

操作系统课程设计实验报告

实验名称: 进程控制
姓名/学号: 郭艺璇/1120161744

一、实验目的

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。

二、实验内容

在 Windows 下实现:

- ① 使用 `CreateProcess()` 来创建进程。
- ② 使用 `WaitForSingleObject()` 在 “mytime” 命令和新创建的进程之间同步。
- ③ 调用 `GetSystemTime()` 来获取时间

在 Linux 下实现:

- ① 使用 `fork()/execv()` 来创建进程运行程序
- ② 使用 `wait()` 等待新创建的进程结束
- ③ 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。`Program1` 可以为自己写的程序，也可以是系统里的应用程序。

```
$ 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

——	名称	版本
Windows 操作系统	Windows 10	专业版
Windows IDE	VS 2017 professional	15.7.3
Linux 操作系统	Ubuntu 虚拟机	内核 4.15.0
Linux 使用的编译器	gcc	7.3.0

四、实验步骤设计与实现

在 Windows 下：

- ① 首先创建主进程 mytime.exe
- ② 然后用主进程创建子进程
- ③ 创建成功则此时设置计时起点，开始计时；若失败则返回
- ④ 运行子进程
- ⑤ 子进程运行结束之后设置计时终点，结束计时

在实际编程过程中，先完成 mytime 主程序的基础功能，使其能够创建子进程和打开系统程序；然后再对 mytime 主程序进行功能拓展，使其具有能够给予子进程传参数的功能，从而控制子进程的执行时间。

实现效果：

- (1) 打开自己写的 cmd_test 程序，子程序的功能是打印 9*9 的-1 矩阵

```
命令提示符
C:\Users\Administrator\Desktop\VSWorkshop\mytime\Debug>mytime.exe cmd_test
mainprogram argv[2]: (null)
subprogram_parameter: cmd_test
Success!
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
0小时 0分 0秒 47毫秒
```

- (2) 打开系统程序 python

```
命令提示符 - mytime.exe python
C:\Users\Administrator\Desktop\VSWorkshop\mytime\Debug>mytime.exe python
mainprogram argv[2]: (null)
subprogram_parameter: python
Success!
Python 3.5.4 (v3.5.4:3f56838, Aug 8 2017, 02:07:06) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- (3) 打开子程序 wastetime, 并执行 1200 毫秒

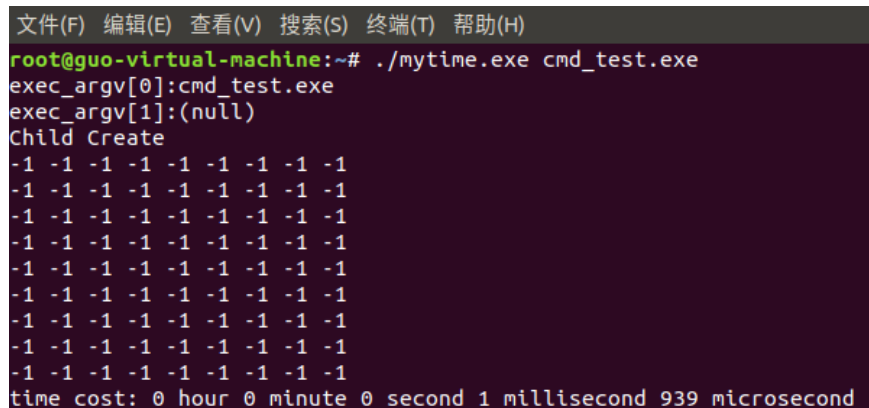
```
命令提示符
C:\Users\Administrator\Desktop\VSWorkshop\mytime\Debug>mytime.exe wastetime 1200
mainprogram argv[2]: 1200
subprogram_parameter: wastetime 1200
Success!
0小时 0分 1秒 250毫秒
```

在 Linux 下：

- ① 首先创建主进程 mytime.exe
- ② 然后用主进程创建子进程
- ③ 创建成功则此时设置计时起点，开始计时；若失败则返回
- ④ 运行子进程
- ⑤ 子进程运行结束之后设置计时终点，结束计时

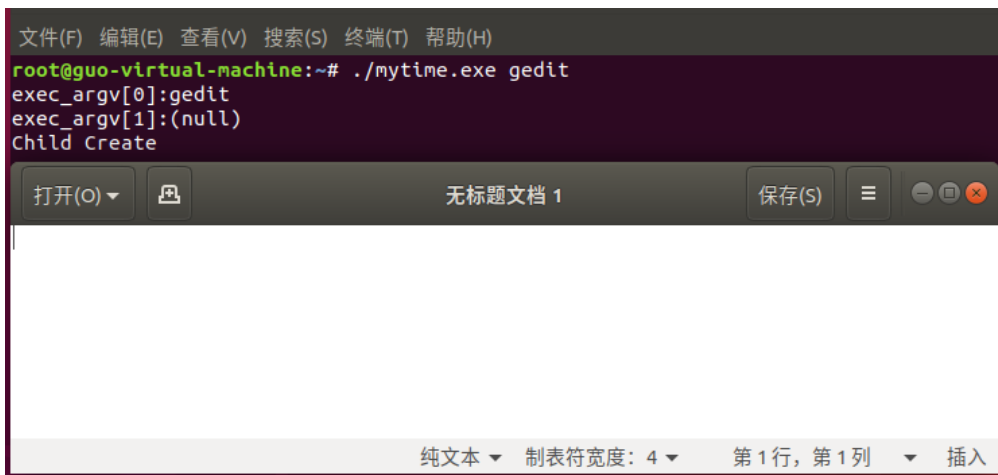
实现效果：

- (1) 打开自己写的程序 cmd_test（功能是打印 9*9 的 -1 矩阵）



```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
root@guo-virtual-machine:~# ./mytime.exe cmd_test.exe
exec_argv[0]:cmd_test.exe
exec_argv[1]:(null)
Child Create
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
time cost: 0 hour 0 minute 0 second 1 millisecond 939 microsecond
```

- (2) 打开系统程序文本编辑器 gedit



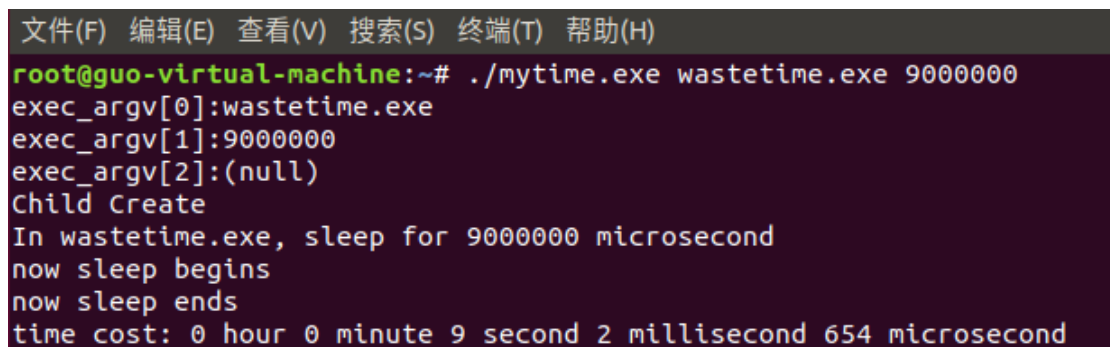
```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
root@guo-virtual-machine:~# ./mytime.exe gedit
exec_argv[0]:gedit
exec_argv[1]:(null)
Child Create
```

无标题文档 1

打开(O) 保存(S)

纯文本 制表符宽度: 4 第 1 行, 第 1 列 插入

- (3) 打开 wastetime，运行 9000000 微秒（9 秒）



```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
root@guo-virtual-machine:~# ./mytime.exe wastetime.exe 9000000
exec_argv[0]:wastetime.exe
exec_argv[1]:9000000
exec_argv[2]:(null)
Child Create
In wastetime.exe, sleep for 9000000 microsecond
now sleep begins
now sleep ends
time cost: 0 hour 0 minute 9 second 2 millisecond 654 microsecond
```

五、实验代码

Windows:

```
#include "stdafx.h"
#include <iostream>
#include <windows.h>
using namespace std;

//打印时间函数
void ShowTime(SYSTEMTIME start, SYSTEMTIME end);

int main(int argc, char *argv[])
{
    SYSTEMTIME start, end;

    //启动设置
    STARTUPINFO si;
    memset(&si, 0, sizeof(STARTUPINFO));
    si.cb = sizeof(STARTUPINFO);
    si.dwFlags = STARTF_USESHOWWINDOW;
    si.wShowWindow = SW_SHOW;

    //拼接字符串给子进程传参数
    char subprogram_parameter[100] = { "\\0" };
    strcat_s(subprogram_parameter,
sizeof(subprogram_parameter), argv[1]);

    if (argv[2] != NULL)
    {
        strcat_s(subprogram_parameter,
sizeof(subprogram_parameter), " ");
        strcat_s(subprogram_parameter,
sizeof(subprogram_parameter), argv[2]);
    }

    //创建进程 pi
    PROCESS_INFORMATION pi;

    //if (!CreateProcess(NULL, argv[1], NULL, NULL, TRUE, 0,
NULL, NULL, &si, &pi))
```

```

    if (!CreateProcess(NULL, subprogram_parameter, NULL, NULL,
TRUE, 0, NULL, NULL, &si, &pi))
    {
        cout << "Create Fail!" << endl;
        exit(1);
    }
    else
    {
        printf("mainprogram argv[2]: %s\n", argv[2]);
        printf("subprogram_parameter: %s\n",
subprogram_parameter);
        cout << "Success!" << endl;
        GetSystemTime(&start);
    }

    WaitForSingleObject(pi.hProcess, INFINITE);

    //计时终点
    GetSystemTime(&end);

    //打印时间
    ShowTime(start, end);

    return 0;
}

void ShowTime(SYSTEMTIME start, SYSTEMTIME end)
{
    int hour, minutes, seconds, milliseconds;
    milliseconds = end.wMilliseconds - start.wMilliseconds;
    seconds = end.wSecond - start.wSecond;
    minutes = end.wMinute - start.wMinute;
    hour = end.wHour - start.wHour;

    if (milliseconds < 0)
    {
        seconds--;
        milliseconds += 1000;
    }
    if (seconds < 0)
    {
        minutes--;

```

```

        seconds += 60;
    }
    if (minutes < 0)
    {
        hour--;
        minutes += 60;
    }
    printf("%d 小时 %d 分 %d 秒 %d 毫秒\n", hour, minutes,
seconds, milliseconds);
}

```

Linux:

```

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>

void show_time(struct timeval start, struct timeval end);

int main(int argc, char *argv[])
{
    int i;
    struct timeval start;
    struct timeval end;

    if (argc==1)
    {
        printf("Error!\n");
        exit(0);
    }
    if (fork()==0)
    {
        char *exec_argv[4];
        for (i=0; i<argc; i++)
        {
            exec_argv[i]=argv[i+1];
            printf("exec_argv[%d]:%s\n", i,exec_argv[i]);

```

```

    }
    if (argv[2]!=NULL)
    {
        char p[50] = {"\0"};
        strcat(p, argv[1]);
        strcat(p, " ");
        strcat(p, argv[2]);
        exec_argv[0]=p;
    }

    printf("Child Create\n");
    execvp(argv[1],exec_argv);

}
else
{
    gettimeofday( &start, NULL );
    wait(NULL);
    gettimeofday( &end, NULL );

    show_time(start, end);
}
return 0;
}

void show_time(struct timeval start, struct timeval end)
{
    int hour, minute, second, millisecond, microsecond;
    microsecond = 1000000 * ( end.tv_sec - start.tv_sec ) +
end.tv_usec - start.tv_usec;

    millisecond = microsecond / 1000;
    microsecond %= 1000;

    second = millisecond / 1000;
    millisecond %= 1000;

    minute = second / 60;
    second %= 60;

    hour = minute / 60;
    minute %= 60;

```

```
    printf("time cost: %d hour %d minute %d second %d  
millisecond %d microsecond\n", hour, minute, second,  
millisecond, microsecond);  
}
```

六、实验收获与体会

这次实验让我感受到了 Windows 和 linux 系统的差别，各自有各自的魅力。Windows 下，父进程创建子进程需要调用 CreateProcess 函数，创建的子进程的参数由 CreateProcess 函数的第二个参数的字符串传入，所以在使用 CreateProcess 函数之前要对字符串进行预处理。Linux 下也是同样的原理，只是使用的函数不同。在 linux 下创建子进程进程，需要调用 fork, execvp 函数。

在使用 linux 的时候，最初写出来的代码编译时会在终端报一大堆 warning，这是因为主函数中使用了某些函数，但是没有 include 包含这些函数的头文件。加上头文件之后，编译就不会有不好看的東西涌现了。

Linux 下并没有一款用着适合 C 语言的 IDE，这对于我这个用惯了 VS 里面代码自动补全的人来说非常难以接受。代码自动补全，语法检查这些功能虽然用着一时爽，而且也极大地提高了编码效率。但如果我们过于依赖这些功能，对我们的代码能力其实是一种束缚。所以在这次实验中，linux 部分花了我最多的时间。