# Concurrent and Parallel Programming, Part I

Programming Languages

CS 214

# Introduction

Suppose you and that "special someone" are shopping for champagne and oysters for a romantic dinner…

You might do your shopping either of two ways:

- The "we can't bear to be apart" approach: together, you
  - find the oysters,
  - find the champagne,
  - pay for what you've selected at the checkout
- The "unromantic but efficient" divide-and-conquer approach:
  - one of you finds the oysters,
  - the other finds the champagne,
  - rendezvous at the checkout to pay for what you've selected

If the average time to find champagne+oysters sequentially is $\tau$, then finding them *concurrently* takes about $\tau/2$.

# Concurrent Programming

Most modern programming languages provide built-in support for concurrent processing.

- Ada provides a *Task* construct, and each distinct task is automatically executed concurrently.
- Java provides a *Thread* class, and each distinct thread can be executed concurrently.
- C++ added a standard *thread* class in C++11.
- Elang, Go, Scala … provide multithreading/processing capabilities

Older languages:

- C relies on external libraries for concurrency (e.g., Unix *fork()*, POSIX *pthreads*, OpenMP, MPI, …).
- E-lisp provides a *start-process* function, but it is *not standard Lisp*.

# Example: Ada

We might represent our sequential approach as follows:

```
procedure RomanticApproach is
begin
  FindOysters;       -- do this
  FindChampagne;     -- then this
  PayAtCheckout;     -- then this
end RomanticApproach;
```

By contrast, our concurrent approach is:

```
procedure EfficientApproach is
  task OysterFinder;
  task body OysterFinder is begin
    FindOysters; RendezvousAtCheckout;
  end OysterFinder;
begin
  FindChampagne; RendezvousAtCheckout;
end EfficientApproach;
```

do this

and this simultaneously

# Example: Java

In Java, our sequential approach is expressed as:

```
class RomanticApproach {
 public static void main(String [] args) {
  findOysters();
  findChampagne();
  payAtCheckout();
 }
}
```

By contrast, our concurrent approach is expressed as:

```
class EfficientApproach {
 class OysterFinder extends Thread {
   public void run() {
    findOysters(); rendezvousAtCheckout();
   }
 }
 public static void main(String [] args) {
  OysterFinder of = new OysterFinder();
  of.start();
  findChampagne(); rendezvousAtCheckout();
 }
}
```

# Terminology

- A *uniprocessor* is a computer with one processing core...
  - With a time-sharing OS, concurrent processing results in pseudo-*Parallel execution* (aka *logical concurrency*), because the OS time-shares the single core among the processes/threads.
- A *multiprocessor* is a computer with multiple cores...
  - Concurrent processing results in parallel execution (aka *true concurrency*), as the OS can simultaneously run different processes/threads on different cores.
  - In a tightly-coupled *multiprocessor*, the cores *Share a common main memory*, and are usually in *close physical proximity*.
  - In a loosely-coupled *multiprocessor*, the cores have no shared memory (each has its own *local memory*), are often *not in close physical proximity*.

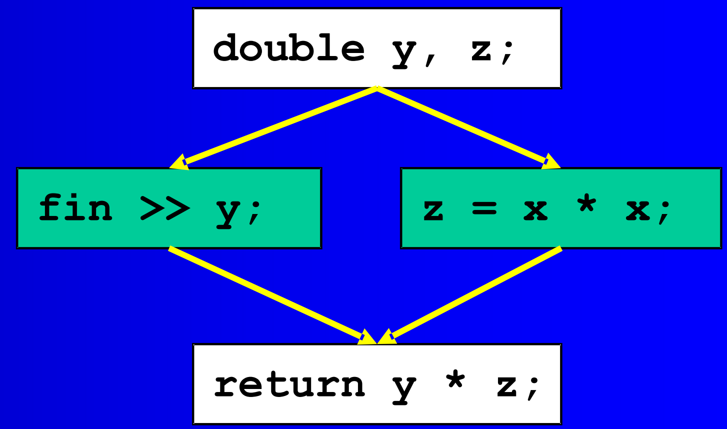
# Parallelizing Compilers

What *dependencies* exist in the following function?

```
void f(double x) {
  double y, z;        // 1
  fin >> y;           // 2 … depends on 1
  z = x * x;          // 3 … depends on 1
  return y * z;       // 4 … depends on 2, 3
}
```

*Parallelizing compilers* build dependency graphs, and use them to identify pieces of code that can be executed concurrently:

- branches in the graph (e.g., 2 & 3) can be safely executed in parallel
- CPU must have the extra hardware to perform *statement-level parallelism*



# Processes and Threads

A computation is sometimes called a *process*:

```
int main() {
   s1;
   s2;
   …
   sn;
}
```

The sequence of *events* that occur as control flows through a process is called a *thread of execution*:

```
s1;
s2;
…
sn;
```

If, for the same inputs, the sequence of events always occurs in the same order, the sequence is called *deterministic*; otherwise, the sequence is called *non-deterministic*.

# Goal: Speedup

One goal of concurrent processing is to achieve speedup:

If a task requires $\tau$ time-units to solve sequentially,

but can be split into p subtasks that can be solved in parallel,

then parallel processing can perform it in $\sim\tau/p$ time-units.

Formally, speedup can be define as:

T1/TN

where: $T_1$ is the time 1 task takes to solve the problem, and

$T_N$ is the time N tasks take to solve the problem.

If it takes one person 10 minutes to find champaign+oysters, but it takes two people 6 minutes, the speedup is 10/6 = 1.67.

# Goal: Responsiveness

… is another goal of concurrent processing.

Example 1: GUIs (Graphical User Interface)

- If an application has a single thread and it has to perform a time-consuming task, then the GUI will “freeze” while the thread is performing that task.
- If the application uses one thread to handle user-interface events and forks a separate thread to perform each task, then the GUI will remain responsive.

Example 2: Network Servers (e.g., a web server)

- If a server has a single thread and has to perform a time-consuming task, then the server will be unable to accept incoming requests while it is performing the task.
- If the server uses one thread to accept incoming requests and forks a new thread to handle each request, the server will accept and handle all requests it receives.

# Event Interleaving

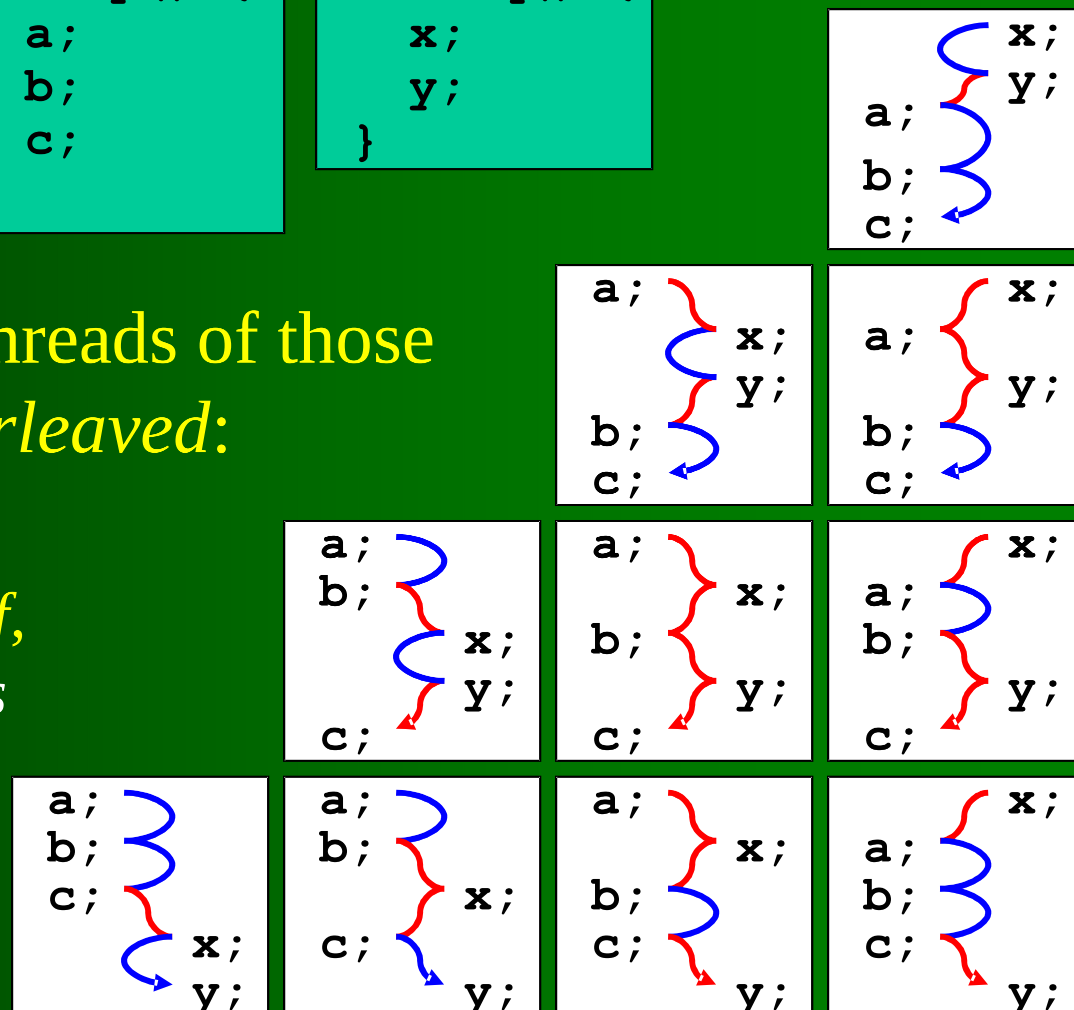
When a computation consists of two or more processes:

```
void p() {
   a;
   b;
   c;
}
```

```
void q() {
   x;
   y;
}
```

then the events in the threads of those processes may be *interleaved*:

– Each sequence is Deterministic *within itself*, but the *sequencing across both processes* is non-deterministic.



# Interleaving in Ada

We can see non-deterministic behavior in Ada as follows:

```
procedure Interleave is
 task A;
 task body A is begin
   put("a");
 end A;
 task B;
 task body B is begin
   put("b");
 end B;
 task C;
 task body C is begin
   put("c");
 end C;
begin
  null;
end Interleave;
```

Sample Run:

```
% ./interleave
___
% ./interleave
___
% ./interleave
___
% ./interleave
___
...
```

How many distinct executions are there?

→ The number of permuattions of {a, b, c}

→ Discrete Math: a set of *n* elements has *n!* permutations, so *3! == 6* distinct possibilities.

# Communication

If a language allows us to divide a task into subtasks, it should also provide a way for those tasks to *communicate*.

- On a tightly-coupled multiprocessor, two tasks can communicate through the *shared memory*:

| S | | R |
|---|---|---|
| msgToR: “Hi” | sharedSpace: “Hi” | msgFromS: “Hi” |
| sharedSpace = msgToR; | | msgFromS = sharedSpace; |

- On a loosely-coupled multiprocessor, they can communicate by *message-passing* (i.e., sending-receiving messages):

| S | | R |
|---|---|---|
| msgToR: “Hi” | → Message-Passing System S:“Hi”:R → | msgFromS: “Hi” |
| Send(R, msgToR); | | Receive(S, msgFromS); |

# Shared Memory Synchronization

If two tightly-coupled processes try to access shared memory simultaneously, the result may be incorrect… What can go wrong?

| TravelAgent1 | | TravelAgent2 |
|---|---|---|
| if (emptySeats > 0)<br>  emptySeats--;<br>else<br>  display("no more seats!"); | emptySeats: 1 | if (emptySeats > 0)<br>  emptySeats--;<br>else<br>  display("no more seats!"); |

Accesses to shared memory must be *synchronizedw* to avoid this:

```
synchronize { // Java
  if (emptySeats > 0)
    emptySeats--;
  else
    display("no more seats!");
}
```

emptySeats: 1

```
synchronize { // Java
  if (emptySeats > 0)
    emptySeats--;
  else
    display("no more seats!");
}
```

Synchronization forces one process to wait until the other is finished.

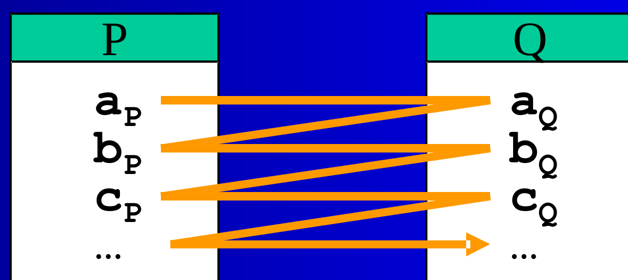
# Synchrony

Concurrent computations lie on a continuum, depending on how much communication/synchronization they entail:

| | |
|---|---|
| Problems whose solutions require *lock-step* synchrony | Problems whose solutions require no synchrony |

minimized — benefit of a concurrent solution — maximize

– *Lock-step synchronous* computations must communicate or be re-synchronized after every step of the computation:



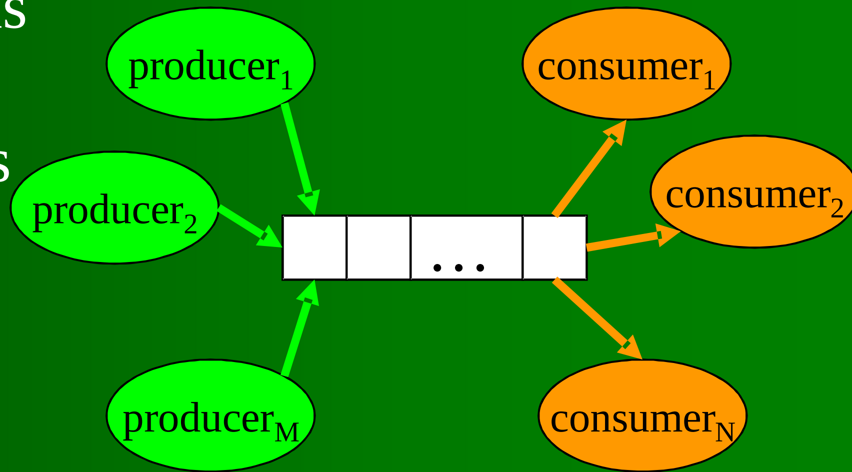

– *Asynchronous* (aka *embarrassingly parallel*) computations require no communication/synchronization of their processes

# The Producer-Comsumer Problem

There are a number of "classic" synchronization problems, one of which is the producer-consumer problem...

There are M *producers* that put items into a buffer. The buffer is shared with N *consumers* that remove items from the buffer.



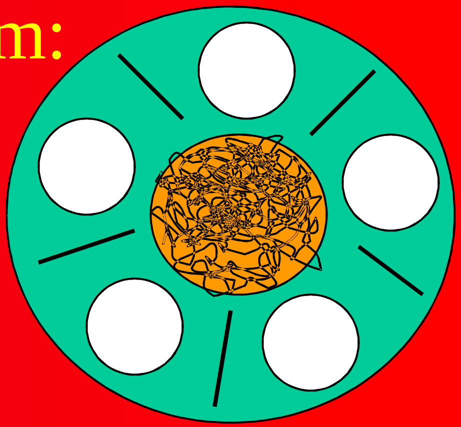

– The problem is to devise a solution that ensures no items are ever lost or duplicated.

Accesses to the buffer must be synchronized: if multiple producers / consumers access it simultaneously, producers may overwrite each other's values, consumers may retrieve the same value, etc.

In the *bounded-buffer version*, the buffer has a fixed-capacity *N*.

# The Dining Philosophers Problem

… is another "classic" synchronization problem:



Five philosophers sit at a table, alternating between eating noodles and thinking. In order to eat, a philosopher must have two chopsticks. However, there is a single chopstick between each pair of plates, so if one is eating, neither neighbor can eat. A philosopher puts down both chopsticks when thinking.

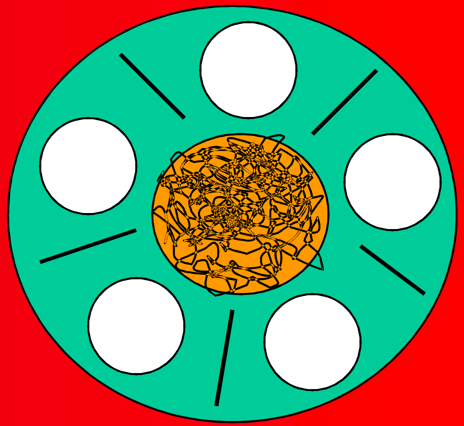
– Devise a solution that ensures:
  - no philosopher starves; and
  - a hungry philosopher is only prevented from eating by his immediate neighbor(s).

```
task philosopher is
  while True begin
     think(randomTime);
     get(left);
     get(right);
     eat();
     release(left);
     release(right);
  end while;
End philosopher;
```

# The Dining Philosophers (ii)

How about this instead?



```
task philosopher is begin
  while True begin
    think(randomTime);
    while not (have(left) and have(right))
     begin
       get(left);
       if notInUse(right) then
         get(right);
       else
         release(left);
       end if;
     end while;
     eat;
     release(left);
     release(right);
 end while;
end philosopher;
```

# Mutual Exclusion

An object that can only be accessed by "one-thing-at-a-time" (e.g., shared memory) is called a *mutually-excluisve* object:

- An access by one process *excludes* other processes from access
- A task may have to 'wait its turn'to access the object
- Many real-world objects (e.g., chopsticks) are mutually exclusive
- Shared-memory *writes* are mutually exclusive: *write-write-conflicts*!

| TravelAgent1 | | TravelAgent2 |
|---|---|---|
| if (emptySeats > 0)<br>  emptySeats--;<br>else<br>  display("no more seats!");<br>end if; | emptySeats: 1 | if (emptySeats > 0)<br>  emptySeats--;<br>else<br>  display("no more seats!");<br>end if; |

- Shared-memory *reads* are not mutually exclusive, unless any task tries to write to the shared memory: *read-write conflicts*!

# Synchronization Primitives

In 1965, Dijkstra proposed the *semaphore*: a shared-memory programming mechanism that can be used to synchronize accesses to a mutually-exclusive resource, with two values: {*locked*, *unlocked*}, and three simple operations:

- *Initialize* the semaphore to *unlocked*
- P: *Lock* the semaphore (wait if it is already *locked*)
- V: *Unlock* the semaphore (awaken the first process waiting for it).

Java 1.7 added a *Semaphore* class:

| Operation | Dijkstra | Java Syntax |
|---|---|---|
| Initialization (unlocked) | s: Semaphore; | s = new Semaphore(1); |
| Lock the semaphore | P(s); | s.acquire(); |
| Unlock the semaphore | V(s); | s.release(); |

# Semaphores and Mutual Exclusion

A semaphore is a shared-memory variable that can be used to enforce mutually exclusive access to other shared-memory variables:

**TravelAgent1**

```
P(s);
if (emptySeats > 0)
  emptySeats--;
else
  display("no more seats!");
V(s);
```

emptySeats: 1
s: unlocked

**TravelAgent2**

```
P(s);
if (emptySeats > 0)
  emptySeats--;
else
  display("no more seats!");
V(s);
```

Whichever travel agent executes P(s) *first* (even by a nanosecond) will lock s, decrement emptySeats, and then unlock s.

Whichever travel agent executes P(s) *second* will find s locked and have to wait until the other agent unlocks s, discover that emptySeats == 0, and then get the “no more seats” message.

# Semaphores and Lockstep Synchrony

A semaphore also permits two processes to execute in lock-step:

| S | | R |
|---|---|---|
| msgToR: | sharedSpace:<br>okToWrite: unlocked<br>okToRead: locked | msgFromS: |
| loop {<br>  P(okToWrite);<br>  sharedSpace = msgToR;<br>  V(okToRead);<br>} | | loop {<br>  P(okToRead)<br>  msgFromS = sharedSpace;<br>  V(okToWrite);<br>} |

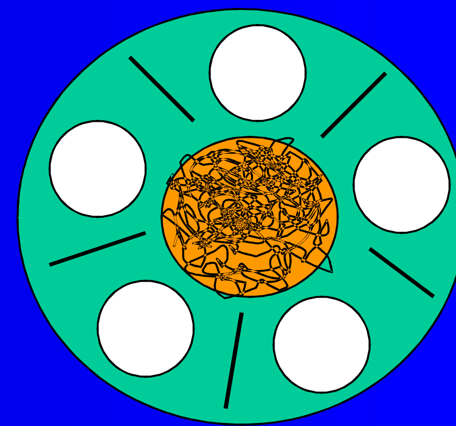
- If *R* executes first, it will wait on the (*locked*) *okToRead* semaphore, until *S* signals (after it writes a value to the shared memory)…
- If *S* executes first, it will lock the (*unlocked*) *okToWrite* semaphore, write its message to shared memory, signal *okToRead*, and then wait on the (locked) *okToWrite* until *R* signals (after it finishes reading).

A *critical section* is a group of statements that access shared space.

# Dining Philosophers (iii)

What if we use a Semaphore?

```
Semaphore: wantBothSticks;
task philosopher is begin
  while True begin
    think(randomTime);
    while not haveBothSticks begin
       P(wantBothSticks);
       if available(leftStick) and
           available(rightStick) then begin
           get(left);
           get(right);
       end if;
       V(wantBothSticks);
     end while;
    eat();
    release(leftStick);
    release(rightStick);
  end while;
end philosopher;
```



This seems to do it, but synchrony is so tricky, it's hard to be 100% certain it's correct ...

# Locks and Condition Variables

A semaphore may be used for either of two purposes:

- *Mutual exclusion*: guarding access to a critical section
- *Synchronization*: making threads/processes suspend/resume

This dual use can lead to confusion: it may be unclear which role a semaphore is playing in a given computation…

For this reason, newer languages provide *distinct constructs for each*:

- *Locks*: guarding access to a critical section
- *Condition Variables*: making threads wait until a condition is true

Locks support mutually-exclusive access to shared memory; condition variables support thread/process synchronization.

# Locks

Like a Semaphore, a lock has two associated operations:

- *acquire()* - try to lock the lock; if it is locked, go to sleep
- *release()* - unlock the lock; awaken a waiting thread (if any)

The *acquire()* is analogous to the Semaphore *P()* operation; the *release()* is analogous to the Semaphore *V()* operation.

These can be used to 'guard' a critical section:

```
sharedLock.acquire();
// access sharedObj
sharedLock.release();
```

```
Lock sharedLock;
Object sharedObj;
```

```
sharedLock.acquire();
// access sharedObj
sharedLock.release();
```

Every Java class inherits a *hidden lock* from class *Object*; the *synchronized* keyword uses it:

```
synchronized {
  // critical section
}
```

# Condition Variables

A *Condition* is a predefined type available in some languages that can be used to declare variables for synchronization.

When a thread needs to suspend execution inside a critical section until some condition is met, a *Condition* can be used.

There are three operations for a *Condition*:

- *wait()*
  - suspend immediately; enter a queue of waiting threads
- *signal()*, aka *notify()* in Java
  - awaken a waiting thread (usually the first in the queue), if any
- *broadcast()*, aka *notifyAll()* in Java
  - awaken all waiting threads, if any

Every Java class inherits it from class *Object* a hidden condition-variable, and the *wait()*, *notify()* & *notifyAll()* methods that use it.

# Monitors

Semaphores, Locks, and Conditions are simple but powerful synchronization tools; but many believe that they are *too powerful for the average programmer* (like the *goto*)…

– Deadlocks/livelocks/non-mutial-exclusion are easy mistakes to make

Just as control structures were "higher level" than the *goto*, language designers began looking for higher level ways to synchronize processes.

In 1973, Brinch-Hansen and Hoare proposed the *monitor*, a class whose methods are automatically accessed in a mutually-exclusive manner.

– A monitor *prevents simultaneous access by multiple threads*

# Mesa-Style Monitors

*Concurrent Pascal* was first to provide a *Monitor* construct:

```
type BoundedBuffer = Monitor
 constant N := 1024;
 myHead, myTail, mySize: integer := 0;
 myValues : array(0..N-1) of Object;
 notEmpty, notFull: Condition;
 procedure put(obj: Object) begin
    while mySize = N do notFull.wait; end;
    myValues(myHead) := obj;
    myHead := (myHead + 1) mod N; mySize := mySize + 1;
    notEmpty.signal;
 end;
 procedure get(var obj: Object) begin
   while mySize = 0 do notEmpty.wait; end;
   obj = myValues(myTail);
   myTail = (myTail + 1) % N; mySize := mySize - 1;
   notFull.signal;
 end;
end;
```

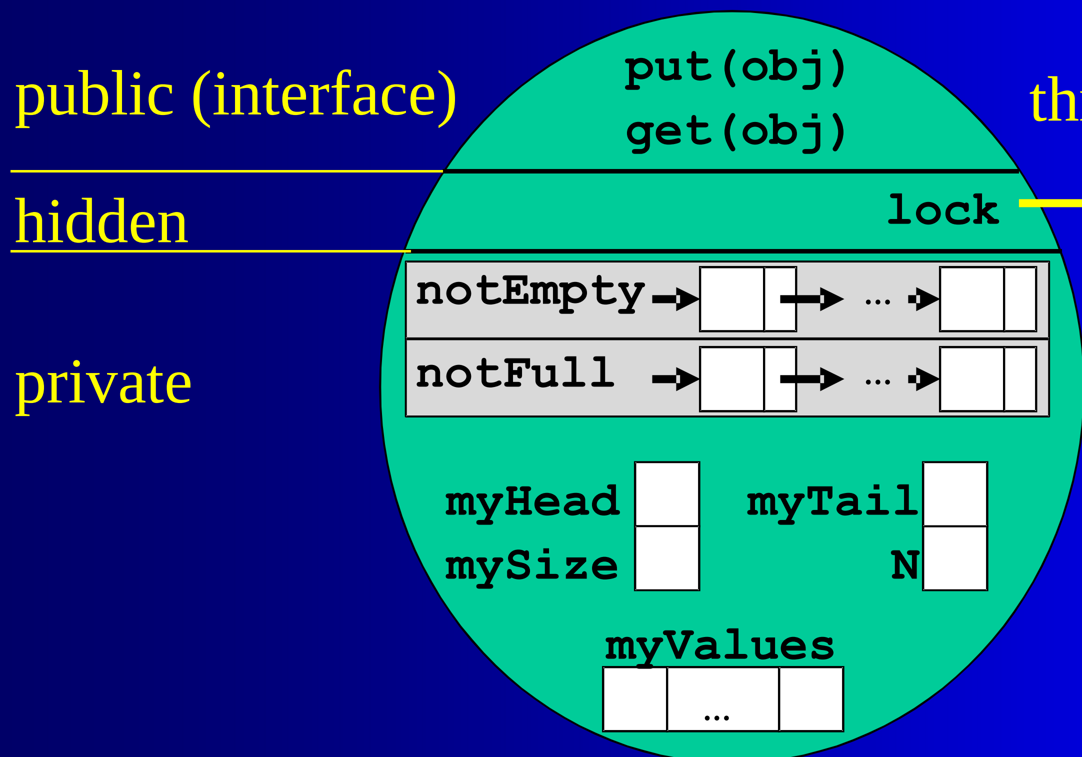
  

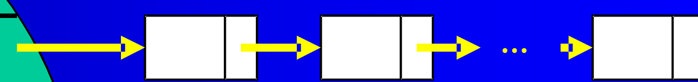
# Monitor Visualisation

The compiler 'wraps' calls to *put()* and *get()* as follows:

```
buf.lock.acquire();
     … call to put or get
buf.lock.release();
```

If the lock is *locked*, the thread enters the entry queue




*entry queue*

Each cond. variable has its own internal queue, in which sleeping threads wait to be signaled…

# Java Monitors

Java classes with *synchronized* methods are monitors…

Example: Let's build a self-synchronizing *BoundedBuffer* class:

```
public class BoundedBuffer extends Object {
 private int mySize, myMax,
             myHead, myTail;
 private Object [] myValues;
 public BoundedBuffer(int n) {
   myMax = n;
   mySize = myHead = myTail = 0;
   myValues = new Object[n];
 }
 public synchronized int size() { return mySize; }
 public int capacity() { return myMax; }
 public synchronized int isFull() { return mySize == myMax; }
 public synchronized int isEmpty() { return mySize == 0; }
 // … continued on next page …
```

A *synchronized* method acquires the class's lock to guarantee "one-thread-at-a-time" execution...

# Buffer Synchrony

```
// … continued from previous page …
 public synchronized void put(Object obj) {
    while ( this.isFull() )
       try{ wait(); } catch(Exception e) {}
    myValues[myHead] = obj;
    myHead = (myHead + 1) % myMax;
    mySize++;
    notifyAll();
 }
 public synchronized Object get() {
   Object result;
   while ( this.isEmpty() )
      try{ wait(); } catch(Exception e) {}
   result = myValues[myTail];
   myTail = (myTail + 1) % myMax;
   mySize--;
   notifyAll();
   return result;
 }
}
```

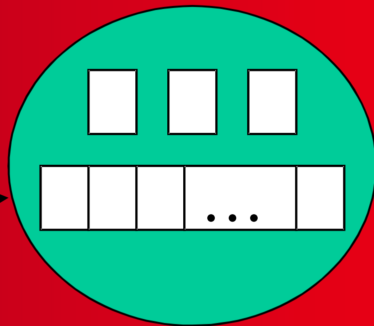
The wait() operation causes the executing thread to *suspend*.

The notifyAll() operation *awakens* all waiting threads.

# Bounded Buffer Producer-Consumer

We can then use our *BoundedBuffer* as follows:

```
// producer thread
for (;;) {
  // produce Item it;
  buf.put(it);
}
```




```
// consumer thread
for (;;) {
  buf.get(it);
  // consume Item it;
}
```

Self-synchrony: No synchronization needed in producer or consumer.

Recall: Every Java class inherits a hidden *lock* from *Object*…

– When a *synchronized* method is called, it tries to *acquire* the lock (waiting if it is already locked), and *releases* the lock on termination.

Every Java class inherits a hidden *condition variable* from *Object*…

– wait() suspends a thread on the condition; notify() awakens a thread waiting on the condition; notifyAll() awakens all waiting threads.

# More Recent Java

In 2005, Java 1.5 added the package java.util.concurrent:

- A ThreadPool class and an Executor framework to make the management of groups of threads easier and more convenient
- Classes for thread-safe data structures (list, queue, map, …)
- Classes for synchronization (semaphore, barrier, mutex, latch, …)
- Classes for creating lock and condition variables;
- Classes for atomic operations (arithmetic, test-and-set, …)

Subsequent Java releases have continued to add features:

- Futures, for asynchronous computations
- ForkJoinTasks and ForkJoinPools for recursive parallelism
- Lambda expressions, CompletableFutures, parallel streams, WorkStealingThreadPools for load-balancing, …

# OpenMP …

… stands for *Open MultiProcessing*

… is an industry-standard library for shared-memory parallel computing in C, C++, Fortran, …

… uses #pragma directives for implicit fork-join multithreading

… simplifies the task of parallelizing legacy code

… was designed by a large consortium in 1997: *AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, Nvidia, Oracle, Redhat, TI, …*

… has "built in" support for many parallel design patterns

… continues to evolve (OpenMP 2.0 in 2000; 3.0 in 2008; 4.0 in 2014, …; current version is 4.5)

# Example: Summing an Array

```
#include <iostream>
#include <omp.h>

// getFileName(), readFile(), ...

int main(int argc, char** argv) {
   string filename = getFileName(argc, argv);
   vector<int> v = readFile(filename);

   #pragma omp parallel for reduction(+:sum)
   for (int i = 0; i < v.size(); i++) {
      sum += v[i];
   }

   cout << "The sum of the values in '"
        << fileName << "' is " <<
        << sum << endl;
}
```

On a machine with *t* cores, this directive implicitly Forks t-1 new threads

An implicit join occurs at the end of the stmt that follows the #pragma

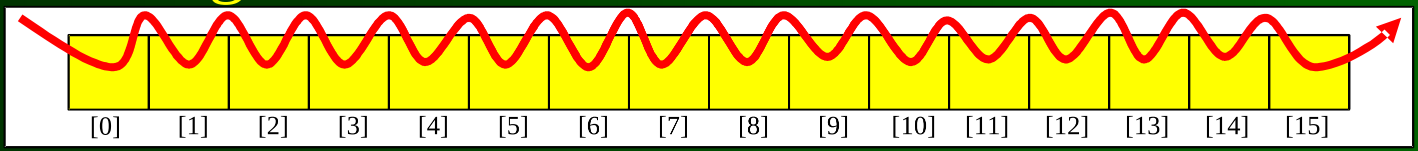
  

# #pragma omp parallel for

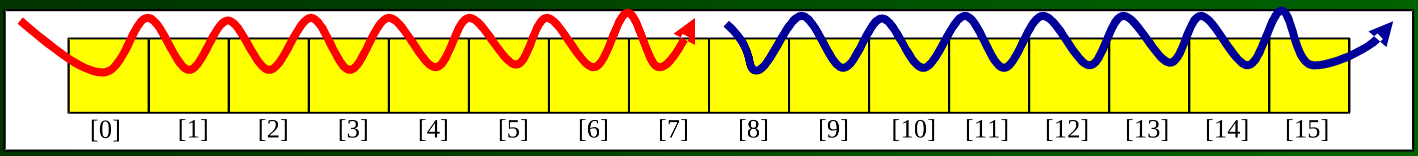
The #pragma omp parallel directive forks the threads…

The for clause auto-divides the iterations of the loop that follows it across those threads.

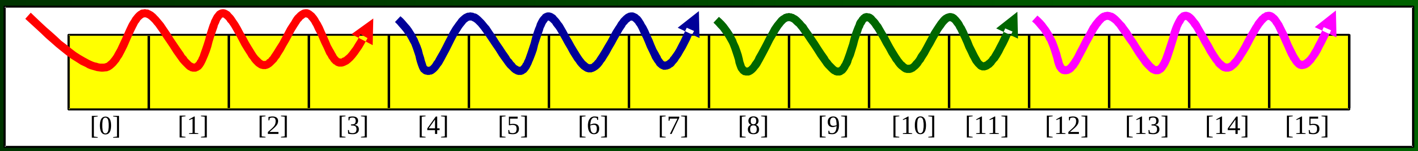
On a single-core machine: one thread…




On a dual-core machine: two threads…



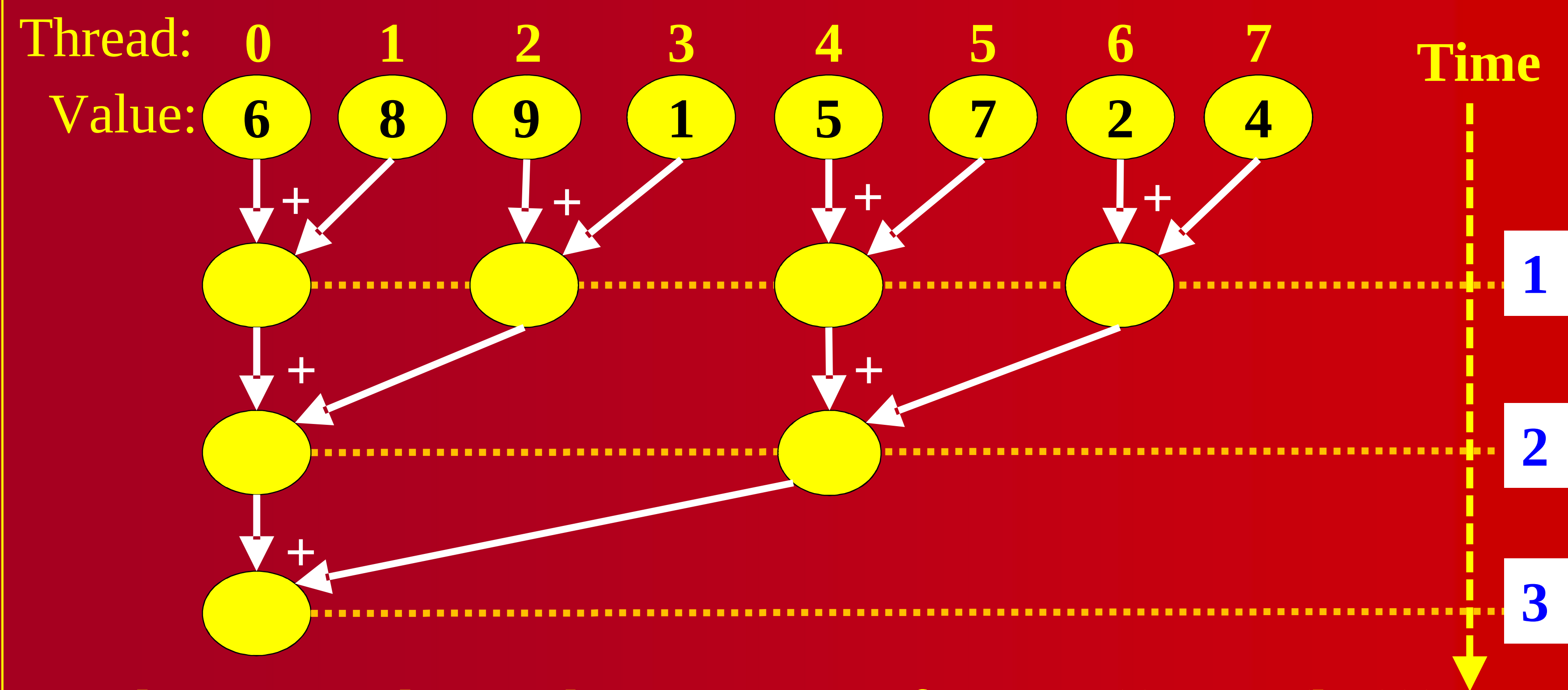

On a quad-core machine: four threads…



# The Reduction Clause

… in the #pragma omp parallel for reduction(+:sum) uses the + operator to combine the partial sums into variable sum.




Reduction reduces the sum-time from *O(t)* to O(lg(t)) …

# OpenMP also supports…

Other ways to reduce local results: +, *, -, /, &, |, ^, &&, ||, …

Other directives: *#pragma omp ________...*

- *critical*
- *atomic*
- *barrier*
- *sections*
- *section*
- *task*
- *task*
- *teams*
- *simd*
- *single*
- *master*
- …

Library functions:

- *omp_set_num_threads()*
- *omp_get_num_threads()*
- *omp_get_thread_num()*
- *omp_get_num_procs()*
- *omp_init_lock()*
- *omp_set_lock()*
- *omp_unset_lock()*
- *omp_test_lock()*
- *omp_get_num_teams()*
- *omp_get_team_size()*
- *omp_get_wtime()*
- …

Much, much more!