Parallel Processing: Performance Limits & Synchronization

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
Administrivia

- Quiz 3: Thursday 7:30-9:30pm, room 50-340 (Walker)
  - Closed book, covers lectures 15-21
- Design project:
  - Part 1 (15 points) is mandatory, part 2 (15 points) is optional
- Grades: We will compute the final grade cutoffs after grading Quiz 3, and will send you e-mail on Friday night
  - We will not include part 2 to determine the grade cutoffs
  - If you are taking the makeup or have missing labs, we will project your grade on those
  - You can then decide how much effort to put on part 2 of the design project based on where you stand
- You must have completed all labs and part 1 of the design project by Thursday 3/17 to pass the course
- Please evaluate 6.S084. Your feedback matters!

May 8, 2018
The Shift to Multicore

- Since 2005, improvements in system performance mainly due to increasing cores per chip
- Why? Limited instruction-level parallelism

[Produced with CPUDB, cpudb.stanford.edu]
Multicore Processors

- If applications have a lot of parallelism, using a larger number of simpler cores is more efficient!

What is the optimal tradeoff between core cost and number of cores?
Amdahl’s Law

- Speedup = \( \frac{\text{time}_{\text{without enhancement}}}{\text{time}_{\text{with enhancement}}} \)
- Suppose an enhancement speeds up a fraction \( f \) of a task by a factor of \( S \)

\[
\text{time}_{\text{new}} = \text{time}_{\text{old}} \cdot ( (1-f) + \frac{f}{S} )
\]

\[
S_{\text{overall}} = \frac{1}{( (1-f) + \frac{f}{S} )}
\]

Corollary: Make the common case fast
Amdahl’s Law and Parallelism

• Say you write a program that can do 90% of the work in parallel, but the other 10% is sequential

• What is the maximum speedup you can get by running on a multicore machine?

\[ S_{\text{overall}} = \frac{1}{(1-f) + \frac{f}{S}} \]

\[ f = 0.9, \ S=\infty \rightarrow S_{\text{overall}} = 10 \]

What \( f \) do you need to use a 1000-core machine well?

\[ \frac{1}{(1-f)} = 1000 \rightarrow f = 0.999 \]
Thread-Level Parallelism

- Divide computation among multiple threads of execution
  - Each thread executes a different instruction stream

- Communication models:
  - Shared memory:
    - Single address space
    - Implicit communication by memory loads & stores
  - Message passing:
    - Separate address spaces
    - Explicit communication by sending and receiving messages

- Pros/cons of each model?
Synchronous Communication

**Precedence Constraints:**

\[ a \preceq b \]

“\( a \) precedes \( b \)”

- Can’t consume data before it’s produced
  \[ \text{send}_i \preceq \text{rcv}_i \]

- Producer can’t “overwrite” data before it’s consumed
  \[ \text{rcv}_i \preceq \text{send}_{i+1} \]
FIFO (First-In First-Out) Buffers

FIFO buffers relax synchronization constraints. The producer can run up to N values ahead of the consumer:

\[ \text{rcv}_i \leq \text{send}_{i+N} \]

Typically implemented as a “Ring Buffer” in shared memory:
Shared-memory FIFO Buffer: First Try

**SHARED MEMORY:**

```c
char buf[N];          /* The buffer */
int in=0, out=0;
```

**PRODUCER:**

```c
void send(char c){
    buf[in] = c;
    in = (in + 1) % N;
}
```

**CONSUMER:**

```c
char rcv(){
    char c;
    c = buf[out];
    out = (out + 1) % N;
    return c;
}
```

Not correct. Why?

Doesn’t enforce any precedence constraints (e.g., `rcv()` could be invoked prior to any `send()`

May 8, 2018
Semaphores (Dijkstra, 1962)

Programming construct for synchronization:
- New data type: semaphore, an integer ≥ 0
  ```
  semaphore s = K;  // initialize s to K
  ```
- New operations (defined on semaphores):
  - `wait(semaphore s)`
    - `wait until s > 0, then s = s - 1`
  - `signal-semaphore s)`
    - `s = s + 1 (one waiting thread may now be able to proceed)`
- Semantic guarantee: A semaphore s initialized to K enforces the precedence constraint:
  $$\text{signal}(s)_i < \text{wait}(s)_{i+K}$$
  The $i^{th}$ call to `signal(s)` must complete before the $(i+K)^{th}$ call to `wait(s)` completes
Semaphores for Precedence

Goal: Want statement A2 in thread A to complete before statement B4 in thread B begins.

Recipe:
- Declare semaphore = 0
- signal(s) at start of arrow
- wait(s) at end of arrow

May 8, 2018
Semaphores for Resource Allocation

Abstract problem:
• Pool of K resources
• Many threads, each needs resource for occasional uninterrupted period
• Must guarantee that at most K resources are in use at any time

Solution using semaphores:

In shared memory:

```c
semaphore s = K;  // K resources
```

Using resources:

```c
wait(s);  // Allocate a resource
...
// use it for a while
signal(s);  // return it to pool
```

Invariant: Semaphore value = number of resources left in pool
FIFO Buffer with Semaphores: 2\textsuperscript{nd} Try

**SHARED MEMORY:**

```c
char buf[N]; /* The buffer */
int in = 0, out = 0;
semaphore chars = 0;
```

**PRODUCER:**

```c
void send(char c) {
    buf[in] = c;
    in = (in + 1) % N;
    signal(chars);
}
```

**CONSUMER:**

```c
char rcv() {
    char c;
    wait(chars);
    c = buf[out];
    out = (out + 1) % N;
    return c;
}
```

Precedence managed by semaphore: send\textsubscript{i} < rcv\textsubscript{i}

Resource managed by semaphore: # of chars in buf

Still not correct. Why? Producer can overflow buffer. Must enforce rcv\textsubscript{i} < send\textsubscript{i+N}!
FIFO Buffer with Semaphores
Correct implementation (for single producer + single consumer)

Resources managed by semaphores: characters in FIFO, spaces in FIFO.
Works with single producer and consumer. But what about multiple producers and consumers?
Simultaneous Transactions

Suppose you and your friend visit the ATM at exactly the same time, and remove $50 from your account. What happens?

```c
void debit(int account, int amount) {
    t = balance[account];
    balance[account] = t - amount;
}
```

What is *supposed* to happen?

// assume t0 has address of balance[account]

<table>
<thead>
<tr>
<th>Thread #1</th>
<th>Thread #2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw t1, 0(t0)</code></td>
<td>...</td>
</tr>
<tr>
<td><code>sub t1, t1, a1</code></td>
<td><code>lw t1, 0(t0)</code></td>
</tr>
<tr>
<td><code>sw t1, 0(t0)</code></td>
<td><code>sub t1, t1, a1</code></td>
</tr>
<tr>
<td></td>
<td><code>sw t1, 0(t0)</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Result: You have $100, and your bank balance is $100 less.
But What If Both Calls Interleave?

We need to be careful when writing concurrent programs. In particular, when modifying shared data.

For certain code segments, called critical sections, we would like to ensure that no two executions overlap.

This constraint is called mutual exclusion.

Solution: embed critical sections in wrappers (e.g., “transactions”) that guarantee their atomicity, i.e., make them appear to be single, instantaneous operations.
Semaphores for Mutual Exclusion

```c
semaphore lock = 1;

void debit(int account, int amount) {
    wait(lock);  // Wait for exclusive access
    t = balance[account];
    balance[account] = t - amount;
    signal(lock); // Finished with lock
}
```

Lock controls access to critical section

Issue: Lock granularity
- One lock for all accounts?
- One lock per account?
- One lock for all accounts ending in 004?

Locks are just one way to implement atomic transactions
Databases → Pessimistic vs optimistic concurrency control
Multicores → Software and hardware transactional memory
Hardware → Bluespec rules

(atomic transactions)

“a precedes b or b precedes a” (i.e., they don’t overlap)
Consider multiple producer threads:

\[ \text{buf}[\text{in}] = c; \quad \text{in} = (\text{in}+1) \mod N; \]

*Problem: Producers interfere with each other*
FIFO Buffer with Semaphores
Correct multi-producer, multi-consumer implementation

**SHARED MEMORY:**

```c
char buf[N];          /* The buffer */
int in = 0, out = 0;
semaphore chars = 0, spaces = N;
semaphore lock = 1;
```

**PRODUCER:**

```c
void send(char c) {
    wait(spaces);
    wait(lock);
    buf[in] = c;
    in = (in + 1) % N;
    signal(lock);
    signal(chars);
}
```

**CONSUMER:**

```c
char rcv() {
    char c;
    wait(chars);
    wait(lock);
    c = buf[out];
    out = (out + 1) % N;
    signal(lock);
    signal(spaces);
    return c;
}
```
The Power of Semaphores

**SHARED MEMORY:**

```c
char buf[N];          /* The buffer */
int in = 0, out = 0;
semaphore chars = 0, spaces = N;
semaphore lock = 1;
```

**PRODUCER:**

```c
void send(char c) {
    wait(spaces);
    wait(lock);
    buf[in] = c;
    in = (in + 1) % N;
    signal(lock);
    signal(chars);
}
```

**CONSUMER:**

```c
char rcv() {
    char c;
    wait(chars);
    wait(lock);
    c = buf[out];
    out = (out + 1) % N;
    signal(lock);
    signal(spaces);
    return c;
}
```

A single synchronization primitive that enforces both:

- **Precedence relationships:**
  \[ send_i < rcv_i \]
  \[ rcv_i < send_{i+N} \]

- **Mutual-exclusion relationships:**
  - protect variables `in` and `out`
Semaphores are themselves shared data and implementing wait and signal operations require read/modify/write sequences that must be executed as critical sections. So how do we guarantee mutual exclusion in these particular critical sections without using semaphores?

Approaches:

- Use a special instruction (e.g., “test and set”) that performs an atomic read-modify-write. Depends on atomicity of single instruction execution. This is the most common approach.

- Implement them using system calls. Works in uniprocessors only, where the kernel is uninterruptible.

- Implement them using the atomicity of individual read and write operations, e.g., Dekker’s Algorithm (Wikipedia). Slow and rarely used.
Synchronization: The Dark Side

The naïve use of synchronization constraints can introduce its own set of problems, particularly when a thread requires access to more than one protected resource.

```java
void transfer(int account1, int account2, int amount) {
    wait(lock[account1]);
    wait(lock[account2]);
    balance[account1] = balance[account1] - amount;
    balance[account2] = balance[account2] + amount;
    signal(lock[account2]);
    signal(lock[account1]);
}
```

What can go wrong here?

Thread 1: `wait(lock[6031]);`
Thread 2: `wait(lock[6004]);`
Thread 1: `wait(lock[6004]);` // cannot complete
    // until thread 2 signals
Thread 2: `wait(lock[6031]);` // cannot complete
    // until thread 1 signals

No thread can make progress → Deadlock
Philosophers think deep thoughts, but have simple secular needs. When hungry, a group of N philosophers will sit around a table with N chopsticks interspersed between them. Food is served, and each philosopher enjoys a leisurely meal using the chopsticks on either side to eat.

They are exceedingly polite and patient, and each follows the following dining protocol:

**Philosopher’s algorithm:**
- Take (wait for) LEFT stick
- Take (wait for) RIGHT stick
- EAT until sated
- Replace both sticks

*Wait, I think I see a problem here... Shut up!!*
Deadlock!

No one can make progress because they are all waiting for an unavailable resource.

**CONDITIONS:**

1) Mutual exclusion: Only one thread can hold a resource at a given time

2) Hold-and-wait: A thread holds allocated resources while waiting for others

3) No preemption: A resource can not be removed from a thread holding it

4) Circular wait

**SOLUTIONS:**

Avoidance - or - Detection and Recovery

Cousin Tom is spared! He still doesn’t look too happy…

May 8, 2018 L22-25
Assign a unique number to each chopstick, request resources in a consistent order:

**New Algorithm:**
- Take LOW stick
- Take HIGH stick
- EAT
- Replace both sticks.

**Simple proof:**

Deadlock means that each philosopher is waiting for a resource held by some other philosopher ...

But, the philosopher holding the highest numbered chopstick can’t be waiting for any other philosopher (no hold-and-wait cycle) ...

Thus, there can be no deadlock.
Can you fix the transfer method to avoid deadlock?

```c
void transfer(int account1, int account2, int amount) {
    int a = min(account1, account2);
    int b = max(account1, account2);
    wait(lock[a]);
    wait(lock[b]);
    balance[account1] = balance[account1] - amount;
    balance[account2] = balance[account2] + amount;
    signal(lock[b]);
    signal(lock[a]);
}
```
Summary

• Communication among parallel threads or asynchronous processes requires synchronization
  – Precedence constraints: a partial ordering among operations
  – Semaphores as a mechanism for enforcing precedence constraints
  – Mutual exclusion (critical sections, atomic transactions) as a common compound precedence constraint
  – Solving Mutual Exclusion via binary semaphores
  – Synchronization serializes operations, limits parallel execution

• Many alternative synchronization mechanisms exist!

• Deadlock:
  – Consequence of undisciplined use of synchronization mechanism
  – Can be avoided in special cases, detected and corrected in others
Thank you!

Next lecture:
Putting it all together