Memory Management and Debugging

V22.0474-001 Software Engineering
Lecture 18
Outline

• Overview of memory management
  - Why it is a software engineering issue

• Styles of memory management
  - Malloc/free
  - Garbage collection
  - Regions

• Detecting memory errors

Adapted from Prof. Necula CS 169, Berkeley
Memory Management

• A basic decision, because
  - Different memory management policies are difficult to mix
    • Best to stick with one in an application
  - Has a big impact on performance and quality
    • Different strategies better in different situations
    • Some more error prone than others
Distinguishing Characteristics

- Allocation is always explicit
- Deallocation
  - Explicit or implicit?
- Safety
  - Checks that explicit deallocation is safe
Explicit Memory Management

• Allocation and deallocation are explicit
  - Oldest style
  - C, C++

    \[
    x = \text{new Foo;}
    \]

    ... 

    delete x;
A Problem: Dangling Pointers

X = new Foo;
...
Y = X;
...
delete X;
...
Y.bar();
A Problem: Dangling Pointers

\[ X = \text{new } \text{Foo}; \]
\[ \ldots \]
\[ Y = X; \]
\[ \ldots \]
\[ \text{delete } X; \]
\[ \ldots \]
\[ Y.\text{bar}(); \]

[Dangling pointers]
Notes

- Dangling pointers are bad
  - A system crash waiting to happen

- Storage bugs are hard to find
  - Visible effect far away (in time and program text) from the source

- Not the only potentially bad memory bug in C
Notes, Continued

• Explicit deallocation is not all bad

• Gives the finest possible control over memory
  - May be important in memory-limited applications

• Programmer is very conscious of how much memory is in use
  - This is good and bad

• Allocation and deallocation fairly expensive
Automatic Memory Management

• I.e., automatic deallocation

• This is an old problem:
  - studied since the 1950s for LISP

• There are well-known techniques for completely automatic memory management

• Until recently unpopular outside of Lisp family languages
The Basic Idea

• When an object is created, unused space is automatically allocated
  - E.g., new X
  - As in all memory management systems

• After a while there is no more unused space

• Some space is occupied by objects that will never be used again
  - This space can be freed to be reused later
The Basic Idea (Cont.)

- How can we tell whether an object will “never be used again”?  
  - in general, impossible to tell  
  - use heuristics

- Observation: a program can use only the objects that it can find:  
  \[ A \times = \text{new } A; \ x = y; ... \]
  - After \( x = y \) there is no way to access the newly allocated object
Garbage

• An object $x$ is **reachable** if and only if:
  - a register contains a pointer to $x$, or
  - another reachable object $y$ contains a pointer to $x$

• You can find all reachable objects by starting from registers and following all the pointers

• An unreachable object can never be used
  - such objects are **garbage**
Reachability is an Approximation

- Consider the program:
  \[
  x = \text{new } A;
  
  y = \text{new } B;
  
  x = y;
  
  \text{if}(\text{alwaysTrue}()) \{ x = \text{new } A \} \text{ else } \{ x.\text{foo}() \}
  \]

- After \( x = y \) (assuming \( y \) becomes dead there)
  - the object \( A \) is unreachable
  - the object \( B \) is reachable (through \( x \))
  - thus \( B \) is not garbage and is not collected
    - but object \( B \) is never going to be used

Adapted from Prof. Necula CS 169, Berkeley
A Simple Example

- We start tracing from registers and stack
  - These are the roots

- Note B and D are unreachable from acc and stack
  - Thus we can reuse their storage
Elements of Garbage Collection

• Every garbage collection scheme has the following steps
  1. Allocate space as needed for new objects
  2. When space runs out:
     a) Compute what objects might be used again (generally by tracing objects reachable from a set of “root” registers)
        b) Free the space used by objects not found in (a)
• Some strategies perform garbage collection before the space actually runs out

Adapted from Prof. Necula CS 169, Berkeley
Notes on Garbage Collection

• *Much* safer than explicit memory management
  - Crashes due to memory errors disappear
  - And easy to use

• But exacerbates other problems
  - Memory leaks can be hard to find
    • Because memory usage in general is hidden
  - Different GC approaches have different performance trade-offs
Notes (Continued)

- Fastest GCs do not perform well if live data is significant percentage of physical memory
  - Should be < 30%
  - If > 50%, quite dramatic performance degradation

- Pauses are not acceptable in some applications
  - Use real-time GC, which is more expensive

- Allocation can be very fast

- Amortized deallocation can be very fast, too
Finding Memory Leaks

• A simple automatic technique is effective at finding memory leaks

• Record allocations and accesses to objects

• Periodically check
  - Live objects that have not been used in some time
  - These are likely leaked objects

• This can find bugs even in GC languages!
A Different Approach: Regions

• Traditional memory management:

<table>
<thead>
<tr>
<th></th>
<th>free</th>
<th>GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Control</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Ease of use</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Space usage</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

• A different approach: regions

  safety and efficiency, expressiveness

Adapted from Prof. Necula CS 169, Berkeley
Region-based Memory Management

- Regions represent areas of memory
- Objects are allocated “in” a given region
- Easy to deallocate a whole region

```c
Region r = newregion();
for (i = 0; i < 10; i++) {
    int *x = ralloc(r, (i + 1) * sizeof(int));
    work(i, x); }
deleteregion(r);
```
Why Regions?

- Performance
- Locality benefits
- Expressiveness
- Memory safety
Region Performance: Allocation and Deallocation

• Applies to delete all-at-once only

• Basic strategy:
  - Allocate a big block of memory
  - Individual allocation is:
    - pointer increment
    - overflow test
  - Deallocation frees the list of big blocks

⇒ All operations are fast

Adapted from Prof. Necula CS 169, Berkeley
Region Performance: Locality

- Regions can express locality:
  - Sequential allocs in a region can share cache line
  - Allocs in different regions less likely to pollute cache for each other

- Example: moss (plagiarism detection software)
  - Small objects: short lived, many clustered accesses
  - Large objects: few accesses
Region Performance: Locality - moss

- 1-region version: small & large objects in 1 region
- 2-region version: small & large objects in 2 regions
- 45% fewer cycles lost to r/w stalls in 2-region version
Region Expressiveness

- Adds some structure to memory management

- Few regions:
  - Easier to keep track of
  - Delay freeing to convenient "group" time
    - End of an iteration, closing a device, etc

- No need to write "free this data structure" functions
Region Expressiveness: lcc

- The lcc C compiler
  - regions bring structure to an application's memory
Region Expressiveness: lcc

- The lcc C compiler, written using unsafe regions
  - regions bring structure to an application's memory
Region Expressiveness: lcc

- The lcc C compiler, written using unsafe regions
  - regions bring structure to an application's memory

---

Adapted from Prof. Necula CS 169, Berkeley
Region Expressiveness: ldc

- The ldc C compiler, written using unsafe regions
  - regions bring structure to an application's memory

Adapted from Prof. Necula CS 169, Berkeley
Region Expressiveness: lcc

- The lcc C compiler, written using unsafe regions
  - regions bring structure to an application's memory
Region Expressiveness: Icc

- The Icc C compiler, written using unsafe regions
  - regions bring structure to an application's memory

Adapted from Prof. Necula CS 169, Berkeley
## Summary

<table>
<thead>
<tr>
<th></th>
<th>regions</th>
<th>free</th>
<th>GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Control</td>
<td>++</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Ease of use</td>
<td>=</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Space usage</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Time</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
Region Notes

• Regions are fast
  - Very fast allocation
  - Very fast (amortized) deallocation
  - Can express locality
    • Only known technique for doing so

• Good for memory-intensive programs
  - Efficient and fast even if high % of memory in use
Region Notes (Continued)

- Does waste some memory
  - In between malloc/free and GC

- Requires more thought than GC
  - Have to organize allocations into regions
Summary

- You must pay attention to memory management
  - Can affect the design of many system components
- For applications with low-memory, no real time constraints, use GC
  - Easiest strategy for programmer
- For high-memory or high-performance applications, use regions
Run-Time Monitoring

• Recall from testing:
  - How do you know that a test succeeds?
  - Can check intermediate results, using assert

• This is called run-time monitoring (RTM)
  - Makes testing more effective
What do we Monitor?

• Check the result of computation
  - E.g., the result of matrix inversion
• Hardware-enforced monitoring
  - E.g., division-by-zero, segmentation fault
• Programmer-inserted monitoring
  - E.g., assert statements
Automated Run-Time Monitoring

- Given a property $Q$ that must hold always
- ... and a program $P$

- Produce a program $P'$ such that:
  - $P'$ always produces the same result as $P$
  - $P'$ has lots of `assert(Q)` statements, at all places where $Q$ may be violated
  - $P'$ is called the instrumented program

- We are interested in automatic instrumentation

Adapted from Prof. Necula CS 169, Berkeley
RTM for Memory Safety

• A technique for finding memory bugs
  - Applies to C and C++

• C/C++ are not type safe
  - Neither the compiler nor the runtime system enforces type abstractions

• Possible to read or write outside of your intended data structure

Adapted from Prof. Necula CS 169, Berkeley
Picture

memory  objects

Access to A  Access to A  Access to A
The Idea

- Each byte of memory is in one of three states:
  - Unallocated
    - Cannot be read or written
  - Allocated but uninitialized
    - Cannot be read
  - Allocated and initialized
    - Anything goes
State Machine

Associate an automaton with each byte

Unallocated → Uninitialized

Uninitialized → Initialized

Initialized → Initialized

Missing transition edges indicate an error
Instrumentation

- Check the state of each byte on each access

- Binary instrumentation
  - Add code before each load and store
  - Represent states as giant array
    - 2 bits per byte of memory

- 25% memory overhead
  - Catches byte-level errors
  - Won’t catch bit-level errors
Note: We can detect invalid accesses to red areas, but not to blue areas.
Improvements

- We can only detect bad accesses if they are to unallocated or uninitialized memory

- So try to make most of the bad accesses be of those two forms
  - Especially, the common off-by-one errors
Red Zones

• Leave buffer space between allocated objects
  - The “red zone”
  - In what state do we put this zone?

• **Guarantees that walking off the end of an array accesses unallocated memory**
**Aging Freed Memory**

- When memory is freed, do not reallocate immediately
  - Wait until the memory has “aged”

- Helps catch dangling pointer errors

- Red zones and aging are easily implemented in the malloc library
Another Class of Errors: Memory Leaks

- A memory leak occurs when memory is allocated but never freed.

- Memory leaks can be even more serious than memory corruption errors

- We can find many memory leaks using techniques borrowed from garbage collection
The Basic Idea

- Any memory with no pointers to it is leaked
  - There is no way to free this memory

- Run a garbage collector
  - But don’t free any garbage
  - Just detect the garbage
  - Any inaccessible memory is leaked memory
Issues with C/C++

• It is sometimes hard to tell what is inaccessible in a C/C++ program

• Cases
  - No pointers to a malloc’d block
    • Definitely garbage
  - No pointers to the head of a malloc’d block
    • Maybe garbage
Leak Detection Summary

- From time to time, run a garbage collector
  - Use mark and sweep

- Report areas of memory that are definitely or probably garbage
  - Need to report who malloc’d the blocks originally
  - Store this information in the red zone between objects
Tools for Memory Debugging

- **Purify**
  - Robust industrial tool for detecting all major memory faults
  - Developed by Rational, now part of IBM

- **Valgrind**
  - Open source tool for Linux
  - [http://valgrind.org](http://valgrind.org)

- “Poor man’s purify”
  - Implement basic memory checking at source code level
  - Sample project includes a simple debugger called simpurify