EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 1: August 24

Lecturer: Vijay Garg *Scribe: Michael Bartling*

1.1 Introduction

In this class we introduce the concept of concurrent programming using multiple libraries including Thread programming in Java, pthreads in C, openMP, and CUDA (for GPU compute). We also introduce the concept of formal verification of parallel programming. Finally, we end with an example of a set of parallel forms of a simple algorithm.

1.2 Special Note

Dr. Gargś office hours are a gun free zone.

1.3 Parallel Frameworks

Note: all of the following code examples can be found at the course GitHub `https://github.com/mbartling/UT-Garg-EE382C-EE361C-Multicore/tree/master/chapter1-threads`

1.3.1 Java

Parallelism in Java can be achieved in one of a few ways. In the first method we just extend the Java Thread class:

```
public class HelloWorldThread extends Thread {
    public void run() {
        System.out.println("Hello World");
    }
    public static void main(String[] args) {
        HelloWorldThread t = new HelloWorldThread();
        t.start();
    }
}
```

Another option is to implement `Run()` such as:

```
class Foo {
    String name;
    public Foo(String s) {
        name = s;
```

```
    }
    public void setName(String s) {
        name = s;
    }
    public String getName() {
        return name;
    }
}
class FooBar extends Foo implements Runnable {
    public FooBar(String s) {
        super(s);
    }
    public void run() {
        for (int i = 0; i < 10; i++)
            System.out.println(getName() + ": Hello World");
    }
    public static void main(String[] args) {
        FooBar f1 = new FooBar("Romeo");
        Thread t1 = new Thread(f1);
        t1.start();
        FooBar f2 = new FooBar("Juliet");
        Thread t2 = new Thread(f2);
        t2.start();
    }
}
```

1.3.2 OpenMP

OpenMP makes trivial parallel paradigms in C, such as parallel for loops and simple threads, well *trivial*. The following code snippet shows a simple task creation, however parallel for loops are about as simple as `parfor` in Matlab. In GCC on Linux you need to use the `-fopenmp` flag to compile with openMP support.

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

main ()  {

int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
```

```
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  } /* All threads join master thread and terminate */

}
```

1.3.3 pthreads

POSIX threads or *pthreads* for short, are a type of thread compatible with the POSIX standard. pthreads are most commonly programmed in C and C++, however they are available in other languages. Furthermore, pthreads are approximately compatible with most common operating systems. Note, in GCC you need to link against the pthreads library to use them, this can be done with the `-lpthread` linker flag.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     5

void *Hello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! from thread %d\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     rc = pthread_create(&threads[t], NULL, Hello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

   pthread_exit(NULL);
}
```

1.3.4 GPGPU's and CUDA Programming

Traditionally, GPU programming has been *locked in* to graphics oriented tasks. Over time this programming model has shifted towards a more software defined model. Nowadays, GPGPU's, or general purpose graphics processing units, handle many highly parallelizable tasks such as computing the back-propagation

algorithm in Deep Neural Networks, Hogwild gradient descent methods, recursive and iterative ray-tracing, and other performance critical parallel computations. The basic programming paradigm is the *kernel*. The kernel describes both a task and how to distribute this task on the GPU including the resources to utilize. Optimizing the kernel is challenging both for programmers and for compiler writers.

NOTE: CUDA programs can be compiled using `nvcc`.

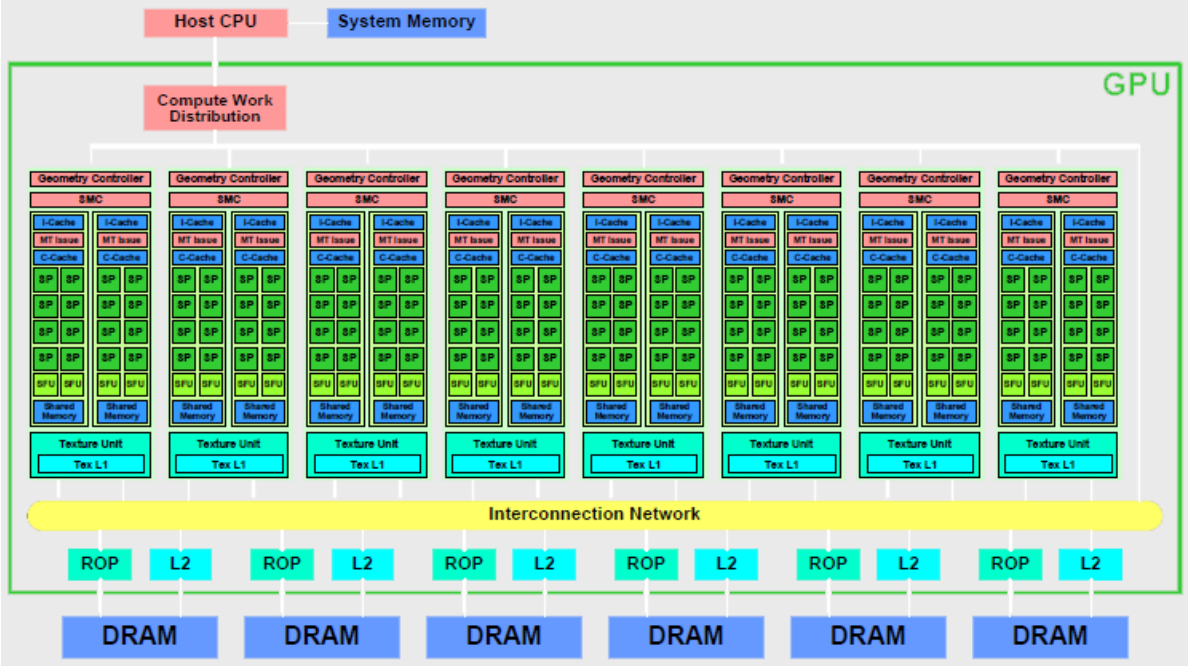


Figure 1.1: GPU Architecture: GPU's are effectively a multicore processor where each *core* (approximately 15-20) has its own set of compute units. These compute units are often confused with cores in the whitepapers (e.g. the Nvidia GTX 980 has 2200 CUDA cores!). The true cores are often referred to as Streaming Multiprocessors.

```
#include <stdio.h>

#define NUM_BLOCKS 16
#define BLOCK_WIDTH 1

__global__ void hello()
{
    printf("Hello world! I'm a thread in block %d\n", blockIdx.x);
}


int main(int argc,char **argv)
{
    // launch the kernel
    hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();

    // force the printf()s to flush
    cudaDeviceSynchronize();
```

```
    printf("That's all!\n");

    return 0;
}
```

1.4 Formal Verification

Oftentimes, determining whether our concurrent model is correct is a non-trivial task. Luckily tools like Promela and Spin exist to formally verify our concurrent model. Spin is used to run promela files and is invoked as following

```
$> spin promelaFile.pml
```

The following is an example of a Promela model:

```
active [2] proctype user()
{
    printf("Hello from thread %d\n", _pid);
}

init {
    printf("Main: Hello from thread %d\n", _pid);
}
```

1.5 Ideal Parallel RAM model: PRAM

We can think of PRAM as an "Ideal Parallel Machine," where each processing element, PE, communicates with a shared memory structure. See 1.5. This model is accessed using multiple methods including:

- CREW
- EREW
- Common CRCW

Where E stands for exclusive, R for read, W for write, and C for concurrent.

1.6 Max Element Algorithm

Suppose we want to find the max element in a sequence or array.

1.6.1 Naive Algorithm

The naive approach is to just access each element 1 time sequentially computing the max. This has the following time and work complexities, assuming all the numbers are unique:

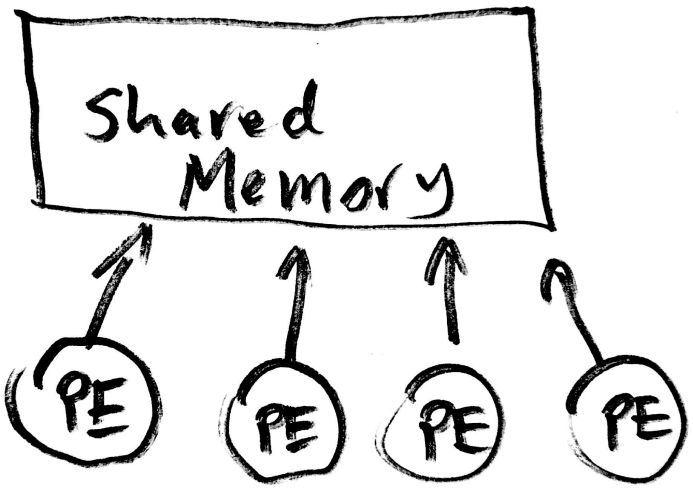


Figure 1.2: PRAM Model

$$T(N) = O(N)$$
$$W(N) = O(N)$$

1.6.2 Binary Tree Algorithm

But can we do better? Yes. If we treat the decision problem as a binary tree then we can get *logn* time complexity. Note, this processes is similar to MapReduce. The basic idea is to divide and conquer. Here, the original list is assigned to a set of workers who make decisions which look like a binary tree. The root of this tree is the max value.

$$T(N) = O(logN)$$
$$W(N) = O(N)$$

Note, we cannot do better than $O(N)$ work complexity since we must visit all nodes. This is just common sense.

1.6.3 Ludicrous Speed

What about an algorithm using as many workers as possible to solve the task? Then we can embed the max value problem into a sort of dependency matrix.

$$T(N) = O(1)$$
$$W(N) = O(N^2)$$

Algorithm 1 Ludicrous Speed Algorithm

```
∀i, isBiggest[i] := 1
for all i, j do
    if A[j] > A[i] then
        isBiggest[i] ← 0
    end if
end for
if isBiggest[i] then
    MAX := A[i]
end if
```

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 1: August 24

Lecturer: Vijay Garg *Scribe: Rahul Jaisimha*

1.1 Introduction

This class was just an introduction to the course. It explained how to write concurrent programs using the different framework/languages we will be using in class. All files used in this lecture can be found in the `chapter1-threads` folder of the class github [1].

1.2 First Things First

Dr. Garg's office is a gun-free zone.

1.3 Writing Concurrent Programs

1.3.1 Java

See github file `java/HelloWorldThread.java`. Concurrent programming in java is implemented by extending the *Thread* class in java and overriding the `run()` function. `thread.start()` is used to start the thread. This program can be run on Unix with:

```
javac HelloWorldThread.java
java HelloWorldThread
```

1.3.2 OpenMP

See github file `openMP/hello.c`. Concurrent programming using openMP is implemented by including `omp.h` and using `-fopenmp` as an argument during compilation (as seen in compile.bat also on github). Use `#pragma` to run things in parallel. In `hello.c`, every thread should have a private copy of `tid`.

1.3.3 Pthreads

See github file `pthreads/hello.c`. This is the longest program. Compilation is normal unlike openMP. The program just has to include `pthread.h`.

1.3.4 Promela

See github file `spin/helloThreads.pml`. The program starts at `init` much like `main()` in other languages. Input the promela file and properties into *Spin* to run.
This program can be run on Unix with:
`spin helloThreads.pml`

1.3.5 Cuda

See github file `cuda/hello.cu`. Cuda is used to write GPU programs. More on Cuda programming in the next section.

1.4 GPU Programming

GPGPU = General Purpose GPU. GPGPUs are useful for matrix multiplication, neural networks, amongst other things. GPU programming relies on SIMD (Single Instruction Multiple Data). This means programming multiple processes to do the same thing on different data.

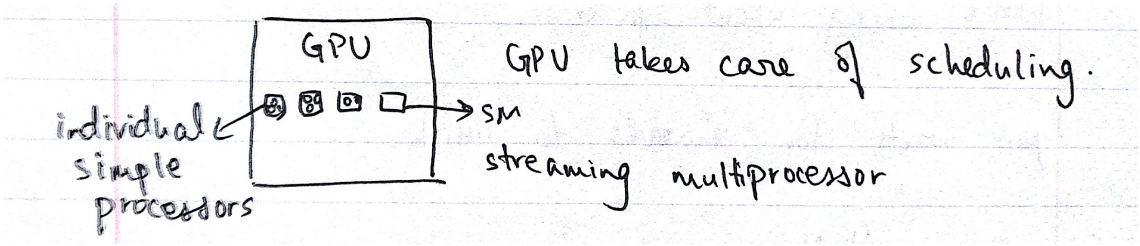


Figure 1.1: GPU

1.4.1 Cuda cont'd.

Cuda uses the notion of kernel. the `__global__` is used to denote running on the gpu. Cuda programs are compiled with the Nvidia Cuda compiler `nvcc`. Note that optimizing GPU programs is a lot harder than optimizing CPU programs.

1.5 PRAM

PRAM = Parallel RAM. PRAM is purely an abstract concept for developing parallel algorithms that assumes shared memory between many processing elements.

Algorithms for accessing the shared memory from each processing element are either of CRCW, CREW, ERCW, or EREW. E: Exclusive, C: Concurrent, R: Read, W: Write.

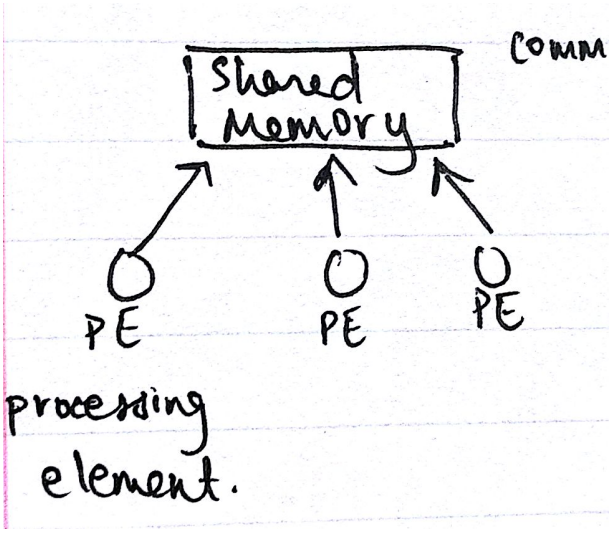


Figure 1.2: PRAM

1.6 The Dr. Garg Traditional First Day of Class Puzzle

Assume you have an array of unique natural numbers (non-negative numbers). Let its size be N. Find the largest element in the array.
In this section we use work complexity to denote number of compares. We also assume we have N^2 cores (or as many cores as we need).

1.6.1 Sequential Algorithm

Time Complexity = $O(N)$
Work Complexity = $O(N)$
Cores needed = 1

1.6.2 Binary Tree Algorithm

Split the array into a binary tree and compare two at a time recursively.

Time Complexity = $O(\log N)$
Work Complexity = $O(N)$
Cores needed = N

1.6.3 "Usain Bolt Algorithm"

Time Complexity = $O(1)$
Work Complexity = $O(N^2)$
Cores needed = N^2

Algorithm 1 "Usain Bolt algorithm"

```
∀i, isBiggest[i] := 1
for all i, j do
    if A[j] > A[i] then
        isBiggest[i] ← 0
    end if
end for
if isBiggest[i] then
    MAX := A[i]
end if
```

1.6.4 Epilogue

Okay so that was pretty good, but can we solve this problem with a good time complexity like $O(\log N)$ without using so many cores?

References

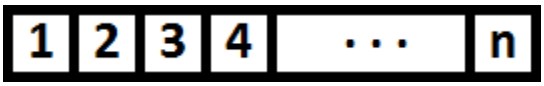
[1] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/tree/master/chapter1-threads

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 2: August 30

Lecturer: Vijay Garg *Scribe: Doyoung Kim*

2.1 Review of Last Lecture



Let's say there is an array with n unique, unsigned integers. If you want to find the max of array, what kind of algorithm would you use to find the max with fastest time and with least amount of work?

2.1.1 Basic Sequential Algorithm

How about good ol' sequential algorithm? First, assume that integer on index 0 is the max. (Let's say we store the value in the variable called *tempMax*) Then, compare *tempMax* with another integer on the next index. Only if 'another integer' is greater than *tempMax*, simply replace *tempMax*'s value with 'another integer'. Continue incrementing the index until the end. This algorithm has execution time of $O(n)$ and work load of $O(n)$.

2.1.2 Binary Algorithm

Another way is to compare integers by pairs. First, compare integers by pairs; first thread compares index 0 and 1, second thread compares index 2 and 3, and so on. Next, compare greater-of-twos by pairs; if index 0 was greater than index 1 and index 3 was greater than index 2, compare index 0 and index 3. Repeat until you get the max. This algorithm has execution time of $O(log(n))$ and work load of $O(n)$.

2.1.3 All Pair Comparison Algorithm

Now, what if you were given huge amounts of threads, let's say n^2? Now, make threads to check all possible, unique combinations and change the value to 0, if value is less than other value. After one run, there will be only one non-zero value and that is the max. This algorithm has execution time of $O(1)$, but work load of $O(n^2)$.

2.2 Ways to Create Threads in Java

There are multiple ways to create threads in Java:

- Extend **Thread** -> Overwrite method **run()** -> Call method **start()** (ex. Fibonacci.java)
- Implement **Runnable** -> Overwrite **run()** -> Create Thread -> Call **start()** (ex. FooBar.java)

- Implement **Callable** -> Overwrite **call()** -> Use **ExecutorService** class to start (ex. Fibonacci2.java)
- Extend **RecursiveTask** -> Overwrite **compute()** -> Use **ForkJoinPool** class (ex. Fibonacci3.java)

Note that **Thread** and **Runnable**'s **run()** do not return anything, while **Callable**'s **call()** and **RecursiveTask**'s **compute()** do.

ExecutorService object lets program to handle the threads, rather than programmer manually handling the threads. Combination of **RecursiveTask** and **ForkJoinPool** also make program to handle the threads. Unlike **ExecutorService**, **RecursiveTask** enables threads to work on other job as waiting for other value(s).

Threads could be **join**ed and those threads wait until all joined threads complete their tasks. **Future** class could be also used, instead of **join**. **Future** class lets thread to run and stops only when program needs computed value.

2.3 Amdahl's Law

Let's say, we have large tasks to complete. If we can define $\mathbf{p}$, a fraction of the work that can be parallelized, we can calculate the limit on speedup for this task by Amdahl's Law. Say, $\mathbf{n}$ is number of cores, $\mathbf{T_p}$ is time on multicore machine, and $\mathbf{T_s}$ is time on sequential process. By Amdahl's Law, we get following formula:

$$\mathbf{T_p} \geq (\mathbf{1} - \mathbf{p})\mathbf{T_s} + \frac{\mathbf{pT_s}}{\mathbf{n}}$$

Which can be used to derive function for **Speedup**:

$$\frac{\mathbf{T_s}}{\mathbf{T_p}} \leq \frac{\mathbf{1}}{\mathbf{1} - \mathbf{p} + \frac{\mathbf{p}}{\mathbf{n}}}$$

2.4 Mutual Exclusion

Suppose there are two threads that are doing same task, using same variable x with initial value of 0 :

$\mathbf{T_1}$	$\mathbf{T_2}$
$x = x + 1$	$x = x + 1$

If those two tasks are executed at the same time, what will x be after the execution?

Answer is 1. Why not 2? It is because this process is not atomic. Because both threads were executed at the same time without variable locking or any check, when they started to run, they both see the initial value of x, which is 0. Both tasks, then, will add 1 to the value they saw, and store it to variable x. Conditions such as this, where more than one threads may read and write on same variable at the same time, is called **critical section**. To avoid *critical section*, code has to be executed atomically.

2.4.1 Door Problem

Assume there are two threads, P_1 and P_2 trying to use same 'door'.

$\mathbf{P_0}$	$\mathbf{P_1}$
loop	loop
— Entry —	— Entry —
Critical Section (CS)	Critical Section (CS)
— Exit —	— Exit —
Not Critical Section (NCS)	Not Critical Section (NCS)
Go back to loop	Go back to loop

If two threads are executed at the same time, how can we prevent *critical section*?

How about using a variable to keep track of the door (Ex. Attempt1.java)? In this process, each thread checks whether door is open or not by looking at boolean variable *openDoor*. While *openDoor* is false, wait until it becomes true. If *openDoor* is true, set it to false and then execute code. After all necessary lines are executed, set *openDoor* back to true. This attempt does not work, because threads may check *openDoor* at the same time before any one thread sets *openDoor* to false.

How about using two variables instead of one (Ex. Attempt2.java)? In this process, there is a boolean array *wantCS* and each index of *wantCS* is assigned to all threads (P_0 gets index 0 and P_1 gets index 1). A thread sets its corresponding *wantCS* to true, right before it checks whether other thread wants or not, This attempt does not work, because if both threads set their *wantCS* to true, none of both threads threads will be able to get past while loop. This situation, which all threads are stuck in some part of the code, is called **deadlock**.

How about manually setting which thread goes first (Ex. Attempt3.java)? In this process, P_0 will execute its code first and then let P_1 to execute. After P_1 executes its code, it will let P_0 to execute. This attempt does work, but since threads don't know whether other thread needs to execute or not, there may be unnecessary waiting time.

2.4.2 Peterson's Algorithm

Peterson's Algorithm (Ex. PetersonAlgorithm.java) uses approaches of Attempt2 and Attempt3 (*wantCS* and *turn*). This process gives each thread a turn, but also checks whether that thread wants to execute the code or not.

Check for Deadlock Freedom

$$Deadlock \equiv (wantCS[1] \wedge turn == 1) \wedge (wantCS[0] \wedge turn == 0)$$
$$\Rightarrow turn == 1 \wedge turn == 0$$
$$\Rightarrow false$$

Check for Mutual Exclusion
Let's add another boolean array, *trying*. Only when thread is checking for while-loop conditions, corresponding index of *trying* becomes true. Otherwise, it is false.

Consider predicate H, where:

$$H(0) \equiv wantCS[0] \wedge [(turn == 1) \vee ((turn == 0) \wedge trying[1])]$$
$$H(1) \equiv wantCS[1] \wedge [(turn == 0) \vee ((turn == 1) \wedge trying[0])]$$

In here, P_0 cannot falsify $H(1)$. Because only P_1 can change the value of wantCS[1], $H(1)$ will be falsified only if $wantCS[1] \wedge turn == 1 \wedge trying[0]$ holds true. When *turn* is set to 1 by P_0, trying[0] is false. From symmetry, P_1 also cannot falsify $H(0)$.

Now, check for the mutual exclusion:

$$\begin{aligned} CriticalSection &\equiv \neg trying[0] \wedge H(0) \wedge \neg trying[1] \wedge H(1) \\ &\Rightarrow turn == 0 \wedge turn == 1 \\ &\Rightarrow false \end{aligned}$$

References

[1] V. K. Garg, Introduction to Multicore Computing, pp. 21-22.

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 2: August 30

Lecturer: Vijay Garg *Scribe: Rohan Nagar*

2.1 Puzzle (Continued)

Recall the puzzle from last time. We have an array of n integers and we want to find the maximum value in the array. Our best attempt so far was the *all pair comparison*, with execution time equal to $O(1)$ and work done equal to $O(n^2)$. Think of it as you have n people each with n helpers. They each can figure out themselves if they are the max value beacuse each of the n helpers can look at the other n numbers and determine if that number is bigger or smaller.

The problem with this algorithm is the work that needs to be done. It requires n^2 processors.

Let us introduce a new algorithm to improve on the amount of work done. We can divide the n integers into $\sqrt{n}$ groups of $\sqrt{n}$. Then, we find the maximun value in each group, and then finally one more step to find the maximum value overall. This still gives us an execution time of $O(1)$, but cuts the work down to $O(n^{\frac{3}{2}})$. All of our solutions so far are listed in the table below.

Algorithm	T	W
Sequential	$O(n)$	$O(n)$
Binary Tree	$O(log(n))$	$O(n)$
All Pair Comparison	$O(1)$	$O(n^2)$
Group-Based	$O(1)$	$O(n^{\frac{3}{2}})$

Question Can you find another solution that improves further on the amount of work done?

2.2 Creating Threads (in Java)

There are three ways to work with threads in Java.

- Extend Thread → call start()
- Implement Runnable or Callable → create new Thread → call start()
- Use Executor Service

2.2.1 Sample Code

All of the sample code examples are available on Dr. Garg's class Github page.

- FooBar.java → Implementing Runnable

  - Motivation for Implementing Runnable: Say, for example, that you already have a class that you want to keep the functionality of. You want to extend this class, but in Java you cannot extend multiple classes. Therefore, instead of extending Thread, you should implement Runnable so that you can still extend the class that you want functionality from.
  - To implement Runnable, override the run() method.
- Fibonacci.java → Waiting with join()
  - join() is used for waiting on a thread. The method is a blocking construct and will stop execution of the current thread to wait for the other thread to finish execution.
  - The join() method call must be surrounded in a try/catch block.
- Fibonacci2.java → Using ExecutorService
  - Think in terms of tasks, not threads. ExecutorService manages threads for you and helps you avoid the overhead associated with creating and running threads.
  - Faster because the system knows best how to manage its resources based on how many cores it has. It is also able to re-use the same thread for multiple tasks instead of throwing each one away and creating a new one.
  - Use submit() to add a task to the ExecutorService thread pool.
- Fibonacci3.java → Extend RecursiveTask
  - In a recursive setting, we will create many threads and run out of memory because threads have to wait for the recursive calls.

2.2.2 Asynchronous Execution

In asynchronous execution, you don't wait for a task to complete or do the work yourself. You wait for 'someone else' to do the subroutine work, and then continue doing something else. When you need the result from that work, then you wait on the task.

One way to do this in Java is to use the Future type. This type has a class method get(). When called, it blocks if the value is not yet set, and returns the value when done with executing the task.

If you want a return value from a thread, then be sure to implement Callable.

2.3 Amdahl's Law

Question Is there a fundamental limit on how much speed-up I can get in a program?

Solution Let p be the fraction of work that can be parallelized. Then $1 - p$ is the fraction that cannot be parallelized. Let n be the number of cores on the machine. Let T_s be the time taken on a sequential processor. Let T_p be the time taken on a multicore machine. Then:

$$T_p \geq (1-p)T_s + \frac{pT_s}{n}$$

$$speedup = \frac{T_s}{T_p} \leq \frac{1}{1-p+\frac{p}{n}}$$

2.4 Mutual Exclusion

Definition Critical Section - A section of the code that can cause race conditions if the code is interleaved.

An example of this could be the line of code $x = x + 1$; If two threads are both in the process of executing this line of code at the same time, problems could arise. Since this breaks down into a load, write, and store instruction, both threads could load the value 0 and add one, then then store that value. In this case, x would be the value 1 and not 2, which may be the expected value.

We want critical sections to be mutally exclusive. If any processor is in the critical section, then no others should be executing that section.

2.4.1 Peterson's Algorithm

This is an algorithm to implement the mutual exclusion construct. The key in this solution is that it is a 'polite' solution - each thread sets the turn variable to the other thread.

P0

```
wantCS[0] = true;
turn = 1;
while (wantCS[1] && turn == 1);
// CS
wantCS[0] = false;
```

P1

```
wantCS[1] = true;
turn = 0;
while (wantCS[0] && turn == 0);
// CS
wantCS[1] = false;
```

Question Does this satisfy Deadlock Freedom?

Proof A deadlock can happen in this code if both while loops are true at the same time.

$$DEADLOCK = wantCS[1] \;\&\&\; turn == 1 \;\&\&\; wantCS[0] \;\&\&\; turn == 0$$

But, we know that $p \wedge q \Rightarrow p$.

$$turn == 1 \wedge turn == 0$$

But, we know that the turn variable cannot be both 0 and 1. This gives us a contradiction, and thus this code is free of deadlock.

Question Does this satisfy Mutual Exclusion?

Informal Argument Assume both threads are in the critical section. This means that both excuted the assignment statement of the variable turn. Let turn = 1. This means that the assignment turn = 0 happened before turn = 1. But, P0 checks that turn = 1 and $wantCS[1]$ = true. This means that P0 could not have entered the critical section. This is a contradiction.

Dijkstra's Proof Let $trying[0]$ be true when the execution point is at the while statement. Let it become false when entering the critical section.

$$H(0) = wantCS[0] \wedge ((turn = 1) \vee ((turn = 0) \wedge trying[1]))$$
$$H(1) = wantCS[1] \wedge ((turn = 0) \vee ((turn = 1) \wedge trying[0]))$$

Based on the above definitions, when P0 gets to the while loop, $H(0) = true$. When P1 gets to the while loop, $H(1) = true$. Let us show that P0 cannot falsify $H(1)$. Then by symmetry, P1 cannot falsify $H(0)$.

Let us look at each part of the statement in turn.

- $wantCS[1]$ - P0 does not touch this variable, so it cannot make it false.
- $turn = 0$ - This value can be changed to $turn = 1$, but then the second part of the statement will become true. This is because when you change the value of turn, you have reached the while loop. This means that $trying[0] = true$ and $turn = 1$.
- $turn = 1 \wedge trying[0]$ - This cannot be falsified because it is equivalent to falsifying $wantCS[1] \wedge turn = 1 \wedge trying[0]$. The first two predicates in that statement are the same condition as the while loop. Due to the entry protocol, we cannot make $trying[0] = false$.

From here, we can assume that both P0 and P1 are in the critical section, and then manipulate the statements to show a contradiction such as $turn = 0 \wedge turn = 1$. This is left as a exercise.

References

[1] V.K. Garg Introduction to Multicore Computing

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 3: September 1

Lecturer: Vijay Garg *Scribe: Aditi Ranganath*

3.1 Introduction

In this lecture we will discuss various mutual exclusion protocols that work for n threads, where n is greater than 2. The scope of this lecture extends to the following algorithms :

1. Proof of Peterson's algorithm (Review)
2. Filter algorithm (Peterson-n algorithm)
3. Tournament algorithm
4. Bakery algorithm

3.2 Peterson's Algorithm (continued)

So far we have developed an informal and a formal proof (Dijktra's proof) for Peterson's two-thread algorithm and discussed the concepts of *Deadlock Freedom* and *Mutual Exclusion* for the same.

Thread 0 (P_0):	Thread 1 (P_1):
wantCS[0] = true;	wantCS[1] = true;
turn = 1;	turn = 0;
while (wantCS[1] && (turn==1)) ;	while (wantCS[0] && (turn==0)) ;
< *critical section* >	< *critical section* >
wantCS[0] = false;	wantCS[1] = false;

Table 3.1: Peterson's algorithm for two threads P_0 and P_1.

In case of *Deadlock Freedom*, combining the conditions in which both P_0 and P_1 are waiting leads to contradiction, hence proving that the algorithm is starvation-free or deadlock-free. That is:

$$(wantCS[1] \;\&\&\; (turn == 1)) \;\&\&\; (wantCS[0] \;\&\&\; (turn == 0))$$

On simplifying, we get (turn==1) && (turn==0) which cannot be true.

In case of *Mutual Exclusion*, we used auxiliary variables $trying[0]$ and $trying[1]$ to prove by Dijktra's method that P_1 cannot falsify predicate $H[0]$ set by P_0 and vice versa, where, $H[0]$ is defined as:

$$H[0] = wantCS[0] \;\&\&\; ((turn == 1)||((turn == 0)\&\&(trying[1])))$$

On simplifying, we again get the contradiction case where (turn==1) && (turn==0) which cannot be true. Thus, Peterson's algorithm for two threads is both *Deadlock Free* and *Mutually Exclusive* [1].

3.3 Filter Algorithm: Peterson-n Algorithm

We now try to extend Peterson's mutual exclusion protocol to work for $n(> 2)$ threads. For this, we keep the algorithm to be symmetric and instead of the semantic *turn*, we use the variable *last*. We expect this to work well as it is easier to know which process wrote into the shared variable at the end. Thus, for P_i processes, where $i \in \{0, 1, 2, .., N-1\}$, we have

```
wantCS[i] = true;
last = i;
while( (∃ j: j≠i: wantCS[j]) && (last==i) ) ;
< critical section >
wantCS[i] = false;
```

Now, lets examine if this is *Mutually Exclusive* : Consider three processes P_0, P_1 and P_2. If P_2 was the last to write into the shared variable, only P_2 waits. Both P_0 and P_1 can now enter the critical section. This is not good as there is no *Mutual Exclusion*. If n threads are at the gate at the same instance of time, only the last one is waiting while the remaining $(n-1)$ enter. In order to ensure only one thread enters the *critical section*, we have to repeat this process $(n-1)$ times. Thus, at each of the $(n-1)$ gates, we have one thread waiting which allows only one thread to enter the *critical section* thereby ensuring *Mutual Exclusion*. With this, the algorithm looks as follows:

```
int n;
int [n]gate;
int [n]last init 0;                                  // last[0] will not be used
for (int k=1 ; k<n ; k++) {                          //entry protocol
    gate[i]=k;                                       // P_i is at gate k now
    last[k]=i;                                       // P_i updates variable last for that gate
    for (int j=0 ; j<n ; j++) {
        while( (j≠i) && (gate[j]≥k) && (last[k]==i) ) ;
    }                                                //inner for − loop
}                                                    //outer for − loop
< critical section >
gate[i]=0;                                           //exit protocol
```

where, i is the process index and k is the gate index. Every process should go through $(n-1)$ gates to enter the *critical section*. Thus, Filter Algorithm can be visualized to be stacking of Peterson's algorithm on one another $(n-1)$ times with the following complexity:
Space complexity : $O(N)$
Time complexity : $O(N^2)$
On further analysis of the above algorithm, it can be shown that if process P_i is pausing at any point, other processes P_j, P_k, etc. can enter *critical section* overtaking P_i arbitrary number of times.

3.4 Tournament Algorithm

Another simple technique to extend the use of a two-thread mutual exclusion algorithm for n threads in using the Tournament Algorithm. In this case, each thread is progressing from the leaf to the root of the tree by participating in a two-thread mutual exclusion algorithm at every step. Thus, a thread has to pass through log_2(N) locks to enter the *critical section*.

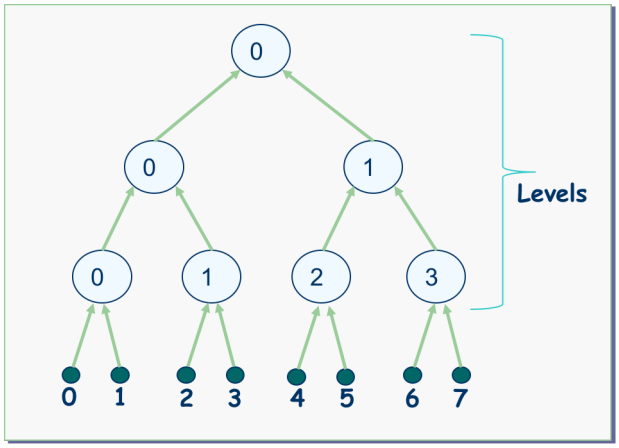


Figure 3.1: Tournament tree with multiple nodes and levels

3.5 Variable read-write Atomicity

A programming language can have three types of variables:

- SRSW : single-reader, single-writer variable
- MRSW : multi-reader, single-writer variable
- MRMW : multi-reader, multi-writer variable

Intuitively, in both SRSW and MRSW, the write order is deterministic. SRSW does not have any concurrency issues. MRSW also possesses sequential consistency but needs some checks for certain scenarios where concurrent reads can possibly return different values. However, in case of MRMW, the writes depend on the time stamps at which the request was made, and hence it requires more checks and locks.

In *Filter algorithm*, we use a MRMW variable *last* which requires some sort of atomicity and mutual exclusion for writes. Thus *Filter algorithm* assumes mutual exclusion for the multi-writer variable. This scenario can be avoided using the Lamport's Bakery Algorithm.

3.6 Bakery Algorithm

The algorithm was developed by Leslie Lamport with an analogy of a bakery with a numbering machine at its entrance so each customer is given a unique number. Numbers increase by one as customers enter the store. A global counter displays the number of the customer that is currently being served. All other customers must wait in a queue until the baker finishes serving the current customer and the next number is displayed. When the customer is done shopping and has disposed the number, the clerk increments the number, allowing the next customer to be served. That customer must draw another number from the numbering machine in order to shop again [2]. Extending this analogy, we have a scenario where every thread has a notion of its own number that can be written on only by itself but multiple threads can read from it. Thus, the algorithm uses a MRSW variable and eliminates the need for MRMW variable. The algorithm works as follows:

1. Every thread enters through a 'doorway' where it takes a number. This thread reads other threads'

numbers to ensure it gets the biggest number.

2. Ideally, only one thread should be at the doorway at a given time to get its number. However, it is possible that two threads enter the bakery at the same time and get the same number. To avoid concurrency issues, a unique ID is issued sequentially from 0 to $(n-1)$ to n threads that enter simultaneously.
3. Once a thread is inside the bakery, it waits till its number is the lowest in the bakery. The lowest thread enters the critical section.
4. If multiple threads have the same number, the one with the lowest unique ID enters the critical section.

```
boolean [ ]choosing;                                                    //init false
int [ ]number;                                                          //init 0
choosing[i]=true;                                                       //Step 1
int t=max(number[0],.....,number[n-1]);
number[i]=t+1;
choosing[i]=false;
for(int j=0; j<n; j++) {                                                //Step 2
  while(choosing[j]) ;                                                  //avoid conflict
  while((number[j]≠0)&&(number[j]<number[i])||(number[j]==number[i]&&(j<i)));
}                                                                       //end of for loop
< critical section >
number[i]=0;
```

Proof:
If P_i is in critical section and P_k (k≠i) has already chosen its number, then, $(number[i], i) < (number[k], k)$
Let t be the time whrn P_i checked choosing [k] and found it false. We have the following two cases:

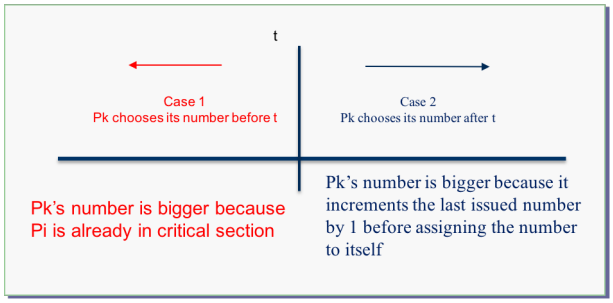


Figure 3.2: Two cases in Bakery algorithm

Thus, no matter when the number is chosen, the thread in the *critical section* is the one with the smallest number.

References

[1] V.K. Garg Introduction to Multicore Computing

[2] https://en.wikipedia.org/wiki/Lamport

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 3: September 1

Lecturer: Vijay Garg *Scribe: Travis Brannen*

3.1 Introduction

This class centered around analysis of various methods to achieve mutual exclusion in a concurrent environment. First, we review the proof of Peterson's Algorithm from last time. Next, we extend Peterson's so that it can be used for more than two processes ultimately arriving a tthe Filter Algorithm. We then briefly cover another extension of Peterson's algorithm, the so called Tournament. We briefly discuss the issue of multiwriter variables. Finally, we discuss another algorithm for concurrent mutual exclusion, the Bakery algorithm.

3.2 Peterson's Algorithm

Previously we developed Peterson's algorithm, and java code for the algorithm can be found in [1]. We also added, for the purpose of our proof, the boolean array *trying*. Here is PetersonAlgorithm.java with the variable *trying*:

```
class PetersonAlgorithm implements Lock {
    boolean wantCS[] = {false, false};
    boolean trying[] = {false, false};
    int turn = 1;
    public void requestCS(int i) {
        int j = 1 - i;
        trying[i] = true;
        wantCS[i] = true;
        turn = j;
        while (wantCS[j] && (turn == j)) ;
        trying[i] = false;
        // process i enters critical section after returning from this method
    }
    public void releaseCS(int i) {
        // process i leaves critical section upon setting wantCS[i] to false
        wantCS[i] = false;
    }
}
```

We review the proof for mutual exclusion that takes advantage of *trying*:

Consider predicate H(0), where:

$$H(0) \equiv wantCS[0] \wedge [(turn == 1) \vee ((turn == 0) \wedge trying[1])]$$

$$H(1) \equiv wantCS[1] \wedge [(turn == 0) \vee ((turn == 1) \wedge trying[0])]$$

P_0 makes $H(0)$ true just before entering the while loop by setting $turn$=1. P_1 makes $H(1)$ true just before entering the while loop by setting $turn$=0.

P_1 cannot falsify $H(0)$.

- Only P_0 can change the value of $wantCS[0]$.
- Additionally,
$$H(0) \Rightarrow turn == 1 \vee (turn == 0 \wedge (trying[1]))$$
Moving line by line through $requestCS(1)$, one can see that although P_1 can change $trying[1]$ and $turn$, $turn$ is only equal to 0 while $trying[1]$ is true. From symmetry, P_1 also cannot falsify $H(0)$.

Now, we will use the above in our proof of mutual exclusion by contradiction. First we suppose that both P_0 and P_1 are in their critical section, a violation of mutual exclusion. For this to be true, $trying$=$\{false, false\}$ since both processes have returned from $requestCS$. Also, $H(0)$ and $H(1)$ are true since both processes made their corresponding predicate true before entering the while loop and the predicates were not made false by any process since then. Therefore,

$$BothInCriticalSection \equiv \neg trying[0] \wedge H(0) \wedge \neg trying[1] \wedge H(1) \Rightarrow turn == 0 \wedge turn == 1 \Rightarrow false$$

/sectionFilter Algorithms Now we try to extend Peterson's Algorithm so that it can be used for N concurrent processes. To do so, we get rid of the *turn* variable and replace it with a variable called *last*. This is to eliminate the need for processes to explicitly reference other processes in their code.

Pseudocode for PetersonN base algorithm:

```
wantCS[i] = true
last = i
while( (exists j such that j!=i and wantCS[j]) and last==i )
***CRITICAL SECTION***
wantCS[i] = false
```

This is very similar to Peterson's algorithm, but close analysis reveals that it will allow N-1 processes to enter the critical section. However, running N-1 of these "gates" in series allows us to go from N processes, to N-1, to N-2, and so on until only one process exits the filter to enter the critical section at a time. This gives us an algorithm to enforce mutual exclusion among N processes such that only 1 process enters the critical section at a time. See PetersonN.java for an implementation of this algorithm[1].

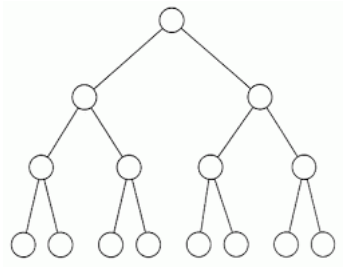
There are two new variables used in this algorithm. *gate* is an array with N integers, with the ith position representing the gate P_i is currently waiting behind. *last* is an array of integers with the value for last for each gate. Since we only need N-1 gates, the 0 position of *last* will be unused. Also, $gate$[i]=0 indicates that P_i is not currently attempting to access the critical section.

This algorithm may also be modified to allow any number of processes between N-1 and 1 into the critical section, for resources that may be shared between a limited number of users.

Space Complexity: O(N) Time Complexity: O(N^2)

3.3 Tournament

The nested for loops in PetersonN.java mean that it is not very temporally efficient. Using a tournament structure as shown below would take less time. Each process would enter a Peterson's Algorithm lock competing with one other process at the bottom. The winner would compete with the winner of an adjacent contest and so on. After passing through log_2(N) such locks, one process would enter the critical section at a time.



3.4 Multwriter Variables

SRSW - Single reader single writer
MRSW - Multireader single writer
MRMW - Multireader Multiwriter

SRSW does not pose a concurrency problem. MRSW allows for sequential consistency but locks are required for strict consistency. MRMW is the most difficult to implement since without locks even sequential consistency cannot be relied on.

3.5 Bakery

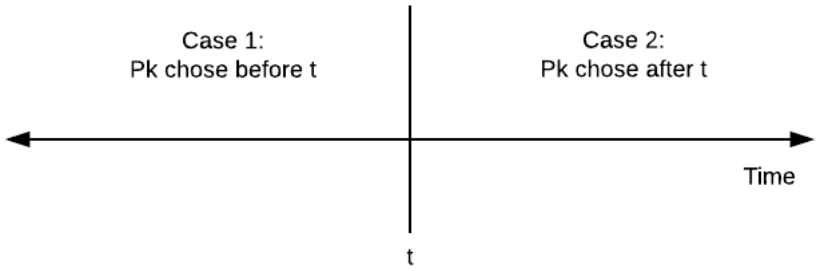
The Bakery Algorithm is another method of locking a shared resource so that only 1 of N processes may use it at once. The basic idea is that there are two steps to acquiring the lock. First, you come in through the "doorway" of the bakery and take a number. This number should be one higher than everyone who is already inside the bakery. Second, you wait until your number is the lowest number of anyone inside the bakery. Because of concurrency, it is possible that two processes will enter the bakery simultaneously and get the same number. Because of this, sequential, persistent process IDs from 0 to N-1 are issued to each process and in the case where two processes have the same number the one with the lower process ID enters the critical section first.

A full java implementation is given in Bakery.java[1]. There are two variables of interest *choosing* and *number*. P_i sets $choosing[i]$ to true when it enters the doorway and to false once it has a *number*. The *choosing* variable allows a process to ensure that any other processes that were choosing while it was choosing have finished by the time it attempts to enter the critical section. $number[i]$ is set by P_i in the doorway to be equal to $max(number) + 1$ and used to determine who should get in to the critical section. *number* is initialized to an array of all zeros because zero is used as a flag indicating that a process is not attempting to enter the critical section. Also for this reason, $number[i]$ is set to 0 after P_i exits the critical section.

Note that in the for loop checking if a process has the smallest number you need not explicitly take care of the case where j=i since the while condition will always return false in this case.

Check for Mutual Exclusion
If P_i is in the critical section and P_k (s.t. k!=i) has already chosen its number, then $(number[i], i) < (number[k], k)$. Then, assume P_i and P_k are in the critical section. This is a contradiction by the following:

t=time when P_i checked $choosing[k]$ and found it false.



In both Case 1 and Case 2, $(number[i], i) < (number[k], k)$.

3.6 Final Thoughts

In Java, even with locking algorithms you can still have mutual exclusion. This is because by default sequential consistency is not enforced. Multiple copies of variables are stored by main memory and the caches for each processor. You can use the volatile keyword for variables and the synchronized keyword for functions to enforce sequential consistency. The atomic keyword for variables can also be used to implement test and set.

References

[1] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/tree/master/chapter1-threads

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 4: September 6

Lecturer: Josep Torrellas *Scribe: Brad VanderWilp*

4.1 Introduction

As technology advances, we are able to build bigger and better computers. More, smaller transistors, larger cloud cores, and 3D stacked chips allow us more processing power is computers. The need to utilize this processing power has pushed research towards many-core computers. However, difficulties arise as power consumption increases and the rate of electrical improvements slows. Thus, research prioritizes energy efficiency, fast communication, and ease of programming as we explore the possibilities of parallelism.

4.2 Memory Fences

One way to improve synchronization is through the use of fences. "Fence" is a primitive used by both programmers and compilers to prevent the reordering of memory accesses. For example, in the sequence $write - x, fence, read - y$, the effects on the read on y cannot be observed by any other process until after the previous write has retired from the pipeline and the write buffer (WB) has been drained.

The following illustrates the effectiveness of fences. Consider the following code:

Process1 $A0 : x = 1; fence; A1 : t0 = y$

Process2 $B0 : y = 1; fence; A1 : t1 = x$

Here, fences prevent the order of retirement from being $A1, B0, B1, A0$. This is a possibility because of the time memory can take to propagate. This shows fences ability to preserve Sequential Consistency (SC).

4.2.1 Work-stealing Algorithm

The order between accesses enforced by fences can be used in work-stealing algorithm. Here, several worker processes are given a queue of tasks, which they execute from the tail-end. If a worker completes all of its tasks, it proceeds to steal a task from the head of a different worker's queue. To avoid interfering with each others' task lists, fences can be where workers and thieves perform their check whether the head of a queue is also the tail. A fence is placed between the thief's *get* of the head and check of the tail, and fence is placed between the worker's *get* of the tail and check of the head

4.2.2 Past implementations

A simple implementation for fences would be to stall all memory operations at a fence until other instructions have retired. This, however, leads to a lot of time wasted waiting. Another option would be to allow data to be loaded speculatively after a fence. If a read after a fence is observed through the cache of another

process, the read is squashed. This implementation also prevents reads from retiring until after the WB is drained. Still, there is a need for yet cheaper fence implementations

4.2.3 WeeFence

WeeFence (WFence) is a proposed type of fence that is skipped and allows statements to execute even before the WB drains. This prevents write misses from piling up before a fence. The only stall that needs to occur is right before a read that would violate SC. This is detected by using a global pending set (PS) table. Before a fence is passed, statements are pushed onto the table. If an instruction occurs after a fence that would violate SC based on some statement in the table, a stall is issues.

This method is successful partially because cycles that break SC are generally uncommon. Additionally, WFence is convenient because no compiler support is required. However, maintaining this global table is expensive. Without a global state to order the PS, deadlock could occur as processors stall for each others' fenced statements. But, if at least one processor stalls before the fence, the deadlock is broken

4.2.4 Asymmetric fences

The asymmetric implementation of WFences uses 1 conventional "strong" fence and N-1 "weak", WFence-type fences for a given conflict cycle of N processors. This eliminates the cost of maintaining a global table at the cost of keeping a slower, strong fence. This can be optimized by strategically placing strong fences at sections of code that occur less frequently, such as at initialization rather than in the loop that uses a variable. In the previously mentioned work-stealing algorithm, for example, the strong fence would be used in the thief code because it is the less common case.

Overall, the use of WFences has been shown to be effective. Tests have revealed WFence to eliminate ninety percent of fence stall time and reduce overhead from forty percent to two percent when compared to normal fences.

4.3 Breaking Serialization

Another major objective for parallelization is to alleviate the bottleneck caused by multiple functions accessing the same variable. One common answer to this problem is to use Compare-and-Set (CAS) algorithms. These operate by obtaining the value, changing it, and then performing an atomic operation to set the value only if the variable has not been altered by a different process. However, this leads to processors wasting time, as only one sets the variable at a time.

4.3.1 CASPAR

A modern implementation attempts to eliminate the time that a process must wait to change and use a shared variable with CAS. First, load requests for the old variable value are queued. This allows only one CAS operation to be performed at a time for completely serial execution.

The next step is an eager forwarding method. The cache containing the variable is forwarded between processes. This prevents stalling as execution occurs in parallel. The new value of the variable is later compared to the actual value.

4.4 Scalable Concurrent Priority Queues

One other goal for improving concurrency is improving implementations of priority queue. Some algorithms work best if certain parallel tasks are executed before others, necessitating an efficient priority queue. However, significant overhead can arise when many processes attempt to simultaneously access the highest priority item. One design involves relaxing the priority and giving different processes sub-queues. However, this potentially leaves the optimum of the sub-queue to be far from the global optimum.

Instead, one implementation uses a mix of hardware and software to organize sub-queues. A hardware register tracks the head of all sub-queues, and all items are enqueued to the local sub-queue. Upon dequeueing an item, all subqueues are visited and sorted. Then, one of the items in the highest range is chosen. By not choosing the highest priority item every time, conflicts between processes are avoided. Such implementations have been found to produce 2-5x speedup.

4.5 Other Projects

Optimizing the usage of growing processor technology is a multidisciplinary task which include many other projects than the ones discussed here. For example, WiSync is a concept which includes antennae within microchips. This improves communication by create a wireless network within a wired network. QuickRec is a system of modifying the cache hierarchy to record the non-deterministic events that occur during execution of a program. This provides a useful debugging tool for parallel systems by using these logs to replicate an execution of a program. ScalCore is a design that generates multiple voltage domains for memory and logic to increase throughput. Since logical data can be transmitted with lower voltage, faster memory accesses can be sent simultaneously, improving speed and conserving energy. Finally, research in control theory has been working to improve controllers for power performance, temperature, and other parameters in such a way as to prevent inefficient competition.

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 4: September 6

Lecturer: Josep Torrella, Dept of Computer Science, UIUC *Scribe: Wenwen Zhang*

4.1 Preface

With the advancement in integrated circuits, we have accelerated progress in transistor integration. We can create new advancements such as large multicore processors for data centers and cloud, and 3D stacked chips. With these new advancements, we also encounter a huge power wall, in such supercomputers would take a lot of power input, even in idle states. Power consumption advancements are relatively slow in comparison to technology improvements, which makes computer architecture improvement high priority. In order to resolve this problem, computer architects are focusing on improving energy efficiency and faster communication/synchronization between computers. In this seminar, Dr. Josep Torrella focuses mostly on new technologies to reduce the cost of basic primitive for parallelism and other challenges in energy and programmability.

4.2 Goal

In order to make synchronization less expensive, we can improve in three different areas.
- make memory fences less costly/free
- break serialization in lock-free synchronisation
- create scalable concurrent priority queues

4.3 Memory Fences

Fir the first suggestion to make synchronization inexpensive, we can improve memory fences. Memory fence is a primitive for parallelism such that it can prevent the computer and hardware from reordering memory accesses inserted by programmers or compilers. It does so by forcing both read and write instructions to be finished and retired from the pipeline before the next instruction. Fences can be achieved by programmers inserting codes with fine-grain sharing then compiler insert fence after the access and not reorder code.

Example for memory fences would be as follows:
Executing the following code:

```
1. A0: x = 1;
2. B0: y = 1;
3. A1: t0 = y;
4. B1: t1 = x;
```

Incorrect execution without fence could have possible execution order such as A1, B0, B1, A0, which would

result in wrong values for t0 and t1, a demonstration of Sequential Consistency (SC) Violation. If we can insert fences after A0 and B0 to force execution of A0 and B0 before A1 and B1.

4.4 WeeFence (WFence)

Modern implementation of fences at hardware scale could be performing speculations on read instructions after fences. In this case, if no processor observes it, no problem would be caused; if coherence transaction received, this rd would be squashed and retired. However, with speculation, the reads are still not able to retire until the WB is drained. This could still make fences costly.

WeeFence is then introduced with a goal to eliminate any stalls in the pipeline. With WeeFence, post-fence read can retire before the pre-fence writes have drained and ?skip? the fence. WeeFence would only stall when a read that violates SC and detects SC by using a global Pending Set (PS) table to store instructions that passes fences.

4.5 Asymmetric Fences

WeeFence has a pretty good improvement in preventing stalls since Cycles that break SC rarely happens. However, a global PS table is rather expensive to maintain. Without the global PS table, deadlocks can occur when all the processors stall themselves after encountering other fenced instructions. Only exception is when one process stall right before the fence. Thus, creating this exception could be the key to resolve the problem.

We can create this exception by having a Strong fence and N-1 Weak fences for a given conflict cycle of N processors. Strong fences are the conventional fences and weak fences are fences like WFence. This can potentially reduce the cost of a global PS table to just a conventional fence. If the strong fence is placed somewhere in the code that less likely to occur, the cost of this strong fence can be further reduced. With implementation of WFence, 90% of fence stall time were eliminated and the overhead of supporting SC went down from 40% to 2%.

4.6 Breaking Serialization

Another bottleneck for increasing power consumption is that many processors have to synchronize on the same variable while executing multiple functions. This case occurs in many areas, such as in OS, databases, language runtimes, and memory allocators. One possible and common solution is to use Compare-and-Swap (CAS), a lock-free synchronization method that uses atomic instructions to manipulate data. However, this could waste a lot of time on processors since only one can set the variable at a time.

In order to reduce the time that other processors have to wait on setting the shared variable using CAS, we could use CASPAR. CASPAR can be done in two steps: first, put the old requests in hardware in queue to allow only one CAS operation at a time; then an eager forwarding method that caches the variable to other processes to prevent stalling. New values after the CAS operation then can be compared with the cache value to ensure accuracy.

4.7 Scalable Concurrent Priority Queues

Since some algorithms work best if some parallel tasks executed before others, we could potentially improve their executing priority to make synchronizations less expensive. However, traditional priority queue always dequeue from head and CASPAR won?t work since it needs to insert nodes like a stack. If all dequeues happen on the highest priority node, it would lead to contention at the head node. In addition, the highest priority node may not be the global best to be dequeued.

An implementation of the new priority queue would be a mix of hardware and software to create and organize sub-queues in different processes to decrease the necessary contention to the high priority node. Hardware collects the nodes at the head of all the queues and sort them to provide one of the top nodes chosen from the highest range. This process resolves the contention to the highest priority item by choosing other items with relatively high priority often. In addition, this implementation reduces 64-threaded application execution time by 2 to 5 times on average.

4.8 Conclusion

In the end, Dr. Torrella briefly talked about other project that he has worked on in the field of processor technology. For example, WiSync, On-Chip Wireless communication concept to include antenna within microchips that would bring wireless network within wired network to improve communication; QuickRec, a prototype of record and Replay (RnR); ScalCore, a core for voltage scalability. With the techniques mentioned above, Dr. Torrella believe that there are a lot of room to innovate in computer architecture field at this time as many exciting interdisciplinary venues of research happens to increase performance, energy-efficiency & programmability.

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 5: September 8

Lecturer: Vijay Garg *Scribe: Nishanth Shanmugham*

5.1 Topics

The topics covered in this lecture are:

- Semaphores
  - Why / what / motivation
  - Usage
  - Implementation
  - Java code on GitHub
    - BinarySemaphore
    - CountingSemaphore
- Consumer-producer problem
- Reader-writer problem
- Dining philosophers problem

5.2 Semaphores

Semaphores provide a means to achieve mutual exclusion.

5.2.1 Why / what / motivation

Think back to Peterson's algorithm. We performed a busy-wait:

```
while (wantCS[1] && (turn == 1)) { no_op(); }
```

This wait consumes the CPU while waiting for the condition to become false. So core execution time is wasted. This busy-wait can be avoided with help from the operating system.

So how does the OS schedule threads?

- The scheduler puts RUNNABLE threads to RUNNING queue.
- If a RUNNING thread issues a blocking call (for example, I/O) then it is put in BLOCKED queue.
- Once the I/O operation returns, the blocked thread can be scheduled for running again by putting in the RUNNABLE queue.

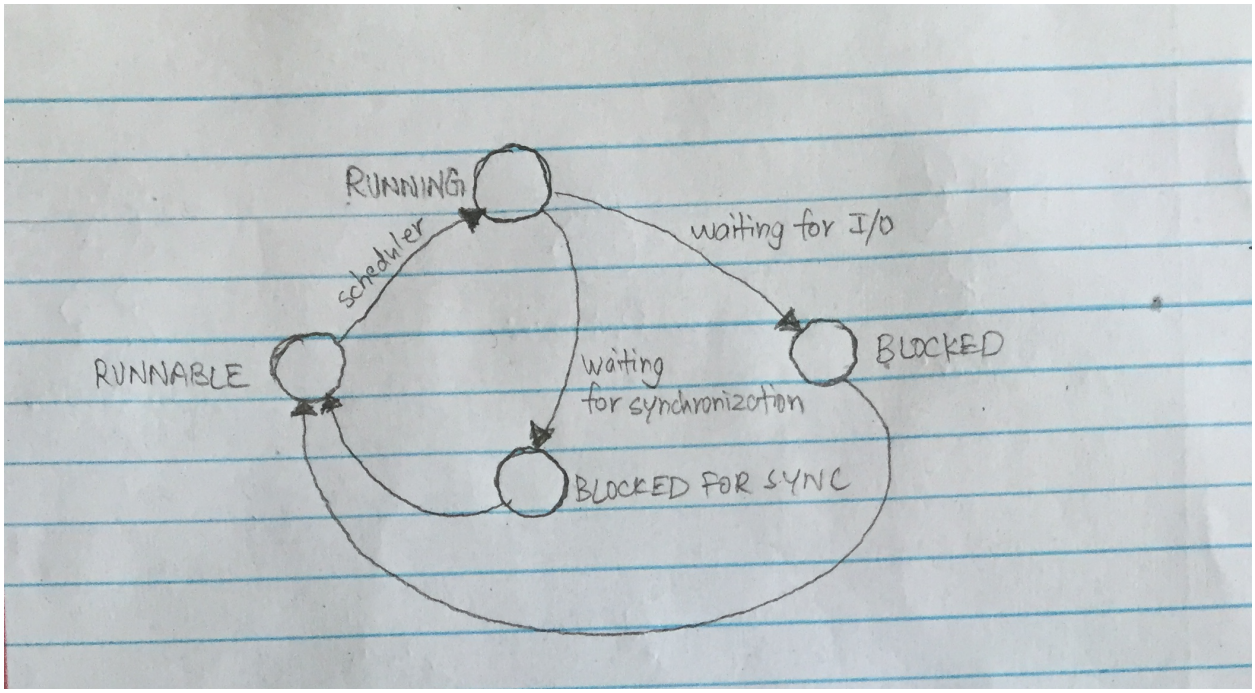


Figure 5.1: OS scheduling queues

- Similarly, instead of running no_op()'s, the OS should be able to take it out of the RUNNING queue and put into the BLOCKED queue waiting for synchronization.

5.2.2 Usage

Semaphores were introduced by Dijkstra in *The Operating System*. A semaphore has two operations defined on it:

- P(), or acquire()
- V(), or release()

That is, if `s` is a semaphore then `s.P()` and `s.V()` are the two possible calls.

5.2.3 Visualization

A simple way to visualize to semaphore is as a bottle with a marble inside it. The marble could either be present inside the bottle or not inside it the bottle.

If a thread wishes to enter a critical section, it should first obtain the marble before it proceeds to enter the CS. If the marble is present, it removes the marble and proceeds to enter the CS. If the marble is not present, it waits until the marble is present to try again.

The `value` variable in the implementation below corresponds to the marble.

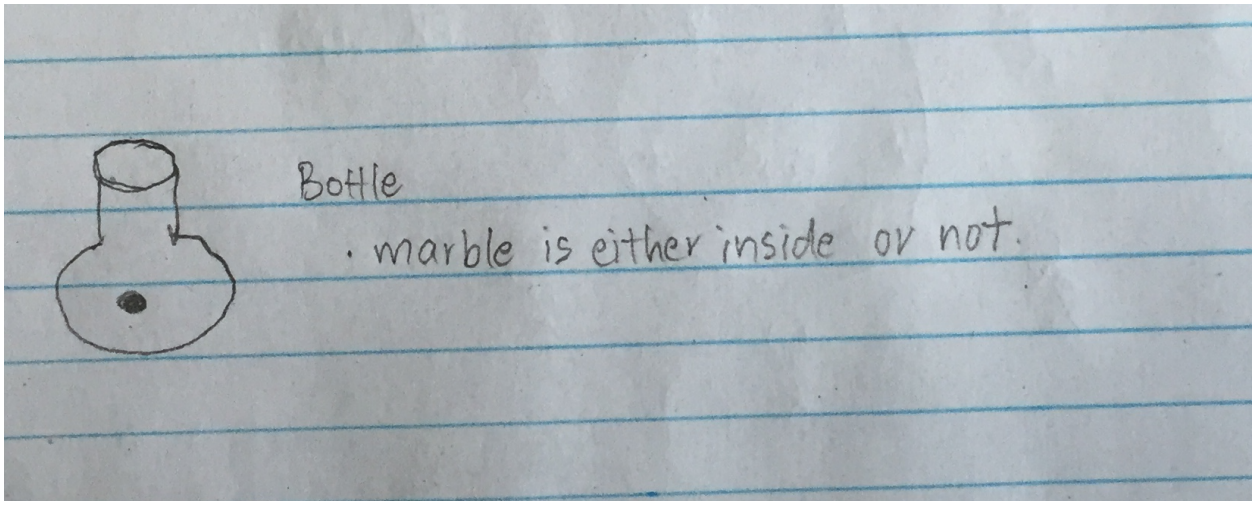


Figure 5.2: Semaphore

5.2.4 Implementation

A semaphore has one field:

```
value:  boolean, initially true
```

The implementation for `P()` is below. Note that both lines would need to execute atomically.

```
while (!value) { wait(); }
value = false;
```

The implementation for `V()` is:

```
value = true;
```

5.2.5 Discussion

But isn't it weird that we have to perform P() atomically? Essentially, it appears like we need to use mutual exclusion to build mutual exclusion.

We will have to use busy-wait to achieve atomicity of P().

- For programmer's CS, we will use semaphore.
- For semaphore's CS, we will use busy-wait.

This is an okay compromise. The reason to not use busy-wait for programmer's CS are that programmer's CS are generally longer and may contain bugs in the implementation. On the other hand, semaphore's CS is known to be small. Thus using a semaphore saves system resources.

5.2.5.1 notify

Sometimes, it might be necessary to wake up one of the sleeping threads once the current threads completes it work in the CS. For this, we use `notify()`. The `notify()` call does not guarantee the order in which threads are woken up—it simply wakes up any one thread.

To guarantee order, Java provides a fairness parameter. But generally, it is good practice to not expect order guarantees in concurrent programs that do not require ordering.

5.2.5.2 Monitors

Semaphores might hinder code readability since semaphores are used for both *mutual exclusion* and *conditional execution*. In addition:

- The order of `P()` calls is important to avoid deadlocks.
- `P()` calls should be paired with corresponding `V()` calls to avoid deadlocks.

Monitors provide a suitable mechanism to handle these, but are explored in a future lecture.

5.2.6 Java code

5.2.6.1 BinarySemaphore

The `BinarySemaphore` class on GitHub implements the kind of semaphore described above.

Of note, the Util.myWait call on line 8 puts the current thread in the blocked queue. Also, the synchronized keyword in the method signatures guarantee that only one thread is executing the method at any given time. This provides the required atomicity.

5.2.6.2 CoutingSemaphore

`CountingSemaphore` allows multiple threads to be in the CS at a given time. The value field is now an int instead of a boolean.

5.3 Producer-consumer problem

5.3.1 Description

There is a producer thread and a consumer thread. The producer deposits to shared buffer, and the consumer reads from the shared buffer. The constraints are:

- Mutual exclusion (no concurrent read/write to shared buffer)
- Conditional synchronization. The consumer has to wait for buffer to be non-empty before read; the producer has to wait for buffer to be non-full before depositing.

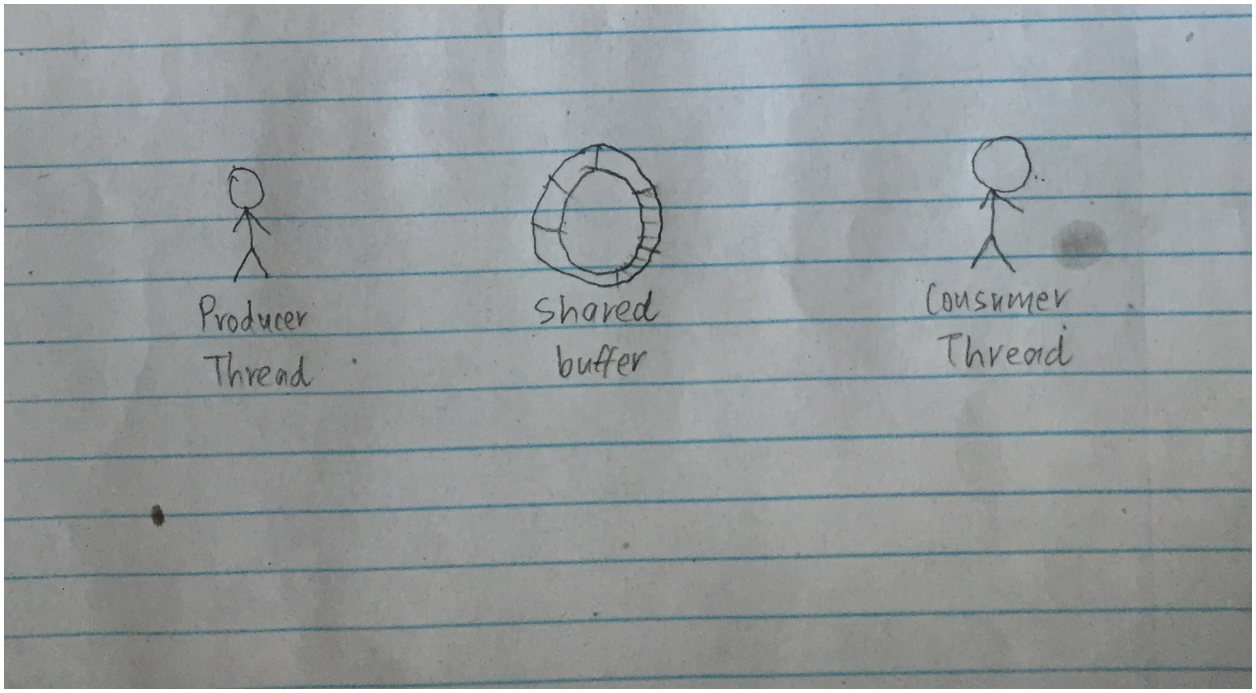


Figure 5.3: Producer-consumer

5.3.2 Implementation

`BoundedBuffer` class on GitHub contains a solution to this problem. The noteworthy observations from the code are:

- The mutex is a BinarySemaphore. It is a common convetion to use a BinarySemaphore for mutual exclusion.
- The conditional synchronization is provided by the CountingSemaphores, namely `isEmpty` and `isFull`.
- The other thread is woken up / notified by the `V()` calls on lines 15 and and 24.

Question: Is it okay to change the order of the calls on lines 10 and 11? No, here's an example execution order that results in deadlock if the order was changed.

- The buffer is full. The producer grabs the mutex. Then the producer waits for the buffer to be not full so that it can deposit.
- The condition that the producer is waiting for will never be fulfilled.
- This is because the consumer will not be able to consume from the buffer, because the mutex is held by the producer.

In the correct order of code on GitHub, this will not happen because consumer and producer only grab the mutex if their entry conditions are already satisfied.

5.4 Reader-writer problem

5.4.1 Description

The problem aims to establish a protocol for multiple readers and writers to safely access a shared database. The constraints are:

- No read-write conflict.
- No write-write conflict.
- Read-read conflict is allowed.
- There is no need to guarantee starvation freedom for writers. It is okay for readers to read repeatedly.

5.4.2 Implementation

`ReaderWriter` class on GitHub contains a solution to this problem.

The `startWrite` and `startRead` methods are the entry protocol for writers and readers respectively. Similary, `endWrite` and `endRead` are the exit protocols for writers and readers respectively.

`startWrite` and `endWrite` are simple. Writers simply need to acquire `wlock` before they can write and release it when done writing. The acquiring of the BinarySemaphore wlock ensures that only one writer is writing at any given time.

Readers on the other hand need to ensure that no writers exist before they can read. The first reader to enter has to wait until all writers leave. When the first reader enters it grabs the wlock (line 8) ensuring no writers can enter until the last exiting readers release wlock (line 14). Readers after the first reader simply need to enter and exit. The last reader to exit has to release wlock as described before.

To track the number of readers, the numReaders variable is used. It is incremented on entry and decemented on exit. It is guarded by mutex to prevent concurrent modification of the variable by readers.

5.5 Dining philosophers problem

5.5.1 Description

n philosophers sit around a circular dining table. There are n forks placed in between the philosophers. To eat, a philosopher requires two forks—the one of her left and the one of her right.

Philosophers switch between these states in order: *thinking* to *hungry*, *hungry* to *eating*, and *eating* back to *thinking*.

The goal is to design a protocol with the following constraint: Two neighboring philosophers cannot eat at the same time.

Philosophers should use an `acquire` call to acquire forks and a `release` call when done eating.

The Philosopher class on GitHub represents a philosopher. DiningPhilosopher provides a simple (but incorrect) implementation of Resource.

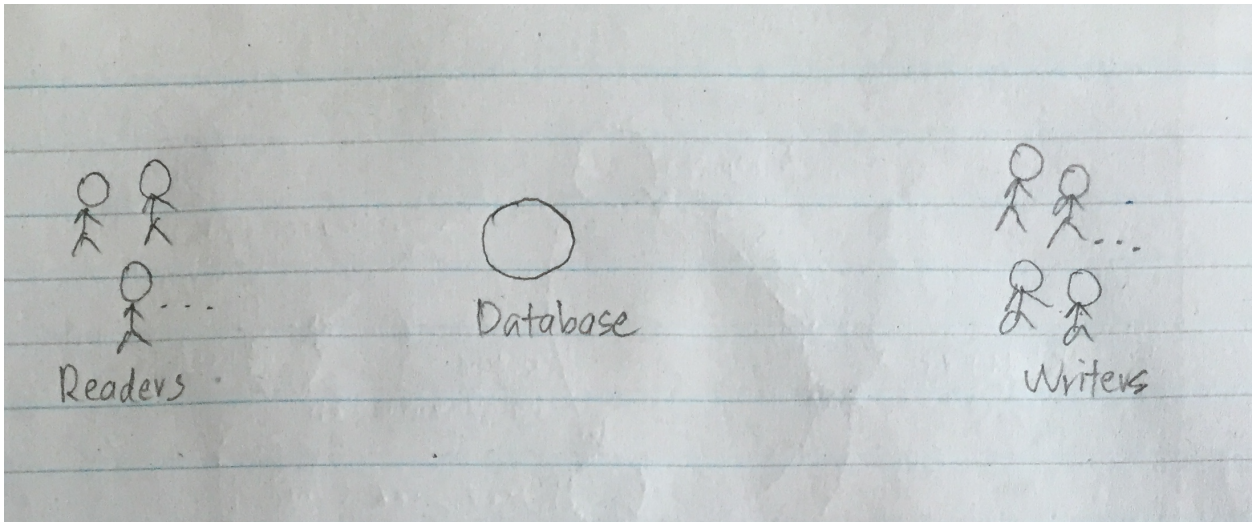


Figure 5.4: Reader-writer

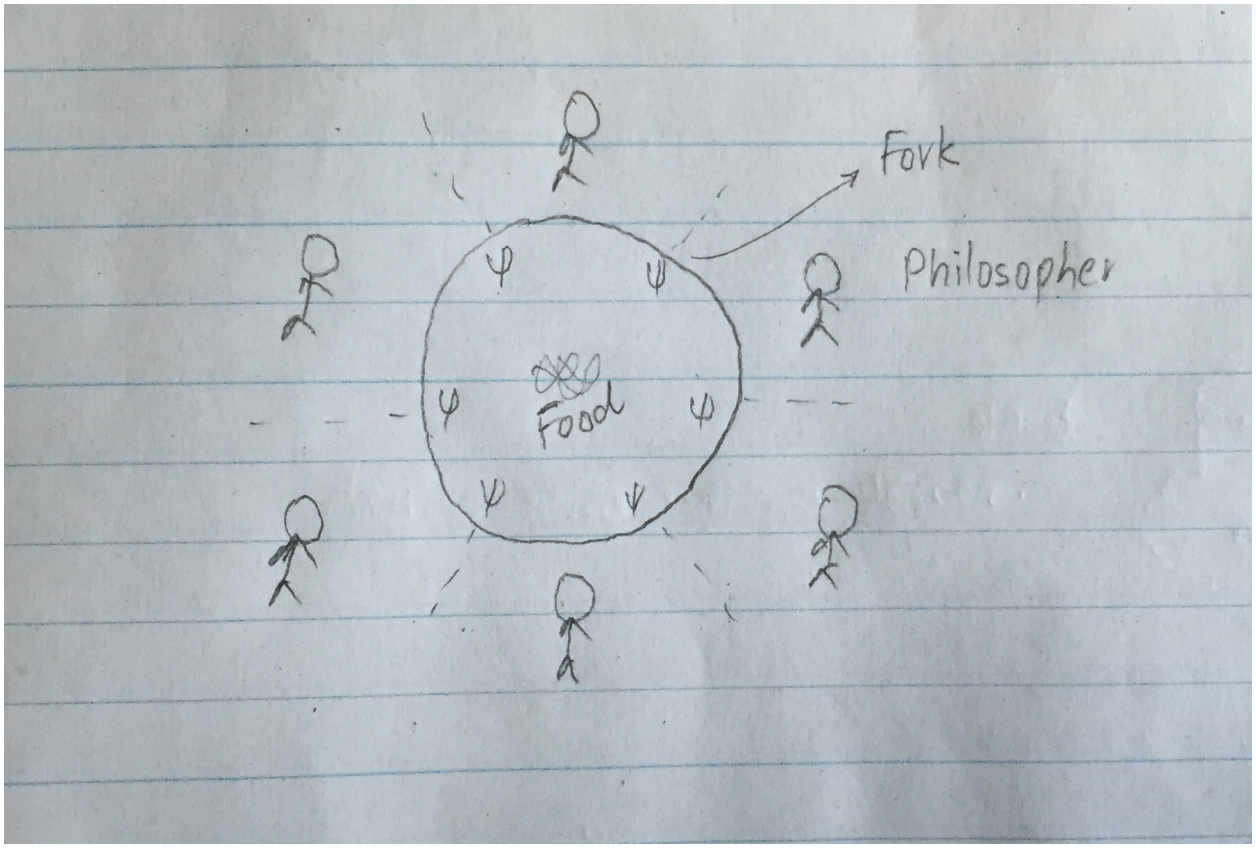


Figure 5.5: Dining philosophers

It is incorrect because it can result in a deadlock. In the acquire method, philosopher's first grab the left fork then the right fork. In a circular dining table, it could happen that the philosophers all grab their left fork first. When the philosophers now try to obtain the right fork, they will fail to do so because it has been obtained by their neighbor to the right (who grabbed it initially as her left fork).

This is problem arises because of symmetry.

5.5.1.1 Fixes

Possible fixes to the DiningPhilosopher class are listed below.

- Breaking symmetry: exactly one philosopher will be instructed to grab her right fork before the left. This will prevent the deadlock.
- Restrict number of philosophers that can be eating at any given time to less than n: For instance, there could be a precursor step of standing up before acquiring a fork. At that point, we can restrict that at most less than n philosophers can be standing at any instant.

References

[GitHub] Multicore source code, *https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore*

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 5: September 8

Lecturer: Vijay Garg *Scribe: Wenbo Xu*

5.1 Semaphore

5.1.1 Motivation

Recall from the Peterson's Algorithm:

```
while (wantCS[j] && (turn == i)){};
```

While threads are blocked to enter the critical section, they consistently check on the condition and running no-ops. This called busy wait and consume a lots of computing resources. With the help of OS, we can save the resource from busy wait.
The OS maintain two queues:

```
Running queue: contains all threads are runnable.
Blocked queue: contains all threads are blocked on other operations.
```

OS scheduler is responsible to put blocked thread from the running queue into blocked queue. And put unblocked queue from blocked queue to running queue. In this way, any threads are blocked would not doing busy waits and the computing resource can be used elsewhere.

5.1.2 Implementation

The concept of Semaphore is introduced by Dijkstra. It has two operations, and one variable

```
Value, indicate whether the CS can be entered
P(), acquire operation
V(), release operation
```

For P() operation, it first check Value to check if the lock is available. If yes, acquire the lock by setting it to false.

```
P(){
        while(Value){}; //no ops
        Value = false;
}
```

For V() operation, it release the lock by simply setting Value to true.

```
V(){
        Value = true;
}
```

The P() operation has to be atomic. And busy wait is used to implement this CS. So, the program use semaphore for its CS, and semaphore use busy wait for its CS. The program's CS is unknown length and is expected to be long, where we want to make is blocked to save computing resource.And Semaphore's CS is small, it is ok to do busy wait.
When a thread finished its CS, a method notify is used to weak up one which is blocked by the same semaphore.

5.1.3 Java Implementation

Here shows the Java code for BinarySemaphore

```
public class BinarySemaphore {
        boolean value;
        public BinarySemaphore(boolean initValue) {
        value = initValue;
        }
        public synchronized void P() {
        while (value == false)
        Util.myWait(this);// in the queue of blocked processes
        value = false;
        }
        public synchronized void V() {
        value = true;
        notify();
        }
}
```

The boolean value indicates whether the semaphore is ready or not. You can initial the semaphore to be ready or not. Unitl.myWait(this) in P() operation puts the thread into blocked queue. The V() operation calls notify() to weak up one thread in the blocked queue. The key work synchronized guarantees the operation is atomic.
The following code shows a CountingSemaphore:

```
public class CountingSemaphore {
        int value;
        public CountingSemaphore(int initValue) {
        value = initValue;
        }
        public synchronized void P() {
        while (value == 0) Util.myWait(this);
        value--;

        }
        public synchronized void V() {
        value++;
        notify();
        }
}
```

In this semaphore, the boolean value is replaced by a integer value. In this way, multiple threads are allowed

to be in CS at the same time.

5.1.4 Producer & Consumer

We have a producer thread and a consumer thread. The two threads share a circular buffer. This type of question has the following constraints:

· **mutual exclusion** shared resources

· **conditional synchronization** consumer must wait for the buffer to become not empty & producer must wait for the buffer to become not full

The following Java class shows how the bounded buffer is implemented.

```
class BoundedBuffer {
        final int size = 10;
        Object[] buffer = new Object[size];
        int inBuf = 0, outBuf = 0;
        BinarySemaphore mutex = new BinarySemaphore(true);
        CountingSemaphore isEmpty = new CountingSemaphore(0);
        CountingSemaphore isFull = new CountingSemaphore(size);

        public void deposit(Object value) {
                isFull.P();// wait if buffer is full
                mutex.P(); // ensures mutual exclusion
                buffer[inBuf] = value; // update the buffer
                inBuf = (inBuf + 1) % size;
                mutex.V();
                isEmpty.V();  // notify any waiting consumer
        }
        public Object fetch() {
                Object value;
                isEmpty.P(); // wait if buffer is empty
                mutex.P();  // ensures mutual exclusion
                value = buffer[outBuf]; //read from buffer
                outBuf = (outBuf + 1) % size;
                mutex.V();
                isFull.V(); // notify any waiting producer
                return value;
        }
}
```

A mutex semaphore is used to is used for mutual exclusion, it is a binary semaphore. Two counting semaphores are used for conditional synchronization. isEmpty is initialed to 0, and isFull is initaled to the size of the buffer.
The order of P() operations and V() operations for isFull, mutex and isEmpty is important. If mutex.P() is called first and mutex.V() is called the last, a producer thread can enter the CS section but wait on the buffer to become not full. At this time, the consumer is waited on mutex.P() to enter the CS so that it can notify the waiting producer to continue. This creates deadlock.

5.1.5 Reader & Writer

Problem is defined as Readers and Writers with a shared database. The constraints are:

· **no read write conflict**

· **no write write conflict**

· **multiple readers are okay**

The following Java class shows how the Reader and Writer is implemented:

```
class ReaderWriter {
        int numReaders = 0;
        BinarySemaphore mutex = new BinarySemaphore(true);
        BinarySemaphore wlock = new BinarySemaphore(true);
        public void startRead() {
                mutex.P();
                numReaders++;
                if (numReaders == 1) wlock.P();
                mutex.V();
        }
        public void endRead() {
                mutex.P();
                numReaders--;
                if (numReaders == 0) wlock.V();
                mutex.V();
        }
        public void startWrite() {
                wlock.P();
        }
        public void endWrite() {
                wlock.V();
        }
}
```

A binary semaphore, mutex, is used for mutual exclusion. Another binary semaphore, wlock, is used as lock for writer.
For the startWrite and endWrite method, the use of wlock ensures only one writer is writing at anytime. For Readers, we have to make sure no writer is writing. For the first reader, the wlock is acquired to make sure no writer can enter CS, and the wlock is released if there is no reader. The number of readers is increased & decreased in CS guaranteed by mutex.

5.1.6 Dining Philosophers Problem

The problem is defined as multiple philosophers (threads) are sitting in a round table. One chopstick is placed between every two philosophers. The philosopher needs both two chopsticks next to him to eat the food. The constraints are:

· **Two neighboring philosophers can not eat the same time**

The following Java class models the philosopher:

```
class Philosopher implements Runnable {
        int id = 0;
        Resource r = null;
        public Philosopher(int initId, Resource initr) {
                id = initId;
                r = initr;
                new Thread(this).start();
        }
        public void run() {
                while (true) {
                        try {
                                System.out.println("Phil " + id + " thinking");
                                Thread.sleep(30);
                                System.out.println("Phil " + id + " hungry");
                                r.acquire(id);
                                System.out.println("Phil " + id + " eating");
                                Thread.sleep(40);
                                r.release(id);
                        } catch (InterruptedException e) {
                                return;
                        }
                }
        }
}
```

Philosophers run in state of thinking, hungry to eat, eating, then back to thinking.
The following Java class implements the shared uses of the resource:

```
class DiningPhilosopher implements Resource {
        int n = 0;
        BinarySemaphore[] fork = null;
        public DiningPhilosopher(int initN) {
                n = initN;
                fork = new BinarySemaphore[n];
                for (int i = 0; i < n; i++) {
                        fork[i] = new BinarySemaphore(true);
                }
        }
        public void acquire(int i) {
                fork[i].P();
                fork[(i + 1) % n].P();
        }
        public void release(int i) {
                fork[i].V();
                fork[(i + 1) % n].V();
        }
        public static void main(String[] args) {
                DiningPhilosopher dp = new DiningPhilosopher(5);
                for (int i = 0; i < 5; i++)
                        new Philosopher(i, dp);
```

```
            }
    }
```

Use binary semaphore for each chopstick. For each philosopher tries to eat, it acquire chopstick from both side.When he finishes eating, release both chopsticks.
However, this implementation can lead to deadlock. For example, all the philosopher picks up the left chopstick at the same time, and has to wait for right chopstick.This is called the problem of symmetry.
There are some possible solutions to solve the problem:

- **Bring asymmetric** asking some philosophers to pick up right chopstick first, while others pick up left chopstick first.
- **Constrain the number of philosophers eligible to acquire is less than the max number**

References

[GITHUB] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 6: September 13

Lecturer: Vijay Garg *Scribe: Sarang Bhadsavle*

Agenda and Announcements

Today's lecture covered the following topics:

- The lower bound on the number of shared locations required for mutual exclusion
- Fischer's Algorithm (A timing-based algorithm)
- Lamport's Fast Mutex Algorithm
  - Splitter Construct

Additionally, remember to watch the video seminar *Toward Extreme-Scale Manycore Architectures* and the video lectures on Semaphores from last week, as they will be covered on the first exam.

6.1 Lower Bound on Number of Shared Locations

Question: Is there any algorithm which, by using just one shared variable, we can achieve mutual exclusion?

Definition 6.1 *Covering State*

The notion of a state in which all shared variables are about to be overwritten by processes and the shared state is consistent with no process in the critical section.

Theorem 6.2 *(Barnes and Lynch) Any mutex algorithm that only uses read-write variables on n processes requires at least n shared locations.*

Proof: (n = 2)

Let P and Q be processes and let A be a shared memory location whose initial value is $\perp$. Let Q run until it is about to write to A, i.e. it is in the Covering State. Now let P run and enter the critical section. Let Q run again. Q writes to A and enters the critical section.

$\therefore$ Two processes are simultaneously in the critical section

$\therefore$ Violation of mutual exclusion ■

This proof can be extended to $n > 2$ by following a similar sequence of events to achieve a Covering State, and then letting processes run and enter the critical section simultaneously, causing a violation of mutual exclusion.

6.2 Fischer's Algorithm

It is possible to achieve mutual exclusion of n processes with a single shared variable, provided that a key assumption is made about the timing of the algorithm. The algorithm along with this assumption is shown below.

Algorithm 1 Fischer's Algorithm (for a process P_i)

```
1: shared var turn = -1; // door is open

2: procedure REQUESTCS
3:     while(true):
4:         while(turn != -1):
5:             noOp();
6:         turn = i;
7:         wait for △t time units;
8:         if(turn == i):
9:             return;

10: procedure RELEASECS
11:     turn = -1;
```

The key assumption with Fischer's Algorithm is that $\triangle t \geq c$, where c is is the maximum time required to close the door i.e. the time required to set *turn* to *i*.

Without this assumption, the following sequence of events could occur and cause a violation of mutual exclusion for two processes, P_i and P_j:

1. P_i reads turn = -1
2. P_j reads turn = -1
3. P_j sets turn = j
4. P_j reads turn = j, and enters critical section
5. P_i sets turn = i
6. P_i reads turn = i and enters critical section

Fischer's algorithm satisfies mutex by ensuring that the time period that a process P_j waits after setting turn = j and before checking if turn == j is greater than the time period that another process P_i takes after reading turn == -1 and before setting turn = i. This ensures that only one process (in this case, P_i) can enter the critical section.

6.3 Lamport's Fast Mutex Algorithm

Lamport's Fast Mutex Algorithm is used to provide mutual exclusion in a very fast way when there is no contention for the critical section. When there is contention, the regular method of mutual exclusion is followed.

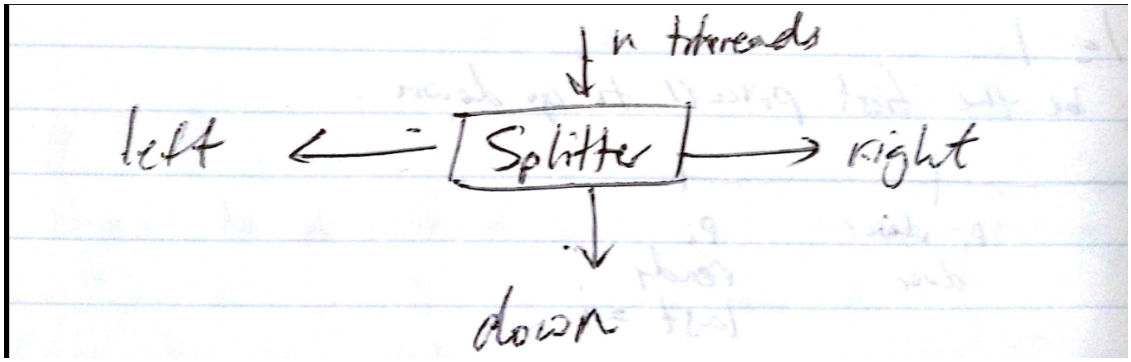


Figure 6.1: A Splitter Construct

Splitter

The construct of a "splitter" is used in Lamport's Fast Mutex algorithm, as seen in Figure 6.1.

The Splitter guarantees that:

- The number of processes sent right $\leq$ n - 1
- The number of processes sent left $\leq$ n - 1
- The number of processes sent down ≤ 1

Algorithm 2 Splitter Construct (for a process P_i)

```
1: shared var door : {open, closed} init open;
2: shared var last : pid init -1;

3: last = i;
4: if(door == closed):
5:     return left;
6: else:
7:     door = closed;
8:     if(last == i): return down;
9:     else: return right;
```

Claim 6.3 $\|left\| \leq n - 1$

Proof: Some process *must* have closed the door. ■

Claim 6.4 $\|right\| \leq n - 1$

Proof: Consider the process P_i such that textitlast = textiti. Then P_i must be part of left or down. ■

Claim 6.5 $\|down\| \leq 1$

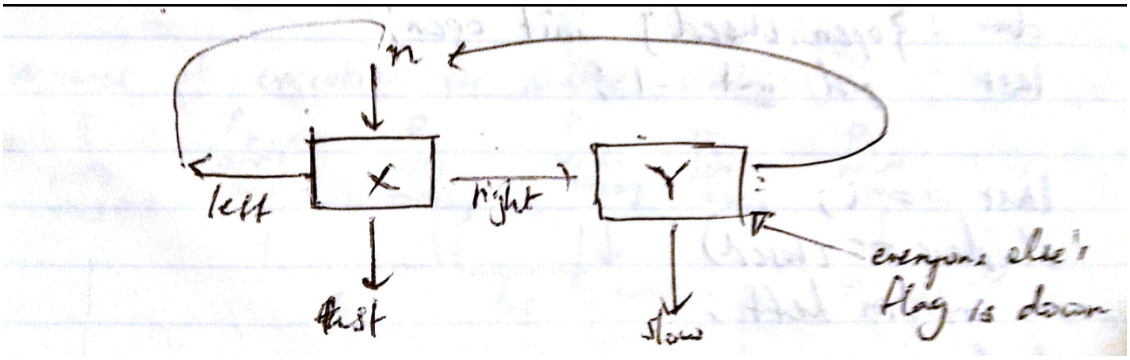


Figure 6.2: Design of Lamport's Fast Mutex Algorithm using Splitter

Proof: Let a process P_i be the first process to go down. In order to go down, P_i must execute 3 steps:

1. P_i writes *last*
2. P_i closes *door*
3. P_i reads *last* $== i$

Suppose there is some other process P_j also in the splitter. P_i must write *last* at some point. There are 3 cases:

- Suppose P_j writes *last* before 1. Then P_j would be the first process to go down, which violates the initial supposition that P_i went down first
- Suppose P_j writes *last* after 1 but before 3. However, this is not possible because P_i read *last* $== i$.
- Suppose P_j writes *last* after 3. However, this is not possible because P_j could not have found *door* to be open in the first place.

$\therefore$ Since all 3 cases are shown to be impossible, there can only be at most one process that goes down in the Splitter. ■

Lamport's Fast Mutex algorithm can be constructed using Splitter, as seen in Figure 6.2.

Proposition 6.6 *Note that Lamport's Fast Mutex Algorithm does not guarantee starvation-freedom*

Algorithm 3 Lamport's Fast Mutex Algorithm (for a process P_i)

```
1:  procedure ACQUIRE
2:      while(true):
3:          flag[i] = up;
4:          x = i;
5:          if(y != i): // splitter's left
6:              flag[i] = down;
7:              waitUntil(y == -1):
8:              continue;
9:          else:
10:             y = i;
11:             if(x == i): return; // fast path
12:             else: // splitter's right
13:                 flag[i] = down;
14:                 waitUntil(∀j : flag[j] == down);
15:                 if(y == i): return; // slow path
16:                 else:
17:                     waitUntil(y == -1);
18:                     continue;

19: procedure RELEASE
20:     y = -1;
21:     flag[i] = down;
```

References

[1] Vijay Garg. EE382C, Multicore Computing, The University of Texas at Austin. September 2016.

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 6: September 08

Lecturer: Vijay Garg *Scribe: Jiaolong Yu*

6.1 Lower Bound on the Number of Shared Memory Location

Theorem 6.1 *(Burns and Lynch)Any mutex algorithem that uses only RW on n processes require at least n shared locations.*

Proof: Consider 2 processes, say P and Q, in competing for Critical Section. They have only one shared memory location A. Let Q run till it's about to write to A. Let P run and enter critical section. Let Q run again. It enters CS. Mutex violation. Consider 3 processes, say P, Q and R. They have 2 shared memory locations A and B. Let P and Q run till they are about to write to A. Let R run and enter critical section. let P and Q run again. One of them will enter critcal section. Mutex violation. With this method we can extend the situation to n processors and prove the theorem. ■

Definition 6.2 *Covering state. All share variables are about to be overwritten by processes and the shared stats is consistant with no process in CS.*

6.2 Fischer's Algorithem

Turn = -1 Means the door is open.
RequestCS:

```
while(ture) {
  while(turn != -1);
  turn = i;
  wait_for_delta_time_units();
  if(turn == i) return;
}
```

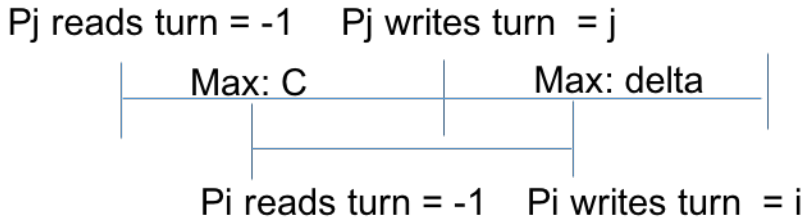
ReleaseCS:

```
turn = -1
```

This algorithm can cause mutex violation. One senario:
P_i reads turn = -1
P_j reads turn = -1
P_j sets turn = j
P_j reads turn = j*andenterCS*
P_i sets turn = i
P_i reads turn = i and enter CS

Theorem 6.3 *Assuming delta >= C, Fischer's Algorithm satisfies mutex. C: maximum time required to close the door, ie. set the turn.*

Proof: Let P_i be the processor that enters CS successfully. Assume there is another processor P_j that may enter CS.

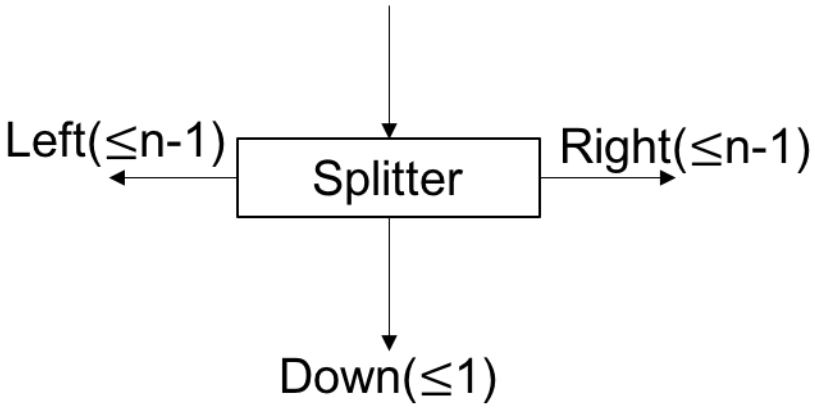


P_j must writes turn after P_i reads it to be -1. If it writes before P_i writes turn, after waiting for delta units of time which is longer than C, then P_i must have already finished writing turn, P_j will defnitely read turn = i then it will not enter CS. If P_j writes turn after P_i writes turn, then P_i will read turn = j after waiting delta units of time, then it will not enter CS. Proved that only one processor will enter CS ■

6.3 Lamport's Fast Algorithm

This algorithm enables for processors to enter CS fast when there is no contention.
No contention – fast path Contention – slow path

6.3.1 Splitter




```
For every P_i:
  Variables:
    door: {open, closed}, initially open
    last: pid, initially -1
  last = i;
  if (door == closed) return left;
  else {
    door = closed;
    if (last == i) return down;
```

```
  else return right;
}
```

Claim 6.4 $|left| <= n - 1$

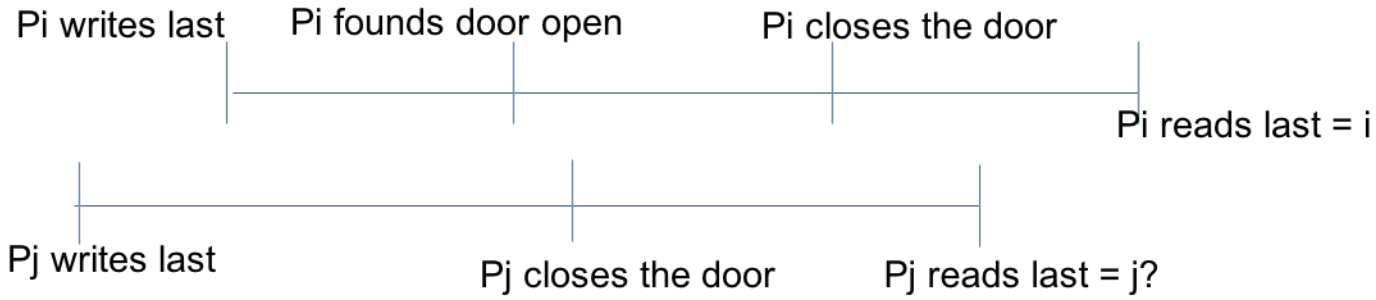
Proof: Someone need to close the door ■

Claim 6.5 $|right| <= n - 1$

Proof: Consider processor P_i such that last = i then P_i is part of left or down. So as at least one process would not go right. ■

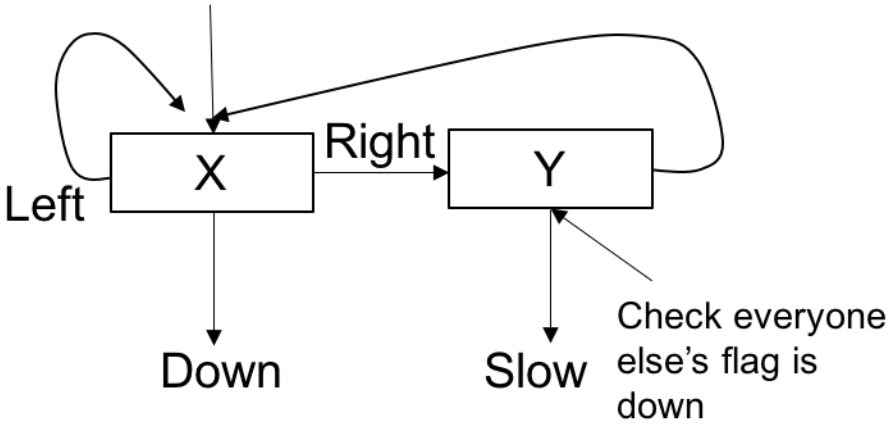
Claim 6.6 $|down| <= 1$

Proof: Let P_i be the first process to go down. Assume there is another processor P_j that may go down.



As shown in the picture, if P_j does everything bofore P_i writes last, then P_j would be the first one to go down, which conflicts with our assumption. However, P_j must writes to last before P_i does otherwise P_i would not be able to read last = i at the end. P_j must closes the door after P_i checks the door, otherwise P_i would have found the door closed, then it cannot go down. Then we have P_i writes to last before P_i checks the door, before P_j closes the door, before P_j reads last. It means P_j must reads last = i then it can not go down. ■

6.3.2 Lamport's Fast Algorithm



RequestCS:

```
while (ture) {
  flag[i] = up;
  x = i;
  if (y != -1) { // split left
    flag[i] = down;
    waituntil( y == -1);
    continue;
  } else {
    y = i;
    if (x == i) return; // down
    else { // right
      flag[i] = down;
      waituntil(y = -1);
      continue;
    }
  }
}
```

ReleaseCS:

```
y = -1;
flag[i] = down;
```

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 7: September 15

Lecturer: Vijay Garg *Scribe: Yichi Zhang*

7.1 Introduction

In this lecture, we will first review the Lamport's Fast Mutex Algorithm and then discuss a couple of basic synchronized constructs of openMP. Finally, we will solve the puzzle mentioned in the first class with best solution balancing the complexity of $Work$ and $Time$.

Outline is lied as follows:
- Lamport's Fast Mutex Algorithm (review)
- Discuss basic knowledge of OpenMP explained with examples
- Solve the previous puzzle (and come up with a new one)

7.2 Lamport's Fast Mutex Algorithm

Code of Fast Mutex Algorithm on next page. Fast Mutex Algorithm is using Splitter method and can be drawn like Figure 7.1.

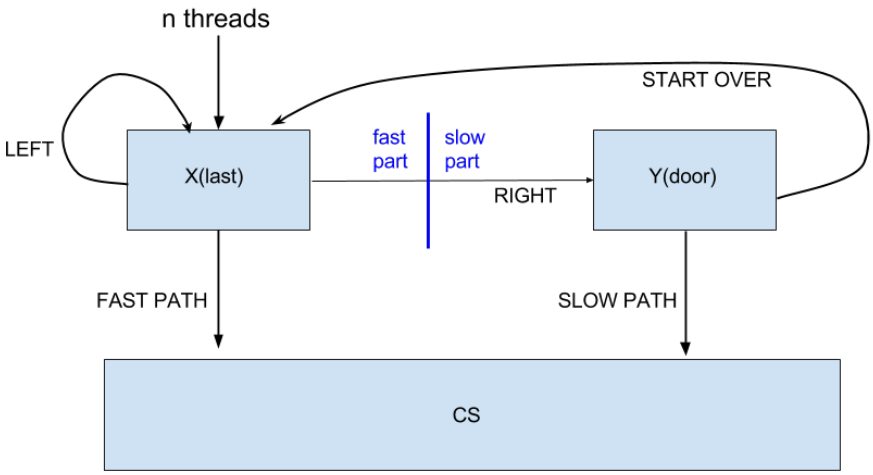


Figure 7.1: Fast Mutex Algorithm Model

Explanation for this algorithm:

There are two shared registers, X and Y, that each processor and read and write. For X, it holds the last process that acquires permission to get into the critical session and is initially set to -1. For Y, it simulates the door, $Y = -1$ indicates the door is open and $Y = i$ means door closed and process P_i is the

```
1   var
2       X, Y: int initially -1;
3       flag: array[1..n] of {down, up};
4
5   acquire(int i)
6   {
7     while(true)
8       flag[i] := up;
9       X := i;
10      if (Y != -1) {                          // splitters left
11          flag[i] := down;
12          waitUntil(Y == -1);
13          continue;
14      }
15      else {
16          Y := i;
17          if (X == i)                         // success with splitter
18              return;                         // fast path
19          else {                              splitter's right
20              flag[i] := down;
21              forall j:
22                  waitUntil(flag[j] == down);
23              if (Y == i) return;             // slow path
24              else {
25                  waitUntil(Y == -1);
26                  continue;
27              }
28          }
29      }
30
31  acquire(int i)
32  {
33      Y := -1;
34      flag[i] := down;
35  }
```

Lamport's Fast Mutex Algorithm

last one that get entered the door. Also, there are n-length array $flag[n]$ for each process, $flag[i]$ could be *up* or *down*, *up* means P_i is contending for mutex using the fast path, *down* means other cases.

As can see from Figure 7.1, at first, process P_i goes into block X, set $X = i$, $flag[i] = up$ and check if the door is open($Y = -1$). If not open($Y >= 0$), it would go to the **left** route and try acquiring again some later. If open($Y = -1$), it would close the door by setting $Y = i$ and check if the last acquiring process is still itself ($X = i$), if so, it would go into CS through the **fast** route, if not, go to the Y block through the **right** route and set $flag[i] = down$. The last chance to get into CS in this iteration is to wait all flags to be down(which means that no process is in the CS) and if so, at this moment, if the last door-closing process is still P_i, then it will enter CS through the **slow** route, otherwise it would start over and try acquiring again.

To sum, Lamports Fast Mutex Algorithm can guarantee deadlock-free but can not guarantee starvation-free. Fast Mutex Algorithm improves Splitter method that it expands Splitter model as two part, fast part

and slow part. For fast part, the **down** route in fast part is similar as the **down** route in Splitter, can guarantee that the **|Down|**, the number of threads through down route, is no more than 1, but could be 0. When this case happens, there is nothing in CS, so the slow part could help this algorithm to pick one thread enter CS through slow path, to ensure there is no deadlock. But unlike PetersonN Algorithm, Fast Mutex Algorithm is not first-come-first-serve algorithm, which means the first-come thread might have to wait infinitely. Thus it cannot guarantee starvation-free.

7.3 Basic synchronized constructs of OpenMP

OpenMP(Open Multi-Processing)[1] is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. By using openMP, it is easier to convert a serial code into parallel and also can run the same code as serial code. Figure 7.2 shows that openMP can run parallel code as serial flow, it can control when to distribute main thread into parallel processes and when to merge all parts as a real sequential program.

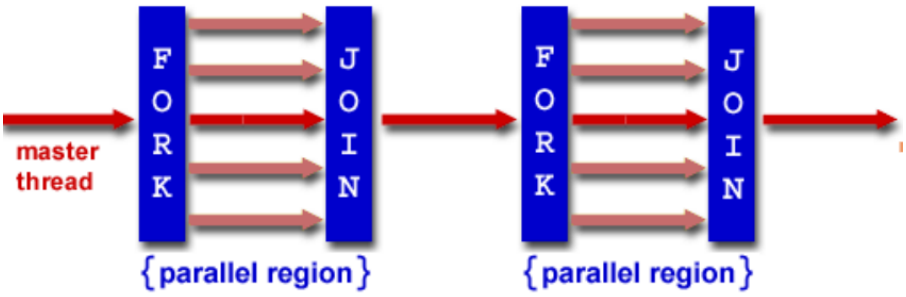


Figure 7.2: Simple Fork-Join Parallelism[2]

Most of the constructs in OpenMP are compiler directives. For example:

$$\#pragma\ omp\ parallel\ num\_threads(4)$$

is to set the number of threads as 4 (there is always one master thread as the main thread so to create 4 threads means to create 3 more threads besides the master thread).

7.3.1 OpenMP variables

Now we will introduce several variables of OpenMP with examples and explanation[3].

7.3.1.1 Critical

Variable *critical* indicates Critical Session, in the code, we have n processors and can execute all iterations in the for-loop at the same time but only one thread would run $Line10$ at a time.

7.3.1.2 Atomic

Variable *atomic* specifies that a memory location that will be updated atomically. For the code, all $foo(i)$ can be executed by n threads at the same time, but r will be updated atomically.

```
1   float result;
2   #pragma omp parallel
3   {
4       float B; int i, id, nthrds;
5       id = omp_get_thread_num();
6       nthrds = omp_get_num_threads();
7       for(i=id; i < N; i = i+nthrds) {
8           B = foo(i);                    // expensive computation
9       #pragma omp critical
10          consume(B, result);
11      }
12  }
```

Variable Critical

```
1   int n,r;
2   #pragma omp parallel shared(n,r)
3   {
4       for(i=0; i < n; i++) {
5       #pragma omp atomic
6           r += foo(i);                   // expensive computation
7       }
8   }
```

Variable Atomic

7.3.1.3 Reduction

```
1   double sum=0.0, ave, A[MAX]; int i;
2   #pragma omp parallel for reduction (+ : sum)
3   for(i=0; i < MAX; i++) {
4       sum += A[i];
5   }
6   ave = sum/MAX;
```

Variable Reduction

Variable *reduction* specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region. For the above code, local copies of variable *sum* would be created and then combined.

7.3.1.4 Barrier

Variable *barrier* synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.

7.3.1.5 Master construct

The target region will be executed only by the *master* thread.

7.3.1.6 Single construct

```
1  #pragma omp parallel
2  {
3      do_many_things();
4      #pragma omp single
5      {exchange_boundaries();}    // target region
6      #pragma omp barrier
7      do_many_other_things();
8  }
```

Variable Single construct

The target region will be executed by just one thread (any thread, not necessary the master). In this case, *Line* 5 would be executed by one thread, if the construct here is *master* instead of *single*, *Line* 5 would only be executed by master thread. And after *Line* 5, there is a barrier at *Line* 6, which makes other threads waiting for the working threads finish their work and then go to *Line* 7 at the same time.

7.3.1.7 Ordered

Locations updated in any order but printed in sequential order.

7.3.1.8 Lock

- omp_init_lock() – Initializes a simple lock.
- omp_set_lock() – Blocks thread execution until a lock is available.
- omp_unset_lock() – Releases a lock.
- omp_test_lock() – Attempts to set a lock but does not block thread execution.
- omp_destroy_lock() – Uninitializes a lock.

7.3.2 OpenMP data attributes

- *Shared* variable is shared among threads.
- *Private* variable(*var*) creates a new local copy of *var* for each thread.
- *Firstprivate* variable initializes each private copy with the corresponding value from the **master** thread.
- *Lastprivate* variable passes the value of a private from the **last** iteration to a global variable.

See the example on next page. We can pick one line from those three: *Line* 4, *Line* 5, *Line* 6. *Line* 4 indicates that the value(*tmp*) is uninitialized and undefined after the region. *Line* 5 indicates that all copies have value of *tmp* initialized as 0. *Line* 6 indicates that not only all copies have value of *tmp* initialized as 0, but after the *for* loop, the variable *tmp* has the value from the last iteration (i.e. j=99).

7.3.3 Dijkstra Algorithm

Dijkstra code example with OpenMP implementation can be seen from [4].

```
1   void Foo()
2   {
3       int tmp = 0;
4       #pragma omp for private(tmp)/
5       #pragma omp for firstprivate(tmp)/
6       #pragma omp for firstprivate(tmp) lastprivate(tmp)
7           for(int j = 0;j < 100; j++ ) {
8           tmp += j;
9           printf("%d\n", tmp);
10      }
11  }
```

OpenMP data attributes

7.4 Solve the puzzle

From previous lectures, we have three ways to solve the find-maximum-from-N-numbers puzzle, listed as Table 7.1

Algorithm	Work	Time
Sequential	O(N)	O(N)
Binary	O(N)	O($log(N)$)
All-pair Algorithm	O(N^2)	O(1)

Table 7.1: Previous result of solving the puzzle

Now we will discuss two new methods to solve the puzzle with better performance.

7.4.1 DoublyLog Algorithm

In All-pair Algorithm, we have the idea that we can divide the N numbers into $\sqrt{N}$ groups with each group has $\sqrt{N}$ numbers. To expand that, we can more deeply divide each $\sqrt{N}$ groups into $\sqrt{\sqrt{N}}$ sub-groups and keep dividing until the size of each group is only 1 or 2. This idea is shown in Figure 7.3.

The height of this tree is k, which is $log(log(N))$, if we have enough processors, the time complexity will be the number of layers, which is $O(log(log(N)))$, and for each layer, the work we will be $O(N)$, so the total work will be $O(Nlog(log(N)))$.

7.4.2 Cascaded Algorithm

To better improve the performance, we try to cascade two different methods to get benefit from both of them. In Cascaded Algorithm, we synthesize Sequential Algorithm and DoublyLog Algorithm.

We divide N numbers into $log(log(N))$-length groups, which means there are $N/log(log(N))$ groups. With sequential algorithm, we can get $N/log(log(N))$ maximum candidates, and then we get the final maximum by compare these candidates using DoublyLog Algorithm. This idea is shown in Figure 7.4.

For the first part, work complexity for sequential is $O(N)$ and time complexity is size of each group,

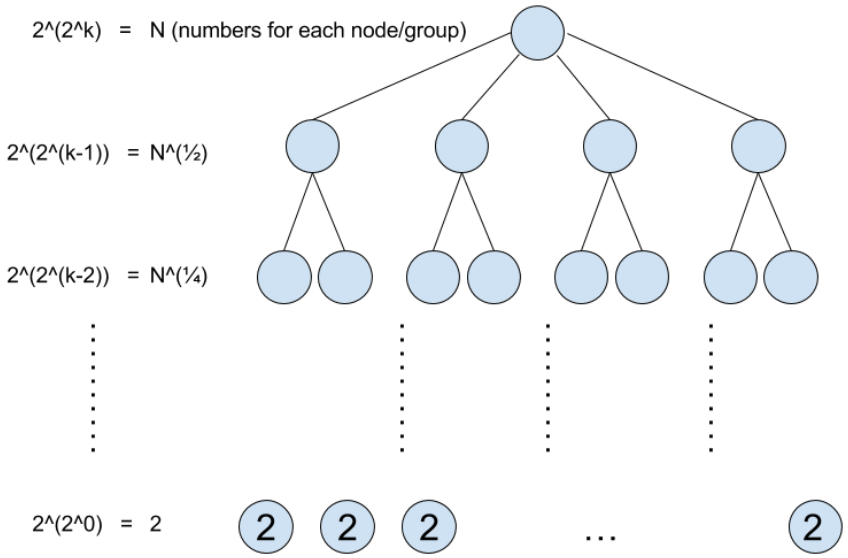


Figure 7.3: DoublyLog Algorithm Model

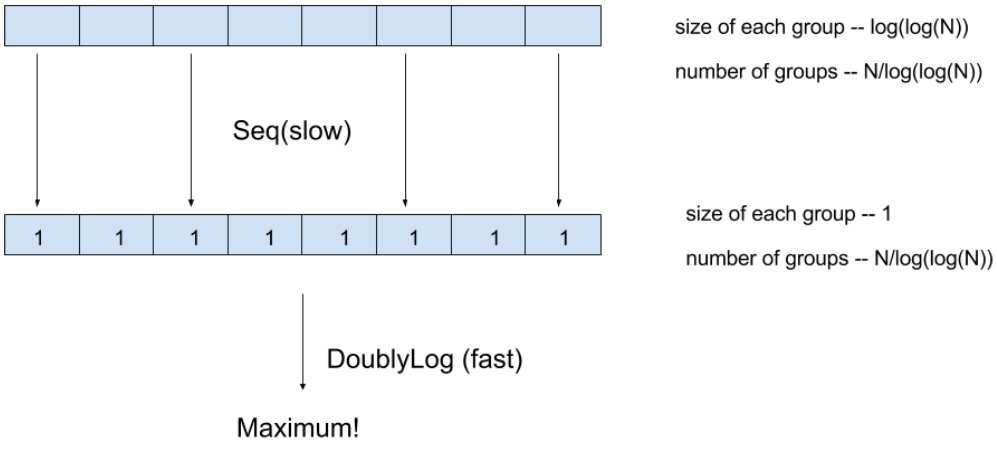


Figure 7.4: Cascaded Algorithm Model

which is $O(log(log(N)))$. For the second part, according to DoublyLog ALgorithm, work complexity is $N/(log(log(N))) * (log(log(N))) = O(N)$, and time complexity is still $O(log(log(N)))$. Now we can expand Table 7.1 as Table 7.2.

Algorithm	Work	Time
Sequential	O(N)	O(N)
Binary	O(N)	O($log(N)$)
All-pair Algorithm	O(N^2)	O(1)
DoublyLog Algorithm	O($N * log(log(N))$)	O($log(log(N))$)
Cascaded Algorithm	O(N)	O($log(log(N))$)

Table 7.2: Current result of solving the puzzle

7.4.3 New puzzle

We have two sorted arrays, say $A[m]$ and $B[n]$, how could we merge them into a larger array, say $C[m+n]$, in parallel?

References

[1] https://en.wikipedia.org/wiki/OpenMP

[2] http://www.llnl.gov/computing/tutorials/openMP

[3] https://msdn.microsoft.com

[4] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/blob/master/openMP/Dijkstra.c

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 7: September 15

Lecturer: Vijay Garg *Scribe: Tong Zhou*

7.1 Outline

The lecture covered the following topics:

- Review of Lamport's Fast Mutex Algorithm
- Introduction to OpenMP
- Two new ways to find maximum from N numbers

7.2 Lamport's Fast Mutex Algorithm

Lamport's fast mutex algorithm uses two shared variables X and Y and a shared single-writer-multiple-reader registers $flag[i]$. A process P_i can enter the critical section either along a *fast* path or along a *slow* path. The algorithm is shown below.

The variable $flag[i]$ is set to value *up* if P_i is actively contending for mutual exclusion using the fast path. When Y is -1, the door is open. A process P_i closes the door by updating Y with i. Lamport's Fast Mutex Algorithm is deadlock-free but allows starvation of individual processes.

In the algorithm, process i first sets X to i, and then checks the value of Y. When it finds $Y = -1$, it sets Y to i and then checks the value of X. If $X = i$ process i enters its critical section along the *fast* path. At most one process can enter its critical section along *fast* path. If a process finds $X \neq i$ then it delays itself by looping until it sees that all the values in the array *flag* are *down*. Checking these n values in array *flag* plays two roles:

1. Say that process i enters its critical section along *slow* path, if it finds $Y = i$ after exiting the for-loop. A consequence of having observed all the values in the array *flag* to be *down* is that the value of Y will not be changed thereafter until process i leaves the critical section. This follows because every other contending process either reads $Y \neq -1$ and waits at $waitUntil(Y == -1)$, or reads $Y = 0$ and then finished the assignment $Y := i$ before setting $flag[i]$ to *down*. Hence, once process i finds $Y = i$ after the loop, no other process can change the value of Y until process i sets Y to -1 in its exit code. It follows that at most one process can enter along *slow* path.

2. The for-loop ensures that if a process enters the critical section along *fast* path, then any other process is prevented from entering along *slow* path. To see this, observe that when a process i enters its critical section along *fast* path, its $flag[i]$ remains *up*. Thus, if another process tries to enter along *slow* path it will find that $flag[i] == up$ and will have to wait until process i exits its critical section and set $flag[i]$ to *down*.

Algorithm 1 Lamport's Fast Mutex Algorithm

```
1: var
2:     X, Y: int initially -1;
3:     flag: array[1..n] of {down, up};

4: acquire(int i)
5: {
6:     while(true):
7:         flag[i] = up;
8:         X = i;
9:         if(Y != -1): // splitter's left
10:            flag[i] = down;
11:            waitUntil(Y == -1):
12:            continue;
13:        else:
14:            Y = i;
15:            if(X == i): return; // fast path
16:            else: // splitter's right
17:                flag[i] = down;
18:                waitUntil(∀j : flag[j] == down);
19:                if(Y == i): return; // slow path
20:                else:
21:                    waitUntil(Y == -1);
22:                    continue;
23: }

24: release(int i)
25: {
26:     Y = -1;
27:     flag[i] = down;
28: }
```

7.3 OpenMP

OpenMP(Open Multi-Processing) is an application programming interface that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, processor architectures and operating systems.

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met. The sequential program evolves into a parallel program.

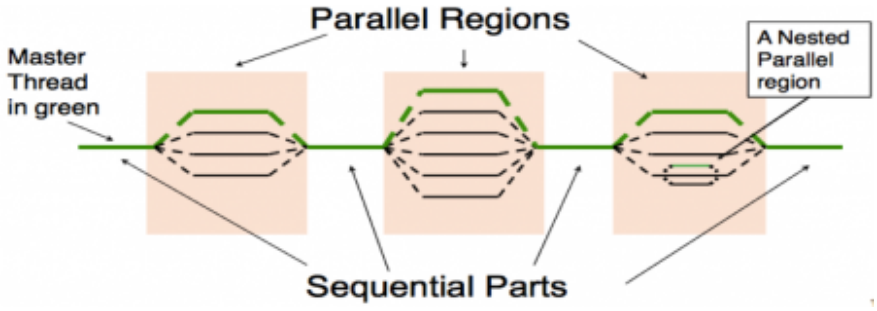


Figure 7.1: Fork-Join Parallelism

We create threads in OpenMP with the parallel construct. For example, to create a 4 thread Parallel region. Each thread calls $foo(ID, A)$ for $ID = 0$ to 3.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    foo(ID, A);
}
```

Thread creation: parallel regions

7.3.1 Synchronization

Synchronization is used to impose order constraints and to protect access to shared data. In OpenMP, we have 4 high level synchronization:

- critical
- atomic
- barrier
- ordered

7.3.1.1 Synchronization: critical

Mutual exclusion: only one thread at a time can enter a *critical* region. Example code:

```
float result;
#pragma omp parallel
{
    float B; int i, id, nthrds;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for(i=id; i<N; i = i+nthrds){
        B = foo(i); // expensive computation
    #pragma omp critical
        consume(B, result)
    }
}
```

7.3.1.2 Synchronization: atomic

Atomic provides mutual exclusion but only applies to the update of a memory location. In the following example code, $foo's$ are run in parallel but r updated atomically.

```
int n,r;
#pragma omp parallel shared(n,r)
{
    for(i=0; i<n; i = i++){
    #pragma omp atomic
        r += foo(i); // expensive computation
    }
}
```

7.3.1.3 Synchronization: barrier

Barrier: Each thread waits until all threads arrive. Example code:

```
#pragma omp parallel shared (A, B, C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
    for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
    #pragma omp for nowait
    for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }
    A[id] = big_calc4(id);
}
```

7.3.1.4 Synchronization: ordered

The *ordered* region executes in the sequential. In the following example code, array *a* updated in any order but printed in sequential order

```
#pragma omp parallel for
{
  for(i=0; i<n; i = i++){
    tid = omp_get_thread_num();
    printf("Thread %d updates a[%d]\n", tid, i);
    a[i] += i;
    #pragma omp ordered
    {
      printf("Thread %d prints value of a[%d]=%d\n",
                tid, i, a[i]);
    }
  }
}
```

7.3.1.5 Synchronization: locks

We also have low level synchronization *locks* (both simple and nested).

- Simple Lock routines: A simple lock is available if it is unset.
  - omp_init_lock()
  - omp_set_lock()
  - omp_unset_lock()
  - omp_test_lock()
  - omp_destroy_lock()
- Nested Locks: A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function.
  - omp_init_nest_lock()
  - omp_set_nest_lock()
  - omp_unset_nest_lock()
  - omp_test_nest_lock()
  - omp_destroy_nest_lock()

7.3.2 Data Environment and Data Attributes

All variables declared outside parallel for pragma are shared by default, except for loop index. *for* index variable is private. One can selectively change storage attributes for constructs using the following clauses.

- shared
- private
- firstprivate

7.3.2.1 Private Clausse

private(var) creates a new local copy of var for each thread. The value is uninitialized and is undefined after the region.

7.3.2.2 firstprivate Clausse

firstprivate is a special case of private. Initializes each private copy with the corresponding value from the master thread. In the following example, All copies have value of tmp initialized as 0.

```
void Foo()
{
    int tmp = 0;
    #pragma omp for firstprivate(tmp)
    for (int j = 0; j < 1000; ++j) {
        tmp += j;
    }
}
```

7.3.2.3 lastprivate Clausse

lastprivate passes the value of a private from the last iteration to a global variable. In the following example, All copies have value of tmp initialized as 0. After the for loop, the variable tmp has the value from the last iteration (i.e. j=99).

```
void Foo()
{
    int tmp = 0;
    #pragma omp for firstprivate(tmp) lastprivate(tmp)
    for (int j = 0; j < 100; ++j)
        tmp += j;
        printf("%d\n", tmp);
}
```

7.4 Find Maximum from N Numbers

In the first lecture, we mentioned four ways to find maximum from N numbers.

	time complexity	space complexity
Sequential	$O(N)$	$O(N)$
Binary Tree	$O(log(N))$	$O(N)$
All-pair	$O(1)$	$O(N^2)$
Comparison	$O(1)$	$O(N^{3/2})$

7.4.1 DoublyLog Algorithm

As we mentioned in the All-pair Algorithm, N numbers can be divided into $\sqrt{N}$ groups. In each group, there are $\sqrt{N}$ numbers. In DoublyLog Algorithm, we further divide every $\sqrt{N}$ group into $\sqrt{\sqrt{N}}$ sub-groups. We keep doing so until the size of every group is 1 or 2. The height of this tree structure is $log(log(N))$. For each layer, the computing time complexity is $O(N)$. The total time complexity for DoublyLog Algorithm is $O(Nlog(log(N)))$.

7.4.2 Cascaded Algorithm

In Cascaded Algorithm, we combine Sequential Algorithm and DoublyLog Algorithm to reduce the number of processors that we need.

Divide N numbers into $N/log(log(N))$ groups. Then, in every group, there are $log(log(N))$ numbers. We use Sequential Algorithm to get $N/log(log(N))$ maximum candidates in every group. For these maximum candidates, we use DoublyLog Algorithm.

For Sequential Algorithm, the work complexity is $O(N)$ and time complexity is $log(log(N))$. To select maximum from candidates with DoublyLog Algorithm, the time complexity is $O(log(log(N)))$ and work complexity is $O(N)$ (There are$N/(log(log(N)))$ groups. Every group has $log(log(N))$ time complexity).

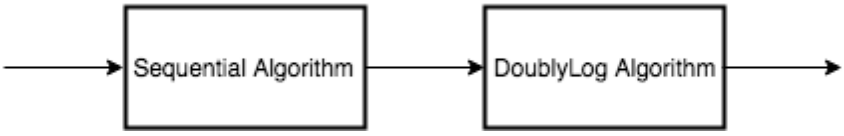


Figure 7.2: Cascaded Algorithm Model

References

[1] Leslie Lamport, A Fast Mutual Exclusion Algorithm(1986).

[2] Michael Merritt, Gadi Taubenfeld, Speeding Lamport?s Fast Mutual Exclusion Algorithm(1991).

[3] Tim Mattson, A "Hands-on" Introduction to OpenMP (2008).

[4] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 8: September 20

Lecturer: Vijay Garg *Scribe: Liheng Ding*

8.1 Introduction

This lecture covers following topics:

1. Locks with Get-and-Set Operation
2. Queue Locks

8.2 Locks with Get-and-Set Operation

Although locks for mutual exclusion can be built using simple read and write instructions, any such algorithm requires as many memory locations as the number of threads. By using instructions with higher atomicity, it is much easier to build locks. For example, the *getAndSet* operation (also called *testAndSet*) allows us to build a lock as shown in Alogrithm 1.

Algorithm 1 Building Locks Using GetAndSet.

```
1:  import java.util.concurrent.atomic.*;
2:  public class GetAndSet implements MyLock {
3:      AtomicBoolean isOccupied = new AtomicBoolean(false);
4:      public void lock() {
5:          while (isOccupied.getAndSet(true))
6:              Thread.yield();
7:              // skip();
8:          }
9:      }
10:     public void unlock() {
11:         isOccupied.set(false);
12:     }
13: }
```

This algorithm satisfies the mutual exclusion and progress property. However, it does not satisfy starvation freedom. Besides, keeping invoking *getAndSet* is not efficient enough since this method will use a shared bus to get access to *isOccupied*. So an alternative implementation is shown in Algorithm 2.

In this implementation, a thread first checks if the lock is available using the *get* operation. It calls the *getAndSet* operation only when it finds the critical section available. If it succeeds in *getAndSet*, then it enters the critical section; otherwise, it goes back to spinning on the *get* operation.

However, high bus contention still happens when all threads get that *isOccupied.get()* is *false* and try to set it to *true* by using *getAndSet()*. So a better implementation using *Backoff* is shown in Algorithm 3. The thread fails to set *getAndSet()* will back off for a certain period of time.

Algorithm 2 Building Locks Using GetAndGetAndSet.

```
1:  import java.util.concurrent.atomic.*;
2:  public class GetAndGetAndSet implements MyLock {
3:      AtomicBoolean isOccupied = new AtomicBoolean(false);
4:      public void lock() {
5:          while (true){
6:              while (isOccupied.get()) {
7:              }
8:              if (! isOccupied.getAndSet(true)) return;
9:          }
10:     }
11:     public void unlock() {
12:         isOccupied.set(false);
13:     }
14: }
```

Algorithm 3 Building Locks Using Backoff.

```
1:  import java.util.concurrent.atomic.*;
2:  public class GetAndGetAndSet implements MyLock {
3:      AtomicBoolean isOccupied = new AtomicBoolean(false);
4:      public void lock() {
5:          while (true){
6:              while (isOccupied.get()) {
7:              }
8:              if (! isOccupied.getAndSet(true)) return;
9:              else {
10:                 int timeToSleep = calculateDuration();
11:                 Thread.sleep(timeToSleep);
12:             }
13:         }
14:     }
15:     public void unlock() {
16:         isOccupied.set(false);
17:     }
18: }
```

8.3 Queue Locks

Previous approaches to solve mutual exclusion require threads to spin on the shared memory location. In this section, we present alternate methods to avoid spinning on the same memory location. All methods maintain a queue of threads waiting to enter the critical section. Anderson's lock uses a fixed size array, CLH lock uses an implicit linked list and MCS lock uses an explicit linked list for the queue. One of the key challenges in designing these algorithms is that we cannot use locks to update the queue.

8.3.1 Anderson's Lock

Anderson's lock uses a circular array **Available** of size n which is at least as big as the number of threads that may be contending for the critical section. The array is circular so that the index i in the array is always a value in the range $0..n-1$ and is incremented modulo n. Different threads waiting for the critical section spin on the different slots in this array thus avoiding the problem of multiple threads spinning on the same variable.

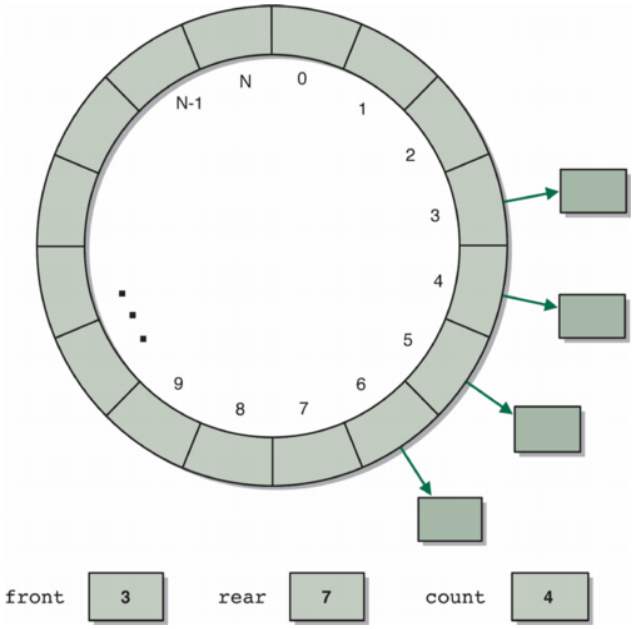


Figure 8.1: Circular Array

Note that the above description assumes that each slot is big enough so that adjacent slots do not share a cache line. Hence even though we just need a single bit to store **Available[i]**, it is important to keep it big enough by padding to avoid the problem of *false sharing*. Also note that since Anderson's lock assigns slots to threads in the FCFS manner, it guarantees fairness and therefore freedom from starvation.

A problem with Anderson lock is that it requires a separate array of size n for each lock. Hence, a system that uses m locks shared among n threads will use up $O(nm)$ space.

Algorithm 4 pseudo code for Anderson Lock

```
1:  public class AndersonLock {
2:      AtomicInteger tailSlot = new AtomicInteger (0);
3:      boolean [ ] Avaliable ;
4:      ThreadLocal <Integer>mySlot; // initialize to 0

5:      public AndersonLock (int n) { // constructor
6:      } // all Available false except Available[0]
7:      public void lock() {
8:          mySlot.set(tailSlot.getAndIncrement() % n);
9:          spinUntil (Available[mySlot]);
10:     }
11:     public void unlock() {
12:         Available[mySlot.get()] = false;
13:         Available[(mySlot.get()+1) % n] = true;
14:     }
15: }
```

8.3.2 CLH Queue Lock

We can reduce the memory consumption in Anderson's algorithm by using a dynamic linked list instead of a fixed size array. Any thread that wants to get lock inserts a node in the linked list. The insertion in the linked list must be atomic. The sub-algorithm is shown in Algorithm 5 and full algorithm can be found at [1]. It is important to note that if the thread exists the CS wants to enter the CS again, it cannot reuse its own node because the next thread may still spinning on that node's field. Therefore, it uses the predecessor in the *unlock* function.

The linked list is virtual. The thread that is spinning has the variable *pred* that points to the predecessor node in the linked list.

Algorithm 5 pseudo code for CLH Queue Lock

```
1:  class Node {
2:      boolean locked;
3:  }
4:  AtomicReference<Node>tailNode;
5:  ThreadLocal<Node>myNode;
6:  ThreadLocal<Node>pred;
7:  lock() {
8:      myNode.locked = true;
9:      pred = tailNode.getAndSet(myNode);
10:     while(pred.locked) { noops; }
11: }
12: unlock() {
13:     myNode.locked = false;
14:     myNode = pred; // reuse pred node for future
15: }
```

8.3.3 MCS Queue Lock

In many architectures, each core may have local memory such that access to its local memory is fast but access to the local memory of another core is slower. In such architectures, CLH algorithm results in threads spinning on remote locations. MCS avoids the remote spinning. Same as CLH, MCS maintain a queue. However, it is an explicit linked list.

The tricky part of MCS is in the *unlock()* function. When a thread *p* exits the CS, it checks if there is any node linked next to its node. If there exits such node, then it makes the *locked* false and removes its node from the linked list. When it finds no node linked next to it, there can be two cases for this. First, no thread has been inserted yet. Therefore, we can find that *tailNode* still points to *myNode*. It can be simply reset to null. In the second case, thread *p* waits until the next is not null. Then it can set the locked field for that node to be false as usual. This is shown in Algorithm 6.

Algorithm 6 pseudo code for MCS Queue Lock

```
1:  class QNode {
2:      boolean locked; // init true
3:      QNode next; // init null
4:  }
5:  lock() {
6:      QNode pred = tailNode.getAndSet(myNode);
7:      pred.next = myNode;
8:      while(myNode.locked) { noops; }
9:  }
10: unlock() {
11:     if (myNode.next == null) {
12:         if ( tailNode.compareAndSet(myNode, null)) return;
13:         while (myNode.next == null) { noops; }
14:     }
15:     myNode.next.locked = false;
16:     myNode.next = null;
17: }
```

References

[1] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 1: September 20

Lecturer: Vijay Garg *Scribe: Shuang Song*

1.1 Introduction

In this lecture, we first go over the GetAndSet operations and how do use them to achieve mutual exclusion. Additionally, we introduce the Queue locks and discuss the pros and cons of each queue lock. Outline is shown as follows:

1. Building locks using GetAndSet
2. Building locks using GetAndGetAndSet
3. Using backoff
4. Anderson's lock
5. CLH queue lock
6. MCS queue lock

1.2 Locks with Get-and-Set Operation

Locks for mutual exclusion can be built using simple rw instructions, which requires n memory locations for n threads. It is easier to use instructions with higher atomicity, such as the GetAndSet operation. The example lock is shown in Algorithm 1.

Algorithm 1 Building Locks Using GetAndSet.

```
1:  import java.util.concurrent.atomic.*;
2:  public class GetAndSet implements MyLock {
3:      AtomicBoolean isOccupied = new AtomicBoolean(false);
4:      public void lock() {
5:          while (isOccupied.getAndSet(true)){          ▷ isoccupied is true, then busy wait
6:              Thread.yield();                                                    ▷ skip();
7:          }          ▷ isoccupied is false, means CS is empty, thread goes to CS and set isoccupied back to true
8:      }
9:      public void unlock() {
10:         isOccupied.set(false);
11:     }
12: }
```

The problem of this approach can be summarized into three points, which are busy-wait, starvation freedom, and expensive getAndSet() operation. Keep checking by using getAndSet operation can cause the high contention for the

Algorithm 2 Building Locks Using GetAndGetAndSet.

```
1:  import java.util.concurrent.atomic.*;
2:  public class GetAndGetAndSet implements MyLock {
3:      AtomicBoolean isOccupied = new AtomicBoolean(false);
4:      public void lock() {
5:          while (true){
6:              while (isOccupied.get()) {
7:              }
8:              if (! isOccupied.getAndSet(true)) return;
9:          }
10:     }
11:     public void unlock() {
12:         isOccupied.set(false);
13:     }
14: }
```

shared bus. The goal here is to optimize the algorithm and alleviate the bus contention. The algorithm to achieve this goal is shown in Algorithm 2.

As can be seen, *isOccupied.get()* is a read instruction, which is served in cache mostly. Checking *isOccupied.get()* before using getAndSet reduces contention of the bus. Therefore, compared to Algorithm 1, the second implementation results in faster accesses to the critical session.

However, if all these threads now get that *isOccupied.get()* is *false* and try to set it to true by using getAndSet, only one of them can succeeds but all of them ending up causing high bus contention again. *Backoff*, illustrated in Algorithm 3, is an useful idea to solve this issue. The thread fails to set getAndSet, it backs off for a certain period of time.

Algorithm 3 Building Locks Using GetAndGetAndSet.

```
1:  import java.util.concurrent.atomic.*;
2:  public class GetAndGetAndSet implements MyLock {
3:      AtomicBoolean isOccupied = new AtomicBoolean(false);
4:      public void lock() {
5:          while (true){
6:              while (isOccupied.get()) {
7:              }
8:              if (! isOccupied.getAndSet(true)) return;
9:              else {
10:                 int timeToSleep = calculateDuration();
11:                 Thread.sleep(timeToSleep);
12:             }
13:         }
14:     }
15:     public void unlock() {
16:         isOccupied.set(false);
17:     }
18: }
```

1.3 Queue Locks

Previous approaches require threads to spin on the shared memory location to achieve mutual exclusion. To avoid spinnings, we discuss three methods here, which are Anderson's lock (fixed size array), CLH lock (implicit linked list) and MCS lock (explicit linked list).

Algorithm 4 pseudo code for Anderson Lock

```
1:  public class AndersonLock {
2:      AtomicInteger tailSlot = new AtomicInteger (0);
3:      boolean [ ] Avaliable ;
4:      ThreadLocal <Integer>mySlot;                          ▷ initialize to 0

5:      public AndersonLock (int n) {                         ▷ constructor
6:      }                                                     ▷ all Available false except Available[0]
7:      public void lock() {
8:          mySlot.set(tailSlot.getAndIncrement() % n);
9:          spinUntil (Available[mySlot]);
10:     }
11:     public void unlock() {
12:         Available[mySlot.get()] = false;
13:         Available[(mySlot.get()+1) % n] = true;
14:     }
15: }
```

1.3.1 Anderson's Lock

Anderson's lock uses a circular array *Available* of size n (shown in Figure 1.1), which is as big as the number of threads. Different threads waiting for the critical section spin on their own slots in the array, so it avoids the problem of multiple threads spinning on the same variable. The *TailSlot* is an atomic integer that is pointing to the next available slot in the array. Any thread acquires the lock will read the value of *tailSlot* in its local variable *mySlot* and calculates the next available slot by using atomic operation *getAndIncrement()*. It then spins on the *Available[mySlot]* until the slot becomes available. In this algorithm, only one thread is affected when a thread leaves the critical section, and all the spinnings result in local cache accesses. Anderson's lock has no starvation problem. It is shown in Algorithm 4.

The problem with Anderson lock is that it requires a separate array for each lock. Like, a system that uses m locks shared among n threads will use up to O(mn) space.

1.3.2 CLH Queue Lock

To reduce the space consumption in Anderson's lock, CLH uses a dynamic linked list instead of fixed size array. Any thread that wants to get lock inserts a node in the linked list. The insertion in the linked list must be atomic. The sub-algorithm is shown in Algorithm 5 and full algorithm can be found at [1]. It is important to note that if the thread exists the CS wants to enter the CS again, it cannot reuse its own node because the next thread may still spinning on that node's field. Therefore, it uses the predecessor in the *unlock* function.
The linked list is virtual. The thread that is spinning has the variable *pred* that points to the predecessor node in the linked list.

Algorithm 5 pseudo code for CLH Queue Lock

```
1:  class Node {
2:      boolean locked;
3:  }
4:  AtomicReference<Node>tailNode;
5:  ThreadLocal<Node>myNode;
6:  ThreadLocal<Node>pred;
7:  lock() {
8:      myNode.locked = true;
9:      pred = tailNode.getAndSet(myNode);
10:     while(pred.locked) { noops; }
11: }
12: unlock() {
13:     myNode.locked = false;
14:     myNode = pred;                          ▷ reuse pred node for future
15: }
```

Algorithm 6 pseudo code for MCS Queue Lock

```
1:  class QNode {
2:      boolean locked;                         ▷ init true
3:      QNode next;                             ▷ init null
4:  }
5:  lock() {
6:      QNode pred = tailNode.getAndSet(myNode);
7:      pred.next = myNode;
8:      while(myNode.locked) { noops; }
9:  }
10: unlock() {
11:     if (myNode.next == null) {
12:         if ( tailNode.compareAndSet(myNode, null)) return;
13:         while (myNode.next == null) { noops; }
14:     }
15:     myNode.next.locked = false;
16:     myNode.next = null;
17: }
```

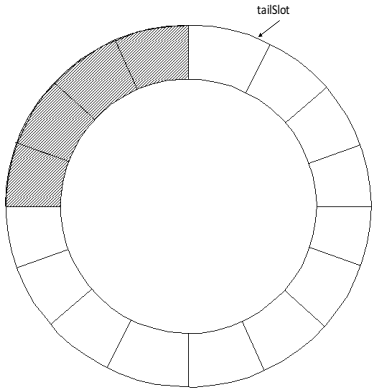


Figure 1.1: Circular Array

1.3.3 MCS Queue Lock

It is known that access to the core's local memory is faster than accessing to the local memory of another core. CLH algorithm results in thread spinning on remote locations in such architecture. MCS avoids the remote spinning. Same as CLH, MCS maintain a queue. However, it is an explicit linked list. The tricky part of MCS is in the *unlock()* function. When a thread *p* exits the CS, it checks if there is any node linked next to its node. If there exits such node, then it makes the *locked* false and removes its node from the linked list. When it finds no node linked next to it, there can be two cases for this. First, no thread has been inserted yet. Therefore, we can find that *tailNode* still points to *myNode*. It can be simply reset to null. In the second case, thread *p* waits until the next is not null. Then it can set the locked field for that node to be false as usual. This is shown in Algorithm 6.

References

[1] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 9: September 22

Lecturer: Vijay Garg *Scribe: Hung-Ming Hsu*

9.1 Agenda

Today's lecture covered the following topics:

- Puzzle Dr. Garg provided last lecture
- Using semaphore to solve two types of synchronization problems: *Mutual Exclusion* and *Conditional Synchronization*
- Using Monitor to solve two types of synchronization problems
- Case studies (code is on course Github)

9.2 Puzzle: merge two sorted array

Puzzle: Given two sorted arrays, A and B, try merging two arrays into one sorted array C in parallel. The faster, less work, the better.

To solve a parallel problem, the first step is to find a sequetial answer. Based on insights obtained from the sequential answer, we understand more about the nature of the problem and can build a parallel algorithm more easily.

9.2.1 Sequential Algorithm

1. Put pointers at the beginning of both A and B.
2. Compare two numbers pointed by the pointers.
3. Copy the smaller number to C, and move the pointer pointing to the smaller number one step forward.
4. Continue *step 2, 3* until all numbers are copied to C. C is sorted.

The time and work complexity are both linear.

- $T(n) = O(n)$
- $W(n) = O(n)$

Now, let's try to construct a parallel algorithm.

9.2.2 Parallel Algorithm 1

1. Pick median of A as pivot. Split A into two arrays at the pivot.
2. Use binary search on B to find where the value of pivot lies in B. Split B into two arrays, one with all numbers smaller than pivot and one with all greater.
3. Now we have to merge two smaller arrays on each side of the pivot. Recursively apply *step 1 and 2* to solve the problem.

Note that when array size is 1, the trivial case can be handled easily and recursion ends. The depth of recursion is $\log_2 n$. At each level a binary search is performed, so time complexity is $O(\log n \log n)$. Total work being done is the time complexity of serial version of it, which is $O(n)$ [W1]; this is optimal because n elements need to be copied into C.

- $T(n) = O(\log n \log n)$
- $W(n) = O(n)$

Let's see if we can be faster.

9.2.3 Parallel Algorithm 2

Define $Rank(x)$ to be total number of elements in both A and B smaller than x.

$$Rank(x) = |\{a \,|\, a \in A \ \wedge \ a < x\}| + |\{b \,|\, b \in B \ \wedge \ b < x\}|$$

$Rank(x)$ essentially is the position of x in C.

1. Calculate $Rank(A[j])$, which is j plus number of elements in B smaller than $A[j]$. This can be computed by binary search.

Step 1 can be computed for each element in A and B in parallel. Thus, the time complexity of this algorithm is $O(\log n)$, and total work is $O(n \log n)$ because total n elements are computed.

- $T(n) = O(\log n)$
- $W(n) = O(n \log n)$

We want to see if we can do optimal work while achieving good time complexity. So the puzzle continues ...

9.3 Semaphore

We have been talking about using algorithms busy waiting for entering critical sections. This is good when critical section is short and quick to exit, but sometimes we do quite a bit of computation, updating data structure, or even waiting for other resources in a critical section. In these cases, it is desirable for other threads to inactively wait in a queue and save CPU time instead of actively busy waiting whenever they get change to be executed.

Semaphore implements such semantics. As we learned in OS course, whenever a process/thread calls $V()$ function, it is moved to an inactive queue, called *blocked*, and essentially goes to sleep until someone else wakes it up by calling $P()$ function to the same semaphore.

Code 1: semaphore example 1

```
void run(int tid) {
    semaphore init true;
    mutex P();
    Critical Section
    mutex V();
}
```

That is one usage of semaphore. There are two types of synchronization: *Mutual Exclusion* and *Conditional Synchronization*. Semaphore can suffice both use.

- **Mutual Exclusion** is to protect one critical section that only one thread can enter at any point of time.
- **Conditional Synchronization** is to synchronize progress of multiple threads at some point of execution.

For example, $T2$ $bar()$ need to be called after $T1$ $foo()$ finishes. Code 2 and 3 show how this can be achieved by semaphore.

Code 2: T1

```
semaphore init false;
foo();
V();
```

Code 3: T2

```
semaphore init false;
P();
bar();
```

However, semaphore has its own drawbacks. It is not easy to use - when acquiring and releasing multiple semaphores, order matters as it is easily getting deadlock if order is not taken care of. Also it is not intuitive; it is hard to tell a semaphore is used for mutual exclusion or conditional synchronization. Thus, *Monitor* is introduced.

9.4 Monitor

Monitor has the same functionality as semaphore, bit it is more intuitive to use. Inside a monitor can be seen as a critical section. At any time, at most one thread can be in the monitor. If a thread tries to enter a monitor but some other thread is currently in it, the thread is blocked until the thread inside exits. If a thread calls $wait()$, it is moved to waiting queue and exits monitor if it was in it. A thread can call $notify()$ to wake up a thread or $notifyAll()$ to wake up all threads in the waiting queue.

Note $notify()/notifyAll()$ is treated as a *hint*. A waken up thread still tries to enter and may fail again. Like in Code 4, it is always a good idea to check condition again before proceed.

Code 4: Monitor example 1

```
while(!B)    // B is some condition needs to be met on proceeding
    wait();
```

In Java every object can be used as a monitor. Use `synchronized` keyword to specify which part of the code is inside the monitor. Code 5 provides some example use. *ReentrantLock* is a specific class that is used as a monitor, and conditional variable can be defined for a ReentrantLock instance.

Code 5: Monitor example 2

```
class foo {
    public void synchronized bar1() {
        // critical section 1
    }
    public void bar2() {
        // non-critical section
        synchronized(this) {
            // critical section 2

            wait(); // release monitor and wait in queue
                    // need to re-acquire monitor again if return
                    // because it is still in CS
        }
        // non-critical section

        notify();   // wake up one thread in the waiting queue
    }
}
```

9.5 Case Studies

9.5.1 BoundedBufferMonitor.java

There is a very common scenario in synchronization study that there are one producer who is generating some data and put into a shared buffer and one consumer who is consuming the data in the buffer. Since the buffer is shared, both people cannot access (update) the buffer simultaneously. One has to wait if the other is doing his job in the critical section or the buffer is full or empty for producer or consumer respectively.

Now this code cannot work if there are multiple producers or consumers because there is chance deadlock could happen when the buffer is full and a producer notifies another producer to wake and vice versa.

9.5.1.1 MBoundedBufferMonitor.java

This deadlock problem can be solved by dividing the unified queue for producers and consumers into two separate queues. For producers, they always notify consumer queue; and for consumers, they always notify producer queue.

9.5.2 BCell.java

Imagine some algorithm employs this parallel swapping mechanism. Each thread acquires the lock for itself and then the lock for target cell and swaps the values. If two threads are swapping with each other and both of them acquire their own lock but not the lock of each other yet, they result in deadlock.

9.5.2.1 Cell.java

To solve this problem, one needs to make sure for any pair of locks a predefined order of acquiring locks is always observed. In this case, the deadlock will never happen because there is only one way to acquire these two locks, not the other way. In the code, the order is determined by the order of hash code of the object, i.e., every lock gets its own id and the lock with smaller id is the first to acquire.

9.5.3 DiningMonitor.java

The dining philosopher problem is well known. I will refer to *Wikipedia* [W2] for more detailed explanation. Note the program in `DiningPhilosopher.java` would result in deadlock.

9.5.4 NestedMon.java

Nested monitor program never works. Because inside the first monitor there is at most one thread, if the thread blocks on the second monitor, it does not release the first monitor and nobody can enter and wake him up.

References

[W1] Parallel merge. In *Wikipedia*. Retrieved September 29, 2016, from https://en.wikipedia.org/wiki/Merge_algorithm#Parallel_merge

[W2] Dining philosophers problem. In *Wikipedia*. Retrieved September 29, 2016, from https://en.wikipedia.org/wiki/Dining_philosophers_problem

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 1: September 22

Lecturer: Vijay Garg *Scribe: Chuiye Kong*

1.1 Introduction

In this lecture, we first go over the puzzle of merging algorithm, then we discuss Monitor. Outline is shown as follows:

1. Sequential and two parallel solutions for Merging algorithm (puzzle)
2. Synchronization
3. Monitor
4. Problems using monitor

1.2 Merging Puzzle

As shown in Figure 1.1, the problem is to merge two sorted array into one sorted array.

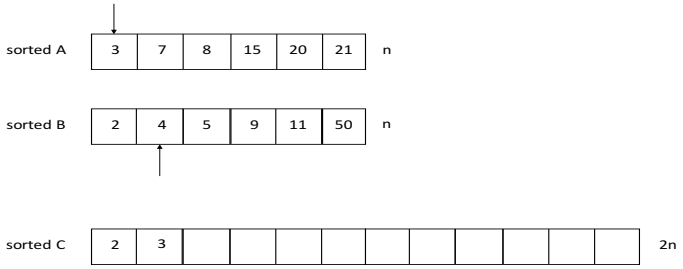


Figure 1.1: Merging Sorted Arrays

The sequential solution is using two pointers to point each array and compare the two pointed array items. The smaller item will be put into array C and its point will move to next element in the array. The timing complexity T(n) of this solution is O(n) and the work W(n) is O(n) as well.

1.2.1 Parallel Solution I

Select a key element in A (for example, we can choose 15), then array A will be split into left and right, which are annotated as A.left and A.right in Figure 1.2. Do the binary search for 15 (the key element we selected) in array B. B

will be divided up into the <15 part and >15 part, and allocated on the left side of the key element and on right side of it. The time complexity T(n) is O(log(n)log(n)) and work is O(nlog(n)log(n)).

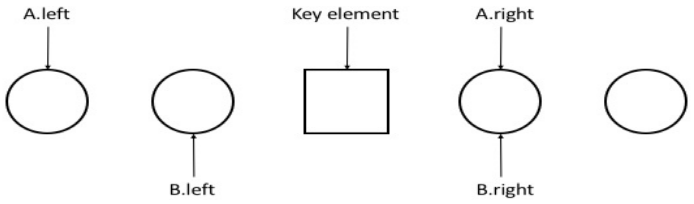


Figure 1.2: Binary Search for Key Element

1.2.2 Parallel Solution II

Now, we define rank(x), which is equal to the sum of the number of elements in A less than X and the number of elements in B less than X. For example, if we are looking for rank(A[j]), then the answer is j + binary search to find rank of A[j] in B, where j is the index in A as shown in Figure 1.3. The timing complexity of this algorithm is O(log) and the work is O(nlog(n)).

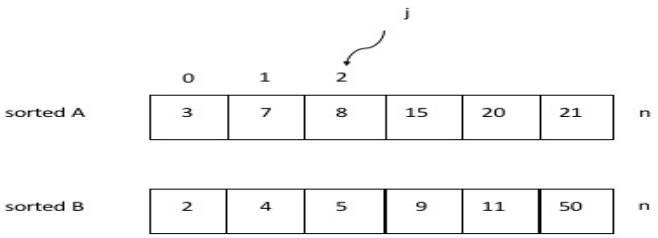


Figure 1.3: Rank example of A[j]

1.3 Synchronization

As far as we know, Synchronization can be used for mutual exclusion and conditional synchronization. To achieve mutual exclusion, waiting thread deploys two methods, which are busy-wait and sleep. Conditional synchronization is needed when a thread must wait for a certain condition to become true. Example is shown in Figure 1.4. We can see foo() has to do v() before T2 go to bar(). The drawbacks of using semaphore to achieve synchronization

can be summarized into two points. First, it is hard to tell the difference between mutual exclusion and conditional synchronization. Secondly, the order of acquire() matters, as it may cause deadlock (refer back to Semaphore lecture). It is not very reasonable to let programmers take care of this ordering problem. Therefore, we will introduce Monitor in the next section.

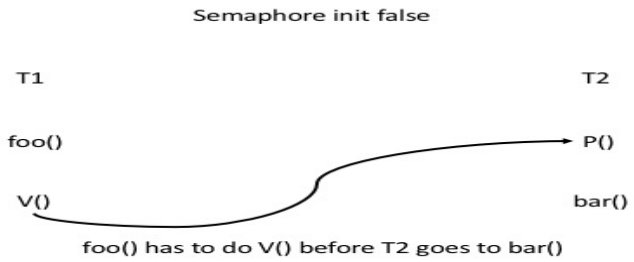


Figure 1.4: Conditional Synchronization Example

1.4 Monitor

We can use Monitor to replace Semaphore for both mutual exclusion and conditional synchronization purposes. There are two types of Monitors we can use, which are synchronized and reenetrant lock. A conditional variable has two operations: *wait* and *notify* (also called *signal*). In Java, the threads usually wait for the condition as shown in Algorithm 1.

Algorithm 1 Conditional Wait.

```
1: while (!B) {
2:         wait();
3: }
```

notify is used to wake up one thread and *notifyAll* is used to wake up all threads in the waiting queue. *Wait* inserts the thread in the wait queue.
Monitor Invariant is defined as: at most one thread can be in the monitor at any given time. A example of monitor is shown in Figure 1.5.

Wait queue and condition queue may both have ready thread that can enter the monitor. The problem here is: which thread should continue after the notify operation. There are two possible answers:

1. One of threads that was waiting on the condition variable continues execution. Monitors that follow this rule are called *Hoare* monitors (or, *signal-and-wait monitors*).

2. The thread that made the notify call continues its execution. When this thread goes out of the monitor, then other threads can enter the monitor. This is the semantics followed in Java and is called *signal-and-continue*.

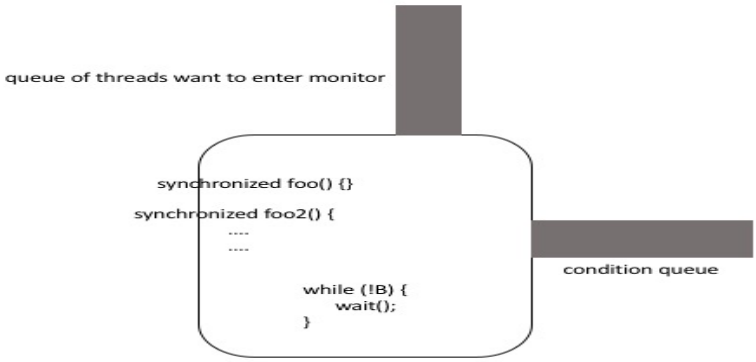


Figure 1.5: Monitor Example

1.5 Problems Using Monitor

1.5.1 Producer and Consumer

The buffer is shown in Figure 1.6. The BoundedBufferMonitor is used for one producer and one consumer, which has two entry methods: *deposit* and *fetch*. If a thread is executing the method *deposit* or *fetch*, then no other thread can execute *deposit* or *fetch*. The code of BoundedBufferMonitor is shown in [1]. This code works correctly only when there is exactly one producer and one consumer. The modified MBoundedBufferMonitor is designed to solve this issue and it can be found at [2].

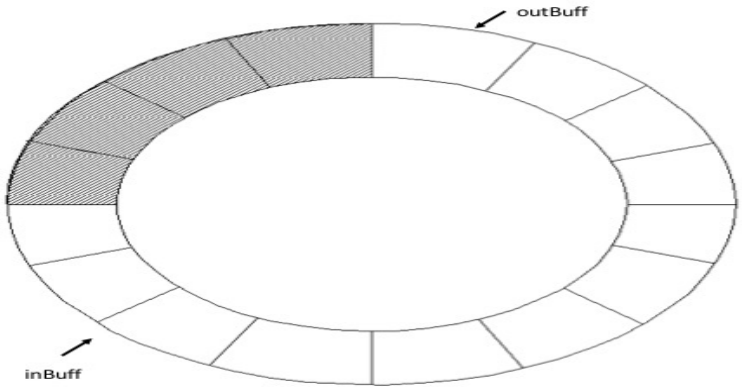


Figure 1.6: Producer and Consumer

1.5.2 Dining Philosophers

The code for using Monitor to solve Dining Philosophers is demoed in [3]. The code guarantees the mutual exclusion and deadlock-free, however, it does not guarantee freedom from starvation.

1.5.3 Nested Monitor

Nested monitor can cause deadlock problem when a thread waits for a condition inside a nested monitor. For example, a thread t1 is inside a monitor for an object *object1* and calls a synchronized method for another object *object2*. If the thread invokes a *wait()* inside it, then it will release the lock on *object2*. However, it will continue to hold the monitor lock of *object1*. If we have a thread t2 can notify thread t1, but thread t2 needs the lock of *object1*, a deadlock appears.

References

[1] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/blob/master/chapter3-synchronization_primitives/BoundedBufferMonitor.java

[2] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/blob/master/chapter3-synchronization_primitives/MBoundedBufferMonitor.java

[3] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/blob/master/chapter3-synchronization_primitives/DiningMonitor.java

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 10: September 27

Lecturer: Vijay Garg *Scribe: Shriya Nair*

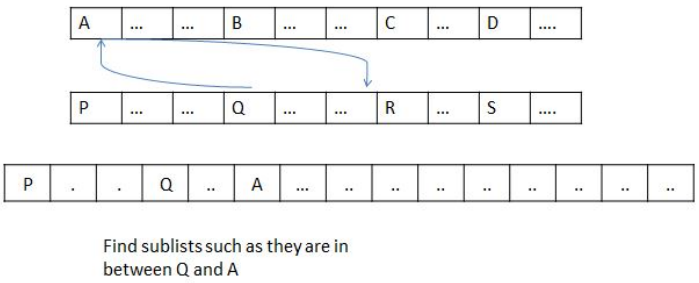
10.1 Puzzle: Sort 2 arrays

Description:
Two sorted A and B of n elements each have to be merged into a sorted array C

Solution 1: Sequential Algorithm
-Time = O(n)
-Work = O(n)
-using two pointers

Solution 2: Parallel Algorithm
- select an element, find its rank (index + position from binary search)
- Time = $O(\log n)$
- Work = $O(n \log n)$

Solution 3: Cascaded Algorithm - the Work-Optimal Solution
-Divide n array into n/log n groups each of size log n.
Number of splitters = $(n/\log n)$
Time taken to search for a splitter = $O(\log n)$
Arrows will never cross because both arrays are sorted



- Fill in only splitters
- Find sublists between and such that they are between Q and A
Max number of sublists = $O(n/\log n)$
Size of each list = $\log n$
Two lists of logn size can be merged in $O(\log n)$

step	T(n)	W(n)
Rank	$O(\log n)$	O(n)
Merging	$O(\log n)$	$O(n/\log n.\log n) = O(n)$
total:	$O(\log n)$	O(n)

10.2 New Puzzle

Problem: Parallel Prefix sum
Description: Consider an array on size n not necessarily sorted. Compute the recurring sum. (also called scan)
Example: 1,2,3,4,5,6
Output : 1,3,6,10,15,21

Solution 1: Sequential Algorithm
T(n) = O(n)
W(n) = O(n)

10.3 Consistency Conditions

- cannot use locks on datastructures- expensive, sequentializing.
- limiting parallelization of code.
Example:

```
---push(40)----  ----pop(?)----
      ----push(30)----
```

When an element is popped will it return 40 or 30?
- Need a definition for consistency.

10.3.1 Sequential Consistency

- Defined by Lamport
Consider a method foo:

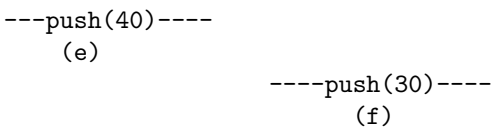
```
s.foo(arg1, arg2)
```

where..
1. s is the object
2. foo: name of the method
3. the statement is the invocation
4. arg1, arg2 are the arguments
5. s.response is the return value. Possible values: OK, value, exception

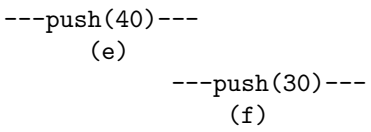
Consider two operations e,f.

Conventions:
inv(e) : invocation of e
resp(e): response generated after invoking e
proc(e) : process e is on

History of operations: (H, $<_H$) where $<_H$ is the real-time order.
Definition: e $<_H$ f = resp(e) occurred before inv(f)
Example:



Consider e, f :



e $\not<_H$ f && f $\not<_H$ e $\implies$ e is concurrent with f

Properties of the relation $<_H$:
1. Irreflexive : e $<_H$ f $\not\Rightarrow$ f $\not<_H$ e
2. Transitive : e $<_H$ f && f $<_H$ g $\implies$ e $<_H$ g
$\implies$ Asymmetric
Hence, (H,$<_H$) is a partially ordered set or poset

10.3.2 Process Order

e $<_H$ f are in process order iff:
proc (e) = proc (f) and resp(e) occurred before inv(f)

$<_{po} \subseteq <_H$
e $<_{po}$ f $\implies$ e $<_H$ f but the reverse may not be true

10.3.3 Sequential History

A history (H, $<_H$) is sequential if $<_H$ is a total order.
A poset (X, <) is a total order if $\forall$ distinct (x,y) belongs to X, (x $<_H$ y) or (y $<_H$ x)

Legal Sequential History:
A sequential history S is legal if it satisfies sequential specifications of the objects.

References

[1] Vijay K Garg, Introduction to Multicore Computing

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 10: September 27

Lecturer: Vijay Garg *Scribe: Prateek Srivastava*

10.1 Introduction

This lecture covers following items.

- Project outlines
- Merging Puzzle
- Consistency conditions

10.2 Projects

Projects need to be done in groups of 2 or 3. Undergrads should do project with other undergrads while grads with grads. For undergrads, aim would be to implement and turn in just one plot of the observations. For grads, they need to pick a topic, implement it and possibly improve on it. Topics can be chosen from recent conference proceedings as well.
If undergrads choose same topics, they can compete among themselves for the performance. Topics are posted on canvas. A brief overview is present here.

- Mutex algorithms not covered in class can be implemented e.g. colored bakery.
- Monitors with additional feature like abort.
- Parallel work scheduling and work distribution algorithms using some language e.g. language CILK.
- Concurrent trees, queues and skiplist. Avoids serialization by locking.
- Concurrent Hash-tables like Cuckoo Hashing, and Hopscotch Hashing.
- Poset, lattice theory. some part will be covered in class.
- Graph shortest path, spanning trees, max flows can be implemented on stampede on GPU.
- Implement sorting and compare performance.
- Image processing on GPU, scientific computation like fft and polynomial. Data mining algorithms like K-means.
- Text analysis used heavily by Google, pattern matching.
- Iphone/Android for this topic need to give exact details on what the app is going to do.

- Run model checker on algorithm and verify correctness.

- Lamport's Temporal Logic of Actions for specifying and verifying the correctness of concurrent algorithms.

- Verification of consistency conditions.

The project should contain some references.

10.3 Merge Puzzle contd...

10.3.1 Problem

Merge two sorted arrays of length n to form a sorted array of length 2n.

10.3.2 Solutions

(a) Sequential algorithm was like the merge sort.
(b) Parallel algorithm computes rank for each element which is the index into the final merged array. Each element computes its rank, using its own position and its position in the other array using binary search. Following table summarizes the time and work complexity of already discussed solution.

	Time	Work
Sequential	O(n)	O(n)
Parallel	O(log n)	O(n log n)

10.3.3 Work optimal parallel solution

We would like to reduce the work from $O(n * logn)$. This can be achieved using cascaded algorithm.
Divide the input array in log n groups. The starting element in each group is called the splitter.
No. of splitters $= O\left(\frac{n}{logn}\right)$

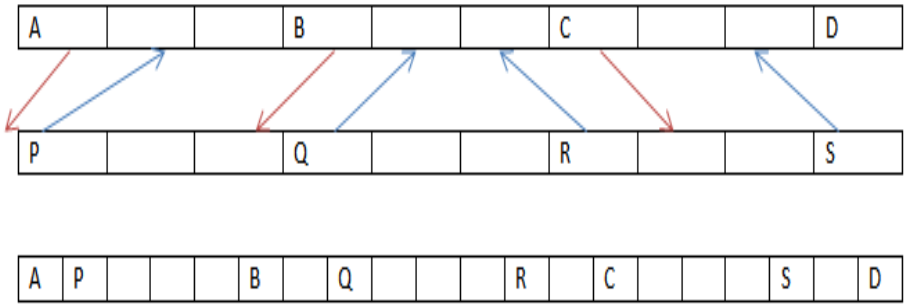


Figure 10.1: Merging Sorted Arrays

Fill the splitter in the target array by finding ranks as done in the parallel algorithm as shown in the figure
Find the sublist in first list say, α and second list say, β such that they are in between the splitters.
Number of such lists = $O\left(\frac{n}{logn}\right)$
Size of each list = $Ologn$)
We know that the $logn$ size lists can be merged in $Ologn$) using the sequential algorithm.

Following table summarizes the steps. Hence, the solution is work as well as time optimal. Although, there

	Time	Work
Step 1	O(log n)	O(n)
Step 2	O(log n)	O(n)
Total	O(log n)	O(n)

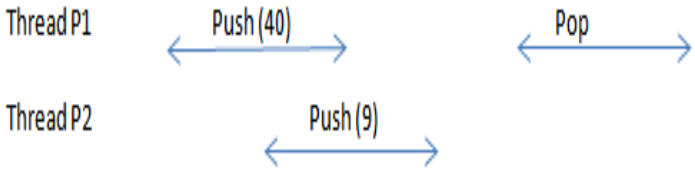
are solutions optimal then this one as well.

10.4 Parallel prefix sum puzzle

Given an array find an output array such that each element in output array is sum of input array elements till that index.
e.g. Input 3, 4, 19, 11, 13
Output 3, 7, 26, 37, 50
The algorithm is called scan.

10.5 Consistency condition

Consider a stack with following operations
push(40) push(9) pop (Pop should return 9 not 40)
This is sequential correctness
Now consider following situation.



The correctness of output depends on the definition of correct output. Lamport wrote a 2 page paper explaining what it means to be sequentially consistent in a multiprocessor environment.

10.5.1 Notations

A method call is split into two events

* *Invocation* method name + args e.g. f.foo(arg1, arg2)

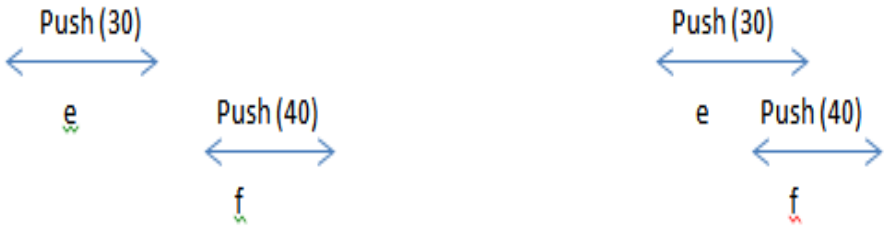
* *Response* result or exception

Consider, P f.foo(arg1, arg2)
This is an invocation of object f on thread P
foo is method name
arg1, arg2 are arguments
Consider, P f.response
This is the return value, for object f on thread P

We define following
inv(e) is invocation of e
resp(e) is response of e
proc(e) is process on which e runs

10.5.2 History

A sequence of invocations and responses constitutes history.
History $(H, <_H)$ is set of operations in real time order.
The relation $e <_H f$ holds, if resp(e) occurred before inv(f)



In first case, response of e occurs before invocation of f. In second case, e overlaps with f or is concurrent with f. There is no particular order

The relation $<_H$ is
(a) irreflexive (each element does not satisfy the relation with itself)
(b) transitive (if relation applies between successive members of a sequence, it must also apply between any two members taken in order)
This two conditions imply that the relation is asymmetric.
Since the relation $(H, <_H)$ is irreflexive and transitive, $(H, <_H)$ is a partial ordered set (poset).

10.5.3 Process Order

$e < f$ is in process order if

proc(e) = proc(f)

resp(e) occurred before inv(f)

10.5.4 Total order

A poset $(H, <_H)$ is a total order if for all distinct x,y $(<_X)$, satisfies $(x < y)$ or $(y < x)$

10.5.5 Sequential History

A history $(H, <_H)$ is sequential if $(<_H)$ is a total order.
A sequential history is legal if it satisfies sequential specs of the objects.

References

[1] V. K. Garg, Introduction to Multicore Computing

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 11: September 29

Lecturer: Vijay Garg *Scribe: William Mills*

11.1 Exam 1 Study Tips

Exam 1 will only cover up-through Lecture 11 (today) There will be a problem from the seminar Some sample problems have been posted on Canvas. It is advised that you follow this order in your studying:

1. Attempt the problems on your own
2. Compare your solutions with your peers
3. Solutions will be posted next week to check for correctness

Additionally, Homework 3 has been posted, and some of its questions will be good preparation (but some of it has not been covered yet) The material to best-prepare you, in descending order:

1. Material covered in both Lecture and Handouts
2. Material covered in Lecture only
3. Material covered in Handouts only
4. Past Assignments

11.2 Concurrent History Equivalence

Recall that a concurrent History $(H, <_H)$ is a sequential history $(S, <_S)$ if $<_S$ is a total order. Two histories are *equivalent* if they are composed of the same operations.

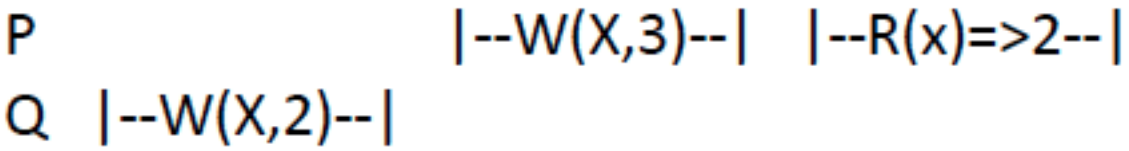


Figure 11.1: H1

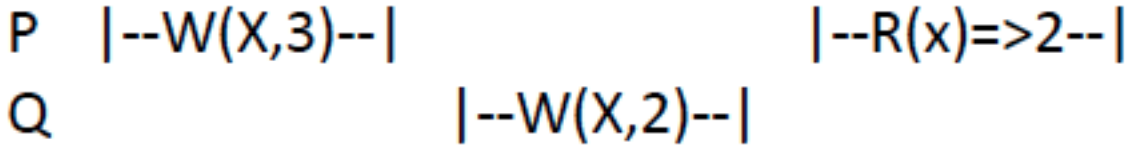


Figure 11.2: H2

Observe that `H2` is simply a permutation of the operations in `H1`, but `H1` is not identical to `H2`. These two histories are equivalent.

11.3 Sequential Consistency

A history $(H, <_H)$ is *sequentially consistent* if there exists a legal sequential history `S` such that:

1. `S` is equivalent to `H`
2. `S` preserves the process order in $(H, <_H)$

Both `H1` and `H2` from the previous section are sequentially consistent. A legal sequential history for either of them would be

$P : W(X, 3), Q : W(X, 2), P : R(X) => 2$

Now consider History `H3`:

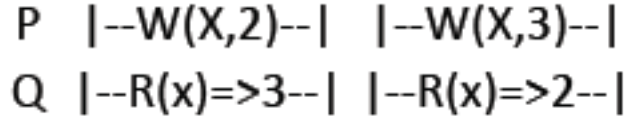


Figure 11.3: H3

`H3` is NOT sequentially consistent. This can be shown by adding arrows that represent the *happened-before* relationship. When a cycle is identified, we know the history is not legal and thus cannot be sequentially consistent. When we place these edges in `H3`, we can see that a cycle exists when we follow the *happened-before* relationship:

$P : W(X, 2), \underline{Q : R(X) => 2, PW(X, 3), Q : R(X) => 3, Q : R(X) => 2}$

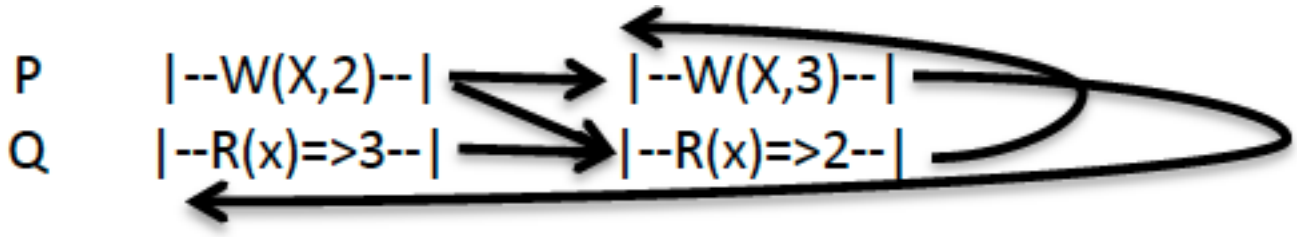


Figure 11.4: H3

11.4 Atomic Consistency (Linearization)

A history $(H, <_H)$ is linearizable/atomically consistent if there exists a legal sequential history `S` such that

1. `S` is equivalent to `H`
2. `S` preserves $<_H$
    - That is, responses occurring before invocations in `H` must preserve that relationship in `S`

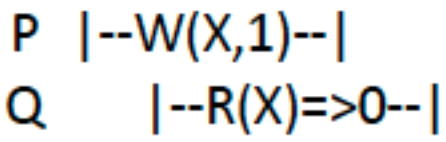


Figure 11.5: H4

Figure 11.6: H5

Thus, `H4` is linearizable, while `H5` is not.

11.4.1 Linearization Points

To help determine if a history is linearizable, We can think of each operation having a single moment where its effect takes place, and try to find an order for them. We call such moments *linearization points*. Theorem: `H` is linearizable if there exist linearization points for all operations such that the resulting sequence is legal.

11.5 Composability (Locality Property)

Linearization is arguably a better tool for software applications, because it has the *locality property*, which says the concurrent history is legal when multiple objects are considered.

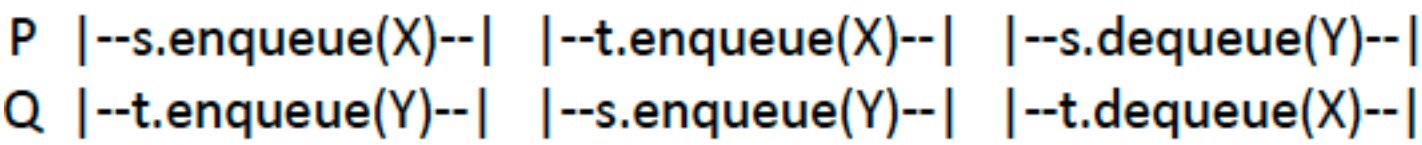


Figure 11.7: H7

Note that `H7` is linearizable when all of object t's operations are ignored. The same holds when all of s's operations are ignored. However, when the two are combined, we see that there is a cycle the same way we did before.

$P : s.enqueue(X), t.enqueue(X), t.enqueue(Y), s.enqueue(Y), s.enqueue(X)$

When a history `H` is projected on an object x, we write the sub-history as $H|x$. Note that $H|s$ and $H|t$ are both sequentially consistent, but `H` is not, because there is a cycle in its $<_H$ trace.

11.5.1 Linearization and the Locality Property

By definition, a consistency condition `CONSISTENCY` satisfies composability if:

$\forall x, H|x satisfies \texttt{CONSISTENCY} \Leftrightarrow H satisfies \texttt{CONSISTENCY}$

Linearizability can be shown to satisfy composability from this definition. It is trivial to show that $H is linearizable \rightarrow \forall x, H|x is linearizable$, so we will focus on the proof for the other direction.

As before, we will place arrows on our history to represent the happened-before relation. The claim is that adding these arrows will maintain an acyclic representation of the history. These arrows can be drawn for two reasons:

1. $<_H$ enforces it
2. $<_x \forall x$

We know arrows from `1` cannot form a cycle because they are enforced by the poset they come from. It remains to be shown that traces including both types of arrows cannot introduce a cycle. We assert that such subpaths including $\texttt{1} \rightarrow \texttt{2} \rightarrow \texttt{1}$ can be re-written as a single arrow of type `1`.

$e <_H f$ – response(e) happened before invocation(f). $f <_x g$ – response(e) happened before invocation(f). $g <_H h$ – response(e) happened before invocation(f).

Transitively, we know that $c <_H h$ holds.

11.6 All Concurrent Histories

There are many other types of histories looser than sequential consistency that we did not discuss. Realize that the weaker your consistency rules are, the better performance you can achieve with the same hardware.

For this class, we will stick to atomic consistency

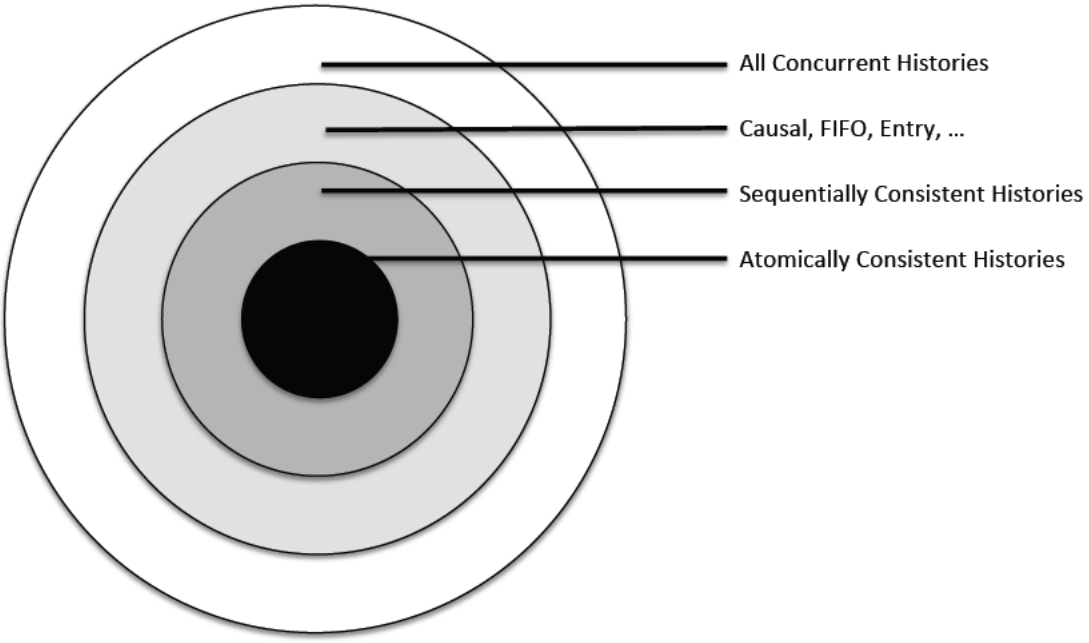


Figure 11.7: Concurrent History Strengths

References

[1] V. K. Garg, Introduction to Multicore Computing

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 11: September 29

Lecturer: Vijay Garg *Scribe: Shruti Subramanian*

11.1 Introduction

Exam Overview, Sequential Consistency, Linearizability, Composability,

11.2 Exam 1 Overview

Exam in class on Thursday 10/6. This is the last day of material for Exam 1. The exam is closed book.

11.2.1 What to Know

The puzzles handout which covers basic parallel algorithms.

Know the meaning of the basic constructs for OpenMP.

Syntax is not important and psudeocode is okay EXCEPT for Condition Variables/Semaphores/Reentrant Locks

HW3 has been posted. The first 3 questions are relevant to the first exam.

Review: Lectures + Handouts + Chapters 1 - 4 + Assignments

11.2.2 How to Study

1. Take sample exam posted online in 75 minutes.
2. Work with friends and discuss practice exam.
3. Check with the solutions posted online.

11.3 Overview

Rememeber that $(H,<_H)$ is the concurrent history.

Sequential history $(S,<_S)$: A history is $(S,<_S)$ is sequential if $<_S$ is total order (total order is from last time)

11.4 Equivalence

Two histories are equivalent if they are composed of the same operations. The processes in both concurrent

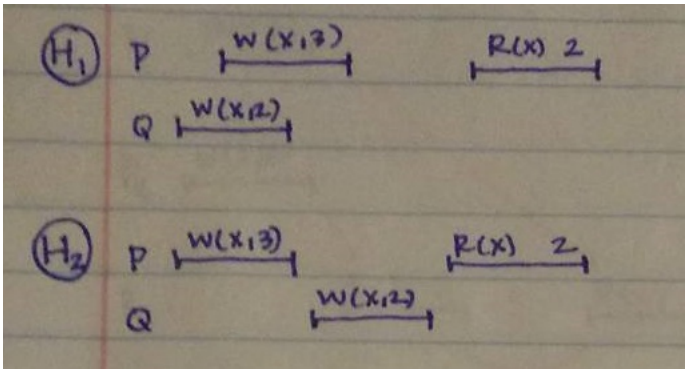


Figure 11.1: Two equivalent histories

histories are equivalent, but not identical since (H,$<_H$) is not the same. In this example the three operations are permuted.

11.5 Lamports Sequential Consistency

A concurrent history (H,$<_H$) is sequentially consistent if there exists a legal sequential history S such that: 1. S is equivalent to H 2. S preserves the process order in (H,$<_H$)

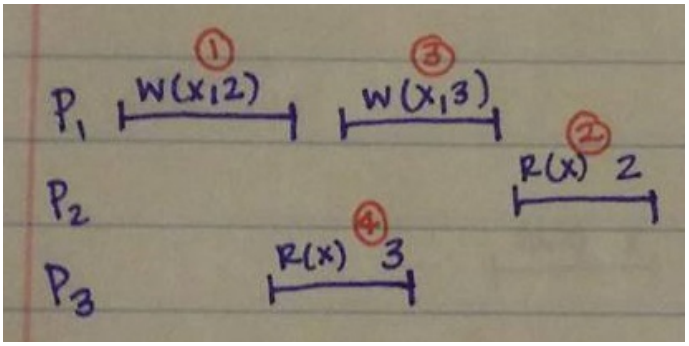


Figure 11.2: A sample sequential consistency

Order across processes do not matter. Both histories from figure 11.1 are sequentially consistent.

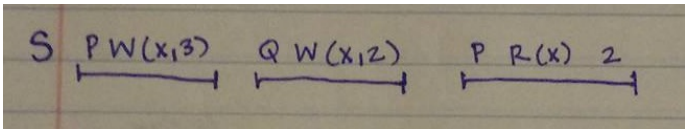


Figure 11.3: Sequential consistency of figure 11.1

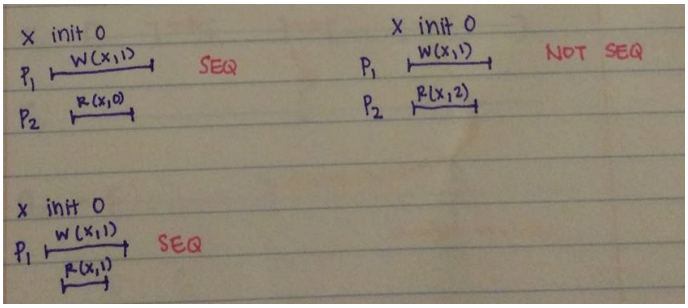


Figure 11.4: Sequential consistency example 1

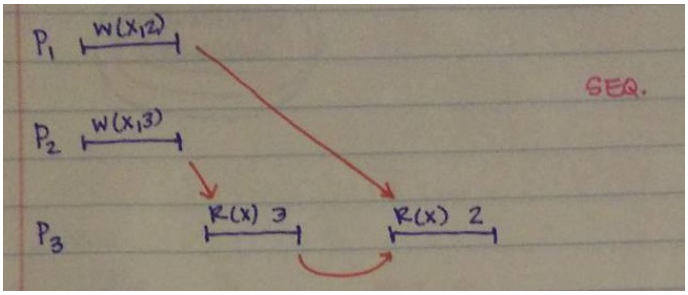


Figure 11.5: Sequential consistency example 2

11.5.1 Not Sequential Consistency

To show that it is not sequentially consistent, you show there is not any way to be sequentially consistent by either:

1. The brute force method where you try out every combination for a case analysis

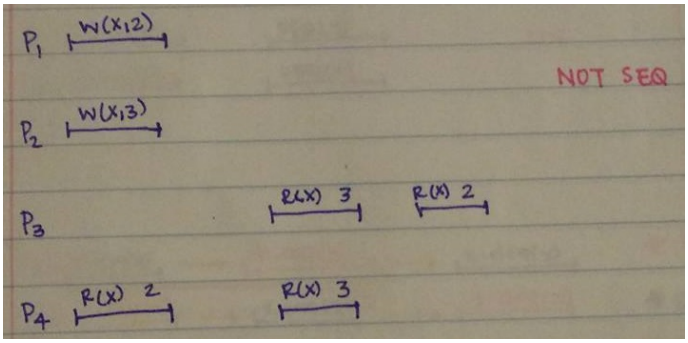


Figure 11.6: Not sequentially consistent

2. Check if there is a cycle such that the order cannot be preserved.

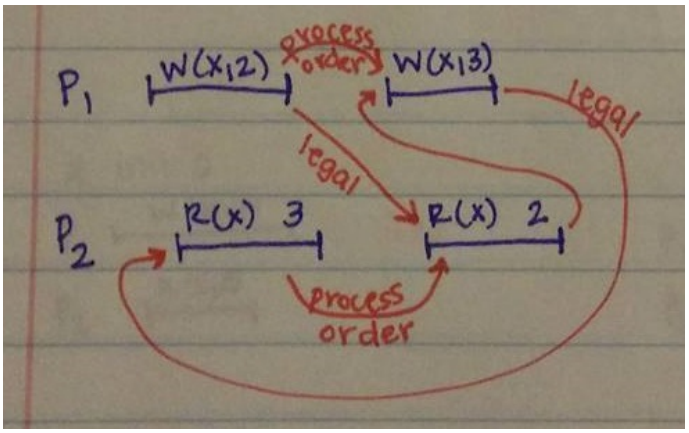


Figure 11.7: Not sequentially consistent cycle

11.5.2 Unfinished history

Extend the unfinished by finishing up the history for a complete history that it is sequentially consistent

11.6 Henlihy and Wings Consistency

This correctness is stronger than sequential consistency and thus better for software purposes. A history (H,$<_H$) is linearizable/atomically consistent if there exists a legal sequential history S such that: 1. S is equivalent to H 2. S preserves $<_H$(the real time order)

11.6.1 Linearizability and Sequential Consistency

Theorem: H is LIN $\rightarrow$ H is SEQ

Proof: e $<_{\text{process order}}$ f $\rightarrow$ e $<_H$ f

The process-order is realtime order on the same process.

11.6.2 Linearization Point

That gives legality without operations. Linearization points should not match and must have some order.

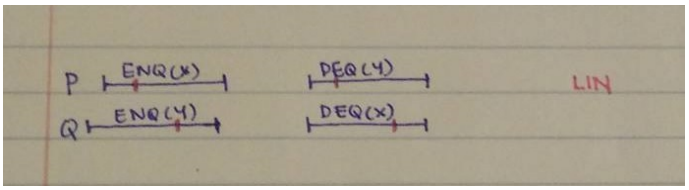


Figure 11.8: A linearizable history with linearization points

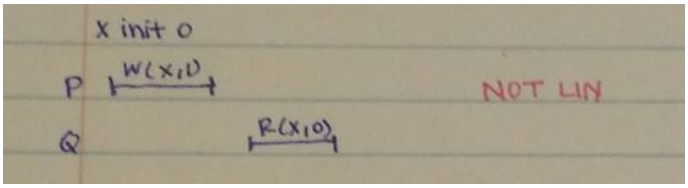


Figure 11.9: A non-linearizable history

S1 = enq(x), eng(y) deq(x) deq(y) S1 = enq(y), eng(x) deq(y) deq(x)

Theorem: H is linearizable iff there exists linearization points for all operations such that the resulting sequence is legal and preserves the real time order.

11.7 Linearizability and Sequential Consistency

Suppose we have a legal program for queues and/or stacks and we join them.

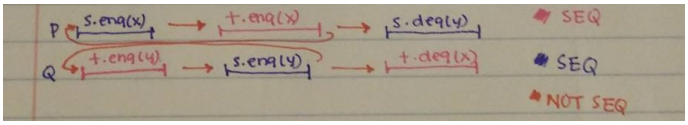


Figure 11.10: Illegal combination of two legal programs

Defn: A consistency condition CONSISTENCY satisfies composability/locatlity if for all x: H | x satisfies consistency $\longleftrightarrow$ H satisfies consistency

11.7.1 Linearizability and Composability

Theorem: LIN satisfies COMPOSABILITY

Proof:

Claim: The graph stays acyclic. Two types of arrows e $<_H$ f resp(e) occurred before inv(f) f $<_n$ g resp(f)

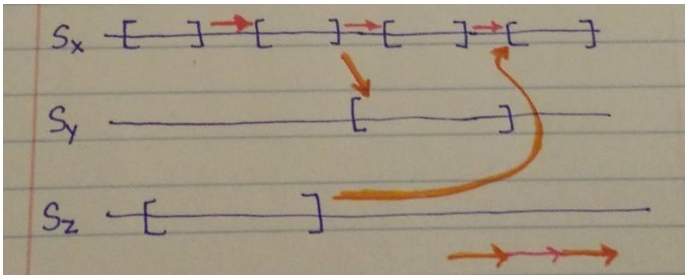


Figure 11.11: Linearizability cycle

occurred before inv(g) g $<_H$ h resp(g) occurred before inv(h) By transitivity $\rightarrow$ resp(e) occurred before inv(h) if any cycle $\rightarrow$ then to begin with thus no cycle Thus, with a cycle we are not sequentially consistent.

11.8 Other types of consistency

Causal consistency, FIFO consistency, Entry Consistency

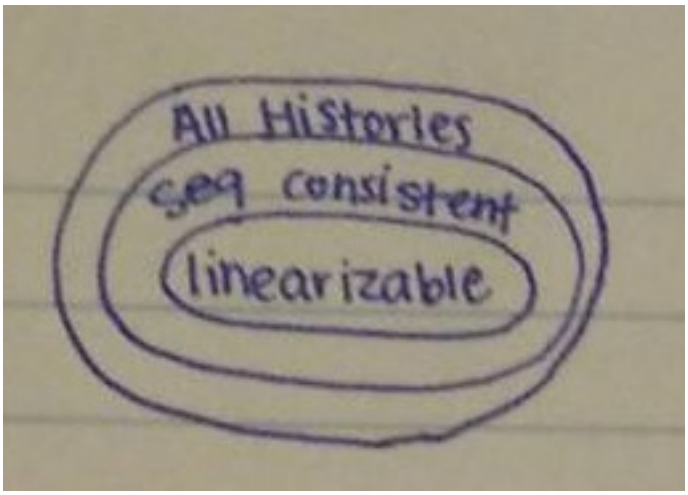


Figure 11.12: Consistency circles

11.8.1 Consistency Tradeoffs

Stronger consistency gaurantees ease of programs, however this causes a loss of performance.

References

[1] Vijay K Garg, Multicore Computing, The University of Texas at Austin. September 2016.

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 12: October 4

Lecturer: Vijay Garg *Scribe: Hui Dong, Mingzhou Jin*

12.1 Prefix Sum

Suppose we have an array $\{3, 1, 2, 1, 5, 4, 6, 9\}$, how to find the prefix sum?

Array	3	1	2	1	5	4	6	9
Prefix Sum	3	4	6	7	12	16	22	31

Sequential Algorithm: for the i^{th} iteration, we calculate the i^{th} prefix sum.
The time and work are $T(n) = O(n)$, $W(n) = O(n)$.
$a[0] = a[0]$
$a[1] = a[0] + a[1]$
$a[2] = a[0] + a[1] + a[2]$
...

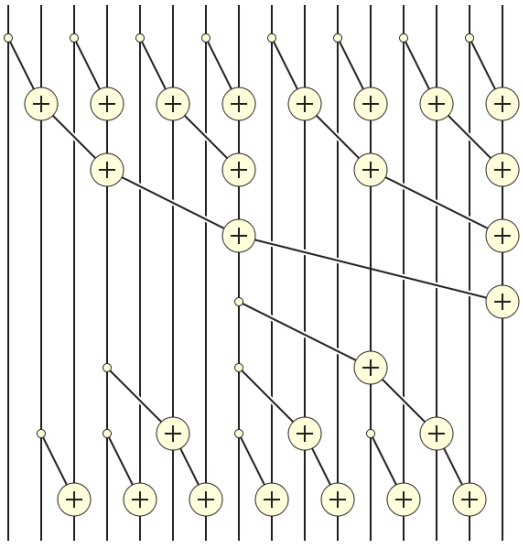
Could we do better? Could we calculate the prefix sum concurrently? Yes.
Add the value of entry which is at a distance of d in front of me, where at the i^{th} iteration $d = 2^i$.
$T(n) = O(log(n))$, $W(n) = O(nlog(n))$.

Array	3	1	2	1	5	4	6	9
d=1	3	4	3	3	6	9	10	15
d=2	3	4	6	7	9	12	16	24
d=4	3	4	6	7	12	16	22	31

Pseudo-code:

```
d = 1;
while (d < n) {
   for all i in parallel do
      if ( i >= d):
         a[i]: = a[i] + a[i-d];
   d = 2*d;
}
```

We can describe the procedure in three steps as follows:
1. Compute the sums of consecutive pairs of items in which the first item of the pair has an even index: z0=a0+a1, z1=a2+a3, ...
2. Recursively compute the prefix sum w0, w1, w2, ... of the sequence z0, z1, z2, ...
3. Express each term of the final sequence y0, y1, y2, ... as the sum of up to two terms of these intermediate sequences: y0 = a0, y1 = z0, y2 = z0 + a2, y3 = w0, etc. After the first value, each successive number yi is either copied from a position half as far through the w sequence, or is the previous value added to one value in the a sequence.



Circuit representation of a 16-input parallel prefix sum

12.2 Lock-free Synchronization

The synchronization mechanism that we have discussed so far are based on locking data structures during concurrent accesses.
Problems with locks:
1. Deadlock: each process is waiting for the other to release a needed resource
2. Prority Inversion: Possible solution–Priority Inheritance
3. Convoying: all the process need a lock A to proceed. However, a slower process acquires A first. Then all the other processes slow down.
4. Kill tolerant availability: a process holding the lock is killed, all other processes would wait forever
5. Preemption tolerance
6. Overall performance

A possible solution of not using locks in synchronization mechanism is called lock-free. If lock-free synchronization also guarantees that each operation finishes in a bound number of steps, then it is called wait-free. Lots of music sharing and streaming applications use lock-free data structures.
PortAudio, PortMidi, and SuperColliderPortAudio.
What we need in lock-free queues, stacks, linked-lists, and sets is Compare and Swap (CAS) primitive: boolean CAS(expectedValue, newValue). The pseudocode below executes atomically without interruption

```
if (expectedValue == compareValue) {
    expectedValue = newValue;
    return true;
} else {
    return false;
}
```

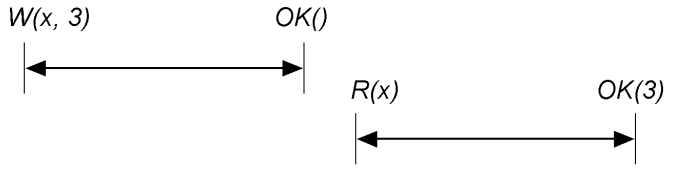
12.2.1 Pointer-swinging technique

In the object pointer-swinging technique, any thread that wishes to perform an operation on an object goes through the following steps:
1. The tread makes a copy of that object.
2. It performs its operation on the copy of the object instead of the original object

12.2.2 Safe, Regular, and Atomic Registers

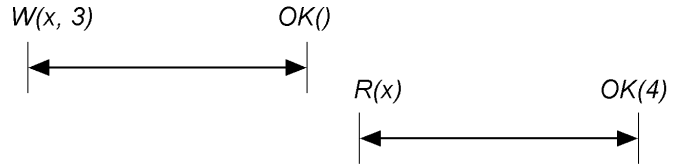
A register is `safe` if a read that does not overlap with the write returns the most recent value. If the read overlaps with the write, then it can return `ANY` value.

(a)



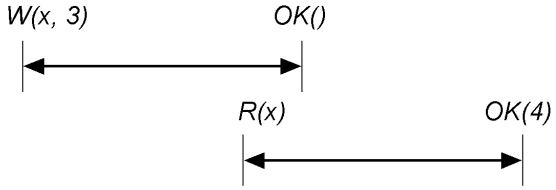
Safe: Read return the most recent write.

(b)



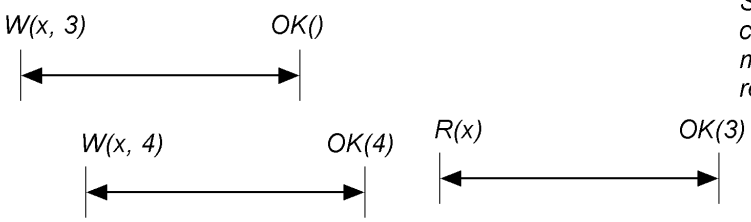
Unsafe: No overlap but read returns 4 which should be 3.

(c)



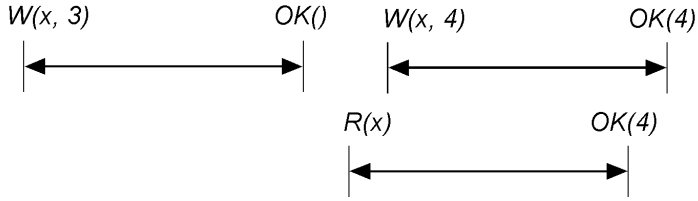
Safe: Read overlaps write and read can return anything.

(d)



Safe: W(x, 3) and W(x, 4) are concurrent. They are both the most recent writes. Read can return either one of them.

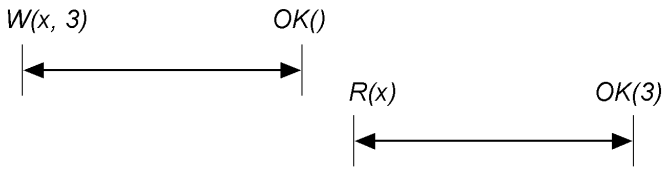
(e)



Safe: Read can return anything due to overlap with write operation.

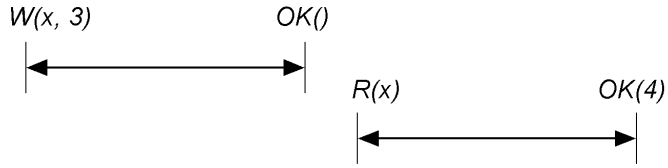
A register is `regular` if it is safe and when the read overlaps with one or more writes, it returns either the value of the most recent write that preceded the read or the value of one of the overlapping writes.

(a)



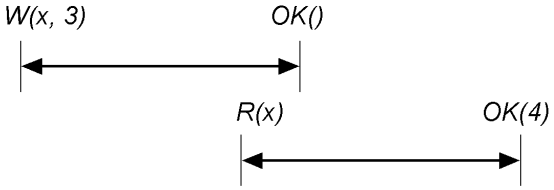
Regular: Read returns the most recent write

(b)



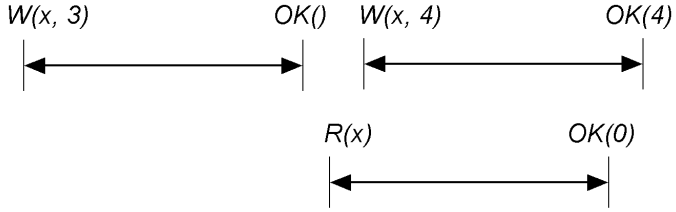
Irregular: it should return the most recent write, OR the overlapping write

(c)



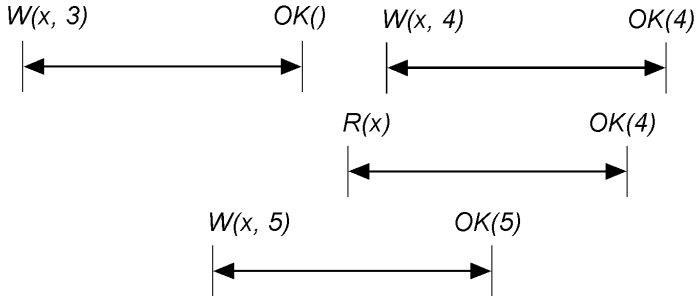
Irregular: stronger than safe which could return anything, regular has to return 3 in this case.

(d)



Irregular: the read should return either 3, the most recent write, or 4, the overlapping write.

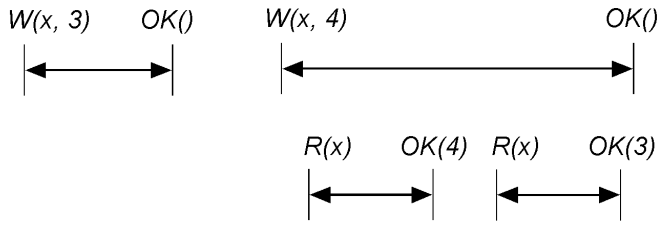
(e)



Regular: the read returns one of the overlapping writes.

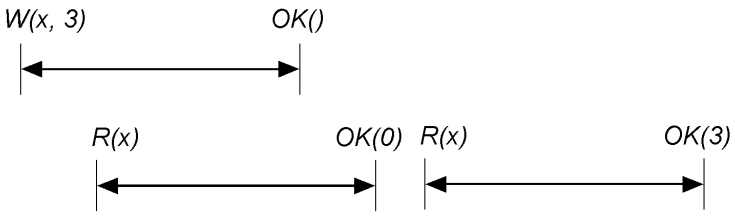
A register is `atomic` if its histories are linearizable.

(a)



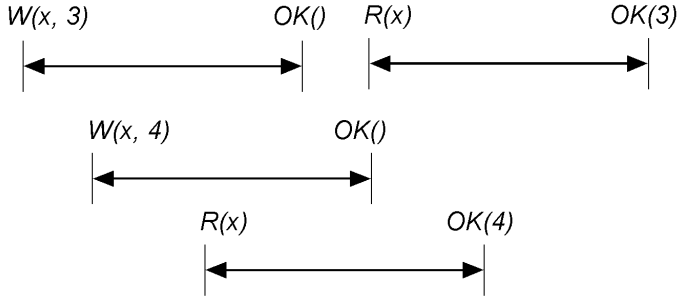
Regular: Read returns overlapping write (4) or most recent write(3). Not atomic: read returns a value (3) which is older than 4.

(b)



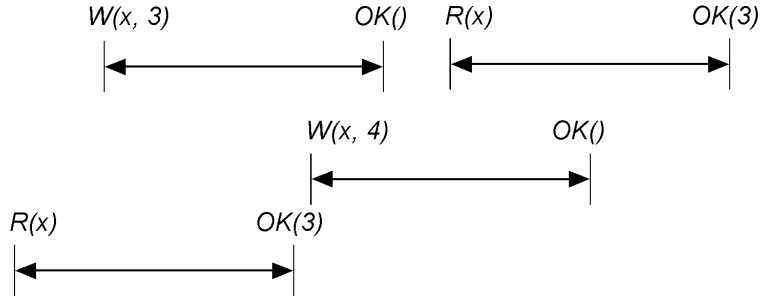
Atomic: Linearizable
R(x, 0), W(x, 3), R(x, 3)

(c)



Atomic: Linearizable
P2: W(x, 4), P3: R(x, 4),
P1: W(x, 3), R(x, 3)

(c)



Not atomic: Read in P1 has to return 4 because both writes in P1 and P2 have finished

Surprisingly, it is possible to build a multiple-reader multiple-writer (MRMW) atomic multivalued register from single-reader single-writer (SRSW) safe boolean registers. This can be achieved by the following chain of constructions:

1. SRSW safe boolean register to SRSW regular boolean register
2. SRSW regular boolean register to SRSW regular multivalued register
3. SRSW regular register to SRSW atomic register

4. SRSW atomic register to MRSW atomic register
5. MRSW atomic register to MRMW atomic register.

12.2.3 Regular SRSW Register

We will use the following Java code as an abstraction for a SRSW safe boolean register.

```
class SafeBoolean {
   boolean value;
   public boolean getValue() {
      return value;
   }
   public void setValue( boolean b) {
      value = b;
   }
}
```

The construction of a regular SRSW register from a safe SRSW is shown below:

```
class RegularBoolean {
   boolean prev; //not shared
   SafeBoolean value; //not shared
   public boolean getValue() {
      return vaule.getValue();
   }
   public void setValue( boolean b) {
      if ( prev != b) {
         value.setValue(b);
         prev = b);
      }
   }
}
```

References

[Vijay K. Garg] , Concurrent and Distributed Computing in Java, *John Wiley & Sons* (Jan 28, 2005)

[WikiPedia] , *https* : *//en.wikipedia.org/wiki/Prefix_sum*,

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 13: October 11

Lecturer: Vijay Garg *Scribes: Rohan Tripathi*

13.1 Concurrent Data Structures

There are 2 choices for building concurrent data structures :-

1. Lock Based
2. Lock Free

The **Lock Based** algorithms can be further divided into:

1. Coarse Grained
2. Fine Grained

The **Lock Free** algorithms can be further divided into:

1. Lock Free
2. Wait Free

Below is an example of a wait free structure:-

```
import java.util.concurrent.atomic.*;
class MyAtomicInteger extends  AtomicInteger {
    public MyAtomicInteger(int val) { super(val);}
    public int myAddAndGet(int delta) {
        for (;;) {
            int current = get();
            int next = current + delta;
            if (compareAndSet(current, next))
                return next;
        }
    }
    public static void main(String[] args) throws Exception {
        MyAtomicInteger x = new MyAtomicInteger(10);
        System.out.println(x.myAddAndGet(5));
    }
}
```

13.1.1 Stack

Stack is one of the simplest concurrent data strucutres in terms of implementation. Stack can be:

- Lock Based
- Lock Free

The 2 major operations associated with stack are *push*(*value*) and *pop*().

The below code represents a lock based Stack :

```
public class Stack<T> {
    Node<T> top = null;

    public synchronized void push(T value) {
        Node<T> node = new Node(value);
        node.next = top;
        top = node;
    }

    public synchronized T pop() throws NoSuchElementException {
        if(top == null) {
            throw new NoSuchElementException();
        } else {
            Node<T> oldTop = top;
            top = top.next;
            return oldTop.value;
        }
    }
}
```

To implement a lock free version of the stack we must make sure that the operations on the data structure are being perfomed atomically even though locks are not being used. A lock free implementation uses *AtomicReference* to achieve this.

```
public class LockFreeStack<T> {
    AtomicReference<Node<T>> top = new AtomicReference<Node<T>>(null);

    public void push(T value) {
        Node<T> node = new Node<T>(value);
                while(true) {
                   Node<T> oldTop = top.get();
                   node.next = oldTop;
                   if (top.compareAndSet(oldTop, node)) return;
                    else Thread.yield();
                }
    }

    public T pop() throws NoSuchElementException {
                while(true) {
                    Node<T> oldTop = top.get();
```

```
                    if(oldTop == null) throw new NoSuchElementException();
                    T val = oldTop.value;
                    Node<T> newTop = oldTop.next;
                    if(top.compareAndSet(oldTop, newTop)) return val;
                    else Thread.yield();
                }
    }
}
```

Speeding up Lock Free stack

The lock free implementation of stack can be made to run faster by adding some design changes. One of them can be $Cancellation - of - Operations$. For example, if a thread is executing a $push(30)$ operation and before the operation can terminate we get a request for $push(70)$ and $pop()$. The last 2 operations can cancel each other out since the effect of physically performing them would have been the same.

13.1.2 Queue

There are 2 types of queues :-

1. Bounded
2. Unbounded

The **Bounded** and **Unbounded** queues can be further divided into:

- Blocking
- Non Blocking

Below is an example of a Bounded Blocking Queue:

```
import java.util.concurrent.locks.*;

class MBoundedBufferMonitor {
   final int size = 10;
   final ReentrantLock monitorLock = new ReentrantLock();
   final Condition notFull  = monitorLock.newCondition();
   final Condition notEmpty = monitorLock.newCondition();

   final Object[] buffer = new Object[size];
   int inBuf=0, outBuf=0, count=0;

   public void put(Object x) throws InterruptedException {
     monitorLock.lock();
     try {
       while (count == buffer.length)
         notFull.await();
       buffer[inBuf] = x;
       inBuf = (inBuf + 1) % size;
       count++;
```

```
      notEmpty.signal();
    } finally {
      monitorLock.unlock();
    }
  }

  public Object take() throws InterruptedException {
    monitorLock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      Object x = buffer[outBuf];
      outBuf = (outBuf + 1) % size;
      count--;
      notFull.signal();
      return x;
    } finally {
      monitorLock.unlock();
    }
  }
}
```

Speeding up Queues

The speedup can be accomplished by having a separate lock for enqueue and dequeue operations.

```
import java.util.concurrent.locks.ReentrantLock;
public class UnboundedQueue<T> {

  ReentrantLock enqLock, deqLock;
  Node<T> head;
  Node<T> tail;
  int size;
  public UnboundedQueue() {
    head = new Node<T>(null);
    tail = head;
    enqLock = new ReentrantLock();
    deqLock = new ReentrantLock();
  }
  public T deq() throws EmptyException {
    T result;
    deqLock.lock();
    try {
      if (head.next == null) {
        throw new EmptyException();
      }
      result = head.next.value;
      head = head.next;
    } finally {
      deqLock.unlock();
    }
    return result;
```

```
  }
  public void enq(T x) {
    if (x == null) throw new NullPointerException();
    enqLock.lock();
    try {
      Node<T> e = new Node<T>(x);
      tail.next = e;
      tail = e;
    } finally {
      enqLock.unlock();
    }
  }
}
```

A special case of queue

The case of single producer and single consumer is faster since there is no need for synchronization and no need for CAS operattion. The enque operation only looks at the tail and dequeue only looks at the head.

```
public class SingleQueue { // Single Producer Single Consumer
  int head = 0;    // slot for get
  int tail = 0;    // empty slot  for put
  Object [] items;
  public SingleQueue(int size) {
    head = 0; tail = 0;
    items = new Object[size];
  }
  public void put(Object x) {
    while (tail - head == items.length) {}; //busywait
    items[tail % items.length] = x;
    tail++;
  }
  public Object get() {
    while (tail - head == 0) {}; // busywait
    Object x = items[head % items.length];
    head++;
    return x;
  }
}
```

It is not possible to write a code for multiple consumers without using a CAS operation.

13.1.3 Michael and Scotts Lock-Free Queue

1. ABA Problem
This problem occurs in Lock free structures due to the ABA problem when we are using memory from the heap. Assume reference X is pointing to object A. A thread T1 reads the value of the reference X and makes a local copy of an updated version. Before it can install the new version using compareAndSet, another thread T2 updates X to B. Furthermore, it deallocates the object A. It again acts on object A and updates X to A. Now if T1 executes compareAndSet,it succeeds because X is still pointing to A. This operation ideally should have failed since the object itself is completely dierent.

2. Helping
Sometimes it is required to perform multiple actions on a lock free implementation. If a thread finds that the remaining actions are idempotent then instead of waiting for the earlier thread to nish its operation, the current thread can simply perform the action before continuing to do its own operation.

13.1.4 Linked List

Linked List has 3major operations $add(value), remove(value)$ and $contains(value)$.

1. Fine Grained Linked List
Fine grained locking uses the principle of implementing multiple locks. In this process a lock is implemented for each node. There is the idea of *LazyDeletion* that can be implemented for hand-over-hand locking in Fine Grained list. Every node has a *isDeleted* bit associated with it. We separate the deletion in 2 parts, the logical deletion and the physical deletion. The logical deletion is followed by the actual physical deletion. When a remove operation is called the isDeleted bit is set to *true*.

For insertion operation, the previous and successive node should be locked. Then we check that both the nodes have not been deleted by checking the *isDeleted* bit. Another thing we have to check is that previous should point to successive node. If there conditions are satisfied the insertion operation occurs otherwise the thread retries.

2. Lock Free Linked List
This can be implemented using principles similar to Free Grained Locking.

References

[1] V. K. Garg, *Introduction to Multicore Computing*

[2] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/tree/master/chapter6-concurrent-data-structures

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 13: October 11

Lecturer: Vijay Garg *Scribe: Harsh Yadav*

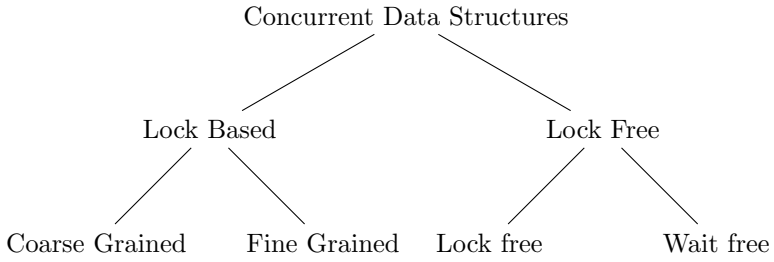
Agenda

The lecture focused on Concurrent Data Structures. Chapter-5 (Wait Free Synchronization) will be continued on next class meeting. Major concepts discussed -:

- Classifications of Concurrent Data Structures
- Concurrent Stacks
- Concurrent Queues
- Concurrent Linked Lists

13.1 Introduction

A **Concurrent Data Structure** is a particular way of storing and organizing data for access by multiple computing threads (or processes) on a computer. Concurrent Data Structures are classified as following -:



The main difference between lock based and lock free implementation lies in the technique to enable mutual exclusion. Lock based primarily works on the concept of *locking, mutually exclusive operation and then unlocking*. However, In lock-free, there is no explicit locking. The mutually exclusive operations happen using *CAS - Compare and Set operation*. The process keeps on trying in a while loop until *CAS* succeeds. *CAS* acts as the linearization point in lock-free scheme. (*Refer to AtomicInteger.Java for an example of lock-free using CAS on course github page*).

Coarse Grained - It refers to locking at entry and unlocking at exit while using a data-structure.It is less efficient in terms of exploiting concurrency. It is relatively easier to implement and should be used when Data-structure is not in the critical path (Data Structure's performance is not of much significance).

Fine Grained - It refers to using multiple locks and implement a more finer locking scheme into various operations instead of explicitly locking at entry and unlocking at exit. (Difficult to implement)

Code 1: Lock Free Implementation

```
int regularLockFreeOperation(){
    ...
    ...
    while(true){
        if(CAS(....)){
            return;
        }else{
            Thread.yield();
        }
    }
}
```

13.2 Concurrent Stacks

The main operations associated with stacks are -:

- **Push**
- **Pop**

The concurrent stack implementation is pretty straightforward.

Lock Based - Lock based stack can be implemented by simple usage of monitors (*Synchronized and Reentrant locks*) for *push* and *pop* operations.

Lock free - Lock free stack implementation can be achieved by standard *CAS* operation in a while loop as discussed in the previous section. (*Refer to LockFreeStack.java on course github page*)

NOTE - One thing to note while doing lock-free implementations using above discussed technique is to make while() loop as tight as possible. (ex - Refrain from allocating memory inside the while() loop)

Speedup trick for Lock-free Stack (Cancellation of Operations) - If there are concurrent push and pop operations happening on stack as shown in Fig 13.1, Stack can be made faster. As shown in the figure, only one of the *push* operation will succeed (either *push(70) or push(30)*).Let's say push(70) succeeded. As, there is a concurrent pop() operation happening, *push(30)* can be cancelled by *pop()* operation. In implementation, an Exchanger has to be implemented to keep record of all the operations and cancel them if a scenario like this exists.

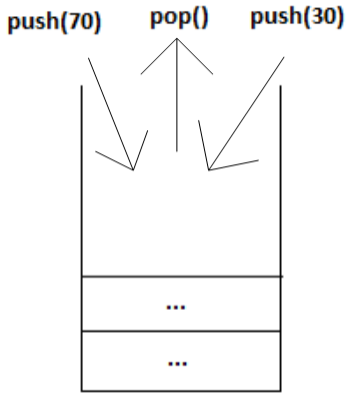
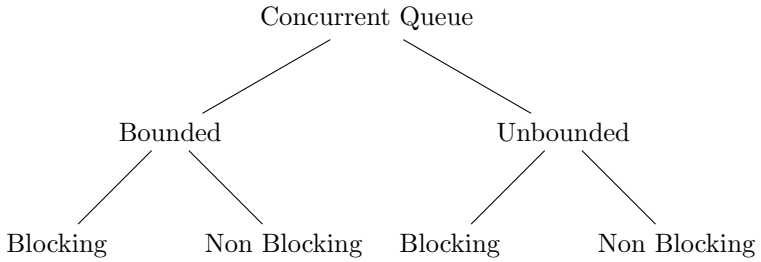


Figure 13.1: Concurrent Stack

13.3 Concurrent Queue

The main operations associated with stacks are -:

- **Enqueue**
- **Dequeue**



Each of the leaf nodes can be further classified into lock based and lock free.

13.3.1 Lock Based

Lock based implementation is usually achieved by using standard monitors. One tweak to fasten lock based implementations is to use multiple locks instead of using single lock. The reason why multiple locks are utilized here and not for Stacks is that operations are happening in queue at both ends *(Enqueue - Tail, Dequeue - Head)* as opposed to Stack where both *(Push and Pop)* operations were happening at the top of the stack. (*Refer to UnboundedQueue.Java and unbounded-total-lock-based-queue.txt on course github page*)

NOTE - Queue implementation can be made more faster by using the concept of *Sentinel* node. As we know, In both *Dequeue & Enqueue* operations, we have to check first that the head is not null. The *if*

statement gets executed in multiple instructions at processor level. These checkings can be avoided if Queue is designed to have at-least one node, even when it is empty (*Sentinel node*). One more thing that can be kept in mind for *Dequeue* operation is to shift the sentinel node(head) to next node instead of pointing the old head to the next. The concep of *Sentinel* node is more beneficial for Lock free implementation.

13.3.2 No lock No CAS queue

This is a special version of queue (bounded) which can be implemented without any Locks as well as without any CAS operations. The important thing to note here is this type of implementation can be done only for *Single Consumer Single Producer* scenario (*Refer to SingleQueue.Java on course github page*). Here *put* is reading the value of *head* and *get* is reading the value of *tail*. The consistency conditions on reading *head & tail* will give either the current value or the previous value depending on whether any concurrent operation is happening to update these registers or not. In both the cases, the legality of both *put() & get()* operations is maintained. It is impossible to make a Lock free and CAS free queue for multiple consumers or multiple producers scenario.

Code 2: No Lock No CAS Queue

```
public void put(Object x) {
  while (tail - head == items.length) {}; //busywait
  items[tail % items.length] = x;
  tail++;
}
public Object get() {
  while (tail - head == 0) {}; // busywait
  Object x = items[head % items.length];
  head++;
  return x;
}
```

13.3.3 Lock Free (Michael and Scott's Algorithm)

The usual Lock free implementation uses CAS operation. One problem that may arise in lock-free queue using CAS operations is **ABA problem**.

13.3.3.1 ABA problem

ABA problem occurs during synchronization, when a location is read twice, has the same value for both reads, and "value is the same" is used to indicate "nothing has changed". However, another thread can execute between the two reads and change the value, do other work, then change the value back, thus fooling the first thread into thinking "nothing has changed" even though the second thread did work that violates that assumption. Suppose a thread is in the process of swinging the *tail* pointer from *A* to *P* (while enquing P) using a CAS operation. Before the CAS happens, another thread T1 comes and swings the pointer to B (B enqueued). Now, CAS fails in compare operation. Now memory location A is free. Suppose another thread comes and swings the Tail pointer to A (in a new enqueue operation). Now the previous CAS operation which failed in compare will succeed and behave as if nothing has changed which is not the case as there is a separate enqueue(B) which happened. Note that this condition is very rare as it is very less probability for a new enqueue operation to exactly use the same memory location (A) just after it got freed.

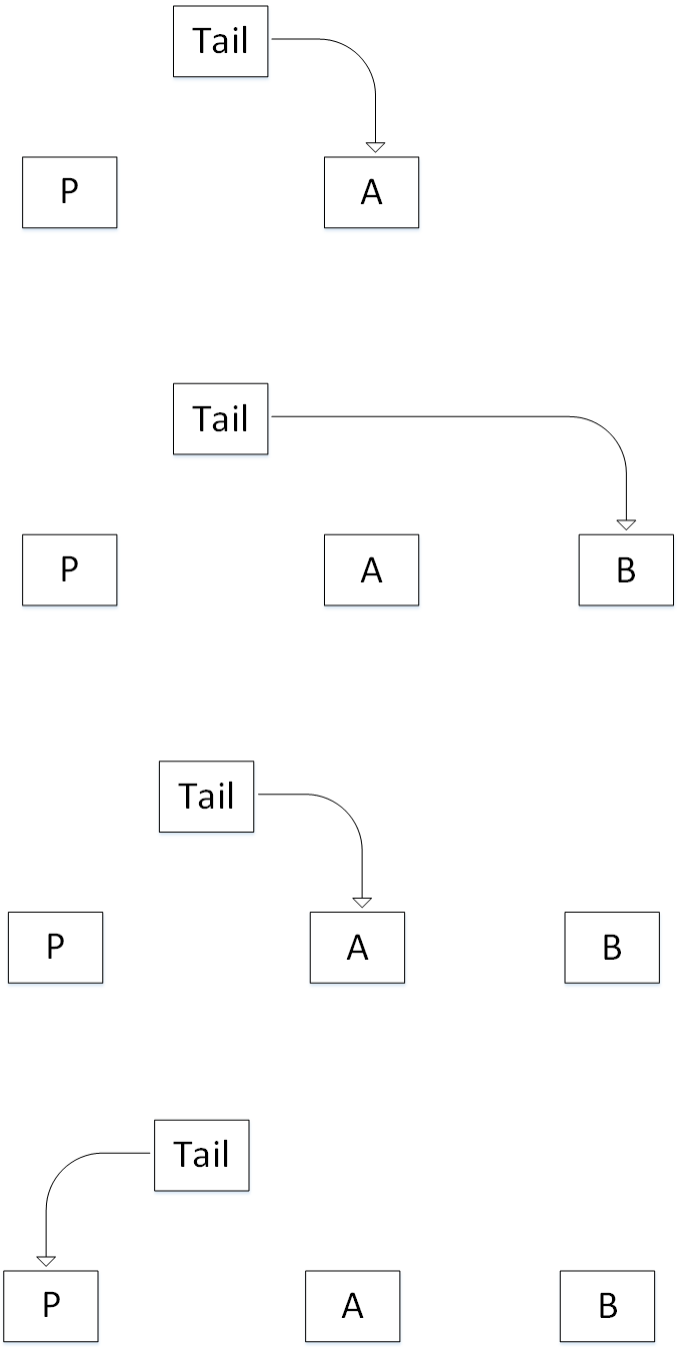


Figure 13.2: ABA problem

Solution - The solution to this problem was given by *Michael and Scott.*The solution is based on attaching a number to the pointer and increment it every time the Tail is assigned to the pointer. In this way, CAS (which reads a complete word) will be able to identify that A is a different version while doing the CAS operation second time in the above discussed scenario.

13.3.3.2 Helping

- Another feature given by *Michael and Scott's* algorithm is *helping*. As, In the lock-free scenario, the state of a process is visible, another process can take on the work of a process (who haven't finished yet) and finish it to speedup the execution. In the M&S imlementation, there are two CAS operations in the main loop *(in both enqueue and dequeue)*. Lets take an instance of enqueue operation, the first CAS is trying to add a new node at the end of the linked list (Linked List implementation of queue). If first CAS is successful, enqueue is completed and process exits from the loop. If CAS is not successful, then the process tries to help some other process (2nd CAS operation in loop) which is in the middle of enqueue to swing tail to the next node (2nd CAS operation - idempotent). After that, the process starts again for its own enqueue.

13.4 Concurrent Linked List

The main operations associated with stacks are -:

- **Add**
- **Remove**
- **Contains** - read only operation (should occur faster, avoid using locks in it)

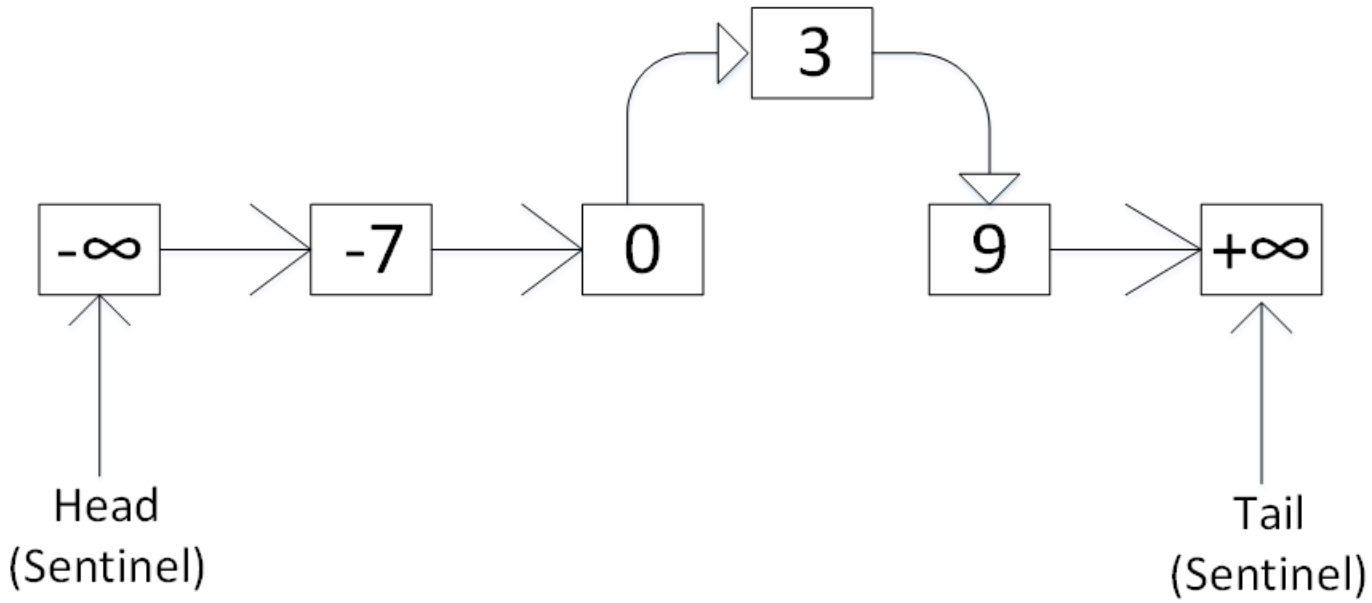


Figure 13.3: Linked List

We need locks for implementing linked lists because any operation (Add and Remove) involves two nodes. If we are doing lock-free implementation, there may be instances when there is an Add operation going on and someone removes previous node. For example, there is an add operation going on for *Node 3* and someone removes *Node 0*, at this point *Node 3* is lost. To prevent this kind of scenario, we need locks. One more

thing about Locked implementation is, it's not sufficient to just lock one element due to possibility of above mentioned scenario. So, we need locks on two elements.

Coarse grained linked list is implemented using usual monitor locks.

Fine grained - As discussed above, fine grained locking requires atleast two locks on successive elements for consistent operations.

Following techniques are used for efficient implementation of Linked Lists -:

- ***Hand Over Hand Locking*** - The idea is pretty simple. Instead of having a single lock for the entire list, you instead add a lock per node of the list. When traversing the list, the code first grabs the next nodes lock and then releases the current nodes lock (which inspires the name hand-over-hand).
- ***Lazy Deletion*** - A delete bit is placed on each node which is set to 1 before attempting the deletion of node. This enables other operations to know that this node is going to be deleted. The updating of this bit is the linearization point. Logical deletion is happening here before Physical deletion.
- ***Validate*** - Before an operation, the delete bit on nodes of importance for that operation should be checked.

NOTE - An important thing to remember in Linked List insert() operation is to do op(2) before op(1). The reason being, if op(1) happens before op(2), it can lead to a misconception of a broken list to a concurrent contains() operation.

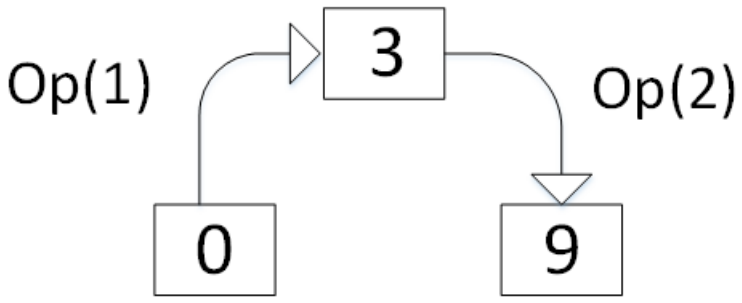


Figure 13.4: Linked List

References

[1] Vijay K Garg, Introduction to Multicore Computing

[2] Mark Moir and Nir Shavit, Concurrent Data Structures

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 14: October 13

Lecturer: Vijay Garg *Scribe: Nishanth Shanmugham*

14.1 Topics

The topics covered in this lecture are:

- GPUs and CUDA Introduction
- CUDA Hello World
- Parallel Computations in CUDA
- Terminology
- CUDA Threads & Identification
- GitHub Links

14.2 GPUs and CUDA Introduction

Heterogeneous computing is programming using both a CPU and a GPU. The following terminology is commonly used to describe the CPU and GPU.

- **Host**: The CPU and its memory (host memory)
- **Device**: The GPU and its memory (device memory)

The general programming process is:

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute,caching data on chip for performance
3. Copy results from GPU memory to CPU memory

The contents of this lecture can be found in `Introduction_to_CUDA_C.pptx` on Canvas [Canvas]. This document aims to capture the essential information from class—not to repeat the contents in the slides.

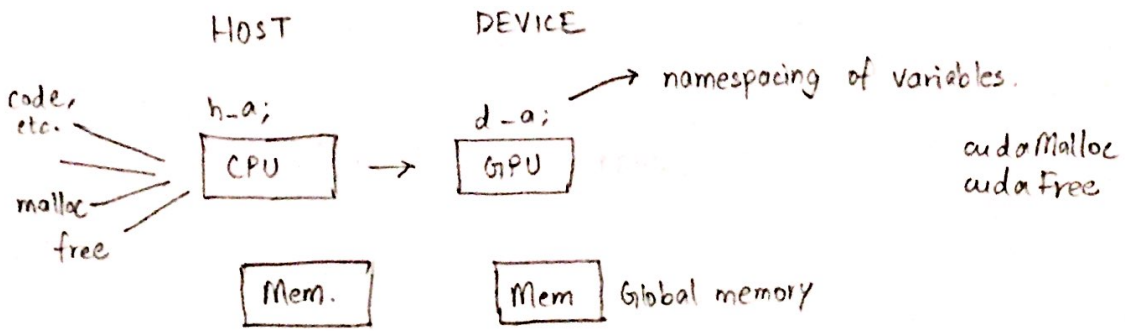


Figure 14.1: Host and Device

14.3 Cuda Hello World

```
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<1,1>>();
    printf(``hello, world\n'');
    return 0;
}
```

The above code is a simple *Hello World* in CUDA in C. First, we declare a function called `mykernel`. The function is then invoked in `main`.

```
mykernel<<N,M>>();
```

The first argument (N) in the angular brackets is the number of *blocks* and the second argument (M) is the number of *threads per block*.

The `__global__` indicates that the function should be run of the device; that is, `mykernel` will be executed on the GPU. In this example, the function does nothing. In future section, we write more useful programs.

14.4 Parallel Computations in CUDA

To run a CUDA function in parallel, we set the argument N (described above) to a value greater than 1. This runs the function in parallel on N blocks on the GPU. Making such a CUDA call can be considered the equivalent of OpenMP's `parallel for loop` construct.

14.4.1 Addition

The following code performs parallel addition on the GPU.

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}


int main(void) {
    int a, b, c;             // host copies of a, b, c
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Setup input values
    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Since the function runs on the device the variables used in the function should reside in device memory. Hence we use the `cudaMalloc` and `cudaFree` functions to allocate and release memory on the device. Moreover, after allocating the memory, the values should be copied from the host to the device using `cudaMemcpy`. Similarly, after the computation is complete and before releasing the device memory, the result must be copied to the host using `cudaMemcpy`. The direction that `cudaMemcpy` should use is specified using the last argument; in this case `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`.

14.5 Terminology

When a GPU is started, a `grid` is started. Figure 13.2 displays the logical construction in a GPU. A grid is made of a set of `blocks`. A block consists of a set of `threads`.

Figure 14.2: Logical construction

14.5.1 Hardware

As show in Figure 13.1, the blocks of a grid are enumerated and distributed to multiprocessors. Shared memory s local to a block. Accessing shared memory is faster than going to global memory. A streaming multiprocessor is composed of multiple streaming processes.

14.6 CUDA Threads & Identification

A block can be split into parallel threads. However, there is a restriction on the maximum number of threads that can run in a single block. If such a situation arises, an option is to use multiple blocks.

Unlike blocks, threads can communicate and synchronize.

In addition, the following tips are useful:

1. If the work is dependent between parallel workers, use the same block. This provides the advantage of shared memory access and communication between threads in the block.
2. If the work is not dependent, then using multiple blocks is faster.

14.6.1 Identifying blocks and threads

The predefined variables `blockIdx`, `blockDim`, and `threadIdx` can be used to designate IDs to threads running in parallel. This allows programmers to assign tasks to each thread based on their ID.

For instance, the index that a particular thread should work on can be computed using:

```
int index = threadIdx.x + (blockIdx.x * blockDim.x);
```

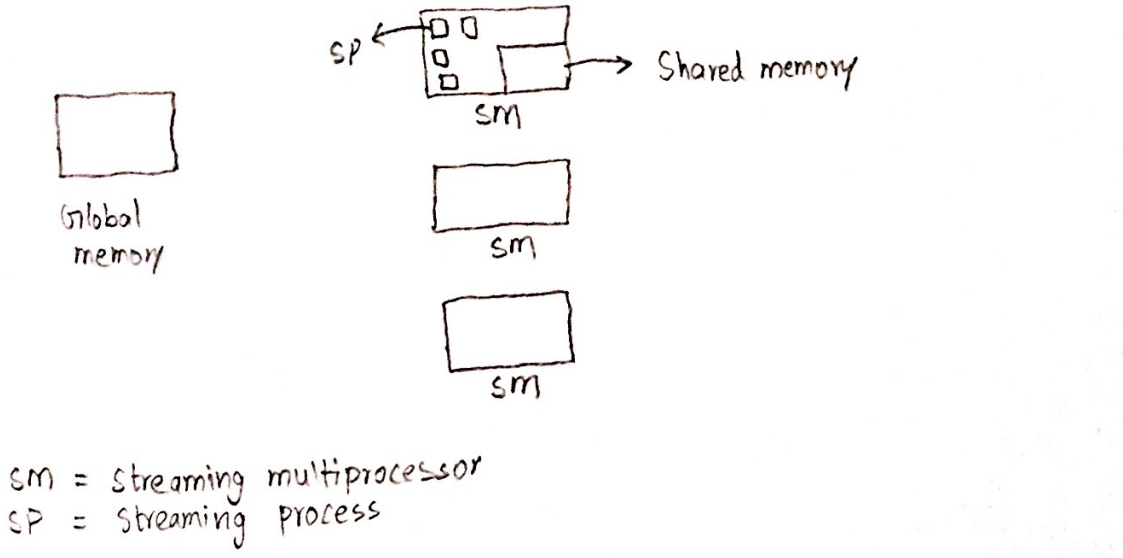


Figure 14.3: Hardware

In addition to the `.x` field, `.y` and `z` fields are also available to facilitate working with multi-dimensional data.

14.6.2 More Terms

- **__shared__**: Declares a variable/array as shared memory..
  - Data is shared between threads in a block.
  - Data is not visible to threads in other blocks.
- **__syncthreads()**: Used as a barrier to wait for threads to prevent data hazards. Similar to the Java construct Threadsjoin.

14.7 GitHub links and Conclusion

The files on GitHub have Cuda examples [GitHub]. See the directory named `cuda`. The code documented with comments. For additional explanation of `reduce.cu`, see the next lecture.

References

[GitHub] Multicore Computing source code, *https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore*

[Canvas] Introduction to CUDA, *https://utexas.instructure.com/courses/1174768/modules/items/8255450*

[Cuda] Cuda Toolkit Documentation*http://docs.nvidia.com/cuda/index.htmlaxzz4NfgY7FeW*

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 15: October 18

Lecturer: Vijay Garg *Scribe: Ben Fu*

15.1 Introduction

The lecture introduces the concept of *scan*, or *prefix-sum*. Scan is first defined and later implemented using three different algorithms.

15.2 Scanning Algorithms

A *scan* algorithm, also known as a *prefix-sum* algorithm, takes a binary associative operator $\oplus$ and an array of n elements:

$$[a_0, a_1, a_2, ..., a_{n-1}]$$

and returns the array with the operator applied cumulatively such as follows:

$$[a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, ..., a_0 \oplus a_1 \oplus a_2 \oplus ... \oplus a_{n-1}]$$

For example, if the operator $\oplus$ is addition and we have an array such as follows:

$$[3, 1, 7, 0, 4, 1, 6, 3]$$

the result of the scan, or prefix-sum, would be the array:

$$[3, 4, 11, 11, 15, 16, 22, 25]$$

For simplicity, we will use addition as the operator $\oplus$ when discussing the following algorithms.

15.2.1 Sequential Scan

Naturally, the sequential algorithm is to iterate through the array, applying the operator to each new element. The pseudocode for the sequential algorithm is as follows [1]:

```
out[0] = in[0]
for all i from 1 to n
    out[i] = out[i-1] + in[i]
```

Code 1: Sequential Scan

The time and work complexity for this algorithm are both O(n).

15.2.2 Hillis-Steele Scan

The second algorithm was introduced by Hillis and Steele, which is a naive parallel implementation of the sequential scan algorithm. The naive parallel algorithm is as follows[1]:

```
out[0] = in[0]
for all i from 1 to log(n)
    for all k in parallel
        if k ≥ 2^d then x[k − 2^(d−1)] + x[k]
```

Code 2: Hillis-Steele Scan

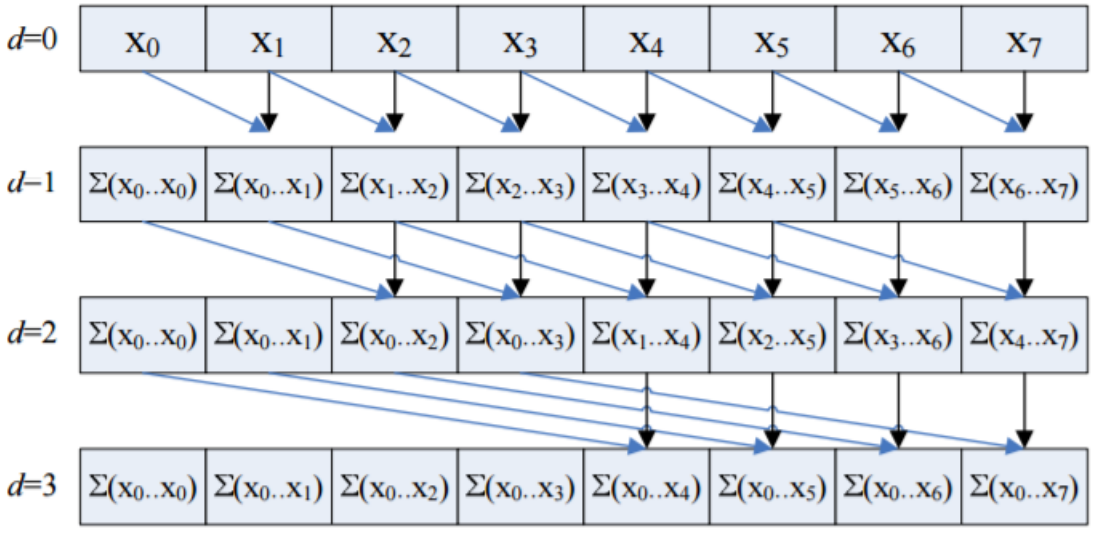


Figure 15.1: Hillis-Steele naive parallel scan

The algorithm is correct but not work optimal. The total number of operations the algorithm performs is $O(nlogn)$, which is sub-optimal.

15.2.3 Blelloch Scan

To build a work-optimal solution to scan, we use a balanced binary tree. Although an actual data structure is not allocated, the solution builds partial sums that can be represented by a balanced binary tree. In this solution, which was formed by Blelloch [2], there are two phases: an *upsweep* and a *downsweep* phase.

In the upsweep phase, each thread computes the partial sums of the two lower-level elements. This can be visualized by the following figure:

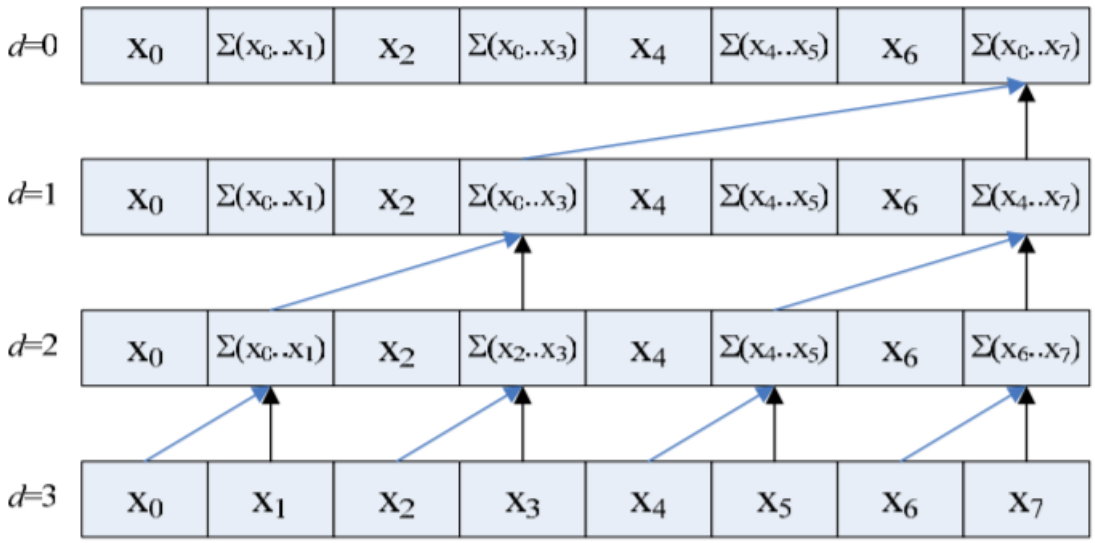


Figure 15.2: Upsweep phase in Blelloch Scan

The algorithm is as follows [1]:

```
for all d from 0 to log(n−1)
    for all k from 0 to n−1 by 2^{d+1} in parallel
        x[k + 2^{d+1} − 1] = x[k + 2^d − 1] + x[k + 2^{d+1} − 1]
```

Code 3: Upsweep Phase

In the downsweep phase, we build the scan using the partial sums computed during the upsweep phase, replacing the elements with the scan output in place. Note that the example provided is an exclusive scan; that is, a zero is inserted into the first step of the downsweep and the final array does not include the total sum.

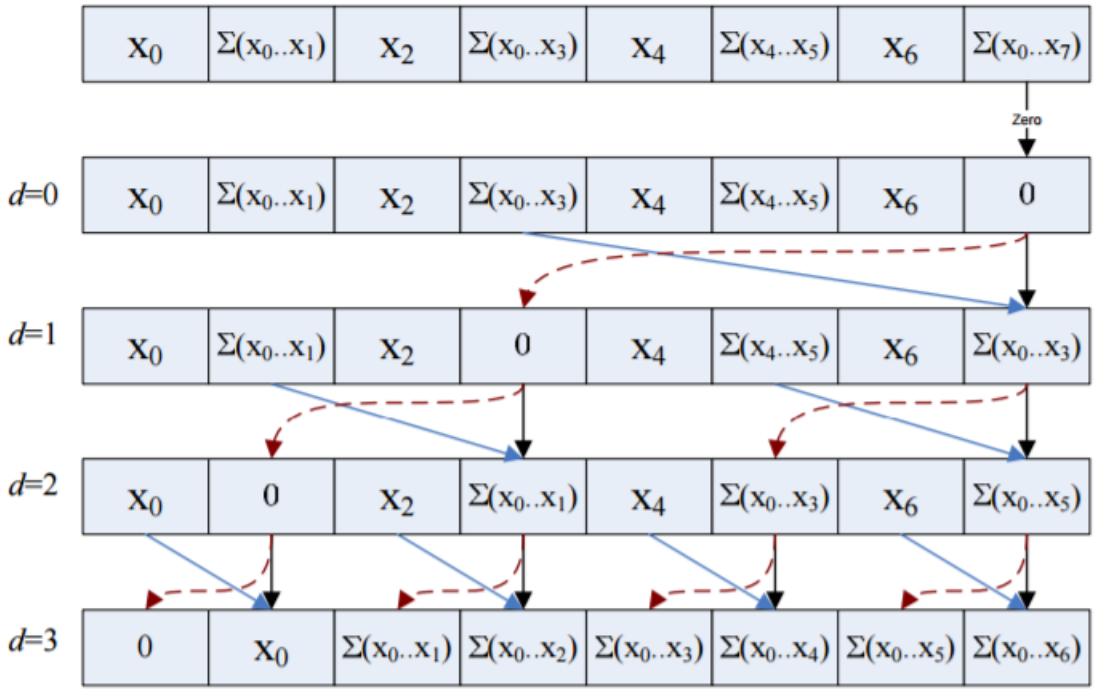


Figure 15.3: Downsweep phase in Blelloch Scan

The algorithm is as follows:

```
x[n - 1] = 0
for all d from log(n) down to 0
    for all k from 0 to n-1 by 2^(d+1) in parallel
        temp = x[k + 2^d - 1]
        x[k + 2^d - 1] = x[k + 2^(d+1) - 1]
        x[k + 2^(d+1) - 1] = temp + x[k + 2^(d+1) - 1]
```

Code 4: Downsweep Phase

A sample implementation of the Blelloch algorithm in CUDA C code is shown on the next page.

```
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
    extern  __shared__  float temp[];// allocated on invocation

    int thid = threadIdx.x;
    int offset = 1;

A   temp[2*thid]   = g_idata[2*thid]; // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];

    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        __syncthreads();

        if (thid < d)
        {
B           int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            temp[bi] += temp[ai];
        }
        offset *= 2;
    }

C   if (thid == 0) { temp[n - 1] = 0; } // clear the last element

    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
        offset >>= 1;
        __syncthreads();

        if (thid < d)
        {
D           int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            float t    = temp[ai];
            temp[ai]   = temp[bi];
            temp[bi] += t;
        }
    }

    __syncthreads();

E   g_odata[2*thid]   = temp[2*thid]; // write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
```

Figure 15.4: CUDA implementation for Blelloch Scan

15.2.4 Complexity Table

The complexity table for the time and work complexities of the previous three algorithms are shown in the following table:

Scan Algorithm	Time Complexity	Work Complexity
Sequential	$O(n)$	$O(n)$
Hillis-Steele	$O(log(n))$	$O(nlog(n))$
Blelloch (Work Optimal)	$O(log(n))$	$O(n)$

Table 15.1: Analysis of Scan Algorithms

References

[1] M. Harris, Parallel Prefix Sum (Scan) with CUDA, *Nvidia Corporation* (2007), pp. 3–10.

[2] G. E. Blelloch, Prefix Sums and Their Applications, *Synthesis of Parallel Algorithms* (1990).

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 15: October 18

Lecturer: Vijay Garg *Scribe: Avin Tung*

15.1 Introduction

This lecture will complete the Cuda examples and begin covering algorithms for parallel programming using GPU:

1. reduce.cu
2. Parallelism Vocabulary
3. Blelloch Scan
4. Brick Sort

15.2 reduce.cu

Check out the Cuda code for reduce.cu [1].

It uses multiple blocks, each with multiple threads, of the GPU to compute the sum of the array by reduction. It splits the array into multiple section for each block to sum using its multiple threads. When each block is complete with its portion, a single block will sum the final sums and return.

Each block runs the following code. The code splits the array in half, then uses n/2 threads to sum entries n+i and n+i+(n/2) for i= 0 to n/2 into a new array of size n/2. Before the next iteration, *syncthreads* function is called to make sure all adds are done before moving on. This is repeated until there is only one entry left which is the sum of the total array. The new value is then written back to global with a single thread.

The key things to note is the program runs a global memory kernel in data and a shared memory kernel in sdata. This example is to show the difference in time between threads running using global memory and threads running on their own shared memory.

15.3 Parallelism Vocabulary

There are specific terms that are often used when talking about writing programs for GPUs.

15.3.1 Map

Map maps every entry to another entry, one to one, and returns it. So for every i, we return it's corresponding f(i). In terms of parallelism, it can be done in O(1) time depending on the complexity of the mapping function.

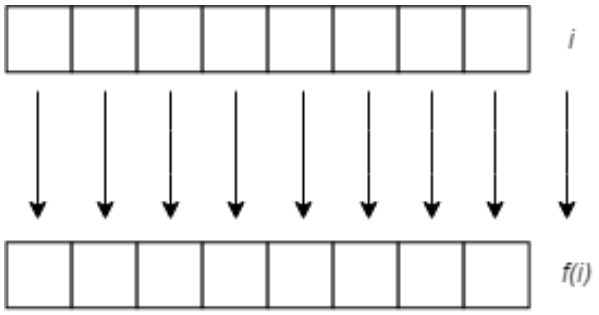


Figure 15.1: Mapping

15.3.2 Reduce

Reduce is taking an associative operator applied to multiple entries and reducing it to one- many to one. The important thing to note is to understand the identity of the operation. For example, the values for sum should be initialized to 0, the values for max should be initialized to -, the values for min should be initialized to , and so on.

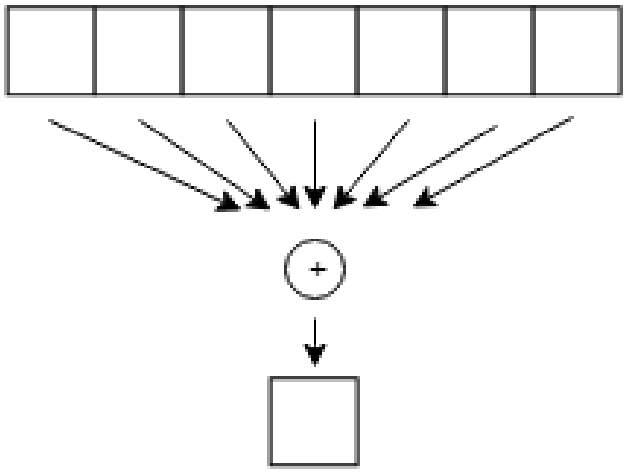


Figure 15.2: Reduce with Add Operation

15.3.3 Scan

Scan takes an array and creates a new array such that the new array's element j is the sum of all the elements before j and including/excluding j. If we include j, it is classified as an inclusive scan, and if j is excluded, then it is an exclusive scan. This is also known as the prefix sum.

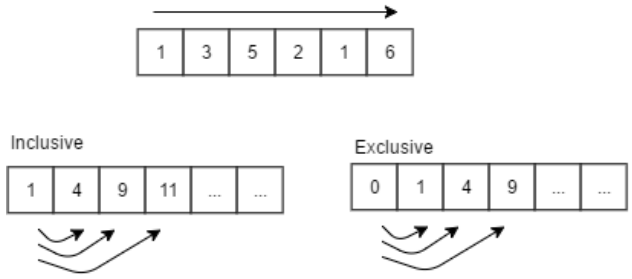


Figure 15.3: Scan

15.3.4 Scatter

Scatter takes an array scatters the entries into a new array. An example of scatter can be merging two sorted arrays into one sorted array using binary search to calculate the new address into the new array.

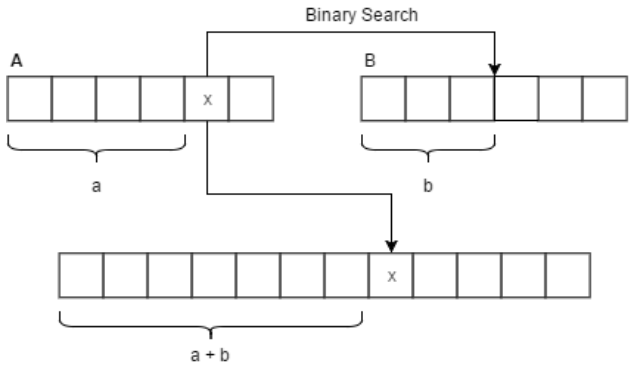


Figure 15.4: Scatter Example: Merging Two Sorted Arrays

15.3.5 Gather

Gather takes multiple entries and does a sequence of operations to them to reduce it to one entry- many to one. The key thing about gather is that some synchronization and atomicity may be used depending on the sequence of operations. For example, we can take 3 entries, and average them into one entry in a new array. This requires an atomic add for each of the 3 entries as well as a barrier before dividing it by 3.

15.3.6 Transpose

Transpose is moving memory from one piece position in memory to another, or liken to transposing a matrix. This is useful in parallel programming since it moving entries into memory that is faster to access and operate on, such as into local cache, can significantly improve the performance time.

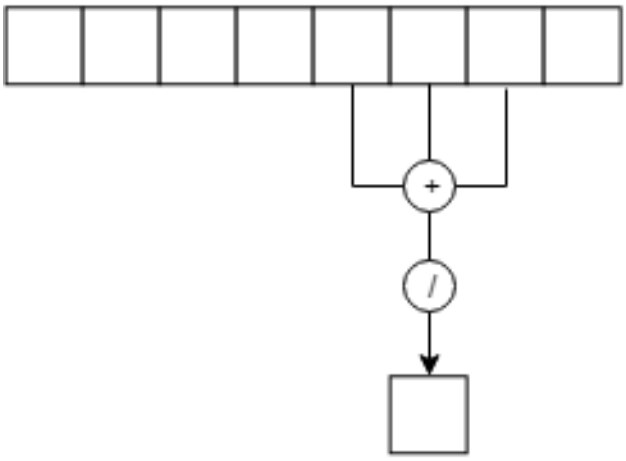


Figure 15.5: Gather Example: Average Example

15.3.7 Stencil

Stencil is like the 3D version of gather. If on a matrix of entries, stencil can take multiple entries surrounding the desired entry, operate on them, and store the value into a new grid of entries.

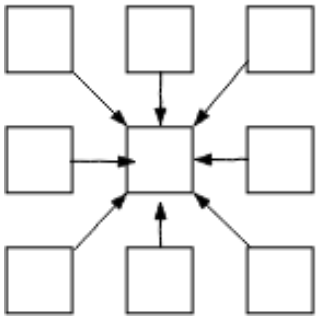


Figure 15.6: Stencil

15.3.8 Compact

Compact is looking at an array, operate on it, and return a compact version of the original array. For example, an array can have multiple threads reading each entry and test whether it is prime, and return only the prime values into a new, smaller array.

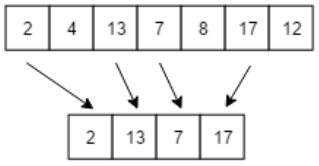


Figure 15.7: Compact Example: Prime

15.4 Blelloch Scan

Blelloc Scan is used to find the total sum of all entries while being both work optimal as well as time efficient. The speed and work is summarized and compared to other total scan sum algorithms we have discussed previously.

Scan Algorithm Complexities		
Scan Algorithm	Time Complexity	Work Complexity
Sequential Scan	O(n)	O(n)
Parallel Prefix Scan (Hillis-Steele)	O(logn)	O(nlogn)
Blelloch Scan	O(logn)	O(n)

For Blelloch Scan, there are two phases to compute the exclusive scan- an upsweep and a downsweep. In the upsweep phase, we do a sum operation on each neighboring pair, keeping track of each of the summed intermediate entries for reference for the downsweep phase. Once the upsweep phase is complete, we set the last node we touch to 0. We then follow it with the downsweep phase, such that at each level we take the previous left entry and add it to the current entry and set that to its new right entry, and carry the current entry to the new left entry.

Upsweep can be computed as:

$$sum[v] = sum[L[v]] + sum[R[v]] \tag{15.1}$$

Downsweep can be computed as:

$$\begin{aligned} scan[L[v]] &= scan[v] \\ scan[R[v]] &= sum[L[v]] + scan[v] \end{aligned} \tag{15.2}$$

Where *sum[L[v]]* is the most recent, nearest, unused summed left value during the upsweep phase.

The parallelism exist since each pair addition is done by an independent thread.

See bottom of document for Blelloch Scan Example

15.5 Brick Sort (Odd-Even Sort)

Brick sort sorts an array of entries like bubble sort with parallelism. It breaks the array into pairs, then compares (and swaps) all pairs into order. Then, on the new array, the entries pair off with the neighbor it did not previously pair with, and repeat. Each pair comparison is handled by an independent thread. This process will take n pairings to complete, therefore O(n).

Brick sort is easy to implement, but is an oblivious sorting algorithm. This means it does not take the values into account and will always take the same amount of time based on the number of entries in the array. Furthermore, brick sort can be viewed as a sorting network, where the entries are the input to the network, and the output will be the sorted entries. In the following figure, the vertical lines are comparators and the horizontal lines are the wire the entries can travel across. The a's are the initial entries and the b's are the sorted entries.

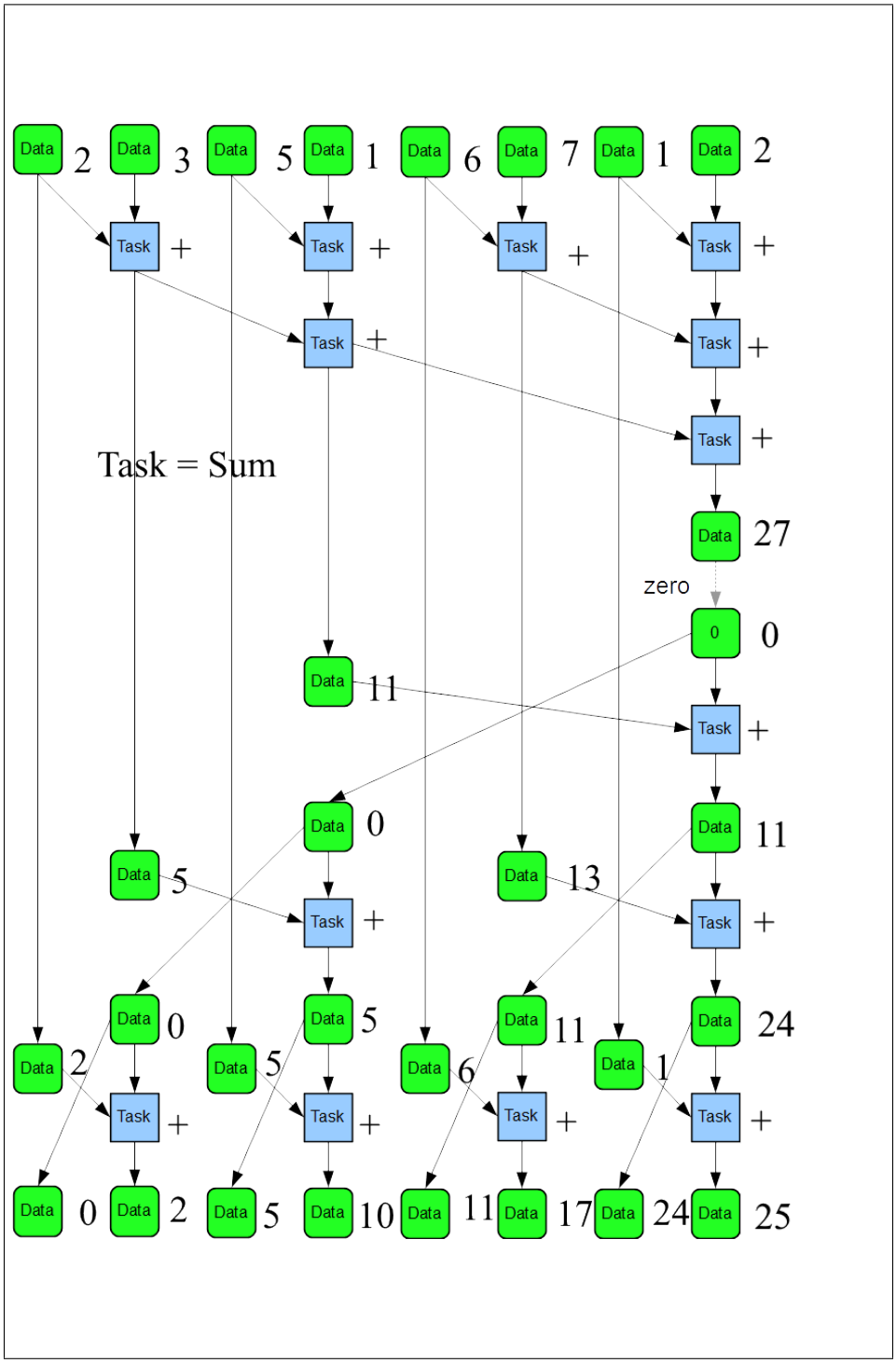


Figure 15.8: Blelloch Scan [4]

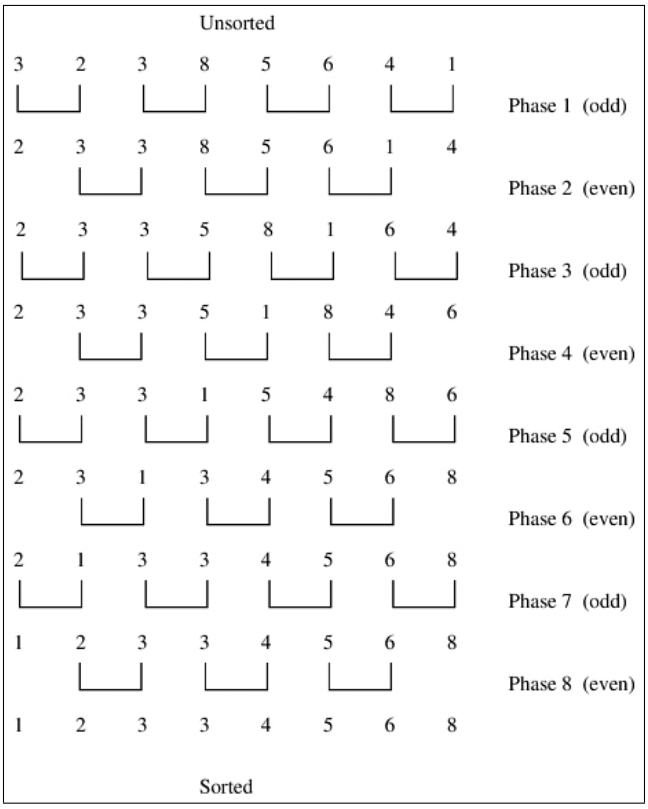


Figure 15.9: Brick Sort [2]

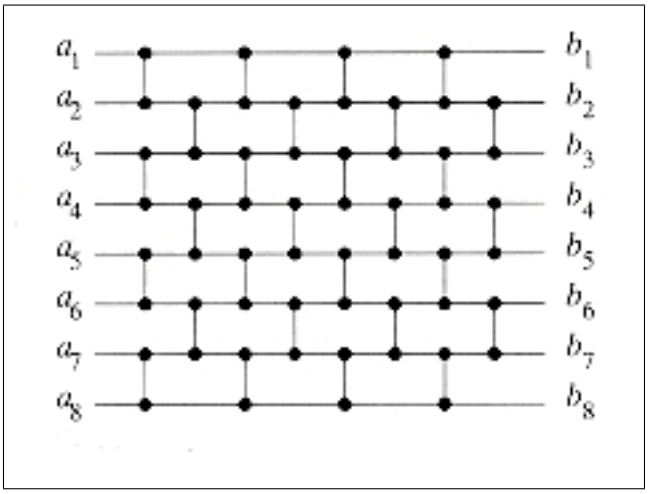


Figure 15.10: Brick Sort Network View [3]

References

[1] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/blob/master/chapter3-synchronization_primitives/BoundedBufferMonitor.java

[2] http://d1gjlxt8vb0knt.cloudfront.net//wp-content/uploads/Even-Odd-Sort.gif

[3] http://staff.ustc.edu.cn/ csli/graduate/algorithms/book6/651_a.gif

[4] https://scs.senecac.on.ca/ gpu621/pages/images/prefix-scan-balanced.png

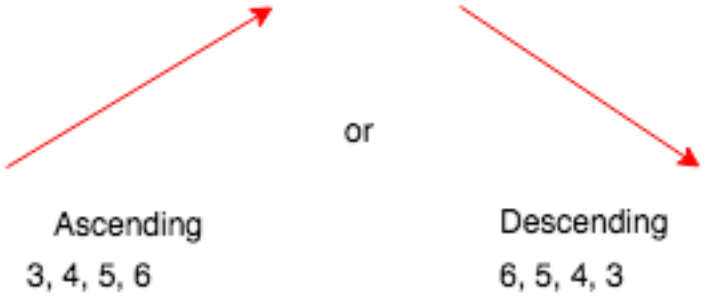
EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 16: October 20

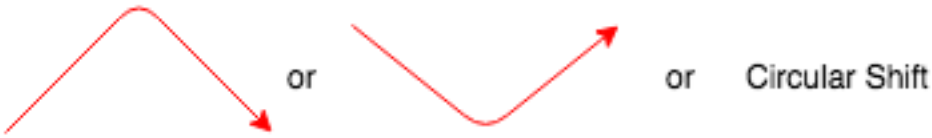
Lecturer: Vijay Garg *Scribe: Yanfeng Zhao*

16.1 Bitonic Sort

16.1.1 Monotonic Sequence



16.1.2 Bitonic Sequence



Def: $X_1 \dots X_n$ is a **bitonic sequence** if there exist k s.t. $X_1 \leq X_2 \leq \dots \leq X_k$ and $X_k \geq X_{k+1} \geq X_{k+2} \geq \dots \geq X_n$ or any circular rotation.

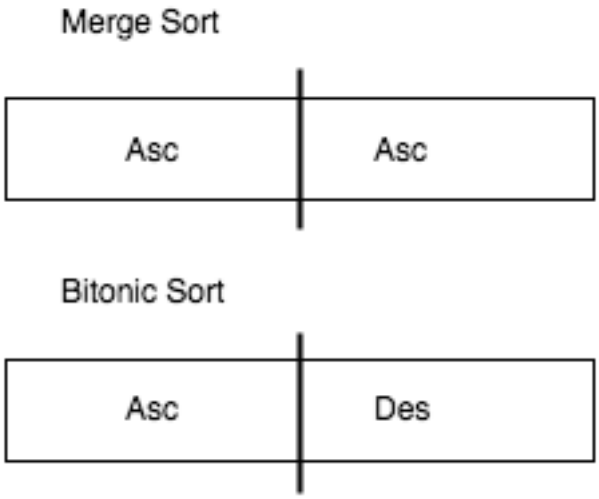
For example:

Bitonic sequence (k = 4): 1, 3, 5, 11, 9, 4, 2

Circular Shift Bitonic Sequence: 2, 1, 3, 5, 11, 9, 4. Notice that 2 was shifted to the front.

Any monotonic sequence is also bitonic sequence.

16.1.3 Why is this useful?



Bitonic Merge

$X_1 \dots X_{n/2} -> $ asc

$X_{n/2+1} \dots X_n$ -> des

Max

- $Y_1 = \max(X_1, X_{n/2+1})$
- $Y_2 = \max(X_2, X_{n/2+2})$
- ...

Min

- $Z_1 = \max(X_1, X_{n/2+1})$
- $Z_2 = \max(X_3, X_{n/2+3})$
- ...

Bitonic Sort Example:

2 7 11 13 9 5 3 1

2 5 3 1 < 9 7 11 15

2 1 < 3 5 < 9 7 < 11 13

1 < 2 < 3 < 5 < 7 < 9 < 11 < 13

16.2 Bitonic Sort Algorithm

BitonicSort (lo, hi) // assume size = 2^k for some k

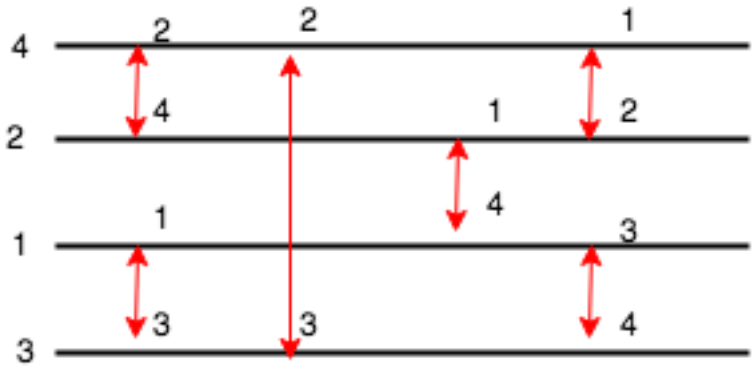
Base Case: If array Size == 1, then return

Recursion:

1. Bitonic Sort left side of the array (left half)
2. Bitonic Sort right side of the array (right half)
3. Compare and exchange if needed pairwise
4. Bitonic Merge (L)
5. Bitonic Merge (R)

Steps 3-5 are part of the bitonic merge.

Example:



16.3 Bitonic Sort Time Complexity Analysis

The algorithm in short:

- BitonicSort(L)
- BitonicSort(R)
- BitonicMerge(L, R)

Notice BitonicSort(L) and BitonicSort(R) can be done in parallel.

$T(n) = T(n/2) + log(n)$

$T(1) = O(1)$

$T(n) = T(n/2) + log(n)$

$= T(n/4) + log(n-1) + log(n)$ ... $= 1 + 2 + 3 + ... + log(n)$

$= O(log^2 n)$

References

[1] V. K. Garg, Introduction to Multicore Computing

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 17: October 25

Lecturer: Vijay Garg *Scribe: Cassidy Burden*

17.1 Sorting

Previously we have talked about three different sorting methods:

1. Brick Sort
2. Bitonic Sort
3. Merge Sort

We will finish parallel sorting by talking about the parallel merge sort algorithm.

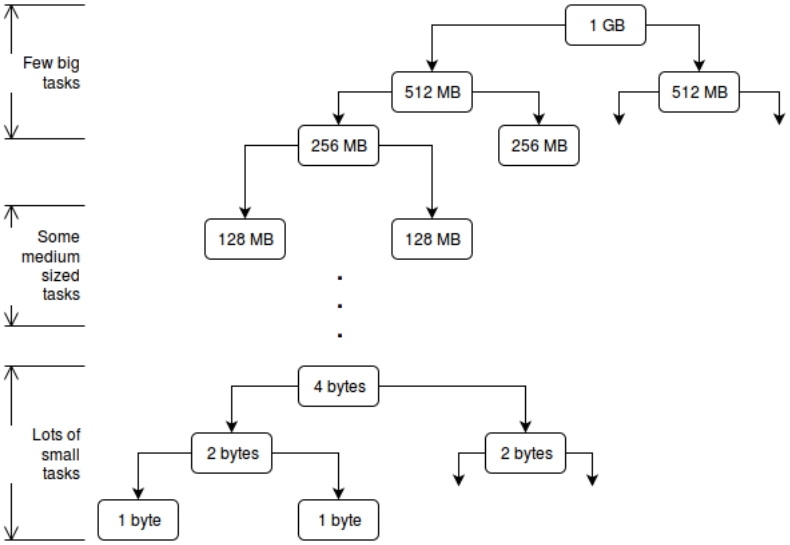
17.1.1 Parallel Merge Sort

Merge sort works by splitting the array into 2 halves, sorting both halves recursively, and then merging the two halves together.

The majority of the work done will be in the merge step. We have previously talked about a work optimal method of merging arrays in parallel by splitting the arrays into chunks and calculating the rank of each element.

We will break the problem into three chunks to allow us to most efficiently distribute our blocks and threads:

1. Large merges: do parallel scatter merge, assign one block per chunk of scatter
2. Medium sized merges: one merge per block, do parallel scatter merge
3. Small merges: one merge per thread, done sequentially



We eventually end up with a time complexity of $O(\log n)$ and work complexity of $O(n \log n)$.

17.2 Wait-free Registers

We now will continue our dicussion on how to build differing levels atomic registers. We will present the following hierarchy in which every register can be built with an unbounded amount of the preceding register.

To recap there are 3 different types of registers that act differently during a concurrent access:

- Safe - Returns any possible value
- Regular - Returns value currently being written or previously written value
- Atomic - Returns values such that history is linearizable

SRSW Safe Bit
(store previous value)
⇓
SRSW Regular Bit
(unary construction)
⇓
SRSW Regular Multivalue
(timestamp, keep seq number)
⇓
SRSW Atomic Multivalue
(replication, timestamp, communication matrix)
⇓
MRSW Atomic Multivalue
(bakery-style choosing of timestamp)
⇓
MRMW Atomic Multivalue

17.2.1 SRSW Regular Bit

Bits can only have the four histories when looking at the two most recent writes:
$(0,0)$ $(0,1)$ $(1,0)$ and $(1,1)$

If we ensure that the only histories are $(0,1)$ and $(1,0)$ then our histories will contain all possible values for a bit. Thus when a concurrent access on the safe register occurs, we will either randomly return the value currently being written or the previous value, satisfying the regular register property. To ensure only those two histories occur, we will ignore all duplicate writes to the register.

17.2.2 SRSW Regular Multivalue

To build a regular register than can hold more than just two values, we will have a max size for our register X. To represent X values we will need X of our SRSW regular bit registers. We will keep these bits in an array and say the first non-zero bit in the array represents the value of our multivalue register.

```
SRSW_Reg_Bit A[X]

read():
  for i = [0 -> X]:
    if A[i] == 1: return i

set(x):
  A[x] = 1
  A[x-1 -> 0] = 0
```

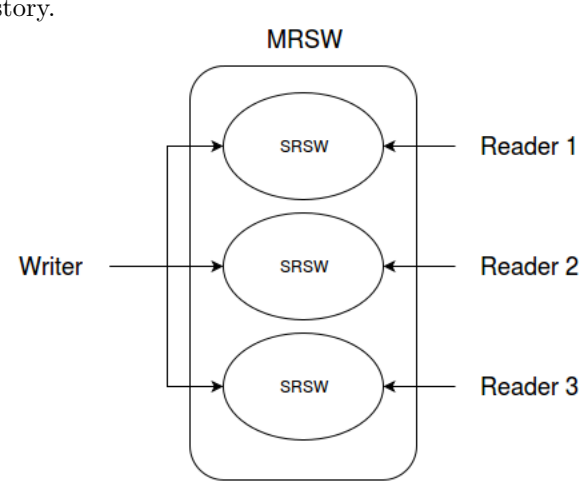
17.2.3 SRSW Atomic Multivalue

Next we will build an atomic register by attaching a sequence number or timestamp to our value. This is with the assumption that we cannot overflow our sequence number.

17.2.4 MRSW Atomic Multivalue

This will be the first register to account for multiple readers. Because we want this register to be atomic we want to make sure that each reader gets the same value once a writer commits a new value.

Because we only have SRSW registers, we will need a register for every reader. The single writer will write to every reader's register, and each reader will read from their own register. However, this is flawed, and can lead to a non-atomic history.

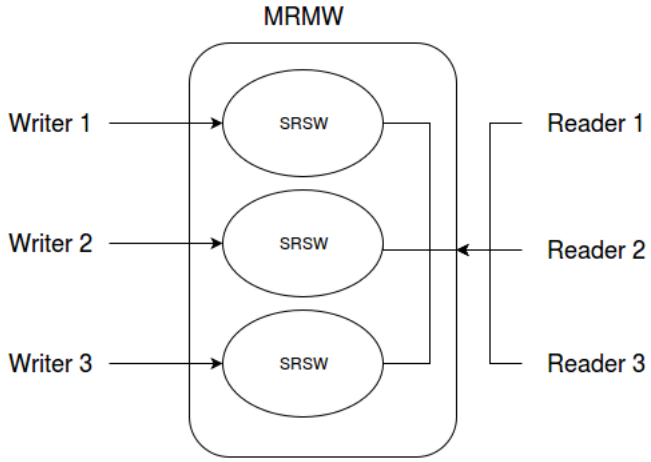


The final solution is to have n^2 SRSW registers, constructing a matrix. We will call this a communication matrix, and it essentially allows each register to share information. When a reader reads from the MRSW register it will want to alert other readers what it just read in order to satisfy atomicity. For example, writing 5 to $comm[i, j]$ is Reader i telling Reader j it just read value 5. Given n readers our MRSW register will work as such:

```
SRSW_Atomic_Multi V[n]
SRSW_Atomic_Multi comm[n, n]

set(x):
  V[0 -> n] = x

read(pid):
  val = max_TS(V[pid], comm[0, pid], comm[1, pid], ..., comm[n-1, pid])
  comm[pid, 0 -> pid] = val
  return val
```

17.2.5 MRMW Atomic Multivalue

For n writers we will need n MRSW registers. Because MRSW only allows one writer, each writer will have to write to their own register. Readers will read from whichever register has the highest timestamp.



However, because we can, in the worst case, have n writers deciding on a new TS at once, we need an algorithm to atomically decide on a timestamp. Lamport's Bakery has already solved this problem, and we will choose our writer's timestamp in the same way processes choose their tickets in Lamport's Bakery:

```
MRSW_Atomic_Multi Regs[n]

set(pid, x):
  ts = max_TS(Regs) + 1
  Regs[pid].set(x, ts, pid) // ts and pid together allow for total ordering

read():
  min_reg = Regs[0]
  for all r in Regs:
    if (min_reg.ts < r.ts) || (min_reg.ts == r.ts && min_reg.pid < r.pid):
      min_reg = r
  return min_reg.read()
```

References

[1] V.K. Garg, Introduction to Multicore Computing

[2] V.K. Garg, https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/tree/master/chapter5-wait-free

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 17: October 25

Lecturer: Vijay Garg *Scribe: Xin Xu*

17.1 Introduction

The topics covered in this lecture are:

1. Merge Sort
2. Wait-Free Synchronization

17.2 Merge Sort

Decompose one task into multiple tasks:

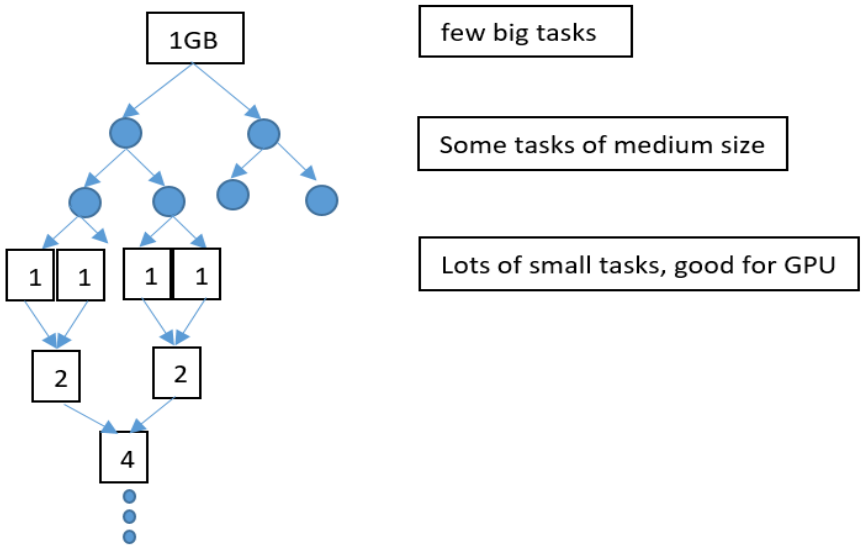


Figure 17.1: Merge Sort

17.2.1 Algorithm for Multiple Merging tasks

1. Determine rank of each element(mentioned in Lecture9). Rank(x) is equal to the sum of number of elements in B less than x and the number of elements in C less than x. Then merge array B and array C to get array D.
2. Cascaded Algorithm(mentioned in Lecture10). Divide n array into n/logn groups. Fill in only splitters.

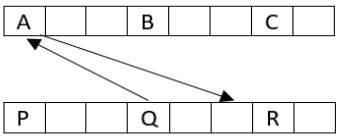


Figure 17.2: Merge Sort

17.3 Wait-Free Synchronization

It is possible to build a multiple reader multiple writer(MRMW) atomic multivalued register from single-reader single-writer(SRSW) safe Boolean registers. This can be achieved by the following chain of constructions:

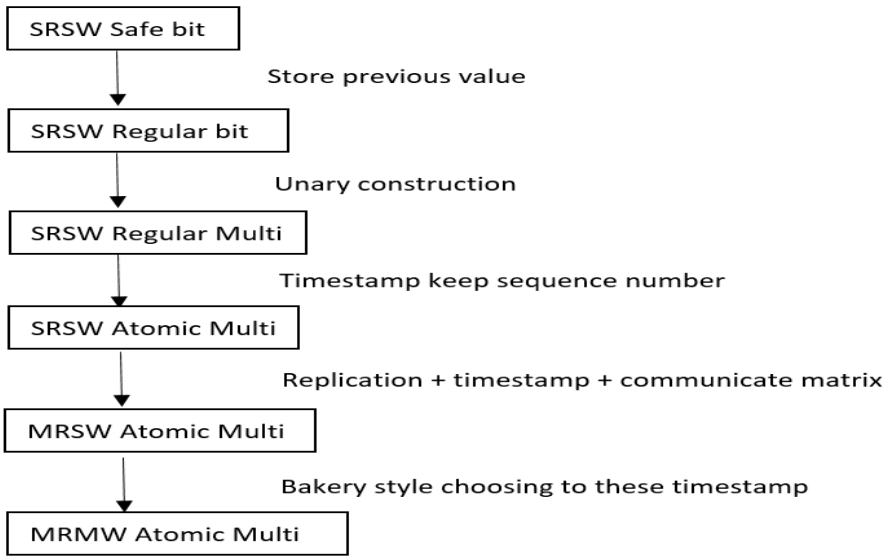


Figure 17.3: construction

17.3.1 Regular SRSW Register

We construct a regular SRSW register from a safe SRSW register using smarter writer which can remember last value.

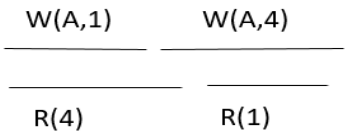
17.3.2 SRSW Multivalued Register

For read: scan the array, looking for the first non-zero bit. The idea is that the reader should return the index of the first true bit. The straightforward solution for writer: Updating the array in the forward direction until it reaches the required index. But it does not work.
Reason: Suppose firstly we set A[3] to 1, and now we need to write A[5] to 1. After the writer executes, the array becomes 0 0 0 0. Before writer set A[5] to 1, the reader comes in, it will find all zeros.
For write, the right operation:

Algorithm 1 RegularBoolean

```
1:  class RegularBoolean {
2:      boolean prev; // not shared
3:      SafeBoolean value;
4:      public boolean getValue() {
5:          return value.getValue();
6:      }
7:      public void setValue(boolean b) {
8:          if (prev != b) {
9:              value.setValue(b);
10:             prev = b;
11:         }
12:     }
13: }
```

•Set A[x] = 1
•Traverse backward resetting bit to zeros
But it is not atomic. This situation can happen:



Code in the handout: the reader first does a forward scan and then does a backward scan at line 14 to find the first bit that is true. Two scans are sufficient to guarantee linearizability.

17.3.3 MRSW Register

We now build a MRSW register from SRSW registers. The straightforward solution is to have an array of n SRSW registers and one writer write to all n registers. This is not linearizable. Instead, we use a sequence number associated with each value. The writer maintains the sequence number and writes this sequence number with any value that it writes. Thus we view our SRSW register as consisting of two fields: the value and ts (for timestamp). In order to build MRSW Register, we use communication matrix. Comm[i][j] is used by the reader i to inform the value it read to the reader j. Reader communicate the value it read to all other readers by writing in the corresponding row.

17.3.4 MRMW Register

The construction of an MRMW register from MRSW registers is to use n MRSW registers for n writers. We use the approach of the Bakery algorithm to assign unique sequence number to each writer.

17.3.5 Atomic Snapshots

Lock-free constructions can be turned into wait-free constructions by using the notion of helping moves. The main idea is that a thread tries to help pending operations. For example, the thread wanting to perform an update operation helps another concurrent thread.

References

[1] V.K. GARG Introduction to Multicore Computing

[2] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 18: November 27

Lecturer: Vijay Garg *Scribe: Huy Doan*

18.1 Atomic Scan

Scan → atomic read of multiple locations Update → atomic write of a single location

Scenario: Assume SWSR and the memory as follow

M0	0
M1	0
M2	0

Figure 18.1: Initial Memory

We are doing two operations W(M0,1), W(M1,1) sequentially. The memory states M0M1M2 we expect to observe are 100, 110. However, a scan may result in the state 010. This result means that the scan operation is not linearizable.

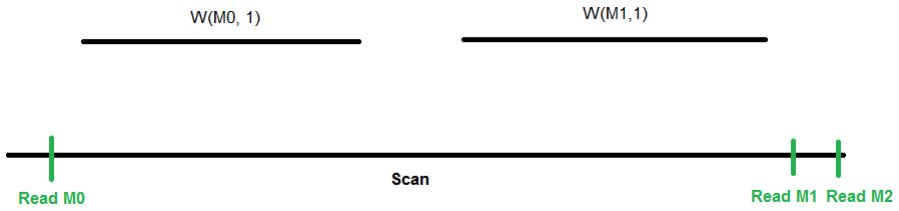


Figure 18.2: Bad Scan

Solution:
Update
Write the new value with the updated timestamp
Scan
Collect an entire array in W
Loop:
Read the entire array again to make sure that there is no change to the timestamp (2nd collect)
If there is some change, go back to loop

→ The algorithm is lock-free but not wait-free because if writes keep coming, scan has to loop and wait

Memory	Value	Timestamp
M0	1	1
M1	1	2
M2	0	0

Figure 18.3: Memory with timestamp

NOTE: It is impossible to solve the dual problem which is write to multiple locations and read a single location in a wait-free manner.

18.2 Consensus

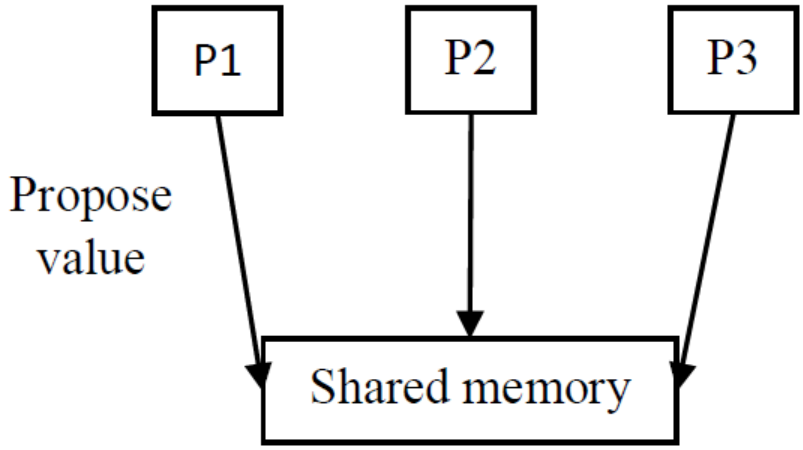


Figure 18.4: Consensus Problem

Consensus is a problem that requires a given set of processes to agree on an input value. As depicted in Figure 18.4, each process proposes its own value and the system has to decide which value among those that all processes will agree on.

The **requirements** on any object implementing consensus are as follows:

- Agreement: No two correct processes decide on different values.
- Validity: The value decided must be proposed by some process.
- Wait-freedom: Decides in a finite number of steps.

18.2.1 Bully Algorithm

Assume that we have two process P0 and P1 and no process fails. The propose values of P0 and P1 are stored in the array prop[2]. The bully algorithm solves the consensus problem by just simply picking prop[0] at all time.

Consensus 1: Bully Algorithm

```
Pi
    Write your proposal to the prop array
    Every process waits until prop[0] becomes non-null
    Decide on prop[0]
```

Obviously, this algorithm is not fair and not fault-tolerant in case that P0 fails.

18.2.2 Global State Overview

Global State: the state of the shared memory

Formally, let $G.V$ is the set of decision values reachable from global state G. We say G is **bivalent** if $|G.V| = 2$ and **univalent** if $|G.V| = 1$. In the latter case, we call G $0-valent$ if $G.V = 0$ and G $1-valent$ if $G.V = 1$. The bivalent state captures the notion of indecision.

A bivalent state is **critical** if any action by any process leads to a 0-valent or 1-valent state

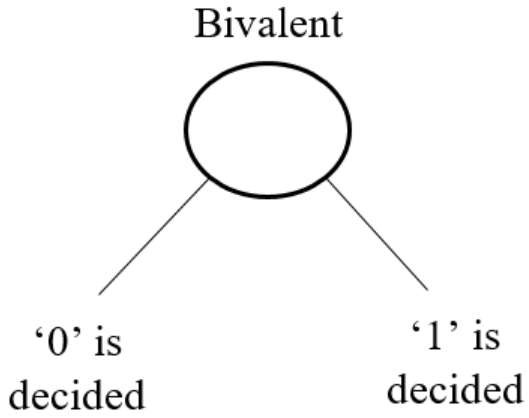


Figure 18.5: Bivalent State

Claim 1: There exists an initial bivalent global state for any consensus protocol.

Proof: Consider the most simple case where there are only two processes. If P1 is slow and P0 runs solo, P0 will eventually decide on '0'. Similarly P1 can run solo and decide on '1'. Therefore, the initial state G is bivalent.

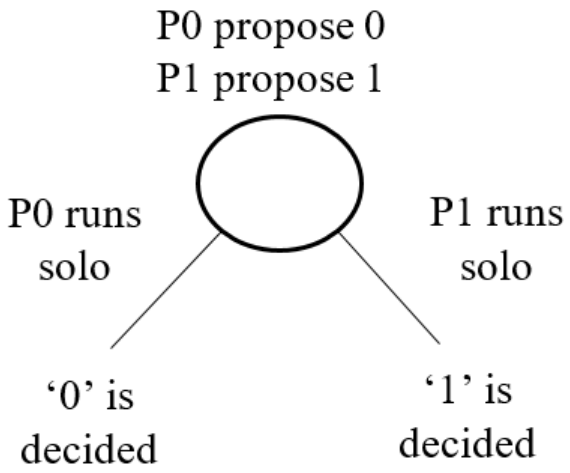


Figure 18.6: Bivalent State of Two Processes

Claim 2: There exists a critical global state for every consensus protocol.

18.2.3 Theorem

It is impossible to solve consensus with just read and write.
Proof: We show that even in a two-process system, atomic registers cannot be used to go to non-bivalent states in a consistent manner. We perform a case analysis of events that can be done by two processes, say, P and Q in a critical state S. Let e be the event at P and event f be at Q be such that e(S) has a decision value dierent from that of f(S). We now do a case analysis:

- Case 1: e and f are operations on different registers. We observe that ef(S) and fe(S) are identical and, therefore, have the same decision value. However, we assume earlier that e(S) and f(S) have different decision values, which implies that e(f(S)) and f(e(S)) have different decision values since the decision values cannot change.

- Case 2: e is read and f is write. When P does e, the state of Q does not change. Therefore, f(S) and f(e(S)) are identical and have the same decision values.

- Case 3: e and f are write on the same registers. Obviously, f(S) and f(e(S)) are identical and have the same decision values.

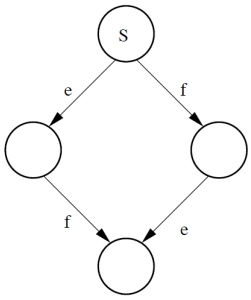


Figure 18.7: Case 1: e and fa are operations on different registers

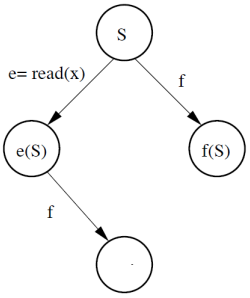


Figure 18.8: Case 2: one read one write

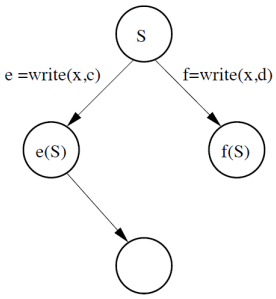


Figure 18.9: Case 3: two writes on the same register

18.2.4 Consensus Number

Consensus number of a shared object class O is the maximum number of processes that use object from class O to solve consensus

Operation	Consensus Number
R/W Register	1
TestAndSet	2
Swap	2
GetAndIncrement	2
CAS	∞

Figure 18.10: Consensus Number of Some Operations

Queue has consensus number 2 Assume a queue has two values *Win* and *Lose*. P0 and P1 can enqueue/dequeue atomically.

Consensus 2: Consensus with Queue

```
Pi
    Write proposal to the prop array
    Dequeue from Q
    If I win, choose my value
    otherwise choose the other's value
```

Theorem: There is no wait-free algorithm to build single producer - multiple consumers Queue using atomic read write registers.

CAS can be used to solve any consensus problem

Consensus 3: Consensus with CAS

```
Register R, initialized to -1
Pi
    Write my proposal to the prop array
    Do R.CAS(-1, mypid)
    if succeed
        decide prop[mypid]
    else
        decide prop[R]
```

References

[1] Vijay K. Garg, Introduction to Multicore Computing

[1] Vijay K. Garg, Concurrent and Distributed Computing in Java (2004), pp. 236

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 18: November 27

Lecturer: Vijay Garg *Scribe: Changyong Hu*

18.1 Atomic Snapshot

Atomic snapshot is a concurrent object that allows us to read multiple memory locations atomically and write one location atomically. A snapshot object is just an array of atomic MRSW registers, one for each thread. It has two methods as below:
Scan : atomic read of multiple locations.
Update : atomic write of a single location.

Scenario: Assume three memory locations and initialized to zero.

M0	0
M1	0
M2	0

Figure 18.1: Initial Memory

Two threads is writing to their own location while one thread is trying to scan. The legal memory state can be 000, 100 and 110. However, scan like below can read memory state as 010, which is inconsistent state.

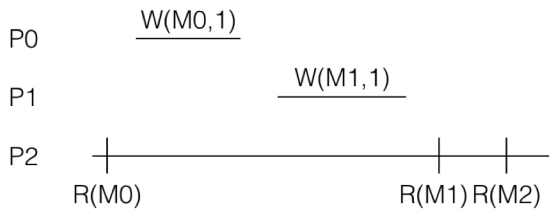


Figure 18.2: Non-atomic scan

Solution:
Update
Write the new value with the updated timestamp

Scan
Collect an entire array
Loop:
Read the entire array again to make sure that there is no change to the timestamp (2nd collect)
If there is some change, go to loop

The algorithm is lock-free but not wait-free because if writes keep coming, scan has to wait forever. But it can be modified to wait-free version by update operation that is helping scan.

It is impossible to solve the dual problem that writes to multiple locations and read a single location atomically in a wait-free manner.

18.2 Consensus

Consensus is a fundamental problem in concurrent system that requires a given set of processors to agree on same value. Each process proposes its own value and eventually decides on some value.

The **requirements** on any object implementing consensus are as follows:

- Agreement: All threads decide the same value.
- Validity: The value decided must be proposed by some processes.
- Wait-free: Every correct process decides in finite number of steps.

18.2.1 State and Valence

Global State: the states of the threads and the shared objects.

Valence: A global state is bivalent if the decision value is not yet fixed: there is some execution starting from that state in which the threads decide 0, and one in which they decide 1. By contract, the global state is univalent if the outcome is fixed: every execution starting from that state decides the same value. A global state is 1-valent if it is univalent and the decision value is 1, and similarly for 0-valent.

Lemma 1: There exists an initial bivalent global state for any consensus protocol.

Proof: Consider the initial state where A has input 0 and B has input 1. If A finishes the protocol before B takes a step, then A must decide on 0, because it must decide some thread's input and 0 is the only input visible to it. A cannot decide on 1 because it's possible that B's input is also 0, that is to say A cannot distinguish the state that B has input 1 to B has input 0 if B didn't take any step. Symmetrically, B must decide on 1. So the initial state is bivalent.

A bivalent state is critical if any threads' moves lead to univalent state.

Lemma 2: Every wait-free consensus protocol has a critical state.

Proof: Suppose not. By Lemma 1, the protocol has an initial bivalent state. Start the protocol from this state. As long as there is some thread that can move without making the protocol state univalent, let that thread move. Because the protocol is wait-free, so eventually the protocol enters a state where no such move is possible, which must be a critical state.

18.2.2 Theorem

It is impossible to solve consensus with only atomic register.

Proof: Suppose there exists a binary consensus protocol for two threads A and B. By Lemma 2, we can run the protocol until it reaches a critical state s. Suppose A's next move leads the protocol to 0-valent state and B's next move leads the protocol to 1-valent state.

First, suppose A reads a register and drive the protocol to 0-valent. B executes one operation and drives the protocol to 1-valent state s'. In the first execution scenario, it executes the same operation for B and drive to s". But s' and s" are distinguishable because read didn't do any change to memory state.

Second, suppose A writes r0 and B write r1, they are commute. Easy to conclude that they cannot reach consensus.

Finally, suppose A and B writes to same memory location r. But write operation can overwrite other's value, so is still distinguishable.

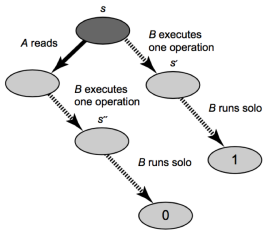


Figure 18.3: Case 1: A reads first

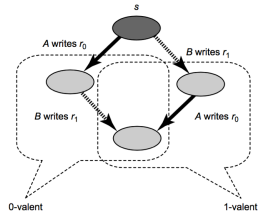


Figure 18.4: Case 2: A and B write to different registers

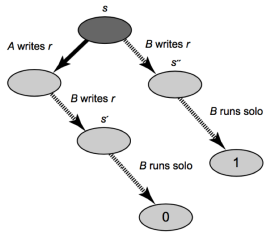


Figure 18.5: Case 3: A and B writes to the same register

18.2.3 Consensus Number

The consensus number of consensus object is the largest n for which that object solves n-thread consensus. If no largest n exists, we say the consensus number of the object is infinite.

The two-dequeuer FIFO queue has consensus number at least 2

Assume a FIFO queue initialized to have two items *Win* and *Lose*. Two threads can dequeue one item atomically.

Consensus 1: Consensus with FIFO Queue

```
Pi
        int i = ThreadID.get();
    propose[i] = value;
    int status = queue.deq();
    if(status = Win)
                return propose[i];
        else
                return propose[1-i];
```

It's trivial that this protocol can solve consensus. If one thread returns it own value, it must have dequeued Win, so the other can only dequeue Lose and decide on other's value.

FIFO queues have exact consensus number 2

Proof: Assume we have a consensus protocol for thread A, B and C. By Lemma 2, the protocol has a critical state s. Without loss of generality, we can assume that A's next move takes the protocol to a 0-valent state and B's next move takes the protocol to a 1-valent state.

First, suppose A and B both call deq(). Let s' be the state if A dequeues and then B dequeues, let s" be the state if the dequeues occur in the opposite order. Then C runs solo on s' and s". But s' and s" are indistinguishable to C, so must decide the same value in both state, while s' is 0-valent and s" is 1-valent. Contradiction.

Second, suppose A calls enq(a) and B calls deq(). If the queue is not empty, then the two methods commute. So C cannot distinguish. If the queue is empty, B dequeues on the empty queue and then A enqueues. C cannot distinguish it with A enqueues alone because it will overwrite B dequeues empty queue.

Finally, suppose A calls enq(a) and B calls enq(b). Consider one situation that A enq(a) and then B enq(b), run A until deq(a) and run B until deq(b), then queue is empty and then run C solo. So C cannot distinguish each other because queue is empty.

Theorem: There is no wait-free algorithm to build single producer multiple consumers Queue using atomic registers.

Proof: Suppose not. Then we can use atomic register to construct single producer multiple consumers queue and use this queue we can solve 2-thread consensus problem, which is contradict with the theorem that using atomic register cannot solve consensus problem. Also we know atomic register has consensus number 1, and FIFO queue has consensus number 2. So object with consensus number 1 cannot be used to build wait-free concurrent object with consensus number 2.

CAS can be used to solve any consensus problem

Consensus 2: Consensus with CAS

```
Register R, initialized to -1
Pi
        int i = ThreadID.get();
        propose[i] = value;
        if(r.compareAndSet(-1, i))
                return propose[i];
        else
                return propose[r.get()];
```

If thread A returns true, then that thread was first in the linearization order, so A decides its own value. Other threads came later will find the current AtomicInteger value, which is threadID of the first thread and decides on the first thread's proposal.

References

[1] Vijay K. Garg, Concurrent and Distributed Computing in Java (2004)

[2] Maurice Herlihy and Nir Shavit, The Art of Multiprocessor Programming (2008)

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 18: October 27

Lecturer: Vijay Garg *Scribe: Yu Sun*

18.1 Atomic Scan

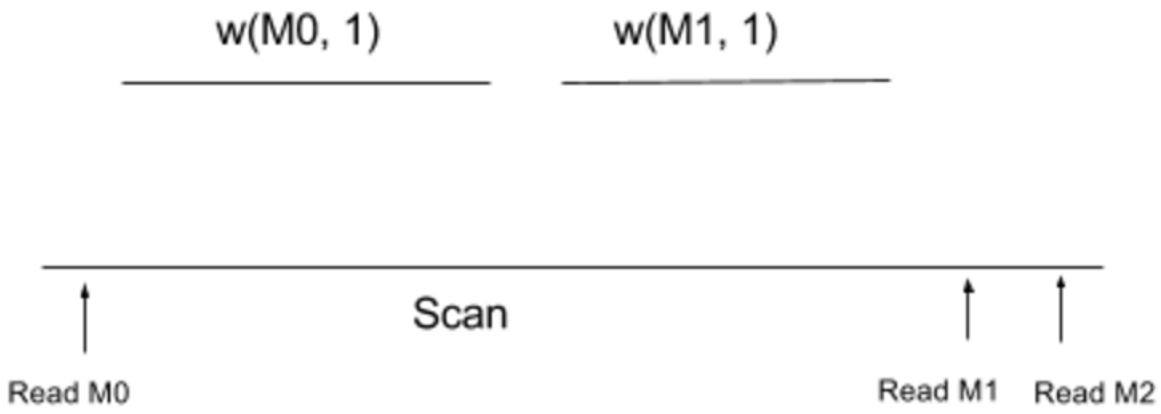
Scan → Atomic read of multiple locations

Update → Atomic write of a single location
Assume the memory state is as following:

M0	0
M1	0
M2	0

We now do two write operations: write (M0, 1) and write (M1, 1)

It is possible that the scan return 010 which is not an atomic execution. Because read(M0) may happen before write(M0, 1) but read(M1) can happen after write(M1, 1).



Non-atomic Scan

Solution

So how to solve this problem? We add a timestamp to each value so that each write operation will update the corresponding timestamp and read operation can guarantee the consistency by compare the timestamps.

Update: Write the new value with updated timestamp.

Scan: Collect the entire array in W.

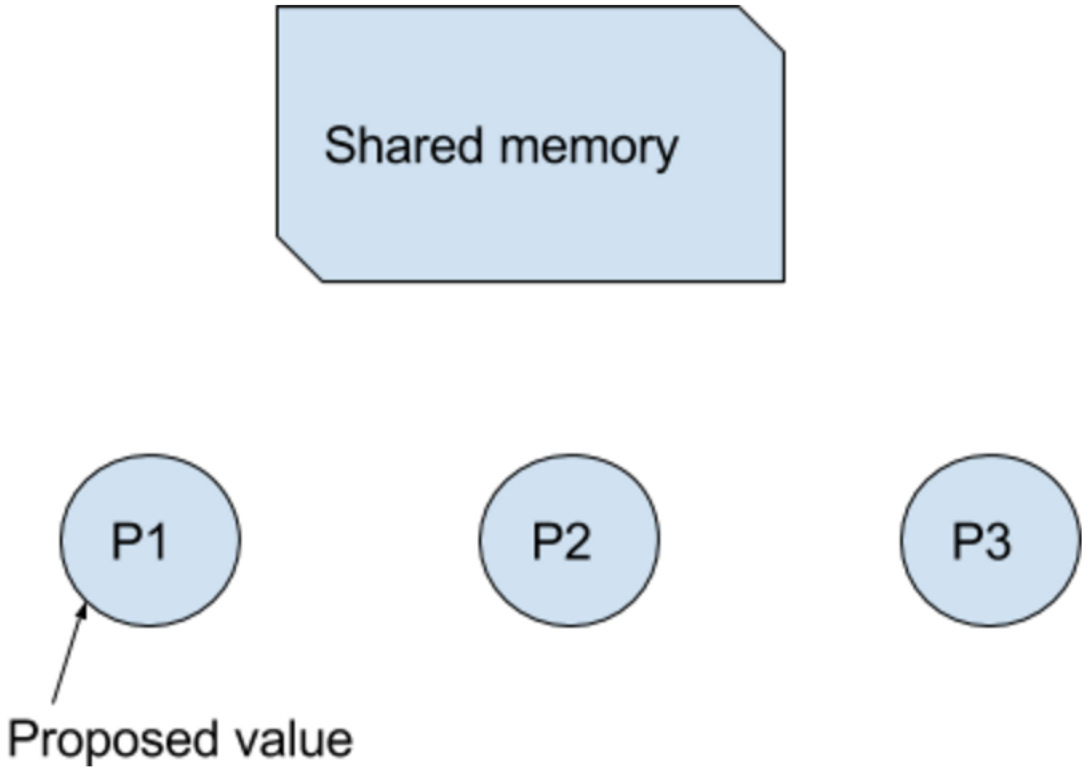
Loop: Read the array again to make sure that there is no change in any timestamp.

If there is some change then go to Loop;

18.2 Consensus

The **consensus** problem requires a given set of processes to agree on an input value.

We abstract the consensus problem as follows: Each process has a value input to it that it can propose. For simplicity, we will restrict the range of input values to a single bit. The processes are required to run a protocol so that they decide on a common value.



The **requirements** on any object implementing consensus are as follows:

- Agreement: No two correct processes decide on different values.
- Validity: The value decided must be proposed by some process.
- Wait-freedom: Decides in a finite number of steps.

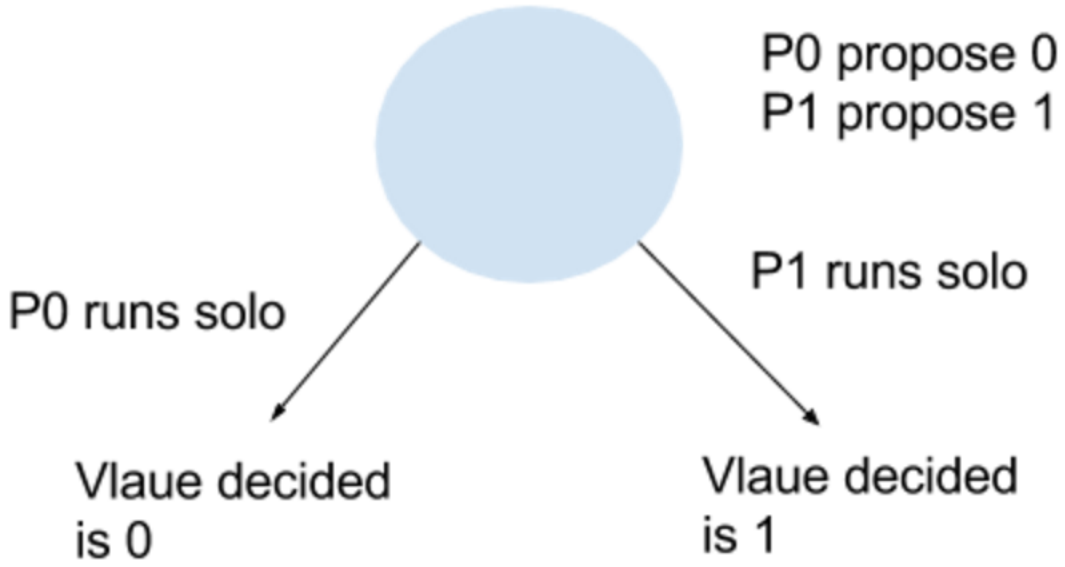
A protocol is in a bivalent state if both the values are possible as decision values starting from that global state.

A bivalent state is a critical state if all possible moves from that state result in nonbivalent states.

18.2.1 Consensus Claims

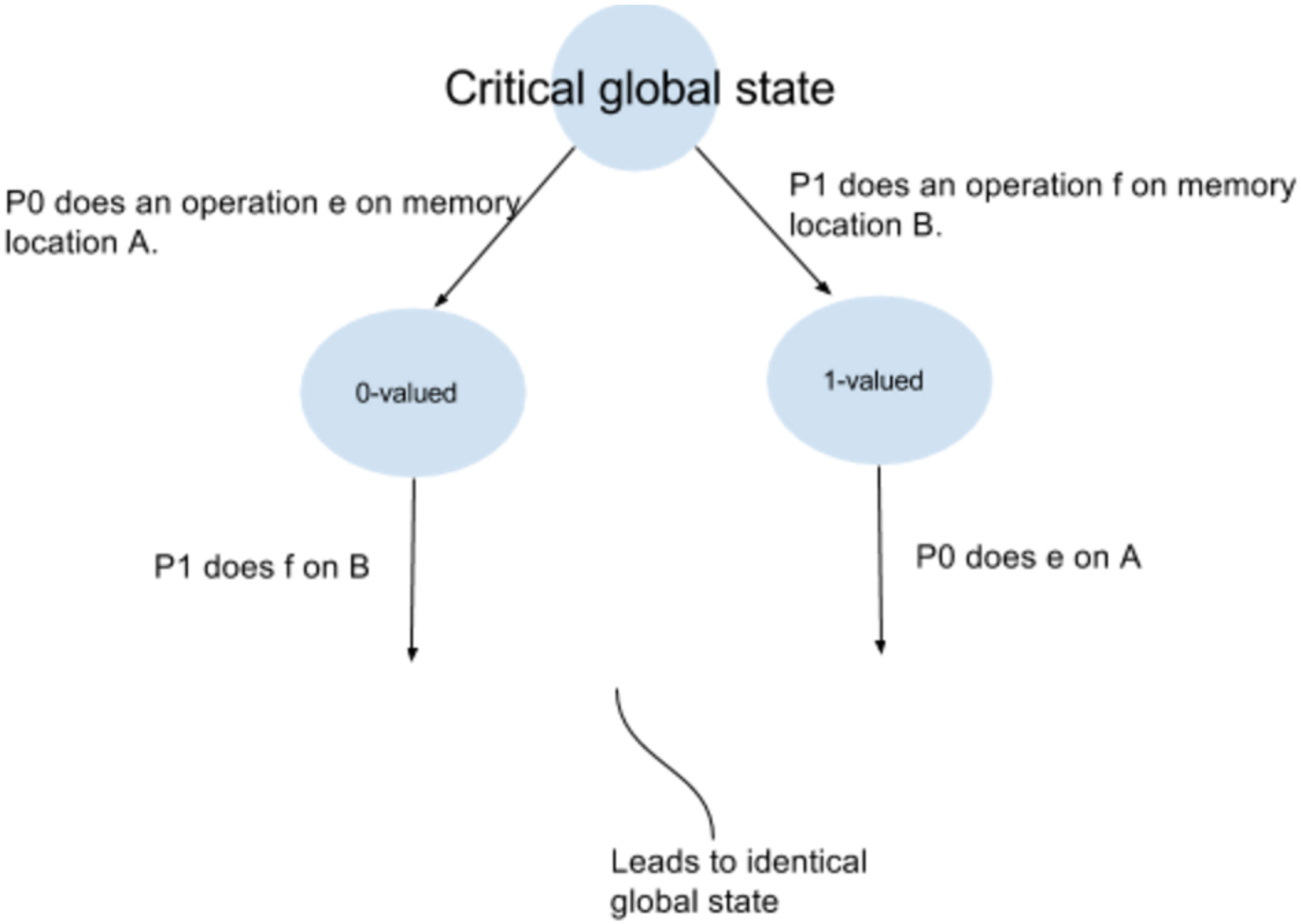
Claim 1: There exists an initial bivalent global state for any consensus protocol.

Proof: Because there exist at least two runs from that state that result in dierent decision values. In the rst run, the process with input 0 gets to execute and all other processes are very slow. Because of wait freedom, this process must decide, and it can decide only on 0 to ensure validity. A similar run exists for a process with its input as 1.

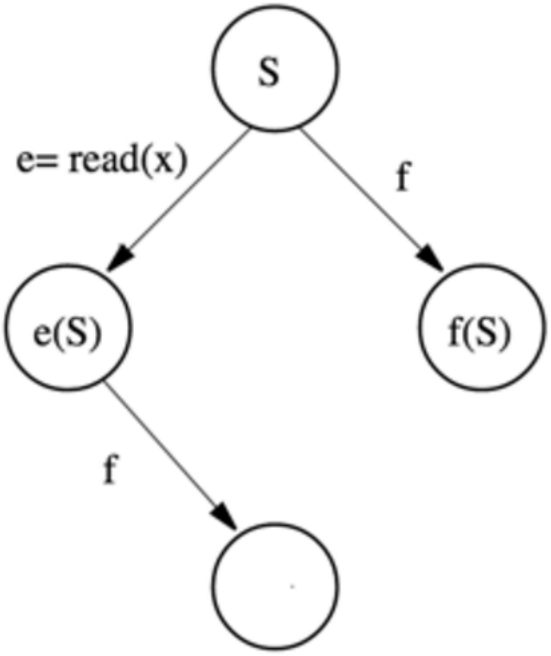


Claim 2: There exists a critical global state for every consensus protocol.

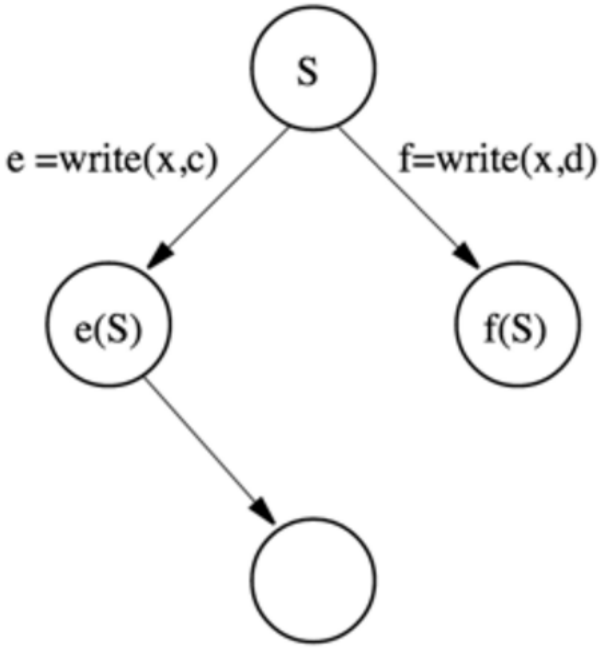
Claim 3: There does not exist any protocol to solve the consensus using atomic registers. **Proof:** We show that even in a two-process system, atomic registers cannot be used to go to non-bivalent states in a consistent manner. We perform a case analysis of events that can be done by two processes, say, P and Q in a critical state S. Let e be the event at P and event f be at Q be such that e(S) has a decision value dierent from that of f(S). We now do a case analysis:



Case 1: e and f are on dierent registers. In this case, both ef and fe are possible in the critical state S. Further, the state ef(S) is identical to fe(S) and therefore cannot have dierent decision values. But we assumed that f(S) and e(S) have dierent decision values, which implies that e(f(S)) and f(e(S)) have dierent decision values because decision values cannot change.



Case 2: Either e or f is a read. Assume that e is a read. Then the state of Q does not change when P does e. Therefore, the decision value for Q from f(S) and e(S), if it ran alone, would be the same; a contradiction.



Case 3: Both e and f are writes on the same register. Again the states f(S) and f(e(S)) are identical for Q and should result in the same decision value.

18.3 SPSC

Assume we have a queue storing values of 'win' and 'lose'.

Pi:

Write my proposal in the prop array.

Deq from Queue

If I win choose my proposal, otherwise choose other guys proposal.

Theorem: There is no wait-free algorithm to build SPMC Queue using atomic read write registers.

Proof: Consensus number of a shared object class O is the maximum number of processes that can use objects from class O to solve consensus.

Consensus number of a shared is the maximum number of processes that can use that object to solve consensus. The following is the operation with its corresponding consensus number.

Operation: Consensus Number

R/W Register: 1

Test And Set: 1

Get And Increment: 2

Swap: 2

CAS: ∞

CAS Operation

Initialize to -1

Pi:

Write my proposal in the prop array

Do R.CAS(-1, my_pid):

Fail: decide prop[R];

Succeed: decide prop[my_id];

References

[1] V. K. Garg, Introduction to Multicore Computing

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 19: November 1

Lecturer: Vijay Garg *Scribe: Zijiang Yang*

19.1 Wait Free Consensus Hierarchy

Consensus number of a shared object is the maximum number of processes that can use that object to solve consensus problem. Different objects have different consensus numbers, as shown in the following table.

Object Type	Consensus Number
Read&Write	1
Queue(Stack), Test&Set, Swap, Get&Increment, (2,1)Objects	2
...	...
Check&Set	Infinity

Based on the hierarchy of consensus objects, we can always use objects with higher consensus number to build objects with lower consensus numbers.

19.1.1 Read-Modify-Write Objects

A Read-Modify-Write object is the abstraction of an general object that has the following structures:

```
private int value;
public int synchronized getAndModify(int x) {
    int prev = value;
    value = f(value, x);
    return prev;
}
```

Each Read-Modify-Write object has an associated value and an atomic operation. The operation will set a new value based on function f and return the old value.

There are a lot of consensus objects which can be viewed as Read-Modify-Write object. Each has its own f function definition:

- Test&Set: f(v, x) = 1
- Swap: f(v, x) = x
- Get&Increment: f(v, x) = v + 1

Read-Modify-Write objects can be characterized by the property of funtion f:

- A Read-Modify-Write object is commute if
  $f_i f_j = f_j f_i$
  where f_i is the funtion applied by process i

- A Read-Modify-Write object is overwrite if
  $f_i(f_j(x)) = f_i(x)$ *for any i, j*
  where f_i is the funtion applied by process i

19.1.2 Consensus Number of Read-Modify-Write Object

A **trivial** Read-Modify-Write object is an Read-Modify-Write object whose f(v, x) = v. It can be viewed as a simple read operation.

Theorem: Any non-trivial Read-Modify-Write object with either commuting or overwriting property has consensus number equal to two.

Proof:

1. Firstly we need to prove that consensus number for RMW object is at least 2.

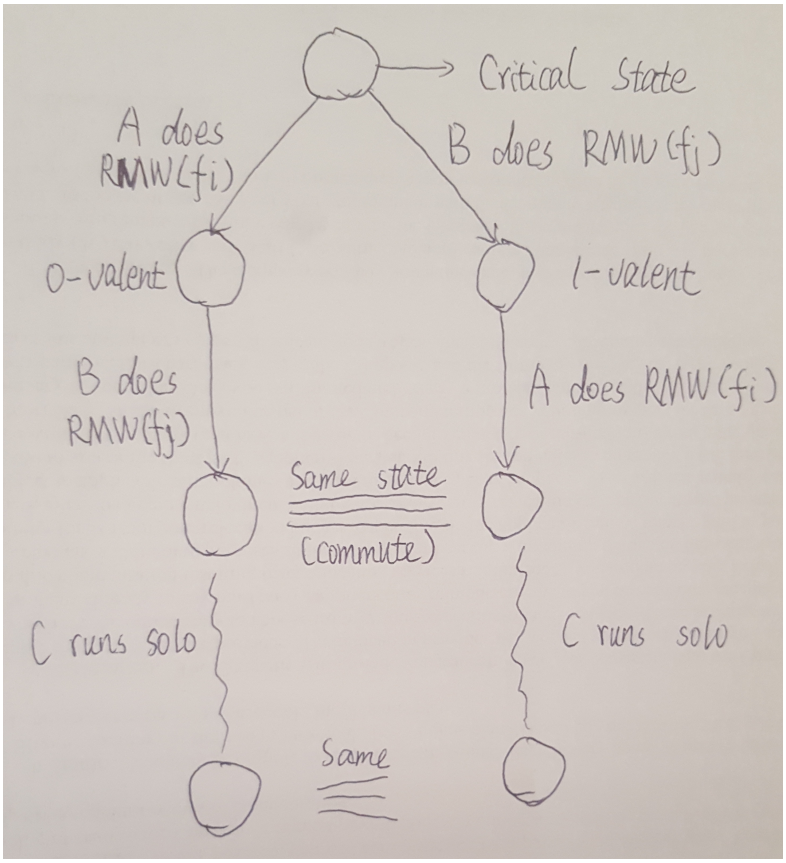
   It is easy to come up with an algorithm to solve consensus problem on 2 processes using the non-triviality property:

```
int[2] A, RMW r (init null)
Pi: write proposol on A[i]
     value = r.getAndModify(i)
     if(value == null) {
         I win
     } else {
         other process win
     }
```

   This algorithm works because the RMW object is non-trivial. Thus there exist v such that f(v, x) != v.

2. Then we need to prove that the consensus number of RMW object is at most 2.

   We can show that there is not a protocol that solves consensus for 3 processes using RMW objects.



In order to reach consensus, a protocol has to reach critical state. Suppose at critical sate A does RMW(fi) and reach to 0-valent state; B does RMW(fj) and reach to 1-valent state. Then at the 0-valent state B does RMW(fj); at the 1-valent state A does RMW(fi). After that, both path reaches the same state. Then C runs solo and will decide the same value from both state. But it contradicts the assumption that one state is 0-valent state and the other state is 1-valent state. Thus it is impossible to reach a critical state for RMW object on 3 processes.

Thus in conclusion, non-trivial Read-Modify-Write object with either commuting or overwriting property has consensus number equal to two.

19.1.3 2-Writes-1-Read Object

We already know that we cannot use atomic read-write register to solve consensus problem. However we can solve this problem if we are allowed to write 2 locations atomically. A 2-Writes-1-Read object (*(2, 1) object*) is an object which can write 2 locations and read 1 location atomically.

Following is a protocol that can solve 2 process consensus problem using (2, 1) object.

```
int[3] A, init null
P0: A[0] = a, A[1] = a (atomically)
P1: A[1] = b, A[2] = b (atomically)
whichever writes first wins
```

There are 4 possible configurations:

- [a, a, null] =>p0 wins
- [a, b, b] =>p0 wins
- [null, b, b] =>p1 wins
- [a, a, b] =>p1 wins

In this way we can solve consensus on 2 processes.

19.2 Universal Constructions

Deterministic Object: given a state and an operation, the resulting state is deterministic.

What we want to do is given some objects of consensus number m, construct a deterministic object with consensus number m' <= m.

An easy way would be to use a consensus object to decide who wins. However, in general consensus objects are not reusable.

The correct way would be use a linked list to record the invocation histories, and maintain a consensus object inside each node. When multiple processes wants to invoke methods, use consensus object of the head node to decide which thread wins.

The linked list is organized as follows:

```
Initial state -> apply method1 -> apply method2 -> apply method3
```

The invocation log is equivalent to the current state of the object.

Sudo code can be found in https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/tree/master/chapter7-consensus

19.3 Hashing

Hashing is a technology to map an object into an index. A HashTable uses hashing to decide where to store an item. The time complexity of HashTable is as follows:

Operation	**Time Complexity**
Insert	Average O(1)
Search	Average O(1)
Remove	Average O(1)

However, generally HashTable may has collisions which may damage the performance. We want to make collisions as infrequent as possible. There are different strategies in case of collision:

- **Chaining:** Store a linked list in each bucket. Add items into the linked list.
- **Open Addressing:**
  - Linear Probing: if the bucket h(i) is occupied, insert item into the next available bucket. This approach may cause clustering effect.
  - Quadratic Probing: if the bucket h(i) is occupied, then try $h(i)+1$. If still occupied, try $h(i)+2^2$, then $h(i)+3^2$, then $h(i)+4^2$ ...

We want to parallelize the chaining HashTable using lock. A naive solution is to hold a lock for all buckets and acquire the lock for all the operations. A better solution is to hold a lock for every bucket, thus we can parallelize the operations on different buckets. However this approach will waste a lot of spaces. An improvement would be hold a lock for every k buckets. Thus we can save a lot of space while maintain the parallel ability.

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 19: November 1

Lecturer: Vijay Garg *Scribe: Yue Zhao*

19.1 Wait Free Consensus Hierachy

Consensus number of a shared object is the maximum number of processes that can use that object to solve consensus problem.

Operation: R/W Register: Consensus number 1;
TestAndSet: Consensus number 2;
GetAndIncrement: Consensus number 2;
Swap: Consensus number 2;
CAS: Consensus number infinity;

19.1.1 Read-Modify-Write Objects

An Read-Modify-Write Object has the following structure:

```
private int value;
public int synchronized getAndModify(int x) {
        int prev = value;
        value = f(value, x);
        return prev;
}
```

Function f sets a new value and returns the old value:

- Test&Set: f(value, x) = 1;
- Swap: f(value, x) = x;
- Get&Increment: f(value, x) = value+1;

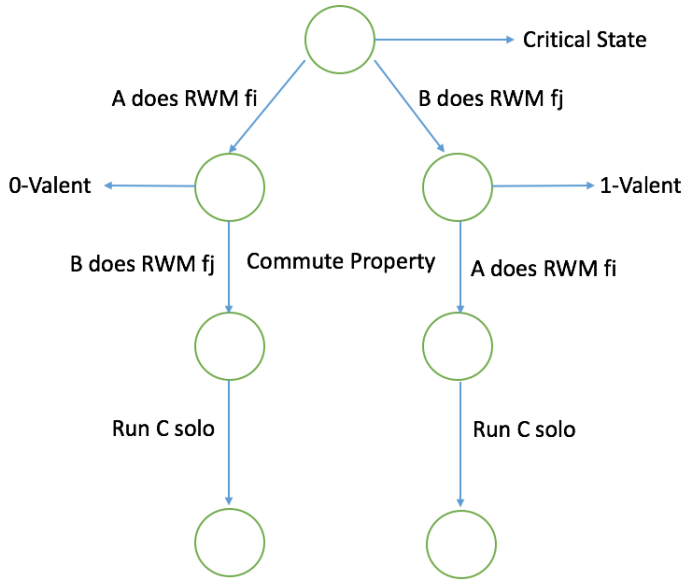
An read-and-modify object can be characterized by function f:
(1) RMW is commute if: fi fj = fj fi (for any i, j), where fi is the function applied by process i;
(2) RMW is overwrite if: fi(fj(x)) = fi(x) (for any i, j);

Theorem:
Any **non-trivial** (f is not identity, if f is idenity, f is read operation f(value, x) = value, which has consensus number 1) RMW object with either commuting or overwriting property has consensus number equals to two.

Proof(consensus number is at least 2):
Use non-triviality: Value init to null. P0 and P1 store their proposals to an int[2] array. There exists value

such that f(value)!=value. Then call RMW, if return initial value then choose my value; else choose the other value;

Proof(consensus number is at most 2):
There is no protocol that solves consensus for 3 processes using RMW object. Any protocol has to reach some critical state:



From critical state, A does RMW(fi) reach 0-valent then B does RMW(fj); for another path, B does RMW(fj) reach 1-valent then A does RMW(fj). Because RMW has commute property, these two path has same state, then C can run. For C, the initial state is same, so, C will always reach same consensus and it cannot decide on different things. However, for the first path, C should reach 0-valent and for the second path, C should reach 1-valent, which contradict with uni-valent property.

19.1.2 2-Write-1-Read-Object ((2, 1) Object)

2-Write-1-Read-Object: Write on two locations atomically, read any one location.
Protocol to solve consensus using (2, 1) object:

int[3] (initial with null);
Write proposal on int[3] array;
P0 writes (a, a) in location 0 and 1;
P1 writes (b, b) in location 1 and 2;
Whichever writes first wins.

Possible configurations during the process:
{a, a, null}, (P0 won at this point);
{a, b, b}, (P0 won);

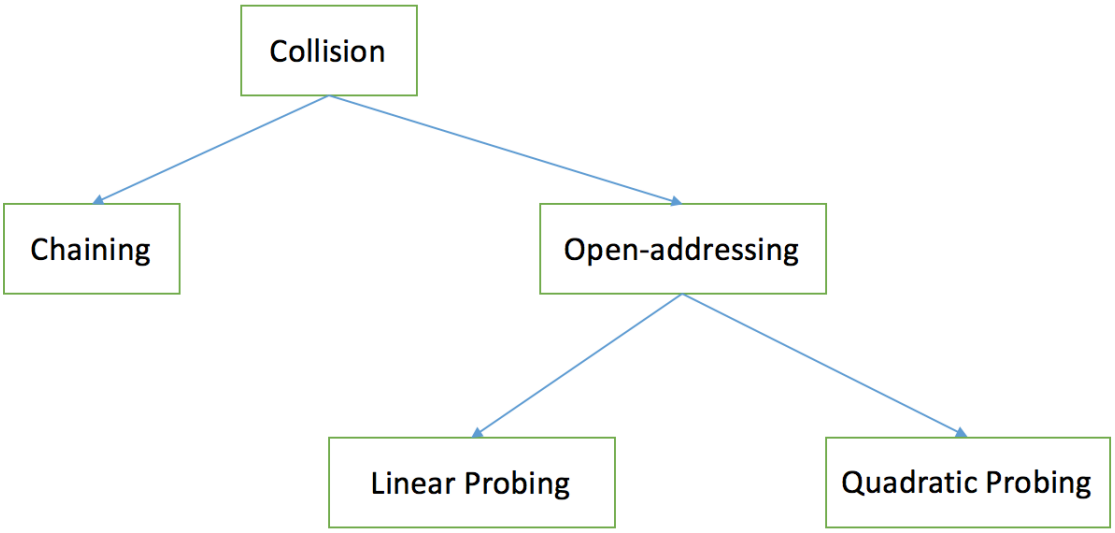
{null, b, b}, (P1 won);
{a, a, b} (P1 won);

This can solve consensus on 2 processes.

19.2 Universal Construction

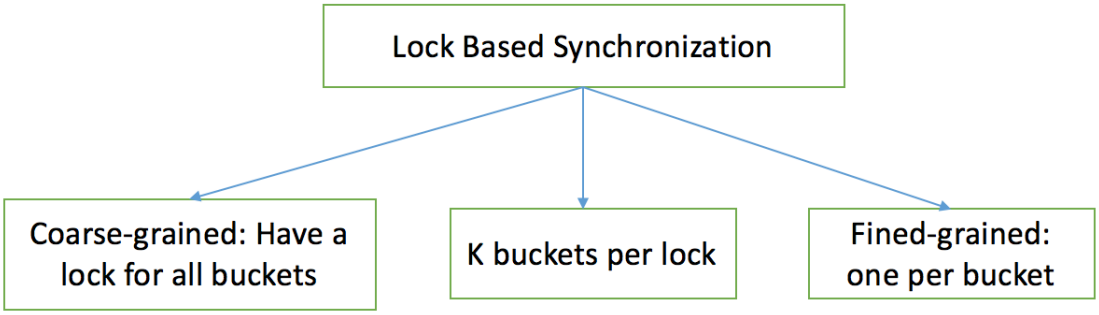
Given some objects of consensus number m, construct an object with consensus number less than or equal to M.
Deterministic Objects: state + operation cause a deterministic state.
Universal Construction only applies to deterministic state.
Correct Way: use a LinkedList, see code in github Initail state(Tail) ->apply method1 ->apply method2 ->apply method3(head)
Each time invoking a method, add a node in the linkedlist as the new head.

19.3 Hashing



Linear Probing: insert in the next empty slot. May cause clustering effect and damage performance.
Quadratic Probing: Try h(i) ->h(i)+1 ->h(i)+2^2 $->h(i)+3^2$ $->h(i)+4^2$

Parallel Chaining:


Lock Based Synchronization
Coarse-grained: Have a lock for all buckets
K buckets per lock
Fined-grained: one per bucket

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 20: November 3

Lecturer: Vijay Garg *Scribe: Haowei Sun*

20.1 Hashing Introduction

Hashing function is uniformly distributed and different items most likely have different hash values. It could be closed addressed, which means each item has a fixed bucket in table and each bucket contains several items or open addressed which means each item could end up in different buckets in table and Each bucket contains at most one item.

For a sequential closed hashmap, when adding items into the map, it may have the problem that buckets are getting too long and the way to solve it is resizing.

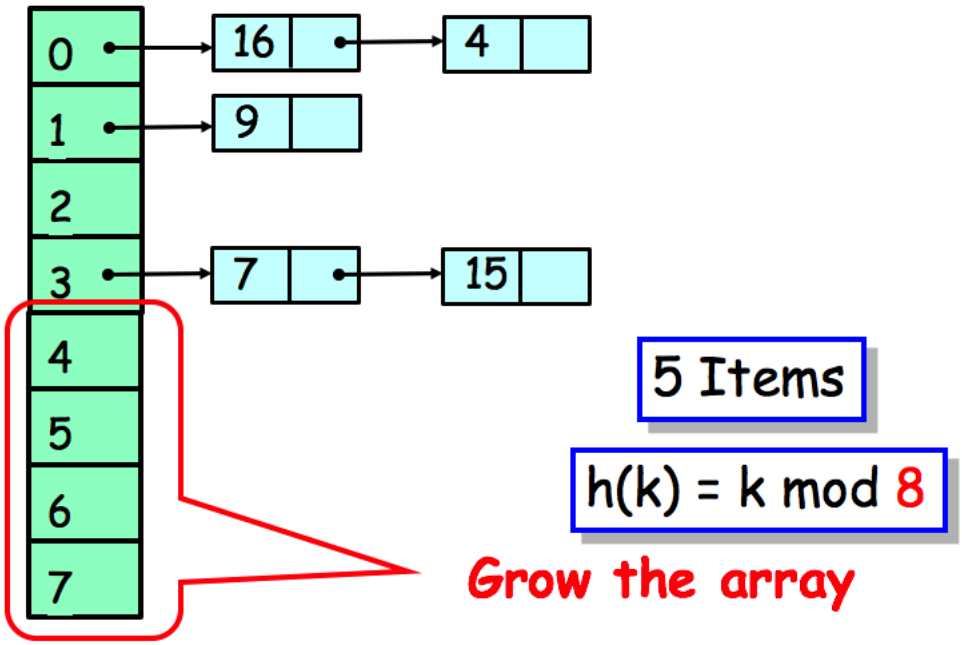


Figure 20.1: Resizing Example

20.2 Resizing Problem

The optimization of operations can be improved due to the frequency of use.When considering the condition to resize, we choose one policy:

- Global threshold : When 1/4 buckets exceed this value
- Bucket threshold : When any bucket exceeds this value

20.2.1 Fine-grained Locking Resizing

In fine-grained locking, each lock is associated with one bucket and table reference didn't change between resize decision and lock acquisition. In the following example, after resize, each lock is associated with two buckets.

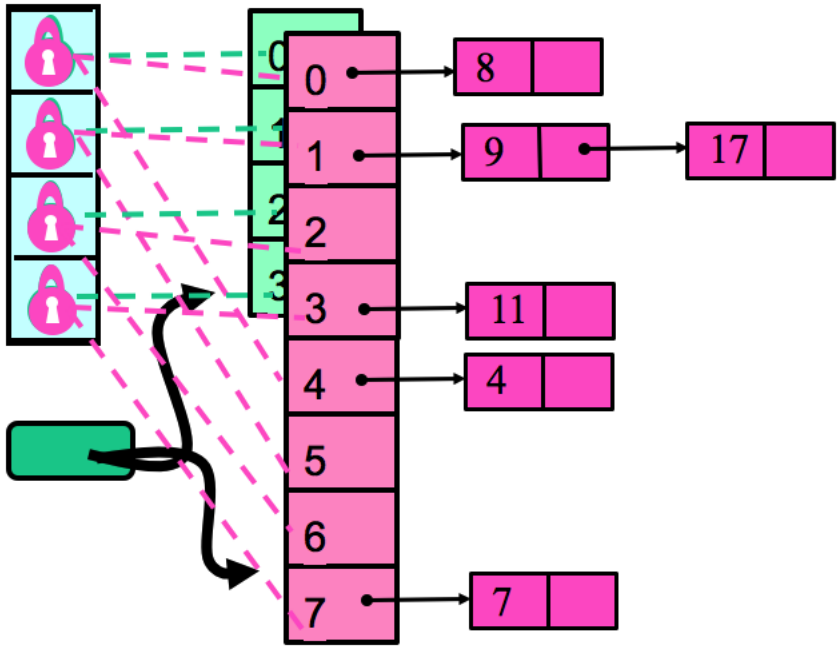


Figure 20.2: Fine-grained Locking Resizing

For read and write locks, they both return associated locks.

20.2.2 Lock-Free Resizing

Since the traditional way of moving items among buckets is hard, our idea was not to move them. Instead, we will flip the idea of bucket-based hashing on its head. We keep the items fixed and move the buckets. We keep the items in an ordered linked list. Even though all the items are reachable from the top bucket, the additional buckets are shortcuts that allow us to reach items in constant time. In order to allow us to redirect the bucket pointers in this recursive manner, we need to insert the items in a special order, and order that allows to recursively split buckets.

Then we need to map the real keys to the split-order.The map simply needs to reverse the order of the bits. That is, the split-order according to which any two keys should be inserted into the list is given by comparing the binary reversed representations of the keys. In split ordered hashing, the order is according to reserved bits and parent always provides a short cut.

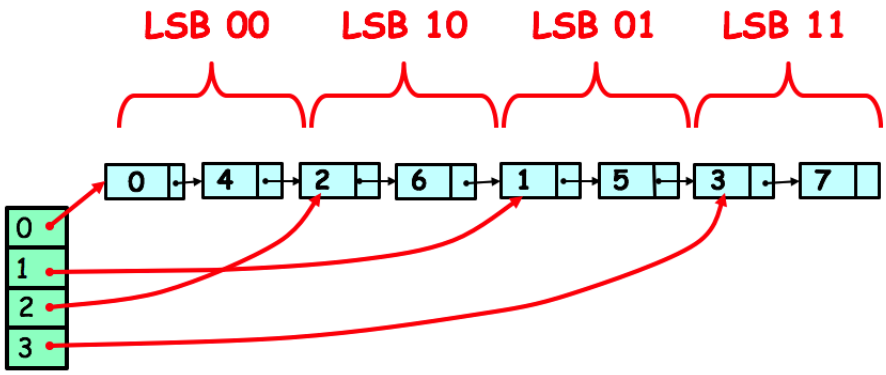


Figure 20.3: Recursive Split Ordering

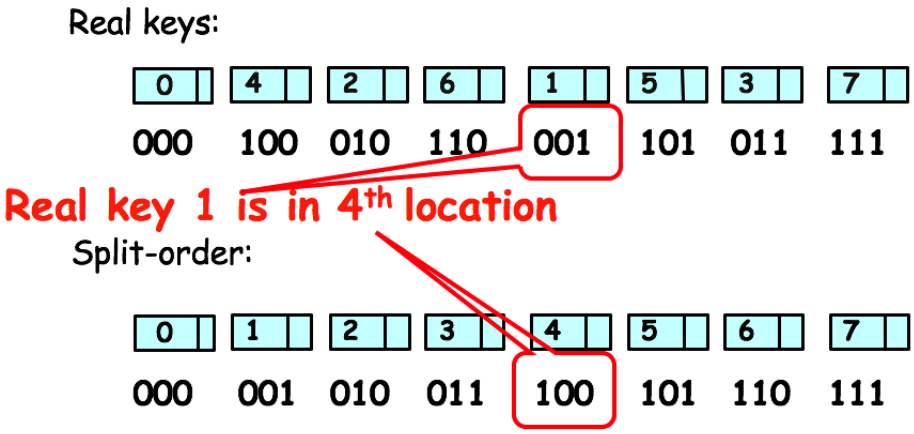


Figure 20.4: Recursive Split Ordering

We use sentinel nodes for each bucket to solve the problem of removing a node pointed by 2 sources using CAS. We set sentinel key for i ordered before all keys that hash to bucket i and after all keys that hash to bucket (i-1).We can now split a bucket in a lock-free manner using two CAS() calls. One to add the sentinel to the list and the other to point from the bucket to the sentinel.

20.3 Chained Hashing

20.3.1 Linear Probing

Advantages :

- Good locality with less cache misses

Disadvantages :

- As items/buckets increases, more cache misses

20.3.2 Cuckoo Hashing

Advantages :

- Contains() : deterministic 2 buckets
- No clustering or contamination

Disadvantages :

- 2 tables
- As items/buckets increases, relocation cycles
- As items/buckets = 0.5, Add() does not work

20.3.3 Hopscotch Hashing

The idea of hopscotch hashing defines neighborhood of original bucket. In neighborhood items are found quickly and we use sequences of displacements to move items into their neighborhood.

Advantages :

- Good locality and cache behavior
- Good performance as table density increases with less resizing
- Pay price in Add() not in frequent Contains()
- Easy to parallelize

References

[1] Maurice Herlihy, Nir Shavit, Hashing and Natural Parallism, Companion slides for The Art of Multiprocessor Programming .

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 20: November 3

Lecturer: Vijay Garg *Scribe: Muhammad Raza Mahboob*

Agenda

This lecture focused on Hashing and introduction of Transactional Memory. Major concepts discussed -:

- Hashing basics
- Resizing Problem for Closed Addressing
- Chained Hashing
- Transactional Memory

20.1 Introduction

Link list provides us with operations having O(n) complexity. In order to reduce the time complexity to constant time, we use Hashing. Hashing function is applied to the object and the result is an index (integer) and the item should be stored at that index in the bucket array. While Hashing we assume that the items are uniformly distributed so that each item most likely have a different hash value. Hashing comes in 2 different flavors:

- Closed Addressing: Each item has a fixed bucket in the table and each bucket can contain several items in the form of a linked list.
- Open Addressing: Each item could end up in a different bucket in the table and each bucket contains at most one item.

20.2 Resizing Problem

Resizing is to allocate a new bigger bucket array and redistribute the existing items using the updated hash function which uses the new size. For closed addressing, we require resizing when the items in the individual buckets increases over a certain threshold which causes a linear search on the bucket list for the Hash operations. There are 2 different criteria for resizing:

- Global threshold: When the size of X number of buckets increases over a certain threshold. (i.e X = 1/4)
- Bucket threshold: When the size of any bucket increases over the threshold value.

The new size of the bucket array is generally twice the original size in order to amortize the cost of copying the existing items in the new bucket array.

There are 3 different techniques of resizing: coarse grained locking, fine grained locking and lock free resizing.

20.2.1 Coarse Grained Locking

The main idea of Coarse grained locking is very simple, there is one lock for the entire bucket array. Acquire that lock and start the resizing process which is to create a new bucket array and copy over all the existing elements in the new array using the updated hash function. Coarse grained lock resizing is very simple and easy to implement but it has the issue of sequential bottleneck. Only one thread can operate at any time which causes performance issues.

20.2.2 Fine Grained Locking

In fine grained lock resizing, there is a separate lock associated with each bucket which has to be acquired before performing any operation (i.e. insert) on that bucket. So while resizing, the first step is to acquire all the locks in ascending order and make sure that the table reference didn't change between the resize decision and the lock acquisition. After acquiring the locks, allocate the new super sized table and start transferring the existing elements into the new table. As shown in 20.1, we take individual locks on each bucket, create a new bigger bucket array, copy over the existing elements and then map any new elements to the new array.

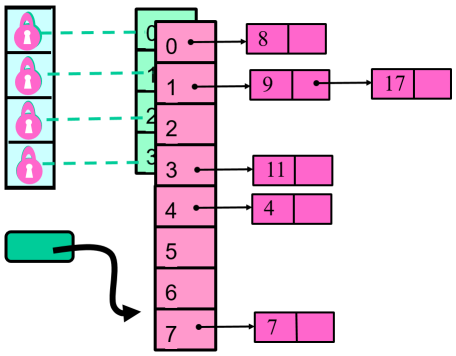


Figure 20.1: Fine Grained Locking

Striped Locks: Instead of allocating new locks for the new array, we use the existing locks. Since the size of the new array is double the size of original array, we use the each existing lock to cover one more bucket in the new array based on the symmetry.

Read Locks: We leverage the insight that most common operations is search on the hash table and so we keep 2 different kinds of locks: read lock and write lock. So multiple threads can take the read lock and start searching on the hash bucket.

20.2.3 Lock Free Resizing

The idea of resizing is to remove the existing element and put it at a new location in the new array. These are 2 different operations but CAS(CompareAndSwap) only allows us to do one operation so we either need

a double-compare-and-swap operation or a new idea for lock free resizing. The idea is to keep the items fixed in an ordered linked list and move the buckets. All the items are reachable from the top bucket but the additional buckets are shortcuts that allow us to achieve constant time retrieval.
We keep all the elements in a linked list with special order. The order will enable us to easily point new buckets in the linked list. We use the Recursive Split Ordering meaning that we recursively divide the existing bucket into two based on the least significant bit of the item. The ordering is based on the least significant bits. Starting from the least significant bit, start ordering and until all the elements are ordered. An example is given in 20.2 where each element is first sorted on the least significant bit and then on the second last significant bit. So if you have, two keys: a and b, a precedes b if and only if the bit reversed (least significant i bits) value of a is smaller than the bit reversed value of b as shown in 20.3.

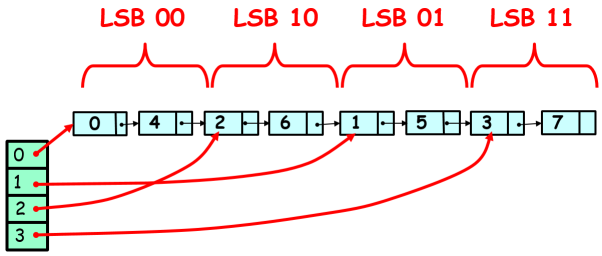


Figure 20.2: Special Ordering of items

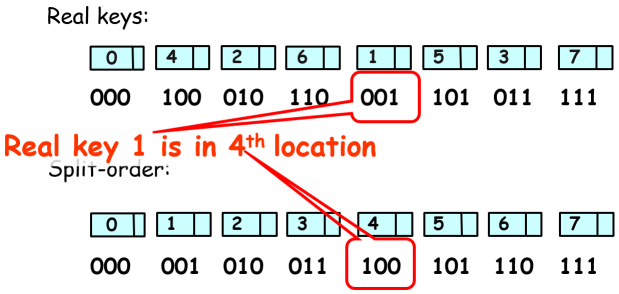


Figure 20.3: Reverse Ordering

Each bucket is mapped according to the reverse order of the bits of the first element in the linked list found by traversing from an existing bucket. But this causes an issue when removing a node having both the bucket pointer and the previous node pointer in the linked list using a single CAS operation. In order to cater for this problem, we use Sentinel Node for each bucket and initialize them whenever we need to point a bucket. So first create the Sentinel node and point it to the real node in the list. After that adjust the previous pointer to point to this sentinel node and finally add the bucket pointer to the sentinel.

20.3 Closed Hashing

20.3.1 Linear Probing

In open addressing we keep the bucket array which hold the actual items instead of pointing to a new list. Whenever an item x needs to added, we apply the hash function on the item and try to put the item at H(x)

index. If that index is already occupied then we move forward from that index until we find an empty spot to allocate x. For the Contain(x) operation, we start from H(x) and then search linearly until we find x or there is an empty bucket.
Advantages:

- Good Locality which means less Cache misses

Disadvantages:

- As the number of total items ratio to bucket size increases (M/N) more cache misses because we maybe searching in several unrelated buckets
- Clustering effect of keys into neighboring buckets

20.3.2 Cuckoo Hashing

Cuckoo hashing leverages the **power of 2** concept and creates 2 bucket arrays instead of one and keep two different hash functions, one for each array. For Insert(x) operation, we take the H1(x) and look at the resulting index in first array. If it's empty then just place x there else if there is an element y at H1(x) then take H2(x) and look at the resulting position in the second array. If it is empty then place x there else evict y, place x at H1(x) and recursively insert y at H2(y) and so on. The same idea is used for searching purposes.
Advantages:

- No clustering issue

Disadvantages:

- 2 tables
- The hash functions could be complex
- As M/N increases, the relocation cycle increases

20.3.3 Hopscotch Hashing

In Hopscotch hashing we have a single array and a simple hash function but we define neighborhood of original bucket. While adding an item, probe linearly to find an open slot and move the empty slot via sequence of displacements into the hop-range of H(x). For Contain(x) operation, search in at most H buckets (the hop range) based on the hop-info bitmap.
Advantages:

- Good locality and cache behavior
- Good performance as table density increases
- Optimize the common operation (Contains) at the expense of Insert

Disadvantages:

- Does not work well in all scenarios like Cuckoo Hashing

20.4 Trasactional Memory

Unitl now we have seen 2 different mechanisms to achieve mutual exclusion i.e. Lock based and lock free. Transactional memory is another concept used in this regard.

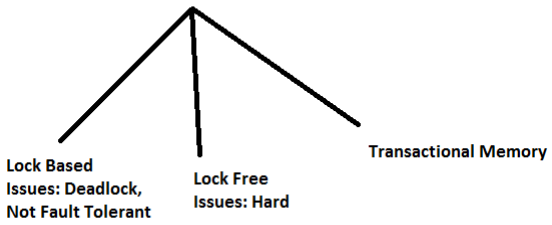


Figure 20.4: Different Concepts for Mutual Exclusion

The concept of Transactional memory is derived from Databases where we have similar issues like Concurrency and Failures. Jim Gray pioneered the concept of transactions for databases. The main construct of a Transaction is that we have *begin.transaction* at the start followed by several reads, writes and abort operations and then finally *end.transaction* which is also called commit transaction. Transactions in databases provide 4 guarantees which are commonly known as ACID:

- Atomicity : All or nothing
- Consistency : transaction will bring the database from one valid state to another
- Isolation : concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially
- Durability : persistence even in the events of power loss or error

References

[1] Vijay K Garg, Introduction to Multicore Computing

[2] Maurice Herlihy, Nir Shavit, Hashing and Natural Parallism, Companion slides for The Art of Multiprocessor Programming

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 21: November 8

Lecturer: Vijay Garg *Scribe: Edward Clinton, TEC553*

21.1 Agenda

This lecture deals with transaction memory. We will focus on 4 characteristics of transactions:

- A - Atomicity
- C - Consistency
- I - Isolation
- D - Durability

Transaction memory: we will look at Atomicity, Consistency and Isolation, not worried about Durability

21.2 Introduction

A transaction is a sequence of events that appears indivisible and instantaneous to an outside observer and has four properties:

- Atomicity: All constituent actions in a transaction complete successfully, or none of these actions appear to start executing.
- Consistency: A transaction can modify the state of the world, i.e. data in a database or memory. These changes should leave this state consistent. If the transaction completes successfully, then the system will be in a valid state. If an error occurs then any change will be rolled back. For example, when transfering money from account A to account B, the system is consistent if the total of all accounts is constant. If an error occurs after removing money from account A but before adding it to account B, the system is no longer consistent since the total would have changed (money disappeared). By rolling back the removal from account A, the system will be back in a consistent state.
- Isolation: Each transaction produces a correct result, regardless of which other transaction are executing concurrently. (i.e. the system state would be obtained if transactions were executed serially)
- Durability: Once a transaction commits, its result is permanent (stored in a durable media such as disk)

Transactions is a language construct that discharges programmers from the management of synchronization issues. (synchronization constructs such as locks are hidden from the programmer).

21.3 Serializability

The correctness condition in transactions is called serializability. Serializability states that the concurrent execution of transactions is equivalent to some serial execution of all transactions. In other words, it appears that one transaction finishes before the next transaction starts. For example, if T_1, T_2 and T_3 execute concurrently, the net result should be equivalent as if these transactions ran sequentialy such as: $T_2 -> T_1 -> T_3$ or $T_3 -> T_2 -> T_1$ etc.. (there are n! possibilities).

21.4 Transaction construct

In the database world, the programmer does not have to worry about using locks for synchronization. Synchronization is all taken care of by the underlying system. To enable transactions we add one additional construct to import in a programming language as follows:

```
atomic{
//do whatever you want
        x:=2;
    while(){
        //do stuff
    };
}
```

This atomic construct assures that the block is executed as one single step such that a process running concurrently can only observe a state that is either before or after the atomic block executes.

At first this construct may seem equivalent to the java synchronized block. The difference is that the synchronized block locks the object implementing the synchronized method such that operations in the synchronized blocks cannot be interleaved. However in transactions, its OK to interleave operations inside an atomic block as long as the effect is the same as if each block executed separately.

Another difference is that transactions use speculative (optimistic) execution to execute multiple transactions in parallel (hoping that nothing bad happens). If contention is low then there is a good chance that nothing bad will happen. If there is a conflict between two atomic blocks the system will resolve it using aborts and roll-backs.

Some issues of locks vs transactions:

- **Deadlocks** , There is no notion of deadlocks in TM. If there is a conflict between some transactions (atomic blocks). The system will resolve it using aborts. Aborts are not available in Java synchronization.
- **Composability problem**. Given two data structures Q1 and Q2 we perform following operation atomically:

  ```
  atomic{
  x = q1.deq();
  q2.enq(x);
  }
  ```

  Invariant: No process shall find a state where x is not in q1 nor q2 (i.e. X has to be in q1 or q2 and not in both). In TM, this is achieved trivially by using atomic block. If using locks, this problem

is non-trivial. Locks cannot be easily composed. In Queue implementations based on monitors, each method acquires the lock internally, so it is essentially impossible to combine two method calls this way.

21.5 Opacity

Opacity is a correctness condition for TMs. An execution satisfies opacity if all the committed transactions and the read prefix of the aborted transactions appear as if they have been executed serially in agreement with real-time occurrence order. Opacity can be viewed as an extension of serializability with the additional requirement that even non-committed transactions are prevented from accessing inconsistent states.

For example:

```
x = 1;
y = 0;

atomic{
x++;
y++;
}

atomic{
z = z/(x - y) //<- can raise a divide by zero exception
//zombie transaction, read values that are impossible (ex x == y)
//if all transactions occured serially

}
```

Opacity guarantees that x != y since x != y is true in all the consistent global states of the multiprocess program (no transaction can read values from an inconsistent global state).

21.6 Implementation and Algorithm

There are multiple implementations (ex Deuce STM , Hardware Transaction memory (HDM) for small transactions). Current state of the art : no algorithms are currently used because of their high overhead (higher than locking , or lock free). But there is optimism that transactions will become useful someday.

In HDM, the idea is that since in a normal machine all updates are stored in the cache and not in main memory (cache block is written-back to main memory before getting evicted, only true for write-back cache) the abortion of a transaction can be implemented by not writing the cache block(s) back to main memory.

We will focus on an STM (software transaction memory) system called TL2 (transactional locking 2) due to D. Dice, O. Shalev, and N. Shavit (2006). This STM system statisfies the opacity consistency condition. The TL2 algorithm is posted on canvas.

The TL2 algorithms uses locks (lock usage is minimized). Locks are used when a transaction is about to be committed to check if the states are consistent and to determine if the transaction needs to be committed or aborted. TL2 also uses atomic instructions.

Properties and control variables of TL2:

- CLOCK: clock is a fetch and add register initialized to 0 and is used as a logical clock to measure the progress of the system. It represents the number of transactions that have been committed so far.
- All registers are MWMR atomic registers. Each register has two fields: *value*, which contains the value of the register, *date* which contains the date of its last update. A lock is associated with each register.
- Each process has two local variables *lrs(T)* and *lws(T)* where T is the transaction currently executing. *lrs(T)* (local read set) contains the names of all registers read by T until now. *lws(T)* (local write set) contains all the registers written up to now.

We will look at an implementation of a *begin* , *end* transaction and how to read and write objects.

```
begin()
{
lst(T) <- {} ; rst(T) <- {} ;
birthdate(T) <- CLOCK + 1 ;
return;
}
```

When reading an object we first read it into a local copy

```
X.readT()
{
if(there is a local copy lc(XX) of XX){
        then return(lc(XX).value)
    }
    else lc(XX) <- copy of XX read from the shared memory;
    if(lc(XX).date < bitrhdate(T)){
        then lrst <- lrst U {X}; return(lc(XX).value)
        }
        else return(abort)
}
```

If the date of the register is not less than the birth date of the transaction then we must abort the transaction to guarantee opacity. A transaction cannot read from the future.

```
X.write(V)
{
if(there is no local copy of XX) then allocate local space lc(XX) for a copy
lc(XX).value <- v ; lwstt <- lwst U {X};
return(ok) //write never aborts
}
```

If a transaction reaches its last statement without having been previously aborted, it invokes the **commit** operation. That operation decides the fate of T by returning commit or abort.

```
try_to_commit()
{
lock all the objects in (lrst U lwst);
for each X in lrst do
{
        //the date of XX is read from the shared memory
    if XX.date >= birthdate(T) then release all the locks; return abort
```

```
}
write_date <- CLOCK.fetch&add();
for each X in lwst do XX <- (lc(XX).value, write_date)
release all the locks; return commit


}
```

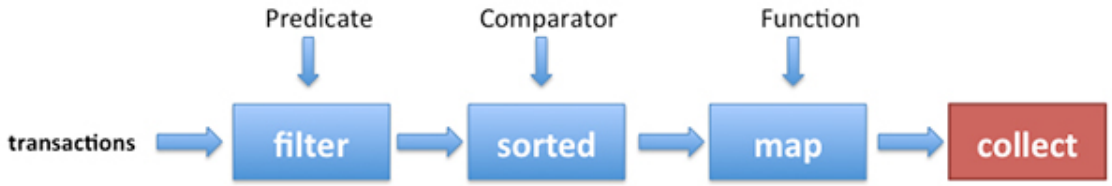
If the read validation succeeds (as explained above), the process has to compute the new date that has to be associated with all the writes issued by T. All written local copies are then stored to shared memory.

This algorithm is slow due to all the copy overhead and local copy write back. When using locks, there is isn't a copy overhead.

21.7 Intro to Java Streams

Stream is a JAVA abstraction introduced to JAVA SE 8. A stream is a sequence of aggregate operations as presented below.

Figure 21.1:




```
List<Integer> transactionsIds =
    transactions.stream()
                .filter(t -> t.getType() == Transaction.GROCERY)
                .sorted(comparing(Transaction::getValue).reversed())
                .map(Transaction::getId)
                .collect(toList());
```

Figure 21.1 illustrates the Java SE 8 code. First, we obtain a stream from the list of transactions (the data) using the stream() method available on List. Next, several operations (filter, sorted, map, collect) are chained together to form a pipeline, which can be seen as forming a query on the data. Stream operations return streams themselves

To parallelize the code, we can use parallelStream() instead of Stream().

References

[R12] M. Raynal, Concurrent Programming: Algorithms, Principles and foundations (2012), pp. 277-289.

EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 21: November 8

Lecturer: Vijay Garg *Scribe: Shengwei Wang*

21.1 Transaction Memory

21.1.1 Transaction

For transcations:

- Atomicity
- Consistency
- Isolation
- Durability

For TM, we are looking at ACI.

Generally, a transaction can be described as:

```
begin transaction
        read(x);
        write(z);
        read(y);
        ...
end transaction
```

There are several read/write operations.

21.1.2 Serializability

The execution is equivalent to some series execution of all transactions.

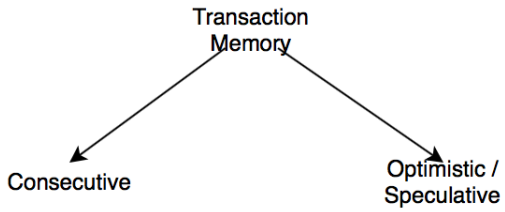
Databases use locks, but not necessary for programmers.

```
atomic {
        x = 2;
        while()
        ...
}
```

Former structure :

```
synchronized (this) {
                ...
        }
```

Difference : By using synchronized key word, we are actually acquiring locks.



When we use locks, we may have problems:

- Deadlocks
- Composability Problem

Example for Composability Problem :

BLOCKING QUEUE q1, q2

To do : get item from either of the queue if not empty.

Problem : when we do q1.deq(), and q1 is empty, then blocking.

21.1.3 Opacity

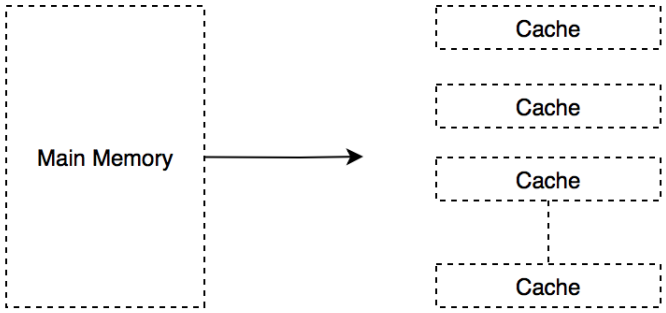
An execution satisfies opacity if all the committed transactions and the read prefix of the aborted transactions appear as if they have been executed serially in agreement with real-time occurrence order.

```
Var x, y init 0

atomic {
        x++;
        y++;
}

atomic {    //Zombie Transaction
        if (x != y) {
                while (true);
        }
}
```

21.2 Software Transaction Memory

There are multiple algorithms.
Deuce STM (Software Transaction Memory)

In hardware TM:



we store data from memory to cache. When abort, we just throw away data in cache.

TL2

21.2.1 Key Notion

- fetch&add
- Clock
- Local copies
- Dates of local copies

Put timestamp for each transaction. If the birthday of transaction 1 if 75, and the birthday of transaction 2 is 80, then transaction 1 is completely earlier than transaction 2.

- read set of transaction : lrst
- write set of transaction : lwst
- local copy : lc (xx)

21.2.2 operations

Begin of transcaion

```
begint()
lrst <- {}; lwst <- {};
birthday(T) = clock + 1; //atomic
```

Read opreation: X.read()

```
if there is a local copy of xx
then
        return lc(xx).value
else
        lc(xx) <- copy of xx from shared memory
        if lc(xx).date < birthday(T)
        then
                lrst(T) <- lrst(T) U {x};
                return lc(xx).value
        else
                abort
```

Write operation: X.write(v)

```
if there is no local copy then allocate space for lc(xx)
loc(xx).value <- v;
lwst <- lwst U {x}
```

Commit operation : Commit()

```
lock all x in lrst, lwst
for all xx <- lrst do
        if xx.date >= birthdate (T)
        then
                release lock;
                abort
write_date <- clock.fetch&add();
for all xx <- lwst do
        xx <- (lc(xx), write_date);

unlock;
return commit;
```

The algorithm is slow because of making local copies When abort, start over again.

21.3 Stream Programming

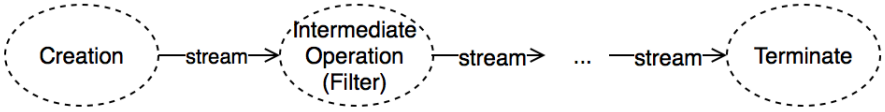
It's like functional programming.
Parallelism is easy if objects don't change.

- lamda function : nameless function
- streams
- parallel stream

lamda Funtion :

```
//using lamda function as the second parameter
Arrays.sort(data, (a,b) -> (b - a));
```

Stream : Think in high level



EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 22: November 10

Lecturer: Vijay Garg *Scribe: Zihan Yang*

Agenda

This lecture is a general introduction of Stream, a new abstract layer in Java 8:

- Stream basics
- Stream and Collection
- Simple examples of Stream
- 2 puzzles

22.1 Introduction

A stream is a sequence of elements supporting sequential and parallel aggregate operations.

To perform a computation, stream operations are composed into a stream pipeline. A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc), zero or more intermediate operations (which transform a stream into another stream, such as filter(Predicate)), and a terminal operation (which produces a result or side-effect, such as count() or forEach(Consumer)).

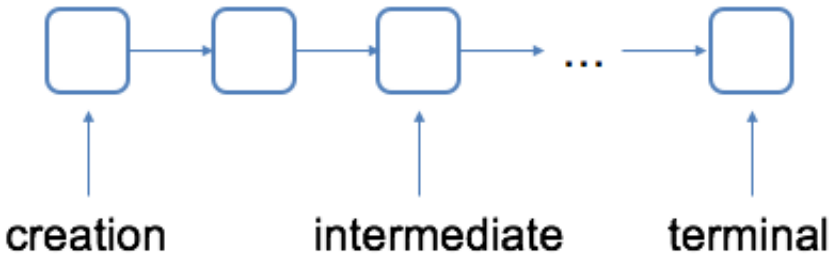


Figure 22.1: Stream in pipeline fashion

Stream pipelines may execute either sequentially or in parallel. This execution mode is a property of the stream. Streams are created with an initial choice of sequential or parallel execution. For example, Collection.stream() creates a sequential stream, and Collection.parallelStream() creates a parallel one. Operations on a sequential stream are processed in serial by one thread. Operations on parallel stream stream are processed in parallel by multiple threads. Most of the methods in the Streams API produce sequential streams by default.

22.2 Stream basics

Stream	Collection
Concept of time	Concept of space
Use it only once	Use it any time
Focus on aggregate operations	Focus on how to store and access elements
Number of elements could be infinite	Always finite
Lazy evaluation	All the elements are always there

In order to compare stream with collection, Fig 22.2 shows a snippet of code in which we square every odd element in an array and return its sum using collection and stream. With the help of stream, we can implement it just in one line. By contrast, the collection implementation is much longer.

```
1
2   /*Square each element in an array and returrn the sum*/
3
4   /*Using Collections*/
5   List<Integers> numbers = Arrays.asList(1,2,3,4,5);
6   int sum = 0;
7   for(int n:numbers){
8       if(n%2==1){
9           int square = n*n;
10          sum = sum + square;
11      }
12      System.out.println(sum);
13  }
14
15  /*Using Stream*/
16  List<Integers> numbers = Arrays.asList(1,2,3,4,5);
17  int sum = numbers.stream().filter(n->n%2==1).map(n->n*n).
18            reduce(0,Integer::sum);
```

Figure 22.2: Comparison between Collection and Stream

Even though Streams and Collections seem to share some superficial similarities. They actually serve for different goals. Collections are primarily concerned with the efficient management of, and access to, their elements. By contrast, streams do not provide a means to directly access or manipulate their elements and are instead concerned with declaratively describing their source and the computational operations which will be performed in aggregate on that source.

Besides, streams are not reusable. A stream can't be reused after calling a terminal operation. Look at the code in Fig 22.3, everything works well in line 13, but the line 16 should be commented because the streams can only be consumed once. However, all the elements in collections are always there and could be operated at any time. Streams are lazy because computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed. To some extend, stream is more like a concept of time while collection is of concept of space.

Another difference is that a collection cannot represent a group of infinite elements whereas a stream can. A stream can pull its elements from a data source. The source can be a collection, an I/0 channel or a function which can generate infinite number of elements.

```java
package lambdasinaction.chap4;

import java.util.*;
import java.util.stream.*;
import static java.util.stream.Collectors.toList;


public class StreamVsCollection {

    public static void main(String...args){
        List<String> names = Arrays.asList("Java8", "Lambdas", "In", "Action");
        Stream<String> s = names.stream();
        s.forEach(System.out::println);
        // uncommenting this line will result in an IllegalStateException
        // because streams can be consumed only once
        //s.forEach(System.out::println);
    }
}
```

Figure 22.3: A stream is not resuable

22.3 Simple examples of Stream

22.3.1 Java Stream Create

We can create stream in the following ways:

- Create Streams from values
- Create Streams from Empty streams
- Create Streams from functions
- Create Streams from arrays
- Create Streams from collections
- Create Streams from files
- Create Streams from other sources

For detailed descriptions and examples, please refer to: `http://www.java2s.com/Tutorials/Java/Java_Stream/0040__Java_Stream_Create.htm`

22.3.2 Java Stream opearations

There are two types of operations:terminal and intermediate. The commonly used stream operations are listed as follows:

- Distinct(intermediate): Returns a stream consisting of the distinct elements by checking equals() method.

- Filter(intermediate): Returns a stream that match the specified predicate.
- FlatMap(intermediate): Produces a stream flattened.
- Limit(intermediate): truncates a stream by number.
- Map(intermediate): Performs one-to-one mapping on the stream
- Peek(intermediate): Applies the action for debugging.
- Skip(intermediate): Discards the first n elements and returns the remaining stream. If this stream contains fewer than requested, an empty stream is returned.
- Sorted(intermediate): Sort a stream according to natural order or the specified Comparator. For an ordered stream, the sort is stable.
- allMatch(terminal): Returns true if all elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
- anyMatch(terminal): Returns true if any element in the stream matches the specified predicate, false otherwise. Returns false if the stream is empty.
- findAny(terminal): Returns any element from the stream. Returns an empty Optional object for an empty stream.
- findFirst(terminal): Returns the first element of the stream. For an ordered stream, it returns the first element; for an unordered stream, it returns any element.
- noneMatch(terminal): Returns true if no elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
- forEach(terminal): Applies an action for each element in the stream.
- reduce(terminal): Applies a reduction operation to computes a single value from the stream.

For detailed descriptions and examples, please refer to: `http://www.java2s.com/Tutorials/Java/Java_Stream/0200__Java_Stream_Operations.htm`

In the class, Prof.Garg walked us through some of the operations and concepts of streams using the examples on the github: `https://github.com/java8/Java8InAction`, including:

- syntax of **lambda function**(`https://github.com/java8/Java8InAction/blob/master/src/main/java/lambdasinaction/chap3/Lambdas.java`)
- **filter** operation(`https://github.com/java8/Java8InAction/blob/master/src/main/java/lambdasinaction/chap1/FilteringApples.java`)
- **sort** operation(`https://github.com/java8/Java8InAction/blob/master/src/main/java/lambdasinaction/chap3/Sorting.java`)
- **reduce** operation(`https://github.com/java8/Java8InAction/blob/master/src/main/java/lambdasinaction/chap5/Reducing.java`)
- mechanism of **lazy evaluation** of stream(`https://github.com/java8/Java8InAction/blob/master/src/main/java/lambdasinaction/chap5/Laziness.java`)
- **parallel streams**(`https://github.com/java8/Java8InAction/blob/master/src/main/java/lambdasinaction/chap7/ParallelStreams.java`)

References

[1] Tutorials on Java Streaming, http://www.java2s.com/Tutorials/Java/Java_Stream/index.htm

[2] Java8InAction on github, https://github.com/java8/Java8InAction/tree/master/src

[3] Documentation of Java stream on Oracle Help Center, https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html