p1q12
SMH

Operating Systems 2001

Answers to Paper 1 Question 12

I/O Models and Device Drivers

The marking scheme here is 5 marks each for the polling case and an interrupt-driven case, 4 marks for the other interrupt-driven case (since it will be substantially the same).

Polling is used in the absence of interrupts, and requires that the host CPU explicitly check ("poll") the status of the device (e.g. by reading its 'status' register). If the device is busy, the CPU simply has to check again, either immediately or at some later time.

Hence a routine to read a character into `(char *)buf` might look like:

```
do {
  status = read_status_register();
} while (status == DEVICE_BUSY);
write_cmd_register(DO_READ);
do {
  status = read_status_register();
} while (status == DEVICE_BUSY);
*buf = read_data_register();
```

In the interrupt-driven programmed I/O case, the "read" routine may *begin* the read operation, but will not wait for it to complete. In fact if the device is busy, it will not even begin the read operation but will rather queue it. It might look like:

```
status = read_status_register();
if(status == DEVICE_BUSY) {
   enqueue(buf, request_queueu);
   return;
}
current = buf;
write_cmd_register(DO_READ);
```

The main control flow logic is in the interrupt handler. This is where read completions are handled, and where enqueued operations are begun. The 'read-done' portion might look something like:

```
*current = read_data_register();        // not reqd if DMA
if(!is_empty(request_queue)) {
   current = dequeue(request_queue);
   write_cmd_register(DO_READ);
}
clear_interrupt();
```

The DMA case is almost identical to the previous one save that the data does not need to be explicitly read when the interrupt occurs; it will already be present in memory. While this doesn't really give us a big win here (with a single byte), it's well worth it when transferring e.g. 4K of data from a disk.

Note: this answer is more precise than perhaps I would expect from the students. In particular, I'd imagine many of them will go for the fuzzier flow char option, and not be particularly concerned with buffer management. With that in mind, do the examiners believe this is a fair question for 14 marks?

I/O Modes and APIs

Marking here is 2 marks per variant described.

In blocking I/O, the invocation does not return until 'complete' — i.e. until the data has been written or read. Hence in our keyboard example, the blocking call would return promptly if data were available in the keyboard buffer, but otherwise would block the process until a key was pressed.

In non-blocking I/O, the invocation always returns immediately, but may or may not be complete. In our keyboard example, the non-blocking read would return immediately with e.g. a return code stating many characters (≥ 0) have been read from the keyboard buffer.

In asynchronous I/O, the invocation always comprises two halves: a request followed sometime later by a response. In our keyboard case, the initial invocation would enqueue a read request along with an I/O completion handler. Then whenever keyboard data became available (perhaps with no delay), the completion handler would be invoked and could then process the data.