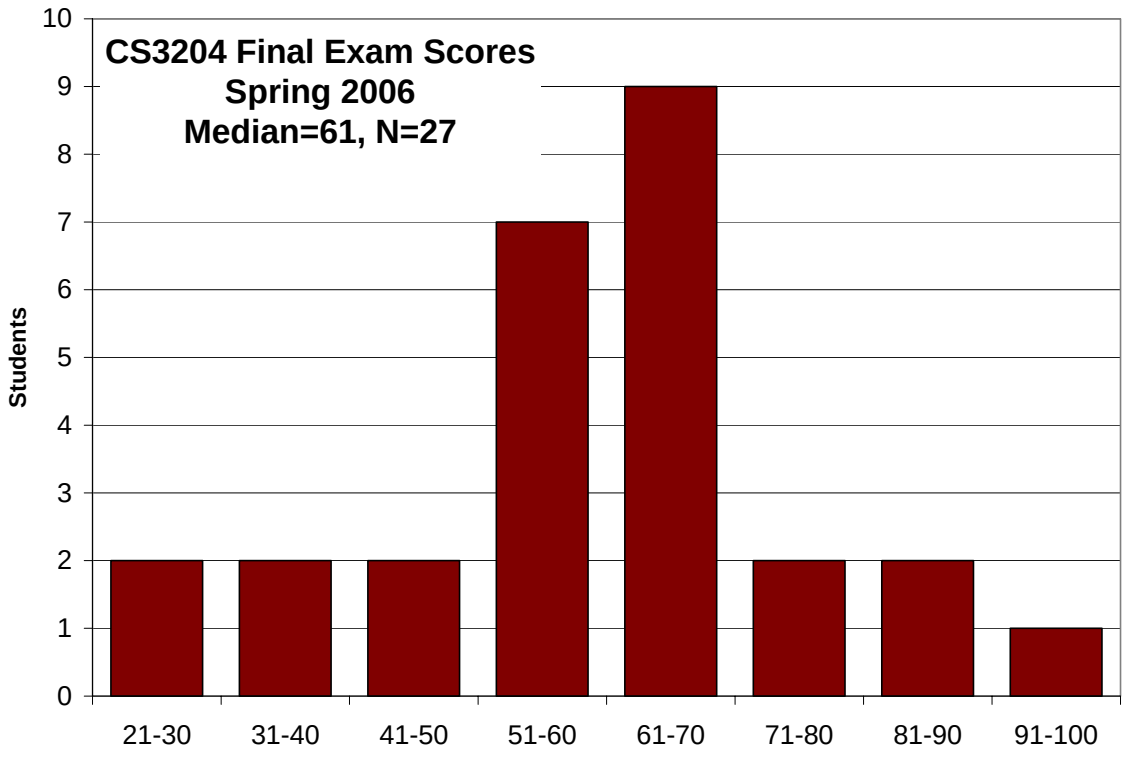
CS 3204 Final Exam Solutions

27 Students took the final exam. The table below shows the statistics for each problem and who graded it. The exams can be picked up at my office in 2160A. Please contact the grader of a problem with questions first.

Problem	***1***	***2***	***3***	***4***	***5***	*Total*
Possible Points	30	16	18	16	20	**100**
Median	17	12	12	9	8	**61**
Average	17.3	10.8	11.8	9.7	9.3	**59.0**
StdDev	5.1	4.8	5.0	3.3	4.6	**16.7**
Min	6	0	0	5	4	**23**
Max	27	16	18	16	19	**92**
Grader	Vijay	Godmar	Godmar	Vijay	Vijay: a), d) Godmar: b), c)	

Here is a histogram of the scores:



Comments regarding the point assignment/grading are shown in this style in the following document.

1 Virtual Memory (30pts)

a) (6 pts) The following code could trigger this vulnerability on a machine with less than 1GB of physical RAM.

```
size_t len = 1024 * 1024 * 1024; // 1GB
void * large = malloc(len);
// make sure [large, large+len] spans a valid virtual address range
sysctl(*, *, large, &len, *, *);
```

where * denotes arguments that are irrelevant. Note in particular that simply passing a large length to sysctl will not trigger the issue. The OS first verify that the memory range is a valid virtual range, this follows from the problem description in the advisory.

We looked for the memory allocation of a "large" buffer and passing that buffer to sysctl in your solutions. Partial credits were assigned if the memory allocation was not shown.

b) (4 pts) If you held a single lock whenever a page is paged in or out, only one process could be in the process of paging. All other processes would have to wait to submit their paging requests, causing undesirable and unnecessary serialization and low utilization.

Solutions mentioning about the unnecessary waits introduced by this lock and the degradation of performance were assigned full credit. Some of you have mentioned that the introduction of this new lock might lead to dead locks. Such solutions received partial credit.

c) (5 pts)

i. (2 pts)

The test `if(uaddr == NULL)` is not necessary because if uaddr is NULL, pagedir_get_page will return NULL since we do not map virtual address NULL.

Most of the solutions said that is_user_addr() would do this check whereas it actually doesn't check for NULL pointer. It just checks if the pointer is lesser than PHYS_BASE. These solutions received no credit.

ii. (3 pts)

The test is not useful either because it optimizes the uncommon case (where uaddr is NULL), but imposes the cost of a redundant check for the common case where uaddr is not NULL.

Solutions which stated that the check is not useful along with a correct explanation received full credit.

d) (3 pts)

Since palloc() allocates one or more pages at a time, there is no unused, wasted space inside an allocated block if all allocation requests are for multiples of PGSIZE. Some of you wrote that there is internal fragmentation if the application does not use all data in an allocated block, and we accepted that answer as well.

If your solution had given one of the 2 possible solutions along with the correct explanation, full credits were awarded. For unsatisfactory or no explanations, partial credits were assigned.

e) (3 pts) Under the same assumption, can external fragmentation occur?
Yes – palloc_get_multiple() may return NULL if it cannot find enough contiguous pages to satisfy the request for the desired number.
(Exception: external fragmentation does not occur if all requests are for exactly 1 page, i.e., there are only calls to palloc_get_page() and none to palloc_get_multiple() – however, the question explicitly included palloc_get_multiple() by stating that all allocation requests are for multiples of PGSIZE.)

For solutions that mentioned no external fragmentation can occur as the case of multiple page allocations was not considered, no points were given.

f) (5 pts) This program increments the major index in the inner loop, causing large deltas in the virtual addresses being accessed. This results in poor locality and a high number of TLB and/or cache misses.
Note that paging or page fault handling is unlikely to be the cause of this, since the machine has enough memory to accommodate 2 * 3000 * 3000 * sizeof(int). The number of page faults that need to be serviced during demand paging should be identical.

For those that mentioned about the locality of reference but not about the cache/TLB hits, partial credits were awarded. Solutions which mentioned about swapping were given partial credits too.

g) (4 pts) The “page present” bit in the hardware page table entry corresponds to the “valid” bit in a buffer cache descriptor. (Some of the literature even calls this bit the “valid” bit also.)

Should have got the name of the bit correct or should have given an explanation of the purpose of the bit to get credits.

2 Symbolic Links (16 pts)

a) (6 pts) Necessary changes include (two are sufficient:)
- Adding a new system call number and system call stub
- Adding a new case or entry to your function call dispatch code

- Adding a function to implement the system call that checks the string arguments for validity and copies them into the kernel, and eventually calls into the file system code.

Solutions that claimed that in-memory data structures that keep track of oldpath & newpath would be maintained in the system call layer (user-side or kernel-side) received no credit.

b) (5 pts) You could handle symbolic links as another file type besides regular files and directories. The file type is stored in the inode. If the type indicates “symbolic link”, then the file data would be the string that was passed to “oldpath” when the link was created. You could store short symlinks directly in the inode, but the PATH_MAX value would prevent you from always doing so. If the path name is longer than what can be stored in the inode, you might need one or two additional blocks, whose numbers you have to store in the inode.

For full credit, you had to mention how the on-disk layout would be changed, so symlinks could be retrieved after shutdown & reboot. That is, you had to talk about changes to either struct inode_disk and/or dir_entry. We also accepted approaches that stored the symlink in the directory, as part of the directory entry. Approaches that rely on changing in-memory data structures will not achieve persistency and did not receive credit. Note also that eager resolution of links would not allow the implementation of the shown ls example.

c) (5 pts) During path resolution, if it determined that a component is a symbolic link, the resolution code has to read the content of the symlink. The path stored in the symlink has to be split into its components, and these components have to be inserted into the original pathname at the point where the symlink component occurred. In addition, if the symlink’s content refers to an absolute path (starts with a /), the directory in which the next component is looked up has to be set to the root directory; for relative paths, no change in the directory is necessary.

For full credit, you had to explain how you integrate the resolution of symlinks into your path resolution routine. A common mistake was to assume that the symlink component could only occur at the end of a path.

3 File Systems (18 pts)

a) (4 pts) Short files could be stored directly in the inode in the space that would otherwise be taken up by direct block pointers. Reiserfs and IBM’s JFS do this.

Most got this right. No credit was given for saying a buffer cache can be used or anything that mentioned direct blocks – as soon as you have pointers (even direct block pointers), you need more than 1 disk access.

b) (6 pts) Given the simplicity of your Project 4 filesystem, there were probably numerous performance problems in your filesystem layout and buffer cache. For instance, the lack of a policy that allocates related blocks closely together would probably lead to similar performance problems as in the original Unix file system.

i. (2 pts) Workload/Scenario would be a workload in which one process slowly grows a large file, while other processes create and destroy many, small files, which prevents the process growing the large file from obtaining contiguous sectors for its blocks.

ii. (2 pts) Performance Problem: Numerous seeks are required when the large file is being accessed sequentially.

iii. (2 pts) Possible Countermeasure: Use clustering: pre-allocate a contiguous number of blocks ahead if a file is being grown.

Any reasonable scenario was accepted, as long as it referred to a performance problem (rather than a correctness issue/bug in your implementation.) You had to be sufficiently precise about the workload, or the scenario of file accesses that would cause the problem.
Some of you mentioned the problem associated with long sequential accesses and an LRU buffer cache policy. To receive full credit for that, you had to mention how LRU affects the performance under sequential reads: namely that it has the potential to evict other, useful data from the cache. As far as the long sequential read itself is concerned, there's no difference between LRU & MFU – it's all cold misses.

c) (8 pts)

iv. (4 pts) Orphaned inodes can occur when a new file is created, its inode written to disk, but the directory entry pointing to it has not reached the disk when a crash occurred. Alternatively, they can occur if a file is deleted and its entry in a directory persistently removed, but the corresponding inode has not been marked as free.

A common mistake was to say that orphaned inodes refer to inodes that were not written to disk when the system crashed. However, if they never made it to disk, fsck will not find them, and can't possibly place them in /lost+found.

v. (4 pts) The Unix version of fsck recovers inodes that have no directory entries pointing to them by scanning the inode table, which is in a separate section of the disk. This requires that inodes are stored in that separate section, which is not true in Pintos where inodes can be stored in any sector. Therefore, unless you changed the way Pintos

allocates sectors for its inodes, you cannot implement it even if you were to implement a version of fsck.

Key to answering this question was realizing that Pintos has no way of finding unlinked inodes (unless you changed that aspect in your implementation, which you'd have to state.) Some of you have described ways in which Pintos could be changed to allow inode recovery or avoid the need for it, but those didn't answer the question that was asked.

4 Networking & Operating Systems (16 pts)

a) (10 pts) The synchronization between layer k-1 and k follows a simple producer/consumer model. A queue could be used to hold packets delivered by layer k-1, but not yet received by layer k. A lock could be used to protect concurrent access to that queue. A condition variable could be used to signal the availability of packets in the receive queue.

// Layer k calls "receive()" to receive a packet	// Layer k-1 calls "Deliver()" when a packet is available
Queue q; Lock l; Condvar c;	

Packet receive() {
  lock(l);
  while (empty(q))
  cond_wait(c, l);
  Packet p = dequeue(q);
  unlock(l);
  return p;
}
	Deliver(Packet p) {
	  lock(l);
	  enqueue(q, p);
	  cond_signal(c);
	  unlock(l) ;
	}

Partial credits were awarded to those solutions that had not used a common buffer, lock and conditional variable depending on the degree of correctness. Some of the solutions had used a semaphore instead of the conditional variable and such solutions were considered to be correct as well.

b) (6 pts)

i. (3 pts) An advantage of paging over the network could be that it's faster than paging to disk, in particular when the remote machine is connected through a high-speed local network.

ii. (3 pts) A disadvantage/complication is that a node's functioning now relies on the correct functioning and availability of another machine on the network.

Any reasonable advantage/disadvantage was accepted.

5 Short Questions (20 pts)

a) (5 pts) The x86 MMU consults the page table of a process, which lists the objects (pages) to which the current process (the subject) has access. This is a capability-based approach.

Solutions that named the correct approach, but got the reasoning incorrect were not given full score.

b) (5 pts) When implementing stack growth in a kernel-level threading systems, it can either happen that the virtual addresses to which the stack should be grown are already used, for instance by some other thread's stack. On the other hand, if large pieces of virtual address space are preallocated to each thread's stack to allow it to grow, virtual address space fragmentation is likely to occur.

The problem with implementing stack growth in a system that supports multiple threads per process is that you cannot grow the stack past an already assigned virtual address range. Therefore, you will either be unable to grow a thread's stack, or run into virtual address fragmentation, and/or you have to put a limit on the number of threads you can support. Partial credit was given for pointing out that concurrency must be managed when multiple threads grow their stacks simultaneously, but this is really a minor problem, especially considering the fact that an OS already has to handle the case where multiple single-threaded processes simultaneously extend their stacks.

c) (5 pts) The "small write" problem in a RAID-5 system refers to the fact that an update to a single sector might cause a read of the sector's old values, a read of the associated parity block, and 2 writes to store the sector's new values and the new parity block. One write results in four disk accesses.

For full credit, you had to mention that a small write causes 2 reads (the original blocks and the parity block) and 2 writes (the updated original block, and the updated parity block. No credit was given for comments like "small writes make inefficient use of space."

d) (5 pts) Pure LRU is usually not the best cache eviction strategy for a buffer cache because sequential accesses to large files would quickly fill the buffer cache with data that is not being used in the future, evicting other data that will be accessed. For this reason, most systems require at least one additional access to a buffer cache block before it is considered worthy of caching.

Some of the solutions mentioned the general drawback of using a LRU replacement policy, but did not take into consideration the special cases introduced by the buffer cache (which is that is has to deal with sequential accesses as the most frequent access pattern.) Such solutions did not receive full credit.