实 验 十二　键盘钩子程序

12.1　实验目的

Windows 平台基于消息机制，消息在分配给应用程序前可利用钩子机制对其处理，本实验介绍使用键盘钩子函数的方法。

12.2　钩子函数

12.2.1　钩子机制介绍

Windows 平台是基于消息驱动的，消息经历产生传输和被目标程序解析的过程。用户的输入操作产生键盘或鼠标消息，窗体的变化也产生消息，程序根据消息的值完成指定功能。例如一个窗口变为活动窗口时，它收到来自 Windows 系统的 WM_ACTIVATE 消息，该消息的编号为 6。对于窗口资源诸如 Open、Activate、MouseDown、Resize 等事件都对应了消息值和处理程序。

消息先存放于系统消息队列中，等待系统分配传送到目标窗体程序。Windows 系统允许在消息分配给应用程前附加一种回调函数，它读取系统消息队列中消息值，并在消息被传送到目标窗体之前执行某些任务，例如监控系统事件或者修改原有的消息值，这种函数即称为钩子函数 (HOOK)。钩子函数根据其作用范围可分为两类，一是应用程序自身的钩子函数，只用来监视程序自己的消息队列，还有一种是全局的钩子函数，它能够监视整个 windows 系统的全局消息队列。全局钩子函数钩子程序能截获其它程序的消息，还能对消息进行修改和控制。图12-1描绘了消息机制中的钩子作用原理。

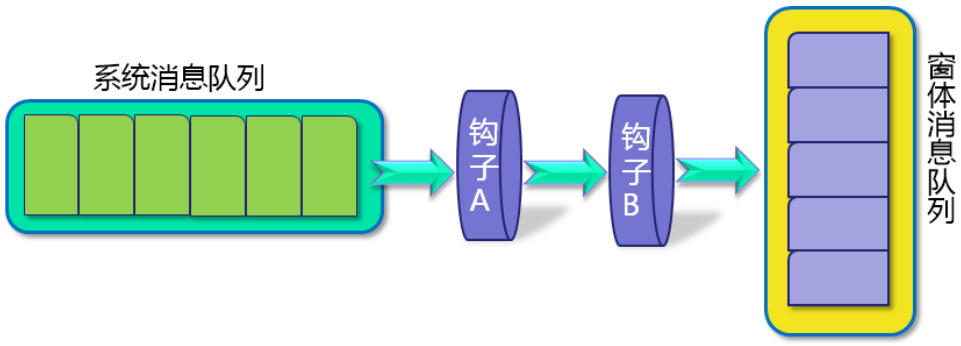

图 12-1 消息机制中的钩子作用原理

根据钩子函数监视的消息类型，Windows 平台可用的钩子类型如表12-1所示：

表 12-1 钩子类型

钩子名称	功能说明
WH_CALLWNDPROC	消息在传送到窗体前。
WH_CALLWNDPROCRET	窗体已经完成消息处理。
WH_CBT	窗体的激活，创建，销毁，最大化，最小化，移动，更改大小。
WH_DEBUG	对钩子函数监视。
WH_FOREGROUNDIDLE	当前端程序变得空闲。
WH_GETMESSAGE	监视消息队列中收到的消息。
WH_JOURNALPLAYBACK	允许进程向系统消息队列中发送消息，它只能是全局 HOOK。
WH_JOURNALRECORD	允许监视和记录系统消息队列消息，它只能是全局 HOOK。
WH_KEYBOARD_LL	允许监视准备放入消息队列的键盘事件。
WH_KEYBOARD	允许监视放入消息队列的键盘消息。
WH_MOUSE_LL	允许监视准备放入消息队列的鼠标事件。
WH_MOUSE	允许监视放入消息队列的鼠标消息。
WH_SHELL	监视 WINDOWS 外壳事件。

钩子函数通过注册的方式关联到特定的消息类型中，当某种消息产生时即调用这个函数。系统允许同一类消息有多个钩子处理函数，每一个 Hook(钩子) 都有一个与之相关联的指针列表，对同一消息处理的钩子函数构成为钩子链。称之为钩子链表，由系统来维护。这个列表的指针指向由应用程序定义的，被 Hook 子程调用的回调函数，也就是该钩子的各个处理子程；当与指定的 Hook 类型关联的消息发生时，系统就把这个消息传递到 Hook 子程，所有钩子函数以回调方式会依次执行，消息会沿链传递。最近安装的钩子函数放在链的开始，而最早安装的钩子放在最后，也就是后加入的先获得控制权。Windows 不要求钩子子程的卸载顺序一定和安装顺序相反。每当有一个钩子被卸载，Windows 便释放其占用的内存，并更新整个 Hook 链表。如果程序安装了钩子，但是尚未卸载钩子就结束了，那么系统会自动为它做卸载钩子的操作。

使用钩子函数分四步：(1) 定义钩子函数 (2) 向系统注册钩子函数 (3) 钩子函数的运行，截获系统消息 (4) 卸载钩子函数。消息沿钩子链传递时，每个钩子函数执行时都会获得消息结构内容，钩子函数对消息可以有下面的处理方式，

1. 不作任何修改，继续传递，比如只是对消息进行记录和监视。
2. 修改消息内容，将新的消息继续传递。
3. 截获消息，不向后传递消息，消息不发送到目标窗体，对消息拦截。

从上面可以看到，由于 windows 系统是基于消息机制的，钩子函数不仅可对系统的消息队列进行监视，还能够修改系统中原有的消息内容。钩子提供的功能是一般的 API 不能比拟的，用户可以利用钩子函数加入自己开发的代码从而丰富程序的功能，或者实现常规方法不能达到的效果。例如我们可以编写程序对其它窗体的状态变化进行截获和处理，对其它程序的键盘或鼠标事件进行截获等。一些应用程序的组合热键功能等也是利用钩子机制来实现的。钩子功能异常强大，对系统中正在运行的程序有较大的影响，还能进一步实现进程注入，多数的木马与病毒程序利用钩子函数实现对机器的非法控制，盗取其它程序信息等。钩子功能增加系统开销，造成机器性能下降，因此只有在必须使用的情况下才使用钩子机制。

12.2.2 使用钩子函数

钩子处理函数是一个回调函数，编写钩子函数需符合规定，它满足钩子函数参数列表固定义，钩子函数的声明必须是下面的样式：

LRESULT CALLBACK HookProc(int nCode, WPARAM wParam, LPARAM lParam);

钩子函数的名称可由用户自定义声明，即 HookProc 可以由用户命名为任意合法的函数名，返回类型固定不变，传入参数的值则由 OS 在调用时填入的，例如键盘的钩子函数在运行时，OS 将会传入键盘值。通过注册添加钩子类型到系统中，注册钩子函数是通过使用 SetWindowsHookEx 函数把自己钩子函数安装到钩子链上。编写钩子函数时一般需要将消息继续传递下去，这是通过在钩子函数最后添加函数 CallNextHookEx，否则极易造成系统问题。因为钩子函数显著降低系统性能，如果不再需要钩子功能，需要卸下钩子的函数，卸下钩子的函数使用的系统调用是 UnhookWindowsHookEx。

12.3 使用键盘钩子的截屏程序

12.3.1 程序功能逻辑

通常的 Windows 窗体程序中，操作系统维护和管理所有窗体对象，确定当前处于激活状态的窗体，操作系统处理用户普通的输入操作时，通常先激活某个窗体再将消息发送到激活窗体。对于载屏程序，界面会影响屏幕像素。要解决接收用户输入同时又不影响当前屏幕显示内容就需要采用特殊方法，钩子函数非常适合这种情况。

本实验的程序与其它应用程序有所区别，Windows 系统默认只将用户输入发送到当前激活窗体，没有界面是不能接收用户输入的。用户每次操作键盘 OS 都生成一个全局的键盘消息，它被放进消息队列中，可以使用 WH_KEYBOARD_LL 钩子类型获取对键盘消息的处理，钩子程序将监视全局键盘事件，即使不属于当前程序的键盘消息也可以获得。

钩子函数 KeyboardHookProc 会在每次系统产生键盘消息时被调用，这个回调函数的 wParam 和 lParam 参数将自动填入合适的值，wParam 值是消息值，如 WM_KEYDOWN 消息，lParam 指向一个内存地址，地址内是具体的按键值，需要进行结构转换；经转换后的按键值在计算机中使用的是键盘的虚拟码，本程序约定用户按下 PrintScreen 键即进行一次的拷幕动作，按下 End 按键程序结束运行。图12-2描绘了键盘钩子线程工作流程。

由于钩子程序与系统运行有很大关系，程序的设计和调试具有一定难度，不能在钩子函数中使用 MessageBox 对话框，因为这将影响消息在钩子链中的传递。程序调试时可借助输出日志的形式获取中间结果，钩子程序稍有出错容易造成系统运行问题。

12.3.2 程序代码

本程序的参考代码如下，首先是常量定义和函数声明：

```
private const int WM_KEYDOWN = 0x100;
private const int WM_KEYUP = 0x101;
private const int WM_SYSKEYDOWN = 0x104;
private const int WM_SYSKEYUP = 0x105;
private const int VK_SNAPSHOT = 0x2C;
```

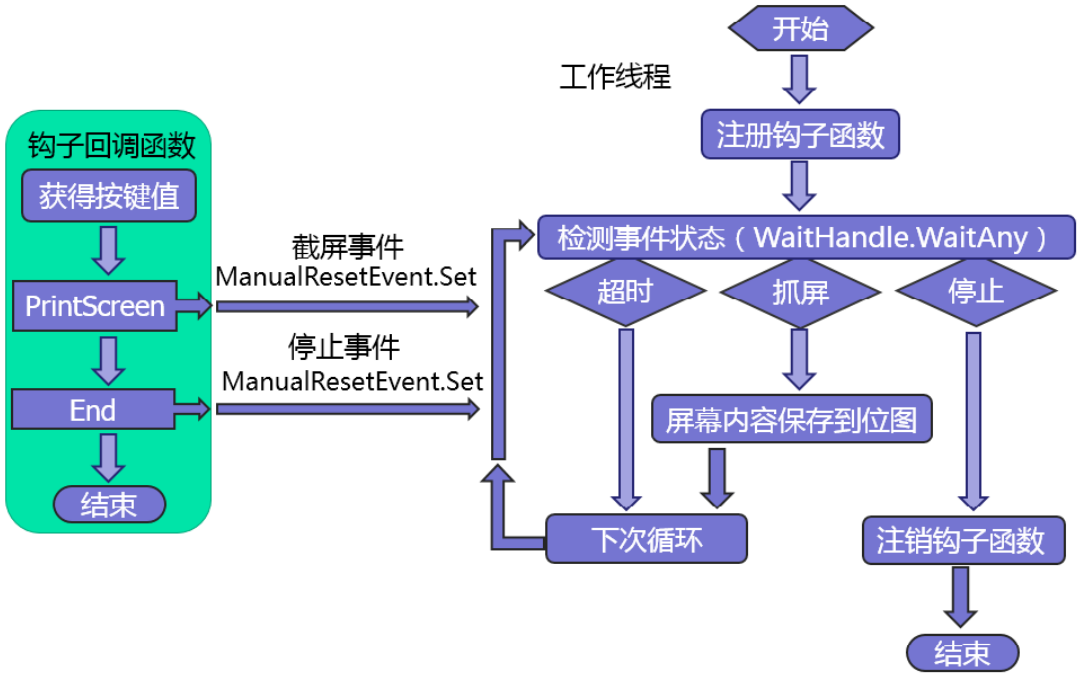

图 12-2 键盘钩子回调函数与工作线程流程

```
//全局的事件
static int hKeyboardHook = 0; //键盘钩子句柄
//鼠标常量
public const int WH_KEYBOARD_LL = 13; //keyboard hook constant
HookProc KeyboardHookProcedure; //声明键盘钩子回调事件.
//声明键盘钩子的封送结构类型
[StructLayout(LayoutKind.Sequential)]
public class KeyboardHookStruct
{
    public int vkCode; //表示一个在 1 到 254 间的虚似键盘码
    public int scanCode; //表示硬件扫描码
    public int flags;
    public int time;
    public int dwExtraInfo;
}
//装置钩子的函数
[DllImport("user32.dll", CharSet = CharSet.Auto, CallingConvention = CallingConvention.StdCall)]
public static extern int SetWindowsHookEx(int idHook, HookProc lpfn, IntPtr hInstance, int threadId);
//卸下钩子的函数
[DllImport("user32.dll", CharSet = CharSet.Auto, CallingConvention = CallingConven-
```

```
tion.StdCall)]
    public static extern bool UnhookWindowsHookEx(int idHook);
    //下一个钩挂的函数
    [DllImport("user32.dll", CharSet = CharSet.Auto, CallingConvention = CallingConvention.StdCall)]
    public static extern int CallNextHookEx(int idHook, int nCode, Int32 wParam, IntPtr lParam);
    [DllImport("user32")]
    public static extern int ToAscii(int uVirtKey, int uScanCode, byte[] lpbKeyState, byte[] lpwTransKey, int fuState);
    [DllImport("user32")]
    public static extern int GetKeyboardState(byte[] pbKeyState);
    [DllImport("kernel32.dll", CharSet = CharSet.Auto, CallingConvention = CallingConvention.StdCall)]
    private static extern IntPtr GetModuleHandle(string lpModuleName);
    [DllImport("kernel32.dll")]
    static extern int GetTickCount();
    public delegate int HookProc(int nCode, Int32 wParam, IntPtr lParam);
    static ManualResetEvent capture_terminate_e;
    static ManualResetEvent capture_this_one_e;
```

安装钩子函数：

```
    private void Start_h()
    {
        //安装键盘钩子
        if (hKeyboardHook == 0)
        {
            KeyboardHookProcedure = new HookProc(KeyboardHookProc);
            Process curProcess = Process.GetCurrentProcess();
            ProcessModule curModule = curProcess.MainModule;
            hKeyboardHook = SetWindowsHookEx(WH_KEYBOARD_LL, KeyboardHookProcedure, GetModuleHandle(curModule.ModuleName), 0);
            if (hKeyboardHook == 0)
            {
                Stop_h();
                throw new Exception("SetWindowsHookEx is failed.");
            }
        }
    }
```

卸下钩子函数：

```
private void Stop_h()
{
    bool retKeyboard = true;
    if (hKeyboardHook != 0)
    {
        retKeyboard = UnhookWindowsHookEx(hKeyboardHook);
        hKeyboardHook = 0;
    }
    //如果卸下钩子失败
    if (!(retKeyboard)) throw new Exception("UnhookWindowsHookEx failed.");
}
    //键盘回调函数
private int KeyboardHookProc(int nCode, Int32 wParam, IntPtr lParam)
{
//lParam 参数只是数据的内存地址
    KeyboardHookStruct MyKeyboardHookStruct =
        (KeyboardHookStruct)Marshal.PtrToStructure(lParam, typeof(KeyboardHookStruct));
    Keys keyData = (Keys)MyKeyboardHookStruct.vkCode;
    if (wParam == WM_KEYDOWN || wParam == WM_KEYDOWN)
    {
        //PrintScreen 负责执行单次捕捉
        if (keyData==Keys.PrintScreen)
        {
            capture_this_one_e.Set();
        }
        //End 负责捕捉的任务结束

        if (keyData == Keys.End)
        {
            capture_terminate_e.Set();
        }
    }
    return CallNextHookEx(hKeyboardHook, nCode, wParam, lParam);
}
```

屏幕抓取线程代码：

```
static void Capture_screen()
{
  int s_wid = Screen.PrimaryScreen.Bounds.Width;
  int s_height = Screen.PrimaryScreen.Bounds.Height;
```

```
  Bitmap b_1 = new Bitmap(s_wid, s_height);
  Graphics g_ = Graphics.FromImage(b_1);
  //图片路径用户须重新设置，才可正常运行
  String init_dir_fn = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
  String dest_fn = null;
  //用事件的方法来抓获图片
  int index = WaitHandle.WaitAny(me_cap, 500);
  while (index != 0)
  {
    if (index == 1)
    {
      dest_fn = init_dir_fn;
      dest_fn += "\\bc\\";
      dest_fn += GetTickCount().ToString();
      dest_fn += "ab.bmp";
      g_.CopyFromScreen(0, 0, 0, 0, new Size(s_wid, s_height));
      b_1.Save(dest_fn, System.Drawing.Imaging.ImageFormat.Bmp);
      capture_this_one_e.Reset();
    }
    index = WaitHandle.WaitAny(me_cap, 500);
  }
  g_.Dispose();
  b_1.Dispose();
}
```

参考的启动工作线程和注册钩子函数的代码：

```
private void button1_Click(object sender, EventArgs e)
{
  Start_h();
  //初始捕获终止事件为未结束
  capture_terminate_e=new ManualResetEvent(false);
  //初始捕获终止状态为未结束
  capture_this_one_e = new ManualResetEvent(false);
  me_cap[0] = capture_terminate_e;
  me_cap[1] = capture_this_one_e;
  //启动捕捉线程
  ThreadStart workStart = new ThreadStart(Capture_screen);
  Thread workThread = new Thread(workStart);
  workThread.IsBackground = true;
  workThread.Start();
}
```

终止钩子的操作：

```
private void button2_Click(object sender, EventArgs e)
{
  Stop_h();
  capture_terminate_e.Set();
}
```

12.3.3 键盘钩子函数的两种类型

SetWindowsHookEx 函数通过使用 WH_KEYBOARD 或者 WH_KEYBOARD_LL 注册键盘钩子函数，功能虽然比较接近，但是回调函数的使用差别很大，表格12-2分别就两种类型的功能及回调函数参数使用进行说明。

表 12-2 两种键盘钩子函数对比

	本地按键消息处理	全局按键消息处理
作用范围	当前进程内	机器上所有按键消息
注册类型	WH_KEYBOARD	WH_KEYBOARD_LL
wParam 参数	键的虚拟码	系统消息常量值
lParam 参数	按键状态位值组合	指向一个 KBDLLHOOKSTRUCT 结构的指针

WH_KEYBOARD_LL 类型可监视系统他部的按键消息包括不属于自己身进程的键盘消息。回调函数 LowLevelKeyboardProc 中 wParam 值为系统消息常量值，取值可以是 WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN, 或者 WM_SYSKEYUP，lParam 值则是指向一个 KBDLLHOOKSTRUCT 结构的指针，该结构中包括按键的虚拟码信息。

注册为 WH_KEYBOARD 类型的钩子只能监视本进程中的键盘消息，回调函数 KeyboardProc 中 wParam 参数值为按键的虚拟码，lParam 为键盘按键状态位值组合。KEY_UP 变量定义为 1 左移 30 位后的整数值，回调函数 KeyboardProc 在系统截获到键盘 WM_KEYUP 或者 WM_KEYDOWN 事件时调用，当 nCode 值为 HC_ACTION 时，wParam and lParam 保存的是键盘按键信息。其中 WPARAM 值为 Virtual-Key 码，lParam 参数是 32 位整数指示按键相关状态信息。lParam 中 0-15 位指示键重复次数，16-23 位指示键扫描码，24 位指示是否为扩展按键，25-28 保留未使用，29 位指示 ALT 键按下状态，30 位指示按键按下状态，31 按键转化状态。图12-3绘制回调函数中 lparam 参数中位的划分。

12.4 实验作业

1. 本实验文档提供了程序的关键代码，请用户自行设计完整应用程序，并调试运行。
2. 编写一个钩子程序，使用热键 (例如 CTRL+F10) 激活特定窗体。

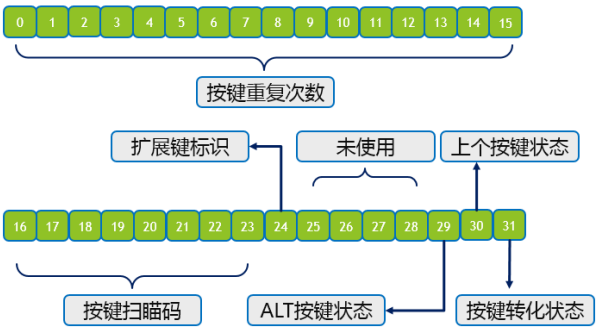

图 12-3 键盘钩子回调函数中 lparam 参数中位划分