**操作系统课程设计实验报告**

实验名称： 进程控制

姓名/学号： 朱婧婧/1120161826

1. **实验目的**

学习并掌握在不同操作系统下创建一个新进程的方式。分别在Windows和Linux上设计并实现Unix的“time”命令，使程序mytime能够创建一个独立的进程运行另个一程序program，并记录该程序运行的时间，同时，可以控制另一个程序的运行时间。

1. **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()/vfork /exec()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

1. **实验环境**

表格 1 环境配置情况表

|  |  |
| --- | --- |
| **配置** | **参数** |
| 虚拟机 | Oracle VM VirtualBox6.0.4 |
| 操作系统（Linux） | Ubuntu 18.04.2 LTS |
| 操作系统（Windows） | Windows 10 Professional 64bit |
| gcc（Linux） | 7.3.0 |
| gcc（Windows） | 6.3.0 |

1. **程序设计与实现**

**4.1 Windows实现**

**4.1.1 设计思路**

在mytime.exe程序中，主进程调用CreateProcess()函数创建子进程，并通过命令行参数调用指定的子程序，并传递时间控制参数。在子进程的执行过程中，主进程通过调用WaitForSingleObject()函数等待子进程执行完毕。在创建子进程前和子进程执行完毕后，调用GetSystemTime()函数分别获取系统时间，作差后得到子进程的运行时间。

子进程执行的程序为系统程序或自己编写的program.exe程序，在program.exe中，通过sleep()函数，控制子程序的执行时间。

**4.1.2 数据结构及函数说明**

1、PROCESS\_INFORMATION结构

包含新创建的进程和它的主线程的有关信息，用于创建进程的函数CreateProcess, CreateProcessAsUser等。

typedef struct \_PROCESS\_INFORMATION {

HANDLE hProcess; //新创建进程的句柄

HANDLE hThread; //新创建进程主线程的句柄

DWORD dwProcessId; //进程的ID

DWORD dwThreadId; //线程的ID

} PROCESS\_INFORMATION, \*PPROCESS\_INFORMATION, \*LPPROCESS\_INFORMATION;

2、SYSTEMTIME结构

时间是世界通用时间UTC或当地时间，取决于调用函数。

typedef struct \_SYSTEMTIME {

WORD wYear; //年，有效范围1601-30827年

WORD wMonth; //月

WORD wDayOfWeek; //星期

WORD wDay; //天，有效范围1-31天

WORD wHour; //小时，有效范围0-23时

WORD wMinute; //分钟，有效范围0-59分

WORD wSecond; //秒，有效范围0-59秒

WORD wMilliseconds; //毫秒，有效范围0-999毫秒

} SYSTEMTIME, \*PSYSTEMTIME, \*LPSYSTEMTIME;

3、CreateProcess函数

创建一个进程和它的主线程。

BOOL CreateProcessW(

//要执行模块的名称

LPCWSTR lpApplicationName,

//要执行的命令行参数

LPWSTR lpCommandLine,

//指向SECURITY ATTRIBUTES结构的指针，决定返回的进程句柄是否可以被当前进程的子进程继承，NULL表示不继承

LPSECURITY\_ATTRIBUTES lpProcessAttributes,

//指向SECURITY ATTRIBUTES结构的指针，决定返回的线程句柄是否可以被当前进程的子进程继承，NULL表示不继承

LPSECURITY\_ATTRIBUTES lpThreadAttributes,

//True则子进程继承父进程的句柄表

BOOL bInheritHandles,

//控制优先类和进程创建的标志

DWORD dwCreationFlags,

//指向新进程环境块的指针

LPVOID lpEnvironment,

//当前进程的完整路径

LPCWSTR lpCurrentDirectory,

//指向STARTUPINFO或STARTUPINFOEX结构的指针

LPSTARTUPINFOW lpStartupInfo,

//指向PROCESS INFORMATION结构的指针

LPPROCESS\_INFORMATION lpProcessInformation

);

4、WaitForSingleObject函数

等待直到指定对象发出信号或超时。

DWORD WaitForSingleObject(

HANDLE hHandle, //当前对象的句柄

DWORD dwMilliseconds //超时间隔

);

5、GetSystemTime函数

得到当前系统的日期和时间。

void GetSystemTime(

LPSYSTEMTIME lpSystemTime //指向SYSTEMTIME结构的指针，获取当前时间

);

**4.1.3 核心代码**

//命令行参数

char command[maxn];

strcpy(command, argv[1]);

if (argc == 3){

strcat(command, " ");

strcat(command, argv[2]);

}

SYSTEMTIME st, et;

GetSystemTime(&st);

//子进程调用

if (!CreateProcess(NULL, command, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)){

printf("CreateProcess failed (%d).\n", GetLastError());

return 1;

}

//等待子进程退出

WaitForSingleObject(pi.hProcess, INFINITE);

GetSystemTime(&et);

**4.1.4 实验结果**

1、分别编译windows\_mytime.cpp和windows\_program.cpp，生成可执行文件mytime.exe和program.exe

g++ -o mytime windows\_mytime.cpp

g++ -o program windows\_program.cpp

2、在命令行中运行mytime.exe，调用program.exe，运行结果如图1所示。

mytime.exe program.exe

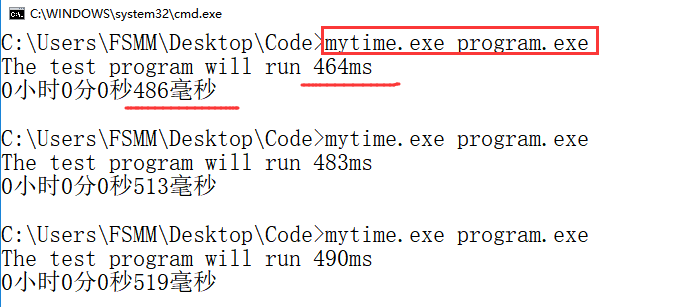


图 1 windows无时间参数运行结果

3、调用系统应用程序，运行结果如图2所示。

mytime.exe Typora.exe

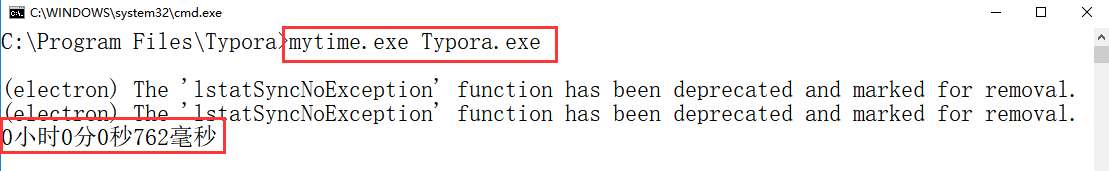


图 2 windows系统应用运行结果

4、指定时间参数，控制program.exe的运行时间，运行结果如图3所示。

mytime.exe program.exe 1500

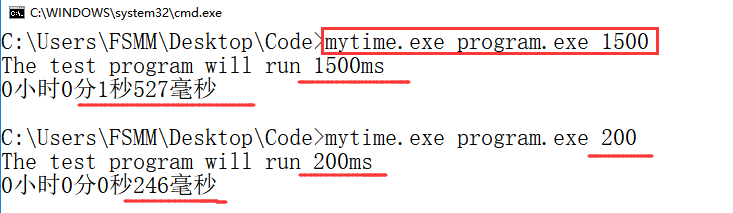


图 3 windows指定时间参数运行结果

**4.1.5 结果分析**

实验结果显示，mytime计时得出的运行时间比子程序的暂停时间略长，经分析，应该是切换进程和调用函数的开销，导致实际运行时间比预估值长。

**4.2 Linux实现**

**4.2.1 设计思路**

在mytime.exe程序中，主进程调用fork()函数创建子进程，并通过execv()函数将子进程替换为新的进程映像，执行指定的子程序，并传递时间控制参数。在子进程的执行过程中，主进程通过调用wait()函数查询子进程的退出状态，等待子进程执行。在创建子进程前和子进程执行完毕后，调用gettimeofday()函数分别获取系统时间，作差后得到子进程的运行时间。

子进程执行的程序为系统程序或自己编写的program.exe程序。在Linux中，sleep()函数的单位是秒，usleep()函数的单位是微秒，在program.exe中，使用usleep()函数，经过转换使时间以毫秒为单位，控制子程序的执行时间。

**4.2.2 数据结构及函数说明**

1、struct timeval结构

struct timeval {

time\_t tv\_sec; //秒

suseconds\_t tv\_usec; //毫秒

};

2、fork函数

创建子进程。系统将创建一个与当前进程相同的新进程，原有进程称作父进程，新创建的进程称作子进程，子进程是父进程的拷贝，获得父进程相同的数据。fork调用一次，返回两次，在父进程中返回子进程的PID，在子进程中返回0，如果调用失败，则返回-1。

pid\_t fork(void);

3、exec函数族

exec名下是由多个关联函数组成的一个完整系列。当进程调用 exec 族函数时，该进程的用户空间代码和数据完全被新程序替换，从新程序的起始处开始执行。调用 exec 族函数并不创建新进程，所以调用 exec 族函数前后该进程的 PID 并不改变。

int execl(const char \*path, const char \*arg, .../\* (char \*) NULL \*/);

int execlp(const char \*file, const char \*arg, .../\* (char \*) NULL \*/);

int execle(const char \*path, const char \*arg, .../\*,(char \*)NULL, char

\* const envp[] \*/);

int execv(const char \*path, char \*const argv[]);

int execvp(const char \*file, char \*const argv[]);

int execvpe(const char \*file, char \*const argv[], char \*const envp[]);

4、wait函数

等待进程改变状态。如果进程子进程正常结束，返回一个非零值；子进程因为捕获信号而终止，返回非零值；如果进程被暂停，返回一个非零值。

pid\_t wait(int \*wstatus);

5、gettimeofday函数

得到当前系统时间，精确到微秒级。

int gettimeofday(struct timeval \*tv, struct timezone \*tz);

**4.2.3 核心代码**

//获取时间

struct timeval st, et;

gettimeofday(&st, NULL);

//创建子进程

pid\_t childpid = fork();

if (childpid == -1){

printf("Fork creatProcess failed.\n");

return -1;

}

//子进程参数调用

if (childpid == 0){

if (execv(argv[1], &argv[1]) < 0){

if (execvp(argv[1], &argv[1]) < 0){

printf("exec %s failed.\n", argv[1]);

return 1;

}

}

exit(0);

}

else{

//等待子进程退出

wait(NULL);

gettimeofday(&et, NULL);

}

**4.2.4 实验结果**

1、分别编译linux\_mytime.cpp和linux\_program.cpp，生成可执行文件mytime和program

g++ -o mytime linux\_mytime.cpp

g++ -o program linux\_program.cpp

2、在命令行中运行mytime，调用program，运行结果如图4所示。

./mytime program

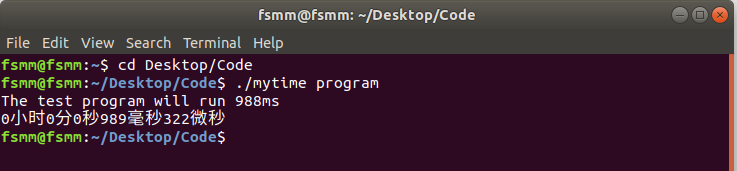


图 4 linux无时间参数运行结果

3、调用系统应用程序，运行结果如图5所示。

./mytime ping

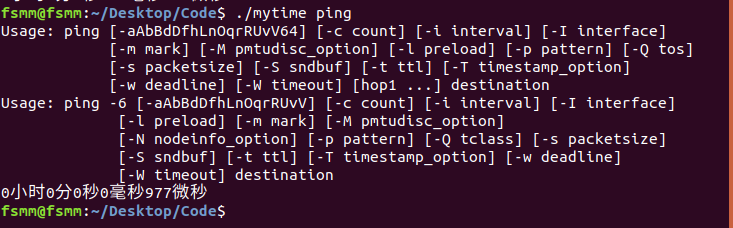


图 5 linux系统应用运行结果

4、指定时间参数，控制program.exe的运行时间，运行结果如图6所示。

./mytime program 1500

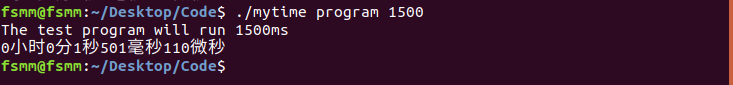


图 6 linux指定时间参数运行结果

**4.2.5 结果分析**

实验结果显示，在linux系统下，mytime计时得出的运行时间同样比子程序的运行时间长，应该也是切换进程和调用函数的开销，但通过实验可以看出，linux下的计时效果比windows环境下好一些，误差更小。

1. **实验收获与体会**

通过本次实验，我学会了如何在windows环境下和linux环境下创建新进程，运行子程序，同时学习了如何使用在两种不同环境下的系统计时工具。多进程多线程运行是计算机发展过程中产生的，充分利用系统资源，提高系统并行性和程序的运行效率，掌握并有效运用多进程多线程编程，是我们必须掌握的一项基本技能。

在实验过程中，虽然有教材的指导，但依旧遇到了很多困难，通过查阅资料得以解决，总结后发现大多是对函数参数记忆不深、理解不够，书本上的学习还远远不够，只有通过实验，才能加强巩固，“纸上得来终觉浅，绝知此事要躬行”。

1. **附录**

**1、windows\_mytime.cpp**

|  |
| --- |
| #include <windows.h>  #include <cstdio>  #include <cstring>  const int maxn = 100005;  int main(int argc, char \*argv[])  {  STARTUPINFO si;  PROCESS\_INFORMATION pi;  ZeroMemory(&si, sizeof(si));  si.cb = sizeof(si);  ZeroMemory(&pi, sizeof(pi));  if (argc < 2){  printf("Usage: %s [cmdline]\n", argv[0]);  return 0;  }  //命令行参数  char command[maxn];  strcpy(command, argv[1]);  if (argc == 3){  strcat(command, " ");  strcat(command, argv[2]);  }    SYSTEMTIME st, et;  GetSystemTime(&st);  //子进程开始  if (!CreateProcess(NULL, // No module name (use command line)  command, //command line  NULL, // Process handle not inheritable  NULL, // Thread handle not inheritable  FALSE, // Set handle inheritance to FALSE  0, // No creation flags  NULL, // Use parent's environment block  NULL, // Use parent's starting directory  &si, // Pointer to STARTUPINFO structure  &pi) // Pointer to PROCESS\_INFORMATION structure  ){  printf("CreateProcess failed (%d).\n", GetLastError());  return 1;  }  //等待子进程结束  WaitForSingleObject(pi.hProcess, INFINITE);  GetSystemTime(&et);  int stime = st.wMilliseconds + st.wSecond \* 1000 + st.wMinute \* 60000 + st.wHour \* 3600000;  int etime = et.wMilliseconds + et.wSecond \* 1000 + et.wMinute \* 60000 + et.wHour \* 3600000;  int usetime = etime - stime;  int hour = usetime / 3600000; usetime %= 3600000;  int minute = usetime / 60000; usetime %= 60000;  int second = usetime / 1000; usetime %= 1000;  int millisecond = usetime;  printf("%d小时%d分%d秒%d毫秒\n", hour, minute, second, millisecond);  //关闭进程和线程  CloseHandle(pi.hProcess);  CloseHandle(pi.hThread);  return 0;  } |

**2、windows\_program.cpp**

|  |
| --- |
| #include <cstdio>  #include <ctime>  #include <Windows.h>  int main(int argc, char \*argv[])  {  int t = 0;  if (argc > 1){  sscanf(argv[1], "%d", &t);  }  else{  srand(time(NULL));  t = rand() % 1000;  }  printf("The test program will run %dms\n", t);      Sleep(t);  return 0;  } |

**3、linux\_mytime.cpp**

|  |
| --- |
| #include <cstdio>  #include <sys/types.h>  #include <sys/time.h>  #include <unistd.h>  #include <stdlib.h>  #include <sys/wait.h>  int main(int argc, char \*argv[])  {  if (argc < 2){  printf("Usage: %s [cmdline]\n", argv[0]);  return 0;  }  //获取时间  struct timeval st, et;  gettimeofday(&st, NULL);  //创建子进程  pid\_t childpid = fork();  if (childpid == -1){  printf("Fork creatProcess failed.\n");  return -1;  }  //调用指定程序  if (childpid == 0){  if (execv(argv[1], &argv[1]) < 0){       if (execvp(argv[1], &argv[1]) < 0){  printf("exec %s failed.\n", argv[1]);  return 1;  }  }  exit(0);  }  else{  //等待子进程结束  wait(NULL);  gettimeofday(&et, NULL);  int usetime = (et.tv\_sec \* 1000000 + et.tv\_usec) - (st.tv\_sec \* 1000000 + st.tv\_usec);  int hour = usetime / 3600000000; usetime %= 3600000000;  int minute = usetime / 60000000; usetime %= 60000000;  int second = usetime / 1000000; usetime %= 1000000;  int millisecond = usetime / 1000; usetime %= 1000;  int haosecond = usetime;  printf("%d小时%d分%d秒%d毫秒%d微秒\n", hour, minute, second, millisecond, haosecond);  }  return 0;  } |

**4、linux\_program.cpp**

|  |
| --- |
| #include <cstdio>  #include <ctime>  #include <unistd.h>  #include <stdlib.h>  int main(int argc, char \*argv[])  {  int t = 0;  if (argc > 1){  sscanf(argv[1], "%d", &t);  }  else{  srand(time(NULL));  t = rand() % 1000;  }  printf("The test program will run %dms\n", t);  usleep(t\*1000);  return 0;  } |