Parallel Programming
Exercise Session 13

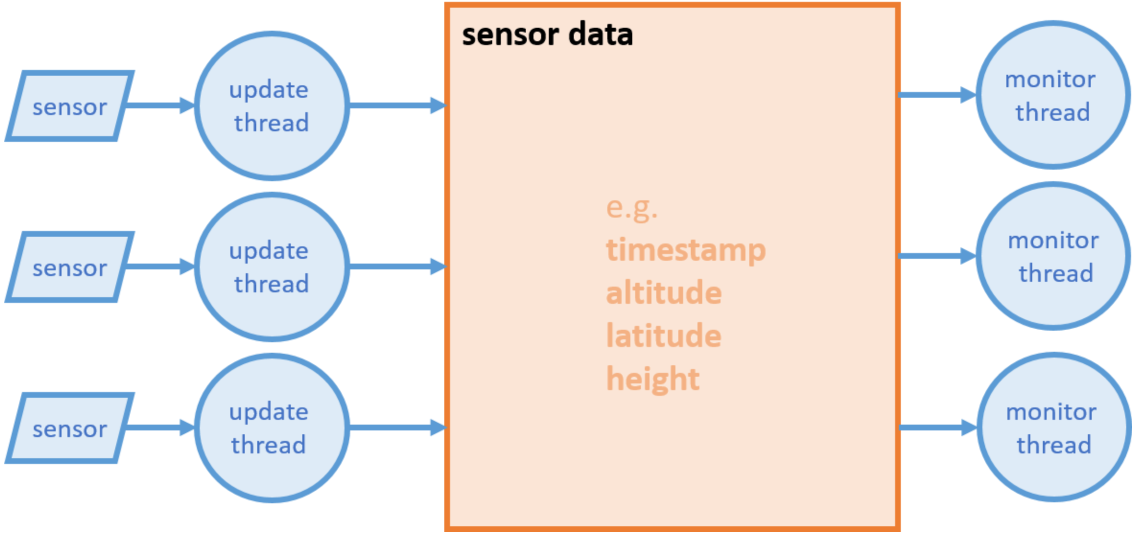
Outline

- Post-Discussion Assignment 12
- Pre-Discussion Assignment 13
- Exam Questions & Theory Recap
- Kahoot (credits: @Bauboo)

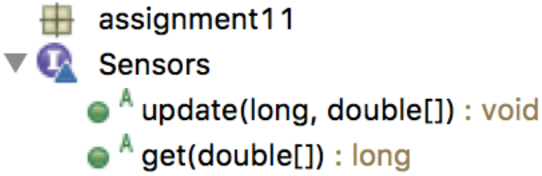
Feedback: Assignment 12

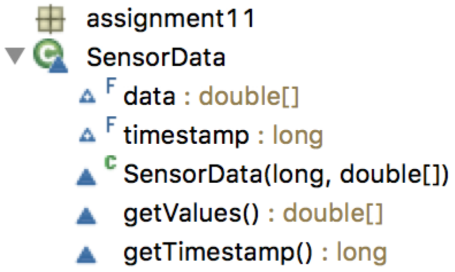
Assignment 12

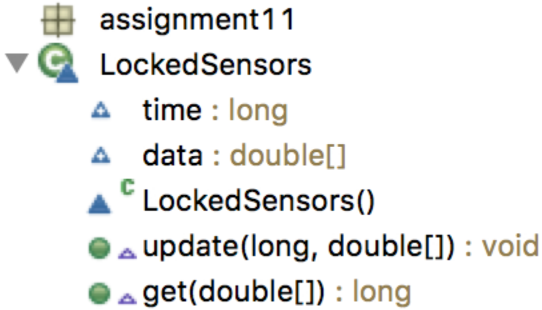
- Multisensor System.

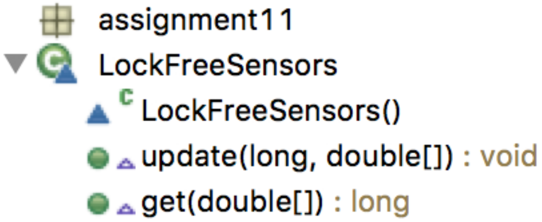


Multisensor System









LockedSensors

```
class LockedSensors implements Sensors {

    long time = 0;
    double data[];

    private ReadWriteLock lock;
    private Lock readlock;
    private Lock writelock;

    LockedSensors() {
        this(new ReadWriteMonitorLock());
    }

    LockedSensors(ReadWriteLock l){
        time = 0;
        lock = l;
        readlock = lock.readLock();
        writelock = lock.writeLock();
    }
```

```
public long get(double val[])
{
    readlock.lock();
    try{
        if (time == 0)
            return 0;
        else{
            for (int i = 0; i<data.length; ++i)
                val[i] = data[i];
            return time;
        }
    }finally {
        readlock.unlock();
    }

}
```

```
public void update(long timestamp, double[] data)
{
    writelock.lock();
    try{
        if (timestamp > time) {
            if (this.data == null)
                this.data = new double[data.length];
            time = timestamp;
            for (int i=0; i<data.length;++i)
                this.data[i]= data[i];
        }
    }
    finally {
        writelock.unlock();
    }
}
```

Lock implementation

```java
public class ReadWriteMonitorLock implements ReadWriteLock{
    private Lock readerlock = new ReadMonitorLock(this);
    private Lock writerlock = new WriteMonitorLock(this);

    //Invariant 0<=readers /\ 0<=writers<=1 /\ readers*writers=0
    private int readers=0;
    private int writers=0;

    private int writersWating=0;
    private int readersWating=0;
    private int readersToWait=0;

    @Override
    public Lock readLock() {
        return readerlock;
    }

    @Override
    public Lock writeLock() {
        return writerlock;
    }
```

```java
private synchronized void aquireRead(){
    readersWating++;
    while(writers>0 || (writersWating>0 && readersToWait<=0)){
        try {
            wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    readersWating--;
    readersToWait--;
    readers++;
}
private synchronized void releaseRead(){
    readers--;
    notifyAll();
}


private synchronized void aquireWrite(){
    writersWating++;
    while(writers>0 || readers>0 || readersToWait>0){
        try {
            wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    writersWating--;
    writers++;
}
private synchronized void releaseWrite(){
    writers--;
    readersToWait = readersWating;
    notifyAll();
}
```

LockFreeSensors

```
class LockFreeSensors implements Sensors {

    AtomicReference<SensorData> data;

    LockFreeSensors()
    {
        data = new AtomicReference<SensorData>(new SensorData(0L, new double[0]));
    }
```

```
public long get(double val[])
{
    SensorData d = data.get();
    double[] v = d.getValues();
    if (v == null) return 0;
    for (int i=0; i<v.length; ++i)
        val[i] = v[i];
    return d.getTimestamp();
}
```

```
public void update(long timestamp, double[] val)
{
    SensorData old_data;
    SensorData new_data = new SensorData(timestamp, val);
    do {
        old_data = data.get();
        if (old_data != null && old_data.getTimestamp() >= new_data.getTimestamp()) {
            return;
        }
    } while (!data.compareAndSet(old_data, new_data));

}
```

Is the previous implementation wait free?

Yes

Given that the threads update (or try to update) the entry in a given time interval only **finitely** often, the method call is bounded and cannot continuously retry. This is the case since the time stamp of the entry is strictly increasing. If our time stamp is older, we return. If our time stamp is newer, there are only **finitely** many other threads with updates which have a time stamp in between the current time stamp and our time stamp. That is why our method call will always finish in a **bounded** number of steps. Hence, it is wait-free.

Pre-Discussion: Assignment 13

Histories and their properties:
Sequential Consistency
Linearizability
Equivalence
Completeness
etc.

Sequential Consistency

For each of the following histories, indicate if they are sequentially consistent or not. In the following the objects r and s are registers (initially zero), q is a FIFO (initially empty).

```
A:   --|r.write(1)|-------------------------
B:   ------|r.read():0|---------------------
C:   ---------------------|r.read():1|------
```

```
A: q.enq(5)
B: q.enq(3)
A: void
B: void
A: q.deq()
B: q.deq()
A: 3
B: 3
```

```
A:   --|s.write(1)|-------------------------
B:   ------|r.read():0|---------------------
C:   ---------------------|r.read():1|------
```

```
A:   --|s.write(1)|-------------------------
B:   ------|r.read():1|---|r.read():0|------
```

Linearizability

Which of the following histories are linearizable? Infer the object type from the supported operations, registers are initially zero, stacks/queues initially empty.

```
A: s.push(1)
A: void
B: s.push(2)
B: void
B: s.pop()
A: s.pop()
B: 1
A: 2
```

```
A:   --|s.write(1)|-------------------------
B:   ------|r.read():1|---|r.read():0|------
```

Equivalence

Give two different well-formed histories H1 and H2, which are equivalent to each other.

Incomplete Histories

When histories are obtained from a program trace, the history might be incomplete, i.e., if tracing stopped before the program completed. In the lecture you learned that this can be dealt with in two ways. Explain them. Why do we need both ways? Give an example where discarding all pending invocations will lead to a non-linearizable history, but adding a response will lead to a linearizable history.

Deifference between Sequential Consistency and Linearizability

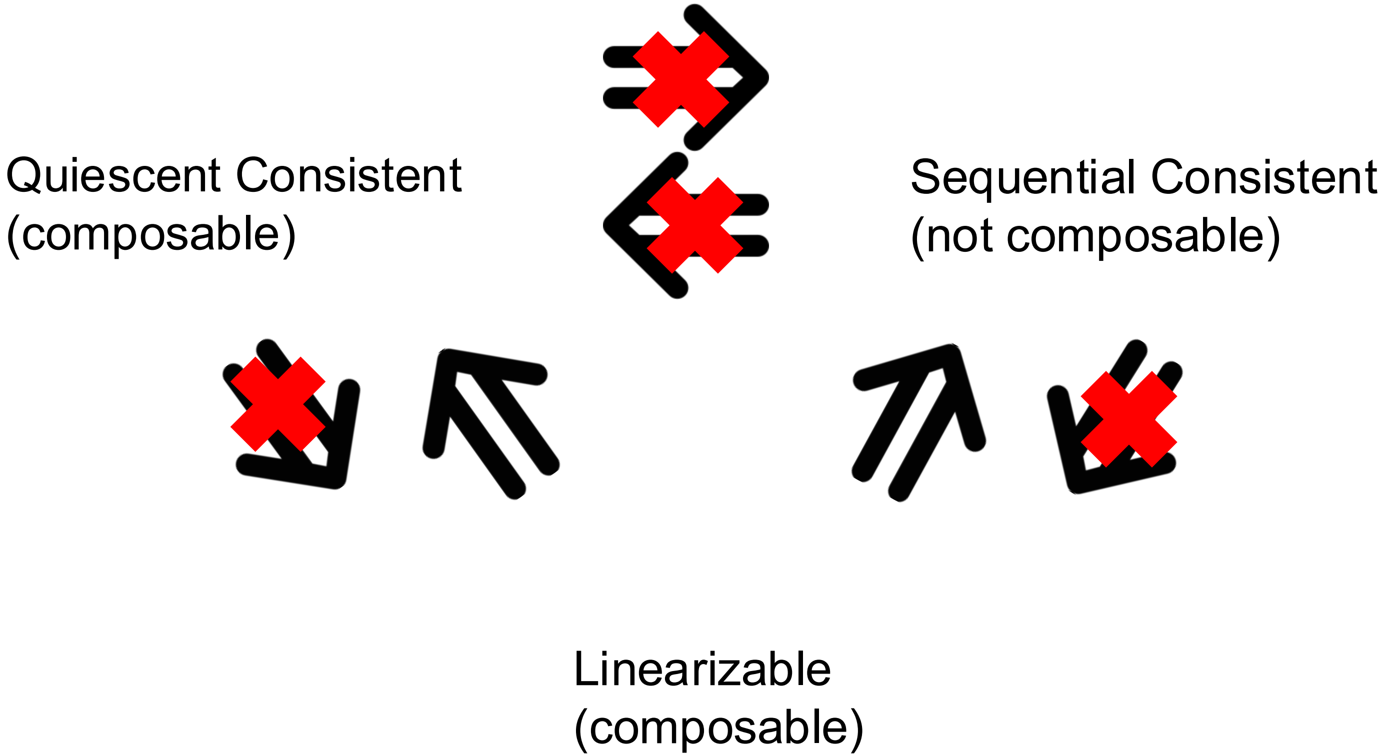
Give a history which is sequentially consistent but not linearizable.

Recap Histories

Histories can be categorized by some fundamental properties:

Sequential: 1st action invocation; no interleavings
Complete: no pending invocations
Equivalence to some other History: for all threads A: H|A = G|A
Legal: for all objects r: H|r is sequential and correct
Well formed: for all threads A: H|A is sequential
Quiescent Consistent: correct with reordering of “overlapping” calls
Sequentially Consistent: correct with reordering regarding threads
Linearizable: choosing linearization points to make execution correct

Note: the above definitions are not formal


Quiescent Consistent
(composable)
Sequential Consistent
(not composable)
Linearizable
(composable)

Exam Questions

Geben Sie eine Definition zu jeder der folgenden Eigenschaften von Locks an:

Give a definition for each of the following lock properties:

Fair

Deadlock Free

Starvation Free

Geben Sie eine Definition zu jeder der folgenden Eigenschaften von Locks an:

Give a definition for each of the following lock properties:

Fair

A lock is fair if it fulfills FIFO order.

Deadlock Free

Starvation Free

Geben Sie eine Definition zu jeder der folgenden Eigenschaften von Locks an:

Give a definition for each of the following lock properties:

Fair

A lock is fair if it fulfills FIFO order.

Deadlock Free

When one or more threads are competing for the lock, at least one of those threads is guaranteed to acquire the lock within a finite amount of time.

Starvation Free

Geben Sie eine Definition zu jeder der folgenden Eigenschaften von Locks an:

Give a definition for each of the following lock properties:

Fair

A lock is fair if it fulfills FIFO order.

Deadlock Free

When one or more threads are competing for the lock, at least one of those threads is guaranteed to acquire the lock within a finite amount of time.

Starvation Free

When one or more threads are competing for the lock, every thread is guaranteed to acquire the lock within a finite amount of time.

Is every lock that is starvation free and deadlock free also fair? Give a proof or counter example.

Is every lock that is starvation free and deadlock free also fair? Give a proof or counter example.

Counter example:

Filter Lock

Is every fair lock also starvation-free and deadlock free? Give a proof or counter example.

Is every fair lock also starvation-free and deadlock free? Give a proof or counter example.

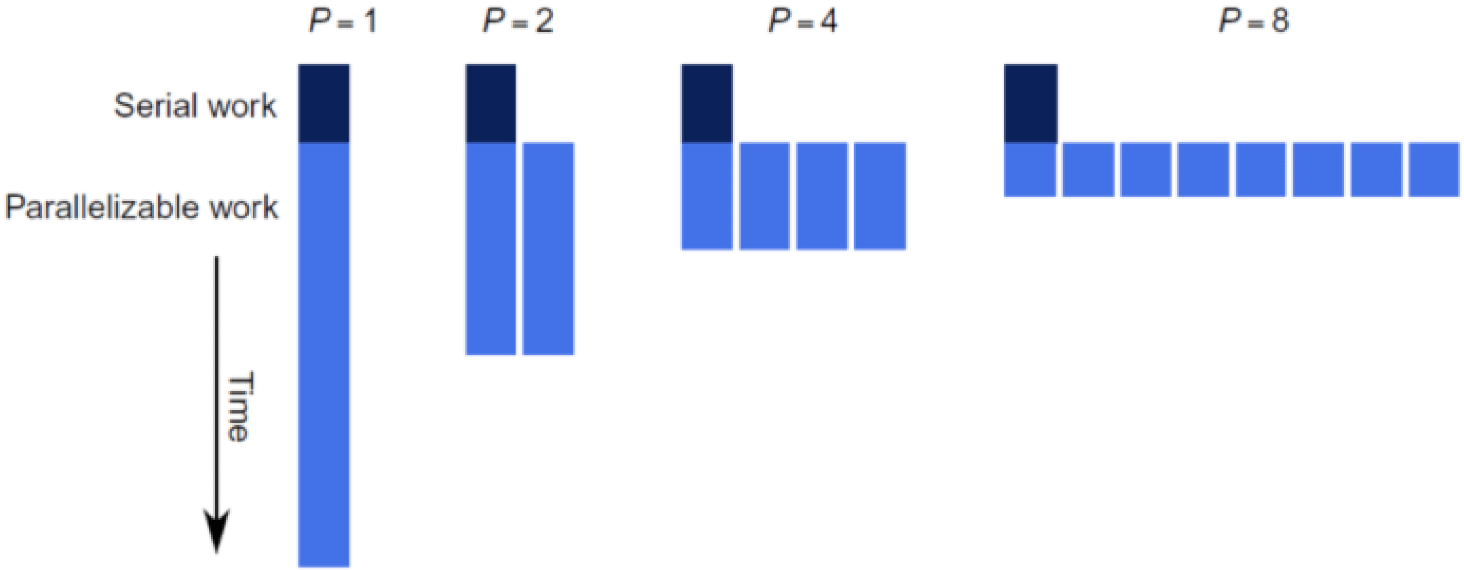
Counter example:

A lock where the lock method never returns.
FIFO not violated but lock is neither starvation-free nor deadlock-free.

	Non-blocking (no locks)		Blocking (locks)
Everyone makes progress	Wait-free	⇍ ⇒	Starvation-free
	⇓ ⇑̸		⇓ ⇑̸
Someone make progress	Lock-free	⇍ ⇒	Deadlock-free

Deadlock-free & fair => Starvation-free

Amdahl’s Law Illustrated



Amdahl's Law – Ingredients

Given P workers available to do parallelizable work, the times for sequential execution and parallel execution are:

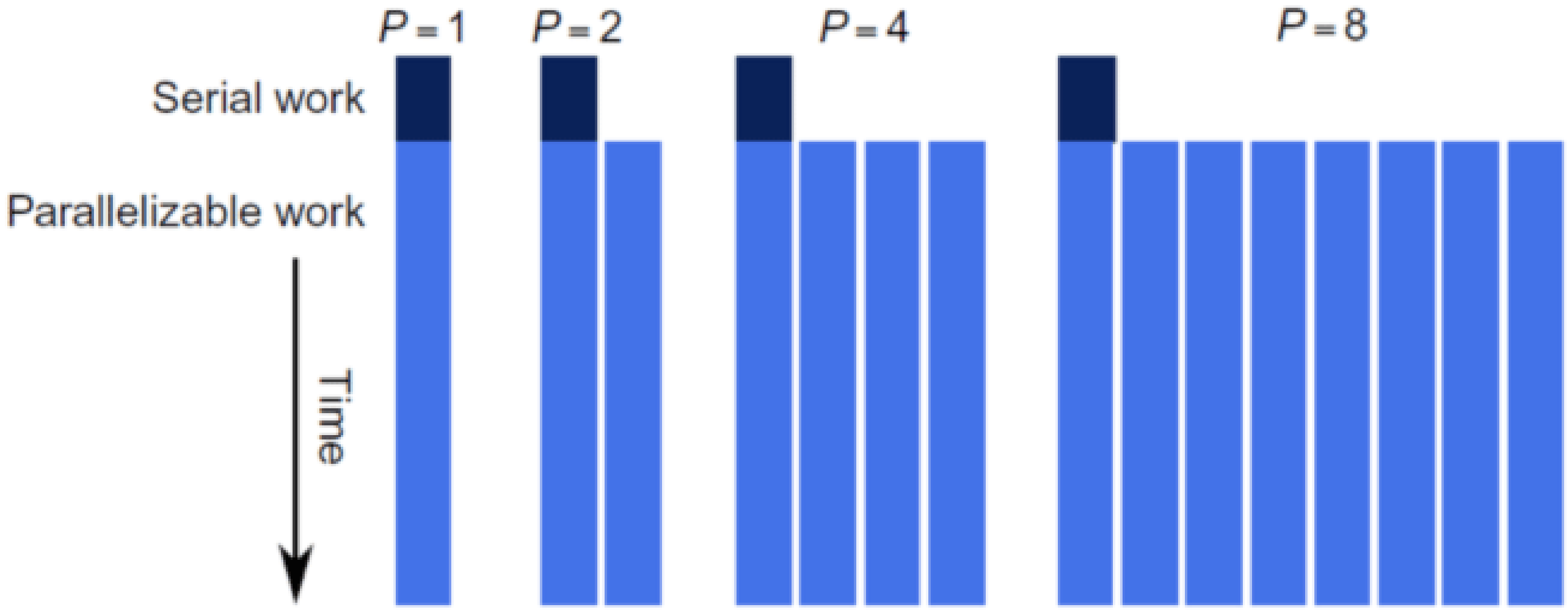
$$T_1 = W_{ser} + W_{par}$$

And this gives a bound on speed-up:

$$T_p \geq W_{ser} + \frac{W_{par}}{P}$$

$$S_\infty \leq \frac{1}{\mathbf{f}} \qquad S_p \leq \frac{W_{ser} + W_{par}}{W_{ser} + \frac{W_{par}}{P}} = \frac{1}{\mathbf{f} + \frac{1-\mathbf{f}}{P}}$$

Gustafson's Law



Gustafson's Law

$$W = \mathbf{f} * W + (1 - \mathbf{f}) * W$$

$$W_P = \mathbf{f} * W + P * (1 - \mathbf{f}) * W$$

$$\begin{aligned} S_P &= \mathbf{f} + P(1 - f) \\ &= P - \mathbf{f}(P - 1) \end{aligned}$$

Consider a program with 20% of the code that is sequential and 80% of the code that is parallelized. Assume that the parallelizable code scales linearly and the sequential runtime of the program is T1 = 100. What is the speedup S8 when executed on 8 CPUs?

Consider a program with 20% of the code that is sequential and 80% of the code that is parallelized. Assume that the parallelizable code scales linearly and the sequential runtime of the program is T1 = 100. What is the speedup S8 when executed on 8 CPUs?

$S_8 = 100/30 = 3.333333\ldots$

A program has 40% sequential code and 60% parallelized code running on a machine with 6 cores. To improve parallel performance, you can either:

A) Hire a developer to parallelize 80% of the code (up from 60%).
B) Buy a better machine with 120 cores. According to Amdahl's law, which of these options leads to better speedup? Briefly explain why.

A program has 40% sequential code and 60% parallelized code running on a machine with 6 cores. To improve parallel performance, you can either:

A) Hire a developer to parallelize 80% of the code (up from 60%).
B) Buy a better machine with 120 cores. According to Amdahl's law, which of these options leads to better speedup? Briefly explain why.

Option A:
We get a speedup of 3

Option B:
We get a speedup < 2.5

Choose option A.

Pipelining: Main Concepts Recap

Latency

time needed to perform a given computation
(e.g., process a customer)

Throughput

amount of work that can be done by a system in a given period of time
(e.g., how many customers can be processed in one minute)

Balanced/Unbalanced Pipeline

a pipeline is balanced if each stage takes the same length of time

Instance vs. Stage

Latency

Generally, you can take the total time of the first instance.
$latency = total\_time(first\_instance) = sum(time(all\_stages))$

If not constant, you can calculate it for the n-th instance.
$$latency = total\_time(first\_instance) + (\max(time\_stage) - time(first\_stage)) \cdot (n-1)$$

Definition 4.2.2. Throughput is the number of elements that exit the pipeline (at full capacity) per a given time unit. Throughput can be calculated as follows for *any* pipeline with one execution unit per stage:

$$Throughput = \frac{1}{max(computationtime(stages))}$$

Definition 4.2.3. Throughput under consideration of lead-in and lead-out time given n elements traverse the pipeline is the average time it takes to output an element. This throughput can be calculated as follows for *any* pipeline with one execution unit per stage:

$$\frac{n}{overall\ time\ for\ n\ elements}$$

$$= \frac{n}{n * max(computationtime(stages)) + sum(computationtime(all\ stages\ except\ longest))}$$

https://polybox.ethz.ch/index.php/s/KwIm2IqvWYP79UB?path=%2FSkript#pdfviewer

Consider a pipeline with three stages with the following execution times:

Stage 1: 50 sec

Stage 2: 25 sec

Stage 3: 25 sec

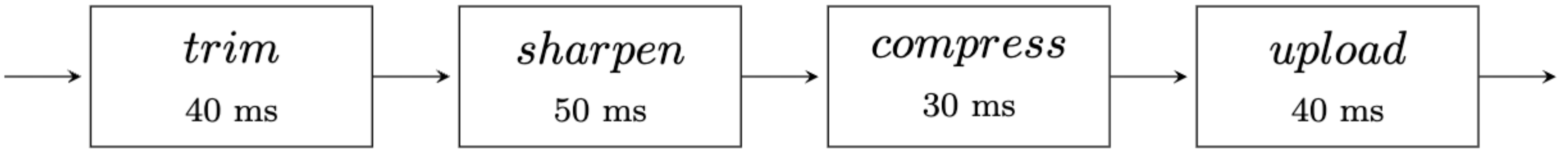
a) What is the throughput of this pipeline?
b) What is the speedup of this pipeline compared to sequential execution?

Consider a pipeline with three stages with the following execution times:
Stage 1: 50 sec
Stage 2: 25 sec
Stage 3: 25 sec

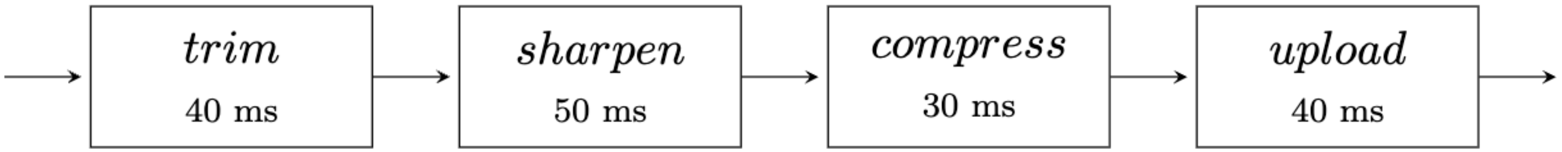
a) What is the throughput of this pipeline?
b) What is the speedup of this pipeline compared to sequential execution?

Throughput
1 / 50 sec

Speedup
S_pipelined = throughput_pipelined / throughput_sequential
= 2

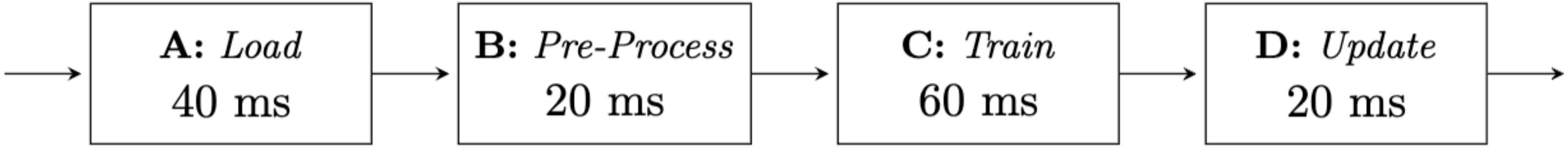


You want to optimize the pipeline by duplicating the execution unit for a stage of your choice (i.e., provide two units that can work in parallel but with the same processing time). What unit would you duplicate? Write down two advantages of the optimized pipeline compared to the original pipeline.

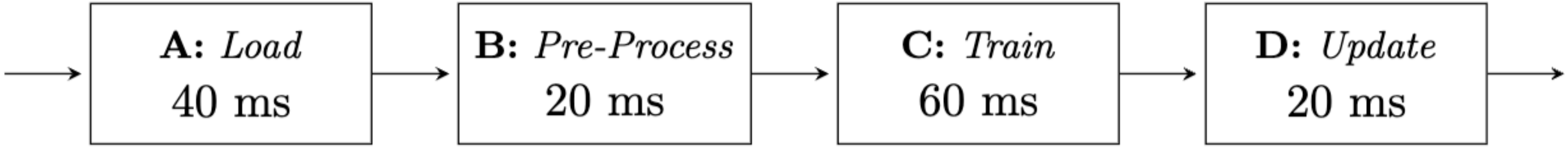


You want to optimize the pipeline by duplicating the execution unit for a stage of your choice (i.e., provide two units that can work in parallel but with the same processing time). What unit would you duplicate? Write down two advantages of the optimized pipeline compared to the original pipeline.

Duplicate sharpen:
Throughput increased
Constant latency

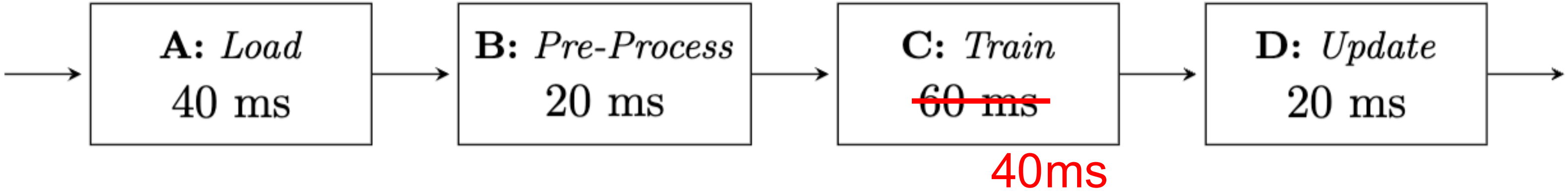


How long should stage ***C*** *take, in order for the pipeline to be balanced?*

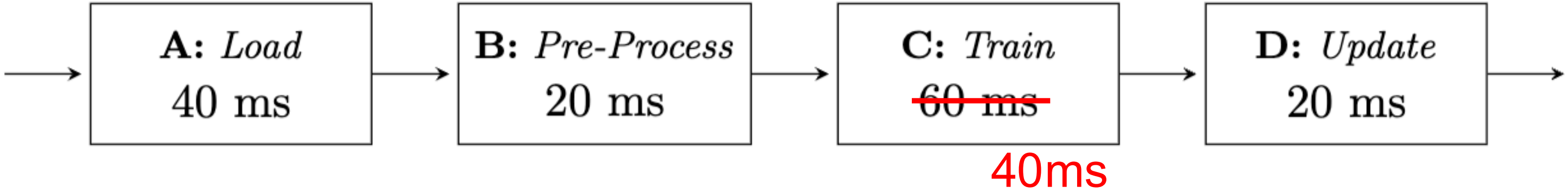


How long should stage ***C*** *take, in order for the pipeline to be balanced?*

40ms



Stage ***C*** *consists of 20% non-parallelizable work. Calculate how many processors are needed to achieve the execution time of stage* ***C*** *that you stated in task c)i), assuming the amount of work stays constant. Use the correct speedup law, and justify your choice in one or two sentences.*



*Stage **C** consists of 20% non-parallelizable work. Calculate how many processors are needed to achieve the execution time of stage **C** that you stated in task c)i), assuming the amount of work stays constant. Use the correct speedup law, and justify your choice in one or two sentences.*

We need 2 processors

Both, ForkJoin and ExecutorService, schedule tasks to threads.

Both, ForkJoin and ExecutorService, schedule tasks to threads.

Both, ForkJoin and ExecutorService, maintain a pool of threads that is reused for multiple tasks.

Both, ForkJoin and ExecutorService, maintain a pool of threads that is reused for multiple tasks.

ExecutorService is better suited than ForkJoin if there are dependencies between tasks.

ExecutorService is better suited than ForkJoin if there are dependencies between tasks.

For each of the following three histories, indicate if it is linearizable and/or sequentially consistent. Assume r is an atomic register which is initialized with 0.

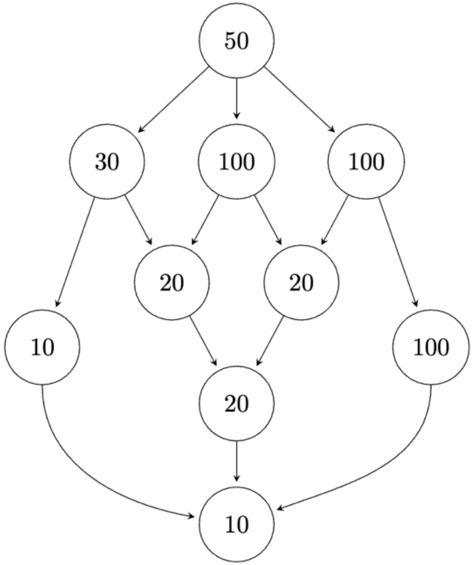
```
H1: A r.write(1)
    B r.write(0)
    B r:void
    A r:void
    A r.read()
    B r.read()
    B r:1
    A r:0
```

For each of the following three histories, indicate if it is linearizable and/or sequentially consistent. Assume r is an atomic register which is initialized with 0.

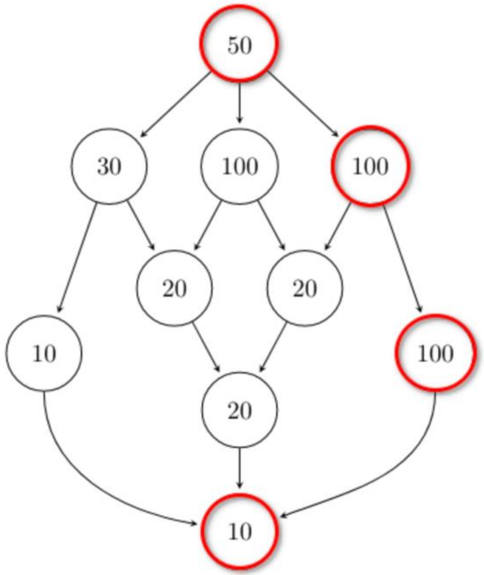
```
H1: A r.write(1)
    B r.write(0)
    B r:void
    A r:void
    A r.read()
    B r.read()
    B r:1
    A r:0
```

Not SC

Thus also not linearizable

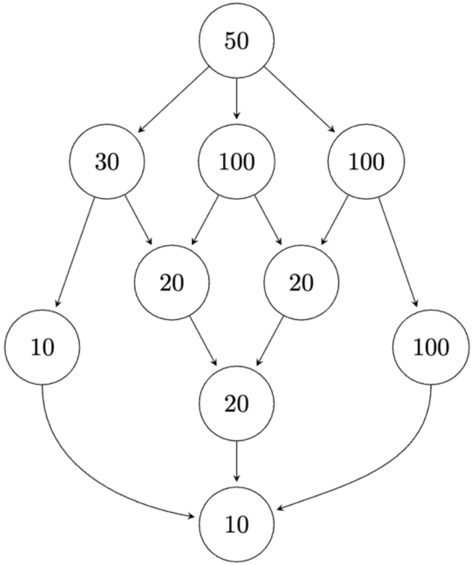


Mark the critical path of the task graph shown in Figure 2. What is the execution time of the critical path?

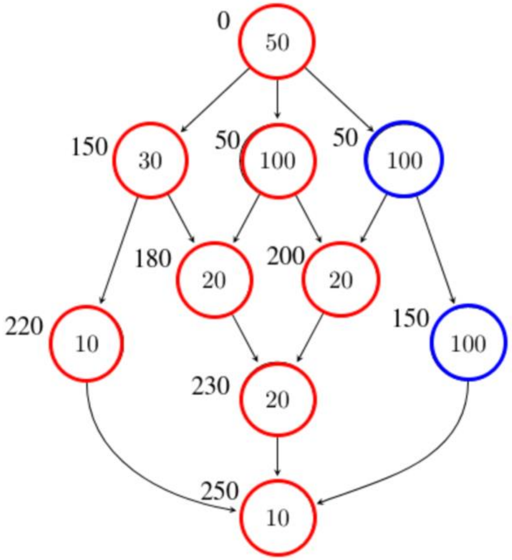


Mark the critical path of the task graph shown in Figure 2. What is the execution time of the critical path?

Execution Time: 260



What is the minimum number of processors necessary to execute the task graph in Figure 2 as quickly as possible?



What is the minimum number of processors necessary to execute the task graph in Figure 2 as quickly as possible?

We need 2 processors

```java
class Position {
private int count = 0;
private int price = 0;
private ReentrantLock lock = new ReentrantLock(true);

public void writePosition(Data data) {
        lock.lock();
        count = data.getCount();
        price = data.getPrice();
        lock.unlock();
}

public Data readPosition() {
        lock.lock();
        Data result = new Data(count, price);
        lock.unlock();
        return result;
}
}
```

- ☐ Der Code ist wait-free. — *The code is wait-free.*
- ☐ Der Code ist starvation-free. — *The code is starvation-free.*
- ☐ Der Code ist lock-free. — *The code is lock-free.*
- ☐ Der Code ist deadlock-free. — *The code is deadlock-free.*

```java
class Position {
private int count = 0;
private int price = 0;
private ReentrantLock lock = new ReentrantLock(true);

public void writePosition(Data data) {
        lock.lock();
        count = data.getCount();
        price = data.getPrice();
        lock.unlock();
}

public Data readPosition() {
        lock.lock();
        Data result = new Data(count, price);
        lock.unlock();
        return result;
}
}
```

- [ ] Der Code ist wait-free. — *The code is wait-free.*
- [x] Der Code ist starvation-free. — *The code is starvation-free.*
- [ ] Der Code ist lock-free. — *The code is lock-free.*
- [x] Der Code ist deadlock-free. — *The code is deadlock-free.*

```java
class Position {
private volatile Data data;

public void writePosition(Data update) {
        data = update;
}

public Data readPosition() {
        return data;
}
}
```

Kreuzen sie alle korrekten Aussagen an

Mark all correct statements.

- ☐ Der Code ist wait-free. — *The code is wait-free.*
- ☐ Der Code ist starvation-free. — *The code is starvation-free.*
- ☐ Der Code ist lock-free. — *The code is lock-free.*
- ☐ Der Code ist deadlock-free. — *The code is deadlock-free.*

```java
class Position {
private volatile Data data;

public void writePosition(Data update) {
        data = update;
}

public Data readPosition() {
        return data;
}
}
```

Kreuzen sie alle korrekten Aussagen an

- [x] Der Code ist wait-free.
- [x] Der Code ist starvation-free.
- [x] Der Code ist lock-free.
- [x] Der Code ist deadlock-free.

Mark all correct statements.

- *The code is wait-free.*
- *The code is starvation-free.*
- *The code is lock-free.*
- *The code is deadlock-free.*

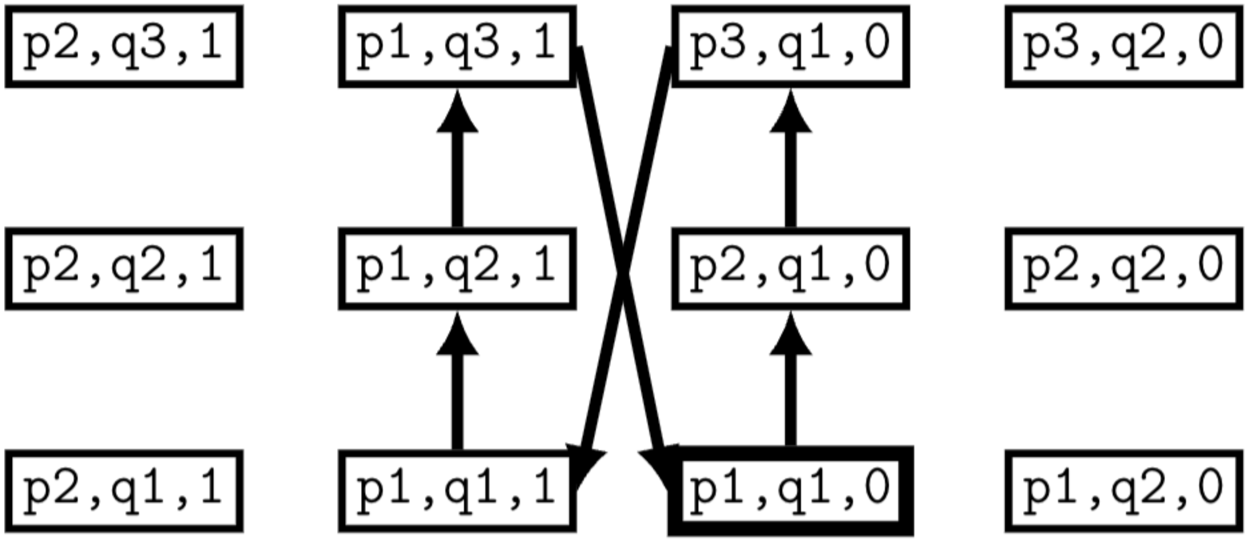
Kreuzen sie alle korrekten Aussagen an. *Mark all correct statements.*

- ☐ Das Peterson Lock ist frei von Starvation. *The Peterson Lock is starvation free.*
- ☐ Das Filter Lock ist fair. *The Filter Lock is fair.*
- ☐ Das Bakery Lock unterstützt mehr als zwei Threads. *The Bakery Lock supports more than two threads.*
- ☐ Das Peterson Lock erweitert das Filter Lock mit Unterstützung für mehr als zwei Threads. *The Peterson Lock extends the Filter Lock to support more than two threads.*

Kreuzen sie alle korrekten Aussagen an. *Mark all correct statements.*

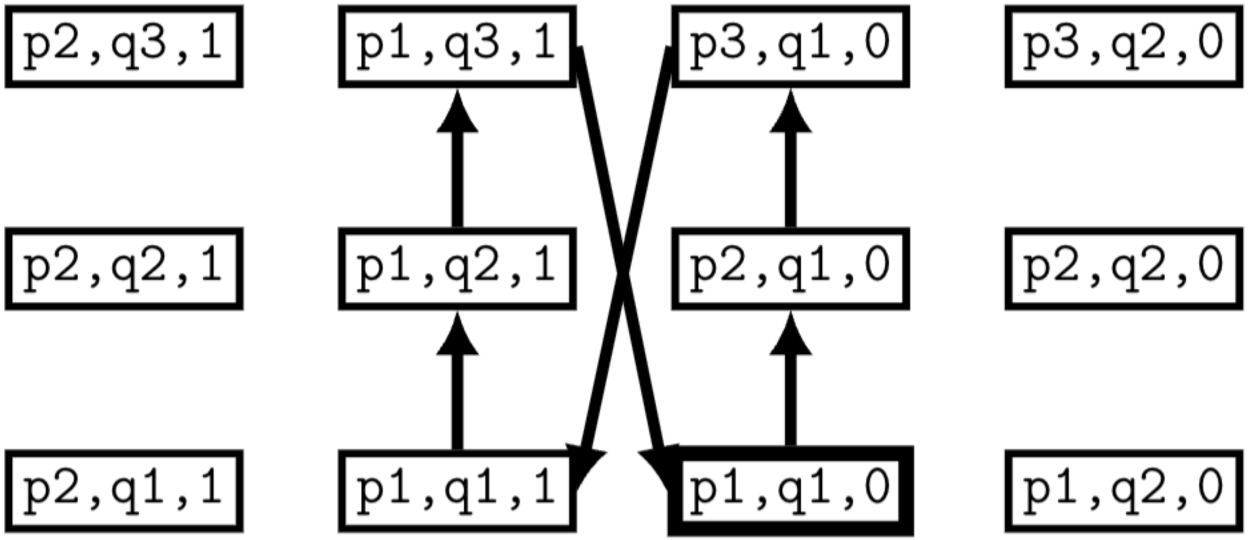
- ☒ Das Peterson Lock ist frei von Starvation. *The Peterson Lock is starvation free.*
- ☐ Das Filter Lock ist fair. *The Filter Lock is fair.*
- ☒ Das Bakery Lock unterstützt mehr als zwei Threads. *The Bakery Lock supports more than two threads.*
- ☐ Das Peterson Lock erweitert das Filter Lock mit Unterstützung für mehr als zwei Threads. *The Peterson Lock extends the Filter Lock to support more than two threads.*

Critical Section: p2 and q2



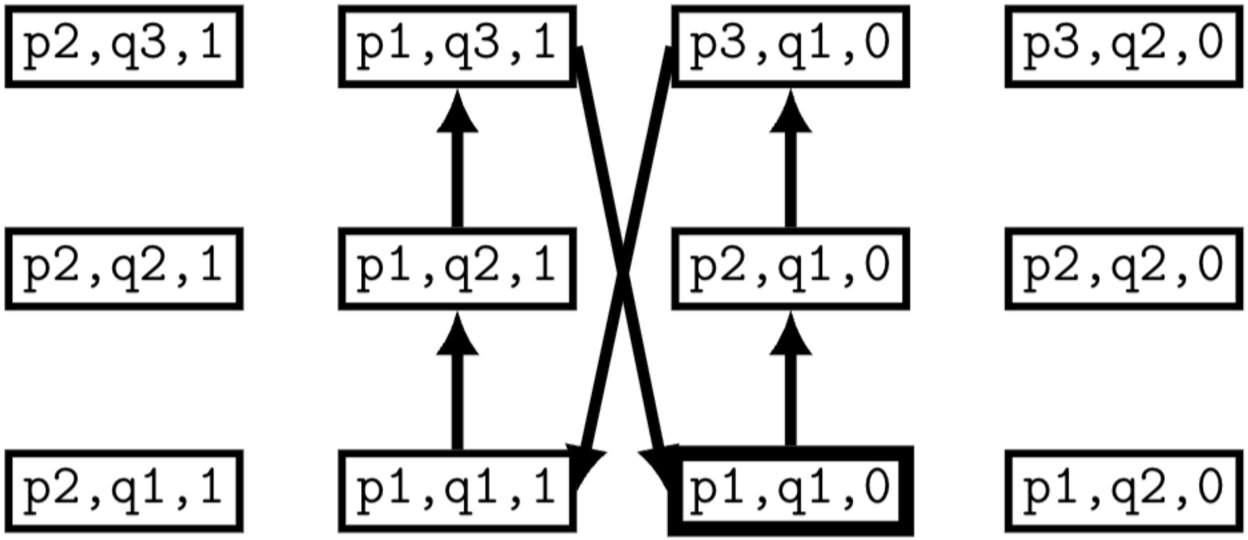
This program can deadlock. ◯ True ◯ False

Critical Section: p2 and q2



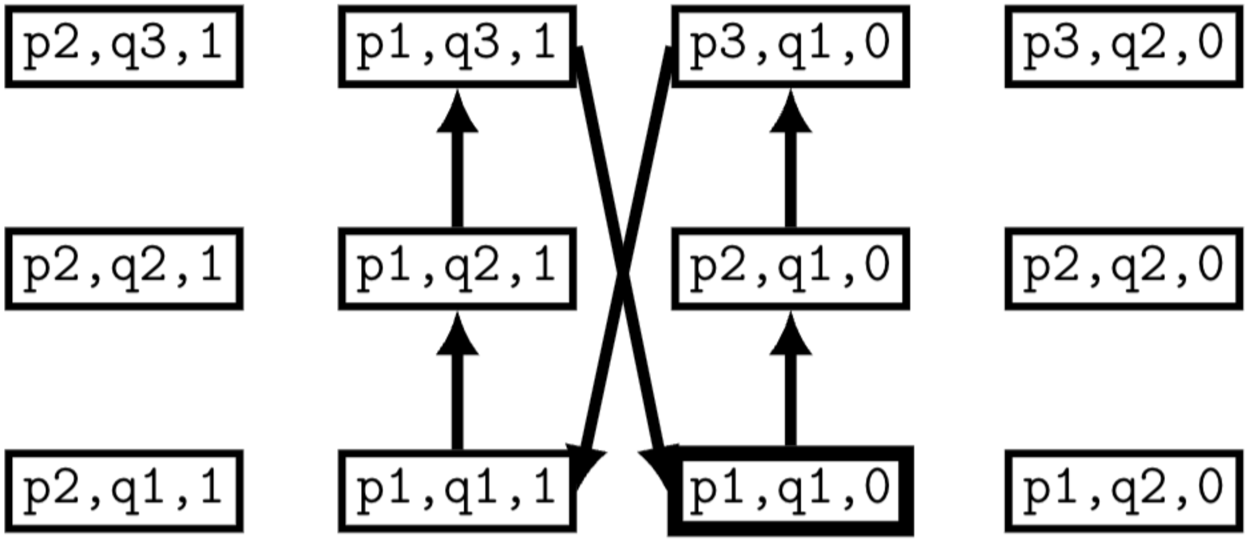
This program can deadlock. ○ True ☒ False

Critical Section: p2 and q2



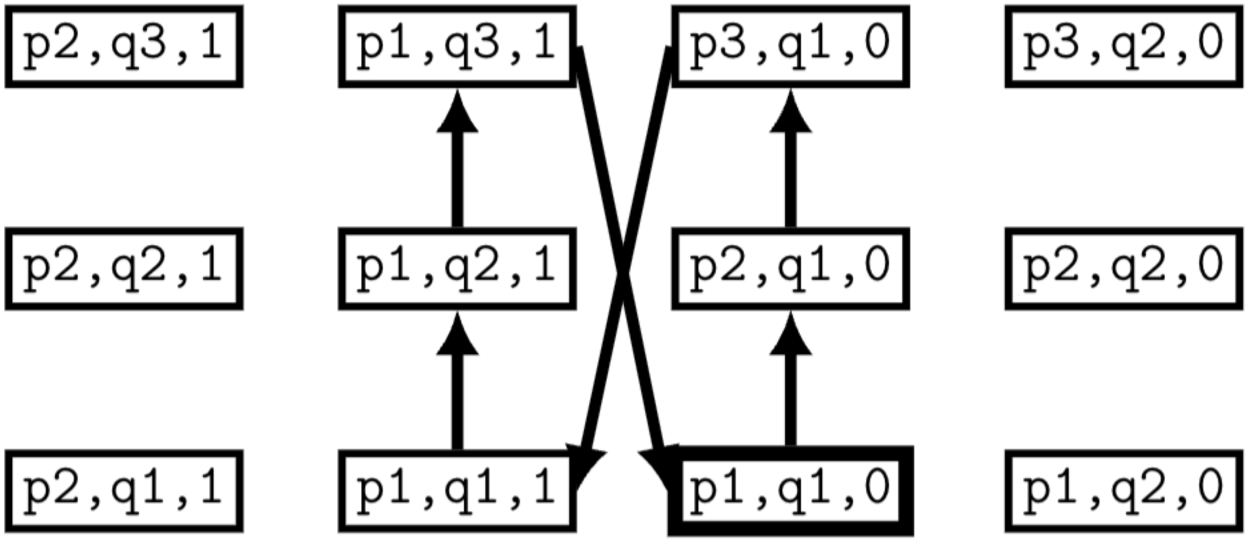
This program can livelock. ◯ True ◯ False

Critical Section: p2 and q2



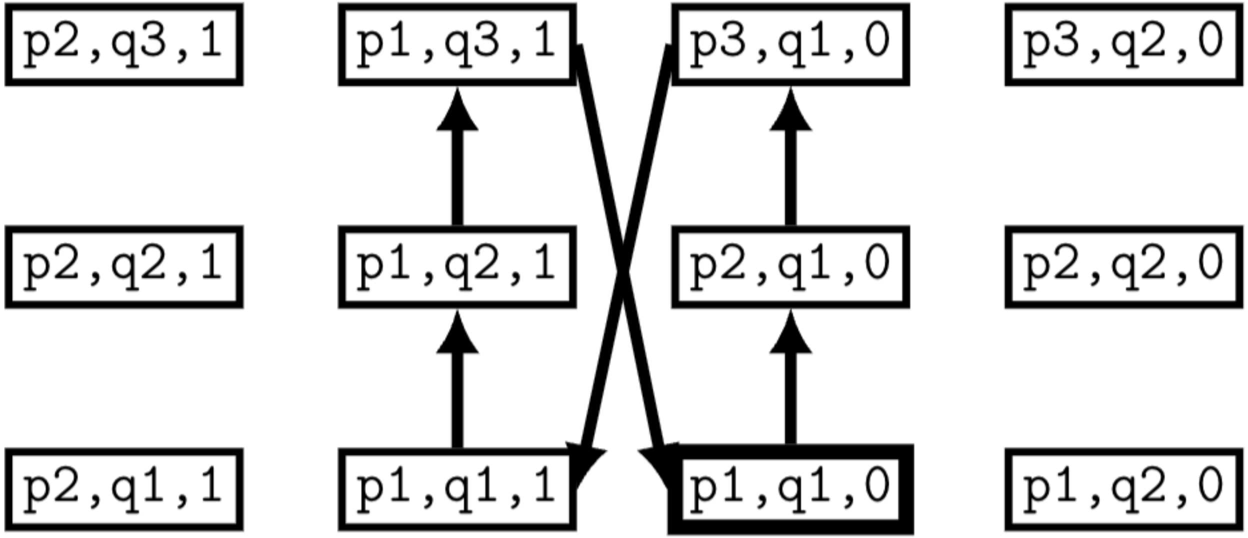
This program can livelock. ○ True ☒ False

Critical Section: p2 and q2



This program provides mutual exclusion. ◯ True ◯ False

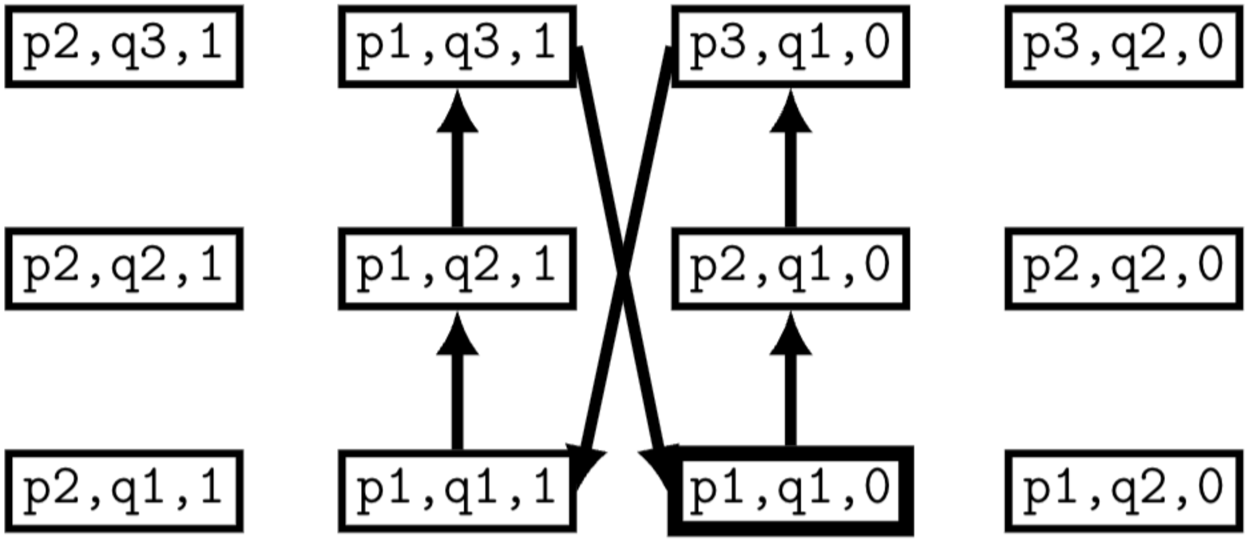
Critical Section: p2 and q2



This program provides mutual exclusion.

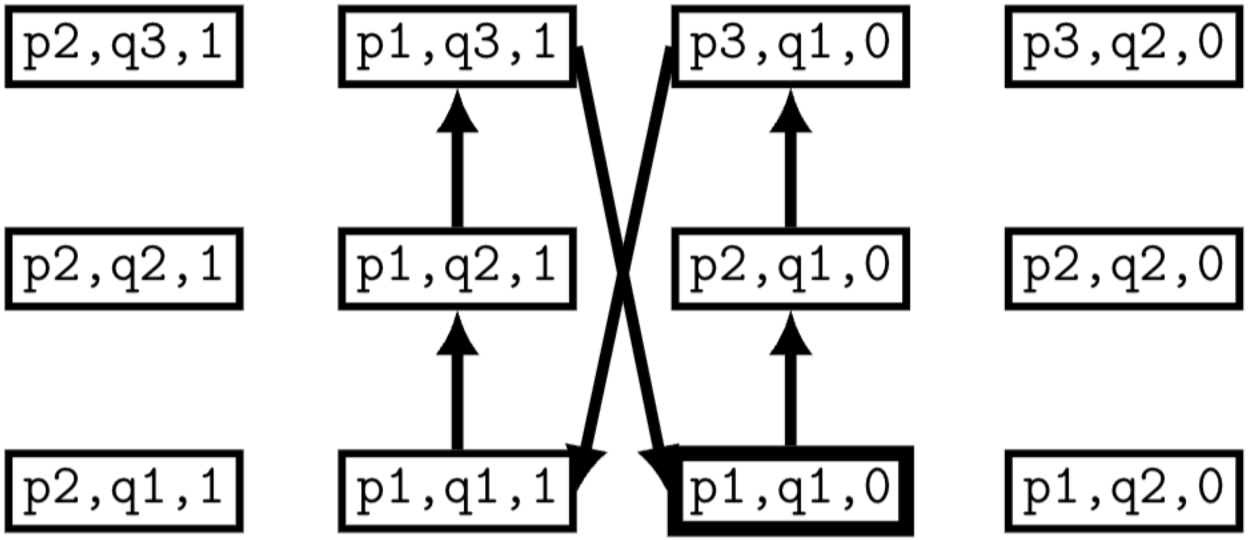
☒ True ○ False

Critical Section: p2 and q2



This program is wait-free (assume critical section is an atomic instruction). ○ True ○ False

Critical Section: p2 and q2



This program is wait-free (assume critical section is an atomic instruction). ○ True ☒ False

Assume p and q both execute a code which is 100 instructions long. The variable n is a 32-bit integer. If we use the notation introduced above, what is the maximum number of states in the state diagram?

Assume p and q both execute a code which is 100 instructions long. The variable n is a 32-bit integer. If we use the notation introduced above, what is the maximum number of states in the state diagram?

100*100*2^32

Parallel Patterns

- We are now quite familiar with how to parallize algorithms
- There are a few recurring patterns that are important to know

Map, Reduction, Stencil, Scan, Pack

Reduction

- A reduction is an operation that produces a single answer from a collection (array etc) via an **associative** operator.
- Needs to be associative. Otherwise divide-and-conquer won‘t work

Example: array sum

Map

- Operates on each element of the input data indenpendently (each array element)
- Output is the same size → no size reduction
- Doesn‘t have to be the same operation on each element

Example: add two arrays

Stencil

- Like map but can take more than one element as input
- Generalization of map and thus also no size reduction

Example:

Image → apply averaging filter on each pixel

Update a value based on its neighbors

Never do it in-place because you would then take values that are already output values.

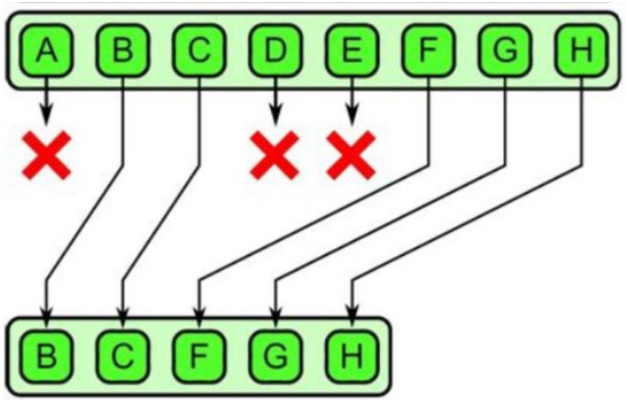
Scan

- Collection of data X → return collection of data Y
- Y(i) = functionOf(Y(i - 1) & X(i))
- Seems sequential because of dependencies
- Can parallelize if function is associative → O(log(n)) span

Example: parallel prefix sum

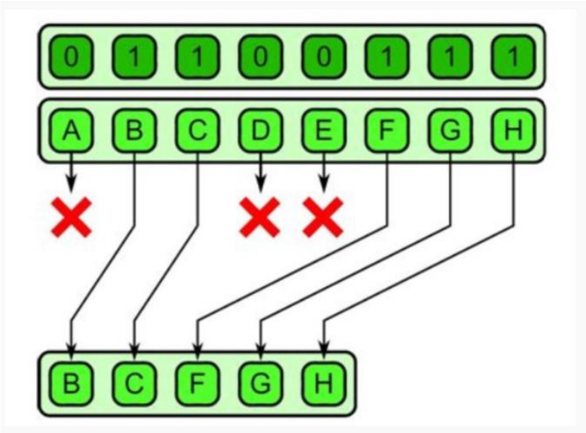
Pack

- Collection of data X → return collection of data X if fulfill condition



Pack

- First compute bit vector
- Then find index in result array (prefix sum on bit vector)



For each of the code snippets below, state whether the operation is a Map, Reduce, Prefix, or Pack and calculate the output of the function.

```
double method_a(){
    double[] vec1 = {10.0, 0.0, 2.0};
    double[] vec2 = {4.0, 4.0, 1.0};
    double sum = 0.0;
    for(int i = 0; i < vec1.length; i++){
        sum += vec1[i] * vec2[i];
    }
    return sum;
}
```

For each of the code snippets below, state whether the operation is a Map, Reduce, Prefix, or Pack and calculate the output of the function.

```
double method_a(){
    double[] vec1 = {10.0, 0.0, 2.0};
    double[] vec2 = {4.0, 4.0, 1.0};
    double sum = 0.0;
    for(int i = 0; i < vec1.length; i++){
        sum += vec1[i] * vec2[i];
    }
    return sum;
}
```

Reduce, sum = 42

```
String[] method_b(){
    String[] vec = {"Apple", "Bean", "Banana", "Pear"};
    boolean[] keepElem = new bool[vec.length];
    int[] numKept = new int[vec.length + 1];
    numKept[0] = 0;

    for(int i = 0; i < vec.length; i++){
        if(vec[i].length() > 4){
            keepElem[i] = true;
            numKept[i+1] = numKept[i] + 1;
        } else {
            keepElem[i] = false;
            numKept[i+1] = numKept[i];
        }
    }

    String[] out = new String[numKept[numKept.length-1]];
    int j = 0;

    for(int i = 0; i < vec.length; i++){
        if(keepElem[i] == true){
            out[j] = vec[i];
            j++;
        }
    }

    return out;
}
```

```
String[] method_b(){
    String[] vec = {"Apple", "Bean", "Banana", "Pear"};
    boolean[] keepElem = new bool[vec.length];
    int[] numKept = new int[vec.length + 1];
    numKept[0] = 0;

    for(int i = 0; i < vec.length; i++){
        if(vec[i].length() > 4){
            keepElem[i] = true;
            numKept[i+1] = numKept[i] + 1;
        } else {
            keepElem[i] = false;
            numKept[i+1] = numKept[i];
        }
    }

    String[] out = new String[numKept[numKept.length-1]];
    int j = 0;

    for(int i = 0; i < vec.length; i++){
        if(keepElem[i] == true){
            out[j] = vec[i];
            j++;
        }
    }

    return out;
}
```

Pack

out = {„Apple“, „Banana“}

```java
int[] method_c(){
    int[] vec = {0, 3, -3, 0, 1};

    for(int i = 0; i < vec.length; i++){
        if(vec[i] < 0){
            vec[i] = -vec[i];
        }
    }
    return vec;
}
```

```java
int[] method_c(){
    int[] vec = {0, 3, -3, 0, 1};

    for(int i = 0; i < vec.length; i++){
        if(vec[i] < 0){
            vec[i] = -vec[i];
        }
    }
    return vec;
}
```

Map, vec = {0, 3, 3, 0, 1}

```java
int[] method_d(){
    int[] vec = {1, 2, 3, 4, 5};
    int[] out = new int[vec.length];
    out[0] = vec[0];

    for(int i = 1; i < vec.length; i++){
        out[i] = vec[i] * out[i-1];
    }
    return out;
}
```

```java
int[] method_d(){
    int[] vec = {1, 2, 3, 4, 5};
    int[] out = new int[vec.length];
    out[0] = vec[0];

    for(int i = 1; i < vec.length; i++){
        out[i] = vec[i] * out[i-1];
    }
    return out;
}
```

Scan, out = {1, 2, 6, 24, 120}

Kahoot