Introduction

- Why concurrency?
- Concurrency problems
- Synchronization
- More Problems
- Lock-free synchronization
- Aura Example
What Is Concurrency?

Concurrent Execution

- With a single CPU,
  - each process runs for awhile
  - processes switch at distinct time points
  - …but…
  - switch can happen at any time
  - on any instruction boundary
- We must assume any ordering of instructions is possible
- With multiple CPUs,
  - Atomic memory operations (read & write)
  - …but…
  - Memory reads and writes are not in instruction order
Concurrent and Parallel

- **Concurrent** means multiple processes (or threads) that either
  - Run in an interleaved fashion, or
  - Run on multiple processors (or cores)
- **Parallel** means the latter: running on multiple processors (or cores)

---

Non-Reasons for Concurrency

- Multiple tasks
  - … but tasks can be interleaved in a single threaded program
    - Example: our discrete event simulations
  - I have to pause task 1 and let others proceed
    - … but you can break up task 1 into multiple code blocks and run them separately
    - … or you can use active objects to retain state
    - … or you can use co-routines (not quite a process because there’s no preemption; aka cooperative multitasking)
More Non-Reasons for Concurrency

- I need to block on I/O devices without blocking other tasks
  - … but you can use asynchronous I/O (sometimes)

Reasons for Concurrency

- **Fault-tolerance**: isolate programs so that bugs do not bring down entire system
- **Time-sharing**:
  - prevent any program from taking control of the computer system
  - allow multiple programs to run without any designed-in cooperative behavior
- **Software Architecture**
  - make programs easier to build and understand
- **Low latency/fast response**:
  - … by preempting a slow process
Concurrent Problems

```c
insert(list_node** list, item)
    node = new(list_node)
    node->value = item
    node->next = *list
    *list = node

node = new(list_node)
    node->value = item
    node->next = *list
    node->next = *list
    *list = node
    *list = node
```

Another Example

```python
def withdraw(m)
    balance = balance - m
    load r1, balance
    load r2, m
    sub r1, r2
    store balance, r1
```

```python
load r1, balance=100
load r2, m=75
sub r1=100, r2=75
load r1, balance=100
load r2, m=60
sub r1=100, r2=60
store balance, r1=40
store balance, r1=25
```

So balance == 25!
Yet Another Example

Parameter Update:
(lowpass filter)

lp_set_cutoff(hz):
\[ b=2.0-\cos(hz\pi/2/sr) \]
\[ c_2=b-\sqrt{(b^2)-1} \]
\[ c_1=1-c_2 \]

(b=2.0-\cos(hz\pi/2/sr) \\
c_2=b-\sqrt{(b^2)-1} \\
c_1=1-c_2)

(maybe the filter runs here in a third thread!)

This c2 is in a CPU register.

Atomicity and Critical Sections

- We say that a set of operations is “atomic” if no other operations can be interleaved or concurrent.
- Some machine steps are always atomic, e.g.
  - Loading a memory word to a register
  - Storing a memory word from a register
- A set of operations that must be atomic for correctness is called a “critical section”
**Critical Sections Can Be Implemented with Locks**

```
insert(list_node**list, item)
    LOCK(list_lock)
    node = new(list_node)
    node->value = item
    node->next = *list
    *list = node
    UNLOCK(list_lock)
```

**Another Example**

```
def withdraw(m)
    LOCK(account)
    balance = balance - m
    UNLOCK(account)
    call LOCK(account)
    load r1, balance=100
    load r2, m=75
    sub r1, r2
    store balance, r1
    call UNLOCK(account)
    load r1, balance=25
    call LOCK(account)
    load r2, m=60
    sub r1, r2
    store balance, r1
    call UNLOCK(account)
    load r1, balance=25
    load r2, m=60
    sub r1, r2
    store balance, r1
    call UNLOCK(account)
    So balance == -35!
```
Yet Another Example

Parameter Update: (lowpass filter)

\(\text{lp}_\text{set}\_\text{cutoff}(\text{hz}):\)

\[
\begin{align*}
&\text{LOCK}(\text{filter\_lock}) \\
b &= 2.0 - \cos(\text{hz} \times \pi^2 / \text{sr}) \\
c_2 &= b - \sqrt{(b^2) - 1} \\
c_1 &= 1 - c_2 \\
&\text{UNLOCK}(\text{filter\_lock})
\end{align*}
\]

(maybe the filter tries to run here in a third thread!)

\[
\begin{align*}
&\text{LOCK}(\text{filter\_lock}) \\
b &= 2.0 - \cos(\text{hz} \times \pi^2 / \text{sr}) \\
c_2 &= b - \sqrt{(b^2) - 1} \\
c_1 &= 1 - c_2 \\
&\text{UNLOCK}(\text{filter\_lock})
\end{align*}
\]

Synchronized Communication Is a Standard Problem

- Process1 puts tasks in a queue for Process2
- What should Process2 do when queue is empty?

```
Process1 → Queue
```
```
loop
  generate data
  queue.insert(data)
```
```
Process2
```
```
loop
  data = queue.remove()
  if data
    process data
  else
    sleep(1)
```

Adds Latency
Events and Signals Are the Standard Alternative to Polling

- Event object
  - States: signaled, nonsignaled
  - Operations: SetEvent, WaitEvent
- SetEvent: sets state of Event to signaled
- WaitEvent:
  - block until state is signaled, then \textit{atomically}:
  - [unblock caller and set state to nonsignaled]
  - \textit{Only one blocked thread is released per SetEvent}

Event/Signal Example

Process1

\begin{verbatim}
loop
  generate data
  queue.insert(data)
  SetEvent(qevt)
\end{verbatim}

Queue

Process2

\begin{verbatim}
loop
  data=queue.remove()
  if data
    process data
  else WaitEvent(qevt)
\end{verbatim}

Proof by contradiction:
assume queue non-empty and waiting forever
in order to be waiting, queue was empty
after queue was empty, it became non-empty
but after an insert, Process1 calls SetEvent
so Process2 will proceed from WaitEvent.

Note many hidden assumptions:
no other processes,
strict execution order,
queue access primitives atomic
Semaphores Are Another Approach to Many Synchronization Problems

- Similar to Event objects, but
- State is an integer
- Signal (V) increments integer (atomically)
- Wait (P) blocks until state > 0, then
  - [decrements integer, unblock caller] atomically
- If initialized to 1, LOCK = P(s), UNLOCK = V(s)
- Useful for queues, allowing \( n \) processes to share a resource, pools of \( n \) resources

Semaphore Example

Initially, \( qsem = 0 \)

```
loop
  generate data
  queue.insert(data)
  V(qsem)
```

```
loop
  P(qsem)
  data = queue.remove()
  process data
```

*Note that we still need mutual exclusion on queue access.*
Readers and Writers Problem

- A classic concurrency problem:
  - Only one process can write at a time
  - Any number of processes can read concurrently
  - Why would you want this?
- We won’t take time to present the solution
- See any OS textbook or the web
- You should recognize the problem when you see it

Fairness and Starvation

- If many threads wait on a lock, a process may never wake up – starvation
- You can wait in a FIFO queue
- You can wake up a random process
- Maybe the process waiting the longest should get the lock next – this is a fairness consideration.
- Fairness requirements can make analysis even more difficult
Deadlock Is Another Potential Problem in Concurrent Programs

LOCK(a)  LOCK(b)
LOCK(b)  LOCK(a)
work with a and b  work with a and b
UNLOCK(b)  UNLOCK(a)
UNLOCK(a)  UNLOCK(b)

OOPS!
OOPS!

Monitors Are an Attempt to Create More Intuitive, High-Level Abstractions for Concurrency

- Roughly speaking, an object that allows at most one process to execute any method is called a Monitor
- Nice abstraction: methods become atomic operations
- Java uses synchronized keyword to require object to be locked before executing the method
Monitor Example

```java
class Queue {
    synchronized void enqueue(Item *item);
    synchronized Item *dequeue();
};
```

Calling `q.enqueue(item)` effectively does this:
```
lock(q.lock);
q.enqueue(item);
unlock(q.lock);
```

Monitors have additional features to block and wake up
(what happens in `dequeue()` when queue is empty?)

Nested Monitor Calls Require Great Care

- Problem:
  - Monitor A calls method in Monitor B
  - Monitor B calls a different method in Monitor A
  - DEADLOCK!
Real-Time Issues: Priority

- Recall that within single applications, the only essential reason for concurrency is to reduce latency
- We want to preempt long-running tasks to meet deadlines
- Two popular methods:
  - Deadline Scheduling
  - Fixed-priority Scheduling

Deadline Scheduling Is Optimal, But Failure Mode Can Be Arbitrarily Bad

- Every task has a deadline
- Run the task with the nearest deadline first
- Optimal, if all deadlines can be met
- But it could force you to miss all deadlines
- Another problem: what’s a deadline?
  - Maybe easy when controlling hardware
  - For audio computation, deadline is when the output buffer runs out of samples
  - Difficult to say when controlling music processes
- Effectively, our class project schedulers are deadline schedulers because they sort events by their ideal execution times and run them in that order.
Fixed Priority Is Commonly Available and Very Usable

- Each process has a fixed priority
- Run the highest priority process that is ready to run
- Often implemented in OS’s
- Often used for periodic tasks of various periods
  - *If the tasks are schedulable*
  - In this case, called *rate-monotonic* scheduling
- Fairly easy mapping to music tasks:
  - Audio computation gets highest priority
  - (MIDI) control gets medium priority
  - Graphical user interface gets low priority

Priority Inversion Can Lead to Disasters

- Static priority scheduling and synchronization primitives can have catastrophic interactions

```
low priority  lock(L)  unlock(L)
  med priority
  high priority  lock(L)  unlock(L)
                  Priority Inversion
```
Solving the Priority Inversion Problem

- **Priority Ceiling**: when you acquire a lock, raise your priority to the highest priority of any other process that might acquire the lock.
- **Priority Inheritance**: make the priority of the lock holder greater than or equal to the priority of any process waiting on the lock.
- **Probably cannot depend on OS solving this problem for you unless you control the OS**

Priority Inversion Solved

- **low priority**
- **med priority**
- **high priority**

- **lock(L)**
- **unlock(L)**
- **unlock(L)**

Priority raised to ceiling or inherited from high priority thread.
Lock-Free Synchronization

- Priority inversion problem can make available synchronization primitives unusable for (reliable) real-time applications
- Alternative: synchronization without locks
- Simplest example: Atomic memory writes
  - you can share a 32-bit value and assume reads/writes are atomic
  - Writer can update value asynchronously
  - Reader always gets an (almost) up-to-date value

Lock-Free Queue

Head

Tail
Single CPU, Single-Reader, Single-Writer Queue

```
hd = 0
tl = 0
q = array(N)
def insert(x):
    if tl < hd + N
        q[tl%N] = x
        tl = tl + 1
def remove():
    if hd < tl
        var x = q[hd%N]
        hd = hd + 1
        return x
    else
        return EMPTY
```

- Note that the order of instructions is critical
- Must store value before incrementing tl
- Must retrieve value before incrementing hd
- Compilers may cause problems: see “volatile” attribute in C compiler
- There are versions without “%” operation, e.g. (hd & mask).
- There are versions without unbounded hd and tl (otherwise this approach would be pretty useless)

Why did we specify “Single CPU” for the Queue Example?

- Multiprocessors rely on multi-level cache
- What happens when there are multiple reads and writes to the same address?
- Modern systems increasingly allow reordering of memory reads and writes(!)
What Can Go Wrong?

```
hd = 0
tl = 0
q = array(N)
def insert(x):
    if tl < hd + N
        q[tl%N] = x
        tl = tl + 1
    def remove():
        if hd < tl
            var x = q[hd%N]
            hd = hd + 1
            return x
        else
            return EMPTY
```

Out of order writes cause problem:
```
store
read
read (the wrong value!)
store
```

This used to be only :-) a problem of preventing the optimizing compiler from reordering assignments, but now write reordering happens in hardware.

A Multiple-CPU, Single Reader, Single Writer FIFO Queue

- Communication through “handshaking”:
  ```
  Process 1:        Process 2:
  while true:       while true:
      if not flag    if flag
      flag = true    flag = false
  Processes synchronize in setting flag to true/false. Depends only on atomic memory reads/writes.
  ```

- Slight change: send non-zero value:
  ```
  var buf = 0
  def send(x):
      while true
          while buf != 0:
              nil
          buf = x
  def receive():
      while true
          var r = buf
          if r != 0
              buf = 0
          return r
  ```
Light Pipe Algorithm -
Alexander Dokumentov

- Expand buf to be a circular buffer:

```
var buf = array(N)
initialize buf to zero
var i = 0
var j = 0
def send(x):
    while buf[i] != 0:
        nil
        buf[i] = x
        i = (i + 1) % N
    nil
    buf[i] = x
    i = (i + 1) % N
def receive():
    while buf[j] == 0:
        nil
    var result = buf[j]
    Buf[j] = 0
    j = (j + 1) % N
    return result
```

Light Pipe Algorithm (2)

- What about zero values?
- Encode M words with zeros as M+1 words:

Reference: http://www.ddj.com/dept/cpp/189401457
Other Lock-Free Algorithms

- Some based on CAS (Compare-and-Swap)
  ```cpp
def cas(a, e, n):
    atomically:
    if (*a == e):
      *a = n;
      return True;
    else:
      return False;
```

- Examples of Lock-Free Algs:
  - FIFO queue
  - Freelist

  “The difficulty of achieving lock-free 64-bit-clean implementations of such mundane data structures strongly suggests that improved hardware support is necessary before practical lock-free data structures will be widely available.”


Memory Consistency and Future Processors

- Memory Barrier Instruction and WriteMB
  - The MB instruction can be used to maintain program order from any memory operations before the MB to any memory operations after the MB.

Blocking vs. Polling

- Lock-free synchronization does not allow processes to block
- Standard solution is polling
  - Wake up every 1ms or so,
  - Do whatever work there is to be done
  - Go to sleep (here’s where blocking takes place) for 1ms or so

Periodically “Waking Up”

- Use an OS call to sleep
- Use an OS blocking call with a timeout
- Block waiting for audio input (wake up every 32 or 64 samples)
- Use a timer facility like Window MM system timer that calls a function periodically
Is Polling Bad?

- Waste of CPU time when nothing to do.
  - But CPU load can be low: 1 to 5%
  - In dedicated systems, there’s no cost (well maybe power)
- Context switches are expensive
  - But if there’s work to do, you’re going to context switch anyway
- Synchronization primitives are expensive too
- Latency: code doesn’t run immediately after data available
  - But if polling frequency is high enough, latency is negligible
  - Real time systems care about being fast enough, not being as fast as possible.
- Polling is more efficient as load increases, so polling can actually be better from a real-time perspective (real time systems care about the worst case, not the average case).

Example: Aura Architecture

- Goal 1: General platform for interactive multimedia
- Goal 2: Open-ended, extensible for video, graphics, networking, software systems.
- Based on Real-Time Distributed Object System
- Objects have globally-unique 96-bit names
- Asynchronous messages
- Location independent
Communication with Aura

- Remote Method Invocation
  - `send_set_hz_to(osc, 440.0)`
  - Automatically generated macros to send messages
  - Receiver is indicated by globally unique ID
- Location Transparency
  - Object in same thread – synchronous call
  - Object in same address space – msg queue
  - Object on remote machine – TCP/IP to msg queue

Messages and Location Transparency

![Diagram showing communication between machine 1 and machine 2 with objects A, B, C, Obj1, Obj2, Obj3, Obj4, and Obj5.]
Aura Details

- Each Zone (thread + memory + scheduler):
  - Memory pool and real-time allocator
  - Calendar Queue-based scheduler
  - Time (seconds) based on audio sample count
- Pre-processor generates:
  - RPC message handlers
  - Stubs to pack parameters into msgs and send
  - Macros to make them easy to call
- Structure by latency, not function

Message Passing Details
Zone Processing Loop

- Every zone runs periodically
- Messages are blocks of memory:
  - [bytecount, timestamp, object-ID, method, arglist]
- Poll:
  - Dispatch any scheduled messages
  - Check each incoming queue for messages
    - Either dispatch immediately (no copy), or
    - Allocate memory, copy, and schedule future msg
  - Actions can send and schedule new messages
- No blocking except:
  - Audio thread does blocking I/O (32 samples = 0.7ms)
  - Midi thread sleeps 1ms when nothing to do
  - Graphics thread run by GUI, uses periodic callback

Aura Objects
Aura ID

- 96-bit globally unique identifier (48 low-order bits of two 64-bit words)

Sending a Message

```c
if space(msg.dest) == my_space_id
    if zone(msg.dest) == my_zone_id
        if msg.timestamp >= NOW
            obj = zone[my_zone_id].lookup(msg.dest)
            obj->msg_handler(msg)
        else
            zone[my_zone_id].schedule(
                msg.timestamp, msg)
    else
        zone[my_zone_id].queue[zone(msg.dest)].enqueue(msg)
else
    space_proxy[space(msg.dest)]->send(msg)
```
Summary

- Concurrency: good reasons and bad reasons
- In real-time systems, preemption->low-latency
- Atomic actions and Critical sections
- Synchronization primitives:
  - locks, events, semaphores, monitors
- The dark side:
  - Starvation, Deadlock, Priority Inversion
- Lock-free structures
- Polling vs Blocking
- Aura