Thread Parallelism

Shared Memory Patterns and Thread Parallelism Paradigms
Topics

- Synchronization Problems
  - Producer-Consumer Problem
  - Readers-Writers Problem
- Semaphores and their Implementation
- Data Races
- Shared Memory Patterns
- Parallelizing Computations
- Partitioning and Problem Decomposition
- Loop Parallelism
Producer-Consumer Problem

• Problem description
  – A producer: in an infinite loop and produce one item per iteration into the buffer
  – A consumer: in an infinite loop and consumes one item per iteration from the buffer
  – Buffer size: can only hold at most N items
  – Need to make sure that
    – Producer does not try to add data into the buffer when it is full
    – Consumer does not try to remove data from an empty buffer
int counter; //initialize to 0
producer
repeat
1 read the counter value
2 if(Counter < MAX_COUNT) {
3 increment the counter;
4 update the counter with the incremented value;
5 Store a value into the buffer
6 //ERROR if buffer full
} else{
7 wait
}
Until YY says stop

consumer
repeat
1 read the counter value;
2 if(Counter > 0) {
3 decrement counter;
4 update the counter with the incremented value;
5 consume a value from buffer
6 //ERROR if buffer empty
} else{
7 wait
}
Until YY says stop
Producer-Consumer Problem

```
int counter; //initialize to 0
Mutex m;

// Producer
repeat
1 Mutex_Lock(&m);
2 read the counter value
3 if(Counter < MAX_COUNT) {
4     increment the counter;
5     update the counter with the incremented value;
6     Store a value into the buffer
7     //ERROR if buffer full
8     Mutex_Unlock(&m)
} else{
9     wait
10 }_until YY says stop

// consumer
repeat
1 Mutex_lock(&m);
2 read the counter value;
3 if(Counter > 0) {
4     decrement counter;
5     update the counter with the incremented value;
6     consume a value from buffer
7     //ERROR if buffer empty
8     Mutex_Unlock(&m)
} else{
9     wait
10 }_Until YY says stop
```
Producer-Consumer Problem

```c
int counter; //initialize to 0
Mutex m;

// Producer
repeat
1   Mutex_Lock(&m);
2   read the counter value
3   if(Counter < MAX_COUNT) {
4       increment the counter;
5       update the counter with the incremented value;
6       Store a value into the buffer
7       //ERROR if buffer full
8       Mutex_Unlock(&m)
} else{
9       Mutex_Unlock(&m)
10      wait
} Until YY says stop

// consumer
repeat
1   Mutex_lock(&m);
2   read the counter value;
3   if(Counter > 0) {
4       decrement counter;
5       update the counter with the incremented value;
6       consume a value from buffer
7       //ERROR if buffer empty
8       Mutex_Unlock(&m)
} else{
9       Mutex_Unlock(&m)
10      wait
} Until YY says stop
```
Semaphores

A synchronization variable that takes on positive integer values

Two operations:

- P(semaphore): an atomic operation that waits for semaphore to become greater than zero, then decrements by 1 (Dutch: proberen)
- V(semaphore): an atomic operation that increments semaphore by 1 (Dutch: verhogen)
Producer-Consumer Problem

Semaphore Full = 0
Semaphore Empty = BUFFER_SIZE
Mutex m; // Equivalent to Semaphore m = 1

// Producer
repeat
1. P(&empty)
2. Mutex_lock(&m);
3. Enqueue new item in buffer
4. Mutex_Unlock(&m);
5. V(&full);
Until YY says stop

// Consumer
repeat
1. P(&full)
2. Mutex_lock(&m);
3. Deque item from buffer
4. Mutex_Unlock(&m);
5. V(&empty);
Until YY says stop
Readers-Writers Problem

A Shared Database
• Two classes of users:
  • Readers – never modify database
  • Writers – read and modify database
• Is using a single lock on the whole database sufficient?
  • Like to have many readers at the same time
  • Only one writer at a time
Readers-Writers Problem

0 Deals with situations in which many threads much access the same shared memory at one time
  0 No thread may access the shared object for reading or writing while another thread is writing to it
  0 Concurrent reads are allowed

0 First Readers-Writers problem: No reader shall be kept waiting if the shared object is currently open for reading

0 Second Readers-Writers problem: No writer, once added to the queue, shall be kept waiting longer than absolutely necessary
Readers-Writers Problem

Basic structure of a solution:

Reader()
  Wait until no writers
  Access database
  Check out – wake up a waiting writer

Writer()
  Wait until no active readers or writers
  Access database
  Check out – wake up waiting readers or writer

State variables (Protected by a lock called “lock”):
  int AR: Number of active readers; initially = 0
  int WR: Number of waiting readers; initially = 0
  int AW: Number of active writers; initially = 0
  int WW: Number of waiting writers; initially = 0
  Condition okToRead = NIL
  Condition okToWrite = NIL
Readers-Writers Problem

Reader() {
    // First check self into system
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++;
        // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    // Perform actual read-only access
    AccessDatabase(ReadOnly);
    // Now, check out of system
    lock.Acquire();
    AR--; // No longer active
    if (AR == 0 && WW > 0) // No other active readers
        okToWrite.signal(); // Wake up one writer
    lock.Release();
}
Readers-Writers Problem

Writer() {
    // First check self into system
    lock.Acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++;
        // No. Active users exist
        okToWrite.wait(&lock);// Sleep on cond var
        WW--;
    }
    AW++;
    // Now we are active!
    lock.release();

    // Perform actual read/write access
    AccessDatabase(ReadWrite);

    // Now, check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){ // No longer active
        okToWrite.signal(); // Give priority to writers
        okToWrite.signal(); // Wake up one writer
    } else if (WR > 0) { // Otherwise, wake reader
        okToRead.broadcast(); // Wake all readers
    }
    lock.Release();
}
Implementation of Semaphores

- No existing hardware implements semaphores directly.
- Semaphores are built up in software using some lower-level synchronization primitive provided by hardware.
- Uniprocessor solution: Disable interrupts.

```c
typedef struct {
    int count;
    queue q; /* queue of threads waiting on this semaphore */
} Semaphore;
```
void P(Semaphore s) {
    Disable interrupts;
    if (s->count > 0) {
        s->count -= 1;
        Enable interrupts;
        return;
    }
    Add(s->q, current thread);
    sleep(); // re-dispatch
    Enable interrupts;
}

void V(Semaphore s) {
    Disable interrupts;
    if (isEmpty(s->q)) {
        s->count += 1;
    } else {
        thread = RemoveFirst(s->q);
        wakeup(thread); /* put thread on the ready queue */
    }
    Enable interrupts;
}
Implementation of Semaphores

0 Multiprocessor Solution:
   0 Can’t turn off all other processors
   0 Can’t just turn off interrupts to get low-level mutual exclusion

0 Most CISC Machines provide some sort of atomic read-modify-write instruction
   0 test&set
   0 swap
   0 compare&swap
Implementation of Semaphores

- Modern RISC machines do not provide read-modify-write instructions
- Instead they provide a weaker mechanism that does not guarantee atomicity but detects interference
  - *load-linked* instruction (ldl): Loads a word from memory and sets a per-processor flag associated with that word (usually stored in the cache)
  - *store-conditionally* instruction (stc): Stores a word iff the processor’s flag for the word is still set; indicates success or failure.
Implementation of Semaphores

Atomic Read-Modify-Write Example in MIPS

```mips
atomic_inc:
    ll $t0, 0($a0)       # load linked
    addiu $t1, $t0, 1   # increment
    sc $t1, 0($a0)      # store cond'l
    beqz $t1, atomic_inc
                      # loop if failed
```
Different Implementations for Mutual Exclusion

Using ldl/stc

```c
int lock;
...
while (ldl(&lock) != 0 || !stc(&lock, 1));
...
critical section
...
lock = 0;
```

Using Test And Set

```c
int lock;
...
while (TAS(&lock, 1) != 0);
...
critical section
...
lock = 0;
```
Using ldl/stc to Implement Semaphores

typedef struct {
    int lock;  /*Initially 0*/
    int count;
    queue q; /* queue of threads waiting on this semaphore */
} Semaphore;
Using ldl/stc to Implement Semaphores

```c
void P(Semaphore s) {
    Disable interrupts;
    while (ldl(s->lock) != 0 || !stc(s->lock, 1));
    if (s->count > 0) {
        s->count -= 1;
        s->lock = 0;
    }
    Enable interrupts;
    Add(s->q, current thread);
    s->lock = 0;
    sleep(); /* re-dispatch */
    Enable interrupts;
}

void V(Semaphore s) {
    Disable interrupts;
    while (ldl(s->lock) != 0 || !stc(s->lock, 1));
    if (isEmpty(s->q)) {
        s->count += 1;
    } else {
        thread = RemoveFirst(s->q);
        wakeup(thread); /* put thread on the ready queue */
    }
    s->lock = 1;
    Enable interrupts;
}
```
What is a Data Race?

0 Two **concurrent** accesses to a memory location at least one of which is a write.

0 Example: Data race between a read and a write

```csharp
int x = 1;
Parallel.Invoke(
    () => { x = 2; },
    () => { System.Console.WriteLine(x); }
);
```

0 Outcome nondeterministic or worse

0 may print 1 or 2, or arbitrarily bad things on a relaxed memory model
Data Races and Happens-Before

Example of a data race with two writes:

```csharp
int x = 1;
Parallel.Invoke(
    () => { x = 2; },
    () => { x = 3; }
);
System.Console.WriteLine(x);
```

We visualize the ordering of memory accesses with a happens-before graph:

There is no path between (write 2 to x) and (write 3 to x), thus they are concurrent, thus they create a data race

(note: the read is not in a data race)
Quiz: Where are the data races?

Parallel.For(1,2, i => {
    x = a[i];
});

Parallel.For(1,2, i => {
    a[i] = x;
});

Parallel.For(1,2, i => {
    a[i] = a[i+1];
});
Quiz: Where are the data races?

Parallel.For(1, 2, i => {
    x = a[i];
});

- Reads a[0]
- Writes x
- Race between two writes.

Parallel.For(1, 2, i => {
    a[i] = x;
});

- Reads x
- Writes a[0]
- Race between a read and a write.

Parallel.For(1, 2, i => {
    a[i] = a[i + 1];
});

- Reads a[2]
- Writes a[1]
- No Race between two reads.

Race between two writes.

Race between a read and a write.
Data Races can be hard to spot

Parallel.For(0, 10000,
    i => {a[i] = new Foo();})

Code looks fine... at first.
Data Races can be hard to spot

Parallel.For(0, 10000,
    i => {a[i] = new Foo();})

Problem: we have to follow calls... even if they look harmless at first (like a constructor).

class Foo {
    private static int counter;
    private int unique_id;
    public Foo()
    {
        unique_id = counter++;
    }
}

Data Race on static field!
Avoiding Data Races

The three most frequent ways to avoid data races on a variable:

- Make it **isolated**
  - Variable is only ever accessed by one task
- Make it **immutable**
  - Variable is only ever read
- Make it **synchronized**
  - Use a lock to arbitrate concurrent accesses
Programming with Shared Memory

0 Keep abstraction level HIGH
0 Temptation: ad-hoc parallelization
   0 Add tasks or parallel loops all over the code
   0 Discover data races/deadlocks, fix them one-by-one
0 Problem (depending on the programmer):
   0 Complexity adds up quickly
   0 Easy to get cornered by deadlocks, atomicity violations, data races
   0 These bugs are often hard to expose
Programming with Shared Memory

- Use well-understood, simple high-level patterns

Architectural Patterns
- Localize shared state
  - Producer-Consumer
  - Pipeline
  - Worklist

Replication Patterns
- Make copies of shared state
  - Immutable Data
  - Double Buffering
Producer-Consumer Pattern

- Also called the Bounded Buffer problem

- One or more producers add items to the buffer

- One or more consumers remove items from the buffer
Producer-Consumer Pattern

1. Item is local to Producer before insertion into buffer
2. Item is local to Consumer after removal from buffer
3. What about buffer?
   - Buffer is thread-safe
   - Blocks when full/empty
Pipeline Pattern

- Generalization of Producer-Consumer
  - One or more workers per stage
  - First stage = Producer
  - Last stage = Consumer
  - Middle stages consume and produce
Worklist Pattern

- Worklist contains items to process
  - Workers grab one item at a time
  - Workers may add items back to worklist
  - No data races: items are local to workers
Immutability

- Remember: concurrent reads do not conflict
- Idea: never write to shared data
  - All shared data is immutable (read only)
  - To modify data, must make a fresh copy first
  - Copy-On-Write
Parallelizing Computations

1. **Partitioning**
   - **Decomposition** of computation in tasks
   - **Assignment** of tasks to processes
   - **Orchestration** of data access, communication, synchronization
   - **Mapping** processes to processors

Sequential computation → Tasks → Processes → Parallel program → Processors
Partitioning

0 Identify concurrency and decide at what level to exploit it
0 Break up computations into tasks to be divided among processes
   0 Tasks may become available dynamically
   0 Number of tasks may vary with time
0 Enough tasks to keep processors busy
0 Decomposition independent of architecture or programming model
0 Structured approaches usually work well
  0 Remember: Shared memory design patterns
An Example: Decomposition
An Example: Decomposition

Task decomposition
• Independent coarse-grained computation
• Inherent to the algorithm
• Sequence of statements (instructions) that operate together as a group
• Corresponds to some logical part of program
An Example: Decomposition

Task decomposition
- Parallelism in the application

Data decomposition
- Same computation is applied to small data chunks derived from a large data set
An Example: Decomposition

Task decomposition
• Parallelism in the application

Data decomposition
• Same computation many data

Pipeline decomposition
• Data assembly lines
• Producer-consumer chains
• Usually observed in case of regular, one-way, mostly stable data flow
Finding Concurrency Design Space

- Programs often naturally decompose into tasks
- Two common decompositions:
  - Function calls
  - Distinct loop iterations
- Dependence Analysis: Given two tasks, how to determine if they can run in parallel?
Data Dependence

Assuming statements S1 and S2, S2 is data-dependent on S1 if:

\[ [I(S1) \cap O(S2)] \cup [O(S1) \cap I(S2)] \cup [O(S1) \cap O(S2)] \neq \emptyset \]

Where,

- I(Si) is the set of memory locations read by Si, and
- O(Sj) is the set of memory locations written by Sj

and there is a feasible runtime execution path from S1 to S2

Called Bernstein Condition
Types of Data Dependence

0 True dependence
$O(S1) \cap I(S2), S1 \rightarrow S2$ and $S1$ writes something read by $S2$

0 Anti-dependence
$I(S1) \cap O(S2)$, mirror relationship of true dependence

0 Output dependence
$O(S1) \cap O(S2)$, $S1 \rightarrow S2$ and both write the same memory location
Control Dependence

- There is a control dependence between two statements S1 and S2 if
  - S1 could be possibly executed before S2
  - The outcome of S1 execution will determine whether S2 will be executed

A: while(node){
B: node = node->next;
C: res = work(node);
D: write(res);
}
Loop Parallelism Patterns

0 Many programs are expressed using iterative constructs
0 Loops are a major part of most programs
0 Loop parallelism especially useful when code cannot be massively restructured
0 Different techniques:
   0 DOALL
   0 DOACROSS
   0 DSWP (Decoupled Software Pipelining)
Consider the following loop

```c
int arr[10], op[10];
int i = 0;
while(i<10) {
    op[i] = arr[i]*arr[i];  // (A)
    i++;
}
```
With Inter-Iteration Dependences?

Consider the following loop

A: while(node){
B:   node = node->next;
C:   res = work(node);
D:   write(res);
}

Here, work may modify list
Consider the following loop

A: while(node){
B: node = node->next;
C: res = work(node);
D: write(res);
}

Here, work may modify list

Communication latency = 1 cycle/iteration
Decoupled Software Pipelining (DSWP)

Consider the following loop

A: while(node){
B:   node = node->next;
C:   res = work(node);
D:   write(res);
}

Here, work may modify list

Communication latency = 1 cycle/iteration