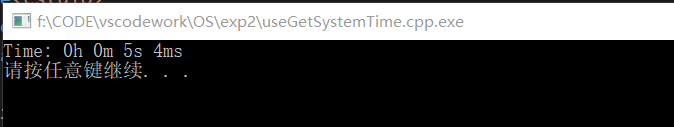
|  |
| --- |
| typedef struct \_SYSTEMTIME {      WORD wYear;      WORD wMonth;      WORD wDayOfWeek;      WORD wDay;      WORD wHour;      WORD wMinute;      WORD wSecond;      WORD wMilliseconds;  } SYSTEMTIME, \*PSYSTEMTIME, \*LPSYSTEMTIME; |

其中可以发现，从年到微秒都可以得到，调用方式非常简单，只需要使用GetSystemTime()将SYSTEMTIME结构变量指针传入。

我对此函数进行了实验：

|  |
| --- |
| #include<Windows.h>  #include<iostream>  #include<cstdio>  #include<cstdlib>  using namespace std;  void printtime(SYSTEMTIME start, SYSTEMTIME end){  int ms = end.wMilliseconds - start.wMilliseconds;  int s = end.wSecond - start.wSecond;  int m = end.wMinute - start.wMinute;  int h = end.wHour - start.wHour;  if(ms < 0){ms += 1000; s -= 1;}  if(s < 0){s += 60; m -= 1;}  if(m < 0){m += 60; h += 1;}    printf("Time: %dh %dm %ds %dms\n", h, m, s, ms);  }  int main(){  SYSTEMTIME starttime, endtime;  GetSystemTime(&starttime);  Sleep(5000);  GetSystemTime(&endtime);  printtime(starttime, endtime);    system("pause");  return 0;  } |

这段程序可以得到将进程挂起5s之后得到的进行运行时间，最后得到的结果为：



在这段程序中，我直接写了函数对于时间直接进行处理，但是微软官方推荐使用SystemTimeToFileTime函数，由于需要计算程序运行的时间，所以需要对得到的两个时间进行数学运算，所以采用FILETIME结构帮助运算。

FILETIME结构体表示的是自1601年1月1日起经过的100ns数，精度为64bit。

结构如下：

|  |
| --- |
| BOOL  WINAPI  SystemTimeToFileTime(      \_In\_ CONST SYSTEMTIME \* lpSystemTime,      \_Out\_ LPFILETIME lpFileTime      ); |

**(4)子程序program1， program2, 以及主程序mytime 附于实验报告后**

2.Linux 环境下的设计与实现

Linux 环境下需要实现的功能与windows 系统下实现的功能是一致的，这是使用的是Linux系统所提供的API;

(1) 使用fork()创建子进程。

fork()在创建子程序时有三种返回形式，如果创建失败，则返回-1；创建成功在子进程中返回0，在原进程中返回进程号，结构如下：

|  |
| --- |
| extern \_\_pid\_t fork(void) \_\_THROWNL; |

如果进程创建成功，则在子进程中使用execl()函数复制子程序。

由于使用fork()得到的子进程基本是父进程的一个副本，还是在执行当前的程序，所以在创建了子进程之后，需要调用exec()函数族来使得子进程启动另外一个程序执行。此处选择了使用的API是execl()，其定义的方式形如

|  |
| --- |
| extern int execl(const char \*path, const char \*arg, ...) |

PATH环境变量包含了一张目录表，系统通过PATH环境变量定义的路径搜索执行码，PATH环境变量定义时目录之间需用用“:”分隔，以“.”号表示结束。PATH环境变量定义在用户的.profile或.bash\_profile中，下面是PATH环境变量定义的样例，此PATH变量指定在“/bin”、“/usr/bin”和当前目录三个目录进行搜索执行码。  
PATH=/bin:/usr/bin:.

此处调用的是子程序的可执行路径。

arg表示的是子程序需要传入的参数。

此实验要求有两种参数的传入方式，所以需要对主程序的参数个数进程判断：

|  |
| --- |
| char \*arg1 = argv[1];  if(argc == 3){  char \*arg2 = argv[2];  execl(argv[1], arg1, arg2, NULL);}  else if(argc == 2){  execl(argv[1], arg1, NULL); |

在判断有三个参数，此时的命令行的调用形式形如

Mytime program2 t

需要把t作为参数传给子程序。

这个创建子程序的代码过程如下：

|  |
| --- |
| if(pid == -1){  perror("fork");  exit(EXIT\_FAILURE);  }  else if(pid == 0){  char \*arg1 = argv[1];  if(argc == 3){  char \*arg2 = argv[2];  execl(argv[1], arg1, arg2, NULL);}  else if(argc == 2){  execl(argv[1], arg1, NULL);  }  }  else{  wait(NULL);  gettimeofday(&end, NULL);  printtime(start, end);  return 0;  } |

**(2)父进程与子进程同步**

调用了Linux提供的API wait();

wait()函数阻塞等待，直到调用进程的任意一个子进程终止运行。调用成功后返回终止子进程的PID码并将状态的说明放入STAT\_LOC参数指明的地址，如果发生错误则返回-1。

|  |
| --- |
| extern \_\_pid\_t wait(\_\_WAIT\_STATUS \_\_stat\_loc); |

**(3）获得子程序运行时间**

获得子程序运行时间调用API为gettimeofday();

在程序开始需要定义

|  |
| --- |
| struct timeval start, end; |

其中timeval 的具体结构如下图所示：

|  |
| --- |
| struct timeval  {      \_\_time\_t tv\_sec;        /\* 秒  \*/      \_\_suseconds\_t tv\_usec;    /\* 微秒  \*/  }; |

在创建子进程的时候，需要得到当前的系统时间：

|  |
| --- |
| gettimeofday(&start, NULL); |

Gettimeofday()的函数API具体定义如下:

|  |
| --- |
| extern int gettimeofday(struct timeval \*\_\_restrict \_\_tv,      \_\_timezone\_ptr\_t \_\_tz); |

在子进程运行结束的时候，同样给end 变量赋值，得到系统的运行时间即可。

使用自定义的printtime()计算时间，定义如下:

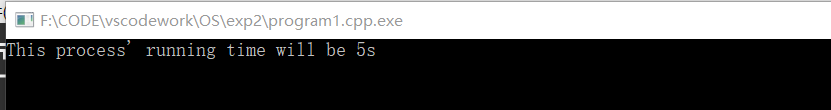
|  |
| --- |
| void printtime(struct timeval start, struct timeval end){  long time\_sec = end.tv\_sec - start.tv\_sec;  long time\_usec = end.tv\_usec - start.tv\_usec;  if(time\_usec < 0){  time\_usec += 1000000;  time\_sec -= 1;  }  long h = time\_sec / 3600;  long m = (time\_sec - h \* 3600) / 60;  long s = time\_sec - h \* 3600 - m \* 60;  long ms = time\_usec / 1000;  long us = time\_usec - ms \* 1000;  printf("The child process' running time is %ldh %ldm %lds %ldms %ldus \n", h, m, s, ms, us);  } |

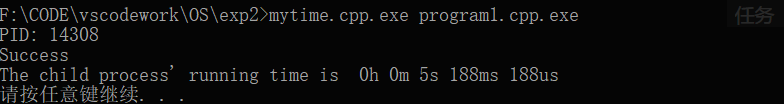
**（4）子程序program1,program2以及主程序mytime 附于实验报告后**

**五、实验结果和分析**

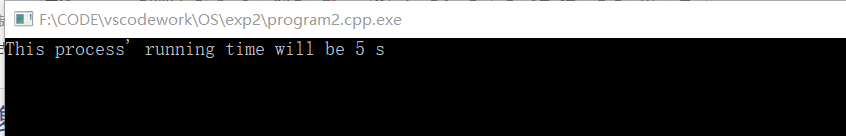
**1.windows 的实验结果**

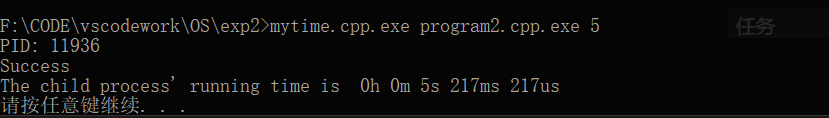
**（1）调用形式形如：mytime program1**





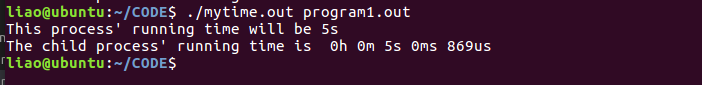
**（2）调用形式形如：mytime program2 t**





**2.Linux 的实验结果**

**（1）调用形式形如：mytime program1**



**（2）调用形式形如：mytime program2 t**



**六、讨论、心得**

**1.心得体会**

完成这部分实验，主要有两个收获：

（1）前两个星期学操作系统这部分课程基本没有进入状态，课后复习学的一些API没有经过实践环节，总是记不住。这个小实验对window与Linux 下的进程控制的API进程了一定的运用，使得我初步了解了整个课程的实验形式和掌握了两种系统下的创建进程，计算进程的运行时间以及同步子进程的方式。

本实验还涉及到一些不同程序之间传递参数的运行，这部分花了不少时间，并且刚开始做实验的时候没有理解好实验的要求，误认为t直接传递到了子程序。

（2）通过这个实验，进一步对Linux系统中的编译代码的原理与方式有了一定的了解，比如编译，连接的过程。

**2.不足与反思**

在Ubuntu中，本来需要学习Vim的方式来进行编写与运行程序，但是由于时间紧张，没有投入时间去进行掌握，只是采用了VisualStudioCode的编译环境对代码进程了一键化的编译和运行过程。虽然这样要方便一些，但是妨碍了对于Linxu系统的进一步学习和适应。

**七、附录 实验代码**

**（1）Windows 系统代码**

* **mytime.cpp**

|  |
| --- |
| #include <Windows.h>  #include <iostream>  #include <stdlib.h>  #include <string.h>  #include <stdio.h>  #include <tchar.h>  void CreateArgv(TCHAR t1[], TCHAR t2[]){  int i = 0;  while(t1[i] != '\0'){  i ++;  }  t1[i] = ' ';  i++;  int j = 0;  while(t2[j] != '\0'){  t1[i++] = t2[j++];  }  t1[i] = '\0';  }    void printtime(LARGE\_INTEGER starttime, LARGE\_INTEGER endtime, double quadpart){  double elapsed = (endtime.QuadPart - starttime.QuadPart) / quadpart;      int h = elapsed / 3600;      int m = (elapsed - h \* 3600) / 60;      int s = elapsed - h \* 3600 - m \* 60;      int ms = (elapsed - h \* 3600 - m \* 60 - s) \* 1000;      int us = (elapsed - h \* 3600 - m \* 60 - s - ms / 1000) \* 1000;      printf("The child process' running time is %dh %dm %ds %dms %dus\n", h, m, s, ms, us);  }  //程序目标：创建新进程并在新进程中显示当前时间  int main(int argc,TCHAR\* argv[]) {      //printf("%s\n",argv[0]);      //printf("%s\n",argv[1]);      // 定义时间参数  LARGE\_INTEGER timeStart, timeEnd;  LARGE\_INTEGER frequency;  QueryPerformanceFrequency(&frequency);  double quadpart = (double)frequency.QuadPart;        //显示进程位置      printf("PID: %d\n", GetCurrentProcessId());        //获取用于当前可执行文件的文件名      TCHAR szFilename[MAX\_PATH];      //GetModuleFileName(NULL, szFilename, MAX\_PATH);        //创建子进程命令行的格式化，获得应用程序的EXE文件名和克隆进程的ID值      TCHAR szCmdLine[MAX\_PATH];      //sprintf((char \*)szCmdLine, "\"%s\"%d", szFilename, nCloneID);      //用于子进程的STARTUPINFO结构      //reinterpret\_cast是C++里的强制类型转换符，只是修改了操作数类型，但仅是重新解释了给出的对象的比特模型而没有进行二进制转换。      STARTUPINFO si;      ZeroMemory(reinterpret\_cast<void \*>(&si), sizeof(si));      si.cb = sizeof(si);      //说明一个用于记录子进程的相关信息的结构变量      PROCESS\_INFORMATION pi;      // typedef struct \_PROCESS\_INFORMATION {      //  HANDLE hProcess;      //  HANDLE hThread;      //  DWORD dwProcessId;      //  DWORD dwThreadId;      // } PROCESS\_INFORMATION      //利用相同的可执行文件和命令行创建进程的符号      //由于mytime有两种调用方式，所以，需要判定参数的个数，实现不同的调用      if(argc == 3) CreateArgv(argv[1], argv[2]);      QueryPerformanceCounter(&timeStart);      BOOL bCreateOK = CreateProcess(          NULL,                                //可执行的应用程序名称          argv[1],                                //指定创建一个子进标识，可以理解为子进程传入参数          NULL,                                   //缺省的进程安全性          NULL,                                   //缺省的线程安全性          FALSE,                                  //不继承打开文件的句柄          CREATE\_NEW\_CONSOLE,                     //使用新的控制台          NULL,                                   //新的环境          NULL,                                   //当前目录          &si,                                    //启动信息          &pi);                                   //返回进程和线程信息        //运行结束，关闭进程和其线程的句柄      // if (bCreateOK) {      //  CloseHandle(pi.hProcess); //关闭进程      //  CloseHandle(pi.hThread); //关闭线程句柄      // }        if(bCreateOK){          printf("Success\n");          WaitForSingleObject(pi.hProcess, INFINITE);          QueryPerformanceCounter(&timeEnd);          printtime(timeStart, timeEnd, quadpart);          }        else printf("Fail\n");  system("pause");      return 0;  } |

* **program1.cpp**

|  |
| --- |
| #include <Windows.h>  #include <iostream>  #include <stdio.h>  #include <stdlib.h>  #include <conio.h>    int main(){      printf("This process' running time will be 5s\n");      Sleep(5000);      return 0;  } |

* **program2.cpp**

|  |
| --- |
| #include <Windows.h>  #include <iostream>  #include <stdio.h>  #include <stdlib.h>  #include <conio.h>    int main(int argc, char \*argv[]){      int t = atoi(argv[1]);      printf("This process' running time will be %d s\n", t);        int truetime = t \* 1000;      Sleep(truetime);        return 0;  } |

**(2) Linux系统代码**

* **mytime.cpp**

|  |
| --- |
| **#include<cstdlib>**  **#include<cstdio>**  **#include<unistd.h>**  **#include<sys/wait.h>**  **#include<sys/types.h>**  **#include<sys/time.h>**  **using namespace std;**  **void printtime(struct timeval start, struct timeval end){**  **long time\_sec = end.tv\_sec - start.tv\_sec;**  **long time\_usec = end.tv\_usec - start.tv\_usec;**  **if(time\_usec < 0){**  **time\_usec += 1000000;**  **time\_sec -= 1;**  **}**  **long h = time\_sec / 3600;**  **long m = (time\_sec - h \* 3600) / 60;**  **long s = time\_sec - h \* 3600 - m \* 60;**  **long ms = time\_usec / 1000;**  **long us = time\_usec - ms \* 1000;**  **printf("The child process' running time is %ldh %ldm %lds %ldms %ldus \n", h, m, s, ms, us);**  **}**  **int main(int argc, char \*argv[]){**  **struct timeval start, end;**  **pid\_t pid = fork();**  **gettimeofday(&start, NULL);**  **if(pid == -1){**  **perror("fork");**  **exit(EXIT\_FAILURE);**  **}**  **else if(pid == 0){**  **char \*arg1 = argv[1];**  **if(argc == 3){**  **char \*arg2 = argv[2];**  **execl(argv[1], arg1, arg2, NULL);}**  **else if(argc == 2){**  **execl(argv[1], arg1, NULL);**  **}**  **}**  **else{**  **wait(NULL);**  **gettimeofday(&end, NULL);**  **printtime(start, end);**  **return 0;**  **}**  **}** |

* **program1.cpp**

|  |
| --- |
| **#include<stdio.h>**  **#include<stdlib.h>**  **#include<unistd.h>**  **#include<sys/time.h>**  **using namespace std;**  **int main(){**  **printf("This process' running time will be 5s\n");**  **sleep(5);**  **return 0;**  **}** |

* **program2.cpp**

|  |
| --- |
| **#include<stdio.h>**  **#include<stdlib.h>**  **#include<unistd.h>**  **#include<sys/time.h>**  **using namespace std;**  **int main(int argc, char \*argv[]){**  **int t = atoi(argv[1]);**    **printf("This process' running time will be %ds\n", t);**  **sleep(t);**  **}** |