Lecture #24

****
Review -- 1 min
****
Disk performance
Response time

****
Outline - 1 min
****
A Little Q Theory
Connection I/O to processors

****
Preview - 1 min
****
I/O Architectures

- disks
- hierarchies

Networks

****
Lecture - 20 min
****
Introduction to Queuing Theory

Disclaimer – just a few very simple ideas
these simple ideas have a lot of power,
but you could take a whole class on queuing theory

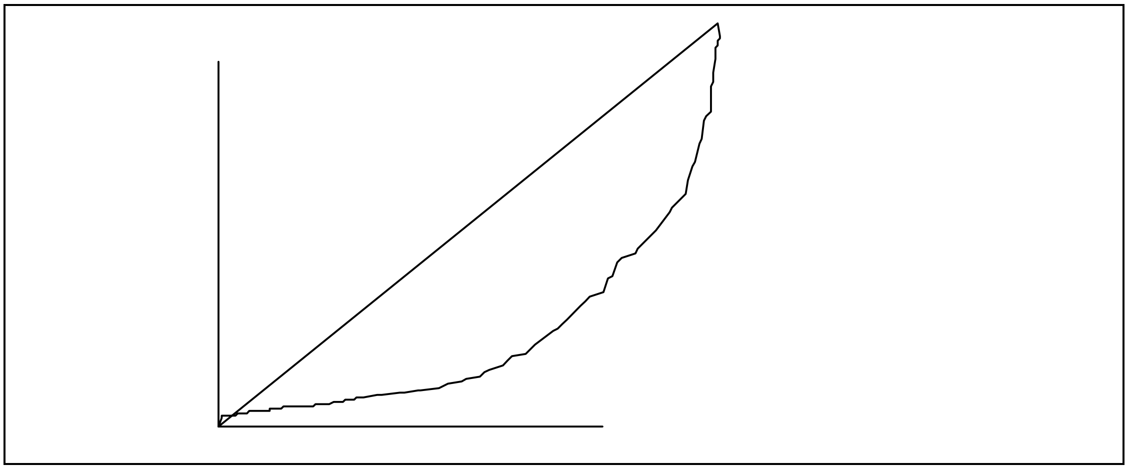
2 results

1. Little's law
   Lsys = rT

#jobs in a system = arrival rate * residence time

what's powerful here: applies to any closed system in equilibrium
closed == black box does not create/destroy jobs
equilibrium == arrival rate = departure rate

2. Response time v. throughput graph



Tqueue = Tservice * u/(1-u)   /* special case */

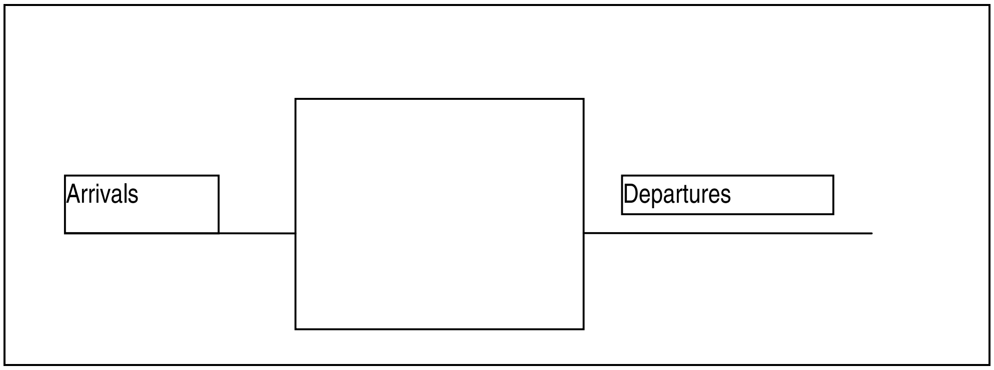
What's powerful here: trade-off between throughput and response time is fundamental. If I drive throughput to 100%, response time goes to infinity.

Also note: variability assumptions are crucial. If arrivals were evenly spaced, could run at 100%.

Little's Law

Well known for a long time
Little proved it, so it gets his name

$L_{sys} = r\,T$



What is interesing – very weak assumptions for this to hold

- system in equilibrium – arrival rate = departure rate
- nothing in black box creates/destroys jobs

e.g.

we have 150 grad students in program
grad student stays in program for 3 years on average
how many students must we accept per year?

The colorado river brings 8M gallons of water into lake travis per hr.
The lake contains 8000M gallons
how long does the average molecule stay in the lake?

I send 50 requests per second to a disk
each request takes 0.1 second to come back
how many requests queued at disk?

L = 50 req/sec * 0.1 sec = 5 req (4 in queue, 1 in disk)

Latency/Throughput Tradeoff

The above disk calculation took response time as a given
How do you predict response time for a system you want to build?
(little's law requires you to know either response time OR
how many items in queue

- these are the things you want to find out

Note that disk access time alone doesn't help –
even if you know that avg disk access time is

1ms (controller) + 3 ms (typical seek) + 4 ms (rot) + 0.5 (xfer)
Tdisk= 8.5 ms

If I'm sending 50 requests/second on average then
Ldisk = .0085 * 50 = .425

QUESTION: How do you interpret that?
A: disk utilization
But that doesn't tell use response time to user.

Central question – burstinesss of requests
Consider two extremes

1. requests arrive every 20ms (= 50 req/sec)
   → no queuing → Tsystem = 8.5 ms

2. 1000 requests arrive at once, then no requests arrive for several seconds
   → first requests takes 8.5 ms, second takes 17, third takes 25.5…
   Tsystem = ?? some big number??

The Queuing part of Queuing theory (finally)

Assume FIFO queue

Tsys = Length_queue * Time_server + residual service time

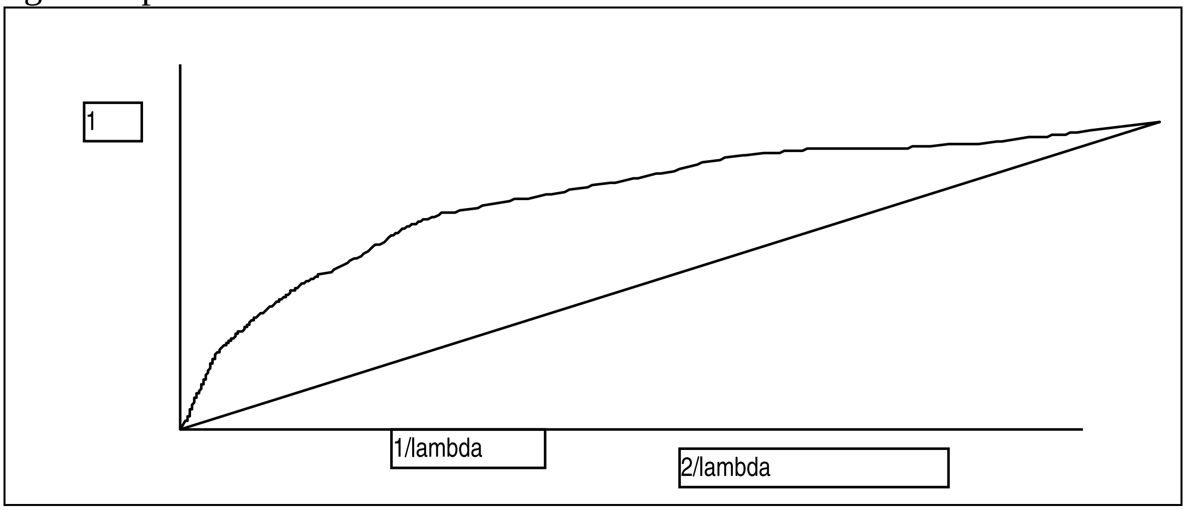
Length_queue * Time_server → how long do I wait for those jobs

last term a bit tricky

RST = mean time to complete tasks being serviced when new task arrives

RST

Assume Memoryless, random arrivals:
e.g. an exponential distribution



P(T<t) = 1 - e^(lambda*t)

What does this mean?

Exponential distribution says that past behavior does not effect future expectations

e.g. if each instruction in stream is equally likely to cause an I/O

e.g. a bunch of different processes are all independently causing I/O to occur

For these distributions squared Coefficient of variation

C = variance / weighted mean time ^2

For exponential distribution → C == 1.0

Hypoexponential → C < 1 – more clustered around average

Hyperexponential → C > 1 – less clustered around average

In such a system

RST = ½ Weighted mean time * (1+C)

Intuition: no variation → C = 0 → makes sense

huge variation → C = infinity → <the burst example above>

For exponential distribution:

Tqueue = Length_queue * Time_server + residual service time
= Length_queue * T_server
+ server utilization * (1/2 * Tserv * (1+C))

= (Arrival Rate * Tqueue) * T_serv
+ Util_serv * Tserv
= Tqueue * Util_serv + Tserv * Util_serv

Solving for Tqueue
→
Tqueue = Tserv * Util_serv/(1-Util_serv) (for C == 1)

Draw Util v. response time curve

Caveats

Queuing system assumptions
1. System is in equilibrium
2. Exponential interrarrival times
3. infinite queue
4. FIFO queue
5. All tasks in line must be completed

These assumptions should make you uncomfortable!

In particular – exponential interrarival times
Used for 2 reasons
1. some workloads kinda look like this (random arrivals)
2. makes math easy

#2 is often the main reason, I fear
What's bad is many workloads don't look at all like this

e.g. network traffic – self-similar
e.g. disk traffic – 30 second write buffer
…

****
Admin - 3 min
****
HW 4 – next Friday (available on line)
Project checkpoint: Friday April 9 (2 weeks)
time to get serious about projects!
key idea – milestones
list the tasks that must be completed to satisfactorily complete your study
prioritize by dependencies
prioritize by importance
indicate which are done, which are pending

example: cache consistency study
1. download seltzer's simulator and traces
2. compile simulator
3. reproduce figure 7 from their paper
4. modify simulator to implement out policy
   sub-task1, subtask2, …
5. simulate our policy
6. reproduce figure 7 with our policy added
7. evaluate and fix deficiencies of study
8. implement prototype

priority: vital: 1-6; optional: 7-8

what I am looking for – have you divided your project
into a set of bite-sized chunks, each of which is straightforward\\
to handle?

****
Lecture - 24 min
****
Processor Interface Issues

interconnections – busses
processor interface

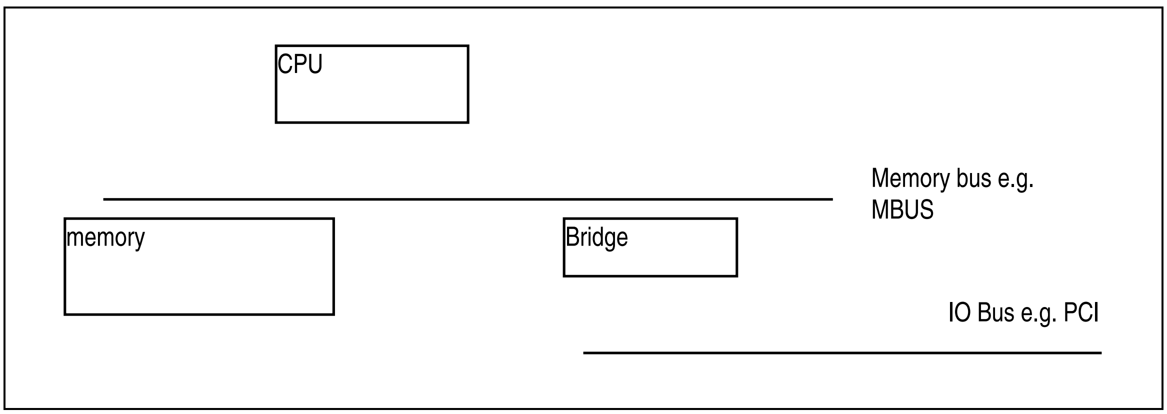
- interrupts
- memory mapped io

I/O Control

- polling
- interrupts
- DMA
- I/O Controllers
- I/O Processors

Interconnections – busses

Need to connection I/O to processor/memory
Common design – 2 busses
CPU-memory
memory bus - I/O bus

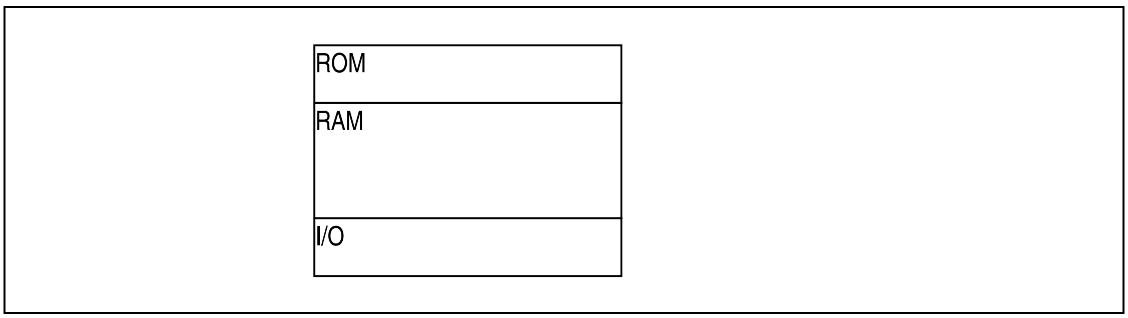


2 busses because motivated by 2 factors

CPU Bus	IO Bus
performance	cost
proprietary (improves speed)	standards-based
few devices (improves speed)	hold many devices
cheap would be nice	performance would be nice

pitfall – this approach moves I/O farther away from CPU example – Myricom advertised Sparcstation 2 performance rather than sparcstation 10 or sparc 20. Because SS2 only had 1 bus (SBUS used for both memory and I/O), better bandwidth from network to memory. (from memory – Sparc 2: 160 Mbit/s max, Sparc20: 136 Mbit/s max)

note interface for this approach: Memory Mapped I/O
→ CPU reads/writes "magic" addresses that correspond to I/O devices



I/O Addresses further devided – specify which I/O device, which register within device, etc…

Contrast with old approach – special I/O instructions at CPU to read and write I/O devices

Processor Interfaces

Polling v. interrupts v. DMA v. I/O processors

most basic – polling

```
loop{
    if(data available)
        read data
}
```

problem: busy wait loop not an efficient way to use CPU
(unless IO device very fast)

solutions: 1) intersperse code with I/O checks
example: wordstar printing

active messages network code

2) interrupts

2. interrupts

I/O device signals interrupt to CPU

how this is done – dedicated lines on bus
e.g. SBUS – 8 extra lines, 1 per SBUS device

→ CPU saves PC and jumps to interrupt service address

problem: interrupt overhead, CPU touches all data

OS + actual time to service interrupt
bad for large transfers of data

3. DMA

device transfers data directly to/from memory
interrupt CPU when done

limitations: works for sequential transfer

(comlplex interaction with CPU; below)

4. I/O processor

example – linked list of I/O's to perform and
places to put data…

Relationship to processor architecture

- Interrupts:

  stack replaced by shadow registers
  handler saves registers and re-enables higher-priority intrrupts
  interrupt types reduced in number – handler must query interrupt controller

- CPU caches cause problems for I/O
  - PIO transfers pollute cache
  - DMA ends with data not in cache → slow to use data

- ♦ reads/writes to I/O devices cause side effects → no caching allowed
- ♦ DMA/IO Processor is second processor → cache consistency problem

    (solutions – flush (expensive), snoop (complex)

- virtual memory makes dma comlex
- stateful processors hard to context switch
    - ♦ registers → shadow registers
    - ♦ cache → cache pollution (slowdown)

- IO Processors – hard to keep up with real CPU

    Intel spends $B's per year making processor faster
    off-loading from CPU can make sense in short term, but
    cost a lot (performance / development) in long term

****
Summary - 1 min
****

Little's law – L = rT
Response time v. throughput