Section 8: Introduction to I/O, Queuing Theory, and RPC

CS162

July 17, 2019

Contents

1 Warm Up

1.1 Short Questions

1. ([✓ True]/False) If a particular IO device implements a blocking interface, then you will need multiple threads to have concurrent operations which use that device.

   block on a request ⇒ cant do any more.

2. (True/[✓ False]) For IO devices which receive new data very frequently, it is more efficient to interrupt the CPU than to have the CPU poll the device.

   Interrupts have high overhead -better for infrequent things.

3. (True/[✓ False]) With SSDs, writing data is straightforward and fast, whereas reading data is complex and slow.

   writes complicated, fast reads

4. (True/[✓ False]) User applications have to deal with the notion of file blocks, whereas operating systems deal with the finer grained notion of disk sectors.

   users don't know about blocks either.

5. What is a block device? What is a character device? Why might one interface be more appropriate than the other?

   lots of data at once ← hard drives
   individual bytes at a time ← keyboard, printer
   ← streaming

6. Why might you choose to use DMA instead of memory mapped I/O? Give a specific example where one is more appropriate than the other?

   DMA - good for large transfers (no CPU intervention)
   MMIO good for direct access to devices w/ CPU

7. Explain what is meant by "top half" and "bottom half" in the context of device drivers.

top half - servicing routines
bottom half - services interrupts

2 Vocabulary

- **I/O** In the context of operating systems, input/output (I/O) consists of the processes by which the operating system receives and transmits data to connected devices.
- **Controller** The operating system performs the actual I/O operations by communicating with a device controller, which contains addressable memory and registers for communicating the the CPU, and an interface for communicating with the underlying hardware. Communication may be done via programmed I/O, transferring data through registers, or Direct Memory Access, which allows the controller to write directly to memory.
- **Interrupt** One method of notifying the operating system of a pending I/O operation is to send a interrupt, causing an interrupt handler for that event to be run. This requires a lot of overhead, but is suitable for handling sporadic, infrequent events.
- **Polling** Another method of notifying the operating system of a pending I/O operating is simply to have the operating system check regularly if there are any input events. This requires less overhead, and is suitable for regular events, such as mouse input.
- **Response Time** Response time measures the time between a requested I/O operating and its completion, and is an important metric for determining the performance of an I/O device.
- **Throughput** Another important metric is throughput, which measures the rate at which operations are performed over time.
- **Asynchronous I/O** For synchronous I/O operations, we can have the requesting process sleep until the operation is complete. On the other hand, asynchronous I/O operations have the requesting process return immediately and continue execution and later notify the process when the operation is complete. , a good way for processes to share data
- **Memory-Mapped File** A memory-mapped file is a segment of virtual memory which has been assigned a direct byte-for-byte correlation with some portion of a file or file-like resource. This resource is typically a file that is physically present on-disk, but can also be a device, shared memory object, or other resource that the operating system can reference through a file descriptor. Once present, this correlation between the file and the memory space permits applications to treat the mapped portion as if it were primary memory.
- **Memory-Mapped I/O** Memory-mapped I/O (not to be confused with memory-mapped file I/O) uses the same address bus to address both memory and I/O devices the memory and registers of the I/O devices are mapped to (associated with) address values. So when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices.
- **Queuing Theory** Here are some useful symbols: (both the symbols used in lecture and in the book are listed)
  - μ is the average service rate (jobs per second)

- T_{ser} or S is the average service time, so $T_{ser} = \frac{1}{\mu}$
- λ is the average arrival rate (jobs per second)
- U or u or ρ is the utilization (fraction from 0 to 1), so $U = \frac{\lambda}{\mu} = \lambda S$
- T_q or W is the average queuing time (aka waiting time) which is how much time a task needs to wait before getting serviced (it does not include the time needed to actually perform the task)
- T_{sys} or R is the response time, and it's equal to $T_q + T_{ser}$ or $W + S$
- L_q or Q is the average length of the queue, and it's equal to λT_q (this is Little's law)

$L_q = \lambda T_q$ ☆

- **RPC** - Remote procedure calls (RPCs) are simply cross-machine procedure calls. These are usually implemented through the use of stubs on the client that abstract away the details of the call. From the client, calling an RPC is no different from calling any other procedure. The stub handles the details behind marshalling the arguments to send over the network, and interpreting the response of the server.
- **IDL** - Interface definition languages (IDLs) are specification languages used to describe a software component's API. They describe an interface in a language-independent way, enabling communication between software components that do not share one language. IDLs are commonly used in RPC where the two endpoints of an RPC communication may be using different operating systems or languages. Protocols specified in an IDL can be compiled into varoius source languages, genearating client and server stubs that handle marshalling/unmarshalling of arguments and code.

3 Problems

3.1 Disabling Interrupts

We looked at disabling CPU interrupts as a simple way to create a critical section in the kernel. Name a drawback of this approach when it comes to I/O devices.

No interrupts from devices/timers now.

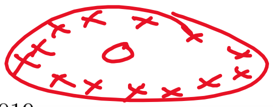
3.2 Disks

What are the major components of disk latency? Explain each one.

SSD

Queuing Time - OS queue
Controller - send message + process request
Seek - disk head moving
rotation - disk rotating into position
transfer - actual transfer of data

In class we said that the operating system deals with bad or corrupted sectors. Some disk controllers magically hide failing sectors and re-map to back-up locations on disk when a sector fails.
If you had to choose where to lay out these back-up sectors on disk - where would you put them? Why?



e.g. outer track

Backups in one spot or spread out

How do you think that the disk controller can check whether a sector has gone bad?

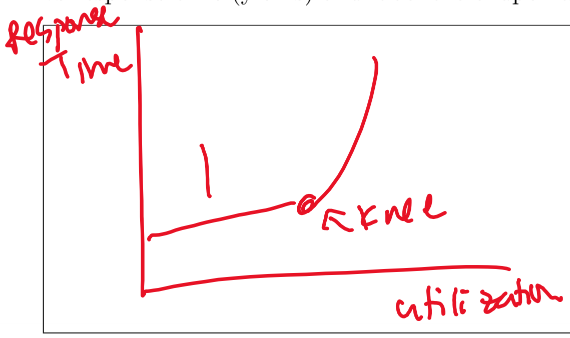
checksum - like 70

Can you think of any drawbacks of hiding errors like this from the operating system?

Can't tell if disk is going to fail ⇒ user can't prepare

3.3 Queuing Theory

Explain intuitively why response time is nonlinear with utilization. Draw a plot of utilization (x axis) vs response time (y axis) and label the endpoints on the x axis.



Server isn't doing work 100% of the time, so at 1 utilization, approaches ∞ response time

If 50 jobs arrive at a system every second and the average response time for any particular job is 100ms, how many jobs are in the system (either queued or being serviced) on average at a particular moment? Which law describes this relationship?

$L_q = \lambda \cdot T_q = 50 \cdot 0.1 = \boxed{5 \text{ jobs}}$

Is it better to have N queues, each of which is serviced at the rate of 1 job per second, or 1 queue that is serviced at the rate of N jobs per second? Give reasons to justify your answer.

One server N jobs/s.
response time 1/N s and no need for load balancing which also takes time.

What is the average queueing time for a work queue with 1 server, average arrival rate of λ, average service time S, and squared coefficient of variation of service time **C**?

avg service rate , $u = \lambda S$

$$T_q = T_{ser}\left(\frac{u}{1-u}\right)\left(\frac{C+1}{2}\right) \leftarrow \text{general formula}$$

What does it mean if **C** = 0? What does it mean if **C** = 1?

$C=0 \Rightarrow$ constant rate
$C=1 \Rightarrow$ Poisson dist of arrivals (memoryless)

3.4 Tying it all together

Assume that you have a disk with the following parameters:

- 1TB in size
- 6000RPM
- Data transfer rate of 4MB/s (4×10^6 bytes/sec)
- Average seek time of 3ms
- I/O controller with 1ms of controller delay
- Block size of 4000 bytes

What is the average rotational delay?

$$\frac{1}{2} \cdot \frac{60\,s/min}{6000\,rpm} = 5ms$$

What is the average time it takes to read 1 random block? Assume no queuing delay.

transfer

$$\frac{4000B}{4000000B/s} = 1ms$$

$$\underbrace{1}_{\text{controller}} + \underbrace{3}_{\text{seek}} + \underbrace{5}_{\text{rotation}} + \underbrace{1}_{\text{transfer}} = \boxed{10ms}$$

Will the actual measured average time to read a block from disk (excluding queuing delay) tend to be lower, equal, or higher than this? Why?

lower. disk op

Assume that the average I/O operations per second demanded is 50 IOPS. Assume a squared coefficient of variation of **C** = 1.5. What is the average queuing time and the average queue length?

3.5 RPC

As mentioned in lecture, RPC provides inter-process communication and location transparency, meaning that processes can be local to the same machine or be between two different machines. Thus, the RPC code used by the client remains the same no matter how the client and server are separated. In this problem, we'll explore the usage of an actual RPC system called gRPC complemented by an IDL called Protocol Buffers. We will implement a basic key-value storage in Python with RPC.

We first start by filing out the necessary code in our `keyvaluestore.proto` file. The two most important concepts are messages and services. Think of messages as a data structure that a client wants to send to a server for it to call a function on. Services are essentially functions in the RPC system that operate on the defined message types. Certain service and message types have already been filled out for you. Your task is to fill out the blanked out service and message types (based on the syntax of the lines that are already filled out):

`keyvaluestore.proto`:

```
syntax = "proto3";

// A simple key-value storage service
service KeyValueStore {
  // Provides a value for a key request
  rpc GetValue (GetRequest) returns (Response) {}

  // Stores a value for a key-value request. Assume service is called StoreValue
  ___________________________________________________________________________;
}

// The request message containing the key
message GetRequest {
  ___________________________________________________________________________;
}

// The request message containing the key and value
message StoreRequest {
  string key = 1;
  string value = 2;
}

// The response message containing the value associated with the key
```

```
message Response {
 ___________________________________________________________________________;
}
```

We can now compile the `.proto` file we wrote by calling `python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. ./keyvaluestore.proto`. This will generate two files: `keyvaluestore_pb2.py` and `keyvaluestore_pb2_grpc` and contain:

- classes for the messages defined in `keyvaluestore.proto`
- classes for the service defined in `keyvaluestore.proto`
  - `KeyValueStoreStub`, which can be used by clients to invoke KeyValueStore RPCs
  - `KeyValueStoreServicer`, which defines the interface for implementations of the KeyValueStore service
- a function for the service defined in `keyvaluestore.proto`
  - `add_KeyValueStoreServicer_to_server`, which adds a RouteGuideServicer to a grpc.Server

Now that we have the necessary compiled stub code, let's create a KeyValueStore server by first implementing the servicer interface generated from our service definitions. This servicer class must implement all the KeyValueStore service methods defined in the `keyvaluestore.proto` file. Please fill out all the following code in `keyvaluestore_server.py`:

```
class KeyValueStoreServicer(keyvaluestore_pb2_grpc.KeyValueStoreServicer):
    """Provides methods that implement functionality of key value store server. """
    def __init__(self):
        ___________________________________________________________________________

    def GetValue(self, request, context):
        if ________________________________________________________________________:
            _______________________________________________________________________
        else:
            return keyvaluestore_pb2.Response(value="Error: %s does not exist." % request.key)

    def StoreValue(self, request, context):
        ___________________________________________________________________________
        ___________________________________________________________________________
```

We next define a `serve()` function (which will be run by the server) that does the following:

- Create a gRPC server.
- Create a KeyValueStoreServicer() instance and add it to the gRPC server
- Set up the gRPC server to listen and accept requests at localhost:50051.
- Start up the server

For those who are curious, it looks something like:

```
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    keyvaluestore_pb2_grpc.add_KeyValueStoreServicer_to_server(KeyValueStoreServicer(), server)
    server.add_insecure_port('[::]:50051')
    server.start()
```

Next, let's create the client by implementing the basic functions of a key-value store: storing a key-value pair and retrieving the value of a key. Client functions operate on its typical arguments and a stub, which is bound to a specific hostname and port. Fill out the following code in `keyvaluestore_client.py`:

```
def store_value(stub, key, val):
    ---------------------------------------------------------------------------
    ---------------------------------------------------------------------------

def get_value(stub, key):
    response = stub.GetValue(keyvaluestore_pb2.GetRequest(key=key))
    print("Stored %s" % (response.value))
```

We can then define a `run()` function that stores key-value pairs into the server and retrieves values from the server for a particular key.

```
def run():
    with grpc.insecure_channel('localhost:50051') as channel:
        stub = keyvaluestore_pb2_grpc.KeyValueStoreStub(channel)
        print("----------------StoreValue----------------")
        store_value(stub, "project2 design doc", "due tonight")
        print("----------------GetValue----------------")
        get_value(stub, "project2 design doc")
        print("----------------Test non-existent key----------------")
        get_value(stub, "my grade")
```

As mentioned earlier, notice that the client connects to a certain server to generate its stub. In this example, the client and server are running on the same machine. However, the server can run on a machine on the other side of the world and the same code that's shown here will still function properly as long as we are able to determine the server's hostname and port. This is the beauty of RPC's *location transparency.*