Linking Process

Static Linking

- Code and variables resolved at compile time by interpreter ld, and copied into target application as a stand-alone executable
- .a filename convention
- Benefits: No dependency problems (single executable file)
- Cost: Large file size, Library code possibly loaded multiple times in memory

```
// file get15.c
// gcc -c get15.c


int get15() {
  return 15;
}

// ar -cur libget15.a get15.o
```

Lib

```
// file main.c
// gcc main.c get15.o
// gcc main.c libget15.a
// gcc main.c -lget15

int main() {
  return get15();
}
```

-lget15
libget15.so.1.0

Main

libldacBT_abr.so

gcc main.c -lldacBT_abr

shortcut

libldacBT_abr.so.2.

convert main.c -> executable object

find the library lldacBT_abr by looking at default locations

```
/lib      ..... liblua.so   -llua
/lib64    .... liblua.so   -llua
```

If found mark the library for linking

If not found:

a) LD_LIBRARY_PATH

b) gcc main.c -L/home/omar/codes/lib -lldacBT_abr

./a.out

load both lib + main exe obj code

speedcrunch
-> libm.so.6

libm.so.7

slowcrunch
-> libm.so.7

Linking Process (cont.)

Dynamic Linking

- Code and variables resolved at load time by runtime interpreter ld-linux.so.2, and copied into target executable as symbols
- .so file convention
- Benefit: Small file size, library code loaded once in shared memory
- Cost: Dependency management

```
// file get15.c
// gcc -shared get15.c -o libget15.so
```

```
// file main.c
// gcc main.c -lget15
```

libget15.so

-lget15 -> libget15

Linking Process (cont.)

```
# Output: readelf -S a.out
[21] .dynamic          DYNAMIC          0000000000003de8  00002de8
     00000000000001f0  0000000000000010  WA       7     0     8

#Output: readelf -d a.out
Dynamic section at offset 0x2de8 contains 27 entries:
  Tag         Type                         Name/Value
 0x0000000000000001 (NEEDED)              Shared library: [libget15.so]
 0x0000000000000001 (NEEDED)              Shared library: [libc.so.6]
 0x000000000000000c (INIT)                0x1000
# ... continues
```

- Use symbolic links when dealing with multiple versions

```
gcc -shared -Wl,-soname,libget15.so.1 -o libget15.so.1.0 get15.c
ln -sf libget15.so.1.0 libget15.so.1
ln -sf libget15.so.1.0 libget15.so
```

libget15.so

-lget15 -> libget15

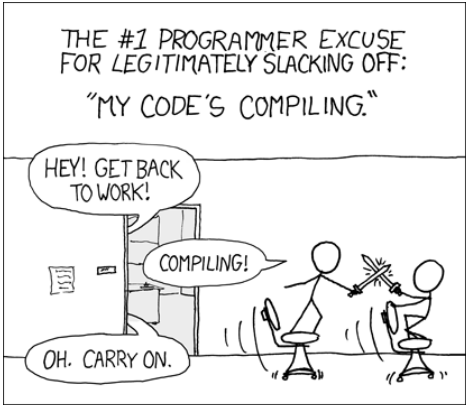
③ gcc main.c -lget15

① libget15.so.1.0

② libget15.so.1
libget15.so

Automated Builds

gcc libreoffice



- Single-file programs do not work well when code gets large
- Larger programs are split into multiple files

Automated Builds (cont.)

- Retyping commands is wasteful (Use ↑ or `CTRL+R` as shortcut)

GNU Make

- A utility for automatically compiling ("building") executables and libraries from source code.
- Often used for C programs, but not language-specific
- Follows Makefile format

```
myprogram : file1.c file2.c file3.c
        gcc -o myprogram file1.c file2.c file3.c
```

- Launch as `make` for first target, or `make myprogram` with direct target name
- Runs commands only if needed (based on timestamp)

Automated Builds (cont.)

automake

shell script

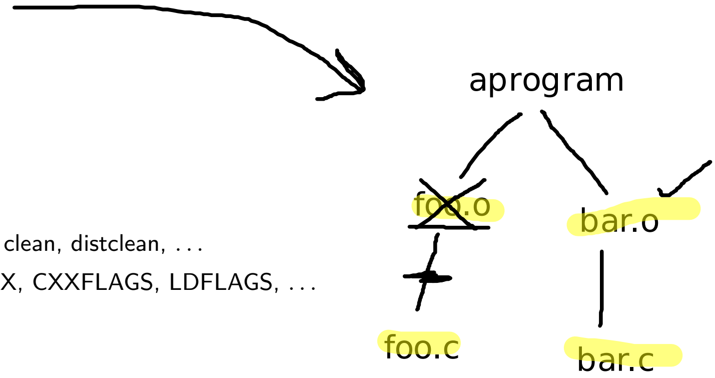
$ make all

```
all: aprogram

aprogram : foo.o bar.o
        gcc -o aprogram foo.o bar.o

foo.o: foo.c
        gcc -c foo.c

bar.o: bar.c
        gcc -c bar.c
```

3

2

1



- Standard Makefile targets: all, install, clean, distclean, . . .
- Standard Variables: CC, CFLAGS, CXX, CXXFLAGS, LDFLAGS, . . .

Debugging Using GDB / DDD

Code Debugging

- Step through a program line by line
- Inspect variables and objects as it steps through
- Inspect disassembled code as it steps through
- Inspect call stack as it steps through

Debugging Using GDB / DDD (cont.)

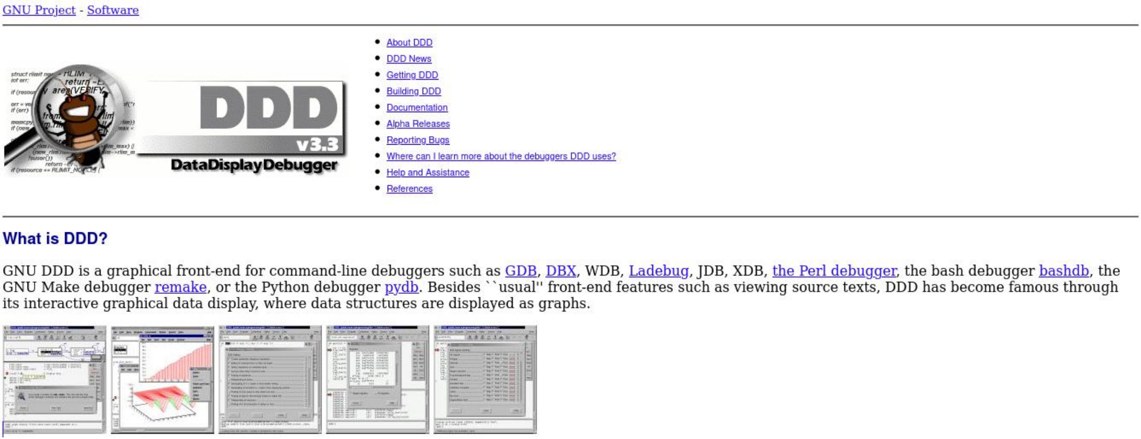
Code Debugging

GNU Debugger GDB

- Compile Time: `gcc -g myCode.c`
- Run Time: `gdb ./a.out`, followed by `run arg1 arg2`
- NCurses based frontend using `gdb ./a.out -tui`, or launching as normal, and issuing `layout src` after inserting any breakpoint.
- Commands:
  - Breakpoints `break file.c:10`, OR `break 10`, OR `break myFunc`
  - Delete breakpoints using `delete` or specifically by name
  - To view code: `list`
  - To view disassembled code: mintinlinebash|disassemble myFunc|
  - Iterate through code: `continue`, `step`, `next`
  - Inspect variables using: `print variableName`

Debugging Using GDB / DDD (cont.)

Code Debugging



GNU Project - Software

- About DDD
- DDD News
- Getting DDD
- Building DDD
- Documentation
- Alpha Releases
- Reporting Bugs
- Where can I learn more about the debuggers DDD uses?
- Help and Assistance
- References

What is DDD?

GNU DDD is a graphical front-end for command-line debuggers such as GDB, DBX, WDB, Ladebug, JDB, XDB, the Perl debugger, the bash debugger bashdb, the GNU Make debugger remake, or the Python debugger pydb. Besides ``usual'' front-end features such as viewing source texts, DDD has become famous through its interactive graphical data display, where data structures are displayed as graphs.

Debugging for Memory Problems using Valgrind

Memory Profilers

Typical Memory Problems

- Uninitialized Variables
- Read/Write to un-allocated Memory (maybe segfaults generated)
- Deleting or Freeing dynamically created memory twice
- Memory Leaks

```
void f() {
  int *x = malloc(10 * sizeof(int));
  x[10] = 0;
}

int main(int argc, char *argv[]) {
  int n, i;
  f();
  for (i = 0; i < n; i++);
  return 0;
}
```

```
valgrind --leak-check=full ./a.out
```

Debugging for Memory Problems using Valgrind (cont.)

Memory Profilers

```
==23294== Invalid write of size 4
==23294==    at 0x108728: f (in /home/omar/work/codes/c/valgrind/a.out)
==23294==    by 0x108749: main (in /home/omar/work/codes/c/valgrind/a.out)
==23294==  Address 0x5204068 is 0 bytes after a block of size 40 allocated
==23294==    at 0x4C2EF1F: malloc (vg_replace_malloc.c:299)
==23294==    by 0x10871B: f (in /home/omar/work/codes/c/valgrind/a.out)
==23294==    by 0x108749: main (in /home/omar/work/codes/c/valgrind/a.out)
==23294==
==23294== Conditional jump or move depends on uninitialised value(s)
==23294==    at 0x10875D: main (in /home/omar/work/codes/c/valgrind/a.out)
==23294==
==23294== More than 10000000 total errors detected. I am not reporting any more.
==23294==
==23294== HEAP SUMMARY:
==23294==    in use at exit: 40 bytes in 1 blocks
==23294==    total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==23294==
==23294== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==23294==    at 0x4C2EF1F: malloc (vg_replace_malloc.c:299)
==23294==    by 0x10871B: f (in /home/omar/work/codes/c/valgrind/a.out)
==23294==    by 0x108749: main (in /home/omar/work/codes/c/valgrind/a.out)
==23294==
```

Debugging for Memory Problems using Valgrind (cont.)

Memory Profilers

```
==23294== LEAK SUMMARY:
==23294==    definitely lost: 40 bytes in 1 blocks
==23294==    indirectly lost: 0 bytes in 0 blocks
==23294==    possibly lost: 0 bytes in 0 blocks
==23294==    still reachable: 0 bytes in 0 blocks
==23294==    suppressed: 0 bytes in 0 blocks
```

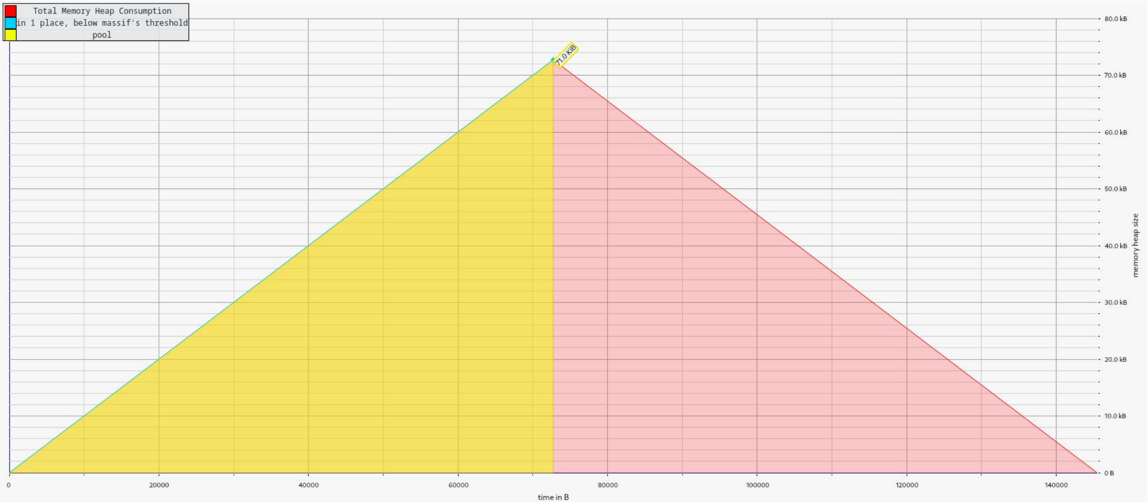
Observing Memory Usage Behavior using Massif

Memory Profilers

- Memory usage as a function of allocation/deallocation event on heap (including stack)
- Usage: `valgrind --tool=massif --time-unit=B ./a.out`
- Visualization usage: `massif-visualizer massif.out.pid`

Observing Memory Usage Behavior using Massif (cont.)

Memory Profilers



Performance Measurement (cont.)

Code Profilers

```c
#include <stdio.h>

void func1(void)
{   printf("\n Inside func1 \n");
    for(int i = 0; i<0xffffffff; i++);
    new_func1();
    return;
}

int main(void)
{
    printf("\n Inside main()\n");
    for(int i = 0; i<0xffffff; i++);
    func1();
    func2();

    return 0;
}
```

```c
void func2(void)
{   printf("\n Inside func2 \n");
   for(int i = 0; i<0xffffffaa; i++);
    return;
}
```

```c
void new_func1(void)
{   printf("\n Inside new_func1()\n");
   for(int i = 0;i <0xffffffee; i++);
    return;
}
```

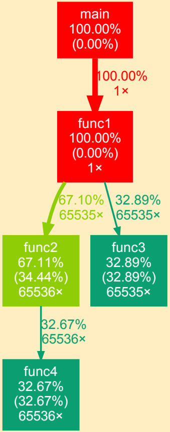
Performance Measurement (cont.)

Code Profilers

gprof2dot

- Can be called with any profiler tools
- Generates interesting graphics for documentation / publications
- Usage:

```
gprof ./a.out | gprof2dot | dot -Teps -o output.eps
```



Cross Compilation

- Compiler support required to merge object files of different languages
- Why? Performance and Native Calls

Wrapper Libraries

- C extension modules (for Python, Cython)
- Java extension modules (for Python, Jython)
- Mex files (for Matlab)
- Swift - Objective C extensions
- . . .

Cross Compilation (cont.)

Example 1 (C Code)

```python
from ctypes import *

so_file="./fputs.so"
myCFunctions=CDLL(so_file)

myCFunctions.myfputs(b"Hello", b"write.txt")
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int myfputs(char *s, char *f) {
  FILE *fp = fopen(f, "w");
  int ret = fputs(s, fp);
  fclose(fp);
  return ret;
}
```

```
gcc -shared file.c -o fputs.so
```