Threads and Synchronization

CS 370 Spring 2019

Overview

- Review
- Deconstructing a Process
- Two Threads are Better Than One
- User level Threads vs Kernel Level Threads
- Convert Single Threaded to Multi Threaded
- POSIX Thread Library (pThread)

Review: Mutual Exclusion

- Correctness Property: Given two critical section executions for thread A and thread B: Mutual Exclusion property states that the ith execution of CS by A, $CS^i{}_A$, will **precede** the jth execution of CS by B, $CS^j{}_B$ or $CS^j{}_B$ will **precede** $CS^i{}_A$.
- Either $CS^i{}_A \rightarrow CS^j{}_B$ or $CS^j{}_B \rightarrow CS^i{}_A$ (they will not execute simultaneously)
- Deadlock satisfies this condition vacuously (no CS, no problems)
- Solutions to Mutual Exclusion also add:
- Deadlock free (always)
- Starvation free (sometimes)

Peterson's 2-Thread algorithm

	Thread A	**Thread B**
pre)	`x: lockA = true;`	`x: lockB = true;`
	`y: turn = B;`	`y: turn = A;`
	`z: while( lockB`	`z: while( lockA`
	`w:       && turn != A );`	`w:       && turn != B );`
post)	`x1: lockA = false;`	`x1: lockB = false;`

- What properties does this satisfy?
- Mutual Exclusion?
- Deadlock Free?
- Starvation Free?
- k-Bounded Waiting?

Bakery Algorithm

```
// declaration and initial values of global variables
Entering: array [1..NUM_THREADS] of bool = {false};
Number: array [1..NUM_THREADS] of integer = {0};

lock(integer i) {
    Entering[i] = true;
    Number[i] = 1 + max(Number[1], ..., Number[NUM_THREADS]);
    Entering[i] = false;
    for (integer j = 1; j <= NUM_THREADS; j++) {
        // Wait until thread j receives its number:
        while (Entering[j]) { /* nothing */ }
        // Wait until all threads with smaller numbers or with the same
        // number, but with higher priority, finish their work:
        while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) {
    }
}
```


```
// declaration and initial values of global variables
Entering: array [1..NUM_THREADS] of bool = {false};
Number: array [1..NUM_THREADS] of integer = {0};

lock(integer i) {
    Entering[i] = true;
    Number[i] = 1 + max(Number[1], ..., Number[NUM_THREADS]);
    Entering[i] = false;
    for (integer j = 1; j <= NUM_THREADS; j++) {
        // Wait until thread j receives its number:
        while (Entering[j]) { /* nothing */ }
        // Wait until all threads with smaller numbers or with the same
        // number, but with higher priority, finish their work:
        while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) {
    }
}
```


- What properties does this satisfy?
- Mutual Exclusion?
- Deadlock Free?
- Starvation Free?
- k-Bounded Waiting?

Processes

- A process is an instance of an executing program
- Include current values of pc, registers, variables.
- **Multiprogramming** is the ability to switch between multiple processes in a system (Concurrency)
- Each process gets its own entry in the Process Control Block (PCB).
- Each process gets its own address space (memory)
- Each process runs a thread of execution and the PCB saves the state of the CPU when performing a context switch.

An Alternative View of a Process

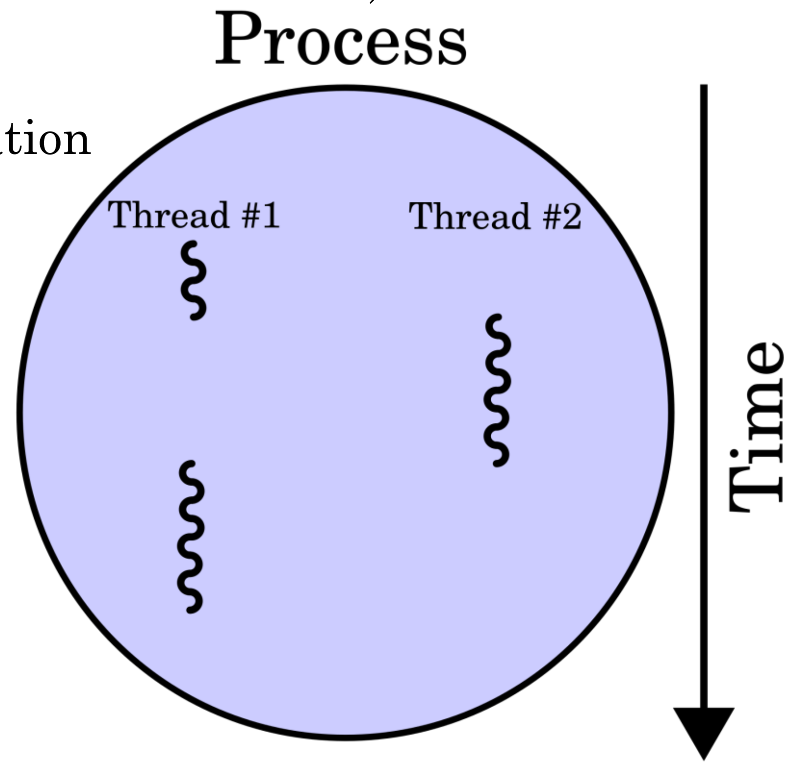
- Two Independent Concepts: **Resource grouping** and **Execution**
- Resources
  - Address space (code and data)
  - Open files
  - Child processes
  - Accounting information (stats for scheduling, etc.)
- Thread of Execution
  - PC
  - Registers
  - Stack
  - State (Ready, Running, Waiting, Terminated)
- Processes are used to group resources together, threads are entities scheduled for execution on the CPU.

Threads

- “Two threads are better than one, but ten threads without wit are good as none.” – John Heywood, 1546
- A “light weight process”
- Each thread shares the same address space and resources with the parent process.
- A process can have multiple threads of execution (threads) which can be scheduled independently of each other.
- Threads work together to complete the task of the process.
- Safety?
  - Processes are isolated from each other
  - Threads are created by a process, so they are assumed to be safe
  - No precautions are taken

Benefits of Using Threads

- An application may have multiple independent tasks to do at once
  - Threading simplifies this by decomposing each into separate task threads
- Faster to create and destroy threads than processes (10-100 times faster)
  - **Economy**
- Share an address space for fast inter-thread communication
  - **Resource Sharing**
- Better performance in the entire application
  - I/O bound threads don't block the entire application
    - Increased **Responsiveness**
- Multiprocessing (Parallelism)
  - **Scalability**

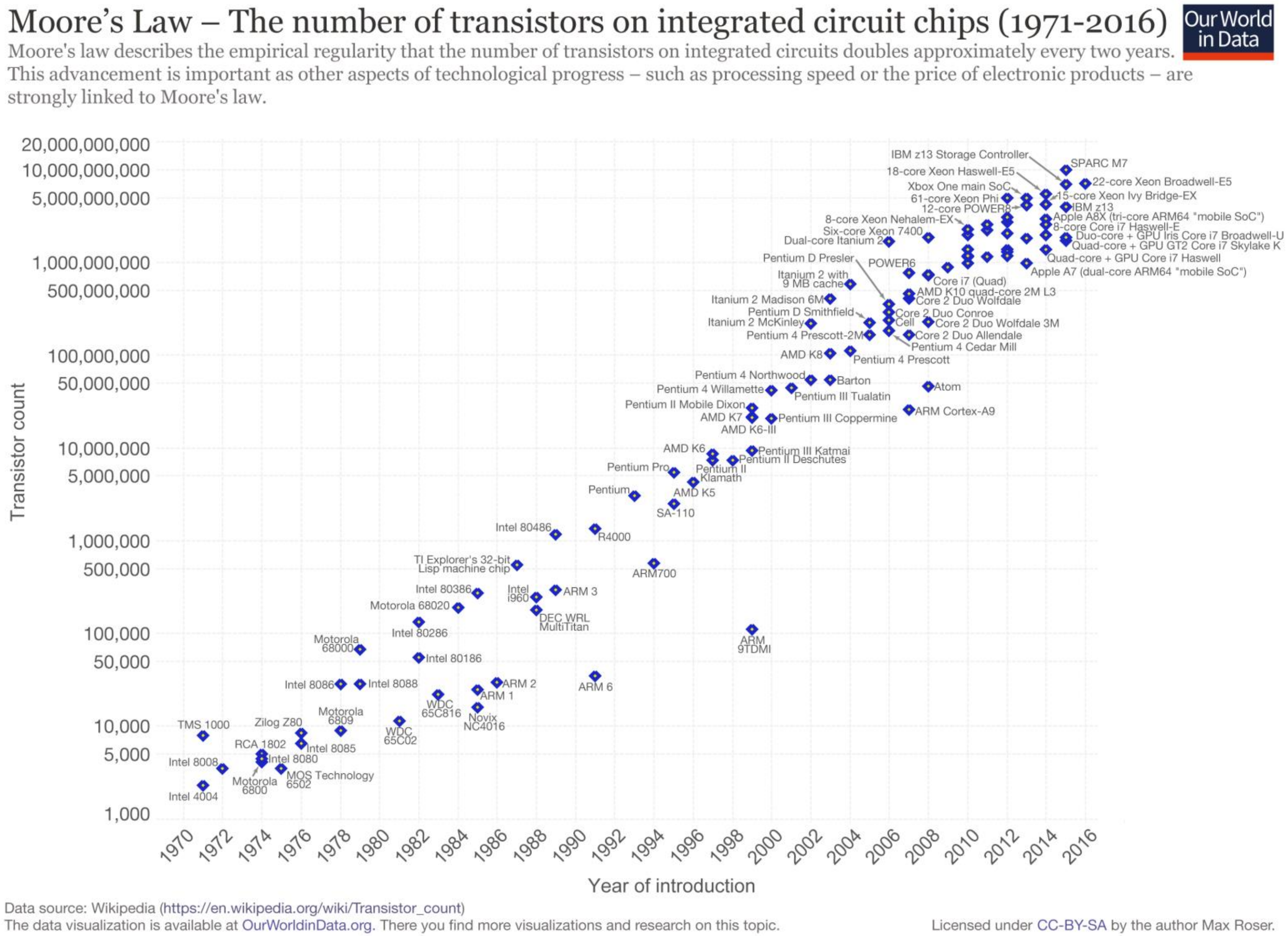


Single Thread to Multi Thread

- What things can go wrong?
- Global variables read and written by two or more threads simultaneously
- Can library calls support calls in parallel?
- Signals like alarms and interrupts
  - Who should receive keyboard interrupt? One thread, all threads?
- Stack management in user level threads

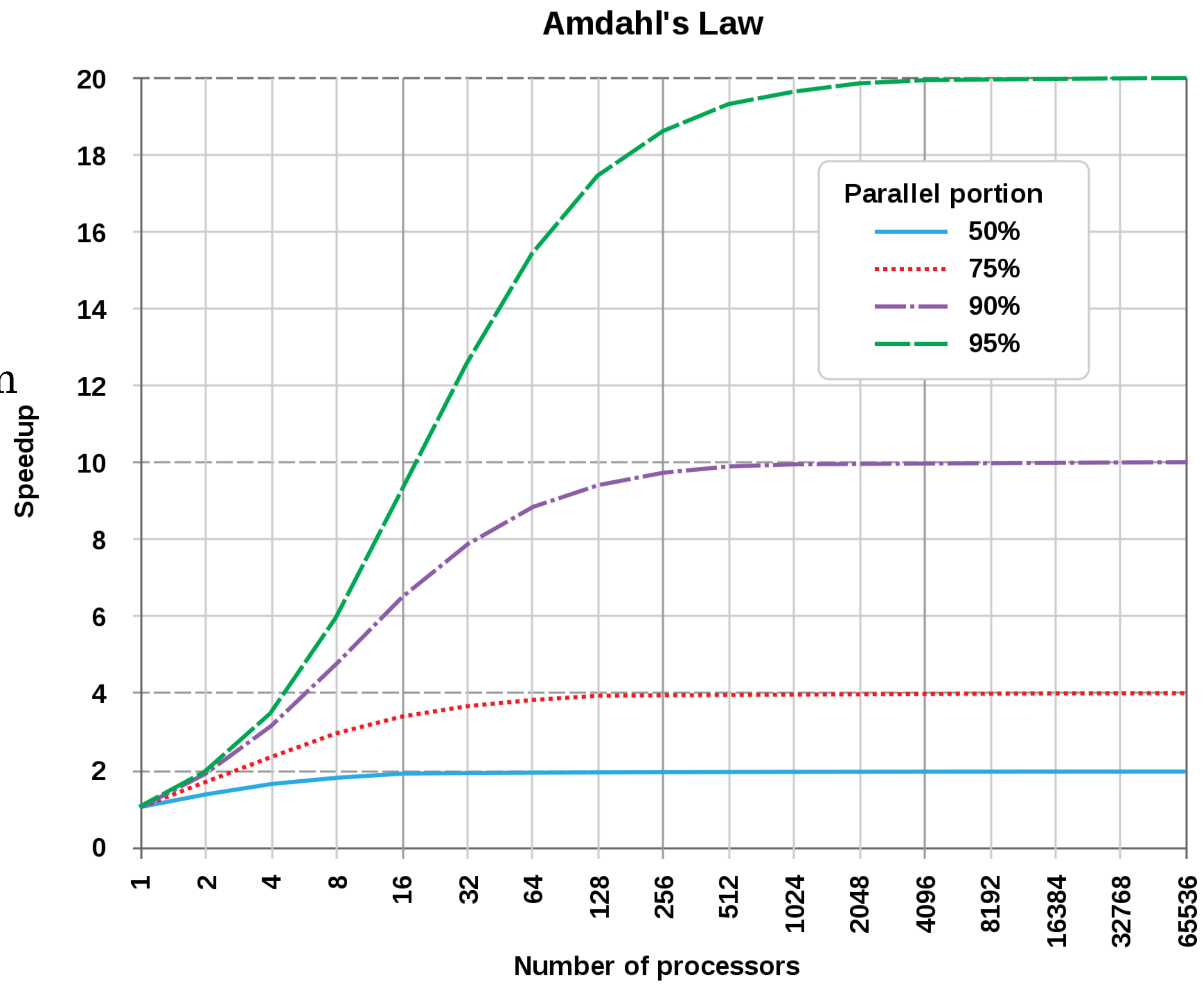
Moore's Law

- Gordon Moore in 2015:
- "I see Moore's Law dying here in the next decade or so."



Amdahl's Law

- Gives an upper bound for the *speedup* factor of adding N cores to a program
- $speedup \leq \frac{1}{S + \frac{1-S}{N}}$
- S is the portion of the program that must be run sequentially
  - $0 \leq S \leq 1$
- $1 - S$ is the parallel portion



User Level vs Kernel Level

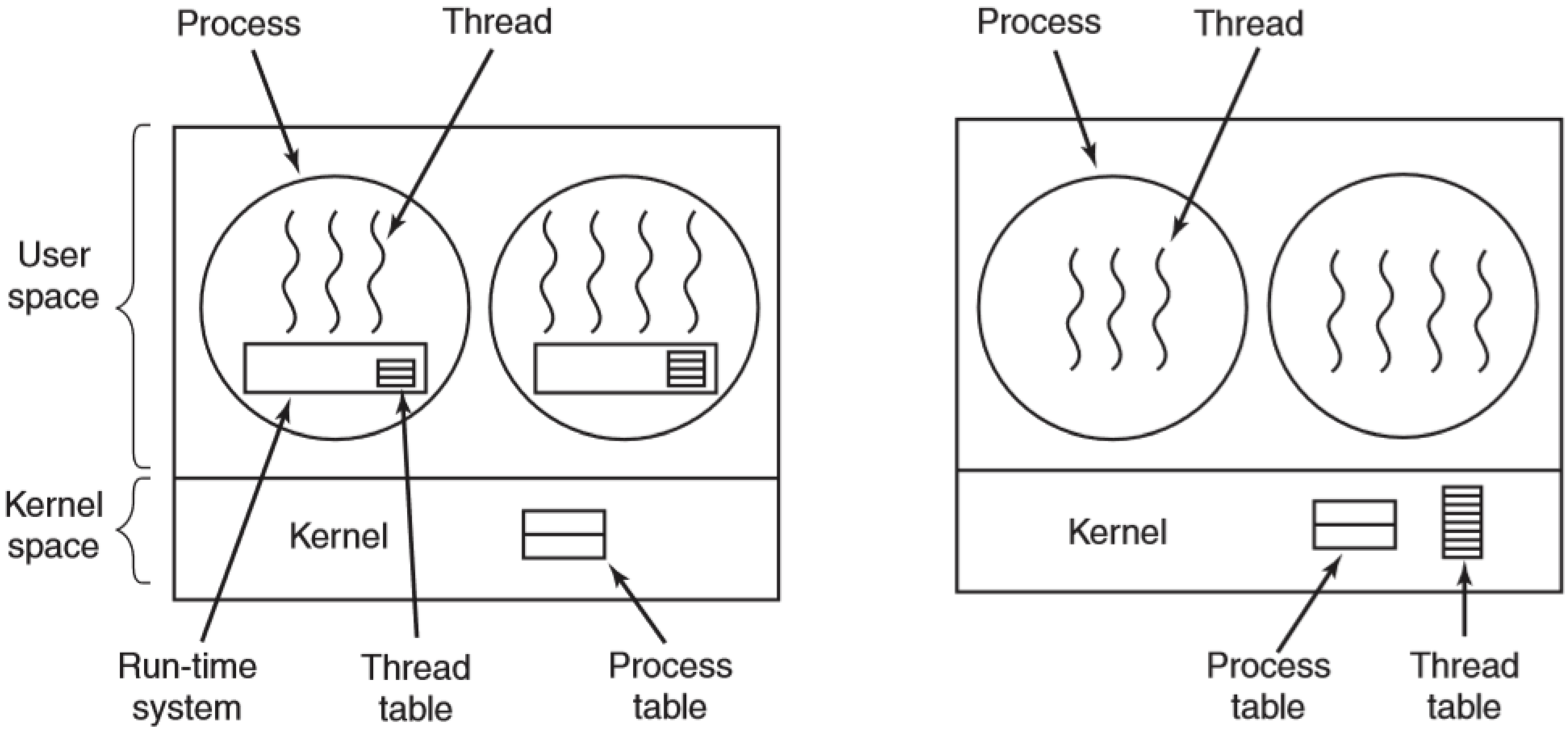


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

User Level Threads

- Each process needs its own thread table (contains pc, registers, state, stack)
- At least 10x faster than invoking the kernel
- User gets to choose proprietary scheduling algorithm
- Can decide when is a good time for a thread to block in terms of the program execution rather than using time slices
- **Problem**: Blocking System Calls
  - What happens when a thread calls *read*()?
- **Problem:** Page Faults Block
- **Problem:** Multiprocessing environments
  - Can user-level threads achieve "true parallelism" if multiple CPUs are available?

Kernel Level Threads

- Kernel keeps the thread table of all threads in the system
  - Keeps track of pc, registers, stack, state
- Thread creation and deletion invokes the kernel (slower)
- Subject to the kernel's scheduling algorithm
- May run another thread of the same process, or a different process's thread
- Resolves problems with system calls
- Can take advantage of multiprocessing!

Multithreading Models

- Many to One
  - Multiple user level threads supported by a single kernel level thread
- One to One
  - Each user level thread is supported by a kernel level thread
- Many to Many
  - Hybrid model, N user level thread supported by up to N kernel threads

Parallelizing a Problem

How to think about Threading

- Embarrassingly Parallel
  - Problem can be broken down without dependence on other threads
  - Rendering frames for display
  - Password Cracking
- Inherently Sequential Problems
  - Problem can not be broken down into threads
  - Iterative methods e.g. Newton's Method
- Matrix Multiplication, C = A*B
  - Fairly difficult to parallelize, can be done
  - Cache becomes important.
  - Use B transpose! Multiply rows in parallel
  - ~8x speedup with 4 threads on 4 core machine and matrix sizes > 2000 x 2000
    - Superlinear speedup! Tomikj and Gusev paper, and Gusev and Ristov paper

Computers are Asynchronous

- Threads can experience sudden, unpredictable delays:
- Cache Misses
  - ~1-10 cycles
- Page faults (long)
  - ~1,000s of cycles
- Scheduling timeslice used up (really long)
  - ~1,000,000s of cycles
- These are common cases for why a thread is called “slow” or “lazy” in reasoning about parallel algorithms.

Race Conditions

Cntr++

Cntr++

4

cntr

Race Conditions

- When the behavior of a system is **dependent** on the sequence or timing of other uncontrollable events
- Ex: Two threads writing simultaneously to a shared counter
- Are race conditions always bad?
  - No!
  - Ex: Two threads read the same constant variable at the same time
- If the counter does not need to return unique values, then race condition is ok
  - Ex: random number generator's seed value in some contexts

Making a Counter Solutions

- readVal = cntr.getAndIncrement()
- Reads the value and increments it
  - Like readVal = cntr++;
- How can we avoid a race condition on cntr?
- cntr.getAndIncrement(){
  - temp = cntr;
  - cntr = temp + 1;
  - return temp;
- }

Making a Counter Work

- Intuition: How should a counter work?
- Correctness Conditions (Safety Properties)
  - Nothing bad will ever happen
- Progress Conditions (Liveness Properties)
  - Something good happens eventually
- What does a **correct** counter do?
- What kind of **progress** can a thread make?

Making a Counter Work

- Specification:
  - Must return unique values to all calling threads
  - Must be monotonically increasing
- No deadlock

Counter Solutions

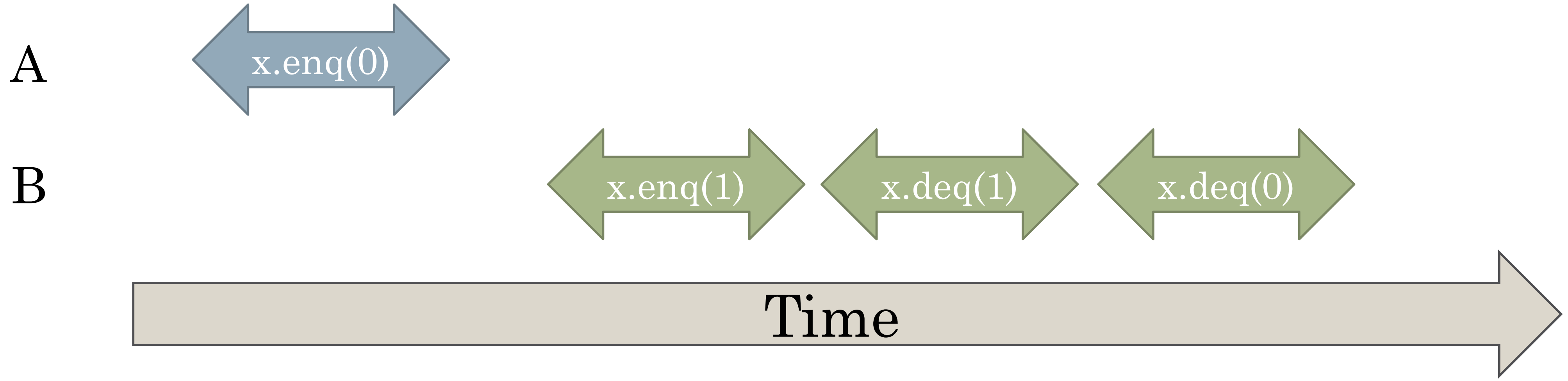
- Do nothing
  - Hope that nothing bad happens
  - Allows race condition, does not solve the problem

- Lock access to function getAndIncrement
  - **Pro**: Solves race condition
  - Uses mutual exclusion to guarantee correctness
  - Algorithm for mutual exclusion will have some progress guarantee
    - Deadlock free, starvation free, etc.
  - **Con**: Added overhead of mutual exclusion algorithm

- Glue instructions together into one indivisible unit
  - **Pro**: Solves race condition
  - Use stronger primitives to avoid using locks
  - Hardware R-M-W operations
  - **Con:** Using stronger instructions too much can slow down performance

Correctness Condition

- **Program Order:** The order of instructions in which a program is written
- **Sequentially Consistent:**
  - Each thread must take effect in program order
  - Can interleave calls of threads in any order we want.
- C11 Atomic class guarantees this!

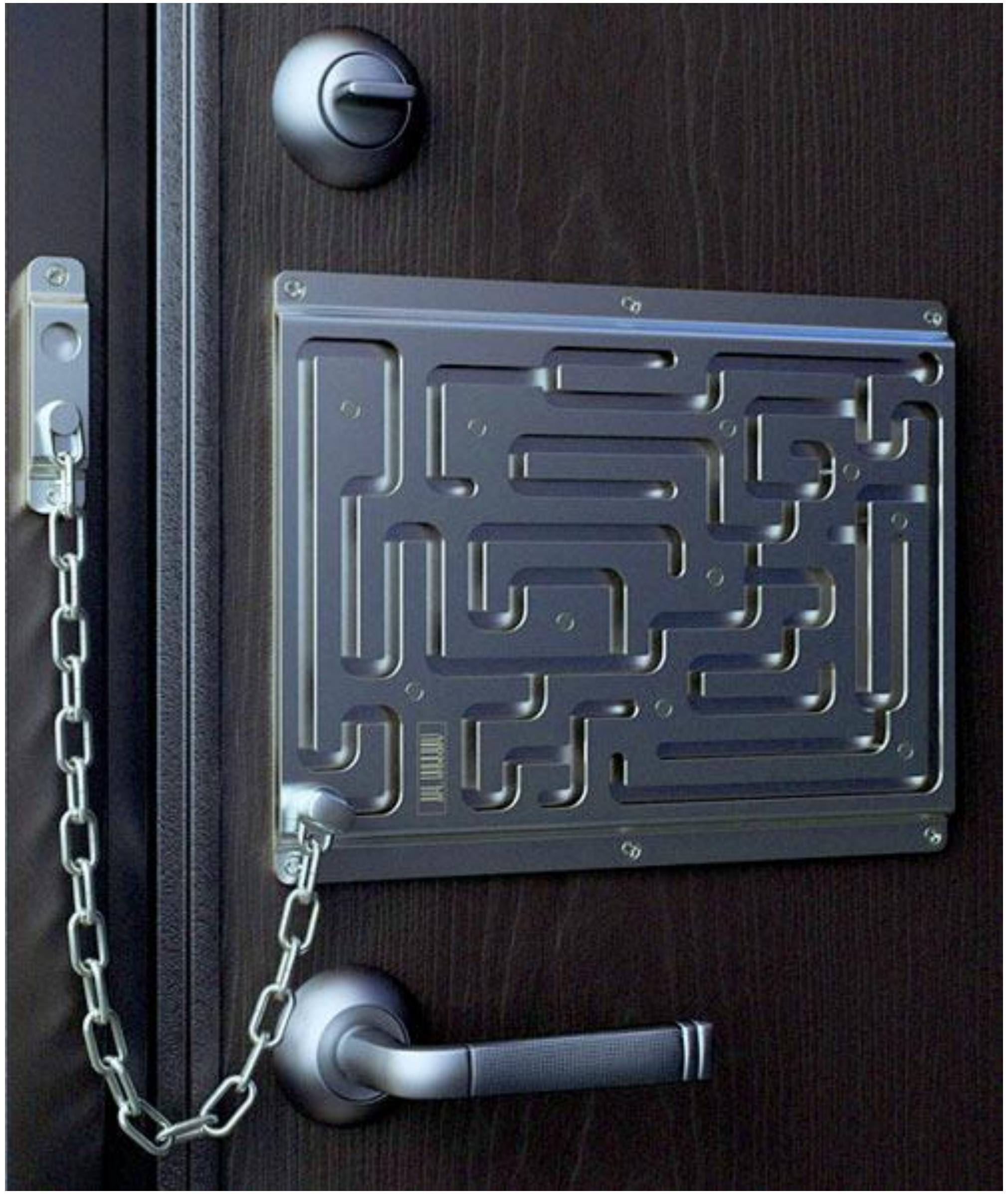


Progress Conditions

- These guarantee some progress will be made
  - “Liveness”
- Deadlock Free
  - Some thread trying to acquire the lock will succeed
- Starvation Free
  - All threads trying to acquire the lock will eventually succeed
- Lock Free
  - Some thread calling a method will return
- Wait Free
  - All threads that call a method will return

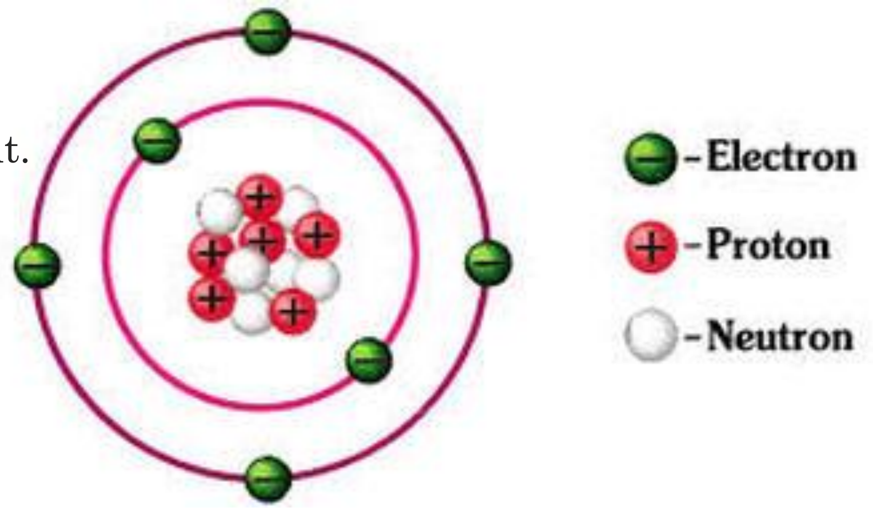
Locks

- Mutexes
  - pthread_mutex_t
  - Lock
    - One thread can acquire the mutex at a time
  - Unlock
    - Owner must unlock the lock
- Semaphores
  - pthread_semaphore_t
    - Initialize the value to an integer
  - Wait
    - Wait for available token (value > 0)
  - Signal/Post
    - Add a token
- Is a semaphore initialized to 1 the same as a mutex?



Atomics R-W

- Atomic means unable to be divided
- Can read or write to a location without being interrupted
- #include <atomic.h>
- atomic_load(atomic_type* A);
- atomic_store(atomic_type* A, type V);
- Sequentially Consistent Correctness Condition
  - Each thread must take effect in program order
  - Can interleave calls of threads in any order we want.



- Are atomic variables lock free?

Stronger Primitives R-M-W

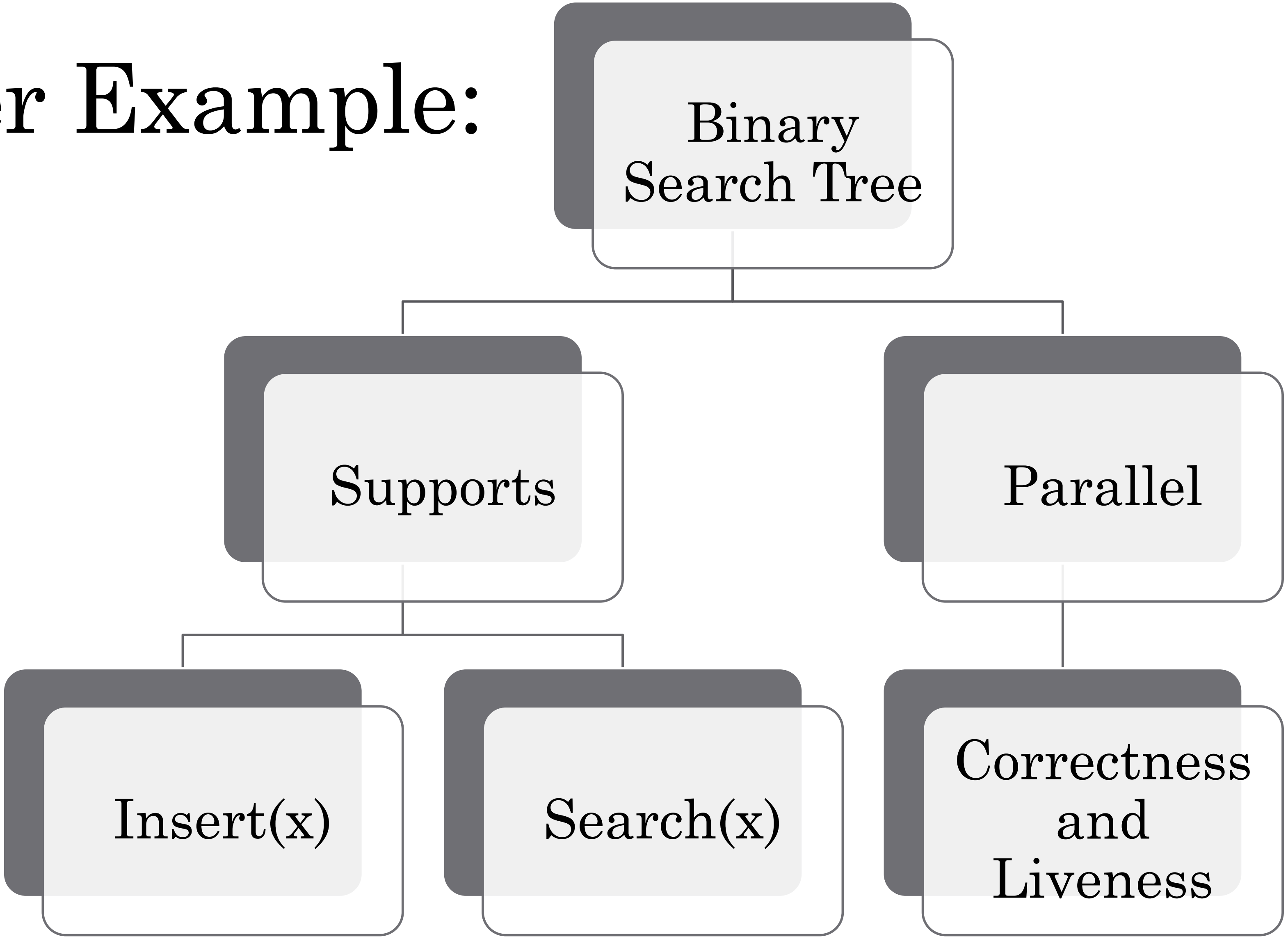
- The Lock-free and Wait-free Toolbox
- Test-and-Set
- Fetch-and-Add
- Compare-And-Set
- RISC: Load-Link (LL) and Store-Conditional (SC)

Examples

Another Example: Primes

- You want to find all the primes between 1 and N:
- Sequentially
- In Parallel with P processors
- Correctness?
- Liveness?

Another Example:



BST Search(x, root)

- Search(x, root){

```
itr = root
while( itr != nil ){
   if(itr.item == x)
      return itr
   if(itr.item < x)
      if(itr.leftChild == nil)
         return itr
      itr = itr.leftChild
   else
      if(itr.rightChild == nil)
         return itr
      itr = itr.rightChild
}
return nil
```

}

BST Insert(x, root)

- Insert(x, root){

```
1. while(true){
2.    node = Search(x, root)
3.    if node.item == x
4.       return false
5.    if( CAS(node, nil, node(k) == true )
6.       return
7.    root = node
   }
}
```

5 Challenges

- Identifying Tasks
  - What ideally independent tasks can be run in parallel?
- Balance
  - Does each thread have nearly equal work?
- Data Splitting
  - How should data accessed and changed by threads be split?
- Data Dependency
  - If two or more threads access/change the same data or depend on data from another, synchronization may be necessary.
- Testing and Debugging
  - Due to asynchrony, threads can run at different times and the interleaving of procedures can occur in many different ways

Types of Parallelism

- Data Parallelism
  - Share data among threads, each thread does the same operation
  - Ex: Two threads sum half the data each and combine
- Task Parallelism
  - Each thread does different tasks
  - Ex: Word processor with different threads for keyboard input, spell checking, saving backups periodically
  - Ex: One thread computes the mean, another thread computes the median
  - Ex: Web server with dispatch and worker threads

Lock or Not?

- Sometimes a clever way to think about the problem can yield a good wait-free or lock-free solution
- Sometimes good wait-free and lock-free solutions are simple!
- Don’t be quick to throw locks at a problem!
- Other times, we can’t eliminate all dependencies without incurring large overhead
  - Use locks!
  - Try to keep Critical Sections small.

POSIX Threads

POSIX Thread Library (pThread)

- A specification **not** an implementation.
- #include <pthread.h>
- Create a thread with:
- **int pthread_create(pthread_t ***thread***, const pthread_attr_t ***attr***, void *(***start_routine***) (void *), void ***arg***);**
- Exit the thread with:
- **void pthread_exit(void ***retval***);**
- Offers synchronization objects too:
  - pthread_mutex_t
  - pthread_semaphore_t
  - pthread_cond_t
- See handout for Project 5 with detailed syntax

pThread Continued

- Main function typically will create the threads
- "With great power comes great responsibility"
- Function that creates the threads should ensure they finish before terminating itself
- Main will **wait** for the threads to finish
- Then main will **join** with the threads
- **int pthread_join(pthread_t *thread*, void ***retval*);**
  - If thread has called exit, then this call just gets return val and completes
  - Else, caller will **wait** until thread has called exit and then get return val

pthread_mutex_t

- pthread_mutex_init(pthread_mutex_t **mutex*, const pthread_mutexattr_t **attr);*
  - Initialize the mutex
- pthread_mutex_destroy(pthread_mutex_t **mutex*);
  - Destroy the mutex
- int pthread_mutex_lock(pthread_mutex_t **mutex*);
  - Lock the mutex, only owner can unlock
- int pthread_mutex_unlock(pthread_mutex_t **mutex*);
  - Unlock mutex

pthread_semaphore_t

- int sem_init(sem_t * *sem*, int *pshared*, unsigned int *value*);
  - Initialize semaphore *sem* to *value*
  - *pshared* flag is **1** for between process semaphore and **0** for thread-access only
- int sem_destroy(sem_t **sem*);
  - Delete the semaphore and mark memory for reallocation
- int sem_post(sem_t **sem*);
  - Signal, add one to the value of semaphore *value*
- int sem_wait(sem_t **sem*);
  - if *value* > *0,* then *value--*
  - if *value* == *0,* wait

pthread_cond_t

- int pthread_cond_wait(pthread_cond_t **cond*, pthread_mutex_t **mutex*);
  - Wait for a signal on the *cond* using *mutex*
- int pthread_cond_signal(pthread_cond_t **cond*);
  - Wakes up one thread arbitrarily that is waiting on *cond*
- int pthread_cond_broadcast(pthread_cond_t **cond*);
  - Wakes up **all** threads that are waiting on *cond*

References

- Tannenbaum, Bos “Modern Operating Systems” Fourth Edition
  - Chapter 2
- Abraham, Silberschatz “Operating System Concepts” 9th Edition
  - Chapter 4
- https://en.wikipedia.org/wiki/Amdahl's_law#/media/File:AmdahlsLaw.svg
- Tomikj, Nikola, and Marjan Gusev. "Parallel Matrix Multiplication." *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (2018): 0204-209. Web. https://doi.org/10.23919/MIPRO.2018.8400039
- Gusev, Marjan, and Sasko Ristov. "A Superlinear Speedup Region for Matrix Multiplication." Concurrency and Computation: Practice and Experience 26.11 (2014): 1847-868. Web. https://doi-org.ezproxy.library.unlv.edu/10.1002/cpe.3102
- https://spectrum.ieee.org/computing/hardware/gordon-moore-the-man-whose-name-means-progress