CS 3204 Sample Final Exam

Solutions are shown in this style. This final exam was given Spring 2006.

1 Virtual Memory (30pts)

a) (6 pts) The following is an excerpt from the NetBSD man page for the sysctl(3) function, which allows retrieval and manipulation of named system parameters:

```
#include <sys/param.h>
#include <sys/sysctl.h>

int
sysctl(int *name, u_int namelen, void *oldp, size_t *oldlenp,
          void *newp, size_t newlen);
```

```
 The sysctl function retrieves system information and allows processes
with appropriate privileges to set system information. […]

 Unless explicitly noted below, sysctl returns a consistent snapshot of
the data requested.  Consistency is obtained by locking the destination
buffer into memory so that the data may be copied out without blocking.
```

On Apr 12, 2006, the NetBSD Team issued an security advisory regarding sysctl(3), part of which is reproduced here:

```
                NetBSD Security Advisory 2006-013
                ================================

Topic:          sysctl(3) local denial of service

Severity:       Any local user can crash the system

Abstract
========

The user supplied buffer where results of the sysctl(3) call are stored
is locked into physical memory without checking its size.  This way, a
malicious user can cause a system lockup by allocating all available
physical memory on most systems.

Technical Details
=================

The system call implementing the sysctl(3) library call tries to lock
the user supplied result buffer into physical memory, to avoid the
interferences of information collection with other system activity.
The size of that buffer is not checked against system limits.

The VM system checks whether the virtual address of the buffer is part
of the user address space, but since the amount of virtual memory a
single user is able to allocate exceeds the available physical memory
```

```
on most systems, a user can cause a system lockup by exhaustion of
physical memory. […]
```

Give a short C program that could exploit this vulnerability. You may assume that NetBSD uses demand paging throughout. Ignore those arguments to sysctl(3) that do not play a role in this vulnerability.

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 verifies that the memory range is a valid virtual range, as is evident from the problem description in the advisory.*

b) (4 pts) In Project 3, if you implemented eviction, a situation could occur where a page frame that was in the process of being evicted to swap disk would be accessed by the thread that still owned it. Handling this case properly required that you either aborted the eviction in this case, or that the owning thread waited until the eviction was complete, then immediately faulted the same page into a new page frame.

   An easier approach would have been to use a lock that is taken on the entry to the page fault handler code (in exception.c). The same lock would also need to be taken whenever a page is being evicted and it would be released afterwards.

   Explain the drawback of this approach!

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. (NB: this is now a design document question for project 3)

c) (5 pts) Many implementations of Pintos used code similar to this one to check the validity of user pointers:

```
static void* resolve_user_address(void *uaddr, struct thread *t,
                                  uint32_t sz)
{
   if(uaddr == NULL)
    return NULL;

  /* Check to see if uaddr is in the user uaddr space */
```

```
  if(!is_user_vaddr(uaddr))
    return NULL;

  /* Get the kernel address for the mapping */
  if((kaddr = pagedir_get_page(t->pagedir, uaddr)) == NULL)
    return NULL;
  … // rest omitted
}
```

i. (2 pts) Explain why the check `if(uaddr == NULL)` is not necessary!

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.

ii. (3 pts) Even though it is not necessary, is it *useful* to have that check? Justify your answer!

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.

d) (3 pts) Assume all allocation requests are multiples of PGSIZE. Under this assumption, does Pintos's palloc() allocator suffer from internal fragmentation? Say why or why not.

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.

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.)

f) (5 pts) Consider the following program:

```
001: #include <stdlib.h>
002:
003: int **
004: allocate_matrix(int n)
005: {
006:     int i, ** m = calloc(n, sizeof(int *));
007:     for (i = 0; i < n; i++) {
008:         m[i] = calloc(n, sizeof(int));
009:     }
010:     return m;
011: }
```

```
012:
013: int
014: main(int ac, char *av[])
015: {
016:     int i, j, n = 3000;
017:     int **m1 = allocate_matrix(n);
018:     int **m2 = allocate_matrix(n);
019:
020:     for (i = 0; i < n; i++)
021:         for (j = 0; j < n; j++)
022:             m2[j][i] += m1[j][i];
023: }
```

When run on a PowerMac G5 with 6.5 GB of physical memory, this program takes about 5 seconds to finish. If I change line 22 to read

```
022:             m2[i][j] += m1[i][j];
```

then the program finishes in about 0.36 seconds.

Explain a possible reason for this behavior!

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 in either version.*

g) (4 pts) In lecture we had discussed the similarity of buffer caching to paged virtual memory. Which bit in a hardware page table entry corresponds to the “valid” bit in a buffer cache descriptor?

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.)

2 Symbolic Links (16 pts)

Symbolic links in Unix are shortcuts to other files. They can be created with the command *ln –s*. Here are examples to remind you of how they work:

```
% mkdir a
% echo hello > a/b
% ln -s a/b c
% cat c
hello
% pwd
/Users/gback/CS3204/final
% ln -s /Users/gback/CS3204/final/a/b d
% cat d
hello
```

```
% ls -l
total 16
drwxr-xr-x   3 gback  gback  102 May  4 23:20 a
lrwxr-xr-x   1 gback  gback    3 May  4 23:20 c -> a/b
lrwxr-xr-x   1 gback  gback   29 May  4 23:20 d -> /Users/gback/CS3204/final/a/b
```

Symbolic links are implemented using the symlink(2) system call, which is defined as follows:

```
#include <unistd.h>
int symlink(const char *oldpath, const char *newpath);
```

Suppose you were to add support for symbolic links to your Project 4 implementation of Pintos (if you did not complete Project 4, answer this problem as if you had completed it.)

a) (6 pts) Name *two* changes you would have to make to the code in the lib/user/* and the userprog/ directory!

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.*

b) (5 pts) How would you change the on-disk layout of your filesystem to store symbolic links persistently? Be specific!
Note: the “oldpath” argument can point to a pathname up to PATH_MAX characters in length. filesys/directory.h defines PATH_MAX to be 1024.

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.

c) (5 pts) Explain how your path resolution routine would have to be modified in order to accommodate symlinks. Discuss both relative and absolute paths.

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.

3 File Systems (18 pts)

a) (4 pts) Ignoring directory lookup, in an indexed filesystem, accessing even a short 20-byte file takes at least two disk accesses: one access to retrieve the inode from disk, and a second access to retrieve the file's data. Explain how you could reduce this number to 1 disk access for such short files.

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.

b) (6 pts) Name one performance problem with your filesystem and/or buffer cache design in Project 4. Be specific. For this performance problem, describe a workload/scenario that would trigger the problem. Name a technique that could alleviate the problem.

i. (2 pts) Workload/Scenario:

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.

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:

Performance Problem: Numerous seeks are required when the large file is being accessed sequentially.

iii. (2 pts) Possible Countermeasure:

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

c) (8 pts) fsck for Unix implements inode recovery - orphaned inodes after a crash are listed in a special directory /lost+found.

i. (4 pts) What are orphaned inodes and how can they occur?

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.

ii. (4 pts) Say you were to write an fsck program for your version of Pintos. Could you implement inode recovery? Say why or why not. State your assumptions if necessary.

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 method 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.

4 Networking & Operating Systems (16 pts)

a) (10 pts) Suppose a host uses a layered network architecture. If a packet is received by layer k-1, it must be processed and eventually passed up to layer k. Write *pseudo code* that could be used by layers k-1 & k for this purpose. Be sure to include all necessary synchronization, but do not worry about memory management.

// Layer k calls “receive()” to receive a packet	// Layer k-1 calls “Deliver()” when a packet is available
// Declare necessary data structures here, if any	

Packet receive() {
 	
}
	Deliver(Packet p) {
	
	}

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);*	*Deliver(Packet p) {* *lock(l);*

```
  while (empty(q))
    cond_wait(c, l);
  Packet p = dequeue(q);
  unlock(l);
  return p;
}
```

```
  enqueue(q, p);
  cond_signal(c);
  unlock(l) ;
}
```

b) (6 pts) It has been proposed to harness unused RAM to implement remote paging, where a page is evicted not to disk, but sent to another machine on the same network. Name one potential advantage and one potential disadvantage/complication of this idea!

i. (3 pts) Advantage

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) Disadvantage

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

5 Short Questions (20 pts)

a) (5 pts) When viewed as a protection system, does the MMU of the 80x86 series of processors use an access-control-list or a capability-based approach? Justify your answer!

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.

b) (5 pts) What problem occurs when implementing stack growth on a system that supports multiple kernel-level threads per process?

When implementing stack growth in a kernel-level threading systems, it can 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 (particularly on systems with a 32bit virtual address space.)

c) (5 pts) What is the "small write" problem in a RAID-5 system?

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.

d) (5 pts) Why is LRU usually not the best cache eviction strategy for a buffer cache?

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.