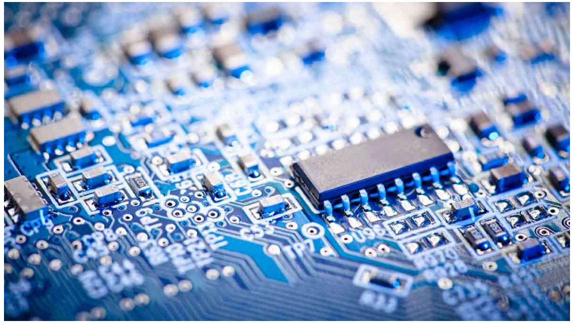
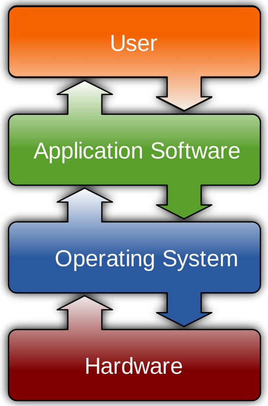
Please do not redistribute these slides without prior written permission



CS 3650 Computer Systems

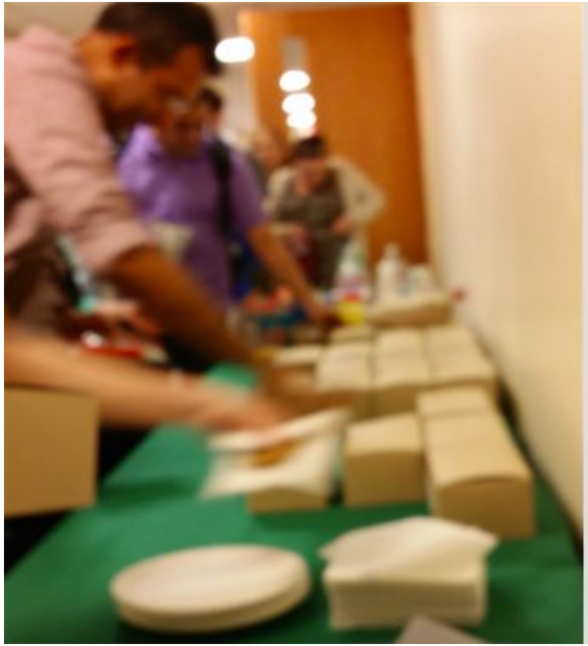
Alden Jackson / Ferdinand Vesely



Intro	Virtualization		Concurrency	Persistence	Appendices
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	*Dialogue*
TOC	4 Processes	13 Address Spaces	26 Concurrency and Threads code	36 I/O Devices	Virtual Machines
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API	37 Hard Disk Drives	*Dialogue*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks	38 Redundant Disk Arrays (RAID)	Monitors
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	*Dialogue*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables	40 File System Implementation	Lab Tutorial
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores	41 Fast File System (FFS)	Systems Labs
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	xv6 Labs
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	
		21 Swapping: Mechanisms	34 *Summary*	44 Flash-based SSDs	
		22 Swapping: Policies		45 Data Integrity and Protection	
		23 Case Study: VAX/VMS		46 *Summary*	
		24 *Summary*		47 *Dialogue*	
				48 Distributed Systems	
				49 Network File System (NFS)	
				50 Andrew File System (AFS)	
				51 *Summary*	

Pre-Class Warmup

- Inefficient vs efficient (parallel) buffet



Course Logistics

- Make sure you are doing the readings on the syllabus
  - They will help prepare you!
- Masks
  - It’s your decision to wear or not wear a mask in class
  - I’m not going to wear one for lecture
- Lab: don’t forget to check in sbrk.c
-

C Corner

Assignment 7 Hint: Ways to find the data

```
typedef struct block{
    size_t size; // How many bytes beyond this block have been allocated in the heap
    struct block* next; // Where is the next block in your linked list
    int free; // Is this memory free?
    int debug; // (optional) Perhaps you can embed other information--remember, you are the boss!
  } block_t;
mymalloc(9)
....
ptr = sbrk(size_passed_to_mymalloc + sizeof(struct block));  \\ sizeof(struct block)) => BLOCK_SIZE
data = (struct block *) ptr + 1
\\ or
struct Slot { struct block header; char data[]; }
Slot* s = sbrk(size_passed_to_mymalloc+sizeof(struct block));
\\ s->header points to beginning of block , s->data points to what is returned to the caller
```

Lecture 9 - Concurrency

Concurrency

con·cur·rence

/kənˈkərəns/

noun

noun: **concurrency**

1. the fact of two or more events or circumstances happening or existing at the same time.

Concurrent thinking

- Humans tend to think sequentially
- Thinking about all the *potential sequences* of events is difficult for humans.
    - https://www.psychologicalscience.org/news/why-humans-are-bad-at-multitasking.html

  Computers on the other hand, can multi-task quite well.

Parallelism vs Concurrency (programming context)

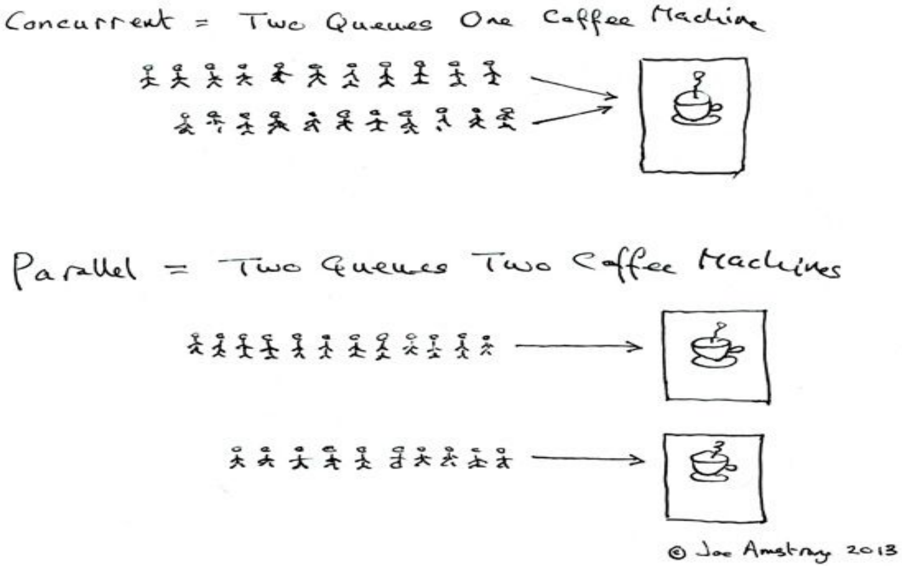
1. Concurrency Definition: Multiple things can happen at once, the order matters, and sometimes tasks have to wait on shared resources.
2. Parallelism Definition: Everything happens at once, instantaneously

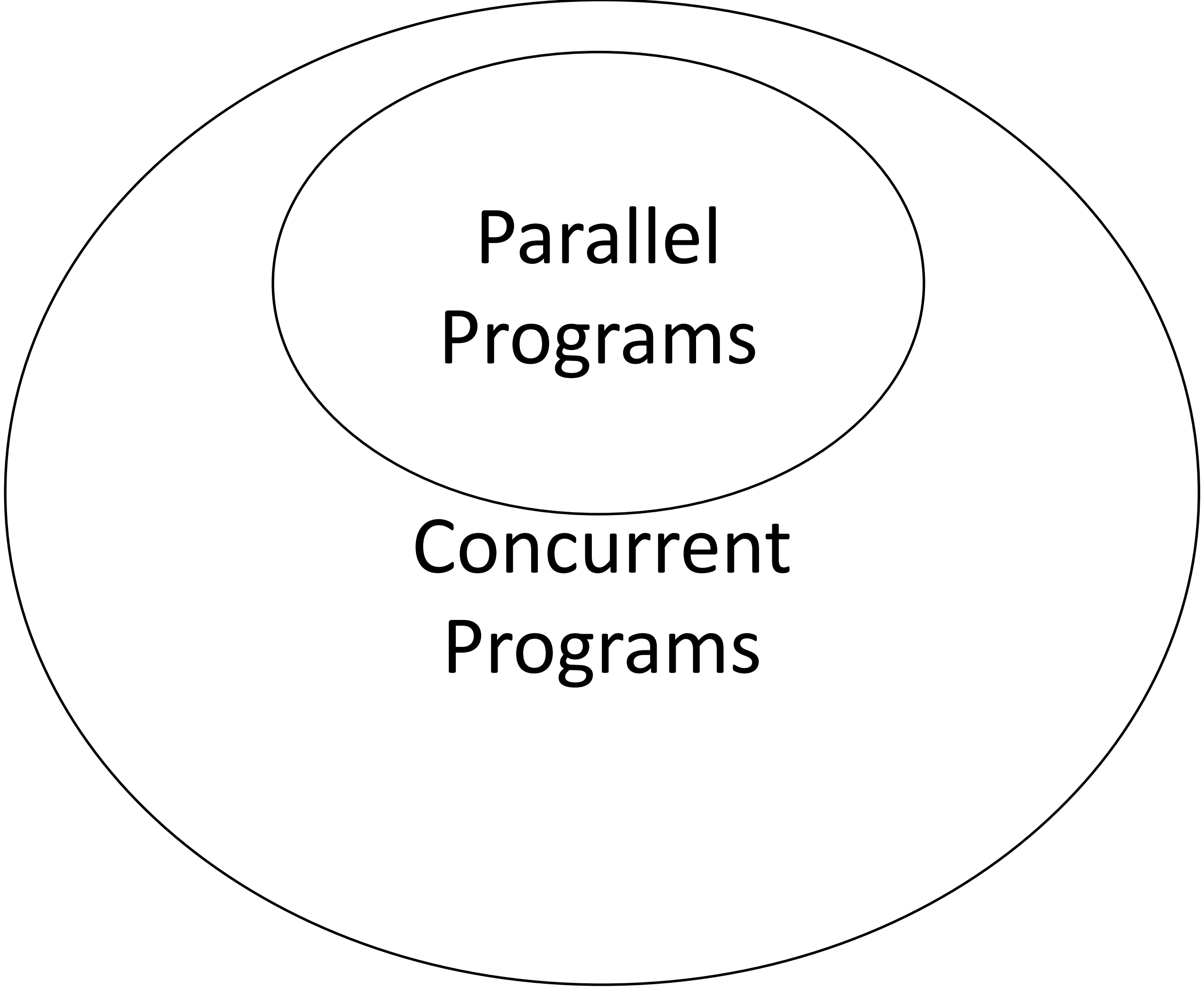
Parallelism vs Concurrency (programming context)

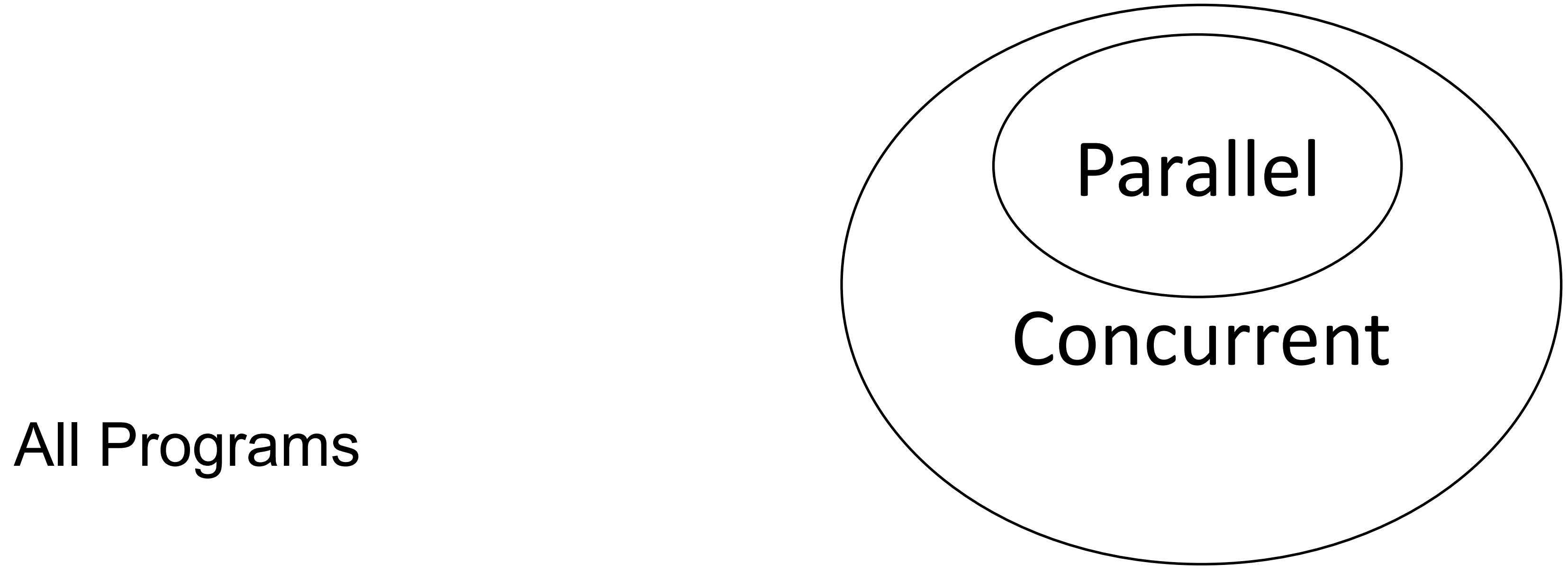
- Concurrency Definition: Multiple things can happen at once, **the order matters, and sometimes tasks have to wait on shared resources.**
- Parallelism Definition: Everything happens at once, instantaneously

Parallelism vs Concurrency (programming context)

- Concurrency Definition: Multiple things can happen at once, **the order matters, and sometimes tasks have to wait on shared resources**.
- Parallelism Definition: Everything happens at once, instantaneously

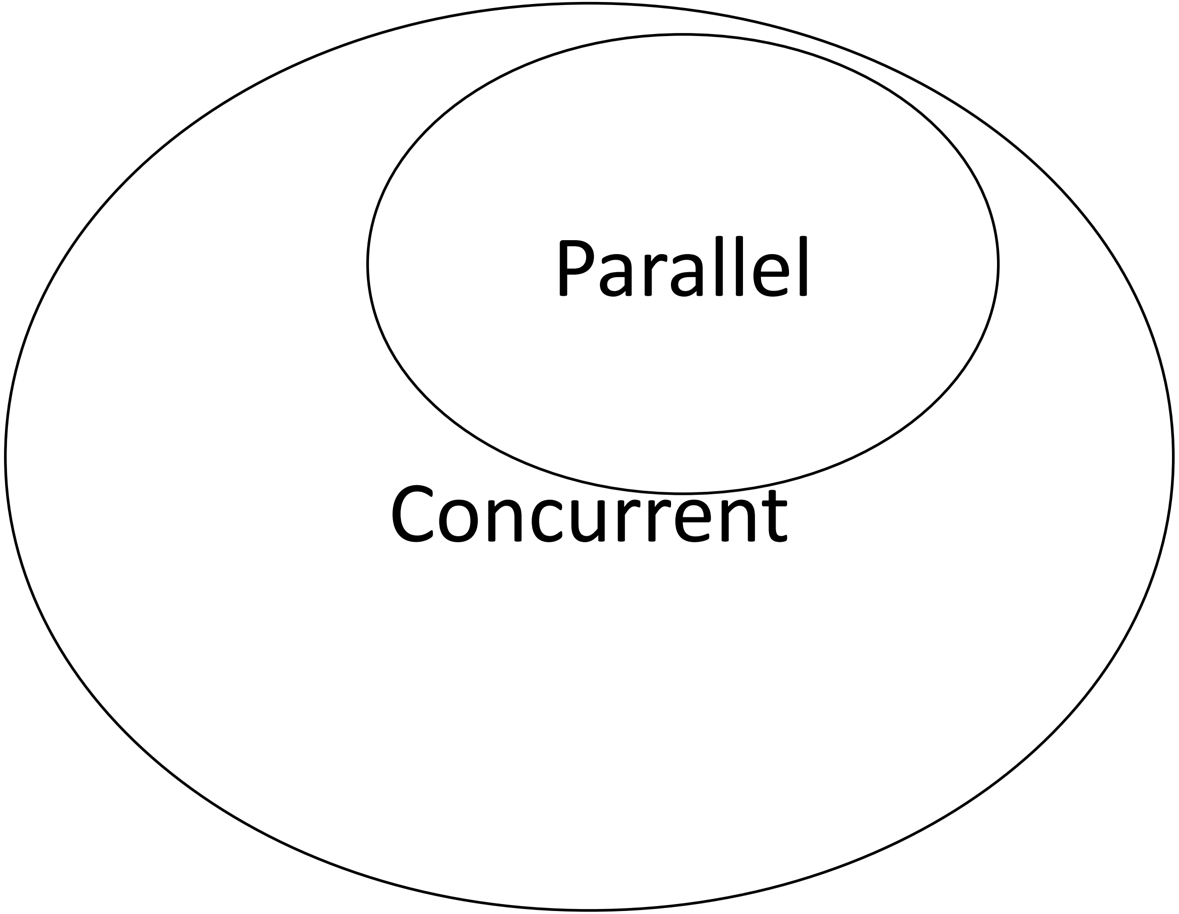



Parallel
Programs
Concurrent
Programs


Parallel
Concurrent
All Programs

All Programs in 10 years? (i.e. more concurrency & parallelism)

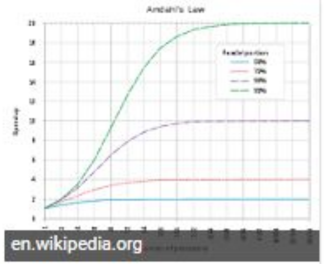
PS: this slide is old



Why Parallel?

- Performance (execution speed)
- But how much performance?

Amdahl's law is a formula used to find the maximum improvement possible by improving a particular part of a system. In parallel computing, **Amdahl's law** is mainly used to predict the theoretical maximum speedup for program processing using multiple processors. ... This term is also known as **Amdahl's** argument.



What is Amdahl's Law? - Definition from Techopedia
https://www.techopedia.com/definition/17035/amdahls-law

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

s = speedup of task that benefits from improved resources
p = portion of execution time benefiting from improved speedup

https://en.wikipedia.org/wiki/Amdahl%27s_law

Applied example: http://web.cs.iastate.edu/~prabhu/Tutorial/CACHE/CompPerf.pdf

Why Concurrency - it's necessary for good music

Good Concurrency = Good Conversation

Good Concurrency = Good Conversation

Concurrency

- In general, concurrency (like parallelism) is used because it is necessary for a system to function.
    - (For example, our jazz ensemble)

- It is also largely motivated by increased performance
    - *The potential* for more tasks to happen at once can thus increases performance (especially, if we have multiple cores on our machine)

Concurrency

- In general, concurrency (like parallelism) is used because it is necessary for a system to
  - (For ex
- It is also la
  - *The po* e (typically if we have m

Concurrency comes with some caveats however (next slide!)

Bad Concurrency = Data Race

- When two (or more) processes contending for one shared resource.

Bad Concurrency = Data Race

- When two (or more) processes contending for one shared resource.

One parking spot 2 cars want to acquire

Data race is not always as obvious...(1/4)

- Imagine you check your fridge and find there is no milk
  - So you run to the store

Data race is not always as obvious...(2/4)

- Imagine you check your fridge and find there is no milk
  - So you run to the store
- Then moments later your roommate checks the fridge and finds it is empty
  - So they run to the store

Data race is not always as obvious...(3/4)

- Imagine you check your fridge and find there is no milk
  - So you run to the store
- Then moments later your roommate checks the fridge and finds it is empty
  - So they run to the store
- Roommate # 3 comes and notices the same
  - ....

Data race is not always as obvious...(4/4)

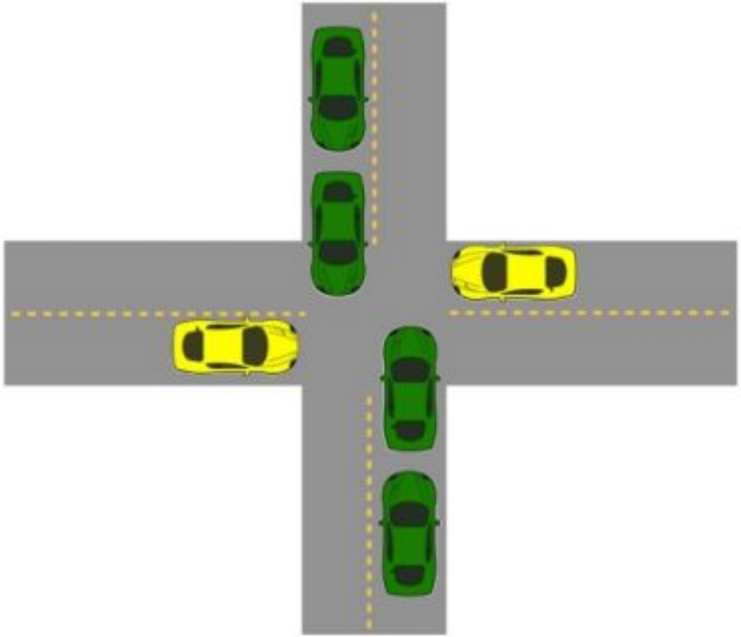
- You get the idea when you then find out you have 3 times as much milk as your house needs when everyone returns.

Bad Concurrency = Deadlock

- Grid lock in a traffic jam
- Each car prevents others from going through a shared resource (the intersection).
- (One car needs a piece of the intersection in order to move forward)

Bad Concurrency = Starvation

- Imagine a constant stream of green cars
- Progress is still being made by the green cars
- The yellow cars can never make progress to get across the street.
  - They are resource starved of a shared resource (again, they cannot cross the intersection)



Concurrent Programming takes some extra care

1. Races: Outcome depends on the arbitrary scheduling decisions elsewhere in the system
   - e.g. Who gets the last seat on the airplane. (soln's to this in Distributed Systems course)
2. Deadlock: Improper resource allocation prevents forward progress
   - e.g. traffic gridlock
3. Starvation/Fairness: External events and/or scheduling decisions can prevent sub-task progress
   - e.g. Someone jumping in front of you in line

- But regardless, concurrent programming is important and necessary to get the most out of current processor architectures!

A Few Approaches to Concurrency

- Process-Based
  - Fork() different processes
  - Each process has its own private address space
- Event-Based
  - Programmer manually interleaves multiple logical flows and polls for events
  - All flows share the same address space
  - Uses technique called I/O multiplexing
- Thread-based
  - Kernel automatically interleaves multiple logical flows
  - Each flow shares the same address space
  - Hybrid of process-based and event-based.

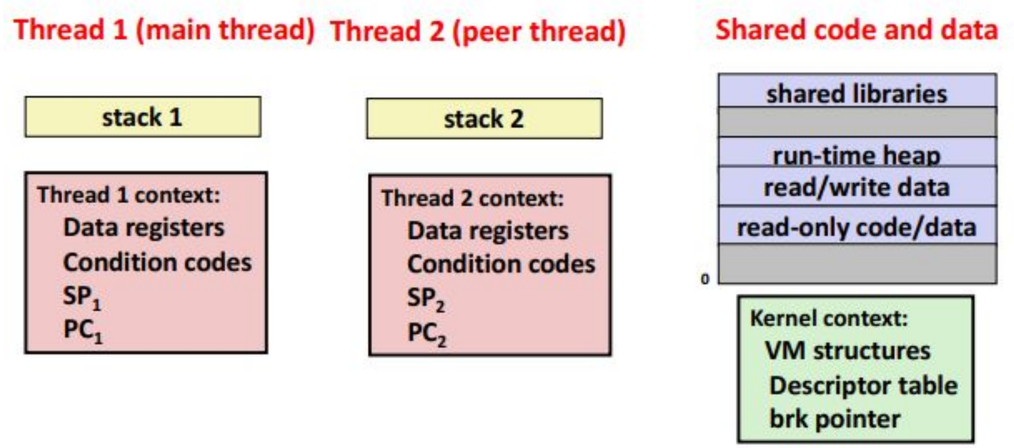
A Few Approaches to Concurrency

- Process-Based
  - Fork() different processes
  - Each process has its own private address space
- Event-Based
  - Programmer **manually** interleaves multiple logical flows and polls for events
  - All flows share the same address space
  - Uses technique called I/O multiplexing
- **Thread-based (Today's focus)**
  - **Kernel automatically interleaves multiple logical flows**
  - **Each flow shares the same address space**
  - **Hybrid of process-based and event-based.**

Threads

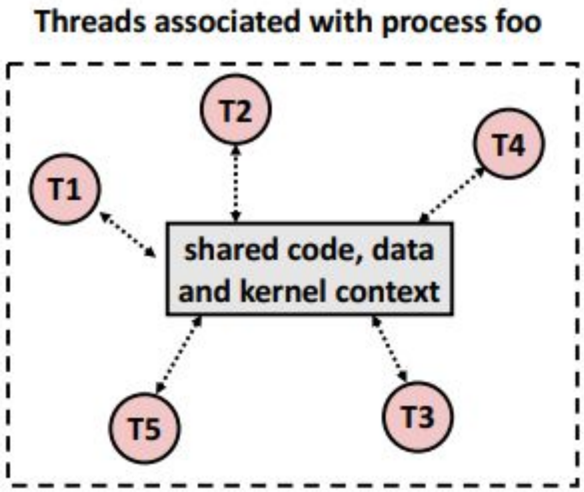
A Process can have Multiple Threads

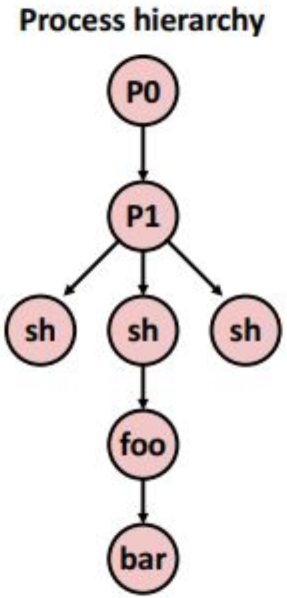
- Each thread shares the same code, data, and kernel context
- A thread has its own thread id (TID)
- A thread has its own logical control flow (no need to exec)
- A thread has its own stack for local variables



View of Threads

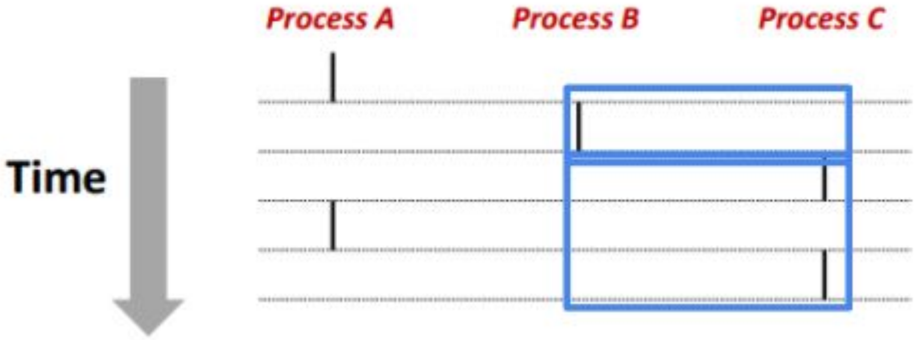
- Threads associated with a process form a “pool” of peers
  - Unlike processes (on the right) which form a tree hierarchy (i.e. parent/child relationship)





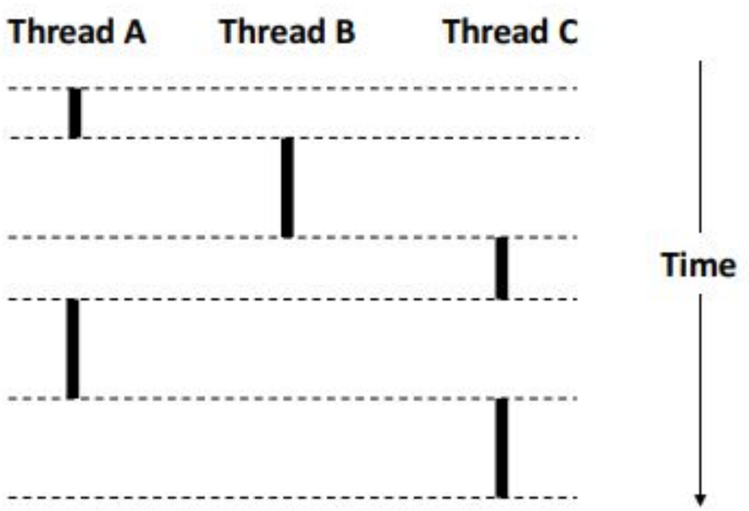
Remember this diagram on Concurrent Processes?

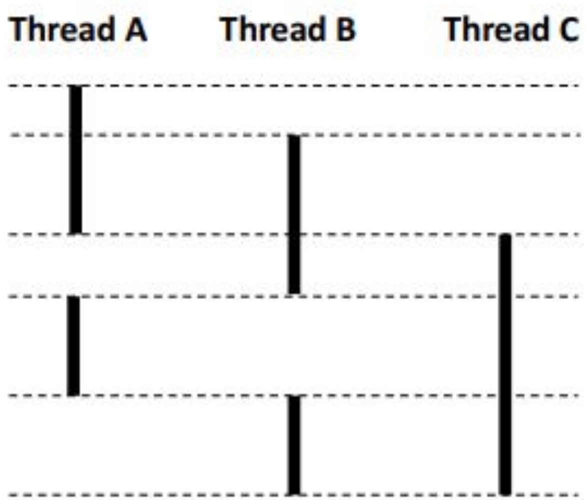
- We looked at multiple processes running on a single core (next slide for multiple cores)
  - On a single core, which processes here are concurrent relative to each other?
    - Concurrent: A&B, A&C
  - Which are sequential?
    - Sequential: B &C



Concurrent Thread (or Process) Execution

- Single Core Process
  - Simulate parallelism by time slicing
- Multi-Core Processor
  - Can have true parallelism
  - Note the longer durations of time spent on each thread without being divided up





Threads vs Processes

- Similarities
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores if available)
  - Each is context switched
- Differences
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not (i.e. fork makes a copy)
  - Threads are usually less expensive than managing processes
    - Process control (creating and reaping) twice as expensive as thread control
      - Linux estimates
        - ~20k cycles to create and reap a process
        - ~10k cycles to create and reap a thread

Posix Threads API (PThreads Interface)

- Known as Pthreads (pronounced as "*p-thread*")
  - Standard of functions for manipulating threads from C Programs
- Sample functions
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
    - `exit()` - Terminates all threads
    - `return` - terminates current thread
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_lock` and `pthread_mutex_unlock`

PThread examples

Hello Thread

- (thread1.c)
- The thread that is “launched” is a function in the program

```c
// Compile with:
//
// clang -lpthread thread1.c -o thread1
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Thread with variable arguments
void *thread(void *vargp){
        printf("Hello from thread\n");
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tid;
        // Create and execute the thread
        pthread_create(&tid, NULL, thread, NULL);
        // Wait in 'main' thread until thread executes
        pthread_join(tid,NULL);
        // end program
        return 0;
}
```

Hello Thread

- (thread1.c)
- The thread that is “launched” is a function in the program
    - This is done when the thread is created
    - Different attributes can be sent to threads (in this case the first NULL)
    - Arguments can also be passed to the function (second NULL)

```c
// Compile with:
//
// clang -lpthread thread1.c -o thread1
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Thread with variable arguments
void *thread(void *vargp){
        printf("Hello from thread\n");
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tid;
        // Create and execute the thread
        pthread_create(&tid, NULL, thread, NULL);
        // Wait in 'main' thread until thread executes
        pthread_join(tid,NULL);
        // end program
        return 0;
}
```

Hello Thread

- (thread1.c)
- The thread that is "launched" is a function in
    - This is
    - Differen threads
    - Argume function (second NULL)

```
// Compile with:
//
// clang -lpthread thread1.c -o thread1
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

");

        pthread_t tid;
        // Create and execute the thread
        pthread_create(&tid, NULL, thread, NULL)
        // Wait in 'main' thread until thread executes
        pthread_join(tid,NULL);
        // end program
        return 0;
}
```

```
-bash-4.2$ ./thread1
Hello from thread
```

Hello Thread

```
-bash-4.2$ ./thread1
Hello from thread
```

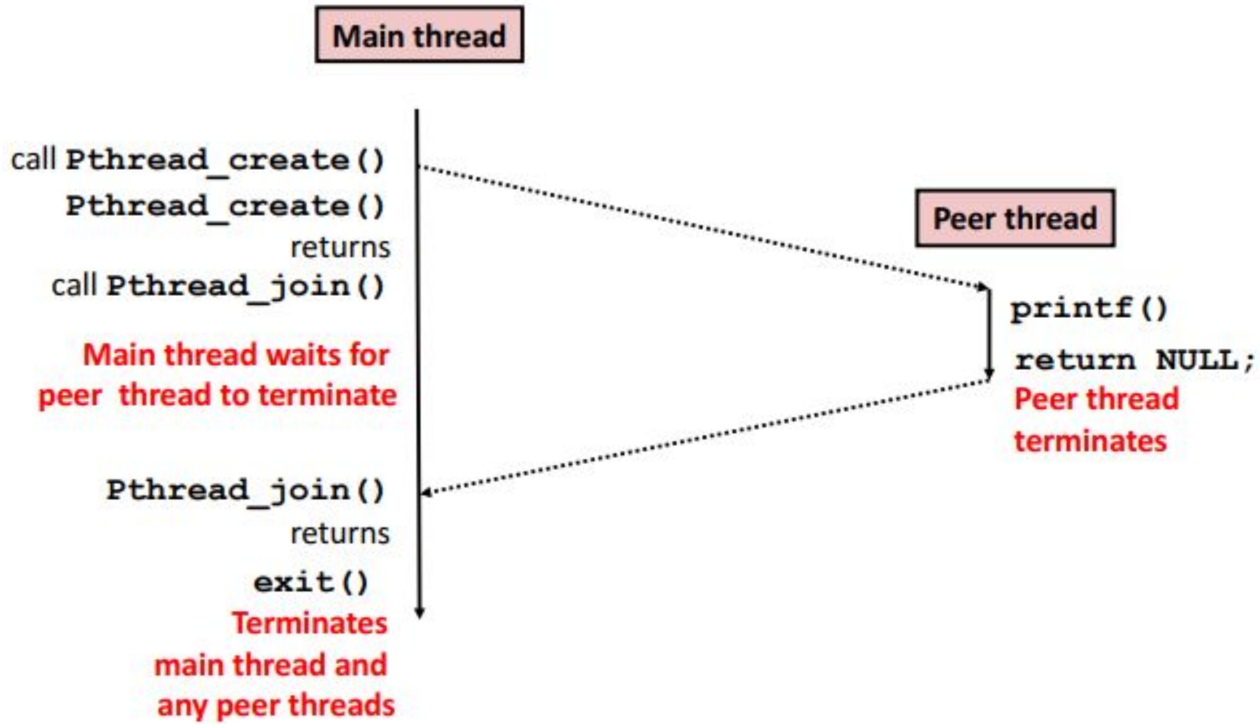
- (thread1.c)
- The thread that is “launched” is a function in the program
  - This is done when the thread is created
  - Different attributes can be sent to threads (in this case the first NULL)
  - Arguments can also be passed to the function (second NULL)
- `pthread_join` is the equivalent to “`wait`” for threads

```c
// Compile with:
//
// clang -lpthread thread1.c -o thread1
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Thread with variable arguments
void *thread(void *vargp){
        printf("Hello from thread\n");
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tid;
        // Create and execute the thread
        pthread_create(&tid, NULL, thread, NULL);
        // Wait in 'main' thread until thread executes
        pthread_join(tid,NULL);
        // end program
        return 0;
}
```

Visual execution of “Hello Thread”



Launching multiple threads

- (thread2.c)
- Store 10 thread ids.

```c
// Compile with:
//
// clang -lpthread thread2.c -o thread2
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10

// Thread with variable arguments
void *thread(void *vargp){
        printf("Hello from thread %ld\n", pthread_self());
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Main thread id: %ld\n",pthread_self());
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Make main wait for each thread
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Main thread returns: %ld\n",pthread_self());
        // end program
        return 0;
}
```

Launching multiple threads

- (thread2.c)
- Launch 10 threads

```c
// Compile with:
//
// clang -lpthread thread2.c -o thread2
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10

// Thread with variable arguments
void *thread(void *vargp){
        printf("Hello from thread %ld\n", pthread_self());
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Main thread id: %ld\n",pthread_self());
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Make main wait for each thread
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Main thread returns: %ld\n",pthread_self());
        // end program
        return 0;
}
```

Launching multiple threads

- (thread2.c)
- Launch 10 threads
- Print out their thread ids to show which thread is executing.

```c
// Compile with:
//
// clang -lpthread thread2.c -o thread2
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10

// Thread with variable arguments
void *thread(void *vargp){
        printf("Hello from thread %ld\n", pthread_self());
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Main thread id: %ld\n",pthread_self());
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Make main wait for each thread
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Main thread returns: %ld\n",pthread_self());
        // end program
        return 0;
}
```

Launching multiple threads

- (thread2.c)
- Launch 10 threads
- Print out their thread ids to show which thread is executing.
- Join all of our threads with the main thread
    - (i.e. make the main thread wait until all 10 threads have executed.)

```c
// Compile with:
//
// clang -lpthread thread2.c -o thread2
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10

// Thread with variable arguments
void *thread(void *vargp){
        printf("Hello from thread %ld\n", pthread_self());
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Main thread id: %ld\n",pthread_self());
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Make main wait for each thread
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Main thread returns: %ld\n",pthread_self());
        // end program
        return 0;
}
```

Launching multiple threads

- (thread3.c)
- *New Program*

```c
// Compile with:
//
// clang -lpthread thread3.c -o thread3
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;

// Thread with variable arguments
void *thread(void *vargp){
        counter = counter +1;
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

Launching multiple threads

- (thread3.c)
- This time launch 10000 threads

```c
// Compile with:
//
// clang -lpthread thread3.c -o thread3
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;

// Thread with variable arguments
void *thread(void *vargp){
        counter = counter +1;
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

Launching multiple threads

- (thread3.c)
- This time launch 10000 threads
- counter is shared between threads

```
// Compile with:
//
// clang -lpthread thread3.c -o thread3
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;

// Thread with variable arguments
void *thread(void *vargp){
        counter = counter +1;
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

Launching multiple threads

- This time launch 10000 threads
- counter is shared between threads
- What is wrong with this program?

```
Counter starts at: 0
Final Counter value: 9998
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9998
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9997
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9999
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9997
```

```c
// Compile with:
//
// clang -lpthread thread3.c -o thread3
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;

// Thread with variable arguments
void *thread(void *vargp){
        counter = counter +1;
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

Synchronization of Threads

- Shared variables are thus handy for moving around data
- But if we do not share properly, we can have synchronization errors!
    - There is a solution however!
    - (recap below)

=

```
Counter starts at: 0
Final Counter value: 9998
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9998
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9997
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9999
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9997
```

Example with lock

(thread4.c)

```
// Compile with:
// clang -lpthread thread4.c -o thread4
// This program fixes a problem with thread3.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

// Thread with variable arguments
void *thread(void *vargp){
        pthread_mutex_lock(&mutex1);
                counter = counter +1;
        pthread_mutex_unlock(&mutex1);
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }

        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }
        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

Example with lock

- Included a pthread_mutex_lock

```
// Compile with:
// clang -lpthread thread4.c -o thread4
// This program fixes a problem with thread3.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

// Thread with variable arguments
void *thread(void *vargp){
        pthread_mutex_lock(&mutex1);
                counter = counter +1;
        pthread_mutex_unlock(&mutex1);
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }

        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }
        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

Example with lock

- Included a pthread_mutex_lock
- lock and unlock protect
- Locks in other words enforce, that we have exclusive access to a region of code.

```c
// Compile with:
// clang -lpthread thread4.c -o thread4
// This program fixes a problem with thread3.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

// Thread with variable arguments
void *thread(void *vargp){
        pthread_mutex_lock(&mutex1);
                counter = counter + 1;
        pthread_mutex_unlock(&mutex1);
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }

        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }
        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

Example with lock

- Included a pthread_mutex_lock
- lock and unlock protect
- Locks in other words enforce, that we have exclusive access to a region of code.

```
mike:8$ gcc thread4.c -o thread4 -lpthread
mike:8$ ./thread4
Counter starts at: 0
Final Counter value: 10000
mike:8$ ./thread4
Counter starts at: 0
Final Counter value: 10000
mike:8$ ./thread4
Counter starts at: 0
Final Counter value: 10000
mike:8$ ./thread4
Counter starts at: 0
^[[AFinal Counter value: 10000
```

```c
// Compile with:
// clang -lpthread thread4.c -o thread4
// This program fixes a problem with thread3.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

// Thread with variable arguments
void *thread(void *vargp){
        pthread_mutex_lock(&mutex1);
                counter = counter +1;
        pthread_mutex_unlock(&mutex1);
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }

        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }
        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

Example with lock

- Also, don’t forget to join!

```c
// Compile with:
//
// clang -lpthread thread4_fixed.c -o thread4_fixed
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

// Thread with variable arguments
void *thread(void *vargp){
        pthread_mutex_lock(&mutex1);
                counter = counter +1;
        pthread_mutex_unlock(&mutex1);
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }

        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }
        printf("Final Counter Value: %d\n",counter);
        // end program
        return 0;
}
```