

CSCI 350 Ch. 7 – Scheduling

Mark Redekopp Michael Shindler & Ramesh Govindan

Overview

- Which thread should be selected to run on the processor(s) to yield good performance?
- Does it even matter?
 - Does the common case of low CPU utilization mean scheduling doesn't matter since the CPU is free more often that it is needed
 - Yes in certain circumstances!
 - Scheduling matters at high utilization (bursts of heavy usage)
 - Google and Amazon estimate they lose approximately 5-10% of their customers if their response time increases by as little as 100 ms (OS:PP 2nd Ed., p. 314)
 - When do you care about scheduling at the grocery store checkout...at 6 a.m. or 5 p.m.
- Many OS scheduling concepts are applicable in other applications: web servers, network routing, etc.



Figure 1. Average CPU utilization of more than 5,000 servers during a six-month period. Servers are rarely completely idle and seldom operate near their maximum utilization, Instead operating most of the time at between 10 and 50 percent of their maximum utilization levels.

"The Case for Energy-Proportional Computing", <u>Luiz André Barroso</u>, <u>Urs Hölzle</u>, *IEEE Computer*, vol. 40 (2007).

2

Choices

- Under heavy utilization important choices must be made
 - Should you turn away some users so others experience reasonable response times?
 - If so, which users should you turn away?
 - How much benefit would additional resources have?
 - In most cloud providers, you can dynamically reprovision (i.e. spin up more servers on the fly)
 - Can you predict the degradation if the number of requests doubles?
 - Might it be worth it to switch scheduling strategies on the fly?
 - Do insights into the context and kind of requests matter?
 - Denial-of-service attack?

Terminology

- Task (job): A user request
- Workload: The mix (type) of tasks and their arrival time
 - Compute bound: Processor resources impose a bound on performance
 - I/O bound: I/O delay imposes a bound on performance
- Response Time (delay): Time from when the user submits the task until the user experiences its completion
- Throughput: Rate at which tasks are completed
- **Predictability**: Low variance in response times of repeated requests
- Scheduling overhead: The time to switch from one task to the next
- Fairness: Equality in the number and timeliness of resources allocated to a task
- Starvation: Lack of progress of a task due to resources given to another (higher-priority) task

Uniprocessors

- Let's start with a simple uniprocessor system assuming:
 - Preemptive multitasking: OS can switch thread at its discretion
 - Work-conserving: If a task is ready, the OS will not leave the processor idle (in preparation for some future event)
- Possible scheduling algorithms:
 - FIFO (FCFS = First come first serve)
 - SJF (Shortest Job First)
 - Time-sliced Round-robin

FIFO

- Under FIFO, the job that arrives first runs to completion
- Avoids overhead increasing throughput
 - Optimal since least possible overhead of context switching
- Maintains a simple queue
- Is it fair?
 - In one sense, yes.
 - But worst-case response times may result if long running job arrives before the short ones (grocery store)
- If jobs are all of equal size, then it can be optimal



6

Shortest Job First (SJF)

- Requires prior knowledge of length of task
 - Impossible?
- Uses some form of priority queue to determine next job to run (i.e. shortest duration)
- It is preemptive!
 - If a shorter job arrives during execution of another, SJF will context switch and run it
 - Thus, it is actually Shortest Remaining Job First
- Provides optimal average response time
- Provides worst-case variance in response time
 - A shorter job can always come in and "cut" in front of a waiting task (i.e. starvation)
- Can you game the SJF system if you are a long task?



Round Robin

- Execute each task for a given time quantum and then preempt
 - No more starvation
- How to choose the time quantum
 - To short, overhead goes up due to excessive context switches (also consider caching effects when switching often)
 - To long, response times suffer (see bottom graphic)
- FIFO and SJF can be thought of as special cases of RR
 - FIFO (RR with time quantum = inf.)
 - SJF (approx. RR with time quantum = epsilon)
 - Assume 0 overhead switch, set epsilon to 1 instruc.
 - Within a factor of n if n schedulable tasks
- Predictable though higher response times
 - Why?



8

School of Engineering

Avg. Resp. time = (60+25+30+35+40)/5 = 38



Round-Robin On Equal Size Tasks

- Poor effect on response time but low variability
 - Consider a server streaming multiple videos



Mixed Workloads

10

- All examples thus far have been compute bound (i.e. tasks are able to use the processor for their entire time quantum)
- Under mixed workloads (some I/O and some compute bound tasks) issues of fairness arise even in round-robin
- Consider an I/O bound process in the presence of two other compute bound tasks (compute for full 100 ms of their time quanta)
 - I/O process starts a 10 ms disk read, compute briefly (1 ms) and then blocks, yielding its time slice
 - Recall, we assume work-conserving so we won't just idle waiting for the disk to finish



Max-Min Fairness

- Idea: Give *priority* to processes that aren't using their fair share of resources
- Note: max-min is not necessarily on top of roundrobin
- Max-min: Maximize (responsiveness to) the minimum request
 - If any task needs less than its fair share, give the smallest (minimum) its full (maximum) request (i.e. schedule)
 - Split the remaining time among the N-1 other requests using the above technique (i.e. recursively)
 - If all tasks need more than an equal share, split evenly and round-robin
- Max-min Approximation: Give priority to task that has received the least processor time
- Originally used/proposed for network link utilization (a short download in the face of a long one)

Example

School of Engineering

Consider 4 programs:

- P1 wants 10% of processor's time
- P2 wants 20% of processor's time
- P3 and P4 each would want 50% of the processor's time on their own.

Fair share would be 25% each

- 1. Since P1 is minimum and wants < 25% we'll always schedule it (maximize it) when it is available in the ready list
- 2. We now have 90% of the processor we can split 3 ways (i.e. fair share is now 30%)
- 3. We recurse and give P2 it's 20% (scheduling it when it's available but P1 isn't).
- We split the remaining 70% between P3 and P4 (35% each) using round-robin as needed

MLFQ

- Multi-Level Feedback Queue
 - Implemented by most modern OSs
 - Unix, Linux, Windows (w/ some variation), Mac OSX?
 - Like round-robin but with multiple queues of different priority
- Goals: Reasonable compromise to achieve:
 - Response time, Low overhead, No-starvation, fairness, de-prioritize background tasks
 - A compromise to achieve similar results as maxmin fairness

MLFQ Rules

- Multiple queues with different priorities
 - Higher priority queues => Smaller time quantum
 - Lower priority queues => Larger time quantum
- Rules:
 - Rule 1: Higher priority always runs, preempting lower priority tasks
 - Rule 2: RR within same priority
 - Rule 3: All threads start at highest priority
 - Rule 4a: If thread uses up quantum, reduce priority (i.e. move to lower priority queue)
 - Rule 4b: If thread gives up processor, stays at same level
 - Alternative: once total quantum is taken up, demote
 - Shorter tasks finish quickly; I/O bound tasks get priority
 - Rule 5: After some time S, move threads back to highest priority
- **Key Idea**: We can't predict the length of a job so assume it is short and then demote it the longer it runs.



Priority

2

3



• Uses recent past to predict future

13

MLFQ Examples

- Example 1: A long running job
 - Starts at high priority and migrates to lower priority with longer time slices
- Example 2: A short job arrives during execution of the long running job
 - Preempts long job and may complete before it reaches Q0



School of Engineering

Figure 8.2: Long-running Job Over Time





Refer to the source of these images for a nice writeup: http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf

MLFQ Examples

- Example 3: I/O bound job and compute bound job
 - I/O bound job preempts compute-bound job
 - Any issue with this scheme?
- Example 4: Intermittent priority boosts to avoid starvation
 - Helps if a compute-bound job transitions to become interactive (I/O-bound)



Figure 8.4: A Mixed I/O-intensive and CPU-intensive Workload





Refer to the source of these images for a nice writeup: http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf 15

MLFQ Examples

- Example 5: Change Rule 4 to avoid gaming the system
 - Consider a program that "sleeps" for 1 ms after computing for 99 ms
 - Rule 4b: If thread gives up processor, stays at same level
 - New Rule 4: Once total quantum is taken up (over several context switches), demote



16



School of Engineering

Effects of caching, false sharing, etc.

MULTIPROCESSOR PERFORMANCE

Cache Coherency

18

- Most multi-core processors are shared memory systems where each processor has its own cache
- Problem: Multiple cached copies of same memory block
 - Each processor can get their own copy, change it, and perform calculations on their own different values...INCOHERENT!
- Solution: Snoopy caches...





Snoopy or Snoopy





Solving Cache Coherency

20

- If no writes, multiple copies are fine
- Two options: When a block is modified
 - Go out and update everyone else's copy
 - Invalidate all other sharers and make them come back to you to get a fresh copy
- "Snooping" caches using invalidation policy is most common
 - Caches monitor activity on the bus looking for invalidation messages
 - If another cache needs a block you have the latest version of, forward it to mem & others



SpinLocks

- Consider a spinlock held by a thread on P3 (not shown) for a "long time" while thread 1 and 2 (on P1 and P2) try to acquire the lock
- Continuous invalidation of each other reduces access to the bus for others (especially P3 when it tries to release)



Thread1

void acquire(lock* 1) { int val = BUSY; while(atomic_swap(val, l->val) == FREE);

Thread2

21

School of Engineering

I wish I

could get the bus!

False Sharing

- Thread-independent (i.e. nonshared) variables allocated on the same cache line
- Can cause a large performance degradation due to cache coherence (invalidates, etc.)



False Sharing Example





22

Is Cache Coherency = Atomicity?

23

- No, cache coherence only serializes writes and does not serialize entire read-modify-write sequences
 - Coherence simply ensures two processors don't read two different values of the same memory location
- Consider our sum example (sum = sum + local_sum;)



MULTIPROCESSOR SCHEDULING



24

Typical Multicore Organization

25

School of Engineering

• How do scheduling choices change when we have multiple processors that can be scheduled at the same time?



Scheduler Data Structure Issues

 Allow processor affinity (i.e. which processor a thread is schedule on) for threads

- Warm caches vs. cold caches

- Single task queue (or MLFQ) or one for each processor
 - Single queue suffers from
 - Locking contention
 - Cache coherence



School of Engineering

If a thread is scheduled on one core, context switched, and then scheduled again on another core, data may need to migrate. This reduces performance.



Cached copies of the MLFQ data structure must be kept coherent as processors modify it.

26

Scheduler Data Structure Solutions

- Each processor can maintain its own queue, reducing lock contention and cache coherence performance penalties
 - Threads essentially stay "pinned" to a certain processor
- Rebalancing across processor scheduling queues can be done only when it is "worth" it
 - i.e. When the benefit of being able to schedule a thread on a different processor outweighs the cost of the locking and caching penalties (both for the scheduler queue and thread data)



Separate scheduling queues avoids costly coherence. Migrate threads (e.g. T1) only when the overhead is outweighed by the rebalancing.

27

Oblivious Thread Scheduling

28

- Consider a single program written to optimize performance by breaking work into many parallel threads
- Knowing the structure of a parallel program can be crucial to scheduling those threads in such a way as to achieve optimal performance
- If the thread scheduler is oblivious to the nature of the parallel program, performance can be severely impaired
 - The next slide(s) show a few examples

Oblivious Scheduling Pitfalls

- Basic problem: Scheduler may treat all threads equally (i.e. many threads from many processes)
 - By not knowing which threads come from what processes or that thread's role in the overall program, performance may suffer
- Various parallel program architectures may exhibit poor performance if threads from the program are improperly scheduled
 - Bulk Synchronous Parallel (BSP): All threads compute, wait for others to finish computing, then exchange data for the next computation period
 - Since threads must wait for all others, delaying a single thread may force all others to wait
 - Staged (Producer/Consumer): Each thread performs one part of the work on an overall task.
 - Delaying one can mean others don't have enough useful work



Bulk Synchronous Parallel



29

More Pitfalls

- Various parallel program architectures may exhibit poor performance if improperly scheduled
 - Critical path: Sometimes certain tasks (threads) are on the critical path of finishing the overall job while others have more slack on their deadlines
 - If the critical path threads don't get scheduled the overall job performance will suffer
 - Preemption of a lock holder
 - Lock holder is context-switched thus holding off other threads from the program





30

Gang Scheduling

 Gang Scheduling attempts to schedule (all of) the threads from one program on the processors at the same time

Assume 1 Progs (PA) with 4 threads and two unrelated background threads

Pro	g. A	Bgrd. D	Bgrd. E
Proc1	Proc2	Proc3	Proc4
T1-A	T2-A	Т3-А	
T4-A		X	X
T1-A	T2-A	Т3-А	
T4-A		Х	X

Assume a BSP style program. T1-T3 can't run again until T4 does.

Proc1	Proc2	Proc3	Proc4
T1-A	T2-A	Т3-А	T4-A
T1-A	T2-A	Т3-А	T4-A
T1-A	T2-A		
		Т3-А	T4-A

Gang Scheduling may allow more progress in the same time window.

31

Law of Diminishing Returns

- If a project would take 12 hours alone, does working in a group of 2 mean it will take 6 hours?
- Likely not. Communication adds overhead.
 - And a team of 4 will almost certainly take much longer than 3 hours
- Many parallel programs do not continue to give linear speedup gains as you add more and more processors



OS:PP 2nd Ed. Fig 7.12

32

USC Viterbi 🗠

School of Engineering

33

Revisiting Gang Scheduling

- Just because we have 4 processors doesn't mean we should use 4 threads for a given program.
- **Space sharing** indicates multiple programs share the physical processors by using different subsets
- This is in contrast to **time sharing** where all processors are used for one program and then is swapped at the next time quantum
- We might achieve better throughput (not response time) for both Prog. A and Prog. B by only using 2 threads
 - Notice here we don't need to context switch!

Assume 2 Progs (PA-PB) each with 4 threads

	Prog	j. A	Prog. B	
Dure	-1	Due e2	Dread	Ducal

Proc1	Proc2	Proc3	Proc4
T1-A	T2-A	Т3-А	T4-A
Т1-В	Т2-В	ТЗ-В	Т4-В
T1-A	T2-A	Т3-А	T4-A
Т1-В	Т2-В	Т3-В	Т4-В

Time Sharing

Proc1	Proc2	Proc3	Proc4
T1-A	T2-A	Т1-В	Т2-В
T1-A	T2-A	Т1-В	Т2-В
T1-A	T2-A	Т1-В	Т2-В
T1-A	T2-A	Т1-В	Т2-В

Space Sharing



School of Engineering

ENERGY-AWARE SCHEDULING

Energy vs. Performance Tradeoffs

35

- Modern HW systems can trade performance for power consumption (i.e. energy)
 - Increase performance (rate of instruction execution) by consuming more power
 - Heterogeneous cores (some high-performance high power cores and some low-performance low power cores)
 - Powering on or off cores and I/O devices

Energy Policies & Scheduling

36

- On battery powered devices (laptops and phones) user's can often select an energy policy
 - Lower performance and greater battery life
 - Better performance and lower battery life
 - Or a blend!
- To achieve this blend the scheduler needs to be involved
 - Should I schedule this thread on the high performance, high power core?
 - Would allowing threads from this program to get all the resources for a few time slices allow the some I/O device to be powered down temporarily?

Basic Approach

37

- If the lower performance is below human perception:
 - Then lower performance and save energy
- If the lower performance is above human perception:
 - Then optimize for performance so the user doesn't notice any difference
- Long-running and background tasks
 - Try to achieve balance taking into account the available energy (i.e. battery level)



School of Engineering

REAL-TIME SCHEDULING

Real-Time Constraints

- Hard and Soft Real-time
 - Hard Real-time: Missing a deadline results in failure (i.e. no value for the computation)
 - Soft Real-time:
 Performance/usefulness
 degrades if deadlines missed
- Programs often have deadlines and scheduler must do its job trying to meet those deadlines



39

School of Engineering

Response Time (Log Scale)

Real-Time Scheduling Strategies

40

- Over-provisioning
 - Ensure the HW is more than needed keep up with the software workload
 - Ensure utilization is never too high
- Scheduling is almost always based on priority
 - Highest priority ready thread is chosen
- A more abstract scheduling strategy is Earliest Deadline First (EDF)
 - Choose the next thread to run based on the earlier deadline
- Priority donation
 - Solves priority inversion by having higher priority tasks that need a resource held by a low priority task to donate its high priority



School of Engineering

Scheduling Review 1

• OS:PP 2nd Ed. Exercise 7.4

Task	Length	Arrival Time	FIFO Completion Time	FIFO Response Time	SJF Completion Time	SJF Response Time	RR (10) Completion Time	RR (10) FIFO Response Time
0	85	0						
1	30	10						
2	35	15						
3	20	80						
4	50	85						
			Average:		Average:		Average:	

Scheduling Review 2

42

- OS:PP 2nd Ed. Exercise 7.13
 - Task A: Arrives first at time 0, and uses the CPU for 100 ms before finishing
 - Task B: Arrives shortly after A, still at time 0. Task B loops ten times; for each iteration of the loop B uses the CPU for 2ms and then it does I/O for 8ms.
 - Task C: Identical to B but arrives after B, still at time 0
 - Assume 0-time context switch, when will each task finish using:

Completion Time:	Α	В	С
FIFO			
RR (1 ms)			
RR (100 ms)			
SJF			
MLFQ (highest priority = 1 ms time slice)			

Scheduling Review 2 Answers

43

- OS:PP 2nd Ed. Exercise 7.13
 - Task A: Arrives first at time 0, and uses the CPU for 100 ms before finishing
 - Task B: Arrives shortly after A, still at time 0. Task B loops ten times; for each iteration of the loop B uses the CPU for 2ms and then it does I/O for 8ms.
 - Task C: Identical to B but arrives after B, still at time 0
 - Assume 0-time context switch, when will each task finish using:

Completion Time:	Α	В	С
FIFO	100	200	300
RR (1 ms)	140	121	122
RR (100 ms)	100	200	22
SJF (on compute)	140	100	102
MLFQ (highest priority = 1 ms time slice)	142	104	106



QUEUEING THEORY

Motivation

- Queuing theory provides some mathematical model of a scheduling system that will allow us to perform "back of the envelope" calculations:
 - Understand response time as a function of arrival rate or service (job execution) time
 - Expected queue sizes
 - Others



45

Definitions

- λ (lambda) for arrival rate (e.g. 500 jobs/second)
- μ (mu) for service rate (e.g. 1000 jobs/second)
- $S = 1/\mu =$ service time
- W (Wait time) = Time spent waiting in a queue to be serviced
- R = Response time = Total time spent in the system
 - R = W + S
- U = Utilization = Percent of time the server is busy
 - $\lambda/\mu =$ when $\lambda < \mu$
 - 1 = when $\lambda \ge \mu$
 - May not always want to maximize utilization
- X = Throughput (jobs processed per unit time)
 - Is $X = \mu$ or λ ?
 - X = λ when U < 1
 - X = μ when U = 1
- N = Number of tasks in the system
 - Q + U = Number of waiters + Number of jobs being serviced



School of Engineering

46

Little's Law

47

School of Engineering

- Stability: When $\lambda < \mu$
 - What if $\lambda \ge \mu$?
 - Delay and queue length will grow without bound
- For a stable system ($\lambda < \mu$)
 - Little's Law says: $N = X^*R$
 - Number in the System (Waiters) = Throughput * Response Time
- Since over the long-term, throughput (X) = λ

 $- N = \lambda^*(W+S) = \lambda^*(W+(1/\mu)) = \lambda^*W + U$

 If we expect 100 jobs/second & service time is 5 ms what utilization will our server run at?

- U = λ / μ = $\lambda *$ S = 100 j/s * .005s = 0.5

- If 10,000 jobs arrive per second and experience 100 ms response time, what is the average number of jobs in the system:
 - N = 10,000 * .1 = 1000
 - True, regardless of what's inside the system

Inter-Arrival Times

- Much of the performance of a system depends on the distribution of interarrival times
- Assume $\lambda < \mu$
 - Example:
 - $\lambda = 1000$ (1 job per 1 ms)
 - μ = 2000 (1 job per 0.5 ms)
- Constant inter-arrival times: If jobs arrived exactly every 1 ms, what would Q (average occupancy/length of the queue) be?
 - Q = 0 !! and R = 0.5 ms
 - So do we not need a queue at all?



Response Time & Throughput as a function of λ for CONSTANT INTER-ARRIVAL times 48

Bursty Inter-Arrival Times

- Much of the performance of a system depends on the interarrival times
- Assume $\lambda < \mu$
 - Example:
 - $\lambda = 1000 (1 \text{ job per } 1 \text{ ms})$
 - μ = 2000 (1 job per 0.5 ms)
- Bursty arrival times: But what if all 1000 jobs arrived at the t=0 sec. and then another 1000 jobs at t = 1 sec.
 - Q≈250
 - R ≈ 250 ms
- Burstiness always increases response time



Response Time & Throughput as a function of λ for <u>BURSTY INTER-ARRIVAL</u> times

49

Modeling Arrivals

- So how should we model arrivals (i.e. inter-arrival time)
- Model both inter-arrival and service times using probabilistic distributions.
- Which distribution?
 - Uniform
 - Gaussian
 - Exponential, $p(t=x) = \lambda e^{-\lambda x}$ because it is memoryless
 - Memoryless: likelihood of an event occurring is independent of how long we've already waited or what other events have already happened
 - This is just a model and not all workloads exhibit its characteristics but many do



50

Using Exponential Distributions

• Under exponential inter-arrival and service times the math says:

$$- R = \frac{S}{1-U} = \frac{\frac{1}{\mu}}{\frac{\mu-\lambda}{\mu}} = \frac{1}{\mu-\lambda}$$

- Example 1 :
 - At 20% utilization:
 - R = S/(1-0.2) = 1.25S
 - At 25% utilization
 - R = S/(1-0.25) = 1.33S
 - 5% increase in U => 8% increase in R
- Example 2
 - Difference at 90% and 95%
 utilization increases R by a factor of
 2 (i.e. 100% increase)



51

What Ifs?

52

- Currently we are using FIFO scheduling; would other policies work better?
 - For exponential service times, FIFO works as well as RR because expected service time remaining is independent of what's already there, you are better off finishing current jobs first
 - So what about non-exponential distributions for service time?
 - Many workloads for serving web pages and tasks in an OS are more bursty and exhibit so called heavy-tailed distributions
 - More long tasks and more shorter tasks thus SJF and RR performs better than FIFO
 - SJF is good, except it can greatly increase average response time at high utilization
 - Why?
- Multiple servers: single queue or multiple queues
 - If multiple queues, the response time curve depends on arrivals to that queue
 - If single queue, response time is always better (likelihood of being queued behind a large task is much less)



OVERLOAD MANAGEMENT

Overload Management

54

- What if burstiness causes a period where $\lambda > \mu$
 - If you use RR what will happen?
- Sometime to give good service to some you must reject others
- What do we do when overload occurs?
 - Drop jobs
 - Decrease service (throttle download bandwidth, disable certain features)
- Algorithms should be designed with overload in mind as many default applications will actually do MORE work under heavy loads
 - Caches under heavy load (thrashing)
 - Naïve network protocols for resending packets when they don't reach the sender (they might have been dropped for a reason!)