Princeton University
COS 217: Introduction to Programming Systems
Fall 2008 Final Exam Answers

The exam was a 3-hour, open-book, open-notes exam.

Question 1 Part A

FALSE -- This would provide less flexibility than using the stack with call and return instructions. For example, you couldn't have one function being called from two different locations.

Question 1 Part B

TRUE -- Registers are typically the fastest kind of memory in the machine.

Question 1 Part C

FALSE -- Underflow, when not an attack, is a programmer error.

Question 1 Part D

FALSE -- The word size determines how much large a number can be used, regardless of which endianness is used.

Question 1 Part E

FALSE -- The registers are typically fixed when the processor's instruction set is designed.

Question 1 Part F

FALSE -- The wait system call harvests return values from child processes, and should be used as appropriate to avoid zombies.

Question 1 Part G

TRUE -- Memory fragmentation wastes memory, which can lead to slower programs.

Question 1 Part H

FALSE -- They are undefined for negative values, not for positives.

Question 1 Part I

FALSE -- Signals are used by the OS to inform the process about something.

Question 1 Part J

TRUE -- Page tables typically hold information about the page permissions, locations, etc.

Question 2 Part A

It will have several side effects: it will no longer initialize the space to zero, it will use a longer machine language encoding, and it will set the flags. The execution time may also be different, depending on circumstances, so if you said either faster or slower and gave a plausible rationale, it was accepted.

Question 2 Part B

See above.

Question 2 Part C

Best fit could reduce the amount of fragmentation generated, which could lead to more efficient use of memory.

Question 2 Part D

Best fit may take more time to scan the free list compared to first fit, and it may be the case that best fit leaves too little space for reallocating the buffer in the event the process calls realloc often.

Question 2 Part E

The page faults -- disk is a few orders of magnitude slower than RAM, so this would dominate the program's performance.

Question 2 Part F

Registers that a function must save before using, and then restore before returning.

Question 2 Part G

The caller-saved registers allow the called function a little scratch space without having to push/restore them. If the function needs more registers, it can save the callee-saved registers and then use them. Caller-saved also allows for greater performance if the caller can avoid keeping needed values in them before calling another function.

Question 3 Part A

The formal parameter that is passed to the function.

Question 3 Part B

A local variable whose value is eventually returned.

Question 3 Part C

It re-enters the loop if the current value of the parameter is greater than zero.

Question 3 Part D

If it fails the test, that means that the low order bit was not equal to zero. In other words, the parameter was odd.

Question 3 Part E

The parameter is odd, and its value is being added to the local variable.

Question 3 Part F

Brute-force answer: This function returns the sum of all odd numbers between zero and the value passed in to it. More elegant: If you think about what happens when you sum a bunch of odd numbers starting at 1, you end up with squares.

Question 4

The array commands should end with a NULL value after quota -- add it.

Need to call signal() to set the signal handler. Call signal(SIGCHLD, SigChildHandler) before the loop entry. Given that there was a correctly-written signal handler, this should have been a reasonable clue that having the parent call wait after each fork was not the desired goal. However, even that was accepted.

The fork system call does not take a parameter -- remove it.

The array args should be of type (char *) instead of just char – add the asterisk.

After a successful execvp, the new program is executing, so the test and printf do nothing -- since you can't indicate success, just get rid of the test and printf.

If you said that the fork command is not checked for failure, that was accepted only if you could explain how to fix the bug correctly. That fix, however, required identifying the problem that this case was not specified.

If you said that execvp was not checked for failure, you had to identify that this would then cause the child to also start forking processes, leading to a fork bomb.

If you said that sleep should just be called once for 10 seconds, you have actually made the program incorrect - the sleep call them gets interrupted by SIGCHLD almost immediately, and almost never sleeps for 10 seconds.

Question 5 Part A

There were several possible scenarios that came up during class, some of which were a better fit for this question than others.

One case is as follows:
a) The process frees a large chunk of memory in the middle of the heap.
b) The malloc library can't give it back to the OS, since it's not at the end of the heap.
c) The OS swaps those pages out to disk due to memory pressure.
d) The process asks for a large chunk of memory, and the library gives it the free chunk.
e) When the process starts using the memory, the OS brings back the useless pages from disk.

In a scenario where the library could have told the OS that those pages are useless, the OS could have just scrapped them, and if the process uses them again, the OS could just fill a page with zeros instead of reading it from disk.

Another scenario, which is related but not quite on target, is to notice that storing free list pointers in the free blocks themselves causes those pages to be accessed frequently just for list traversals. When that happens, the OS thinks that memory is being used, which needless increases memory pressure. Worse, though, would be if those pages somehow got swapped out -- then the next time the free list is traversed, the malloc library would suffer a big performance hit as those pages got swapped back in. This is why modern allocators typically keep the free list in separate memory than the data blocks itself. Note that this explanation is not a perfect answer to the question, since these decisions could still be made with a system that used brk/sbrk.

Another possible answer that would get partial credit is to observe that a program might want to read a very large file but not modify the contents of that file. If the program uses malloc, a lot of memory gets used to copy something from disk, and if there's memory pressure, the OS has no way of knowing that the data isn't modified from what's already on disk. Those pages then get written back to swap. Instead, with memory-mapped files, the OS could just give up those pages, since they can be pulled back from the file itself when needed. In this scenario, the bad interaction with brk/sbrk occurs because the address space for memory-mapped files has to come from somewhere, and it's typically placed between the heap and stack. While stacks are more or less expected to be contiguous, there's nothing that says the heap as a whole has to be in contiguous memory, so if a malloc library only uses brk/sbrk, it may run out of address space sooner than if it used
mmap.

Question 5 Part B

Some stack-smashing attacks rely on placing instructions onto the stack, and then executing those instructions. So, if you were to disable instruction execution on stack pages, you could defeat stack-smashing attacks that used that exact mechanism. However, that would not be enough to defeat stack-smashing attacks in their entirety.

The attacks would simply have to find another source of convenient instructions and use them. One possibility is the program itself – if there are already routines in the program that can cause damage, change passwords/permissions, or otherwise allow arbitrary commands, then the necessary arguments can be placed on the stack, and the relevant piece of code can be used as the return address.

In the event that the program itself may not have the right code, the other approach is to use code in the C library, known as a return-to-libc attack. Of particular interest would be something like the system() function, which allows arbitrary commands to be executed. So, the command is placed on the stack, and the return address is set to the entry of system().

Question 5 Part C

One simple approach is to place known values in the header and footer of each buffer allocated. These values could be fixed, or they could be calculated based on the location and/or size of the buffer. The idea is that these values could be checked, either periodically, or on calls to free or realloc. If these values have changed, it means that there's been an overflow/underflow. In the case that these values are checked only during free or realloc, the overhead is constant per call. In the event that all of these values are checked periodically, the overhead grows with the number of items allocated. This approach does not catch all possible overflows/underflows. For example, if the overflow writes starting at N bytes before or after the buffer, then the header/footer must be more than N bytes large to catch the overflow/underflow. In general, any overflow or underflow that is discontiguous and larger than the size of the header or footer may be missed.