Q1: (10 points) Does the xv6 kernel use cooperative approach or non-cooperative approach to gain control while a user process is running? Explain how xv6's approach works using xv6's code.

-> In the xv6 kernel, the approach used to gain control while a user process is running is a non-cooperative approach. It uses a trap mechanism to switch from user mode to kernel mode when needed. When a trap happens, the processor transfers control to a specific interrupt handler in the kernel, which performs the required operations. Once the handler completes its task, control is returned to the interrupted user process, allowing it to continue execution as if nothing happened.

Code from trap() in trap.c:

```
void

trap(struct trapframe *tf)

{

  if(tf->trapno == T_SYSCALL){ ------- checks if the trap is associated with a system call and handles the system call

    ...

  }


  switch(tf->trapno){ ----------- handles the interrupts

  case T_IRQ0 + IRQ_TIMER:

            ...

  case T_IRQ0 + IRQ_IDE:

            ...

  case T_IRQ0 + IRQ_IDE+1:

            ...

  case T_IRQ0 + IRQ_KBD:

            ...

  case T_IRQ0 + IRQ_COM1:

            ...

  case T_IRQ0 + 7:
```

```
  case T_IRQ0 + IRQ_SPURIOUS:
        ...
}
```

Q2: (10 points) After fork() is called, why does the parent process run before the child process in most of the cases? In what scenario will the child process run before the parent process after fork()?

-> After a fork() system call is called in xv6, the kernel creates a new process called the child process, which is an exact copy of the parent process. When fork() is called, The parent process calls fork(), and the kernel creates a new process (the child process) by duplicating the parent's address space, file descriptors, and other resource.After the child process is created, both the parent and child processes are marked as runnable and added to the scheduler's run queue.The scheduler then selects a process from the run queue to run on the CPU. In many operating systems, including xv6, the scheduler often gives priority to the parent process over the child process when both are in the runnable state. As a result, the parent process is more likely to be scheduled to run first after a fork(). One common scenario is when the parent process explicitly waits for the child process to terminate using the wait() system call. In this scenario, the parent process may yield the CPU to allow the child process to start execution, and then the parent process waits for the child to complete before continuing its own execution.

Q3: (10 points) When the scheduler de-schedules an old process and schedules a new process, it saves the context (i.e., the CPU registers) of the old process and load the context of the new process. Show the code which performs these context saving/loading operations. Show how this piece of code is reached when saving the old process's and loading the new process's context.

Code from void scheduler(void) in proc.c:

```
...
// Switch to chosen process.  It is the process's job
// to release ptable.lock and then reacquire it
// before jumping back to us.
c->proc = p;
switchuvm(p);
p->state = RUNNING;
```

```
swtch(&(c->scheduler), p->context);
switchkvm();


// Process is done running for now.
// It should have changed its p->state before coming back.
c->proc = 0;
…
```

The scheduler function loops through the process table looking for a process to run, depending on whether its state is RUNNABLE or RUNNING. The selected process will run in the scheduler until they finish executing or their time is up. The scheduler uses switchuvm(p) to switch to the memory contents of process p. It sets the state of p to RUNNING and gets executed by the CPU. Then, the swtch() function performs a context switch which saves the current/old process's context (like CPU registers) into the scheduler field of c. It then loads the context of the new process.