CS354 Final Solution, spring 2017

P1(a) 25 pts

A delta list is a data structure that stores timer events where time stamps of consecutive events represent their difference.
3 pts

enqueue: linear
dequeue: constant
4 pts

When updating the time stamps of events, all that is needed is to decrement the first event in the delta list, hence constant overhead.
3 pts

sleep dequeue: lower half of kernel
sleep enqueue: upper half of kernel
2 pts

sleep dequeue: clock interrupt handler clkdisp(), clkhandler()
(which calls other kernel functions)
sleep enqueue: sleepms()
2 pts

A global variable preempt is decremented at each tick by the clock interrupt handler.
3 pts

Overhead: constant
2 pts

When a new process is context switched in, insert its time slice event into the delta list (each event needs to be marked to distinguish between the two types of events). When a process blocks, then remove its time slice event from the delta list.
4 pts

A XINUSIGXTIME timer event is inserted into the delta list. A signal handler/callback function for XINUSIGXTIME is invoked when the timer expires.
2 pts

P1(b) 25 pts

Real-world processes w.r.t. their memory demand follow a "mice and elephants" property: most processes have small memory demand, a few have very large demand. Since most processes exert small memory demand, there is no need to maintain a page table that translates all possible virtual addresses into physical addresses. (That is, we can compress/reduce the size of page tables of most processes.)
4 pts

12 bits are used to specify the offset within a 4 KB page, hence they do not change. The remaining 20 bits are split into two groups, say, 10 bits and 10 bits. The outer (i.e., most significant) 10 bits represent 2^10 regions/blocks in 4 GB memory each of size 4 MB. The inner 10 bits represent 2^10 regions within one of the outer blocks.
4 pts

A page table cache, TLB, is provided that enables fast page table lookup of frequently used/translated addresses. In some systems, hardware manages TLB. In others, kernel does.
4 pts

A page fault, an interrupt, is raised when a page needed by a process is not resident in memory (i.e., RAM). Kernel handles page fault interrupts.

4 pts

Rationale: Since disk (to a lesser extent but still the case for SSD) is significantly slower than RAM, the hardware managing page faults would imply that the system busy waits until disk I/O completes to bring in the missing frame. This wastes CPU time since other ready processes are not executed. Hence it makes more sense for the kernel to handle page faults where it context switches out the process that page faulted and context switches in a ready process to utilize the CPU/core.
4 pts

With the help of a crystal ball, a page that will be used farthest in the future, i.e., that won't be needed for the longest time (Belady's criterion) is optimal to select w.r.t. reducing the total number of page faults incurred by a process.
3 pts

If the optimal page according to Belady had been modified (i.e., written to) and there is a slightly suboptimal page that has only been read but not written to, then due to the need to write the modified page back to disk/SSD to achieve coherence/ consistency and its associated overhead, evicting a suboptimal but not modified page may be preferred.
2 pts

P2(a) 25 pts

Top half of lower half handles fast chores with interrupt disabled.
Bottom half of lower half handles slower interrupt handling chores with interrupts enabled.
4 pts

Top half: Interface with device controller, copy data from device buffer to RAM (and vice versa).
Bottom half: Perform processing that may take additional time such as decapsulate network packet headers, perform checksums, etc.
4 pts

One: Borrow context of the current process to run bottom half.
Two: Use separate kernel process/thread to run bottom half.
4 pts

Pro of context borrowing: Lower overhead.
Con of context borrowing: Cannot make blocking calls.
Vise versa for kernel process implementation.
4 pts

Web cam streaming results in frequent interrupts (e.g., one interrupt per 125 microseconds) which would put a significant burden on CPUs taking away time to run application processes.
2 pts

Top half w/o DMA support copies data from device buffer to kernel buffer in RAM (and vice versa). With DMA support, the DMA controller handles copying. The top half instructs DMA what to do (what and how much to copy to where in RAM).
2 pts

Interrupt results in top half of the lower half to be executed. Without DMA support, the top half copies the data to a kernel buffer allocated for input from the device in question. With DMA support, the DMA controller handles the copying operation. If further processing of the arrived data is required, the bottom half of the kernel is executed. After the bottom half has completed it will unblock the process that called read() and call the scheduler. The scheduler, at some point, may decide to context switch in the process that called read() which is waiting in ready state. When this process runs, the upper half of the kernel resumes execution and copies data from the kernel buffer to user space buffer. The read() system call returns which puts the process back in user mode and it accesses the

data in its user buffer.
5 pts

P2(b) 25 pts

The app generates approximately 1000 packets per second.
4 pts

Although the app process calls sleepmicro() with 500 microseconds as argument after sending out a packet, the tickful Linux kernel only raises a clock interrupt every 1 millisecond. Hence, on average, after 1 msec the sleeping process is woken up and removed from the sleep queue. The loops starts anew, the process sends out another packet and calls sleepmicro() with 500 microseconds as argument. Thus the process manages to send out 1 packet per millisecond which results in a data rate of 1000 packets per second
6 pts

When Linux is configured as a tickless kernel, upon calling sleepmicro() in the loop the upper half of Linux inserts a sleep timer event for the app process in the sleep/event queue and programs a clock (e.g., programmable interval timer PIT in x86) to raise a clock interrupt after 500 microseconds. Thus after approximately 500 microseconds
an interrupt will be triggered, the lower half of the kernel wakes up the sleeping app process which sends out another packet and calls sleepmicro() again. Hence, the app is able to achieve the desired 2000 packets per second data rate.
6 pts

A potential drawback is that if processes invoke many timer events such as sleep where the events are spaced close together in time (e.g., microseconds apart), the number of clock interrupts that the kernel must handle per second is very large and may exert significant overhead that takes away CPU cycles from app processes.
4 pts

An alternative approach uses a tickful Linux kernel but sets its tick value to a smaller number such as 500 microseconds. That addresses the needs of this application.
3 pts

However, an application that needs an even faster data rate (and is coded in the manner outlined above) will face a similar problem as before. Also, even if applications
do not require fine granular timer events, the kernel will handle interrupts every 500 microseconds which incurs unnecessary overhead and consumes energy.
2 pts

Bonus 10 pts

Locality of reference which allows caching to improve performance.
2 pts

Mice and elephants property of process memory demand which allows the size of page tables of most processes which are mice to be significantly reduced.
2 pts

Mice and elephants property of file sizes which allows file systems (e.g., UFS) to use indexing structures where small files are accessed using direct pointers in constant time and only large files incur logarithmic overhead through look-up of indirect pointers in a file system's indexing tree.
2 pts

Performance of systems would significantly degrade.
2 pts

Locality of reference may be considered the most important since it is far reaching. That is, it impacts caching of data and page tables in virtual memory

management, page replacement policies that try to keep recently used pages in RAM, caching of files in RAM, etc.
2 pts