LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 491: Professional Experience

Programming Tools

Instructor: Haidar M. Harmanani

What are tools for?

- Creating code modules
  - compiler
- Creating program from modules
  - linker
- Compiling groups of programs (dependencies)
- Debugging code
  - tracer, debugger, code checker
- Profiling and optimization
- Documentation: derive from code
- Coordination and “memory”
- Testing
- User installation
- User feedback

Compiler

- Convert source code to object modules
- .o: external references not yet resolved

```
$ nm
         U printf
0000000000000000 t
0000000000000000 d
0000000000000000 b
0000000000000000 r
0000000000000000 ?
0000000000000000 a *ABS*
0000000000000000 T c
0000000000000000 a const.c
0000000000000048 T main
```


Linker

- Combine .o and .so into single a.out executable module
- .so/.dll: dynamically loaded at run-time
- see “dl”
- `$ ldd a.out`

```
libc.so.1 =>     /usr/lib/libc.so.1
    libdl.so.1 =>    /usr/lib/libdl.so.1
    /usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1
```

Creating a static library

- static library for linking: libsomething.a
  - create .o files: gcc –c helper.c
  - ar rlv libsomething.a *.o
  - ranlib libsomething.a
  - use library as gcc –L/your/dir –lsomething

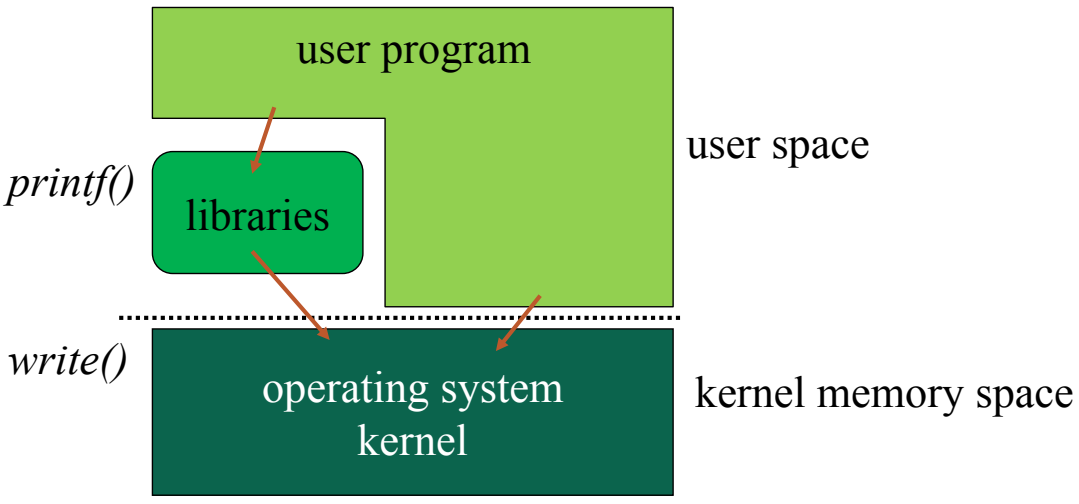
Creating a dynamic library

- Details differ for each platform
- `gcc -shared -fPIC -o libhelper.so *.o`
- use same as for static (-l*library*)
- also `LD_LIBRARY_PATH`

Testing

- Every module and functionality needs to have an (automated) test
- Regression testing: change -> test old functionality
- Easy for simple functions
- Screen input/output?
- Complicated “test harness”

Program tracing



strace

- Simple debugging: find out what system calls a program is using
- -T for timing
- `$ strace –t –T cat foo`

```
14:26:59 open("foo", O_RDONLY|O_LARGEFILE) = 3 <0.000712>
14:26:59 fstat(3, {st_mode=S_IFREG|0644, st_size=6, ...}) = 0 <0.000005>
14:26:59 brk(0x8057000)                 = 0x8057000 <0.000011>
14:26:59 read(3, "hello\n", 32768)      = 6 <0.000010>
14:26:59 write(1, "hello\n", 6hello
)         = 6 <0.000015>
14:26:59 read(3, "", 32768)             = 0 <0.000005>
14:26:59 close(3)                       = 0 <0.000010>
14:26:59 _exit(0)                       = ?
```


Memory utilization: top

Show top consumers of CPU and memory

```
load averages:  0.42,  0.22,  0.16                                  14:17:35
274 processes: 269 sleeping, 1 zombie, 3 stopped, 1 on cpu
CPU states: 81.3% idle,  5.2% user, 13.4% kernel,  0.1% iowait,  0.0% swap
Memory: 512M real, 98M free, 345M swap in use, 318M swap free

   PID USERNAME THR PRI NICE  SIZE   RES STATE    TIME    CPU COMMAND
   144 root       1  53    0 3384K 1728K sleep   33.3H  3.67% ypserv
 11011 hgs        1  48    0 2776K 2248K sleep    0:00  0.57% tcsh
 11040 hgs        1  55    0 1800K 1352K cpu/0    0:00  0.39% top
   281 root       1  58    0 4240K 2720K sleep  313:03  0.38% amd
 10933 kbutler    1  58    0   11M 8376K sleep    0:00  0.17% lisp
  1817 yjh9       1  58    0 8968K 7528K sleep    0:39  0.10% emacs
 13955 yjh9       1  58    0 8496K 7200K sleep    2:47  0.09% emacs
```

Debugging

- Interact with program while running
  - step-by-step execution
    - instruction
    - source line
    - procedure
  - inspect current state
    - call stack
    - global variables
    - local variables

Debugging on Linux

- Requires compiler support:
  - generate mapping from PC to source line
  - symbol table for variable names
- Steps:

```
$ gcc -g -o loop loop.c
$ gdb loop
(gdb) break main
(gdb) run foo
Starting program: src/test/loop

Breakpoint 1, main (argc=2, argv=0xffbef6ac) at loop.c:5
5         for (i = 0; i < 10; i++) {
```

gdb

```
(gdb) n
6 printf("i=%d\n", i);
(gdb) where
#0  loop (i=1) at loop.c:4
#1  0x105ec in main (argc=2,
      argv=0xffbef6a4) at loop.c:11
(gdb) p i
$1 = 0
```

```
(gdb) break 9
Breakpoint 2 at 0x105e4: file
     loop.c, line 9.
(gdb) cont
Continuing.
i=0
i=1
...
Breakpoint 2, main (argc=1,
     argv=0xffbef6ac) at
     loop.c:9
9  return 0;
```


gdb hints

- Make sure your source file is around and doesn't get modified
- Does not work (well) across threads
- Can be used to debug core dumps:

```
$ gdb a.out core
#0  0x10604 in main (argc=1, argv=0xffbef6fc) at loop.c:14
*s = '\0';
(gdb) print i
$1 = 10
```

gdb - execution

run *arg*	run program
call *f(a,b)*	call function in program
step *N*	step *N* times into functions
next *N*	step *N* times over functions
up *N*	select stack frame that called current one
down *N*	select stack frame called by current one

gdb – break points

break *main.c:12*	set break point
break *foo*	set break at function
clear *main.c:12*	delete breakpoint
info break	show breakpoints
delete *1*	delete break point 1
display *x*	display variable at each step

Other Debuggers

- Built-in debuggers in Xcode, MS Visual Studio, Eclipse, Netbeans, …
- Much simpler to use
- Commands are integrated in a GUI

Installation

- Traditional:
  - tar (archive) file
  - compile
  - distribute binaries, documentation, etc.
- InstallShield
- Linux RPM

Building programs

- Programs consist of many modules
- Dependencies:
  - if one file changes, one or more others need to change
  - `.c` depends on `.h` -> re-compile
  - `.o` depends on `.c` -> re-compile
  - executable depends on `.o`'s -> link
  - library depends on .o -> archive
  - recursive!

make

- `make` maintains dependency graphs
- based on modification times
- Makefile as default name
- `make [-f makefile] [target]`
- if node newer than child, remake child

```
target ...: dependency
  command
```

make

```
all: hello clean
clean:
 rm –f *.o
helper.o: helper.c
OBJ = helper.o \
  hello.o
hello: $(OBJ)
 $(CC) $(CFLAGS) $(LDFLAGS) –o $@ $(OBJ)
```


make variables

- implicit rules, e.g., a .c file into .o
  - .c.o:
  - $(CC) $(CFLAGS) $<

$@	name of current target
$?	list of dependencies newer than target
$<	name of dependency file
$*	base name of current target
$%	for libraries, the name of member

make depend

```
depend: $(CFILES) $(HFILES)
 $(CC) $(CFLAGS) –M $(CFILES) > .state
# works for GNU make and BSD make
#if 0
include .state
#endif
#include ".state"
```


make environment

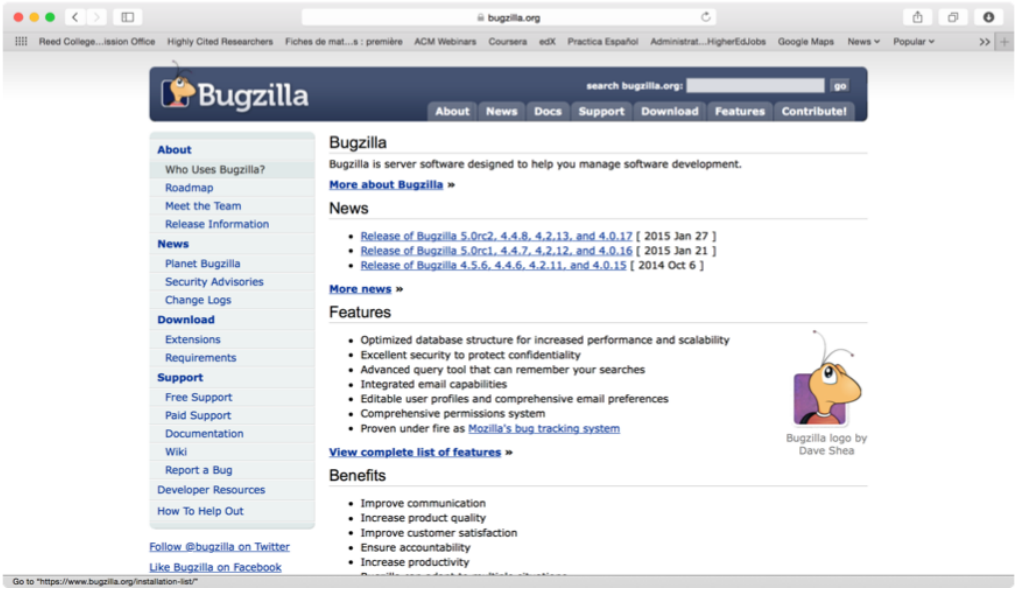
- Environment variables (**PATH, HOME, USER**, etc.) are available as **$(PATH)**, etc.
- Also passed to commands invoked
- Can create new variables (gmake):

```
export FOOBAR = foobar
```

User feedback – bug tracking

- Automatically capture system crash information
  - non-technical users
  - privacy?
  - e.g., Netscape Talkback
- User and developer bug tracking
  - make sure bugs get fixed
  - estimate how close to done

Bug tracking: Bugzilla



Development models

- Integrated Development Environment (IDE)
  - integrate code editor, compiler, build environment, debugger
  - graphical tool
  - single or multiple languages
  - Eclipse, netbeans (Cross Platforms)
  - Xcode (MacOS)
  - VisualStudio (Windows)
- Unix model (Linux and MacOs)
  - individual tools, command-line
  - On Mac: macports Solutions (https://www.macports.org/)

Source code management

- problem: lots of people working on the same project
  - source code (C, Perl, ...)
  - documentation
  - specification (protocol specs)
- versions
  - released – maintenance only
  - stable – about to be released, production use
  - development, beta
- different hardware and OS versions

Version control systems

- Version control (or revision control, or source control) is all about managing multiple versions of documents, programs, web sites, etc.
  - Almost all "real" projects use some kind of version control
  - Essential for team projects, but also very useful for individual projects
- Some well-known version control systems are CVS, Subversion, Mercurial, and Git
  - CVS and Subversion use a "central" repository; users "check out" files, work on them, and "check them in"
  - Mercurial and Git treat all repositories as equal
- Distributed systems like Mercurial and Git are newer and are gradually replacing centralized systems like CVS and Subversion

Why version control?

- For working by yourself:
  - Gives you a "time machine" for going back to earlier versions
  - Gives you great support for different versions (standalone, web app, etc.) of the same basic project
- For working with others:
  - Greatly simplifies concurrent work, merging changes
- For getting an internship or job:
  - Any company with a clue uses some kind of version control
  - Companies without a clue are bad places to work

cvs: overview

- version control system
- see also RCS or SCCS
- collection of directories, one for each module
- release control
- concurrent revisions: “optimistic”
- network-aware
- single master copy (‘repository’) + local (developer) copies
- see http://www.cs.columbia.edu/~hgs/cvs

What cvs isn’t/doesn’t...

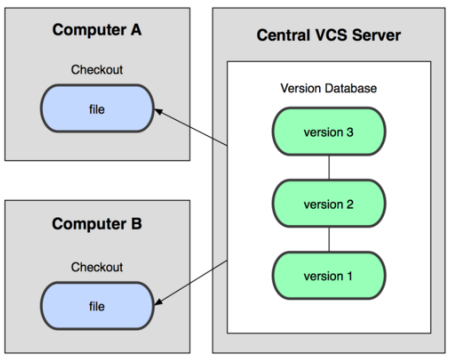
- build system
- project management
- talking to your friends
- change control:
  - all changes are isolated vs. single logical change
  - bug fix tracking
  - track change verification
- testing program (regression testing)
- work flow or process model

Why Git?

- Git has many advantages over earlier systems such as CVS and Subversion
  - More efficient, better workflow, etc.
  - Came out of Linux development community (Linus Torvalds, 2005)
- Initial goals:
  - Speed
  - Support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects like Linux efficiently
- Best competitor: Mercurial
  - Same concepts, slightly simpler to use
  - Not popular

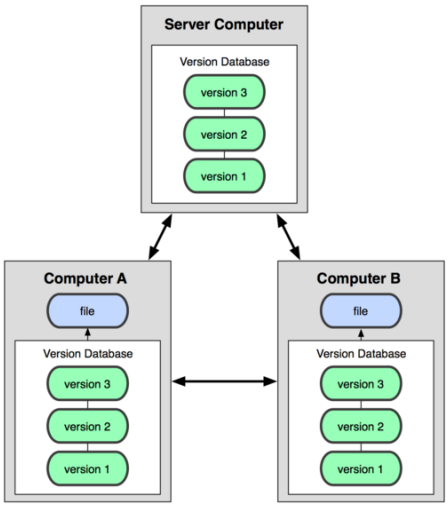
Git uses a distributed model

Centralized Model



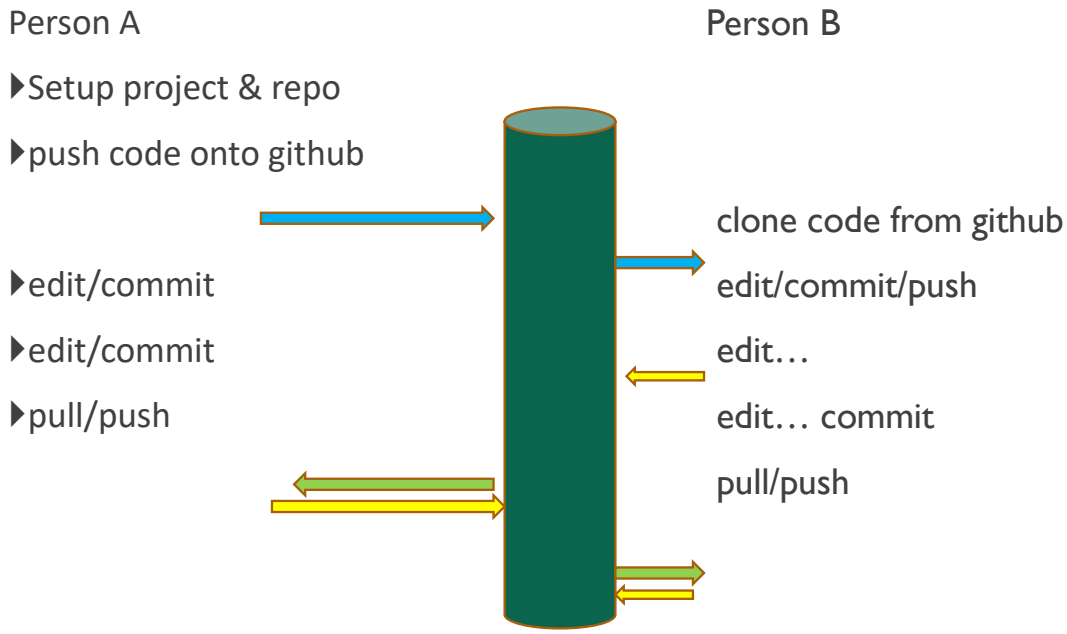
(CVS, Subversion, Perforce)

Distributed Model

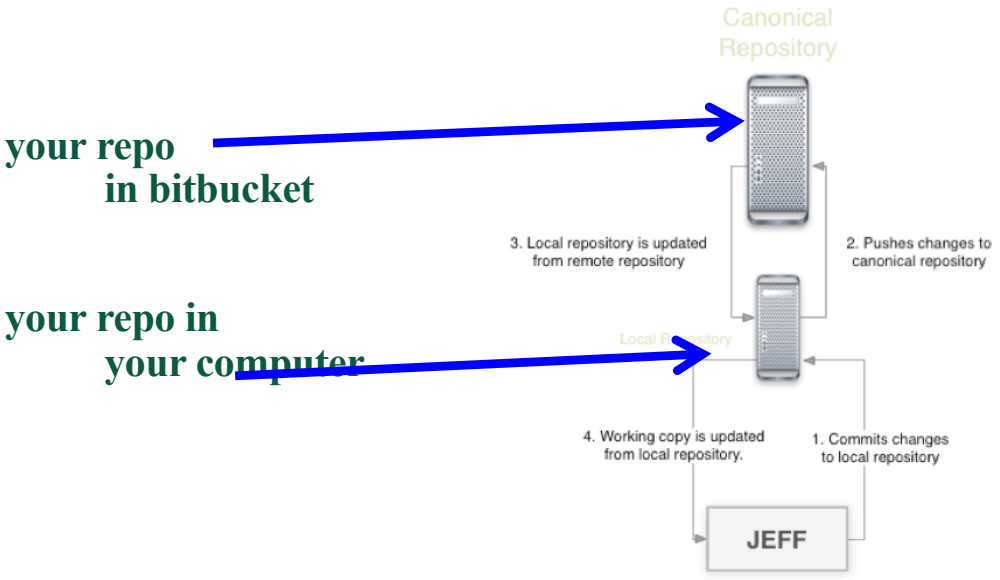


(Git, Mercurial)

Typical workflow

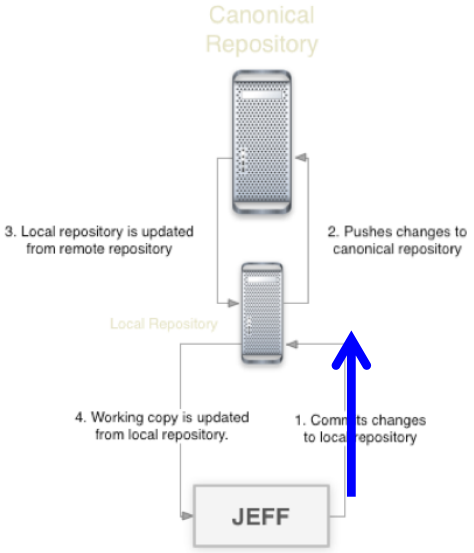


Bitbucket: A free online git repository



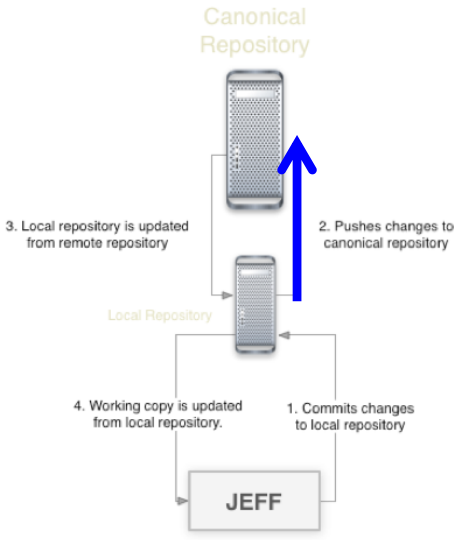
Bitbucket: A free online git repository

- add and commit

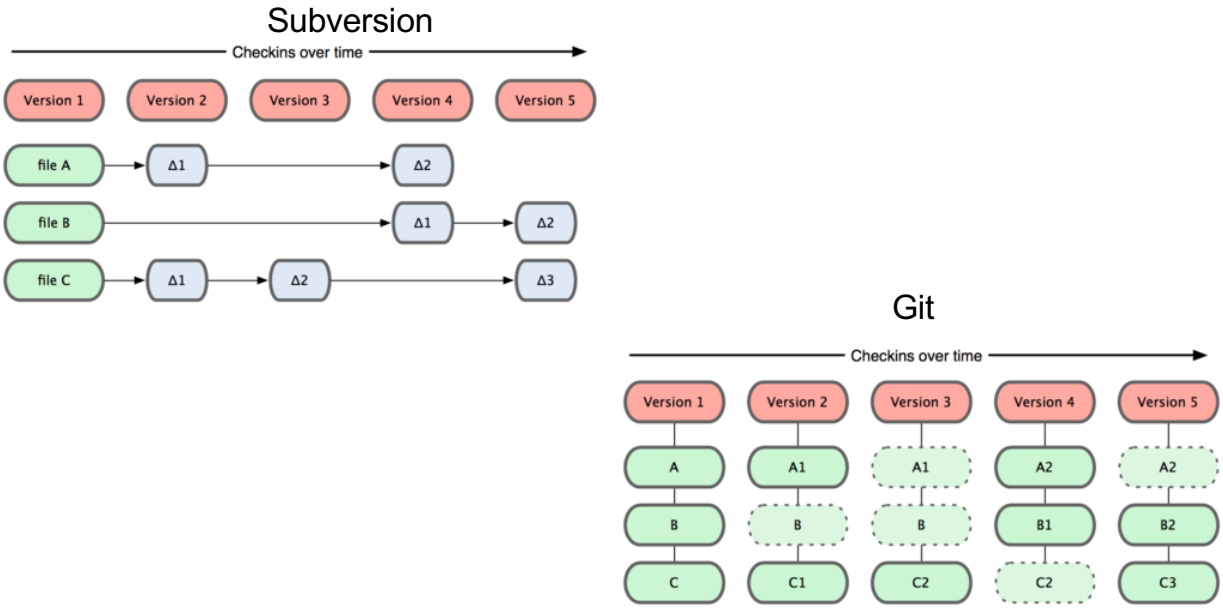


Bitbucket: A free online git repository

- push



Git takes snapshots



Git uses checksums

- In Subversion each modification to the central repo incremented the version # of the overall repo.
- How will this numbering scheme work when each user has their own copy of the repo, and commits changes to their local copy of the repo before pushing to the central server?????
- Instead, Git generates a unique SHA-1 hash – 40 character string of hex digits, for every commit. Refer to commits by this ID rather than a version number. Often we only see the first 7 characters:
  - 1677b2d Edited first line of readme
  - 258efa7 Added line to readme
  - 0e52da7 Initial commit

Git commands

command	description
`git clone` ***`url [dir]`***	copy a git repository so you can add to it
`git add` ***`files`***	adds file contents to the staging area
`git commit`	records a snapshot of the staging area
`git status`	view the status of your files in the working directory and staging area
`git diff`	shows diff of what is staged and what is modified but unstaged
`git help` ***`[command]`***	get help info about a particular command
`git pull`	fetch from a remote repo and try to merge into the current branch
`git push`	push your new branches and data to a remote repository
others: `init, reset, branch, checkout, merge, log, tag`	

GitHub

GitHub

- Largest open source git hosting site
- Public and private options
- You can get free space for open source projects or you can pay for private projects.
  - Special discount for educational purposes
- User-centric rather than project-centric
- Try it
  - Log onto github
  - Walk through the tutorials

GitHub

Using GitHub

- Set up a user account
  - public account is free, students can request 5 private for $7 a month
- Create a repository
  - e.g. JavaDemos
- Install local client
- Add github as a “remote”
- Push your master branch
- Add collaborators (if public, everyone can read, but not write. If private, must be collaborator to even read)

http://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes

Memory leaks and overruns

- see http://www.cs.colorado.edu/homes/zorn/public_html/MallocDebug.html
- Graphical tool: purify
- Simple library: ElectricFence
  - catches
    - overruns a `malloc()` boundary
    - touch (read, write) memory released by `free()`
  - places inaccessible (VM) memory page after each allocation
  - only for debugging (memory hog)

ElectricFence

```
gcc -g test.c -L/home/hgs/sun5/lib -lefence
-o test
```

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *s = malloc(5);
  strcpy(s, "A very long string");
  return 0;
}
```

use gdb:

```
Program received signal SIGSEGV, Segmentation fault.
0xff2b2f94 in strcpy () from /usr/lib/libc.so.1
(gdb) up
#1  0x10adc in main (argc=1, argv=0xffbef684) at test.c:10
10        strcpy(s, "A very long string");
```


dmalloc – memory leaks

```
$ dmalloc -l logfile -i 100 high
```

```
setenv DMALLOC_OPTIONS
debug=0x4f47d03,inter=100,log=logfile
```

create file

```
#ifdef DMALLOC
#include "dmalloc.h"
#endif
```

link: `gcc -g -DDMALLOC dmalloc.c -L/home/hgs/sun5/lib/ -ldmalloc -o dm`

run program

Profiling: execution profile of call graph

```
int inner(int x) {
  static int sum;
  sum += x;
  return sum;
}
double outer(int y) {
  int i;
  double x = 1;
  double sum = 0;
  for (i = 0; i < 10000; i++) {
    x *= 2; sum += inner(i + y);
  }
  return sum;
}
```

```
int main(int argc, char *argv[])
{
  int i;
  for (i = 0; i < 1000; i++)
 {outer(i);}
  exit(0);
}
```

Where your program spent its time and which functions called which other functions while it was executing

profiling

- `gcc –pg nested.c –o nested`
- change function invocation to do logging (call `_mcount`)
- also, PC sampling (e.g., 100 times/second)
- generate a *call graph*
- `gprof nested gmon.out`

gprof flat profile

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 59.50      2.88     2.88                             internal_mcount
 21.69      3.93     1.05     1000     1.05     1.92  outer
 17.15      4.76     0.83 10000000     0.00     0.00  inner
  0.83      4.80     0.04     1000     0.04     0.04  _libc_write
  0.83      4.84     0.04                             _mcount
  0.00      4.84     0.00     2000     0.00     0.00  _realbufend
  0.00      4.84     0.00     2000     0.00     0.00  ferror_unlocked
  0.00      4.84     0.00     1890     0.00     0.00  .mul
  0.00      4.84     0.00     1000     0.00     0.04  _doprnt
  0.00      4.84     0.00     1000     0.00     0.04  _xflsbuf
  0.00      4.84     0.00     1000     0.00     0.00  memchr
  0.00      4.84     0.00     1000     0.00     0.04  printf
```


gprof call graph

Time spent in function and its children

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]     60.0    2.88    0.00                 internal_mcount [1]
                0.00    0.00       1/3           atexit [15]
-----------------------------------------------
                1.05    0.87    1000/1000        main [3]
[2]     40.0    1.05    0.87    1000         outer [2]
                0.83    0.00 10000000/10000000     inner [5]
                0.00    0.04    1000/1000        printf [6]
-----------------------------------------------
                0.00    1.92       1/1           _start [4]
[3]     40.0    0.00    1.92       1         main [3]
                1.05    0.87    1000/1000        outer [2]
                0.00    0.00       1/1           exit [19]
-----------------------------------------------
                                                 <spontaneous>
[4]     40.0    0.00    1.92                 _start [4]
                0.00    1.92       1/1           main [3]
                0.00    0.00       2/3           atexit [15]
-----------------------------------------------
                0.83    0.00 10000000/10000000     outer [2]
[5]     17.3    0.83    0.00 10000000         inner [5]
```

doc++

- documentation system for C/C++ and Java
  - generate LaTeX for printing and HTML for viewing
  - hierarchically structured documentation
  - automatic class graph generation (Java applets for HTML)
  - cross references
  - formatting (e.g., equations)

doc++

- Special comments: /** */, ///

int main (int argc, char *argv[])

This is the famous "hello world" program, with more comments than code

---

Documentation

This is the famous "hello world" program, with more comments than code.

Returns:
0 if no error s
Parameters:
argc - number of argument
argv - command-line arguments
Author:
H.W. Programmer

alphabetic index hierarchy of classes

---

this page has been generated automatically by doc++

doc++

```
/**
This is the famous "hello world" program, with more comments than
 code.
@author H.W. Programmer
@return 0 if no error
@param argc  number of argument
@param argv  command-line arguments
@returns
*/
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("Hello World!");
  return 0;
}
```


doc++

- docify to create minimal version
- doc++ -d outdir hello.c

Other tools useful to know

- configuration:
  - autoconf: configuration files
  - automake: make files
- code generation:
  - indent (e.g., `indent -kr -i2 hello.c`): automated indentation for C programs
  - lex, flex: lexical analyzers
  - yacc, bison: compiler generator

Other Tools

NETWORKING

Communication: ssh, sftp, mirror

Peer-to-peer: uTorrent, BitTorrent, …

EDITING/TYPESETTING

vi, emacs (edit)

Latex (typsetting)

BibDesk (Managing References)

MS Project/OmniPlan (Project Management)