P1(a) 20 pts

RMS pro: simpler since priority of does not change.
RMS con: lower utilization (i.e., schedulability test < 1).
4 pts

EDF pro: optimum utilization (100%).
EDF con: more complex due to dynamic priority adjustment.
4 pts

Estimation of computation time requirement C for each task.
Computation time requirement C is not constant (worst-case) but time-varying.
Accounting for kernel overhead.
Designing maximally preemptive kernel.
6 pts
// Any subset of three receive full credit.

Both upper and lower halves of XINU kernel are non-preemptible. [Hence processing of time critical events may be delayed due to running kernel code with external interrupts disabled.]
// Elaborations noted in square brackets [...] are for clarity.
6 pts

P1(b) 20 pts

System call API: wrapper function to trap to kernel code (system call dispatcher) by executing trap instruction [and pass arguments].
3 pts

System call dispatcher: kernel code in lower half that serves as entry point to kernel code in the upper half.
3 pts

Kernel functions in the upper half called by system call dispatcher that perform requested service.
3 pts

XINU system calls start by disabling interrupts but before doing so remember the value of x86's IF flag in EFLAGS. XINU system calls before returning restore IF to the value it had before a system call was called. Therefore, even though legacy XINU system calls were used as internal kernel functions called by the system call dispatcher, remembering the value of IF and restoring it prevented interrupts from being enabled when executing system call dispatcher kernel code.
5 pts

No additional complications would arise for blocking system calls such as sleepms(). Expanded process context including user stack pointer ESP, user stack SS, user code EFLAGS, user code CS, user code return address (EIP), and stack frames pushed by internal kernel function calls are preserved with the help of a per-process kernel stack which is saved/restored when a process executing upper half kernel code is context-switched out/in.
6 pts

P2(a) 20 pts

Interrupt disabling is suited for uniprocessor environments [where kernel code executing with interrupts disabled are short in duration].
4 pts

Test-and-set is suited for multiprocessor/core environments [where processes/threads executing on a processor may afford to busy wait].
4 pts

Instead of busy waiting using test-and-set, semaphore based mutual exclusion allows a process to block, i.e., be context-switched out, thus freeing the CPU/core to be used by a ready process. When the semaphore is released, the blocked process may be unblocked and execute its critical section.
6 pts

tset() assigns constant 1 to EAX and executes xchg with operands EAX and the address of the semaphore variable passed as argument. Then tset() returns. [If the semaphore
variable contained 0, tset() returns 0 since EAX = 0 which allows tset() to break out from an infinite while-loop. Since xchg atomically sets the semaphore variable to 1, processes that execute xchg later will be stuck in their while-loop until the semaphore is reset to 0 using the mov instruction.
6 pts

P2(b) 20 pts
Dequeque/extract operation: loop from high to low priority which has constant overhead since the number of priority levels is constant. At the first priority level where a process exists, dequeue the process from front of FIFO queue using head pointer which incurs constant overhead.
3 pts
Enqueue/insert operation: use priority of ready process as index into multilevel feedback queue. Use tail pointer to insert process at the end of the FIFO queue. Both operations incur constant overhead.
3 pts
Linux CFS keeps track of process CPU usage to select a process of least CPU usage to run next when the scheduler is invoked. Since CPU usage has non-constant wide range, performing insert/exact operations using a balanced tree/heap structure incurs logarithmic overhead.
3 pts
First heuristic: a newly created process cannot be assigned zero CPU usage but is assigned an artificial CPU usage so that it does not starve existing processes that have accumulated significant CPU usage.
3 pts
Second heuristic: when an I/O-bound process that blocks for an extended time becomes ready, it must be assigned an artificially increased CPU usage so that it does not starve CPU-bound processes that accumulated significant CPU usage while the I/O-bound process was blocked.
3 pts
A second balancing criterion is priority in order to avoid a group of high priority ready processes being assigned to one core, while a group of low priority ready processes are assigned to another core. From a CPU load perspective the two cores would be balanced. However, across the two cores high priority processes would not receive a larger share of CPU time than low priority processes. (Hence composition of processes assigned to a core w.r.t. their priorities must be balanced, i.e., diverse so that not only at each core but across cores high priority processes receive preference over low priority processes.)
3 pts
Process lifetime of real-world workloads where many processes are short-lived needs to be considered. Given the cost associated with migrating a process from one core to another, migrating short-lived processes should be avoided.
2 pts
P3 20 pts
A process that becomes current for the first time is set up to appear as if it had executed before and was context-switched out. This allows the code of ctxsw() to work for both "veteran" and "newbie" processes. The main difference between the two cases is the return address of ctxsw() stored in the runtime stack where for veteran processes it points to the instruction in resched() following the call to ctxsw(). In the newbie process, the return address points to the first instruction of the function that the process is tasked to execute. In the newbie case ctxsw() executes iret to untrap to user mode whereas in the veteran case ret is executed to return to resched().
4 pts
create() artificially sets up a newly created process's kernel stack so that the above outcome [and illusion] for the newbie case is induced.
2 pts
Since for a newly created process that executes for the first time ctxsw() does not return to resched() but jumps to the first instruction of the function to be executed by the new process, the IF flag in EFLAGS must be set to 1 so that user code runs with interrupts enabled. If EFLAGS is restored first, ctxsw() may be preempted by an interrupt that leaves the context of the newbie process in an inconsistent state: ESP and EBP point to kernel stacks belonging to two different processes. Restoring EBP first prevents this from happening.
x86 with hardware support for stack switching allows kernel code to use the current process's per-process kernel stack to manage kernel function calls (including ctxsw()) and save SS, ESP, EFLAGS, CS, EIP of user mode code and user stack to preserve the state of the process to be context-switched out. The address of the top of the kernel stack must be saved when checkpointing. ESP0 of TSS must be set to the bottom of the kernel stack of the process to be context-switched in. For a newly created process this is performed in ctxsw() before executing iret which atomically jumps to the function to be executed by the new process and changes CS and SS to point to user code and user data entries in GDT. For a process that ran before, ESP0 can be updated by the system call dispatcher before executing iret.

For a newly created process to start executing in user mode, create() must configure the per-process kernel stack of a newly created process so that the top of the stack contains SS, ESP, EFLAGS, CS, EIP where EIP pointing to the address of the user code to be executed is at the very top. CS and SS point to user code and data entries of GDT, ESP points to the top of the user stack, EFLAGS is initialized with IF = 1 to enable interrupts. DS is set to point to the user data entry in GDT. Then iret is executed.

Just updating ESP0 of TSS in the system call dispatcher (or ctxsw()) so that it points to the per-process kernel stack of the context-switched in process will suffice. [The remaining state information SS, ESP, EFLAGS, CS, EIP to switch to user code running in user mode using user stack is contained in the process's kernel stack which will be popped by iret which empties it.]

A third stack to be used for interrupt handling in XINU's lower half is not necessary.

Since both upper half and lower half of XINU execute with interrupts disabled, during the time a system call is being serviced by the upper half it cannot be preempted by the lower half in response to an interrupt. As noted in Problem 3, when system call processing in the upper half completes the current process's per-process kernel stack pointed to by ESP0 is back to empty when the process untraps to user mode.

An interrupt may preempt the current process when running in user mode. Transition to kernel mode switches the runtime stack from user stack to the empty per-process kernel stack pointed to by ESP0. Since the lower half of XINU also runs with interrupts disabled, until its work is complete (and untraps to user mode) it cannot be preempted. When the lower half executes iret and switches back to user mode and user stack, the per-process kernel stack that was borrowed to execute lower half code is again empty. Thus complications that arise in kernels where upper and lower halves are preemptible do not occur in XINU.