STACKPOINTER

3-1987. Organ för Datorföreningen STACKEN, KTH.

TORSDAG

7

MAJ

Macafton i sal E7
klockan 1900.

MÅNDAG

8

JUNI

iskt högskole-
föreningsmöte
öping.

SÖNDAG

7

JUNI

liskt högskole-
föreningsmöte
öping.

LÖRDAG

6

JUNI

Nordiskt högskole-
datorföreningsmöte
i Linköping.

Datorföreningen STACKEN

STACKEN är datorföreningen på KTH. Vi har en mängd intressanta verksamheter på gång. Dock hänger det ytterst på den enskilde medlemmen att avgöra vad han eller hon vill göra för föreningen. STACKEN är en ideell förening, där intresse för datorer är den gemensamma faktorn. Sedan föreningen grundades 1978 har vi (bland annat) åstadkommit:

- samköp av mikrodatorbyggsatser
- kurser, föredrag och studiebesök
- AMIS, den portabla EMACS-kompatibla editorn för TOPS-10, VMS, PRIME, NORD, ...
- STACKPOINTER, vår tidning
- en egen datorhall för vår DEC-10 och våra andra stordatorer

Sedan ett och ett halvt år har vi en egen DEC-10, som vi installerat, felsökt och kör på. Det är en gammal modell med KA10-processor. Vi kallar henne KATIA. Hon står i vår maskinhall "B30", på Brinellvägen 30 (V-sektionen) på gaveln mot Lill-Jansskogen (där vi har en egen ingång). En våning ovanför finns en hörsal (V4), där vi håller till vid större möten.

Ordinarie möten är första torsdag i varje månad, kl 19, vid datorhallen eller i hörsalen. Är Du intresserad av föreningen, är Du välkommen till något av våra möten. Vill Du sedan bli medlem, lämnar Du en skriftlig ansökan till styrelsen eller skickar den till vår postadress.

STACKPOINTER

STACKPOINTER är organ för Datorföreningen STACKEN på KTH. Den utkommer när material i tillräcklig mängd finns, förhoppningsvis 4–5 gånger per år. Återgivande av delar av innehållet är tillåtet när källan anges.

Redaktör: Hans Nordström
I redaktionen: Jan Michael Rynning och Mats O Jansson
Tillika ansvarig utgivare: Mats O Jansson

Färdigställd: 1987-04-23

I detta nummer

Macafton 5
STACKEN ordnar Macintoshafton i E7 torsdagen den 7 maj klockan 19.
Nordiskt datorföreningsmöte 6
STACKEN och CD ordnar nordiskt högskoledatorföreningsmöte i Linköping i pingst (6–8 juni).
GNU's Bulletin #2 10
GNU's Task List 22
Lista över de mest angelägna arbetsuppgifterna i GNU-projektet. Finns där något du kan hjälpa till med?
RMS föredrag 1986 24
Föredraget som Richard M. Stallman höll när han besökte KTH.
Bruksanvisning 60

Adresser m m

POST till föreningen skickas till nedanstående adress eller läggs i postfacket på NADA.

Datorföreningen STACKEN
c/o NADA
KTH
100 44 STOCKHOLM

Medlemsavgift: 87 kronor för 1987

Ordf: Stellan Lagerström {08-787 78 14 (arb) 08-63 68 25 (hem)
Sekr: Olle Betzén {0758-318 48 (hem)
Kass: Mats O Jansson {08-34 93 20 (arb) 08-711 70 37 (hem)
Red: Hans Nordström {0760-615 80 (arb)
Hexm: Henrik Björkman {08-30 89 01 (hem)
Suppl: Henning Croona {08-799 80 74 (arb) 08-739 17 40 (hem)
Suppl: Thord Nilson {08-712 00 90 (arb) 08-749 21 92 (hem)

Postgiro: 433 01 15-9
Bankgiro: 344-3595

Klubblokal: Osquars Backe 4 (ingång från valvet, Osquars Backe 1)
Datorhall: Brinellvägen 32 (på gaveln mot Lill-Jansskogen)

SUNET DECnet: STACKEN@VERA (VMSmail: VERA::STACKEN)
UUCP: {seismo,mcvax}!enea!ttds!stacken
EAN: stacken@cs.kth.sunet
PSI: PSI%0240200101905::VERA::STACKEN
ARPA: enea!ttds!stacken@seismo.CSS.GOV
CSnet: enea.uucp!ttds!stacken@chalmers.csnet

Redaktörn

BÖRJAN AV APRIL hade I*M presskonferens och presenterade sin nya produktlinje av persondatorer. Nu blev det klart att det skall vara 3,5"-disketter framöver. Vill man använda 5 1/4"-disketter så finns det givetvis att köpa som tillbehör. Till de laserskrivare som också lanserades kommer sidbeskrivningsspråket POSTSCRIPT att användas. En reaktion som noterades dagen efter på Wall Street var att Apples aktier gick upp 7%. Även Tandy och Compaq kunde glädja sig åt stigande aktiekurs. Hur gick det då med I*M-aktien? Den föll med 2%!

Så kan det gå.

En kväll härförleden stannade jag kvar på arbetet. (Övertid kallas det.) Lite problem med den mänskliga faktorn och systemen. Sitter alltså vid terminalen och glaner på programkoden. Jag blev lite försjunken i hur det hela fungerade. Vaknar upp (nä, jag sov inte) vid att jag hör städerskan komma släpande på dammsugaren i korridoren. Innan jag reagerade så stoppade hon stickkontakten i urtaget, där firmans Mac också var ansluten och igång med dammsugaren. Mycket försiktigt närmade jag mig Macen. Jodå, det var inga problem med den.

Hur många startar dammsugaren från samma urtag som datorn är ansluten till?

Ja, det var det hele.

/hn

Här kommer tet som föll bort i förra utgåvan

t

/hn

Macafton

TORSDAGEN DEN 7 MAJ KL. 1900 blir det träff i sal E7. Kvällen kommer att vara influerad av Apple Macintosh. Leksaksdatorn som har vuxit till sig.

Under första kvartalet 1984 kom första versionen med 128Kb, enkelsidiga 400Kb disketter och nästan inga program. Följande år växte den lite till och fick 512Kb, mera program och sällskap av LaserWriter. Förra året kom Mac Plus med 1Mb, dubbelsidiga 800Kb disketter och även en LaserWriter Plus. I början av mars i år kom så den öppna Mac II med stora utbyggnadsmöjligheter.

Med under kvällen kommer att vara folk från svenska Apple. De har 3 st Mac II. En av dessa är trasig. På Apple vet de inte om de kan laga felet. Men OM de klarar av det så kommer den att visas upp under kvällen.

Alla är välkomna.

/hn

Unix

MARK CRISPIN, Tops-20-programmerare vid Stanford, som även har en DEC-2020 hemma i ett ombyggt sovrum, Rainbow Suite, har skaffat sig hund av något fint slag. För mig är en hund en hund, så jag har glömt vad det var för modell. Nå, hunden skulle ju heta något och i ett hem där man får uppmaningar typ "Ring mig, om jag inte är hemma kan min fru hjälpa till med enklare saker som bandmonteringar etc..." måste ju husdjuret ha ett med datorer förknippat namn.

"Vi provade med allt möjligt trevligt som, Tenex, Jsys, UUO, Tops-20, Tops-10, Midas... men djuret reagerade inte. En dag råkade någon av misstag kalla den för Unix och den for runt, runt och verkade lystra till det. Så den stackars arma kraken går numera under namnet Unix."

Peter Löthberg

Nordiskt högskole-datorföreningsmöte i Linköping 1987.

HALMERS DATORFÖRENING OCH DATORFÖRENINGEN STACKEN PÅ KTH har beslutat att ordna ett möte mellan datorföreningarna vid högskolorna i de nordiska länderna. Mötet kommer att avhållas i Linköping under pingsthelgen, lördag 6:e–måndag 8:e juni 1987.

Planerade aktiviteter.

Lördag 6:e juni (pingstafton och svenska flaggans dag):

Förmiddag:	Ankomst till Linköping.
Eftermiddag:	Studiebesök på företag/institutioner.
Kväll:	Gemensam kvällsmat.
	Föreningspresentationer.

Söndag 7:e juni (pingstdagen):

Förmiddag:	Redaktionsarbete på gemensam tidning.
	Diskussioner om förhållande till respektive högskolor, resurser, tiggeriverksamhet m.m.
	Bildande av samarbetsorganisation.
Mitt på dagen:	Gemensam lunch.
Eftermiddag:	Lekar och spel.
	Presskonferens.
Kväll:	Bankett.

Måndag 8:e juni (annandag pingst):

Förmiddag:	Föredrag (föreningsprojekt m.m.).
Eftermiddag:	Avfärd från Linköping.

Frågor och anmälan.

Chalmers Datorförening besvarar frågor och tar emot anmälningar.

Telefon:	031-811727
Adress:	Chalmers Datorförening
	412 96 Göteborg
SUNET DECnet:	TEKN01::F01841467
UUCP:	cd-cd@gustaf.CS.CHALMERS.SE
	...enea!chalmers!cd3200!xt
	matte@chalmers.CS.CHALMERS.SE

Mötesplanering.

Konferenssystemet KOM på datorn Kicki i Stockholm används för planering av mötet.

Nordic University Computer Club Conference in Linköping 1987.

DURING the Whitsuntide, Saturday 6–Monday 8 June, the Chalmers Computer Society (at the Chalmers University of Technology in Gothenburg, Sweden) and the STACKEN Computer Club (at the Royal Institute of Technology in Stockholm, Sweden) will arrange a joint meeting in Linköping, with participants from the computer clubs at the Nordic universities.

Preliminary programme.

Saturday 6 June (Whitsun Eve):

Morning:	Participants arrive in Linköping.
Afternoon:	Visits to companies/institutions.
Evening:	Joint dinner.
	Presentations of the clubs.

Sunday 7 June (Whit Sunday):

Morning:	Editorial work on joint newsletter.
	Discussions on the relations to the schools, resources available to the clubs, etc.
Noon:	Joint lunch.
Afternoon:	Games and competitions.
	Press conference.
Evening:	Banquet.

Monday 8 June (Whit Monday):

Morning:	Seminars (project presentations, etc.).
Afternoon:	Participants leave.

Questions and registration.

Questions and registration will be handled by the Chalmers Computer Club.

Telephone:	+46 31 811727
Address:	Chalmers Datorförening
	S-412 96 Göteborg
	Sweden
UUCP:	cd-cd@gustaf.CS.CHALMERS.SE
	...{seismo,mcvax}!enea!chalmers!cd3200!xt
	matte@chalmers.CS.CHALMERS.SE

Conference planning.

The KOM teleconferencing system on the Kicki computer in Stockholm will be used for planning the conference. The discussions will probably be held in Scandinavian.

$1 00

January 1987 G N U ' S B U L L E T I N Volume 1 No 2

Contents

Gnu's Who	2
What is the Free Software Foundation?	3
GNU Project Status	4
GNU Software Available Now	6
How To Get GNU Software	7
Emacs version 18 improvements	9
GNU Wish List	10
Free Software Foundation Order Form	11
Thank Gnus	12

January 1987 G N U ' S B U L L E T I N Volume 1 No 2

Gnu's Who

In the first Bulletin there was a piece Gnu's Zoo telling of the various people working on Project Gnu and connecting them with an appropriate animal. Matching menageries of people to menageries of animals gets increasingly hard to do. So I have settled for presenting just the biography without the bestiary.

Paul Rubin started working for the Foundation full time this summer and is now helping us again in January. During the school year he studies mathematics at UC Berkeley. He's written a number of GNU utilities including the C Compatible Compiler Preprocessor (CCCP), worked on getting the printed Emacs manuals made, and is now developing kernel maintenance tools for TRIX. He likes jazz and classical music and hates cats.

hack (Jay Fenlason) joined project GNU full time this fall. Jay is finishing the awk program started by Paul Rubin. Jay says of himself:

"I've been a UNIX hacker since high school, I wrote the original version of Hack, and various obscure utilities. I'm most famous for my work on various Logo interpreters, including LSRHS/Childrens Museum logo, and TLC logo for the Commodore Amiga. When I'm not hacking, I read, write poetry, and play role-playing games."

Diane Wells has been helping all summer and fall and winter, answering the mail and filling orders.

Stephen Gildea redesigned the Emacs reference card for version 18. The new reference card source uses TeX instead of a proprietary formatting program.

Pierre MacKay typeset the masters that the Emacs manual pages were shot from on his high quality phototypesetter.

G N U ' S B U L L E T I N

Editor: Jerome E. Puzo
Asst. Editor Paul Rubin

January 1987 G N U ' S B U L L E T I N Volume 1 No 2

What is the Free Software Foundation?

by Richard M. Stallman

The Free Software Foundation is dedicated to eliminating restrictions on copying, redistribution, understanding and modification of software.

The word "free" in our name does not refer to price; it refers to freedom. First, the freedom to copy a program and redistribute it to your neighbors, so that they can use it as well as you. Second, the freedom to change a program, so that you can control it instead of it controlling you; for this, the source code must be made available to you.

The Foundation works to give you these freedoms by developing free compatible replacements for proprietary software. Specifically, we are putting together a complete, integrated software system "GNU" that is upward-compatible with Unix. When it is released, everyone will be permitted to copy it and distribute it to others; in addition, it will be distributed with source code, so you will be able to learn about operating systems by reading it, to port it to your own machine, to improve it, and to exchange the changes with others.

There are already organizations that distribute free CPM and MSDOS software. The Free Software Foundation is doing something different.

1. The other organizations exist primarily for distribution; they distribute whatever happens to be available. We hope to provide a complete integrated free system that will eliminate the need for any proprietary software.

2 One consequence is that we are now interested only in software that fits well into the context of the GNU system Distributing free MSDOS or Macintosh software is a useful activity, but it is not part of our game plan.

3. Another consequence is that we will actively attempt to improve and extend the software we distribute, as fast as our manpower permits For this reason, we will always be seeking donations of money, computer equipment or time, labor, and source code to improve the GNU system.

4 In fact, our primary purpose is this software development effort, distribution is just an adjunct which also brings in some money We think that the users will do most of the distribution on their own, without needing or wanting our help.

Why a Unix-Like System?

It is necessary to be compatible with some widely used system to give our system an immediate base of trained users who could switch to it easily and an immediate base of application software that can run on it (Eventually we will provide free replacements for proprietary application software as well, but that is some years in the future)

[cont'd on next page]

January 1987 G N U ' S B U L L E T I N Volume 1 No 2

We chose Unix because it is a fairly clean design which is already known to be portable, yet whose popularity is still rising. The disadvantages of Unix seem to be things we can fix without removing what is good in Unix.

Why not imitate MSDOS or CPM? They are more widely used, true, but they are also very weak systems, designed for tiny machines. Unix is much more powerful and interesting. When a system takes years to implement, it is important to write it for the machines that will become available in the future; not to let it be limited by the capabilities of the machines that are in widest use at the moment but will be obsolete when the new system is finished.

Why not aim for a new, more advanced system, such as a Lisp Machine? Mainly because that is still more of a research effort; there is a sizeable chance that the wrong choices will be made and the system will turn out not very good. In addition, such systems are often tied to special hardware. Being tied to one manufacturer's machine would make it hard to remain independent of that manufacturer and get broad community support.

Status of the GNU project, last updated 3 January 1987.
by RMS

(See also the article "GNU Software Available Now", on page 6 of this issue).

* GNU Emacs and GDB.

GNU Emacs and GDB are already released. Berkeley is distributing GNU Emacs with the 4.3 distribution, and DEC is going to distribute it with Unix systems on Vaxes.

* gsh, the GNU imitation C shell.

Beta-test release of a C shell with input editing and compilation of shell scripts is expected at the end of January.

The same program is supposed to imitate sh, but that doesn't work yet.

* Kernel.

I am planning to use a remote procedure call kernel called TRIX, developed at MIT, as the GNU kernel. It runs, and supports basic Unix compatibility, but needs a lot of new features Its authors have decided to distribute it free. It was developed on an obscure, expensive 68000 box designed years ago at MIT.

In December 1986, we started working on the changes needed to TRIX.

[cont'd]

January 1987 G N U ' S B U L L E T I N Volume 1 No 2

* C compiler

I am now working on finishing a new portable optimizing C compiler. It supports the Oct 1986 draft of ANSI C and has compiled both itself and GNU Emacs. However, I plan to make some rearrangements in order to enable compilation of arbitrarily large functions in bounded amounts of memory, though with some decrease in optimization compared to what can be done with lots of memory.

The compiler performs automatic register allocation, common subexpression elimination, invariant code motion from loops, constant propagation and copy propagation, delaying popping of function call arguments, plus many local optimizations that are automatically deduced from the machine description. By the time it is finished it will probably also know when to keep constant addresses in registers.

It makes shorter and faster 68020 code than the sun compiler with -O.

A new cpp was written last summer. It is as fast as the Unix cpp. PHR is now making it support the Oct 1986 standard.

* Assembler

An assembler has been written. It works well on Vaxes but proves to be harder to port than I had hoped, so some rewriting is needed to simplify the interface between the machine-dependent portions and the machine-independent ones.

* Window system.

I plan to use the X window system written at MIT. This system is already available free.

* Documentation system

I now have a truly compatible pair of programs which can convert a file of texinfo format documentation into either a printed manual or an Info file.

Documentation files are needed for many utilities.

* Stdio

A free stdio system has just been received

* Other utilities.

The GNU 'ls', 'grep', 'make' and 'ld' are in regular use 'tar' recently appeared on USENET net sources The other object-file management utilities are written too 'cron' and 'at' were recently submitted, and so was 'm4' The assembler works for the Vax, but proves to be hard to port, so it may need considerable rewriting

'awk' is now in final testing stages 'diff' is making progress We have a program like 'lex' but not fully compatible, work is required on it

[continued on page 8]

January 1987 G N U ' S B U L L E T I N Volume 1 No.2

GNU Software Available Now

* GNU Emacs

In 1975, Richard Stallman developed the first Emacs: the extensible, customizable real-time display editor. GNU Emacs is his second implementation of Emacs. It's the first Emacs available on Unix systems which offers true Lisp, smoothly integrated into the editor, for writing extensions. It also provides a special interface to MIT's free X window system, which makes redisplay very fast.

GNU Emacs has been in widespread use since 1985 and often, as at MIT's Project Athena, displaces proprietary implementations of Emacs because of its greater reliability as well as its good features and easier extensibility.

GNU Emacs has run on many kinds of Unix systems: those made by Alliant (system release 1 or 2), AT&T (3b machines and 7300 pc), Celerity, Digital (Vax, not PDP-11), Dual, Encore, Gould, HP (9000 series 200 or 300 but not series 500), IBM (RT/PC running 4.2), Integrated Solutions (Optimum V with 68020 and VMEbus), Masscomp, Megatest, NCR (Tower 32), Plexus, Pyramid, Sequent, Stride (system release 2), Sun (any kind), Tahoe, Tektronix (NS16000 system), Texas Instruments (Nu), Whitechapel (MG1), and Wicat. These include both Berkeley Unix and System V (release 0, 2 or 2.2). It also runs on Apollo machines and on VAX/VMS.

GNU Emacs use is described by the GNU Emacs Manual, available from the Free Software Foundation.

* GDB

GDB is the source-level C debugger written for the GNU project in 1986. It offers many features not usually found in debuggers on Unix, such as a history that records all values examined within the debugger for concise later reference, multi-line user-defined commands, and a strong self-documentation capability. It currently runs on Vaxes and Suns (systems version 2 and 3).

A users' manual for GDB is available from the Foundation.

* GNU CC

The GNU C compiler is a fairly portable optimizing compiler. It generates good 68000 and 68020 code and generated good Vax code when it was last tested for the Vax. It features automatic register packing that makes register declarations unnecessary. It supports full ANSI C as of the latest draft standard. We expect to release the compiler in 1st quarter 1987.

* Bison

Bison is an upward-compatible replacement for YACC, with some additional undocumented features. It has been in use for a couple of years.

[Cont'd]

January 1987 G N U ' S B U L L E T I N Volume 1 No 2

* X Window System

X is a portable, network transparent window system for bitmap displays written at MIT and DEC. It currently runs on DEC VAXstation, Lexidata 90, and most Sun Microsystems displays, with others in the works. X supports overlapping windows, fully recursive subwindows, and provides hooks for several different styles of user interface. Applications provided include a terminal emulator, bitmap editor, several window managers, clock, window dump and undump programs, hardcopy printing program for the LN03 printer, several typesetting previewers, and more.

* MIT Scheme

Scheme is a simplified, lexically scoped dialect of Lisp, designed at MIT and other universities for two purposes: teaching students of programming, and researching new parallel programming constructs and compilation techniques. MIT Scheme is written in C and runs on many kinds of Unix systems.

Sorry, there is no documentation for the current distribution version of MIT Scheme. A new standard for Scheme has been designed by the various labs that work on Scheme, and work is going on at MIT to change MIT Scheme to fit. Once that is done, the standard will serve as a manual for MIT Scheme. At that time, we will distribute both the new release of Scheme and the standard.

* GNU Chess

GNU Chess was written in 1986 by Stuart Cracraft, who is continuing to develop it. It comes with an interface to the X window system to display a pretty chessboard. It also has an opening book which is being added to all the time.

* Hack

Hack is a display oriented adventure game similar to Rogue.

H O W T O G E T G N U S O F T W A R E

All software and publications are distributed with a permission to copy and redistribute. The easiest way to get a copy of GNU Software is from someone else who has it. You need not ask for permission, just copy it.

If you have access to the Internet, you can get the latest distribution version of GNU Software from host: 'prep.ai.mit.edu' For more info read '/u2/emacs/GETTING.GNU.SOFTWARE' on said host.

If you cannot get a copy in any of these ways, you can order one from the Free Software Foundation Please consult the accompanying Order Form for prices and details

January 1987 G N U ' S B U L L E T I N Volume 1 No 2

GNU PROJECT STATUS, continued from page 5

* Free Software Foundation.

The foundation exists for two purposes: to accept gifts to support GNU development, and to carry out distribution. We are now tax exempt; you can deduct donations to us on your tax returns.

Our address is

Free Software Foundation
1000 Mass Ave
Cambridge, MA 02138

and our phone number is (617) 876-3296.

* Service directory.

The foundation now maintains a Service Directory; a list of people who offer service to individual users of GNU Emacs and, eventually, all parts of the GNU system. Service can be answering questions for new users, customizing programs, porting to new systems, or anything else.

* Possible target machines.

GNU will require a cpu that uses 32-bit addresses and integers and addresses to the 8-bit byte. 1 meg of core should be enough, though 2 meg would probably make a noticeable improvement in performance. Running much of the system in 1/2 meg may be possible, but certainly not GNU Emacs. I do not expect that virtual memory will be required, but it is VERY desirable in any case.

GNU Emacs requires more than a meg of addressable memory in the system, although a meg of physical memory is probably enough if there is virtual memory.

A hard disk will be essential; at least 20 meg will be needed to hold the system plus the source code plus the manual plus swapping space. Plus more space for the user's files, of course. I'd recommend 80meg for a personal GNU system.

This is not to say that it will be impossible to adapt some or all of GNU for other kinds of machines; but it may be difficult, and I don't consider it part of my job to try to reduce that difficulty.

I have nothing to say about any specific models of microcomputer, as I do not follow hardware products.

* Porting.

It is too early to inquire about porting GNU (except GNU Emacs). First, we have to finish it.

January 1987 G N U ' S B U L L E T I N Volume 1 No 2

Emacs 18 runs on Vax VMS.

* GNU Emacs now runs on Vax VMS.

* Searching is several times faster.

* Running out of memory is never fatal.
Memory usage for strings is cut in half by a new garbage collector.

* GNU Emacs can emulate other editors: EDT, VI, Gosmacs

* New major modes for LaTeX, Fortran, Scribe, Modula2 and Prolog

* Terminal-independent function keys.

The first, terminal-dependent level converts a terminal's function key codes into standard codes. The second level maps these into commands Users can customize the second level and enjoy the same results automatically on all terminal types.

* All C-c LETTER keys are reserved for users. Such commands previously defined by Mail mode, Picture mode and Telnet mode have been moved.

* New Commands

** Buffer-sorting commands.

Various new commands sort the lines, paragraphs or pages in the region; they can also sort lines according to fields or columns

** 'occur' output now serves as a menu.

'M-x occur' now allows you to move quickly to any of the occurrences listed. To do this, select the '*Occur*' buffer that contains the output of 'occur', move point to the occurrence you want, and type C-c C-c.

** Meta-TAB performs completion on the Emacs Lisp symbol name in the buffer

** Dynamic abbreviation package.

The new command Meta-/ expands an abbreviation in the buffer before point by searching the buffer for words that start with the abbreviation

** 'c-tab-always-indent' parameter tells TAB in C mode to insert a tab character when used in the middle of a line

** Outline mode is customizable

You can now specify with a regexp which lines are outline headings Lines that separate pages are always considered headings

* File saving changes

Undo says "not modified" only when the buffer matches the disk file

[cont'd on next page]

January 1987 G N U ' S B U L L E T I N Volume 1 No.2

** Auto save file name now has '#' at end.
For a file 'foo', the auto save file is now called '#foo#'. This is so that '*.c' in a shell command will never match auto save files.

** M-x recover-file checks file dates.
M-x recover-file is used to recover a file's contents from its auto save file. Now this command checks the date of the auto save file and offers to recover from it only if it is newer.

** Modifying a buffer whose file is changed on disk is detected instantly. Thus, you are warned that something is wrong before you go ahead and create a skewed version of the file.

** Exiting Emacs offers to save '*mail*'.

** M-x ftp-find-file and M-x ftp-write-file read and write files via Internet.

** Precious files. If you mark a buffer "precious", Emacs will save it by renaming so that there is no time between the disappearance of the old file and the appearance of the new one. This is used for RMAIL files.

* Existing Emacs usable as a server for 'mail', etc.
Programs that invoke a user-specified editor as a temporary inferior can now be told to use an existing Emacs process instead.

* M-x disassemble disassembles byte-compiled Emacs Lisp functions.

* 'substitute-key-definition' finds all keys defined as one command and redefines them all as another command.

* New hooks for file I/O.
You can set up multiple hooks for finding and saving files. These can arrange automatically to get files via RCS, uncompression, ftp, etc.

* New data structure controls mode line format.
Now it is possible to change one aspect of what appears in the mode line independently of what is being done with the rest of the mode line.

GNU Wish List

The GNU project can always use donations of money or equipment. Specifically, we could use:

* Salary for two more full time programers.

* A computer powerful enough to develop the GNU kernel on. This means a 68xxx/32xxx class processor with several meg of main memory and an 80 meg disk.

* Local volunteers to help mail tapes and manuals to our clients, and answer mail. We need about 10 person-hours/week of help doing this.

* Dedicated people, with C and Unix knowledge, especially those with a local (Cambridge and surrounds) address, to write programs and documentation. Ask for our task list if you want to help.

Free Software Foundation Order Form
January 1987

All software and publications are distributed with permission to copy and redistribute.

Quantity Price

_____ $150 GNU Emacs source code, on 1600bpi industry standard magnetic tape in tar format. The tape also contains Scheme, Hack, Bison, GNU Chess, GDB, and the X window system.

_____ $175 Same data as above, on a DC300XL 1/4" cartridge tape.

_____ $150 GNU Emacs source code, on 1600bpi industry standard magnetic tape in VMS interchange format.

_____ $15 GNU Emacs manual. This includes a reference card. The source for this manual also comes with the tape. (~300 pages)

Thus, one 1600bpi tape and one manual come to $165.

_____ $60 Box of six GNU Emacs manuals, shipped book rate.

_____ $1 GNU Emacs reference card.

_____ $5 Ten GNU Emacs reference cards.

_____ $10 GDB manual. The source for this manual also comes with the source for GDB. (~50 pages)

_____ $10 TeXinfo manual. The source for this manual also comes with the Emacs source. (~30 pages)

$_____ 5% Massachusetts sales tax, if applicable.

$_____ Optional tax deductable donation

$_____ Total amount enclosed

Shipping outside of North America is normally by surface mail, which is very slow. For air mail delivery, please add $15 per tape or manual, $1 for an individual reference card, or 50 cents per card in quantities of 10 or more.

Orders are filled upon receipt of check or money order. We do not have the staff to handle the billing of unpaid orders. Please help keep our lives simple by including your payment with your order. Make checks payable to the Free Software Foundation, and mail orders to:

Free Software Foundation phone: (617) 876-3296
1000 Massachusetts Avenue
Cambridge, MA 02138

Prices are subject to change without notice. All software from the Free Software Foundation is provided on an ''as is'' basis, with no warranty of any kind

January 1987 G N U ' S B U L L E T I N Volume 1 No 2

Thank Gnus

The Free Software Foundation would like to send special thank gnus to the following:

Thanks to Stacy Goldstein. Stacy answered the mail and filled orders for FSF. Her efforts got us thru a very busy season. She then left to continue her studies in Hawaii which she claims "is as good as they say".

Thanks to Todd Cooper and Henry Mensch. They also helped out in the mail room.

Thanks to the MIT Laboratory for Computer Science. The LCS has provided FSF with the loan of a TI Nu machine and a Microvax for program development.

Thanks to Professor Dertouzos, head of LCS. His specific decision to support us is greatly appreciated.

Thanks to the MIT Artificial Intelligence Laboratory for invaluable assistance of many kinds.

Thanks to Lisp Machine, Inc. LMI has generously provided office space, computer resources and a mailing address for FSF.

Thanks to the European Unix Users' Group of Sweden and the Swedish Royal Institute of Technology for their generous donations.

Thanks to those who sent money and offered help. Thanks also to those who support us by ordering Emacs manuals and distribution tapes.

The creation of this bulletin is our way of thanking all who have expressed interest in what we are doing.

end

Free Software Foundation, Inc.
1000 Mass Ave
Cambridge, MA 02138

stamp
here

GNU task list

NU task list, last updated 11 February, 1987.

0. Documentation

We very urgently need documentation for some parts of the system that already exist or will exist very soon:

- A C reference manual.
- A GNU Emacs Lisp programming manual. (There is a team working on this, but they need more people to write chapters).
- A Yacc manual.
- A SH manual.
- A stdio manual.

Reference cards for various programs (and texinfo has been suggested).

1. Imitations of standard parts of Unix:

dbm, diction, diff3, explain, graph, join, lint, more, plot, mt, sdiff, style, tip

We have something like lex, but it is not fully compatible with Unix lex. It needs to be fixed.

kwik indexer.

nroff/troff, eqn, tbl, and standard macro packages. Probably the equivalent of troff should output dvi files (TEX output files). Part of this has already been written.

Finish an incomplete vi clone.

Finish an incomplete implementation of diff.

An imitation of uucp. Specs for the uucp communication protocol were recently published; we can send you a copy. A free "uuslave" program has been written by John Gilmore.

Mostly machine independent floating point print routines. This isn't as hard as it sounds; talk to me about it. This is important.

A variant of the regex library that uses finite state machine algorithms (à la egrep) rather than backtracking. This is important.

A mail-delivery system (replacing sendmail). It does not need to have all the hair that sendmail has. Please speak to me about the details if you are interested.

2. GNU-specific projects:

info, a program for perusing node-structured documentation files, equivalent to M-x info in GNU Emacs but not dependent on GNU Emacs.

texinfo, a program for converting texinfo format files into node-structured info files, equivalent to M-x texinfo-format-buffer in GNU Emacs but not dependent on GNU Emacs.

3. Other random projects:

- An imitation of dbase2 or dbase3 (How dbased!)
- An imitation of Page Maker.
- paint and draw programs for the X window system.
- A music playing and editing system.
- A postscript interpreter that can output pictures to screens (using X) and to printers.
- A Forth system.
- A Smalltalk system.
- A Prolog interpreter and compiler.
- A Logo system. (We have one that you can start with, but certain parts of it are poorly written and must be replaced.)

Note that graphics programs should be written to work with the X window system, a free portable window system from MIT and DEC that we will be using.

4. Compilers for other batch languages.

These cannot be worked on until the C compiler is released, but that is expected in a few months. At that time it will be possible for people to write parsers/front ends for other languages such as Fortran, Pascal, Algol 60, Algol 68, Modula, PL/I, Ada, or whatever.

Likewise, it will be possible to adapt the C front end as a lint; preferably not as stupidly stubborn as the Unix lint.

5. Games.

- Empire
- Adventure
- Backgammon
- imitations of your favorite video games

We do not need rogue, as we have hack.

RMS Lecture at the RIT 30 October 1986

[Richard M. Stallman gave this lecture on 30 October 1986 at the Royal Institute of Technology in Stockholm, Sweden, at a meeting arranged by the STACKEN Computer Club and attended by more than 200 persons.

Hans Nordström recorded the speech on tape, and Bjørn Remseth later wrote it down.

Note: This is a sligthly edited transcript of the talk. As such it contains false starts, as well as locutions that are natural in spoken English but look strange in print. It is not clear how to correct them to written English style without 'doing violence to the original speech'.]

It seems that there are three things that people would like me to talk about. On the one hand I thought that the best thing to talk about here for a club of hackers, was what it was like at the MIT in the old days. What made the Artificial Intelligence Lab such a special place. But people tell me also that since these are totally different people from the ones who were at the conference Monday and Tuesday that I ought to talk about what's going on in the GNU project and that I should talk about why software and information can not be owned, which means three talks in all, and since two of those subjects each took an hour it means we're in for a rather long time. So I had the idea that perhaps I could split it in to three parts, and people could go outside for the parts they are not interested in, and that then when I come to the end of a part I can say it's the end and people can go out and I can send Jan Rynning out to bring in the other people. (Someone else says: "Janne, han trenger ingen mike" (translation: "Janne, he doesn't need a mike")). Jan, are you prepared to go running out to fetch the other people? Jmr: I am looking for a microphone, and someone tells me it is inside this locked box. Rms: Now in the old days at the AI lab we would have taken a sledgehammer and cracked it open, and the broken door would be a lesson to whoever had dared to lock up something that people needed to use. Luckily however I used to study Bulgarian singing, so I have no trouble managing without a microphone.

Anyway, should I set up this system to notify you about the parts of the talk, or do You just like to sit through all of

it? (Answer: Yeaaah)

When I started programming, it was 1969, and I did it in an IBM laboratory in New York. After that I went to a school with a computer science department that was probably like most of them. There were some professors that were in charge of what was supposed to be done, and there were people who decided who could use what. There was a shortage of terminals for most people, but a lot of the professors had terminals of their own in their offices, which was wasteful, but typical of their attitude. When I visited the Artificial Intelligence lab at MIT I found a spirit that was refreshingly different from that. For example: there, the terminals was thought of as belonging to everyone, and professors locked them up in their offices on pain of finding their doors broken down. I was actually shown a cart with a big block of iron on it, that had been used to break down the door of one professors office, when he had the gall to lock up a terminal. There were very few terminals in those days, there was probably something like five display terminals for the system, so if one of them was locked up, it was a considerable disaster.

In the years that followed I was inspired by that ideas, and many times I would climb over ceilings or underneath floors to unlock roooms that had machines in them that people needed to use, and I would usually leave behind a note explaining to the people that they shouldn't be so selfish as to lock the door. The people who locked the door were basically considering only themselves. They had a reason of course, there was something they thought might get stolen and they wanted to lock it up, but they didn't care about the other people were affecting by locking up other things in the same room. Almost every time this happened, once I brought it to their attention, that it was not up to them alone whether that room should be locked, they were able to find a compromise solution: some other place to put the things they were worried about, a desk they could lock, another little room. But the point is that people usually don't bother to think about that. They have the idea: "This room is Mine, I can lock it, to hell with everyone else", and that is exactly the spirit that we must teach them not to have.

But this spirit of unlocking doors wasn't an isolated thing, it was part of an entire way of life. The hackers at the AI lab were really enthusiastic about writing good programs, and interesting programs. And it was because they were so eager to get more work done, that they wouldn't put up with having the terminals locked up, or lots of other things that people could do to obstruct useful work. The differences between people with high morale who really care about what they're trying to do, and people who think of it as just a job. If it's just a job, who cares if the people who hired you are so stupid they

make you sit and wait, it's their time, their money but not much gets done in a place like that, and it's no fun to be in a place like that.

Another thing that we didn't have at the AI lab was file protection. There was no security at all on the computer. And we very conciously wanted it that way. The hackers who wrote the Incompatible Timesharing System decided that file protection was usually used by a self-styled system manager to get power over everyone else. They didn't want anyone to be able to get power over them that way, so they didn't implement that kind of a feature. The result was, that whenever something in the system was broken, you could always fix it. You never had to sit there in frustration because there was NO WAY, because you knew exactly what's wrong, and somebody had decided they didn't trust you to do it. You don't have to give up and go home, waiting for someone to come in in the morning and fix the system when you know ten times as well as he does what needs to be done.

And we didn't let any professors or bosses decide what work was going to be done either, because our job was to improve the system! We talked to the users of course; if you don't do that you can't tell what's needed. But after doing that, we were the ones best able to see what kind of improvements were feasible, and we were always talking to each other about how we'd like to see the system changed, and what sort of neat ideas we'd seen in other systems and might be able to use. So the result is that we had a smoothly functioning anarchy, and after my experience there, I'm convinced that that is the best way for people to live.

Unfortunatly the AI lab in that form was destroyed. For many years we were afraid the AI lab would be destroyed by another lab at MIT, the Lab for Computer Science, whose director was a sort of empire builder type, doing everything he could to get himself promoted within MIT, and make his organization bigger, and he kept trying to cause the AI lab to be made a part of his lab, and nobody wanted to do things his way because he believed that people should obey orders and things like that.

But that danger we managed to defend against, only to be destroyed by something we had never anticipated, and that was commercialism. Around the early 80's the hackers suddenly found that there was now commercial interest in what they were doing. It was possible to get rich by working at a private company. All that was necessary was to stop sharing their work with the rest of the world and destroy the MIT-AI lab, and this is what they did despite all the efforts I could make to prevent them.

Essentially all the competent programmers except for me, at the AI lab were hired away, and this caused more

than a momentary change, it caused a permanent transformation because it broke the continuity of the culture of hackers. New hackers were always attracted by the old hackers; there were the most fun computers and the people doing the most interesting things, and also a spirit which was a great deal of fun to be part of. Once these things were gone, there is nothing to recommend the place to anyone new, so new people stopped arriving. There was no-one they could be inspired by, no-one that they could learn those traditions from. In addition no-one to learn how to do good programming from . With just a bunch of professors and graduate students, who really don't know how to make a program work, you can't learn to make good programs work. So the MIT AI lab that I loved is gone, and after a couple of years of fighting against the people who did it to try to punish them for it I decided that I should dedicate my self to try to create a new community with that spirit.

But one of the problems I had to face was the problem of proprietary software. For example one thing that happened at the lab, after the hackers left, was that the machines and the software that we had developed could no longer be maintained. The software of course worked, and it continued to work if nobody changed it, but the machines did not. The machines would break and there would be no-one who could fix them and eventually they would be thrown out. In the old days, yes we had service contracts for the machines, but it was essentially a joke. That was a way of getting parts after the expert hackers from the AI lab fixed the problem. Because if you let the field-service person fix it it would take them days, and you didn't want to do that, you wanted it to work. So, the people who knew how to do those things would just go and fix it quickly, and since they were ten times as competent as any field service person, they could do a much better job. And then they would have the ruined boards, they would just leave them there and tell the field service person "take these back and bring us some new ones".

In the real old days our hackers used to modify the machines that came from Digital also. For example, they built paging-boxes to put on the PDP-10's . Nowadays I think there are some people here [in Stocholm] who do such things too, but it was a pretty unusual thing in those days. And the really old days, the beginning of the 1960's people used to modify computers adding all sorts of new instructions and new fancy timesharing features, so that the PDP-1 at MIT by the time it was retired in the mid-seventies had something like twice as many instructions as it had when it was delivered by Digital in the early sixties, and it had special hardware scheduler assisting features and strange memory-mapping features making it possible to assign individual hardware devices to particular timesharing jobs and lots of things that I

hardly really know about. I think they also built in some kind of extended addressing modes they added index registers and indirect addressing, and they turned it essentially from a weak machine into a semi-reasonable one.

I guess it is one of the disadvantages of VLSI that it's no longer so feasible to add instructions to your machines.

The PDP-1 also had a very interesting feature, which is that it was possible to interesting programs in very few instructions. Fewer that any other machine since then. I believe for example that the famous display hack "munching squares" which made squares that get bigger and break up into lots of smaller squares which gets bigger and break up into smaller squares . That was written in something like five instructions on the PDP-1. And many other beautiful display programs could be written in few instructions.

So, that was the AI lab. But what was the culture of hackers like aside from their anarchism? In the days of the PDP-1 only one person could use the machine, at the beginning at least. Several years later they wrote a timesharing system, and they added lots of hardware for it. But in the beginning you just had to sign up for some time. Now of course the professors and the students working on official projects would always come in during the day. So, the people who wanted to get lots of time would sign up for time at night when there were less competition, and this created the custom of hackers working at night. Even when there was timesharing it would still be easier to get time, you could get more cycles at night, because there were fewer users. So people who wanted to get lots of work done, would still come in at night. But by then it began to be something else because you weren't alone, there were a few other hackers there too, and so it became a social phenomenon. During the daytime if you came in, you could expect to find professors and students who didn't really love the machine, whereas if during the night you came in you would find hackers. Therefore hackers came in at night to be with their culture. And they developed other traditions such as getting Chinese food at three in the morning. And I remember many sunrises seen from a car coming back from Chinatown. It was actually a very beautiful thing to see a sunrise, cause' that's such a calm time of day. It's a wonderful time of day to get ready to go to bed. It's so nice to walk home with the light just brightening and the birds starting to chirp, you can get a real feeling of gentle satisfaction, of tranquility about the work that you have done that night.

Another tradition that we began was that of having places to sleep at the lab. Ever since I first was there, there was always at least one bed at the lab. And I may have done a little bit more living at the lab than most people because every year of two for some reason or other I'd

have no apartment and I would spend a few months living at the lab. And I've always found it very comfortable, as well as nice and cool in the summer. But it was not at all uncommon to find people falling asleep at the lab, again because of their enthusiasm; you stay up as long as you possibly can hacking, because you just don't want to stop. And then when you're completely exhausted, you climb over to the nearest soft horizontal surface. A very informal atmosphere.

But when the hackers all left the lab this caused a demographic change, because the professors and the students who didn't really love the machine were just as numerous as before, so they were now the dominant party, and they were very scared. Without hackers to maintain the system, they said, "we're going to have a distaster, we must have commercial software", and they said "we can expect the company to maintain it". It proved that they were utterly wrong, but that's what they did.

That was exactly when a new KL-10 system was supposed to arrive, and the question was, would it run the Incompatible Timesharing System or would it run digital's Twenex system. Once the hackers were gone who probably would have supported using ITS, the academic types chose to run the commercial software, and this had several immediate effects. Some of them weren't actually so immediate but they followed inevitably as anyone who thought about it would see.

One thing was that that software was much more poorly written, and harder to understand; therefore making it harder for people to make the changes that were in fact needed. Another was, that that software came with security, which had the inevitable effect of causing people to cooperate with each other less. In the old days on ITS it was considered desirable that everyone could look at any file, change any file, because we had reasons to. I remember one interesting scandal where somebody sent a request for help in using Macsyma. Macsyma is a symbolic algebra program that was developed at MIT. He sent to one of the people working on it a request for some help, and he got an answer a few hours later from somebody else. He was horrified, he sent a message "so-and-so must be reading your mail, can it be that mail files aren't properly protected on your system?" "Of course, no file is protected on our system. What's the problem? You got your answer sooner; why are you unhappy? Of course we read each other's mail so we can find people like you and help them." Some people just don't know when they're well off.

But of course Twenex not only has security, and by default turns on security, but it's also designed with the assumption that security is in use. So there are lots of things that are very easy to do that can cause a lot of damage, and the only thing that would stop you

from doing them by accident, is security. On ITS we evolved other means of discouraging people from doing those things by accident, but on Twenex you didn't have them because they assumed that there was going to be be strict security in effect and only the bosses were going to have the power to do them. So they didn't put in any other mechanism to make it hard to do by accident. The result of this is that you can't just take Twenex and turn of the security and have what you'd really like to have, and there were no longer the hackers to make the changes to put in those other mechanisms, so people were forced to use the security. And about six months after the machine was there they started having some coups d'etat. That is, at first we had the assumption that everyone who worked for the lab was going to have the wheel bit which gave full powers to override all security measures, but some days you'd come in some afternoon and find out that the wheel bits of just about everybody had been turned off.

When I found out about those, I overthrew them. The first time, I happened to know the password of one of the people who was included among the elite, so I was able to use that to turn everyone back on. The second time he had changed his password, he had now changed his sympathies, he was now part of the aristocratic party. So, I had to bring the machine down and use non-timeshared DDT to poke around. I poked around in the monitor for a while, and eventually figured out how to get it to load itself in and let me patch it, so that I could turn off password checking and then I turned back on a whole bunch of people's wheel bits and posted a system message. I have to explain that the name of this machine was OZ, so I posted a system message saying: "There was another attempt to seize power. So far the aristocratic forces have been defeated—Radio Free OZ" Later I discovered that "Radio Free OZ" is one of the things used by Firesign Theater. I didn't know that at the time.

But gradually things got worse and worse, it's just the nature of the way the system had been constructed forced people to demand more and more security. Until eventually I was forced to stop using the machine, because I refused to have a password that was secret. Ever since passwords first appeared at the MIT-AI lab I had come to the conclusion that to stand up for my belie, to follow my belief that there should be no passwords, I should always make sure to have a password that is as obvious as possible and I should tell everyone what it is. Because I don't believe that it's really desirable to have security on a computer, I shouldn't be willing to help uphold the security regime. On the systems that permit it I use the "empty password" , and on systems where that isn't allowed, or where that means you can't log in at all from other places, things like that, I use my login name as my password. It's about

as obvious as you can get. And when people point out that this way people might be able to log in as me, i say "yes that's the idea, somebody might have a need to get some data from this machine. I want to make sure that they aren't screwed by security"

And an other thing that I always do is I always turn of all protection on my directory and files, because from time to time I have useful programs stored there and if there's a bug I want people to be able to fix it.

But that machine wasn't designed also to support the phenomenon called "tourism". Now "tourism" is a very old tradition at the AI lab, that went along with our other forms of anarchy, and that was that we'd let outsiders come and use the machine. Now in the days where anybody could walk up to the machine and log in as anything he pleased this was automatic: if you came and visited, you could log in and you could work. Later on we formalized this a little bit, as an accepted tradition specially when the Arpanet began and people started connecting to our machines from all over the country. Now what we'd hope for was that these people would actually learn to program and they would start changing the operating system . If you say this to the system manager anywhere else he'd be horrified. If you'd suggest that any outsider might use the machine, he'll say "But what if he starts changing our system programs?" But for us, when an outsider started to change the system programs, that ment he was showing a real interest in becoming a contributing member of the community. We would always encourage them to do this. Starting, of course, by writing new system utilities, small ones, and we would look over what they had done and correct it, but then they would move on to adding features to existing, large utilities. And these are programs that have existed for ten years or perhaps fifteen years, growing piece by piece as one craftsman after an other added new features.

Sort of like cities in france you might say, where you can see the extremely old buildings with additions made a few hundred years later all the way up to the present. Where in the field of computing, a program that was started in 1965 is essentially that. So we would always hope for tourists to become system maintainers, and perhaps then they would get hired, after they had already begun working on system programs and shown us that they were capable of doing good work.

But the ITS machines had certain other features that helped prevent this from getting out of hand, one of these was the "spy" feature, where anybody could watch what anyone else was doing. And of course tourists loved to spy, they think it's such a neat thing, it's a little bit naughty you see, but the result is that if any tourist starts doing anything that causes trouble there's always

somebody else watching him. So pretty soon his friends would get very mad because they would know that the continued existance of tourism depended on tourists being responsible. So usually there would be somebody who would know who the guy was, and we'd be able to let him leave us alone. And if we couldn't, then what we would was we would turn of access from certain places completely, for a while, and when we turned it back on, he would have gone away and forgotten about us. And so it went on for years and years and years.

But the Twenex system wasn't designed for this sort of thing, and eventually they wouldn't tolerate me with my password that everybody knew, tourists always logging in as me two or three at a time, so they started flushing my account. And by that time I was mostly working on other machines anyway, so eventually I gave up and stopped ever turning it on again. And that was that. I haven't logged in on that machine as myself...[At this point RMS is interrupted by tremendous applause]...for.

But when they first got this Twenex system they had several changes they wanted to make. Changes in the way security worked. They also wanted to have the machine on both the Arpa network and the MIT-chaos network, and it turns out that they were unable to do this, that they couldn't get anyone who was sufficiently competent to make such changes. There was no longer talent available to do it, and it was to hard to change. That system was much harder to understand, because it was to poorly written, and of course, Digital wouldn't do these things, so their ideas that a commercial system would essentially maintain it self, proved to be mistaken. They had just as much need for system hackers, but they had no longer the means to entice system hackers. And nowadays at MIT there are more people interested in hacking on ITS, than there are interested in hacking on Twenex.

And the final reason why this is so, is that Twenex can't be shared. Twenex is a proprietary program and you're only allowed to have the sources if you keep them secret in certain nasty ways, and this gives them a bad flavor. Unless a person is oblivious (which some people in computers are, there's some people who'll do anything if it's fun for them, and won't think for a minute whether they're cooperating with anyone else, but you'd have to be pretty oblivious to not to notice what a sad thing it is to work on a program like that, and that is a further discuragement). And if that isn't enough there is the fact that every year or so they're going to give you a new release full of 50,000 additional lines of code all written by monkeys. Because they generally follow the "million monkeys typing, and eventually they'll come up with something useful" school of system development.

It was clear to me from what I saw

happening to these proprietary systems that the only way we could have the spirit of the old AI lab was to have a free operating system. To have a system made up of free software which could be shared with anyone. So that we could invite everyone to join in improving it. And that's what led up to the GNU project. So I guess we've arrived at the second part of the talk.

The GNU Project

About three and a half year ago it was clear to me that I should start developing a free software system. I could see two possible kinds of systems to develop: One: A LISP-machine-like system, essentially a system just like the MIT LISP machine system that had just been developed, except free, and running on general purpose hardware, not on special LISP machines. And the other possibility was a more conventional operating system, and it was clear to me that if I made a conventional operating system, I should make it compatible with UNIX, because that would make it easy for people all around to switch to it. After a little while, I concluded I should do the latter and the reason was, that I saw that you can't have something really like the LISP machine system on general purpose hardware. The LISP machine system uses special hardware plus special writable microcode to gain both good execution speed and robust detection of errors at runtime, specially data-type errors. In order to make a LISP system run fast enough on ordinary hardware, you have to start making assumptions. Assuming that a certain argument is the right type, and then if it isn't the system just crashes.

Of course you can put in explicit checks, you can write a robust program if you want, but the fact is that you are going to get things like memory addressing errors when you feed a function an argument of the wrong type if you did NOT put in things to check for it.

So the result is then that you need something running underneath the LISP system to you catch these errors, and give the user the ability to keep on running, and debug what happened to him. Finally I concluded that if I was going to have to have a operating system at a lower level, I might as well make a good operating-system—that it was a choice between an operating system and the lisp, or just an operating system; therefore I should do the operating system first, and I should make it compatible with UNIX. Finally when I realized that I could use the most amusing word in the English language as a name for this system, it was clear which choice I had to make. And that word is of course GNU, which stands for "Gnu's Not Unix". The recursive acronym is a very old tradition among the hacker community around MIT. It started, I believe with an editor called TINT, which means: "Tint Is

Not Teco", and later on it went through names such as SINE for "SINE Is Not Emacs", and FINE for "Fine Is Not Emacs", and EINE for "Eine Is Not Emacs", and ZWEI for "Zwei Was Eine Initially", and ultimately now arrives at GNU.

I would say that since the time about two and a half years ago when I actually started working on GNU, I've done more than half of the work. When I was getting ready to start working on the project, I first started looking around for what I could find already available free. I found out about an interesting portable compiler system which was called "the free university compiler kit", and I thought, with a name like that, perhaps I could have it. So, I sent a message to the person who had developed it asking if he would give it to the GNU project, and he said "No, the university might be free, but the software they develop isn't", but he then said that he wanted to have a UNIX compatible system too, and he wanted to write a sort of kernel for it, so why didn't I then write the utilities, and they could both be distributed with his proprietary compiler, to encourage people to buy that compiler. And I thought that this was despicable and so I told him that my first project would be a compiler.

I didn't really know much about optimizing compilers at the time, because I 'd never worked on one. But I got my hands on a compiler, that I was told at the time was free. It was a compiler called PASTEL, which the authors say means "off-color PASCAL".

Pastel was a very complicated language including features such as parametrized types and explicit type parameters and many complicated things. The compiler was of course written in this language, and had many complicated features to optimize the use of these things. For example: the type "string" in that language was a parameterized type; you could say "string(n)" if you wanted a string of a particular length; you could also just say "string", and the parameter would be determined from the context. Now, strings are very important, and it is necessary for a lot of constructs that use them to run fast, and this means that they had to have a lot of features to detect such things as: when the declared length of a string is an argument that is known to be constant throughout the function, to save to save the value and optimize the code they're going to produce, many complicated things. But I did get to see in this compiler how to do automatic register allocation, and some ideas about how to handle different sorts of machines.

Well, since this compiler already compiled PASTEL, what i needed to do was add a front-end for C, which I did, and add a back-end for the 68000 which I expected to be my first target machine. But I ran into a serious problem. Because the PASTEL language was defined not to require you to declare something before you used

it, the declarations and uses could be in any order, in other words: Pascal's "forward" declaration was obsolete, because of this it was necessary to read in an entire program, and keep it in core, and then process it all at once. The result was that the intermediate storage used in the compiler, the size of the memory needed, was proportional to the size of your file. And this also included stack-space, you needed gigantic amounts of stack space, and what I found as a result was: that the 68000 system available to me could not run the compiler. Because it was a horrible version of unix that gave you a limit of something like 16K words of stack, this despite the existance of six megabytes in the machine, you could only have 16Kw of stack or something like that. And of course to generate its conflict matrix to see which temporary values conflicted, or was alive at the same time as which others, it needed a quadratic matrix of bits, and that for large functions that would get it to hundreds of thousands of bytes. So i managed to debug the first pass of the ten or so passes of the compiler, cross compiled on to that machine, and then found that the second one could never run.

While I was thinking about what to do about these problems and wondering whether I should try to fix them or write entirely new compiler, in a roundabout fashion I began working on GNU Emacs. GNU Emacs is the main distributed portion of the GNU system. It's an extensible text editor a lot like the original emacs which I developed ten years ago, except that this one uses actual LISP as its extension language. The editor itself is implemented in C, as is the LISP interpreter, so the LISP interpreter is completely portable, and you don't need a LISP system external to the editor. The editor contains its own LISP system, and all of the editing commands are written in LISP so that they can provide you with examples to look at for how to write your own editing commands, and things to start with, so you can change them into the editing commands that you really want.

In the summer of that year, about two years ago now, a friend of mine told me that because of his work in early development of Gosling Emacs, he had permission from Gosling in a message he had been sent to distribute his version of that. Gosling originally had set up his Emacs and distributed it free and gotten many people to help develop it, under the expectation based on Gosling's own words in his own manual that he was going to follow the same spirit that I started with the original Emacs. Then he stabbed everyone in the back by putting copyrights on it, making people promise not to redistribute it and then selling it to a software-house. My later dealings with him personally showed that he was every bit as cowardly and despicable as you would expect from that history.

But in any case, my friend gave me this program, and my intention was to

change the editing commands at the top level to make them compatible with the original Emacs that I was used to. And to make them handle all the combinations of numerical arguments and so on that one might expect that they would handle and have all the features that I wanted. But after a little bit of this, I discovered that the extension language of that editor, which is called MOCK-LISP was not sufficient for the task. I found that that I had to replace it immediatly in order to do what I was planning to do. Before I had had the idea of someday perhaps replacing MOCK-LISP with real LISP, but what I found out was that it had do be done first. Now, the reason that MOCKLISP is called MOCK, is that it has no kind of structure datatype: it does not have LISP lists; it does not have any kind of array. It also does not have LISP symbols, which are objects with names: for any particular name, there is only one object, so that you can type in the name and you always get the same object back. And this tremendously hampers the writing of many kinds of programs, you have to do things by complicated string-manipulation that don't really go that way.

So I wrote a LISP interpreter and put it in in place of MOCKLISP and in the process I found that I had to rewrite many of the editor's internal data structures because I wanted them to be LISP objects. I wanted the interface between the LISP and the editor to be clean, which means that objects such as editor buffers, sub-processes, windows and buffer-positions, all have to be LISP objects, so that the editor primitives that work on them are actually callable as LISP functions with LISP data. This meant that I had to redesign the data formats of all those objects and rewrite all the functions that worked on them, and the result was that after about six months I had rewritten just about everything in the editor.

In addition, because it is so hard to write things in MOCKLISP, all the things that had been written in MOCK-LISP were very unclean and by rewriting them to take advantage of the power of real LISP, I could make them much more powerful and much simpler and much faster. So I did that, and the result was that when I started distributing this program only a small fraction remained from what I had received.

At this point, the company that Gosling thinks he sold the program to challenged my friend's right to distribute it, and the message was on backup tapes, so he couldn't find it. And Gosling denied having given him permission. And then a strange thing happened. He was negotiating with this company, and it seemed that the company mainly was concerned with not having anything distributed that resembled what they were distributing. See, he was still distributing, and the company where he worked, which is Megatest, was still distributing the same thing he had given me, which really was

an old version of Gosling Emacs with his changes, and so he was going to make an agreement with them where he would stop distributing that, and would switch to using GNU Emacs, and they would then acknowledge that he really had the permission after all, and then supposedly everyone would be happy. And this company was talking to me about wanting to distribute GNU Emacs, free of course, but also sell various sorts of supporting assistance, and they wanted to hire me to help do the work. So it's sort of strange that they then changed their mind and refused to sign that agreement, and put up a message on the network saying that I wasn't allowed to distribute the program. They didn't actually say that they would do anything, they just said that it wasn't clear whether they might ever someday do something. And this was enough to scare people so that no one would use it any more, which is a sad thing.

(Sometimes I think that perhaps one of the best things I could do with my life is: find a gigantic pile of proprietary software that was a trade secret, and start handing out copies on a street corner so it wouldn't be a trade secret any more, and perhaps that would be a much more efficient way for me to give people new free software than actually writing it myself; but everyone is too cowardly to even take it.)

So I was forced to rewrite all the rest that remained, and I did that, it took me about a week and a half. So they won a tremendous victory. And I certainly wouldn't ever cooperate with them in any fashion after that.

Then after GNU Emacs was reasonably stable, which took all in all about a year and a half, I started getting back to other parts of the system. I developed a debugger which I called GDB which is a symbolic debugger for C code, which recently entered distribution. Now this debugger is to a large extent in the spirit of DBX, which is a debugger that comes with Berkeley Unix. Commands consist of a word that says what you want to do, followed by arguments. In this debugger, commands can all be abbreviated, and the common commands have single character abbreviations, but any unique abbreviation is always allowed. There are extensible HELP facilities, you can type HELP followed by any command or even subcommands, and get a lengthy description of how to use that command. Of course you can type any expression in C, and it will print the value.

You can also do some things that are not usual in symbolic C debuggers, for example: You can refer to any C datatype at any memory address, either to examine the value, or to assign the value. So for example if you want to store a floating point value in a word at a certain address, you just say: "Give me the object of type FLOAT or DOUBLE at this address" and then assign that. Another thing you can do

is to examine all the values that have been examined in the past. Every value examined gets put on the "value history". You can refer to any element in the history by its numerical position, or you can easily refer to the last element with just dollar-sign. And this makes it much easier to trace list structure. If you have any kind of C structure that contains a pointer to another one, you can do something like "PRINT *$.next", which says: "Get the next field out of the last thing you showed me, and then display the structure that points at". And you can repeat that command, and each time you'll see then next structure in the list. Whereas in every other C debugger that I've seen the only way to do that is to type a longer command each time. And when this is combined with the feature that just typing carriage-return repeats the last command you issued, it becomes very convenient. Just type carriage-return for each element in the list you want to see.

There are also explicitly settable variables in the debugger, any number of them. You say dollar-sign followed by a name, and that is a variable. You can assign these variables values of any C datatype and then you can examine them later. Among the things that these are useful for are: If there's a particular value that you're going to examine, and you know you are going to refer to it a lot , then rather than remember its number in the history you might give it a name. You might also find use for them when you set conditional breakpoints. Conditional breakpoints are a feature in many symbolic debuggers, you say "stop when you get to this point in the program, but only if a certain expression is true". The variables in the debugger allow you to compare a variable in the program with a previous value of that variable that you saved in a debugger variable. Another thing that they can be used for is for counting, because after all, assignments are expressions in C, therefore you can do "$foo=+5" to increment the value of "$foo" by five, or just "$foo++" you can do. You can even do this in a conditional breakpoint, so that's a cheap way of having it break the tenth time the breakpoint is hit, you can do "$foo—==0". Does everyone follow that? Decrement foo and if it's zero now, break. And then you set $foo to the number of times you want it to skip, and you let it go. You can also use that to examine elements of an array. Suppose you have an array of pointers, you can then do:

PRINT X[$foo++]

But first you do

SET $foo=0

Okay, when you do that [points at the "PRINT" expression] , you get the zeroeth element of X, and then you do it again and it gets the first element, and suppose these are pointers to structures, then you probably put an asterisk there [before the X in the PRINT

expression] and each time it prints the next structure pointed to by the element of the array. And of course you can repeat this command by typing carriage-return. If a single thing to repeat is not enough, you can create a user-defined-command. You can say "DEFINE MUMBLE", and then you give some lines of commands and then you say "END". And now there is defined a "MUMBLE" command which will execute those lines. And it's very useful to put these definitions in a command file. You can have a command file in each directory, that will be loaded automatically when you start the debugger with that as your working directory. So for each program you can define a set of user defined commands to access the datastructures of that program in a useful way. You can even provide documentation for your user-defined commands, so that they get handled by the "help" features just like the built-in commands.

One other unusual thing in this debugger, is the ability to discard frames from the stack. Because I believe it's important not just to be able to examine what's happening in the program you're debugging, but also to change it in any way conceivable. So that after you've found one problem and you know what's wrong, you can fix things up as if that code were correct and find the next bug without having to recompile your program first. This means not only being able to change the data areas in you program flexibly, but also being able to change the flow of control. In this debugger you can change the flow of control very directly by saying:

SET $PC=<some number>

So you can set the program counter. You can also set the stack pointer, or you can say

SET $SP+=<something>

If you want to increment the stack-pointer a certain amount. But in addition you can also tell it to start at a particular line in the program, you can set the program counter to a particular source line. But what if you find that you called a function by mistake and you didn't really want to call that function at all? Say, that function is so screwed up that what you really want to do is get back out of it and do by hand what that function should have done. For that you can use the "RETURN" command. You select a stack frame and you say "RETURN", and it causes that stack-frame, and all the ones within it, to be discarded as if that function were returning right now, and you can also specify the value it should return. This does not continue execution; it pretends that return happened and then stops the program again, so you can continue changing other things.

And with all these things put together you thus have pretty good control over what's going on in a program.

In addition one slightly amusing thing: C has string constants, what happens if you use a string constant in an expression that you're computing in the debugger? It has to create a string in the program you were debugging. Well it does. It sets up a call to MALLOC in that debugged program, lets MALLOC run, and then gets control back. Thus it invisibly finds a place to put the string constant.

Eventually when this debugger is running on the real GNU system, I intend to put in facilities in the debugger to examine all of the internal status of the process that is running underneath it. For example to examine the status of the memory map, which pages exist, which are readable, which are writable, and to examine the inferior program's terminal status. There already is a bit of a command; this debugger, unlike the debuggers on UNIX, keeps the terminal status completely separate for the debugger and the program you're debugging, so that it works with programs that run in raw mode, it works with programs that do interrupt driven input, and there's also a command that enables you to find out something about the terminal settings at the program you're debugging is actually using. I believe that in general a debugger should allow you to find out everything that's going on in the inferior process.

There are two other main parts of the GNU system that already exist. One is the new C compiler, and one is the TRIX kernel.

The new C compiler is something that I've written this year since last spring. I finally decided that I'd have to throw out PASTEL. This C compiler uses some ideas taken from PASTEL, and some ideas taken from the University of Arizona Portable Optimizer. Their interesting idea was to handle many different kinds of machines by generating simple instructions, and then combining several simple instructions into a complicated instruction when the target machine permits it. In order to do this uniformly, they represent the instructions in algebraic notation. For example, an ADD instruction might be represented like this:

```
r[3]=r[2]+4
```

This would be a representation inside their compiler for instruction to take the contents of register two, add four and store it in register three. In this fashion you can represent any possible instruction for any machine. So they actually did represent all the instructions this way and then when it came time to try to combine them, they would do this by substituting one expression into another, making a more complicated algebraic expression for the combined instruction.

Sometimes depending on whether the result of the first instruction had

any further use, it might be necessary to make a combined instruction with two assignment operators. One for this value [pointing at ???] and another one with this value [pointing at ???] substituted in it with what came from the second instruction. But if this value was only used that once, you could eliminate it after substituting for it; there'd be no need to compute it any more. So it's actually somewhat complicated doing the substitution correctly checking that the intervening instructions don't change any of these values and other such things. When you support such things as auto-increment and auto-decrement addressing, which I do now, you also have to do various checks for those to check for situations where what you're doing is not value preserving.

But after checking all those things, then you take the substituted combined expression and put it through a pattern matcher, which recognizes all the valid instructions of your chosen target machine. And if it's recognized, then you replace those two instructions with the combined one, otherwise you leave them alone. And their technique is to combine two or three instructions related by dataflow in this way.

In the Arizona compiler, they actually represent things as text strings like this, and their compiler is horribly slow. First I had some idea of just using their compiler and making changes in it, but it was clear to me I had to rewrite it entirely to get the speed I wanted, so I have rewritten it to use list structure representations for all these expressions. Things like this:

```
(set (reg 2)
     (+  (reg 2)
         (int 4)))
```

This looks like Lisp, but the semantics of these are not quite LISP, because each symbol here is one recognized specially. There's a particular fixed set of these symbols that is defined, all the ones you need. And each one has a particular pattern of types of arguments, for example: "reg" always has an integer, because registers are numbered, but "+" takes two subexpressions, and so on. And with each of these expressions is also a data type which says essentially whether it's fixed or floating and how many bytes long it is. It could be extended to handle other things too if you needed to.

And the way I do automatic register allocation is that when I initially generate this code, and when I do the combination and all those things, for every variable that conceivably go into a register, I allocate what I call a pseudo register number, which is a number starting at sixteen or whatever is to high to be a real register for your target machine. So the real registers are numbered zero to fifteen or whatever and above that comes pseudo registers. And then one of the last parts of the compiler consists of going through and changing

all the pseudo registers to real registers. Again it makes a conflict graph, it sees which pseudo registers are alive at the same point and they of course can't go in the same real register, and then it tries packing pseudo registers into real registers as much as it can, ordering them by priority of how important they are.

And finally it then has to correct the code for various problems, such as happen when there were pseudo registers that don't fit in the real registers, that had to be put into stack slots instead. When that happens on certain machines, some of the instructions may become invalid. For example on the 68000 you can add a register into memory and you can add memory into register, but you can't add one memory location into another. So if you have an ADD instruction, and you're headed for a 68000 and both of the things end up in memory, it's not valid. So this final pass goes through and copies things into registers and out of registers as needed to correct those problems.

Problems can also arise with index registers. If you're trying to index by something, then most of the time that code will become invalid if the index quantity is in memory, except in a few cases on some machines where you can it with indirect addressing. In the cases when you're doing auto-increment on an index register you may have to copy the value into a register, do the instruction, and then copy the incremented value back to the memory slot where it really lives.

There's got room for a lot of hair, and I've not finished implementing all the hair needed to make really fully efficient.

This compiler currently works by having a parser which turns C code into effectively a syntax tree annotated with C datatype information. Then another pass which looks at that tree and generates code like this [LISP like code]. Then several optimization passes. One to handle things like jumps across jumps, jumps to jumps, jumps to .+1, all of which can be immediatly simplified. Then a common subexpression recognizer, then finding basic blocks, and performing dataflow-analysis, so that it can tell for each instruction which values are used in that instruction and never used afterward. And also linking each instruction to the places where the values it uses were generated, so if I have one instruction which generates pseudo register R[28], and then another instruction later which uses R[28] and it's the first place to use R[28], I make the second one point back to the first one, and this pointer is used to control the attempts to combine the instructions. You don't combine adjacent instructions, you combine an instruction that uses a value with the instruction that produced that value. Even if there are other instructions in between, they don't matter for this, you just have to check them to make sure they don't do anything to in-

terfere. Then after the combiner comes the dynamic register allocator, and finally something to convert it into assembly code.

In the Arizona compiler the instruction recognizer was generated with LEX. Your machine description was simply a LEX program that LEX would turn into a C function to recognize valid instructions as strings. What I have is instead a special purpose decision tree that's generated from a machine description written in this syntax as if it were LISP. And this recognizer is used as a subroutine for many different parts of the compiler.

Currently this compiler runs about as fast as PCC. It runs noticeably faster if you tell it not to do the hairy register allocation, in which case it allocates registers the same way as PCC does. In its super hairy mode it does a much better job of allocating registers than PCC, and I observe that for the VAX it generates the best code I've seen from any C compiler on the VAX.

For the 68000 the code is still not ideal. I can see places where early stages do things that are not the best, because it can't fully look ahead. It has a choice in an early stage, and it does the thing that it thinks is going to be best, but really if it did the other one, a later stage is actually smart enough to do something even better. But the early stage doesn't know what the later stage is going to do, so I have more work to do on some of these things.

Sometimes this causes it to free up registers unnecessarily. Because when things wind up in memory and it needs to copy them into registers, it needs to get registers to copy them into. This means taking registers that it has already allocated to, and kicking those temporary quantities out to stack slots. Of course this may invalidate more instructions now that those things are in memory, not registers, so it has to check again and again. Sometimes it thinks it has to copy things to registers and really it isn't going to have to, so it may free up too many things and thus not use all the registers that it could.

(Question: Do you have a code generator for 32000?) Not yet, but again, it's not a code generator it's just a machine description that you need. A list of all the machine instructions described in this [LISP like] form. So in fact aside from the work of implementing the idea of constraints on which arguments can be in registers and which kind of registers, which is something which was needed for the 68000 and was not needed for the VAX, the work of porting this compiler from the VAX to the 68000 just took a few days. So it's very easy to port.

The compiler currently generates assembler code and it can generate debugging information either in the format that DBX wants, or in the special internal format of GDB. I'd say the

only work needed on this compiler is in three areas. One: I have to add a "profiling" feature, like the one that the UNIX compilers have. Two: I have to make these register allocation things smarter, so that I can stop seeing stupid things appearing in the output. And three: There are various bugs, things that doesn't handle correctly yet, although it has compiled itself correctly. I expect this will just take a few months, and then I will release the compiler.

The other sizable part of the system that exist, is the kernel. (Question: A pause?) Ah, yeah I guess we've forgotten about breaks. Why don't I finish talking about the kernel, which should only take about five minutes, and then we can take a break.

Now, for the kernel I am planning to use a system called TRIX (it doesn't stand for anything that I know of) which was developed as a research project at MIT. This system is based on Remote Procedure Call. Thus programs are called domains. Each domain is a address space and various capabilities, and a capability is none other than the ability to call a domain. Any domain can create "capability ports" to call it, and then it can pass these ports to other domains, and there is no difference between calling the system and calling another user domain. In fact you can't tell which you have. Thus it is very easy to have devices implemented by other user programs. A file system could be implemented by a user program, transparently. It's also transparent to communicate across networks. You think that you're directly calling another domain, but really you're calling the network server domain. It takes the information that you gave in the call, and passes this over the network to another server program which then calls the domain that you're trying to talk to. But you and that other domain see this as happening invisibly.

The TRIX kernel runs, and it has a certain limited amount of UNIX compatibility, but it needs a lot more. Currently it has a file system that uses the same structure on disk as the ancient UNIX file system does. This made it easier to debug the thing, because they could set up the files with UNIX, and then they could run TRIX, but that file system doesn't have any of the features that I believe are necessary.

Features that I believe must be added include: Version numbers, undeletion, information on when and how and where the file was backed up on tape, atomic superseding of files. I believe that it is good that in Unix when a file is being written, you can already look at what's going there, so for example, you can use "tail" to see how far the thing got, that's very nice. And if the program dies, having partly written the file, you can see what it produced. These things are all good, but, that partly written output should not ever be taken for the complete output that you expected to have eventually. The previous ver-

sion of that should continue to be visible and used by everyone who tries to use it, until the new version is completely and correctly made. This means that the new version should be visible in the file system but not under the name it is supposed to have. It should get renamed when it's finished. Which is by the way what happens in ITS, although there each user program has to do this explicitly. For UNIX compatibility with the user programs, it has to happen invisibly.

I have a weird hairy scheme to try to make version numbers fit with the existing UNIX user programs. And this is the idea that you specify a file name leaving the version number implicit, if you just specify the name in the ordinary way. But if you wish to specify a name exactly, either because you want to state explicitly what version to use, or because you don't want versions at all, you put a point at the end of it. Thus if you give the filename "FOO" it means "Search the versions that exist for FOO and take the latest one". But if you say "FOO." it means "use exactly the name FOO and none other". If you say "FOO.3." it says "use exactly the name FOO.3" which of course is version three of FOO and none other. On output, if you just say "FOO", it will eventually create a new version of "FOO", but if you say "FOO." it will write a file named exactly "FOO".

Now there's some challenges involved in working out all the details in this, and seeing whether there are any lingering problems, whether some UNIX software actually breaks despite feeding them names with points in them and so on, to try to make it get the same behavior.

I would expect that when you open a file for output whose name ends in a point, you should actually open that name right away, so you get the so you get the same UNIX behavior, the partially written output is immediatly visible, whereas when you output a name that doesn't end in a point, the new version should appear when you close it, and only if you close it explicitly. If it gets closed because the job dies, or because the system crashes or anything like that, it should be under a different name.

And this idea can be connected up to "star matching", by saying that a name that doesn't end in a point is matched against all the names without their version numbers, so if a certain directory has files like this:

foo.1
foo.2
foo.3

If I say "*", that's equivalent to

foo
bar

because it takes all the names and gets rid of their versions, and takes all the

distinct ones. But if I say "*." then it takes all the exact names, puts a point after each one, and matches against them. So this gives me all the names for all the individual versions that exist. And similar, you can see the difference between "*.c" and "*.c." this [the first]would give you essentially versionless references to all the ".c" files, whereas this [the second] will give you all the versions .....well this actually wouldn't, you'd have to say "*.c.*.". I haven't worked out the details here.

Another thing, that isn't a user visible feature and is certainly compatible to put in, is failsafeness in the file system. Namely, by writing all the information on disk in the proper order, arranging that you can press "halt" at any time without ever corrupting thereby the file system on disk. It is so well known how to do this, I can't imagine why anyone would neglect it. Another idea is further redundant information. I'm not sure whether I'll do this or not, but I have ideas for how to store in each file all of its names, and thus make it possible if any directory on disk is lost, to reconstruct it from the rest of the contents of the disk.

Also I think I know how to make it possible to atomically update any portion of a file. Thus if you want to replace a certain subrange of a file with new data in such a fashion that any attempt to read the file will either see only the old data, or only the new data. I believe I can do that, without any locking even.

For network support, I intend eventually to implement TCP/IP for this system. I also think it's possible to use KERMIT to get something effectively equivalent to UUCP.

A shell I believe has already been written. It has two modes, one imitating the BOURNE shell, and one imitating the C-shell in the same program. I have not received a copy of it yet, and I don't know how much work I'll have to do on it. Also many other utilities exists. A MAKE exists, LS, there's a YACC replacement called BISON which is being distributed. Something pretty close to a LEX exits, but it's not totally compatible, it needs some work. And, in general what remains to be done is much less that what's been done, but we still need lots of people to help out.

People always ask me "When is it going to be finished?" Of course I can't know when it's going to be finished, but that's the wrong question to ask me. If you were planning to pay for it, it would make sense for you to want to know exactly what are you going to get and when. But since you're not going to pay for it, the right question for you to ask is "how can you help make it get finished sooner?" I have a list of projects, it is on a file at MIT, and people who are interested in helping could send me mail at this internet address, and I will send back a list of projects.

(I wonder if this is will work (looking at the chalk)). Is this readable? This is "RMS@PREP.AI.MIT.EDU" (just follow the bouncing ball.) And now let's take a break, and after the break, I will say some really controversial things. So don't leave now. If you leave now, you're going to miss the real experience.

[Here we had a 15 min. break]

Why Software Can't Be Owned

I've been asked to announce how you can get copies of GNU software. Well, one way of course is if you know a friend who has a copy, you can copy it, but if you don't know a friend who has a copy, and you're not on the Internet, you can't FTP it, then you can always order a distribution tape, and send some money to the Free Software Foundation. Of course free programs is not the same thing as free distribution. I'll explain this in detail later.

Here I have an EMACS manual, of the nicely printed variety. It has been phototypeset and then offset printed. Although you can also print it yourself from the sources that come in the EMACS distribution, you can get these copies from the Free Software Foundation. You can come afterwards and look at this and also this contains an order for you might copy some information from, and this [front] picture has also sometimes been enjoyed. This [pointing at a figure being chased by RMS riding a gnu] is a scared software hoarder, I'll be talking about him in a moment.

Software is a relatively new phenomenon. People started distributing software perhaps thirty years ago. It was only about twenty years ago that someone had the idea of making a business about it. It was an area with no tradition about how people did things, or what rights anybody had. And there were several ideas for what other areas of life you might bring traditions from by analogy.

One analogy that is liked by a lot of professors in Europe, is that between programs and mathematics. A program is sort of a large formula. Now, traditionally nobody can own a mathematical formula. Anybody can copy them and use them.

The analogy that's most meaningful to ordinary people is with recipes. If you think about it, the thing that you have in ordinary life that's most like program is a recipe, it's instructions for doing something. The differences come because a recipe is followed by a person, not by a machine automatically. It's true there's no difference between source code and object code for a receipe, but it's still the closest thing. And no-one is allowed to own a receipe.

But the analogy that was chosen was the analogy with books, which have

copyright. And why was this choice made? Because the people that had the most to gain from making that particular choice were allowed to make the decision. The people who wrote the programs, not the people who used the programs, were allowed to decide, and they decided in a completely selfish fashion, and as a result they've turned the field of programming into an ugly one.

When I entered the field, when I started working at MIT in 1971, the idea that programs we developed might not be shared was not even discussed. And the same was Stanford and CMU, and everyone, and even DIGITAL. The operating system from DIGITAL at that time was free. And every so often I got pieces of program from DIGITAL system such as a PDP-11 cross assembler, and I ported it to run on ITS, and added lots of features. It was no copyright on that program.

It was only in the late seventies that this began to change. I was extremely impressed by the sharing spirit that we had. We were doing something that we hoped was useful and were happy if people could use it. So when I developed the first EMACS, and people wanted to start use it outside of MIT, I said that it belongs to the EMACS "Commune", that in order to use EMACS you had to be a member of the commune, and that meant that you had the responsibility to contribute all the improvements that you made. All the improvements to the original EMACS had to be sent back to me so that I could incorporate them into newer versions of EMACS, so that everyone in the community could benefit from them.

But this started to be destroyed when SCRIBE was developed at CMU, and then was sold to a company. This was very disturbing to a lot of us at many universities, because we saw that this was a temptation placed in front of everyone, that it was so profitable to be uncooperative and those of us who still believed in cooperation had no weapon to try to compel people to cooperate with us. Clearly, one after another, people would defect and stop cooperating with the rest of society, until only those of us with very strong consciences would still cooperate. And that's what happened.

The field of programming has now become an ugly one, where everyone cynically thinks about how much money he is going to get by not being nice to the other people in the field, and to the users.

I want to establish that the practice of owning software is both materially wasteful, spiritually harmful to society and evil. All these three things being interrelated. It's spiritually harmful because it involves every member of society who comes in contact with computers in a practice that is obviously materially wasteful to other people. And

every time you do something for your own good, which you know is hurting other people more that it helps you, you have to become cynical in order to support such a thing in your mind. And it's evil because it is deliberatly wasting the work done in society and causing social decay.

First I want to explain the kinds of harm that are done by attempts to own software and other information that's generally useful, then I'll go on to rebutt the arguments made to support that practice, and then I want to talk about how to fight that phenomenon, and how I'm fighting it.

The idea of owning information is harmful in three different levels. Materially harmful on three different levels, and each kind of material harm has a corresponding spiritual harm.

The first level is just that it discourages the use of the program, it causes fewer people to use the program, but in fact it takes no less work to make a program for fewer people to use. When you have a price on the use of a program this an incentive, that's the word these software hoarders love to use, the price is an incentive for people not to use the program, and this is a waste. If for example only half as many people use the program because it has a price on it, the program has been half wasted. The same amount of work has produced only half as much wealth.

Now in fact, you don't have to do anything special to cause a program to get around to all the people who want to use it, because they can copy it themselves perfectly well, and it will get to everyone. All you have to do after you've written the program is to sit back and let people do what they want to do. But that's not what happens; instead somebody deliberatly tries to obstruct the sharing of the program, and in fact, he doesn't just try to obstruct it, he tries to pressure other people into helping. Whenever a user signs a nondisclosure agreement he has essentially sold out his fellow users. Instead of following the golden rule and saying, "I like this program, my neighbour would like the program, I want us both to have it", instead he said, "Yeah, give it to me. To hell with my neighbour! I'll help you keep it away from my neighbour, just give it to me!", and that spirit is what does the spiritual harm. That attitude of saying, "To hell with my neighbours, give ME a copy".

After I ran into people saying they wouldn't let me have copies of something, because they had signed some secrecy agreement, then when somebody asked me to sign a thing like that I knew it was wrong. I couldn't do to somebody else the thing that had made me so angry when it was done to me.

But this is just one of the levels of harm. The second level of harm comes when people want to change the program, because no program is re-

ally right for all the people who would like to use it. Just as people like to vary recipes, putting in less salt say, or maybe they like to add some green peppers, so people also need to change programs in order to get the effects that they need.

Now, the software owners don't really care whether people can change the program or not, but it's useful for their ends to prevent people. Generally when software is proprietary you can't get the sources, you can't change it, and this causes a lot of wasted work by programmers , as well as a lot of frustration by users. For example: I had a friend who told me how she worked for many months at a bank where she was a programmer, writing a new program. Now, there was a commercially available program that was almost right, but it was just not quite the thing they needed, and in fact as it was it was useless for them. The amount of change it would have taken to make it do what they needed was probably small, but because the sources of that program were not available, that was impossible. She had to start over from scratch and waste a lot of work. And we can only speculate about what fraction of all programmers in the world are wasting their time in this fashion.

And then there is also the situation where a program is adequate make do, but it's uncomfortable. For example: The first time we had a graphics printer at MIT, we wrote the software ourselves, and we put in lots of nice features, for example it would send you a message when your job had finished printing, and it would send you a message if the printer ran out of paper and you had a job in the queue, and lots of other things that were what we wanted. We then got a much nicer graphic printer, one of the first laser printers, but then the software was supplied by Xerox, and we couldn't change it. They wouldn't put in these features, and we couldn't, so we had to make do with things that "half worked". And it was very frustrating to know that we were ready, willing and able to fix it, but weren't permitted. We were sabotaged.

And then there are all the people who use computers and say that the computers are a mystery to them, they don't know they work. Well how can they possibly know? They can't read the programs they're using. The only way people learn how programs should be written, or how programs do what they do, is by reading the source code.

So I could only wonder whether the idea of the user who just thinks of the computer as a tool is not actually a self-fulfilling profecy, a result of the practice of keeping source code secret.

Now the spiritual harm that goes with this kind of material harm, is in the spirit of self-sufficiency. When a person spends a lot of time using a computer system, the configuration of that computer system becomes the city that

he lives in. Just as the way our houses and furniture are laid out, determines what it's like for us to live among them, so that the computer system that we use, and if we can't change the computer system that we use to suit us, then our lives are really under the control of others. And a person who sees this becomes in a certain way demoralized: "It's no use trying to change those things, they're always going to be bad. No point even hassling it. I'll just put in my time and .....when it's over I'LL go away and try not to think about it any more ". That kind of spirit, that unenthusiasm is what results from not being permitted to make things better when you have feelings of public spirit.

The third level of harm is in the interaction between software developers themselves. Because any field of knowledge advance most when people can build on the work of others, but ownership of information is explicitly designed to prevent anyone else to doing that. If people could build on other people's work, then the ownership would become unclear, so they make sure each new entry to the field has to start from the beginning, and thus they greatly slow down the advance of the field.

So we can see: How many spreadsheet systems were made all by different companies , all without any benefit of understanding how it was done before? Yes it's true, the first spreadsheet written wasn't perfect. It probably only ran on certain kinds of computers, and it didn't do some things in the best possible way. So there were various reasons why certain people would want to rewrite parts of it. But if they had only to rewrite the parts that they really wanted to improve, that would have made for a lot less work. You may see how to make one aspect of a system better, you may not see how to make another aspect of the same system any better, in fact you might have a great deal of trouble doing it as well. Now if you could take the part that you like and redo only the part that you have an inspiration for, you could have a system that's better in all ways, with much less work than it now takes to write a completely new system. And we all know that system can often benefit from being completely rewritten, but that's only if you can read the old one first.

Thus, the people in the programming field have evolved a way of wasting a lot of their time and thus making apparently a need for more programmers than we really need. Why is there a programmer shortage? Because with intellectual property programmers have arranged to waste half the work they do, so we seem to need twice as many programmers. And so, when people point to the system of intellectual property and say "look at the large employment statistics, look at how big this industry is" what that really proves is that people are wasting a lot of money and time. If they talk about looking for

ways to improve programmer productivity, they're happy to do this if it involves superior tools, but to improve programmer productity by getting rid of the explicit things that is done to reduce programmer productivity, that they're against. Because that would reduce the number of programmers employed. There's something a little bit schizophrenic there.

And the spiritual harm that corresponds to this level of material harm is to the spirit of scientific cooperation, which used to be so strong that scientists even in countries that were at war would continue cooperating, because they knew that what they were doing had nothing to do with the war, it was just for the long term benefit of humanity. Nowadays, people don't care about the long term benefit of humanity any more.

To get an idea of what it's like to obstruct the use of a program, let's imagine that we had a sandwich, that you could eat, and it wouldn't be consumed. You could eat it, and another person could eat it, the same sandwich, any number of times, and it would always remain just as nourishing as originally.

The best thing to do, the thing that we ought to do with this sandwich is carry it around to the places where there are hungry people; bringing it to as many mouths as possible, so that it feeds as many people as possible. By all means, we should not have a price to eat from this sandwich, because then people would not afford to eat it, and it would be wasted.

The program is like this sandwich, but even more so because it can be in many different places at once being eaten, used by different people one after the other. It is as if this sandwich was enough to feed everyone, everywhere, forever, and that were not allowed to happen, because someone believed he should own it.

Now, the people who believe that they can own programs, generally put forward two lines of argument for this. The first one is "I wrote it, it is a child of my spirit, my heart, my soul is in this. How can anyone take it away from me? Wherever it goes it's mine, mine, MINE!!". Well, it's sort of strange that most of them signs agreements saying it belongs to the company they work for.

So I believe this is one of the things you can easily talk yourself into believing is important, but you can just as easily convince yourself it doesn't matter at all.

Usually, these people use this argument to demand the right to control even how people can change a program. They say: "Nobody should be able to mess up my work of art". Well, imagine that the person who invented a dish that you plan to cook had the right to control how you can cook it, because it's his work of art. You want to leave

out the salt, but he says "Oh, no. I designed this dish, and it has to have this much salt!" "But my doctor says it's not safe for me to eat salt. What can I do?".

Clearly, the person who is using the program is much closer to the event. The use of the program affects him very directly, whereas it only has a sort of abstract relation to the person who wrote the program. And therefore, for the sake of giving people as much control as possible over their own lives, it has to be the user who decides those things.

The second line of argument they make is the economic one. "How will people get payed to program?" they say, and there's a little bit of real issue in this. But a lot of what they say is confusion. And the confusion is, it's not at all the same to say "if we want to have a lot of people programming we must arrange for them not to need to make a living in any other fashion" on the one hand, and to say "We need to have the current system, you need to get rich by programming" on the other hand. There's a big difference between just making a living wage and making the kind of money programmers, at least in the US make nowadays. They always say: "How will I eat?" , but the problem is not really how "Will he eat?", but "How will he eat sushi?". "How will I have a roof over my head?", but the real problem is "How can he afford a condo?".

The current system were chosen by the people who invest in software development, because it gives them the possibility of making the most possible money, not because it's the only way anyone can ever come up with money to support a system development effort. In fact, even as recently as ten and fifteen years ago it was common to support software development in other ways. For example, those DIGITAL operating systems that were free, even in the early seventies, were developed by people who were paid for their work. Many useful programs has been developed at universities. Nowadays those programs are often sold , but fifteen years ago they were usually free, yet the people were paid for their work.

When you have something like a program, like an infinite sandwich, like a road, which has to be built once, but once it is built it pretty much doesn't matter how much you use it, there's no cost in using it, generally it's better if we don't put any price on using it. And there are plenty of those things that we develop now, and pay people to build. For example, all the streets out there. It's very easy to find people who will program without being paid; it really is impossible to find people who will build streets without being paid. Building streets is not creative and fun like programming. But we have plenty of streets out there, we do come up with the money to pay them, and it's much better the way we do it than if if we said: "Let's have companies go and build

streets and put toll booths up, and then every time you turn another street corner, you pay another toll. And then the companies that picked the good places to put their streets, they will be profitable, and the others will go bankrupt."

There's a funny thing that happens whenever someone comes up with a way of making lots of money by hoarding something. Until that time you've probably had lots and lots of people who were really enthusiastic and eager to work in that field, the only sort of question is how can they get any sort of livelihood at all. If we think of mathematicians for example, there are a lot more people who want to be pure mathematicians than there is funding for anybody to be pure mathematicians. And even when you do get funding, you don't get very much, they don't live well. And for musicians it's even worse. I saw a statistics for how much the average musician, the average person devoting most of his time trying to be a musician, in Massachusetts made; it was something like half the median income or less. It is barely enough to live on, it's difficult. But there are lots of them trying to do that. And then, somehow when it gets generally possible to get very well paid to do something, all those people disappear, and people start saying "nobody will do it unless they get paid that well".

And I saw this happen in the field of programming. The very same people who used to work at the AI lab and get payed very little and love it, now wouldn't dream of working for less than fifty thousand dollars a year. What happened? When you dangle before people the possibility of making lots of money, when they see that other people doing similar work are getting paid that much money, they feel that they should get the same, and thus noone is willing to continue the old way. And it's easy after this has happened to think that paying people a lot of money is the only way it could be, but that's not so. If the possibility of making a lots of money did not exist, you would have people who would accept doing it for a little money, specially when it's something that is creative and fun.

Now I saw the unique world of the AI lab destroyed, and I saw that selling software was an intrinsic part of what had destroyed it, and I saw also, as I explained before, how you need to have free software in order to have a community like that. But then thinking about it more, I realized all these ways in which hoarding software hurts all of society, most specially by pressuring people to sell out their neighbours and causing social decay. The same spirit that leads people to watch while somebody in the street is getting stabbed and not tell anyone. The spirit that we can see so many companiess all around us displaying all the time. And it was clear to me I had a choice, I could become part of that world and feel unhappy about what I was doing with my life, or I could decide to fight it. So I de-

cided to fight it. I've dedicated my career to try to rebuild the software sharing community, to trying to put an end to the phenomenon of hoarding generally useful information. And the GNU system is a means to this end. It is a technical means to a social end. With the GNU system, I hope to vaccinate the users against the threat of the software hoarders.

Right now the hoarders essentially claims the power to render a person's computer useless. There used to people in the US, most commonly about fifty years ago, they were in the Mafia, they would go up to stores and bars, esspecially bars when bars were illegal of course. They would go up and say: "A lot of places around here have been burning down lately. You wouldn't want your place to burn down, would you? Well we can protect you from fires, you just have to pay us a thousand dollars a month, and we'll make sure you don't have a fire here". And this was called "the protection racket". Now we have something where a person says "You got a nice computer there, and you've got some programs there that you're using. Well, if you don't want those programs to disappear, if you don't want the police to come after you, you better pay me a thousand dollars, and I'll give you a copy of this program with a licence", and this is called "the software protection racket".

Really all they're doing is interfering with everybody else doing what needs to be done, but they're pretending as much to them selves as to the rest of us, that they are providing a useful function. Well, what I hope is that when that software mafia guy comes up and says, "You want those programs to disappear on your computer?", the user can say "I'm not afraid of you any more. I have this free GNU software, and there's nothing you can do to me now."

Now, one of the justifications people sometimes offer for owning software, is the idea of giving people an incentive to produce things. I support the idea of private enterprise in general, and the idea of hope to make money by producing things that other people like to use, but it's going haywire in the field of software now. Producing a proprietary program is not the same contribution to society as producing the same program and letting it be free. Because writing the program is just a potential contribution to society. The real contribution to the wealth of society happens only when the program is used. And if you prevent the program from being used, the contribution dosn't actually happen. So, the contribution that society needs is not these proprietary programs that everyone has such an incentive to make, the contribution we really want is free software, so our society is going haywire because it gives people an incentive to do what is not very useful, and no incentive to do what is useful. Thus the basic idea of private enterprise is not being followed, and you

could even say that the society is neurotic. After all when an individual encourages in others behavior that is not good for that individual we call this a neurosis. Here society is behaving in that fashion, encouraging programmers to do things that is not good for society.

I'm unusual. I'd rather believe that I'm a good member of society and that I'm contributing something, than feel that I'm ripping society off successfully, and that's why I've decided to do what I have done. But every one is at least a little bit bothered by the feeling that they are getting paid to do what's not really useful. So let's stop defending this idea of incentives to do the wrong thing and let's at least try to come up with arrangements to encourage people to do the right thing, which is to make free software.

Thank you.

[After this RMS answered questions for about an hour. I have only included a very few of the questions and answers in this version. The tape was bad, and I didn't have the time to do a proper job on all of it]

Q: Has anyone tried to make problems for you?

A: The only time anyone has tried to make a problem for me was those owners, so called, self-styled owners of Gosling Emacs. Aside from that they have no grounds to do so, so there is not much they can do. By the way, I'd like to call everyone's attention to the way in which people use language to try to encourage people to think certain thoughts and not think of others. Much of the terminology current in the field was chosen by the self-styled software owners to try to encourage you to try to make you see software as similar to material objects that are property, and overlook the differences. The most flagrant example of this is the term "pirate". Please refuse to use to use the term "pirate" to describe somebody who wishes to share software with his neighbour like a good citizen.

I forgot to tell you this: The idea of copyright was invented after the printing press. In ancient times authors copied from each other freely, and this was not considered wrong, and it was even very useful: The only way certain authors works have survived, even in fragments, is because some of them were quoted at length in other works which have survived.

This was because books were copied one copy at the time. It was ten times as hard to make ten copies as it was to make one copy. Then the printing press was invented, and this didn't prevent people from copying books by hand, but by comparison with printing them, copying by hand was so unpleasant that it might as well have been impossible.

When books could only be made by mass production, copyright then started

to make sense and it also did not take away the freedom of the reading public. As a member of the public who didn't own a printing press, you couldn't copy a book anyway. So you weren't loosing any freedom just because there were copyrights. Thus copyright was invented, and made sense morally because of a technological change. Now the reverse change is happening. Individual copying of information is becoming better and better, and we can see that the ultimate progress of technology is to make it possible to copy any kind of information. [break due to turning of tape]

Thus we are back in the same situation as in the ancient world where copyright did not make sense.

If we consider our idea of property, they come from material objects. Material objects satisfy a conservation law, pretty much. Yes it's true I can break a chalk in half, that's not it, and it gets worn down, it gets consumed. But basically this is one chair [pointing at a chair]. I can't just sort of snap my finger and have two chairs. The only way to get another one is to build it just the way the first one was build. It takes more raw materials, it takes more work of production, and our ideas of property were evolved to make moral sense to fit these facts.

For a piece of information that anyone can copy, the facts are different. And therefor the moral attitudes that fit are different. Our moral attitudes comes from thinking how much it will help people and how much it will hurt people to do certain things. With a material object, you can come and take away this chair, but you couldn't come and copy it. And if you took away the chair , it wouldn't be producing anything, so there's no excuse. I somebody says: "I did the work to make this one chair, and only one person can have this chair, it might as well me", we might as well say: "Yeah, that makes sense". When a person says: "I carved the bits on this disk, only one person can have this disk, so don't you dare take it away from me", well that also make sense. If only one person is going to have the disk, it might as well be the guy who owns that disk.

But when somebody else comes up and says: "I'm not going to hurt your disk, I'm just gonna magically make another one just like it and then I'll take it away and then you can go on using this disk just the same as before", well it's the same as if someone said: "I've got a magic chair copier. You can keep on enjoying your chair, sitting in it, having it always there when you want it, but I'll have a chair too". That's good.

If people don't have to build, they can just snap their fingers and duplicate them, that's wonderful. But this change in technology doesn't suit the people who wants to be able to own individual copies and can get money for individual copies. That's an idea that only fits con-

served objects. So they do their best to render programs like material objects. Have you wondered why, when you go to the software store and buy a copy of a program it comes in something that looks like a book? They want people to think as if they were getting a material object, not to realize what they have really got in the form of digital copyable data.

What is a computer after all but a universal machine? You've probably studied universal Turing machines, the machines that can imitate any other machine. The reason a universal machine is so good is because you can make it imitate any other machine and the directions can be copied and changed, exactly the things you can't do with a material object. And those are is exactly what the software hoarders want to stop the public from doing. They want to have the benefit of the change in technology, to universal machines, but they don't want the public to get that benefit.

Essentially they are trying to preserve the "material object age", but it's gone, and we should get our ideas of right and wrong inync" with the actual facts of the world we live in.

Q: So it boils down to ownership of information. Do you think there are any instances where, you opinion, it's right to own information?

A: With information that's not generally useful, or is of a personal nature, I would say it's OK. In other words not information about how to do things, but information about what you intend to do. Information whose only value to others is speculative, that is they can take some money away from you, but they can't actually create anything with it. It's perfectly reasonable I'd say to keep that sort of thing secret and controlled.

But in terms of creative information, information that people can use or enjoy, and that will be used and enjoyed more the more people who have it, always we should encourage the copying.

Därför kallas den "han"

Nu har vi kommit på varför man säger "han" om datorn:

Den bör skötas av kvinnor, gärna så många som möjligt.

Den måste matas med information, saknar förmåga till egna initiativ.

Den tål inte överbelastning.

Den gör många fel, utan att själv rå för det.

Den försöker att hålla måttet, befaras med åren bli utbytt mot intelligentare utländska konkurrenter.

Den måste underhållas hela tiden.

Den är tämligen enkelspårig.

Den fordrar stort tålamod vid kontaktsvårigheter.

Den behövs för framtida utveckling.

Den är ett ekonomiskt alternativ.