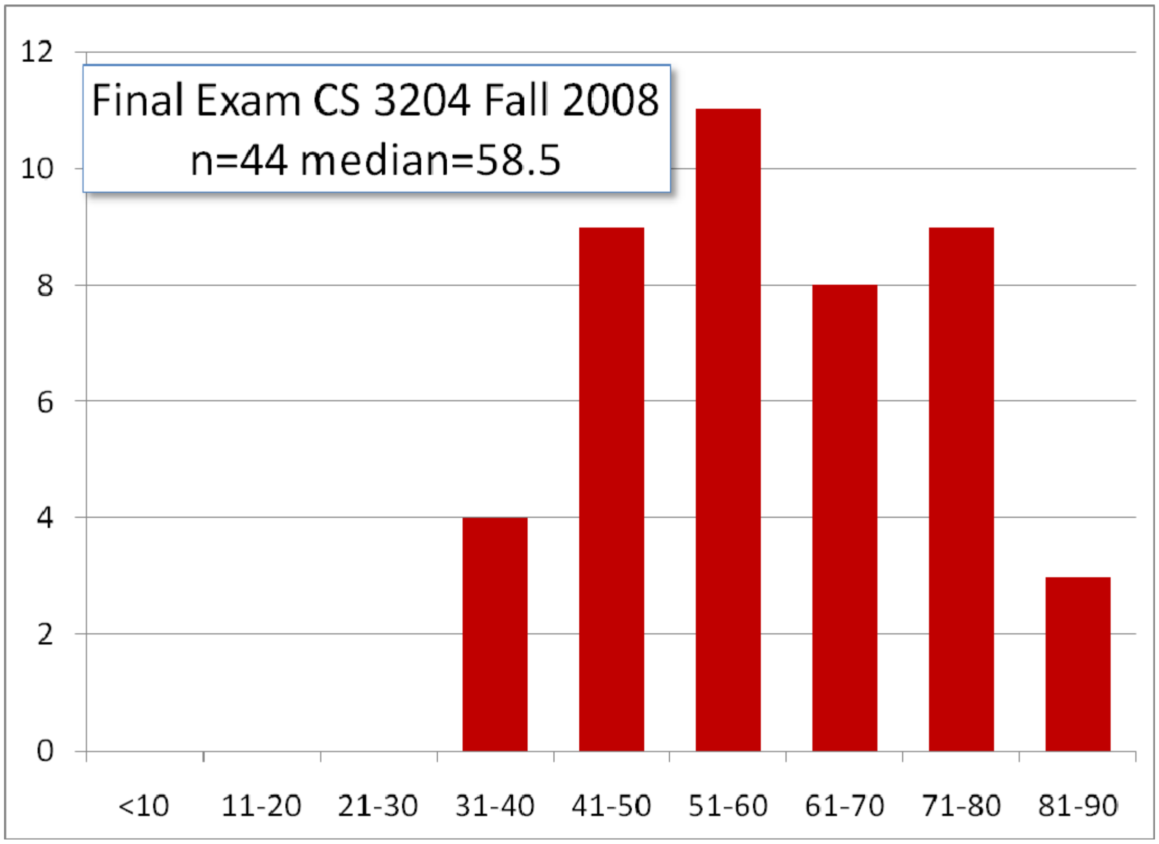
# CS 3204 Final Exam Solutions

44 Students took the final exam. The table below summarizes the results and who graded which problem. Exams are available for review in my office.

| **Problem** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | |
|---|---|---|---|---|---|---|---|---|
| **Possible** | 16 | 18 | 18 | 20 | 12 | 12 | 4 | 100 |
| **Average** | 12.0 | 11.0 | 5.9 | 10.6 | 9.3 | 7.3 | 4.0 | 60.0 |
| **Std Dev** | 3.3 | 4.4 | 3.9 | 4.4 | 2.1 | 3.5 | 0.0 | 14.1 |
| **Min** | 5 | 4 | 0 | 4 | 4 | 2 | 4 | 35 |
| **Max** | 18 | 18 | 15 | 19 | 12 | 12 | 4 | 84 |
| **Grader** | Peter | Godmar | Peter | Godmar | Peter | Godmar | Godmar | |



*Solutions are shown in this style.*
Grading comments are shown in this style.

## 1 Security (16 pts)

a. If you ran the following program on a recent version of Linux:

```
// print_addr_of_ac.c
#include <stdio.h>
int
main(int ac)
{
        printf("%p\n", &ac);
}
```

Then you might see output such as the following:

```
gback@ghestal [19](~/tmp) > ./print_addr_of_ac
0xbf95eac0
gback@ghestal [20](~/tmp) > ./print_addr_of_ac
0xbf90da70
gback@ghestal [21](~/tmp) > ./print_addr_of_ac
0xbf8579b0
gback@ghestal [22](~/tmp) > ./print_addr_of_ac
0xbfaaec10
gback@ghestal [23](~/tmp) > ./print_addr_of_ac
0xbfceae50
```

i. (3 pts) Which specific security-related technique explains this output?

*The output is caused by address space randomization (ASR). Rather than placing a new process's stack always at the same address, the Linux operating system starts the stack of each program it executes at a random address, thus resulting in different values for the address of main's 'ac' argument.*

An extremely easy question if you attended the lecture where I demoed this.

ii. (3 pts) What kind of attacks is above technique designed to prevent or make more difficult?

*It is intended to prevent buffer overflow attacks, because the attacker no longer knows the virtual address at which a given stack frame is allocated, making it more difficult to divert control to code injected by the attacker.*

b. A 'rootkit' is a set of software tools used by a hacker after gaining (unlawful) access to a computer system in order to conceal the altering of files and other activities. Many rootkits include code that augments or replaces code in the running kernel.

i. (4 pts) Explain why an attacker may choose to target the kernel (rather than, or in addition to, replacing user or system programs.)

*The entire kernel is part of the trusted computing base (TCB) of typical operating systems, meaning that control over the kernel provides control over the entire security system, including all authentication and access control mechanisms. It also provides an ideal means for interception, since all programs use the kernel to communicate with the outside world or each other.*

ii. (6 pts) Suppose you wanted to implement a mini-rootkit for Pintos, which allowed an attacker to hide a file "/cracked" in the Pintos file system such that only the attacker would see it. Which of the system calls you implemented in this semester would you have to patch, and how?

*You would need to change (at least) the readdir(), create(), unlink(), mkdir(), and open() calls. Each call would have to act differently when a user program controlled by a legitimate user or system admin executes. The readdir() call would have to be patched such that "cracked" is no longer returned as a directory entry when listing the contents of "/", and the open() call would have to be changed such that open("/cracked") will not succeed initially. Create() and mkdir() would have to be changed such that create("/cracked") or mkdir("/cracked") does succeed, but does not actually overwrite "/cracked" – maybe by creating a file or directory with a different name. Subsequent calls to open("/cracked") should open that file instead. Unlink("/cracked") must fail, except when an alternative version of "/cracked" was created.*
*A fully complete implementation would probably also need to modify the filesize() and tell() system calls so that the hidden entry in the root directory could not be inferred from the size of the root directory file or from how the offset in the root directory changes while iterating over its contents via readdir().*

## 2 Implementing Kernel-level Threads (18 pts)

The Pintos kernel, though fully preemptive, currently supports only 1 thread per process. In this question, you are asked to discuss how to implement support for multi-threaded processes in Pintos.

a) (3 pts) Describe 1 application scenario that would particularly suffer from the lack of support for multiple threads within one process.

*Support for multiple threads within a process is useful for many application scenarios, such as scientific numerical applications, which may benefit from distributing work across multiple cores or processor.*

I accepted pretty much every scenario that uses multiple threads within one process; but not scenarios that either use multiple processes (such as "make –j 4") or examples where lack of multi-threading probably doesn't hurt much – such as disk device bound tasks such as enumerating the files on a disk.

b) (7 pts) Describe which data structures you have to introduce, and which you'd have to change to implement kernel-level threads following the 1:1 model used in Linux or Windows. Describe which kinds of control blocks you would need to introduce, their relationships, and which changes, if any, you would need to make to the ready queue implementation. Draw a sketch if needed.

*To implement multiple kernel-level threads per process, you need to separate the thread control block (TCB) from the process control block (PCB). The existing 'struct thread' can continue to be used for the TCB, but all fields that refer to data that is shared by threads within a process will need to be placed into the PCB. Each TCB would contain a pointer to the PCB of its containing process. The data contained in the PCB includes the file descriptor table, the page directory, supplemental page table, current working directory, etc. Fields that are used by the scheduler, such as the priority or the linkage elements to keep a thread in the ready queue don't have to be changed, and the ready queue also should not need any modifications.*

c) (2 pts) To start new threads, you would need to implement a function 'thread_create()' that user processes could use. Would this function be implemented as a user level library function or as a system call?

*Since the problem asks you to implement kernel-level threads, and not user-level threads, it must be a system call so the kernel can create a new thread and a new TCB.*

Frequent problems in parts a) and b) included a confusion about the concepts of user-level threads (which are implemented as a user-level library) and kernel-level threads (in which threads are first-class entities within a process, supported by the kernel.)
Some students confused kernel-level threads with kernel threads (those that execute within the kernel's address space their entire life. The latter assumption could be ruled out since Pintos already supports these.

d) (3 pts) Would your data structures and your implementation of the file system related system calls have to change (e.g., open(), read(), write(), etc.)? If so, say how. If not, say why not.

*All threads in a given process share the same set of file descriptors. The file descriptor table therefore would need to be placed into the PCB, as outlined above. In addition, the file descriptor table is now subject to concurrent accesses by multiple threads, so locks must now be used when allocating and deallocating file descriptors. No other changes should be needed.*

e) (3 pts) Consider that multiple threads may fault within the same page. Would your data structures and your implementation of virtual memory/paging have to change? If so, say how. If not, say why not.

*All threads in a given process share the same address space. The mapping structures describing this address space (such as the hardware page directory and supplemental page table) therefore would need to be placed into the PCB, as outlined above. In addition, these structures are now subject to concurrent accesses by multiple threads, so locks must now be used when inserting and removing page table entries. If two threads fault on the same page, you would need to ensure that only one of them performs the actual I/O to fault in the page while the other(s) wait.*

## 3 Virtual Memory (18 pts)

a) (12 pts) ***Debugging Virtual Memory.*** During project 3, some of you had trouble getting lazy loading, stack growth, and eviction to work correctly. In many cases, the behavior of a user program could have given you an indication of which aspect of your virtual memory implementation was buggy. Consider the following test program:

```
#include <stdio.h>
#include <assert.h>

int buf[2048];
char * progname = "TestVM";

int f(int n) {
   char tmp[1024];
   return n > 1 ? n * f(n-1) : 1;
}

int
main(int ac, char *av[])
{
    printf("%s: starting\n", progname);  // [1]
    printf("10! = %d\n", f(10));        // [2]
    assert(buf[0] == 0);                 // [3]
    progname[0] = 'X';
    printf("test failed (should have been terminated)\n"); // [4]
}
```

The correct output, which you can verify on Pintos or Linux, is:

```
TestVM: starting
10! = 3628800
```

Followed by a message such as `Segmentation fault (core dumped)`.

For each of the following outputs, explain in which part of your VM implementation you would be looking for bugs! Be specific!

i. (3 pts) The program outputs "`(null): starting`" in line [1]

*This output would indicate that the initial value of the global variable 'progname' was not available to the program, indicating a bug in the code that reads global initialized data from the executable into memory.*

ii. (3 pts) The program crashes with an unhandled page fault before reaching line [2].

*The call to f() recurses 10 levels deep, allocating at least 1024 bytes at each level. This code therefore exercises stack growth beyond the initial 4KB stack page; faulting here likely indicates a bug in the stack growth logic.*

iii. (3 pts) The program fails the assertion on line [3].

*Global uninitialized variables in C programs must be zero. If they are not, you likely have a bug in the code that zeros out memory when faulting in uninitialized data (pages that are either partially zero or all zeros).*

iv. (3 pts) The program reaches line [4] (it should have been terminated after line [3]).

*String literals such as "TestVM" are placed in a read-only section, which becomes part of the text segment, which is mapped read-only. Attempts to write to the first element of this string must therefore fail. If they are allowed, you may have a bug in the code that maintains and sets the permissions for pages you map during a page fault.*
*Note: don't confuse 'progname', which is a global, writable pointer variable, with "TestVM", which is a zero-terminated read-only array of characters.*

Note that none of the failures can be attributed to eviction – this simple, small test isn't going to cause any pages to be evicted.
The four cases a) through d) included very specific clues about where the error might lie; for full credit, you had to be specific enough to show that you understood the relationship between a program's variables and the action an OS takes to ensure their extent and values.

b) (6 pts) ***Extreme Virtual Memory***. Virtual Memory is said to provide the illusion that a process has access to all of a machine's memory, when in fact each process shares memory with other processes on the system. Your CS 3204 teammate proposes to implement this idea in an (almost) literal manner: whenever an access by a process to a user virtual address causes a page fault, the OS will service the fault, allocate physical memory, and create a suitable virtual-to-physical mapping at that address. Discuss the merits of this idea.

i. (3 pts) What would be a benefit of this approach?

*There would be no need for system calls such as sbrk() by which the user program asks the OS for virtual memory, possibly simplifying the design of user-level memory allocators. Stack growth also would work automatically and would no longer need treatment as a special case.*
*Finally, programs would continue where they would otherwise segfault, possibly increasing availability.*

ii. (3 pts) Why do real virtual memory implementations not exploit this seemingly cool idea?

*With this implementation, user programs would be much less likely to segfault when performing out-of-bounds accesses or dereferencing uninitialized pointers. As a result, many errors would go undetected and instead lead to the silent corruption of the program's data.*

Note that "Extreme Virtual Memory" does not violate inter-process isolation or protection. Processes will not have access to kernel data nor will they obtain access to other process' data using the proposed approach. The proposed design affects only how an individual process negotiates with the OS about which parts of its address space are mapped and which ones are not. Some answers implied an overhead/performance trade-off, but I cannot see one (except for wildly misbehaving programs, in which case Extreme Virtual Memory might lead to large supplemental page tables – but this may also occur if a process first asks the OS for that virtual address space before accessing it).

## 4 File Systems (20 pts)

a) (9 pts) ***Filesystem Recovery***. To boot Pintos on real hardware, a roughly 4MB disk image (representing a continuous region of sectors) must be copied to a USB flash device via the Unix 'dd' command. Some students issued that command wrongly such that the device to which the image was copied was their internal hard disk, which overwrote the first 8192 sectors of that disk. In this question, you should discuss recovery options and explain what, if anything, these students may be able to recover under different assumptions.

i. (3 pts) Suppose the internal hard disk was formatted using your project 4 Pintos file system (and assume it contained useful data. If you didn't complete project 4, answer this question as if you had). State your assumptions if necessary.

*Unless you changed it in your implementation, the Pintos file system allocated the freemap file's inode and the inode of the root directory at the beginning of the disk, and the freemap and root directory files themselves right after. These would be lost in the scenario described. In addition, since any sector on the disk may contain an inode, it would be very difficult to discern for any recovery program which sectors contain inodes and which contain file data or are unused. Most likely, little or nothing useful would be recoverable.*

ii. (3 pts) The internal hard disk was formatted using the Fast Filesystem (FFS) discussed in class, with multiple replicated superblocks, and with the disk divided into multiple cylinder groups, each containing its own inode table and file data block section.

*In this scenario, probably only the first cylinder group is wiped out. The replicated superblock should allow a recovery program to find the locations of the other cylinder groups. Within each cylinder group, it should be possible to recover the files whose inodes are allocated in the corresponding inode table and whose data is allocated within one of the groups not affected. As a result, many if not most files should be recoverable.*
*If the root directory (or a directory close to the root) was allocated in the cylinder group(s) that were wiped out, a fair amount of manual effort may be required to place the restored files from lost+found back into the file system hierarchy.*

Note that cylinder groups do no longer have any significance since modern disks hide the physical layout of the disk and instead provide a linear addressing mode. I gave full credit if I could tell that you understood the general layout of a Unix-style file system and its implications on recoverability.

iii. (3 pts) Suppose the internal hard disk were using Sun's advanced ZFS file system.

*ZFS uses physical replication of blocks (so-called ditto blocks), with each block being replicated at least twice. ZFS promises to survive the loss of 1/8 of the platter, so if one is to believe ZFS's developers, all data on this disk should be fully recoverable if only 8192 sectors at the beginning are wiped out. In addition, it may be possible to retrieve old snapshots of the data located at these sectors.*

ZFS can be configured for different degrees of replication (by default, only metadata is replicated – so we cannot be certain that all data could be recovered). For full credit, I expected you to know that physical replication and/or snapshots are advanced features of modern file systems, as we discussed in class.

b) (6 pts) **File System APIs.** Many file systems in the Unix heritage represent directories as files. However, they do not allow processes to access these directory files using the read() and write() system calls. Instead, the directory contents are read using a specialized readdir() call, and are updated as part of file system operations. Explain the rationale for this design!

i. (3 pts) write() is not supported because:

*Writing to a directory file places new entries into the directory. If user programs were allowed to do that, it would be much more difficult, or even impossible, to enforce the consistency guarantees a file system makes, such as that directory* *entries within a directory are unique, that file creation is atomic, and that directory entries always point to allocated inodes.*

ii. (3 pts) read() is not supported because:

*To interpret the data read from a directory file, knowledge of the internal directory format is necessary. This information is part of a file system's internal design and implementation, which should not be exposed to user programs.*
*Note that this argument is an encapsulation argument, not a protection argument. In fact, older versions of Unix (which supported only a single file system type!) did leave the parsing of directory contents to user programs.*

An easy question if you attended the class in which I answered this question.

c) (5 pts) **File Allocation Strategies.** NetApp, a company specializing in storage systems that exploit high-performance RAID hardware, uses a file system which they call "WAFL." WAFL stands for "Write Anywhere File Layout."

i. (3 pts) Explain what "write anywhere" means and why "writing anywhere" is beneficial when it comes to a file system used for RAID arrays.

*"Writing anywhere" refers to where new versions of data or metadata is written. Unlike in traditional file systems, in which each file's data is allocated to a sector once and then updated in place, "write anywhere" assigns a new sector number whenever a new version of the data is written, and then updates the metadata structures that record where the new data is located.*
*In a RAID system, this approach avoids "small writes" (and the associated small write penalty) because writes can now be arranged to always span an entire stripe.*

Many answers implied that WAFL can simply choose from any free block/sector when allocating a new block or sector, and thus can spread writes out over multiple disks to increase parallelism. I gave partial credit for that answer. The hint that WAFL was designed for RAID systems, and the fact that RAID systems incur a small-write penalty when in-place updates must be done, combined with the in-class discussion of avoiding in-place updates as an advanced feature of modern file systems, should have led you to the right insight about the nature of NetApp's innovation.

ii. (2 pts) Would "write anywhere" also be beneficial when it comes to single-disk file systems? Justify your answer!

*Yes, it would also benefit single-disk file systems because it can make writes more sequential, reducing the number of seeks. In fact, 'online defragmentation' is now being considered in some modern file systems, such as ext4.*

Some answers implied that "write-anywhere" would increase fragmentation, not reduce it.

## 5 The Future (12 pts)

Operating systems are a rapidly developing area that is strongly influenced by short term and long term technology trends. In this question, you are asked to discuss how some of these trends may affect the design of operating systems. For each question, give 2 examples of how techniques learned in this class will fare under such changes. Choose 1 example of a technique that will be in need of major overhaul or may even become redundant due to a trend. Choose 1 example of a technique that will remain relevant. You only need to name the technique, but be specific enough so that the relationship to the technology trend is clear.

a) The trend towards solid state drives, related to techniques for designing file systems and buffer caches.

  i. (2 pts) In need of overhaul or may become irrelevant:

*Techniques designed to reduce or minimize seeks, such as defragmentation, may become less relevant. If the speed with which solid state drives operate approaches the speed of RAM, the role of buffer caches itself may change.*

  ii. (2 pts) Likely to stay relevant:

*Most file system related techniques will probably stay relevant. Examples may include techniques to speed up or reduce the space complexity of the metadata mapping, or techniques for providing consistency (as a software failure is still possible between writes to different sectors).*

b) The trend of physical memory sizes to outpace the requirements of typical desktop applications, as it relates to virtual memory techniques.

  i. (2 pts) In need of overhaul or may become irrelevant:

*Page replacement algorithms, and any technique that optimizes for frequent replacement, will likely become less relevant. So will techniques that perform mid-term scheduling (e.g., which suspend processes when there is not enough memory). If the trend continues, swapping to swap space itself may no longer be supported on all systems.*

  ii. (2 pts) Likely to stay relevant:

*Virtual memory techniques such as stack growth, memory-mapped files, and on-demand paging (possibly with increased prefetching) will likely be unaffected by this trend. So will the use of translation to provide multiple address spaces and protection domains.*

c) The trend to place multiple cores on a single chip, as it relates to scheduling and synchronization techniques.
   i. (2 pts) In need of overhaul or may become irrelevant:

*All single-processor algorithms (uniprocessor lock and semaphore implementations, enabling critical sections by disabling interrupts, etc.) will be less relevant.*
*So would be schedulers that cannot support multiple CPUs in some way; although most multi-CPU schedulers divide the ready queue into multiple, per-CPU queues that are then scheduled using a single-CPU scheduling algorithm.*

   ii. (2 pts) Likely to stay relevant:

*Certainly, all multi-processor techniques (spinlocks, use of atomic instructions, etc.) will remain relevant since multi-core processors operate almost indistinguishable from multi-processor systems. Barring a breakthrough in new parallel programming model, all techniques that exploit the fundamental abstractions provided by locks, semaphores, and monitors will stay relevant as well.*

## 6 Essay Question: Synchronization Trade-Offs (12 pts)

d) (12 pts) Describe the trade-off between granularity of synchronization (fine-grained vs. coarse-grained), overhead and performance, and the potential for deadlock in the area of operating systems. Describe situations in which this trade-off matters and which factors you may need to take into account when choosing or designing a file system.

***Note:*** *This question will be graded both for content/correctness (8 pts) and for your ability to communicate effectively in writing (4 pts). Make sure you define the trade-off clearly, and elaborate on its meaning and consequences. Your answer should be well-written, organized, and clear.*

*Systems that support multiple, concurrent threads or processes must provide means of synchronizing the activities of those threads. This synchronization ensures correct results even when these threads execute simultaneously on multiple processors or cores, or under the regime of a preemptive scheduler on a single CPU. A common method of synchronization is the serialization of access to shared data structures using locks or monitors.*

*In a fine-grained synchronization approach, a single lock protects only one or a few shared data structures. This approach maximizes the potential for concurrency and can lead to increased performance when multiple computing elements are available. However, repeatedly acquiring and releasing locks may incur higher overhead as well, particularly for hotly contended locks.*

*A coarse-grained approach, on the other hand, assigns only one or a few locks to protect a larger number of data structures, resulting in a simpler design with less overhead. On the downside, coarse-grained synchronization may introduce unnecessary serialization, resulting in a loss of parallelism and subsequently performance.*

*Fine-grained synchronization also makes the occurrence of deadlocks more likely since it increases the likelihood of a process holding one lock while waiting for another, which is one of the necessary conditions for deadlock to occur.*

Some answers implied that there is a trade-off between safety/correctness and performance; this is not a generally accepted view. Code must first and foremost be correct and safe; then we'll consider its overhead and performance properties.
The addition "when choosing or designing a file system" was a copy-and-paste mistake. Since I hadn't underlined it, I accepted both answers that included a discussion of how granularity of synchronization applied to file systems and those that did not. The change from a single global lock in project 2 to a fine-grained approach in project 4 provided a good example for this discussion.

## 7 Race Condition Checker Survey (4 pts)

There are no right or wrong answers in this question, but credit is given for participation (at least answering parts a) to d)).

I'd like to thank everybody for providing feedback on the use of the race condition checker.