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

    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 %)

(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 %).

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

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

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:

   typedef struct {
     long pid; /* 4 byte task identifier */
     long time_remaining; /* how many time units remain for service */
   } TASK_REC;
   TASK_REC task_list[NUM_TASKS]; /* NUM_TASKS is a constant */

   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 the 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 %)
   (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 %)?

5. Scheduling Algorithms: Processor Affinity (PA) scheduling uses user defined priorities and cache man-
   agement 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 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) */
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 pseudocode 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 running schedule_job at the same time.