Solutions for CS 354 Final, Fall 2015

P1(a) 12 pts

Optimal offline: Assuming one knows the future, pick the page to evict
that will be accessed latest/farthest in the future.
3 pts

LRU: Least recently used, looks into the past, and picks the page to
evict that hasn't been accessed the longest time.
3 pts

LRU is similar to optimal but only considers the past (online). If
locality of reference holds, LRU should approximate optimal offline.
1 pts

LRU is not used in practice because of the hardware cost needed to realize
it.
1 pts

Global clock uses 2 bits -- one for access, one for write -- to determine
if a page has been recently written or read. The two bits act as a counter
that is updated by the hardware when a page is accessed or written to,
and the bits are updated periodically by the kernel which decrements the
counter value: 11 -> 10 -> 00 where the first bit refers to access, the
second bit refers to write. When the bits reach 00, a page is marked for
eviction. Pages which are written to (hence requiring writing back to disk)
are given priority over pages that have only been read.
3 pts

Global clock tries to approximate LRU by incorporating recency of access
without the costly hardware associated with LRU.
1 pts

P1(b) 12 pts

The inode kernel data structure has a number of direct pointers to data
blocks that allow identification of sectors on which a small file is located.
For large files, indirect points are used to find the pointers to data
blocks which can take logarithmic overhead. Constant overhead for small
files which are the most frequent.
5 pts

Both reads/writes can be cached in RAM which speeds up subsequent access
assuming locality of reference holds. Write operations require the cached
copy to be periodically written back to disk (or SSD) for reliability/
consistency.
3 pts

Since most files are small and they are the most frequently accessed, a
sequence of file I/O requests serviced by a disk requires that it constantly
spin the disks to locate the sectors where the files are stored. The
resultant
delay (or seek time) can significantly reduce utilization of disk bandwidth.
For large files, disk bandwidth can be more fully utilized.
4 pts

P1(c) 12 pts

Enqueue: The priority of a process acts as an index into the multilevel

feedback
queue arrary. After this constant cost random access, a pointer that points to
the end of the FIFO list of processes of the same priority is used to insert
process. Hence total overhead remains constant.
4 pts

Dequeue: To find a highest priority ready process, search the array from high
to low until a non-empty FIFO list is found. The overhead is linear in the number
of priorities, however, since we treat the number of priorities (60) as
constant, this search incurs constant overhead. After finding a non-empty
FIFO list, dequeue the front element. Hence total overhead is O(1).
4 pts

The multilevel feedback queue data structure is not extensible to fair scheduling
since 1/CPU-time-received can be real number and one that is very large. Since
each process may have received a different amount of cumulative CPU time at an
instance, each process may have a different priority value. Hence the number of
priorities not constant (60) anymore as in Solaris TS scheduling.
2 pts

A simple bound on the overhead is linear (in the number of processes). By using
a heap/balanced tree, the overhead may be reduced to logarithmic.
2 pts

P2(a) 12 pts

Memory trashing occurs in an on-demand paging VM kernel when the amount of memory
needed by processes exceeds available memory to such an extent that (a) processes
constantly page fault, (2) to bring in the missing pages other resident pages need
to be evicted, (3) but the evicted pages are needed shortly thereafter, which
then repeats the vicious cycle.
4 pts

Typical symptoms include high page fault rate, high disk I/O, low CPU utilization
(i.e., CPU is utilized by the null/idle process since processes are blocking on
disk I/O completion).
2 pts

Memory thrashing requires on-demand paging/VM since without it, processes are
loaded in physical memory in their entirety without performing virtual to physical
address mappings.
2 pts

Increasing RAM reduces memory pressure, hence helps alleviate thrashing.
2 pts

Increasing CPU speed has little effect since during memory thrashing the

CPU(s)
are underutilized. CPU is not a bottleneck.
2 pts

P2(b) 12 pts

In a tickless kernel, instead of using a fixed tick value that periodically triggers
a clock interrupt, tick values are allowed to be variable. The kernel maintains an
event queue of future events that it needs to handle, and an interval timer is
programmed to interrupt at the time of the first event.
4 pts

Strength: By only triggering clock interrupts when there is an event that requires
kernel attention, CPU cycles and power consumed by the CPU to run clock handler
instructions is not wasted.
3 pts

Weakness: If the number of events is dense (e.g., tens of events within a 1 msec
time window), then the interval timer is programmed to go off very quickly (e.g.,
microsecond granularity) which can flood the CPU with too many clock interrupts and
resultant overhead. In this case, it is more efficient with respect to CPU load
to process a batch of events every fixed tick.
3 pts

Monitor if the number of future events is dense or not. If not, act as a tickless
kernel to reduce unnecessary clock interrupt handling and conserve energy. If the
number of events is dense, act as a tickful kernel.
2 pts

P2(c) 12 pts

The top half of the lower half acts as "first responder" that interfaces with the
hardware controller to copy data from hardware buffer to kernel buffer (in the case
of read). It runs with interrupts disabled.
3 pts

The bottom half of the lower half handles the kernel processing of the data copied
to kernel memory by the top half. This tends to involve more instructions and thus
take more time. It may run with interrupts enabled (so that the top half can
preempt it).
3 pts

Bottom half design/implementation: (a) borrow context of the current process, or
(b) run bottom half code in the context of a separate kernel process/thread.

2 pts

Pro of context borrowing: less overhead compared kernel thread.
Con of context borrowing: cannot make blocking calls.
(Vice versa for kernel thread.)
2 pts

With DMA, top half instructs DMA how much and where in memory to copy data to.
1 pts

DMA hides a number of interrupts from the CPU (and its interrupt controller) which
frees up the CPU to do other tasks.
1 pts

P3(a) 14 pts

Increase of context switch cost is due to enlarged process context, in particular,
per-process page table and TLB flushing of cached entries this may entail. Upon
context switching, invalidated caches must be flushed and repopulated with valid
entries belonging to the new process.
5 pts

Significant benefits include handling external memory fragmentation, providing a
simplified abstracted memory model to programmers, and facilitating hierarchical
memory in the form of disk/SSD-based persistent storage where parts of process
memory may be kept.
5 pts

If the speed gap between main memory (RAM) and persistent memory (disk/SSD) is
sufficiently narrowed, the I/O operation needed to bring in a missing page may be
fast enough so that a context switch is not needed. In this case, the kernel
can just return to the current page faulting process after the page has been
read into RAM. Part of the kernel processing for demand paging may even be
delegated to hardware if standardized.
4 pts

P3(b) 14 pts

When XINU's clock interrupt handler runs, to update the time remaining to wake up
sleeping processes, only the process at the front of the list needs to be updated
which incurs constant overhead.
4 pts

The elements of the sleep queue would be generalized so that an additional field
specifies the type of event (sleep or time slice). The enqueue/dequeue operations
for sleep work as before. For time slice, instead of using the global

variable
preempt to decrement by 1 msec each time the clock interrupt handler runs,
when
a process is context switched in, a new event is enqueued into the unified
event
queue (a delta list) with the process's time slice as the key. If the current
process depletes its time slice, this will be detected by the kernel because
of
the time slice event that was inserted into the event queue. If the current
process does not deplete its time slice and makes a blocking system call,
then
the the time slice event is removed from the event queue. As an optimization,
to avoid searching through the event list to find the time slice event (which
could be linear time), a field in the process table may be added that
contains
a pointer to the time slice event. If the delta list is made bidirectional,
the
event queue can be updated with constant overhead. The clock interrupt
handler,
as before with the sleep queue, decrements the timer value of the front
element
of the event queue when it runs every millisecond.
6 pts

The turn the above into a tickless kernel, instead of programming the
internal
timer to go off every 1 msec, the clock interrupt handler is programmed to
interrupt
after the time duration of the front element in the event queue. This way,
only
when events need to be processed is a clock interrupt triggered.
4 pts

Bonus

Without isolation/protection, bugs in app code or malicious user code, would
both
be able to compromise the entire computing system. Such a system is not
reliable
and of very limited outside of specialized computing environments (e.g.,
embedded
systems).
6 pts

When XINU responds to a send() system call, it checks if the recipient
process has
registered a callback function. If so, the callback function is invoked at a
future
time when the scheduler decides to context switch in the recipient process as
the
last step of the context switch routine. Hence the callback function runs in
the
context of the recipient that registered the callback function, and by doing
it as
the last step, we are able to run it after switching back to user mode (even
though
in XINU there is no such separation).
6 pts