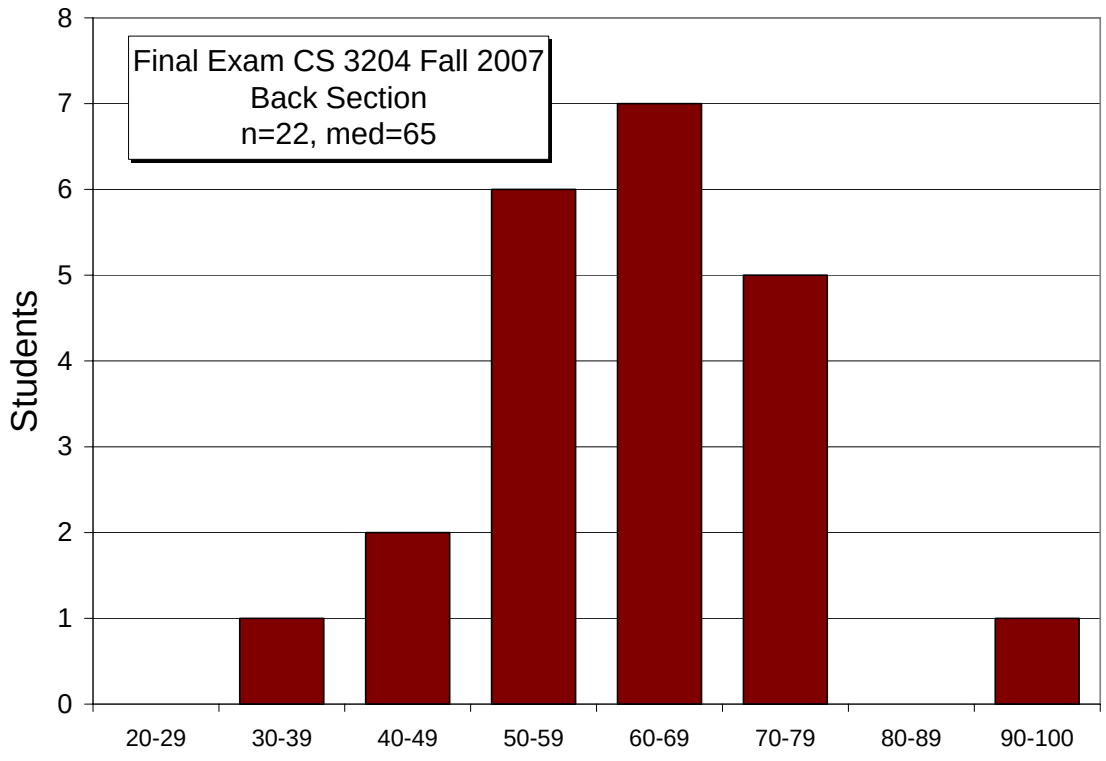
CS 3204 Final Exam Solution

22 students took the final exam. The table below summarizes the results. Exams can be picked up in my office.

Problem	1	2	3	4	5	Total
Median	8	16	14	13.5	13.5	65
Average	7.3	15.1	14.0	12.3	14.3	63.0
Std Dev	2.0	5.0	4.1	3.7	5.0	13.0
Min	2	5	4	5	6	39
Max	10	22	20	19	23	91
Out of	10	22	20	20	28	100
Grader	Godmar	Jai	Jai	Xiaomo	a,b: Xiaomo c: Godmar	



Solutions are shown in this style.

Grading Comments are shown in this style.

1 Failure-Oblivious Computing (10 pts)

If a program performs an illegal memory access, the operating system usually terminates the program. Recently, some researchers have proposed the idea of "failure-oblivious computing." A failure-oblivious system continues despite faults such as illegal memory accesses. Instead, read accesses read synthetic (made-up) values, whereas write accesses are ignored entirely.

a) (5 pts) Explain how you would implement this technique in the Pintos kernel. Which function would you have to change and how? Be specific about how you ensure that the faulting program can continue to make progress after the fault. Address both the read and the write case.

The page fault handler (page_fault in exception.c in Pintos) needs to be changed to examine the fault, manufacture a read value if needed, skip the instructions, and resume execution at the next instruction. The following pseudocode demonstrates this idea:

```
void page_fault(struct intr_frame *f) {
    ….
    instruction faulting_ins = ins_get_instruction_at_pc(f->eip);
    if (ins_is_read (ins))
        f->{ins_targetregister(ins)} = manufactured_value();
    f->eip += ins_length(ins);
    return;
}
```

In addition, all system calls that expect pointers to user-provided locations (read, write, open, remove, etc.) would need to be changed to simply return (and not terminate the process) if the pointer being passed is invalid.

You needn't mention the system call handler part for full credit. You needed to mention how values would be manufactured (by directly changing the state of the user process) and that in order to guarantee progress, execution must skip the instruction (otherwise, if you return as now, the instruction would simply fault again.)
Many here were confused about the distinction between the page fault handler and the system call handler. Please note that the problem was concerned with memory accesses, not read()/write() system calls. Maybe I should have used the terms loads and stores.

b) (5 pts) Discuss the merits of this technique. Name, briefly, 1 argument in favor of it and 1 argument against it.

As you can imagine, the idea is controversial. The main argument in favor of it is that it can increase availability: instead of crashing, a program may provide reduced service – in the case of a server program, maybe it cannot handle one piece of malformed input, but will have recovered in time for the next.

The primary counterargument points out that hiding such failures could lead to disastrous results if manufactured values are subsequently used in computations

with real side-effects, such as transferring money between accounts. It may also lead to waste of resources by stretching out ultimately failing computations.

Some of you pointed out that if ignored, errors may go undetected. I should point out that the actual implementation of failure-oblivious computing will still log illegal accesses to inform the programmer.
Some of you pointed out that it's better than a kernel panic, but that's not the alternative: an illegal memory access by a user program should never crash the kernel. Finally, note that protection is not compromised: illegal writes are ignored, which means that no other data to which a process would not normally have access can't be altered even if this idea is used.

2 Shared Memory (22 pts)

Most OS provide some form of shared memory capability to their user processes. For instance, POSIX's `int shm_fd = shm_open(name,…)` system call can be used to obtain a file descriptor, based on a name, which can then be mmap'ed into the processes' address spaces. If two processes use shm_open using the same name, they should see the same data.

Suppose you added an implementation of shared memory to the virtual memory manager you implemented in project 3.

a) (10 pts) You would need to provide a system call

```
bool mmap_shared(int shm_fd, void *buf, size_t len);
```

to make a shared memory area visible in the user's address space. `shm_fd` in this example is an open file descriptor pointing to the shared memory object, whereas `buf` is the user virtual address at which the user process wishes to access the shared area. Describe how you would implement this system call. You may refer to specific design in project 3, or describe the solution in general terms. You may assume that shm_open has already been implemented. Describe which data structures you would have to update, and what checks you needed to perform.

mmap_shared needs to:

- *Check that shm_fd is valid and came indeed from shm_open.*
- *Check that the virtual address range from buf to buf+len isn't already used, e.g., that there are no entries in the page table for that range.*
- *Add entries to the page table that refer to the shared object such that subsequent accesses will create mappings to the frames in which the shared object is located.*

For the remainder of this question, suppose the physical frames holding shared memory are subject to page frame reclamation. Below, describe what actions would be necessary to evict such a page frame's content. Assume the x86 architecture.

b) (4 pts) How would you accurately determine whether a shared memory page has been recently accessed? What changes, if any, would be needed to your page or frame table designs?

We can now have multiple page table entries (in multiple processes) point to the same frame table entry. Therefore, when determining recent access, we must check all access bits in all page table entries referring to the page frame. Likely, this will require expanding the frame table data structure: instead of having a single back pointer to the (supplemental) page table entry pointing to it, you would now need to keep track of a list of page table entries.

For complete credit, you needed to explain how the access bits are accessed when a frame is evicted – by determining the referring page table entries.

c) (4 pts) How would you ensure that the TLB is kept consistent with the entries in the hardware page directory/tables after a shared memory frame has been evicted?

To ensure TLB consistency, all page table entries referring to an to-be-evicted frame must be cleared before evicting it. If any of the referring page table entries are part of the current process's hardware page table, the TLB also has to be flushed to evict now stale entries. This is done in Pintos via pagedir_clear_page().

The question specifically asked about TLB consistency, so for full credit, you needed to mention that the TLB may need to be flushed if a page table entry changes.

d) (2 pts) Where you would store evicted shared memory frames?

Evicted shared memory frames would need to be stored in swap space since they are not directly related to frames.
Alternatively, if you assume that mmap_shared works like mmap(MAP_SHARED) in Unix, you could also state that they would be written to the backing file.

e) (2 pts) Is it necessary to determine whether a shared memory page's content is dirty before evicting it? If not, why not? If so, how would you do it?

Both answers are possible:

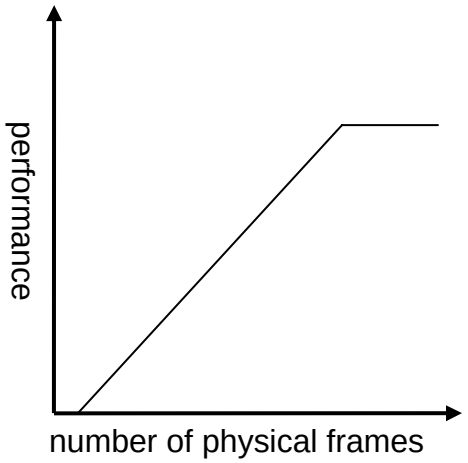
- *It's not necessary to check the dirty bit since the data isn't anywhere on disk, so must be written regardless.*
- *It may be useful to check, but only if the page has been evicted before and you keep track of where it was evicted to the last time. In that case, as in*

the case of access bits, the dirty bits of all referring page table entries must be checked.

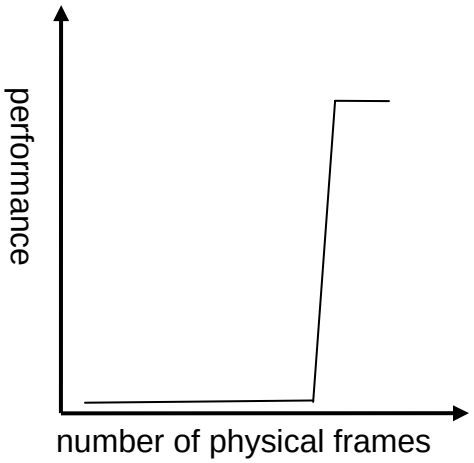
3 Memory Performance (20 pts)

a) (8 pts) Recently, researchers have been working on kernels that can support the dynamic variation of physical memory at runtime. Combined with hardware support, machines running those kernels have the ability to turn physical page frames on and off. To turn a frame off, its contents have to be evicted to disk first. Below are two graphs that show application performance vs. the amount of physical memory that is currently turned on.

A.



B.



i. (3 pts) What can you infer about the character of the workload used in scenario A? (Be sure to discuss the workload, not just the graph.)

The workload shows a regular access pattern with good locality when it comes to its virtual memory accesses. Adding more memory leads to a reduction in the page fault rate.

ii. (3 pts) What can you infer about the character of the workload used in scenario B?

This workload shows poor locality. For instance, the access pattern could be irregular, fooling the page replacement policy of the OS.

iii. (2 pts) Why do both curves flatten out once the number of physical frames exceeds a certain threshold?

At the threshold, all data is kept in physical memory – that is, the OS can assign enough page frames to hold the applications' working set in memory. Therefore,

the performance of virtual memory is equivalent to that of physical memory and any memory-related performance bounds are removed.

b) (4 pts) Some versions of Unix support a system call mlock(2), which allows a user to instruct the operating system to pin down (and not evict!) a particular virtual memory page in physical memory. The number of pages a particular user can lock in this manner is subject to an administrator-defined limit.
Say why the number of pages must be limited in this manner!

If not regulated, a user could drastically reduce the number of physical page frames that is left for other applications, leading to reduced performance and, in the worst case, thrashing.

c) (4 pts) (multi-oom) Consider the multi-oom test from project 2. To recap, multi-oom spawns an identical copy of itself as a child process, then waits for this child to terminate. multi-oom terminates if the child process could not be spawned for any reason. It records the depth of the chain of child processes it was able to create.

What would happen if you attempted to run multi-oom on a system that supports page reclamation, such as your Pintos project 3 kernel? Consider all system resources!

Once all user memory page frames are used, the system should continue spawning children by evicting pages from other processes. Eventually, one of two things will happen: the system could start thrashing, or, since creating a new process also always allocates kernel memory that may not be subject to paging, the kernel may run out of kernel memory (as it did in project 2).

d) (4 pts) The Linux kernel uses the buddy allocator scheme to manage its physical page frames, even though this allocation scheme incurs much higher internal and external fragmentation than, for instance, a first-fit or best-fit allocator would.
Give 2 reasons for why it is used in this situation despite these disadvantages!
   i. Reason #1:

A buddy allocator is fast: it needs to check at most n lists (n=10 in Linux's case) to find a free block or determine that none is available.

   ii. Reason #2:

Internal fragmentation does not matter here since the virtual memory hardware does not support subpage mappings anyway, so entire pages must be used.

External fragmentation is not (very) important here since most requests are for single page frames in this allocator (few kernel components require multiple, physically contiguous pages.)

4 Buffer Caching (20 pts)

a) (6 pts) Microsoft's latest OS, Vista, includes a feature called ReadyBoost. ReadyBoost allows the memory of a flash drive to be used as a cache for the hard disk's swap space. Microsoft states that ReadyBoost is only used for small, random reads and not for large, sequential reads. Explain why!

ReadyBoost is used as a write-through cache, so all data is written to both flash and the hard disk. Hard disks are preferred for sequential reads since doing so usually incurs a single seek and rotational delay, following by a continuous transfer from media. We must conclude that a hard disk's media transfer speed exceeds the read speed of a USB device.

b) (8 pts) Assume that you are using the LRU cache replacement policy, and that the number of buffer cache entries is 5. Give an example of a reference stream for which LRU will not be an optimal replacement policy. Show your work (showing both LRU, and the optimal policy!)

Hint: A reference stream is a sequence of blocks that are accessed by a program. For instance, if a file of length 4 blocks were read from start to end, the resulting reference stream, expressed in block numbers, would be 1, 2, 3, 4.

Looping accesses are an example of where LRU is not optimal. For instance, consider the reference stream: 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6. The following table shows which block, if any, a particular strategy would evict.

	1	2	3	4	5	6	1	2	3	4	5	6
LRU	-	-	-	-	-	1	2	3	4	5	6	1
OPT	-	-	-	-	-	5	-	-	-	-	*	-

** any block except block 6*

LRU incurs 7 misses, but an optimal strategy would only incur 2.

c) (6 pts) CVS stores the entire revision history of a file file.c in a single file called file.c,v. To ensure checkins are atomic, it uses the following strategy: a temporary file temp.c is created, based on the old file.c,v and the to-be-checked-in changes. After the temporary file has been written (via write(2)) and closed (via close(2)), the temporary file temp.c is renamed to file.c,v. Assume that the underlying file system uses a Unix-like approach to store files, directories, and inodes, and assume FFS-like consistency semantics for its metadata operations (in particular, assume that rename is atomic)!

Why could developers still lose the entire content of file.c,v in the event of an interruption, such as a power outage?

Even though the changes to the directory and inode will have reached the disk atomically, FFS does not guarantee that the file data itself has reached the disk. Ensuring that would have required an explicit request ("fsync").

This was a difficult question, and the answer is obvious only in hindsight. If you didn't get it, don't feel bad – neither did CVS's developers.

5 File Systems (28 pts)

a) (5 pts) When would you choose RAID-1 over RAID-5?

Compared to RAID-5, RAID-1 can provide you with higher read throughput if you can spread requests over multiple disks. If requests can be scheduled based on latency, it may also provide lower read latency. Also, when recovering after the loss of a disk drive, RAID-1 usually outperforms RAID-5. Finally, you may not want to pay the price if you need the redundancy, but not the additional capacity of a RAID-5 configuration.

A single reason sufficed for full credit.

b) (8 pts) Some recent file system designs, such as ext4, use an extent-based approach to reduce the per-file metadata overhead that is encountered in a traditional Unix-like file system.

i. (4 pts) Provide a formula f(l) for the overhead of representing metadata for a file of length l. State your assumptions and introduce variables as needed!

Overhead for metadata representation in a Unix-like system includes the inode and any needed index blocks. Assume that the number of indices stored per index block is N. A key observation is that all data blocks are referred to by one pointer, so the overhead can be roughly approximated by ceil(l/N). In addition, every index block is referred to by one pointer, adding ceil(l/N^k) for k level of depth.
Because a multi-level index uses one or multiple indices, depending on the length l, the actual formula is more complicated. Assuming that a triple-indirect blocks are used and assuming a sufficiently large l, the overhead would be 1 block for the inode containing D direct block pointers, 1 block for the single-indirect index block, 1 block for the double-indirect block and N blocks for the index blocks referred from the double-indirect block.

f(l) = 1 + 1 + (1 + N) + (1 + ceil(L/N^2) + ceil(L/N))

*where L = l – D – N – N * N is the number of blocks that cannot be indexed using the single and double-indirect indices. The dominating term here is L/N.*

Since this question ended up being more complicated than intended, we accepted the approximation l/N + l/N^2 + 1 for full credit, and l/N for 3 pts of partial credit.

ii. (4 pts) What is the minimum (best-case) overhead that an improved design such as the one proposed in ext4 could achieve?

The best case would be if the entire file were laid out contiguously in a single extent, in which case we would need to keep only the start block and the length as metadata, which, as in the Pintos base system, could probably be included in the inode so no additional metadata blocks would be required.

c) (15 pts) Describe the trade-off between performance and consistency in the area of file 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 (10 pts) and for your ability to communicate effectively in writing (5 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.*

The trade-off between performance and consistency arises from the enormous difference between random memory and disk access speeds. This difference forces the use of buffer caches, which keep copies of on-disk data in volatile RAM. Updates to this data must eventually be propagated to disk to become persistent. If there is an interruption while the data is kept in volatile RAM, it may be lost.

The trade-off arises from the choice of time window between an update and when the data is written to disk. Writing the data synchronously, or within a short time window, ensures that the on-disk data is always consistent with in-memory updates. This approach, however, leads to poor performance because applications have to wait for the disk operations to complete when performing updates. All update operations now proceed at disk speed, leading to processes that are blocked from making progress, and indirectly to poor CPU and RAM utilization. On the other hand, delaying the propagation of updates would improve performance by allowing processes to proceed even before the data has been written to disk, but consistency would be reduced, since the on-disk data may now lag substantially behind the cached in-memory content.

For these reasons, most file systems and buffer cache implementation define a set of consistency guarantees that allows them to recover the file system to a

well-defined state after an interruption. To achieve these guarantees, they either use synchronous writes, journaling, or write ordering. Applications must explicitly request performance-reducing synchronous writes if they wish to obtain additional consistency guarantees.

The grading criteria for this question say to assign “excellent”, “good”, or “unsatisfactory” rating to each answer in each of the categories “Knowledge” and “Writing.” I mapped these to points as follows:
Knowledge: Excellent 10 pts, Good 6 pts, Unsatisfactory 0 pts.
Writing: Excellent 5 pts, Good 3 pts, Unsatisfactory 0 pts.