1 Rules of the Exam

This examination is open book and notes. Calculators are permitted. Networked devices are strictly prohibited. The questions are marked as to their relative value, the exam will be scored out of 100% but is worth 25 points towards your course grade. Relax and try to do what you can.

2 The Problem Set

1. Systems software design (total 20 %): To secure data from misuse or inappropriate access by intruders, modern designers currently apply two approaches:
   - Securing the system: Making successful attacks hard (slow down the intruder)
   - Intrusion Detection: Detecting what the attacker did (so that any damage can be contained and eventually fixed).

   Consider the following (currently used) approaches to security and describe what kinds of kernel features might be needed to implement them. Under Linux there are several toolkits used to monitor system calls.
   (a) Strace - A process can be run under strace by entering

   \[
   \text{strace cmd [parameters]}
   \]

   where cmd is a command and parameters are the run time parameters. The strace command uses a normal kernel and will cause every system call made by that process to be output to the standard output. What run time support is needed to make strace work? (10 %)
   Strace uses the same facilities as the debugger, what it does is trap on every system call by inserting trace instruction, and has a handler that decodes the name of the system call and then writes a textual representation of the system call along with its parameters to the standard output. So the required support mechanism is the ptrace system call (and perhaps the /proc file system) along with the ability to place processes in trace mode.
   It is also necessary to allow communication between the traced process and the process tracing it.

   (b) The Linux Trace Toolkit (LTT) and Intersect Alliance’s Snare are two popular toolkits which instrument the kernel to allow (optional) logging of any and all system calls. LTT is implemented by adding source code to the kernel (using an automated tool called patch) and then recompiling the kernel. Snare on the other hand is compiled separately from the kernel and is loaded at run
time. Describe what kinds of systems software support must be necessary for Snare to work (10 %).

Snare has several components that are used to support it and is designed much like the Basic Security Module (BSM) in SUN Solaris. Snare has the following components:

- A Loadable Kernel Module (LKM) - This module when started preserves the address of every system call in a vector (that is an array of pointers) and replaces system calls with versions that copy information into a queue of buffers. The LKM allocates and manages both the memory for the vector and the queue of buffers. The information includes the system call and its parameters, the process that invoked it, the uid of the process and the time of invocation.

- A Daemon - The daemon runs in the background and is responsible for loading (installing) the kernel module when the daemon starts and removing the kernel module when the daemon terminates. The daemon and the LKM communicate using signals. When the LKM’s buffers fill the LKM signals the daemon, causing the daemon to copy the data from the LKM’s buffer queue to a file on disk.

- A GUI - The GUI interacts with the daemon, and allows easier browsing of the logs as well as supporting a menu driven interface to the control parameters.

2. Readings (total 10 %) Note: Word for word copying from the readings will not get credit, answer IN YOUR OWN WORDS:

(a) Lampson describes using brute force approaches, and Gabriel describes Unix as a “worse is better” design. Does Unix benefit from a worse is better design philosophy (5 %)?

Brute force programming refers to the substitution of simple techniques for more sophisticated approaches. Brute force approaches can at times sacrifice correctness/completeness by handling common cases and ignoring degenerate cases. Unix definitely does this, this is a key part of “worse is better”. Algorithms with worse asymptotic complexity (that is the big O notation) for more sophisticated approaches. Brute force programming succeeds when the program is not run frequently (so that the reduction in development time exceeds the gains in run time), when an increase in response time due to using simple algorithms is not perceptible to a user or when the number of inputs are small relative to the constants hidden in the big O notation. Unix tends to be constructed using simple algorithms because they are “fast enough” and enhancements/optimizations are done when needed.

(b) Your instructor uses Linux, Gcc, and Emacs. This exam was typeset using TeX. Does Pike think that is a cutting edge solution? (5 %)

Pike would say I’m one of these folks who lives in the past and sticks with old tools because of inertia (and to some extent he would be right).

3. Page Replacement Strategies (total 20 %)

(a) Consider the reference string:

\[ \omega = 1, 7, 3, 2, 0, 5, 0, 8, 0, 7 \]

Please be sure that to count all pages loaded into memory as page faults (not just replacements). Circle page faults in your chart (much like in the lecture notes). Recall that I treat any page table miss as a page fault (so initial loads count, not just replacement operations). Assume that you have 3 page frames of memory available, numbered 0, 1 and 2.

i. (5 %) Compute the set of resident pages for \( \omega \) given the (least recently used) LRU algorithm. Give the total number of page faults.

Following the lecture notes we can model the fault behavior of the system as follows:
ii. (5 %) Compute the set of resident pages for the simple clock algorithm, and the number of page faults. Be sure to underline those pages with their use bits asserted and to record the position of the hand. You should assume the use flag is set whenever a page is loaded.

Following the lecture notes we can model the fault behavior of the system as follows (we assume the hand initially points to frame 0):

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_i )</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Fault?</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Faults</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

(b) Your boss hears that your compiler can do static analysis, and can accurately predict the page frame of the next \( n \) references (where \( n \) is a constant).

- What is the best case behavior of this method (justify your analysis)? (5 %)

If we let \( m \) be the number of page frames of main memory available and \( n \geq m \), then the using the lookahead could potentially tell us exactly when all of the resident pages are going to be used in the future if they were all referenced in the lookahead window. This would support the optimal algorithm (OPT). We can slightly tighten the bound if the window is one reference less than the memory size, so that for the case \( n = m - 1 \), all but one resident page, say page \( k \), \( 0 \leq k < m \) could have references in the window. We would know that all pages except \( k \) were used in the next \( n = m - 1 \) references, but page \( k \) is either never referenced again, or is used some time after that. Thus we could still do the Optimal algorithm. If \( n < m - 1 \), the algorithm would lack enough knowledge to accurately do OPT.

- Is this method a stack replacement strategy (show your analysis) (5 %)?

It depends on how the algorithm selects the victim in the case where the lookahead does not provide enough information to do the OPT algorithm. When the lookahead is not sufficient to do OPT (that is less than \( m \) distinct page references appear in the window). The right thing to do if a replacement is performed under that condition is to select a victim from the set of resident pages not contained in the lookahead window. If we use a nonstack algorithm (e.g. randomized replacement or FIFO) to pick at this point, the resulting technique will not be a stack algorithm. However, using a stack replacement algorithm should satisfy the inclusion property, so if say LRU was employed, it could be shown that the resulting algorithm is indeed a stack page replacement algorithm.

4. Systems Program Design and Memory Performance: (25 %) Consider a Shortest Remaining Time scheduler, which organizes the ready queue using an array of tasks as follows:

```c
typedef struct {
    long pid; /* 4 byte task identifier */
```
Suppose that the mean queue length is known to be $N$, $0 < N < \text{NUM\_TASKS}$. There are two possible approaches that you can use to structure the ready queue. The first is to use a priority queue (using a binary heap) allowing insertion of a job requiring $2 \log N + 16$ operations on average, and a routine to delete a job from the ready queue requiring $4 \log N + 64$ operations on average. Alternatively, an unsorted list implementation can be used, which on average requires 20 operations for an insertion and $2N + 4$ operations to delete the minimum value (delete min) from the list.

(a) If each operation costs 1 nanosecond, under what range of values (if any) is linear search and an unsorted array implementation faster than using the priority queue approach? (10 %)

Each task in the system is enqueued and dequeued once for each visit to the CPU, assuming that the probability of the CPU being idle is small. Otherwise it is neither enqueued nor dequeued and choice of implementation does not matter. So what we want to know is when:

\[\text{Operations of Priority Queue} > \text{Operations of Unsorted Array}\] (1)

\[\text{Priority Queue Insertion Cost} + \text{Priority Queue Delete Min Cost} >\]

\[\text{Insertion Cost into Unsorted Array} + \text{Delete Min Cost of Unsorted Array}\] (2)

\[(2 \log N + 16) + (4 \log N + 64) > 20 + (2N + 4)\] (3)

\[6 \log N + 80 > 2N + 24\] (4)

\[6 \log N + 56 > 2N\] (5)

\[56 > 2N - 6 \log N\] (6)

\[28 > N - 3 \log N\] (7)

\[28 > N - 3 \log N\] (8)

\[0 > N - 3 \log N - 28\] (9)

I was not able to find a simple closed form solution for this, equation\(^1\). However, a quick way to check is to try values of $N$ looking for the root using the method of bisection.

When $N = 1$ the RHS is -27. When $N = 64$ the RHS is 18. Trying $N = 32$, the RHS = -11. A few more iterations would eventually show that the root lies between 44 and 45 (near 44.419)\(^2\).

(b) Suppose each operation costs 1 nanosecond, and each cache miss causes a 50 nanosecond delay. Suppose that the priority queue implementation’s insertion and delete min operations cause $\log N$ cache misses (or 1 if $N < 64$), and that the unsorted array has 1 cache miss for an insertion and $\frac{N}{64} + 1$ misses for the delete min operation (or 1 if $N < 64$). For what range of values of $N$ is the unsorted array approach able to outperform a priority queue (15 %)?

This is similar to last time, but now we have to add in cache overhead, and try 2 cases, when $N < 64$ and when $N > 64$. Let’s first try when $N < 64$ (maybe we’ll get lucky

\[^1\] Maple found roots using the LambertW function

\[^2\] Maple also found a root around 0.00155, but that mean queue length is so short that there would be a negligible amount of queuing, so it probably would not matter at that point. The presence of this root was not intended in the original problem design.
since our last answer was less than 64).

\[
\text{Operations of Priority Queue + Cache Misses of the Priority Queue} > \\
\text{Operations of Unsorted Array + Cache Misses of Unsorted Array} \quad (10)
\]

\[
(2 \log N + 16 + 50 \times 1) + (4 \log N + 64 + 50 \times 1) > \\
(20 + 50 \times 1) + (2N + 4 + 50 \times 1) \quad (11)
\]

\[
6 \log N + 180 > 2N + 124 \quad (12)
\]

\[
6 \log N + 56 > 2N \quad (13)
\]

\[
56 > 2N - 6 \log N \quad (14)
\]

\[
28 > N - 3 \log N \quad (15)
\]

\[
28 > N - 3 \log N \quad (16)
\]

\[
0 > N - 3 \log N - 28 \quad (17)
\]

Which exactly matches the solution of the previous problem (since the cache overhead is identical for both algorithms), and thus has the same roots, meaning that when \(1 \leq N \leq 44.4\), the unsorted array implementation is faster.

\[
\text{Operations of Priority Queue + Cache Misses of the Priority Queue} > \\
\text{Operations of Unsorted Array + Cache Misses of Unsorted Array} \quad (18)
\]

\[
(2 \log N + 16 + 50 \times \log N) + (4 \log N + 64 + 50 \times \log N) > \\
(20 + 50 \times 1) + (2N + 4 + 20 + 50 \times (\frac{N}{64} + 1 + 1)) \quad (19)
\]

\[
106 \log N + 80 > \frac{2 \times 64 + 50}{64} N + 124 \quad (20)
\]

\[
106 \log N + 80 > \frac{178}{64} N + 124 \quad (21)
\]

\[
106 \log N - 44 > \frac{89}{32} N \quad (22)
\]

\[
0 > \frac{89}{32} N - 106 \log N + 44 \quad (23)
\]

The method of bisection works here too, it but the numbers are not as straight forward. It can be shown that there is a root at around \(n = 297.3\) for this case, and that for when \(64 < N < 297.3\) the unsorted array is faster than the priority queue. Thus the range of values for the solution is the union of the ranges, telling us that the unsorted array implementation is better when \(1 < n < 44.4\) and when \(64 < N < 297\).

5. Scheduling Algorithms: Processor Affinity (PA) scheduling uses user defined priorities and cache management for scheduling in multi processor systems with shared memory.

Jobs of the same priority are run round robin (RR) on their processor. However, jobs that have high cache miss rates are “banished” to a sort of pool of processors reserved for such misbehaving processes. Banished jobs which show good behavior can get “promoted” back to the pool of jobs with good cache behavior. If a processor in either pool goes idle, it can “beg” for a job from the head of the other processor’s queue. You can assume that each processor runs a special idle process when no jobs are available and that the banish and promote functions will invoke the scheduler if a processor in the target pool is idle (i.e. running the idle process). Suppose all jobs have the same priority (so PA acts like RR on each processor) and you have the routines and data structures:
const int MISS_THRESHOLD = N; /* Upper bound on the number of misses  
for the low miss rate pool */
const int IDLE_PID = 0; /* the PID of the idle process */
const int NUM_POOLS = 2; /* How many classes of jobs */
QueueType q_list[NUM_POOLS] /* one queue per pool +of jobs */
void banish( int pid ); /* process id banished */
int beg(); /* returns pid or -1 if no jobs available */
void promote( int pid ); /* process to promote */
int which_pool(); /* returns 0 if processor is in the  
low cache miss rate pool,  
otherwise returns 1 (high miss rate pool) */
int num_misses(); /* number of cache misses in the last  
CPU burst */
int getpid(); /* returns the pid of the cpu resident job */
void enqueue(QueueType *q_ptr, /* insert a job into a queue */
int pid);
int dequeue(QueueType *qptr); /* Remove a pid from the head of the queue, 
return -1 if the queue is empty */
void runjob(int pid); /* Makes the proces with pid resident  
on the processor that issued this call */

Write C like pseudo-code for the function:

/* Called to schedule a job on a processor */
void schedule_job(int last_pid_run); /* the pid of the last job run or  
-1 if the processor was idle */

You can assume that the operating system ensures that there is never more than one processor is 
schedule job at the same time.

A solution might look something like this:

void schedule_job(int last_pid_run){
    int next_job = dequeue(q[which_pool()]);
    int evicted = FALSE; /* Was the job evicted from its pool? */
    if (last_pid_run != -1){ /* Was the CPU Idle? */
        /* No, check if the job should remain in its current pool */
        if ((which_pool == 0) && (num_misses() > MISS_THRESHOLD)){
            banish(last_pid_run); /* no, belongs in high miss rate pool */
            evicted = TRUE; /* remember that it was evicted */
        } else if ((which_pool == 1) && (num_misses() <= MISS_THRESHOLD)){
            promote(last_pid_run); /* no belongs in low miss rate pool */
            evicted = TRUE; /* remember that it was evicted */
    }
}

if (next_job == -1){ /* queue associated with this pool is empty? */
    /* yes, have to select another pid to run */
    if (evicted){ /* Did we kick the job out of the pool ? */
        next_job = beg(); /* yes, beg for job from the other pool */
        if (next_job == -1){ /* Does the other pool have any jobs? */
            next_job = IDLE_PID; /* no, idle the processor */
    }
} else { /* The last job was not evicted */
        next_job = last_pid_run; /* so might as well keep running it */
}
if (next_job != last_pid_run){ /* are we changing jobs? */
    if (!evicted){ /* was it evicted? */
        /* no, so reinsert into the queue */
        enqueue(q[which_pool()], last_pid_run);
    }
}

runjob(next_job); /* runjob never returns, so no need to return */