Garbage collection for C

Jon Rafkind

Issue

- Memory must be managed manually in C
- Notoriously difficult to get right(memory leaks, double free, dangling references)

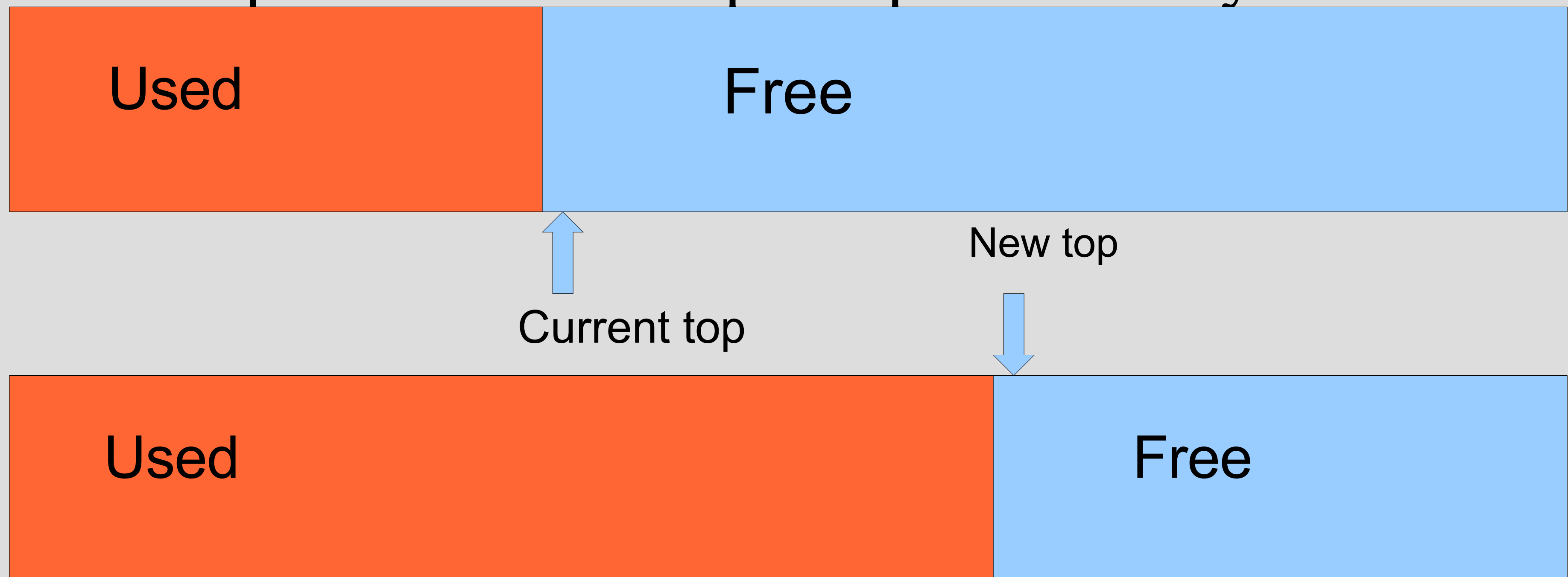
Solution: garbage collection

Program lifetime

- Modifies pointers
- Allocates objects from memory
- When nursery runs out of space, perform a GC

Allocation

- Allocations come from the nursery, unless the object is too big
- Nursery is an array of bytes, allocate from the current pointer and bump the pointer N bytes



Conservative collection

Scan heap for words that look like they might be pointers, mark those objects as live

0x23	0xffab432f	0x84356	0x0	0xdeadbeef	0x42

Pros: easy to use, sound

Cons: inaccurate

Precise collection

Traverse all data structures, only mark live pointers

0xfafaabe1	0xba54af02	0xcedd4422	0xbe400219	0xbb123456

0xba54af02 : struct foo *

Pros: accurate
Cons: incompatible with naïve libraries(future research)

Magpie

Precise, moving(generational) garbage collector using mark/sweep

Originally written by Adam Wick in C and Scheme
Scheme frontend rewritten in Ocaml using Cil

http://www.cs.utah.edu/~rafkind/projects/magpie/
http://hal.cs.berkeley.edu/cil

GC process

1. Mark/traverse roots and stack
2. For each object added, mark/traverse that object as well
3. Move objects in the nursery to older pages, make a new nursery

What is needed for precise collection?

Need to know:

- Root pointers
- Pointers on the stack
- How to traverse objects in the heap

Root pointers

Global variables – pointers and structures

Call from main():

```
void register_globals(){
    add_global( &some_global, gc_atomic );
    ....
}
```

Pointers on the stack

```
void foo( int * x ){
    int * y;
    ...
}
```

Store pointers in a shadow stack

```
extern void * GC_shadow_stack[];
GC_shadow_stack[0] = last_shadow_stack;
GC_shadow_stack[1] = &x;
GC_shadow_stack[2] = &y;
```

Traverse objects in the heap

For each allocation, compute type and store it somewhere

Types:

- gc_atomic – int, char, etc
- gc_array – arrays of pointers
- gc_struct_* - unique type for each struct/union

Traversal routines for structs

Need to traverse data in structs, generate a routine to do it

```
void gc_struct_traverse_foo( struct foo * f ){
    traverse( f->x );
    traverse( f->y );
    ...
}

gc_struct_foo = struct{ gc_struct_traverse_foo, ... };
```

Using it

Works fine for code that naively uses malloc, calloc, realloc (no program modifications necessary)

- Lame mp3 encoder
- Top
- xterm

Doesn't work as easily with programs that wrap malloc with complicated procedures (gcc, perl)

- safemalloc from perl
- obstack from gcc

Quick instructions

Step 1: Run the C source through the preprocessor. I like to put the resulting file in a different directory, usually called 'gc'.

```
$ gcc -E src/file.c -o gc/file.i
```

Step 2: Run magpie over the preprocessed file.

```
$ cilly.asm.exe --out gc/file.c --noPrintLn --dooneRet --doallocanalysis gc/file.i
```

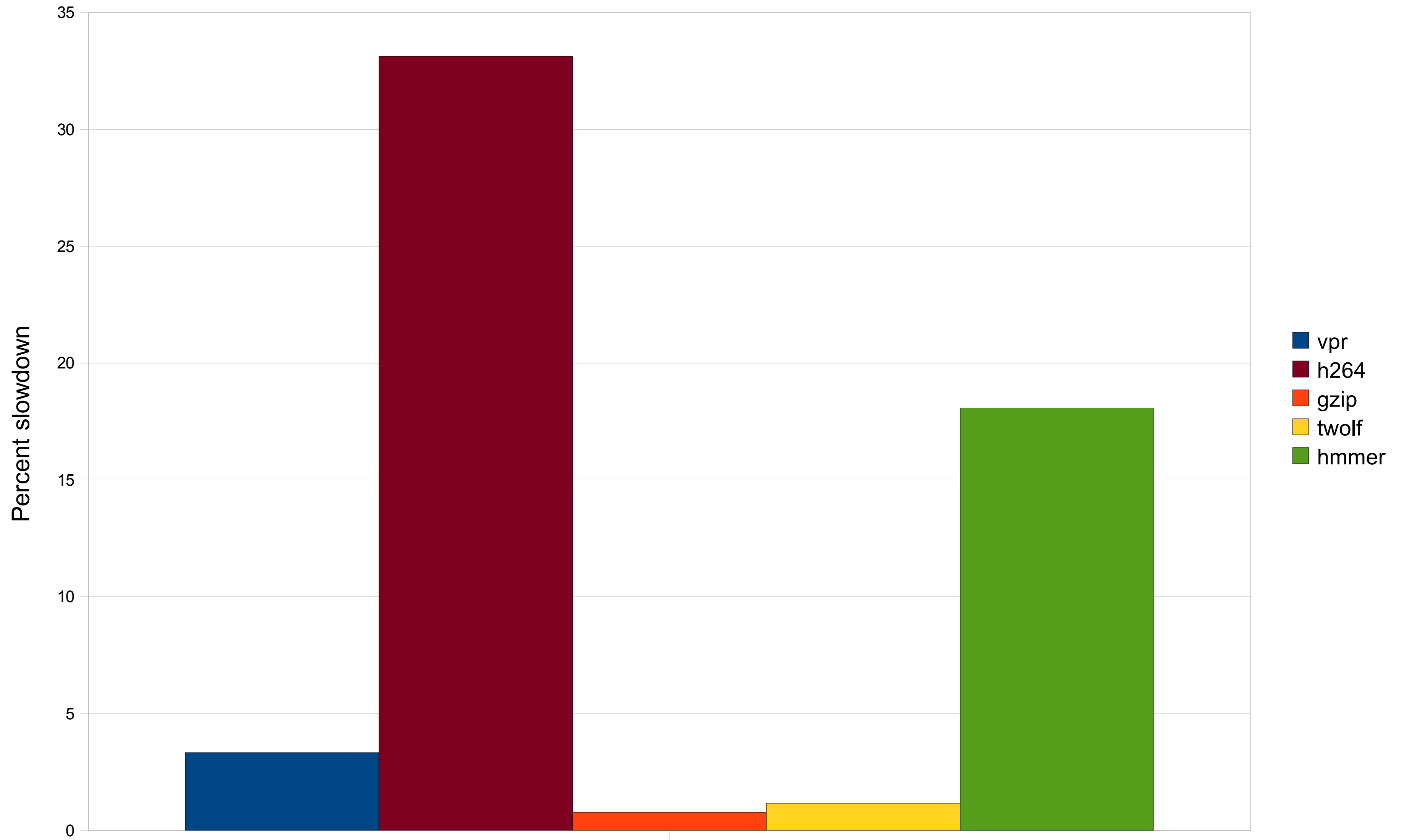
Step 3: Compile the file.

```
$ gcc -c gc/file.c -o gc/file.o
```

Step 4: Link the program together with the garbage collector. You will also have to compile gc_implementation.o, obviously.

```
$ gcc gc/file.o gc/gc_implementation.o -o program
```

Spec2000



Linux

Just another C program(mostly)

- Non-uniform allocators (kmem_cache_alloc)
- Reference counting
- Bit twiddles memory (red-black trees)
- Interrupts
- Rcu (Read-Copy update)
- Multiple stacks

Allocators

- Slab allocators – created by some modules to speed up allocation
- Kmalloc, kzalloc – basically like malloc
- __get_free_page – close enough to malloc
- Vmalloc – close but strange
- Ioremap – maps physical memory to virtual memory

Reference counting

Reference counting via kref objects

Cannot be removed easily because kref is intertwinned with sysfs

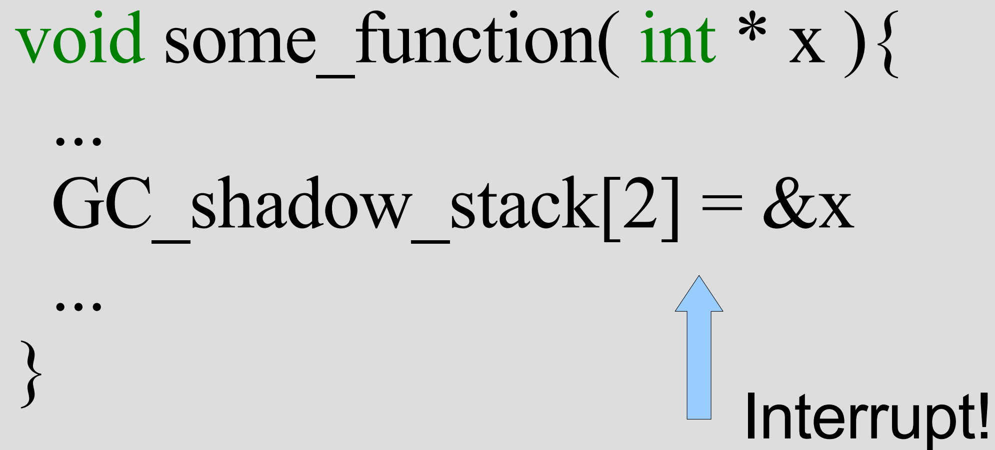
Bit twiddling

Red black tres operate on the lower 2 bits

```
static inline void rb_set_parent(struct rb_node *rb, struct rb_node *p){
    rb->rb_parent_color = (rb->rb_parent_color & 3) | (unsigned long)p;
}

static inline void rb_set_color(struct rb_node *rb, int color){
     rb->rb_parent_color = (rb->rb_parent_color & ~1) | color;
}
```

Interrupts



- Disable during GC
- Hurts performance

Rcu

Read-copy update stores values in a per-cpu variable

Last cpu to access the variable cleans it up, has own cleanup facilities that are not orthogonal to garbage collection

So far

- Stand alone modules work
- Attempted to GC ext3 but..

Filesystems are at the heart of Linux. GC'ing the filesystem requires the entire system to be GC'd as well.

Summary

How to incrementally GC linux?
Debugging is hard!

The end

Questions?