Name:____________________________________

ID number:_________________________________

Instructions: answer all 6 questions (answering in Hebrew is fine)

Q1: ______
Q2: ______
Q3: ______
Q4: ______
Q5: ______
Q6: ______
Grade: ________________________________
1. [16 points] The designers of Determinator identify four aspects one must tackle in order to have an operating system that allows for pervasive determinism. Enumerate these four aspects and briefly explain them:
2. [16 pts] In the lecture about machine virtualization, you’ve learned about three ways to accomplish I/O virtualization. Enumerate these ways and briefly explain their pros and cons: 

______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
______________________________________________________________________________________
3. [18 pts] Define a “wild backfill” operation and give a brief example.
4. [17 pts] Context switching in JOS:
   a. [8 pts] JOS contains the following code:

   ```c
   void env_pop_tf(struct Trapframe *tf) {
       __asm __volatile("movl %0,%%esp\n"
                       "tpopl\n"
                       "tpopl %%es\n"
                       "tpopl %%ds\n"
                       "taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
                       "iret"
                       : : "g" (tf) : "memory");
       panic("iret failed"); /* mostly to placate the compiler */
   }
   ```

   A student decided to "optimize" it by converting the `iret` instruction into a regular `ret`. Is this optimization correct? If yes, explain why. If not, explain why not.
b. [9 pts] The same student also decided to optimize `env_run()` by commenting out the line `lcr3(curenv->env_cr3)`. Explain the resulting bug. Explain in what cases the new code would still do the right thing.

```c
void env_run(struct Env *e) {
    curenv = e;
    curenv->env_runs++;
    /* lcr3(curenv->env_cr3); */
    env_pop_tf(&curenv->env_tf);
}
```
5. [17 pts] xv6 contains the following loop in the scheduler:

```c
void scheduler(void) {
    struct proc *p;
    for(;;){
        // Enable interrupts on this processor.
        sti();
        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;
            // Switch to chosen process. It is the process's job to release ptable.lock
            // and then reacquire it before jumping back to us.
            proc = p;
            switchuvm(p);
            p->state = RUNNING;
            swtch(&cpu->scheduler, proc->context);
            switchkvm();
            // Process is done running for now.
            // It should have changed its p->state before coming back.
            proc = 0;
        }
        release(&ptable.lock);
    }
}
```

a. [2 pts] What scheduling policy does this scheduler implement?

_____________________________________________________________________
_____________________________________________________________________
b. [9 pts] As you might recall, **switch()** stops running the old context (&cpu->scheduler here) and starts running the new context (proc->context here). Since the old context is the scheduler itself, the loop over the processes is stopped in the middle once the scheduler finds a new process to run and starts running it. Will this cause some processes to never run? If yes, explain why the xv6 designers made this design decision. If not, explain why all processes will eventually run.

c. [6 pts] Why does the scheduler call **sti()** unconditionally to enable interrupts? Hint: Assume that there are no currently runnable processes and at least one process is not runnable because it is waiting for I/O (e.g., for the user to press the keyboard or for a network packet). What would happen in this case without the **sti()**?
6. [16 pts] SMP and page faults:
   a. [6 pts] Why does an SMP kernel need multiple kernel stacks?
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________

   b. [6 pts] Give one reason why page fault handling should happen in user space (as we
implemented in lab #4) and one reason why it should happen in kernel space (as done in Linux, for example)
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________

   c. [4 pts] A CPU designer at "We Make CPUs Inc." decided to optimize her CPU by **not**
having the CPU store the "fault address" in the CR2 register during page faults. Suggest how a kernel
programmer could work around this "optimization" to find the faulting address.
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________