CS 350

Systems & Abstractions

9/7/22

Design Concepts:

Systems as Data Flow & Control Flow:

Computing = Data path + control path

Networking = topology + routing + accounting

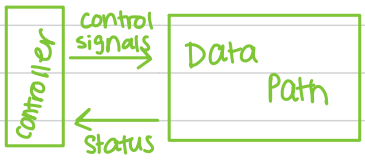
Layering to hide system complexity

Programming = microarchitecture > ISA > OS Syscalls > VMs

Networking = Physical > IP > TCP > Applications

System:

- interconnected operation blocks
- has input and output



Controller:

- cycle through a sequence of states. In each state, send controls to data path

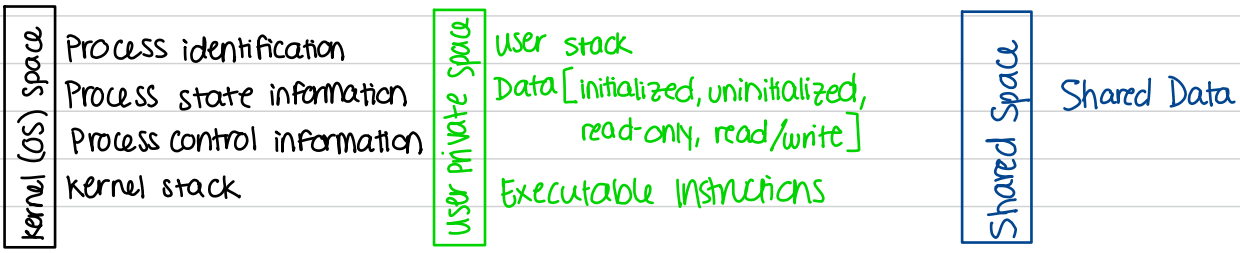
Classes of Interaction

9/12/22

Synchronous: exceptions, memory faults, system calls

Asynchronous: hardware faults, I/O event, timer fired

Processes: program in execution on a system

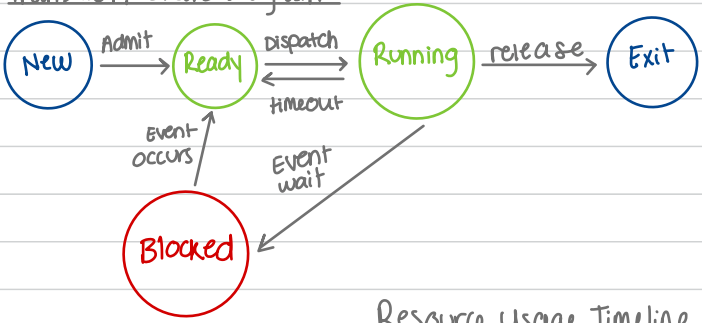


Macrophases:

1. consume some resource (CPU)
2. Request/wait for an external service/event
3. Do nothing until external service done
4. And back up

} still alive

Transition State Diagram



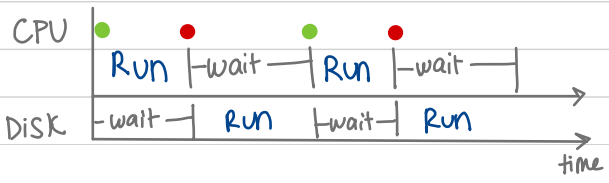
Ready = ready to consume the resource

Running = currently consuming the resource

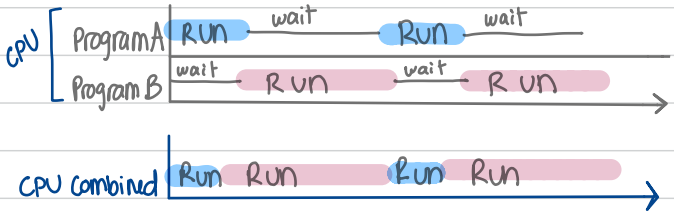
Blocked = Not competing for resource because waiting for some event

Resource Usage Timeline

while blocked on I/O, CPU idle



Multiprogramming: multiple alive processes, and different resource, they progress in parallel



utilization: fraction of time over a given time window when resource is busy (not idle)

throughput: number of completed requests over a given time window

} increasing MPL increases these

capacity: maximum throughput

Response time: finish time - start time

Bottleneck: when one resource reaches 100% utilization = capacity reached

Discussion 1

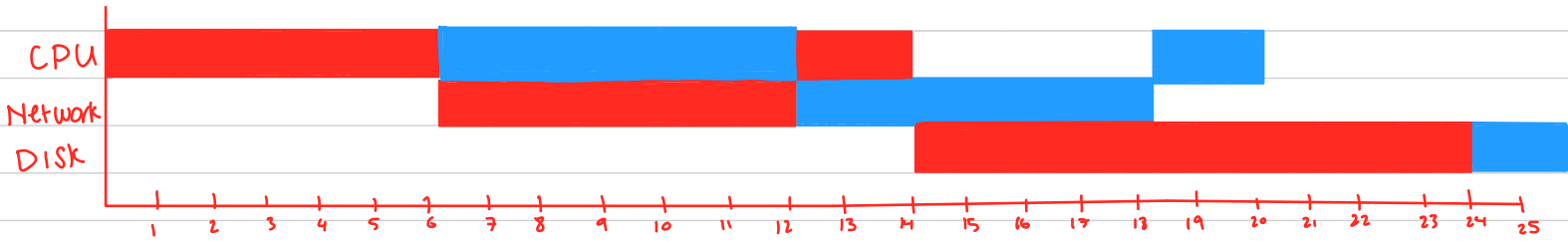
9/14/22

CPU has registers → cache → memory → disk
gets slower →

Multiprocessing = more than 1 CPU with multiple processes

example:

- process uses CPU for 6 ms
- issues network I/O request
- Block for 6 ms until network I/O request is done
- Use CPU for 2ms
- Issue disk I/O request
  Block for 10ms until disk I/O is done
- Exit



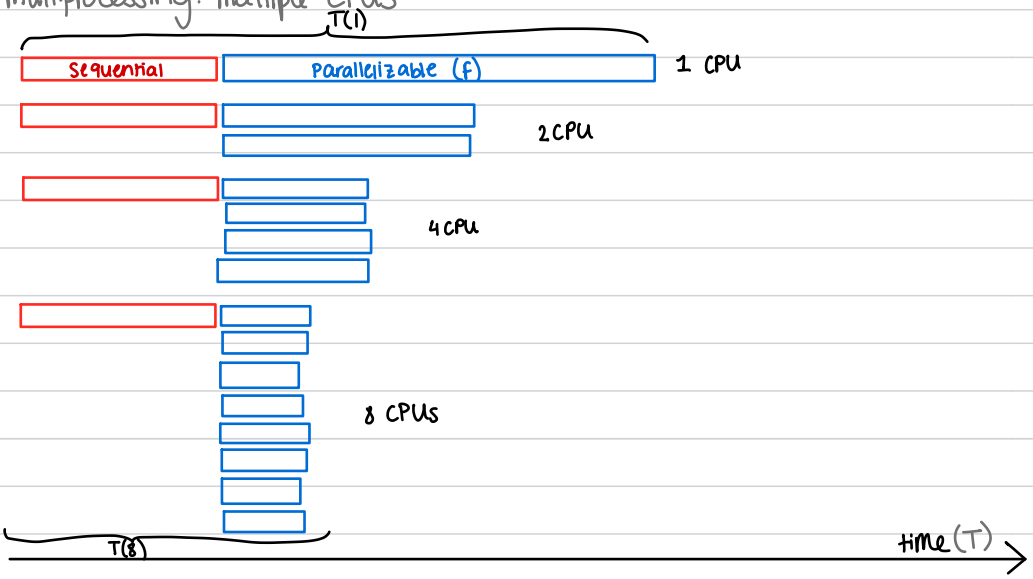
steady-state utilization of CPU = $\frac{8}{24}$
of Network = $\frac{6}{24}$
of Disk = $\frac{10}{24}$
throughput = $\frac{1}{24}$

2 processes: MPL = 2 (keep throwing processes until see pattern)
steady state utilization of CPU = $\frac{16}{28}$
of network = $\frac{12}{28}$
of Disk = $\frac{20}{28}$
throughput = $\frac{2}{28}$

bottleneck: Disk
capacity: $\frac{2}{20}$

9/19/22

multiprocessing: multiple CPUs

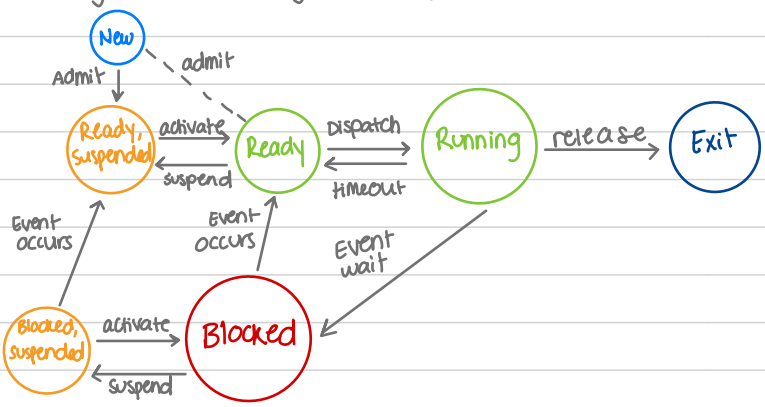


$$\text{Speedup} = \frac{T}{T - fT + \frac{fT}{N}} = \frac{1}{1 - f(1 - \frac{1}{N})}$$ by Amdahl's Law

Managing Concurrency

9/19/22

increasing multiprocessing (concurrency) is a good idea, sometimes. There is a limit.



System Metrics

Measuring:

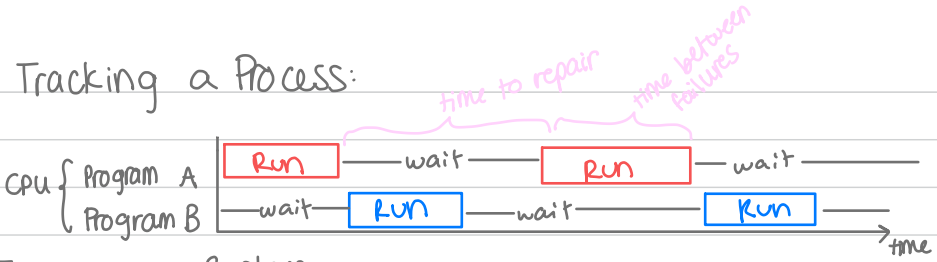
- simulation method (build simulator)
- Build and then measure (very expensive, too set in stone, can't alter)
  Analysis (no code, just math)

System: Resources + Processes

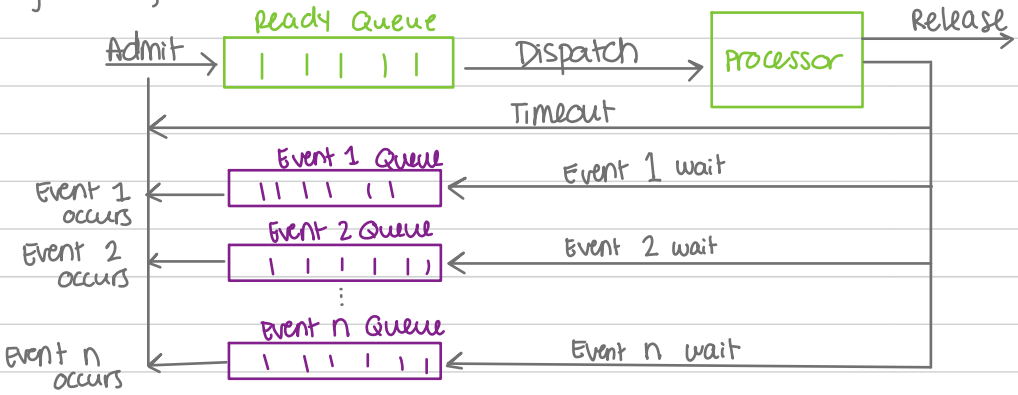
To model: track state of resources and processes

a process is just a consumer of a resource

Tracking a Process:



Tracking a System:



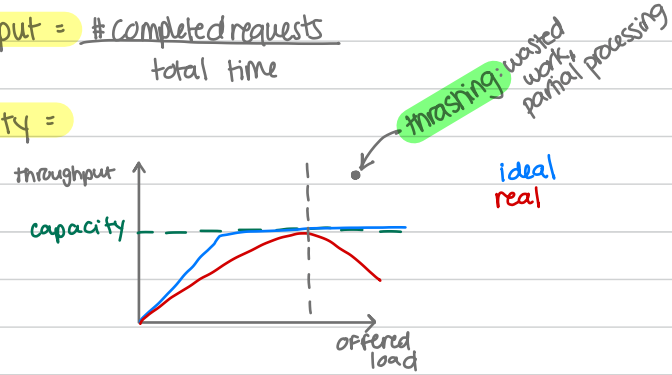
Perspectives to Performance

1. System-centric (more for engineer)
   - are resources being utilized?
   - is revenue being maximized?
   - is system's wear minimized?
2. Process-centric (more for user)
   - average processing time?
   - worst-case processing time?
   - waiting time before service?

Utilization = $\frac{\text{time busy}}{\text{total time}} = 1 - \frac{\text{time idle}}{\text{total time}}$

throughput = $\frac{\#\text{completed requests}}{\text{total time}}$

capacity =



higher **utilization** is always the **bottleneck**, if workload doesn't change

Response time = latency = turnaround time

Service time: process running **waiting time**: from arrival to start of service

Predictability: Difference between best- or average-case response time and worst-case response time

= worst - best

or St. dev

or X percentile

Lateness: how late requests complete service with respect to their deadline

= complete time − deadline

* for entire system, get average of all resources

Availability: Likelihood that the system will be "up" when a request is submitted (per resource)

$$= \frac{\text{up time}}{\text{total time}} = \frac{MTBF}{MTBF + MTTR}$$

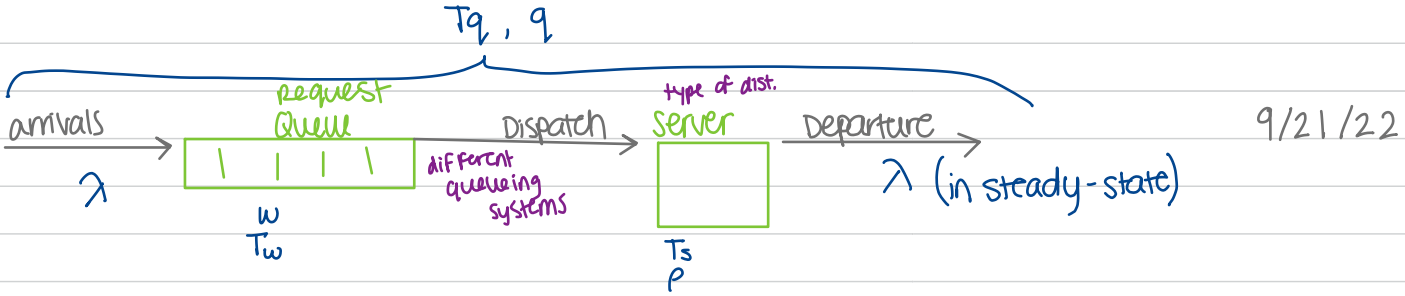
MTBF → mean time btw failures

MTTR → mean time to repair

Reliability: likelihood that the system will be "up" for a certain amount of time

$$R(t) = e^{-t/MTBF}$$

t ← length of time you're asking about



9/21/22

λ = arrival rate

Ts = avg service time

Tw = avg wait time

Tq = avg response time

w = avg. items waiting

ρ = utilization

q = avg items in whole system

* if no steady-state, there is no avg.

assumptions:

1. infinite population - arrival of requests does not impact the arrival of future requests
2. unlimited queue

Calculations:

utilization $\rho = \lambda \cdot T_s$, valid if: $\lambda < \frac{1}{T_s}$ = steady state

single queue only

response time $T_q = T_w + T_s$

Little's Law: Regardless of complexity of system / # of resources

assumption: input flow rate = output flow rate. Implies system at steady-state, no unaccounted deaths (no finite queue)

$q = \lambda \cdot T_q$

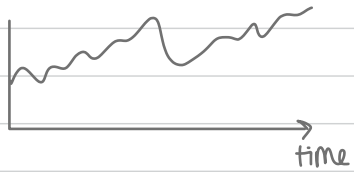
$w = \lambda \cdot T_w$

Using Littles Laws: (still for single resource)

$q = w + \rho$

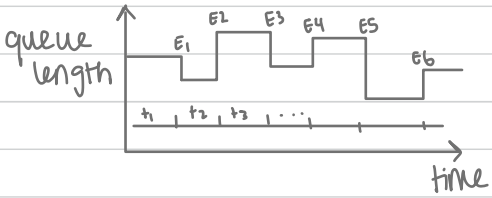
Discrete Event Simulation

Continuous simulation



at each point in time
- weather prediction
- rocket trajectory
- electric signals

Discrete Event Simulation



- queuing systems
- network modeling

IAT = inter arrival time = $1/\lambda$

variables

inputs	outputs
λ	q
T_s	T_q
simulation duration T	ρ

Events *not requests!

in another event queue?
- Birth: event of new request arriving
- Death: the event of service completion
- monitor: the event of logging statistics

request = arrival time
start time
finish time

Birth Event:

1. create request (set): arrival time, start time, finish time (parameters)
2. Add request to queue
3. Generate next Birth event (λ)
4. If this is top of queue, serve it, set start time, start death event (T_s)

Death Event:

1. Remove request from queue, set finish time
2. update statistics
3. Generate Birth at next server
4. If queue not empty, serve next, generate next death event (T_s)

Monitor

1. take snapshot - update statistics
2. add next monitor event

What does controller do?

1. Initialize the simulated system state
2. keep track of time and event timeline
3. Dispatch events at the correct time

★ Building simulator code on slides ★

Probability & Distributions

9/26/22

Indep:

$P(B|A) = P(B)$

$P(A \cup B) = P(A) + P(B)$

$P(A \cap B) = P(A) \cdot P(B)$

RV X:

PDF: $f(x) = P(X = x)$

CDF: $F(x) = P(X \leq x)$

Discrete: $\sum_{-\infty}^{\infty} f(x) = 1$, $\sum_{-\infty}^{x} f(x) = F(x)$ ← can change ranges if $P(x_1 \leq x \leq x_2)$=?

continuous: $\int_{-\infty}^{\infty} f(x)\,dx = 1$, $\int_{-\infty}^{x} f(x)\,dx = F(x)$

$E[X] = \sum_{-\infty}^{\infty} x \cdot f(x) = \mu$

$E[X] = \int_{-\infty}^{\infty} x \cdot f(x)\,dx = \mu$

$E[g(x)] = \sum_{-\infty}^{\infty} g(x)\, f(x)$ ★ integral for continuous

$E[c] = c$

$E[c \cdot g(x)] = c \cdot E[g(x)]$

$E[g(x) + h(x)] = E[g(x)] + E[h(x)]$ } indep.

$E[g(x) \cdot h(x)] = E[g(x)] \cdot E[h(x)]$ }

Distributions

Bernoulli:

$f(x) = \begin{cases} p & \text{(prob of failure)} \\ 1-p \end{cases}$

$\mu = 1 - p$

$\sigma^2 = p(1-p)$

Geometric: # of unsuccessful trials before success (not including success)

$f(x) = p^x \cdot (1-p)$

$\mu = \frac{p}{1-p}$

$\sigma^2 = \frac{p}{(1-p)^2}$

eg: How many transmission attempts on avg needed to send network packet if the network drops 1 packet every 10?

$P(\text{packet loss}) = P(\text{failure}) = p = 0.1$

$\mu = \frac{p}{1-p} = \frac{0.1}{0.9} = 0.111$ (# failures)

∴ mean # of transmission attempts = $\mu + 1$ = 1.111

Binomial: # of successes out of n trials

$f(x) = \binom{n}{x} p^x (1-p)^{n-x}$

↑ success prob (whatever the Q is asking)

$\mu = np$

$\sigma^2 = np(1-p)$

eg: Website has 3 servers. Availability of one server is 0.7. What is the prob that exactly 2 servers are "up"?

$P(\text{a server is up}) = p = 0.7$, $n = 3$, $x = 2$

$f(2) = \binom{3}{2} \cdot 0.7^2 \cdot 0.3^1 = 0.441$

Poisson: arrival rates of binomial trials 5, $n \to \infty$

$\lambda = np = \mu = \sigma^2$

$f(x) = \frac{e^{-\lambda} \cdot \lambda^x}{x!}$

eg:

Exponential: time of interarrival between events

$f(x) = \lambda \cdot e^{-\lambda x}$

$F(x) = 1 - e^{-\lambda x}$

$\mu = 1/\lambda$

$\sigma^2 = 1/\lambda^2$

eg: on avg SSD lives for 5 years. If this duration is exp, what is the prob it last more than 3 years?

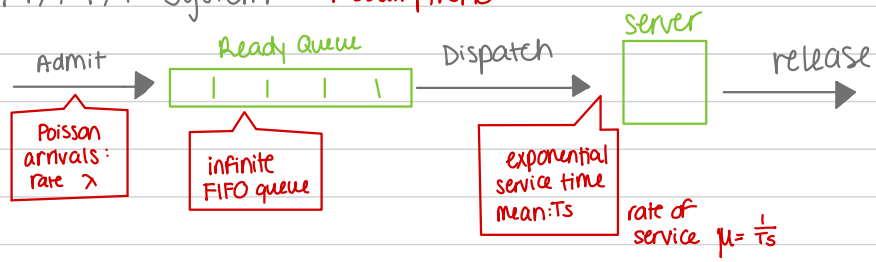
$1/\lambda = 5 : \lambda = 1/5, \quad x > 3$

$P(x > 3) = 1 - p(x \leq 3) = e^{-0.2*3} = \boxed{0.549}$

M/M/1 Queuing System

9/28/22

M/M/1 System - Assumptions



Sj: system state where there is a total of j requests (queued or being served)

h: a period of time small enough such that no arrivals will happen or at most one

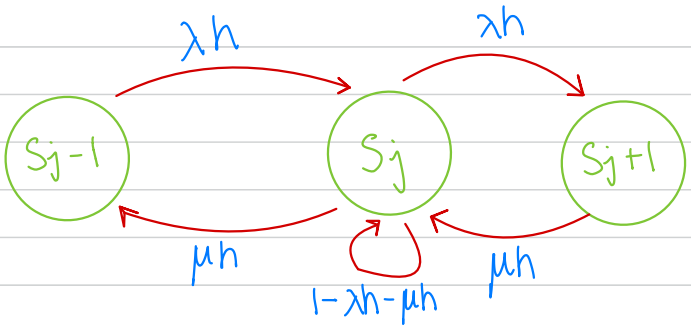
λh: rate of arrival during time window h

Poisson Arrivals: compute $P(n \text{ arrivals in } h) = \frac{(\lambda h)^n}{n!} e^{-(\lambda h)}$, $n = 1, 2, 3 \ldots$

$P(\text{no arrivals in } h) = e^{-(\lambda h)} = 1 - \lambda h + \ldots$, $n = 0$

$P(1 \text{ or more arrivals in } h) = 1 - P(\text{no arrivals in } h) = \lambda h$

$\rho = \lambda/\mu$

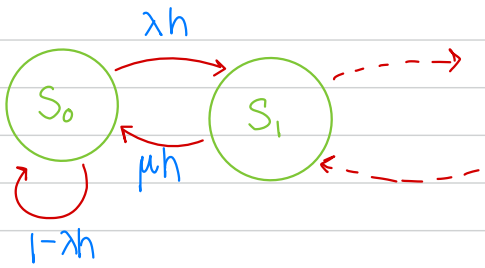


← Prob. of being in Sj

$P(S_j) = \lambda h \, P(S_{j-1}) + \mu h \, P(S_{j+1}) + (1 - \lambda h - \mu h) P(S_j)$

↳ simplified $P(S_{j+1}) = (1+\rho) P(S_j) - \rho P(S_{j-1})$

S_0: system state when there are no requests (system idle)



$P(S_0) = (1-\lambda h)\,P(S_0) + \mu h\,P(S_1)$ (simplified)

$P(S_1) = \rho\,P(S_0)$

↳ $P(S_j) = \rho^j P(S_0)$

how to compute $P(S_0)$:

must hold: $\sum_j P(S_j) = 1$

$$\sum_j \rho^j P(S_0) = \frac{P(S_0)}{1-\rho}$$

$\therefore P(S_0) = 1-\rho$ $\quad P(S_j) = \rho^j(1-\rho)$

"prob that # of requests in system (q) is j"

★ geometric dist: $\mu = E[\#\text{ of requests in system}]$

$\therefore q = \frac{\rho}{1-\rho}$ in single resource system

In M/M/1:

$q = \frac{\rho}{1-\rho}$ $\quad \rho = \lambda \cdot T_s$

$q = w + \rho$

↳ $w = \frac{\rho^2}{1-\rho}$ $\quad A = 1$

slowdown because of queue $= \frac{T_q}{T_s} = \frac{1}{1-\rho}$

From Littles Law: $T_q = \frac{T_s}{1-\rho}$ $\quad T_w = \frac{\rho T_s}{1-\rho} = \frac{w}{\lambda}$

10/3/22

example: hits to a single-process, single threaded web server follow a poisson arrival process with mean 8 req/sec. T_s per request is Exp dist with mean 100msec.

Calculate T_q and slowdown.

$\lambda = 8$ req/sec $\quad T_q = ?$

$T_s = 0.1$ sec $\quad$ slowdown $= ?$

model as M/M/1 after checking properties

$\rho = \lambda \cdot T_s = 0.8$

$T_q = \frac{T_s}{1-\rho} = \frac{0.1}{1-0.8} = \frac{0.1}{0.2} = 0.5$ sec

slowdown $= \frac{T_q}{T_s} = \frac{0.5}{0.1} = 5$ (times slower)

- Due to an ad campaign, hits to the web server went up 20%. What is the new T_q?

$\lambda' = \lambda * 1.20 = 9.6$ req/sec

$\rho' = \lambda' * T_s = 9.6 * 0.1 = 0.96$

$q' = \frac{\rho'}{1-\rho'} = \frac{0.96}{0.04} = 24$

$T_q' = q'/\lambda' = 24/9.6 = 2.5$ sec

- how much faster should the web server be so that the slowdown after the ad campaign would be no more than 8 as required by a service level agreement (SLA).

$$\text{slowdown} = \frac{1}{1-\rho} \leq 8$$

$1 \leq 8(1-\rho)$

$1 \leq 8 - 8\rho$

$7 \geq 8\rho$

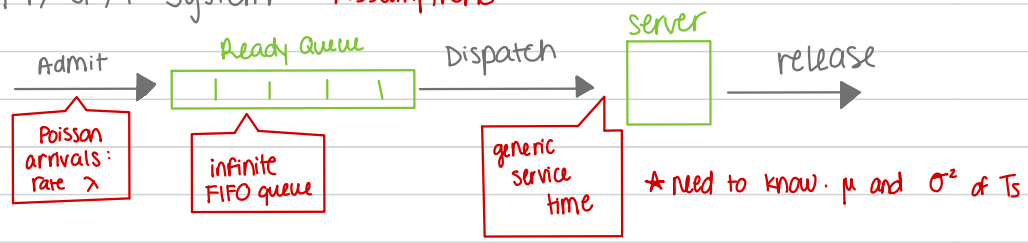
$\rho \leq 7/8 = 0.875$

$T_S \leq 0.875/\lambda$

$\leq 0.875/9.6 = 0.091$ sec

M/G/1 Queuing System

M/G/1 System - Assumptions



* need to know μ and σ^2 of T_S

$$A = \frac{1}{2}\left[1 + \left(\frac{\sigma_{T_S}}{T_S}\right)^2\right]$$

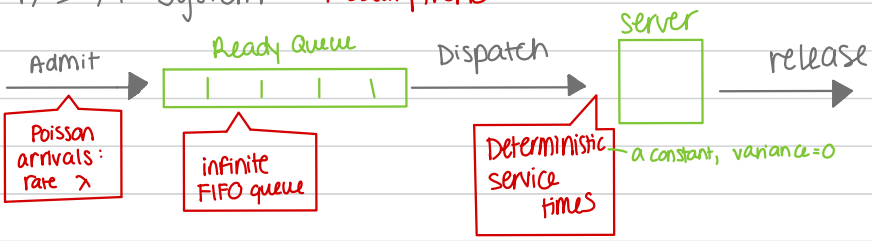
$$\rho = \lambda * T_S$$

$$q = \frac{\rho^2 A}{1-\rho} + \rho$$

$$w = \frac{\rho^2 A}{1-\rho}$$

M/D/1 Queuing System

M/D/1 System - Assumptions



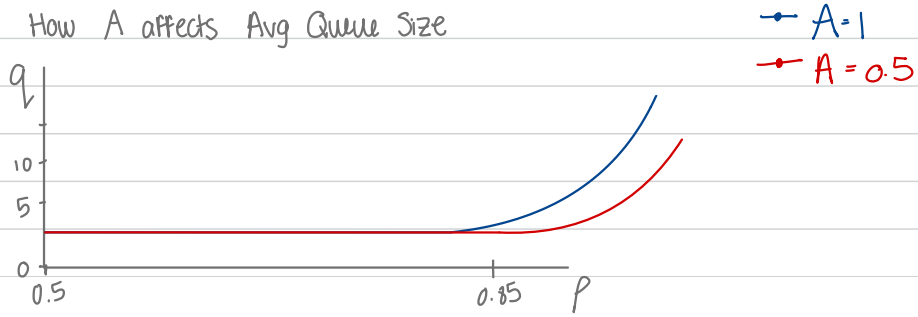
$$A = \frac{1}{2}$$

$$\rho = \lambda * T_S$$

$$q = \frac{\rho^2}{2(1-\rho)} + \rho$$

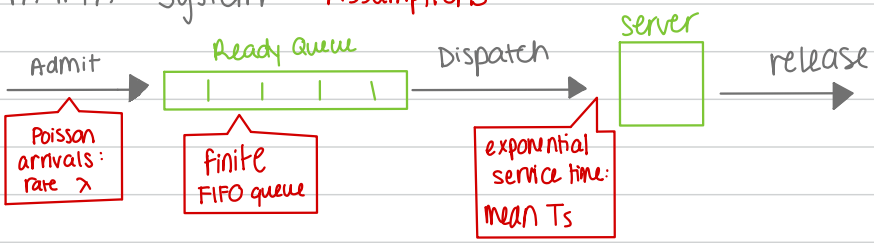
$$w = \frac{\rho^2}{2(1-\rho)}$$

How A affects Avg Queue Size



M/M/1/K Queuing System

M/M/1/K System - Assumptions (K: length of queue)



request dropped if queue is full

at most K states/requests

$$P(S_i) = \begin{cases} \frac{(1-\rho)\rho^i}{1-\rho^{K+1}}, & \text{for } \rho \neq 1 \\ \frac{1}{K+1}, & \text{for } \rho = 1 \end{cases}$$

$\rho = \lambda T_s$, can be greater than 1 (ρ and λ: original ρ and λ)

$$q = \begin{cases} \frac{\rho}{1-\rho} - \frac{(K+1)\rho^{K+1}}{1-\rho^{K+1}}, & \rho \neq 1 \\ \frac{K}{2}, & \rho = 1 \end{cases}$$

If there's a rejection = M/M/1/K

$$T_q = \frac{q}{\lambda'}$$

if queue is full: $\lambda' = \lambda \cdot (1 - P(S_K))$

$$T_q = \frac{q}{\lambda'}$$

$$P(\text{rejection}) = P(S_K) = \begin{cases} \frac{(1-\rho)\rho^K}{1-\rho^{K+1}} & \text{if } \rho \neq 1 \\ \frac{1}{K+1} & \text{if } \rho = 1 \end{cases}$$

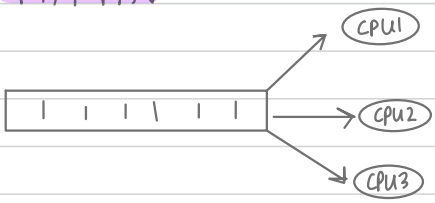
can apply Little's law after this $P(S_K)$ is found

Discussion

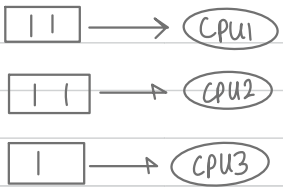
10/5/22

Better! CPUs never idle, higher utilization

M/M/N



N * M/M/1



The buffer of a streaming media application (e.g., mp3 player) can hold up to N packets. The processing time for each packet to be played out was found to be exponential with a mean of 25 msec. Degradation in the quality of the playout occurs as a result of buffer over-runs or as a result of a buffer under-runs. Buffer over-runs occur when a packet is dropped due to a filled buffer (causing a "blip" in the playout). Buffer under-runs occur when there are no packets to playout (causing a period of "plop" in the playout). It was measured that the packets arrive to the application as a Poisson process with a mean of 36 packets per second. Answer the following questions:

$T_s = 25\text{ ms} = 0.025\text{sec}$ M/M/1

$\lambda = 36$

Assuming an infinite buffer, how many packets do you expect to find in the buffer (i.e., being played out or waiting to be played out)? What is the delay between receipt of a packet and the beginning of its playout?

$\rho = \lambda \cdot T_s = 36 \cdot 0.025 = 0.9$

$q = \frac{\rho}{1-\rho} = \frac{0.9}{0.1} = \boxed{9}$

$w = q - \rho = 9 - 0.9 = 8.1$

$T_w = \frac{w}{\lambda} = \frac{8.1}{36} = \boxed{0.225}$ seconds

Assuming an infinite buffer and assuming that a streaming media object consists of 10,000 packets, how many "blips" and how many "plops" do you expect to hear for that object?

blips = 0, because queue is infinite

plops = $10000 * (1-\rho) = 10000 * (0.1) = 1000$

Assuming N=4 packets, how many packets do you expect to find in the buffer? What is the mean delay between receipt of a packet and the beginning of its playout?

M/M/1/4

$q = \frac{0.9}{0.1} - \frac{5(0.9)^5}{1-0.9^5} = 1.79$

$\lambda' = \lambda \cdot (1 - P(S_k)) = 36(1-0.16) = 36(0.84) = 30.24$

$w = q - \rho' = 1.79 - 0.75 = 1.04$

$T_w = \frac{w}{\lambda'} = \frac{1.04}{30.24} = 0.034$

Assuming N=4 packets and that a streaming media object consists of 10,000 packets, how many "blips" and how many "plops" do you expect to hear for that object?

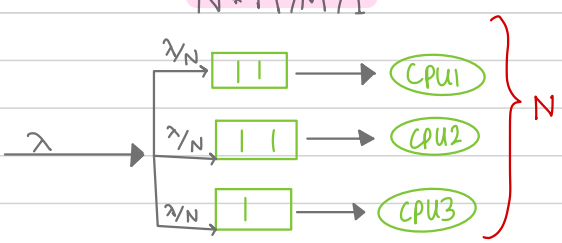
blips = $10000(0.16) = 1600$

plops = $(1-\rho')(10000) = 2500$

Queuing Networks

10/5/22

N * M/M/1



$\rho' = \frac{\lambda T_s}{N}$ for each individual server

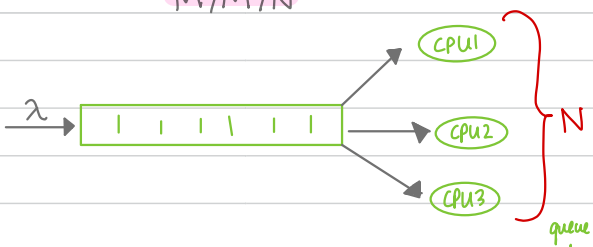
$q' = \frac{\rho'}{(1-\rho')}$ at one server.

for whole thing, multiply by N

$T_q = \frac{q'}{\lambda/N}$

$\lambda' = \frac{\lambda}{N}$, then all same formulas as M/M/1

M/M/N



$P(\text{all servers busy}) = C = \frac{1-K}{1-\rho K}$

Poisson ratio = $K = \frac{\sum_{i=0}^{N-1} \frac{(N\rho)^i}{i!}}{\sum_{i=0}^{N} \frac{(N\rho)^i}{i!}}$

$q = \frac{\rho}{1-\rho} \cdot C + N\rho$ (queue build up; requests being served)

of entire system

$\rho = \frac{\lambda T_s}{N}$ of single server

Under same condition, which one has lower T_q? M/M/N , in general
Behave in similar way?

light load, pretty similar
heavy load, pretty similar
in between, M/M/N has higher utilization

2 servers. traffic of 20 req/sec. $T_q = 0.08$ sec. Compare response time M/M/2 vs. 2* M/M/1

$T_q = 0.08$ sec
$\lambda = 20$ req/sec
$N = 2$

2* M/M/1

$\rho = \frac{\lambda T_s}{N} = 0.8$ per server

$q' = \frac{\rho}{1-\rho} = 4$ per server

entire system:
$q = 4 * N = 4 * 2 = 8$
$T_q = q/\lambda = 8/20 = 0.4$

M/M/2

$\rho = \frac{\lambda T_s}{N} = 0.8$ per server

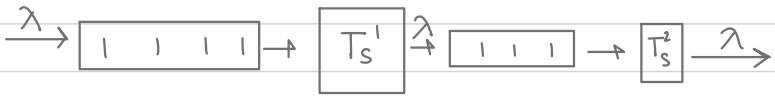
$K = 0.67$
$C = 0.71$
$q = 4.44$ entire system
the smaller the better

$T_q = \frac{q}{\lambda} = 0.22$

∴ M/M/2 is better

Pipelining

at steady state, rate in = rate out



Splitting

Merging

Jackson Theorem

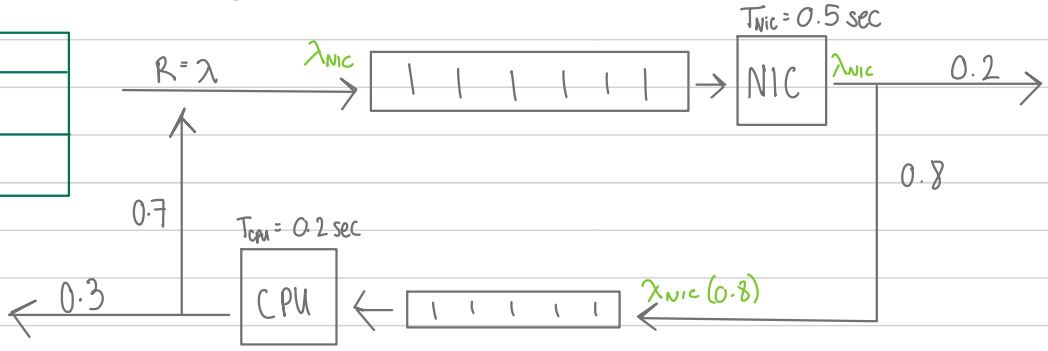
assumption: steady-state, arrival from outside is Poisson, service time is exponential, infinite queues with FIFO.

in a network of queues, each queue can be analyzed in isolation of all others

10/11/22

Consider a web crawling engine. When a webpage is fetched, the network interface (NIC) is first involved in fetching the content of an HTML page. With 20% probability, the network transaction fails and nothing else needs to be done. Otherwise, the obtained HTML page is parsed at the CPU. With probability 0.7 after CPU parsing, the page contains a link that needs to be recursively crawled, as if it was a new crawling request. If the page contains no further link (or an already visited link), crawling completes. The processing times for NIC and CPU are exponentially distributed and given in the table below. Assume that requests at the engine arrive according to a Poison distribution with rate R.

Request Processing Times	
CPU	0.2 sec
NIC	0.5 sec



★ throughput must be R if steady-state

an going into NIC, 2 diff paths

$\lambda_{NIC} = R + \lambda_{NIC} * 0.8 * 0.7$

rearrange: $\lambda_{NIC} = \frac{R}{1 - 0.8 * 0.7} = \frac{R}{0.44}$

OR

input rate / output rate

$R = \lambda_{NIC} * 0.2 + \lambda_{NIC} * 0.8 * 0.3$

rearrange: $\lambda_{NIC} = \frac{R}{0.2 + 0.8 * 0.3} = \frac{R}{0.44}$

Find bottleneck: (highest utilization)

$\rho_{NIC} = \underbrace{\frac{R}{0.44}}_{\lambda_{NIC}} * T_{NIC}$ ← bottleneck

$\lambda_{CPU} = \lambda_{NIC} * 0.8 = \frac{0.8R}{0.44}$

$\rho_{CPU} = \frac{0.8R}{0.44} * T_{CPU}$

capacity:

find max value of R, such that $\rho_{NIC} \to 1$

$\frac{R^* \cdot T_{NIC}}{0.44} = 1$

cap of system: $R^* = \frac{0.44}{T_{NIC}} = 0.88$

$\mu_{NIC} = \frac{1}{T_{NIC}} = \frac{1}{0.5} = 2$

Capacity Review

single resource:

cap = $\mu = \frac{1}{T_S}$, when $\rho \to 1$

throughput = λ

multiple: find bottleneck, then input rate = 1, find capacity

From M/M/1 to GPS

10/17/22

Generalized processor sharing (GPS)

- partition the capacity among users who want access to it
- each request class is assigned a weight $w \in [0, 1]$. The bigger the weight, the bigger the share
  - weights $w_1 + w_2 + \dots w_n \leq 1$
    - can save some weight for the future

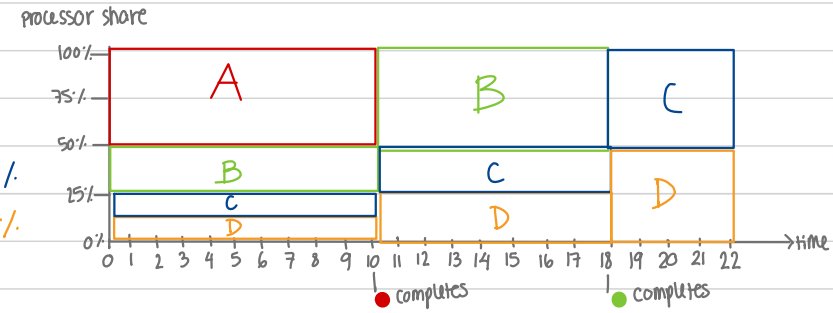
example:

A - 50%

B - 25%

C - 12.5%

D - 12.5%



Granularity vs Reality: no matter which system, perfect capacity distribution is unfeasible

A "practical" GPS:

μ_s = server capacity: req/sec $\left.\vphantom{\sum_{i=1}^{n}}\right\} \sum_{i=1}^{n} w_i \leq 1$

$w_i = w_1 = w_2 = w_3 \dots = w_n$

|↔| = time slice (quantum s)

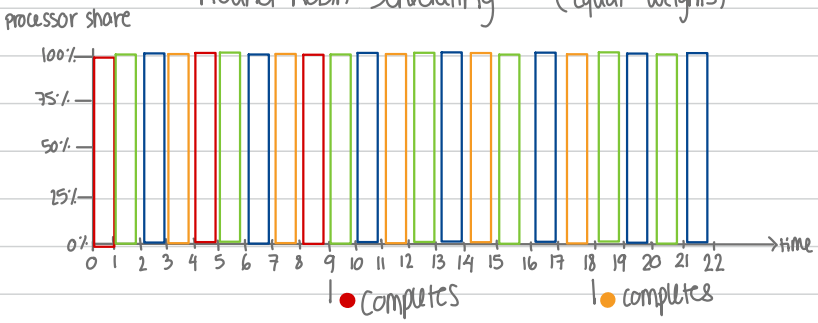
Round-Robin Scheduling (equal weights)

A - 25%

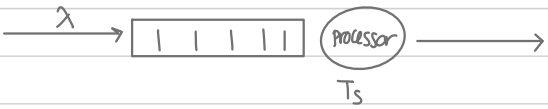
B - 25%

C - 25%

D - 25%



Round Robin as GPS



assume: RR slice of length s

each job has its own class

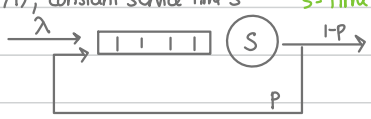
- we dont know length of each job

So,

p: prob that job will not be done at the end of a slice

1-p: prob that a job will be done at the end of a slice

(M/D/1), constant service time s s = time slice



Need to find λ'

$\lambda = \lambda'(1-p)$

$\lambda' = \frac{\lambda}{1-p}$

$\rho = \lambda' \cdot s = \frac{\lambda s}{1-p} = \lambda \cdot T_S$

(M/M/1), s is exponentially distributed T_S with mean s

$t = (m+1)s$ t = time units to complete the job

$f(m) = p^m (1-p)$ m = how many times you fail and ask for service

↳ m ~ Geometric (p)

$E[m] = \frac{p}{1-p}$

$T_S = E[t] = (E[m]+1)s = \frac{s}{1-p}$

$q = \frac{\rho}{1-\rho}$

$T_q = \frac{q}{\lambda}$

☆ a work conserving scheduling policy that is also length-unaware does not impact system averages

* when the system is heavily utilized, and we can differentiate length and/or request classes, scheduling has a big impact on the perceived service from a user-centric perspective

Resource Scheduling

10/19/22

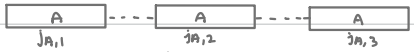
optimal scheduler: the scheduler that always takes the best decision to optimize some metric

- the more resources, the harder it is to get a perfect scheduler
- the type of resources affect the process of getting a perfect scheduler
- take into consideration the metric you are looking for

Consider one resource - **Processor**:

- can it be interrupted to do something else? (yes, **preemptive**) (**non preemptive** = cannot be interrupted) ← FIFO is non-preemptive
- **stateful**: the time to serve a request is impacted by history vs. **stateless**
- How to describe a scheduler
- cost of scheduler?
- **I/O-bound process** vs **processor-bound process** (most time spent on I/O or processor, respectively)

Tasks & Jobs

task: A - 

a sequence of **jobs**

j_A, a generic job of task A

Evaluating a Scheduler

system centric: utilization, throughput, complexity

process centric: response time ($R_A = t_f - t_r$), fairness (how different is the quality of service received by tasks [ie: slowdown imbalance])

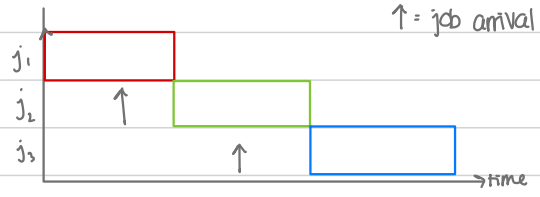
t_f: finish time of task A; t_r: arrival time of task A

A Scheduler

① Scheduler Invocation: when it wakes up and takes a decision

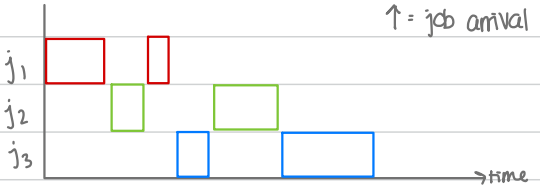
② Scheduling Policy: how the decision is taken

schedule produced by scheduler:



quantum slice size

$q = 1$



if wait times (since last time they received service) are equal, tie-breaker: job with higher ID gets the processor

FIFO

avg response time for FIFO: $\bar{R} = \frac{1}{n}\sum_{i=1}^{n} R_i$

Fairness in FIFO: qualitatively fair (will always eventually be served)

quantitatively fair (response time may be worse off)

consider slowdown of shortest job = $\frac{\text{response time}}{\text{service time}} = \frac{R}{C}$

Round Robin

avg response time is calculated the same as in FIFO

fairness: slowdown of shortest job

$q \to \infty$ then it becomes non preemptive FIFO

$q \to 0$ then it becomes a fair GPS

$q = q \gg 100 \cdot o_{rr}$

↖ overhead = wasted time on implementing/calculating the policy, not actually doing the tasks

Job-Aware Scheduling

10/24/22

Shortest Job Next (SJN):

scheduler invocation: completion of a job (non-preemptive)

scheduler policy: rank tasks by length C_i, shortest goes first after its arrival time

performance:

calculate all response times R_i = finish time − arrival time

Avg. response time = $\frac{1}{n}\sum_{i=1}^{n} R_i$

fairness: slowdown of slowest job

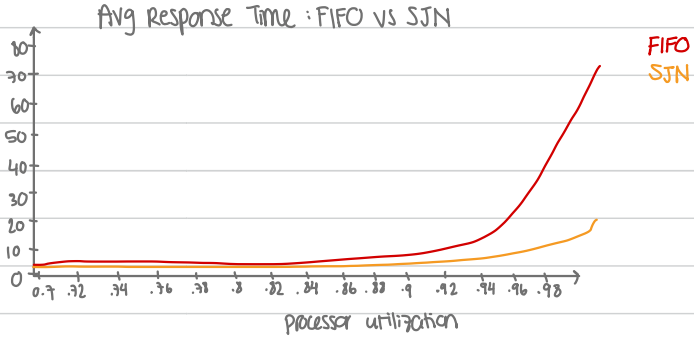
problems with SJN:

1. not preemptive
2. longer jobs may not execute if have a train of shorter jobs
   ↳ "starvation"

Avg. Response time shortest → longest

1. SJN
2. FIFO } with many jobs, these are very similar.
3. RR

Avg Response Time : FIFO vs SJN



Shortest Remaining Time (SRT):

scheduler invocation: arrival of job, completion of job (preemptive)

scheduler policy: rank tasks by remaining execution time, execute tasks with shortest remaining time

still have starvation!

Solving Starvation

- consider how long a job has been waiting?
  - job wait time grows ⟶ slowdown grows

minimum slowdown at $t = \frac{t - a_i + C_i}{C_i}$ (a_i ← arrival time of T_i (task i); C_i ← length of job or task)

Highest Slowdown Next (HSN)

scheduler invocation: completion of a job (non-preemptive)

scheduler policy: rank tasks by minimum slowdown at t (current time), execute task with highest slowdown

To Find Job Length:

1. static analysis - dissect the binary, model the hardware, find the time
2. measurements - run it enough times offline, then use that online
3. estimations - as it runs, collect measurements and predict

- observe $C_{i_1}, C_{i_2}, \ldots C_{in}$ for task T_i

$$\bar{C}(n) = \frac{1}{n}\sum_{k=1}^{n} C_{i,k}$$

↳ use this as length of $j_{i,n+1}$

problem: takes up a lot of memory, poor accuracy if job length fluctuates over time

solution: keep only $\bar{C}(n-1)$ $\sum_{k=1}^{n-1} C_{i,k}$

then $\bar{C}(n) = \frac{\bar{C}(n-1)\cdot(n-1) + C_{i,n}}{n}$

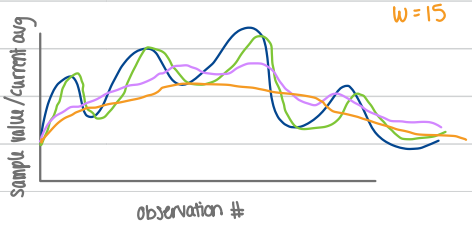
Sliding Window Avg:

- only use w most recent samples, then calculate that avg

$$\bar{C}(n) = \frac{1}{w}\sum_{k=n-w+1}^{n} C_{i,k}$$

$$= \bar{C}(n-1) - \frac{C_{i,n-w}}{w} + \frac{C_{i,n}}{w}$$

rolling avg



A scheduling Algorithm: 10/26/22

- used to determine which ready processes should be selected to execute on processor
- can be non-work conserving
- potentially makes big difference in avg response time if work-conserving
- can impact capacity of stateless resource
- can behave better than FIFO if job lengths are known

Exponentially weighted Averages

- decide the weight $\alpha \in [0,1]$

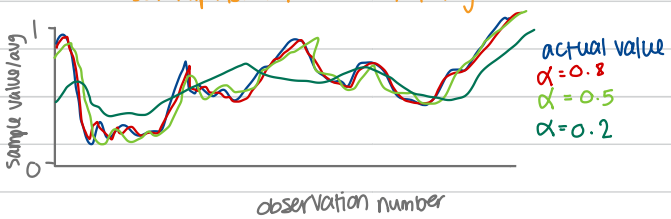
$\bar{C}(n) = \alpha C_{i,n} + (1-\alpha)\bar{C}(n-1)$

↳ recursive

recursion: k steps

$\bar{C}(n) = \alpha[C_{i,n} + (1-\alpha)C_{i,n-1} + \ldots + (1-\alpha)^k C_{i,n-k}] + (1-\alpha)^{k+1}\bar{C}(n-(k+1))$

* loses importance faster if k is high

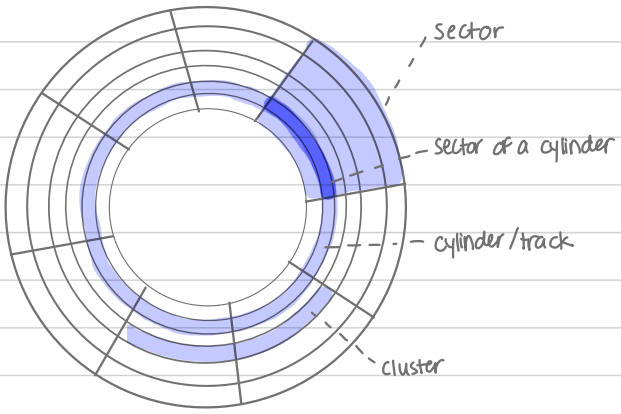


Scheduling a Stateful Resource

Stateless: if the amount of time to service of a given request is independent from the history of requests that have been served

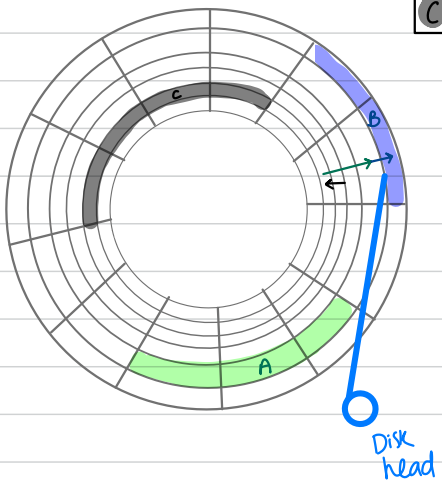
stateful: .. it does depend on previously serviced workloads

Hard Disk:



each read/write request targets a cluster

Shortest Scan First (SSF):



Implementation: ✓

need to know the disk's geometry

need to keep track of head position and reorder queue

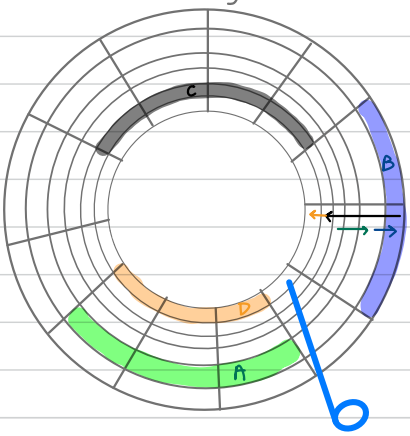
Head Movement: ✓

- Tries to minimize head movement
- Not necessarily the best EVER, though

Fairness: ✗

- A bunch of co-located requests can delay indefinitely
- Head could stay stuck for a while in one spot

SCAN Scheduling (elevator):



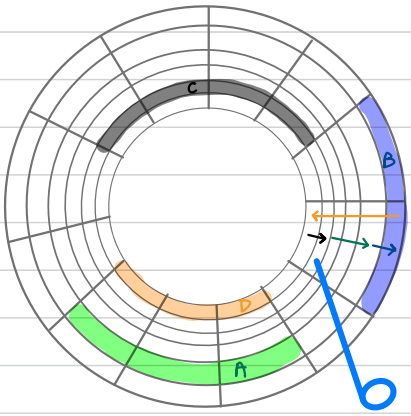
Direction = Down

Implementation: ✓
- need to know the disk's geometry
- need to keep track of head position and reorder queue

Fairness: ✓
- Stickiness between few cylinders solved
- pretty bad worst case

Head Movement: ✓
- Tries to minimize head movement
- Head does not jump all over the place

C-SCAN Scheduling:



Implementation: ✓
- need to know the disk's geometry
- need to keep track of head position and reorder queue

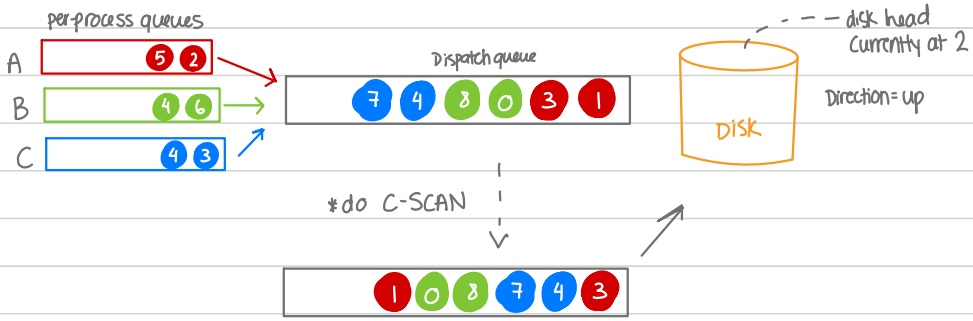
Fairness: ✓
- Stickiness between few cylinders solved
- better worst case than SCAN

Head Movement: ✓
- Tries to minimize head movement
- Head does not jump all over the place

Complete Fair Queuing Scheduling (CFQ):

Round Robin

$q_{CFQ} = 2$



Implementation: ✗
- need to keep track of head position reorder queue
- keep a queue per each process and perform RR

Fairness: ✓
- no stickiness possible
- delay always bounded thanks to RR

Head movement: ✓ • minimize head movement in the dispatch queue • always move in one direction (up)

10/31/22

DRAM: store memory between CPU and disk

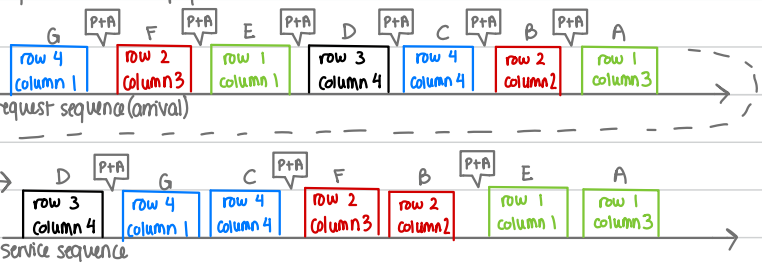
- each read/write request targets a col/row

FR-FCFS DRAM Scheduler:

"First ready - first come, first serve"

- for any stateful resource
- starvation exists (not fair)

at t=0, row buffer is empty



implementation: ✓

- has to be done in hardware
- only keep track of currently open row

Fairness: X

- a request can be indefinitely delayed
- starvation of memory requests

memory throughput: ✓

- Maximum throughput for outstanding requests
- Least amount of P+A operations ???

FR-FCFS + CAP DRAM Scheduler:

- maximum reordering until someone suffers

at t=0, row buffer is empty

[X] = suffered delay, cap = 2!

implementation: ✓X

- keep track of currently open row
  track accumulated delay per request

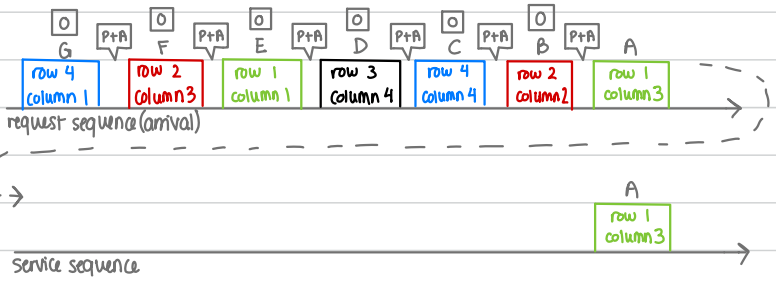
Fairness: ✓

- capped re-ordering
- No starvation of memory requests

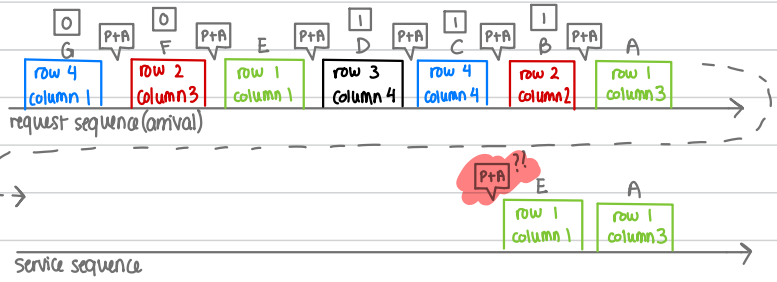
Memory Throughput: ✓

- good throughput for outstanding requests
- Some extra P+A but nothing DRAMatic

(1)

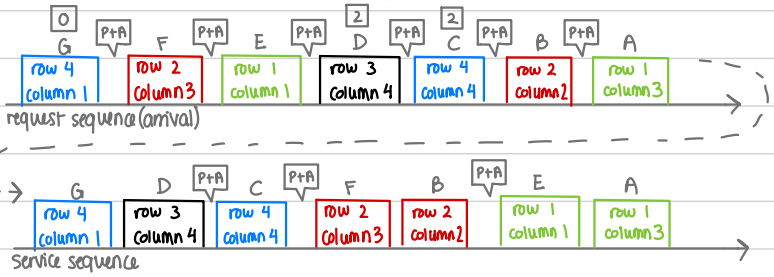


(2)



cap reached

(3)



Real Time Scheduling

predictability

real-time system: always consider worst-case

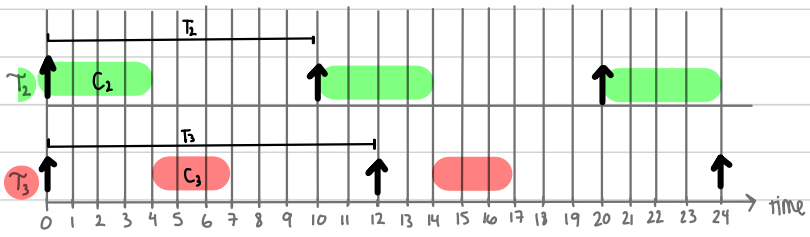
Real-Time System Model

C_i = task τ_i's worst case time length (WCET)

T_i = task τ_i's period = inter-arrival of job arrivals for τ_i

D_i = task τ_i's deadline (for us, $D_i = T_i$)

task	T	C
τ_2	10	4
τ_3	12	3



Static Priority Scheduling

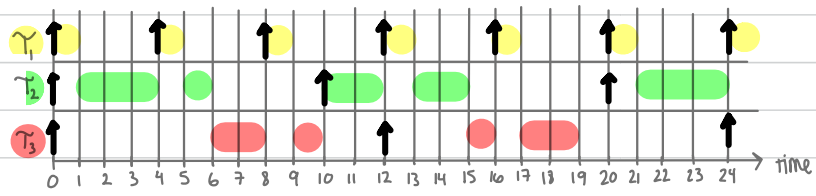
1. look at parameters of task set (C, T)
2. Assign offline a priority to each of them
3. At runtime, select the ready task with higher priority

Rate Monotonic Scheduler (RM):

scheduler invocation: arrival of a job, completion of a job (preemptive)

scheduler policy: rank tasks by period, execute ready task with shortest period

task	T	C
τ_1	4	1
τ_2	10	4
τ_3	12	3



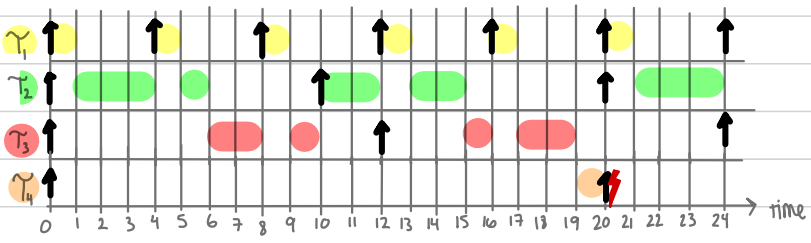
•• task set is schedulable under RM

→ no deadline misses

11/2/22

ϟ = deadline miss

task	T	C
τ_1	4	1
τ_2	10	4
τ_3	12	3
τ_4	20	2



•• task set is not schedulable under RM

→ deadline miss

can we know in advance if they are schedulable? YES: $\frac{1}{T_i} \cdot C_i = U_i$

for the whole set: $U = \sum_{i=1}^{m} \frac{C_i}{T_i}$

schedulable if: $U \leq m(2^{\frac{1}{m}} - 1)$

when $m \to \infty$ = 0.69

Dynamic Priority Scheduling

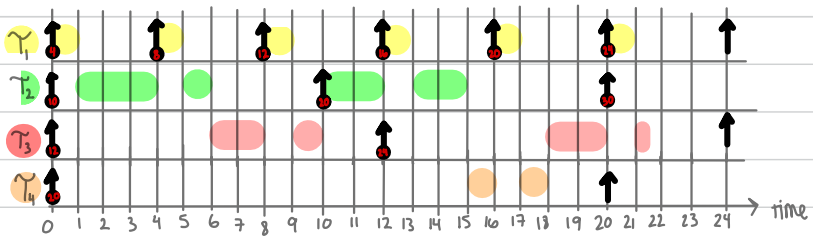
1. run the system and track tasks online
2. for each job (not task!), decide it's priority
3. select the job with higher priority

Earliest Deadline First Scheduler (EDF):

Scheduler invocation: arrival of a job, completion of a job (pre-emptive)

Scheduler policy: online, rank jobs by deadline's proximity, execute ready task with closest deadline

task	T	C
τ_1	4	1
τ_2	10	4
τ_3	12	3
τ_4	20	2



tie breaker: lower job ID wins

∴ task set is schedulable under EDF

→ no deadline misses

★ EDF is optimal on single-core processors

can we know in advance if they are schedulable? YES: $\frac{1}{T_i} \cdot C_i = U_i$

for the whole set: $U = \sum_{i=1}^{m} \frac{C_i}{T_i}$

schedulable iff: $U \leq 1$

In real-time system

RM ≤ 69% EDF ≤ 100%

better because:

- complexity
- workload composition
- robustness to tardiness

Multi-Processor RT Scheduling

Partitioned Multi-Processor Scheduling: (like N*M/M/1)

1. offline, consider parameters and assign tasks to processors
   - 1.1). try to assign a task
   - 1.2). check schedulability of that particular processor
2. Apply single-core scheduling on each core independently

Partitioned RM

Task	T_i	C_i	U_i
τ_1	10	3	0.3
τ_2	15	6	0.4
τ_3	20	6	0.3
τ_4	32	18	0.56
τ_5	8	2	0.25

$U_1 \leq 0.69$ T_1? → P1

$U_1 + U_2 \not\leq 0.69$ but $U_1 + U_2 \leq 2(2^{\frac{1}{2}} - 1) = 0.83$

★ steps on slides!!

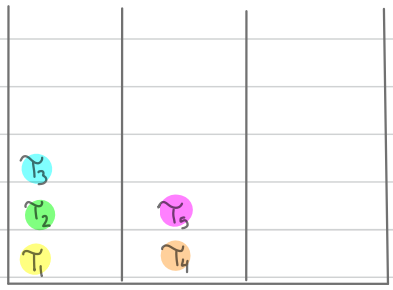
Partitioned RM + First-Fit Assignment

for whole set: $U = \sum_{i=1}^{m} \frac{C_i}{T_i}$

Schedulable under RM-FF if: $U \leq N \cdot (\sqrt{2} - 1)$

Partitioned EDF:

Task	T_i	C_i	U_i
τ_1	10	3	0.3
τ_2	15	6	0.4
τ_3	20	6	0.3
τ_4	32	18	0.56
τ_5	8	2	0.25



$U_1 \leq 1$ — τ_1? yes

$U_1 + U_2 \leq 1$ — τ_2? yes

...

* Steps on slides!!

Partitioned EDF + First-Fit Assignment

for whole set: $U = \sum_{i=1}^{m} \frac{C_i}{T_i}$

Schedulable under EDF-FF if: $U \leq \frac{\beta N + 1}{\beta + 1}$, $\beta = \left\lfloor \frac{1}{\max_k U_k} \right\rfloor$

↳ each cpu is schedulable

↳ task with max utilization

Global Multi-Processor Scheduling (like M/M/N)

11/7/22

1. online, rank all the tasks/jobs in the shared ready queue
2. select top N to run on the N available processors

max Utilization of 2 processors is 2 ... so on

Global EDF:

← check what to do at each time unit

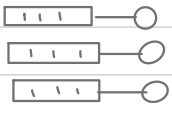
Scheduler invocation: arrival of a job, completion of a job (pre-emptive)

Scheduler policy: online, rank jobs by deadline's proximity, execute N ready task with closest deadline

example on slides!

* some low-utilization (compared to # cpus you have) task-sets are not schedulable, even with large N

↳ The Dhall's Effect

Global Multi-Processor Scheduling	Partitioned Multi-Processor Scheduling
	
- high complexity - good efficiency if have good policy	- not good efficiency - very easy to perform

Concurrency & Mutual Exclusion

Concurrency:

multiprogramming: no real concurrency - interleaved execution at different resources

multiprocessing: real concurrency & interleaved execution

Distributed processing: all of the above, but with large communication delays

Concurrent Processes:

1. might need to share resources (I/O device, co-processor)
2. might need to share data (memory data structure, file)

uncontrolled access to shared resources/data is recipe for disaster!

Uncontrolled Sharing: → data race - ending up with different results

step	task 1	task 2
1	x = c	
2	x = x+1	
3	c = x	
4		x = c
5		x = x+1
6		c = x

c = 17

or

step	task 1	task 2
1	x = c	
2		x = c
3		x = x+1
4		c = x
5	x = x+1	
6	c = x	

c = 16

new way of drawing, no time units

Concurrency done Right:

Critical sections: when a process manipulates shared resources/data

★ goal: How to design a protocol for critical sections so that the final result does not depend on exact scheduling or interleaving?

Mutual Exclusion: sharing resource R is safe if at any time only one process is in a critical section that manipulates R

↳ at most one process can execute code in critical section

idea: place extra code each time a process needs to enter/leave a critical section

entry section: section of code that implements the entry protocol, only one process can go into critical section

exit section: section of code that implements the exit protocol

Code to Achieve Mutual Exclusion:

assumptions: 1. processes are alive (progress at non-zero speed)
2. progress is unknown
3. schedule is unknown (assume the worst)
4. many cpus in system (simultaneous access to the same memory location not possible)

properties: 1. mutual exclusion (at most one process in a given critical section CS)
2. isolation (only processes trying to enter a CS get to decide which one enters)
3. progress - no deadlocks (if no process in CS and processes want to enter, then some one will)
4. no starvation (no process waits an infinite amount of time before entering a CS)

Possible Solution: Disable Scheduler, not possible!

2 Party Mutual Exclusion

Process i — index of current process

```
repeat:
    remainder section
    [entry section]
    critical section!
    [exit section]
    remainder section
forever
```

taking turns: Process i

```
repeat:
    while (turn != i): — busy-loop until it is process i's turn
    critical section!   (not in loop)
    turn = j — once done, give turn to process j
forever
```

Isolation is not satisfied

11/9/22

lock variable: initialized to false

```
Process i:
repeat:
    while (locked):
    locked = true
    critical section
    locked = false
forever
```

step	task 1	task 2
1	while (locked)	
2		while (locked)
3		locked = true
4		crit. section
5	locked = true	
6	crit. section	

mutual exclusion not satisfied!

signaling intent: flag per process where they notify intent to enter CS

```
Process i:
repeat:
    flag[i] = true
    while (flag[j]):
    critical section
    flag[i] = false
forever
```

step	task 1	task 2
1	flag[1] = T	
2		flag[2] = T
3	while flag[2]	
4		while flag[1]
5	while flag[2]	
6		while flag[1]

Deadlock!

signaling intent and backing off:

```
Process i:
repeat:
flag[i] = true
while (flag[j]) {
    flag[i] = false
    delay();
    flag[i] = true}
critical section
flag[i] = false
```

if other process has also signaled intent, back off for a bit.

step	task 1	task 2
1	flag[1] = T	
2		flag[2] = T
3	flag[1] = F	
4		flag[2] = F
5	flag[1] = T	
6		flag[2] = T

no guaranteed Progress!

Dekkers Algorithm: signaling intent and switching who should back off:

```
Process i:
repeat:
    flag[i] = true
    while (flag[j]) {
        if (turn == j) {
            flag[i] = false
            while (turn == j);
            flag[i] = true
        }
    }
    critical section
    turn = j
    flag[i] = false
forever
```

with turn n set to j, it's task i's turn to back off

after CS, release resource and switch turn

- mutual exclusion ✓
- isolation ✓
- progress ✓
- no starvation ✓
  - ↳ no deadlock
- fairness not guaranteed

↳ **Peterson's Algorithm:** Fairness fixed

```
Process i:
repeat:
    flag[i] = true          } entry section / lock procedure
    turn = j
    while (flag[j] && turn == j);
    critical section
    flag[i] = false         } exit section / unlock procedure
forever
```

- mutual exclusion ✓
- isolation ✓
- progress ✓
- no starvation ✓
  - ↳ no deadlock
- fair ✓

Out of Order Processors (OOO)

Barriers: place between instructions, limit reordering

Spinlock Implementation

Semaphore implementation (using spinlock)