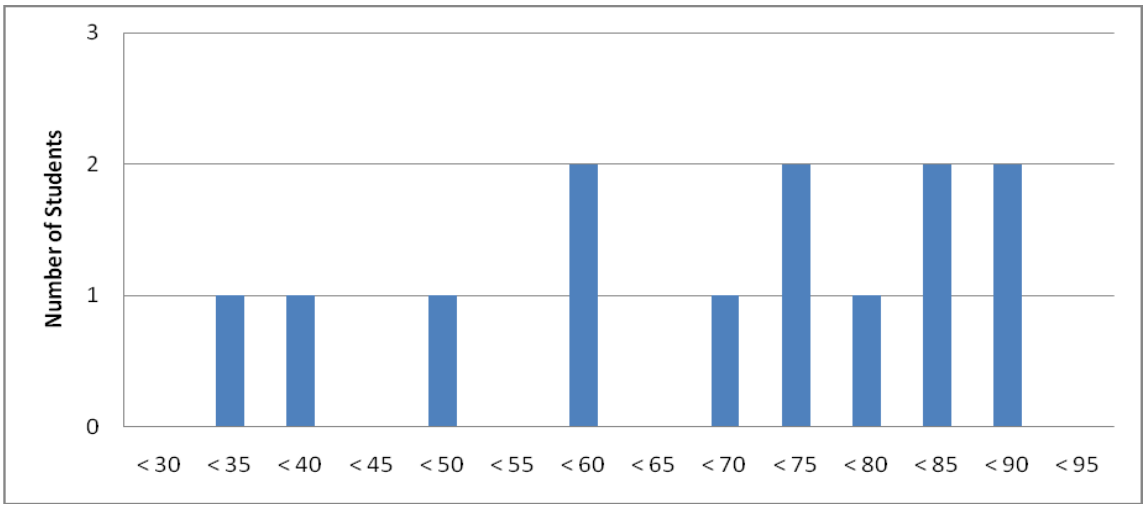
CS 3204 Final Exam

13 students took this exam. The following histogram and table shows the statistics. The exams can be picked up from my office.



#	Problem	Points	Min	Max	Mean	Median	SD
1	Checkpointing	25	5	24	16.62	17	4.82
2	Buffer Caching Algorithms	20	7	20	15.15	17	3.69
3	File Systems	20	2	20	13.46	17	6.83
4	Avoiding Stack Overflow	20	0	20	9.54	8	6.70
5	Large vs. Small Kernel	15	10	15	12.23	12	2.17
	Total	**100**	**31**	**89**	**67.00**	**75**	**19.13**

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

1 Checkpointing (25 pts)

Modern scientific applications that model complex physical phenomena may run for days and even months on end. To protect the results of these applications from machine failures, a technique referred to as checkpointing is used. Essentially, the application is paused, and its entire state is saved to the disk in a checkpoint file. If the application later fails, the last saved state is loaded from the file into memory and the application can resume without losing days/months worth of work.

a) (8 pts) Outline how will you incorporate checkpointing in Pintos kernel that has virtual memory support, especially, focusing on how you will pause the application, what features you will leverage and extend, what kernel-level data structures you will use to determine what should be saved, and how the checkpoint file should be created? For full-credit you should name specific Pintos structures/functions.

This requires writing all process associated pages to the checkpoint file. The pages include in-memory frames as well as swapped out pages. The supplemental page table is used to determine what application pages are stored where. The supplemental page table itself also needs to be written to the disk. In addition, the information needed at context-switch also needs to be saved, e.g., CPU registers, stack pointer etc. You also have to keep track of all the open files, and store that information in the checkpoint file.

The best way to pause (or take a snapshot of) the application is as follows. First, a fork of the original process is done to create a copy of the process (including multiple threads). Then the newly created process is moved to the suspended state, and all its contents written to the checkpoint file. The original process can continue to execute meanwhile.

Many students used timer_interruppt to pause the application. This will work for Pintos as there is only one running thread, but will not work if a process has multiple threads. I gave partial credit for this. Most students missed open files and other information.

b) (2 pts) Comment on whether virtual memory facilitates or hinders your checkpointing approach, and why?

First, VM reduces the added complexity of checkpointing as all the necessary information is already available in the VM supplemental page tables. Second and very importantly, VM allows checkpointed pages to be restored to different (available) physical frames while preserving the process' original view of its virtual address space. It would be extremely hard or even impossible to support checkpointing/restore without virtual memory.

I gave partial credit to answers that gave weaker reasons for benefits of VM.

c) (5 + 3 pts) Outline the procedure for resuming an application from a checkpoint. You should provide at least two optimizations to improve the performance of this process.

A new thread is created. The supplemental page table is loaded from the checkpoint file to memory. All the in-memory pages are reloaded in memory, and all the swapped out pages from the checkpoint file are moved to the swap space. This may also require swapping other processes' pages from core memory to make room for pages being restored pages. All the files that were originally open should be opened again, and the associated location pointers updated to correct locations. The saved CPU registers and state is loaded into the thread structure. Since, we forked the original process and suspended it, the restored process is a copy of the original in suspended state. Once all the state is ready, the newly created process is moved to the ready state and the kernel will context switch to it when it is selected by the scheduler. The process will resume from the same point as where it was suspended.

Opt1: Use lazy load of the checkpoint pages. Requires modification to the kernel supplemental pages to also include a "location" of CHECKPOINT_FILE, in addition to executable and swap.

Opt2: Don't write any code segment and load that direct from the application executable. Requires careful handling while writing the checkpoint file, and availability of the executable when restoring.

d) (2 pts) Give a scenario where checkpointing/resume will not work.

If the application is communicating with the outside world, e.g., over the network, with a user, then the checkpoint cannot capture this interaction. That outside data cannot always be resend so it may not be possible to always restart.

Failure of checkpointing process, and corruption of checkpoint file are naïve reasons, I only gave partial credit for these.

e) (5 pts) It is possible to checkpoint on one machine, transfer the checkpoint file to a different machine, and resume the application there. This is called application migration. List two main challenges that your checkpointing approach faces if it has to support application migration.

Hardware software difference may make it impossible to migrate. So the checkpoint has to be done in a portable manner. That is simply writing CPU or thread states won't work and require special consideration so both the machines can understand the checkpoint file and interpret it in a meaningful way. Think about how you would migrate a Linux process to a Windows machine! Second, the machine on which a process is being restored should have all the data files etc. that were opened and available on the checkpointing machine.

2 Buffer Caching Algorithms (20 pts)

For this question, assume a buffer cache that can only hold 4 data blocks. You can ignore the need for storing meta-data etc.

a) (4 + 4 pts) An application with a working set size of greater than 4 will have poor performance with our cache. Construct an example reference stream with the working set size of 5 that will give the worst hit ratio. Construct another example that will have a better hit ratio than the worst case. [*Assume LRU for calculating hit ratios*]

A reference stream with 5 blocks, e.g. A, B, C, D, E repeated in a looping fashion will give worst case behavior. Any combination of these (as long as it as 5 different blocks) will give better hit ratio. An example A, A, B , C, D, E repeatedly accessed in a loop.

b) (5 + 3 pts) For your worst case example from (a), determine the hit ratio under Belady's optimal cache replacement algorithm. Based on your results, suggest a practical replacement algorithm that will result in close to optimal hit ratio for your example.

Assuming A, B, C, D, E … Beladys's algorithm will give behave as follows:
M M M M M H H H M H H H M H H H M

Hence, over time the hit ratio with Beladys under steady state will approach 75%.

Most Recently Used is the policy that will give close to Belady's performance here with 60% hit ratio.

I also awarded partial credit for stating the LIFO algorithm instead of MRU.

c) (4 pts) For this part, assume that you have an extremely fast disk that supports access times similar to memory. Do we still need to have a buffer cache? Why?

The cache is designed to hide disk latency. However, having a cache allows the disk to be not used, while requests are being serviced from the cache. This provides opportunities for the disk to be shut down to save energy. So even with a fast disk, a buffer cache is useful.

This question was meant to make students think beyond simple principles. I was hoping to have at least one student identify the energy saving perspective. Anyways, I awarded credit to answers that identified the main purpose for the cache is to hide disk latency from applications. Full credit was awarded to students who gave a reasonable discussion of why buffer cache is used in the first place and how it complicates OS design.

3 File Systems (20 pts)

a) (8 pts) You are to design a special-purpose file system for storing digital images. Each image is of [*the same*] fixed-size and stored in a different file. The name of the file is a 128-bit unique number. Your file system should be able to store an extremely large number of such files. List the main design decisions of your file system, and compare and contrast your approach to your implementation of Pintos Project 4 on file systems.

Since the file sizes are fixed, you do not have to support file extension. Also, given that the size of the disk is known, you can calculate at the beginning how many files can be stored and created available map, file entries etc. statically. You also do no need to have index inodes, as contagious allocation can be done. You can choose a flat directory hierarchy to simply things.

The challenge is to find files, especially when a large number of files are to be stored. The disk can be divided into fixed-size slots for file size each. Using a hash function on the name of the file, a unique slot for storing the file can be determined. If the slot is already full, a different hash can be created.

Compared to P4, this does not require multi-level indices, nor need to support subdirectories and extensible files. The on-disk data structures are also simpler.

Most students got this right. However, full credit was only given if you discussed how an extremely large number of files can be managed, either through a hash mapping, or through some name-based sub-directories (to manage the files in a tree).

b) (3 pts) Give the steps to delete a file in your file system.

Locate the file to be deleted, remove its entry from the directory, and mark the corresponding disk space as available for other files. The slot can be used by other files now.

c) (4 pts) What can you say about fragmentation in your file system compared to your Project 4 implementation?

All files are of equal size, so there is no external fragmentation. Also the size of the file is known in advance so internal fragmentation can be minimized by proper selection of block sizes, or even eliminated. This is not easy to do in Project 4 as the file sizes are unknown and vary.

d) (2 + 3 pts) You are given two logical volumes that are formatted according to your file system design above. One volume is completely full, the other has more than the first's size available and free. Explain how you will merge the two volumes into one. Explain how you will merge the volumes, if both are almost full.

For case 1, simply repeat the following. Take a file from the smaller volume, copy it to the larger volume, delete the original from the smaller volume. Do this for all the files. Then add the smaller volumes disk space to the large volume, and update freemap and other structures on the larger volume.

For case 2, the above process is done but in steps. Taking a number of files from smaller volume, adding it to the large volume till the large volume is full. Then space from smaller volume is added to the larger volume, and the process repeated.

This question is actually more complex than the above discussion. But as long as the answers hit on the above points, I gave credit.

4 Avoiding Stack Overflow (20 pts)

A number of computer security issues arise when a user can overflow the stack so as to overwrite the valid program loaded into memory with malicious code. Once, the malicious code is executed, it can give the user unauthorized access. In this question, you will design a technique to avoid such a problem.

a) (15 pts) Write a C program that uses **mmap** to protect itself from stack overflow attacks. For full credit, you must use the appropriate functions, and provide correct addresses for mmaping.

```
void my_function (void)
{
        int foo;
        int fd = fopen("my_test_file", "r");
        mmapid_t protector mmap(fd, &foo-ALLOWED_LIMIT);

        // do stuff

        munmap(protector);
        fclose(fd);
}
```

You program has to do the following things: Determine the address of the stack. As long as you defined a local variable and used its address with some offset this was fine. Mmap the file read only between the stack and the executable code.

b) (5 pts) Stack overflow attacks require in-depth understanding of how the stack memory is assigned with respect to the users program in virtual memory. Assuming we cannot detect stack overflow, suggest how you can make it more difficult for a would-be attacker to compromise the system using stack overflow.

You can randomize the location of the stack in the virtual memory with respect to the executable. This will make it very difficult for a stack overflow attack to do harm. (It can still cause the process to crash though). Alternatively, you can make the stack portion non-executable. This will allow access to the stack but will not allow any execution of code in the stack range.

This was intentionally repeated from the midterm. Only students who have paid attention to the in class discussion regarding this received full credit.

5 Large vs. Small Kernel (15 pts)

Describe the trade-offs of having a large and complex operating system with a lot of functionality in the kernel versus having a minimal kernel and letting applications devise their own custom solutions to various OS tasks. Describe situations in which these trade-offs matters and which factors you may need to take into account when choosing or designing an OS.

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.*

Three tradeoffs: Flexibility, security, efficiency. Small kernel provides more flexibility. A larger kernel may be more efficient by allowing users to focus on their applications rather than reinventing the wheel. Management of shared resources such as memory, disk, etc. is difficult to do in user space, and must be incorporated in the kernel. Also, a smaller kernel actually improves security with fewer kernel debilitating bugs etc.

As long as you mentioned two of the tradeoffs, and stated your answer in a clear and meaningful context I awarded credit.

For content: Excellent answers clearly stated at least two of the above constraints. Good answers made some mention of these issues, but gave fewer details. There were no unsatisfactory answers.

For writing: Excellent answers used precise terms and no language mistakes. Good answers relied on inexact wording. Once again, I did not come across any unsatisfactory answers.

Rubric for converting Excellent/Good/Unsatisfactory to points:

	Writing	Content
Excellent	5	10
Good	3	7
Unsatisfactory	0	0