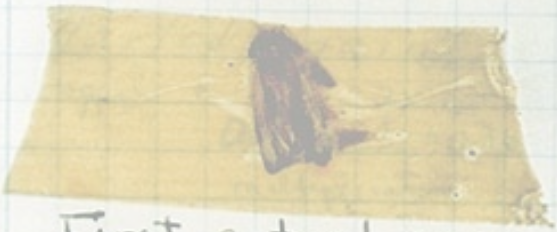
Debugging Linux Applications

Version 1.0



Copy and other Rights

The following Wikipedia articles were used in the making of this study material:

- Call_stack
- Debugging
- Buffer_overflow
- Dynamic memory allocation

Additional material from the GNU libc and GDB manual.

Original content by:
Gilad Ben-Yossef
Yariv Shivek

Debugging

" From then on, when anything went wrong with a computer, we said it had bugs in it."

Rear Admiral Grace Hopper, coined the term ***debugging*** in 1945.

Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

1545
1700 closed down.

Debugging

- **Debugging** is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware thus making it behave as expected.
- Debugging is an art form as much as a science, but many useful debugging methods can be taught and learned.
- This is what this course is about.

Agenda

- Inside the Build Process
- GDB, the GNU Debugger
- Smashing the Stack for Fun and Profit
- The Heap and Dynamic Allocations
- Multi-Threading and Concurrency
- Programmed Debug Assistance
- Post-Mortem
- Debugging Tools

Debugging Linux Applications

Chapter 1

Inside the Build Process

ELF and DWARF

- The Executable and Linking Format (ELF) is the file format standard for executables, objects, shared libraries (and core dumps) in Unix
- Use **`readelf(1)`** to obtain ELF information
- The newest debug information format, compatible with ELF, is DWARF 3

Header Paths

- GCC looks for system headers (included with triangle brackets) in predefined paths
- Only afterwards, GCC searches for your headers (included with quotation marks), according to the **`-I`** options:

```
$ gcc -I/path/to/my/headers src.c
```

Library Paths

To link with libraries, you have to provide the linker with the following:

- **`-larchive`**: Link against ***`archive`***, which may be a statically linking library (.a), or a dynamically linking shared object (.so).
- **`-L`*`searchdir`***: Add ***`searchdir`*** to the list of paths the linker will search. The directory is added before the system default directories.

Debug Information

- The `-g` flag instructs GCC to add debug information
- Even without `-g`, GCC still includes minimal DWARF debug information
- Therefore, before shipping the application, you may wish to strip the executable from all debug symbols:

```
$ strip -d application
```

Optimizations

- The `-On` flag determines the current optimization level (n)
- The Linux kernel, for example, must compile with `-O2`
- When debugging, it is better not to optimize – the code will be easier to follow in the debugger

ldd(1)

- Use **`ldd(1)`** to view the dynamic library dependencies of an application:

```
$ ldd main_dyn
linux-gate.so.1 =>  (0xffffe000)
libdyn.so.1 => libdyn.so.1 (0xb7fe3000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e93000)
/lib/ld-linux.so.2 (0xb7fe7000)
```

Symbols

- Global functions and variables in the source files turn to symbols in the object files.
- The linker statically links object files together using symbols.
- Dynamic linkage uses the GOT (Global Offset Table) and the PLT (Procedure Linkage Table)

Symbols and Name Mangling

- Symbols usually have the same name as their function or variable name.
- However, dynamic linkage (allowing multiple versions of the same DSO) and C++ (with its overloading), create complications.
- Therefore, symbol names will sometimes be mangled

nm(1) – view object symbols

Use **nm(1)** to list symbols from any ELF object files.

```
...
000018b0 D lib_a_global_inited_var
000018bc B lib_a_global_var
000004fd T lib_a_print_msg
000004bc t lib_a_static_print_msg
000018b8 b lib_a_static_var
000005c5 T lib_b_print_msg
00000584 t lib_b_static_print_msg
         U printf@@GLIBC_2.0
```

objdump(1) – ELF Contents

`objdump(1)` contains many options for inspecting the insides of ELF files:

- **`-d/-D`** – disassemble/disassemble all
- **`-S`** – intermix source code with assembly
- **`-x`** – display all available header info
- **`-C`** – demangle symbol names into user-level names (C++)

Using objdump(1) and nm(1)

- **`objdump(1)`** is most commonly when we only have a crash address (`EIP`) and need to understand where we crashed
- **`nm(1)`** is usually used to resolve linkage problems – in order to understand which symbols are defined where

Static Linkage Problems

- Missing symbols
- Using the same name for a static symbol in two different files by mistake (meaning to have only one variable, but defining it static in a header file, for example)
- symbol collisions (sometimes preprocessor macros are used to change function names)

Dynamic Linkage Problems

- As in static, plus:
- Library interface changes require change to the DSO version

Debugging Linux Applications

Chapter 2

GDB, the GNU Debugger

GDB

- GDB is a symbolic source level debugger for Linux (and other) systems. It supports:
  - Starting programs , attaching to running programs or debugging crashed programs
  - Debugging locally or remotely (via gdbserver)
  - Setting breakpoints and watchpoints
  - Examining variables, registers and call stack
  - Changing data and calling functions
  - Automating debug tasks
  - Multi threaded programs

GDB Interfaces

- GDB can be used in two ways:
  - As a standalone interactive program running in the shell which receives text based commands
  - As a back-end to a GUI program, communicating in the MI protocol
- The text interface is more difficult to use, but can do things that the GUI cannot, like automation.
  - We'll get to the GUI later...

Build for Debugging

- In order to effectively debug a program, we need to build the program with debug information
- Debug information saves the relation between the program binary and its source and is required for proper GDB operation
- To build a binary with debug information, pass the `-g` argument to GCC
  - Usually passed via the `CFLAGS` Makefile variable

Starting GDB

- Invoke GDB by running the program `gdb`:
  - `$ gdb application`
- You can also start with a core file:
  - `$ gdb application core`
- You can specify a process ID if you want to debug a running process:
  - `$ gdb program 1234`
    - Make sure not to have a file “1234” in the current directory, or GDB will treat it as a core file...

GDB Commands

- A GDB command is a single line of input
- Command may be truncated if that abbreviation is unambiguous
- Pressing TAB completes commands for you
- A blank line (typing just RETURN) means to repeat the previous command
- Any text from a # to the end of the line is a comment; it does nothing
- The help command is useful :-)

Starting Your Program

- To start your program, use the command `run`
- You can pass any parameters you pass to the program as parameters of the `run` command.
- Example:
  - `(gdb) run 1 "hello world"`
- The program to run is the the one specified on the GDB command line

Controlling Program Execution

- Hitting CTRL+C stops the program
  - In a multi-threaded program all threads stop
- The command `step` steps into the current function
  - Use `stepi` for single instruction stepping
- The command `next` advances to the next source line
- The command `continue` lets the program continue as normal

Source Code Listing

- The command `list` shows the source code at the current position
  - Additional `list` commands show the next lines of source
- You can ask to list a specific file or function using:
  - `(gdb) list my_func`
    - List the source for function `my_func()`
  - `(gdb) list hello.cpp:3`
    - List the file hello.cpp starting from line 3

Symbolic Debugging

- Use the `print` command to show the value of a variable, address or expression:
  - `(gdb) print my_var`
  - `(gdb) print (struct object *)*my_char_ptr`
- Use `display` to follow the changes in above while stepping through the code
- Use `call funcname(params...)` to call a function
  - `(gdb) call debug_func(1)`

Misc. Information

- Use `info threads` to view information on current threads
- Use `info registers` to list the CPU registers and their contents, for a selected stack frame
  - Register name as argument means describe only that register
- Many more `info` commands available
  - See `help info` for detail

Breakpoints

- Use `break [LOCATION] [thread THREADNUM] [if CONDITION]` to set a break point
  - `LOCATION` is a line number, function name or * and an address
  - `THREADNUM` is a thread number. The breakpoint will only be valid for that thread
  - `CONDITION` is a boolean expression. The breakpoint will happen only if it is true
    - Multiple breakpoints at one place are permitted, and useful if conditional

Breakpoints Examples

- `(gdb) break main`
  - Break at entrance to `main()`
- `(gdb) break hello.cpp:3`
  - Break in the third line of hello.cpp
- `(gdb) break thread 3 *0xdeadbeef`
  - Break when `EIP` is 0xdeadbeef in thread 3
- `(gdb) break func if (my_global==42)`
  - Break in `func()` if `my_global` equals 42

Remote Debugging

- Sometime we want to debug a program on a different machine:
  - Problem occurs on remote customer server accessible only via network
  - Can't put source on customer machine
  - Don't have room on customer machine for source and debug information
  - No debugger on customer machine
  - Embedded device

Remote Debugging (cont')

- Let's define terms:
  - Our workstation with the debugger and source shall be called **host**
  - Customer machine, with the executable is called **target**
- On the host we need:
  - Debugger, source, executable and libraries with debug information
- On the target we need:
  - Executable and debugger agent

Debugging Agent

- We don't need a full debugger on the target
- We need a small agent that the debugger can talk to
- The GDB agent is called GDBServer
- It can start a program under the agent or attach to an already running program
- GDB and GDBServer communicate either via a TCP socket or via the serial port

Remote Debug Example

- On the target:
  - **`$ gdbserver /dev/ttyS0 my_prog 12 3`**
    - Load `my_prog` with parameters 12 3 and wait for the debugger to connect on the first serial port
  - **`$ gdbserver 0.0.0.0:9999 my_prog 12 3`**
    - Load `my_prog` with parameters 12 3 and wait for the debugger to connect on TCP port 9999
  - **`$ gdbserver 0.0.0.0:9999 –attach 112`**
    - Attach agent to the process with PID 112 and wait for the debugger to connect on TCP port 9999

Remote Debug Example (cont')

- On the host:
  - **`$ gdb my_prog`**
    - Start GDB
  - **`(gdb) set solib-absolute-prefix /dev/null`**
  - **`(gdb) set solib-search-path /path/to/target/libs`**
    - Set host path to target libraries with debug information
  - **`$ target remote 192.1.2.3:9999`**
    - Connect to GDBServer on IP 192.1.2.3 port 9999

Remote Debugging Tips

- Your first automatic breakpoint will be before `main()` in the guts of the C library
  - So just do `break main` and `continue`
- If the path to the copy of target libraries on the host is wrong or the target and host files do not match, you will see many strange errors
- If the target GDBserver is not installed or the file libthread_db.so is missing, you will not be able to see or debug threads

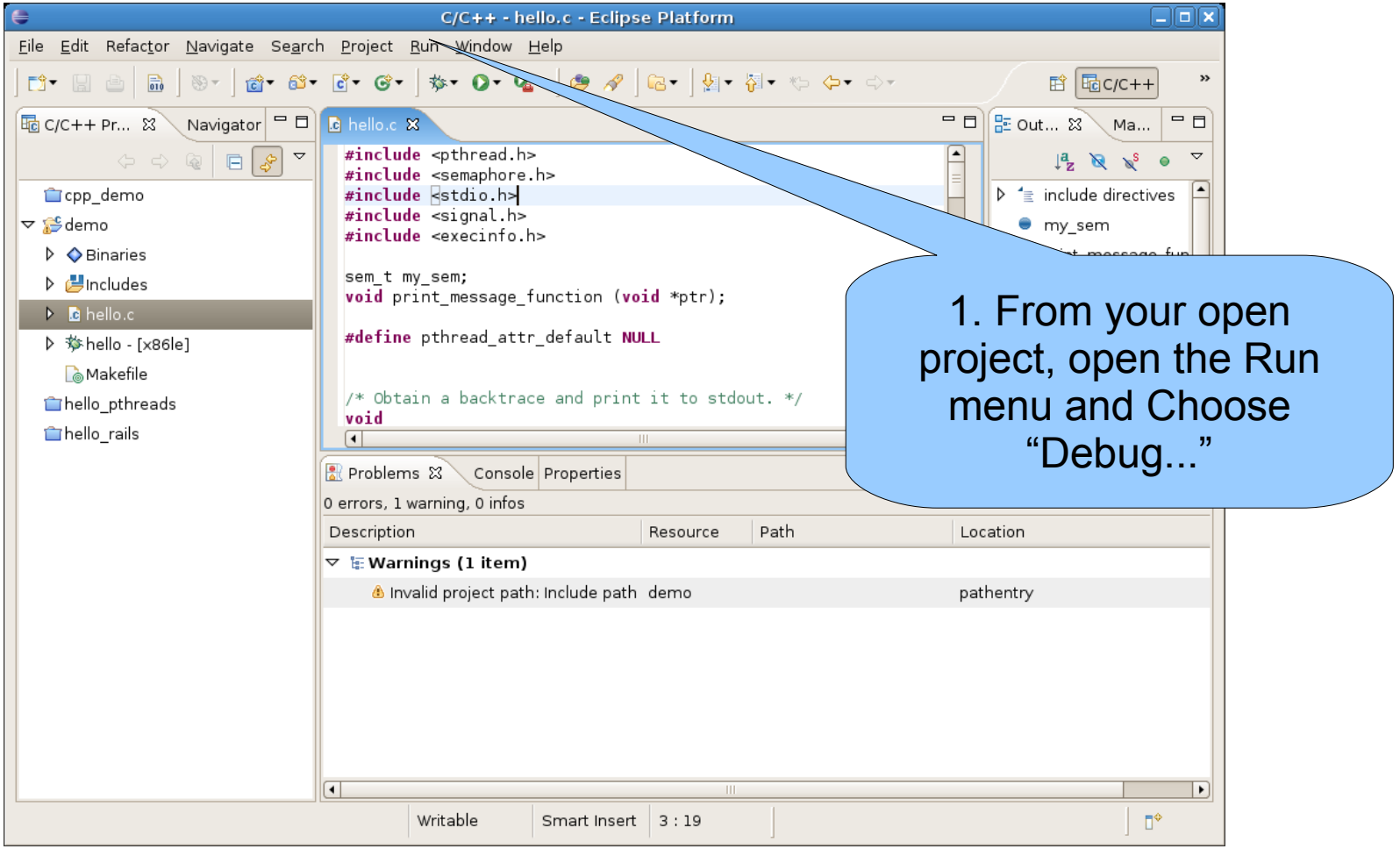
Automation

- You can specify GDB commands to be run automatically when a breakpoint is hit using the `command <breakpoint number>` command
- You can put GDB command in a text file and run it by using the `source <file name>` command
- The file `.gdbinit` is called in such way automatically every time GDB starts

Eclipse

- Eclipse is an open source community whose projects are focused on building an open development platform comprised of extensible frameworks, tools and run times for building, deploying and managing software across the life cycle.
- CDT is the name of the C/C++ development plug-in.
- Eclipse/CDT contains a graphical GDB front end.

Debugging With Eclipse/CDT

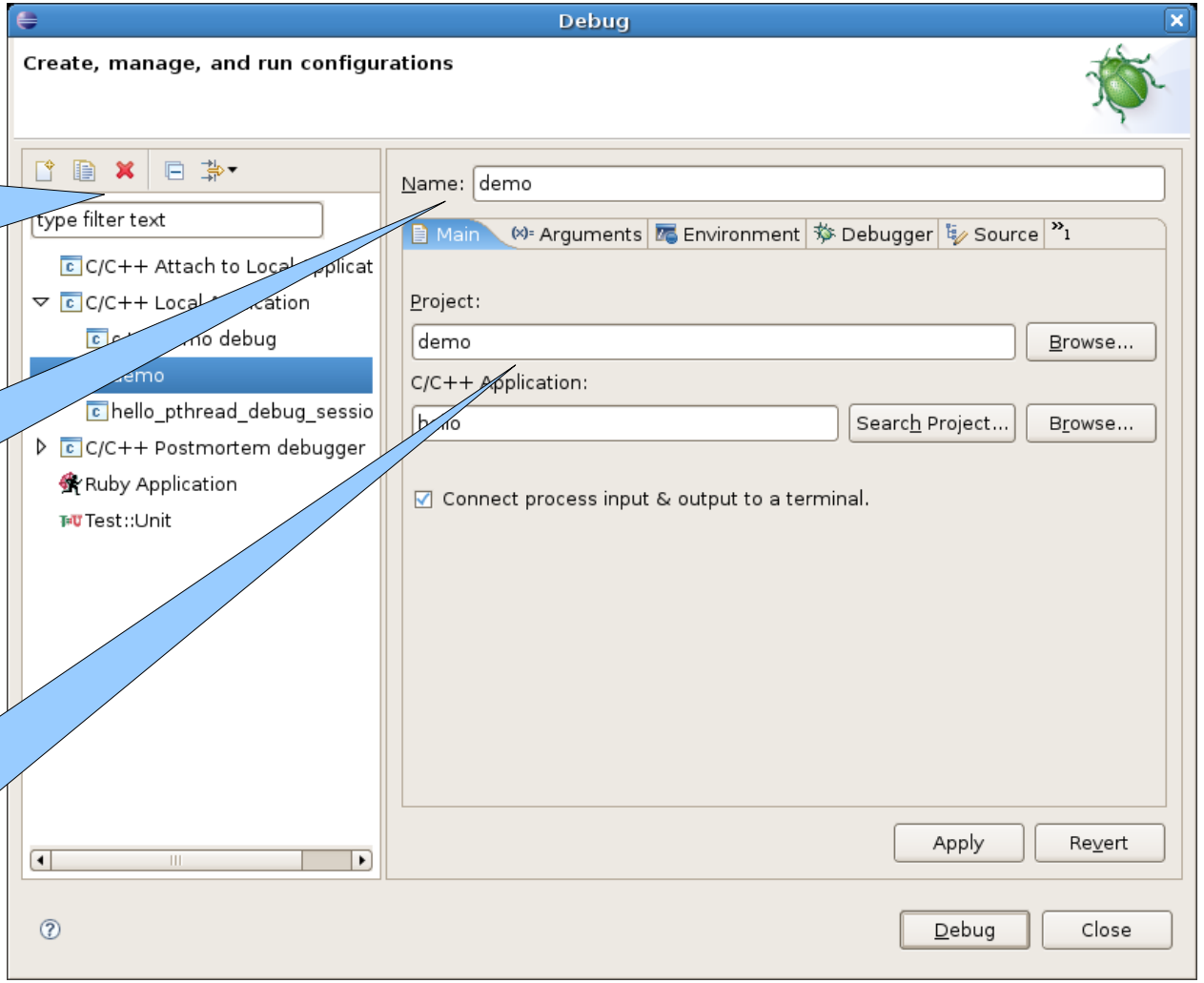


Creating a Debug Profile

2.Create a new debug profile

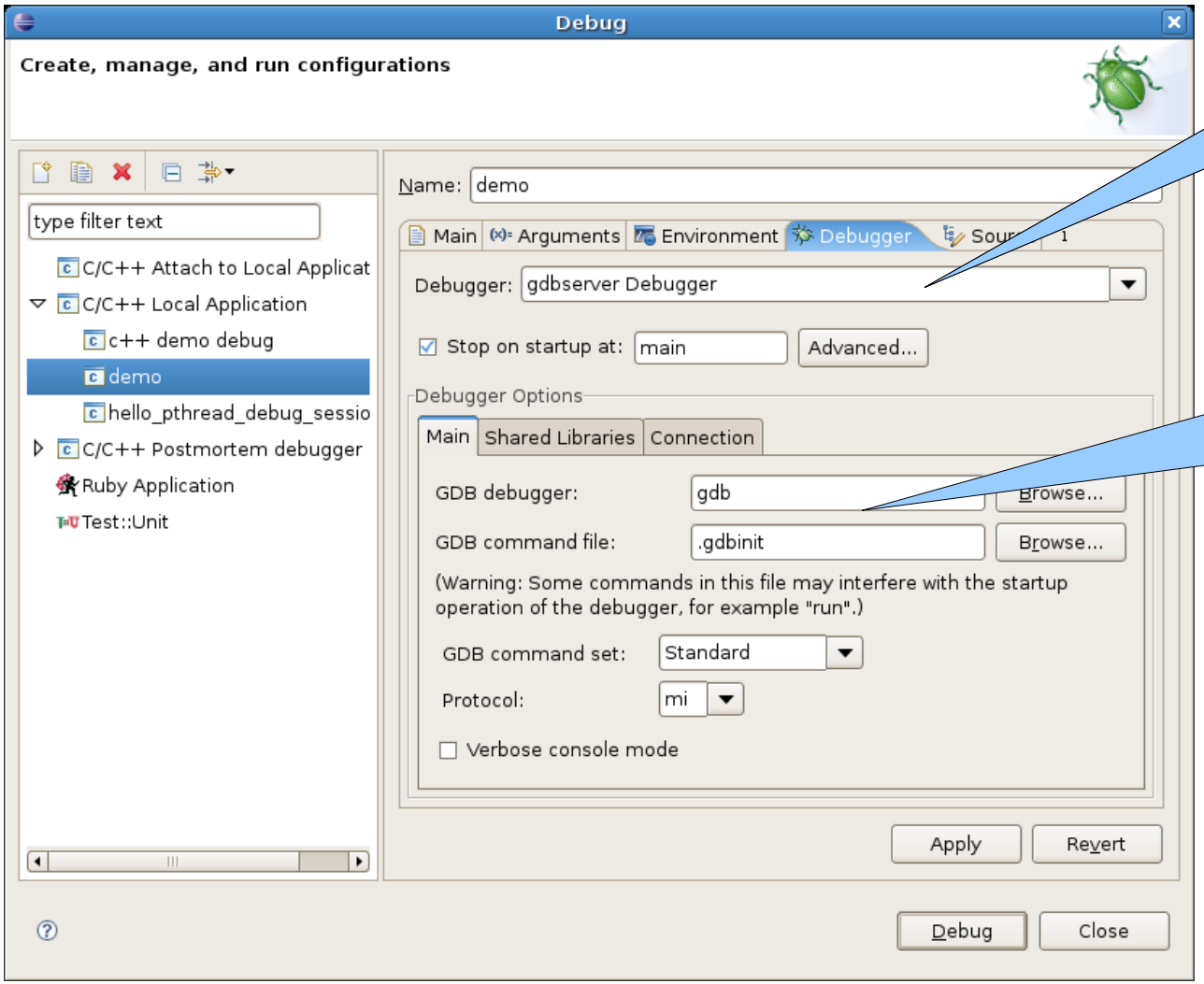
3.Give it a name

4. Choose project and binary



Codefidence
a name you can trust™

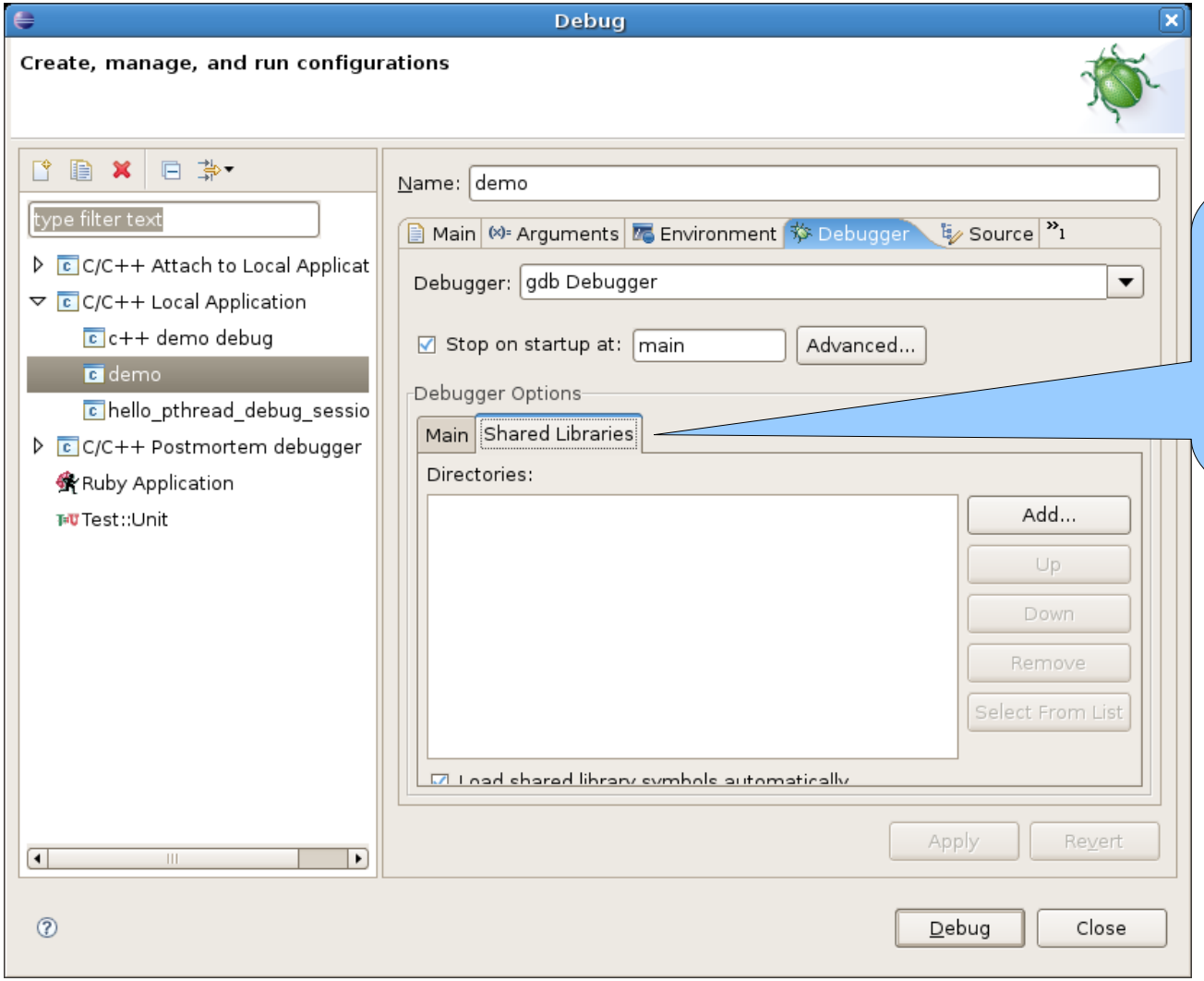
Configuring The Debugger



5. Choose type (local/remote)

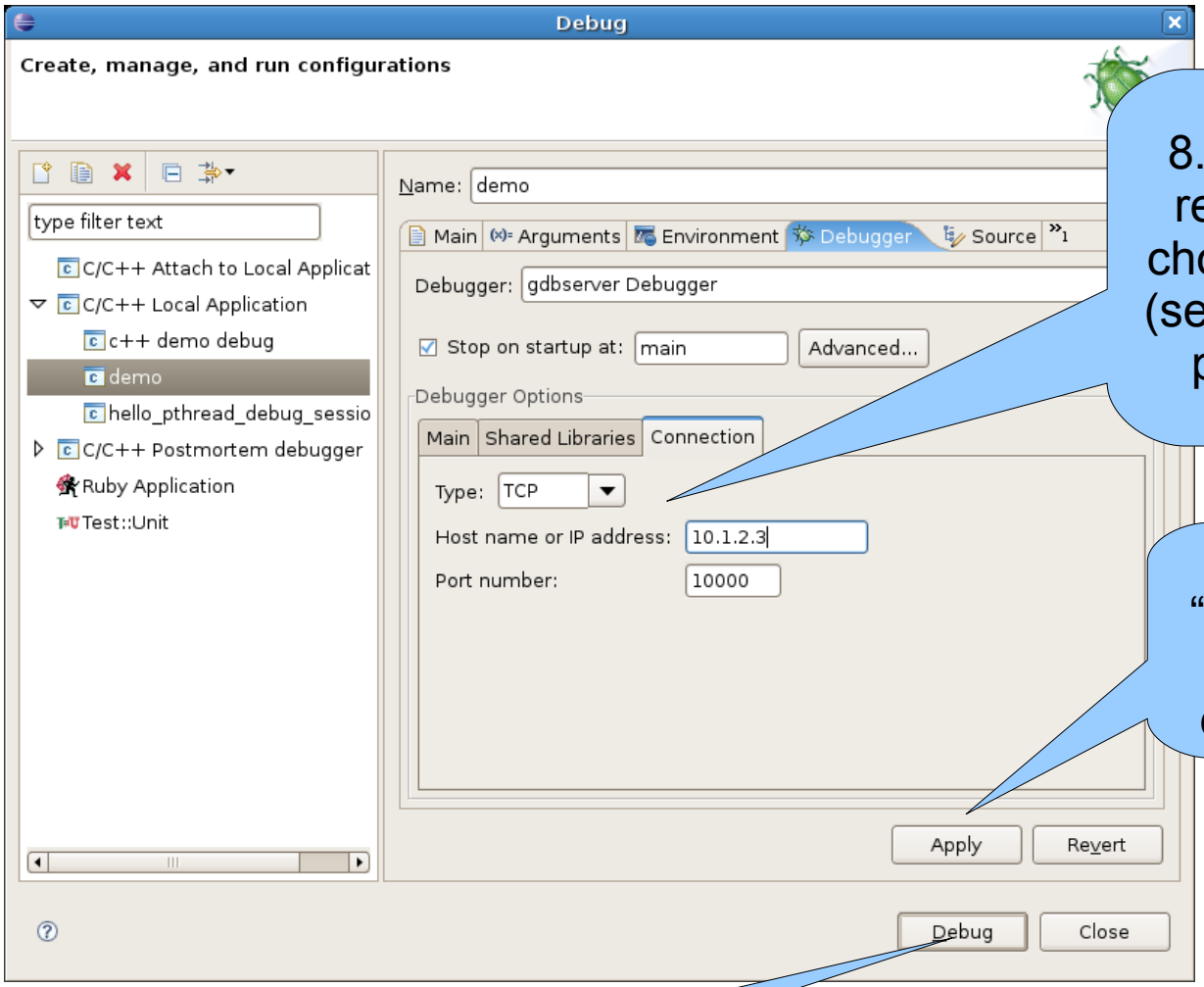
6. Add path to GDB and init file

Configure Dynamic Libraries



7. If debugging a remote machine, configure path to shared libs.

Choosing Connection Type



8. If debugging a remote machine choose connection (serial or TCP) and provide details

9. Use the "Apply" button to save your debug profile

10. Click "Debug" to start debugging.

Eclipse Debug session

Step, Next, Continue buttons

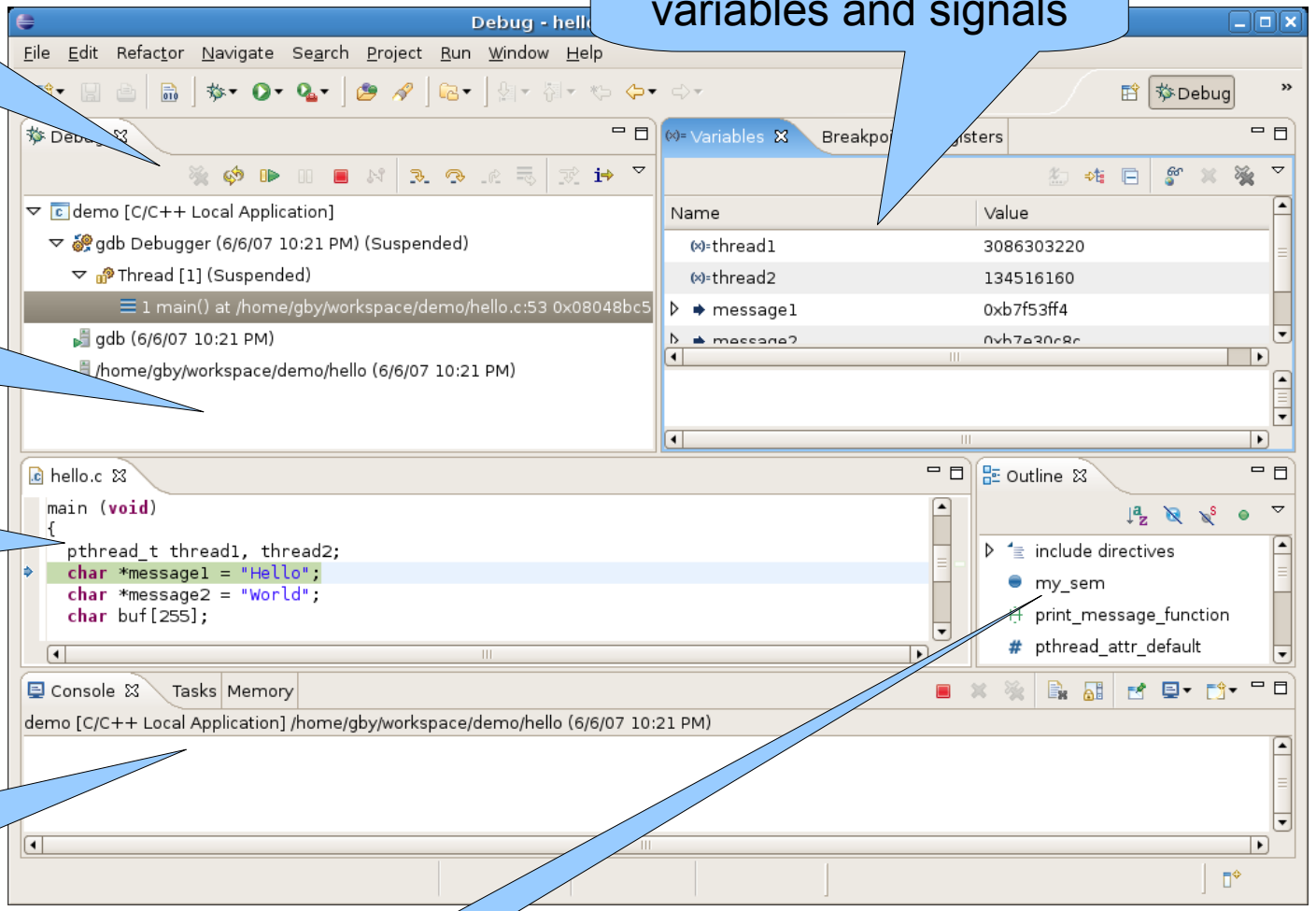
Threads and call stacks

Source code

Status, memory watches

Breakpoints, registers, variables and signals

Symbols



Debugging Linux Applications

Chapter 3

Smashing the Stack for Fun and Profit

The Call Stack

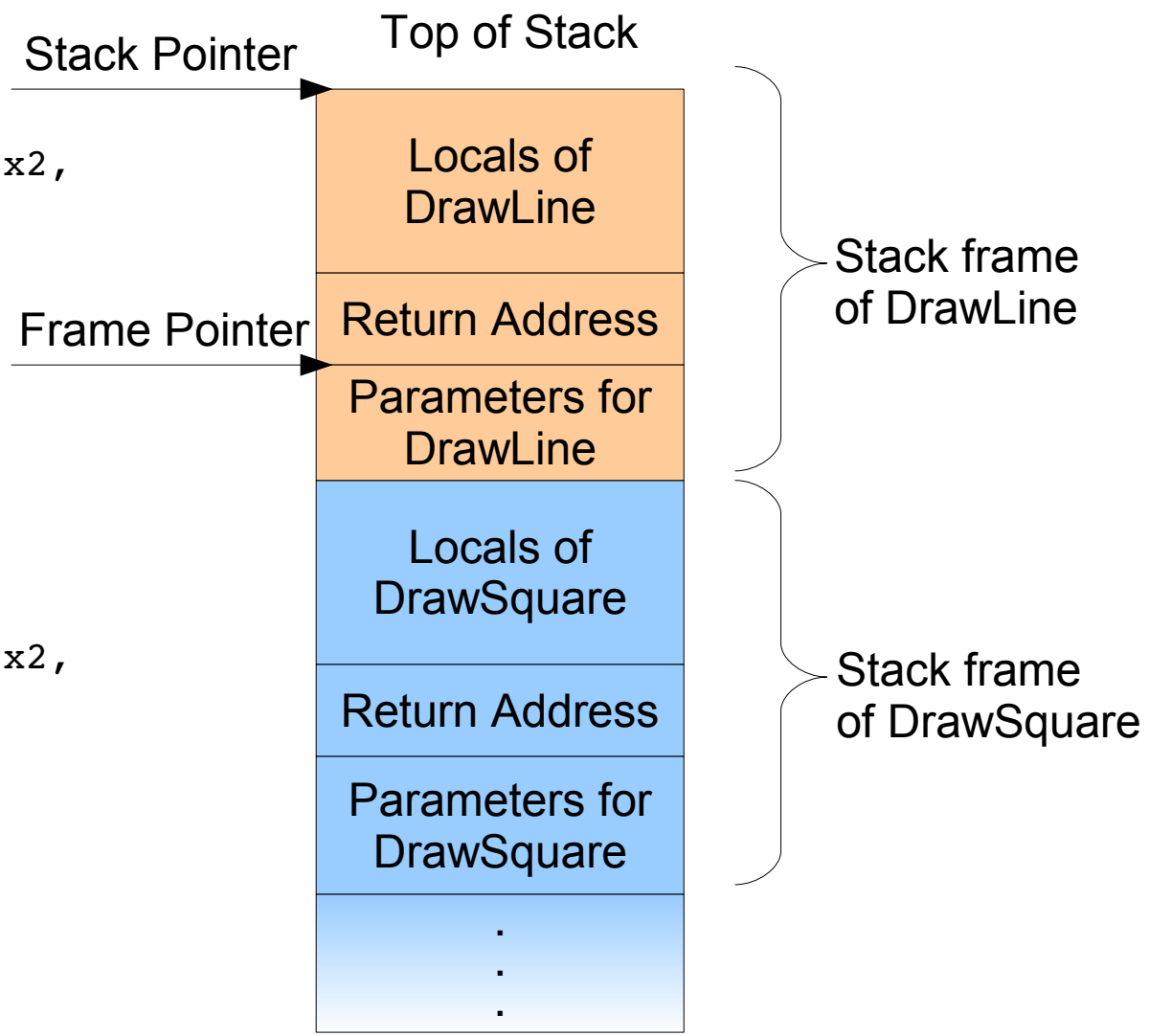
- The *call stack* stores information about the active subroutines of a computer program.
- The active subroutines are those which have been called but have not yet completed execution by returning.
- This kind of stack is also known as an *execution stack*, *control stack*, *function stack*, or *run-time stack*, and is often abbreviated to just "*the stack*".

Functions of the Call Stack

- **Storing the return address**
  - So we'll know to which function to return to.
- **Local data storage**
  - This is where the *tmp*, *i* and *myStr*'s are placed in memory.
- **Parameter passing**
  - This is how parameters are passed to functions.
- **Pointer to current instance**
  - C++ *this.*

Call Stack Structure

```
int
DrawLine
(int x1, int y1, int x2,
int y2)
{
     int tmp;
     char myarr[254];
     ...
     return 0;
}

int
DrawSquare
(int x1, int y1, int x2,
int y2)
{
     int i;
     ...
     return 0;
}
```



Use of the Call Stack

- **Call site processing**
  - Push parameters into the stack and call.
- **Callee processing**
  - Subroutine prologue gets parameters, saves room for locals.
- **Return processing**
  - Subroutine epilogue undo prologue, pop stack, return to caller
- **Unwinding**
  - Exception handling

Buffer Overflow

- A *buffer overflow* is an anomalous condition where a process attempts to store data beyond the boundaries of a fixed-length buffer.
- The result is that the extra data overwrites adjacent memory locations.
- The overwritten data may include other buffers, variables and program flow data and may cause a process to crash or produce incorrect results.

Buffer Overflow on the Stack

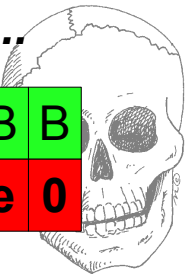
```
int main(void)
{
    int B = 3;
    char A[8];

    printf("%d\n", B);
    strcpy(A,"excessive");
    printf("%d\n", B);

    return 0;
}

bash$ ./program
3
228
bash$
```

Buffer before *strcpy...*

A	A	A	A	A	A	A	A	B	B
0	0	0	0	0	0	0	0	0	3

Buffer after *strcpy...*

A	A	A	A	A	A	A	A	B	B
'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	**e**	**0**



Writing to A changed the value of B because we overflowed the buffer of A.

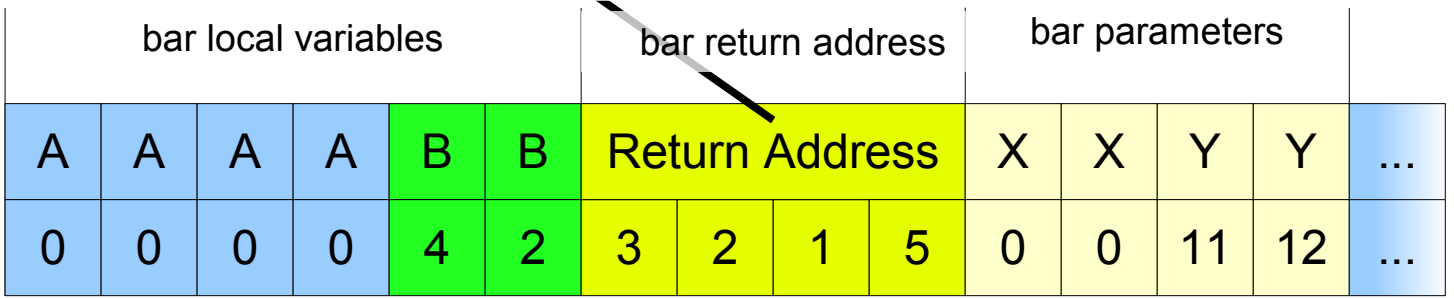
Codefidence
a name you can trust™

Buffer Overflow on the Stack (cont')

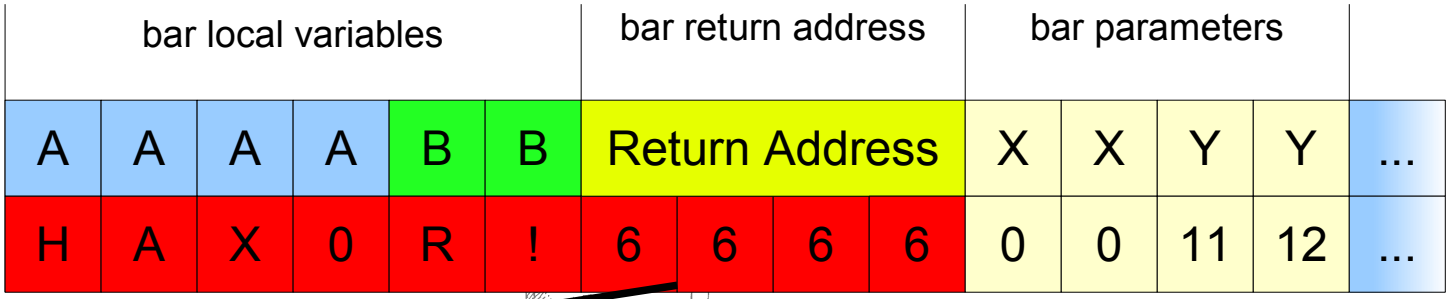
```
foo(Z,N)
{
    int A;
    int B;
    klunky(0,0)
    ...
}
```

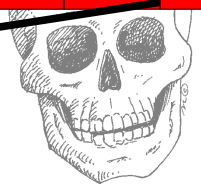
```
bar(X,Y)
{
    char A[4];
    int B;
    strcpy(A,"HAX0R!#@!%");
    return;
}
```

Normal Stack (return address points to foo)



Stack after overflow (return address points somewhere else)





Protecting from Stack Buffer Overflows

- **Choice of programming language**
  - Python, Ada, JAVA and C# all have internal bound checking as a language feature
- **Use of safe libraries and calls**
  - Use *strncpy* rather then *strcpy,* etc...
  - Use verified libraries, such as Vstr or Erwin
- **Stack-Smashing Protection (SSP)**
  - Compiler or library extension that adds automatic checks against stack overruns

Example of Stack Buffer Overflow

- The GoAhead web server is an open standards web server for embedded systems
- Version 2.1 shipped with a stack buffer overflow bug
- The bug could be used to force the web server to run arbitrary commands
- Let's see if you can spot it

Stack Overflow Example

```
int websValidateUrl(webs_t wp, char_t *path)
{
        /* Array of ptr's to URL parts */
        char_t  *parts[64];
        char_t  *token, *dir, *lpath;
        int             i, len, npart;
...

/* Copy the string so we don't destroy the original
*/

        path = bstrdup(B_L, path);
        websDecodeUrl(path, path, gstrlen(path));

        len = npart = 0;
        parts[0] = NULL;
        token = gstrtok(path, T("/"));

/*    Look at each directory segment and process
 *    "." and ".." segments. Don't allow the browser
 *    to pop outside the root web. */

        while (token != NULL) {
                if (gstrcmp(token, T("..")) == 0) {
                        if (npart > 0) {
                                npart--;
                        }

           } else if (gstrcmp(token, T(".")) != 0) {
                        parts[npart] = token;
                        len += gstrlen(token) + 1;
                        npart++;
                }
                token = gstrtok(NULL, T("/"));
        }
...
```

```
/*    Look at each directory segment and process
 *    "." and ".." segments
 *    Don't allow the browser to pop outside the
 *    root web.
 */
        while (token != NULL) {
                if (gstrcmp(token, T("..")) == 0) {
                        if (npart > 0) {
                                npart--;
                        }

           } else if (gstrcmp(token, T(".")) != 0) {
                        parts[npart] = token;
                        len += gstrlen(token) + 1;
                        npart++;
                }
                token = gstrtok(NULL, T("/"));
        }

/*   Create local path for document. Need extra
 *   space all "/" and null.
 */

if (npart || (gstrcmp(path, T("/")) == 0)
          || (path[0] == '\0')) {
...

} else {
                bfree(B_L, path);
                return -1;
        }
        return 0;

}
```

Example Explanation

```
/*********************************************
 *
 *      Validate the URL path and process ".."
 *      path segments. Return -1 if the URL
 *      is bad.
 */

int websValidateUrl(webs_t wp, char_t *path)
{
        /* Array of ptr's to URL parts */
        char_t  *parts[64];

        char_t  *token, *dir, *lpath;
        int             i, len, npart;


        a_assert(websValid(wp));
        a_assert(path);

        dir = websGetRequestDir(wp);
        if (dir == NULL || *dir == '\0') {
                return -1;
        }

/*
 * Copy the string so we don't destroy the original
 */
        path = bstrdup(B_L, path);
        websDecodeUrl(path, path, gstrlen(path));
```

```
/*      Look at each directory segment and process
 *      "." and ".." segments
 *      Don't allow the browser to pop outside the
 *      root web.
 */
        while (token != NULL) {
                if (gstrcmp(token, T("..")) == 0) {
                        if (npart > 0) {
                                npart--;
                        }

        } else if (gstrcmp(token, T(".")) != 0) {
                        parts[npart] = token;
                        len += gstrlen(token) + 1;
                        npart++;
                }
                token = gstrtok(NULL, T("/"));
        }

/*      Create local path for document. Need extra
 *      space all "/" and null.
 */

if (npart || (gstrcmp(path, T("/")) == 0)
          || (path[0] == '\0')) {
...

} else {
                        bfree(B_L, path);
                        return -1;
                }
               return 0;

}
```

There are only 64 places in the parts array, but the loop never checks for overflow!

Codefidence
a name you can trust™

Use After Free

- Use after free is a common programming error in which a resource is used after it no longer belongs to the user
- Use after free bugs can be sometimes very difficult to spot, because the location of the program crash or error is often where the resource has been allocated legally by another user of the resources, which is suddenly corrupted

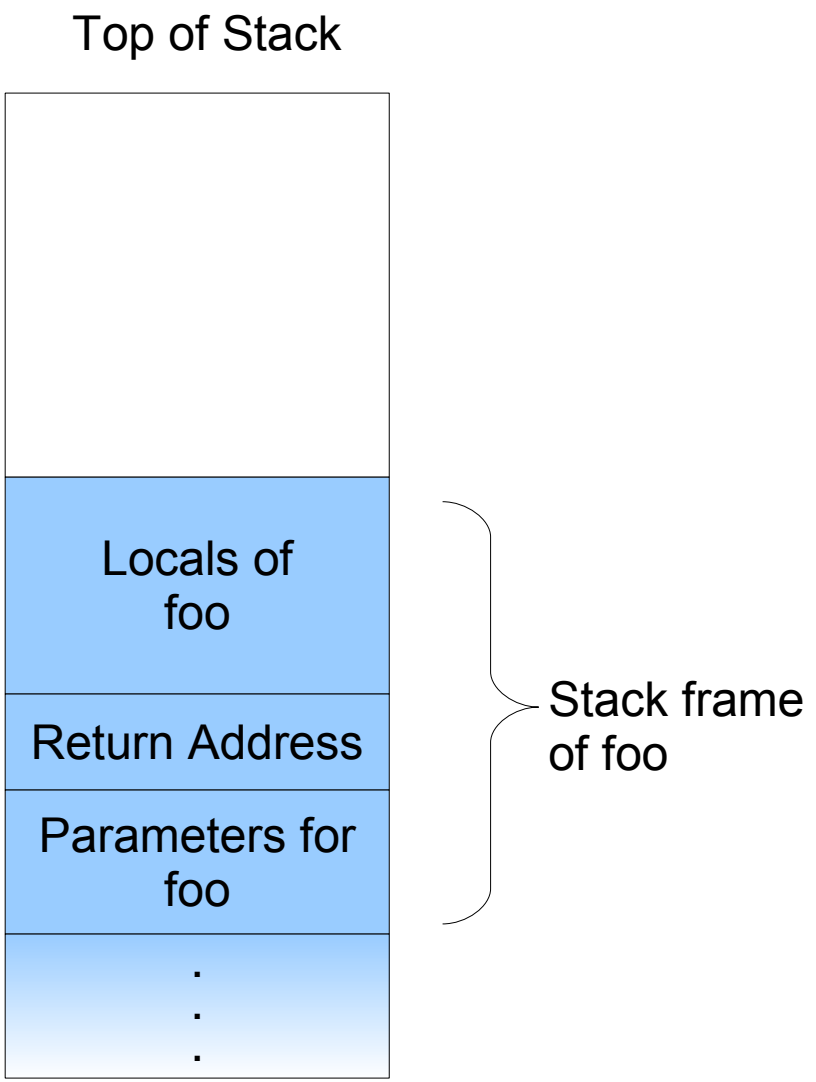
Volatile Stack References

- Local variables are stored on the stack
- After returning from the function, the stack is unwound
- Returning a reference (a pointer to) information on the stack is a common bug
- However, because of the stack behavior, the data on the stack remains valid until another function is called

Volatile Stack Reference

```
int * bar(int y)
{
    int i=5;
    ...
    return &i;
}

int baz(int z)
{
    char arr[5];

    memset(arr,5,z);
    ...
    return 0;
}

int * foo(int x)
{
    int * p;
    p = bar(x+5);
    baz(z);

    return p;
}
```

Foo() is called.

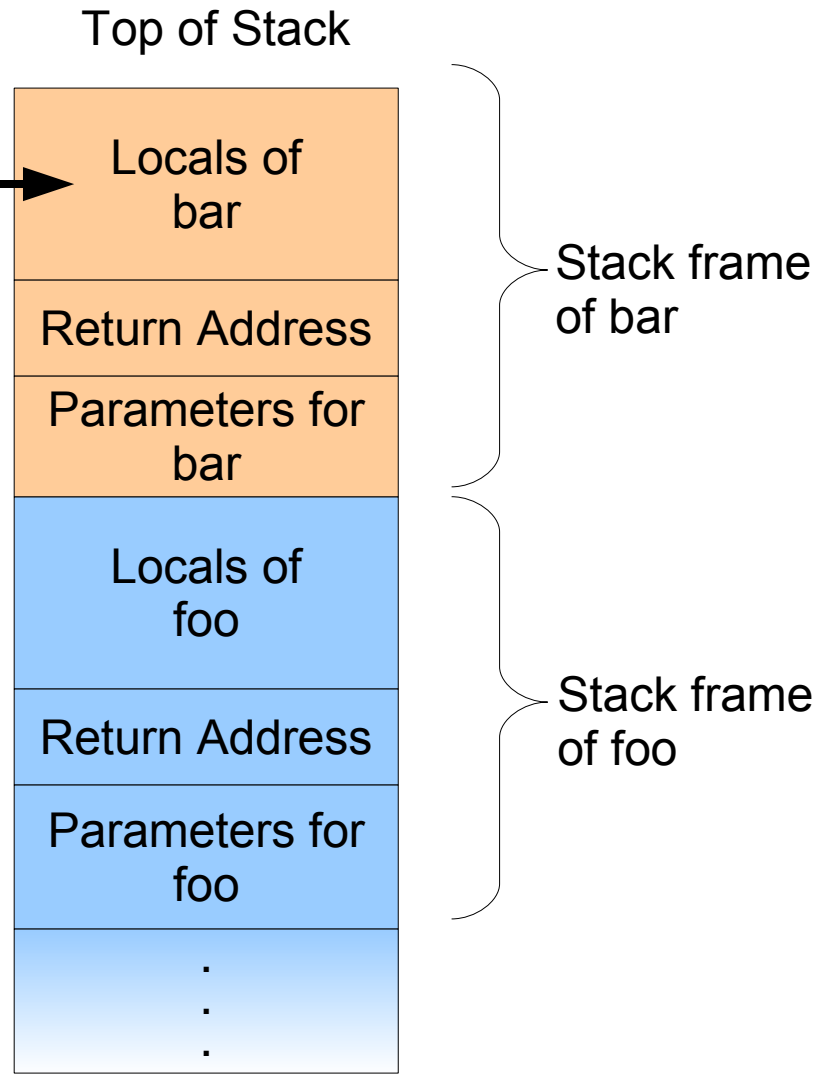


Volatile Stack Reference

```
int * bar(int y)
{
    int i=5;
    ...
    return &i;
}
```

Foo() calls bar().

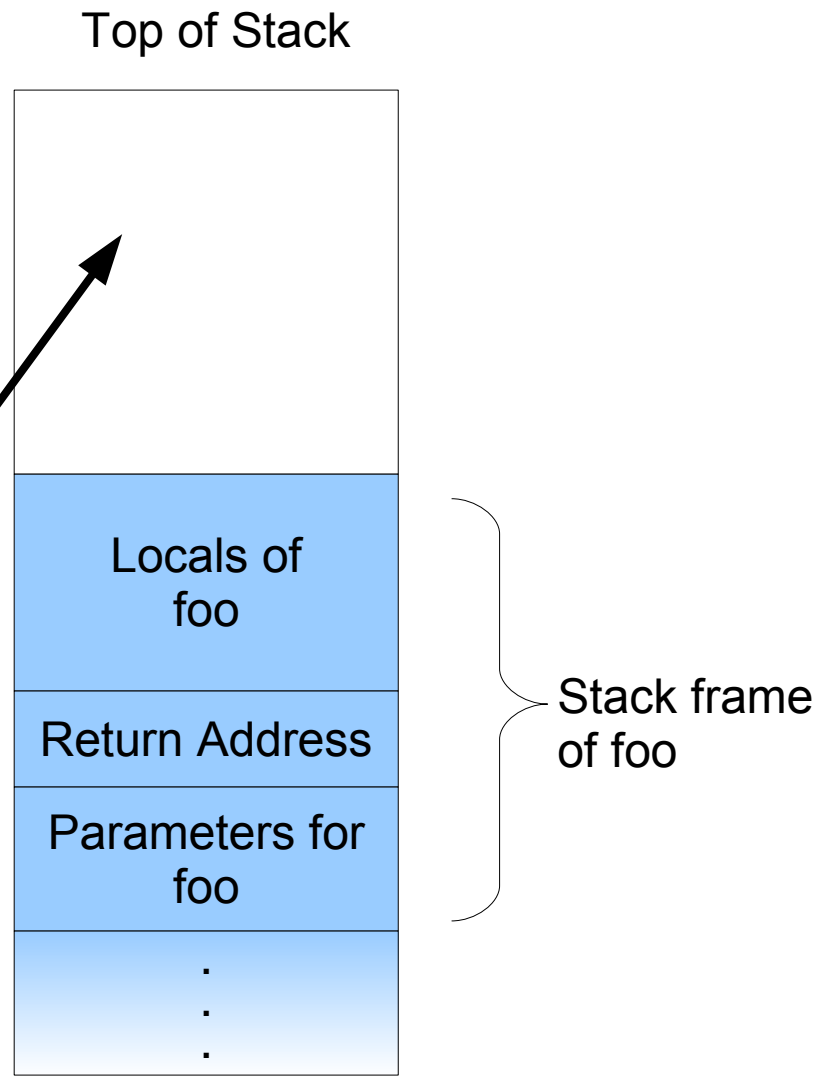
```
int baz(int z)
{
    char arr[5];

    memset(arr,5,z);
    ...
    return 0;
}

int * foo(int x)
{
    int * p;
    p = bar(x+5);
    baz(z);

    return p;
}
```



Volatile Stack Reference

```
int * bar(int y)
{
    int i=5;
    ...
    return &i;
}

int baz(int z)
{
    char arr[5];

    memset(arr,5,z);
    ...
    return 0;
}

int * foo(int x)
{
    int * p;
    p = bar(x+5);
    baz(z);

    return p;
}
```

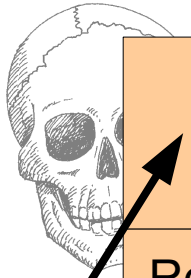
bar() returns control to foo().

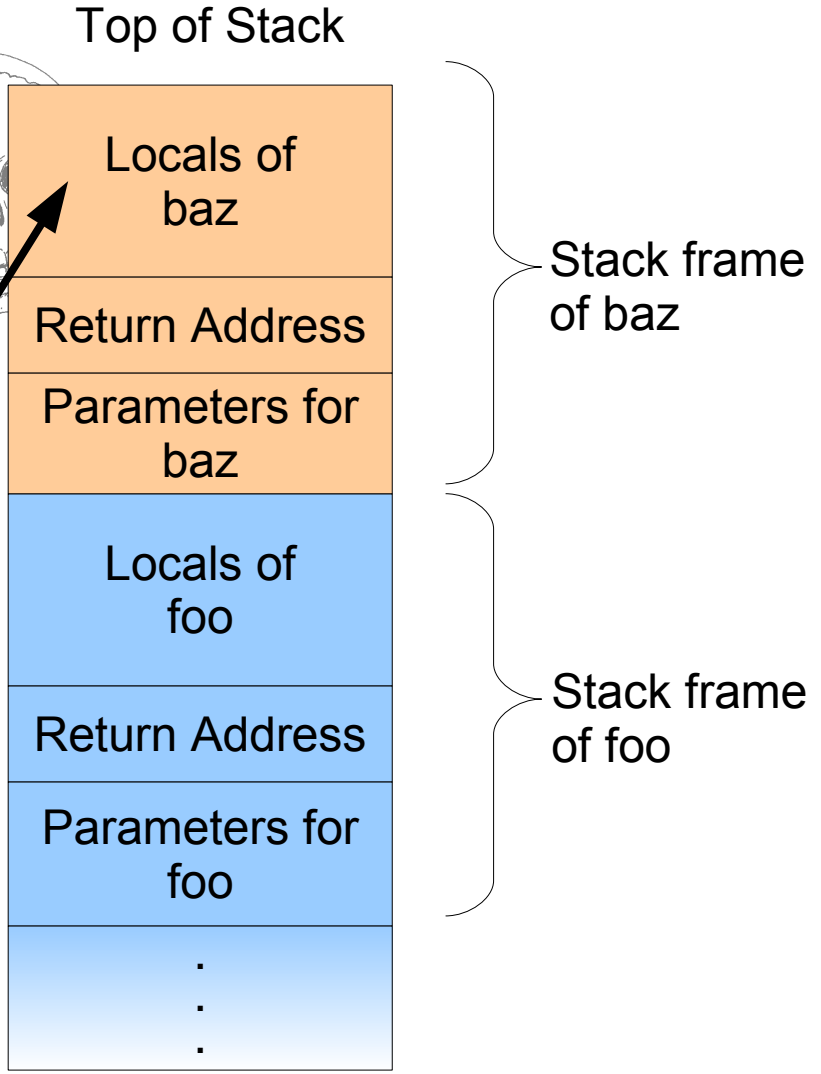


Volatile Stack Reference

```
int * bar(int y)
{
    int i=5;
    ...
    return &i;
}

int baz(int z)
{
    char arr[5];

    memset(arr,5,z);
    ...
    return 0;
}

int * foo(int x)
{
    int * p;
    p = bar(x+5);
    baz(z);

    return p;
}
```

foo() calls baz().





Stack Overflow

- In some operating systems, the size of the stack that is allocated to a process (or thread) is fixed
- On these systems, a program can overflow is stack – write more to the stack then is allocated to it
- In Linux, the stack is allocated dynamically per use, so this bug cannot occur

Debugging Linux Applications

Chapter 4

The Heap and Dynamic Allocations

Dynamic Allocations

- Dynamic memory allocation is the allocation of memory storage for use in a program during runtime
- The amount of memory allocated is determined at the time of allocation and need not be known in advance.
- A dynamic allocation exists until it is explicitly released, either by the programmer or by a garbage collector

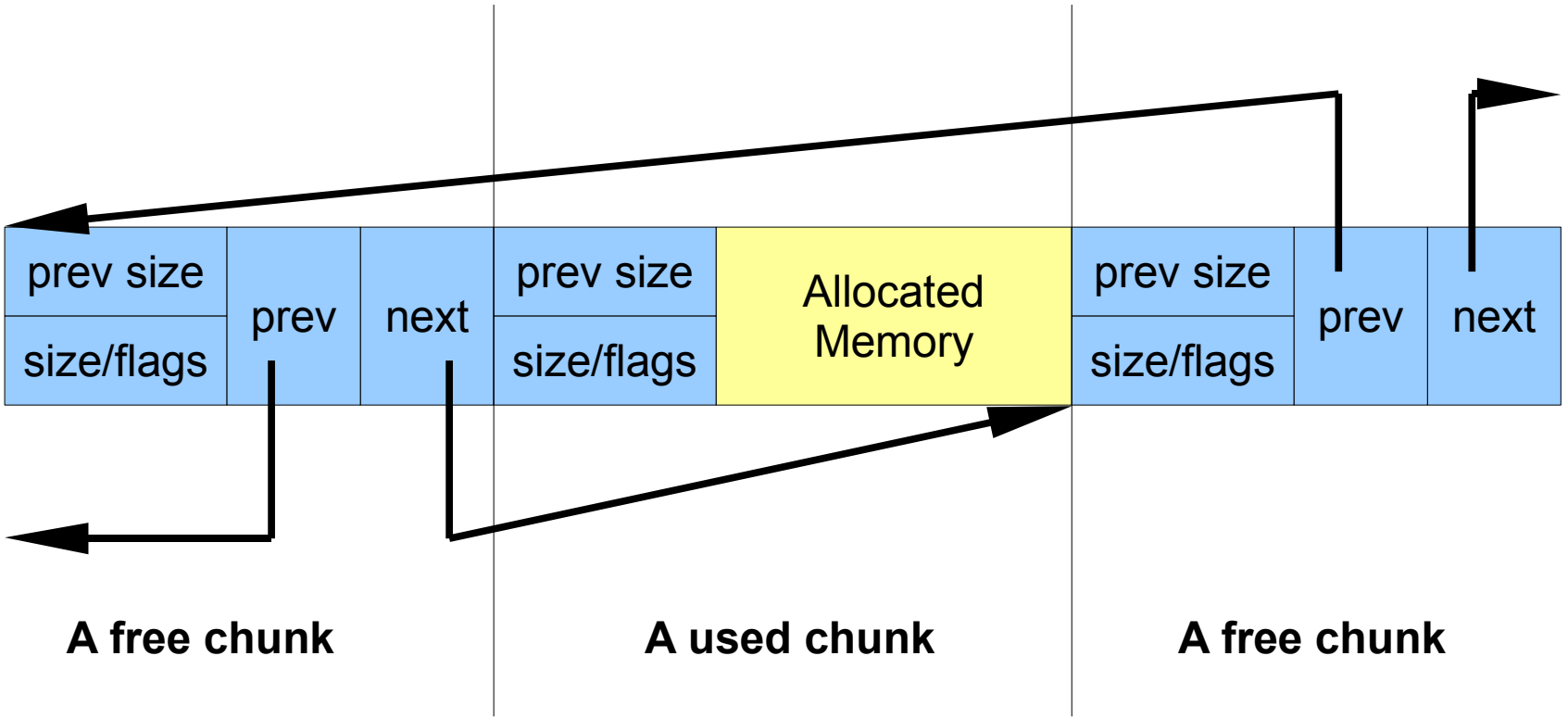
The Heap

- Usually, memory is allocated from a large pool of unused memory area called the heap
- Since the location of the allocation is not known in advance, the memory is accessed indirectly, via pointers
- The allocator can expand and contract the heap to fulfill allocation requests
- The heap method suffers from a few inherent flaws, stemming from fragmentation

The glibc Allocator

- The GNU C library (glibc) uses the ptmalloc allocator which uses both `brk(2)` and `mmap(2)`
- The `brk(2)` system call changes the size of the heap to be larger or smaller as needed
- The `mmap(2)` system call is used when extremely large segments are allocated
- The data structure used to keep track of allocation is called the “malloc arena”

The Malloc Arena



Prev size: sizeof previous chunk

Flags: PREV_INUSE (previous chunk is in use)
IS_MMAPPED (chunk is mmapped)

Allocation Failure

- malloc is not guaranteed to succeed — if there is no memory available, or if the program has exceeded the amount of memory it is allowed to reference, malloc will return a NULL pointer
- Many programs do not check for malloc failure. Such a program would attempt to use the NULL pointer returned by malloc as if it pointed to allocated memory, and the program would crash

Memory Leaks

- When a call to malloc, calloc or realloc succeeds, the return value of the call should eventually be passed to the free function
- If this is not done, the allocated memory will not be released until the process exits — in other words, a memory leak will occur
- In long running programs, memory leaks will result in allocation failure

Memory Leak Example

```
void *ptr;
size_t size = BUFSIZ;

ptr = malloc(size);

/* some further execution happens here... */

/* now the buffer size needs to be doubled */
if (size > SIZE_MAX / 2) {
  /* handle overflow error */
  return (1);
}
size *= 2;

ptr = realloc(ptr, size);

if (ptr == NULL) {
  return (1);
}

...
```

Example Explained

```
void *ptr;
size_t size = BUFSIZ;

ptr = malloc(size);

/* some further execution happens here... */

/* now the buffer size needs to be doubled */
if (size > SIZE_MAX / 2) {
  /* handle overflow error */
  return (1);
}
size *= 2;

ptr = realloc(ptr, size);

if (ptr == NULL) {
  return (1);
}

...
```

If realloc returned NULL because the re-allocation failed, realloc will not free the old buffer and we just lost track of it because the return value was written to the variable ptr which held the pointer to it!

Allocation Debugging

- The malloc implementation in the GNU C library provides simple means to detect leaks and obtain information to find the location
- To do this the application must be started in a special mode which is enabled by an environment variable
- There are no speed penalties for the program if the debugging mode is not enabled

Tracing malloc

- **`void mtrace(void)`**
  - This function looks for an environment variable MALLOC_TRACE. This variable is supposed to contain a valid file name
  - All uses of malloc(), realloc() and free() are traced and logged into the file
- **`void muntrace(void)`**
  - This function deinstalls the handlers and then closes the log file

Use After Free

- After a buffer has been freed, using it is forbidden. The pointer to the buffer may still be used by mistake:

```
int *ptr = malloc(sizeof (int));
free(ptr);
...
*ptr = 0;
```

- The system reuses freed memory, so writing to a deallocated region of memory results in overwriting another piece of data somewhere else in the program

Double Free

- With dynamic allocation, a particularly bad example of the use after free problem is if the same pointer is passed to free twice, known as a double free
- This returns the same memory buffer to the allocator
- It either causes the allocator to give the same buffer twice later, or crash the allocator (we'll say why later)

Freeing Unallocated Memory

- Another problem is when free is passed an address that wasn't allocated by malloc
- This can be caused when a pointer to a literal string or the name of a declared array is passed to free, for example:

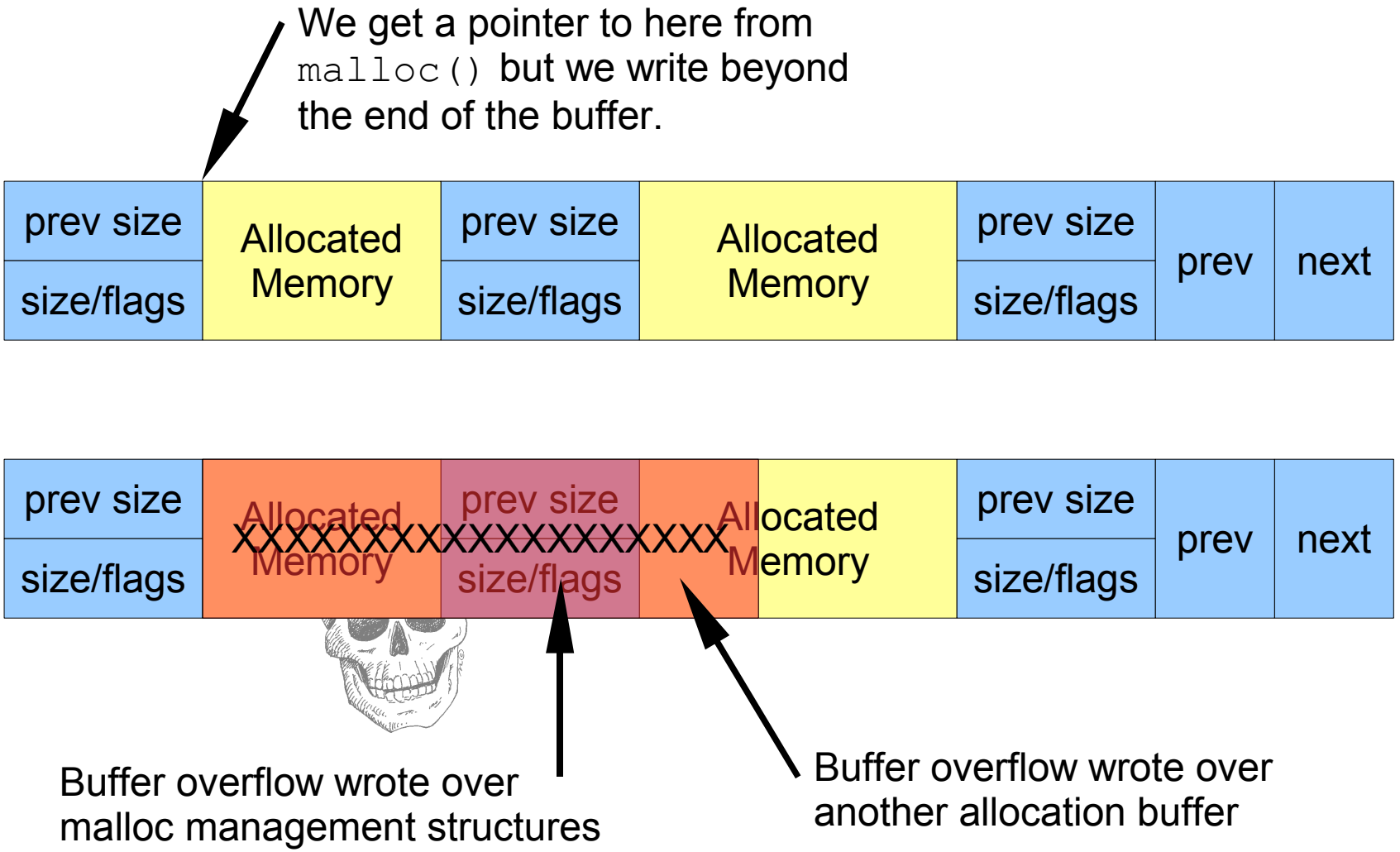
```
char *msg = "Default message";

int tbl[100];
```

- passing either of the above pointers to free will result in undefined behavior

Heap Buffer Overflow

- Just like with a stack buffer, a common bug involves writing beyond the end of the allocated buffer
- Because of the way the malloc arena is built, this can manifest in two ways:
  - We overwrite a buffer that was allocated to some other part in the program, making random changes to its content
  - Corruption of the malloc arena. Typically showing itself as crashes in calls to the `malloc()` and `free()` functions

Heap Buffer overflow scheme



Heap Buffer Overflow Example

- Can you tell what is wrong here?

```
p = malloc(strnlen(mystr, MAX_LEN));
strncpy(p, mystr, MAX_LEN);
```

Example Explained

- Can you tell what is wrong here?

```
p = malloc(strnlen(mystr, MAX_LEN));
strncpy(p, mystr, MAX_LEN);
```

- In short:
  - Assuming size of `mystr` is smaller than `MAX_LEN`, `strnlen()` will return the size of `mystr`
  - However, strncpy will write `MAX_LEN` bytes, even if `mystr` is smaller
  - In doing so, we'll overrun the `p` buffer

Heap Consistency Checking

- glibc has two heap debugging functions:
  - `int` **`mcheck`**`(void (*abortfn) (enum mcheck_status status))`
    - Calling `mcheck()` tells `malloc()` to perform occasional consistency checks
    - If a check fails it calls the callback function
    - These will catch things such as writing past the end of a block that was allocated with `malloc()`
  - `enum mcheck_status` **`mprobe`**`(void *pointer)`
    - The `mprobe()` function lets you explicitly check for inconsistencies in a particular allocated block
    - You need to call `mcheck()` first

Heap Consistency Checking

- Another possibility to check heap related bugs is to set the environment variable `MALLOC_CHECK_`
- It causes glibc to use a special implementation of malloc which is designed to catch simple errors
  - Errors such as double calls of free with the same argument, or overruns of a single byte (off-by-one bugs)

Hook For Malloc

- glibc lets you modify the behavior of malloc, realloc, and free by specifying appropriate hook functions
- You can use these hooks to help you debug programs that use dynamic memory allocation by writing your own call back functions to track allocations
- The hook variables are declared in `malloc.h`
- The `mcheck()` function works by installing such hooks

Debugging Linux Applications

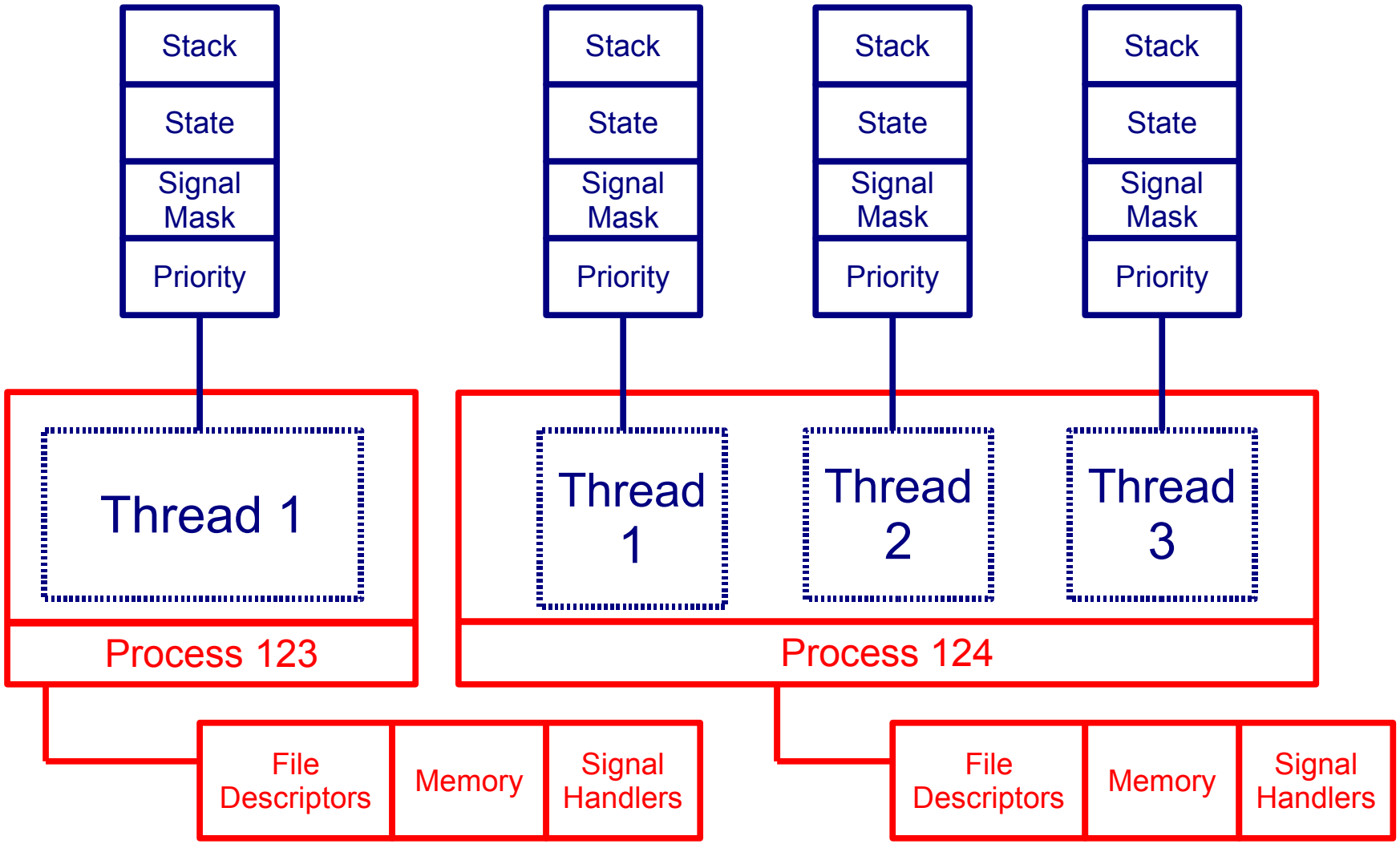
Chapter 5

Multi-Threading and Concurrency

Threads

- Threads in Linux are co-tasks running in the same process
- Threads share an address space, the heap, process ID and signal handlers
- Each thread has their own stack, signal mask and priority (nice level)
- Threads typically run concurrently, at the same time (in multi processor systems), or appear to do so (in single CPU systems)

Threads and Processes



Concurrency

- Threads are one of the major sources (but not the only one!) of concurrency
- Concurrency is the state in which several task are executing at the same time, and potentially interacting with each other
- Concurrency requires synchronization of several threads
- When synchronizations fails, we get bugs

Race Condition

- The main property of concurrency bugs is that they are time dependent:
  - A block of code may function correctly or not, depending on the exact timing it is called by one or more threads
- Because of the reference to time, these bugs care some time referred to as “Race Conditions”
- They are very tricky to debug

Race Condition Explained

- Consider this piece of code:

```
static int num = 0;
void AddOne(void) {
   num++;
}
```

- The `AddOne()` function is compiled into:
  - Read the value of `num` from memory into temporary register
  - Increment the temporary register
  - Write back the new value of `num` into memory

Race Condition Explained (cont')

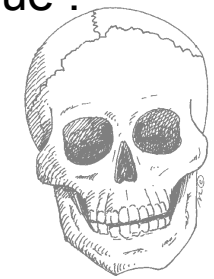
- If only a single copy of `AddOne()` runs at a time (and no one else touches `num`) everything is fine
- This will (usually) be the case in a single threaded application
- What can go wrong in a multi-threaded application?
- There is a chance that two threads will call `AddOne()` at the same time

The Lucky Case

Thread A		Thread B
	num=1	
Read num to register 1.		
Increment register 1.		
Write back new value .		
	num=2	
		Read num to register 2.
		Increment register 2.
		Write back new value .
	num=3	

The not so Lucky Case

Thread A		Thread B
	num=1	
Read num to register 1.		
		Read num to register 2 .
Increment register 1.	num=2	
		Increment register 2.
Write back new value .		
		Write back new value .
	num=2	



The value of `num` depends on the time the two threads calls `AddOne()`!

Determining Thread Safety

- Good places to look for problems:
  - Accessing global variables or the heap
  - Allocating/reallocating/freeing resources that have global limits (files, sub-processes, etc.)
  - Indirect accesses through handles or pointers
  - Any visible side-effect (e.g., access to volatile variables in the C programming language)
- A subroutine is thread-safe, if it only uses stack variables, arguments passed in, and calls thread-safe subroutines

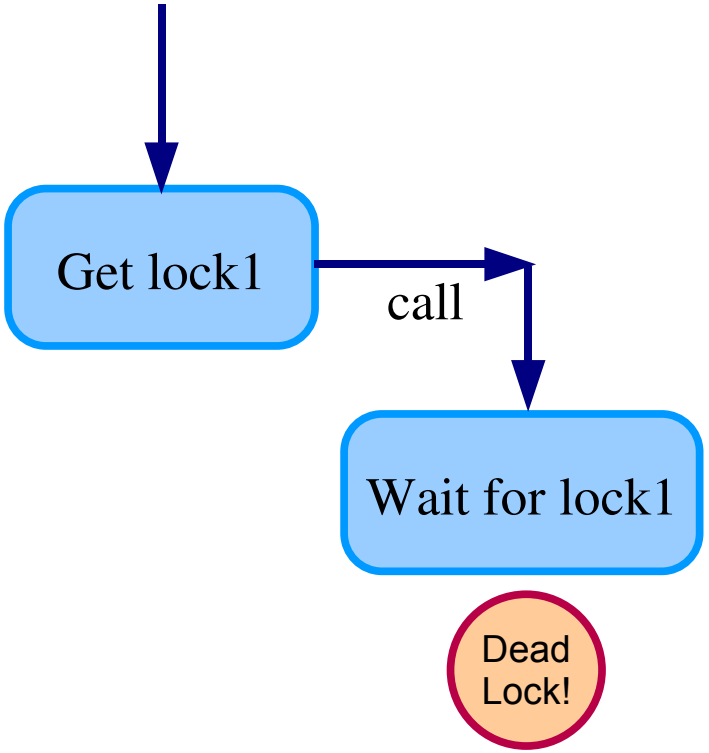
Achieving Thread Safety

- **Re-entrancy**
  - Write code that can be run in parallel
  - Don't use static or global variables
- **Mutual exclusion**
  - Access to shared data is serialized using locks
- **Thread-local storage**
  - Use variables local to each thread, instead of global ones

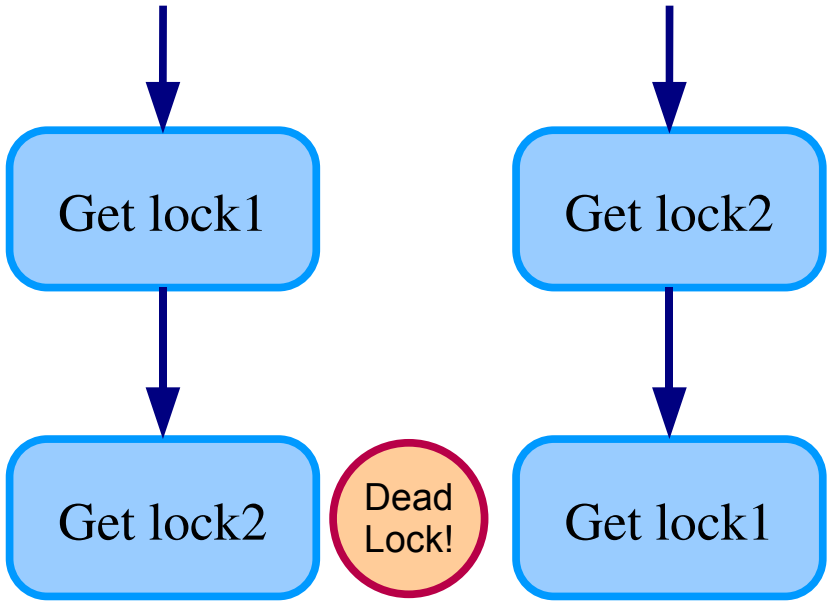
Lock Problems

Locks can also introduce bugs:

Don't call a function that can try to get access to the same lock



Holding multiple locks is risky!



POSIX Mutexes Debugging

- The POSIX mutexes used in Linux can be initialized with the attribute `PTHREAD_MUTEX_ERRORCHECK`
- This will cause the locking functions to be slower, but will catch many common errors:
  - `pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK);`
  - You will need to add the `-D_GNU_SOURCE` compiler flag to use this option

Compiler Optimizations

- The compiler is your friend... Usually. Sometime it can surprise you
- Compilers work on code serially and thus have no knowledge of concurrency
- This can lead to correct code being optimized into bad code by the compiler
  - Compiler might put frequently accessed variables into registers
  - Compiler might re-order code for performance reasons

Compiler Optimization Example

```
static int flag = 0;
```

Thread A

```
DoSomeStuff();

DoSomeMoreStuff();

flag = 1;
```

Thread B

```
/* Wait for thread A */

while (!flag) {

    sched_yield();

}

DoSomething();
```

If the compiler puts value of `flag` in thread A into a temporary register, this will not work and thread B will be stuck in the loop forever!
Or it might call `DoSomething()` before the while...

The C Volatile Keyword

- The C language gives us a way to tell the compiler not mess with our code – the `volatile` keyword
- When applied to variables it will cause the compiler to not make any optimization to access to the data
- It will solve the problem, but at considerable cost – code slow down
- It is the only option when accessing hardware memory mapped registers

Memory Barriers

- Most of the time, there is a much better solution: Using a memory barrier
- A memory barrier tells the compiler to make sure that all the stores that were coded before this point are seen before any of the stores coded after this point
- Careful use of barriers is much better than using `volatile`

Memory Barriers in GCC

- This is how to put a memory barrier in code in GCC:

```
asm volatile ("" : : : "memory");
```

- This tell GCC to:
  - Not move code around across the barrier
  - Write back to memory the values of all variables in registers before the barrier
  - Assume that the content of the variables may have changed after the barrier

Optimization Example Revisited

```
static int flag=0;
#define barrier() \
   asm volatile ("" : : : "memory");
```

Thread A

```
DoSomeStuff();

DoSomeMoreStuff();

flag = 1;
```

Thread B

```
/* Wait for thread A */

while (!flag) {

   barrier();

   sched_yield();

}

DoSomething();
```

Now the code is correct!

CPU Optimizations

- The compiler is not the only entity that can change order of code or memory operations – so can the CPU
- The issues caused by this are very similar to the issues caused by the compiler optimizations
- Hardware memory barriers are CPU specific assembly instructions
  - Different from CPU to CPU...

Bugs and Concurrency

- Many of the bugs presented thus far are made more complicated to find and fix when threads are involved
- The symptoms become non deterministic due to the element of timing
- Adding debugging code or running under debugger may change the symptoms

Other Sources of Concurrency

- Signal handlers are also sources of (pseudo) concurrency
- A signal may be caught at any time. The signal handler interrupts any context
- Unlike threads, locks do not provide a solution as the signal handler cannot block
- Any use of lock or accessing a shared resources in a signal handler is suspect
  - This includes global variables, pointers etc.

Debugging Linux Applications

Chapter 6

Programmed Debug Assistance

Programmed Debug Assistance

- Good logging practices
- Using the preprocessor:
  - Leveled logging macros
  - Assert macros
- Signals
- Writing a stack dump function

The System Log

- The system log file is typically /var/log/messages
- Applications may use the **syslog(3)** function in order to log messages to it, via **syslogd(8)**
- **syslogd(8)** supports remote logging
- **syslogd(8)** configuration file is **syslog.conf(5)**

The System Log (cont')

- You can use **`logger`** from within shell scripts or the command line to log:

```
$ logger -p err “this goes to the log”
```

System Log Example

```
...
Jun  4 16:59:40 slimshady kernel: NET: Registered
  protocol family 10
Jun  4 16:59:40 slimshady kernel: Disabled Privacy
  Extensions on device c02bf040(lo)
Jun  4 16:59:40 slimshady kernel: IPv6 over IPv4
  tunneling driver
Jun  4 16:59:42 slimshady dhcdbd: Started up.
Jun  4 16:59:46 slimshady hpiod: 1.7.3 accepting
  connections at 2208...
Jun  4 17:19:43 slimshady -- MARK --
Jun  4 17:39:43 slimshady -- MARK --
Jun  4 17:49:41 slimshady exiting on signal 15
...
```

Logging with syslog(3)

- Use **`syslog(3)`** to write messages to the system log:

```
#include <syslog.h>

void openlog(const char *ident, int option, int
  facility);

void syslog(int priority, const char *format, ...);

void closelog(void);
```

syslog(3) Tips

- Use the correct log level for every message. This will enable you later to filter only critical messages or debug messages
- Sometimes, per-module logging may be helpful. Add another prefix before every log message (besides the **`ident`** variable).
- syslog is implemented using a Unix domain socket: the most efficient IPC. You won't beat its performance, so don't try...

Leveled Logging

- It is recommended to use a system-wide configurable log level threshold:

```
int current_log_level = LOG_ERR;


#define LOG(lvl, msg...) do { \
    lvl <= current_log_level ? \
    syslog(lvl, msg) : 0; \
    } while (0)
```

assert()

- Use glibc's **`assert()`** macro in your debug builds
- Defined in `<assert.h>`
- In production build, disable assert by defining the preprocessor token `NDEBUG` (no debug)

Signals

- Signals are asynchronous notifications sent to a process by the kernel or another process
- Signals interrupt whatever the process was doing at the time to handle the signal
- The application may register a signal handler for each signal. The signal handler is a function that gets called when the process receives that signal

Signals (cont')

- Two default signal handlers also exist:
  - **SIG_IGN**: Causes the process to ignore the specified signal
  - **SIG_DFL**: Causes the system to set the default signal handler for the given signal
- There are two signals which cannot be caught, blocked or ignored:
  - **SIG_KILL**
  - **SIG_STOP**

Catching Signals

- You can register your own signal handler by calling **`signal(3)`**, from `<signal.h>`:

```
void (*signal(int sig, void (*func)(int)))(int);
```

- Confused? Well, I am. This is the equivalent of:

```
typedef void (*sighandler_t)(int);

sighandler_t signal(int sig, sighandler_t);
```

Catching Signals (cont')

- The new signal registration API uses **`sigaction(3)`**, from `<signal.h>`:

```
int sigaction(int sig, const struct sigaction *act,
  struct sigaction *oact);
```

- **`sigaction(3)`** is better than **`signal(3)`** since:
  - It is more portable (Sys V)
  - It is feature-rich
- However, it is a bit more complicated to use...

Printing a Stack Dump

- A backtrace is a list of function calls currently active in a thread
- glibc provides backtrace support for your program via:

```
#include <execinfo.h>
int backtrace(void **buffer, int size);
char **backtrace_symbols(void *buffer,
    int size);
```

Debugging Linux Applications

Chapter 7

Post-Mortem

Check the Log

1. Run `tail -f /var/log/messages` and then start the application which fails from a different shell. Maybe you already got some hints of what's going wrong in the log
2. If step 1 is not enough then edit `/etc/syslog.conf` and change `*.info` to `*.debug`. Run `/etc/init.d/syslog restart` and repeat step 1.

Core Dumps

- Some exceptions may result in a core dump – a record of the application state (memory) when it crashed
- Since core dump take up a lot of disk space, most distributions disable the feature
- To enable core dumps of up to 500 MB (1024 blocks of 512 bytes):

```
$ ulimit -c 1024
```

- Or better – unlimited:

```
$ ulimit -c unlimited
```

Core Dump Configuration

- `/proc/sys/kernel/core_pattern` (together with `/proc/sys/kernel/core_uses_pid`) describe the core file pattern
- Usually the default is **`core.<pid>`**
- So we will see a file such as:

`core.4455`

Core Dump Configuration (cont')

- For example:

```
$ echo "core.%e.%p" > /proc/sys/kernel/core_pattern
```

- Produces the following core dump files names:

core.<executable>.<pid>

Using the Core File

- First make sure this is your dump by using the **`file`** command:

```
$ file ./core.4455
./core.4455: ELF 32-bit LSB core file Intel 80386,
  version 1 (SYSV), SVR4-style, from 'dump'
```

- We then start GDB with the core file and the misbehaving application:

```
$ gdb ./dump core.4455
```

Using the Core File (cont')

- gdb will provide you with the `EIP` which caused the crash, and tell you which signal caused the program to be terminated:

```
$ gdb ./dump core

...

Core was generated by `./dump'.

Program terminated with signal 11, Segmentation
  fault.

#0  0x08048384 in croak () at dump.c:9

9             *ip = 7;

(gdb)
```

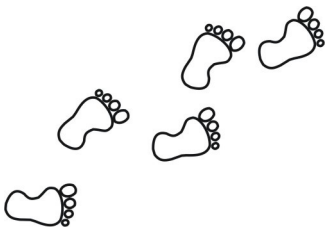
Debugging Linux Applications

Chapter 8

Debugging Tools

Debugging Tools

- strace(1)
- ltrace(1)
- POSIX Threads Trace Toolkit
- Dmalloc
- Valgrind



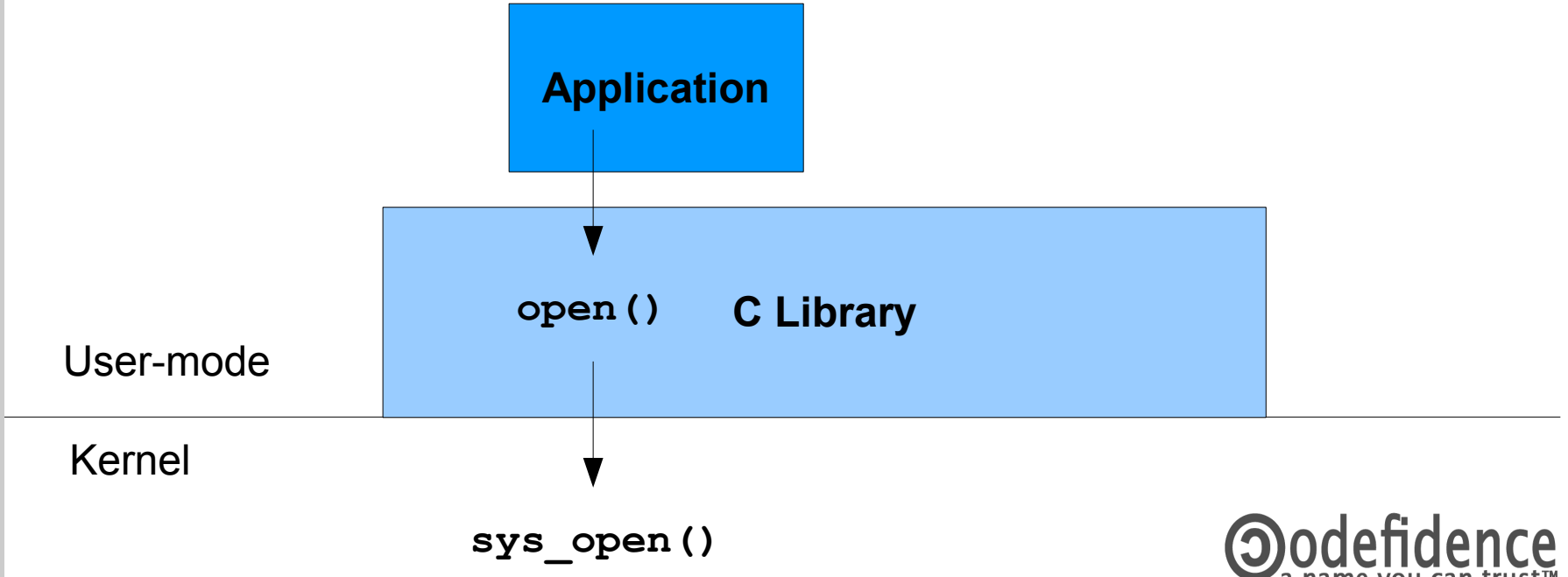
strace(1)

Use **`strace(1)`** to trace system calls and signals when:

- The source code is not available, or
- The application hangs or fails unexpectedly and you wish to quickly get more information

What Are System Calls

- The Unix OS provides the application services via the system call interface
- The C library implements the system calls, thus programmers need only call C library functions



Output Example

```
yariv@slimshady:~$ strace ls xxx.c

...

stat64("xxx.c", 0x80620fc)                    = -1 ENOENT (No
   such file or directory)

lstat64("xxx.c", 0x80620fc)                   = -1 ENOENT (No
   such file or directory)

write(2, "ls: ", 4ls: )                       = 4

write(2, "xxx.c", 5xxx.c)                     = 5

...

write(2, ": No such file or directory", 27: No such file or
   directory) = 27

write(2, "\n", 1)                             = 1

close(1)                                      = 0

exit_group(2)                                 = ?
```

Advanced strace(1)

- You can filter interesting system calls using the various **`-e`** flags:
  - `-e trace=open,close` (only open and close)
  - `-e trace=file` (all file related system calls)
- **`strace(1)`** can attach to a running process using the **`-p <pid>`** option
- You can also have **`strace(1)`** follow forking children with the **`-f`** option

ltrace(1)

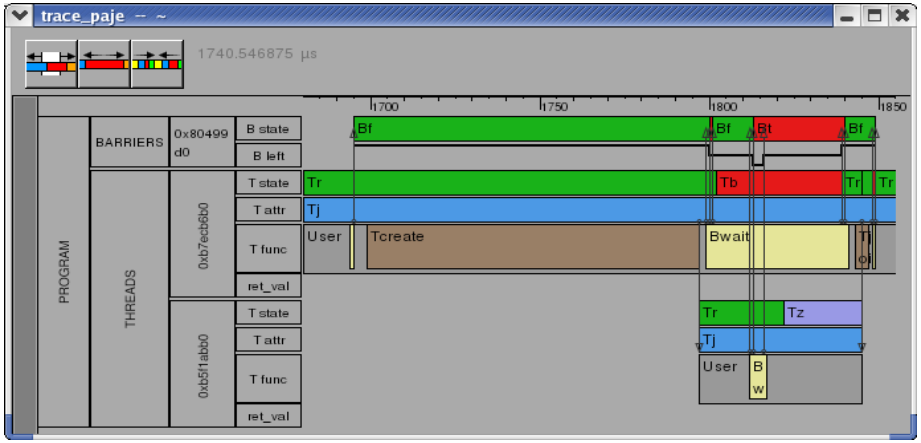
- Use **`ltrace(1)`** to trace dynamic library calls
- Very similar to **`strace(1)`**, in command-line options as well
- It cannot trace DSOs which were opened by `dlopen()`

Output Example

```
yariv@slimshady:~$ ltrace ls xxx.c

__libc_start_main(0x804e1c0, 2, 0xbffffa74, 0x8056930,
   0x8056920 <unfinished ...>

setlocale(6, "")
                              = "en_US.UTF-8"

bindtextdomain("coreutils", "/usr/share/locale")
                              = "/usr/share/locale"

textdomain("coreutils")
                              = "coreutils"

__cxa_atexit(0x8050090, 0, 0, 0xb7faaff4, 0xbffff9d8)
                              = 0

isatty(1)
                              = 1

...
```

POSIX Threads Trace Toolkit

- PTT (http://nptltracetool.sourceforge.net/)
- Building it requires patching gcc and rebuilding it
- Generates a trace describing the behavior of NPTL (Native POSIX Threads Library) threads
- Also provides a graphical trace (Paje-compatible):



PTT (cont')

```
#        PTT version 0.90.0.

#        The trace was generated on i686 on Mon, 22 May 2006 10:04:16 +0200.

#        This file is the number 0.

#

#

0000000.016463 Pid 19493, Thread 0x4014aa90 starts user function, version=900000

0000000.016581 Pid 19493, Thread 0x4014aa90 enters function pthread_barrier_init,
   barrier=0x8049c78, attribute=(nil), count=2

0000000.016581 Pid 19493, Thread 0x4014aa90 initializes barrier 0x8049c78, left=2

...

0000000.016617 Pid 19493, Thread 0x4014aa90 is blocked on barrier 0x8049c78

0000001.018918 Pid 19493, Thread 0x4074dbb0 enters function pthread_barrier_wait,
   barrier=0x8049c78

0000001.018918 Pid 19493, Thread 0x4074dbb0 requires barrier 0x8049c78, lock=0

0000001.018918 Pid 19493, Thread 0x4074dbb0 is blocked

0000001.018918 Pid 19493, Thread 0x4074dbb0 is resumed
```

Dmalloc

- Debug Malloc Library (http://dmalloc.com/)
- Dmalloc is a dynamic allocations (heap) debugging library only
- Dmalloc is extremely portable and is known to work on AIX, *BSD, GNU/Hurd, HPUX, Irix, Linux, Mac OSX, NeXT, OSF/DUX, SCO, Solaris, Ultrix, Unixware, MS Windows, and Unicos.
- Dmalloc replaces `malloc()`, `free()` and friends with its own macros when you `#include <dmalloc.h>`

Dmalloc (cont')

- Dmalloc supports multi-threaded applications
- Dmalloc has some C++ support (not as good as its C support)
- Dmalloc produces a log explaining the problems found.
- By default, Dmalloc immediately core dumps when it finds any heap memory problems

Dmalloc Log Example

```
1181060733: 10: Dmalloc version '5.5.2' from 'http://dmalloc.com/'

1181060733: 10: flags = 0x4f48d03, logfile 'dmalloc.log'

1181060733: 10: interval = 10, addr = 0, seen # = 0, limit = 0

1181060733: 10: starting time = 1181060733

1181060733: 10: process pid = 20563

1181060733: 10:   error details: checking user pointer

1181060733: 10:   pointer '0xb7f66f78' from 'unknown' prev access
   'bad_alloc.c:20'

1181060733: 10:   dump of proper fence-top bytes: 'i\336\312\372'

1181060733: 10:   dump of '0xb7f66f78'-8:
   '\033\253\300\300\033\253\300\300\000\000\000\000\t\000\000\000
   '

1181060733: 10:   next pointer '0xb7f66f80' (size 4) may have run
   under from 'bad_alloc.c:20'

1181060733: 10: ERROR: _dmalloc_chunk_heap_check: failed OVER
   picket-fence magic-number check (err 27)
```

Valgrind

- Valgrind (http://valgrind.org/)
- Suite of tools for debugging and profiling Linux applications
- The most useful of the tools is called Memcheck (also the default tool). We will focus on it
- Runs on x86, AMD64, PPC 32/64

Valgrind Tools

- Memcheck – detects memory management problems
- Cachegrind – cache profiler for I1, D1 and L2 CPU caches
- Helgrind – multi-threading data race detector
- Callgrind – heavyweight profiler
- Massif – heap profiler

How Valgrind Works

- The Valgrind core simulates every instruction the application executes
- It therefore increases code size considerably and makes the code run much slower
- For Memcheck:
  - Size increased by at least x12
  - Running speed is 25-50 times slower

Memcheck

- Valgrind's Memcheck can find:
  - Memory leaks
  - Illegal read/write operations
  - Uses of uninitialized values
  - Illegal frees
  - Bad system call parameters
  - Overlapping blocks in memcpy operations

Valgrind Output Example

```
$ valgrind ./bad_alloc 2

...

==9472== Use of uninitialised value of size 4

==9472==    at 0x406F69B: (within /lib/tls/i686/cmov/libc-2.5.so)

==9472==    by 0x407361B: vfprintf (in /lib/tls/i686/cmov/libc- 2.5.so)

==9472==    by 0x4079652: printf (in /lib/tls/i686/cmov/libc-2.5.so)

==9472==    by 0x804856C: uninit (bad_alloc.c:43)

==9472==    by 0x80485F4: main (bad_alloc.c:57)

...

==9472== LEAK SUMMARY:

==9472==    definitely lost: 26 bytes in 2 blocks.

==9472==      possibly lost: 0 bytes in 0 blocks.

==9472==    still reachable: 0 bytes in 0 blocks.

==9472==         suppressed: 0 bytes in 0 blocks.

==9472== Use --leak-check=full to see details of leaked memory.
```

Debugging Linux Applications

Last words of advice...

The Zen of Debugging

"A novice was trying to fix a broken Lisp machine by turning the power off and on.

Knight, seeing what the student was doing, spoke sternly: "You cannot fix a machine by just power-cycling it with no understanding of what is going wrong."

Knight turned the machine off and on.

The machine worked."

-- From the MIT AI Lab Koan collection, via the Jargon file.

Debugging Linux Applications

Appendix A

Libraries

Static Library Creation

- Use the archive utility – **ar(1)** to create a static library
- Makefile rule example:

```
libstat.a:
        gcc -g -c -Wall -o libstat_a.o lib_a.c
        gcc -g -c -Wall -o libstat_b.o lib_b.c
        ar rcs $@ libstat_a.o libstat_b.o
```

Dynamic Library Creation

- Use gcc to create relocatable objects and then link them together into a DSO (Dynamic Shared Object)
- Makefile rule example:

```
libdyn.so:
        gcc -fPIC -g -c -Wall -o libdyn_a.o lib_a.c
        gcc -fPIC -g -c -Wall -o libdyn_b.o lib_b.c
        gcc -shared -Wl,-soname,libdyn.so.1 \
          -o $@.1.0 libdyn_a.o libdyn_b.o -lc
        ln -s $@.1.0 $@.1
        ln -s $@.1.0 $@
```

Static and Dynamic Linkage

- The process of linking an application with a library is the same, regardless of whether it is a static library or a dynamic library
- Whether we're linking against `libmylib.a` or `libmylib.so`, the syntax remains the same:

```
$ gcc -o myapp main.c -L/path/to/lib -lmylib
```