CS354 Midterm1 Solution, spring 2017

P1(a) 15 pts

Upper half: respond to system calls.
Lower half: respond to interrupts.
4 pts

Disable all interrupts.
2 pts

IF flag in the EFLAGS register is set to 0.
2 pts

XINU does not implement trap (i.e., system calls are regular function calls) hence the upper half runs without isolation/protection.
4 pts

upper half: sleepms()
lower half: clkdisp()
neither: resched()
3 pts

P1(b) 15 pts

The last action associated with a running process is used to determine if a process's recent behavior is CPU- or I/O-bound.
If a process makes a blocking system call: I/O-bound.
If a process depleted its time slice: CPU-bound.
4 pts

Solaris:
If CPU-bound: decrease priority, increase time slice.
If I/O-bound: opposite.
4 pts

A process that is I/O-bound and relinquishes the CPU voluntarily without depleting its time slice is "rewarded" by promoting its priority which makes it likely to be scheduled sooner than CPU-bound processes when it becomes unblocked in the future. This enhances its responsiveness which is beneficial to interactive applications. Since it does not need much CPU time, the time slice is kept small.
2 pts

On the other hand, a process that is CPU-bound and depletes its time slice has its priority lowered to allow I/O-bound processes timely access to CPU cycles. Its time slice is increased to reflect its CPU-bound nature. This does not starve I/O-bound processes since their higher priority allows them to preempt the lower priority CPU-bound process when they become unblocked.
2 pts

In fair scheduling where we use CPU usage as an indicator of priority (smaller CPU usage corresponds to higher priority), I/O-bound processes whose CPU usage tends to be small get higher priority compared to CPU-bound processes.
3 pts

P1(c) 15 pts

In Linux/UNIX and Windows, by default, a process has its own text, data, heap, and stack area in memory that is not shared with other processes. In XINU, text, data, and heap are shared, and stack areas of processes are placed adjacent to each other.
4 pts

XINU memory layout corresponds to the layout of a single heavy-weight process that has multiple light-weight processes which share text, data, and heap, but have individual/separate stacks per light-weight process.
3 pts

Although unlikely due to the large memory afforded in today's systems, a very large number of nested function calls could cause the stack area of a process to overflow into its heap area.
3 pts

Depending on the stack size specified when spawning a process using create(), a moderate number of nested function calls could overflow into the stack area of another process since in XINU's memory layout stack areas of processes are placed adjacent to each other.
3 pts

We may use stack segment (SS) memory bound checking provided by x86 to prevent a process from exceeding the memory allocated to its stack area. By default, the base is set to zero and the limit to maximum memory by kernel software so that memory bound checking is effectively disabled.
2 pts

P2(a) 17 pts

Hardware support features:
privileged/non-privileged instructions
user mode/kernel mode
trap instruction/mechanism to enter kernel mode
memory protection
4 pts

Software support:
per-process kernel stack
2 pts

An attacker can create two processes that share their memory. One process makes a blocking system call that puts the process in kernel mode and while in kernel mode pushes stacks frames when making kernel function calls into the process's run-time stack. The second process, because it shares memory with the first process, access the run-time stack and changes the return address in one of the kernel stack frames to point to malware. When the blocked process resumes, it eventually jumps to malware code while in kernel mode.
6 pts

No. Super-user/root are concepts that relate to the notion of users of an operating system, i.e., user name/user ID. This is primarily a software construct. Kernel mode is part of core hardware support for isolation/protection.
3 pts

As is, XINU is a single user system without a notion of user name/user ID. The XINU version we are using does not have a file system and does not implement isolation/protection. If these two components are added, it may be meaningful to add user name/user ID to XINU that supports the notion of ownership of files.
2 pts

P2(b) 17 pts

EFLAGS
8 "general purpose" registers (EAX, EBX, ECX, EDX, ESP, EBP, two more)
4 pts

ESP is also separately stored in the saved stack pointer field (prstkprt) of the process's process table entry. To restore the hardware using the information saved on the process's stack, we need to know the address of the top of the stack.
3 pts

In XINU all code (i.e., text) is shared by all processes and that includes the code of the kernel. This implies that kernel functions such as resched() and ctxsw() reside at the same address for all processes. Since both the process being context-switched out and the process being context-switched in call the same functions to checkpoint and restore their states, there is no need to save EIP.
7 pts

This is not specific to XINU. In modern kernels such as Linux/UNIX and Windows, kernel code (but not user code) is shared by all processes. Therefore addresses of kernel functions are the same for all processes.
3 pts

P3 21 pts

XINU static priority scheduler:
Dequeue: constant (i.e., O(1)) since XINU's ready list is a priority queue that sorts ready processes by their priority in nonincreasing order. Therefore a highest priority ready process is always at the head of the list.
Enqueue: in the worst case, the ready list has to be traversed until the end to find the place where a ready process has to be inserted. Since the ready list can include all other processes, the cost is linear (i.e., O(n) where n is the number of processes).
6 pts

Solaris TS with multilevel feedback queue:
With 60 priority levels for TS processes (i.e., number of priorities is fixed or constant):
Dequeue: find the highest priority level at which there are one or more ready processes enqueued. At worst, this requires 60 comparisons. After finding the highest non-empty priority level, we pick the first in the list of processes pointed to by the data structure since all processes at the same priority level are serviced round-robin. Hence overhead is constant.
Enqueue: the priority of the ready process to be inserted acts as the index to the data structure (array of struct which contains two pointers) which is constant. We use the second pointer which points to the end of the list to insert the process. This incurs constant overhead. Again, this only works because all processes of equal priority are assumed to be served round-robin.
8 pts

In Linux fair scheduling where CPU usage is utilized in determining the priority of processes, a process's priority is inversely related to its CPU usage. The kernel's scheduling invariant -- always pick a highest priority ready process to run next -- dictates that we choose a ready process which has received least CPU usage. Instead of using a linear cost priority queue, we can use data structures that implement balanced trees to enqueue and dequeue ready processes in logarithmic time (i.e., the depth of a balanced tree is logarithmic in the number of elements stored). Linux uses a particular data structure called R-B tree for this purpose.
4 pts

Practically, it is not such a big weakness since the logarithm of even a super astronomical number such as 2^300 is still 300, a relatively small constant. For servers which may manage thousands of processes, the depth will be in the teens or less. Reducing constants is practically important hence logarithmic cost is a weakness, just not a major weakness.
3 pts

Bonus 10 pts

One of the tasks of the clock interrupt handler is to implement count down of a process's time slice. If the count reaches zero (i.e., a process has depleted its time slice), the clock interrupt handler calls the scheduler so that it may decide which process to run next.
5 pts

Assuming a process is not being preempted by a higher priority process that has become unblocked and ready, we need to ensure that the current process (which has the least CPU usage) does not excessively use CPU cycles uninterrupted where it may overtake the process with the second least CPU usage. Thus we need to apply a time limit so that if the process does not give up the CPU before then, the kernel is able to intervene and decide who to schedule next.
5 pts