Faculty of Economic Sciences, Communication and IT.

Department of Computer Science, Karlstad University

Ali Yanar and Natalie Lidén

Degree Project of 15 credit points

Wote Text Editor

Computer Science

C-uppsats

Date/Term: 12-06-05

Supervisor: Kerstin Andersson

Examiner: Donald Ross

Serial Number: C2012:08

Wote Text Editor

Ali Yanar and Natalie Lidén

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

_____

Ali Yanar

_____

Natalie Lidén

Approved, 20120605

_____

Advisor: Kerstin Andersson

_____

Examiner: Donald Ross

Abstract

Text editors are used by most people today who wish to write some sort of document or notes. The text editor in this project was to have a special function very similar to a keylogger. This function would enable the user to write in the text editor without really setting it as a focused window. When a user is reading and browsing through another document, whilst for instance writing comments to this document, the Wote keylogger will prove very useful, as the user will not be forced to switch between different windows when using them. To reach this goal, the Wote keylogger must be able to fetch the characters to be typed into the Wote document, whilst leaving other appropriate keyboard commands for the other application.

The result is a text editor for Linux, with a keylogger function. However, the keylogger is not completely functional in all environments. The environment used in this project was a Linux virtual machine, where the keylogger was functional except for a few bugs. In the virtual environment, the bugs involved the keylogger reading mouse input and risks of the arrow keys freezing if pressed for a longer time.

SAMMANFATTNING

Textredigerare används av de flesta idag som vill anteckna eller skriva dokument. Textredigeraren i detta projekt skulle ha en speciell funktion som liknar en keylogger. Denna funktion ger användaren möjlighet att skriva i editorn utan att fönstret behöver vara i fokus. När en användare läser och bläddrar i ett annat dokument, samtidigt som han till det nämda dokumentet skriver kommentarer, är Wote-keyloggern mycket användbar, eftersom användaren inte tvingas växla mellan olika fönster när de används. För att uppnå detta mål, måste Wote-keyloggern kunna hämta tecken som ska skrivas ut i Wote-dokumentet, samtidigt som lämpliga tangenbordskommandon hanteras av den andra applikationen.

Resultatet är en texteditor till Linux, med en keylogger-funktion. Men keyloggern är inte fullständigt funktionell i alla miljöer. Den miljö vi har använt är en virtuell maskin med Linux, där keyloggern fungerar, men några buggar kvarstår. I den virtuella miljön, innefattade buggarna att keyloggern läste indata från musen och piltangenterna riskerade att låsa sig om de var nedtryckta för länge.

Acknowledgements

We would like to express our gratitude to Department of Computer Science, Karlstad University for giving us the opportunity to do this project. We would also like to thank researcher Martin Bloom for his advice.

And we would like to thank our academic supervisor Kerstin Andersson for her excellent guidance, constructive inputs and advices.

May, 2012
Ali Yanar and Natalie Lidén

INNEHÅLLSFÖRTECKNING

ILLUSTRATIONSINDEX

1. INTRODUKTION

Vi, Ali Yanar och Natalie Lidén, har fått uppdraget Wote Text Editor av Martin Blom, som är universitetslektor på avdelningen Datavetenskap på Fakulteten för Ekonomi, Kommunikation och IT. Anledningen till att vi valde detta arbete var att vi har intresse för programmering och Wote verkade som ett unikt och spännande projekt.

Projektet Wote Text Editor är en idé av Martin Blom. Önskemålet med den här editorn är att man ska kunna skriva text utan att ha fönstret aktivt och att fönstret ska kunna göras transparent. Detta är för att man ska kunna läsa dokument och till exempel skriva kommentarer till dokumenten samtidigt som man läser, utan att behöva byta mellan fönster eller ha dem bredvid varandra.

Målet är att den färdiga editorn ska kunna ta emot text innan någon annan applikation gör det. All text från tangentbordet ska hamna i editorn istället för i en annan applikation vars fönster är aktivt. Man ska helst också kunna välja om man vill ha den här funktionen aktiverad eller inte.

I sektion 2 kommer vi att ge en kort beskrivning av den tidigare versionen av Wote, och sedan gå över till några delar som har spelat roll i utvecklingen av den version av Wote som vi har arbetat med. Bland dessa delar förekommer keylogging, TTY, kernelmoduler och GTK.

I sektion 3 kommer vår version av Wote-applikationen att beskrivas med tekniska detaljer och olika delar av koden kommer att förklaras. Applikationen är uppdelad i en grafisk del, där GTK-biblioteket används, och en del för keylogger-funktionaliteten. Delarnas samarbete kommer att beskrivas, samt de problem som uppstått under implementationen.

Sektion 4 innehåller en mindre teknisk illustration av hur Wote fungerar. Alla programmets användarfunktioner kommer att förklaras på ett förståeligt sätt. Vi kommer även att gå igenom de erfarenheter vi har fått av arbetet.

Till sist, i sektion 5, tar vi upp de uppfyllda kraven, samt de som inte har uppfyllts. Några utvecklingsmöjligheter och rekommendationer för Wote följer i sektion 5.3.

2. BAKGRUND

Wote började som ett projekt baserat på Martin Bloms idé om en specifik sorts textredigerare, och detta projekt gick ut på skapandet av en Wote texteditor för Windows. Implementeringen av Wote version 1.0 utfördes av Jonas Larsson och Martin Bengtsson[7], två studenter, och resulterade i en nästan helt komplett Wote-applikation enlig Martin Bloms förväntningar.

Den första implementationen av Wote skulle skrivas i C++, men eftersom detta var tidskrävande, skrevs hela applikationen i stället i C# [7]. Programmet använder bland annat biblioteket win32 för att uppfylla de krav om teckenhantering som Wote har.

Det vi hade att arbeta med i det nya Wote-projektet var att göra en version som går att köra i Linux. C# -koden för programmet behövde också översättas till C++ eller C för att bli snabbare.

För att få en allmän bild av vad Wote innebär, så förklaras begreppet textredigerare i sektion 2.1. Hur Windows-versionen ser ut kommer att förklaras i sektion 2.2.1, följt av sektion 2.2.2 med en beskrivning av de delar som spelat roll i utvecklingen av Linux-versionen.

2.1 TEXTREDIGERARE

En textredigerare eller texteditor har nog använts av de flesta som arbetar vid datorer. Denna typ av program används för att skriva och redigera text, men ibland även hantera bilder, tabeller, diagram etc. beroende på hur avancerad texteditorn är. Textredigerare har jämfört med en vanlig skrivmaskin fördelarna att text bland annat kan tas bort, ersättas med ny text, kopieras och klistras in. De flesta textredigare har möjlighet att skriva ut de dokument man har skapat om man har en skrivare kopplad till datorn eller nätverket. Eftersom text kan redigeras med teckenstilar, färger etc. så har olika textredigerare olika sätt att koda dokumentet. En .odt-fil kan till exempel inte öppnas med Note Pad [22], som är en mycket simpel texteditor gjord för .txt-filer. [1]

2.2 WOTE

Projekt Wote innefattar en texteditor med vissa funktioner. Funktionerna ska ge användaren möjlighet att enkelt kunna läsa ett dokument och samtidigt skriva, till exempel kommentarer, i Wote-editorn. Detta innebär att användaren inte ska tvingas växla fram och tillbaka mellan olika fönster, utan ska kunna ha bra kontroll över Wote och en annan applikation samtidigt. Wote-editorn ska därför kunna ligga överst, även då man använder sig av ett annat fönster. Inställningarna i Wote ska även ge en möjlighet för användaren att kunna skriva in text i editorn medan ett annat fönster är aktivt, till exempel möjligheten att kunna skriva i Wote samtidigt som man bläddrar i ett annat dokument med musen. För att Wote fönstret inte ska vara i vägen när det ligger överst, ska man även kunna ändra genomskinligheten på själva Wote fönstret till en nivå som passar.

2.2.1 WINDOWS-VERSION

För Windows finns det en nästan komplett version av Wote, utvecklat av Jonas Larsson och Martin Bengtsson [7]. Applikationen är skriven i programspråket C#, och har de vanliga funktionerna hos en editor, som öppna, spara, starta nytt dokument, sök osv. De specifika Wote-funktionerna är “on top”, “hook”, “steal” och “trackbar”. För att få Wote-fönstret överst, klickar man på “on top”-knappen och för att skriva ut tecken i Wote från tangentbordet klickar man på “hook”. Eftersom “hook” får tecken att kopieras, kan man även välja “steal” som hämtar tecken innan någon annan applikation får dem; det är denna funktion som gör att man kan skriva i Wote, utan att påverka en annan applikation som då är i fokus. Den så kallade “trackbar” är det verktyg man använder för att modifiera transparensen av fönstret. För att undvika att det blir osynligt och tappas bort av användaren, blir det aldrig fullständigt genomskinligt. Man kan även trycka Ctrl + Q för att ändra transparensen till maximum.

Nedan visas två bilder på Bengtssons och Larssons version av Wote, den ena efter man har skrivit text i fönstret (se Illustration 1) och den andra när man har fått fönstret att bli genomskinligt genom att ha använt “trackbar”-verktyget (se Illustration 2) [7].

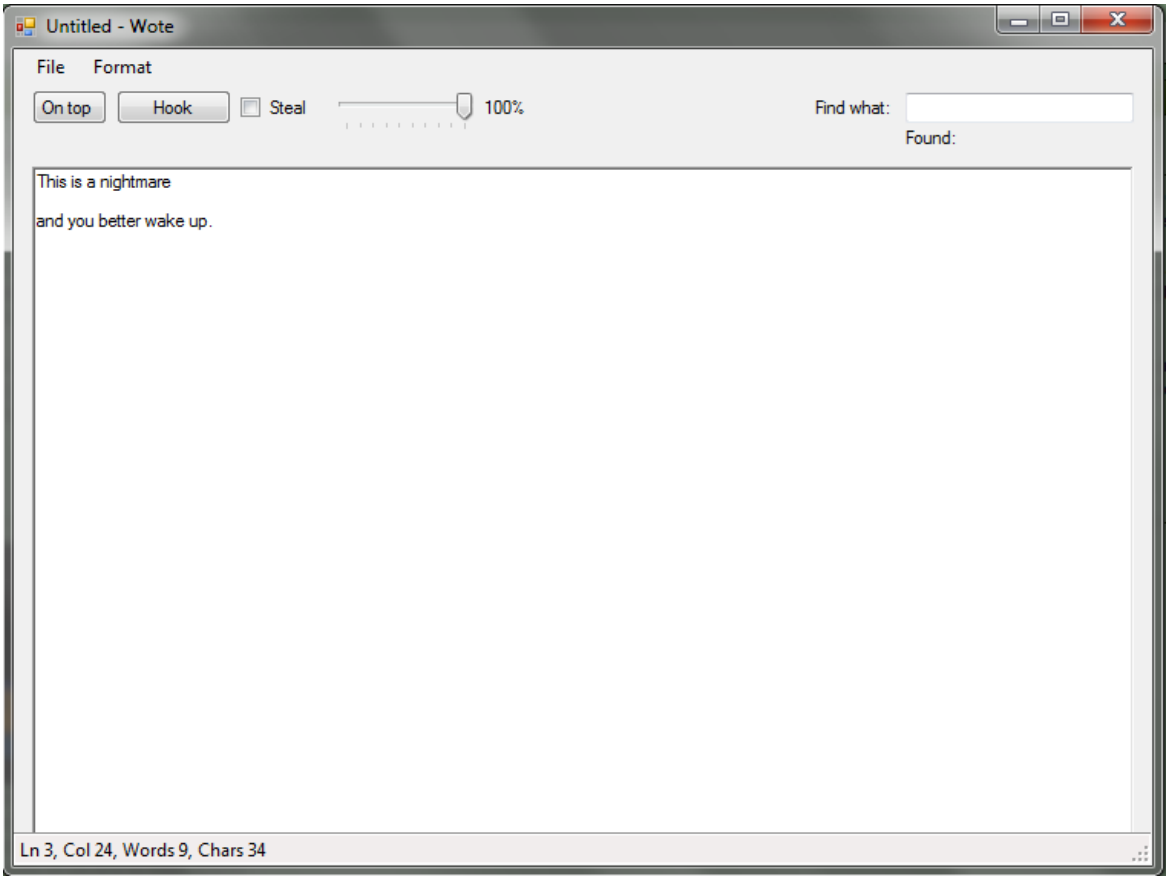

Illustration 1: Windows-versionen av Wote.

Illustration 2: Windows-versionen av Wote, där man har sänkt transparensen till 50%.

2.2.2 LINUX-VERSION

Linux-versionen av Wote ska byggas efter samma krav och önskemål som Windows-versionen, men kommer förhoppningsvis att bli litet snabbare, då denna kommer att skrivas i C++ eller C istället för C#. Versionen för Linux kommer att se mycket annorlunda ut kodmässigt, eftersom de två operativsystemen skiljer sig från varandra med olika bibliotek och så vidare. Wote-applikationens viktigaste funktion är att kunna vara aktiv och ta emot tecken, även då en annan applikation egentligen är den som är i fokus. Eftersom tecknen bör hämtas direkt från tangentbordet, så kommer man att behöva programmera nära hårdvaran eller operativsystemet. Det underlättar därför om man förstår hur Linux och tangentbordet fungerar. De delar som har blivit relevanta eller åtminstone stötts på under arbetet kommer att förklaras. Först kommer själva hanteringen av tecken att beskrivas, som innefattar en teknik som kallas keylogging. Sedan kommer en beskrivning av kernelmoduler och hur man kör dessa i Linux. Slutligen presenteras också en kort introduktion till GTK, ett bibliotek som vi har använt för applikationens grafik.

2.2.2.1 KEYLOGGING

För att kontrollera indata från tangentbordet, kan man använda en teknik som kallas keylogging. Anledningen till att vi ska använda oss av detta, är att vid användning av Wote-applikationen ska det finnas möjlighet för användaren att ställa in hur tecken från tangentbordet ska hanteras, till exempel att tecken går till Wote innan någon annan applikation kan hämta upp dem.

Enligt en artikel [2] om keylogging i Linux, skickar tangentbordet sina indata till en drivrutin för tangentbord. Indatan kallas scancodes och när man trycker ned en tangent kan det skickas en sekvens med upp till sex scancodes. Strömmen av scancodes hanteras av funktionen `handle_scancode()` som konverterar den till keycodes med hjälp av en tabell via funktionen `kbd_translate()`. Dessa keycodes representerar att man har tryckt ned eller släppt en tangent, och varje tangent har en unik keycode, en keycode för att blivit tryckt och en keycode för att blivit släppt.

Scancodes kan även användas till exempel för att meddela om det har uppstått ett fel med kommunikationen med tangenbordet eller som svar på kommando. Det finns även scancodes som

fungerar som prefix, till exempel `e0` och `e1`. Scancode `e0` används för att utöka kombinationer, och `e1` innebär att tangenten bara skickar scancodes när den blivit tryckt och inte skicka scancodes när den är släppt [18].

Kombinationen av indata från tangentbordet konverteras sedan till tangentsymboler [2]. En kombination kan till exempel vara Shift + A. De tecken som genereras lagras i tty-kön som kallas `tty_flip_buffer`. Funktionen `receive_buf()` hämtar tecken från `tty_flip_buffer` och lägger dem i `tty_ldisc`-bufferten. Därifrån kan tecknen sedan läsas till en av de tty som finns i /dev-katalogen, till exempel tty0. Sedan kan `sys_read()` användas för att läsa tecken; denna funktion anropar `read()` som finns definierad i `file_operations`, en struct, och funktionen läser från en tty.

Vad tty används för, kommer att förklaras i sektion 2.2.2.2. I artikeln om keylogging finns en bild som illustrerar hur indatan från tangentbordet bearbetas genom systemet. Se Illustration 3[2].

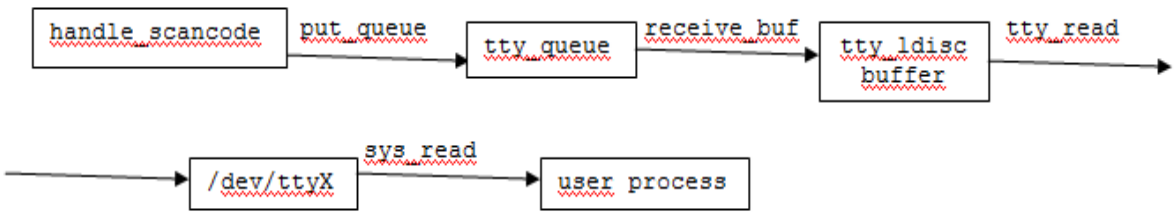

Illustration 3: En enkel illustration om vad som händer från det att indata från tangentbordet kommer in, tills att det läses av en applikation.

2.2.2.2 TTY

I /dev finns filer som tty0, ttyS0, ttyS1 och så vidare, där “tty” är en förkortning av “Teletype”. Teletype är ett märke på en teleprinter. En teleprinter liknar en skrivmaskin som kan skicka data, och dessa maskiner var de första terminalerna som användes. De tty-filer som finns i /dev hör till datorns serieportar, och det brukade vara vanligt att terminaler var kopplade till de portarna. Skriver man till en tty, så skickar man detta till den terminal som finns på motsvarande tty [6].
I Linux finns många olika funktioner som hanterar data via en tty. Bland annat finns `tty_read()`

som läser från en tty och `receive_buf()` som hämtar en buffert med tecken från en viss tty (se Illustration 3). För en programmerare kan en tty och dessa funktioner kommas åt genom att man öppnar en tty-fil och via den får en struct-pekare, enligt

```
struct tty_struct *tty = file->private_data;
```

2.2.2.3 KERNELMODULER

Kernelmoduler körs på kernelnivå i Linux, och är därför integrerade med operativsystemet och kan påverka detta. Förutom kernel space, där modulerna körs, finns även user space, där bland annat vanliga användarapplikationer körs. De två nivåerna brukar kommunicera med varandra genom systemanrop, som exempelvis `sys_read()`.

För att kunna köra en kernelmodul, måste man ha en startpunkt, ungefär som `main()` i ett vanligt c-program, och en utgång [3]. Dessa defineras med `module_init` och `module_exit`, enligt

```
module_init(logger_init);
module_exit(logger_exit);
```

I detta fall kommer programmet att börja köra i funktionen `logger_init()` och avslutas med `logger_exit()`. Den kernelmodul som genereras efter kompilering är en .ko-fil som kan integreras med operativsystemet via kommandot `insmod`. Det första som då körs är init-funktionen, som i fallet ovan är `logger_init()`. För att sedan ta bort modulen, använder man kommandot `rmmod`, och då körs exit-funktionen, till exempel `logger_exit()`, innan programmet terminerar.

För att kunna kompilera en kernelmodul som använder kernelbiblioteken, måste man ha tillgång till rätt källkod och tillhörande header-filer [8]. Header-filerna inkluderas, som till exempel `#include<file.h>`, för att man ska komma åt definierade funktioner och så vidare. Eftersom de header-filer man vanligtvis inkluderar ligger i /usr/include, måste man tala om att man vill använda header-filerna för kernelmoduler istället. Detta gör man genom att i sin makefile skriva

```
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
```

Kommandot `uname -r` används för att få namnet på den version av operativsystemet man har. Själva modulen skapas med

```
obj-m :=    logger.o
```

Exempel på en makefil för kernelmoduler kan ses i appendix A1.

2.2.2.4. GTK

GTK är ett bibliotek som används för att skapa grafiska gränssnitt i bland annat C, C++ och Python och står för GIMP Toolkit [9]. GIMP står för GNU Image Manipulation Program, som är en bildredigerare med öppen källkod [21]. GTK-biblioteket går att använda i olika miljöer, som till exempel Linux där vår Wote-applikation ska utvecklas. För att installera GTK 2.0 i Linux, kör man `apt-get isntall libgtk2.0-dev` som kommando i terminalen.

När man skriver en fönsterapplikation med GTK, inkluderar man header-filen med `#include <gtk/gtk.h>` för att få tillgång till funktioner och de så kallade Widgets. En Widget kan vara ett fönster, en knapp, en textruta och så vidare. Man kan skapa och visa ett fönster med koden

```
GtkWidget *window;
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_widget_show(window);
```

Alla GTK-applikationer behöver ha anropen `gtk_init(&argc, &argv)` och `gtk_main()` [10]. Det första initialiserar GTK och vid `gtk_main()` kommer GTK att vänta på händelser som till exempel att användaren trycker på en knapp. Initialiseringen ska ligga innan anrop till GTK-funktioner och `gtk_main()` efter.

Vill man skapa en händelse i sin GTK-applikation använder man

```
g_signal_connect(window, "delete-event", G_CALLBACK (delete_event),
NULL);
```

Händelsen kommer att aktiveras vid en “delete-event”, alltså då man stänger fönstret.

`G_CALLBACK(delete_event)` avgör vilken funktion händelsen utgör, i detta fall `delete_event()`. Både `window` och `NULL` är parametrar till funktionen `delete_event()`, som visas nedan.

```
static gboolean delete_event(GtkWidget *widget, GdkEvent *event,
gpointer data)
{
   gtk_main_quit();
   return FALSE;
}
```

När `delete_event()` anropas, avslutas programmet med `gtk_main_quit()`. Funktionen returnerar `FALSE`, vilket innebär att GTK kommer att fortsätta med vad det skulle göra vid en "delete_event". Tar man bort anropet till `gtk_main_quit()` och returnerar `TRUE`, kommer applikationen inte att avslutas.

För att kompilera ett GTK-program med till exempel biblioteket för GTK 2.0 kan man använda gcc och skriva i terminalen

```
gcc `pkg-config --cflags --libs gtk+-2.0` hello.c -o hello
```

I exemplet ovan kompilerar vi filen hello.c till en körbar fil hello.

2.3 SAMMANFATTNING

Vi har nu gått igenom vad Projekt Wote innebär och vad keylogging, kernelmoduler och GTK är. Denna information behövde vi för att börja implementationen av Wote till Linux. Det viktigaste var att känna till keylogging och något grafiskt bibliotek, som till exempel GTK. Under vårt arbete stötte vi även på det så kallade tty, men det kommer senare visa sig att det inte blev någon stor del av resultatet.

Mer information om keylogging, kernelmoduler och GTK följer i sektion 3, implementationen. Där kommer vi att gå igenom hur utvecklingen av den nya Wote gick till.

3. IMPLEMENTATION

Språket vi valde att programmera i, när vi skulle skapa en Wote-applikation, var C, eftersom detta var mer bekant för oss än C++. C är känt för att vara svårare än C++ när man ska koda grafiska applikationer, men det såg vi som en spännande utmaning.

Vi kommer nu först att gå igenom lösningen med `receive_buf`, hur vi gick till väga och vilka resultat vi fick. Sedan kommer den mer lyckade lösningen med Uberkey att presenteras. Användargränssnittet för Wote är programmerat i språket C med biblioteket GTK 2.0, och detta arbetet kommer att beskrivas efter de olika keylogger-lösningarna har gåtts igenom.

3.1 KEYLOGGING

För att lösa problemet med att styra indata från tangentbordet till Wote, var vi rekommenderade att använda keylogging. Vi började arbetet med att söka information om keylogging. Därefter valde vi en lösning att försöka implementera. Det första vi testade var att hämta tecken via en funktion i Linux, kallad `receive_buf()`, och detta gjordes på kernelnivå, med en kernelmodul (se sektion 3.1.1). Eftersom tecknen i bufferten kommer från en tty och terminalen, var det svårt att få ihop detta med fönsterapplikationer. Bufferten visade sig också vara krånglig att hantera, för vi var inte säkra på exakt vilka tecken som den innehöll. Lösningen med `receive_buf()` verkade inte leda någonstans, speciellt inte då den koncentrerade sig mycket på tecken i terminalen, så vi övergav detta och gick över till en annan lösning.

Den alternativa lösningen är en färdig keylogger kallad Uberkey, men som inte alltid fungerar som förväntat. För att bygga vidare på den här lösningen, skulle Uberkey förbättras och anpassas till Wote (se sektion 3.1.2).

3.1.1 LÖSNING I: RECEIVE_BUF

Vi började med att testa en `tty_ldisc`-buffert-lösning för keylogging. Via `tty_ldisc` kan man komma åt `receive_buf()`. Som nämnts tidigare (se sektion 2.2.2.1) är `receive_buf()` en funktion som hämtar en `tty_flip_buffer` innehållande de tecken som lästs från en tty. Ett exempel på denna keyloggerlösning finns i artikeln writing-linux-kernel-keylogger.txt [2]. Den går ut på att implementera en ny variant av funktionen `receive_buf()` som finns i `struct tty_ldisc`. För att göra detta skapar man en struct med en filpekare till den tty vars indata man vill komma åt. Genom denna pekare kan man nå en pekare till `tty_struct` och via den komma åt `ldisc.receive_buf`. Koden här är ett exempel från artikeln [2]

```
int fd = open("/dev/tty0", O_RDONLY, 0);
struct file *file = fget(fd);
struct tty_struct *tty = file->private_data;
old_receive_buf = tty->ldisc.receive_buf;
tty->ldisc.receive_buf = new_receive_buf;
```

Enligt koden ovan öppnas en tty med funktionen `open()`, och sedan hämtas en pekare till filen med `fget()`. När man har en pekare till filen kan man komma åt filens `private_data`. För en tty finns en pekare i `private_data`, och denna pekare pekar till en tty struct. Det är via den man kan komma åt `receive_buf`.

Det visade sig, vid vår implementation, att `ldisc.receive_buf` inte kunde hittas. I `tty_struct` finns `ldisc`, men i `tty_ldisc` finns även `ops`, så vi ändrade en rad i koden till
`old_receive_buf = tty->ldisc->ops->receive_buf;`
för att komma åt `receive_buf`.

Det man sedan gör, när man har fått tillgång till `receive_buf`, är att skapa en funktionspekare som håller reda på `receive_buf()`, eftersom man ska ändra på den befintliga pekaren och få den att peka mot den nya implementationen av `receive_buf`. Den nya `receive_buf`-funktionen kommer att anropas istället för den gamla på grund av pekaren. I den nya funktionen kan man till exempel hämta de tecken man vill ha, och man bör även anropa den gamla

receive_buf-funktionen för att operativsystemet ska fortsätta fungera som det ska.

För att kunna använda sig av funktioner som receive_buf(), behövde vi de header-filer som används för kernelmoduler. Man behöver bland annat denna rad

```
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
```

i make-filen för att komma åt rätt header-filer [3].
I koden vi hittade finns ett anrop till funktionen open(). Den hittades inte på kernelnivå och kompilatorn gav därför felmeddelandet “implicit declaration of function ‘open’”.

Funktionen open() används för att öppna en tty, så att man därefter ska kunna få en filpekare med fget(). Eftersom open() inte verkade fungera i kernel space, ersatte vi både open() och fget() med filp_open() som både öppnar filen och returnerar en filpekare. Via till exempel tty0 kan vi komma åt pekaren till en tty struct. För en tty-fil finns pekaren lagrad i private_data.

I fallet av vår keylogger, är det viktigt att i logger_exit(), den funktion som körs innan modulen tas bort, få receive_buf-pekaren att peka till den gamla receive_buf-funktionen igen, eftersom den nya fuktionen försvinner när programmet avslutas. Har funktionspekaren då fortfarande adressen till den nya funktionen, så får vi en “dangling pointer” och operativsystemet riskerar att krascha eller låsa sig.

3.1.1.1 TEST AV BUFFERT

När vi sedan testade att skriva ut de tecken som fanns lagrade i bufferten via new_receive_buf, fick vi ut samma tecken som man hade skrivit i terminalen. All utdata hamnar i en loggfil, eftersom vi använder printk(). Funktionen printk() finns på kernelnivå, och vi använde den därför att printf() inte var tillgänglig. Skrev vi till exempel “welcome to hell” i terminalen, så kom dessa tecken med i loggfilen som man läser med kommandot dmesg. Men det kom även med dubletter av tecken och “skräptecken” från någon annanstans. För att undvika dubletterna, skrev vi ut bufferten varannan gång, men andra onödiga tecken fanns fortfarande kvar. Relevanta tecken

befann sig alltid på index 0 i bufferten, så det var den enda positionen som vi behövde skriva ut. Här är koden för att skriva ut små bokstäver varannan gång från bufferten:

```
static unsigned int bit = 0;
if (!bit && ((cp[0] < 123 && cp[0] > 96) || cp[0] == 32))
printk("%c", cp[0]);
bit = ~bit;
```

Eftersom bufferten bara ska skrivas ut varannan gång, använder vi en statisk variabel för att kontrollera när bufferten ska skrivas ut och när denna del ska hoppas över. Vi använder en statisk variabel för att den ska finnas kvar även när funktionen lämnas för att sedan kontrolleras vid nya anrop. Den sätts alltså till värdet 0 första gången, men kommer att ha kvar andra värden, som den möjligen har tilldelats, vid nya anrop av funktionen.

Om det statiska variabeln bit är satt till 0 och tecknet på buffertens första position har ett värde mellan 96 och 123, skrivs tecknet ut med `printk()`. Det som testas är om tecknet `cp[0]` är en bokstav a till z. Tecknet i `cp[0]` jämförs även med värdet 32, som är värdet för ett mellanslag.
Värdena för varje tecken är så kallade keycodes och följer en ASCII-tabell. ASCII är en av de tabeller som används för att matcha keycodes med tecken.

Efter utskrift av `cp[0]` kommer variabeln bit att inverteras. Alla binära tal inverteras så att nollor blir ettor och ettor blir nollor. Det betyder att när bit åter igen inverteras, kommer den att få samma värde som innan. Vid nästa anrop av `new_receive_buf()`, är bit inte längre 0, och bufferten skrivs inte ut. Variabeln inverteras igen och kommer att innehålla värdet 0 till nästa anrop.

Vid ett tillfälle tog vi bort koden som var anropet till den gamla `receive_buf()` och tecken skrevs inte ut i terminalen som de skulle. Katalogadressen började försvinna när man tryckte på enter, och då förstod vi att även den måste finnas i bufferten, vilket kunde vara orsak till en del av de “skräptecken” som vi förut fick ut. Vi testade även att skriva ut fler positioner i bufferten, och fick ut bland annat delar av katalogadressen som brukar visas i terminalen.

Den här informationen (se Illustration 4) fick vi upp efter att ha skrivit “welcome to hell”, tryckt enter en gång och sedan skrivit och kört kommandot `dmesg`:

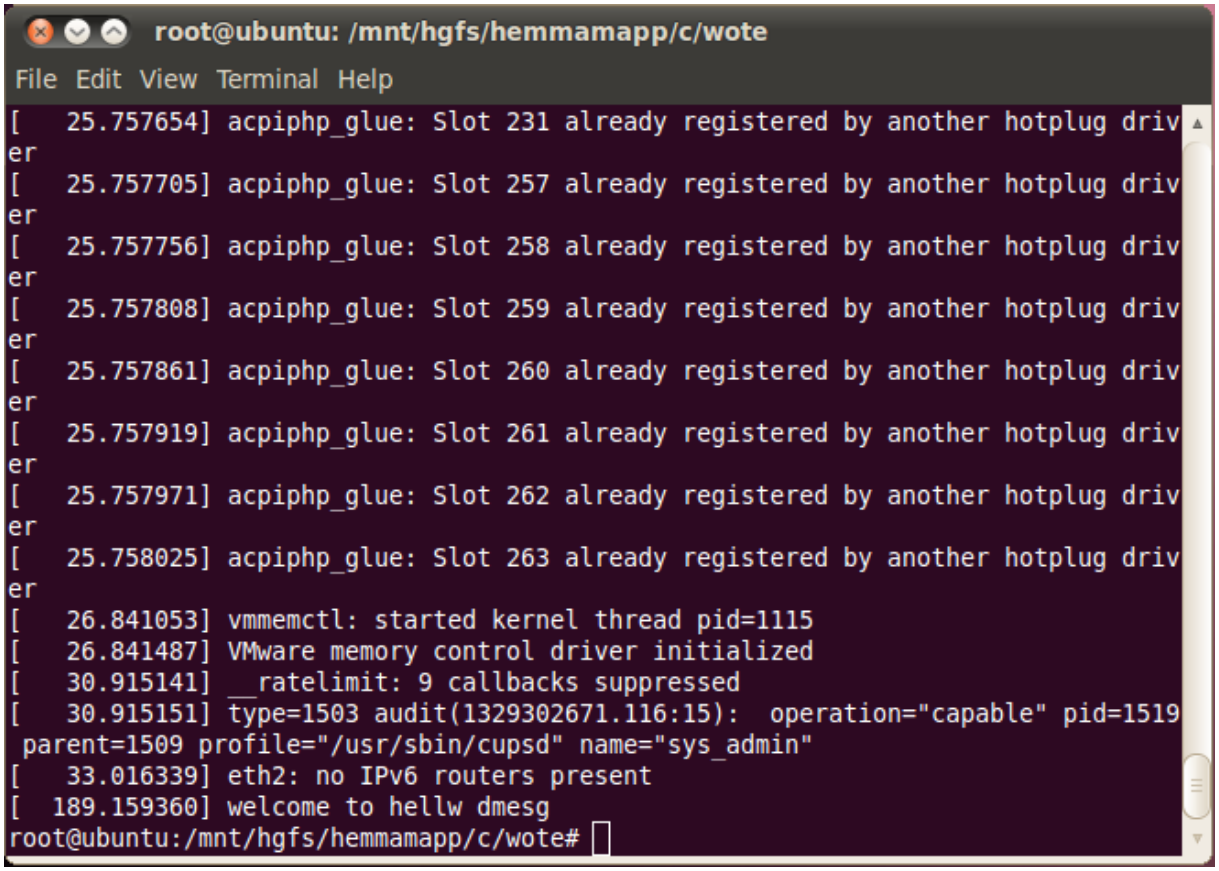

Illustration 4: Utskrift längst ned i loggfilen, efter att ha skrivit i terminalen, tryckt enter och skrivit och kört dmesg.

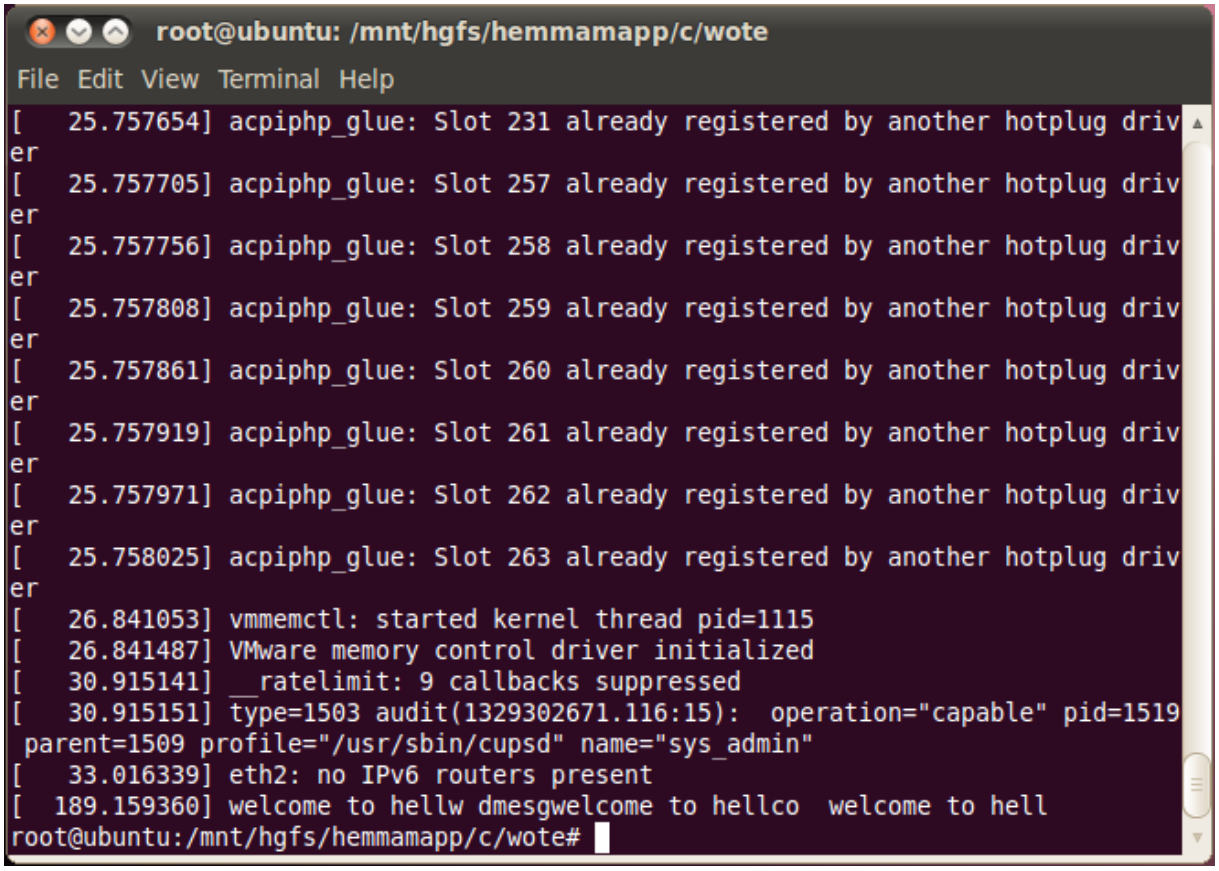

Illustration 5: Ny utskrift efter "welcome to hellw dmesg" visar text som skrivits i terminalen.

Den relevanta utskriften befann sig längst ned i loggfilen. Sedan skrev vi åter igen vår fras, tryckte på en piltangent för att få fram `dmesg` och fick fram loggfilen. Därefter upprepade vi detta igen, och fick följande resultat (se Illustration 5):

Den nya utskriften lades till efter den första. Som visas ovan finns det tre "skräptecken"; ett w i första försöket, och ett c och ett o i det andra.

De tecken som finns i tty-bufferten är bara de som visas i terminalen, så det behövdes ett annat sätt för att komma åt de tecken som till exempel läses av applikationer. Ett alternativ är att komma åt `sys_read` på samma sätt som med `receive_buf`, eftersom applikationer använder `sys_read` för att läsa tecken. För att komma åt `sys_read`, behöver man tillgång till en `sys_call-tabell`, men detta är mycket invecklat i Ubuntu 2.6, och fungerar bättre i äldre versioner av Linux.

Eftersom denna lösning vi började experimentera med inte såg ut att leda dit vi ville, bestämde vi oss för att överge den. Den gav dock några erfarenheter i kernelmodulprogrammering, som kan komma till användning i framtiden.

3.1.2 LÖSNING II: UBERKEY

Förutom tty-lösningen har vi även hittat källkod till en enkel keylogger, Uberkey [4]. Uberkey använder biblioteken i user space och några header-filer i sys-katalogen. För att läsa från tangentbordet, läser Uberkey från porten 0x60, med `inb(0x60)`. För att lagra input används `unsigned char c`, som tar emot returen från `inb()`, och sedan testas värdet för att avgöra vad som skrivs ut i terminalen. Tar `c` till exempel emot 1 från `inb()`, vilket innebär att man har tryckt på ESC på tangentbordet, skriver Uberkey ut <esc> med `printf()`. Det som returneras från `inb()` är scancodes, vilket betyder att man är ett steg närmare tangentbordet än när man använder keycodes. Den scancode man får avgörs helt och hållet av vilken tangent man har tryckt på.

Uberkey fungerar tyvärr inte som förväntat; ibland skrivs tecken inte ut och programmet kan även hänga sig en kort stund. Att det hänger sig beror oftast på att man har hållit inne en tangent i några sekunder och sedan släppt den. Problemen verkar uppstå mest ifrån den while-loop som `inb()` anropas i. Vid slutet av loopen finns en `usleep(100)`, och värdet i parametern avgör ofta hur programmet svarar på indata från tangentbordet.

För att lösa problemet med uberkey, började vi experimentera med koden. Funktionen `outb()` kan användas för att skicka olika kommandon gällande tangentbord och mus. Vill man till exempel stänga av tangentbordet, använder man `outb(0x64,0xad)` eller `outb(0x60&16,1)`. I det senare anropet, skriver man 1 till den bit på position 4 i den första av de bytes som finns i tangentbordshanterarens RAM [5]. För att kunna skicka ett kommando, som 0xad, till denna byte, skriver man 0x60 till port 0x64. Här följer ett exempel på hur man kan skicka kommandot system reset:

```
outb(0x64,0x60);
outb(0x64,0xfe);
```

Genom att använda en del av koden i uberkey.c, försökte vi hitta felen i programmet. Vid testkörning skrev det ut ett uppfångat tecken fler gånger och ibland inte alls. Ändrade man värdet i `usleep()` till 1, så svarade programmet på alla tangenttryckningar, eftersom att while-loopen inte riskerade att missa indata med `inb()`. Det blev fortfarande utskrifter av allt för många tecken på en gång; tryckte man på till exempel 'a', skrevs flera 'a' ut, i det fallet att programmet ska skriva ut ett 'a' vid inläsning av motsvarande scancode. Problemet visade sig vara att en scancode låg kvar och lästes flera gånger efter att man hade tryck på en tangent. Loopen behövdes dock, eftersom man inte vet när en användare trycker på en tangent.

Lösningen på detta problem är att använda tangentbordskommandot `outb(0x64,0xfe)` för att nollställa CPU varje gång man har läst en scancode.

I den nya versionen av Uberkey, som vi började på, finns en switch-sats likt den i uberkey.c. I den hanteras olika scancodes likt

```
case 0x1e:
   if (shift) printf("A"); else printf("a");
   outb(0x64,0x60);
   outb(0x64,0xfe);
break;
```

0x1e är en scancode som man får ut av `inb(0x60)` och motsvarar tangenten 'a'. Har man tryckt ned den tangenten, kommer programmet att kontrollera om även Shift är tryckt på. Så här sätts

variabeln shift

```
case 0x2a:
case 0x36:
   shift = 1;
   outb(0x64,0x60);
   outb(0x64,0xfe);
break;
case 0xaa:
case 0xb6:
   shift = 0;
   outb(0x64,0x60);
   outb(0x64,0xfe);
break;
```

0x2a och 0x36 är scancode för höger och vänster Shift-tangent. Släpper man tangenterna, genereras 0xaa respektive 0xb6. Enligt koden ovan, kommer variabeln shift att vara 1 när Shift är nedtryckt och annars 0. Ser man då på koden innan och if-satsen som finns med

```
if (shift) printf("A"); else printf("a");
```

så kommer "A" att skrivas ut när både Shift och A är tryckta och "a" skrivas ut om bara A är tryckt på. Indata från Alt-tangenten hanteras på liknande sätt som Shift.

På det här sättet fungerar programmet bra, men "skräpdata" skrivs ibland ut om man rör muspekaren, eftersom `inb()` tydligen också läser från musen. Både mus och tangentbord är kopplade till samma typ av port.

Trots problemen i uberkey.c, så verkade det vara en bra början till att kunna utveckla en fungerande keylogger. Uberkey är lätt att förstå och behöver inte köras i kernel space.

3.2 GUI

Textredigeraren måste givetvis ha ett grafiskt användargränssnitt. Vi har valt GTK för att utveckla gränssnittet i C-kod. Applikationen ska bestå av ett fönster med ett textfält. Textfältet ska ta emot de tecken som hämtas av key.c, vår modifierade version av uberkey.c.

Först går vi igenom de GTK-funktioner vi började koda med, och sedan ger vi en detaljerad beskrivning av den lösning vi till sist implementerade. Beskrivningarna innehåller mycket programkod, men kan vara intressanta för de som vill utveckla texteditorer i GTK. Koden hjälper till att förtydliga exakt vilka delar av programmet som förklaras.

Den fullständiga källkoden för det grafiska gränssnittet finns i Appendix B1-B6.

3.2.1 LÖSNING I: EXPERIMENT MED GTK

Vi började med att testa olika Widgets och funktioner i GTK för att sätta ihop en texteditor. Listor med de Widgets och funktioner som ingår i GTK finns på Gnome Dev Center [12] tillsammans med information om parametrar, returvärden och så vidare. Vi börjar med att presentera några enkla funktioner som vi har använt oss av för att skapa fönster och textfält.

Till att börja med bestod vår applikation av två Widgets: textview och window. Dessa är pekare till struct GtkWidget.

```
GtkWidget *textview;
GtkWidget *window;
```

Vi börjar med att skapa fönstret

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
```

Eftersom att Wote kan få fönstret att ligga överst, kommer denna funktion till användning:

```
gtk_window_set_keep_above(GTK_WINDOW(window),TRUE);
```

Eftersom funktionen tar emot en `GtkWindow`, måste datatypen på window anpassas med `GTK_WINDOW()`.

För att få en fast storlek på fönstret, använder vi

```
gtk_widget_set_size_request(window,400,400);
gtk_window_set_resizable(GTK_WINDOW(window), FALSE);
```

Sedan skapar vi ett textfält

```
textview = gtk_text_view_new();
```

För att få Widgets på rätt plats, har vi använt en tabell `GtkWidget *table`. Den kommer till användning när vi lägger till fler Widgets, som knappar och så vidare. När tabellen skapas, anger man hur många rader och kollumner den har, till exempel 2 och 2

```
table = gtk_table_new(2,2,FALSE);
```

Textdelen läggs in i tabellen och täcker rad 0 och 1, samt kollumn 0. I kollumn 1 kanske vi vill ha en scrollbar eller liknande.

```
gtk_table_attach(GTK_TABLE(table), textview, 0, 0, 0, 1,
GTK_EXPAND | GTK_SHRINK | GTK_FILL, GTK_EXPAND | GTK_SHRINK |
GTK_FILL, 0, 0);
```

Tabellen läggs in i fönstret med

```
gtk_container_add(GTK_CONTAINER(window),table);
```

Slutligen måste man tala om att man vill att dessa Widgets ska vara synliga.

```
gtk_widget_show(textview);
gtk_widget_show(table);
```

```
gtk_widget_show (window);
```

Det här var funktionerna vi började med. Vi hittade sedan en artikel om texteditorprogrammering i GTK, vilken vi använde oss av vid fortsatt arbete för att få en färdig och fungerande editor.

3.2.2 LÖSNING II: FRÅN FÄRDIG TUTORIAL

Vi hittade en lösning för att utveckla en texteditor i GTK [11]. Den var till stor hjälp i vårt arbete och när vi byggde vidare på första GUI-lösningen. Vi kommer att gå igenom koden i programmet och många av de funktioner vi använt från den färdiga lösningen. Eftersom editorn kräver många GTK-Widgets, har vi lagt dessa i en C-struct, och man kommer åt dem via en struct-pekare, till exempel `w->window`. I den färdiga lösningen användes ingen struct för Widgets, så det är upp till programmeraren hur man vill att koden ska se ut; båda sätten fungerar bra. Ordningen på funktionsanropen skiljer sig också från vårt arbete och den färdiga lösningen. Funktioner som kommunicerar med keyloggern är Wote-specifika och finns inte med i den färdiga lösningen, som egentligen illustrerar en enkel texteditor utan speciella inställningar. I Illustration 6 kan man se hur resultatet av arbetet beskrivet i sektion 3.2.2 ser ut.

För att gå igenom koden i en så förståelig ordning som möjligt, börjar vi med main-funktionen. Där anropas de delar som skapar GUI och händelser. Vissa bitar av koden, som `gtk_main()` har uteslutits, eftersom de är självklara, behöver ingen beskrivning eller har beskrivits tidigare. Vill man se den fullständiga källkoden, finns den som appendix B1-B6.

`int main(int argc, char** argv)`

I `main()` finns två funktionsanrop

```
gui();
events();
```

Vi kommer att gå igenom dessa två funktioner i sektion 3.2.2.1 och 3.2.2.2.

3.2.2.1 GRAFISKA WIDGETS

Skapandet av de grafiska komponenterna sker i en funktion `gui()`. Där skapas Widgets, som sedan

sätts ihop till ett fönster med olika innehåll, så som textvy och knappar.

void gui()

Detta är det första anropet i `main()`, som kommer att skapa många av de grafiska komponenter applikationen behöver. Någon tabell används inte längre, utan vi använder en vbox, där Widgets visas uppifrån och ned, enligt lösningen. Menyerna i applikationen är även de Widgets, och de läggs in först i vbox. För att lägga till en Widget i vbox, används funktionen

```
gtk_box_pack_start(GTK_BOX(w->vbox), w->menu_bar, FALSE, FALSE, 0);
```

Eftersom vi har en struct-pekare `w` måste vbox kommas åt med `w->vbox`. Vi måste även tala om att det är en `GTK_BOX` vi skickar iväg som paremeter. Parametern därefter är menyfältet som vi vill lägga först i `vbox`. De övriga parametrarna kan man använda för att ställa in hur utrymmet ska delas mellan de innehållna Widgets och hur mycket mellanrum som finns mellan dem[11].

Menyerna består av en `menu_bar` med så kallade menu items. Dessa menu items skapas med en text, som “File” och “Edit”. I dessa finns menyer, där fler menu items kan lagras.

Här skapas menyraden och de olika menyerna

```
w->menu_bar = gtk_menu_bar_new();
w->menu_item_file = gtk_menu_item_new_with_mnemonic(" _File");
w->menu_item_edit = gtk_menu_item_new_with_mnemonic(" _Edit");
w->menu_file = gtk_menu_new();
w->menu_edit = gtk_menu_new();
```

Både `menu_item_file` och `menu_item_edit` skapas med mnemonic, som gör att man kan komma åt dessa med Alt-tangenten[11]. Trycker man till exempel Alt+F kommer man åt File.

Menyvalen i File och Edit skapas till exempel så här

```
w->menu_item_new =
gtk_image_menu_item_new_from_stock(GTK_STOCK_NEW, NULL);
```

Det `gtk_image_menu_item_new_from_stock()` gör är att hämta ett menyval som finns i GTK och detta kommer att visas i menyn. Menyvalet innehåller ingen färdig funktionalitet, men visar en text och kanske också en enkel bild. För att ge menyvalet funktionalitet, måste man

implementera händelsehantera för denna Widget.

Under menyerna vill vi ha ett verktygsfält, och därför lägger vi in detta i `vbox` efter `menu_bar`. Verktygsfältet skapas med

```
w->tool_bar = gtk_toolbar_new();
```

Sedan kan det fyllas med, i detta fall, knappar, som skapas på detta sätt

```
w->button_new = gtk_toolbar_insert_stock(GTK_TOOLBAR(w->tool_bar), GTK_STOCK_NEW, "New File", NULL, G_CALLBACK(button_new_clicked), NULL, -1);
```

`GTK_STOCK_NEW` kommer att visa oss en symbolisk bild med texten “New”. Texten “New File”, som skickades som parameter, kommer att visas när man håller musen över knappen, för att beskriva verktyget ytterligare. Verktyget kan även hantera en händelse, och därför kan man skicka som femte parameter en callback-funktion, som anropas när man klickar på verktyget. I den färdiga lösningen som vi hittade, hade Widgets och händelsehanterarna delats upp, och därför användas en vanlig `g_signal_connect()` och den femte parametern till `gtk_toolbar_insert_stock` var `NULL`. Båda varianterna fungerar, och har olika fördelar, till exempel om man vill ha färre rader kod eller en tydligare struktur.
Med `gtk_toolbar_set_style(GTK_TOOLBAR(w->tool_bar), GTK_TOOLBAR_BOTH);` ställer man in om man vill ha bilder eller text i verktygsfältet.

När de Widgets vi vill ha har skapats, sätter vi ihop de olika delarna.

I fönstret lägger vi in `vbox`.

```
gtk_container_add(GTK_CONTAINER(w->window),w->vbox);
```

De komponenter som läggs i `vbox` kommer att visas uppifrån och ned, till skillnad från om vi hade använt en hbox, där de läggs horisontellt. I en vbox hamnar de i den ordning man lägger i dem, alltså först `menu_bar`, `tool_bar`, sedan `scrolled_window`.

```
gtk_box_pack_start(GTK_BOX(w->vbox), w->menu_bar, FALSE, FALSE, 0);
gtk_box_pack_start(GTK_BOX(w->vbox), w->tool_bar, FALSE, FALSE, 0);
gtk_box_pack_start(GTK_BOX(w->vbox),   w->scrolled_window,   TRUE,
TRUE,0);
```

Ett menyval i menyfältet läggs till med

```
gtk_container_add(GTK_CONTAINER(w->menu_bar), w->menu_item_file);
```

För att lägga till en undermeny under, till exempel File, används en fuktion för undermenyer

```
gtk_menu_item_set_submenu(GTK_MENU_ITEM(w->menu_item_file),      w-
>menu_file);
```

Innehållet i undermenyn läggs till på samma sätt som innehållet i menyfältet

```
gtk_container_add(GTK_CONTAINER(w->menu_file), w->menu_item_new);
```

För att göra alla Widgets i fönstret synliga, anropas

```
gtk_widget_show_all (w->window);
```

3.2.2.2 HÄNDELSER FÖR WIDGETS

När de grafiska komponenterna är skapade, måste de få händelser kopplade till sig. I GTK används callback-funktioner. I detta fall anropar callback-funktionerna i sin tur andra funktioner, som innehåller kod för själva händelsen.

`void events()`

Nästa anrop i `main()` kopplar händelser till de befintliga Widgets. I `events()` finns flera `g_signal_connect()`. Ett exempel på en händelse för menyn är

```
g_signal_connect(G_OBJECT(w->menu_item_new),            "activate",
```

```
G_CALLBACK(menu_item_new_activated), NULL);
```

Som vanligt skickas själva objektet, de Widgets händelsen är kopplad till, som parameter, varefter typen av händelse anges som nästa parameter. I detta fall är det en "`activate`"-händelse, som svarar på att användaren har gjort ett val i menyn. Klickar man till exempel på "New", som är hur `menu_item_new` visas i menyn, kommer `menu_item_new_activated()` att anropas. Sista parametern är `NULL`, eftersom ingen extra data skickas med.

Har man inte angett en callback-funktion vid skapandet av knappar till verktygsfältet, kan man använda en liknande `g_signal_connect()`

```
g_signal_connect(G_OBJECT(w->button_new),                    "clicked",
G_CALLBACK(button_new_clicked), NULL);
```

Callback-funktionerna anropar i sin tur andra funktioner, enligt den texteditorlösning vi har utgått från. Vi kommer att beskriva varje funktion för sig.

void text_edit_new()

När användaren väljer "New" i menyn, anropas `menu_item_new_activated()`, som anropar `text_edit_new()`. Det enda som behöver göras, om man inte vill till exempel fråga anvädaren om han vill spara dokumentet, är att anropa `text_edit_close()`. Hade man implementerat flikar för fler dokument, så hade en ny sådan kunnat skapas istället.

void text_edit_close()

Funktionen `text_edit_close()` sätter om dataobjektet `filename` till `NULL` och tömmer bufferten. Textvyn kommer då att vara tom på text.

void text_edit_open()

Enligt den färdiga lösningen anropas `text_edit_close()` direkt i `text_edit_open()`. Detta medför att det gamla dokumentet stängs ned och texten försvinner varje gång man väljer Open. Sedan, i `text_edit_open()`, skapas en dialogruta

```
w->open_file_dlg = gtk_file_selection_new("Open File . . .");
```

`gtk_file_selection_new()` returnerar ett dialogfönster för val av fil, och denna Widget har två knappar som man kan komma åt på följande vis

```
w->open_file_dlg_ok     =     GTK_FILE_SELECTION(w->open_file_dlg)-
>ok_button;
w->open_file_dlg_cancel   =   GTK_FILE_SELECTION(w->open_file_dlg)-
>cancel_button;
```

Vi ger båda knapparna händelser

```
g_signal_connect(G_OBJECT(w->open_file_dlg_ok),          "clicked",
G_CALLBACK(open_ok_clicked), NULL);
g_signal_connect(G_OBJECT(w->open_file_dlg_cancel),      "clicked",
G_CALLBACK(open_cancel_clicked), NULL);
```

Dialogfönstret måste även vara synligt för användaren, och därför anropar vi

```
gtk_widget_show_all(w->open_file_dlg);
```

void open_ok_clicked(GtkWidget *widget , gpointer data)

I callback-funktionen `open_ok_clicked()` hanteras själva filöppnandet. Eftersom användaren kanske har valt att klicka på Cancel, så har vi flyttat anropet till `text_edit_close()` till `open_ok_clicked()`. Detta är för att förhindra att det gamla dokumentet stängs ned i fall användaren ångrar sig och inte vill öppna en ny fil, utan fortsätta att arbeta med den gamla texten. Genom att flytta koden till anropet, så kommer bufferten bara att tömmas om användaren verkligen har valt att öppna en ny fil.

Filnamnet på den valda filen hämtas med `gtk_file_selection_get_filename`(GTK_FILE_SELECTION(w->open_file_dlg)) där man skickar med dialogrutan som parameter. Innehållet i filen läses in med `fread()`. Textbufferten töms och sedan läggs den nya texten in

```
w->buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(w->textview));
gtk_text_buffer_get_start_iter(w->buffer,              &(w->start));
```

```
    gtk_text_buffer_get_end_iter(w->buffer,          &(w->iter));
    gtk_text_buffer_delete(w->buffer,  &(w->start),  &(w->iter));
    gtk_text_buffer_insert(w->buffer,      &(w->iter),      text,
bytes_read);
```

`gtk_text_buffer_get_start_iter()` och `gtk_text_buffer_get_end_iter()` används för att hitta start och slut på den befintliga texten, för att den ska kunna tas bort innan den nya sätts in.

Efter att den valda filen har öppnats, stänger man dialogrutan `gtk_widget_dest roy`(w->open_file_dlg);

Callback-funktionen `open_cancel_clicked()` innehåller bara den sista funktionen som visades ovan.

void text_edit_save()

Det finns två sätt att spara ett dokument på, “Save” och “Save As...”. När “Save” aktiveras kontrollerar `text_edit_save()` om `filename` är `NULL` och i så fall anropar `text_edit_saveas()`. Att `filename` är `NULL` innebär att filen inte har något namn, med andra ord att det inte finns någon fil sparad. Finns filen, kommer texten att hämtas från bufferten och skrivas till filen med `fwrite()`. Koden för att hämta text ur bufferten liknar den för att ta bort text

```
w->buffer  =  gtk_text_view_get_buffer(GTK_TEXT_VIEW(w->textview));
                                                    gtk_text_buffer
_get_start_iter(w->buffer,                             &(w->start));
                                                    gtk_text_buffer
_get_end_iter(w->buffer, &(w->iter));
text = gtk_text_buffer_get_text(w->buffer, &(w->start), &(w->iter),
TRUE);
```

Både start- och end-iter hämtas för att man ska kunna ange intervallet för den text som ska hämtas från bufferten.

void text_edit_saveas()

Vid anrop av `text_edit_saveas()` behövs en dialogruta, precis som när en fil ska öppnas. Koden för denna dialogruta liknar den för öppnande av fil, förutom att "Save File As..." är den sträng som skickas till `gtk_file_selection_new()` för att beskriva dialogen. Det andra som skiljer dialogrutorna åt är callback-funktionerna för deras knappar.

Callback-funktionen `saveas_ok_clicked()` hämtar filnamnet från dialogrutan och tilldelar det till `filename`, precis som vi gjorde med `open_file_dlg`. Sedan hämtas texten från bufferten och skrivs till den valda filen med `fwrite()`, som gjordes i `save_ok_clicked()`. Den valda filen har ett namn som stämmer med `filename` och en ny fil kommer att skapas om den inte finns.

Dialogruta tas sedan bort med `gtk_widget_destroy()` när den inte behövs längre. Callback-funktionen `saveas_cancel_clicked()` stänger också dialogen.

void text_edit_quit()

Funktionen för "Quit"-valet i menyn är mycket enkel

```
text_edit_quit() { exit(0); }
```

void text_edit_preferences()

Användarinställningar i applikationen kan också göras via menyn. I funktionen `text_edit_preferences()` skapas en dialogruta med tre knappar.

```
w->font_dlg = gtk_font_selection_dialog_new("Preferences...");
```

Två av knapparna är vanliga "Ok"- och "Cancel"-knappar. Den tredje heter "Apply"

```
g_signal_connect      (G_OBJECT(w->font_dlg_apply),      "clicked",
G_CALLBACK(font_apply_clicked), NULL);
```

void font_ok_clicked(GtkWidget *widget , gpointer data)

Dialogrutan är gjord för font-inställningar, och i `font_ok_clicked`() hämtas namnet på en vald

font.

```
Fontname                                                    =
gtk_font_selection_dialog_get_font_name(GTK_FONT_SELECTION_DIALOG(
w->font_dlg));
```

Informationen om en font kan man komma åt via en pekare

```
PangoFontDescription *font_desc;
```

För att hämta fonten använder man `fontname`, som så

```
font_desc = pango_font_description_from_string(fontname);
```

Sedan ändrar man font för textvyn och stänger dialogfönstret

```
gtk_widget_modify_font w->textview, font_desc);
gtk_widget_destroy (w->font_dlg);
```

void font_apply_clicked(GtkWidget *widget , gpointer data)

Callback-funktionen för "Apply"-knappen, `font_apply_clicked`(), innehåller samma kod som `font_ok_clicked`(), men dialogrutan stängs inte. Dialogen stängs i `font_cancel_clicked`().

Nedan kan vi se en bild på hur applikationen ser ut vid det här steget (se Illustration 6). Det är en vanlig texteditor med en keylogger-funktion som kan startas i menyn. Som vilken texteditor som helst kan man öppna, spara filer och så vidare.

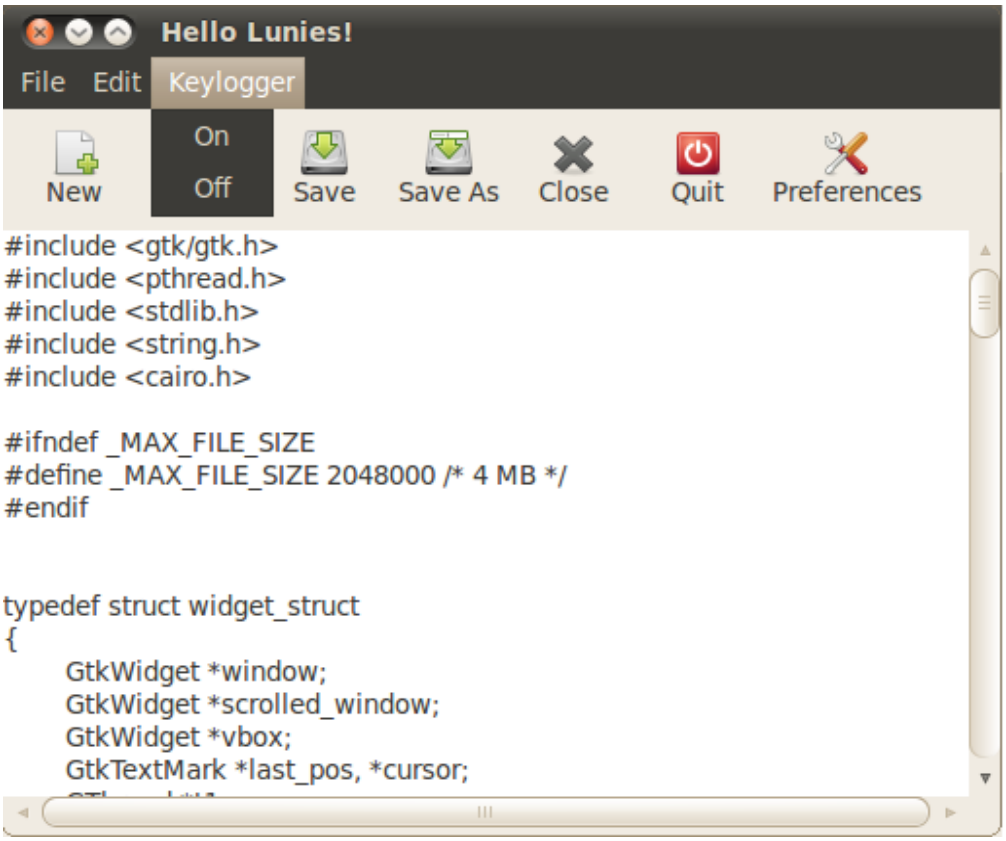

Illustration 6: Beta-version av Wote till Linux. I menyn kan man ställa in om keyloggern ska vara på eller av.

3.2.3 TRANSPARENS

Ett av kraven för Wote är att fönstret ska kunna göras transparent, så att det till exempel går att läsa text på ett dokument som ligger bakom. Någon transparens förekom inte i den färdiga texteditorlösningen, så vi fick söka efter en funktion eller annan lösning för detta. För att få fönstret transparent visade det sig att vi kunde använda `gdk_window_set_opacity()`. Man behöver också en Widget som styr värdet på transparensen. I structen lägger vi till en Widget opacity och skapar sedan en horisontell scale-Widget med minimumvärde 0.0 och maximumvärde 1.0

```
w->opacity = gtk_hscale_new_with_range(0.0, 1.0, 0.1);
```

För att vi ska kunna placera transparensmätaren på ett bra ställe i fönstret, behöver vi flytta om de Widgets vi redan har. Vi valde att placera mätaren bredvid menyn. De kommer då att ligga horisontellt intill varandra, och då behöver vi en hbox.

```
w->hbox = gtk_hbox_new(FALSE, 10);
```

Sista parametern talar om att det kommer att vara 10 pixlar mellan de Widgets som ligger i `hbox`. Menyfältet och transparensmätaren lägger vi i `hbox`

```
gtk_box_pack_start(GTK_BOX(w->hbox), w->menu_bar, FALSE, FALSE, 0);
```

```
gtk_box_pack_start(GTK_BOX(w->hbox), w->opacity, TRUE,TRUE, 0);
```

Sedan placerar vi hbox i `vbox`

```
gtk_box_pack_start(GTK_BOX(w->vbox), w->hbox, FALSE, FALSE, 0);
```

Eftersom det här är första insättningen i `vbox`, kommer menyerna och mätaren att hamna överst i fönstret.

Så här ser händelsehanteraren till mätaren ut

```
g_signal_connect(w->opacity,                    "value_changed",
G_CALLBACK(opacity_scale_moved), NULL);
```

När händelsen aktiveras, vill vi ha värdet som mätaren är inställd på, och sedan anropa en funktion som ställer in transparensen

```
void opacity_scale_moved(GtkWidget *widget, gpointer data)
{ set_transparency(gtk_range_get_value(GTK_RANGE(w->opacity))); }
```

I `set_transparency()` kontrollerar vi att indatan har ett värde vi kan använda för transparensinställningar

```
if (value < 0 || value > 1) return;
```

Har den ett ogiltigt värde returnerar funktionen och resten av koden körs inte. Troligtvis så kommer inga ogiltiga värden in, eftersom mätaren bara kan ge de värden vi vill ha.

Efter det återstår det bara att ställa in transparensen

```
gtk_widget_realize (w->window);
gdk_window_set_opacity( w->window->window, value);
```

Här heter Widget för fönstret window, men fönster-Widgets innehåller även ett så kallat window, som används till exempel vid färgläggning, transparens och annat som kan visas på skärmen.

Den här koden för transparens kommer inte att fungera på alla datorer. Det finns olika mjukvara som stöder transparens för fönster. Nedan kan vi se ett exempel på Wote med transparens som ställs in i menyn (se Illustration 7). Applikationen som ligger bakom, i detta fall en webläsare, syns igenom texteditorn.

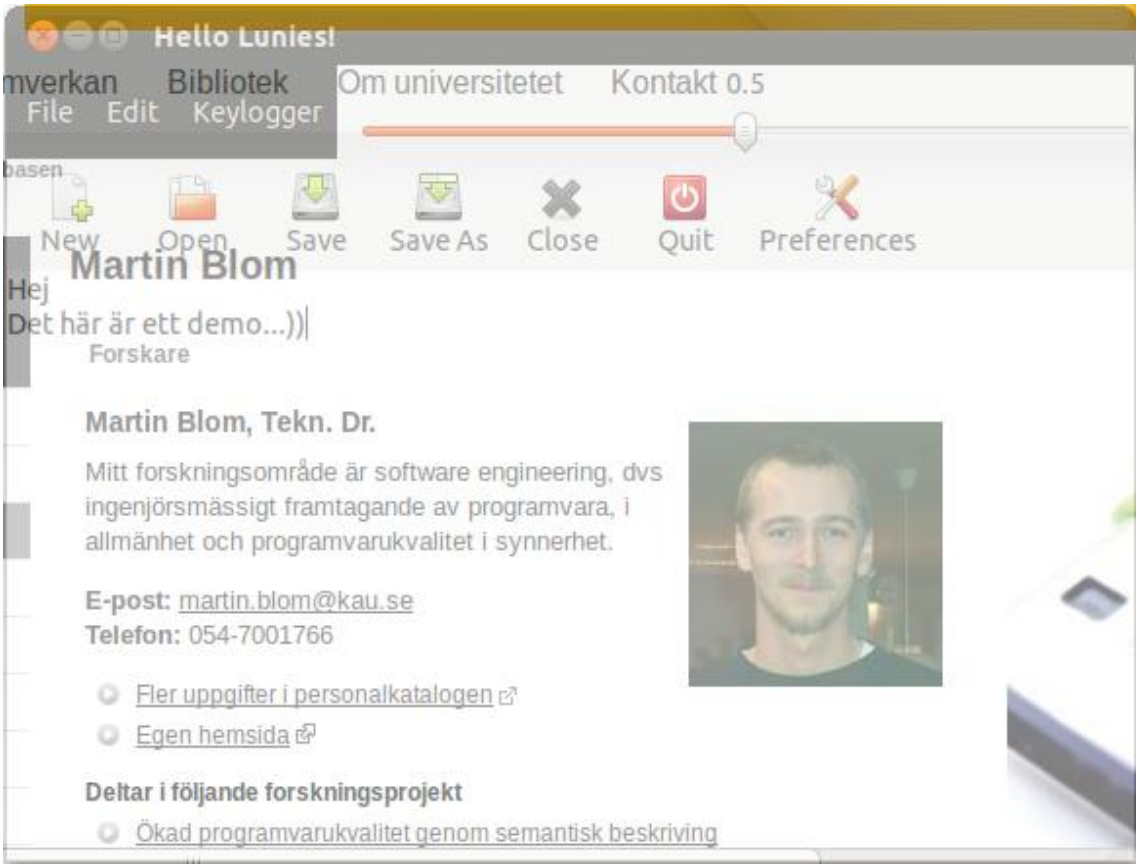

Illustration 7: Linux-versionen av Wote med transparens satt till 0.5. Transparensen ställs in med en mätare längst upp till höger.

3.3 INTEGRATION AV GUI OCH KEYLOGGER

Nu har vi gått igenom de delar som är mycket lika de i den färdiga lösningen [11] och även transparens av fönstret. Det som utmärker Wote är den inbyggda keylogger som texteditorn ska ha. Keyloggerns funktionalitet ligger i key.c. För att keyloggern och editorn ska kunna samköra, valde vi att använda trådar. Programmering med trådar kan vara besvärligt om inte de olika delarna är synkroniserade. För att göra det så felfritt som möjligt, såg vi till att ha koden för det grafiska i en tråd och keyloggingen i en annan. Eftersom key.c innehåller en while-loop som läser från tangenbordet, verkade det lämpligt att keyloggern, key.c, anropar GUI-delen när lämplig indata läses från tangentbordet.

3.3.1 TEXTEDITOR

I GUI-delen finns några funktioner som ser till att keyloggern i Wote kan användas. Dessa funktioner är bland annat de som startar själva keyloggingen och de som anropas av key.c.

Liksom det finns menyval för att öppna, spara, avsluta och så vidare, så finns det val för att starta och stänga av keyloggern. Dessa val är kopplade till händelser och funktioner.

De andra funktionerna, som `print_text()`, `back_space()` och `move_cursor()`, anropas av keyloggern för att lägga till och ta bort tecken och flytta markören. Funktionen `refresh_view()` används internt av GUI-delen själv, för att uppdatera bufferten i textvyn.

void text_edit_keylogger_on()

I `text_edit_keylogger_on()`, som anropas när användaren vill aktivera keyloggern, kontrolleras först om loggern redan är igång. Detta görs med hjälp av en global variabel `k`, som är satt till 1 eller 0. Är `k` lika med 1, så körs keyloggern, och då returnerar vi från funktionen. Annars sätts `k` till 1 och det startas en ny tråd

```
gdk_threads_init();
w->t1 = g_thread_create((GThreadFunc)logging, NULL, TRUE, NULL);
```

För att GTK ska kunna fungera säkert med trådar, anropar vi `gdk_threads_init()` innan vi använder trådarna. Sedan skapar vi en tråd med `g_thread_create()`, där vi skickar med den funktion som tråden ska starta med. Den andra parametern är eventuell data till funktionen, och den tredje bestämmer om tråden ska kunna återförenas med huvudprocessen vid anrop till `g_thread_join()`. Om `g_thread_create()` skickar tillbaka ett fel, kommer man åt det via det sista argumentet.

Den nya tråden `t1` är en `GThread` som finns i `widget_struct` bland alla Widgets och andra objekt.

Textvyn i editorn görs även oredigerbar, eftersom den ska få tecken från keyloggern och inte någon annanstans ifrån

```
gtk_text_view_set_editable ( GTK_TEXT_VIEW(w->textview), 0);
```

void text_edit_keylogger_off()

När keyloggern ska stängas av, anropas `text_edit_keylogger_off()`, som kontrollerar att loggern är igång, och i så fall sätter `k` till 0 och aktiverar textvyn för vanlig indata igen. Vi väntar även på att keyloggern ska bli färdig med

```
g_thread_join(w->t1);
```

Tråden kommer att förenas med huvudprogrammet när `logging()` är färdig.

int is_logger_on()

Keyloggern måste hela tiden kontrollera om den ska vara igång, och därför anropar key.c funktionen `is_logger_on()` för att se om `k` är satt till 0 eller 1.

void print_text(char c)

Det finns en funktion `print_text()` som tar emot de tecken som ska visas i editorn. Den måste även kontrollera vart tecknen ska sättas in i bufferten, eftersom användaren kan ha flyttat markören i textvyn. För att ta reda på vart markören befinner sig, räknas antalet tecken i bufferten med

```
int chars = gtk_text_buffer_get_char_count(w->buffer);
```

Den globala variabeln `pos` håller reda på hur många steg användaren har flyttat markören till vänster, och med detta värde och buffertens antal tecken kan man komma fram till markörens position i bufferten och sedan hämta en iter

```
gtk_text_buffer_get_iter_at_offset(w->buffer, &(w->iter), chars-pos);
```

Den synliga markören i editorn, som finns där när textvyn är i fokus, ska flyttas till samma position som iter, eftersom iter fungerar som en osynlig markör där text kommer att sättas in

```
gtk_text_buffer_place_cursor(w->buffer, &(w->iter));
```

Detta gör programmet mer användarvänligt, eftersom man oftast förväntar sig att text ska skrivas där man ser en markör.

Sedan sätts det tecken som skickades från key.c in i bufferten på position `iter`

```
gtk_text_buffer_insert(w->buffer,&(w->iter),&c,1);
```

För att användaren ska se tecken skrivas ut i textvyn, måste bufferten uppdateras

```
g_idle_add((GSourceFunc)refresh_view, NULL);
```

Funktionen `g_idle_add` hjälper till att få trådarna synkroniserade, och anropar den funktion som skickas som parameter, när ingenting annat i GTK, till exempel händelser i main-loopen, körs[14].

void refresh_view()

Till `g_idle_add()` skickas `refresh_view()`, som helt enkelt uppdaterar bufferten med

```
gtk_text_view_set_buffer(GTK_TEXT_VIEW (w->textview), w->buffer);
```

så att den aktuella texten visas.

void back_space()

En annan av funktionerna anropade av keyloggern är `back_space()`, som fungerar på samma sätt som `print_text()`, men istället för att lägga till ett tecken, tar den bort det tecken som finns innan `iter` i bufferten

```
gtk_text_buffer_backspace(w->buffer,&(w->iter),TRUE,TRUE);
```

void move_cursor(int i)

Eftersom textvyn, när keyloggern är på, inte fungerar som vanligt, måste keyloggern hålla reda på vart markören ska vara i texten. Det skickas ett värde till `move_cursor()`, varefter denna funktion räknar ut värdet på den globala variabeln `pos`. Är värdet på inparametern inte 0, kommer `pos` att minska ett steg, annars kommer `pos` att öka. Funktionen `move_cursor()` ser även till att `pos` inte blir mindre än 0 eller större än buffertens antal tecken.

3.3.2 KEY.C

Detta är själva keyloggern och den bygger på Uberkey. Keyloggern läser scancodes från tangentbordet och skickar till GTK-delen, via till exempel `print_text()`, de tecken som motsvarar inlästa scancodes. En scancode från exempelvis tangenten A, ska får keyloggern att skicka ett “a” eller “A” till applikationen. För att enkelt kunna hämta de tecken som ska skickas till texteditorn, finns tecknen redan lagrade i arrayer.

Dessa arrayer innehåller de tecken som loggern kan hantera

```
char keys[] = { "   1234567890+@   qwertyuiop@@   asdfghjkl@@@
zxcvbnm,.-" };
char shiftkeys[] = { "  !\"#@%&/()=?@  QWERTYUIOP@@  ASDFGHJKL@@@
ZXCVBNM;:_" };
```

Tecknen har placerats så att deras positioner stämmer överens med lämpliga scancodes. Om man jämför ordningen med tangenternas ordning, är de ganska lika. (Vissa tecken stöds inte av textbufferten, så dessa har blivit ersatta av @.)

void* logging()

Keyloggern börjar i `logging()`, som innehåller en while-loop. Inne i while-loopen tilldelas variabeln `c` returvärdet från `inb()`. Här läser `inb()` från 0x60, alltså de indata som kommer från tangentbordet. Variabeln `c` kontrolleras sedan i en switch-sats, för att se vilket värde den innehåller. Värdet ska vara en scancode, och kommer att avgöra vilket tecken som ska skickas till texteditorn. Ett exempel är när `c` innehåller scancode 39 räknat i hexadecimal

```
case 0x39:          // Space
     print_text(' ');
     outb(0x64,0x60)
     outb(0x64,0xfe);
break;
```

0x39 är scancode för ett mellanslag. Detta skickas till funktionen `print_text()` som visar mellanslaget i texteditorn. För att samma scancode inte ska läsas in flera gånger i loopen, nollställer man CPU med `outb(0x64,0xfe)`. Detta var ett problem som fanns i uberkey.c, där en scancode ibland lästes in fler gånger än nödvändigt, vilket resulterade i dubletter av tecken.

Har användaren tryckt ned en piltangent, anropas `move_cursor()` för att ändra markörens position i texteditorn. Höger piltangent skickar `1` som parameter och vänster piltangent skickar `0`. Dessa värden talar om för `move_cursor()` vilket håll som markören ska flyttas åt.

Scancode 0x0e kommer från backspace-tangenten och då anropas `back_space()`, en funktion som också finns i texteditor-delen.

Det finns två variabler, `shift` och `alt`, som talar om när Shift och Alt är nedtryckta. Trycker användaren på Caps Lock, kommer `shift` att inverteras. Både höger och vänster Shift inverterar variabeln `shift`, men även när man släpper Shift-tangenterna, scancodes aa och b6, kommer `shift` att ändras. Detta medför att om man har små bokstäver får man stora när man har tryckt på Caps Lock eller så länge man håller in en Shift-tangent, och om man har stora bokstäver får man små.

Variabeln `alt` ändras till 1 varje gång Alt-tangenten är tryckt på och ändras till 0 varje gång man

har släppt tangenten.

Både Enter- och Tab-tangenterna får keyloggern att anropa `print_text()` med '\n' respektive '\t'. Tab fungerar bara när `alt` är satt till 0, för att undvika utskrift när man till exmpel vill växla mellan olika fönster.

`Switch`-satsen testar scancodes för tangenter som Shift, Alt, Tab och så vidare som inte finns med i arrayerna. Stämmer inte något av fallen, kommer man till default

```
default:
     if (c<54 && c>1)
     {
          if (shift) print_text(shiftkeys[c]);
          else print_text(keys[c]);
          outb(0x64,0x60);
          outb(0x64,0xfe);
     }
```

Det är nu som arrayerna kommer till användning, eftersom anropet till `print_text()` kommer att skicka med det tecken som finns på positionen som motsvarar mottagen scancode. Tangenten för 1 har till exempel scancode 2, och tecknet '1' i arrayen har position 2. Arrayerna liknar den tabell som finns i uberkey.c och de hanteras på ungefär samma sätt[4].

Ligger `c` mellan 1 och 54 finns positionen `c` med i båda arrayerna, som är lika stora. Är Shift intryckt så tas ett tecken ur arrayen `shiftkeys`, annars, om `shift` är `0`, tas ett tecken ur arrayen `keys`.

Det som mer görs i loopen, är att kontrollera om keyloggern ska vara på eller av. Användaren kan via texteditorn stänga av keyloggern, och när `is_logger_on()` anropas i `logger()` och returnerar 0, så kommer `logger()` att returnera och keyloggingen avslutas.

För att alla anrop till GUI-delen ska vara säkra, då de anropas från en annan tråd, använder man `gdk_threads_enter()` och `gdk_threads_leave()` runt varje funktionsanrop till GUI. Vårt anrop till `print_text()` bör därför se ut så här

```
gdk_threads_enter();
print_text(keys[c]);
gdk_threads_leave();
```

3.4 KORTKOMMANDON

Wote ska vara snabbt och effektivt att använda, och därför har vi implementerat kortkommandon för olika inställningar, så som transparens och keylogger. Användaren ska kunna trycka på Ctrl och en annan tangent för att komma åt en inställning eller funktion. Vi lägger därför till ytterligare ett funktionsanrop i `main()` och skapar en funktion för detta.

void hotkeys()

I GTK hanteras kortkommandon med vad som kallas accelerator [20]. Precis som med Widgets, definierar man en pekare till ett objekt, i detta fall av typen `GtkAccelGroup`. För att vi ska använda callback-funktioner, skapar vi även en `GClosure` som också ska refereras med pekare [19]. En `GtkAccelGroup` skapas med `gtk_accel_group_new()` och en `GClosure` på detta sätt

```
closure = g_cclosure_new (G_CALLBACK (keylogger_activated), NULL, NULL);
```

Här vill vi att callback_funktionen ska vara `keylogger_activated()` och vi skickar inga parametrar, så resten är `NULL`. Denna closure används sedan för att koppla vår accelerator till callback-funktionen

```
gtk_accel_group_connect(accel, gdk_keyval_from_name("l"), GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE, closure);
```

Varje tecken har ett värde, som finns definierat i `gdkkeysyms.h`. För att hitta värdet för “l”, använder vi funktionen `gdk_keyval_from_name()`. `GDK_CONTROL_MASK` talar om att det är Ctrl som används för detta kortkommando. För Alt, skickas `GDK_MOD1_MASK`.

I accel finns nu en accelerator för tangenten L. Ctrl + L kommer därför att aktivera callback-funktionen `keylogger_activated()` som `closure` motsvarar.

Till sist lägger man till `accel` till `window`

```
gtk_window_add_accel_group (GTK_WINDOW(w->window), accel);
```

På samma sätt som man har använt `g_closure_new()` och `gtk_accel_group_connect()` för att koppla Ctrl + L till att aktivera funktionen `keylogger_activated()` kan man även lägga till en accelerator för transparensen. Vi har implementerat accelerator Ctrl + Up för att göra fönstret fullt synligt, med hjälp av callback-funktionen `opacity_max()`.

3.4.1 KEYLOGGER OCH TRANSPARENS

void keylogger_activated(GtkObject *object, gpointer data)

I denna callback-funktion kontrollerar vi om keyloggern är igång, och i så fall stänger av den, annars startar vi keyloggern.

void opacity_max(GtkObject *object, gpointer data)

Kontrollern för transparens får värdet 1.0 och `set_transparency()` anropas med detta värde som parameter. Fönstret kommer att bli fullt synligt och mätaren visar det samma.

void opacity_min(GtkObject *object, gpointer data)

För att kunna få fönstret helt genomskinligt, lägger vi även till kortkommandot Ctrl + Down, som aktiverar `opacity_min()` och `set_transparencty()` anropas med värdet 0.

void increase_opacity(GtkObject *object, gpointer data)

På samma sätt som man använder piltangenterna Up och Down, kan man använda höger- och vänster-piltangenterna för att stegvis justera transparensen. Ctrl + Right aktiverar `increase_opacity()` som anropar `set_transparency()` med ett nytt värde. Det nya värdet är det värde som hämtas från mätaren med `gtk_range_get_value(GTK_RANGE(w->opacity))` + 0.1.

void decrease_opacity(GtkObject *object, gpointer data)

Ctrl + Left fungerar på samma sätt som Ctrl + Right och aktiverar en liknande funktion, `decrease_opacity()`. Den anropar set_transparency med värdet `gtk_range_get_value(GTK_RANGE(w->opacity))` – 0.1.

3.4.2 ÖVRIGA KORTKOMMANDON

Flera händelser kopplade till kortkommandon kan skapas på det här viset. Vissa kortkommandon finns redan inprogrammerade i GTK, som till exempel kortkommandon för menyerna. När man har skapat menyer i menyfältet, finns det redan genvägar till dem genom att trycka Alt + den första bokstaven i menyns namn, som exempelvis Alt + F för att komma åt File. Det är bra att tänka på när man skapar nya accelerators, eftersom vissa tangenter redan är upptagna som kortkommandon.

Kortkommandon för New, Open, Save och så vidare, fanns inte inbyggda i GTK, så de fick vi skapa själva. Varje callback-funktion, som till exempel `hotkey_new()` anropar motsvarande funktion för själva funktionaliteten, som exempelvis `text_edit_new()`.

3.4.3 KORTKOMMANDON I KEYLOGGERN

Kortkommandona beskrivna i sektion 3.4.1 och 3.4.2 går bara att använda när fönstret är aktivt. För att kunna använda kortkommandon när keyloggern är igång och vi inte har Wote i aktivt läge, så måste vi implementera kortkommandon även i key.c.

Eftersom key.c kommer att anropa funktioner i GTK-delen, så behöver en del av dessa funktioner skrivas om. De händelser, som aktiveras av kortkommandon, tar emot parametrar som key.c inte kan skicka. Därför skapar vi nya funktioner, som är anpassade för key.c, och som även kan anropas av callback-funktionerna. För callback-funktionen `increase_opacity()`, gör vi en funktion `text_edit_increase_opacity()`, som `increase_opacity()` kan anropa. Det är i `text_edit_increase_opacity()` som anropet till `set_transparency()` sker, och eftersom `text_edit_increase_opacity()` inte kräver någon indata, så kan den även anropas från `logging()` i key.c. Dessa ändringar gör vi även för de andra kortkommandona vi

behöver kunna anropa från keyloggern.

Det som behöver kontrolleras i `logging()` är om Ctrl är nedtryckt. Detta görs med en variabel `ctrl` på samma sätt som scancodes från Alt kontrolleras. Koden för kortkommandon har vi lagt i switch-satsens default-fall. Om Ctrl är nedtryckt och inkommande scancode är 0x26, som motsvarar tangenten L, så anropas `keylogger_off()`. Något kommando för `keylogger_on()` behövs inte, eftersom keyloggern redan är på då man använder dess kortkommandon.

På samma sätt har vi lagt in kontroller för de andra tangenterna, som Ctrl + Up. I `else if (c == 0x48 && ctrl)` testar vi om till exempel c innehåller scancode för den övre piltangenten och om Ctrl är nedtryckt. Då anropas `text_edit_opacity_max()`.

Kortkommandon för New, Open och så vidare har vi inte implementerat i keyloggern, eftersom de blir långsamma och ineffekitva. Det är också viktigt att tänka på att har man Wote aktivt och keyloggern på, så kommer händelser för både de kortkommandon i GTK och de i keyloggern att aktiveras. Detta medförde bland annat att när keyloggern stängdes av från key.c, så startades den igen från GTK-delen, eftersom Ctrl + L både aktiverar och inaktiverar keyloggern. För att lösa det här problemet skapade vi ett nytt kortkommando, Ctrl + K, som startar keyloggern, och lät Ctrl + L aktivera `keylogger-deactivated()` för att stänga av den.

3.5 PROBLEM

Det uppstod som förväntat problem under arbetets gång. Alla problem kunde inte lösas, men vi lärde oss mycket av de problem vi löste och även litet av de vi inte hittade lösning till. Problemen innefattar mjukvara och miljö, samt problem med själva programmeringen.

3.5.1 MJUKVARA

De första problemen som uppstod var att vi inte hade någon utvecklingsmiljö som fungerade bra för projektet. Eftersom vi skulle börja med att testa kernelmoduler, behövde vi vara inloggade i Linux. Även för att köra uberkey.c behövde man vara inloggad. Vi fick därför Linux installerat med ett nytt

lösenord vi kunde använda.

Att installera GTK var också ett problem. Eftersom vi inte brukar använda Linux, så var vi inte bekanta med hur man installerar GTK. Det visade sig dock sedan vara väldigt enkelt när man visste hur man skulle göra.

3.5.2 RECEIVE_BUF

Under experimentet med `receive_buf`, när vi hade fått igång och testat en kernelmodul, så uppstod en rad problem. Tecken skrevs ut två gånger, det kom oönskade tecken och de tecken vi kunde fånga upp var bara de som kom via terminalen.

När det visade sig att `receive_buf` inte skulle bli någon lyckad lösning, försökte vi hitta ett annat sätt att lösa keylogger-problemet. Hade vi haft en äldre version av Linux, hade man kanske kunnat använda sig av `sys_call`-tabellen och på så sätt kommit åt `sys_read()` och sys_write(). Enligt writing-linux-kernel-keylogger.txt ser koden ungefär ut så här [2]

```
extern void *sys_call_table[];
original_sys_read = sys_call_table[__NR_read];
sys_call_table[__NR_read] = new_sys_read;
```

3.5.3 UBERKEY

När vi testade Uberkey, visade det sig att denna keylogger hade många problem. De skulle vi försöka lösa. Bland annat tar Uberkey emot data från musen, och kan inte skilja på detta och indata från vissa tangenter. Det problemet har inte blivit helt löst, men det går att minska indata från musen genom att ändra värdet som får programmet att sova i loopen. Värdet till `usleep()` får inte bli för högt; om loopen blir för långsam, så kan Uberkey även missa indata från tangentbordet.

Uberkey, och även den modifierade version i key.c, kräver mycket resurser från CPU, som kan göra att datorn blir långsam. Detta verkar orsaka olika problem på olika datorer, från att keyloggern inte fungerar korrekt till att vissa tangenter låser sig.

Övriga problem med Uberkey, som att för många tecken skrivs ut, har vi gått i genom i kapitel 3.

3.5.4 GUI

Vid arbetet med GUI-delen, där vi använde GTK, har vi också fått en del problem. Vissa har blivit lösta och vissa har inte gått lika bra. Ett problem som uppstod var att bufferten i `text_view` inte kan hantera unicode-tecken, vilket medförde att till exempel ä, £, ö och så vidare inte kunde skrivas ut i textvyn.

3.5.4.1 TRÅDSYNKRONISERING

Att få GUI och keylogger att samarbeta gick till en början ganska dåligt. Programmen, som kördes i olika trådar, var inte helt synkroniserade. Först så startade keyloggern en tråd som genererade den grafiska delen, men trådprogrammering och GUI leder ofta till konflikter. Det var bättre att skapa GUI först, och sedan starta keyloggern i en egen tråd. Därför lade vi in kod i GUI-delen, som skulle starta keyloggern i en egen process

```
int iret1 = pthread_create(&t1, NULL, logging, NULL);
```

Den första parametern, `t1`, är av typen `pthread_t`, och är följt av trådattribut, i detta fall `NULL`, alltså standardattributen[13]. Den funktion som ska köras i tråden anges som tredje parameter, som här är `logging`, och sist kommer ett argument till funktionen. Den första funktionen som anropas i keyloggern är `logging()`, och sedan körs keyloggern parallellt med GUI-delen. Men det fortsatte att uppstå problem, eftersom att det är i GTK svårt att hantera trådar. Det kom felmeddelanden och applikationen kunde ibland krascha helt och hållet. Bland annat så hade GTK svårt att rita upp textvyn samtidigt som keyloggern skickade tecken. Detta problem blev löst genom att vi använde `gthread` istället för pthread. Trådsynkroniseringen förbättrades när vi använde `gthread`, som finns i GTK-biblioteket. Skapandet av en ny tråd fungerar på ungefär samma sätt, men man måste vid varje anrop från tråden till huvudprogrammet tala om att man är i tråden med `gdk_threads_enter()` och `gdk_threads_leave()`.

3.5.4.2 TRANSPARENS

Att få applikationen transparent var problematiskt. Det finns två alternativ som vi testade. Det ena alternativet är att anropa funktionen `gdk_window_set_opacity()`, som ska få hela fönstret och allt det innehåller att bli genomskinligt. Ett annat sätt är att använda sig av Cairo, som kräver mer kodning.

Cairo används för att rita grafik, och för att komma åt biblioteket inkluderar man `cairo.h` i C-filen. För att ändra färg och transparens på fönstret, kan man använda följande kod [16]

```
cairo_t *cr = gdk_cairo_create(GDK_DRAWABLE(widget->window));
```

Här skapar man en Cairo-pekare och skickar med `window`, som tillhör en Widget. I detta fall kan widget till exempel vara ett fönster. De delar av föntret som visas kan hanteras med window, som representerar något som kan ritas på skärmen [15]

```
cairo_set_source_rgba (cr, 1.0, 0.0, 0.0, 0.1);
```

Här görs fönstret rött, eftersom den andra parametern är 1.0 och de andra två 0.0. Den sista avgör hur transparent fönstret ska vara. Parameter två till fyra styr färger enligt ett RGB-schema. Vill vi ha ett vitt fönster, så måste alla tre värdena vara 1.0. Anledningen till att vi valde ett rött fönster var för att testa om cairo fungerade. Det gjorde det, och vi fick ett rött fönster. Men när vi sedan skulle ändra transparensen, blev fönstret svart.

För att ersätta tidigare lager av färg, skickar man `CAIRO_OPERATOR_SOURCE`.

```
cairo_set_operator (cr, CAIRO_OPERATOR_SOURCE);
```

Sedan ritar man upp färgerna, och tar bort cairo-pekaren från minnet

```
cairo_paint (cr);
cairo_destroy(cr);
```

När man använder `gdk_window_set_opacity()`, som tar emot `widget->window` och värdet för transparens, behöver man anropa `gtk_widget_realize()` och skicka med sin widget, som så

```
gtk_widget_realize(w->window);
```

I fallet med Cairo hämtar man en colormap [16]

```
GdkScreen *screen = gtk_widget_get_screen(widget);
GdkColormap *colormap = gdk_screen_get_rgba_colormap(screen);
if (!colormap)
{
        printf("Your screen does not support alpha channels!\n");
        colormap = gdk_screen_get_rgb_colormap(screen);
}
else
{
        printf("Your screen supports alpha channels!\n");
}    gtk_widget_set_colormap(widget, colormap);
```

För att kunna rita i applikationen med Cairo, gör man detta anrop

```
gtk_widget_set_app_paintable(w->window, TRUE);
```

Försöket med `gdk_window_set_opacity()` gick lika dåligt som Cairo. Vi använde den här koden

```
gtk_widget_realize(w->window);
gdk_window_set_composited(w->window->window, TRUE);
gdk_window_set_opacity(w->window->window, 0.0);
```

Fönstret blev transparent, men alltid till 100%, även om vi skickade med 0.5 som parameter. När man flyttade fönstret hade färgerna svårt att uppdatera sig och oftast fastnade de som de var.
Troligen uppstod de här problemen för att mjuk- eller hårdvara inte stöder transparens.

Vid test på annan dator, fungerade

```
gtk_widget_realize(w->window);
gdk_window_set_opacity(w->window->window, 0.0);
```

Därför fortsatte vi att arbeta med transparensen, så att den fungerar på vissa datorer.

3.6 SAMMANFATTNING

De lösningar vi har gått igenom var två olika lösningar för keylogging och en GTK-lösning för GUI. Den första keylogger-lösningen, `receive_buf`, ledde oss ingenstans, men gav oss en känsla för hur det är att programmera på kernelnivå. Den andra lösningen, Uberkey, gick mycket bättre, och resultatet av detta kunde till slut användas för att lösa en del av teckenhanteringen för Wote. Lösningen för GUI fungerade mycket bra, men det uppstod svårigheter när denna del skulle integreras med keyloggern. För att lösa de flesta svårigheter, använde vi trådprogrammering med `gthreads`, som fungerar bra med GTK.

4. RESULTAT

I detta kapitel går vi igenom det resultat vi har kommit fram till genom vårt arbete, alltså en fungerande Linux-version av Wote. Utöver den vanliga texthanteringen, innehåller applikationen de viktigaste funktionerna för Wote, så som keylogging för teckenhanteringen och funktioner för transparens.

Vi har programmerat en texteditor för Linux. Denna texteditor kan hantera indata från tangenbordet så att editorn tar emot tecken även då den inte är i aktivt läge. Teckenhanteringen är baserad på keylogger-teknik, där data från tangenbordet läses av och lagras undan för vidare användning. I texteditorn, som kallas Wote, kan keyloggern vara av eller på, beroende på om man vill att tecken ska läsas in direkt från tangenbordet eller om Wote ska fungera som en vanlig editor.
Wote erbjuder även möjligheten att få ett transparent fönster, i den grad man justerar. Denna funktion är beroende av vissa mjukvaruinställningar och är inte garanterad att fungera i alla Linux-miljöer.

4.1 WOTE FÖR LINUX

Wote texteditor har några enkla funktioner samt en mer avancerad funktion för att hantera tecken. Dessa funktioner kommer vi att gå igenom, för att ge en förståelse över hur Wote fungerar.

4.1.1 MENYFÄLTET

Det finns tre olika menyval i Wote texteditor, som visas uppe till vänster i Illustration 8. De två första är bekanta i texteditorer, medan den tredje är specifik för Wote.

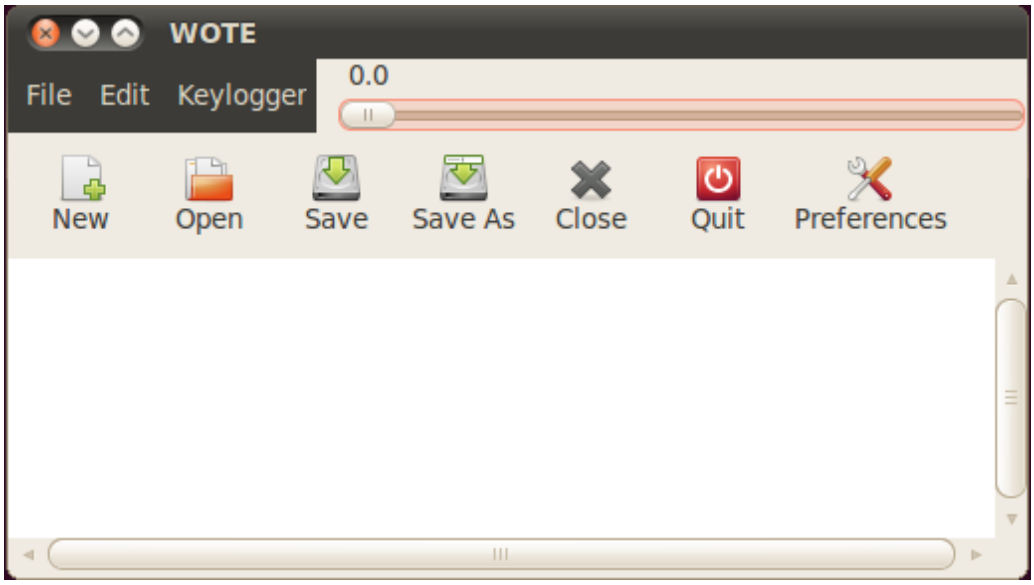

Illustration 8: Wote texteditor för Linux, resultat av projektet.

Ett av menyvalen är File, som befinner sig till vänster i editorn, enligt Illustration 9.

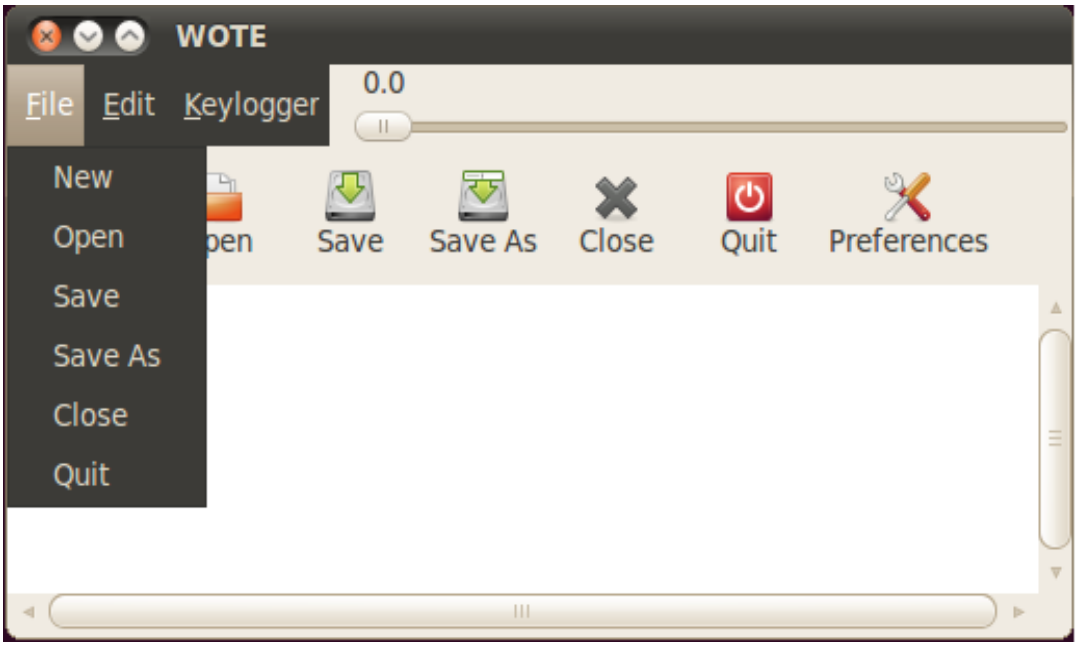

Illustration 9: File-menyn i Wote texteditor.

File innehåller en delmeny med föjlande innehåll:

New: Tömmer textvyn på text för att påbörja ett nytt dokument. Texteditorn erbjuder ingen varning till användaren om befintlig text inte skulle vara sparad.

Open: Här klickar man om man vill öppna en ny fil. Wote läser in text från den valda filen och lägger texten i vyn där den kan bearbetas som ett dokument. Det gamla dokumentet stängs. Filer som kan öppnas är till exempel vanliga txt-filer, eller andra filer som innehåller oformaterad text.

Save: Om dokumentet finns sparat, skrivs den gamla filen över. Till exempel om man har öppnat en textfil i Wote, sparas det nya Wote-dokumentet över den öppnade filen. Har man valt New och startat ett nytt dokument och använder sedan Save för att spara, kommer ingen fil att skrivas över, utan man kommer att hamna i samma tillstånd som när man har valt Save As.

Save As: Sparar dokumentet i en textfil i den målmapp man väljer. Man får möjlighet att bläddra genom katalogerna till den plats där man vill lagra filen.

Close: Fungerar på samma sätt som New; textvyn töms på text, så att man kan påbörja ett nytt dokument. Hade Wote tillåtit flikar för att hantera olika textvyer, så hade Close kunnat få en annan

funktion, till exempel att stänga en flik. Close hade då skillt sig från New, som möjligtvis hade öppnat en ny flik i det fallet.

Quit: Avslutar applikationen. Wote stängs. Det finns inga varningar för text som inte är sparad.

Nästa delmeny är Edit, som innehåller endast ett val; Preferences (se Illustration 10).

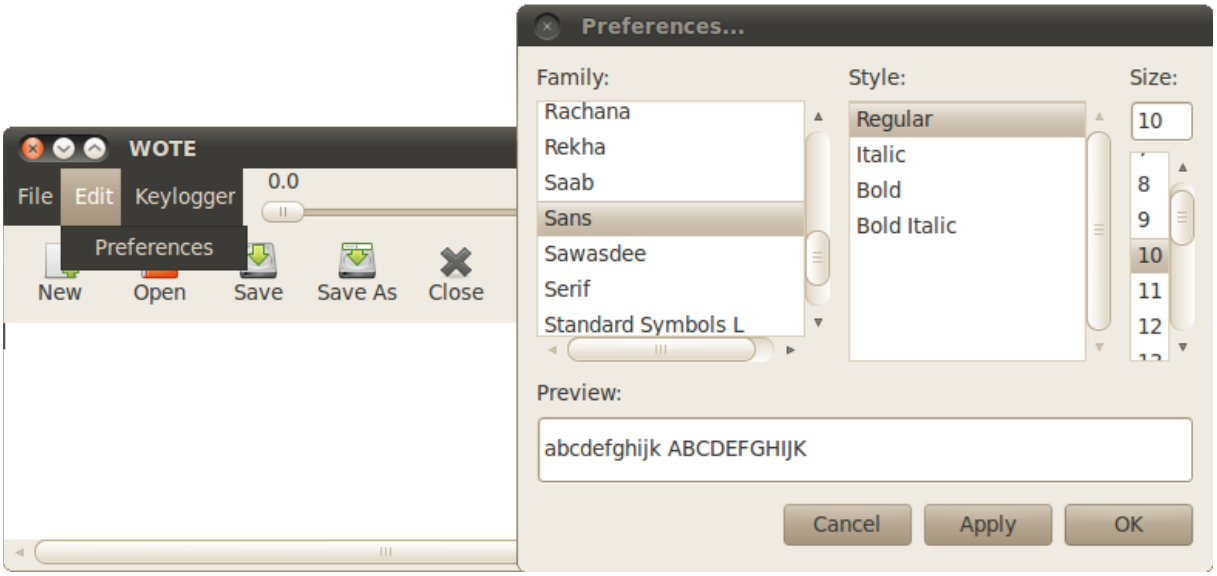

Illustration 10: Edit innehåller valet Preferences, som låter användaren ställa in utseendet på texten.

Preferences: När man väljer Preferences, visas ett fönster där man kan ändra teckeninställningar. Man kan byta font, storlek och välja om texten ska vara kursiv, fet eller normal. Inställningarna gäller för all text i textvyn, så det går inte att blanda olika inställningar.

Sist i menyfältet finns Keylogger, som styr teckenhanteringen för Wote (se Illustration 11).

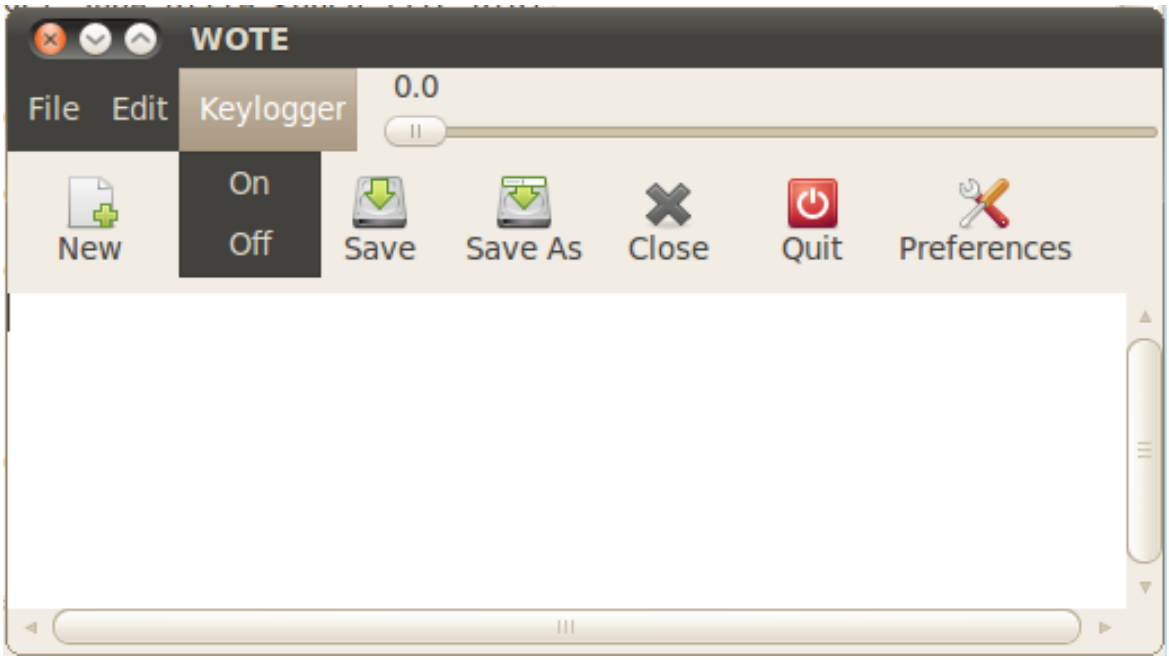

Illustration 11: I menyn kan keyloggern slås på eller av.

On: Är keyloggern av, kan man starta den genom att klicka på On. När keyloggern är på, läser Wote scancodes direkt från tangentbordet. Dessa data används för att avgöra vilka tecken som ska skickas till själva texteditorn. Själva funktionen tillåter användaren att skriva i editorn oberoende av vilken applikation som har fokus. Så länge keyloggern är på, kommer Wote alltid att lägga till inmatade tecken i textvyn. Övriga tangenter, som piltangenter och mellanslag, hanteras på samma sätt.

Off: Är keyloggern på, stänger man av den med Off. Då kommer Wote sluta att läsa direkt från tangenbordet och återgår till att hantera tecken som en vanlig texteditor.

4.1.2 ÖVRIGA VERKTYG

Under menyfältet finns ett verktygsfält. Det innehåller exakt samma funktioner som den övre menyn, förutom keyloggerinställningarna. I verktygsfältet kan man snabbt och enkelt komma åt de funktioner som även finns under File och Edit. Ett verktygsfält förekommer i många andra texteditorer, som exempelvis Gedit[23].

Till höger om menyfältet finns en mätare som visar transparensen. Denna kan även styras för att användaren ska kunna justera transparensen från 0.0 till 1.0. Är detta verktyg markerat, kan man även styra transparensen med piltangenterna.

Nedanför verktygsfältet har vi själva textvyn, där användaren kan arbeta med vanlig text. När keyloggern är på, så skrivs alla inlästa tecken ut i textvyn, så att den fungerar på samma sätt som vid vanlig behandling av text.

Texten i vyn har inga gränser för längd, men det går att "scrolla" horisontellt och vertikalt. Nya ord kommer att hamna på ny rad om de är längre än textvyns bredd. Textvyn stöder även funktioner som att kopiera och klistra in text.

4.1.3 ÖVRIGA FUNKTIONER

För de tre menyerna i menyfältet finns kortkommandon. Dessa är vanliga kortkommandon där man trycker på Alt och sedan den bokstav menynamnet börjar med.

Alt + F → File
Alt + E → Edit
Alt + K → Keylogger

För menyernas innehåll, används Ctrl-tangenten för kortkommando. Close har inget kortkommando, eftersom Ctrl + C brukar användas för att kopiera text och kortkommando för Close verkade inte viktigt, som det till exempel är för Save.

Funktionerna för att få fönstret överst respektive underst finns inte i menyn, utan aktiveras enbart av kortkommandona Ctrl + T och Ctrl + B. Här följer de Ctrl-kortkommandon som finns i Wote:

Ctrl + N → New
Ctrl + O → Open
Ctrl + S → Save
Ctrl + Q → Quit
Ctrl + P → Preferences

Ctrl + T → Window on Top
Ctrl + B → Window to Bottom
Ctrl + K → (Keylogger) On
Ctrl + L → (Keylogger) Off

4.2 ERFARENHETER

Under vårt arbete har vi fått lära oss bland annat hur man programmerar grafiska användargränssnitt i språket C. Vi har sett hur GTK används i olika språk, som Python och C++, men mest har vi lärt oss om hur GTK används i C, det språket som vi har arbetat med.

Våra tidigare erfarenheter med grafiska användargränssnitt är programmering i Java och C#. GUI-programmering med GTK i C innebär att man får mindre stöd från själva biblioteket och de färdiga funktionerna än vad man får i Java och C#. Ett exempel är användingen av textbufferten i GTK, när man vill sätta in text eller flytta markören i textvyn. Här följer exempel på C-kod för att tömma bufferten på text

```
buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(textview));
gtk_text_buffer_get_start_iter(buffer, &(start));
gtk_text_buffer_get_end_iter(buffer, &(iter));
gtk_text_buffer_delete(buffer, &(start), &(iter));
```

Först hämtar man bufferten från textvyn. För att kunna sätta in eller ta bort text behöver man en så kallad iter. Mer om detta förklarades i sektion 3.2.2, där vi gick igenom implementationen av texteditorn. När man har en iter för början av bufferten och en för slutet, kan man använda dessa för att ta bort all text i det intervall dessa iter markerar.

I C#, för att ta bort text ur `RichTextBox`, den textvy som erbjuds av `System.Windows.Forms`, kan man skriva

```
textview.Text = "";
```

Här ser vi referensen `textview` till en textvy, som har egenskapen `Text`, som motsvarar

bufferten i GTK.

Skillnaderna beror främst på att i C programmerar man på lägre nivå och närmare det som faktiskt händer när ett program körs. När man använder egenskapen `Text` i C#, hanteras möjligen någon typ av buffert som inte programmeraren ser.

Det finns mycket att lära sig av att programmera på lägre nivå. GTK har dock ganska många färdiga funktioner, som textvyns funktionalitet och olika dialogfönster för att öppna och spara filer.

Vi har också fått erfarenheter av Linux-miljön, och upplever att olika versioner av Linux skiljer sig mycket, speciellt när det gäller kompatiblitet av olika program. Till exempel när vi skulle testköra en kernelmodul som hanterar interrupts, så hittades inte header-filerna. När vi testade en uppgraderad version av denna kernelmodul, var det fortfarande en del kod som behövdes anpassas till den version av Linux vi använde.

4.3 SAMMANFATTNING

Resultatet gällande applikationen är en enkel texteditor med en inbyggd keylogger-funktion. Texteditorn har menyer och kortkommandon som låter användaren komma åt de olika funktionerna. Under utvecklingen av applikationen har vi lärt oss hur man kan programmera grafiska användargränssnitt i språket C.

5. SLUTSATS

Målet med arbetet har varit att skapa en texteditor som kan hämta tecken från tangentbordet utan att vara i aktivt läge. Tecken ska kunna kopieras till applikationen, men även kunna hämtas utan att någon annan applikation får tillgång till dem. Ytterligare krav har varit att fönstret ska kunna ligga överst och kunna göras transparent. Vi har under arbetet försökt att uppfylla dessa krav.

5.1 UPPFYLLDA KRAV

Av de inkluderade uppgifterna har vi lyckats med att få texteditorn att hämta tecken direkt från tangentbordet. Dock kan Wote inte ta tecken istället för en annan applikation, utan andra program kan fortfarande läsa in tecken som vanligt.

Kravet för ett genomskinligt fönster är också uppfyllt, men funktionen för transparens kan bete sig olika i olika miljöer, beroende på till exempel vilken mjukvara som hanterar fönster.

Att få Wote-applikationen att ligga överst, oberoende av om den är aktiv, har vi också gjort möjligt, och det går även att lägga Wote under de övriga fönster som visas på skärmen. Att justera fönstrets nivå på skärmen var ett av de önskemål som vår uppdragsgivare hade.

Texteditorn har en del kortkommandon för att underlätta användningen av applikationen. Dessa har vi implementerat både i texteditorns normala läge och då keyloggern är på. De kortkommandon som var mest efterfrågade av uppdragsgivaren var kommandon för fönstrets nivå relaterat till andra fönster och även kommandon för starta och stänga av keyloggern.

5.2 KVARSTÅENDE PROBLEM

Det viktigaste kravet var själva keylogger-funktionen, och det var denna funktion vi arbetade mest med. Keylogger-problemet var det första vi försökte lösa, för att komma så långt som möjligt på det området. Det är under arbetet med keyloggern som de flesta problem har uppstått, och tyvärr har inte alla blivit lösta. Det har varit mycket osäkerhet om vi har varit på rätt väg, till exempel om vi skulle behöva programmera en kernelmodul eller ett vanligt C-program. Vi valde att basera lösningen på en befintlig keylogger, kallad Uberkey, eftersom den var enkel att förstå och vi inte hade några tidigare erfarenheter av keyloggers. Det fanns dock vissa risker med Uberkey, eftersom den från början inte fungerade som den skulle. Vi tog därför utmaningen att försöka förbättra denna keylogger innan vi anpassade den till Wote.

Trots de försök till att förbättra keyloggern som gjordes, så återstår fortfarande ett allvarligt problem. Detta problem märktes inte av lika mycket på den dator vi arbetade med som det gjorde på andra

datorer. På vissa datorer hade texteditorn svårt att uppdatera tecken från keyloggern och vid alla test vi gjorde fanns det risk för att piltangenterna låste sig om man höll in dem för länge. Varje gång man startade keyloggern i Wote, gick användningen av CPU upp till 100%.

Vi misstänker att de olika problemen orsakas av att datorn blir överbelastad av keyloggerns process. Det som gör keyloggern krävande att köra, är den `while`-loop, innehållande flera rader kod, som finns i programmet. I denna loop kontrolleras hela tiden indata från tangenbordet. Lösningen till detta problem skulle vara att istället få keyloggern att vara still och vänta på indata. Vi har testat bland annat funktioner som `getch()`, som väntar på tangenttryckningar, men inte lyckats lösa problemet.

Under projektets gång har vi testat två lösningar för Wote-applikationens specialhantering av tecken. Det fanns inte mycket tid till att testa fler lösningar, och därför valde vi att integrera den senaste lösningen med resten av applikationen. Förhoppningsvis finns det möjligheter att förbättra denna lösning, för att uppnå en fullt fungerande Wote-applikation.

Implementeringen av teckenhanteringen i Wote har även stött på ett annat problem. Enligt de krav som vi har fått, ska keyloggern i Wote även kunna hämta tecken på så sätt att andra applikationer inte har tillgång till dem. Det ska gå att, till exempel, skriva i Wote men samtidigt bläddra i ett annat dokument, vilket innebär att vissa tangenbordskommandon ska hanteras av keyloggern och andra kommandon inte ska hanteras på detta sätt, beroende på hur användaren vill ha det. Det vi har kommit fram till, är en keylogger som läser av indata från tangenbordet, men som inte kan hindra att andra applikationer läser indata.

Eftersom textbufferten GTK inte hanterar alla möjliga tecken, så finns det vissa tecken, exempelvis 'ä', 'ö', '£' osv, som inte kan skrivas ut i vyn via keyloggern. Orsaken är att textbufferten bara hanterar tecken av UTF-8-kodning. Trots våra försök att konvertera teckenkodning, har vi inte kunnat lägga till alla specialtecken i bufferten.

5.3 FRAMTIDA ARBETE

Det finns möjligheter att utveckla Wote vidare och rätta till de problem som fortfarande finns i applikationen. Främst är det keyloggern som bör förbättras, men det finns även mycket som kan

göras för att uppgradera själva textredigeraren. Wote texteditor är, bortsett från dess keylogger-funktion, en simpel editor med ganska få funktioner. Anledningen till att vi inte gick så långt med utvecklingen av texteditorn, var att vi försökte koncentrera oss på keyloggern.

För att förbättra keyloggern, behöver främst `while`-loopen ersättas av en bättre lösning, till exempel interrupts, som reagerar på indata från tangentbordet. Man får tänka på att interrupts är ofta maskinberoende.

REFERENSER

[1].[http://en.wikipedia.org/wiki/Text_editor, 2012-05-11]

[2].[http://thc.org/papers/writing-linux-kernel-keylogger.txt , 2012-02-02]

[3].[http://crashcourse.ca/introduction-linux-kernel-programming/lesson-4-writing-and-running-your-first-kernel-module, 2012-02-06]

[4].[http://gnu.ethz.ch/linuks.mine.nu/uberkey/uberkey-1.2/, 2012-02-03]

[5].[A. Brouwer, 2009, http://www.win.tue.nl/~aeb/linux/kbd/scancodes-11.html , 2012-02-22]

[6].[D. S. Lawyer, 2002, http://www.linuxselfhelp.com/HOWTO/Text-Terminal-HOWTO-6.html , 2012-02-15]

[7].[J. Larsson & M. Bengtsson, Laborationsrapport, Avdelingen för datavetenskap, Karlstads Universitet, 2010]

[8].[V. Gite, 2006, http://www.cyberciti.biz/tips/build-linux-kernel-module-against-installed-kernel-source-tree.html , 2012-02-22]

[9].[http://www.gtk.org/, 2012-02-22]

[10].[http://developer.gnome.org/gtk-tutorial/2.90/, 2012-02-22]

[11].[M. A. Aslan, 2005, http://webmail.aast.edu/~maslan/files/pgtk+.pdf , 2012-03-05]

[12].[http://developer.gnome.org/gtk/2.24/, 2012-02-01 --> 2012-]

[13].[http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html, 2012-03-07]

[14].[http://developer.gnome.org/glib/2.30/glib-The-Main-Event-Loop.html, 2012-03-07]

[15].[http://developer.gnome.org/gdk/stable/gdk-Windows.html, 2012-03-12]

[16].[http://www.linuxquestions.org/questions/programming-9/gtk-window-transparency-660677/, 2012-03-12]

[17].[http://cairographics.org/manual/cairo-cairo-t.html#cairo-operator-t , 2012-03-12]

[18]. [A. Brouwer, 2009, http://www.win.tue.nl/~aeb/linux/kbd/scancodes-1.html#ss1.1]

[19]. [http://developer.gnome.org/gobject/unstable/gobject-Closures.html#GClosure , 2012-03-21]

[20].[http://developer.gnome.org/gtk/2.24/gtk-Keyboard-Accelerators.html, 2012-04-03]

[21].[http://en.wikipedia.org/wiki/GIMP, 2012-04-10]

[22]. [http://en.wikipedia.org/wiki/Notepad_%28software%29, 2012-06-05]

[23].[http://en.wikipedia.org/wiki/Gedit, 2012-06-05]

APPENDIX

A1. Exempel på makefile till kernelmodul

Detta är ett exempel på en makefile för att kompilera en kernelmodul. Makefilen genererar en .ko-fil.

```
ifeq ($(KERNELRELEASE),)

KERNELDIR ?= /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

.PHONY: build clean

build:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:

rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c

else

$(info Building with KERNELRELEASE = ${KERNELRELEASE})

obj-m := log
```

B1. Makefile till Wote-lösningen

Detta är makefilen som används för Wote-applikationen. Som syns i första raden, används GTK-biblioteket.

```
GTKFLAGS=-export-dynamic `pkg-config --cflags --libs gtk+-2.0`

testox : test.o ../key.o gui.o
	gcc -o testox test.o ../key.o gui.o $(GTKFLAGS)

test.o : test.c widgets.h
	gcc -c test.c $(GTKFLAGS)

key.o : ../key.c
	gcc -c ../key.c

gui.o : gui.c widgets.h
	gcc -c gui.c $(GTKFLAGS)

clean:
	rm *.o
```

B2. test.c (GTK-delen)

Även om filnamnet inte är så beskrivande, så är detta GUI-delen för Wote-applikationen. Koden innehåller kommunikation med keylogger, GTK-händelser och så vidare.

```
#include <gtk/gtk.h>
#include <stdlib.h>
#include <string.h>
#include "widgets.h"

#ifndef _MAX_FILE_SIZE
#define _MAX_FILE_SIZE 2048000 /* 4 MB */
#endif
```

```
GThread *t1;
const gchar *filename;
int k = 0; // Is keylogger running?
int pos = 0;

int is_logger_on()
{
      return k;
}

/*****************************
 * BEGIN KEYLOGGING FUNCTIONS *
 *****************************/

void* logger(); // Function in key.c

// Refresh the textview:
void refresh_view()
{
      gtk_text_view_set_buffer(GTK_TEXT_VIEW (w->textview), w->buffer);
}

void move_cursor(int i)
{
      int chars = gtk_text_buffer_get_char_count(w->buffer);
      if (i) { pos--; if (pos < 0) pos = 0;}
      else { pos++; if (pos > chars-1) pos = chars-1;}
      gtk_text_buffer_get_iter_at_offset(w->buffer, &(w->iter), chars-
pos);
      gtk_text_buffer_place_cursor(w->buffer, &(w->iter));
}

// Print backspace from keylogger:
void back_space()
{
      gtk_text_view_set_buffer(GTK_TEXT_VIEW (w->textview), w->buffer);
```

```
      /*int chars = gtk_text_buffer_get_char_count(w->buffer);
      gtk_text_buffer_get_iter_at_offset(w->buffer, &(w->iter), chars-
pos);
      gtk_text_buffer_place_cursor(w->buffer, &(w->iter));*/
      w->cursor = gtk_text_buffer_get_insert(w->buffer);
     gtk_text_buffer_get_iter_at_mark (w->buffer, &(w->iter), w->cursor);
      gtk_text_buffer_backspace(w->buffer,&(w->iter),TRUE,TRUE);

}

// Print character from keylogger:
void print_text(char * c)
{
     /*w->buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW (w->textview));
      g_assert(w->buffer != NULL); */
      //refresh_view();
      //char * new_c;
      //int chars = gtk_text_buffer_get_char_count(w->buffer);

      //gtk_text_buffer_get_end_iter(w->buffer, &(w->iter));
      //gtk_text_buffer_insert(w->buffer,&(w->iter),&c,1);
      /*if (!strcmp(c,"/xb6"))
      {
            new_c = g_locale_to_utf8("b",1,NULL,NULL,NULL);
            gtk_text_buffer_insert_at_cursor(w->buffer, "/xb6", 1);
      }*/
      gtk_text_buffer_insert_at_cursor(w->buffer, c, 1);
      g_idle_add((GSourceFunc)refresh_view, NULL);
}

/****************************
 * END KEYLOGGING FUNCTIONS *
 ****************************/

/*****************************
 * BEGIN TEXT EDITOR FUNCTIONS *
 *****************************/
```

```
static gboolean delete_event(GtkWidget *widget, GdkEvent *event, gpointer
data)
{
      gtk_main_quit();
      return FALSE;
}

void set_window_below()
{
      gtk_window_set_keep_above(GTK_WINDOW(w->window), FALSE);
      gtk_window_set_keep_below (GTK_WINDOW(w->window), TRUE);
}

void set_window_above()
{
      gtk_window_set_keep_below (GTK_WINDOW(w->window), FALSE);
      gtk_window_set_keep_above(GTK_WINDOW(w->window), TRUE);
}

void set_transparency(gdouble value)
{
      if (value < 0 || value > 1) return;
      gtk_widget_realize(w->window);
      gdk_window_set_opacity(w->window->window, value);
      gtk_range_set_value(GTK_RANGE(w->opacity), value);
}

void text_edit_increase_opacity()
{
      set_transparency(gtk_range_get_value(GTK_RANGE(w->opacity))+0.1);
}

void text_edit_decrease_opacity()
{
      set_transparency(gtk_range_get_value(GTK_RANGE(w->opacity))-0.1);
}

void text_edit_opacity_min()
```

```
{
      gtk_range_set_value(GTK_RANGE(w->opacity), 0.0);
      set_transparency(0.0);
}

void text_edit_opacity_max()
{
      gtk_range_set_value(GTK_RANGE(w->opacity), 1.0);
      set_transparency(1.0);
}

void font_cancel_clicked(GtkWidget *widget , gpointer data)
{
      gtk_widget_destroy(w->font_dlg);
}

void font_apply_clicked(GtkWidget *widget , gpointer data)
{
      gchar *fontname;
      PangoFontDescription *font_desc;
      fontname =
gtk_font_selection_dialog_get_font_name(GTK_FONT_SELECTION_DIALOG(w-
>font_dlg));
      font_desc = pango_font_description_from_string(fontname);
      gtk_widget_modify_font(w->textview, font_desc);
}

void font_ok_clicked(GtkWidget *widget , gpointer data)
{
      gchar *fontname;
      PangoFontDescription *font_desc;
      fontname =
gtk_font_selection_dialog_get_font_name(GTK_FONT_SELECTION_DIALOG(w-
>font_dlg));
      font_desc = pango_font_description_from_string(fontname);
      gtk_widget_modify_font(w->textview, font_desc);
      gtk_widget_destroy(w->font_dlg);
}
```

```
void text_edit_preferences()
{
     w->font_dlg = gtk_font_selection_dialog_new("Preferences...");
     w->font_dlg_ok = GTK_FONT_SELECTION_DIALOG(w->font_dlg)->ok_button;
     w->font_dlg_cancel = GTK_FONT_SELECTION_DIALOG(w->font_dlg)-
>cancel_button;
     w->font_dlg_apply = GTK_FONT_SELECTION_DIALOG(w->font_dlg)-
>apply_button;
     g_signal_connect(G_OBJECT(w->font_dlg_ok), "clicked",
G_CALLBACK(font_ok_clicked), NULL);
     g_signal_connect(G_OBJECT(w->font_dlg_cancel), "clicked",
G_CALLBACK(font_cancel_clicked), NULL);
     g_signal_connect(G_OBJECT(w->font_dlg_apply), "clicked",
G_CALLBACK(font_apply_clicked), NULL);
     gtk_widget_show_all(w->font_dlg);
}

void saveas_cancel_clicked(GtkWidget *widget , gpointer data)
{
     gtk_widget_destroy(w->saveas_file_dlg);
}

void saveas_ok_clicked(GtkWidget *widget , gpointer data)
{
     FILE *fp;
     gchar *text;
     filename = gtk_file_selection_get_filename(GTK_FILE_SELECTION(w-
>saveas_file_dlg));
     w->buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(w->textview));
     gtk_text_buffer_get_start_iter(w->buffer, &(w->start));
     gtk_text_buffer_get_end_iter(w->buffer, &(w->iter));
     text = gtk_text_buffer_get_text(w->buffer, &(w->start), &(w->iter),
TRUE);
     fp = fopen(filename, "w");
     if (!fp) return;
     fwrite(text, strlen(text), 1, fp);
     fclose(fp);
```

```
	gtk_widget_destroy(w->saveas_file_dlg);
}

void text_edit_saveas()
{
	w->saveas_file_dlg = gtk_file_selection_new("Save File As...");
	w->saveas_file_dlg_ok = GTK_FILE_SELECTION(w->saveas_file_dlg)-
>ok_button;
	w->saveas_file_dlg_cancel = GTK_FILE_SELECTION(w->saveas_file_dlg)-
>cancel_button;
	g_signal_connect(G_OBJECT(w->saveas_file_dlg_ok), "clicked",
G_CALLBACK(saveas_ok_clicked), NULL);
	g_signal_connect(G_OBJECT(w->saveas_file_dlg_cancel), "clicked",
G_CALLBACK(saveas_cancel_clicked), NULL);
	gtk_widget_show_all(w->saveas_file_dlg);
}

void text_edit_save()
{
	if (filename == NULL)
	{
		text_edit_saveas();
	}
	else
	{
		FILE *fp;
		gchar *text;
		w->buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(w-
>textview));
		gtk_text_buffer_get_start_iter(w->buffer, &(w->start));
		gtk_text_buffer_get_end_iter(w->buffer, &(w->iter));
		text = gtk_text_buffer_get_text(w->buffer, &(w->start), &(w-
>iter), TRUE);
	fp = fopen(filename, "w");
	if (!fp) return;
	fwrite(text, strlen(text), 1, fp);
	fclose(fp);
	}
```

```
}

void open_cancel_clicked(GtkWidget *widget , gpointer data)
{
      gtk_widget_destroy(w->open_file_dlg);
}

void text_edit_close()
{
      filename = NULL;
      w->buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(w->textview));
      gtk_text_buffer_get_start_iter(w->buffer, &(w->start));
      gtk_text_buffer_get_end_iter(w->buffer, &(w->iter));
      gtk_text_buffer_delete(w->buffer, &(w->start), &(w->iter));
      g_idle_add((GSourceFunc)refresh_view, NULL);
}

void open_ok_clicked(GtkWidget *widget , gpointer data)
{
      int bytes_read;
      FILE *fp;
      gchar text[_MAX_FILE_SIZE];
      filename = gtk_file_selection_get_filename(GTK_FILE_SELECTION(w-
>open_file_dlg));
      if ((fp = fopen(filename, "r")) == NULL) exit(1);
      while (!feof(fp)) bytes_read = fread(&text, sizeof(gchar),
_MAX_FILE_SIZE, fp);
      fclose(fp);

      text_edit_close();
      w->buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(w->textview));
      gtk_text_buffer_get_start_iter(w->buffer, &(w->start));
      gtk_text_buffer_get_end_iter(w->buffer, &(w->iter));
      gtk_text_buffer_delete(w->buffer, &(w->start), &(w->iter));
      gtk_text_buffer_insert(w->buffer, &(w->iter), text, bytes_read);
      gtk_widget_destroy(w->open_file_dlg);
}
```

```
void text_edit_open()
{
     w->open_file_dlg = gtk_file_selection_new("Open File . . .");
     w->open_file_dlg_ok = GTK_FILE_SELECTION(w->open_file_dlg)-
>ok_button;
     w->open_file_dlg_cancel = GTK_FILE_SELECTION(w->open_file_dlg)-
>cancel_button;
     g_signal_connect(G_OBJECT(w->open_file_dlg_ok), "clicked",
G_CALLBACK(open_ok_clicked), NULL);
     g_signal_connect(G_OBJECT(w->open_file_dlg_cancel), "clicked",
G_CALLBACK(open_cancel_clicked), NULL);
     gtk_widget_show_all(w->open_file_dlg);
}

void text_edit_quit() { exit(0); }

void text_edit_new() { text_edit_close(); }

void keylogger_off()
{
     k = 0;
     gtk_text_view_set_editable( GTK_TEXT_VIEW(w->textview), 1);
}

void text_edit_keylogger_off()
{
     if (!k) return;
     keylogger_off();
     g_thread_join(t1);
}

void text_edit_keylogger_on()
{
     if (k) return;
     //pthread_t t1, t2;

     k = 1;
```

```
      //g_thread_init(NULL);
     gdk_threads_init();


      gtk_text_view_set_editable( GTK_TEXT_VIEW(w->textview), 0);
      t1 = g_thread_create((GThreadFunc)logger, NULL, TRUE, NULL);
      //int iret1 = pthread_create(&t1, NULL, logging, NULL);
}

/*****************************
 * END TEXT EDITOR FUNCTIONS *
 *****************************/

/****************
 * BEGIN EVENTS *
 ****************/

// Menu events:
void menu_item_new_activated(GtkWidget *widget, gpointer data)
{ text_edit_new();}
void menu_item_open_activated(GtkWidget *widget, gpointer data)
{ text_edit_open();}
void menu_item_save_activated(GtkWidget *widget, gpointer data)
{ text_edit_save(); }
void menu_item_saveas_activated(GtkWidget *widget, gpointer data)
{ text_edit_saveas(); }
void menu_item_close_activated(GtkWidget *widget, gpointer data)
{ text_edit_close(); }
void menu_item_quit_activated(GtkWidget *widget, gpointer data)
{ text_edit_quit(); }
void menu_item_preferences_activated(GtkWidget *widget, gpointer data)
{ text_edit_preferences(); }
void menu_item_keylogger_on_activated(GtkWidget *widget, gpointer data)
{ text_edit_keylogger_on(); }
void menu_item_keylogger_off_activated(GtkWidget *widget, gpointer data)
{ text_edit_keylogger_off(); }

// Button events:
```

```
void button_new_clicked(GtkWidget *widget, gpointer data)
{ text_edit_new();}
void button_open_clicked(GtkWidget *widget, gpointer data)
{ text_edit_open();}
void button_save_clicked(GtkWidget *widget, gpointer data)
{ text_edit_save(); }
void button_saveas_clicked(GtkWidget *widget, gpointer data)
{ text_edit_save(); }
void button_close_clicked(GtkWidget *widget, gpointer data)
{ text_edit_close(); }
void button_quit_clicked(GtkWidget *widget, gpointer data)
{ text_edit_quit(); }
void button_preferences_clicked(GtkWidget *widget, gpointer data)
{ text_edit_preferences(); }

// Scale event:
void opacity_scale_moved(GtkWidget *widget, gpointer data)
{ set_transparency(gtk_range_get_value(GTK_RANGE(w->opacity))); }

// Hotkey events:
void hotkey_new(GtkObject *object, gpointer data)
{ text_edit_new(); }
void hotkey_open(GtkObject *object, gpointer data)
{ text_edit_open(); }
void hotkey_save(GtkObject *object, gpointer data)
{ text_edit_save(); }
void hotkey_quit(GtkObject *object, gpointer data)
{ text_edit_quit(); }
void hotkey_preferences(GtkObject *object, gpointer data)
{ text_edit_preferences(); }
void hotkey_set_window_above(GtkObject *object, gpointer   data)
{ set_window_above(); }
void hotkey_set_window_below(GtkObject *object, gpointer   data)
{ set_window_below(); }
void keylogger_activated(GtkObject *object, gpointer data)
{ text_edit_keylogger_on(); }
void keylogger_deactivated(GtkObject *object, gpointer data)
{ text_edit_keylogger_off(); }
```

```
void opacity_max(GtkObject *object, gpointer   data)
{ text_edit_opacity_max(); }
void opacity_min(GtkObject *object, gpointer   data)
{ text_edit_opacity_min(); }
void increase_opacity(GtkObject *object, gpointer   data)
{ text_edit_increase_opacity(); }
void decrease_opacity(GtkObject *object, gpointer   data)
{ text_edit_decrease_opacity(); }

void events()
{
     // Menu events:
     g_signal_connect(G_OBJECT(w->menu_item_new), "activate",
G_CALLBACK(menu_item_new_activated), NULL);
     g_signal_connect(G_OBJECT(w->menu_item_open), "activate",
G_CALLBACK(menu_item_open_activated), NULL);
     g_signal_connect(G_OBJECT(w->menu_item_save), "activate",
G_CALLBACK(menu_item_save_activated), NULL);
     g_signal_connect(G_OBJECT(w->menu_item_saveas), "activate",
G_CALLBACK(menu_item_saveas_activated), NULL);
     g_signal_connect(G_OBJECT(w->menu_item_close), "activate",
G_CALLBACK(menu_item_close_activated), NULL);
     g_signal_connect(G_OBJECT(w->menu_item_quit), "activate",
G_CALLBACK(menu_item_quit_activated), NULL);
     g_signal_connect(G_OBJECT(w->menu_item_preferences), "activate",
G_CALLBACK(menu_item_preferences_activated), NULL);
     g_signal_connect(G_OBJECT(w->menu_item_keylogger_on), "activate",
G_CALLBACK(menu_item_keylogger_on_activated), NULL);
     g_signal_connect(G_OBJECT(w->menu_item_keylogger_off), "activate",
G_CALLBACK(menu_item_keylogger_off_activated), NULL);

     // Toolbar events:
     g_signal_connect(G_OBJECT(w->button_new), "clicked",
G_CALLBACK(button_new_clicked), NULL);
     g_signal_connect(G_OBJECT(w->button_open), "clicked",
G_CALLBACK(button_open_clicked), NULL);
     g_signal_connect(G_OBJECT(w->button_save), "clicked",
G_CALLBACK(button_save_clicked), NULL);
```

```
	g_signal_connect(G_OBJECT(w->button_saveas), "clicked",
G_CALLBACK(button_saveas_clicked), NULL);
	g_signal_connect(G_OBJECT(w->button_close), "clicked",
G_CALLBACK(button_close_clicked), NULL);
	g_signal_connect(G_OBJECT(w->button_quit), "clicked",
G_CALLBACK(button_quit_clicked), NULL);
	g_signal_connect(G_OBJECT(w->button_preferences), "clicked",
G_CALLBACK(button_preferences_clicked), NULL);

	g_signal_connect(w->opacity, "value_changed",
G_CALLBACK(opacity_scale_moved), NULL);
	g_signal_connect(w->window, "delete-event", G_CALLBACK
(delete_event), w->textview);
}

void hotkeys()
{
	GtkAccelGroup *accel;
	GClosure *closure;

	accel = gtk_accel_group_new();

	closure = g_cclosure_new (G_CALLBACK (hotkey_new), NULL, NULL);
	gtk_accel_group_connect(accel, gdk_keyval_from_name("n"),
GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE, closure);
	g_closure_unref (closure);
	closure = g_cclosure_new (G_CALLBACK (hotkey_open), NULL, NULL);
	gtk_accel_group_connect(accel, gdk_keyval_from_name("o"),
GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE, closure);
	g_closure_unref (closure);
	closure = g_cclosure_new (G_CALLBACK (hotkey_save), NULL, NULL);
	gtk_accel_group_connect(accel, gdk_keyval_from_name("s"),
GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE, closure);
	g_closure_unref (closure);
	closure = g_cclosure_new (G_CALLBACK (hotkey_quit), NULL, NULL);
	gtk_accel_group_connect(accel, gdk_keyval_from_name("q"),
GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE, closure);
	g_closure_unref (closure);
```

```
      closure = g_cclosure_new (G_CALLBACK (hotkey_preferences), NULL,
NULL);
      gtk_accel_group_connect(accel, gdk_keyval_from_name("p"),
GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE, closure);
      g_closure_unref (closure);

      // Window shortcut
      closure = g_cclosure_new (G_CALLBACK (hotkey_set_window_above),
NULL, NULL);
      gtk_accel_group_connect(accel, gdk_keyval_from_name("t"),
GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE, closure);
      g_closure_unref (closure);
      closure = g_cclosure_new (G_CALLBACK (hotkey_set_window_below),
NULL, NULL);
      gtk_accel_group_connect(accel, gdk_keyval_from_name("b"),
GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE, closure);
      g_closure_unref (closure);

      // Keylogger hotkeys:
      closure = g_cclosure_new (G_CALLBACK (keylogger_activated), NULL,
NULL);
      gtk_accel_group_connect(accel, gdk_keyval_from_name("k"),
GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE, closure);
      g_closure_unref (closure);
      closure = g_cclosure_new (G_CALLBACK (keylogger_deactivated), NULL,
NULL);
      gtk_accel_group_connect(accel, gdk_keyval_from_name("l"),
GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE, closure);
      g_closure_unref (closure);

      /*
      #define GDK_Left 0xff51
      #define GDK_Up 0xff52
      #define GDK_Right 0xff53
      #define GDK_Down 0xff54

      */
```

```
      // Transparency:
      closure = g_cclosure_new (G_CALLBACK (opacity_max), NULL, NULL);
      gtk_accel_group_connect(accel, 0xff52, GDK_CONTROL_MASK,
GTK_ACCEL_VISIBLE, closure);
      g_closure_unref (closure);
      closure = g_cclosure_new (G_CALLBACK (opacity_min), NULL, NULL);
      gtk_accel_group_connect(accel, 0xff54, GDK_CONTROL_MASK,
GTK_ACCEL_VISIBLE, closure);
      g_closure_unref (closure);
      closure = g_cclosure_new (G_CALLBACK (increase_opacity), NULL,
NULL);
      gtk_accel_group_connect(accel, 0xff53, GDK_CONTROL_MASK,
GTK_ACCEL_VISIBLE, closure);
      g_closure_unref (closure);
      closure = g_cclosure_new (G_CALLBACK (decrease_opacity), NULL,
NULL);
      gtk_accel_group_connect(accel, 0xff51, GDK_CONTROL_MASK,
GTK_ACCEL_VISIBLE, closure);
      g_closure_unref (closure);

      gtk_window_add_accel_group (GTK_WINDOW(w->window), accel);
}

/**************
 * END EVENTS *
 **************/

int main(int argc, char** argv)
{
      gtk_init (&argc, &argv);
      void gui();
      gui();                // gui.c
      hotkeys();
      events();

      gtk_main ();
    return 0;
}
```

B3. gui.c (Widgets)

Här finns koden för alla Widgets. Händelserna kopplade till dem finns i test.c.

```
#include <gtk/gtk.h>
#include <stdlib.h>
#include "widgets.h"
//#include "event_fun.h"

void gui()
{
      w = malloc(sizeof(struct widget_struct));
      w = g_slice_new(widgets);

      w->window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
      w->scrolled_window = gtk_scrolled_window_new(NULL, NULL);
      w->textview = gtk_text_view_new();
      w->buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(w->textview));
//gtk_text_buffer_new(NULL);
      w->vbox = gtk_vbox_new(FALSE, 0);
      w->hbox = gtk_hbox_new(FALSE, 10);
      w->opacity = gtk_hscale_new_with_range(0.0, 1.0, 0.1);

      // Menu:
      w->menu_bar = gtk_menu_bar_new();
      w->menu_item_file = gtk_menu_item_new_with_mnemonic(" _File");
      w->menu_item_edit = gtk_menu_item_new_with_mnemonic(" _Edit");
      w->menu_item_keylogger = gtk_menu_item_new_with_mnemonic("
_Keylogger");
      w->menu_file = gtk_menu_new();
      w->menu_edit = gtk_menu_new();
      w->menu_keylogger = gtk_menu_new();

      // Menu items:
     w->menu_item_new = gtk_image_menu_item_new_from_stock(GTK_STOCK_NEW,
NULL);
      w->menu_item_open =
```

```
gtk_image_menu_item_new_from_stock(GTK_STOCK_OPEN, NULL);
     w->menu_item_save =
gtk_image_menu_item_new_from_stock(GTK_STOCK_SAVE, NULL);
     w->menu_item_saveas =
gtk_image_menu_item_new_from_stock(GTK_STOCK_SAVE_AS, NULL);
     w->menu_item_close =
gtk_image_menu_item_new_from_stock(GTK_STOCK_CLOSE, NULL);
     w->menu_item_quit =
gtk_image_menu_item_new_from_stock(GTK_STOCK_QUIT, NULL);
     w->menu_item_preferences =
gtk_image_menu_item_new_from_stock(GTK_STOCK_PREFERENCES, NULL);
    w->menu_item_keylogger_on = gtk_image_menu_item_new_from_stock("On",
NULL);
     w->menu_item_keylogger_off =
gtk_image_menu_item_new_from_stock("Off", NULL);

     // Toolbar items:
     w->tool_bar = gtk_toolbar_new();
     w->button_new = gtk_toolbar_insert_stock(GTK_TOOLBAR(w->tool_bar),
GTK_STOCK_NEW, "New File", NULL, NULL, NULL, -1);
     w->button_open = gtk_toolbar_insert_stock(GTK_TOOLBAR(w->tool_bar),
GTK_STOCK_OPEN, "Open File", NULL, NULL, NULL, -1);
     w->button_save = gtk_toolbar_insert_stock(GTK_TOOLBAR(w->tool_bar),
GTK_STOCK_SAVE, "Save File", NULL, NULL, NULL, -1);
     w->button_saveas = gtk_toolbar_insert_stock(GTK_TOOLBAR(w-
>tool_bar), GTK_STOCK_SAVE_AS, "Save As File", NULL, NULL, NULL, -1);
    w->button_close = gtk_toolbar_insert_stock(GTK_TOOLBAR(w->tool_bar),
GTK_STOCK_CLOSE, "Close File", NULL, NULL, NULL, -1);
     w->button_quit = gtk_toolbar_insert_stock(GTK_TOOLBAR(w->tool_bar),
GTK_STOCK_QUIT, "Quit", NULL, NULL, NULL, -1);
     w->button_preferences = gtk_toolbar_insert_stock(GTK_TOOLBAR(w-
>tool_bar), GTK_STOCK_PREFERENCES, "Preferences", NULL, NULL, NULL, -1);

     gtk_window_set_title(GTK_WINDOW(w->window), "WOTE");
     gtk_window_set_keep_above(GTK_WINDOW(w->window),TRUE);
     gtk_text_view_set_wrap_mode(GTK_TEXT_VIEW(w->textview),
GTK_WRAP_WORD);
     gtk_container_set_border_width(GTK_CONTAINER(w->window), 10);
```

```
      gtk_widget_set_size_request(w->window,512,256);
      gtk_toolbar_set_style(GTK_TOOLBAR(w->tool_bar), GTK_TOOLBAR_BOTH);

      // Put the widgets together.
      gtk_container_add(GTK_CONTAINER(w->window),w->vbox);
      gtk_box_pack_start(GTK_BOX(w->hbox), w->menu_bar, FALSE, FALSE, 0);
      gtk_box_pack_start(GTK_BOX(w->hbox), w->opacity, TRUE,TRUE, 0);
      gtk_box_pack_start(GTK_BOX(w->vbox), w->hbox, FALSE, FALSE, 0);
      gtk_container_add(GTK_CONTAINER(w->menu_bar), w->menu_item_file);
      gtk_container_add(GTK_CONTAINER(w->menu_bar), w->menu_item_edit);
      gtk_container_add(GTK_CONTAINER(w->menu_bar), w-
>menu_item_keylogger);
      gtk_menu_item_set_submenu(GTK_MENU_ITEM(w->menu_item_file), w-
>menu_file);
      gtk_menu_item_set_submenu(GTK_MENU_ITEM(w->menu_item_edit), w-
>menu_edit);
      gtk_menu_item_set_submenu(GTK_MENU_ITEM(w->menu_item_keylogger), w-
>menu_keylogger);
      gtk_container_add(GTK_CONTAINER(w->menu_file), w->menu_item_new);
      gtk_container_add(GTK_CONTAINER(w->menu_file), w->menu_item_open);
      gtk_container_add(GTK_CONTAINER(w->menu_file), w->menu_item_save);
     gtk_container_add(GTK_CONTAINER(w->menu_file),  w->menu_item_saveas);
      gtk_container_add(GTK_CONTAINER(w->menu_file), w->menu_item_close);
      gtk_container_add(GTK_CONTAINER(w->menu_file), w->menu_item_quit);
      gtk_container_add(GTK_CONTAINER(w->menu_edit), w-
>menu_item_preferences);
      gtk_container_add(GTK_CONTAINER(w->menu_keylogger), w-
>menu_item_keylogger_on);
      gtk_container_add(GTK_CONTAINER(w->menu_keylogger), w-
>menu_item_keylogger_off);
      gtk_box_pack_start(GTK_BOX(w->vbox), w->tool_bar, FALSE, FALSE, 0);
     gtk_box_pack_start(GTK_BOX(w->vbox), w->scrolled_window, TRUE, TRUE,
0);
      gtk_container_add(GTK_CONTAINER(w->scrolled_window),w->textview);
      gtk_container_border_width (GTK_CONTAINER (w->window), 0);

      gtk_widget_show_all (w->window);
}
```

B4. test.h

Här är bland annat funktioner som anropas av keyloggern.

```
#ifndef TEST_H
#define TEST_H

#define nfound -1

void back_space();
void print_text(char * c);
int kill_thread();
void move_cursor(int i);
void text_edit_opacity_max();
void text_edit_opacity_min();
void text_edit_increase_opacity();
void text_edit_decrease_opacity();
void keylogger_off();
void set_window_above();
void set_window_below();
//void * appli_init( void *w);

#endif
```

B5. widgets.h

Här finns alla Widgets definierade. De används av gui.c och test.c.

```
#ifndef WIDGETS_H
#define WIDGETS_H

#define nfound -1

#include <gtk/gtk.h>
```

```
typedef struct widget_struct
{
      GtkWidget *window;
      GtkWidget *scrolled_window;
      GtkWidget *opacity;
      GtkWidget *vbox, *hbox;
      GtkTextMark *last_pos, *cursor;
      GtkWidget *textview;
      GtkTextBuffer *buffer;
      GtkTextIter iter, start;
      // Menu items:
      GtkWidget *menu_bar;
      GtkWidget *menu_item_file, *menu_item_edit, *menu_item_keylogger;
      GtkWidget *menu_file, *menu_edit, *menu_keylogger;
      GtkWidget *menu_item_new;
      GtkWidget *menu_item_open;
      GtkWidget *menu_item_save;
      GtkWidget *menu_item_saveas;
      GtkWidget *menu_item_close;
      GtkWidget *menu_item_quit;
      GtkWidget *menu_item_preferences;
      GtkWidget *menu_item_keylogger_on, *menu_item_keylogger_off;
      // Toolbar items:
      GtkWidget *tool_bar;
      GtkWidget *button_new;
      GtkWidget *button_open;
      GtkWidget *button_save;
      GtkWidget *button_saveas;
      GtkWidget *button_close;
      GtkWidget *button_quit;
      GtkWidget *button_preferences;
      // Dialogues:
      GtkWidget *open_file_dlg, *open_file_dlg_ok, *open_file_dlg_cancel;
      GtkWidget *saveas_file_dlg, *saveas_file_dlg_ok,
*saveas_file_dlg_cancel;
      GtkWidget *font_dlg, *font_dlg_ok, *font_dlg_cancel,
*font_dlg_apply;
```

```
}widgets;

widgets *w;

#endif
```

B6. eventfun.h

Definition av händelser, som är kopplade till Widgets.

```
#ifndef EVENT_FUN_H
#define EVENT_FUN_H

#define nfound -1

#include <gtk/gtk.h>

void menu_item_new_activated(GtkWidget *widget, gpointer data);
void menu_item_open_activated(GtkWidget *widget, gpointer data);
void menu_item_save_activated(GtkWidget *widget, gpointer data);
void menu_item_saveas_activated(GtkWidget *widget, gpointer data);
void menu_item_close_activated(GtkWidget *widget, gpointer data);
void menu_item_quit_activated(GtkWidget *widget, gpointer data);
void menu_item_preferences_activated(GtkWidget *widget, gpointer data);
void menu_item_keylogger_on_activated(GtkWidget *widget, gpointer data);
void menu_item_keylogger_off_activated(GtkWidget *widget, gpointer data);
void button_new_clicked(GtkWidget *widget, gpointer data);
void button_open_clicked(GtkWidget *widget, gpointer data);
void button_save_clicked(GtkWidget *widget, gpointer data);
void button_saveas_clicked(GtkWidget *widget, gpointer data);
void button_close_clicked(GtkWidget *widget, gpointer data);
void button_quit_clicked(GtkWidget *widget, gpointer data);
void button_preferences_clicked(GtkWidget *widget, gpointer data);
void opacity_scale_moved(GtkWidget *widget, gpointer data);
void keylogger_activated(GtkObject *object, gpointer data);
void opacity_max(GtkObject *object, gpointer data);
```

```
#endif
```

C1. key.c

Här har vi keyloggern.

```
#include <time.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/param.h>
#include <stdio.h>
#include "GUI/test.h"

#ifdef BSD

#include <err.h>
#include <machine/sysarch.h>
#include <machine/cpufunc.h>
#else
#include <sys/io.h>
#endif

void* logger()
{

    if (getuid()!=0) {
           printf("Must be root to run, you're only %u.\n",getuid());
        //printf("Must be root to run %s, you're
only %u.\n",argv[0],getuid());
        return; //1;
    }

#ifdef BSD
    if (i386_set_ioperm(0x60,5,1) == -1)
```

```
        err(1,"i386_set_ioperm failed");
#else
    iopl(3);
#endif

     char keys[] = { "  1234567890+@  qwertyuiop@@  asdfghjkl@@@
zxcvbnm,.-" };
     char shiftkeys[] = { "  !\"#@%&/()=?@  QWERTYUIOP@@  ASDFGHJKL@@@
ZXCVBNM;:_" };

     unsigned char c;
     short shift = 0, ctrl = 0, alt = 0;

     while(1)
     {
          printf("key pressed");
          while((inb(0x64)&32));
          c=inb(0x60);
          outb(0x64,0x60);
          outb(0x64,0xfe);
          switch(c)
          {
               case 0x0e: // Backspace
                    gdk_threads_enter(); back_space();
gdk_threads_leave();
               break;
               case 0x1c: // Enter
                    gdk_threads_enter(); print_text("\n");
gdk_threads_leave();
               break;
               case 0x39: // Space
                    gdk_threads_enter(); print_text(" ");
gdk_threads_leave();
               break;
               case 0x0f: // Tab
                    if (!alt)
                    {
                         gdk_threads_enter();
```

```
                print_text("\t");
                gdk_threads_leave();
        }
break;
case 0x3a: // CAPS LOCK
        shift = ~shift;
break;
case 0x2a:      // Left Shift pressed
case 0x36: // Right Shift pressed
        shift = ~shift;
break;
case 0xaa:      // Left Shift released
case 0xb6: // Right Shift released
        shift = ~shift;
break;
case 0x1d: // Ctrl pressed
        ctrl = 1;
break;
case 0x9d: // Ctrl released
        ctrl = 0;
break;
case 0x38: // Alt pressed
        alt = 1;
break;
case 0xb8: // Alt released
        alt = 0;
break;
default:
        if (c<54 && c>1)
        {
                if (shift)
                {
                        gdk_threads_enter();
                        print_text(&shiftkeys[c]);
                        gdk_threads_leave();
                }
                else
                {
```

```
            if (!alt && !ctrl)
            {
                  gdk_threads_enter();
                  print_text(&keys[c]);
                  gdk_threads_leave();
            }
            // Keylogger off:
            else if (c == 0x26 && ctrl)
            {
                  gdk_threads_enter();
                  keylogger_off();
                  gdk_threads_leave();
                  return;
            }
            // Window top or bottom:
            else if (c == 0x14 && ctrl)
            {
                  gdk_threads_enter();
                  set_window_above();
                  gdk_threads_leave();
            }
            else if (c == 0x30 && ctrl)
            {
                  gdk_threads_enter();
                  set_window_below();
                  gdk_threads_leave();
            }
      }
}
// Opacity max:
else if (c == 0x48 && ctrl)
{
      gdk_threads_enter();
      text_edit_opacity_max();
      gdk_threads_leave();
}
// Opacity min:
else if (c == 0x50 && ctrl)
```

```
        {
            gdk_threads_enter();
            text_edit_opacity_min();
            gdk_threads_leave();
        }
        // Increase opacity or move cursor right:
        else if (c == 0x4d)
        {
            if (ctrl)
            {
                gdk_threads_enter();
                text_edit_increase_opacity();
                gdk_threads_leave();
            }
            else
            {
                gdk_threads_enter();
                move_cursor(1);
                gdk_threads_leave();
            }
        }
        // Decrease opacity or move cursor left:
        else if (c == 0x4b)
        {
            if (ctrl)
            {
                gdk_threads_enter();
                text_edit_decrease_opacity();
                gdk_threads_leave();
            }
            else
            {
                gdk_threads_enter();
                move_cursor(0);
                gdk_threads_leave();
            }
        }
}
```

```
            fflush(0);
            usleep(1000);
            if (!is_logger_on()) return;
        }
    }
```