COMP 322: Fundamentals of Parallel Programming

Lecture 22: Actors (continued)

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Worksheet #21: Interaction between finish and actors

What output will be printed if the end-finish operation from slide 15 is moved from line 13 to line 11 as shown below?

```
1. finish(() -> {
2.    int threads = 4;
3.    int numberOfHops = 10;
4.    ThreadRingActor[] ring = new ThreadRingActor[threads];
5.    for(int i=threads-1;i>=0; i--) {
6.      ring[i] = new ThreadRingActor(i);
7.      ring[i].start(); // like an async
8.      if (i < threads - 1) {
9.        ring[i].nextActor(ring[i + 1]);
10.     } }
11. }); // finish
12.ring[threads-1].nextActor(ring[0]);
13.ring[0].send(numberOfHops);
14.
```

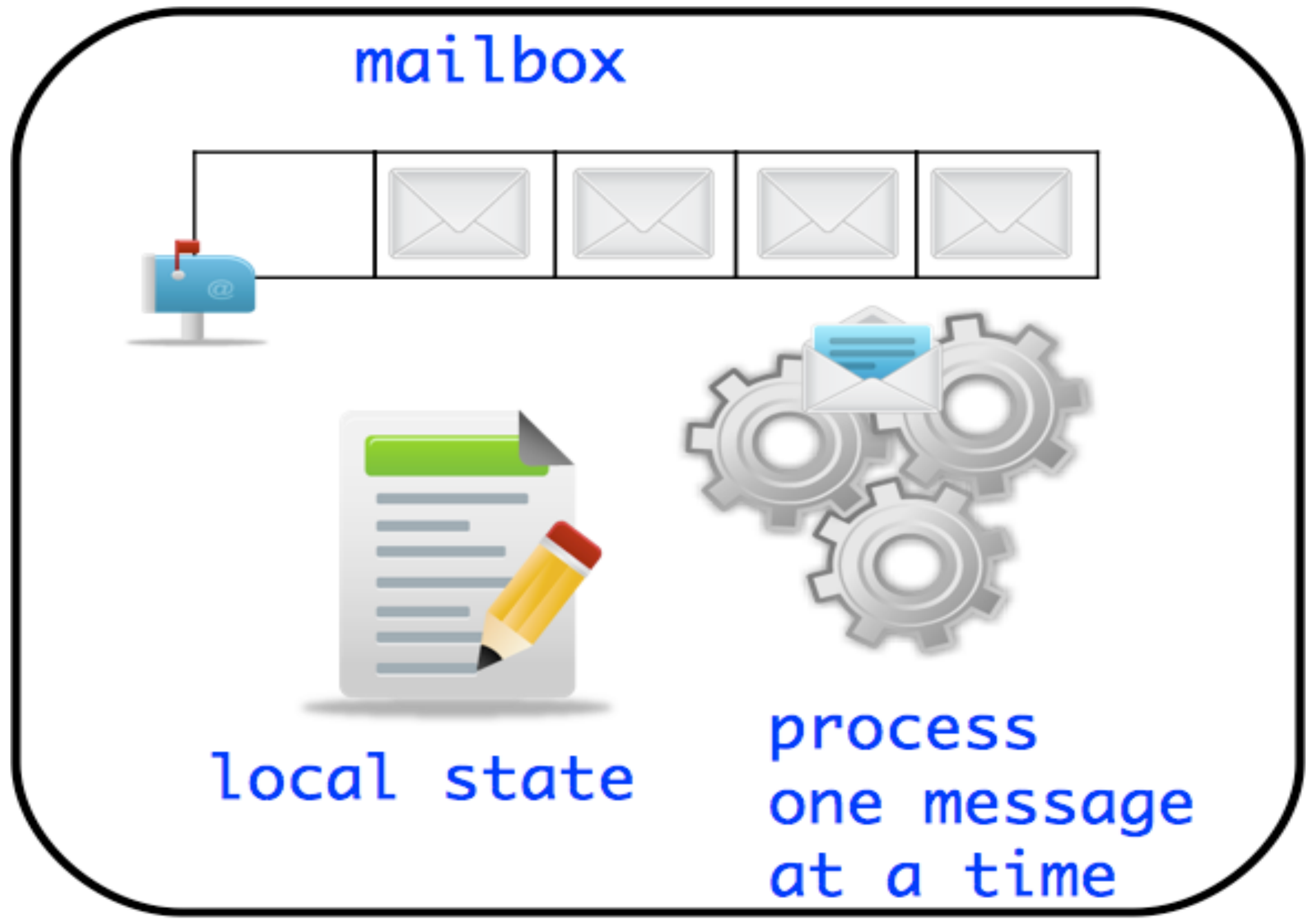
Deadlock (no output): the end-finish operation in line 11 waits for all the actors started in line 7 to terminate, but the actors are waiting for the message sequence initiated in line 13 before they call exit().

Announcements

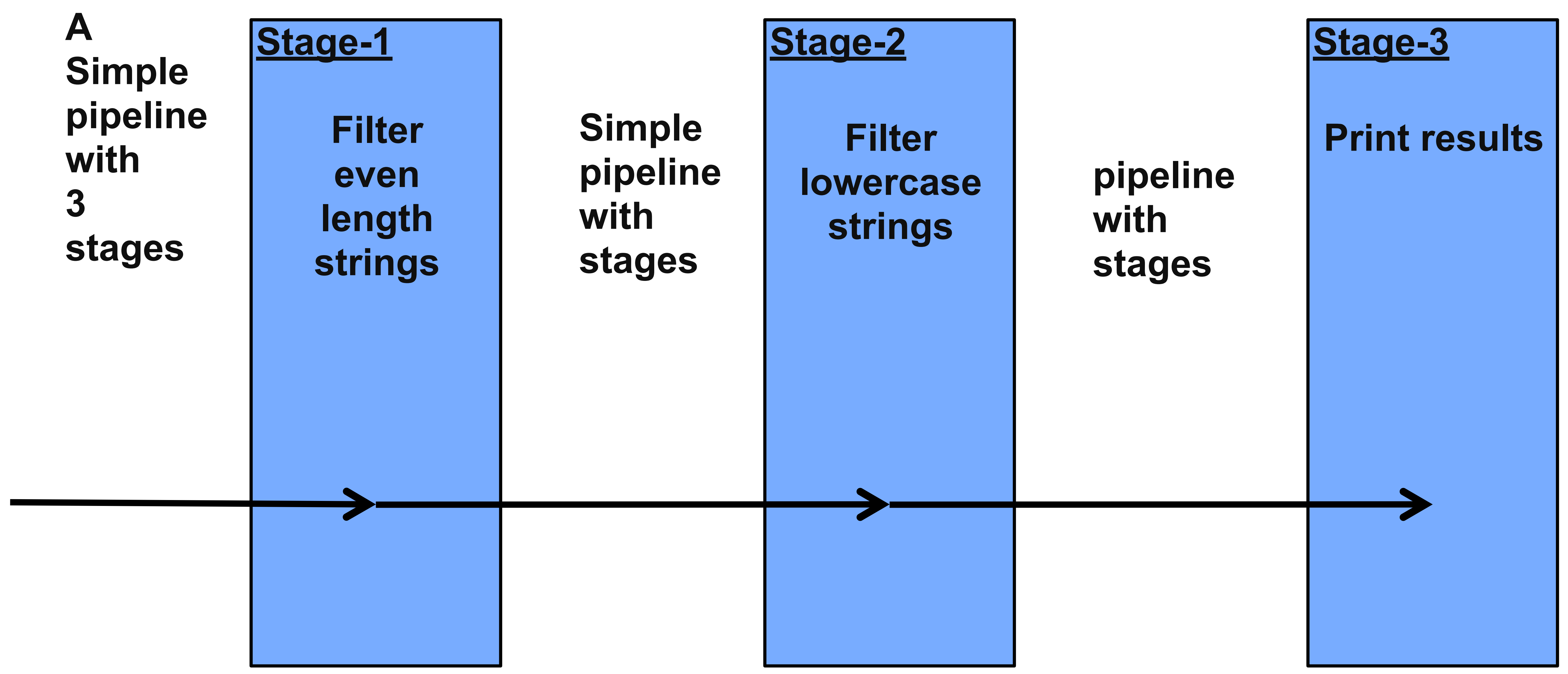
- HW #3 Checkpoint 1 is due today by 11:59pm
- The entire written + programming homework (Checkpoint #2) is due by Monday, April 5th
- Quiz for Unit 5 is due Monday, March 29th at 11:59pm

Recap of Actors

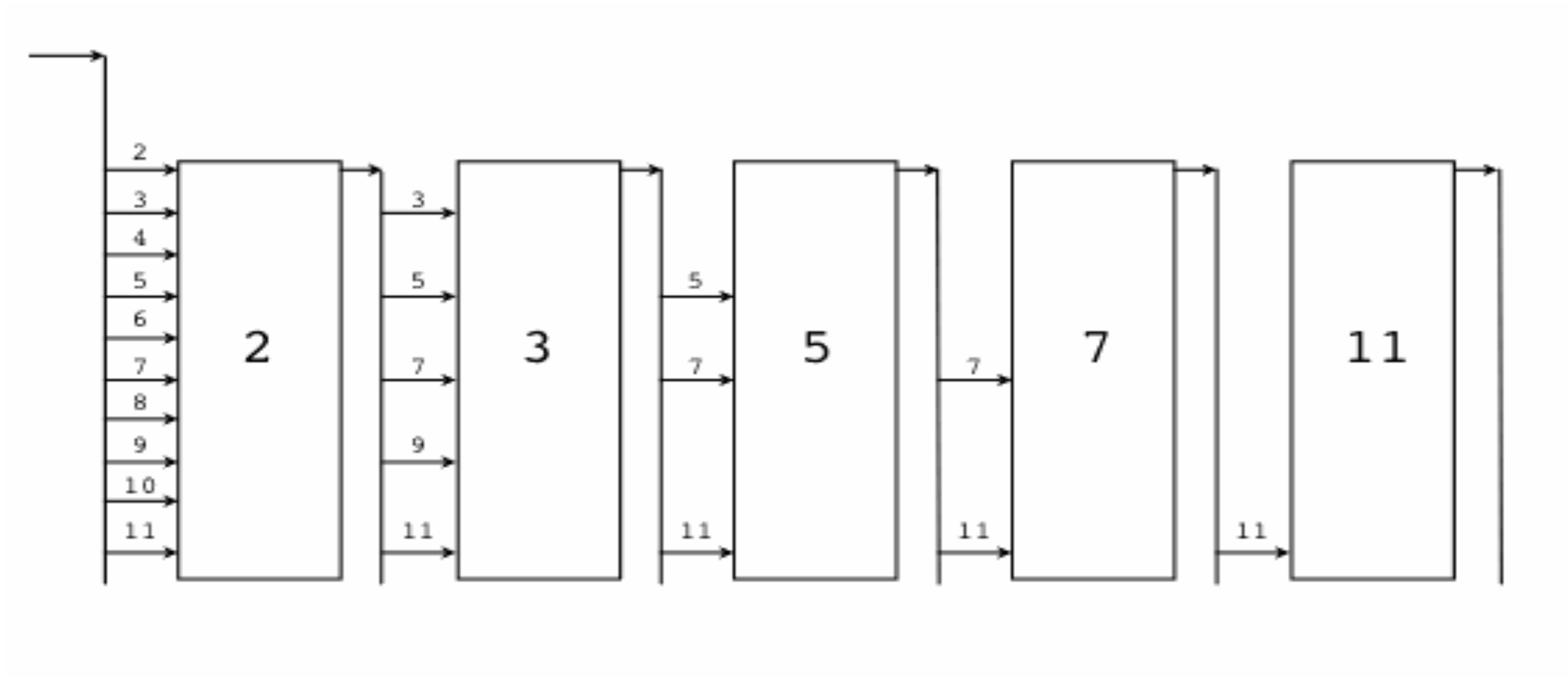
- Rely on asynchronous messaging
- Message are sent to an actor using its `send()` method
- Messages queue up in the mailbox
- Messages are processed by an actor after it is started
- Messages are processed asynchronously
  - —one at a time
  - —using the body of `process()`



Simple Pipeline using Actors


A Simple pipeline with 3 stages
Stage-1
Filter even length strings
Simple pipeline with stages
Stage-2
Filter lowercase strings
pipeline with stages
Stage-3
Print results

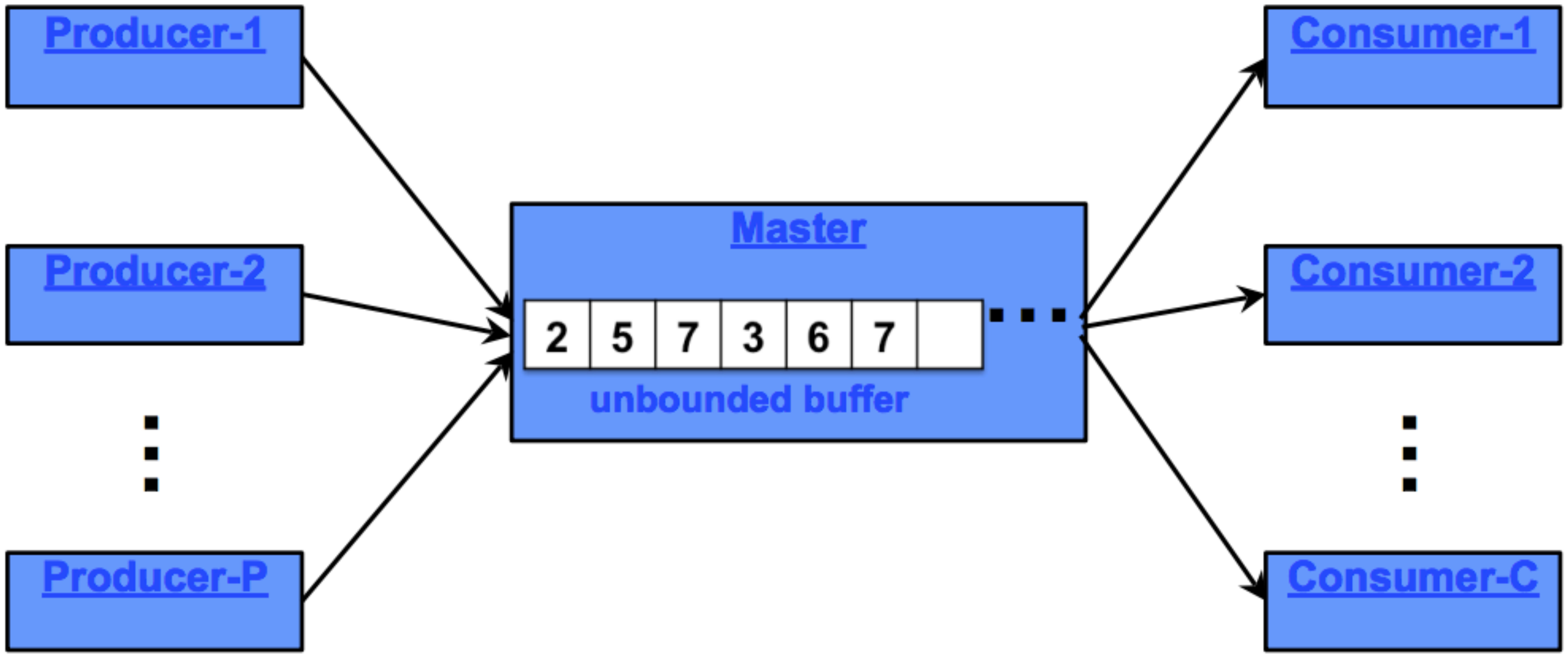
Sieve of Eratosthenes using Actors



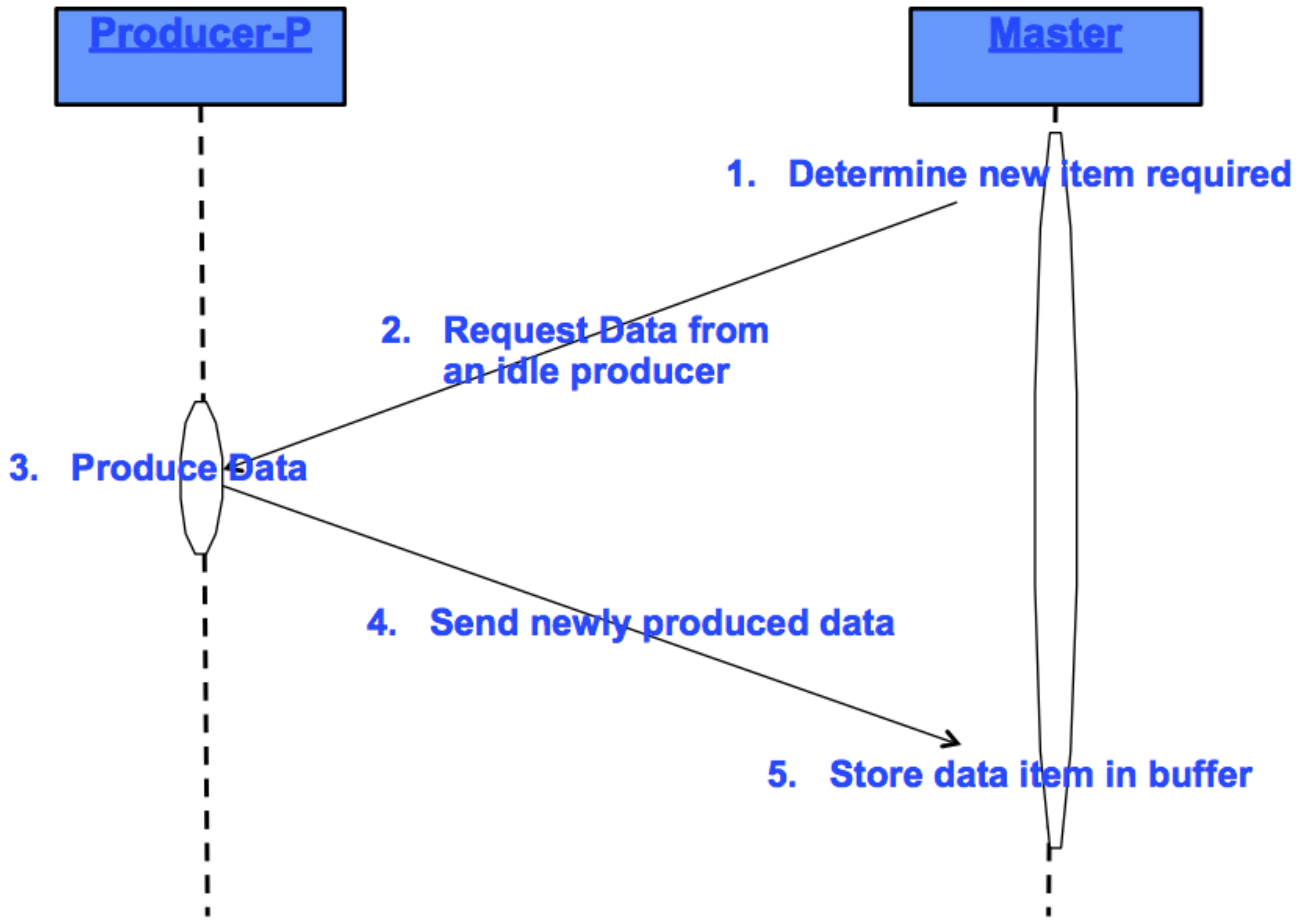
Limitations of Actor Model

- Deadlocks possible
  - —Occurs when all started (but non-terminated) actors have empty mailboxes
- Data races possible when messages include shared objects
- Simulating synchronous replies requires some effort
  - —e.g., does not support addAndGet()
- Implementing truly concurrent data structures is hard
  - —No parallel reads, no reductions/accumulators
- Difficult to achieve global consensus
  - —Finish and barriers not supported as first-class primitive

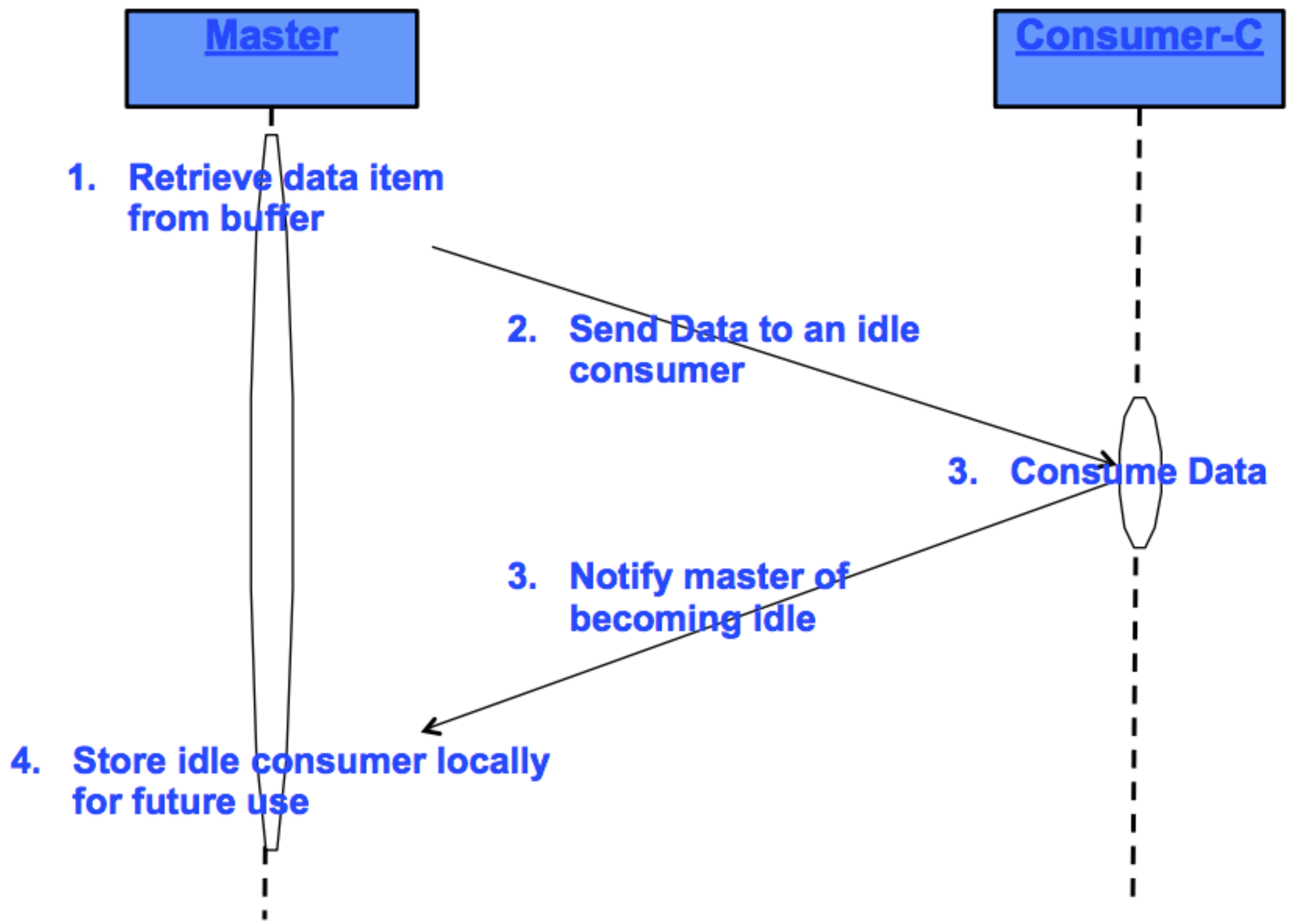
Implementing an Unbounded Buffer using Actors



Unbounded Buffer Actor Interaction Diagram


Producer-P
Master
1. Determine new item required
2. Request Data from an idle producer
3. Produce Data
4. Send newly produced data
5. Store data item in buffer

Unbounded Buffer Actor Interaction Diagram (cont.)


Master
Consumer-C
1. Retrieve data item from buffer
2. Send Data to an idle consumer
3. Consume Data
3. Notify master of becoming idle
4. Store idle consumer locally for future use

Poll: Is Main Actor needed for Producer-Consumer model?

Under which of the following scenarios is a main actor needed to model producer-consumer relationship with an unbounded buffer? Assume Producer(s) have access to Consumer list and Consumer(s) have access to Producer list.

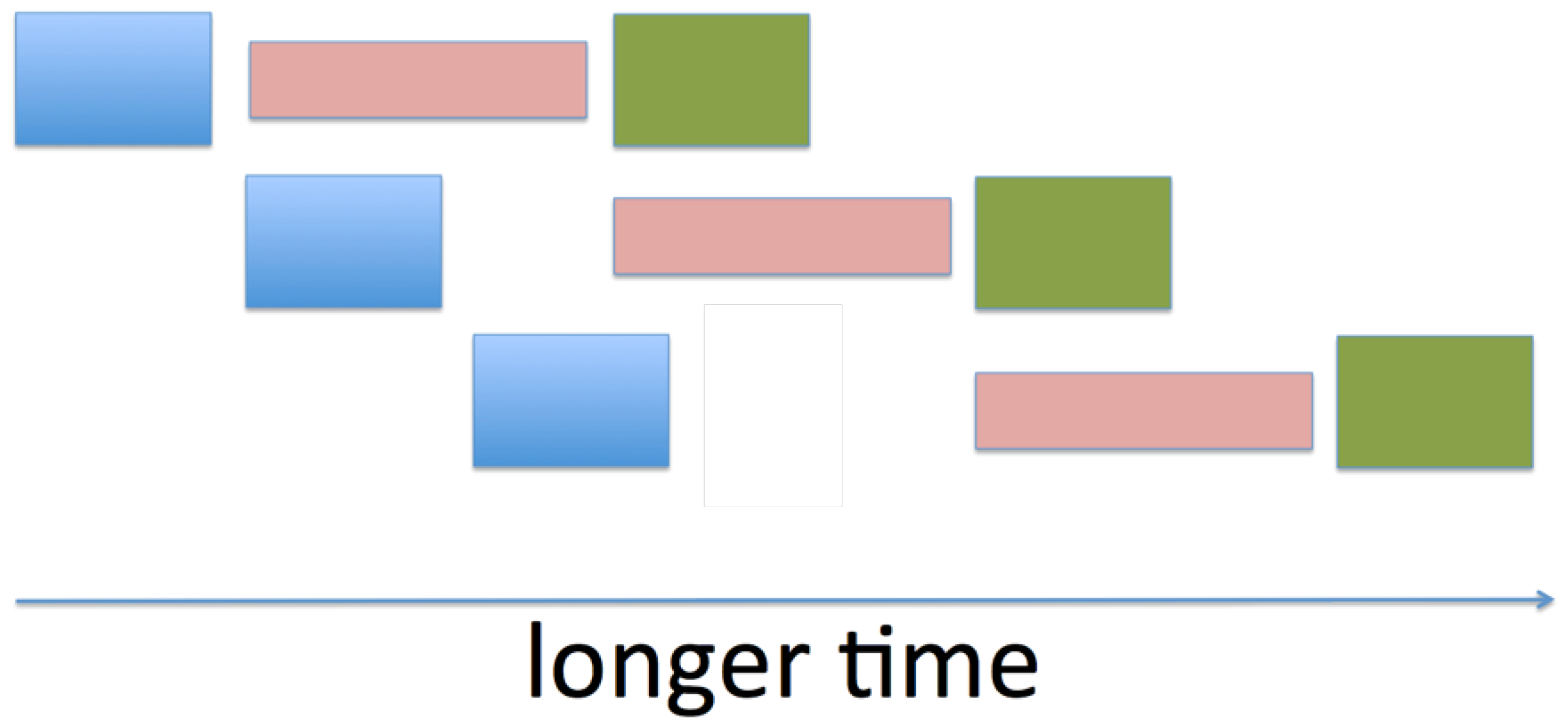
- 1 producer, 1 consumer
- 1 producer, many consumers
- Many producers, 1 consumer
- Many producers, many consumers

Under which of those scenarios is having a main actor more efficient?

Pipeline and Actors

Pipelined Parallelism:

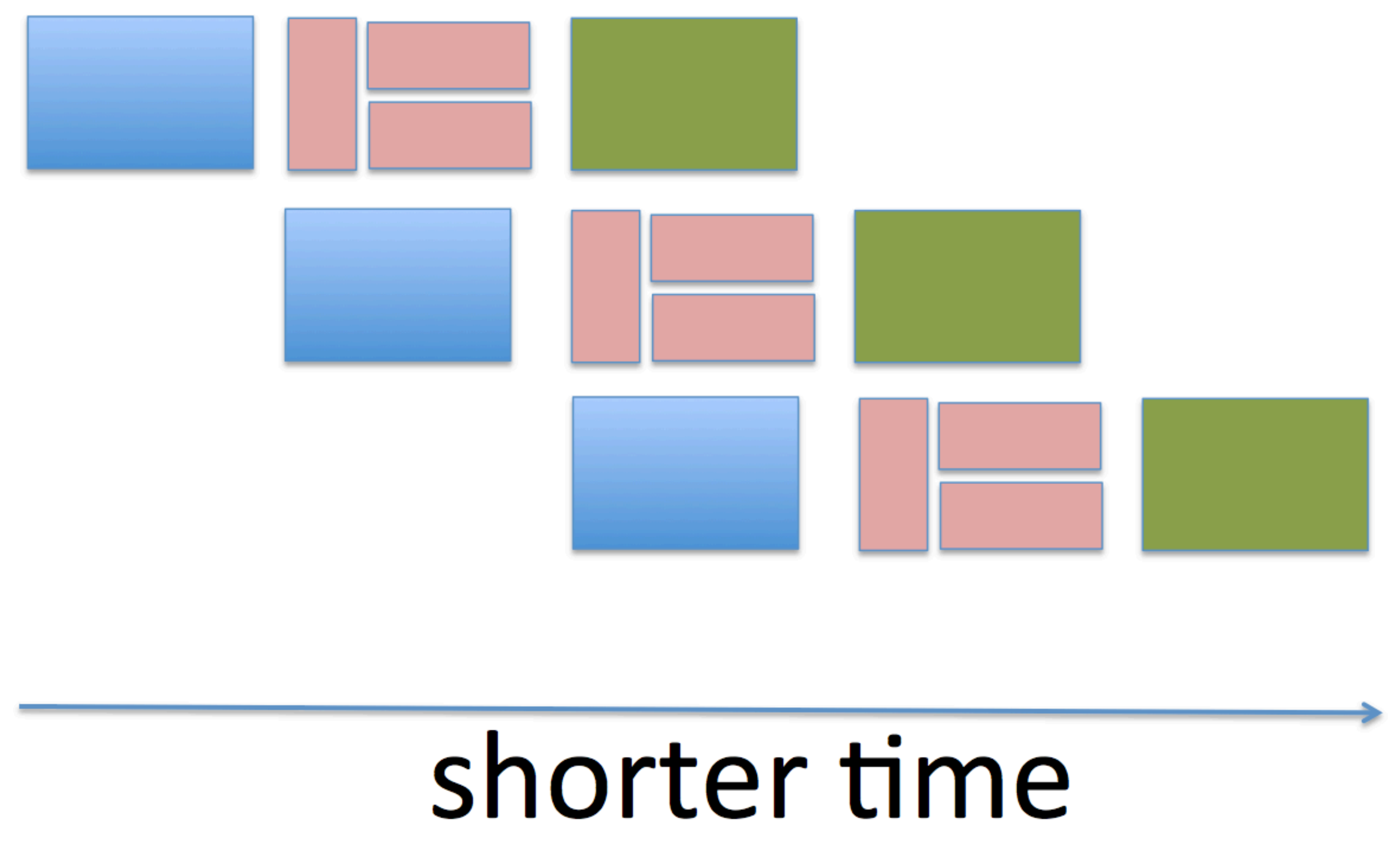
- Each stage can be represented as an actor
- Stages need to ensure ordering of messages while processing them
- Slowest stage is a throughput bottleneck



Motivation for Parallelizing Actors

Pipelined Parallelism:

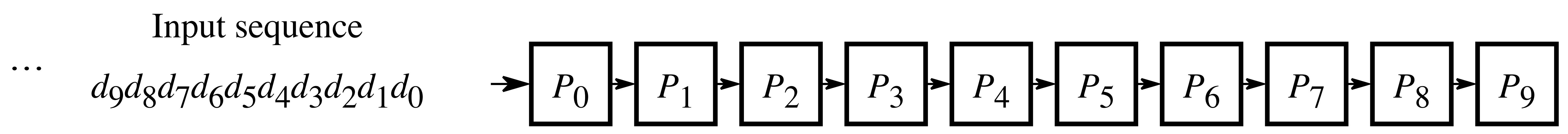
- Reduce effects of slowest stage by introducing task parallelism.
- Increases the throughput.



Worksheet #22: Analyzing Parallelism in an Actor Pipeline

Consider a three-stage pipeline of actors (as in slide 5), set up so that P0.nextStage = P1, P1.nextStage = P2, and P2.nextStage = null. The process() method for each actor is shown below.

Assume that 100 non-null messages are sent to actor P0 after all three actors are started, followed by a null message. What will the total WORK and CPL be for this execution? Recall that each actor has a sequential thread.



```
protected void process(final Object msg) {
    if (msg == null) {
        exit();
    } else {
        doWork(1); // unit work
    }
    if (nextStage != null) {
        nextStage.send(msg);
    }
}
```