CS4410

Operating Systems

Lecture 8:
CPU scheduling (wrap up)
Concurrency—Understanding the problem

Rachit Agarwal

Goal of Today’s Lecture

- Finish up our scheduler
- Understand the concurrency problem

Let us design our own CPU scheduler

An ideal scheduler?

- Each thread gets an equal share of CPU
    - While ensuring that time-sensitive jobs are not blocked
- What is the "mechanism" we should use?
    - Priorities? Nah. We already saw issues.
    - Number of quantum used? Close, but can be cheated.
    - Why not directly track CPU time per thread?
- Scheduling decision
    - Among all "ready" threads
        - Choose the thread with minimum CPU time so far

An ideal scheduler?

- Scheduling decision:
    - Among all “ready” threads
        - Choose the thread with minimum CPU time so far
- Why may this work?
- I/O bound jobs: issue next file op, and wait
    - Blocked/sleeping threads don’t advance their CPU time
    - When ready, get boosted!
- Interactive jobs: respond to an input, and wait
    - Blocked/sleeping threads don’t advance their CPU time
    - When ready, get boosted!
- CPU-bound jobs: grind away all the remaining CPU cycles
    - While getting a fair allocation of CPU cycles
    - Cannot cheat!—kernel maintains CPU time for each job

An ideal scheduler?

- Scheduling decision:
  - Among all "ready" threads
    - Choose the thread with minimum CPU time so far
- But what if too many I/O bound and/or interactive jobs?
  - Starvation of CPU-bound jobs, or even priority inversion
  - How to avoid this?
- Idea 2: Introduce "target latency"
  - Period of time over which every thread should get some CPU cycles
  - Define quantum = target-latency/n
  - Every target-latency period,
    - Each thread gets at least a quantum worth of CPU time

An ideal scheduler?

- Scheduling decision:
  - Among all "ready" threads
    - If a thread has not been scheduled for target-latency time
      - Schedule it for a quantum worth of CPU time
        - Where quantum = target-latency/n
    - Else, choose the thread with minimum CPU time so far
- Problem?
  - Target latency = 20 ms, 200 threads
    - Each thread gets 0.1ms of CPU time
    - Large context switching overheads
- Idea 3: introduce a "minimum granularity"
  - Minimum time a thread must run, when scheduled

An ideal scheduler?

- Scheduling decision:
  - Among all "ready" threads
    - If a thread has not been scheduled for target-latency time
      - Schedule it for X worth of CPU time
      - Where X = maximum (quantum, min. granularity)
    - Else, choose the thread with minimum CPU time so far
- Problem?
  - Target latency = 20ms, minimum granularity = 1ms, 20,000 threads
    - Each thread gets 1ms worth of CPU time
    - But….
      - Some thread may have to wait for 20,000ms.
      - Back to being problematic for I/O and interactive jobs.

An ideal scheduler?

- We have been using priorities the wrong way all along
- We should use priorities to reflect "share" rather than preference
  - **nice** jobs: willing to give up for important jobs
  - **nice** values range from -20 to 19
  - If you are nice(r)—higher nice value—you will let important tasks run
- **Idea 3: Assign CPU cycles to threads using priorities as "weights"**
  - Each nice value is assigned a weight
    - Weight ~ $1024/(2)^{nice}$
  - Share of thread i
    - (its weight/(sum of all thread weights)) * target-latency

An ideal scheduler?

- Scheduling decision:
  - Among all "ready" threads
    - If a thread has not been scheduled for target-latency time
      - Schedule it for X worth of CPU time
      - Where X = maximum (thread's share, min. granularity)
      - Where thread's share depends on
        - thread's nice value & other threads' nice value
    - Else, choose the thread with minimum CPU time so far
- Problem?
  - Starvation for CPU-bound jobs if new I/O jobs keep arriving
  - Solution to starvation problem: FCFS queues!

An ideal scheduler?

- Scheduling decision:
  - Among all "ready" threads
    - If a thread has not been scheduled for target-latency time
      - Add it to a FCFS queue
      - Schedule the head of the queue for X worth of CPU time
      - Where X = maximum (thread's share, min. granularity)
      - Where thread's share depends on
        - thread's nice value & other threads' nice value
    - Else, choose the thread with minimum CPU time so far

- Would this work for all mix of jobs?
  - Let us see!

An ideal scheduler? Example 1

- Among all "ready" threads
  - If a thread has not been scheduled for target-latency time
    - Add it to a FCFS queue
    - Schedule the head of the queue for X worth of CPU time
    - Where X = maximum (thread's share, min. granularity)
    - Where thread's share depends on
      - thread's nice value & other threads' nice value
  - Else, choose the thread with minimum CPU time so far

- Target latency = 20ms, Minimum granularity = 1ms
- Two CPU-bound jobs (nice = 20)
  - Each thread's share = (1/2)*20 = 10ms!
  - Each thread runs for 10ms, before the other gets CPU!

An ideal scheduler? Example 2

- Among all "ready" threads
  - If a thread has not been scheduled for target-latency time
    - Add it to a FCFS queue
    - Schedule the head of the queue for X worth of CPU time
    - Where X = maximum (thread's share, min. granularity)
    - Where thread's share depends on
      - thread's nice value & other threads' nice value
  - Else, choose the thread with minimum CPU time so far

- Target latency = 20ms, Minimum granularity = 1ms
- A CPU-bound jobs (nice value = 20), and an I/O job (nice value = -19)
  - Thread shares will be: tiny (cpu-bound job), large (I/O job)
  - CPU-bound job can block I/O job for at most target-latency
  - I/O job will not block CPU-bound job—will go to sleep/block

An ideal scheduler?

- Among all "ready" threads
    - If a thread has not been scheduled for target-latency time
        - Add it to a FCFS queue
        - Schedule the head of the queue for X worth of CPU time
        - Where X = maximum (thread's share, min. granularity)
        - Where thread's share depends on
            - thread's nice value & other threads' nice value
    - Else, choose the thread with minimum CPU time so far

- Very close to today's Linux CFS scheduler!
    - The only difference is Linux does scheduling on "virtual runtimes"
        - Rather than real CPU times (implementation issue)
    - Nicer job => lower weight => virtual runtime increases more quickly
    - Less Nicer job => higher weight => virtual runtime increases less quickly

Houston,
We have a CPU scheduler!

- Designed by you!
  - Pretty close to ideal ….
  - Actually used by millions today …
- You now know CPU scheduling
- Network/Disk/… scheduling very similar

Concurrency

And

Synchronization

Concurrency and Synchronization

- Threads cooperate in multithreaded processes
  - To share resources, access shared data structures
    - e.g., threads accessing a memory cache in a web server
  - Also, to coordinate their execution
    - E.g., a disk reader thread reads a block of data and …
    - hands off the blocks to a network writer thread

Concurrency and Synchronization

- For correctness, we have to control this cooperation
  - Must assume threads interleave executions arbitrarily
  - Must assume threads execute at different speeds
    - Modern CPU schedulers are preemptive
    - Modern servers are multicore
    - CPU scheduling is not under application writer's control
- **Synchronization:** the process of coordination between multiple threads
  - Enables us to carefully restrict the interleaving of executions
  - Note: this applies also to processes, not just threads

Shared resource

- We will focus on coordinating access to shared resources
- Basic problem:
  - Two (or more) concurrent threads are accessing a shared variable
  - Both threads may read/modify/write the variable
  - The results must be deterministic
    - Multiple runs should get the same output
- **Over the next few lectures:**
  - Why is this a hard problem?
  - What are the basic mechanisms to solve this problem?
  - Applying basic mechanisms to different scenarios

Example 1: The racing threads

Two threads: Thread A and Thread B, operating on a shared variable value (initiated to 0)

```
value = value + 1;
If (value =/= -1)
      print ("Thread A wins");
```

```
value = value - 1;
If (value == -1)
      print ("Thread B wins");
```

Which thread wins?

(Suppose Thread A is scheduled at t=0)

Example 1: The racing threads

Two threads: Thread A and Thread B, operating on a shared variable value (initiated to 0)

```
value = value + 1;
If (value =/= -1)
      print ("Thread A wins");
```

```
value = value - 1;
If (value == -1)
      print ("Thread B wins");
```

Whats happening under the hood (inside the loop)?

(If each thread were the only thread running)

```
rA = 0 <- load rA, value
rA = 1 <- add rA, rA, 1
value = 1 <- store rA, value
```

```
rB = 0 <- load rB, value
rB = -1 <- sub rB, rB, 1
value = -1 <- store rB, value
```

Time

Example 1.1: The racing threads (one possible scenario)

Two threads: Thread A and Thread B, operating on a shared variable value (initiated to 0)

```
value = value + 1;
If (value =/= -1)
    print ("Thread A wins");
```

```
value = value - 1;
If (value == -1)
    print ("Thread B wins");
```

Whats happening under the hood (inside the loop)?

(If threads were running concurrently)

```
rA = 0 <- load rA, value

rA = 1 <- add rA, rA, 1

value = 1 <- store rA, value
```

```
rB = 0 <- load rB, value

rB = -1 <- sub rB, rB, 1

value = -1 <- store rB, value
```

Time

Whats value after these executions?

Example 1.2: The racing threads (another possible scenario)

Two threads: Thread A and Thread B, operating on a shared variable value (initiated to 0)

```
value = value + 1;
If (value =/= -1)
    print ("Thread A wins");
```

```
value = value - 1;
If (value == -1)
    print ("Thread B wins");
```

Whats happening under the hood (inside the loop)?

(If threads were running concurrently)

```
rA = 0 <- load rA, value

rA = 1 <- add rA, rA, 1



value = 1 <- store rA, value
```

```

rB = 0 <- load rB, value

rB = -1 <- sub rB, rB, 1
value = -1 <- store rB, value
```

Time

Whats value after these executions?

The crux of the problem

- Two concurrent threads (or processes)
  - Accessing a shared resource (account)
  - Without any coordination—with “synchronization"

- Lack of synchronization
  - Creates race conditions
  - Non-deterministic outputs, depending on thread scheduling

- In scenarios involving Shared resources + concurrent execution
  - We need mechanisms for synchronization
  - Ensure that we can reason about execution outputs
  - Ensure deterministic outputs

Recall: what resources are shared?

- Local variables are *not* shared
  - Refer to data on the stack, each thread has its own stack
  - Never pass/share a pointer to a local variable to other thread's stack

- Global variables are shared
  - Stored in the static data segment, accessible by any thread

- Dynamic objects are shared
  - Stored in the heap, shared if you can name it

Example 1: Potential solution?

Two threads: Thread A and Thread B, operating on a shared variable value (initiated to 0)

```
value = value + 1;
If (value =/= -1)
      print ("Thread A wins");
```

```
value = value - 1;
If (value == -1)
      print ("Thread B wins");
```

```
Make value "unreadable/unwritable"
value = value + 1;
Make value "readable/writable"
If (value =/= -1)
      print ("Thread A wins");
```

```
Make value "unreadable/unwritable"
value = value - 1;
Make value "readable/writable"
If (value == -1)
      print ("Thread B wins");
```

Which thread wins?

(Suppose Thread A is scheduled at t=0)

Example 2: The complicated racing threads

Two threads: Thread A and Thread B, operating on a shared variable value (initiated to 0)

```
while (value < 10)
      value = value + 1;
print ("Thread A wins");
```

```
while (value > -10)
      value = value - 1;
print ("Thread B wins");
```

Which thread wins?

(Suppose Thread A is scheduled at t=0)

Example 2: Potential solution?

Two threads: Thread A and Thread B, operating on a shared variable value (initiated to 0)

```
while (value < 10)
      value = value + 1;
print ("Thread A wins");
```

```
while (value > -10)
      value = value - 1;
print ("Thread B wins");
```

```
while (value < 10)
      Make value "unreadable/unwritable"
      value = value + 1;
      Make value "readable/writable"
print ("Thread A wins");
```

```
while (value > -10)
      Make value "unreadable/unwritable"
      value = value - 1;
      Make value "readable/writable"
print ("Thread B wins");
```

Which thread wins?

(Suppose Thread A is scheduled at t=0)

Example 2: Potential solution?

Two threads: Thread A and Thread B, operating on a shared variable value (initiated to 0)

```
while (value < 10)
      value = value + 1;
print ("Thread A wins");
```

```
while (value > -10)
      value = value - 1;
print ("Thread B wins");
```

```
Make value "unreadable/unwritable"
while (value < 10)
      value = value + 1;
print ("Thread A wins");
Make value "readable/writable"
```

```
Make value "unreadable/unwritable"
while (value > -10)
      value = value - 1;
print ("Thread B wins");
Make value "readable/writable"
```

Which thread wins?

(Suppose Thread A is scheduled at t=0)

Example 3: The real-world ATM banking example

- Suppose we want to implement a function to do the following
  - There is a bank account (shared resource)
  - Shared by you and your significant other (threads)
  - Each of you can operate independently (e.g., at different ATM)

- **Here is one template for withdraw:**

```
int withdraw (account, amount) {
        read_balance (account);
        balance = balance - amount;
        write_balance (account, balance);
        return balance;
}
```

- Suppose the initial balance is $1000
  - Both of you go to separate ATM machines, and withdraw $500
  - What happens?

Example 3: The real-world ATM banking example

Initial balance: $1000; both of you execute **withdraw (account, 500) at the same time**

```
int withdraw (account, amount) {
        balance = read_balance (account);
        balance = balance - amount;
        write_balance (account, balance);
        return balance;
}
```

```
int withdraw (account, amount) {
        balance = read_balance (account);
        balance = balance - amount;
        write_balance (account, balance);
        return balance;
}
```

Example 3: The real-world ATM banking example

Initial balance: $1000; both of you execute **withdraw (account, 500) at the same time**

```
balance = read_balance (account);
balance = balance - amount;
write_balance (account, balance);
return balance;
```

```
balance = read_balance (account);
balance = balance - amount;
write_balance (account, balance);
return balance;
```

Example 3.1: The real-world ATM banking example

Initial balance: $1000; both of you execute **withdraw (account, 500) at the same time**

```
balance = read_balance (account);
balance = balance – amount;
write_balance (account, balance);
return balance;
```

```
balance = read_balance (account);
balance = balance – amount;
write_balance (account, balance);
return balance;
```

Time

- What is the final balance?
  - 500? 1000? 0?
  - Everyone is happy!

Example 3.2: The real-world ATM banking example

Initial balance: $1000; both of you execute **withdraw (account, 500) at the same time**

```
balance = read_balance (account);
balance = balance – amount;



write_balance (account, balance);
return balance;
```

```
balance = read_balance (account);
balance = balance – amount;
write_balance (account, balance);
return balance;
```

Time

- What is the final balance?
  - 500? 1000? 0?
  - Bank goes berserk!

Example 3: Potential solution?

Initial balance: $1000; both of you execute **withdraw (account, 500) at the same time**

```
int withdraw (account, amount) {
        Freeze account;
        balance = read_balance (account);
        balance = balance – amount;
        write_balance (account, balance);
        Unfreeze account;
        return balance;
}
```

```
int withdraw (account, amount) {
        Freeze account;
        balance = read_balance (account);
        balance = balance – amount;
        write_balance (account, balance);
        Unfreeze account;
        return balance;
}
```

Why is return outside of freeze/unfreeze?

Is that still correct?

Example 4: Too-much-milk problem

You in your lovely, cozy, non-shared apartment

3:00	Look in fridge. Out of milk.
3:05	Leave for store.
3:10	Arrive at store.
3:15	Buy milk.
3:20	Arrive home. Put milk in fridge.
3:25	
3:30	

Drink milk, be strong!

Example 4: Too-much-milk problem

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

3:00	Look in fridge. Out of milk.	
3:05	Leave for store.	
3:10	Arrive at store.	Look in fridge. Out of milk.
3:15	Buy milk.	Leave for store.
3:20	Arrive home. Put milk in fridge.	Arrive at store.
3:25		Buy milk.
3:30		Arrive home. Put milk in fridge.

Too much milk!

Example 4: Too-much-milk problem

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

```
If (no Milk) {
      Buy milk;
}
```

```
If (no Milk) {
      Buy milk;
}
```

Too much milk!

Example 4: Potential solution? Attempt 1

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Attempt 1: Let us try the “freezing” idea

```
If (no Milk) {
    If (no Note) {
        Leave note;
        Buy milk;
        Remove note;
    }
}
```

```
If (no Milk) {
    If (no Note) {
        Leave note;
        Buy milk;
        Remove note;
    }
}
```

Does this work?

Example 4: Potential solution? Attempt 1

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

No!

```
If (no Milk) {


    If (no Note) {
        Leave note;
        Buy milk;
        Remove note;
    }
}
```

```
If (no Milk) {
    If (no Note) {






        Leave note;
        Buy milk;
        Remove note;
    }
}
```

Example 4: Potential solution? Attempt 2

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Attempt 2: Let us get smarter: freeze first

```
Leave note;
If (no Milk) {
      If (no Note) {
            Buy milk;
      }
}
Remove note;
```

```
Leave note;
If (no Milk) {
      If (no Note) {
            Buy milk;
      }
}
Remove note;
```

Does this work?

Example 4: Potential solution? Attempt 2

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

No!

```
Leave note;
If (no Milk) {
      If (no Note) {
            Buy milk;
      }
}
Remove note;
```

```
Leave note;
If (no Milk) {
      If (no Note) {
            Buy milk;
      }
}
Remove note;
```

Nobody ever buys milk!

Example 4: Potential solution? Attempt 3

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Attempt 3: May be different interpretations of notes

```
If (no Note) {
      If (no Milk) {
            Buy milk;
      }
      Leave note;
}
```

```
If (Note) {
      If (no Milk) {
            Buy milk;
      }
      Remove Note;
}
```

Does this work?

Example 4: Potential solution? Attempt 3

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

No! Starvation!

```
If (no Note) {
      If (no Milk) {
            Buy milk;
      }
      Leave note;
}
```

Example 4: Potential solution? Attempt 4

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Attempt 4: Perhaps two notes?

```
Leave noteA;
If (no noteB) {
    If (no Milk) {
        Buy milk;
    }
}
Remove noteA;
```

```
Leave noteB;
If (no noteA) {
    If (no Milk) {
        Buy milk;
    }
}
Remove noteB;
```

Does this work?

Example 4: Potential solution? Attempt 4

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Even worse! Lockup, deadlock, starvation!

```
Leave noteA;

If (no noteB) {
	If (no Milk) {
		Buy milk;
	}
}
Remove noteA;
```

```
Leave noteB;

If (no noteA) {
	If (no Milk) {
		Buy milk;
	}
}
Remove noteB;
```

Example 4: Potential solution? Attempt 5

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Attempt 5: What are we missing?

"If roommate is not doing something, I should do it"

"If roommate is doing something, I should not do it"

```
Leave noteA;
While (noteB) {
      Do nothing;
}
      If (no Milk) {
            Buy milk;
      }
Remove noteA;
```

```
Leave noteB;
While (no noteA) {
      Do nothing;
}
      If (no Milk) {
            Buy milk;
      }
Remove noteB;
```

Does this work?