ECE498SL: Engineering Software Systems **Spring 2009**
Lab #1 **Due: Thursday 26 February**

Implementing Memory Debugging with Pin

This document describes an approach for identifying errors in the use of a dynamic memory library using the Pin tool mentioned earlier. The first section uses a formal model in an attempt to provide some sort of systematic identification of potential problems as well as the necessity of explicitly specifying one's assumptions when using such abstractions. The second section discusses implementation with the Pin tool, reasons about the overhead of various approaches, and describes a few practical tricks that people have used in the past to implement such debugging tools more effectively. The final section specifies your implementation.

Identifying Dynamic Memory Errors

One can use a model of the system to identify subtle failure cases. A certain amount of abstraction is necessary to define such a model, which can also obscure failure cases, so it's important to think carefully about assumptions made implicitly.

We'll start by viewing the problem from the point of view of the state of a given block of dynamic memory. By adopting such a viewpoint, we've already lost the ability to reason about inter-block relationships that may be important in a language that supports pointer arithmetic. We also obscure temporal relationships that may only become clear by reasoning about sequences of states. Finally, we implicitly ignore issues with static and stack memory.

A given dynamic block is either allocated or unallocated. Unallocated could mean that the block has never been allocated or that it was allocated some number of times, but is currently free. We use the symbol A to denote this bit of state.

The other bit of state that we will track in our model is whether or not the block is referenced by the program using it. What exactly do we mean by "referenced?" Accesses to any memory location can be expressed easily in C (or C++), so in a strict definition, all blocks are always referenced. Instead, let's assume that the program has some reasonable set of data structures and that the code only uses those data structures in reasonable ways to reach pointers to dynamic blocks of memory. In that case, we could theoretically ask one of the authors at any given point of time, "True or false: you can find this block using your data structures?" The answer to that question is what we'll call R.

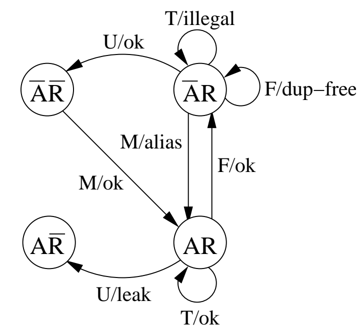
The next step is to identify the set of events that occur from a block's point of view. One way to think about the events to be considered is to start with state-changing events, although such a set won't be complete. Block references are controlled by the user code, but control of allocation is shared with the library code, thus events can be generated from both sources.

The library initiates only one kind of event for any given block: selecting it for allocation to user code in response to a call to `malloc`. We use M to denote this event. In general, we might not want to assume that the library contains no bugs. In the (typical) absence of protection boundaries between the user code and the library data, we might also not want to assume that the user code has not somehow corrupted the library's data (or, technically, code) in a way that causes the library to produce errors. However, in the interest of simplicity, we'll start by ignoring these possibilities and assume that M events only occur when $\overline{A}$.

The user code can generate three types of events on a block. As we did with the definition of the reference state R, we once again assume that user code does not generate events unless R. Note that these assumptions are substantially less likely to hold, given that blocks are often adjacent in memory and dynamic memory is often used for arrays. The first event generated by user code, F, occurs when the code calls `free` on the block. The second event, U, represents the elimination of the last remaining reference to the block. Finally, we must consider events that do not change a block's state. For a memory block, we define a third event, T, that represents access to the block (either read or write) by user code. We ignore the symmetric event in which library code accesses the block, implicitly making the assumption that the library never does so incorrectly.

The state and events for a dynamic memory block are summarized in the table below. Now we're ready to develop the transition diagram for a block and to identify error conditions, for which the final product is shown to the right of the table. We begin by drawing and marking the four states and adding the arcs for the expected use case. A block starts in the $\overline{A}\overline{R}$ state in the upper left. In this state, only an M event is possible; such an event takes the state to AR, reflecting both the change in allocation and the implicit creation of a reference. The user code makes use of the block for a while, as indicated by the self-looping T event, and eventually frees the block, bringing the state up to $\overline{A}R$. After freeing the block, the user code discards any remaining references to it, and the resulting U event brings the block state back to the original $\overline{A}\overline{R}$, at which point the cycle can begin again.

block state	
A	block is currently allocated to user code
R	block is referenced by user code state, *i.e.*, a pointer to the block is stored in the reachable user data and is considered to be valid
events generated by library (can only occur when $\overline{A}$, assuming correct and uncorrupted library code)	
M	occurs when block is allocated by the library
events generated by user code (can only occur when R, assuming reasonably sane C code)	
F	occurs when block is freed
T	occurs when block is accessed (read or written)
U	occurs when last reference to block is destroyed



We next add arcs corresponding to other possible events. The F, T, and U events can occur from both $\overline{A}R$ and AR. Recall that in the normal use model, a block in the $\overline{A}R$ state has been freed, but references to it have not been eliminated. If F occurs in this state, the user code has made a duplicate call to `free`. Similarly, if T occurs in this state, the user code has accessed the block after freeing it. Both are errors, but neither affects the state of the block, as indicated by the self-loops in the figure. In contrast, if U occurs in the AR state, the block transitions to $A\overline{R}$, and no further events are possible for the block, since the library views the block as allocated, but the user code has no way to access it. This transition thus corresponds to a memory leak: the memory has been lost to the system.

Finally, the M event can also occur from the $\overline{A}R$ state, in which case references to an old, freed object or objects remaining in the user code's data structures now alias to the newly allocated object or objects.

Through this process, we have identified four kinds of errors. Now let's review some of our assumptions.

- ignore non-dynamic memory blocks (static and stack data)
- user code does not act on unreferenced blocks
- rely on programmer knowledge for state R
- library is correct and cannot be corrupted
- (sub-property) library does not touch allocated blocks inappropriately

We started by ignoring static and stack memory. While we don't want to generalize our model to include all aspects of memory management, such as retaining a reference to a stack variable, we do want to capture cases in which non-dynamic memory is mistakenly treated as dynamic memory. An M event for non-dynamic memory would imply library code errors or corruption, which we shall continue to ignore. On the other hand, an F event for a non-dynamic block represents a bad call to `free`, which we add to our list of errors.

Generalizing our approach to avoid assumptions about the user code event generation does not add much to our abstract model, although in practice doing so can be helpful. Consider adding arcs for F and T events to the $\overline{A}\overline{R}$ and $A\overline{R}$ states. Adding arcs for U events is meaningless given our definition of R. Arcs from $\overline{A}\overline{R}$ are the same as those from $\overline{A}R$: whether or not the programmer considers a block to be part of the user data is not particularly important after the block has been freed. We're already starting to question the practical likelihood of knowing R dynamically; if we expect to know R in practice, we might also choose to signal errors for both of these arcs to signify violations of program semantics. For the same reason, touching and freeing blocks in state $A\overline{R}$ is hard to differentiate from the same actions in AR. The F events happen to get lucky, freeing memory that had previously leaked. As you might imagine, detecting that a programmer accidentally violated program semantics in order to free a block that leaked is quite challenging.

errors identified by model	
dup-free	duplicate `free` call made on block
illegal	block accessed without ownership rights
alias	references retained to old freed block alias with newly allocated block
leak	references discarded without freeing block
errors on non-dynamic memory blocks	
bad-free	`free` call made on non-dynamic memory

The table above lists the five types of errors that we have identified. As a final topic in this section, we discuss the difficulty in defining the system state in terms of a programmer's knowledge, in particularly through the R bit.

Can we ever know whether a given block is in R or $\overline{R}$? Certainly such knowledge is not free: even if we were able to "ask one of the authors," they are human and prone to making mistakes when evaluating complex data structures. Asking them to identify all referenced data systematically is also error-prone, and **the information is certainly not free.** A few people mentioned, for example, checking for leaks by identifying remaining memory allocations at the end of a program. Not all programs terminate—high-availability servers, for example, are generally not designed to clean themselves up by shutting down. Also, operating systems generally ensure that resources in use when a program terminates are freed for use by other programs. Although some thin support packages such as DOS did not provide such guarantees, most programs are written assuming this level of support.

How hard is it for a programmer to write routines that identify R? Doing so is not particularly hard in practice if a memory debug package is available: given an initial definition for R, the package identifies data that should have been included but were accidentally overlooked as memory leaks. The programmer can thus extend the definition to include these blocks and try again. Using this feedback-based process, constructing code to identify R correctly becomes relatively simple. Nevertheless, some work is necessary.

As an alternative strategy, we can leverage the distributions of bit patterns in pointers and integers to try to automatically determine R. Information about which regions of virtual memory correspond to which types of data are available with most operating systems, and are exposed to debugging interfaces as well (try "info sharedlibrary" in `gdb` or look in `/proc/<pid>/maps` under Linux). Most integers tend to be small, but most operating systems do not make use of virtual addresses near zero, so differentiating them by bit pattern is usually fairly accurate. Pointers stored in memory are also generally aligned to 4-byte addresses. Using this information, one can recursively search memory, starting with the stack and static data areas, to find all regions for which the program currently has a pointer. Debugging information can eliminate most of the guesswork by providing type information about the contents of both. Knowing whether a given pointer refers to an object or an array is not necessarily straightforward, thus the size of objects in the heap is difficult to know; here, a memory debugging package can track information about object size and, if necessary, the source code line responsible for allocation.

One can also make assumptions about pointer values based on alignment: while a character pointer might take any address, a structure pointer is generally aligned to the size of its largest member, thus a structure with another pointer must be 4-byte-aligned. Regions referenced by pointers without such alignment must still be marked, but need not be searched for more pointers.

Using Pin to Find Dynamic Memory Errors

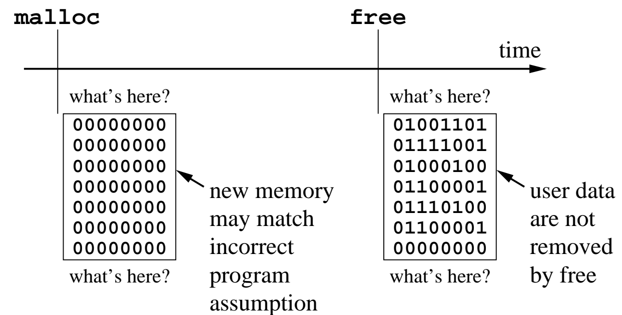
Pin is a powerful infrastructure, but using it at a low-level incurs substantial overhead. Examining the entire dynamic instruction stream for a program, for example, suffices to catch most of the errors that we identified in the previous section, assuming that we either obtain information about R from the programmer or try to infer it directly.

However, executing even a brief function for each dynamic instruction is likely to slow down program execution by several orders of magnitude, which is not an acceptable practical impact except for pure debugging purposes. For the more subtle errors that might crop up only after a program has been running for a few days and is supporting, for example, hundreds of remote users, we need an approach that incurs overheads of perhaps 5-10%.

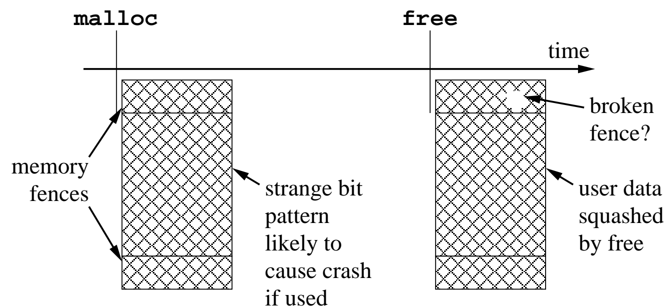
Pin can, of course, let us operate in this manner as well, by capturing only calls to the dynamic memory management routines, for example. Such calls are infrequent enough that we can afford to execute some simple code without incurring unacceptable overhead.

This approach is sufficient for identifying duplicate and bad calls to `free`. To identify aliasing and memory leaks, a fair amount of work is still necessary to identify R, thus such checks are run periodically rather than on each call, with the frequency of checks chosen to be small enough to meet overhead constraints. Illegal accesses are the most difficult type of error to catch without examining actual memory references. For this type of error, we can fall back on a few practical mechanisms designed for this purpose, including memory fences and scrubbing.

The mechanisms that we now discuss are motivated by the most common types of errors in the general category of illegal accesses, which can be broadly classified into temporal and spatial errors. Temporal errors include mistakes such as failure to initialize memory after it has been allocated and freeing a block just before making a final use of its contents. Spatial errors include accessing elements beyond the ends of a dynamically-allocated array.



In order to reason about how one can identify such errors efficiently, we first consider the contents and position of dynamic memory blocks as they appear immediately after library calls, as shown to the right. Initialization errors due to partial initialization of structures are fairly common, and are exacerbated by implicit initialization of static data as well as the availability of library calls that zero new dynamic blocks (which encourages programmers to save a few cycles by leaving some fields in a structure uninitialized). As you may recall, operating systems zero out physical pages before allocating them to a given process in order to prevent information (e.g., an unencrypted password) from leaking to the process. Many memory allocations, and perhaps all memory allocations if a system is not tested using a significant workload, may thus produce blocks of memory that are in fact filled with zero when returned from `malloc`. Similarly, implementations of `free` do not typically overwrite user data stored in the block before returning, thus improper accesses to the data soon after the call to `free` are likely to obtain the results that the programmer expected, *i.e.*, valid data. Finally, accesses to memory past the bounds of an allocated block have unpredictable results, in large part because the adjacent memory could in theory serve a wide variety of purposes, and its contents are also unpredictable.



Two mechanisms help to address these issues. Memory fences address the spatial issues: as shown to the right, rather than allocating a block with exactly the number of bytes requested, we change the memory system to allocate extra memory before and after the block. The purpose and contents of this extra memory is thus predictable, and accesses beyond the bounds of the allocated memory are easier to detect. Some memory debug systems take advantage of operating system support by lining up the lower edge of objects (which is more likely to be overrun than the upper, or negative, direction) with a page boundary and making the next page inaccessible. With such support, accesses immediately generate exceptions. For our purposes, we will make do with trying to crash the program if it reads the data and trying to detect writes when the memory is freed.

The second mechanism is the use of bit patterns that are likely to lead to crashes if they are used. For example, one can use copies of a large integer referencing invalid memory if interpreted as an address. Blocks are then filled with this bit pattern before they are returned from `malloc`, and user data is overwritten with this pattern when a block is freed (sometimes called scrubbing) to ensure that later accesses do not retrieve valid data. Memory fences can be filled with a similar pattern—using a distinct pattern is slightly better than using exactly the same one for inside and outside the blocks. The integrity of the fences can then be checked periodically and when the block is freed to detect inappropriate changes.

Obviously, these mechanisms are not infallible. Large integers do not necessarily lead to crashes. An invalid program may happen to write exactly the same bits into a fence. However, such coincidences are unlikely, and the mechanisms are fairly effective in practice.

The Implementation

For this assignment, you must write a generic memory debugging package based on the Pin tool. You must catch all calls to the standard C library interfaces (`malloc`, `calloc`, `realloc`, and `free`) and manipulate the calls to implement the mechanisms described in the previous section. You must also track all memory allocations in order to identify duplicate and bad calls to `free`. Errors of these types should be reported (giving the source file and line number of the call site as well as the address passed to `free`) and squashed (return without actually calling `free`). Broken memory fences should also generate messages. Although you will not be able to identify a source line for the access in these cases, your message should include information about the address and allocation source line for the object with a broken fence.

These mechanisms will give you some probabilistic coverage of illegal accesses, and you should try to scan registers for your bad bit pattern if a program crashes (*e.g.*, receives an unexpected signal).

You do not need to implement a mechanism through which you determine R, and thus you may not be able to identify memory leaks or alias problems. However, you should dump a list of currently allocated blocks (block address and actual size along with source file and line) when the program terminates or crashes.

You may work in teams of three for this assignment. Let me know if you need Unix groups on the EWS machines.

errors	strategy
dup-free bad-free	track allocated memory blocks and use to identify when `free` call occurs; report source file and line number of call site and address passed to `free`; your error messages do not need to differentiate these two errors; squash the bad calls
illegal	allocate blocks with memory fences (multiples of 16B to maintain alignment needs); fill new blocks and fences with appropriate bit patterns of your choice; catch abnormal program termination, try to identify bit pattern in registers (and report it); check memory fence when block is freed; report block address and allocation source line if fence has been broken; fill objects with appropriate bit patterns when they are freed
alias leak	on program termination (normal or abnormal), dump block address, block size, source file, and source line

The following links provide documentation and an example of obtaining information about program sources with Pin:

- `http://www.pintool.org/docs/24110/Pin/html/group__DEBUG__API.html` (these are both double underscores)
- `pin-2.6/source/tools/ManualExamples/statica.cpp` (from the Pin sources)