CS444/544
Operating Systems II

Lecture 12

Multi-threading and Synchronization

5/15/2024

Acknowledgement: Slides drawn heavily from Yeongjin Jiang

Oregon State University

Odds and Ends

- Lab 4 posted
- Lab 2 grades posted
- Lab 3 due Monday (5/20) midnight

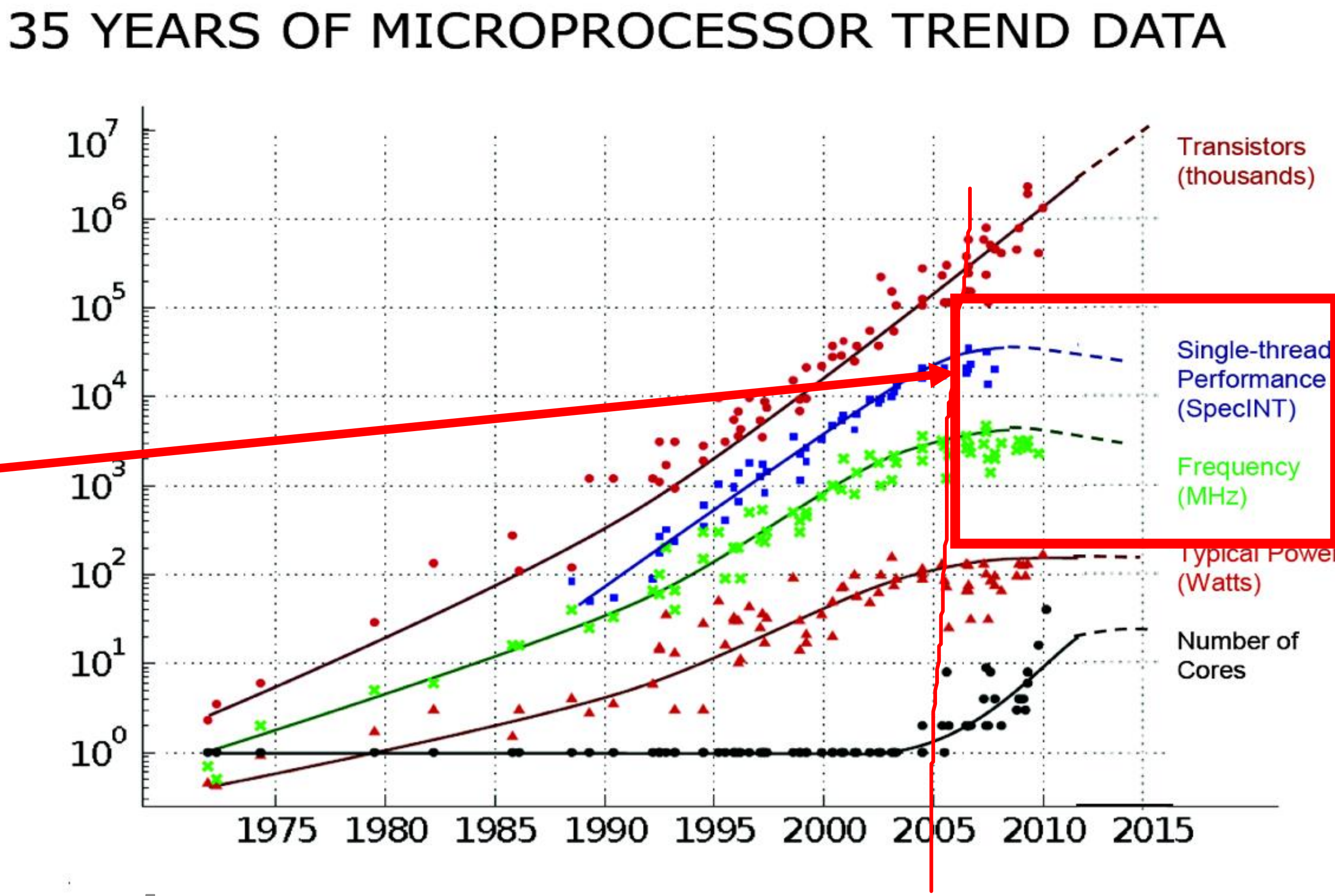
Quiz 2

Process/Thread/Synchronization

- We will learn:
  - Why concurrency is useful?
  - Differences between Process and Thread
  - Data racing issue
  - Synchronization (Mutual Exclusion)

Single-threaded CPU Performance

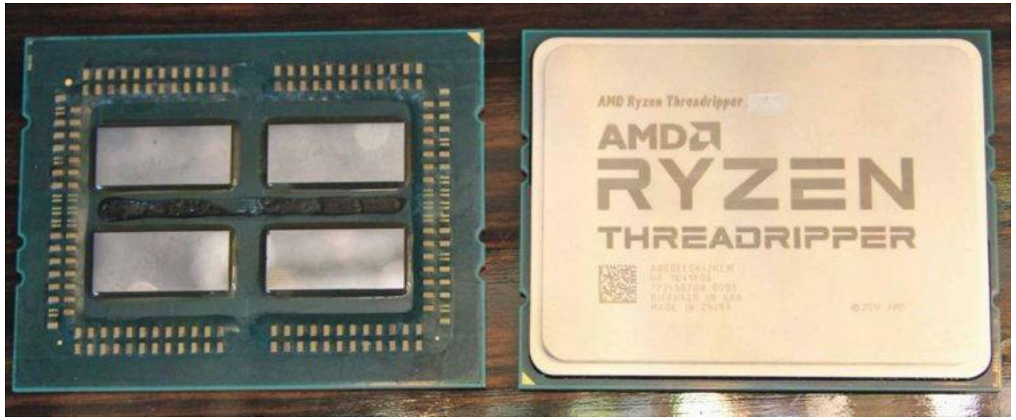
- # of transistors
  - Increasing linearly
- Performance
  - **Not increasing linearly…**

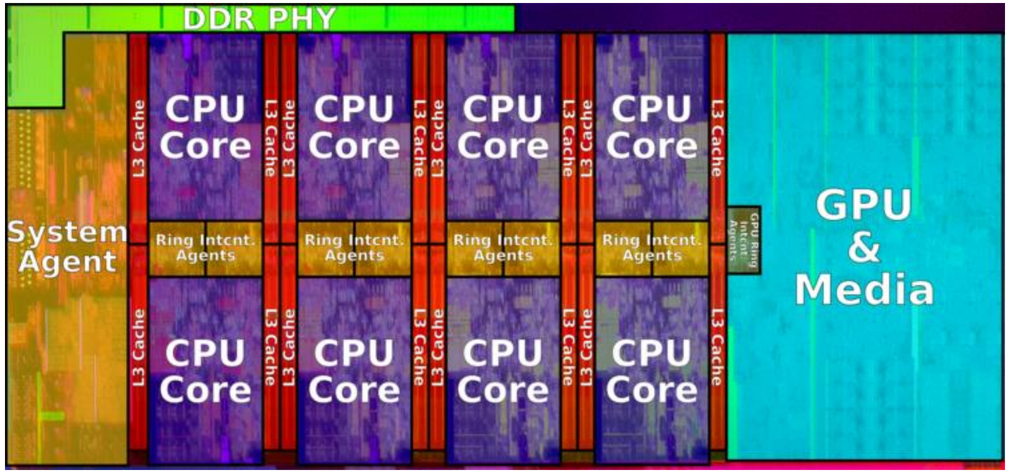


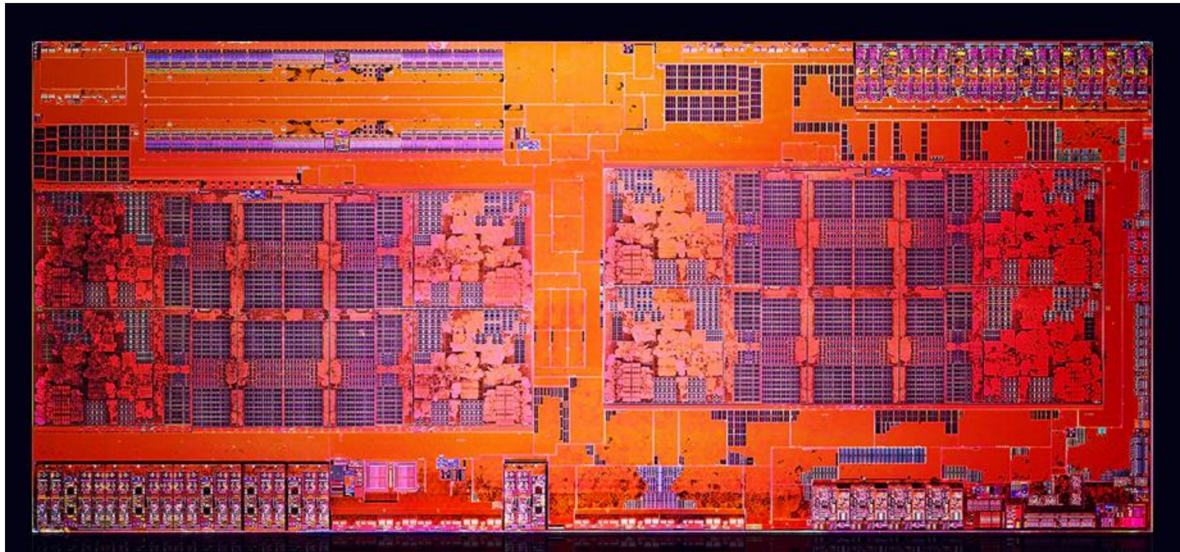
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

CPU Speed Capped by Frequency/Power

- How to get a better performance?

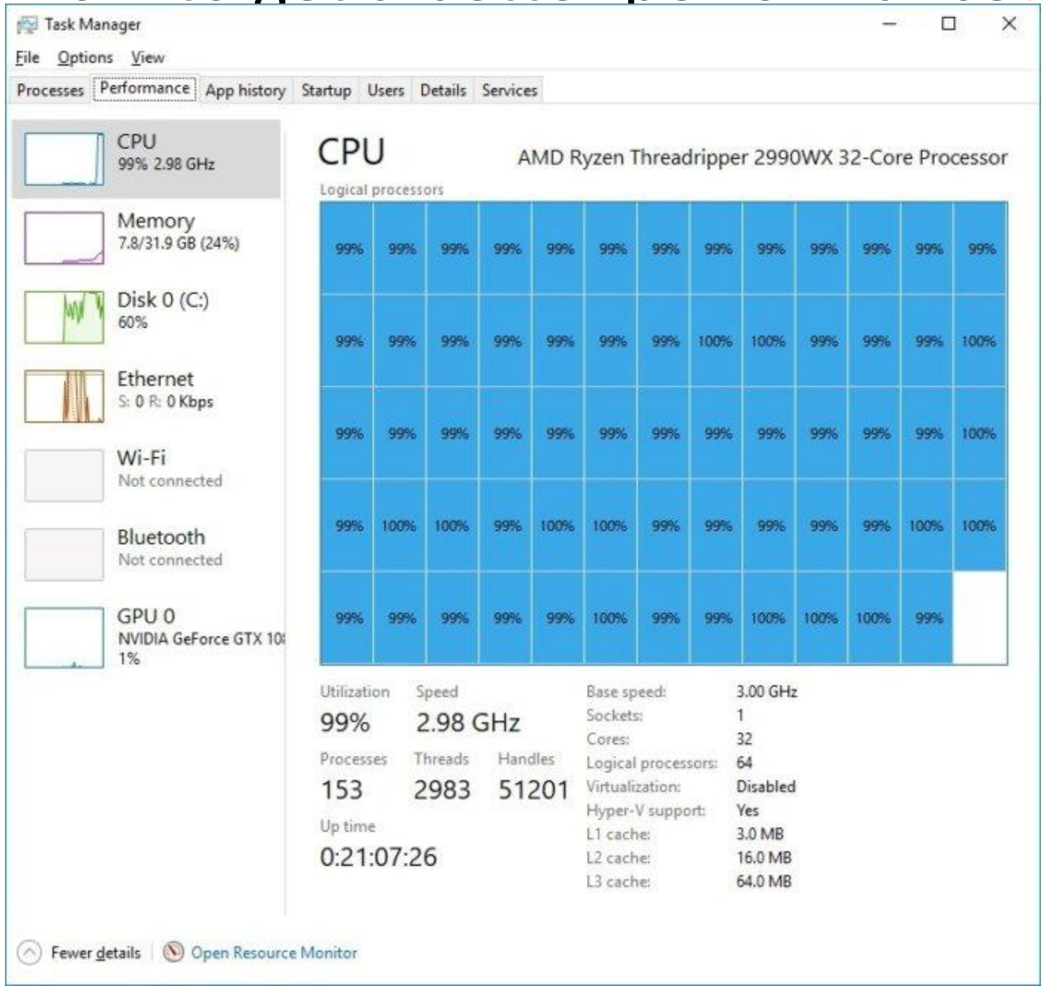


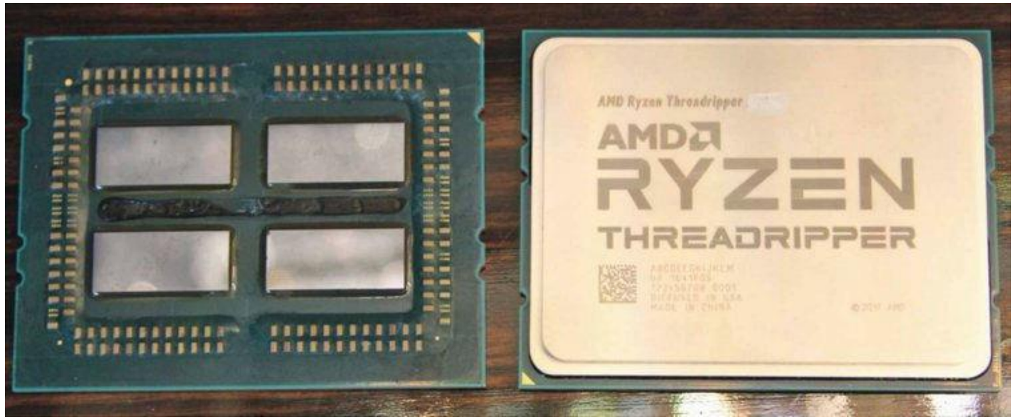


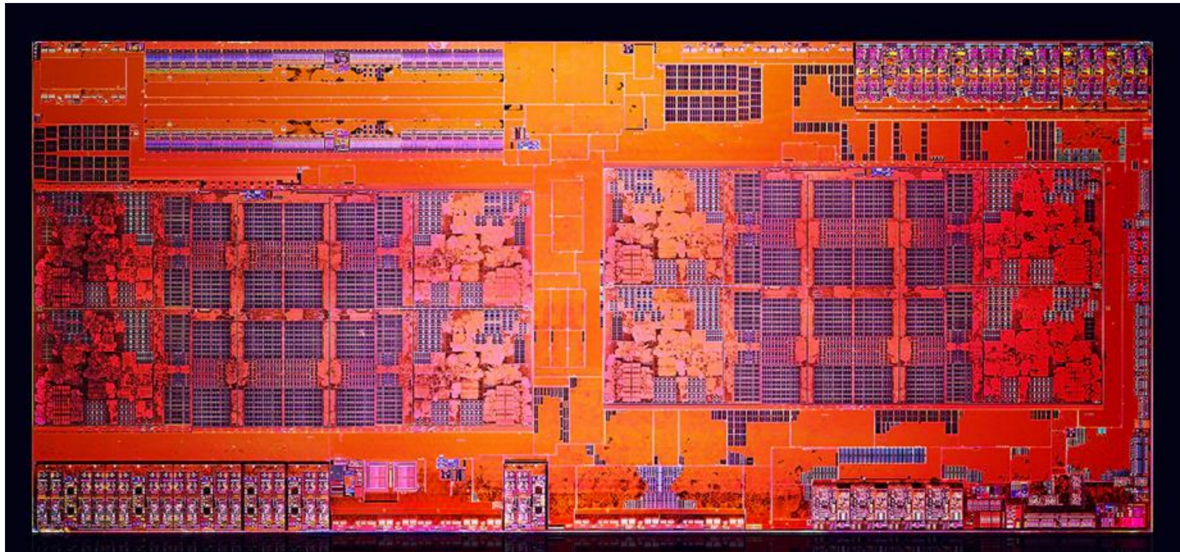


CPU Speed Capped by Frequency/Power

- How to get a better performance?

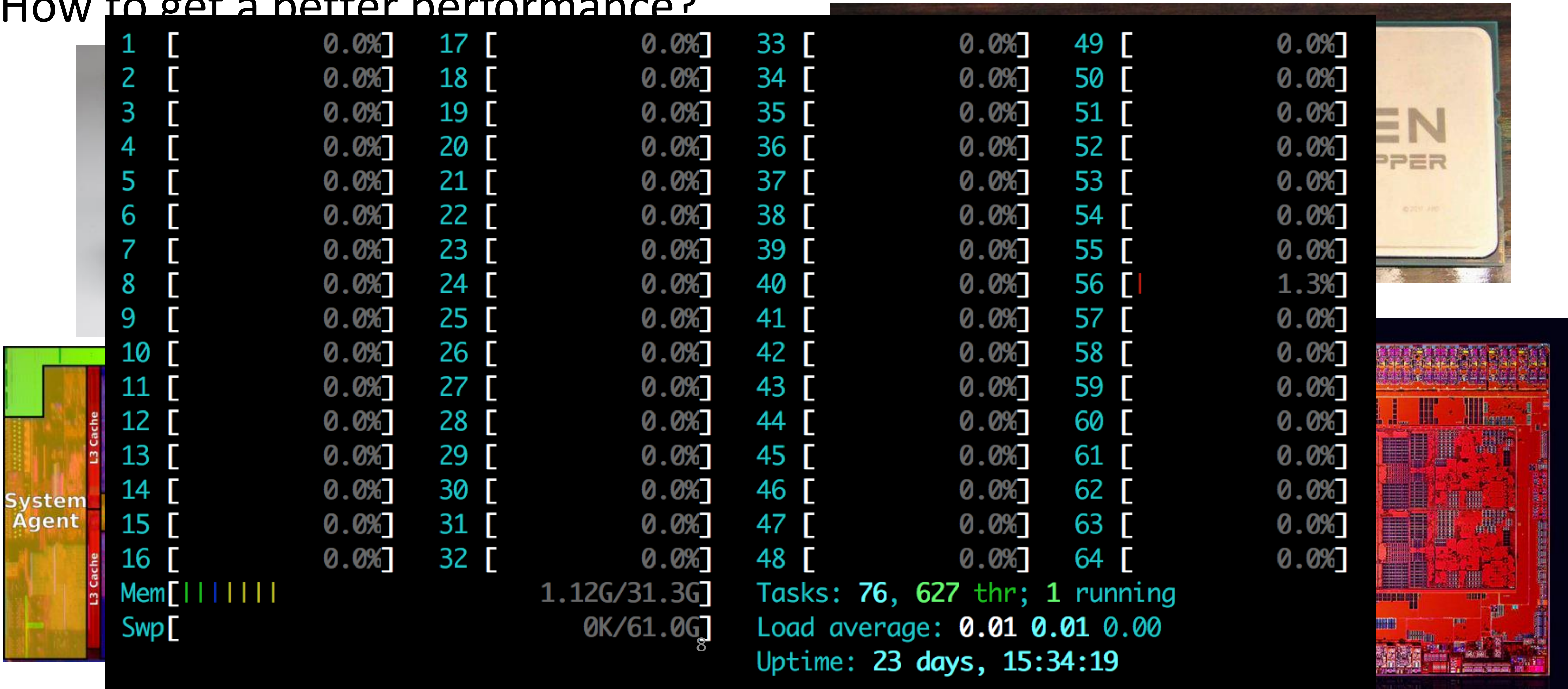






CPU Speed Capped by Frequency/Power

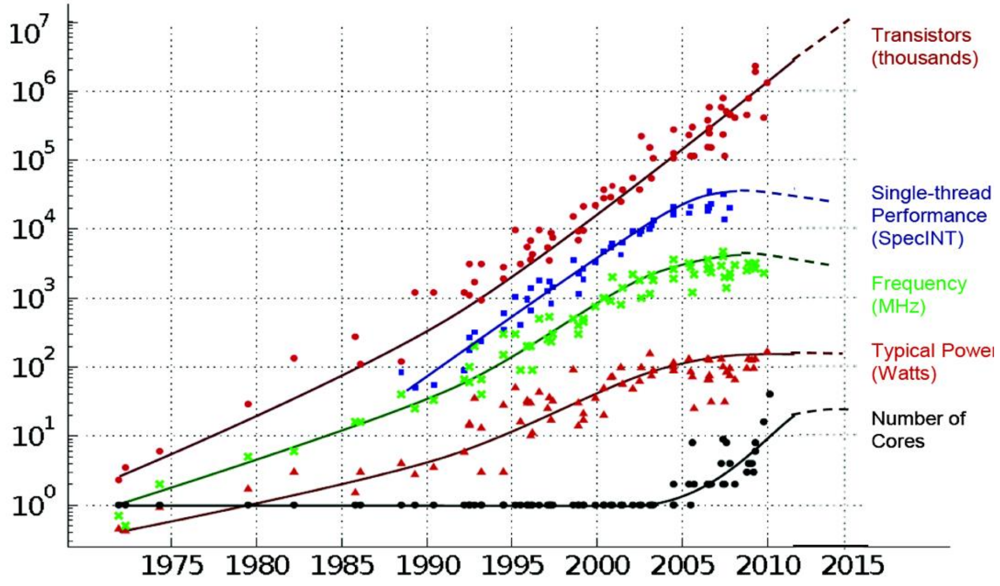
- How to get a better performance?



Motivation for Concurrency

- Trend in CPU
  - Same clock speed, more CPU cores
- Increase System Performance
  - Run many jobs at the same time to fully utilize multiple cores
- How to increase application performance?
  - Run multiple functions as separate jobs at the same time!
  - Processes, Threads, etc…

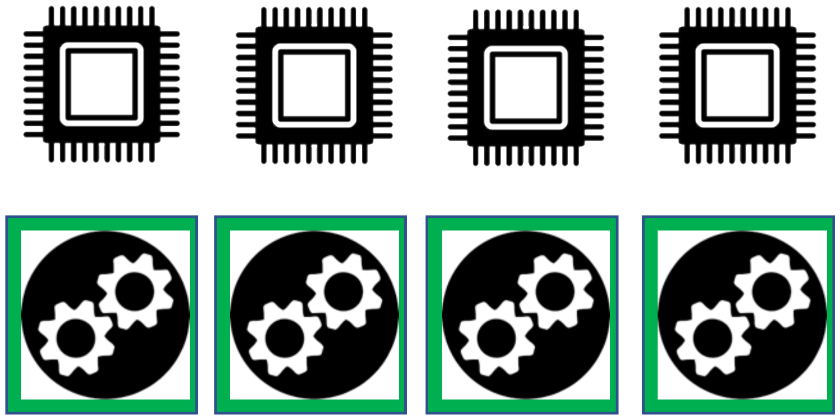
35 YEARS OF MICROPROCESSOR TREND DATA



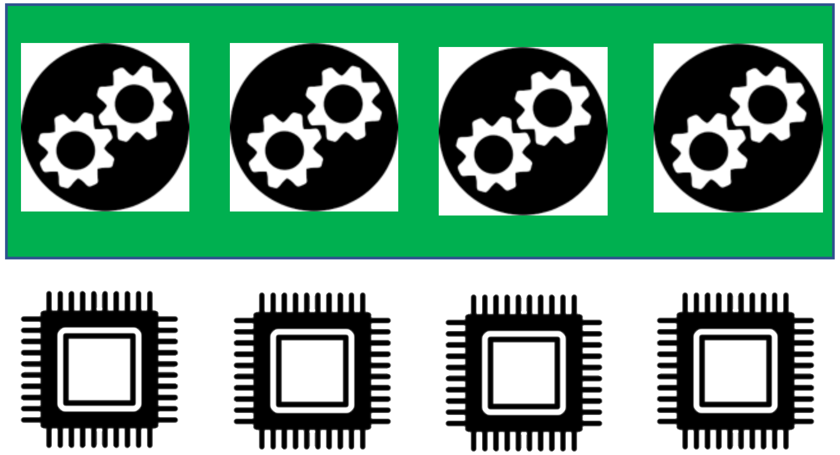
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Options for Concurrency

- Process
  - Run program as a separate instance

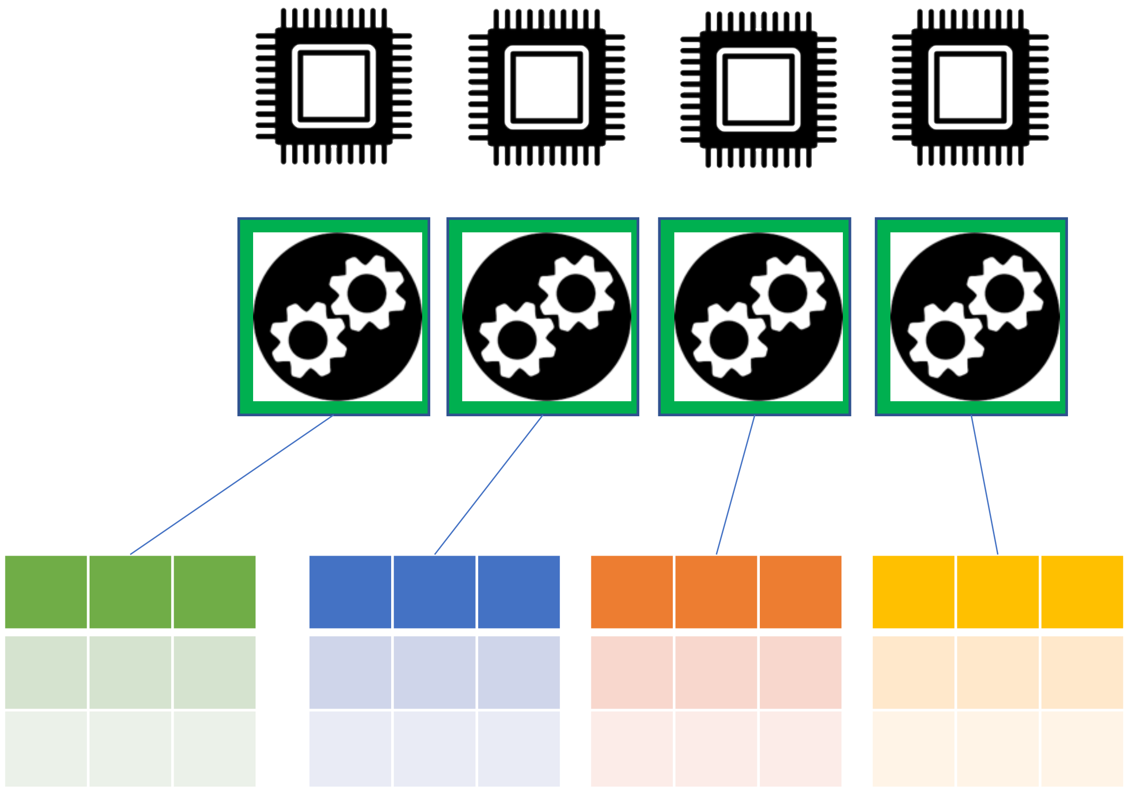


- Thread
  - Run program as a same instance

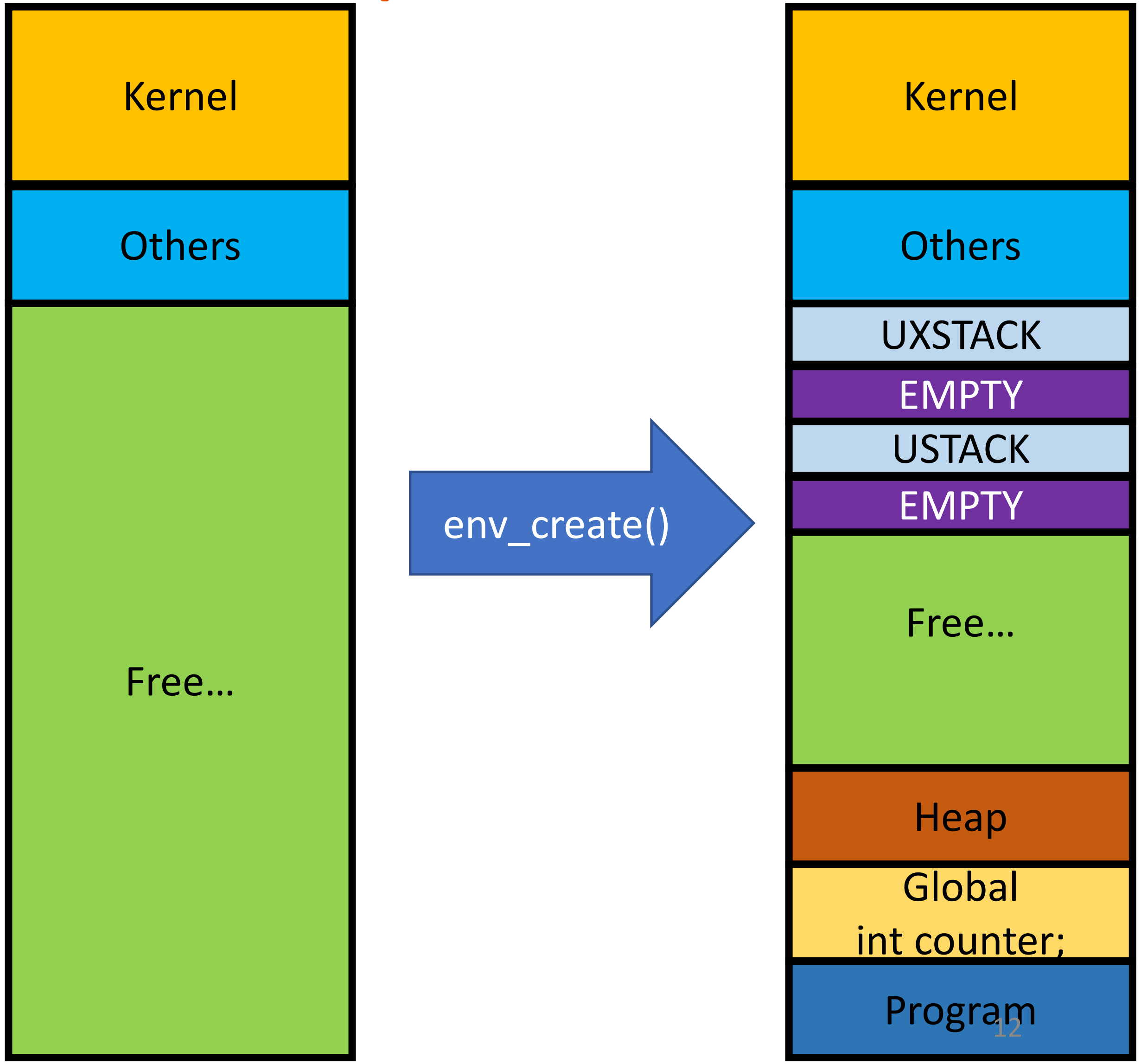


Process

- Each execution runs in an isolated environment
- Does not share memory space
  - Each has own page table
- Requires Inter-Process Communication for data sharing
  - File(), Pipe(), socket(), shared memory, etc..



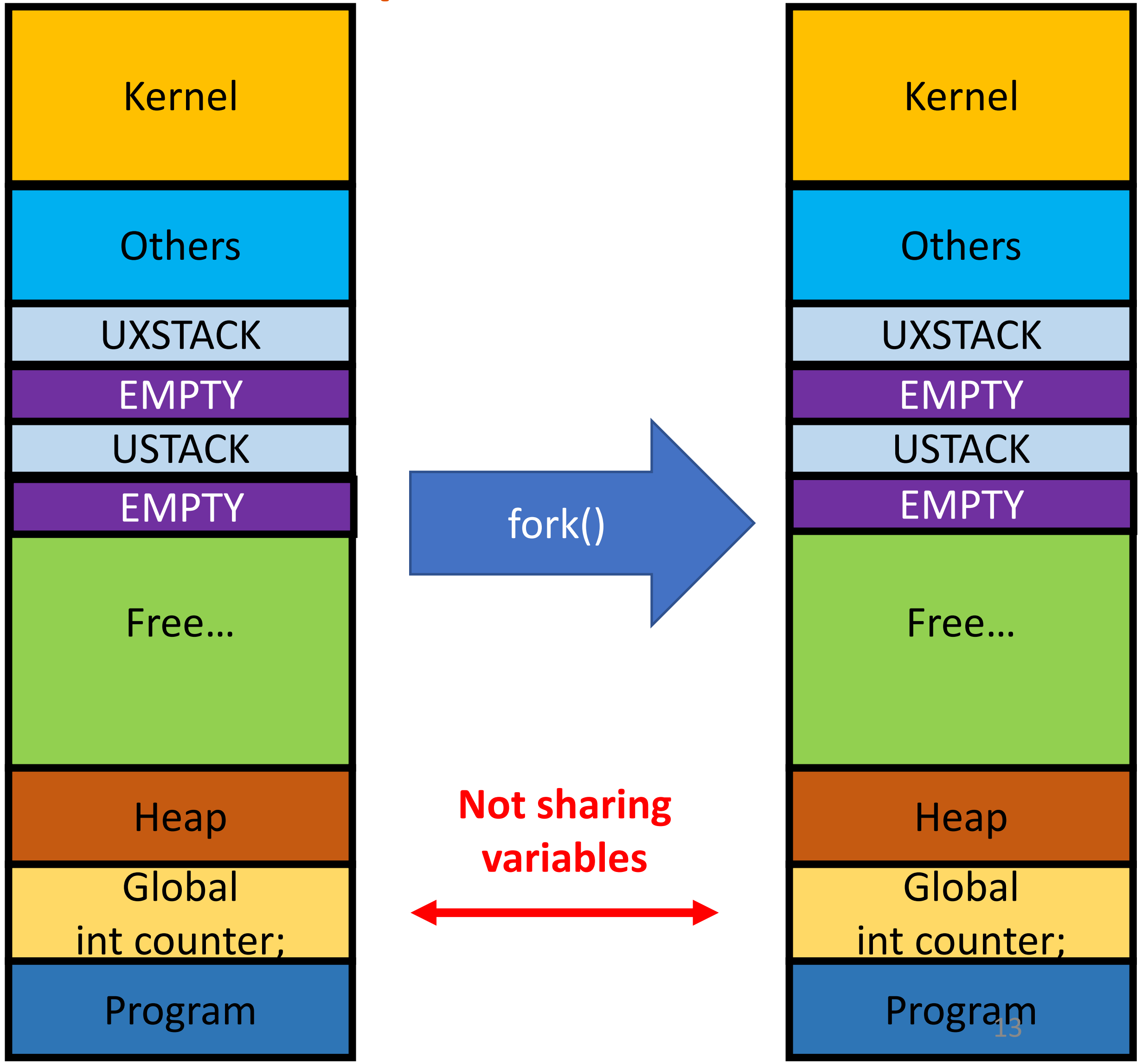
Process (Environment in JOS)



Process creates a new PRIVATE memory space

Parent **Child**

Process (Environment in JOS)



Fork() creates new process by copying memory space
Process creates a new PRIVATE memory space

```
#include <stdio.h>
#include <unistd.h>

int counter;
volatile int value = 1;

void countup() {
    for(int i=0; i<1000000; ++i) {
        counter += value;
    }
}

int main() {
    pid_t pid = fork();
    countup();
    printf("%s: %d\n", pid ? "Parent" : " Child", counter);
}
```

```
Parent: 1000000
 Child: 1000000
```

Process (Pros/Cons)

- Pros
  - Do not have to modify program to achieve parallelism
    - Just run multiple instances, or fork()!
- Cons
  - Use some additional memory to run same programs
    - Any write will incur memory duplication even in CoW fork()
  - Cannot directly read memory of other processes
    - Inter-process Communication (IPC) is available, but slow
- Use
  - Suitable for parallel ‘isolated’ execution
  - Not suitable for parallel execution on shared data

Can We Share a Memory Space and Run Jobs in Parallel at the Same Time?

- Yes! Thread: here I am!

- What is a thread?
  - Process: creates a new PRIVATE memory space and run concurrently
  - Thread: creates a SHARED memory space and run concurrently

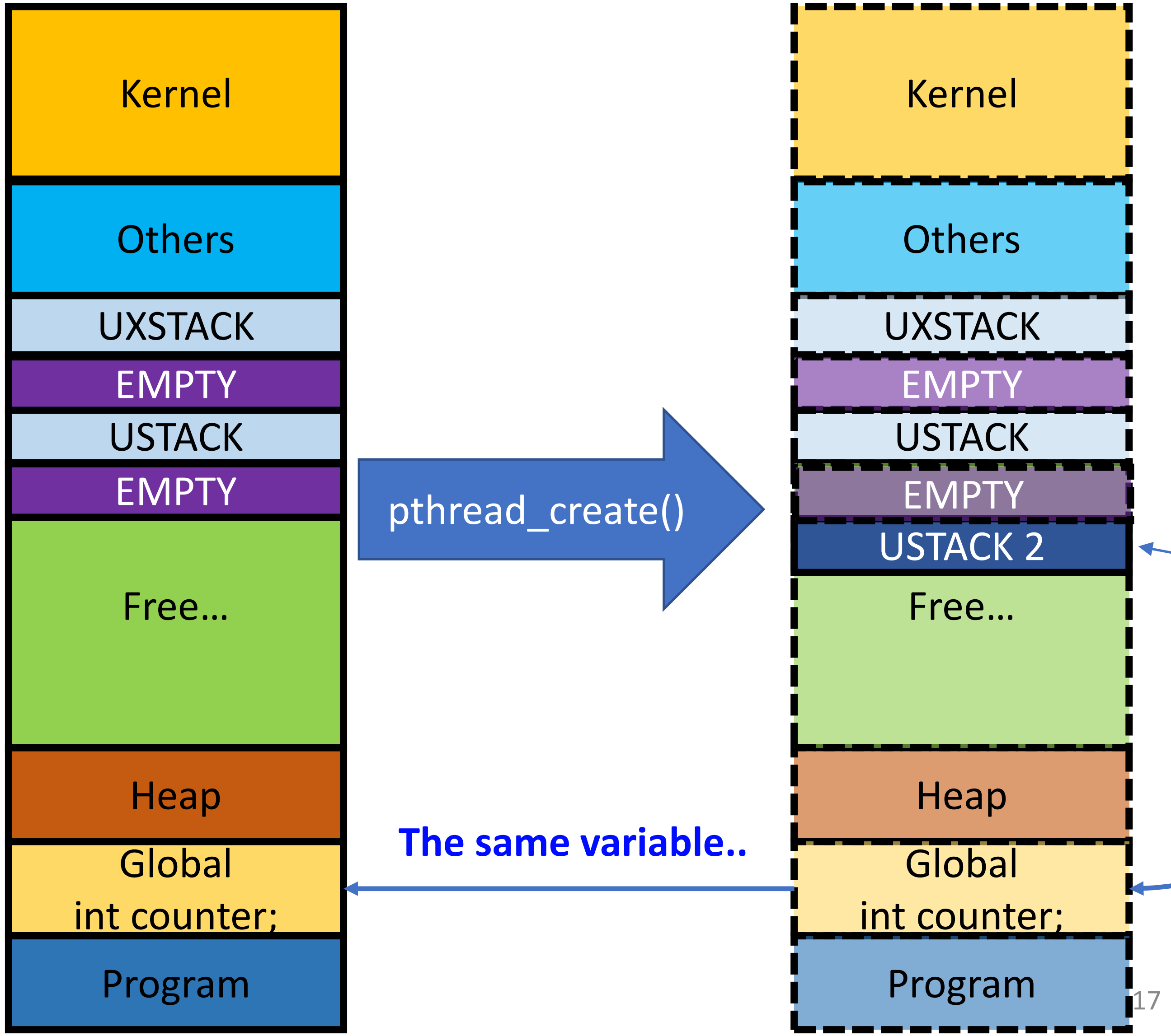
- SHARE?
  - Can access the same memory space, e.g., global variables, etc.

Thread: How Can We Share Memory Space Among Threads?

- Process Creation via Fork()
  - Naïve design: copy all physical pages, and create **a new page directory/table** that has the same virtual mapping (to new, corresponding physical pages)
  - Copy-on-write: do not copy all physical pages but keep the same mappings by read-only at the **new page directory/table** and provide a private copy when write on COW page occurs…

- Thread Creation
  - Get **a new execution environment**
  - Assign **the same page directory/table** (e.g., assign the same CR3)
  - Create **a new stack / storage for register context** to store execution context separately
    - Use less memory than fork()…

Thread

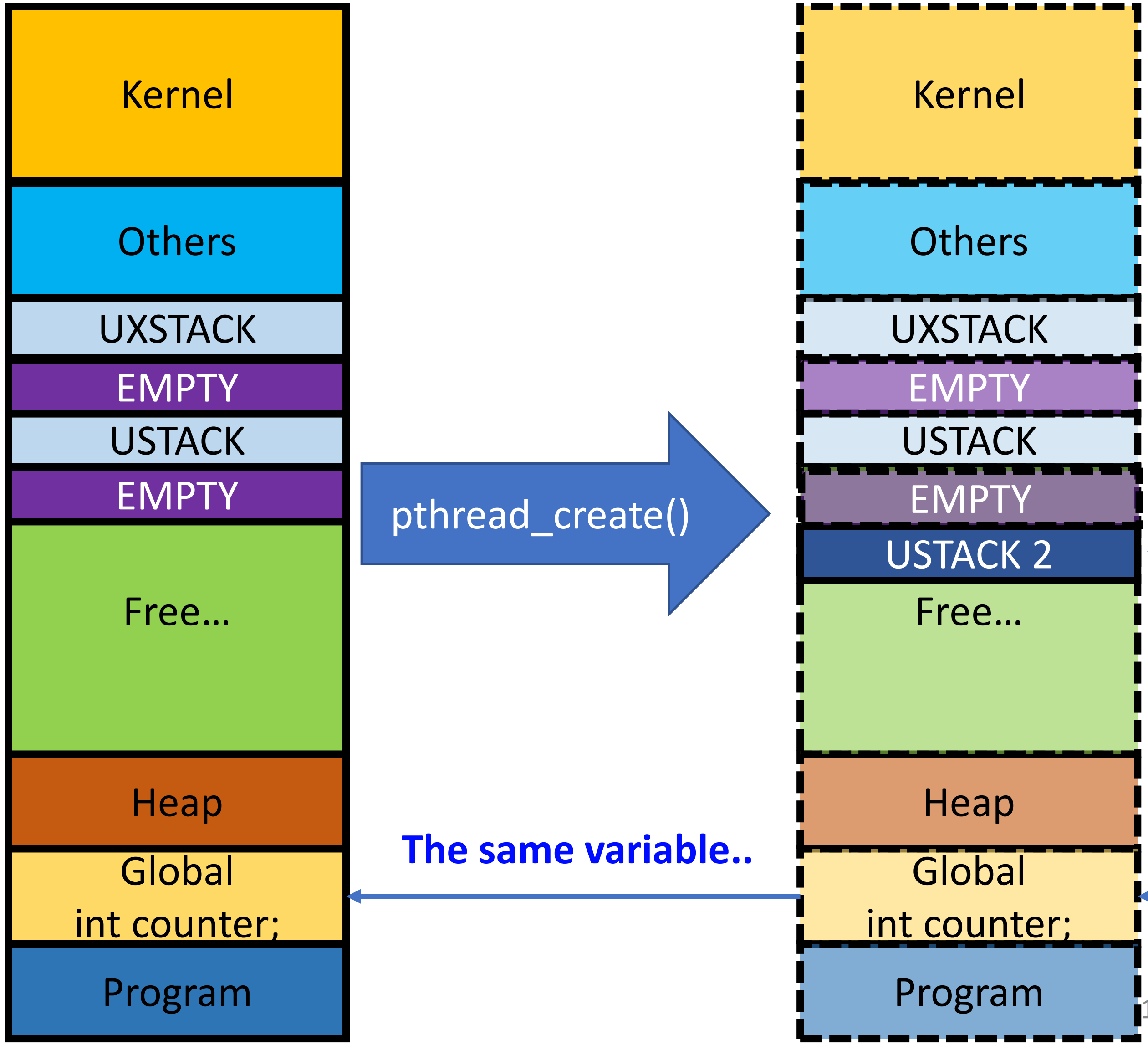



```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int counter;
volatile int value = 1;

void * countup(void *arg) {
    for(int i=0; i<1000000; ++i) {
        counter += value;
    }
    printf("%s: %d\n", arg ? "Parent" : " Child", counter);
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, countup, NULL);
    countup((void*) 1);
    pthread_join(thread, NULL);
}
```

Thread (Pros/Cons)

- Pros
  - Threads can directly access memory space of other threads
    - Sharing data!
  - Require less memory than fork()
    - A stack and few more..
- Cons
  - No isolated execution; the programmer needs to be careful
- Use
  - Suitable for parallel execution on shared data
  - Not suitable for having a private execution

Synchronization Issue..




```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int counter;
volatile int value = 1;

void * countup(void *arg) {
    for(int i=0; i<1000000; ++i) {
        counter += value;
    }
    printf("%s: %d\n", arg ? "Parent" : " Child", counter);
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, countup, NULL);
    countup((void*) 1);
    pthread_join(thread, NULL);
}
```

```
Child: 1092487
Parent: 1221966
```

```
Child: 975822
Parent: 1081479
```

Why not 2000000?

Add a new stack!

Adding value..

Data Race

- A thread's execution result could be inconsistent if other threads intervene its execution…

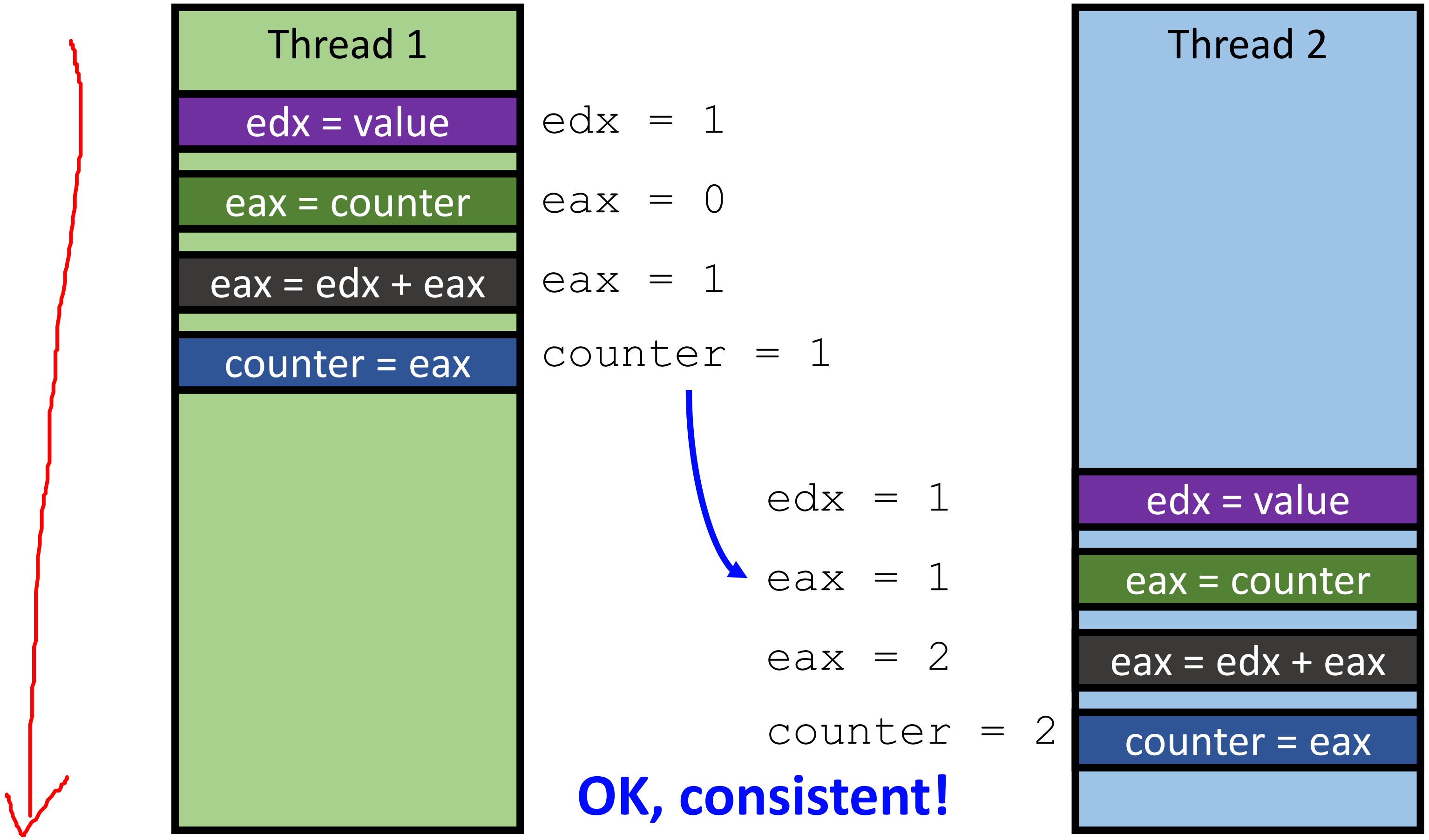
- counter += value
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`

```
mov     0x20087b(%rip),%edx          # 0x201010 <value>
mov     0x20087d(%rip),%eax          # 0x201018 <counter>
add     %edx,%eax
mov     %eax,0x200875(%rip)          # 0x201018 <counter>
```

Data Race Example (No race)

- counter += value
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`

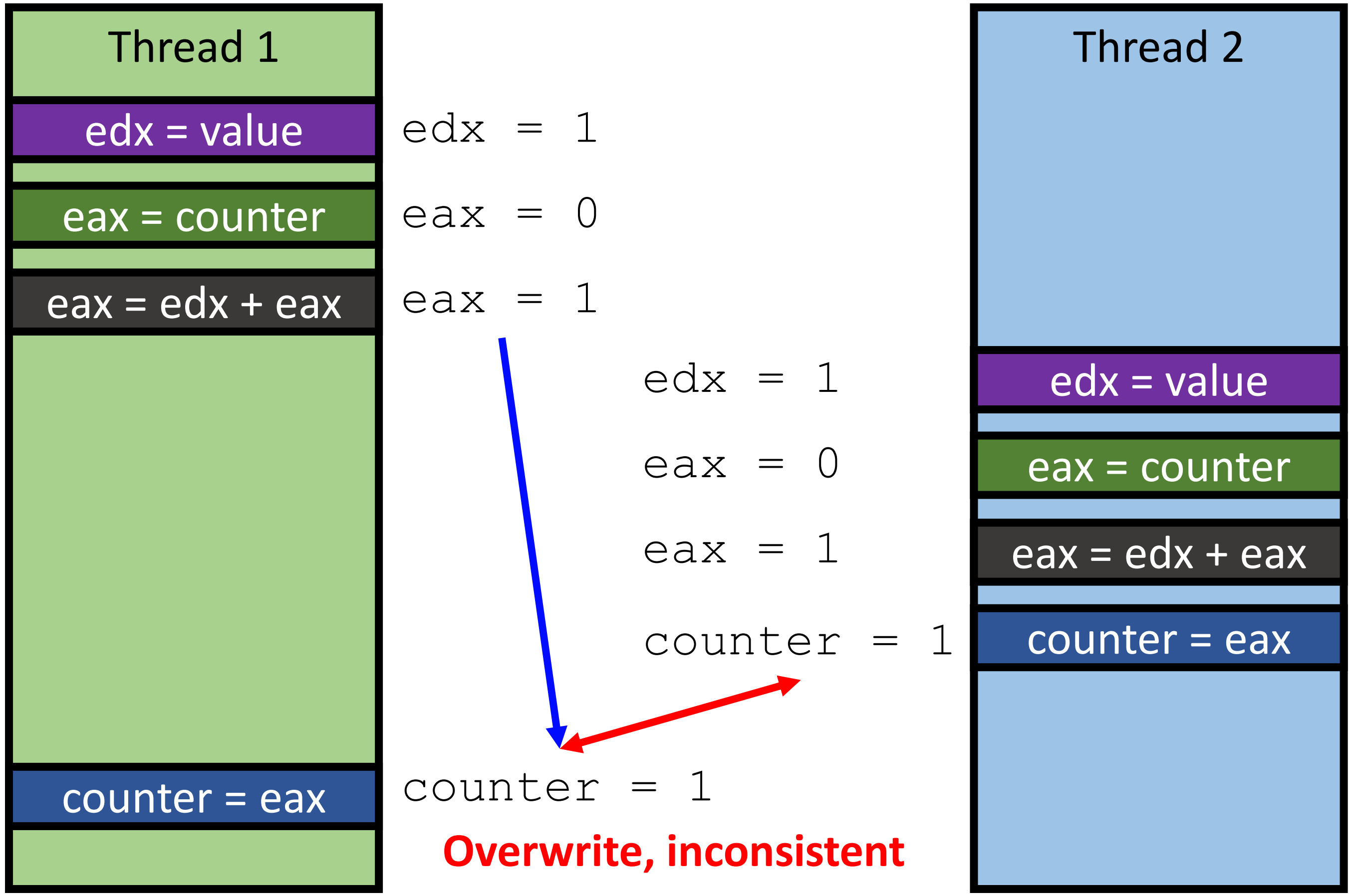
- Assume counter = 0 at start, and value = 1;



Data Race Example (Race cond.)

- counter += value
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`

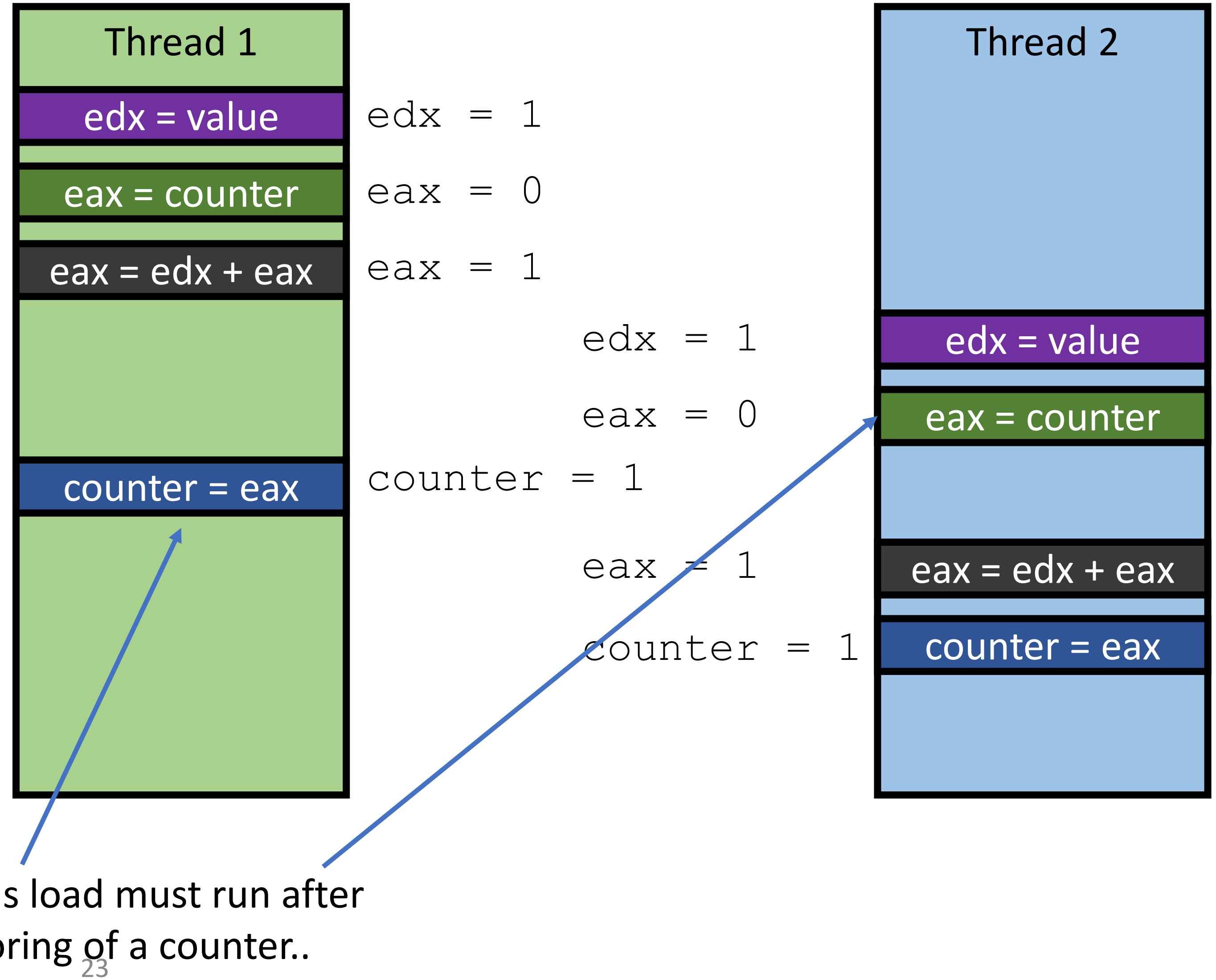
- Assume counter = 0 at start, and value = 1;



Data Race Example (Race cond.)

- counter += value
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`

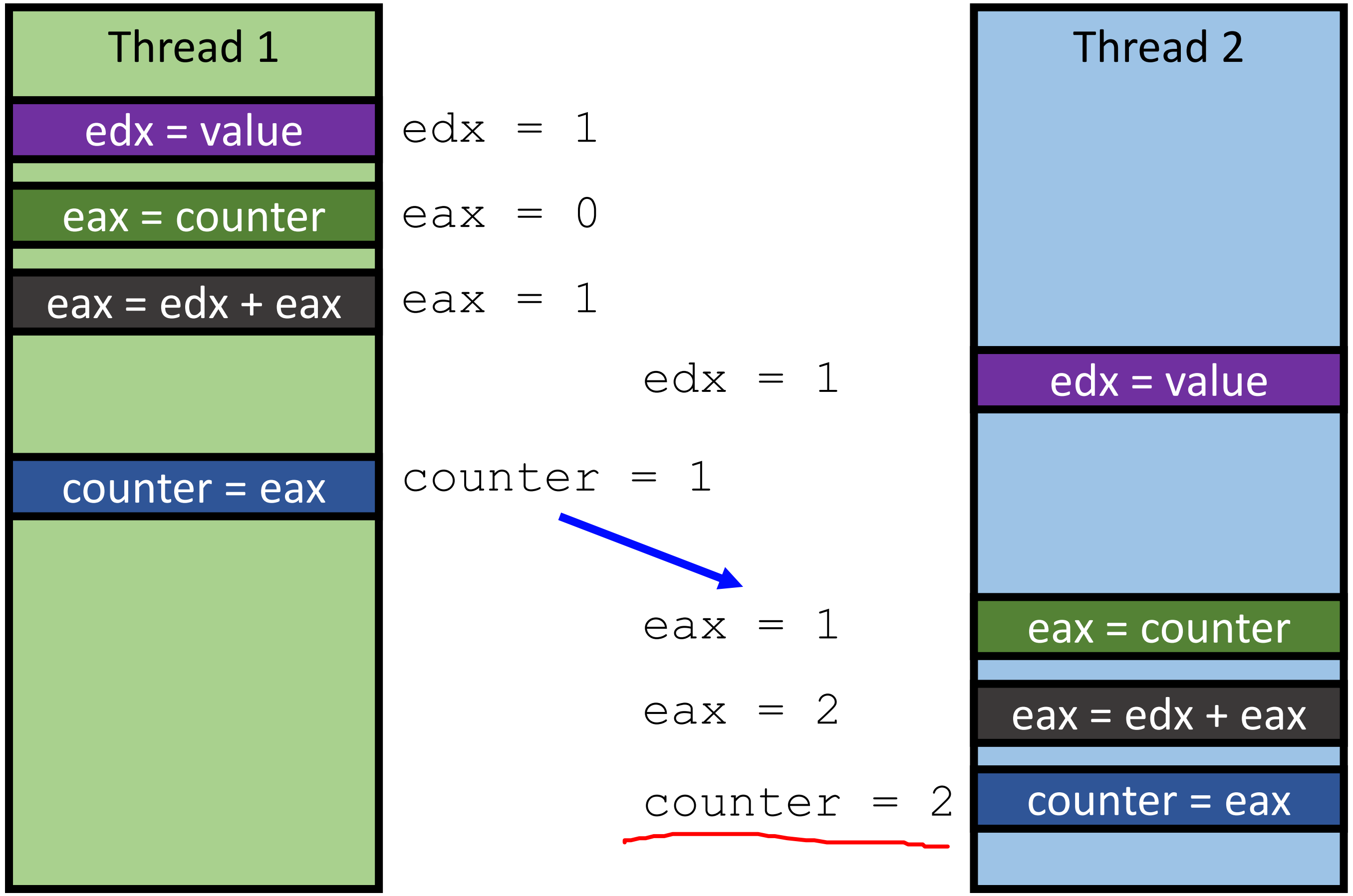
- Assume counter = 0 at start, and value = 1;



Data Race Example (Race cond.)

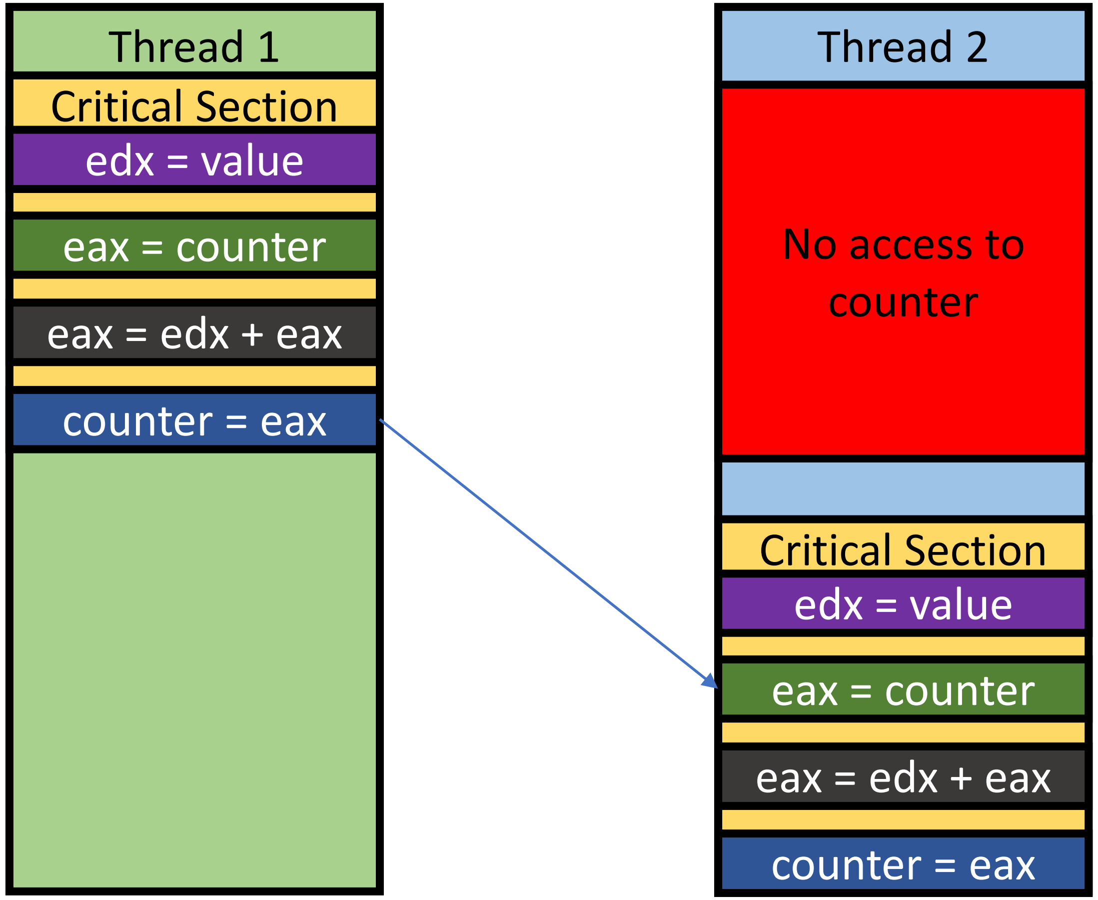
- counter += value
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`

- Assume counter = 0 at start, and value = 1;



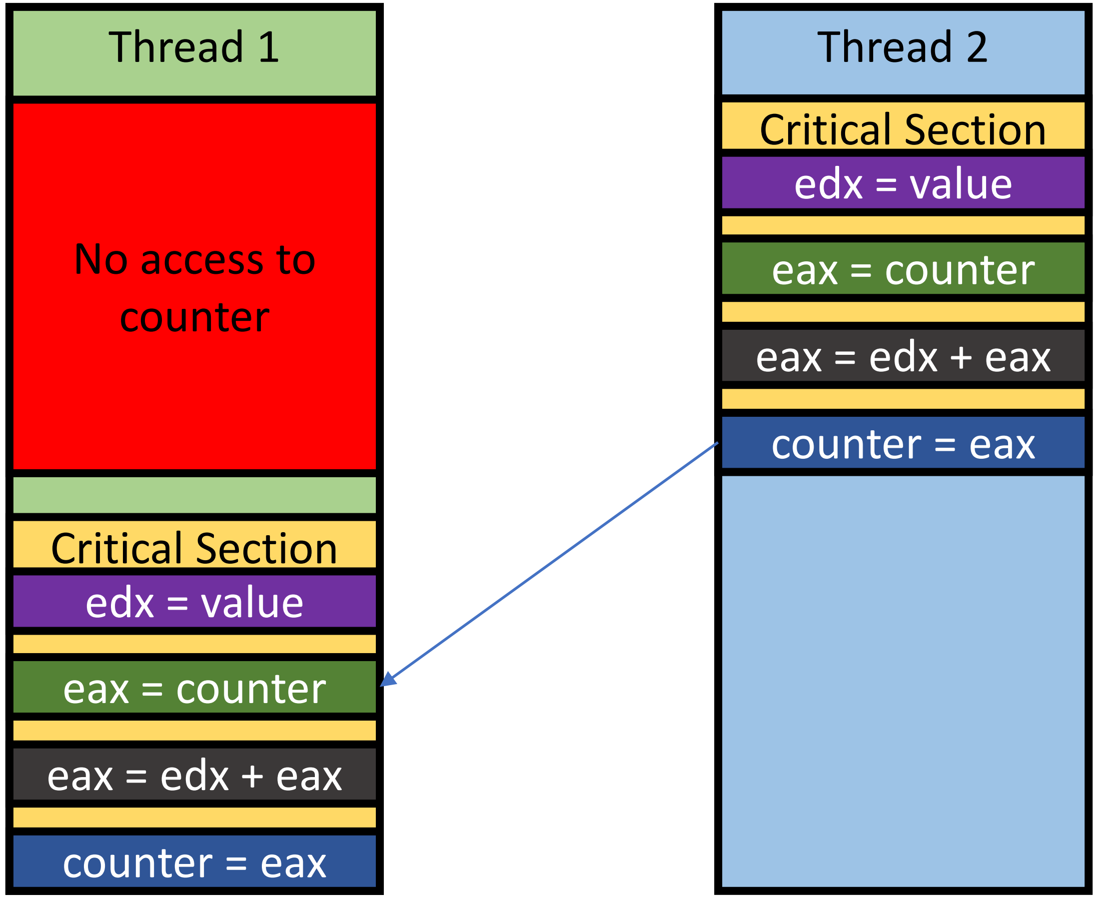
How to Prevent Data Racing?

- Mutual Exclusion / Critical Section
  - Combine multiple instructions as a chunk
  - Let only one chunk execution runs
  - Block other executions

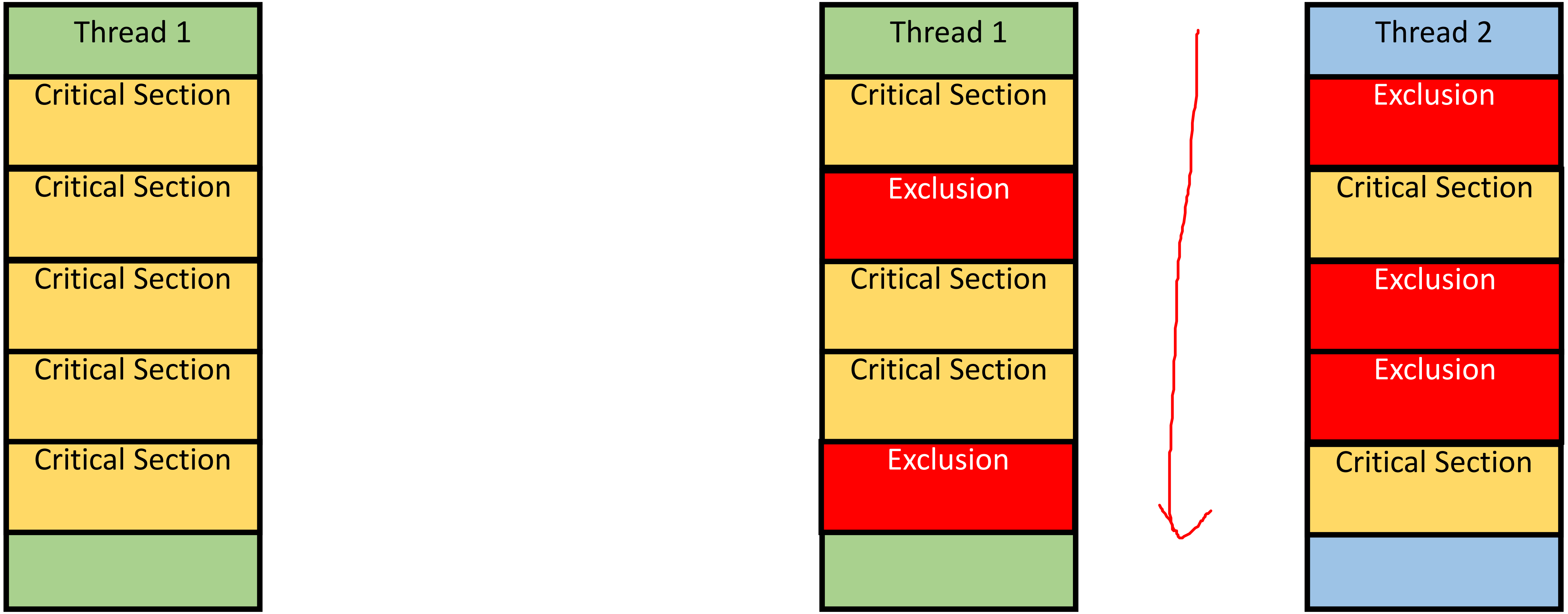


How to Prevent Data Racing?

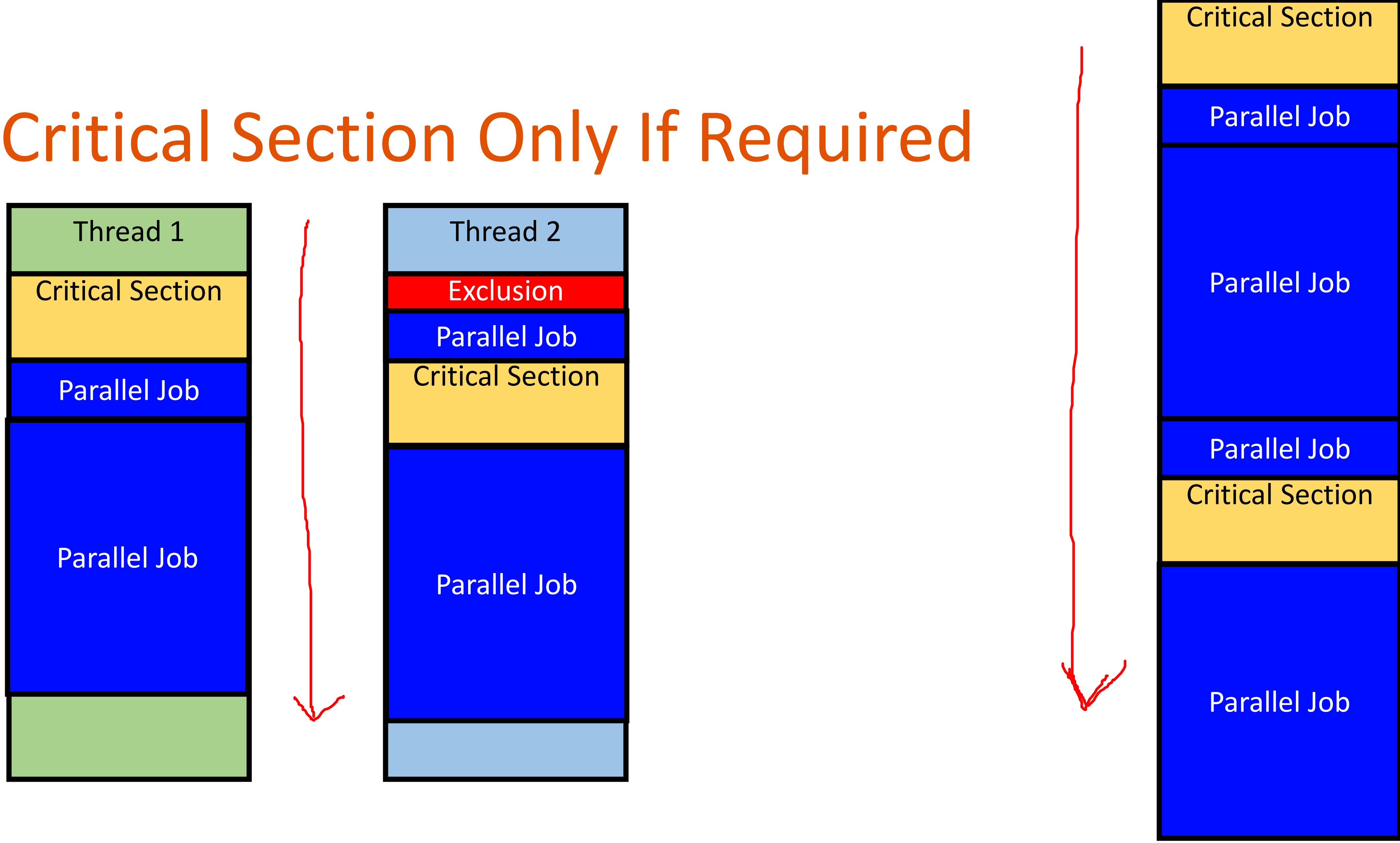
- Mutual Exclusion / Critical Section
  - Combine multiple instructions as a chunk
  - Let only one chunk execution runs
  - Block other executions



Would Mutex Render Threading Useless?



Use Critical Section Only If Required



Caveat: Apply Mutex only if required

- Mutex can synchronize multiple threads and yield consistent result
  - No read before previous thread stores the shared data
- Making the entire program as critical section is meaningless   bad!
  - Running time will be the same as single-threaded execution
- Apply critical section as short as possible to maximize benefit of having concurrency
  - Non-critical sections will run concurrently!

Enabling Mutual Exclusion

- `cli`, in a single processor computer
  - Clear interrupt bit
- CPU will never get interrupt until it runs `sti`
  - Set interrupt bit
- There will be no other execution
  - Any problems?
  - Multi CPU?
  - `cli/sti` available in Ring 0

- counter += value
  - **`cli`**
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`
  - **`sti`**

Mutex (Mutual Exclusion)

- Lock
  - Prevent others enter the critical section
- Unlock
  - Release the lock, let others acquire the lock

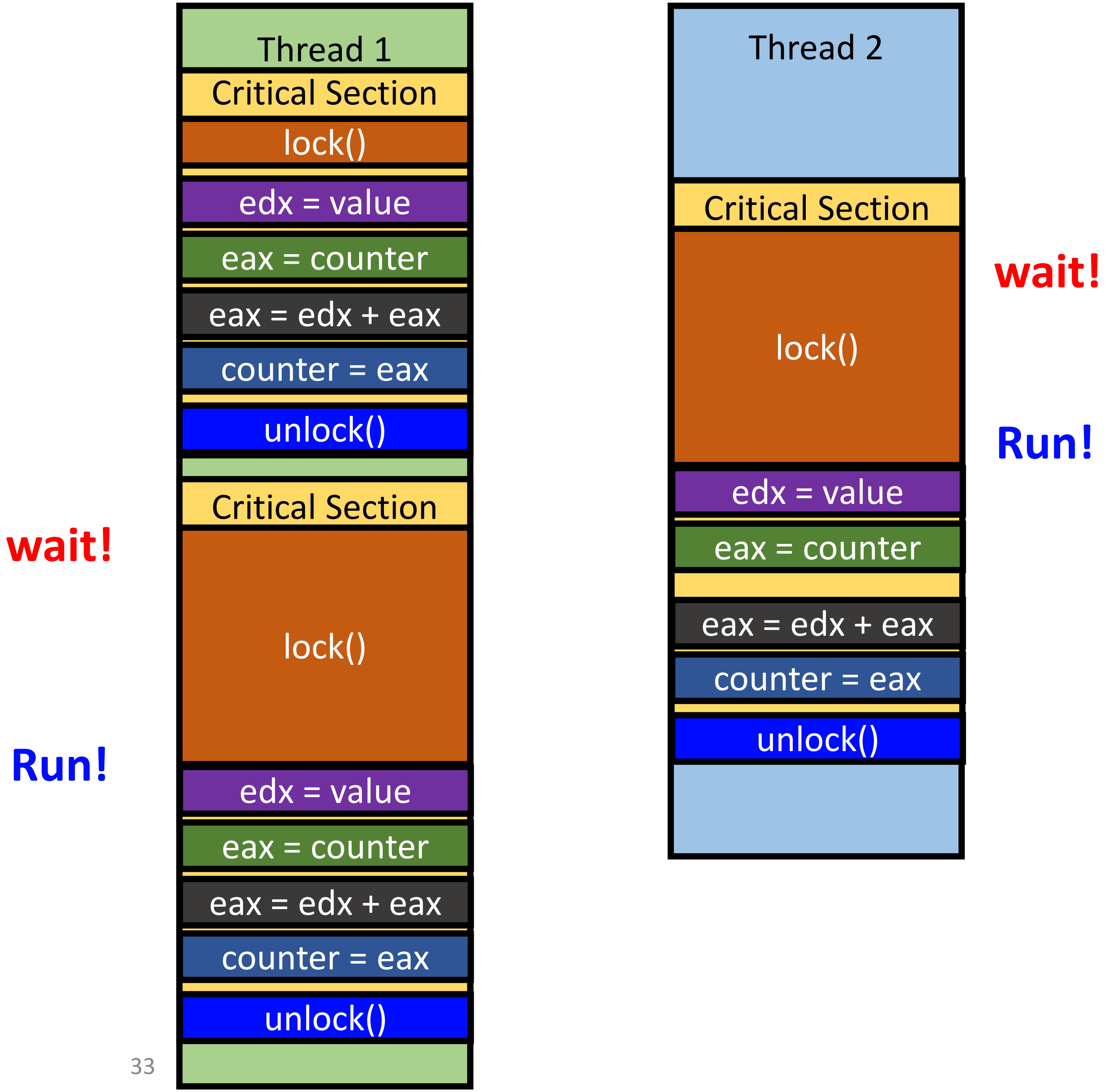
- counter += value
  - **lock()**
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`
  - **unlock()**

Mutex (Mutual Exclusion)

- Lock
  - Prevent others enter the critical section
- How?
  - Check if any other execution in the critical section
    - If it is, wait; busy-waiting with while()
  - If not, acquire the lock!
    - Others cannot get into the critical section
  - Run critical section
  - Unlock, let other execution know that I am out!

- counter += value
  - **`lock()`**
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`
  - **`unlock()`**

Mutex Example



Summary

- Single-threaded CPU performance does not increase linearly anymore
  - CPU contains many cores to speed up by concurrent execution
- Process and Thread are two options for exploiting concurrency
  - Process: new page directory/table; do not share memory; isolated
  - Thread: shares CR3 (page directory/table); shared memory; not isolated
- Data race could happen if two or more threads access same memory
  - Mutex is one way of avoiding this..