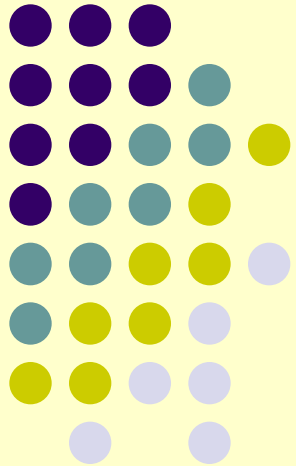
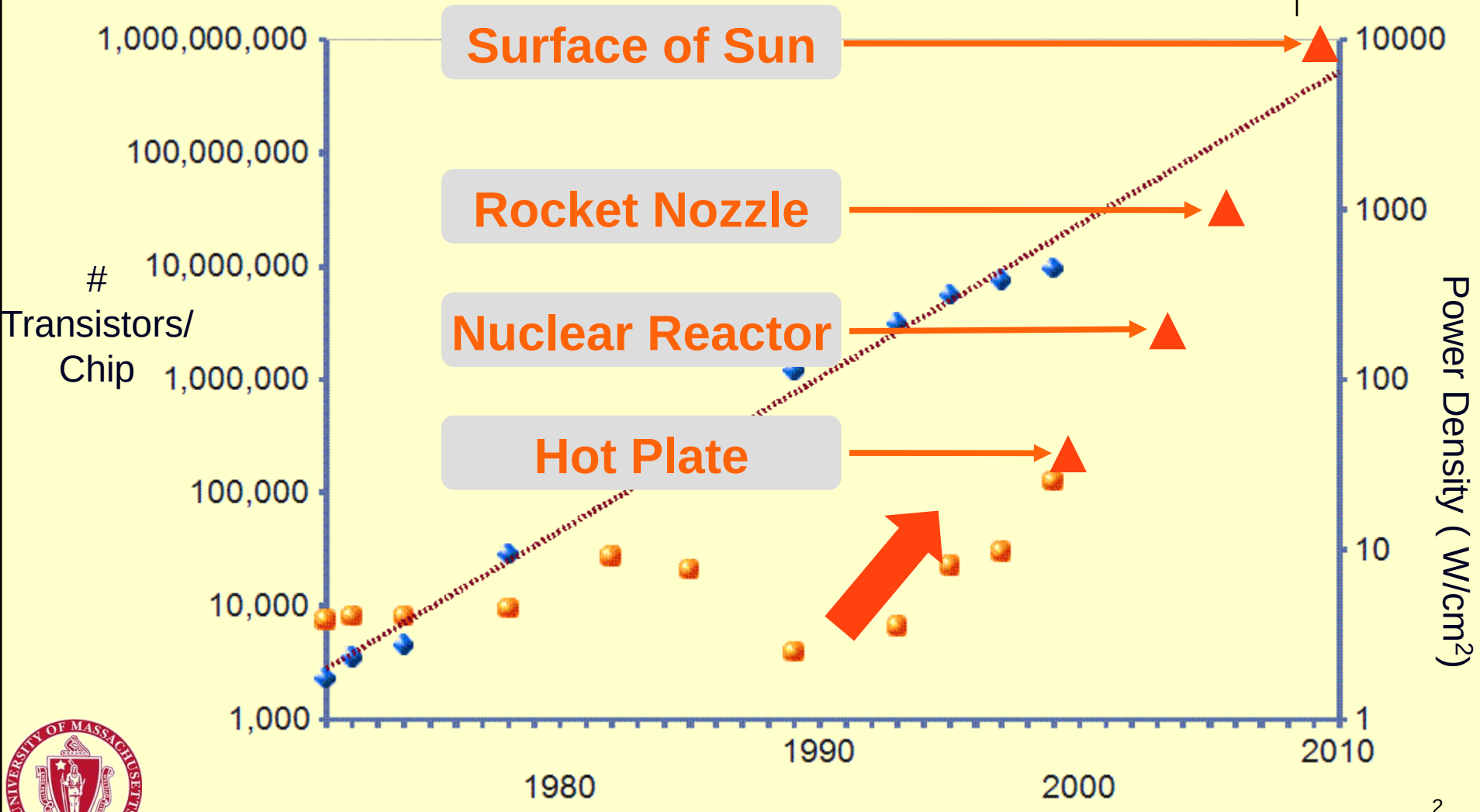
Concurrent Programming

Yannis Smaragdakis
University of Massachusetts, Amherst

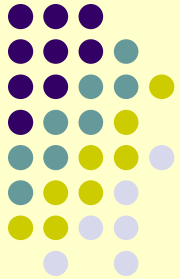


Obligatory Moore's Law Slide

- Number of transistors doubles every 2 years

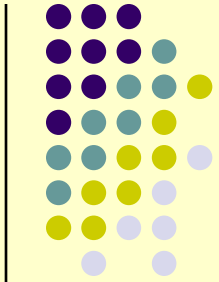


You Have Heard of Multi-Cores



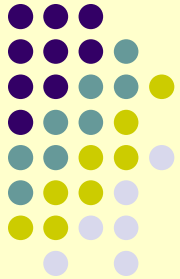
- You have probably also heard that parallel programming is hard
- Many ways to do parallel programming
  - message passing
  - multithreading
  - …
- Suited for different kinds of parallelism
  - e.g., shared memory vs. distributed memory

This Talk: Only About Multithreading



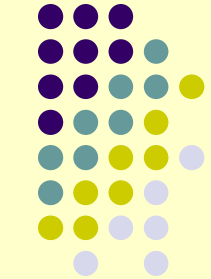
- Most explicit parallel programming model, unlikely to be completely superseded for general purpose shared memory parallelism

Quick Review of Multithreading



- A thread is a single sequential flow of control
  - a process can have many threads and a single address space
  - threads share memory and, hence, need to cooperate to produce correct results
  - shared memory: heap, global variables
  - thread-private memory: stack, other only if explicitly designated

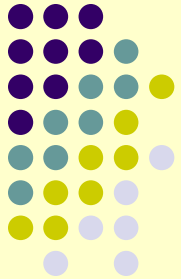
Programming Models for Multithreading



- Most thread programming nowadays is "monitor style" programming
  - introduced in 70s, subsequently dominated all alternatives as a general-purpose facility
    - alternatives: semaphores, low-level atomic operations, etc.
  - universal mechanism: can do anything

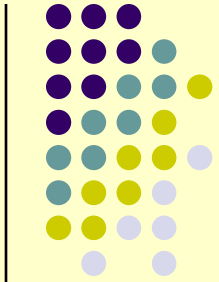
Outline

1. Monitor style programming: traditional multithreading
   - if you already know mutexes, bear with me for 30 mins and you'll see new stuff
2. Complexities of the programming model
3. Transactional memory: an emerging MT programming model
   - Complexities
   - Implementation



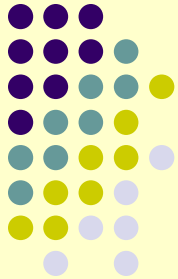
MONITOR STYLE PROGRAMMING

Monitor Style Programming



- Two components:
  - locks/mutexes (lock)
  - condition variables (wait, signal, broadcast)
- Mapping of abstract to concrete:
  - Java: every object is a mutex and a condition var
    - lock -> synchronized
    - wait -> Object.wait
    - signal -> Object.notify
    - broadcast -> Object.notifyAll

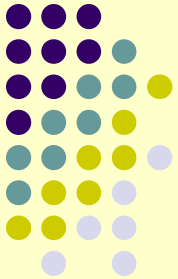
Other Incarnations



- PThreads (C, C++, etc.): can declare explicit mutex and condition var objects
  - pthread_mutex_t, pthread_cond_t
  - operations:
    - lock ->

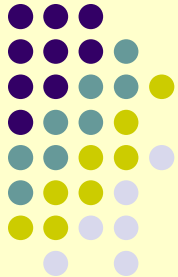
      pthread_mutex_lock … pthread_mutex_unlock
    - wait, signal, broadcast -> pthread_cond_{wait,signal,broadcast}

For Our Discussion



- Default Java distorts and handicaps the model a bit. We will use an imaginary language with the full power (also supported in Java through library classes)
  - Java but with condition variables declared explicitly as objects of type `Cond`
- Thread creation is an uninteresting detail
  - in Java threads are instances of class `Thread`, they begin execution when the `start` method is called

Mutexes



- Mutexes are used to control access to shared data
  - only one thread can execute inside a `synchronized` clause
  - other threads who try to enter synchronized code, are blocked until the mutex is unlocked
- As we said, in Java, every object is a (or “has an associated”) mutex!

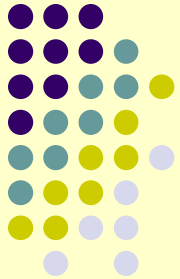
Mutexes

```
class List {
  public synchronized int insert(int i)
  { [BODY] }
}
```

- same as

```
class List {
  public int insert(int i)
  { synchronized(this) [BODY] }
}
```

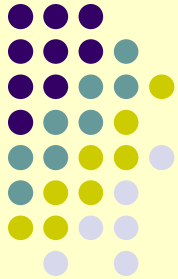
Mutexes Prototypical Example



```
class Account {
  int balance = 0;
  public synchronized int withdraw(int amt) {
   if (balance >= amt) {
     balance -= amt;
     return amt;
   } else return 0;
  }
  public synchronized void deposit(int i) {
   balance += i;
  }
}
```

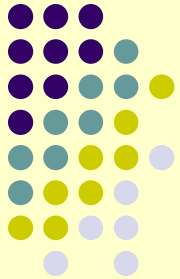
- what errors are prevented?
- we'll return to this example

Using Mutexes: Deadlocks



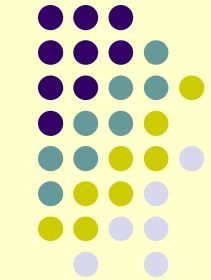
- Example deadlock:
  - A locks M1, B locks M2, A blocks on M2, B blocks on M1
- Techniques for avoiding deadlocks:
  - Fine grained locking (but then greater risk of races)
  - Two-phase locking: acquire up front all the locks you'll need, release all if you fail to acquire any one
  - Order locks and acquire them in order (e.g., all threads first acquire M1, then M2)

Condition Variables



- Used to wait for specific events (especially for long/indefinite waits)
  - free memory is getting low, wake up the garbage collector thread
  - 10,000 clock ticks have elapsed, update that window
  - new data arrived in the I/O port, process it
- Each condition variable is associated with a single mutex
  - In Java, every object is cond. var (ugly!)

Condition Variables in Our Language

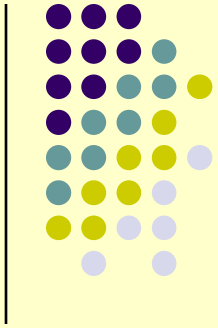


- Cond type with methods
  - `wait(m)`: ***atomically*** unlocks the mutex (as many times as needed) and blocks the thread
  - `notify`: awakes some blocked thread
    - the thread is awoken inside `wait`
    - tries to lock the mutex (maybe many times)
    - when it (finally) succeeds, it returns from the `wait`
  - `notifyAll`: like `notify` but for all blocked threads

Condition Var Example

```
class Buffer {
  Port port;
  Cond c = new Cond();
  public synchronized void consume() {
    while (port.empty())
      c.wait(this);
    process_data(port.first_data());
  }

  public synchronized void produce() {
    port.add_data();
    c.notify();
  }
}
```

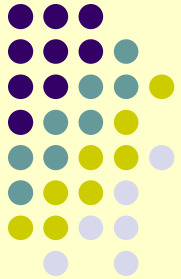
Using Condition Variables



```
class Buffer {
  Port port;
  Cond c = new Cond();
  public synchronized void consume() {
    while (port.empty())
      c.wait(this);
    process_data(port.first_data());
  }
  public synchronized void produce() {
    port.add_data();
    c.notify();
  }
}
```

- Why not “if”? When can we use `notifyAll`? Could we replace condition variables with messaging?

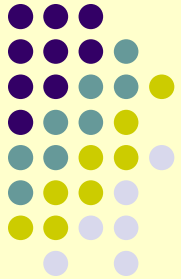
Monitor Style Programming General Pattern



- Armed with mutexes and condition variables, one can implement *any kind of critical section*
  - *CS.enter(); [controlled code] CS.exit();*
- class CS {

```
  [shared data, including c]
  public synchronized void enter() {
    while (![condition]) c.wait(this);
    [change shared data to reflect in_CS]
    [notify as needed]
  }
  public synchronized void exit() {
    [change shared data to reflect out_of_CS]
    [notify as needed]
  }
}
```

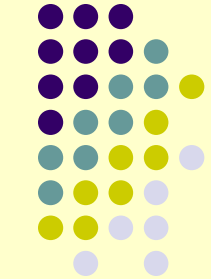
Simplest Example: Implement an Unstructured Mutex



- class Mutex {

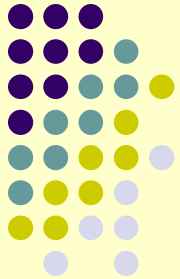
```
    public synchronized void acquire() {


    }
    public synchronized void release() {


    }
}
```

Simplest Example: Implement an Unstructured Mutex



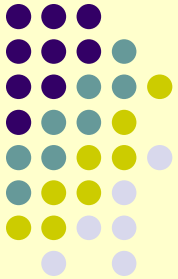
```
class Mutex {
  boolean locked = false;
  Cond c = new Cond();
  public synchronized void acquire() {
    while (locked) c.wait(this);
    locked = true;
  }
  public synchronized void release() {
    locked = false;
    c.notify();
  }
}
```

Classic Example: Readers-Writers Lock



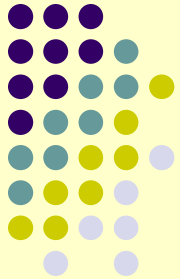
```
class RWLock {
  int readers = 0; boolean writer = false;
  Cond readPhase = new Cond();
  Cond writePhase = new Cond();

  public synchronized void enterRead() {
    while (writer) readPhase.wait(this);
    readers++;
  }
  public synchronized void exitRead() {
    readers--;
    if (readers == 0) writePhase.notify();
  } …
```

Classic Example: Readers-Writers Lock



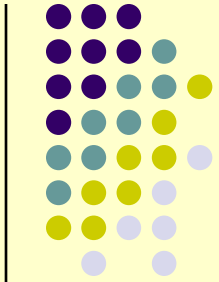
```
…
public synchronized void enterWrite() {
  while (readers != 0 || writer)
    writePhase.wait(this);
  writer = true;
}
public synchronized void exitWrite() {
  writer = false;
  readPhase.notifyAll();
  writePhase.notify();
}
}
```

Comments on Readers/Writers Example

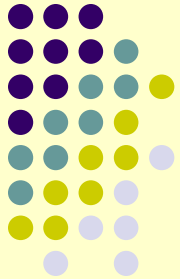


- How would it be different in plain Java?
  - single condition variable for phase changes
- Note the use of `notifyAll`
  - also wakes up many readers that will contend for a mutex
- Writer notifies many threads. Not all can proceed, however (*spurious wake-ups)*
  - how can we avoid this?
- Unnecessary lock conflicts may arise (especially for multiprocessors):
  - both readers and writers signal condition variables while still holding the corresponding mutexes

Many Other Examples (try them!)

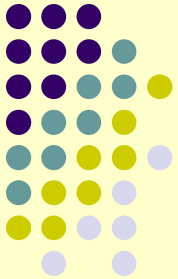


- CS with red/green threads, up to 3 in CS, not all the same color
- Red/green threads, up to 3, red have priority
  - no green can enter if a red is waiting to enter



COMPLEXITIES OF MONITOR STYLE PROGRAMMING

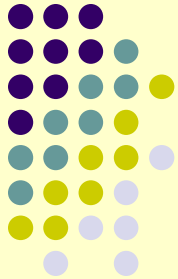
Monitor Style Programming Errors



- Most problems with concurrent programming are simple oversights that are easy to introduce due to partial program knowledge and near-impossible to debug!
  - more on that later
- People forget to access shared variables in locks, forget to signal when a condition changes, etc.

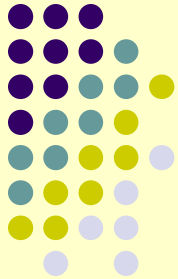
The Golden Rules

(best currently known–still easier said than done)



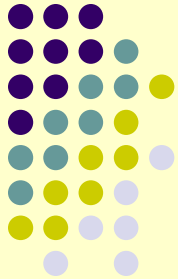
- Associate each shared location with a single mutex, access only while holding the mutex
- Associate each condition variable with a boolean condition (expressed in terms of program variables). Every time the value of the boolean condition may have changed, call `notifyAll` on the cond var
  - only call `notify` when you are certain that *any and only one waiting thread can enter the critical section*
- Globally order locks, acquire in order in all threads

Why Is This Hard In Practice?



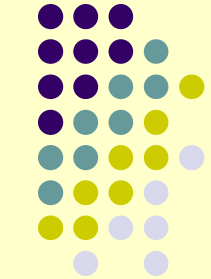
- Holding locks is a global property: affects entire program, cannot be hidden behind an abstract interface
- Results in lack of modularity: callers cannot ignore what locks their callees acquire or what locations they access
  - necessary for race avoidance, but also for global ordering to avoid deadlock
  - part of a method's protocol which lock needs to be held when called, which locks it acquires

Why Is This Hard In Practice?



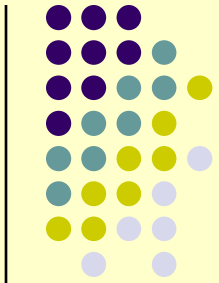
- Cond vars are also non-local: every time some value changes, we need to know which condition var may depend on it to signal it!
- Everything exacerbated by aliasing in imperative languages
  - hard to tell what location a program expression refers to
    - are two locks the same?
    - are two data locations the same?

Why Is This Hard In Practice?



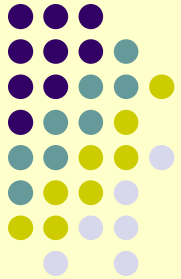
- Even worse: lack of composability, cannot build safe services out of other safe services

Back to Earlier Example



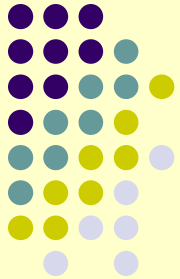
```
class Account {
 int balance = 0;
 public synchronized int withdraw(int amt) {…}
 public synchronized void deposit(int i) {…}
}
class Client1 {
 public synchronized void move (Account a1, Account a2) {
  a2.deposit(a1.withdraw(10));
 }
}
class Client2 … // same as Client1
```

- What if move truly needs to be atomic?
- Deadlock? How can it be avoided?
  - All clients need to know each other!



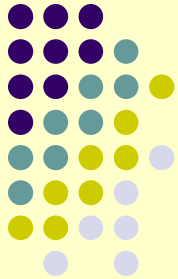
TRANSACTIONAL MEMORY

What is Transactional Memory (TM) ?



- Instead of mutexes and condition variables, atomic code sections
  - `atomic(expr) { … }`
    - block until expr is true (wake up automatically when it changes)
  - typically `abort` and `retry` statements also supported
- The runtime system ensures that the critical section executes transactionally
  - *"as if atomic"*

Semantics (Informally)

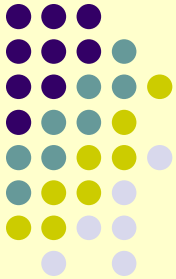


- Transaction properties: atomicity, isolation
  - other threads cannot see intermediate values of the transaction (all-or-nothing)
  - the transaction cannot see values of other transactions in the middle of its execution
    - other transactions have to appear to either have committed before the current one, or after
      - "serializability"

Advantages

- No deadlock!
  - we never refer to locks explicitly
- Composability, modularity
  - no need to know what callees do w.r.t. synchronization

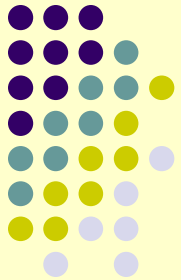
Back to Earlier Example



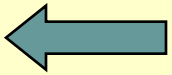
```
class Account {
 int balance = 0;
 public int withdraw(int amt) {atomic{…}}
 public void deposit(int i) {atomic{…}}
}
class Client1 {
 public void move (Account a1, Account a2) {
  atomic{ a2.deposit(a1.withdraw(10)); }
 }
}
class Client2 … // same as Client1
```

- No deadlock problem, atomicity enforced
- `Client1` doesn't need to know anything about `Client2`

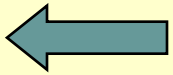
Foundation: Transactional Retrying

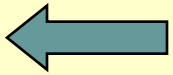


- ```
  atomic {
      x = 3;
      y = x+2;
      …
      z = y + x;
      …
  }
  ```


Begin transact

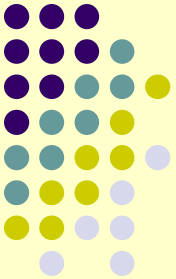
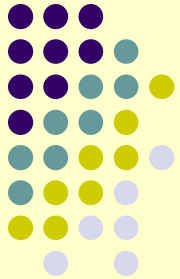
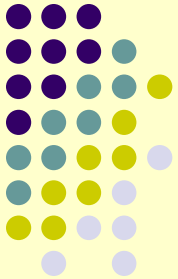
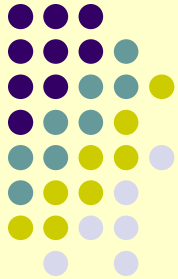
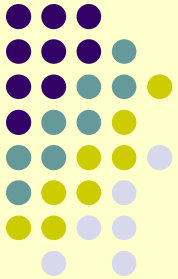
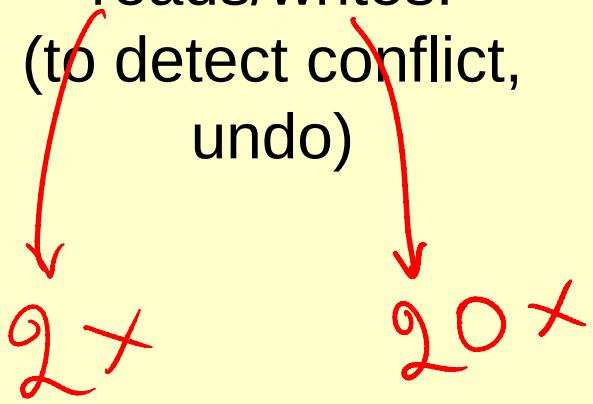

change shared


change shared

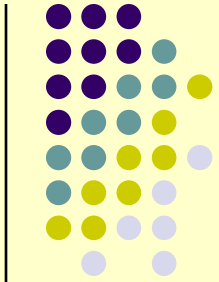

…


if conflict, undo, retry


End transact

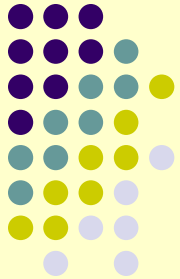

  ```

Implementations



- TM can be implemented in software ("Software TM"—STM) or hardware ("Hardware TM"—HTM)
- Realistically, the (first?) mainstream incarnations will be hybrids
  - processors are expected to offer some support
  - software will take over for more expensive transactions

Let's Think What's Needed

- How would you implement an STM?
  - trivial: acquire a single global lock, right?
  -
```
atomic {
    x = 3;
    y = x+2;
    …
    z = y + x;
    …
}
```

Begin transact
change shared
change shared
…
if conflict, undo, retry
End transact

Implementation Choices: Pessimistic



- Acquire locks before accessing shared data

```
atomic {
    x = 3;
    y = x+2;
    …
    z = y + x;
    …
}
```

Begin transact

acquire lock, change shared

acquire lock, change shared

…

try lock, detect deadlock, undo, retry

End transact

Implementation Choices: Optimistic



- Log all reads/writes to shared data, detect conflicts, at commit acquire locks, post updates
  - `atomic {` ← Begin transact
    - `x = 3;` ← log write, change shared
    - `y = x+2;` ← log read, write, change shared
    - … ← …
    - `z = y + x;` ← try read, detect conflict, undo, retry
    - …
  - `}` ← End transact

A Myriad Variations



- Logging/commit/serialization protocols
  - decided depending on common access patterns
    - e.g., how often do we have read/write conflicts, how often write/write? How many writes vs. reads?
  - early/late conflict detection, locking protocol at commit, etc.
- Locking granularity
  - word level, object level, cache line level
    - also false sharing issues

Software Transactional Memory: Heavy Overheads



- ```
  atomic {
    x = 3;
    y = x+2;
    …
    z = y + x;
  }
  ```

optimistic: need to track all shared memory reads/writes! (to detect conflict, undo)



- Also: overhead for retrying, wasted effort on abort, overhead of lock acquisition


  ```

Footnote: Semantic Complexities



- What happens between transactions and non-transactional code?
- Many different semantic models. Broad categories (very rough, imprecise classification)
  - Strong atomicity: every memory reference outside a transaction appears as if in a little transaction of its own
  - Weak atomicity: few/no guarantees if a memory reference outside a transaction is also accessed inside a transaction

Semantic Complexities, Surprising Example ("privatization")



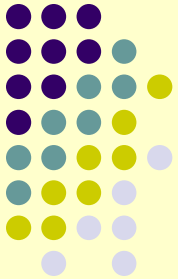
Thread1

```
Item item;
atomic {
  item =
    list.removeFirst();
}
int r1 = item.val1;
int r2 = item.val2;

// Can r1 != r2 ?
```

Thread2

```
atomic {
  if (!list.isEmpty()) {
    Item item =
      list.getFirst();
    item.val1++;
    item.val2++;
  }
}
```

Programming Model Complexities

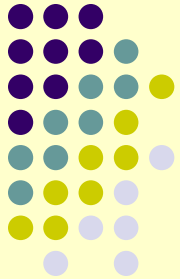


- Important: the potential for “undoing” is inevitable!
  - in both pessimistic and optimistic implementations
    - in pessimistic because of possible deadlock
    - in optimistic because of conflict

Programming Model Complexities

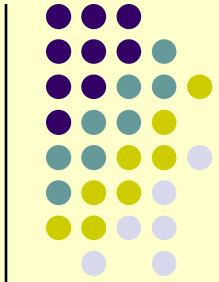
- What if we have done something that cannot be undone?
- Irreversible operations
  - launch
  - format
  - take_medicine
  - shutdown
  - send_email_to_boss_after_three_beers

Problem In Principle



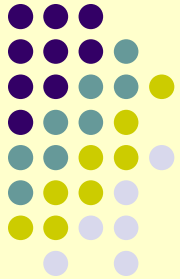
- Destroys the composability properties
  - performing irreversible operations becomes a global property, just as holding locks was a global property
  - caller needs to know whether callee performs irreversible operations
    - why depends on the specific treatment of irreversible operations

Ideas for Addressing: #1



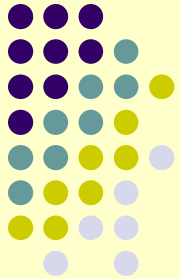
- Completely disallow irreversible operations in transactions
  - typically also enforce/propagate property with a type system
    - a la Haskell
  - draconian, global property, hard to adapt to pre-existing libraries

Ideas for Addressing: #2



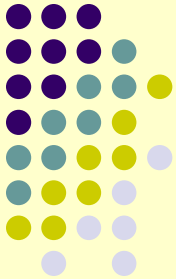
- When encountering irreversible operations, roll back the transaction, acquire single global lock, restart
  - one transaction at a time can be performing irreversible actions
  - limits performance, makes clients non-independent
    - exposes as a performance impact the fact that irreversible actions are a global property

Ideas for Addressing: #3

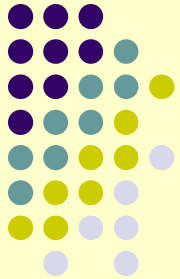


- When encountering irreversible operations, commit the transaction, breaking it up in two ("punctuation")
  - violates atomicity, needs type system for warning to the user, all callers need to know
  - can be combined with system for restoring "local" invariants when a transaction is punctuated
    - in practice turns out to be quite easy to retrofit, even for large applications and external opaque libraries

Ideas for Addressing: #4

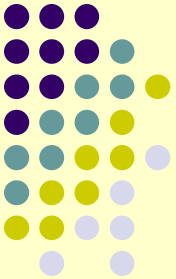


- (Not really addressing the problem, but limiting its impact.)
- I/O is the biggest problem, but most I/O is perfectly reversible
  - only read after write is irreversible
- Buffer reads, delay writes, as long as they are not in a read-after-write pattern
  - perform writes at end of transaction, replay reads on retry



CONCLUSIONS

Concurrent Programming Overview



- You got a good idea of concurrent programming techniques
  - what is used in current practice
    - and how to use it correctly
  - what people do research on
    - and what problems people focus on the most
  - what are the possibilities for the future