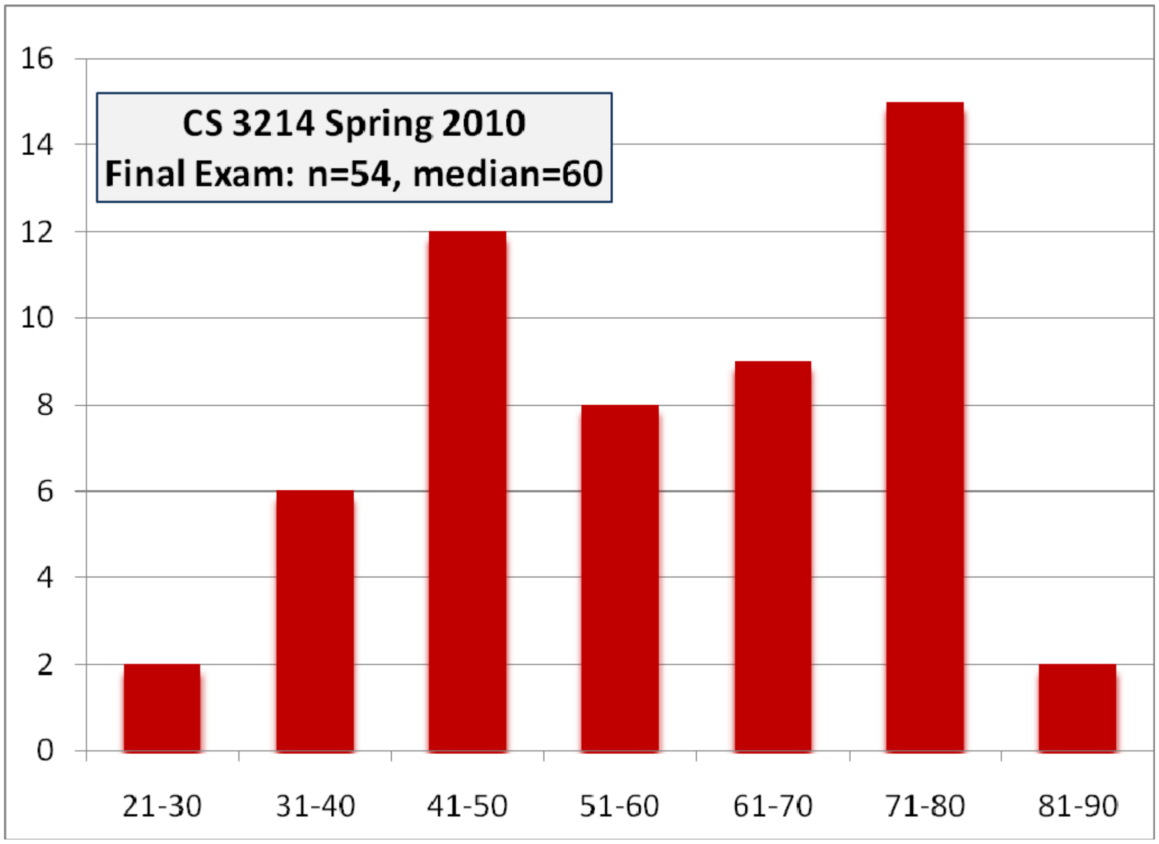
CS 3214 Final Exam Solutions

54 students took the final exam. The table below shows who graded which problem and statistics about it. The exams are kept by your section's instructor. Arrange by email if you wish to look at your exam.

	P1	P2	P3	P4	P5	P6	Total
Min	0	3	4	1	3	0	26
Max	12	17	26	14	15	16	85
Possible	12	18	24	14	16	16	100
Avg	6.5	9.4	18.3	7.7	8.7	7.4	58.6
StDev	3.5	3.8	5.6	3.3	2.8	4.3	15.8
Median	7	9	19.5	7.5	9	6	60
Grader	Godmar	Godmar	Peter	Scott	Ali	Ali	



1. Linking and Loading (12 pts)

The following questions are related to linking and loading in a C/Unix environment.

a) (4 pts) A coding style rule that is used in many C-based projects is that variables that are used only within one compilation unit be declared static. Sketch briefly how you could create a script as part of the build process that would check if programmers followed this rule!

Use nm or a similar tool and verify that for every defined global symbol (recognizable with a capital letter such as 'T', 'C', or 'D' in nm), there exists at least one undefined ('U') reference in another .o file.

Some answers suggested a source code analysis approach by looking for 'extern' declarations. While possible, this is more difficult because simply declaring a variable or function extern does not produce an undefined reference unless the variable or function is actually used.

A common mistake was to propose an algorithm that would check that static variables with the same name aren't used outside the current compilation unit. This is wrong: static variables are local to a compilation unit, so using a static variable with the same name in different files is entirely ok (and occurs quite often).

b) (4 pts) Consider the following two .c files:

```
// inck.c
extern int k;

void inc_k()
{
    k++;
}
```

```
// main.c
extern void inc_k();

int main() {

    inc_k();
    return 0;
}
```

What error message would the command `gcc inck.c main.c -o main` produce if one attempted to compile and link this program?

```
$ gcc -Wall inck.c main.c -o main
/tmp/cc4HL3gU.o: In function `inc_k':
inck.c:(.text+0x4): undefined reference to `k'
inck.c:(.text+0xc): undefined reference to `k'
collect2: ld returned 1 exit status
```

The symbol 'k' is referenced in inck.o, but never defined. 'extern' declares, but does not define a symbol.

c) (4 pts) Address space randomization is a defensive technique used to increase an attacker's difficulty of succeeding with overflow attacks that require a priori knowledge of where a program's data is located in its virtual address space. For instance, the location of the stack and the start of the heap are randomized on modern systems, so that they vary between different runs of a program. Could the location of global variables be similarly randomized? State your assumptions if necessary!

No or yes, depending on your assumptions.

Assuming the way executables are customarily built, the answer is no because the linker determines the location of all global variables at link time, then places the computed addresses directly into the executable.

This could be avoided by building position-independent executables that use indirection via a global offset table for each access to global variables, just like shared libraries do. Note that shared libraries can access global variables used in their implementation, even though they may be located at different virtual addresses in the processes that use this shared library.

Some answers proposed randomization as part of the linking process, not the loading process. In this case, the locations of global variables would be the same from run to run, although they would vary between systems (and an attacker may not know the location on the system they are attacking.) This approach means that custom binaries need to be built for each system. This approach has so far been used only in experimental settings.

2. Running Programs On Unix (18 pts)

The following questions relate to how programs execute on Unix-like operating systems.

a) (4 pts) Consider the following program contained in a file prog1.c

```
// prog1.c
#include <stdio.h>
#include <unistd.h>

int
main()
{
    printf("one\n");
    write(STDOUT_FILENO, "two\n", 4);
    return 0;
}
```

When run on the command line, this program outputs:

```
$ ./prog1
one
```

```
two
```

However, if the output of the program is first piped through the 'cat' program, the following is displayed on the terminal:

```
$ ./prog1 | cat
two
one
```

Explain what could cause this behavior!

printf() uses buffered standard I/O, and write() is a system call that immediately invokes the kernel to output the data.

The stdio FILE buffer is flushed when a newline \n is encountered, but only if the FILE is connected to a terminal. If stdout is connected to a pipe, as in the second run, flushing is delayed until the program exits.

Common mistakes were to assume that printf() and write() go to different file descriptors, or that printf() goes directly to the terminal, or that standard I/O uses some kind of asynchronous buffering where flushing is done concurrently by an independent thread. Neither is the case. printf() merely provides buffering for writes() to file descriptor 1 – flushing the buffer is eventually accomplished by issuing a write(1, …) system call. The difference is the flushing policy – standard I/O tries to avoid flushing when it can, and attaching the file descriptor to a pipe means no flushing on \n.
We awarded 3 out of 4 points if you recognized that the issue was related to flushing.

b) (4 pts) When a Unix process exits, the OS will automatically reclaim some of the system resources it was using. Give one example of a resource that the OS will reclaim, and one example of a resource that the programmer must reclaim before exiting!

i. (2 pts) A resource the programmer does not need to worry about when a program exits:

- Open file descriptors for files, pipes, sockets, etc. – these are all closed by the OS on exit() or termination, as if the programmer had called close().
- The entire virtual address space, including the runtime stacks of all threads, dynamically allocated heap or other anonymous memory, mmap()'d memory segments and the program code and global variables.
- Exit statuses of unwaited-for children (e.g., zombies) – these are reaped automatically if their parent exits.
- Any threads started by this process are terminated automatically when a process exits.

ii. (2 pts) A resource the programmer needs to worry about:

- Named resources, including (temporary or not) files, named pipes, named shared memory. These persist even after the process creating them has terminated. An exemption are temporary files that have been unlinked right after creation, as per the pattern discussed in class.
- Still running child processes (orphans: with respect to their resource consumption, not with respect to their eventual reaping). The programmer needs to decide if these processes should be terminated or not, depending on the desired application semantics.

The key message is that the OS attempts to reclaim as many resources as it can so that the fate of an individual process does not affect the stability of the remaining system. It's one of the fundamental services an OS provides.

A very common mistake was to say that a programmer needs to worry about dynamically allocated memory when a program exits. If this were the case, a computer would become quickly unusable from running programs, as most programs do not ensure that dynamically allocated memory is deallocated before exiting. (And, it would mean that the system would leak memory whenever your program is terminated by a signal.)

Perhaps valgrind was more confusing than helpful here: valgrind reports leaked memory at the end of each run. The intention here is to help programmers create applications that don't leak memory in the long run. For instance, if your actual program does `while (1) doit();`, then letting valgrind examine leaks in `int main() { doit(); }` will help you avoid leaks in the actual deployment situation.

A second mistake was to say that you have to worry about zombies when exiting. This is not so: you have to worry about zombies (unreaped children) while you're running, not after you're dead.

c) (10 pts) Consider the following interaction of a user running bash in a terminal. In the table below, list the following events: (1) process-related system calls (fork, exec, exit, kill, wait/waitpid) and (2) file descriptor related calls (open, close, pipe, dup, dup2)

User input **shown in bold,** terminal output *in italics*	Shell Process	Child Process$_1$	Child Process$_2$
$* **cat < main.c \| grep main** *int main() {* *$	pipe([RD, WR])		
	fork()		
	close(WR)	close(RD)	
	fork()	dup2(WR,1)	
	close(RD)	close(WR)	dup2(RD, 0)
		open("main.c") = FD	close(RD)
		dup2(FD, 0)	exec("grep")

		close(FD)	exit()
		exec("cat")	
		exit()	
	waitpid(-1)		
	waitpid(-1)		

Use different rows to express if events are synchronized, i.e., if it is guaranteed that event n occurs after event m, then event n should be listed in a row below event m. Note that not all rows/columns will have entries.

The 'close()' calls are shown for completeness, but our grading focused on the remaining calls. Also note that waitpid() is synchronized with exit() in the sense waitpid() will not return until after the process has exited – it's possible for the parent shell to reach waitpid() before the child processes completed their execution (and it typically does.)

Note that you cannot do dup2() in the shell process because that would affect the shell's stdin/stdout file descriptor.

3. Multi-Threading (24 pts)

a) (4 pts) A number of students used the following approach when implementing the thread pool in the accompanying exercise.

```
thread_pool_get(threadpool *p, future **t)
{
  lock(p->lock);
  while (threadpool_empty(p) && !threadpool_shutdown(p))
     cond_wait(p->lock, p->cond);
  // …
  *t = threadpool_pulltask(p);
  unlock(p->lock);
}

threadpool_submit(threadpool *p, future *t)
{
  lock(p->lock);
  threadpool_add(t);
  unlock(p->lock);
  cond_signal(p->cond);
}
```

Could this implementation lead to lost wake-ups?

No. With the exception of issuing cond_signal() outside the monitor lock, this code follows exactly the established idiom.

Every submission of a future is eventually followed by a signal, and every worker thread will either see the future in the queue when acquiring the monitor, or the signal will arrive after the thread has added itself to the monitor's condition variable queue. Note that it may happen that a thread wakes up only to see that another worker thread “stole” the added future from it. That’s okay, and the code is designed for it. The worker thread will simply go back to wait, and the enqueued future was processed.

This implementation has a minor efficiency problem, however – by delaying the call to cond_signal to outside the monitor, worker threads may have to wait longer to be woken up than if the call occurred inside the monitor. More specifically, each worker thread’s progress depends on how the scheduler treats the submitting thread between the unlock() and the cond_signal() call. This is generally undesirable. The recommended style is to call cond_signal inside the monitor (in Java’s adaption of monitors, this behavior is enforced via a ‘IllegalMonitorStateException’ thrown if notify() is called on an unsynchronized object).

One section received a broken version of this question, which is why we didn’t grade it. Everybody was awarded 4 points for this question. Kudos to those who noticed it.

b) (3 pts) Some students implemented mutual exclusion in program 5 using the following sequence:

```
void doit(int fd)
{
    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_lock(&lock);
    ...
    // access some shared resource
    pthread_mutex_unlock(&lock);
}
```

Explain why this is wrong!

It creates a local, non-shared mutex for each thread. Only this thread will acquire this mutex, and other threads will acquire theirs, leading to loss of mutual exclusion. Using locks for mutual exclusion requires that all threads aiming to access a shared resource contend for the same mutex, which only one thread will be able to lock at a time.

c) (9 pts) In the next 2 subparts of this question, we will explore the concept of rendezvous, which is a point that two threads have to reach before any of them can proceed further. Consider the following implementations of rendezvous using two semaphores.

(V1)	(V2)	(V3)

`sem_t aHere;`
`sem_t bHere;`
 	
`// initial value is zero`
`sem_init(&aHere, 0, 0);`
	`sem_t aHere;`
	`sem_t bHere;`
	
	`// initial value is zero`
	`sem_init(&aHere, 0, 0);`
		`sem_t aHere;`
		`sem_t bHere;`
		
		`// initial value is zero`
		`sem_init(&aHere, 0, 0);`

sem_init(&bHere, 0, 0);	sem_init(&bHere, 0, 0);	sem_init(&bHere, 0, 0);
thread_A_code	thread_A_code	thread_A_code
{	{	{
…	…	…
sem_wait(&bHere);	sem_wait(&bHere);	sem_post(&aHere);
sem_post(&aHere);	sem_post(&aHere);	sem_wait(&bHere);
…	…	…
}	}	}
thread_B_code	thread_B_code	thread_B_code
{	{	{
…	…	…
sem_wait(&aHere);	sem_post(&bHere);	sem_post(&bHere);
sem_post(&bHere);	sem_wait(&aHere);	sem_wait(&aHere);
…	…	…
}	}	}

i. (3 pts) Discuss the correctness and expected performance of variant V1. Identify any problems!

V1 leads to deterministic deadlock. Thread A waits for semaphore bHere to be signaled before signaling aHere, but this will never happen because thread B waits for aHere to be signaled before signaling bHere.

ii. (3 pts) Discuss the correctness and expected performance of variant V2. Identify any problems!

V2 is correct in that it guarantees rendezvous. Thread A will block on bHere until thread B signals it. Subsequently, thread A will signal aHere, allowing thread B to progress past its sem_wait call.
This implementation has a performance drawback, however. If thread A arrives at the rendezvous point first, it will block in the sem_wait call, but then thread B will very likely also block because it must wait until thread A was resumed from its sem_wait() and issued the sem_post() call. Conversely, if thread B arrives first, thread A will not have to block.

iii. (3 pts) Complete V3 so that the problems identified in i) and ii) no longer occur! (Fill in your solution above).

Ordering the sem_post call before the sem_wait call removes this asymmetry. It ensures that the second thread to arrive at the rendezvous will not block, independent of whether thread A or B arrives first.

d) (8 pts) The generalization of rendezvous to N threads is called a *barrier*. Consider the following implementation of a barrier:

```
int count = number_of_threads;
pthread_mutex_t barrier_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
sem_t barrier_signal;

sem_init(&barrier_signal, 0, 0); // init to 0

void barrier() // barrier function is called by each thread
{
   pthread_mutex_lock(&barrier_mutex);
   count--;
   pthread_mutex_unlock(&barrier_mutex);

   if (count == 0)
      sem_post(&barrier_signal);

   sem_wait(&barrier_signal);
   sem_post(&barrier_signal);
}
```

i. (4 pts) Suppose N threads are running on a single single-core CPU under the regime of a non-preemptive scheduler (so that context switches can occur only at synchronization points, such as pthread_mutex_lock and sem_wait()). Provide the sequence of events when the 1st, 2nd, etc. up to the nth thread are calling this function!

Each arriving thread decrements the count by one, and all but the last will end up waiting at the barrier_signal semaphore. When the last thread arrives at the barrier, it will post the semaphore, allowing one blocked thread to move past it. This thread will in turn post the semaphore again, allowing the next thread to move past it, and so on until all threads have moved past the 'barrier' function after the arrival of the last thread.

You will note that this (incomplete) barrier implementation cannot be used more than once, because the final value of the semaphore will be 1 after each thread called barrier() once.

2 pts of extra credit if you realized the point made in the last paragraph.

ii. (4 pts) When run on a contemporary system with multiple cores and/or CPUs running under the regime of a preemptive scheduler, this code contains a race condition. Explain why the code will not work correctly under these assumptions, and suggest a fix for the algorithm!

There is a race condition in the test count == 0, which is done outside the barrier_mutex. It could be fixed by placing the test inside the lock/unlock bracket.

That said, the race condition doesn't affect the correctness of barrier – it simply means that the final value of the semaphore is undefined (could be larger than one if multiple threads see count to be 0 when they test it). This will make the problem of not being able to use the barrier multiple times harder to repair.

As an aside, real barriers are not implemented like this; this example serves only for illustrative purposes.

4. Dynamic Storage Allocation (14 pts)

a) (4 pts) The performance of dynamic memory allocators is a trade-off between peak utilization and throughput. High throughput often implies low utilization, and higher utilization reduces throughput. Name 2 allocation strategies that represent the extreme points of this spectrum!

i. (2 pts) High throughput, low utilization:

The naïve “bump-a-pointer, never free” allocator provided in the sample code for program 4 has high throughput and very low utilization.

ii. (2 pts) Low throughput, high utilization:

Any type of best-fit scheme using a single list would have these properties.

b) (4 pts) When debugging C code, some of you saw messages during calls to free() such as this one:

```
*** glibc detected ***
/home/ugrads/ugrads1/g/gpb/cs3214/Project5/sysstatwebservice/statd:
double free or corruption (!prev): 0x09b96e50 ***
```

Based on your knowledge of dynamic memory allocators, explain how GNU libc’s memory allocator would be able to implement such diagnostics!

GNU libc’s malloc uses a variant of Knuth’s boundary tag method to record a block’s status. The bits contained in those tags allow GNU’s malloc to implement a number of lightweight consistency checks. For instance, it can check that a block’s ‘in-use’ bit is actually set to 1 when the user issues a free() request for this block.

This concrete error message, btw, occurs when the next block’s ‘previous-in-use’ bit does not indicate that the previous block is in use at the time the user attempts to free() it. GNU malloc optimizes away the footer boundary tag for used blocks by stealing one bit from the next block’s header. In practice, a likely reason for this error message is overrunning the allocated space (and thus corrupting the next block’s header.)

c) (6 pts) The valgrind analysis tool, in addition to detecting common memory corruption errors, analyzes the C heap for unreachable objects. These are objects to which no pointers are found anywhere in the heap. For instance, you may see this output before a program exits.

```
==23082== LEAK SUMMARY:
==23082==    definitely lost: 1248 bytes in 2 blocks.
==23082==      possibly lost: 680 bytes in 5 blocks.
==23082==    still reachable: 172 bytes in 3 blocks.
==23082==         suppressed: 0 bytes in 0 blocks.
==23082== Reachable blocks (those to which a pointer was found) are not
shown.
```

i. (2 pts) Why can't the same method be applied to finding memory leaks in Java programs?

Because Java's garbage collector reclaims unreachable objects to which no pointers exist.

ii. (4 pts) Sketch how you might be able to find memory leaks in Java programs dynamically (e.g., by monitoring a program's activity while it runs).

Memory leaks in Java occur if objects are still reachable, but in fact won't be accessed in the future. This is a crucial difference from memory leaks in C. A dynamic tool could find leak candidates by observing how long ago an object was last accessed by the program. (This is in fact how some tools work.) This will not prove that an object is leaked, but alert the programmer to the fact that reachable objects have not been accessed for a long time.

5. TCP/IP Networking (16 pts)

a) (5 pts) Consider the following flawed attempt at implementing the sysstatd web service in Project 5:

```
void tcp_server_loop()
{
   int  socket;
   struct sockaddr_in serveraddr, clientaddr;
   socklen_t clientaddrlen = sizeof(clientaddr);

   if ((socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_TCP)) < 0)
      return -1;

   bzero((char *) &serveraddr, sizeof(serveraddr));
   serveraddr.sin_family = AF_INET;
   serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
   serveraddr.sin_port = htons((unsigned short)port);
   if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
      return -1;

   while (accept(s, &clientaddr,  &clientaddrlen) != -1) {
      char buf[1024], *url;

      int rc = read(s, buf, 1024);    // read HTTP request
```

```
            parse_http_request(buf, &url);  // parse it
            if (strcmp(url, “/loadavg”) == 0) {
               char reply[1024];
               char *jsonMsg = read_json(“/loadavg”);
               snprintf(reply, sizeof reply,
                  “Content-Length: %s\r\n”, strlen(jsonMsg));
               write(s, reply, strlen(reply));
               write(s, jsonMsg, strlen(jsonMsg));
            } else {
               send_404(s);
            }
         }
      }
```

This code contains multiple errors related to the use of TCP sockets and to the implementation of the HTTP protocol. Identify 5 such mistakes! (Ignore potential memory management problems such as limited buffer sizes or lack of memory deallocation. Assume that code not shown is implemented correctly!)

The code contains at least the following mistakes:

TCP related:

1. A TCP socket is created by passing SOCK_STREAM, not SOCK_DGRAM.
2. There is no call to listen()
3. The socket returned by accept() is ignored; the listening socket cannot be used to communicate with the client
4. Related, the accepted connection is not independently closed
5. Short reads() are not handled when reading the HTTP request

HTTP related

1. Code does not send a status line (e.g. HTTP/1.1 200 Ok)
2. Code does not end HTTP header from \r\n

We also accepted that ‘listenfd’ and ‘s’ were not defined/initialized, which was due to a typo. ‘socket’, ‘s’, and ‘listenfd’ were meant to refer to the same variable.

We did not accept errors such as “void function returns -1” because they are not TCP- or HTTP-related.

b) (4 pts) The widely used Apache webserver can be configured to use different “Multi-Processing Modules” (MPM) that allow an administrator to choose different server models. The default configuration is ‘prefork,’ which implements a non-threaded, pre-forking web server. According to the Apache documentation, in pre-fork mode, “a single control process is responsible for launching child processes which listen for connections and serve them when they arrive. Apache always tries to maintain several spare or idle server processes, which stand ready to serve

incoming requests."
Considering the many possible uses of web servers, give 2 reasons for why the Apache developer might have chosen this model as their default configuration!

Using a multiple process model provides the highest degree of isolation between requests, so that any faults occurring during the handling of one request do not affect others.

Preforking ensures that a server process is already waiting when a request arrives, reducing latency.

While not the most efficient model, it works well in most real-world use cases.

c) (3 pts) XMPP is an Internet protocol used for instant messaging. An XMPP server will typically entertain a very large number of long-lived TCP connections to multiple clients, each of which is only sporadically active. What server model would be most appropriate for this use case, and why?

Neither a thread-per-connection nor a process-per-connection model would scale to very large numbers of connections, so an I/O multiplexing/event-based server model would likely be adopted, using select() or similar notification mechanisms.

d) (4 pts) Network Address Translation. Many home users use multiple machines behind a NAT gateway. In the past, some ISPs threatened to ban users for using such devices unless they signed up for more expensive service tiers. Could an ISP reliably detect that a customer is using a NAT device by monitoring the TCP/IP traffic that is sent from/to the customer's IP address? Justify your answer!

No and yes.

No, because NAT guarantees that all traffic appears to be originating from the single IP address assigned to the external facing gateway of the NAT device. NAT works just because the outside Internet is entirely unaware, and in fact cannot tell, that multiple devices are located inside the private network. As one solution aptly said: trying to guess if a machine is using NAT is like trying to guess the number of rooms in a house by looking at it from the outside.

Yes, because in practice an analysis of traffic patterns could identify patterns that are unlikely to be produced by a single machine. In addition, OS fingerprinting (which measures unique characteristics of individual TCP/IP network stack implementations, such as timing-related characteristics) could be applied to deduce the device type. Such determination is unlikely to be as reliable as ISPs would have needed to enforce such stipulations, which is why so far (to the best of my knowledge) no ISP ever decided to enforce this ban.

6. Essay Question: Virtual Memory Trends (16 pts)

In recent years, the amount of physical memory available in many desktop machines has increased at a faster pace than the working set sizes of many commonly used applications.
Discuss the implications of this trend on application programmers! Consider the impact of this trend separately for each of virtual memory's multiple design features!

Note: *This question will be graded both for content/correctness (10 pts) and for your ability to communicate effectively in writing (6 pts). Your answer should be well-written, organized, and clear.*

No sample answer is provided.

A correct answer should have provided separate discussion of per-process address spaces (and its implication on the linking/loading process and tool chain); isolation and protection (the ability to prevent processes from accessing each other's memory and kernel memory); and resource virtualization (on-demand paging, file caching, page faults, and thrashing).