Sample Final Exam (Fall 2014)

Solutions are shown in this style. This exam was given in Fall 2014.

1. Explicit Memory Management (14 pts)

The following questions relate to explicit memory management.

a) (3 pts) *Detecting usage errors.* Programs that use explicit memory management are notorious for the potential for programmer mistakes. Consider the following C program as an example:

```
#include <stdlib.h>
int main()
{
    void * ptr = malloc(256);
    free(ptr);
    free(ptr);
}
```

When run under Linux, it outputs:

```
*** Error in `./dfree': double free or corruption (top):
0x0000000000f10010 ***
Aborted (core dumped)
```

How would you add the ability to output such warnings to your project 3 malloclab implementation?

This can be done simply by checking that the boundary tag header immediately preceding the pointer passed to free() has the 'inuse' bit set. On the first free(), the bit is reset. Subsequent attempts to free(), if they find the bit clear, will issue the warning. Note that this check may have false negatives if intervening allocations and deallocations reused the memory in which the boundary tag header is located, or if the inuse bit was cleared by memory corruption.

b) (4 pts) *Tolerating usage errors.* Unlike in the previous question, where you were asked to add facilities that would help detect usage errors, in this question, you should describe 2 different techniques that you could add to your allocator such that it would tolerate memory-related programming errors!

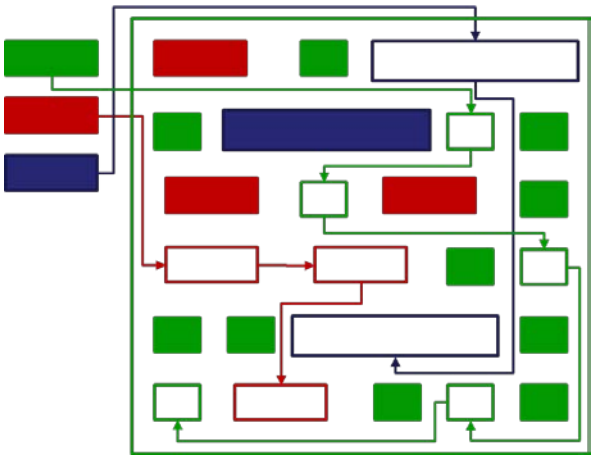
Possible ideas include:

- Padding of allocated areas to tolerate index out-of-bound errors
- Delayed free to tolerate access-after-free errors

- Zero-filling to avoid uninitialized access errors (e.g., unterminated strings)

For a system that does this, see “Rx: Treating Bugs As Allergies— A Safe Method to Survive Software Failures” by Qin et al, SOSP’ 2005. http://dx.doi.org/10.1145/1275517.1275519

c) (3 pts) Considering the following picture from the lecture slides, which shows the memory layout of a segregated fit allocator.



If you carefully study the image, you find that it violates an invariant that is commonly found in allocators.

Which invariant is violated in the depiction shown?

The picture shows two free blocks (for instance, the first two on the red list) adjacent to each other, which means they were not coalesced properly (unless the allocator uses delayed coalescing, which is less commonly done.) In an ordinary segregated fit allocator, the blocks should have been coalesced and added to the blue free list.

d) (4 pts) In project 2, some students implemented a custom memory allocator for futures by maintaining a list of free futures for each worker thread, as sketched below:

```
/* Submit a callable to FJ pool, returns future */
struct future * thread_pool_submit(struct thread_pool * pool,
                   fork_join_task_t callable,
                   void *callable_data)
{
    struct future * f;
    struct worker * worker = current_worker; // thread-local variable
    if (worker == NULL || worker->nfree_futures == 0) {
        f = malloc(sizeof *f);
    } else {
        worker->nfree_futures--;
        f = list_entry(list_pop_front(&worker->free_futures),
                       struct future, elem);
    }
    // code where future is initialized and submitted to the pool elided
    return f;
}

/* Deallocate this future. */
```

```
void
future_free(struct future *f)
{
    struct worker * worker = current_worker;
    if (worker == NULL) { // external thread, e.g., main
        free(f);
    } else {
        list_push_back(&worker->free_futures, &f->elem);
        worker->nfree_futures++;
    }
}
```

Describe one advantage and one disadvantage of this idea!

i. (2 pts) Advantage

Advantages include:

- No potential for contention when acquiring a lock in malloc() since the free list is per-thread.
- Constant time allocations: allocations from the per-worker free list are guaranteed to be constant time O(1), which may or may not be true for the general malloc().

ii. (2 pts) Disadvantage

As with all custom memory allocators, the key disadvantage is the potential for lower utilization, for two reasons:

- Unused memory on the per-worker free list cannot be used for other purposes, particular if, as in the implementation shown, the list is allowed to grow unbounded.
- There is a heightened potential for fragmentation since unused blocks kept on the per-worker free list cannot be coalesced with other free blocks.

2. Automatic Memory Management (14 pts)

The following questions relate to automatic memory management.

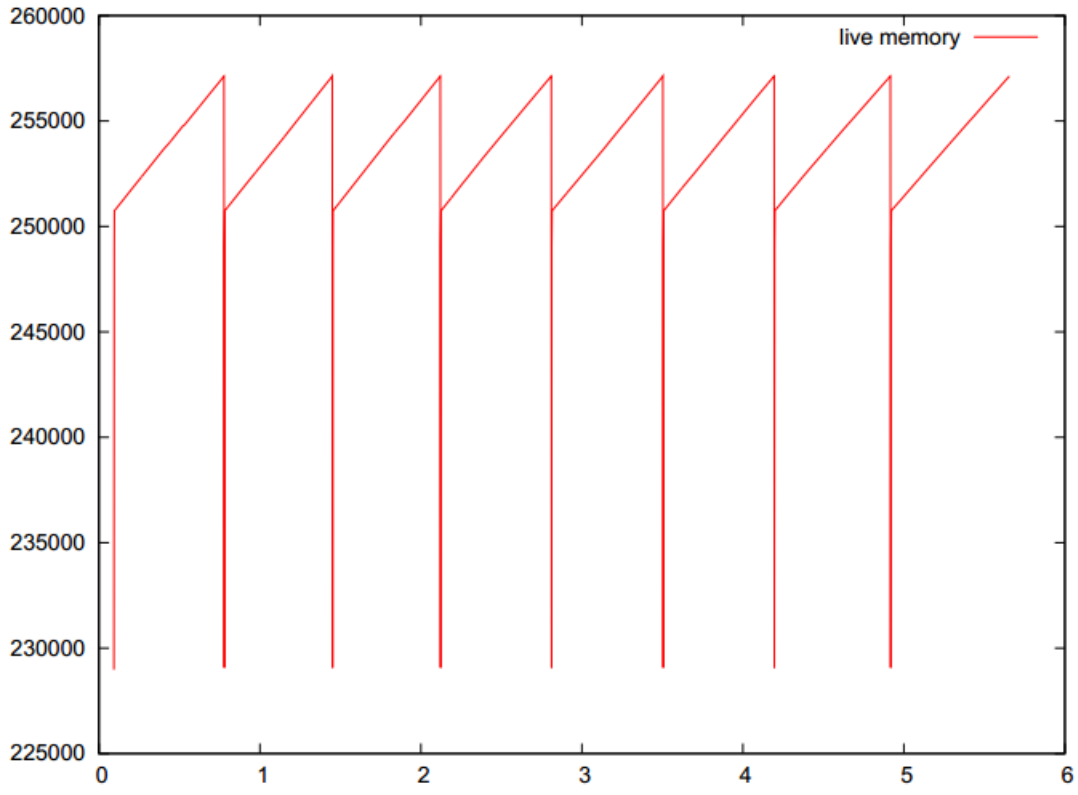
a) (8 pts) Sketch the live memory graph of the following Java program (the API is the same as in exercise 4 – every call to takeLiveHeapSample() outputs a data point with the size of the live heap.)

```
import java.util.*;

public class MysteryFigure
{
    static void a() throws Exception {
        int [] a = new int[5000];
```

```
            HeapTracker.takeLiveHeapSample();   // take sample
            List<Object> list = new ArrayList<Object>(400);
            for (int i = 0; i < 400; i++) {
                list.add(new Object());
                HeapTracker.takeLiveHeapSample();   // take sample
            }
        }

        public static void main(String []av) throws Exception {
            HeapTracker.startTrace();   // start tracing
            for (int i = 0; i < 8; i++) {
                HeapTracker.takeLiveHeapSample();   // take sample
                a();
            }
            HeapTracker.stopTrace();   // stop tracing
        }
    }
```

Your sketch/drawing goes here:

It should look like a saw blade. Note the deep “cuts” when the int [] array is allocated:



b) (3 pts) Why can many garbage collectors move objects in order to compact the heap whereas explicit allocators (e.g. general malloc()) are usually unable to do so?

Because these garbage collectors operate in execution environments for typesafe languages in which the collector can precisely identify the locations of pointers, and in which the program's code cannot arbitrarily produce pointers by casting integers. This property holds true for languages such as Java, JavaScript, Python, and many others. Conversely, general-purpose explicit allocators have no knowledge of where a program may store pointers returned by calls to malloc(), so they cannot move the blocks of memory to which those pointers refer.

c) (3 pts) In exercise 5, you implemented a simulation of a mark-and-sweep collector. How do cycles in the reachability graph affect a mark-and-sweep collector?

Not at all. If there is a cycle in the reachable part of the heap, the mark-and-sweep's per-node 'visited' flag will ensure that the mark algorithm terminates even though there are cycles – as in any graph traversal algorithm. If there is a cycle in the unreachable part of the heap, it does not affect the collector's ability to sweep unreachable objects – all objects contained in a cycle are still identified as garbage. This property is unlike reference-counting based schemes which cannot easily handle cycles in the unreachable part of the heap. Exercise 4 part1 was intended to reinforce how mark-and-sweep works.

3. Virtual Memory (20 pts)

a) (6 pts) Computer systems did not always exploit virtual memory in the way they do today. For instance, in some systems, all movement of data between main memory and secondary storage was done by the programmer. In fact, an advertisement from May 1971 (depicted on the right) points out that some large computer manufacturers of that time (IBM) were initially skeptical of the idea.

List 2 distinct technical challenges that had to be overcome before virtual memory could become a viable technology!

The main concerns centered on the overhead and its impact on speed/performance, in particular:

- Translation overhead: a VM system requires a virtual-to-physical translation for each instruction (so the processor can fetch it), which in turn was solved by introducing special-purpose hardware (TLB) to cache these translations.

- Page fault frequency: for the average access speed of the resulting system to approach that of RAM, page fault rates must be kept very low. This problem was solved by the introduction of the concept of a working set and good page replacement algorithms.

b) (4 pts) For programs that must process large amounts of data in memory, which one is faster: declaring a global `char[]` array of maximum size or mmap'ing the necessary amount of anonymous memory? Justify your answer!

From a virtual memory point of view, the two approaches are nearly indistinguishable: when loading a program with a large global array (declared in its bss section) the program loader will simply reserve virtual addresses for it, exactly in the same way as mmap() will – in fact, inside the kernel, the same functions are called. Physical memory is allocated only when the process accesses the individual pages. The additional overhead of the mmap() system call is insignificant when compared to the cost of the minor page faults to bring in the pages in the area, especially if it is large as stated in the problem.

c) (4 pts) We know that the operating system deallocates all memory a process has allocated when the process exits, thus it is not required to call free() before exiting. Though not required, could there be benefits to doing so? Alternatively, could there be drawbacks? Or does it not matter?

There are only drawbacks: In the best case, CPU time is wasted in adjusting pointers and lists in pages that are shortly overwritten. In the worst case, the OS will need to bring in swapped-out pages from disk in order to perform those meaningless operations.

The only argument in favor might be one of software evolution, which doesn't justify actually calling free() before exit(), but may justify leaving free() calls in code that might later be used repeatedly in long running programs, in which case not calling free() would create memory leaks.

d) (6 pts) Consider the following program:

```
// infrec.c
static void recurse() {
    char buffer[8192];
```

```
        buffer[0] = 'A';
        recurse();
    }

    int main() { recurse(); }
```

When run on `boxelder` (one of the machines on our rlogin cluster), the program does not appear to finish. After 1 minute, the output of the Unix command `top` shows the information below, which includes the amount of virtual memory used (VIRT), the amount of physical memory used (RES), the state of the process (D – sleeping on disk), and the CPU utilization of the process:

```
top - 11:55:20 up 120 days, 15:23,  4 users,  load average: 2.97, 1.17, 0.77
Tasks: 390 total,   1 running, 386 sleeping,   0 stopped,   3 zombie
Cpu(s):  0.0%us,  1.3%sy,  0.0%ni, 83.7%id, 15.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  49409336k total, 49219968k used,   189368k free,     2748k buffers
Swap: 14335992k total,  2760476k used, 11575516k free,    45088k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
30892 cs3214    20   0 96.6g  46g  260 D 10.0 97.7   0:24.07 infrec
```

i. (4 pts) Briefly explain what is going on from a virtual memory perspective!

Thrashing. At each recursive call, a new stack frame of size ~8KB is allocated and immediately touched, causing a page fault that will cause the OS to allocate physical memory. After 1 minute, all physical memory (48GB) was exhausted, at which point the machine started to swap memory to disk to satisfy the program's implicit requests. As a result, it became I/O bound (in the D – waiting for disk state, with very low CPU utilization). Linux's global replacement policy allowed the program to take up all available physical memory on the machine.

ii. (2 pts) How could this behavior have been prevented?

Put a limit on the amount of virtual addresses that can be taken up by the stack segment, i.e., limit the size of the stack and terminate programs that attempt to use more.

This is commonly done and is in fact the default configuration in Linux: a soft limit of, say 8192 kB, and a hard limit set to 'unlimited'. If a program suffers from infinite recursion, like this one does, it will fail quickly without bringing the machine into the state of thrashing. If there is a legitimate reason for using large stacks (as is the case for some programs), the user can use ulimit(1)/setrlimit(2) to explicitly increase the maximum stack size before running the program. After inquiring, I learned that our system administrators removed the default limit to accommodate an old & buggy version of a commercial Fortran compiler. A better approach is to have only those users who need to run this compiler remove the stack limit.

4. Networking & Servers (14 pts)

a) (4 pts) *Internet Design.* A key design principle of the Internet is sometimes referred to as "smart host, dumb network." Explain what is meant by that, using examples of specific design decisions!

Fundamentally, it means that the network's primary function is to forward packets from a source host to a destination host. Everything else is handled at the end points, including

- transport layer protocols such as TCP that implement connection establishment and reliable data transmission
- application code and application-level protocols
- metadata protocols such DNS, the domain name service that maps domain names to IP addresses

There are exceptions to this in practice (e.g. middleboxes) but those are implemented in a way that the user does not perceive their existence.

b) (4 pts) *IPv4 vs. IPv6.* Even though the shortage of IPv4 addresses has been known for almost two decades, and even though IPv6 was (more or less) finalized more than a decade ago, only a miniscule fraction of Internet traffic today uses IPv6. Describe 2 technical reasons for why this transition is going so slowly!

Possible technical reasons include:

- The inability of IPv4-only hosts to communicate with IPv6-only hosts due to a lack of embedding of the IPv4 address space in the IPv6 address space. As such, end host operators must assign a separate IPv6 address to each of their hosts, without necessarily gaining immediate benefits in return.

- The need to change applications to understand IPv6 addresses, as you have experienced in the project – even 2014 textbooks do not include the necessary information as you've seen. Some application protocols that embedded IPv4 addresses needed changes as well, e.g. ftp.

- The existence and widespread deployment of NAT as a solution for the common case in which no public IP addresses are needed.

For more details, see D.J. Bernstein (ca. 2002), "The IPv6 Mess", http://cr.yp.to/djbdns/ipv6mess.html

c) (6 pts) *Software Engineering Considerations.* Engineering *single process* network server programs such as your HTTP server exhibits unique software engineering challenges, whether they are multi-threaded or written using a single-threaded, event-based model. Describe 2 challenges unique to this particular model of server!

The key challenges are likely:

- Error recovery: if an error occurs, it must be handled without exiting the process. In a multi-threaded design, if an error causes a thread to exit, it must do so in a way that leaves critical global data structures in a consistent state.
- Resource reclamation: any resource leak (file descriptors, database connections, etc.) caused by a single connection has the potential to affect the entire server.

These challenges apply to any single-process design, be it multi-threaded or event-based. By contrast, multi-process design often have the option to exit() to recover from error situations, in which case the OS (often) ensures that no resources are leaked.

5. HTTP (20 pts)

a) (5 pts) CGI (Common Gateway Interface) is a way by which a HTTP server can delegate the generation of dynamic content to an external program. The tiny.c sample program, which many of you used as a starting point for implementing project 4, contains the following function that implements it:

```
void serve_dynamic(int fd, char *filename, char *cgiargs)
{
    char buf[MAXLINE], *emptylist[] = { NULL };

    /* Return first part of HTTP response */
    sprintf(buf, "HTTP/1.1 200 OK\r\n"); // modified from HTTP/1.0
    Rio_writen(fd, buf, strlen(buf));
    sprintf(buf, "Server: Tiny Web Server\r\n");
    Rio_writen(fd, buf, strlen(buf));

    if (Fork() == 0) { /* child */
       /* Real server would set all CGI vars here */
       setenv("QUERY_STRING", cgiargs, 1);
       Dup2(fd, STDOUT_FILENO);         /* Redirect stdout to client */
       Execve(filename, emptylist, environ); /* Run CGI program */
    }
    Wait(NULL); /* Parent waits for and reaps child */
}
```

Based on that code, sketch a CGI program in a language of your choice that responds to a request! The dynamic content it generates does not matter, but make sure that the HTTP/1.1 protocol is implemented correctly!

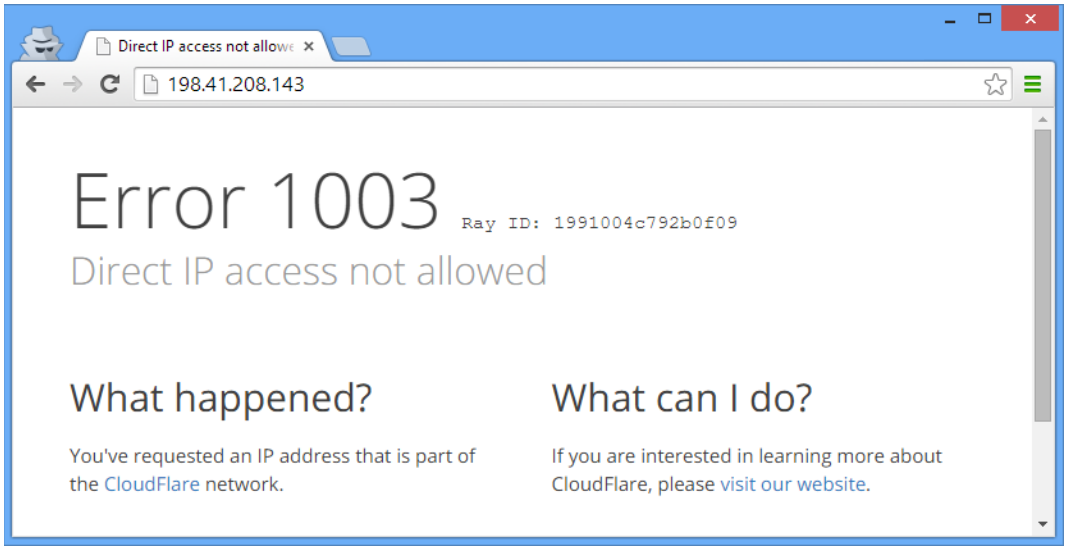
The CGI program inherits the open client connection as its stdout. The server has already written the 200 response and the `Server:` header. The CGI program must add a `Content-Length` header and add any remaining headers it wishes, then must output a CRLF and the body of the response, e.g..

```
#!/usr/bin/python

from sys import stdout as out
body = "Hello, World\n"
out.write("Content-Type: text/plain\r\nContent-Length: %d\r\n\r\n%s" %
(len(body), body))
```

Note that the question asked to sketch how CGI would work "based on the given code" – in real-life CGI (RFC 3875), there are variations: by default, the process forked will add headers, but no content-length header – instead, the parent will count how many bytes are in the body and add a `Content-Length` header field. An exception to this are NPH (non-parsed header script), in which case the CGI script takes full responsibility for the response processing. In HTTP/1.0, the script could simply exit and this would close the connection.

Some of you wrote just a function, but the problem clearly asked for a program, and it's clear from context (fork() + execve()) that a program is needed.

b) (5 pts) In TCP/IP, addressing is done by a combination of a globally unique IPv4/v6 address and a port number. Domain names such as www.reddit.com cannot be used directly; rather, programs first need to look up the corresponding IP address before calling `connect()`, e.g. 198.41.208.143. However, when one attempts to enter the IP address directly, the following ominous screen appears:



How does the destination server (here: www.reddit.com) know whether you typed in the hostname or the IP address in the browser, and why does it care?

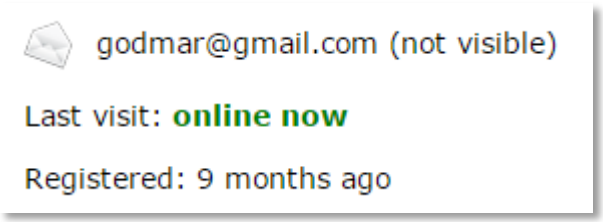
i. (2 pts) It knows it …

Because it is sent along in the `Host:` header of the HTTP request.

ii. (3 pts) It cares because …

The same IP address could (and probably is) used to host multiple websites with different domain names. CloudFlare likely does not have enough IP addresses to have a separate set for each of their many customers.

c) (5 pts) Some community websites implement a feature that lets users see which other users are currently viewing the site, i.e., are "online."



Based on your knowledge of the HTTP protocol and its connection management, how would you implement such a feature?

HTTP does not have a way for a server to contact a client and check its presence; there is also no protocol element that would announce to a server when the user has closed a web page or the browser has exited or crashed. Consequently, such features must be implemented using JavaScript code that periodically polls the server by requesting some agreed-upon URL. The server must take the absence of such client-initiated pings as a sign the user has left the page.

Side note: the DOM window also fires an 'unload' event when a page is torn down; an attached JavaScript event handler can send a message to the server right before the user navigates away from a page, but this event will not fire if the user disconnects because of a network disconnect or browser crash.

d) (5 pts) SPDY (pronounced (SPeeDY) is a protocol proposed by Google to replace HTTP/1.1. Two of its core features are (a) multiplexed streams that reuse a single TCP connection and (b) the ability of a server to respond to requests in any order. Describe the motivation for adding these features, particularly when compared to HTTP/1.1's persistent connections!

The use of a single connection is beneficial as it reduces connection overhead, which in particular reduces the amount of resources a server has to allocate for

each client – something very important to large scale providers such as Google. HTTP/1.1 already allows the pipelining of multiple requests and the retrieval of multiple objects over the same connection, but single connections are not used in practice since clients risk "head-of-line" blocking where requests a server could answer more quickly have to wait behind requests that take longer. HTTP/1.1 cannot allow those requests to "jump the line," but the out-of-order response feature in SPDY can, thus making it risk-free for clients to pipeline requests on the same connection.

6. Essay Question: You Be The Judge! (18 pts)

Some students were surprised to learn that they did not meet the basic requirements in the fork/join thread pool project when their code failed during grading, even though (they say) it did not fail during testing. Other students saw their code sometimes fail during testing, but were relieved to learn that it did not fail during grading.

Suppose you are hired as a UTA for next semester's offering of CS 3214. You are being asked to design and defend a sensible strategy for grading this project. Based on your knowledge of current computer architectures, multi-threaded programming techniques, as well as the capabilities and limitations of emerging or established tools, develop, describe, and defend a possible strategy!

Note: *This question will be graded both for content/correctness of your technical arguments (12 pts) and for your ability to communicate effectively in writing (6 pts). Your answer should be well-written, organized, and clear.*

A complete answer should discuss the nature of and reasons for intermittent failures in multi-threaded programs, which include latent bugs that manifest itself when certain environmental conditions are present.
Existing tools (e.g. race condition checkers) can be effective, but they suffer from false negatives and (in some cases) false positives. False negatives can occur if a bug is not triggered for a given input or the race condition is hidden by spurious happens-before relationships; false positives may occur due to limitations in the race-detection algorithm used or because advanced synchronization constructs are used. Students may also apply techniques to placate the tool, but which introduce atomicity violations. Finally, the tool itself may have bugs.
There are emerging tools for formal verification that can prove certain aspects of a program's correctness, but they are not ready for classroom use.
Repeated stress testing is an alternative; while it also suffers from false negatives, it does – by definition – not have false positives (i.e., correct programs that fail the provided tests).
Several other ideas are worth considering, including intentional variations in the environmental conditions under which a program runs (number of cores, core

assignment, background CPU load), or the use of deterministic multi-threading that would allow a replay of conditions under which a program failed.
A suitable strategy should take these facts into account and provide concrete suggestions for how to include them in a grading strategy.

Arguments that are less technically sound include attempts at ensuring a pristine and consistent environment; although this helps with reducing the variance observed in performance benchmarking, it tends to mask intermittent concurrency-related issues. Another idea was to average correctness scores and awarding partial credit if problems fail only so many times. Such a strategy is in my opinion not appropriate for crucial infrastructure such as a thread pool, for which basic tests should never fail.