Parallel Programming

Overall Goal

- Acquire mental model of how parallel computers work
- Learn basic guidelines on designing parallel programs
- Know what you need to look up later
- Emphasis is on multithreaded shared-memory programs, but most of the high-level concepts apply to any parallel program.

Why parallel programming?

- Functionality
- Throughput
- Turn-around
- Capacity

Functionality

- Sometimes parallel version is easier to write.
  - Often the case in stream processing
  - expand | sed's/ */ /'| fold -s
  - Responsiveness of human interface
    - Decouple human interface from computing portions
      - Compute in background
  - Decouple portions of user interface
    - Not have window block because another window's dialog box is open.

Throughput

- Thoughput = useful work per unit time
- Parallel program can improve throughput even on a single CPU
  - Example: hide I/O latency

Turn-around

- Solve problem faster
  - Typically by applying more resources
  - Parallel solution is sometimes better algorithm!
    - Parallel branch-and-bound may tighten bound quicker, resulting in less wasted work.
- Examples
  - Practical weather prediction
    - 24-hour forecast that takes 24 hours to compute is not useful.
  - Real time (e.g. video games, cybernetics)
    - Must meet deadline.

Basic parallel programming models

- Single Instruction Multiple Data (SIMD)
  - Single thread operating on long vector
- Multiple Instruction Multiple Data (MIMD)
  - Message passing
    - Each thread has its own address space
    - Threads communicate by sending and receiving messages
    - Example: MPI
  - Shared memory
    - Threads share a common address space
    - Threads communicate by writing and reading shared memory
    - Example: OpenMP
- Hybrid
  - Networks of shared-memory machines
  - Virtual shared memory across a cluster
- Totally implicit
  - E.g., applicative programming languages

Speedup

- Speedup measures improvement from parallelization

$$speedup(p) = \frac{best\_serial\_version}{version\_with\_p\_threads}$$

- Efficiency is relative to ideal speedup

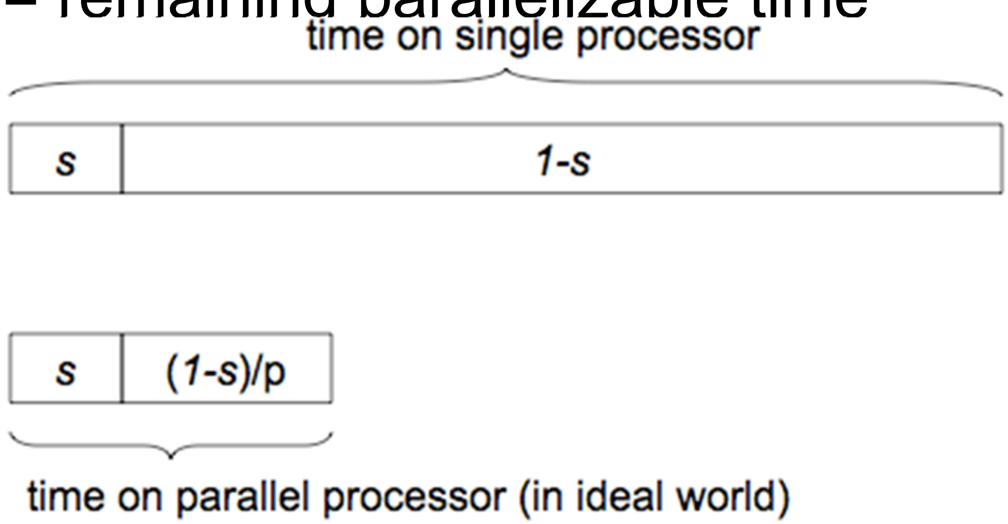
$$effiency(p) = \frac{speedup}{p}$$

Amdahl's "Law"

- Suppose program takes 1 unit of time to execute serially.
- Part of program is inherently serial (unparallelizable).

s= fraction of time spent in serial portion

1-s= remaining parallelizable time



Maximum speedup

$$speedup(p) = \frac{1}{s + \frac{1-s}{p}}$$

- Speedup is limited by serial portion of programs

$$speedup(p) \to \frac{1}{s}$$

- Situation is worse in real world, because parallel portion usually has inefficiencies

- This is depressing. Anyway to evade Amdahl’s law?

Gustafson/Barsis's argument

- Scale up problem size N as number of processors increases.
  - Assume that work in parallel portion grows as N does
    - E.g., finer mesh and more particles for gorier video game
    - Parallelizable portion now requires time N(1-s) to execute serially
- Keep serial portion independent of problem size
  - This is often achievable, though it requires care.
  - E.g., must never loop over Nin serial portion.
  - E.g., if amount of I/O depends on N, I/O must be parallel too.
- Net effect is to shrink serial fraction as number of processors grows.

Maximum speedup

$$speedup(p) = \frac{s + N(1-s)}{s + \frac{N(1-s)}{p}}$$

$$speedup(p) \to s + p(1-s) \quad \text{when N=p}$$

Multithreaded processors

- Simultaneous Multithreading (SMT)
  – Feed execution units from multiple instruction streams
  – Good fit for out-of-order machine
- Switch On Event Multithreading (SOEMT)
  – Switch to another instruction stream while waiting on an outer-level cache miss.
  – Good fit for in-order machine

Multicore processors

- Multiple CPUs in same socket or die.
- Multiple cores on same die has both advantages and disadvantages
  - Enables communication at chip speed instead of slower board speed.
  - Lower yield from wafer. Assuming random distribution of failures with good rate p, then only p2 cpu-pairs are good. Even lower for quad-core processors.

Where is future?

- Future speed improvements will mostly be from multithreading.
- Improvements in past have been for single thread
  - E.g., Higher clock rate, Bigger functional units (e.g. full array multipliers),Pipelining and out-of-order execution.
  - These all consume superlinear transistors or power
    - Approximately quadratic
    - Cooling capacity limited (cannot put toaster in lap)
- Improvements in future are for multiple threads
  - Multiple cores and more threads per core
  - As long as program scales significantly better than sqrt(p), we are ahead.

Communication between cores

- Intrachip locality is good. Interchip locality is bad.
- Shared bus
  - All devices on bus share the bus bandwidth
  - Uniformly slow for everyone
- Point to point
  - Non-Uniform Memory Access (NUMA)
  - Bus bandwidth grows as nodes are added

Bandwidth vs. latency

- Bisection bandwidth
  - Adversary cuts machine in half.
  - Measure bandwidth between halves.
- Latency
  - Near vs. far reference
  - For message passing, software adds order of magnitude (or worse) to hardware latency.

Shared Memory Is Really Message Passing

- Messages are cache lines/sectors
  - Typically around 32-128 bytes per line
  - Pentium®4 Processor has sectored lines
    - 64-byte “sector”and 128-byte “line”
    - Writes invalidate sectors
    - Reads pull in lines
  - For cache without sectors, sector is same as line
- When a processor writes to a cache sector, the other processors have to be notified of the change.

Programming for parallelism

- Decomposition and collaboration
  - “Many hands make light work”

Decomposition

- Parallel programming is about how to decompose serial work.
- Functional decomposition
  - Each thread performs a different function
  - Usually not scalable
- Data domain decomposition
  - Each thread works on different portion of data
  - Scales as domain grows
  - Scales as domain can be more finely partitioned

Four key considerations

- Granularity
  - The size of each chunk of work
- Load balance
  - How work is spread across processors
- Communication
  - The bandwidth and latency for exchanging information between processors or memory.
- Synchronization
  - Incurs latency, but is necessary for correctness

Granularity

- If grain size is too large
  - Limits parallelism to number of grains
- If grain size is too small
  - Parallel overheads swamp useful work.
    - Synchronization between grains
    - Communication between grains

Load balancing

- Unbalanced load can leave some processors idle.
- Excessive effort to balance load can hurt too.
  - Consumes cycles and bandwidth

Communication

- Too much communication hurts performance
  - Cost of latency
  - Swamp bandwidth

Synchronization

- Synchronization for correctness
  - Avoid race conditions
- Synchronization for performance
  - Match concurrency to available parallelism
- Fine-grain versus coarse-grain synchronization
  - Fine-grain improves concurrency
  - Fine-grain may incur more overhead
    - More synchronization operations
    - Synchronizers occupy more space

Excessive concurrency?

- Oversubscription = more logical threads than physical threads. ·
- Incurs overhead
  - Context-switching
  - Cache sharing by logical threads
  - Lock preemption
  - Each grain requires memory while running

Typical tradeoffs

- Smaller granularity $\Rightarrow$better load balancing, but more synchronization and often more bandwidth
- Better load balancing $\Rightarrow$more synchronization and bandwidth
  - Migration is not free
- Less synchronization $\Rightarrow$more memory space
  - More tasks active at the same time

Scheduling in OpenMP

- static
  - Each thread gets 1/p of iterations
  - Tiny amount of communication
  - Poor load balancing if iterations differ in time
- dynamic
  - Threads grab iterations dynamically
  - Very good load balancing
  - Lots of communication via central shared counter
- guided
  - Threads grab chunks of iterations in exponentially decreasing size
  - Better load balancing than static, but worse than dynamic. More communication than static, but less than dynamic

Decomposition patterns

- Work queue
- Geometric
- Divide and conquer
- Parallel pipeline

Work queue

- Idle threads grab work from queue
  - Or master dishes out the work to slaves
- Good load balancing
- Queue contention can be problem
  - Distributed queue helps
- Poor locality
  - FIFO by nature works against LRU cache
  - FIFO can cause breadth-first behavior with high space demand

Geometric decomposition

- Stencil computations are quite common for solving partial differential equations (PDE) for physical systems
  - Heat flow
  - Wave propagation Reservoir simulation
- Each cell value is function of neighbors or nearby cells
  - E.g., bi,j = f(ai+1,j, ai-1,j ai,j+1,ai,j-1)

Divide and conquer

- Recursively divide problem into parallel subproblems
- Some parallel programming languages use this approach for everything
  - Excellent space behavior
  - Good load balancing
  - Good locality and cache behavior
- Requires care in scheduling to avoid exponential space!

Hybrid decomposition

- Divide-and-conquer on geometric decomposition
  - Recursively subdivide grid until processors are busy
  - All the advantages of cache-oblivious programming and good load balancing!

Parallel pipeline

- Input is sequence of items
  - E.g., sound or video frames
- Two kinds of stages
  - Serial: can operate on only one piece of data at a time
    - E.g., counters, filters with feedback loops
  - Parallel: can operate on more than one piece of data simultaneously
    - E.g., FFT, convolutions, etc.
- Scales well if serial stages are few and fast

Specification of parallelism and synchronization

- Parallel languages
- Pragmas
- Run-time libraries

Parallel languages

- Examples
  - Sunss proposed Fortressmakes loops parallel by default
  - Fortran 90 has parallel array operations
- Issues
  - Give compiler a lot of leverage
  - Do not work well with legacy code

Pragmas

- Example: OpenMP
  - Programmer can add pragmas to existing code to parallelize it
  - Works for Fortran, C, and C++ that look like Fortran
- #pragma omp parallel for
  - for(int i=start; i<=end; i+=2)
    - if(TestForPrime(i))
- #pragmaompatomic
      - PrimesFound++;

Run-time library

- Message-passing libraries
  - Message Passing Interface (MPI)
- Threading libraries
  - POSIX pthreads
  - Windows threads

Typical thread library

- Create thread
- Wait for thread to finish
- Mutual exclusion (critical sections)
  - mutexes, condition variables
- Thread-private storage
- Thread cancellation
- Atomic operations

Race conditions

- These are the primary correctness headache.
  - Non-deterministic results
- Example:

<u>Initial State</u>
int X = 0;

<u>Thread 1</u>
t1 = X
t1 = t1 + 1
X = t1

<u>Thread 2</u>
t2 = X
t2 = t2 + 2
X = t2

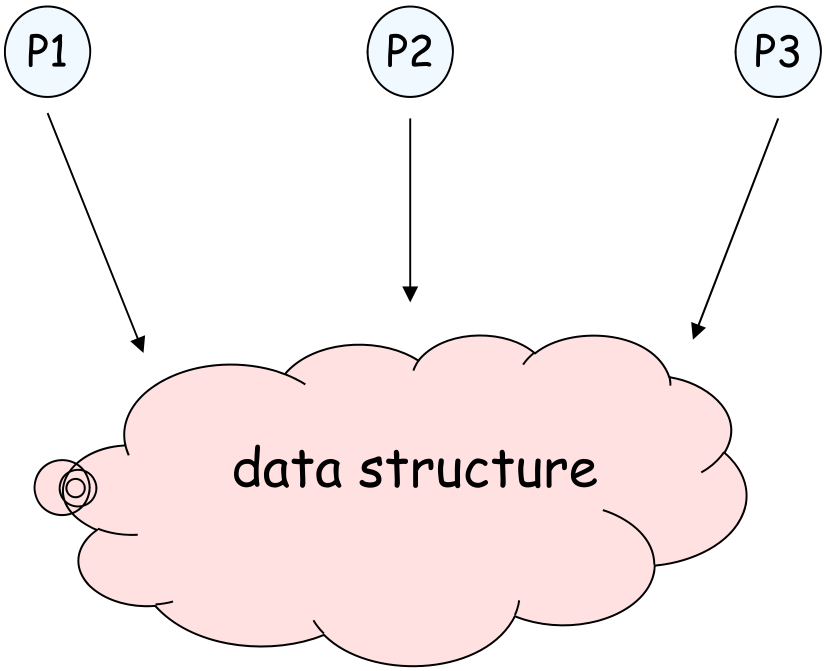
<u>Final State</u>
$X \in \{1,2,3\}$

Synchronization

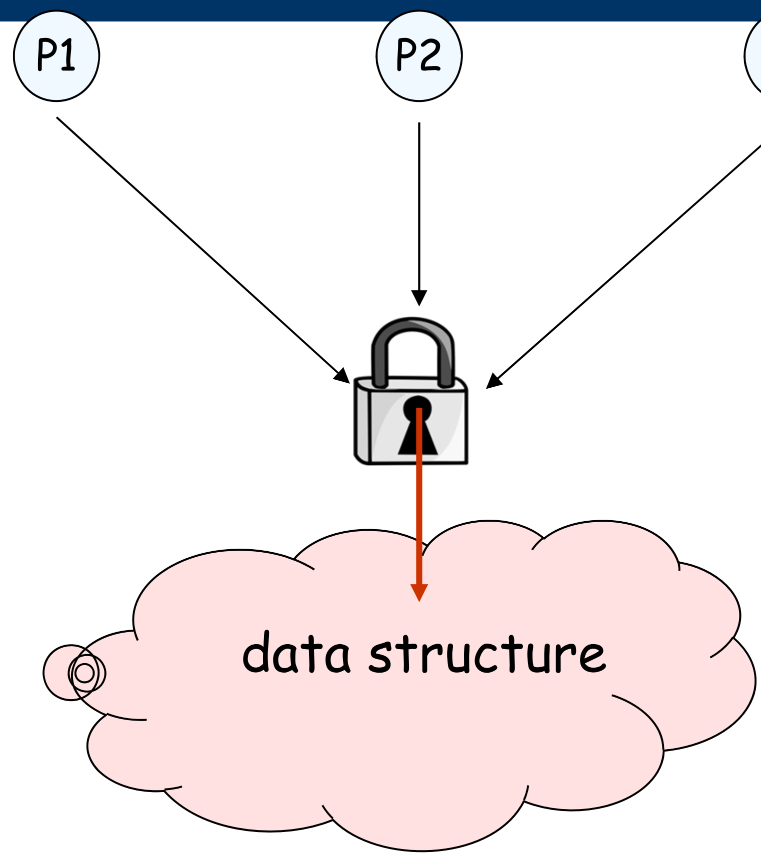
- Low-level
  - Mutexes, condition variables, events
  - Atomic operations
  - Emphasis is on pair of threads
- High-level
  - Parallel loops
  - Pipelines
  - Barriers
  - Queues

Blocking and Non-blocking Synchronization
Concurrent data structures

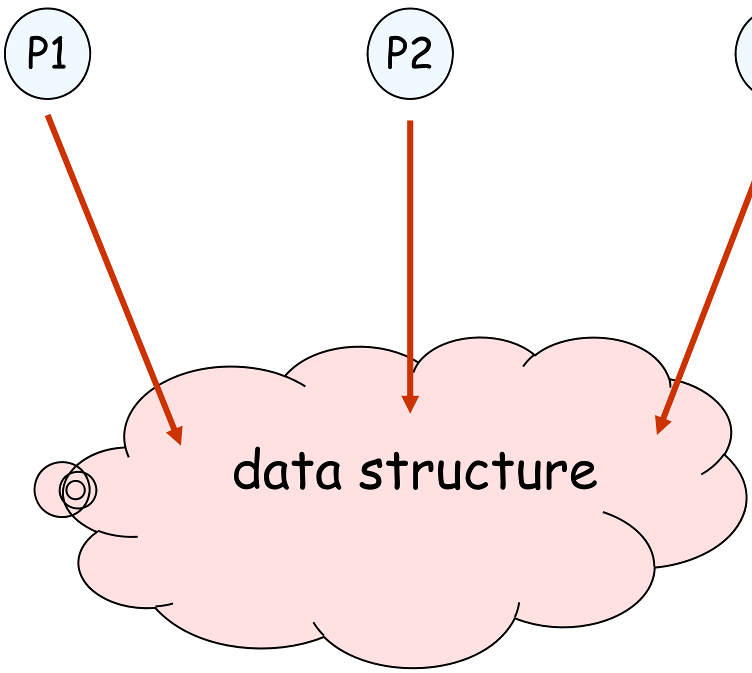
counter, stack, queue, link list



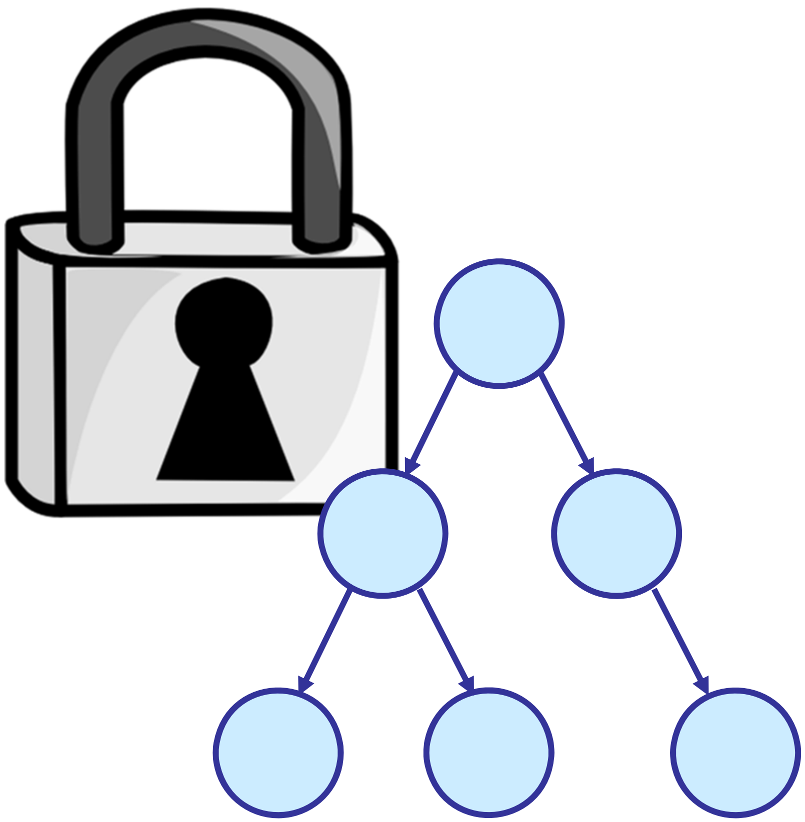
Blocking



Non-blocking

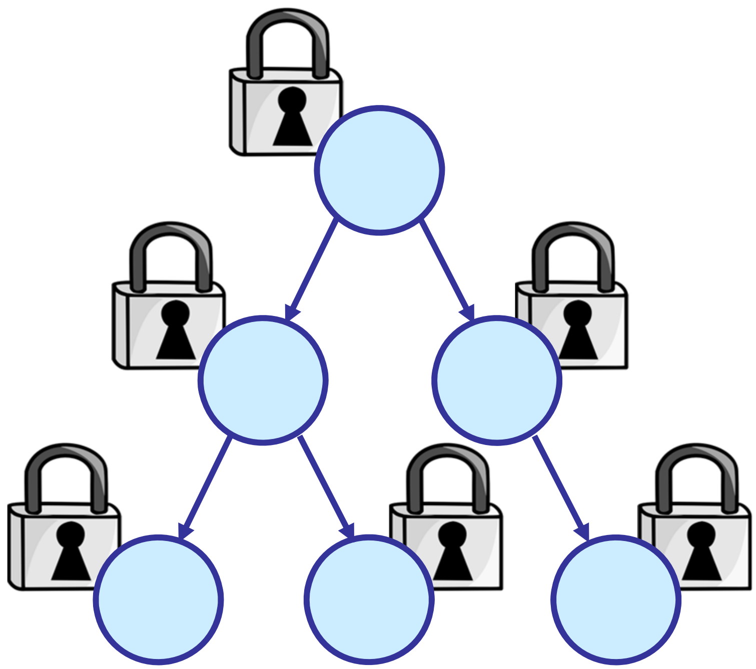


Coarse-Grained Locking

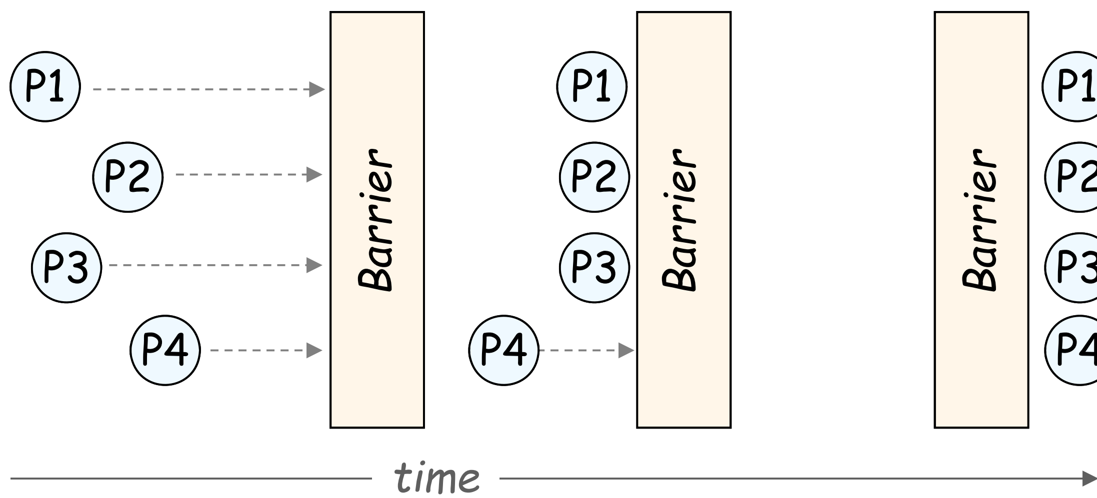


Easier to program, but not scalable.

Fine-Grained Locking



Efficient and scalable, but
too complicated (deadlocks, etc.).



four processes approach the barrier

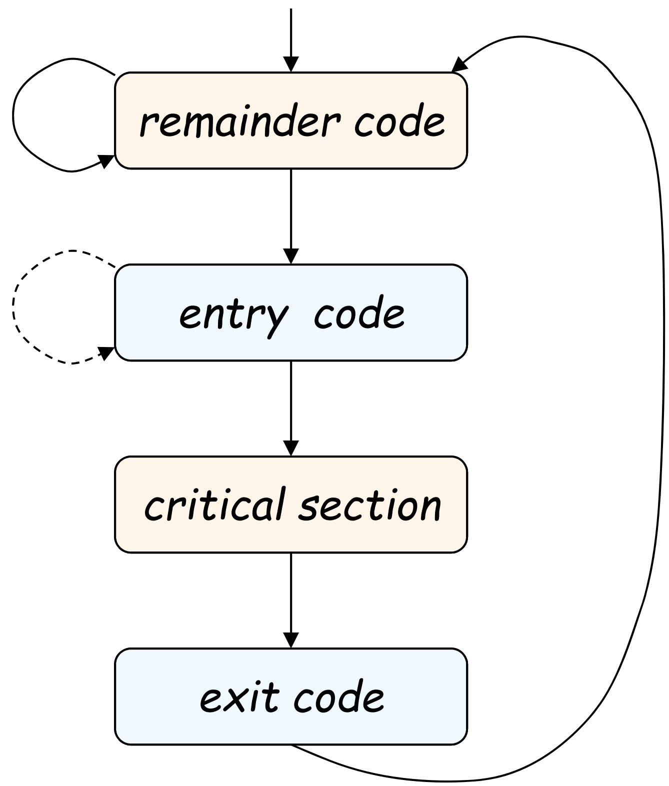
all except P4 arrive

Once all arrive, they continue

Flavors of mutual exclusion

- Mutex, critical section
  - Let one thread in at a time
- Semaphore
  - Let <=N threads in at a time.
- Reader-writer lock
  - Let multiple readers or one writer in at a time
  - Useful when reading dominates writing
- Condition variables
  - Allows threads to wait for state protected by mutex to change, without holding the mutex.
  - And without creating timing holes!

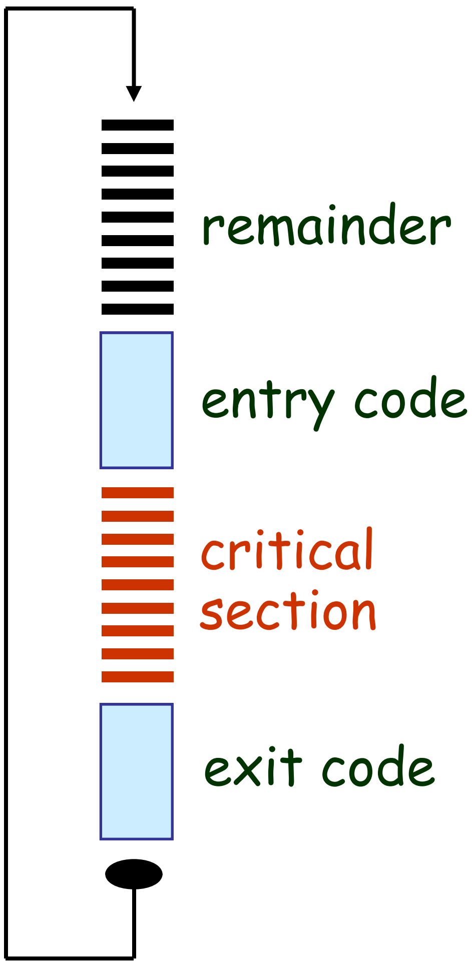
The mutual exclusion problem



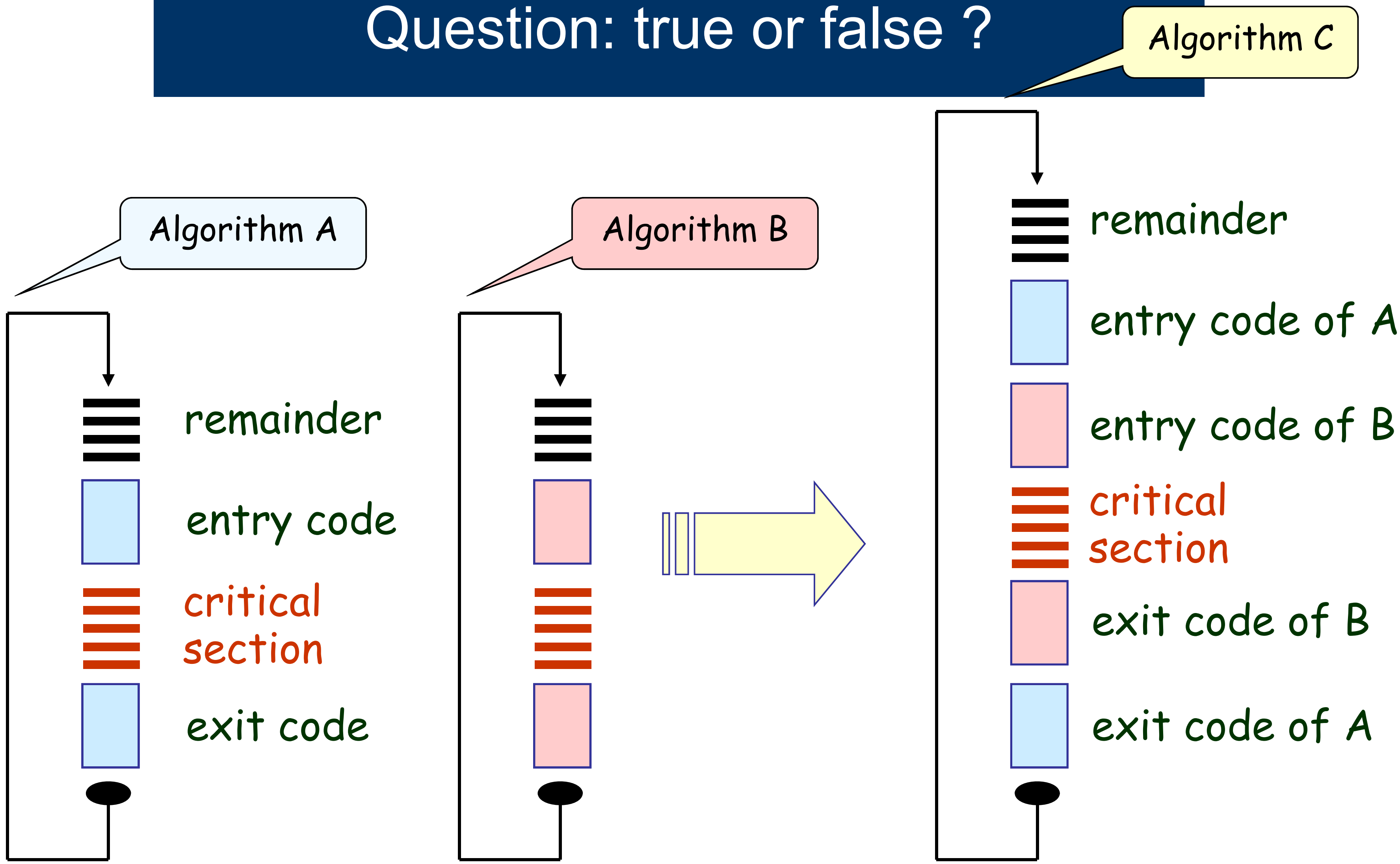
The problem is to design the entry and exit code in a way that guarantees that the mutual exclusion and deadlock-freedom properties are satisfied.

The mutual exclusion problem

- Mutual Exclusion: No two processes are in their critical sections at the same time.
- Deadlock-freedom: If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.
- Starvation-freedom: If a process is trying to enter its critical section, then this process must eventually enter its critical section.

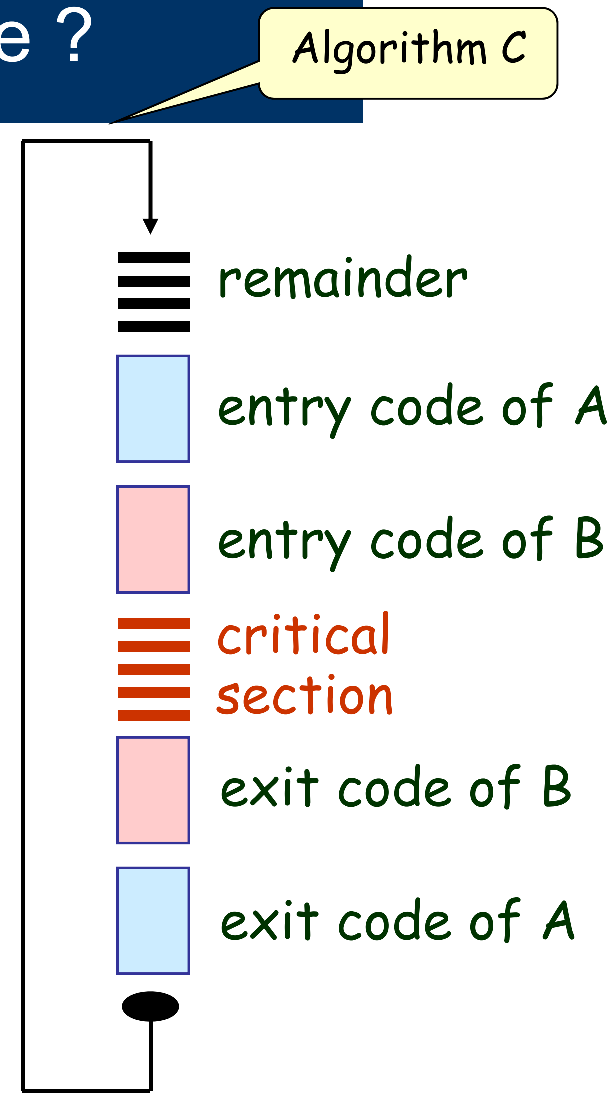


Question: true or false ?



Question: true or false ?

- A and B are deadlock-free ➔ C is deadlock-free.
- A and B are starvation-free ➔ C is starvation-free.
- A or B satisfies mutual exclusion ➔ C satisfies mutual exclusion.
- A is deadlock-free and B is starvation-free ➔ C is starvation-free.
- A is starvation-free and B is deadlock-free ➔ C is starvation-free.



Proposed lock solution 1

Thread 0	Thread 1
flag[0] = true	flag[1] = true
while (flag[1]) {skip}	while (flag[0]) {skip}
critical section	*critical section*
flag[0] = false	flag[1] = false

flag

0 false

1 false

✓ mutual exclusion

X deadlock-freedom

Does it work?

Proposed lock solution 2

Thread 0	Thread 1
while (flag[1]) {skip}	while (flag[0]) {skip}
flag[0] = true	flag[1] = true
critical section	*critical section*
flag[0] = false	flag[1] = false

flag

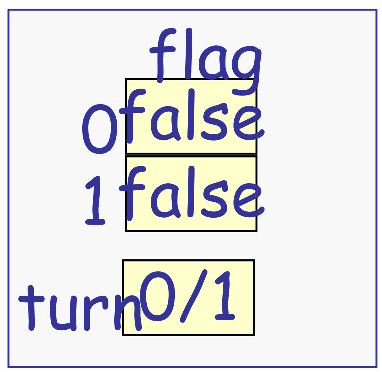
0	false
1	false

X mutual exclusion
✓Deadlock-freedom

Does it work?

Peterson’s algorithm

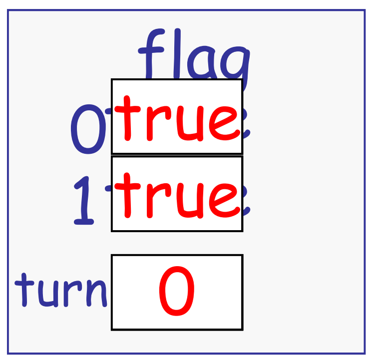
Thread 0	Thread 1
flag[0] = true	flag[1] = true
turn = 1	turn = 0
while (flag[1] and turn = 1)	while (flag[0] and turn = 0)
{skip}	{skip}
critical section	*critical section*
flag[0] = false	flag[1] = false



A variant of Peterson’s algorithm

Is it correct ?

Thread 0	Thread 1
turn = 1	turn = 0
flag[0] = true	flag[1] = true
while (flag[1] and turn = 1)	while (flag[0] and turn = 0)
{skip}	{skip}
critical section	*critical section*
flag[0] = false	flag[1] = false



Problems with locks

- Composition
  - Locking lower level operations does not guarantee that higher-level operation is race-free.
- Deadlock
  - Cycle of threads that have each acquired a lock, and are waitingto acquire another's thread's lock.
- Convoying
  - If owner of lock is preempted, other threads wait behind it.
  - If owner of lock crashes, other threads wait forever
- Priority Inversion
  - Can occur with prioritized preemptive scheduling
    - Low-priority thread is preempted while holding lock
    - Medium-priority thread runs in preference to low-priority thread
    - High-priority thread waits forever on lock

Compose locks

- Composing thread-safe operations does not guarantee that result is thread-safe.
- Example: implementing a set using a thread-safe list
  – Invariant is “each key occurs once in list”

```
void add_if_not_present(key) {
    if( !list.contains(key) )
        list.append(key);
}
```

Between the time we check for the key and insert it, another thread might append the same key.

Remember to lock outermost invariant.
Locks in inside levels just waste time.

This is a challenge for reusable components.

Profiling parallel code

- Use a profiler to find where time is spent
  - Do not waste time rewriting portions that contribute little to run time.
  - Poor man’s profiler
    - Stop program manually at random times
- If goal is turn-around time, find portions that are on critical path
  - These are what affect total time
- What if goal is throughput?