P1(a) 12 pts

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

The current process.
Upper half: the current process invoked the system call.
Lower half: kernel borrows the context of the current process.
6 pts

P1(b) 12 pts

1. No isolation/protection.
2. XINU implements static priority scheduling.
3. XINU implements 1-word IPC, no other mechanisms such as
asynchronous IPC with callback function.
4. XINU does not utilize multiple core/CPUs.
There are others and any three -- must include
feature #1 though -- are enough.
6 pts

Isolation/protection is most critical since without it a shared
computing system such as a server may be trivially compromised by
hackers, and desktop/mobile devices would crash frequently due
to app software bugs.
6 pts

P1(c) 12 pts

Interrupt disabling.
3 pts

While interrupts are disabled, time critical and important events
such as clock interrupts, packets arriving on network interfaces,
etc. would be ignored while system call code in the upper half of
a kernel is running.
3 pts

Counting semaphores.
3 pts

Counting semaphores enforce mutual exclusion only during critical
section code (only part of system call code is CS code) and processes
that attempt to access shared data structures that have been already
acquired/locked (in XINU through wait()) are blocked in a waiting
queue to achieve orderly access.
3 pts

P2(a) 16 pts

System calls in the upper half that may block the calling process.
Interrupt routine in the lower half that unblock previously blocked
processes.
5 pts

resched() is an internal kernel function which is invoked by system
calls/interrupt routines. In XINU, it is the latter that disable
interrupts before calling resched() so no need for resched() to do so.
5 pts

ebp, flags, and 8 "general purpose" registers which include eax, ebx,
ecx, edx.
3 pts

The process's stack pointer.
3 pts

P2(b) 16 pts

To maintain isolation/protection and prevent the entire system from crashing or being hacked into, each system call must carefully check the validity of the arguments passed in the system call and their correct use.
6 pts

Changing from user to kernel incurs additional overhead whose magnitude varies depending on specific hardware support and kernel implementation.
4 pts

Storing kernel information in user space run-time stack which is readable/writable by user mode processes presents a vulnerability through which a kernel can be compromised/hacked into.
6 pts

P3(a) 16 pts

Blocking: When the kernel is not in possession of the data item to be received/read, the calling process is blocked by calling the scheduler which context switches it out.
3 pts

Nonblocking: When the kernel is not in possession of the data item to be received/read, the system call returns to the caller with a return value indicating that the requested data item is not available.
3 pts

A process registers a user space callback function, say myfunc, with a kernel specifying when it should be executed: for example, upon receiving a message from another process or network. The process runs code without blocking on this event to occur. When the event occurs, the kernel runs the function myfunc.
5 pts

The event that triggers running myfunc may occur at any future time, such as when another process is occupying the CPU. Two issues arise: one, the kernel must run the user function in user mode to shield the kernel from buggy or malicious user code, two, myfunc should be run in the context of the process that registered it to protect the current process from buggy/malicious code belonging to a different process.
5 pts

P3(b) 16 pts

When a process depletes its assigned time slice which
is detected by the clock interrupt handler, the scheduler is invoked which assumes that the process is CPU-intensive.
When a process makes a blocking system call before depleting its time slice, the scheduler which is invoked by the upper half of the kernel assumes the process is I/O-intensive since it gives up the CPU voluntarily.
4 pts

If deemed CPU-intensive, Solaris decreases the process's priority and increases its time slice.
If deemed I/O-intensive, the opposite holds true.
4 pts

If there are no ready I/O-intensive processes, a CPU-intensive process can occupy a CPU for a prolonged period without the clock interrupt handler calling the scheduler every time the time slice is depleted. This avoids unnecessary overhead.
If a blocked I/O-intensive process becomes ready, then its higher priority enables it to preempt the CPU-intensive process, thereby, over time, allowing "fair" sharing of CPU cycles.

4 pts

No. The above mechanism is conducive to "fair" sharing of CPU cycles
but does not guarantee that a process may not receive CPU cycles for
a prolonged period. To prevent that, additional safety net mechanisms
must be instituted as discussed in class.
4 pts

Bonus

Hardware support for:
- implementing isolation/protection
- run-time stack manipulation
- implementing mutual exclusion
- multithreading
- etc.
Any three suffice for full credit.
10 pts