System Assertions

Andreas Zeller

1

System Invariants

Some properties of a program must hold over the entire run:

- must not access data of other processes
- must handle mathematical exceptions
- must not exceed its privileges

Typically checked by hardware and OS

2

2

Memory Invariants

Even within a single process, some invariants must hold over the entire run

- code integrity
- data integrity

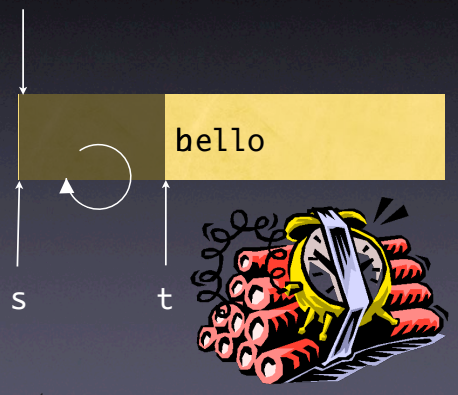
This is a major issue in C and C++

3

3

Heap Misuse

```
s = malloc(30)
free(s)
t = malloc(20)
strcpy(t, “hello”)
s[10] = ‘b’
free(s)
```



4

Heap Assertions

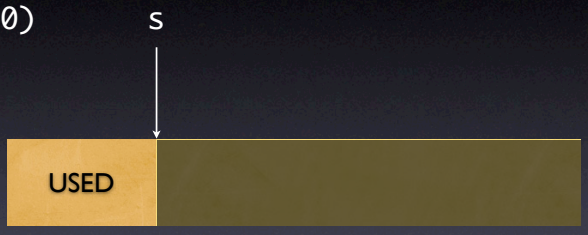
The *GNU C runtime library* provides a simple check against common errors:

```
$ MALLOC_CHECK_=2 myprogram myargs
free() called on area that was already free'd()
Aborted (core dumped)
$ _
```


5

Heap Assertions

```
s = malloc(30)
free(s)
free(s)
```




```
free() called on area that was already free'd()
Aborted (core dumped)
```


6

Array Assertions

The *Electric Fence* library checks for array overflows:

```
$ gcc -g -o sample-with-efence sample.c -lefence
$ ./sample-with-efence 11 14
Electric Fence 2.1
Segmentation fault (core dumped)
$ _
```


7

Array Assertions

```
s = malloc(30)
s[30] = 'x'
```


```
Segmentation fault (core dumped)
```


8

Memory Assertions

The *Valgrind* tool checks *all* memory accesses:

```
$ valgrind sample 11 14
Invalid read of size 4
  at 0x804851F: shell_sort (sample.c:18)
  by 0x8048646: main (sample.c:35)
  by 0x40220A50: __libc_start_main (in /lib/libc.so)
  by 0x80483D0: (within /home/zeller/sample)
```

Valgrind works as an *interpreter* for x86 code

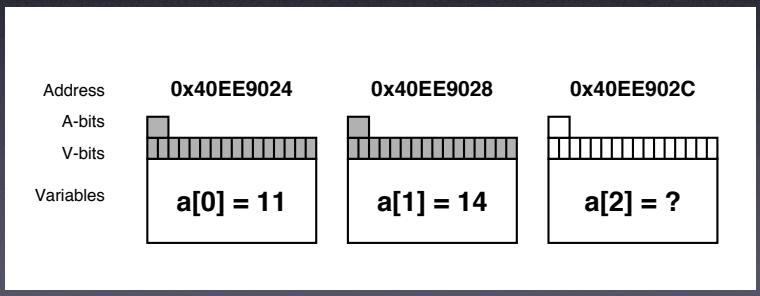
9

Valgrind Checks

- Use of uninitialized memory
- Accessing free'd memory
- Accessing memory beyond malloc'd block
- Accessing inappropriate stack areas
- Memory leaks: allocated area is not free'd
- Passing uninitialized memory to system calls

Shadow Memory

- V-bit set = corresponding bit is initialized
- A-bit set = corresponding byte is accessible



V-Bits

- When a bit is first written, its V-bit is set
- Simple read accesses to uninitialized memory do not result in warnings:

```
struct S { int x; char c; };
struct S s1, s2;
s1.x = 42;
s1.c = 'z';
s2 = s1;
```

5 bytes initialized

8 bytes copied (no warning)

V-Bits Warnings

Reading uninitialized data causes a warning if

- a value is used to generate an *address*
- a *control flow decision* is to be made
- a value is passed to a *system call*

A-Bits

- When the program starts, all global data is marked "accessible" (= A-bits are set)
- malloc() sets A-bits for the area returned; free() clears them
- Local variables are "accessible" on entry and "non-accessible" on exit
- Accessing "non-accessible" data ⇒ error

Overhead

Tool	GNU C Library	Electric Fence	Valgrind
Space	2 bytes/ malloc	1 page/ malloc	100%
Time	negligible	negligible	2500%

Preventing Misuse

- CYCLONE is a C dialect which prevents common pitfalls of C
- Most important feature: *special pointers*

16

Non-NULL Pointers

```
int getc (FILE @fp);
```

fp may not be NULL

```
extern FILE *fp;
char c = getc(fp);
```

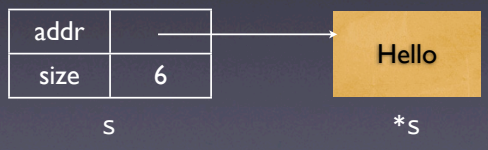
```
warning: NULL check inserted
```


17

Fat Pointers

- A fat pointer holds address *and* size
- All accesses via a fat pointer are automatically bounds-checked

```
int strlen(const char? s)
```



18

CYCLONE Restrictions

- NULL checks are inserted
- Pointer arithmetic is restricted
- Pointers must be initialized before use
- Dangling pointers are prevented through region analysis and limitations on free()
- Only "safe" casts and unions are allowed

Production Code

- Should products ship with active assertions?

Things to Check

- Critical results. If lives, health, or money depend on a result, it had better be checked.
- External conditions. Any conditions which are not within our control must be checked for integrity.

Points to Consider

- The more active assertions, the greater the chance to catch infections.
- The sooner a program fails, the easier it is to track the defect.
- Defects that escape into the field are the hardest to track.

More to Consider

- By default, failing assertions are not user-friendly.
  - ➡ *Handle assertions in a user-friendly way*
- Assertions impact performance.
  - ➡ *First measure; then turn off only the most time-consuming assertions*

Concepts

- ★ To check memory integrity, use specialized tools to detect errors at run time
- ★ Apply such tools before any other method
- ★ To fully prevent memory errors, use another language (or dialect, e.g. Cyclone)
- ★ Turning assertions off seldom justifies the risk of erroneous computation

25