Parallel Processing: Performance Limits & Synchronization

- The shift to multicore
- Amdahl’s Law
- Synchronization
  - For resource allocation
  - For mutual exclusion
- Deadlock

This week:
- Lab 7 due Thursday
The Shift to Multicore

- Since 2005, improvements in system performance mainly due to increasing cores per chip
- Why? Limited instruction-level parallelism
  End of voltage scaling
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 \( S \)
  \[ \text{time}_{\text{new}} = \text{time}_{\text{old}} \cdot ( (1-f) + \frac{f}{S} ) \]
  \[ S_{\text{overall}} = \frac{1}{(1-f) + \frac{f}{S}} \]

\[
\begin{align*}
\text{time}_{\text{old}} & \quad \text{(l - f)} \quad f \\
\text{time}_{\text{new}} & \quad \text{(l - f)} \quad f/S
\end{align*}
\]

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?

\[ S_{\text{overall}} = 500 \rightarrow f = 0.998 \]
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:**

- Can’t consume data before it’s produced
  - \( \text{send}_i \preceq \text{rcv}_i \)
- Producer can’t “overwrite” data before it’s consumed
  - \( \text{rcv}_i \preceq \text{send}_{i+1} \)

**Example:**

- Producer:
  - Loop: \(<xxx>\); send(c); goto loop

- Consumer:
  - Loop: \(c = \text{rcv}(); <yyy>\); goto loop

- Example scenarios:
  - \(<xxx>_1\) to \(\text{send}_1\)
  - \(<xxx>_2\) to \(\text{send}_2\)
  - \(<xxx>_3\) to \(\text{send}_3\)
  - \(\text{rcv}_1\) to \(<yyy>_1\)
  - \(\text{rcv}_2\) to \(<yyy>_2\)
  - \(\text{rcv}_3\) to \(<yyy>_3\)
RELAXES interprocess synchronization constraints. Buffering relaxes the following OVERWRITE constraint to:

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

"Ring Buffer:"

- **Read ptr**
  - \(<xxx>;\) send(c₀);
  - \(<xxx>;\) send(c₁);
  - \(<xxx>;\) send(c₂);
  - \(<xxx>;\) send(c₃);

- **Write ptr**
  - \(<xxx>;\) send(c₀);
  - \(<xxx>;\) send(c₁);
  - \(<xxx>;\) send(c₂);
  - \(<xxx>;\) send(c₃);

\( \text{rcv}; //c_0 <yyy>; \)
\( \text{rcv}; //c_1 <yyy>; \)
\( \text{rcv}; //c_2 <yyy>; \)

*Time*
Example: Bounded Buffer Problem

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

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

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

Problem: Doesn’t enforce precedence constraints (e.g. rcv() could be invoked prior to any send() )
Semaphores (Dijkstra)

Programming construct for synchronization:

- NEW DATA TYPE: *semaphore*, an integer $\geq 0$
  
  `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 process may now be able to proceed)*

- SEMANTIC GUARANTEE: A semaphore $s$ initialized to $K$ enforces the constraint:

  $\forall i \leq K : signal(s)_i \prec wait(s)_{i+K}$

  *Often you will see $P(s)$ used for $wait(s)$ and $V(s)$ used for $signal(s)$!*  
  
  $P =$ “proberen” (test) or “pakken” (grab)  
  $V =$ “verhogen” (increase)

*This is a precedence relationship: the $i^{th}$ call to $signal$ must complete before the $(i+K)^{th}$ call to $wait$ will succeed.*
Semaphores for Precedence

Goal: want statement A2 in process A to complete before statement B4 in Process B begins.

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

semaphore s = 0;

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1;</td>
<td>B1;</td>
</tr>
<tr>
<td>A2;</td>
<td>B2;</td>
</tr>
<tr>
<td>A3;</td>
<td>B3;</td>
</tr>
<tr>
<td>A4;</td>
<td>B4;</td>
</tr>
<tr>
<td>A5;</td>
<td>B5;</td>
</tr>
</tbody>
</table>
Semaphores for Resource Allocation

Abstract problem:
- POOL of K resources
- Many processes, each needs resource for occasional uninterrupted period
- MUST guarantee that at most K resources are in use at any time.

Semaphore Solution:

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
Bounded Buffer Problem
w/ Semaphores

**SHARED MEMORY:**

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

**PRODUCER:**

```c
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;
}
```

Does this work?

**PRECEDENCE** managed by semaphore: \( \text{send}_i < \text{rcv}_i \)

**RESOURCE** managed by semaphore \( \text{chars} \): # of chars in \( \text{buf} \)
Flow Control Problems

Q: What keeps PRODUCER from putting N+1 characters into the N-character buffer?

A: Nothing.

Result: OVERFLOW.

WHAT we’ve got thus far:

\[ \text{send}_i < \text{rcv}_i \]

WHAT we still need:

\[ \text{rcv}_i < \text{send}_{i+N} \]
Resources managed by semaphore: characters in FIFO, spaces in FIFO. Works with single producer, consumer. But what about multiple producers and consumers?

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

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

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

Debit(int account, int amount) {
    t = balance[account];
    balance[account] = t - amount;
}

What is supposed to happen?

Process # 1
LD(R10, balance, R0)
SUB(R0, R1, R0)
ST(R0, balance, R10)
...

Process # 2
LD(R10, balance, R0)
SUB(R0, R1, R0)
ST(R0, balance, R10)
...

NET: You have $100, and your bank balance is $100 less.
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;

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

RESOURCE managed by “lock” semaphore:
Access to critical section

ISSUES:
Granularity of lock
1 lock for whole balance database?
1 lock per account?
1 lock for all accounts ending in 004

Look up "database" on Wikipedia to learn about systems that support efficient transactions on shared data.
Consider multiple PRODUCER processes:

```
P_1
...
buf[in] = c;
in = (in+1) % N;
...
P_2
...
```

**BUG:** Producers interfere with each other.
Bounded Buffer Problem

w/ even more Semaphores

**SHARED MEMORY:**

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

**PRODUCER:**

```c
send(char c)
{
    wait(space);
    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(space);
    return c;
}
```
The Power of Semaphores

**SHARED MEMORY:**

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

**PRODUCER:**

```c
send(char c) {
    wait(space);
    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(char);
    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`
Semaphore Implementation

Semaphores are themselves shared data and implementing WAIT and SIGNAL operations will 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. We will see how to implement these instructions later in the course.

- SVC implementation, using atomicity of kernel handlers. Works in uniprocessors only, where the kernel is uninterruptible.

- Implementation using atomicity of individual read or write operations. For example, see “Dekker’s Algorithm” on Wikipedia.
The naïve use of synchronization constraints can introduce its own set of problems, particularly when a process requires access to more than one protected resource.

```c
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]);
}
```

**Synchronization: The Dark Side**

**DEADLOCK (aka “deadly embrace”)!**
Dining Philosophers

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 process can hold a resource at a given time

2) Hold-and-wait - a process holds allocated resources while waiting for others

3) No preemption - a resource can not be removed from a process holding it

4) Circular wait

SOLUTIONS: Avoidance or Detection and Recovery
One Solution

KEY: Assign a unique number to each chopstick, request resources in globally 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.
Dealing With Deadlocks

Cooperating processes:
- Establish a fixed ordering to shared resources and require all requests to be made in the prescribed order

```c
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]);
}
```

Unconstrained processes:
- O/S discovers circular wait & kills waiting process
- Transaction model
- Hard problem
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