Review for final

1. Lab1: Get information
2. Lab2: Change priority
3. Lab3: Fair scheduling
4. Lab4: Go to the call back function
5. Lab5: Call back function for receiving message
6. Lab6: Alarm

7. The Solaris TS scheduler implemented in lab3 achieves constant (i.e., O(1)) overhead with respect to time complexity. Explain why this is the case by considering enqueue and dequeue operations separately

   Enqueue: The priority of a process acts as an index into the multilevel feedback queue array. 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.
   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).

8. In lab2, a form of fair scheduling was implemented where the priority of a process was set to 1/CPU-time-received. Thus a process that has received relatively little CPU time would get a higher priority than a process that has consumed more CPU time. Why is a multilevel feedback queue not extensible for fair scheduling?

   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.

9. What is the scheduling overhead associated with fair scheduling?

   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.

10. What is the delta list?

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

11. What is its time complexity?

    enqueue: linear
    dequeue: constant

12. Why is a delta list well suited for managing kernel timer events including sleep events?

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

13. Which part of a kernel handles sleep dequeue and enqueue events?
    sleep dequeue: lower half of kernel
    sleep enqueue: upper half of kernel

14. In XINU, what specific kernel function calls are invoked by the two parts?
    sleep dequeue: clock interrupt handler clkdisp(), clkhandler(),
    (which calls other kernel functions)
    sleep enqueue: sleepms()

15. How is time slice bookkeeping(countdown) of the current process managed in XINU? What is its overhead?
    A global variable preempt is decremented at each tick by the clock interrupt handler.
    Overhead: constant

16. How can time slice bookkeeping be carried out by using a unified event queue that keeps track of both sleep and time slice depletion events via a single delta list?
    The elements of the sleep queue would be generalized so that an additional field specifies the type of event (sleep or time slice).
    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 in the newly added field).
    When a process blocks, then remove its time slice event from the delta list.
    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.

17. XINUSIGXTIME that represents a process's wall time exceed event was added to XINU's signal handling subsystem. How can XINUSIGXTIME.
    A XINUSIGXTIME timer event is inserted into the delta list. A signal handler/callback function for XINUSIGXTIME is invoked when the timer expires.

18. Why is isolation/protection an essential feature of general-purpose computing systems? How did isolation/protection influence the design and implementation of asynchronous IPC with callback function in lab4?
    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).

    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).

19. In the asynchronous IPC with callback function lab, we considered an implementation in XINU where the principle of preserving isolation/protection—to the extent feasible—was followed even though XINU does not support isolation/protection. How was this done?

    In XINU where upper and lower halves run with interrupts disabled, the design goal was to insert run-time hooks so that a callback function was executed in the context of the process that registered the callback function after it resumes and just after interrupts are restored/enabled.

20. Suppose a process that registered a callback function for message reception has low priority and a message is transmitted to this process. Under XINU's legacy scheduler, will be callback function be executed in a timely manner?

    If the process has low priority, it may take a while for the scheduler to select this process to run. Since the callback function is executed in the context of the process that registered it, actual message receive by the process may be significantly delayed.

21. What about the case of the Solaris TS scheduler?

    There is no fundamental difference with XINU. Under the assumption that priority is low, significant delay may be incurred.
    [One difference is that in Solaris I/O-bound process tend to get a priority boost so the assumption of low priority may not hold.]

22. Describe deadlock by providing a simple example involving processes and semaphores.

    Two processes p1 and p2, two semaphores s1 and s2.
    Sequence of events:
    p1 runs and acquires s1; p2 runs and acquires s2;
    p1 runs and tries to acquire s2 which causes it to block;
    p2 runs and tries to acquire s1 which causes it to block.

23. How would a kernel go about detecting deadlock in app processes?

    Cycle detection in resource graph comprised of nodes (processes and semaphores) and directed links (has-semaphore and want-semaphore edges).

24. What is the resultant overhead?

    Cycle detection takes linear time (e.g., breath-first expansion).

25. What is the approach adopted by modern operating systems when it comes to dealing with deadlocks, and why is the approach viewed as viable?

    Modern kernel adopt the Ostrich approach: ignore the problem and let users/apps in user space handle it. This approach is viable since the impact of a deadlock is limited to the deadlocked processes.

26. Beyond overhead, what is the rationale that justifies operating systems not getting involved in deadlock detection?

    The consequence of deadlock is limited to processes that deadlock, i.e.,applications that have a deadlock problem. Outside of consuming some system resources (e.g., process table entries, swap space), the impact does not spread to the rest of the system which is in keeping with the principle of preserving isolation/protection.

27. What is memory thrashing?

    In memory thrashing, a memory-intensive process (or processes) requires memory (i.e., number of pages) that far exceeds physical memory. As a consequence, even with locality reference, pages that have been recently evicted are needed again in the near future which leads to a cycle of elevated page faults.

    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
    (1) 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.

28. What are its characteristic and observable performance symptoms?

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

29. Is memory thrashing possible if virtual memory is turned off (i.e., the default configuration of the XINU kernel we have been using)?

    Not possible. 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.

30. Would increasing CPU processing power(speed) help alleviate the problem?

    Increasing CPU power does not help since during thrashing a CPU is idle most of the time (unless there are unrelated CPU-intensive processes to run) waiting for disk I/O to complete to load a missing page. CPU is not a bottleneck.

31. Would increasing main memory help? Explain your reasoning.

    Increasing memory helps since this will decrease the likelihood of a page fault. If memory demand is still excessive compared to physical memory, this will serve to delay the onset of thrashing.

32. What are the main components of context-switch cost under process management (material covered before the midterm)?

    Saving registers onto process stack, updating process table information, etc.

33. What additional costs are introduced under memory management (material covered after the midterm)?

    Saving more process state (per-process page table register values), flushing of L1, TLB, L2 cache to avoid aliasing.

34. Which cost may be higher and why?

    The latter is considered higher due to misses (TLB, page faults, L1, L2) that result from flushing of caches. And initial cache misses when populating the caches with instructions/data/TLB entries referenced by the context switched in process.

35. What is a TLB miss?

The virtual address to be translated cannot be found in the TLB cache that holds part of the process's page table.

36. In computing system where TLB misses are handled by an operating system, what are the steps undertaken by a kernel to resolve a TLB miss?

Locate in RAM the process's page table. Assuming the process table is locked into RAM, read the missing entry from RAM into TLB. Resume (i.e., re-execute) which will now result in TLB hit.

37. Assuming a disk is used for persistent storage, when is disk I/O needed to handle a TLB miss?

Disk I/O is needed if some page tables are located on disk.

38. What is a common method used by kernels to avoid such disk I/O?

Lock page tables into RAM.

39. Why are 2-level page tables relevant for avoiding disk I/O for TLB misses?

2-level page tables decrease the total space required by all process' page tables in RAM. Hence it reduces the need to put some page tables on disk.

40. What happens after a TLB miss has been resolved?

The content of the physical address translated from virtual address is accessed in L2 or RAM.

41. The context switch cost increased significantly when we introduced virtual memory-based demand paging. What are the reasons for that?

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.

42. Despite the increased context switch cost and extra hardware expenditure incurred to build VM support, what benefits enabled by VM outweigh the negatives?

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.

43. What is external fragmentation? Give an example of external fragmentation in memory management.

In external fragmentation, there is, in total, sufficient memory to satisfy a dynamic memory allocation request, however, the free memory is fragmented so that contiguous memory allocation is not possible.

Example: Process A wants 100 MB memory (contiguous) but RAM has only two free regions: one of size 70 MB, another of size 30 MB. Although the total free memory matches the 100 MB, it is not contiguous hence system call will fail.

44. What is the solution utilized by modern kernels, and why is hardware support an essential component?

    Indirection that maps virtual (or logical) addresses to physical addresses achieves defragmentation, but the translation must be done with hardware support since software indirection would incur too much overhead.

45. What is the conceptual solution adopted for this problem and why is hardware support essential to make the solution viable?

    Indirection that uses a table to translate virtual addresses into physical addresses. For the above example, in the virtual space, the fragmented 70+30 MB can be made to seem contiguous.

    Since this is done for every memory reference that needs virtual to physical address translation, it must be fast and cannot be handled by kernel software.

46. What is the typical role played by L1 caches in today's systems with respect to memory referencing, and why does it help mitigate the external fragmentation problem?

    An L1 cache stores frequently accessed instructions/data in small but fast memory that is addressed by a CPU using virtual addresses. Since virtual addresses are used, assuming a cache hit, there is no need to perform additional hardware assisted address translation with the help of TLB.

47. Page table, in general, can exert significant memory pressure in demand paging kernels. What does memory demand of real-world processes look like, and how does this help reduce page table memory pressure?

    Real-world processes with reference to 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 address.

48. In a 32-bit architecture with 4KB pages, describe how a 2-level page helps reduce memory consumption of page tables.

    12 bits are used to specify the offset within a 4KB page, hence they do not change. The remaining 20 bits are split into 2 groups, 10 bits and 10 bits.
    The outer (most significant) 10 bits represent 2^10 region.blocks in 4GB memory each of size 4MB. The inner 10 bits represent 2^20 region within on of the outer block.s

49. To speed up virtual-to-physical address translation using page table look up, what hardware support is provided and who - kernel or hardware - manages this support?

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

50. What is a page fault and who - kernel or hardware - handles it in modern kernels?

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

51. Why are page faults handled by software (i.e., kernel) and not hardware?

    Assuming persistent storage that is significantly slower than main memory such as hard disks, the time needed to load a missing page into RAM is very slow. Hence, to not idle a CPU for an extended period, the process that page faulted is context switched out and a process that can make use of the CPU context switched in. Scheduling and context switching take many instructions to complete and are the purview of software (i.e., kernel).

    The rationale for responding to page faults via interrupt handlers as part of a kernel is: fetching the missing page from swap space incurs significant slow-down (e.g., disk is about 1000 times slower than RAM). Hence context switching out the process that page faulted so that a CPU may be utilized by other processes while disk I/O is on-going is necessary to not idle the CPU. Since context switching entails scheduler intervention, this task is well-suited for kernel execution.

52. What is the relational for this design?

    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 complete to bring in the missing frame.

53. What is the optimal offline page replacement strategy (i.e., Belady)?

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

54. What is the LRU policy and how is it related to the optimal offline strategy?

    LRU: Least recently used, looks into the past, and picks the page to evict that hasn't been accessed the longest time.
    LRU is similar to optimal but only considers the past (online). If locality of reference holds, LRU should approximate optimal offline.

55. Why is LRU not used in practice?

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

56. What is global clock and how is it related to LRU?

    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.

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

57. Suppose a page fault occurs and memory is completely filled, (there's no free frames). If a kernel had crystal ball and could see into the future, which page would be optimal to evict?

    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.

58. If this page had been modified (written to), while in resident in RAM, would it still be the best page to evict to disk or SSD?

    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.

59. In what sense can LRU be viewed as approximating Belady's optimal page replacement?

    LRU selects a page that has not been accessed for the longest time (i.e., farthest in the past) to evict. Under locality of reference, pages in the recent past and near future will significantly overlap. Hence both LRU and Belady tend to evict pages that do not satisfy locality of reference. In this sense, LRU approximates Belady using information available from the past.

60. What is global clock and how is it related to LRU

    Global clock uses periodic sweeping across pages to give modified pages two chances before being selected for eviction and pages that have been only read a single chance. Global clock can be viewed as approximating LRU by using the time to make a full sweep as a measure of recency. Pages that have not been accessed (read or written) during this time are marked for eviction.

61. In modern operating systems, why are pages not evicted at the time of page fault interrupts?

    To reduce the overhead/delay incurred when a page fault occurs, it is preferable to evict pages via periodic sweeping (asynchronous to page faults) so that some number of free frames are available. By doing so, when a page fault occurs, only reading in a missing page is required (but not determining a page to evict or writing back a page if it has been modified).

62. Page faults in kernels with disk-based file systems induce a context switch because disks are significantly slower than main memory (i.e., RAM): a process that page faulted cannot continue execution until the missing page is brought in. Suppose in the future the speed difference between RAM and flash memory (i.e., SSD)—used increasingly in place of disks—narrows to the extent that there is only a negligible gap. How might page fault handling be done differently in such an environment?

    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.

63. What is the motivation for diving the lower half of a kernel into two subsystems—top and bottom halves—in today's operating systems?

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

    Lower half kernel chores triggered by device interrupts need to be promptly completed. Some devices inherently require more processing by the lower half than others. By dividing the lower half into top (fast) and bottom (less fast) halves, the top half performs urgent chores that don't take much time such as copying bits from device buffer to kernel memory with interrupts disabled (to prevent data structure corruption). The bottom half performs slower chores with interrupts enabled so that when future interrupts are raised they can preempt the bottom half so that responsiveness is preserved.

64. What are the tasks assigned to a top half?

    Top half: Interface with device controller, copy data from device buffer to RAM (and vice versa).

65. How does a kernel contribute to degradation of video quality

    If the upper half does not read the kernel buffer sufficiently frequently, the buffer may build up and overflow thus losing video information.

66. What are the chores assigned to a bottom half? Real world example?

    Bottom half: Perform processing that may take additional time such as decapsulate network packet headers, perform checksums, etc.

    TCP/IP packet processing. Packets are copied by top half (assuming no DMA support) from device buffer to kernel memory by top half. Further processing is required by the bottom half to strip headers, calculate checksums, etc. to ready the payload of TCP for copying to user space buffer.

67. What are the two design choices for implementing a bottom half?

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

68. What are their pros/cons?

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

69. Why is hardware support in the form of DMA important for delivering adequate performance in web cam streaming applications?

    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.

70. What is the main performance benefit that DMA brings?

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

71. How does the role of the top half change when DMA is used to assist in video streaming?

    Top half without 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).

    In isochronous mode used for video streaming, an interrupt is trigged every 125 microsecond when a packet containing part of image data arrives. Without special hardware support (i.e., DMA), a CPU would be overwhelmed executing lower half code to service interrupts. DMA shields the CPU from interrupts by performing data copy from device buffer to memory on its own which allows the CPU to perform other tasks. After a certain number interrupts stemming from streaming packets have been copied does the DMA let the next interrupt through so that the CPU runs its lower half which instructs the DMA what to do next (usually to handle the next N packet interrupts). Thus interrupt load of the CPU is significantly reduced.

72. Suppose a process makes a read() system call to read data from an interface (e.g., USB) which is to be copied to a user space buffer. Assume that at the moment the read() system call is invoked, the requested data has not arrived and the kernel buffer allocated for the device is empty. Since read(), by default, is blocking, the process that invoked it is context switched out. At some point in the future, the requested data arrives at the device interface which triggers an interrupt. Describe the subsequent sequence of events, including who the producers and consumers are of kernel and user space buffers, that results in delivery of the data to user space buffer.

    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.

73. We discussed producer/consumer queues in the context of process coordination/synchronization as part of IPC. That is, a process acts as a writer of a FIFO queue and another process acts as its reader. Suppose we are considering device I/O where a user process performs a read() system call, and the upper and lower halves coordinate with the help of a shared FIFO queue to copy data bits from a device buffer (e.g., Ethernet card) through the shared kernel FIFO queue to a user space buffer whose pointer is provided through read(). Explain why the techniques studied in the context of IPC apply for device I/O. Use an example to specify who the writers/readers of the kernel FIFO queue and user space buffer are. For this example, there is no need to subdivide the lower half into top and bottom halves.

    We discussed using counting semaphores to achieve sharing of producer/consumer queues for IPC between a writer/producer and reader/consumer (i.e., two processes).

    When performance device I/O using read(), a process p1 makes a read() system call which is handled by the upper half of the kernel. In general, data requested by read() may not be available in which case p1 is context switched out and another process p2 is context switched in.

    Suppose while p2 is running, data arrives at a device interface which raises an interrupt.

    The lower half of the kernel (ignoring the details of top and bottom halves) copies data from device buffer to the kernel FIFO queue that is shared between upper and lower halves of the kernel. That is, it acts as the writer/producer of the shared FIFO queue.

    The lower half then readies the p1 which was blocked. Key point: the lower half kernel code that writes to the shared FIFO queue is process p2.

    At some future time, the scheduler selects p1 to run. p1 resumes where it left off -- in the upper half of the kernel performing the read() system call -- which entails copying the data from the shared FIFO queue to a user buffer whose pointer is provided by read().

    That is, the upper half is the reader/consumer of the shared kernel buffer and writer/producer of the user buffer.

    The process that performs said reading is p1. Hence in this example of read() device I/O, process p2 which executes the code of the lower half writes to the kernel FIFO queue and process p1 which executes the code of the upper half reads from the kernel FIFO queue. That is, at the end of the day, two processes are communicating using a shared buffer and the counting semaphore techniques of IPC carry over.

74. What differentiates tickless kernels from tickful kernels?

    Tickful: a fixed tick value which determines a periodic interval at which system timer clock interrupts are raised.

    Tickless: no such fixed tick value, i.e., tick can vary over time. 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.

75. Why might a tickless kernel be well-suited for mobile devices if they are not actively used?

    If a mobile device is not actively used, it may not have enough future events to handle. In a tickful kernel, e.g., with tick value 1msec (as in XINU), the mobile would repeatedly run lower half code every 1msec for no useful purpose which expends energy and reduces battery charge. By using a tickless kernel, system timer clock interrupts are raised only when there is a chore to be performed by the kernel's lower half.

    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.

    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.

76. How might a hybrid design that aims to achieve the best of both worlds work?

    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.

77. We know how XINU's lower half is designed to handle clock interrupts as a tickful kernel. Describe how you would modify XINU's lower half so that it operates as a tickless kernel. Consider two tasks only: sleep queue and time slice management.

    For sleep and time slice events, manage them using a unified delta list.

    In the case of time slice management, XINU uses a global variable preempt to decrement it by 1msec each time the clock interrupt handler runs.

    Instead, insert into the unified delta list when a process is context switched in an element that specifies that it's a time slice event with the time value (i.e., key of the delta list) being the initial quantum.

    All the other elements in the unified delta list are sleep elements.

    XINU programs an interval timer to raise an interrupt after X msec specified in the first member of the delta list.

    The first clock interrupt occurs after X msec at which time XINU processes the delta list element at the head (and any element following it with 0 offset).

    Then XINU programs the interval timer to raise an interrupt after Y msec where Y is the positive time offset of the next delta list element.

78. Suppose an app programmer is tasked with writing an app that sends out data packets through a network interface using a system call, sendpacket(), where packets are to be spaced 0.5 msec apart. This should result in 2000 packets being sent out during a 1 second time interval. Assume CPU and network interface speeds are more than adequate for handling this load. The app programmer proceeds to write code whose structure is a loop wherein sleepmicro()—a system call that sleeps for a given number of microseconds—is called with argument 500 after a packet is sent out using sendpacket(). The app runs on a tickful Linux kernel with the tick value set to 1 msec (as in XINU). What approximate data rate in units of packets per second (the goal is 2000) does the app actually generate when it is executed? Explain how you arrive at your answer.

    The app generates approximately 1000 packets per second.
    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.

79. Suppose a colleague suggests to this programmer that configuring Linux as a tickless kernel may help achieve the desired data rate. Why is this a meaningful solution?

    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.

80. Given the benefit that a tickless kernel may bring, what is its potential drawback?

    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.

81. What is an alternate solution that preserves the default tickful mode of Linux but reconfigures it?

    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.

82. What is its potential drawback?

    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.

83. What is the key difference between XINU and UNIX file systems when it comes to locating the data blocks of a file?

    XINU: Linear cost to access data blocks from iblock structure.
    UNIX: Constant cost for small files using direct pointers, logarithmic cost using indirect pointers.

84. What are three real-world properties (two pertain to processes) discussed in class that operating systems exploit to significantly improve performance?

    1. Locality of reference which allows caching to improve performance.

    2. 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.

    3. 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.
    (90% of files are small and take up 10% of space, 10% of files are large and take up 90% of the space)

85. Locality of reference example

    File caching in RAM.
    Page replacement policy in VM through global clock that mimics LRU.

86. What is the key difference between the Xinu and UNIX file systems with respect to performance?

    Accessing a byte in a file in Xinu FS requires linear overhead to search through the linked list index blocks. Doing so in a UNIX FS requires logarithmic overhead due to the tree structure of the inode pointers comprised of indirect, doubly-indirect, and triply-indirect pointers.

87. What additional optimization is implemented in UNIX file systems to address the idiosyncrasy of file sizes ("mice and elephants") in real-world file systems?

    By the 90/10 rule, most files in real-world file systems are small. UNIX FS deals with this property by allocating 10 direct pointers that reference file data up to 5KB assuming 512B block size.

88. How does mice and elephants influence the file system?

    Although traditional/UNIX file systems use indirect (single, double, triple) pointers to index data blocks using a tree of logarithmic depth for large files, a number of direct pointers are kept in the i-node to facilitate direct access to data blocks of small files without traversing the tree.

    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.

89. Why are kernels able to handle read requests more efficiently than write requests during file I/O?

    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.

90. Suppose for a disk-based file server with disk bandwidth B bytes per second that is busy serving client requests, it is found that only 15% of B is being utilized. Given that the server fields many requests, what may cause its disk bandwidth to be severely underutilized?

    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.

91. How would the XINU file system fare under real-world file sizes?

    XINU's file system uses a linked list of index blocks each containing a fixed number of pointers to data blocks. As such, the overhead of finding a data block is linear in the size of the file.

92. What would happen for kernel design and system performance if we lived in a world where these properties did not hold?

    Performance of systems would significantly degrade.

93. Which of them is the most important, and why?

    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.

94. In Lab 2, asynchronous nonblocking IPC with callback function was implemented in Xinu. Although Xinu does not implement isolation/protection, what design choices were followed to achieve isolation/protection which would have been necessary had the implementation been ported to Linux/Windows? Asynchronous nonblocking IPC with callback function can be adopted to device I/O where, instead of a writer process sending a message to a receiver process, an interrupt handler for a received message (e.g., Internet packet) copies the message to a receiver process. What is the reason that the technique for inter-process communication carry over to device I/O?

    The two design choices for achieving isolation/protection were: execute the user supplied callback function
    1) after switching back to user mode from kernel mode,
    2) switching to the context of the process that registered the callback function.

    The lower half of a kernel which handles interrupt related processing, is either executed while borrowing the context of the current process or as a separate kernel thread. In both cases, the lower half which acts as a writer (or reader) is just another process. Hence, the coordination required to achieve device I/O is again between a pair of processes as in IPC.

95. We noted only two exceptions where a kernel's service was implemented as a process, i.e., kernel thread. What were these two exceptions?

    Two exceptions:

        pageout kernel process
        bottom half of lower half implemented as kernel thread.

96. What is the main rationale for constructing a kernel as a library of function calls instead of a pool of kernel threads?

    The main rationale is that context borrowing incurs less overhead than context switching between processes/threads. Also, if kernel chores are implemented as threads, IPC is needed for coordination which incurs additional overhead. Since speed/efficiency is a major concern in kernel design, context borrowing using function libraries has dominated.