15-213

Introduction to Computer Systems

Memory Management III: Perils and pitfalls Oct 15, 1998

Topics

- Review of C pointer references
- Memory-related bugs
- Debugging versions of malloc
- Binary translation
- Garbage collection

C operators

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * & (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= != <<= >>=	right to left
,	left to right

Note: Unary +, -, and * have higher precedence than binary forms

C pointer declarations

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

Memory-related bugs

Dereferencing bad pointers

Reading uninitialized memory

Overwriting memory

Referencing nonexistent variables

Freeing blocks multiple times

Referencing freed blocks

Failing to free blocks

Dereferencing bad pointers

The classic scanf bug

```
scanf("%d", val);
```


Reading uninitialized memory

Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```


Overwriting memory

Allocating the wrong sized object

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```


Overwriting memory

Off-by-one

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

Overwriting memory

Off-by-one redux

```
int i, done=0;
int s[4];

while (!done) {
   if (i > 3)
     done = 1;
   else
     s[++i] = 10;
}
```


Overwriting memory

Forgetting that strings end with '/0'

```
char t[7];
char s[8] = "1234567";

strcpy(t, s);
```


Overwriting memory

Not checking the max string size

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```


Overwriting memory

Referencing a pointer instead of the object it points to

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```

Overwriting memory

Misunderstanding pointer arithmetic

```
search(int *array, int val) {

   while (*p && *p != val)
      p += sizeof(int);
}
```


Referencing nonexistent variables

Forgetting that local variables disappear when a function returns

```
int *foo () {
   int val;
   return &val;
}
```


Freeing blocks multiple times

Nasty!

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);

y = malloc(M*sizeof(int));
<manipulate y>
free(x);
```


Referencing freed blocks

Evil!

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);
...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

Failing to free blocks (memory leaks)

slow, long-term killer!

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```


Failing to free blocks (memory leaks)

Freeing only part of a data structure

```
typedef LISTP {
   int val;
   LISTP *val;
}

foo() {
   LISTP *listhead = malloc(sizeof(LISTP));
   LISTP *listitem = malloc(sizeof(LISTP));

   listhead->next = listitem;
   <create and manipulate the rest of the list>
   ...
   free(listhead);
   return;
}
```


Dealing with memory bugs

Conventional debugger (gdb)

- **good for finding bad pointer dereferences**
- **hard to detect the other memory bugs**

Debugging malloc (CSRI UToronto malloc)

- **wrapper around conventional malloc**
- **detects memory bugs at malloc and free boundaries**
  - memory overwrites that corrupt heap structures
  - some instances of freeing blocks multiple times
  - memory leaks
- **Cannot detect all memory bugs**
  - overwrites into the middle of allocated blocks
  - freeing block twice that has been reallocated in the interim
  - referencing freed blocks

Dealing with memory bugs (cont)

Binary translator (Atom, Purify)

- **powerful debugging and analysis technique**
- **rewrites text section of executable object file**
- **can detect all errors as debugging malloc**
- **can also check each individual reference at runtime**
  - bad pointers
  - overwriting
  - referencing outside of allocated block

Garbage collection (Boehm-Weiser Conservative GC)

- **let the system free blocks instead of the programmer**

Debugging malloc

mymalloc.h:

```
#define malloc(size) mymalloc(size, __FILE__, __LINE__)
#define free(p) myfree(p, __FILE__, __LINE__)
```

Application program:

```
ifdef DEBUG
#include <mymalloc.h>
#endif

main() {
    ...
    p = malloc(128);
    ...
    free(p);
    ...
    q = malloc(32);
    ...
}
```

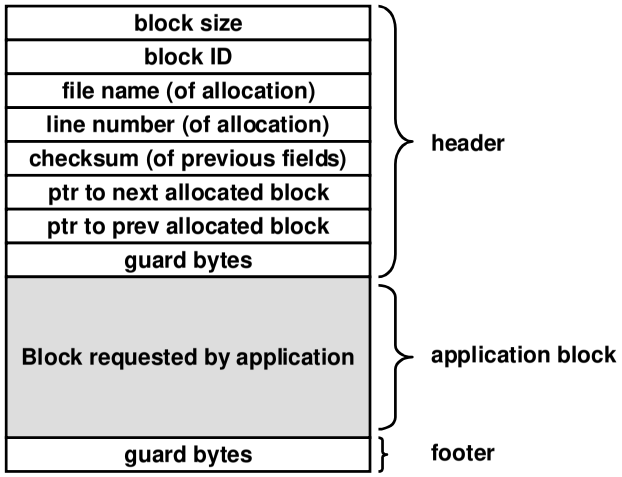

Debugging malloc (cont)

Debugging malloc library:

```
void *mymalloc(int size, char *file, int line) {
   <prologue code>
   p = malloc(...);
   <epilogue code>
   return q;
}

void myfree(void *p, char *file, int line) {
   <prologue code>
   free(p);
   <epilogue code>
}
```


Debugging malloc (cont)



Debugging malloc (cont)

mymalloc(size):

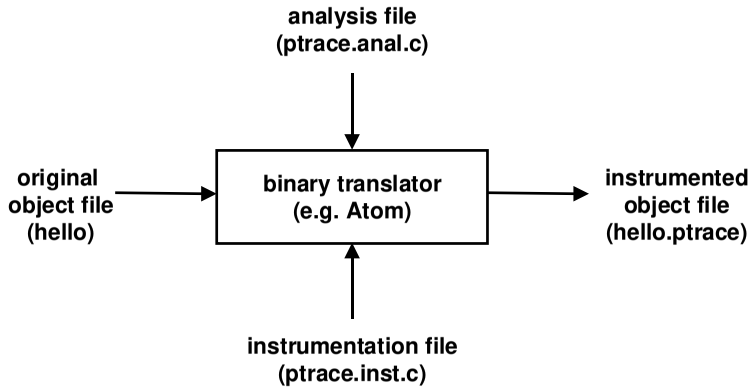
- p = malloc(size + sizeof(header) + sizeof(footer));
- add p to list of allocated blocks
- initialize application block to 0xdeadbeef
- return pointer to application block

myfree(p):

- already free (line # = 0xfefefefefefefefe)?
- checksum OK?
- guard bytes OK?
- free(p - sizeof(hdr));
- line # = 0xfefefefefefefefe;

Binary translator

Converts an executable object file to an instrumented executable object file.



Atom example (procedure call tracing)

Instrumentation file (ptrace.inst.c):

```
#include <stdio.h>
#include <instrument.h>

Instrument() {
   Proc *proc;
   AddCallProto("ProcTrace(char*)");

   for (proc = GetFirstProc(); proc != NULL; proc = GetNextProc(proc))
      AddCallProc(proc, ProcBefore, "ProcTrace", ProcName(proc));
}
```

Analysis file (ptrace.anal.c):

```
#include <stdio.h>

void ProcTrace(char *name) {
   printf("%s\n", name);
}
```


Instrumenting "hello,world"

```
% cc -Wl,r -non_shared hello.c -o hello.rr
% atom hello.rr ptrace.inst.c ptrace.anal.c -o hello.ptrace
% hello.ptrace
```

```
__start
__init_libc
__tis_init
__libc_locks_init
calloc
malloc
__sbrk
_errno
__getpagesize
__sbrk
__tis_mutex_lock
_unlocked_sbrk
__tis_mutex_unblock
memset
calloc
malloc
__sbrk
memset
main
printf
__tis_mutex_lock
_doprnt
__getmbcurmax
memcpy
__fwrite_unlocked
_wrtchk
_findbuf
__geterrno
__isatty
__ioctl
_cerror
__seterrno
__geterrno
__seterrno
memcpy
__tis_mutex_unblock
exit
__ldr_atexit
__fini_libc
_cleanup
__tis_mutex_trylock
__fclose_unlocked
__tis_mutex_trylock
__fflush_unlocked
__close_nc
__close
__tis_mutex_trylock
__fclose_unlocked
__tis_mutex_trylock
__fflush_unlocked
_xflsbuf
__write_nc
__write
hello, world
__close_nc
__close
__tis_mutex_trylock
__fclose_unlocked
__tis_mutex_trylock
__close_nc
__close
```

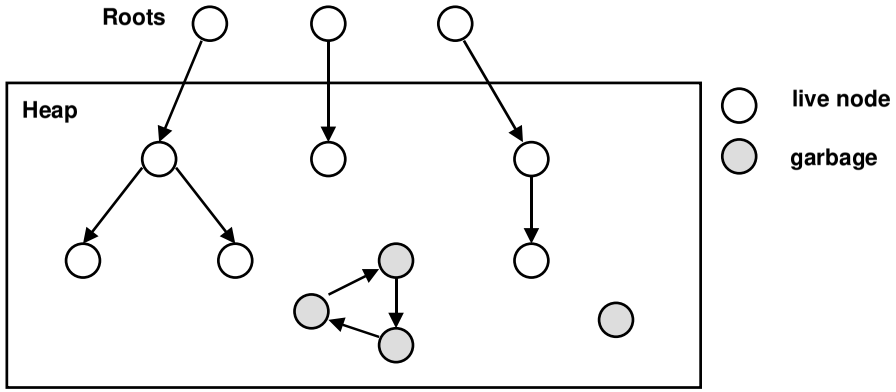

Atom tools

iprof - instruction profiling
liprof - instruction profiling at basic block level
syscall - syscall trace and performance analyzer
memsys - memory system simulator
io - io performance summary
gprof - call graph execution time profile
3rd - memory checker and leak finder (like Purify)
pixie - basic block profiling

Garbage collector

Garbage: unreachable allocated memory blocks (nodes).

- no pointers in registers, on stack, or in global variables (roots)
- no pointers in any allocated heap blocks reachable from the roots



Garbage collector

Garbage collection: automatic reclamation of heap-allocated storage after its last use by a program

```
void foo() {
   int *p = malloc(128);
   return; /* p block is now garbage */
}
```

Common in functional languages and modern object oriented languages:

- Lisp, ML, Java

Variants (conservative garbage collectors) exist for C and C++

Classical GC algorithms

Reference counting (Collins, 1960)

- described here

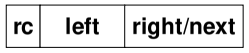
Mark and sweep collection (McCarthy, 1960)

Copying collection (Minsky, 1963)

- not described here
- see *Jones and Lin, "Garbage Collection: Algorithms for Automatic Dynamic Memory", John Wiley & Sons, 1996.*

Reference counting

Assume this format for each node



rc: reference count
left: ptr to left node
right: ptr to right node
or next node in free list

```
New () {
   newcell = allocate();
   newcell->rc = 1;
   return newcell;
}
```

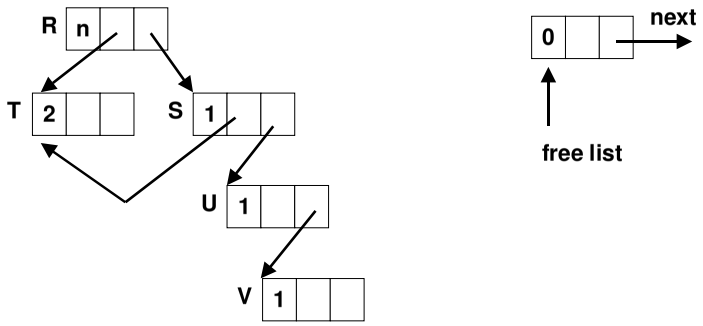
```
Free(U) {
   U->next = free_list;
   free_list = U;
}
```

```
Delete(T) {
   T->rc--;
   if (T->rc == 0) {
      foreach (U in Children(T))
         Delete(U);
      Free(T);
   }
}
```

```
Update(R,S) {
   Delete(R);
   S->rc++;
   R = S;
}
```

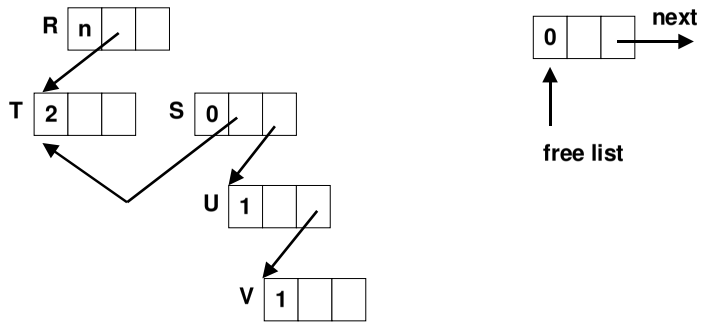
Reference counting example

Initially, before Update(right(R), NULL)



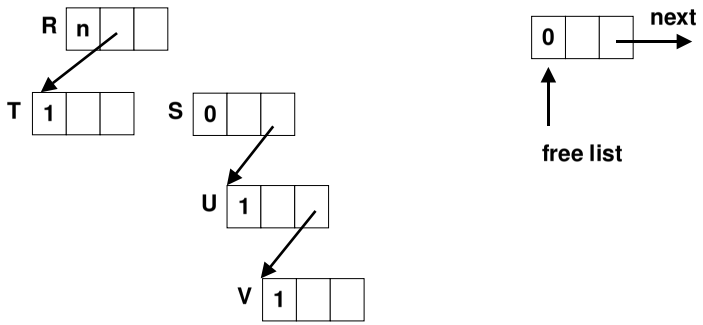
Reference counting example

after Delete(right(R))



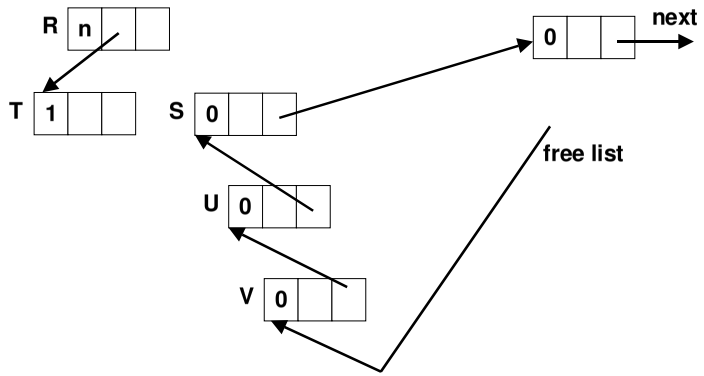
Reference counting example

after Delete(left(S))



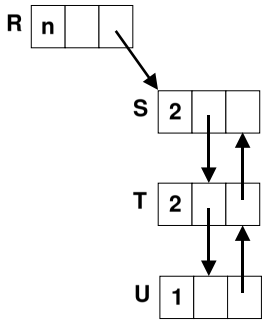
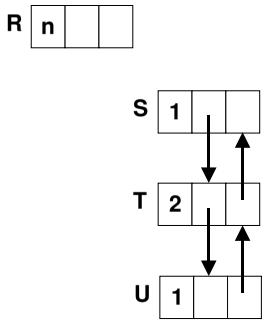
Reference counting example

after Update(right(R), NULL)



Reference counting cyclic data structures

before Delete(right(R))



after Delete(right(R))