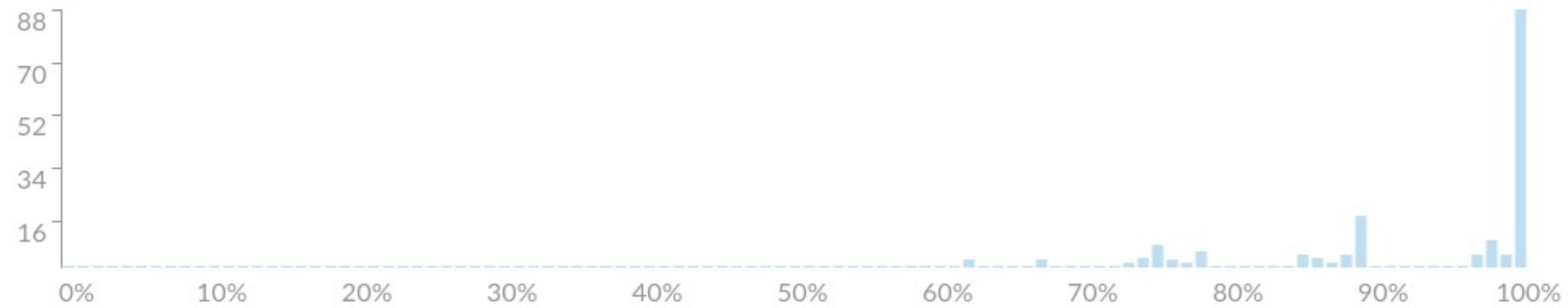
Prof. Sibin Mohan

Spring 2022 | Lec. 11: Multithreading and Synchronization

Adapted from content originally created by: Prof. Yeongjin Jang

Quiz 2

Average Score	High Score	Low Score	Standard Deviation	Average Time
94%	100%	62%	0.87	45:25



Administrivia

- Lab 3 due date extended: **May 20, 2022** [**Friday**] at **11:59 PM**!
  - Lots of office hours/lab sessions this week and next

- Watch all **Tutorials** and go through the slides/textbook

Virtualization → Concurrency

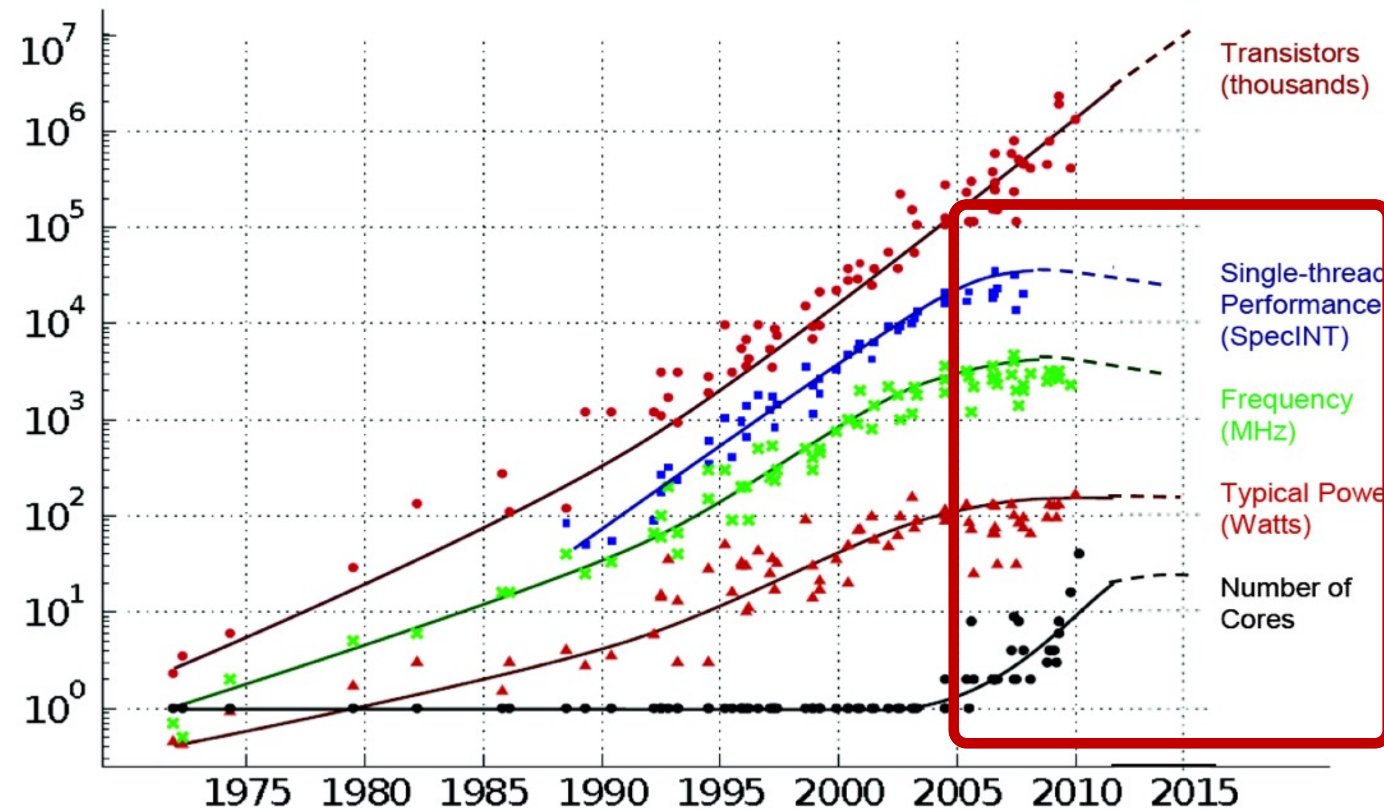
Topics discussed today

- Process / Thread
- Synchronization via Mutex
- Concurrency bugs / Deadlock

- Why is concurrency useful?
- Difference between Process/Thread
- Data racing issues
- Synchronization (Mutual Exclusion)

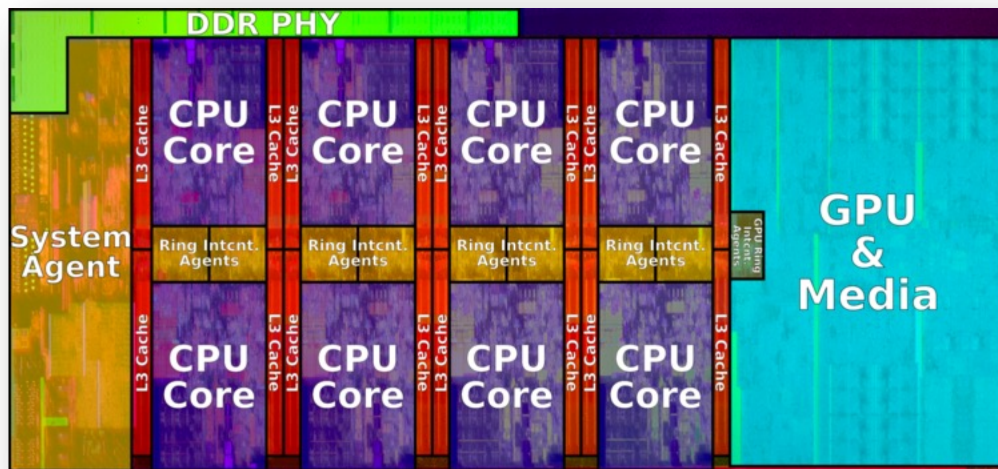
Single-threaded CPU Performance

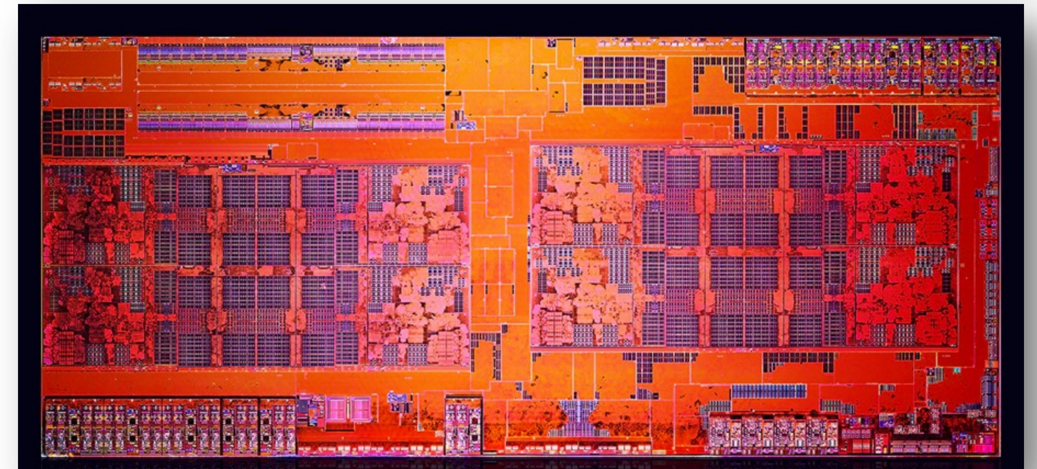
- # of transistors
  - Increases linearly
- Performance
  - **No longer increasing linearly**



CPU Speed Capped by Frequency/Power

- How to get a better performance?



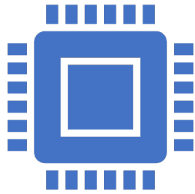


CPU Speed Capped by Frequency/Power

- How to get a better performance?

Motivation for Concurrency

vs



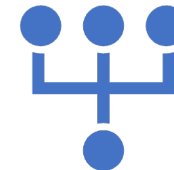
Trend in CPU

Same clock speed (3~5Ghz), **more CPU cores**

Increase System Performance

Run **many jobs at the same** time

fully utilize multiple cores

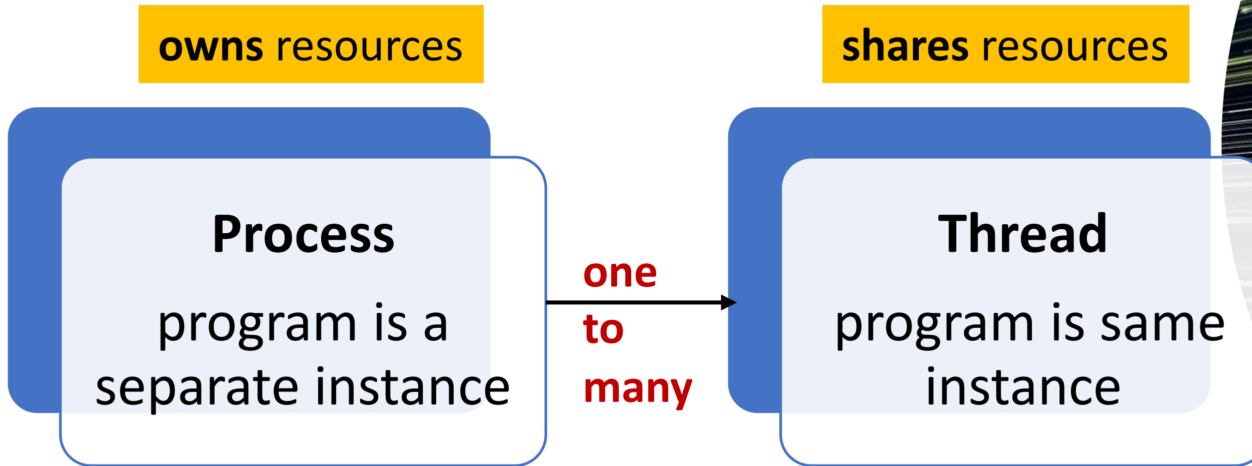


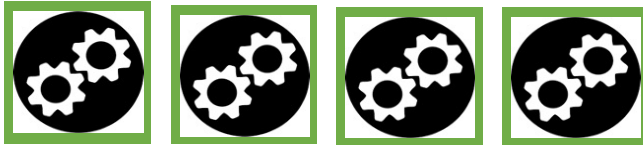
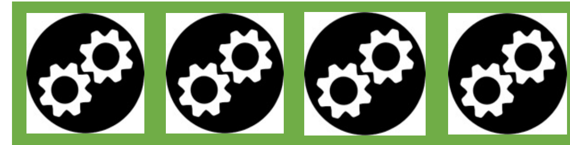
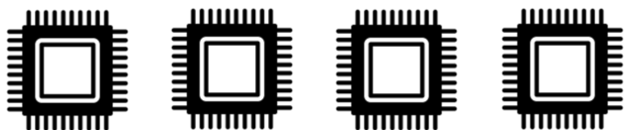
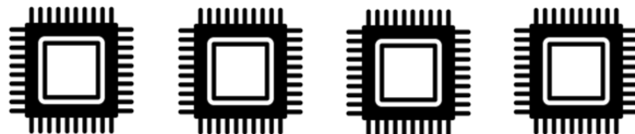
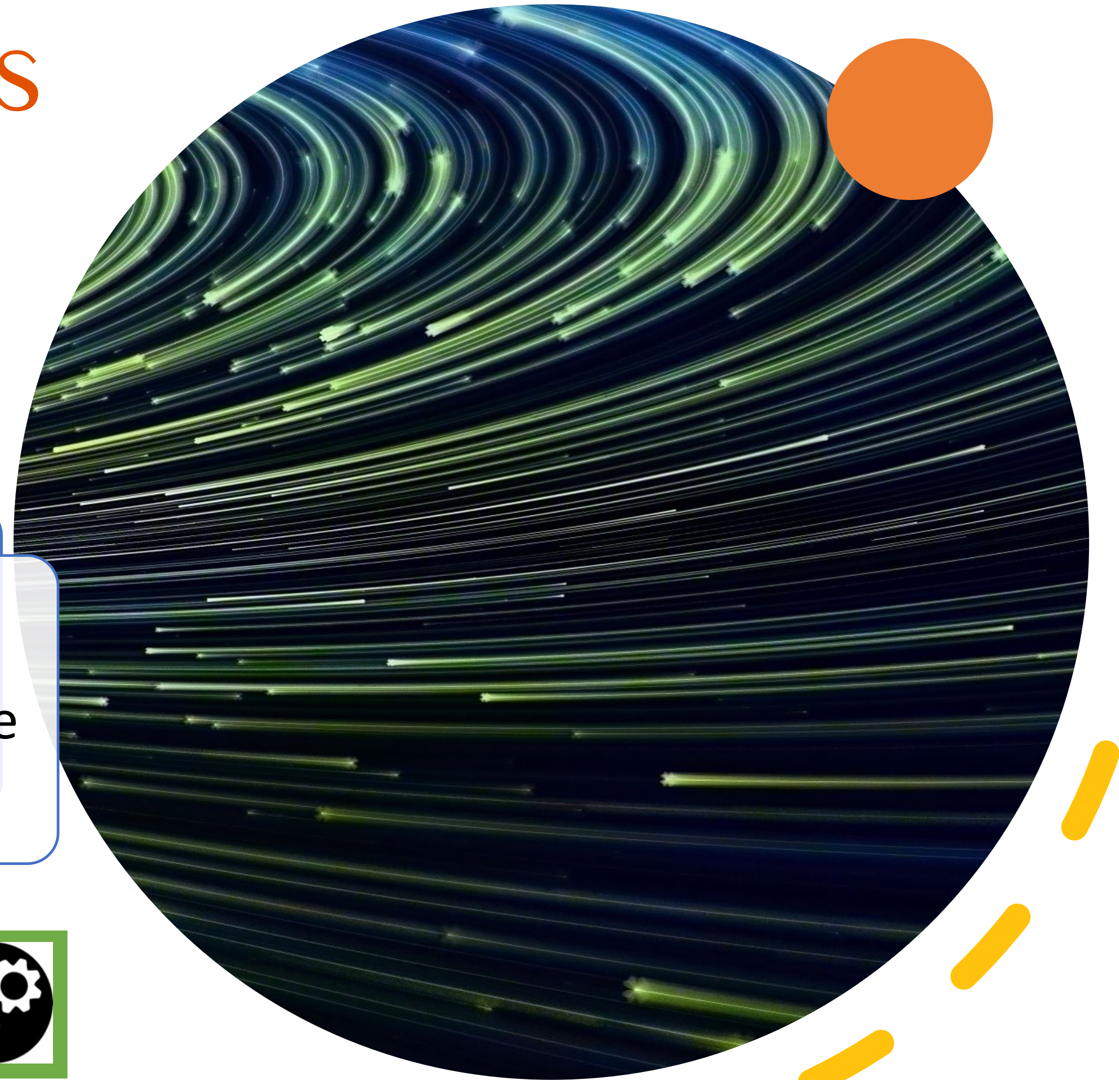
How to increase application performance?

Run **multiple functions as separate jobs** at the same time!

Processes, Threads

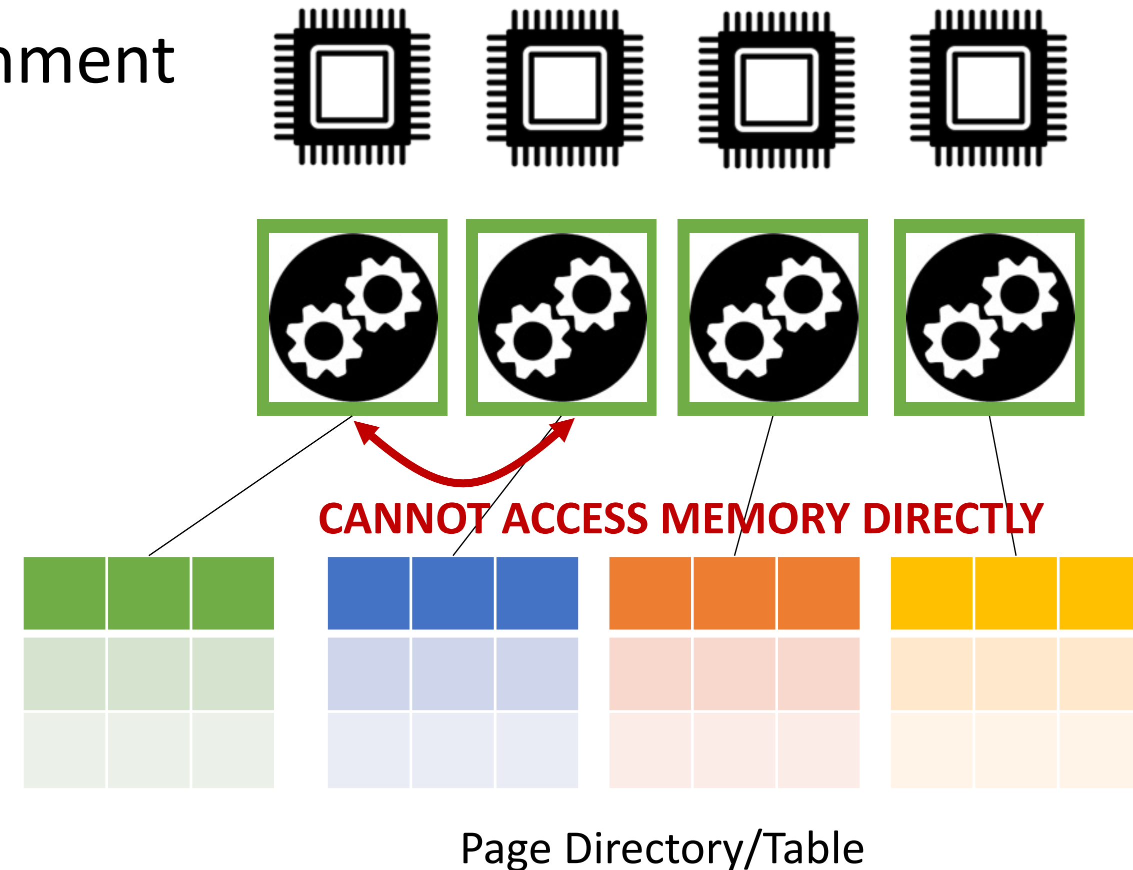
Concurrency Options



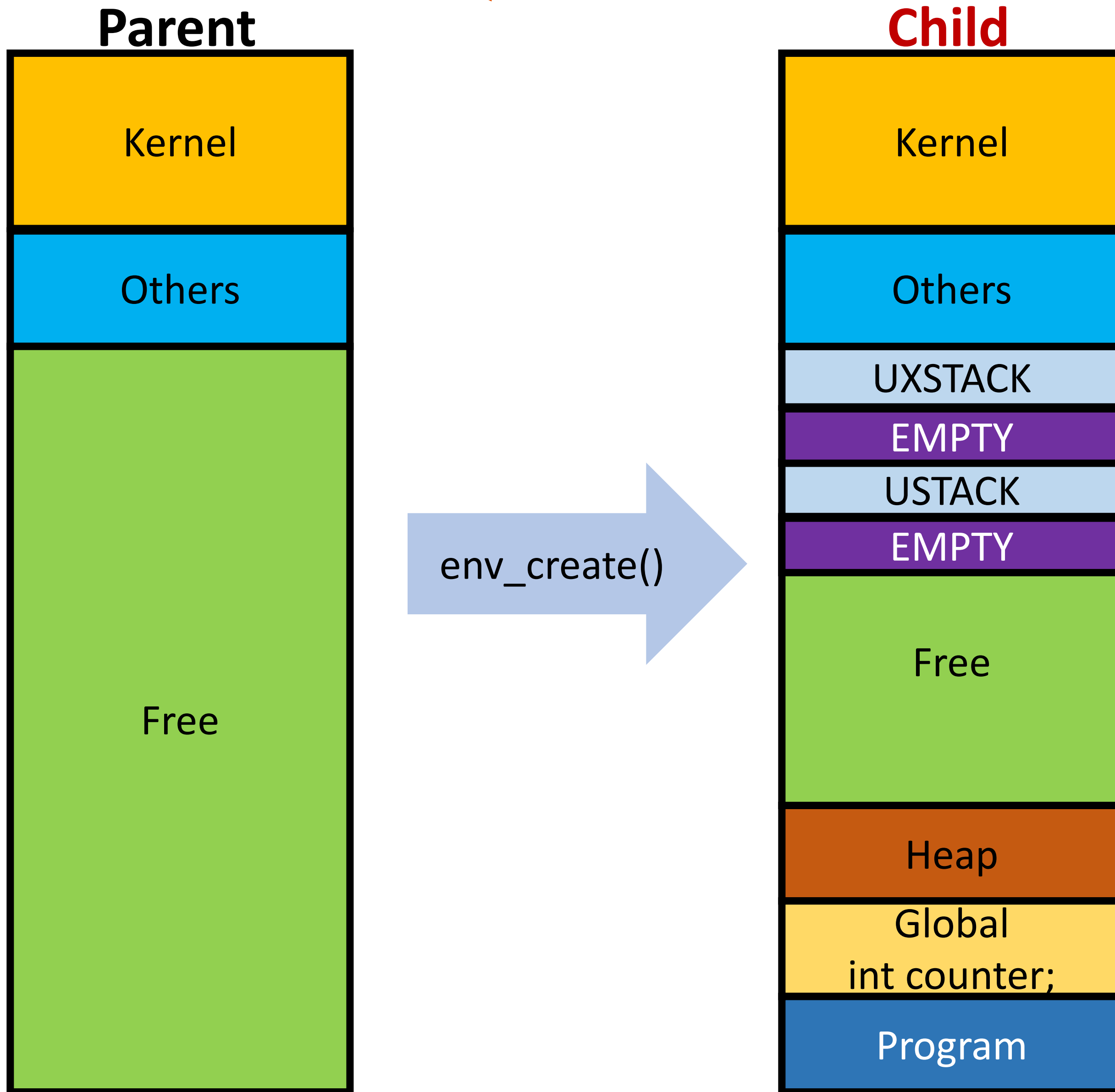






Process

- Each execution runs in isolated environment
- Does not share memory space
  - Each process its has own page table
- Inter-Process Communication
  - for data sharing
  - file, pipe, socket(), shared memory, etc.

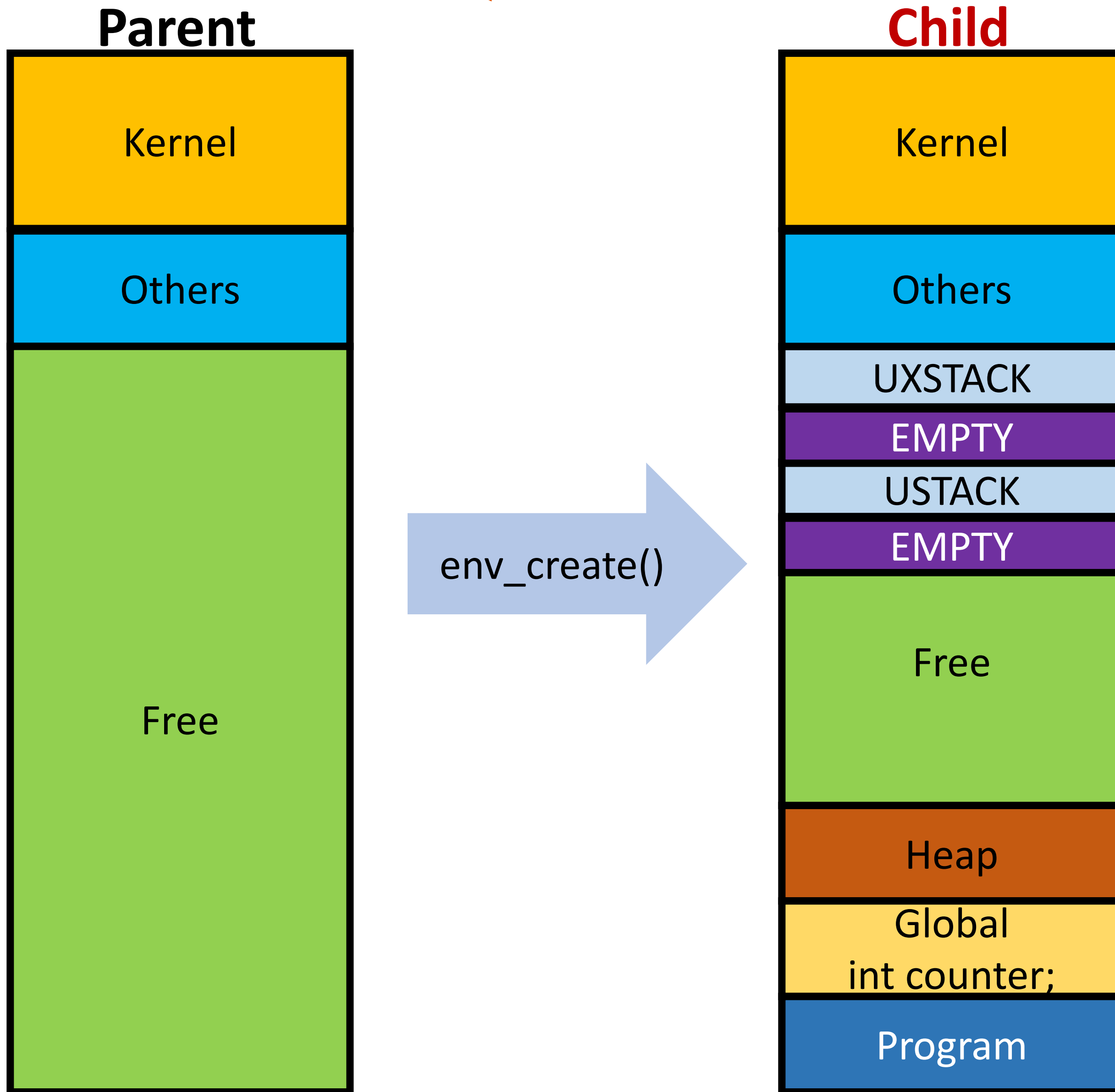


Process (Environment in JOS)



Process creates a **new private** memory space

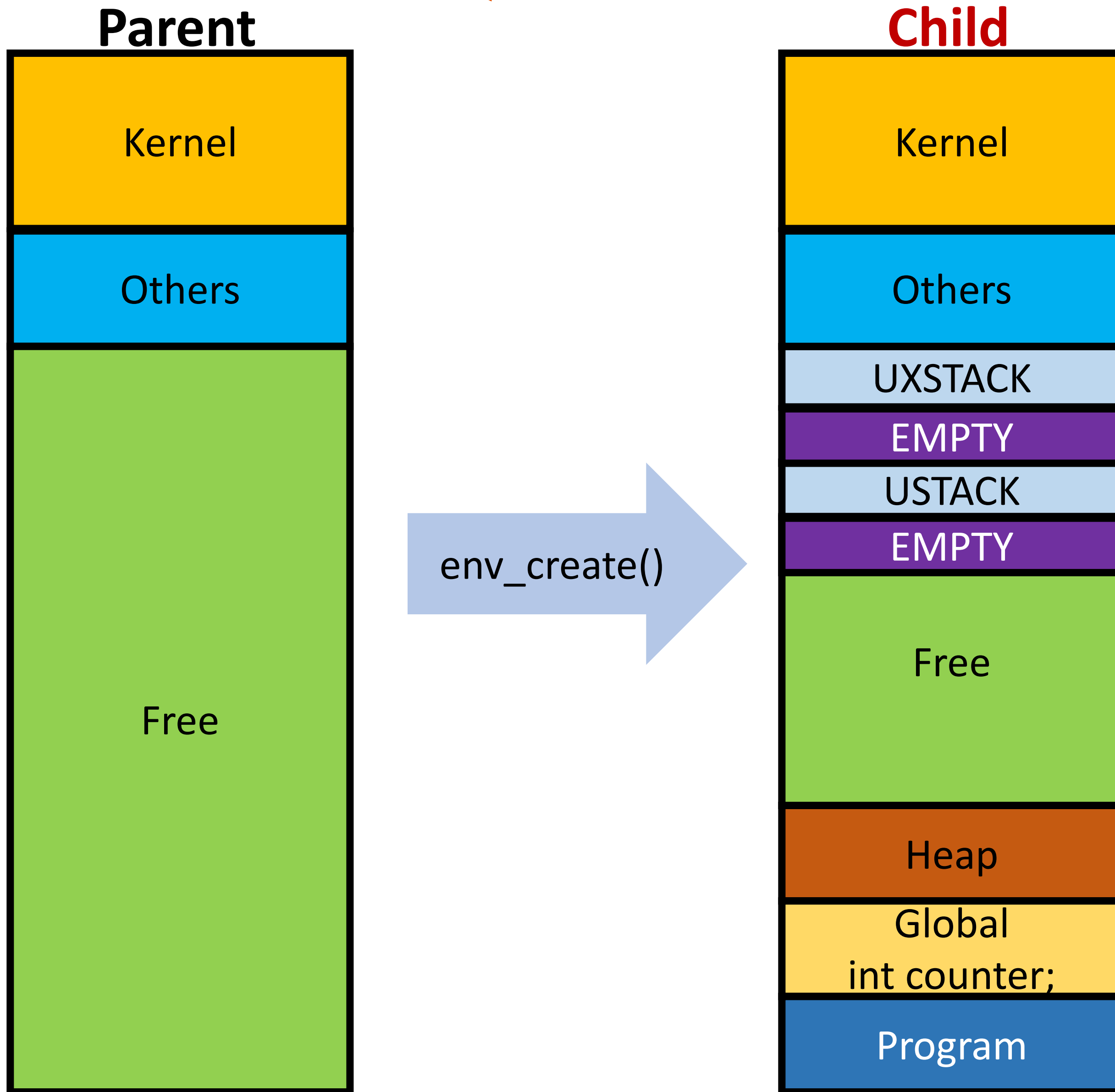
Process (Environment in JOS)



Process creates a **new private** memory space

```
int foo(arguments)
{
      some code ;

      some other code ;
}
```

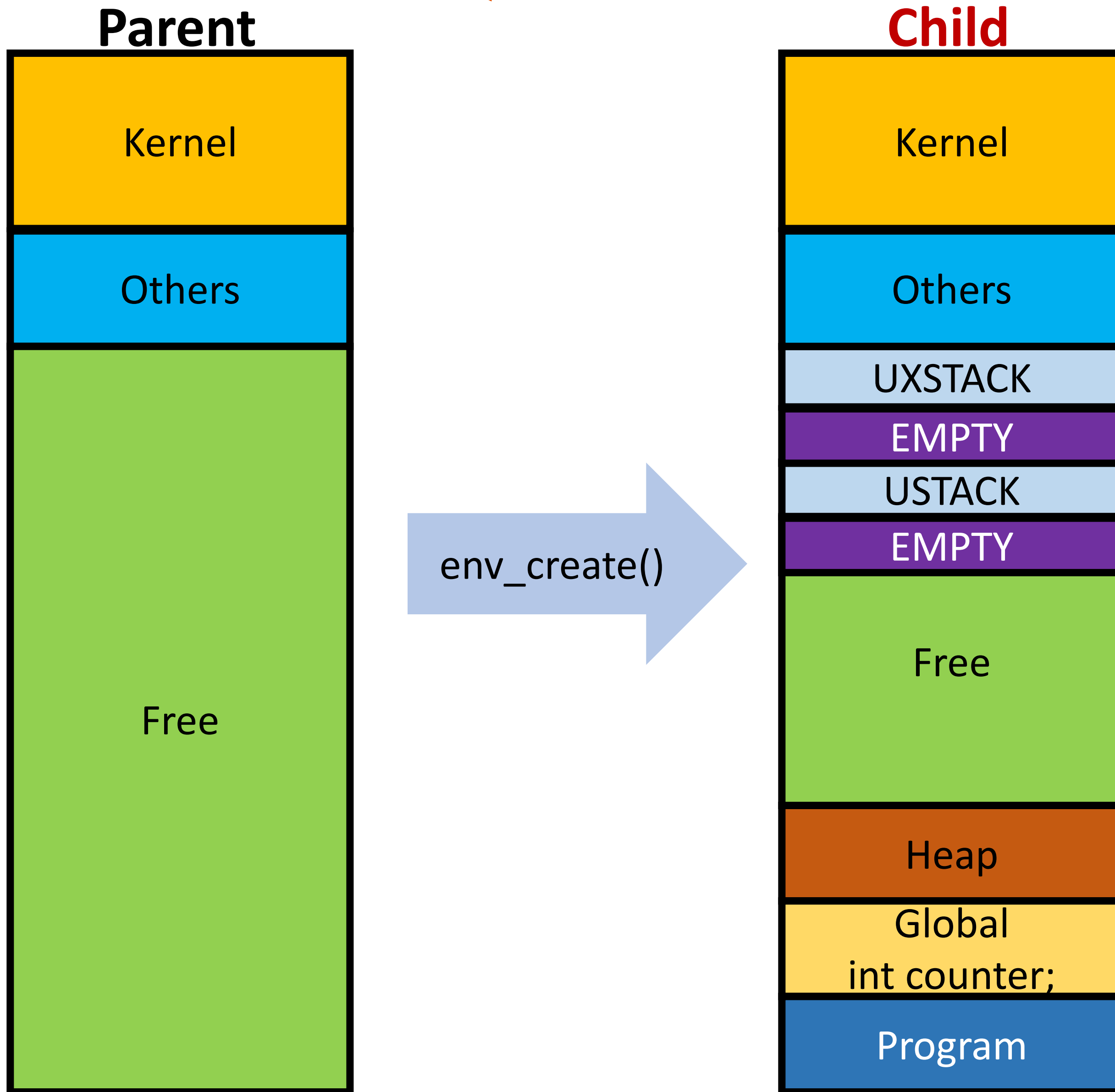
Process (Environment in JOS)



Process creates a **new private** memory space

```
int foo(arguments)
{
    some code ;
    fork() ;
    some other code ;
}
```

Process (Environment in JOS)



Process creates a **new private** memory space

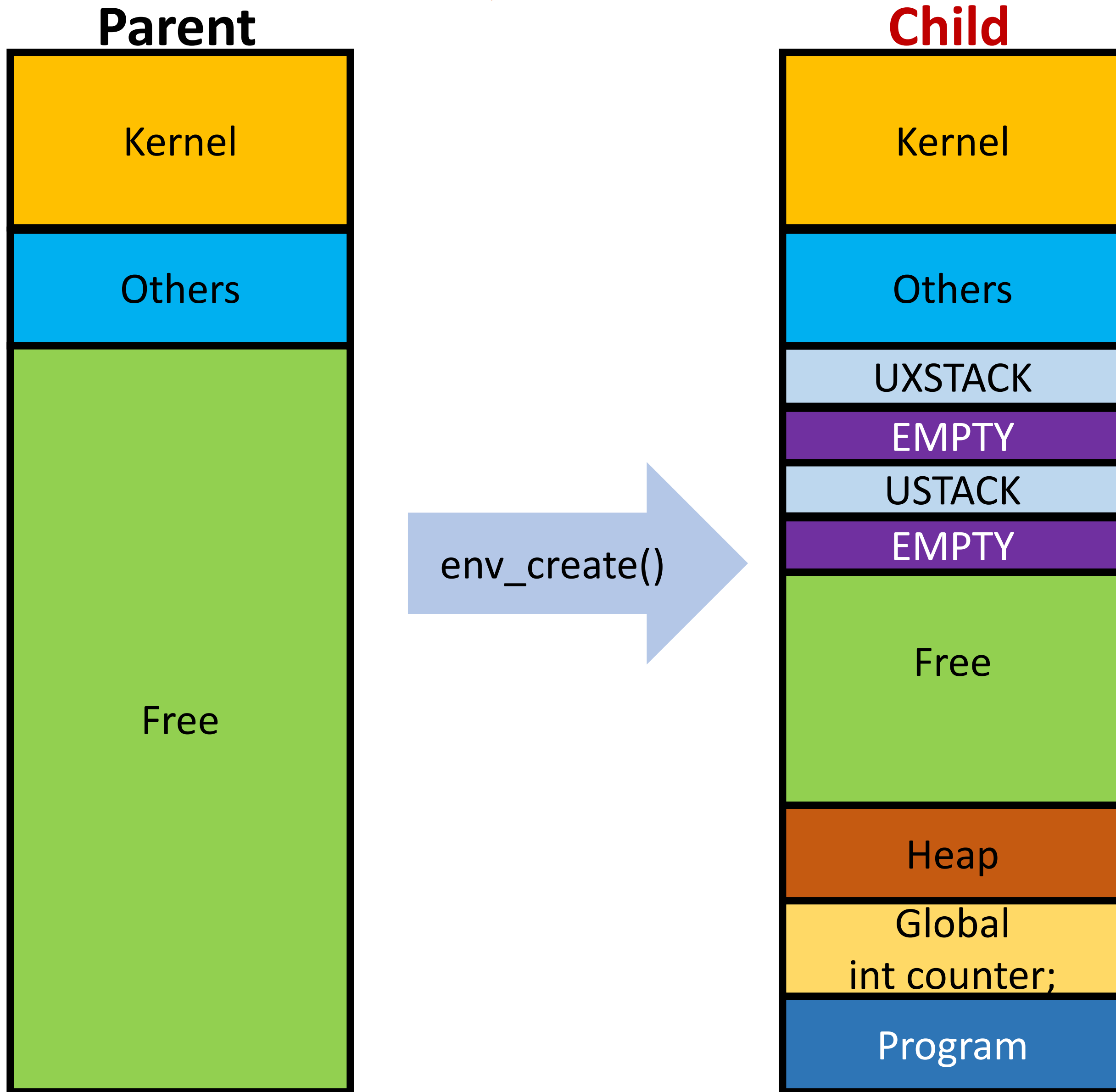
Parent

```
int foo(arguments)
{
    some code ;
    fork() ;
    some other code ;
}
```

Child

```
int foo(arguments)
{
    some code ;
    fork() ;
    some other code ;
}
```

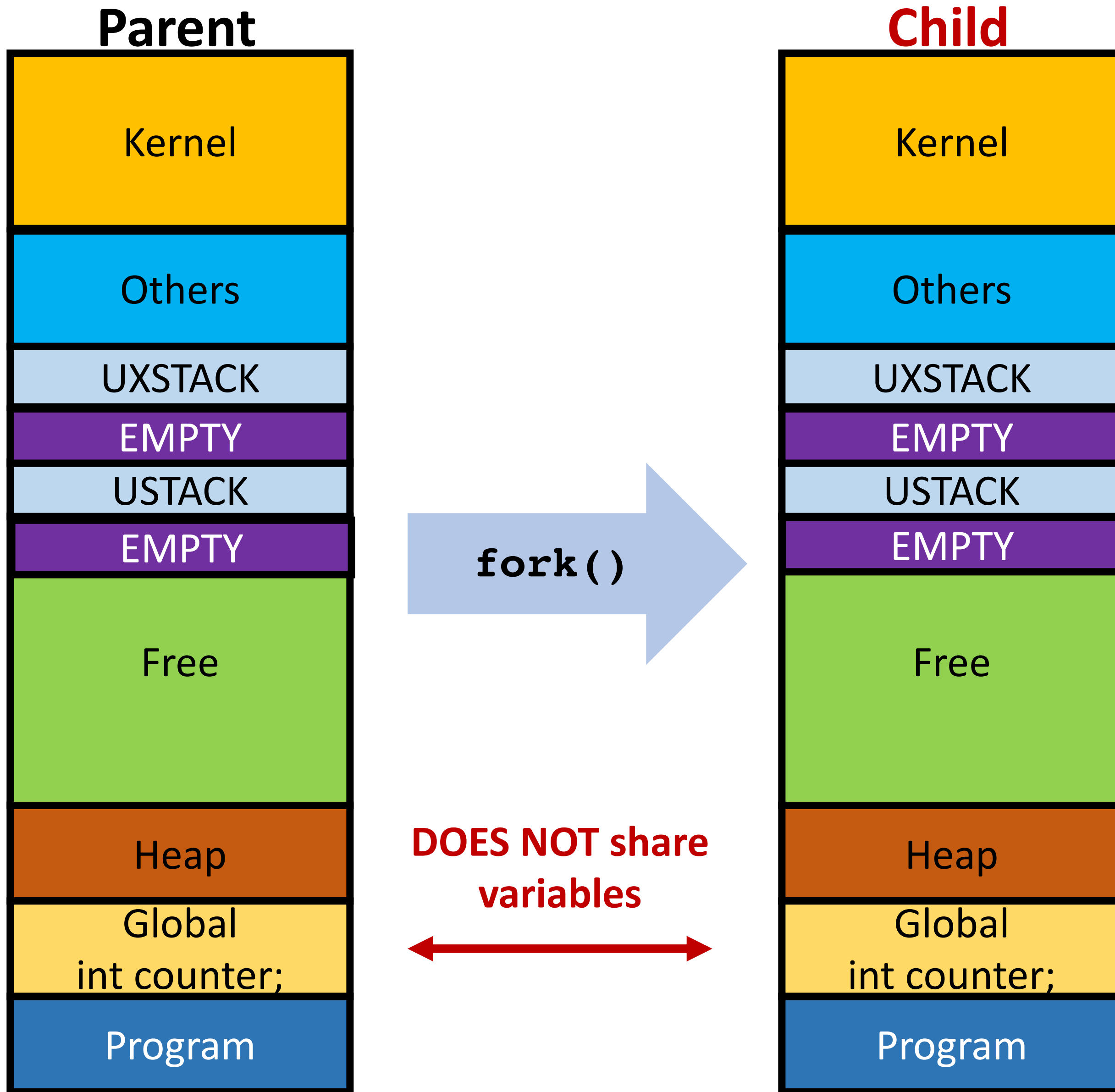
Process (Environment in JOS)



Process creates a **new private** memory space

```
int foo(arguments)
{
     some code ;
     pid_t pid = fork() ;
     if (pid == parent)
        parent_function(arguments);
     else
        child_function(arguments);
}

int parent_function(arguments)
{...}

int child_function(arguments)
{...}
```

Process (Environment in JOS)



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

```c
#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 vs Cons

Pros	Cons
Parallelism without program modifications	More memory requirements → one per process

Process | Pros vs Cons

Pros	Cons
Parallelism without program modifications	More memory requirements → one per process
Separation of memory spaces [**isolation**]	Separation of memory spaces [**no data sharing**]

Process | Pros vs Cons

Pros	Cons
Parallelism without program modifications	More memory requirements → one per process
Separation of memory spaces [**isolation**]	Separation of memory spaces [**no data sharing**]
Easy to start → fork()	Each process needs to be ended separately

Process | Pros vs Cons

Pros	Cons
Parallelism without program modifications	More memory requirements → one per process
Separation of memory spaces [**isolation**]	Separation of memory spaces [**no data sharing**]
Easy to start → fork()	Each process needs to be ended separately

- Any write will incur memory duplication even in CoW fork()
- Inter-process Communication (IPC) is available, but slow
- Suitable for **parallel 'isolated'** execution
- Not suitable for parallel execution on **shared data**

Two Issues

Parallelism

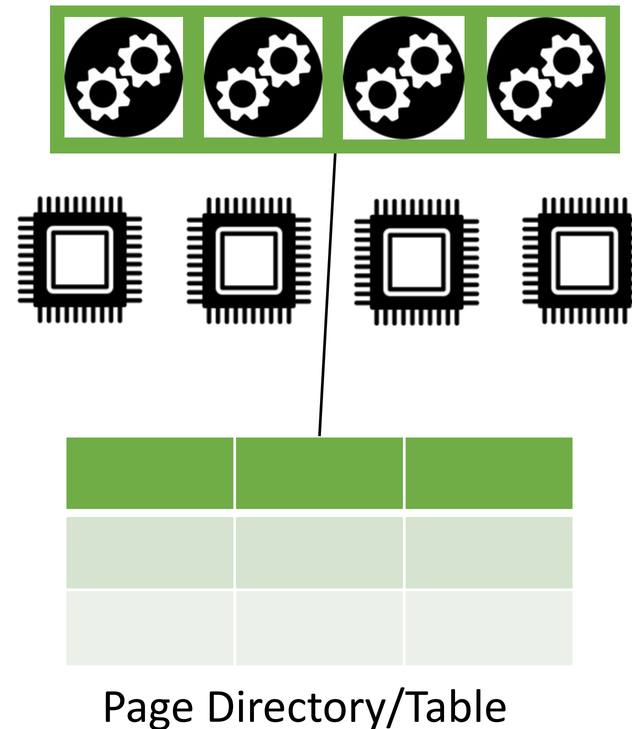
Share a memory space

Threads!

Threads

Process
- creates a new **private** memory space
- runs concurrently

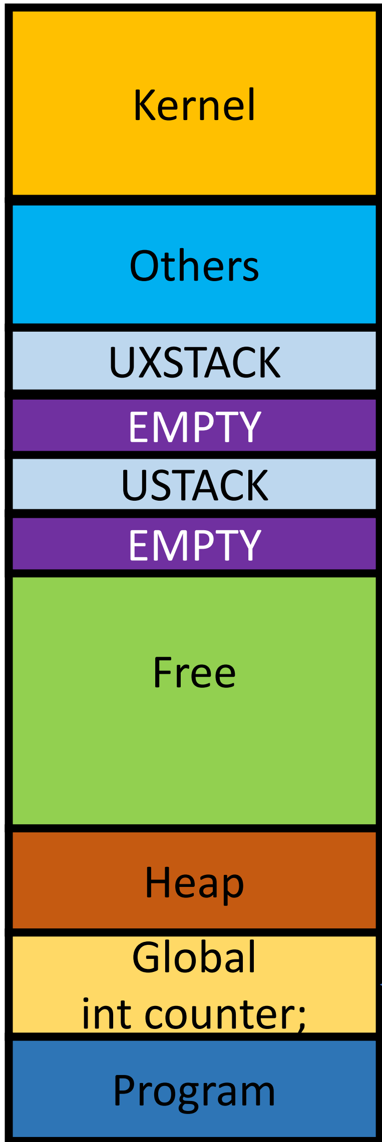
- What is a thread?
  - creates a **shared** memory space
  - runs **concurrently**
- **Sharing**
  - **access the same memory space**
  - e.g., global variables, etc.
- A process contains 1 or more threads

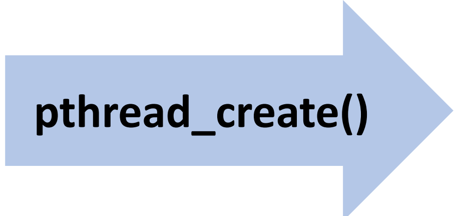


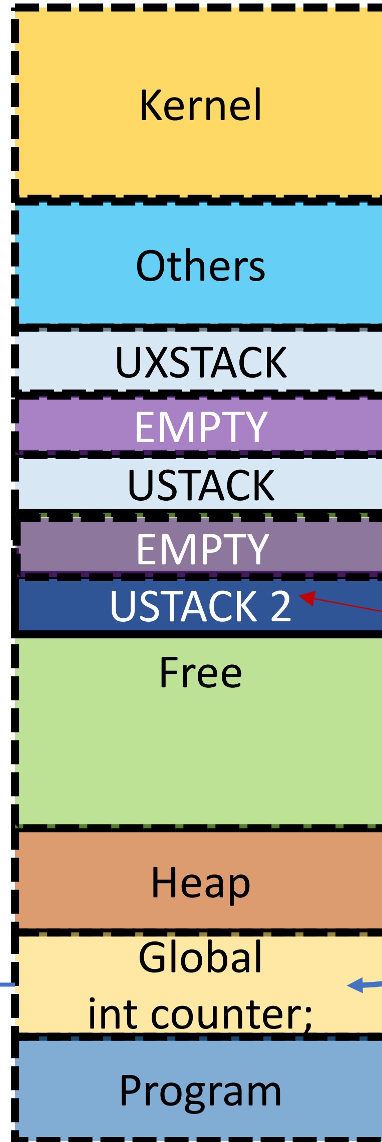
Thread | Sharing Memory

- **Process** Creation via `fork()`
  - Naïve design
    - copy all physical pages
    - create **a new page directory/table**
    - has same virtual mapping (to new, corresponding physical pages)
  - Copy-on-write
    - do not copy all physical pages but keep the same mappings
    - 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







The same variable

Add a new stack!

Adding value

```
#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 vs Cons

Pros	Cons
Easier to share memory across threads	No isolation! Programmers must be careful

Thread | Pros vs Cons

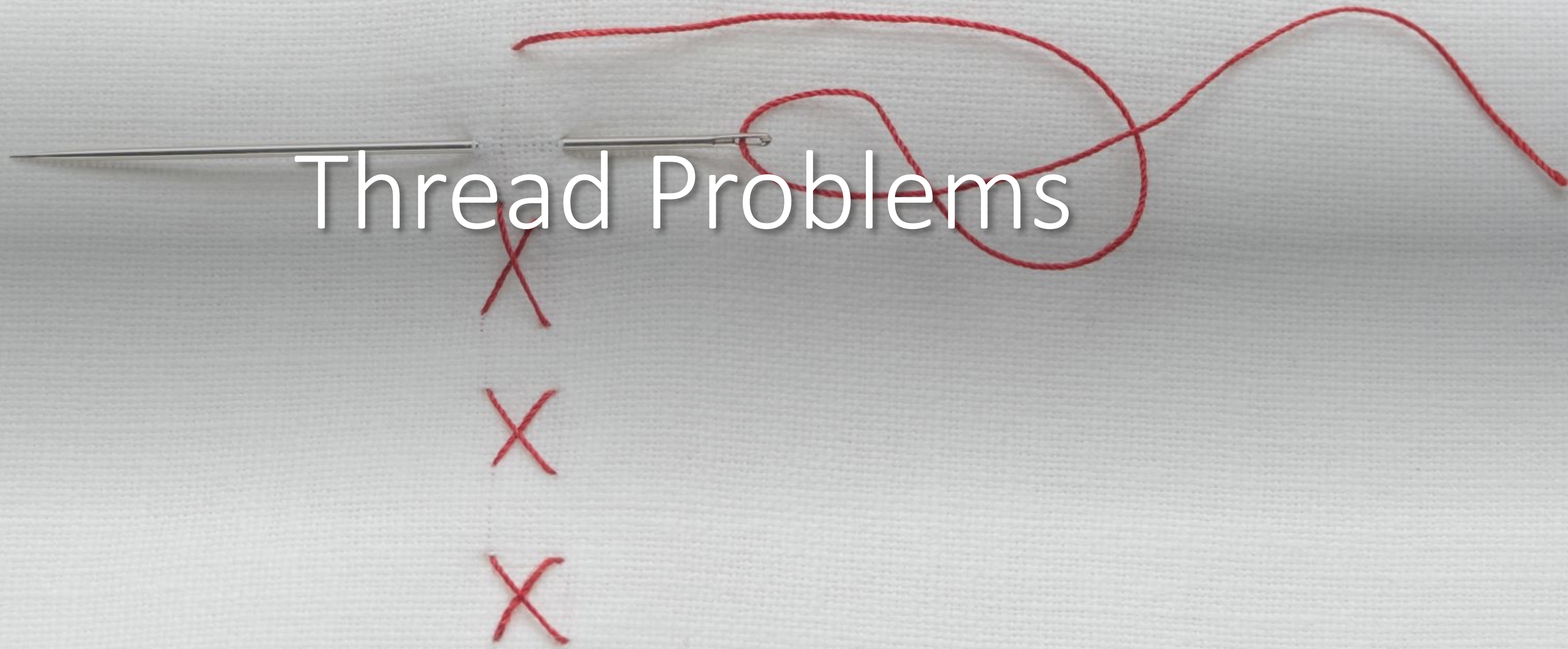
Pros	Cons
Easier to share memory across threads	No isolation! Programmers must be careful
Less memory, compared to fork [only new stack, registers]	All threads share same address space

Thread | Pros vs Cons

Pros	Cons
Easier to share memory across threads	No isolation! Programmers must be careful
Less memory, compared to fork [only new stack, registers]	All threads share same address space

- Suitable for parallel execution on shared data
- Not suitable for having a private execution

Thread Problems



Thread

Process (before)	Process (after pthread_create())
Kernel	Kernel
Others	Others
UXSTACK	UXSTACK
EMPTY	EMPTY
USTACK	USTACK
EMPTY	EMPTY
	USTACK 2
Free	Free
Heap	Heap
Global int counter;	Global int counter;
Program	Program

pthread_create()

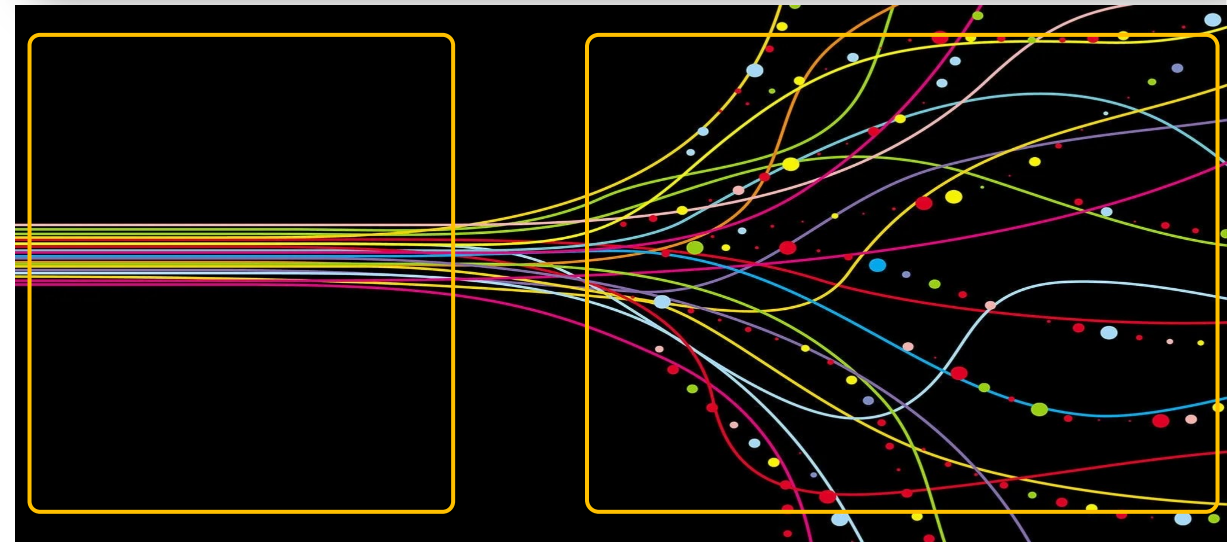
The same variable

```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);
}
```

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

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

Why not 2000000?



Data Race

- A thread’s execution result could be **inconsistent**
- **other threads intervene its execution!**

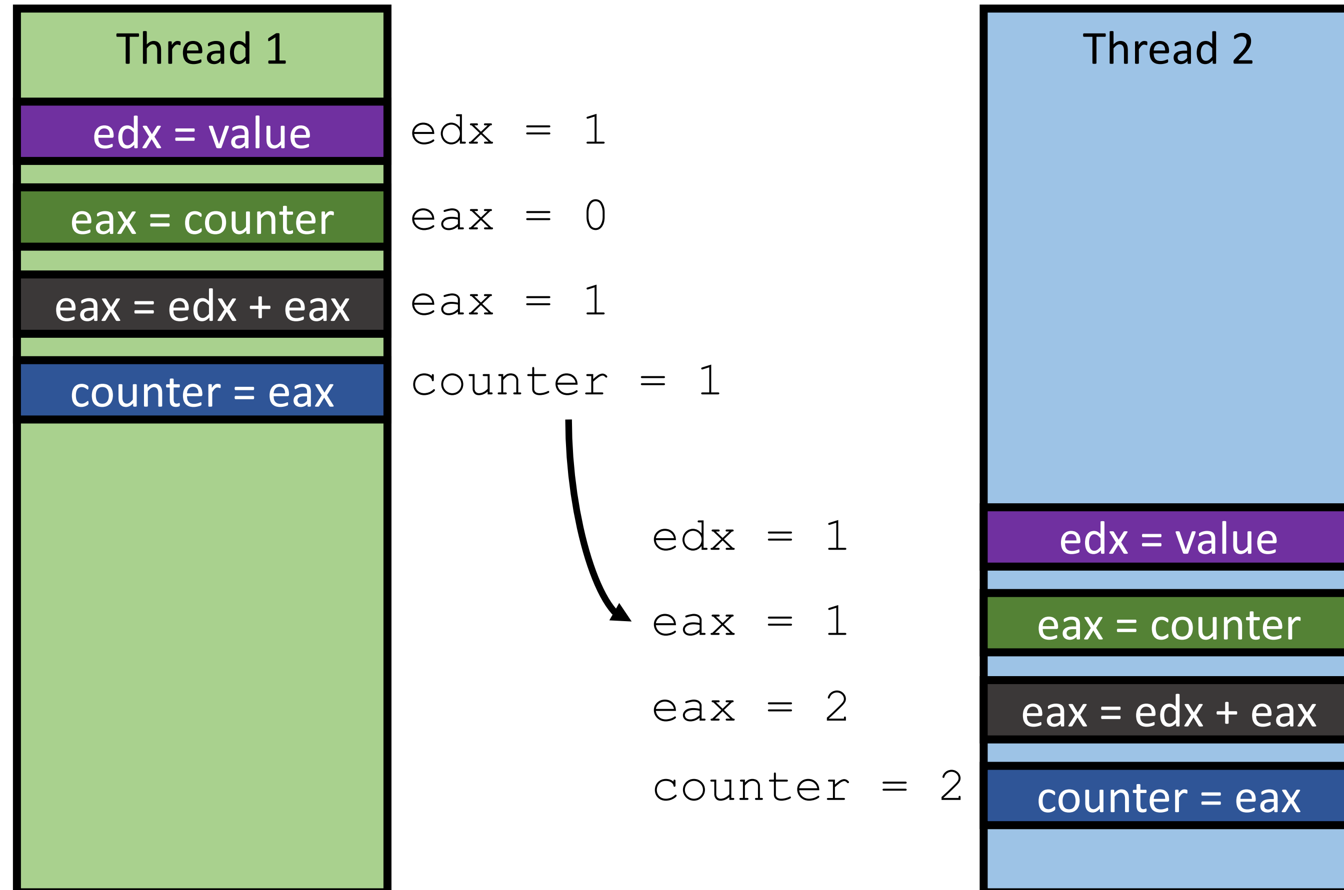
- counter += value
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`

```
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 at start,
  - counter = 0
  - value = 1

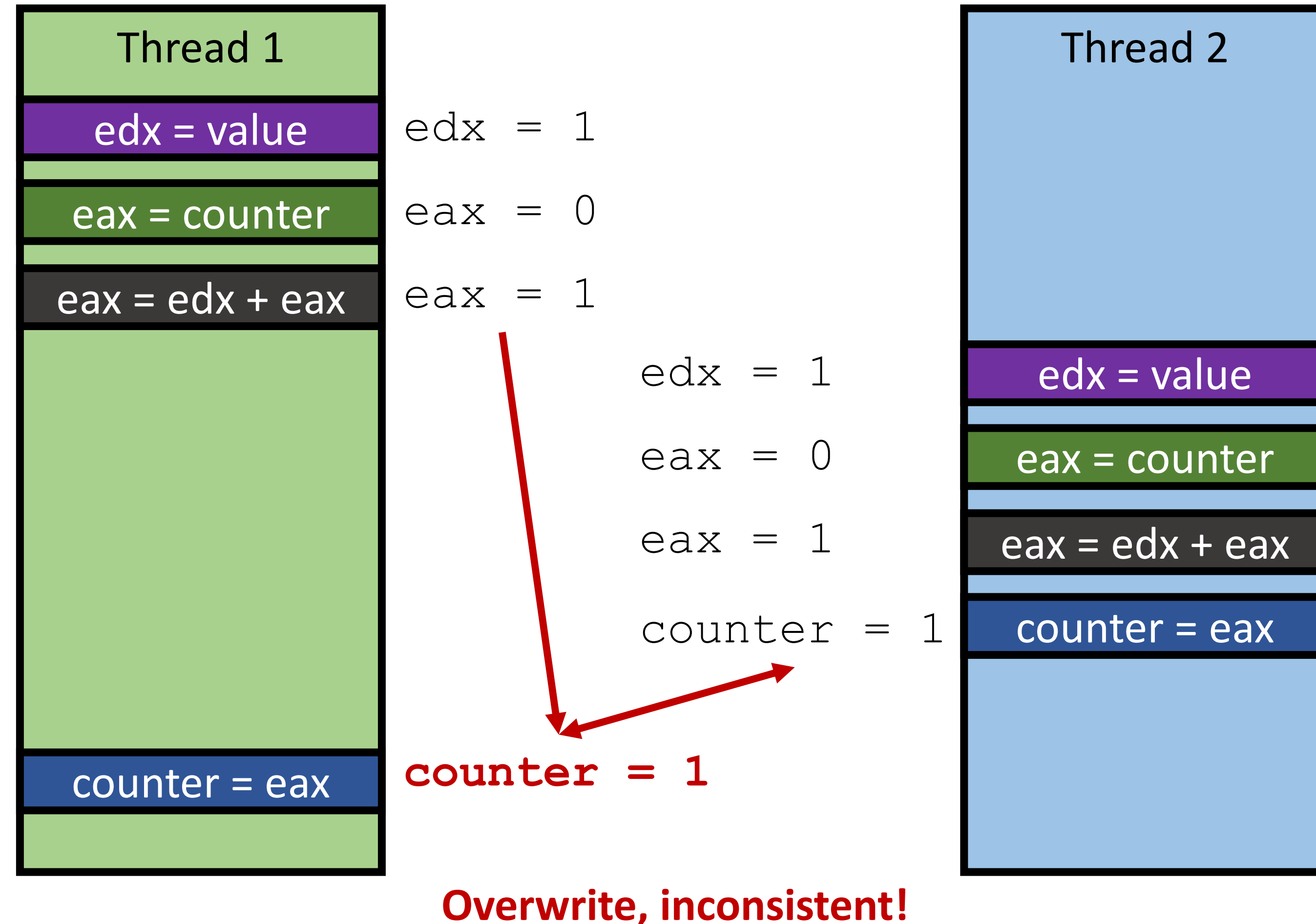


OK, consistent!

Data Race Example (Race cond.)

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

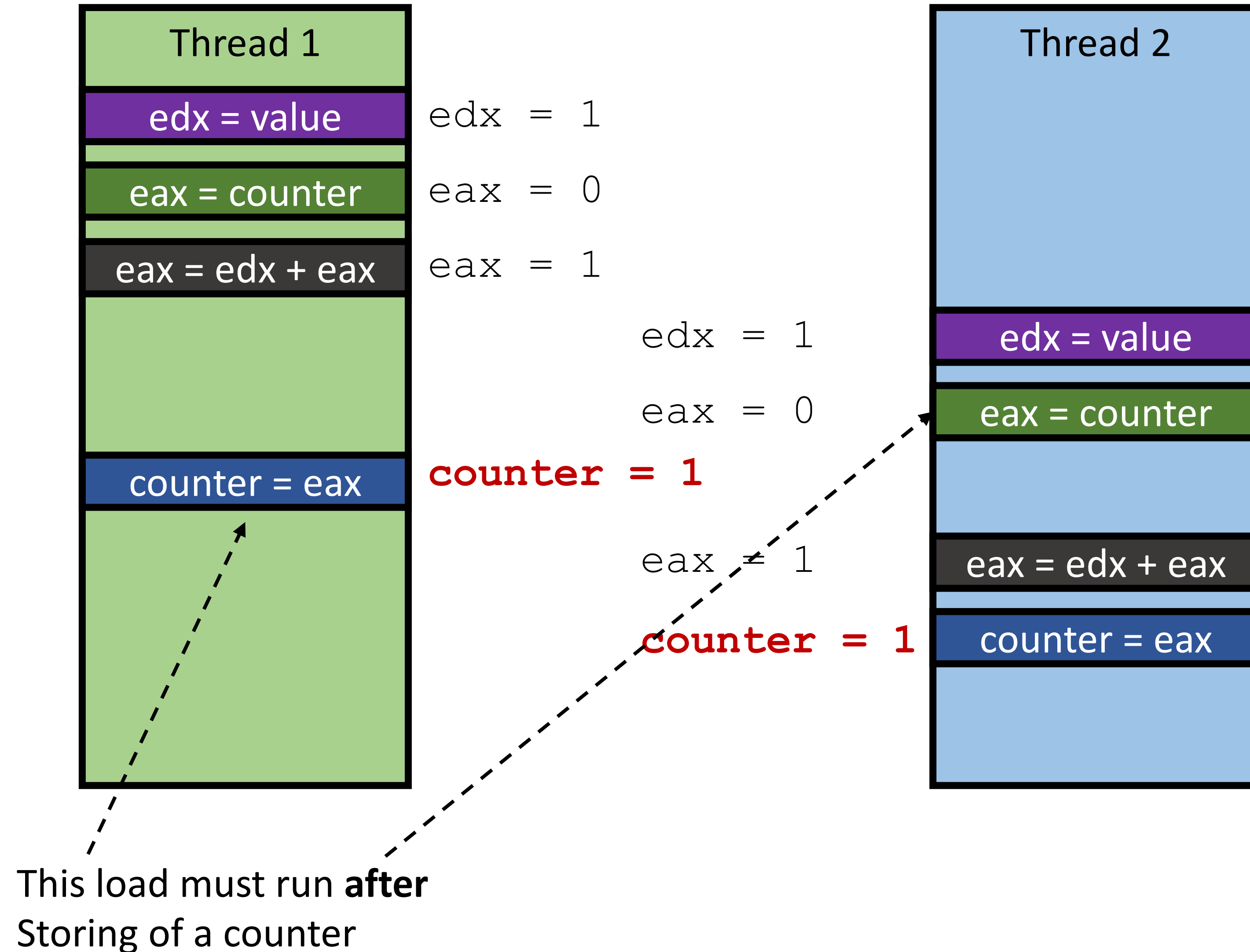
- Assume at start,
  - counter = 0
  - value = 1



Data Race Example (Race cond.)

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

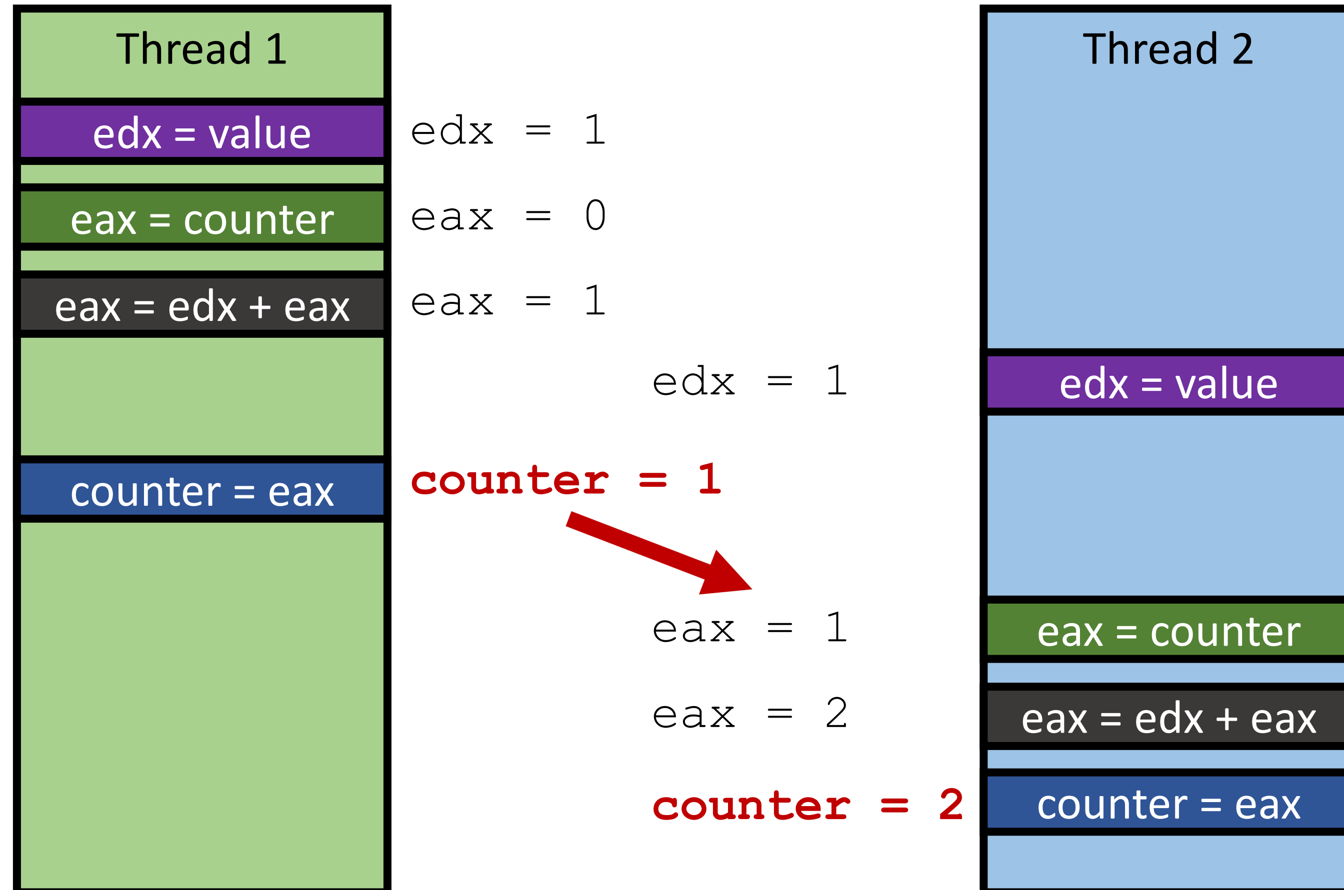
- Assume at start,
  - counter = 0
  - value = 1



Data Race Example (Race cond.)

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

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

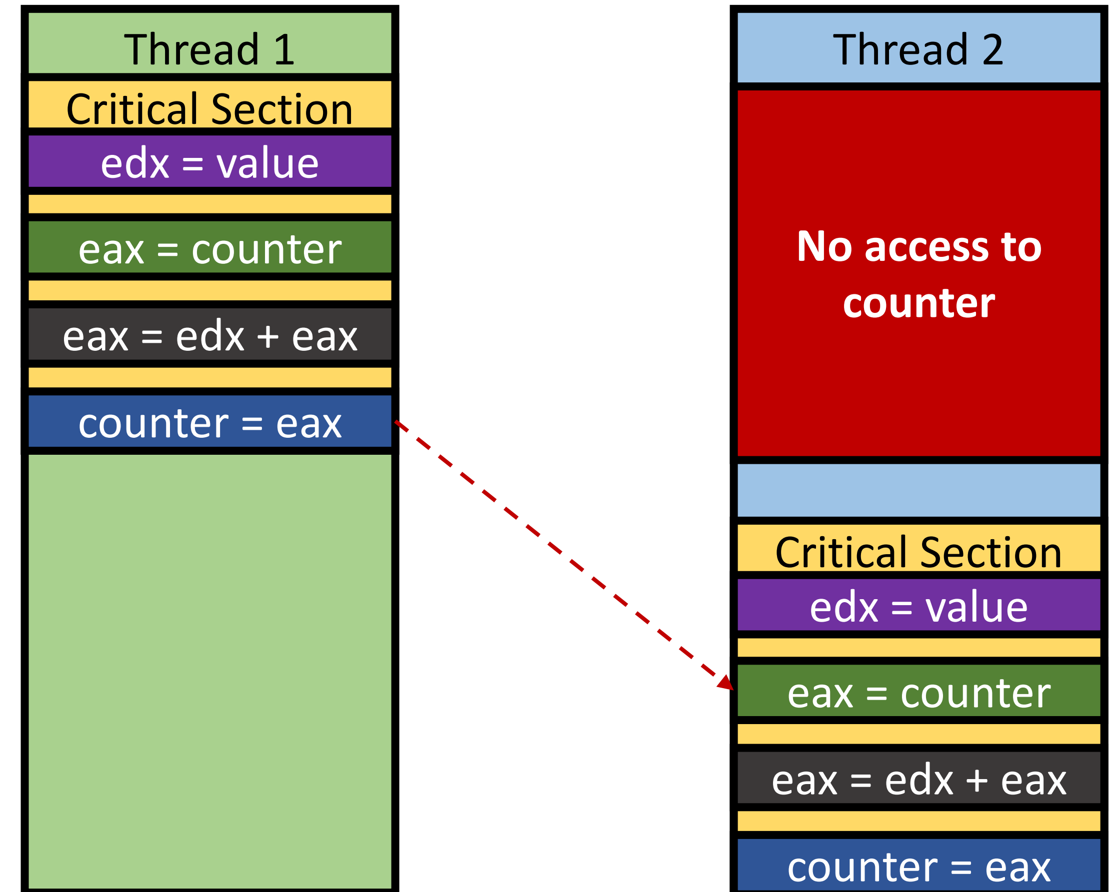


How to Prevent Data Races?

- Root-cause
  - A thread may load the '**previous version**' of shared data (counter = 0)
  - Before the previously running thread properly stores it (counter += 1)
- Store instruction of the previous **thread must finish**
  - before the load instruction of the next thread
- Solution
  - **Make all loads on shared variable wait until previous load-store finishes**
  - **Mutual exclusion**

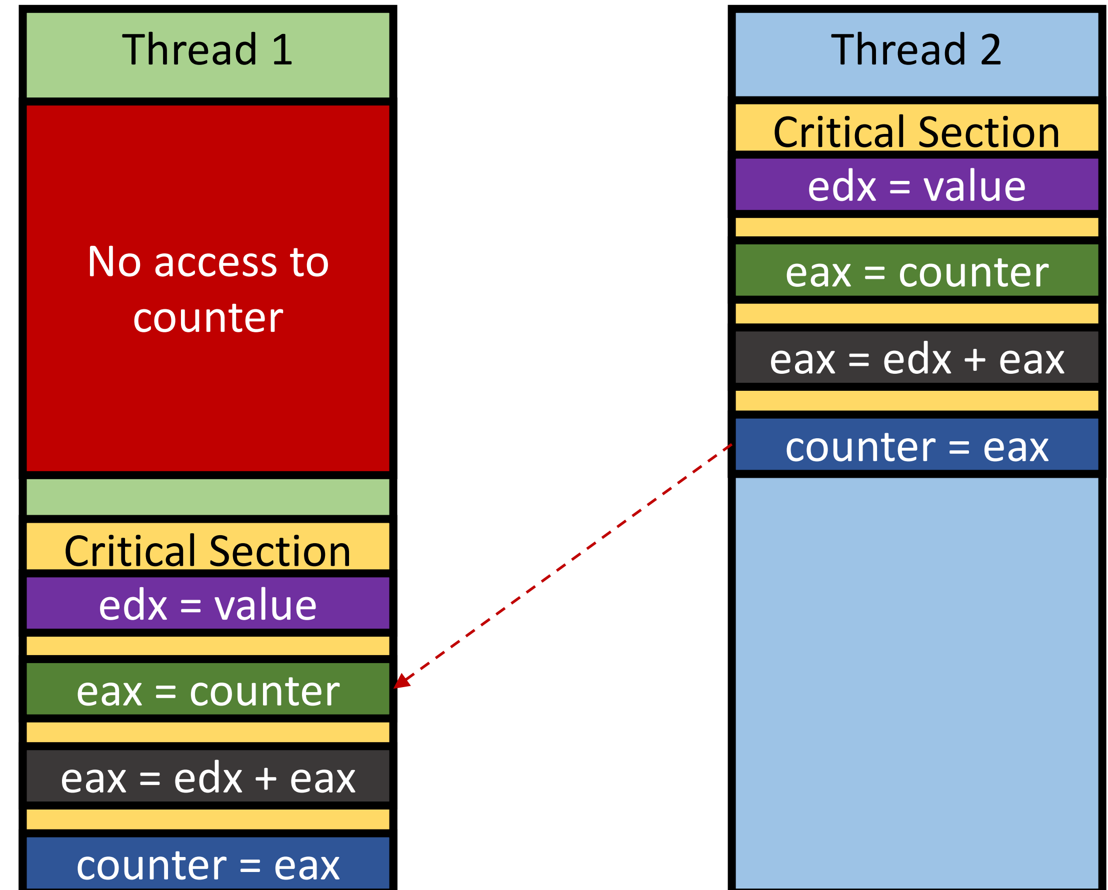
How to Prevent Data Racing?

- **Mutual Exclusion / Critical Section**
  - Combine multiple instructions as a **chunk**
  - Let only one chunk execution run
  - Block other executions
  - Next execution
    - only after finishing all previous critical sections
- `pthread_mutex()` does this for you
  - learn how we can implement locks soon



How to Prevent Data Racing?

- **Mutual Exclusion / Critical Section**
  - Combine multiple instructions as a **chunk**
  - Let only one chunk execution run
  - Block other executions
  - Next execution
    - only after finishing all previous critical sections
- `pthread_mutex()` does this for you
  - learn how we can implement locks soon

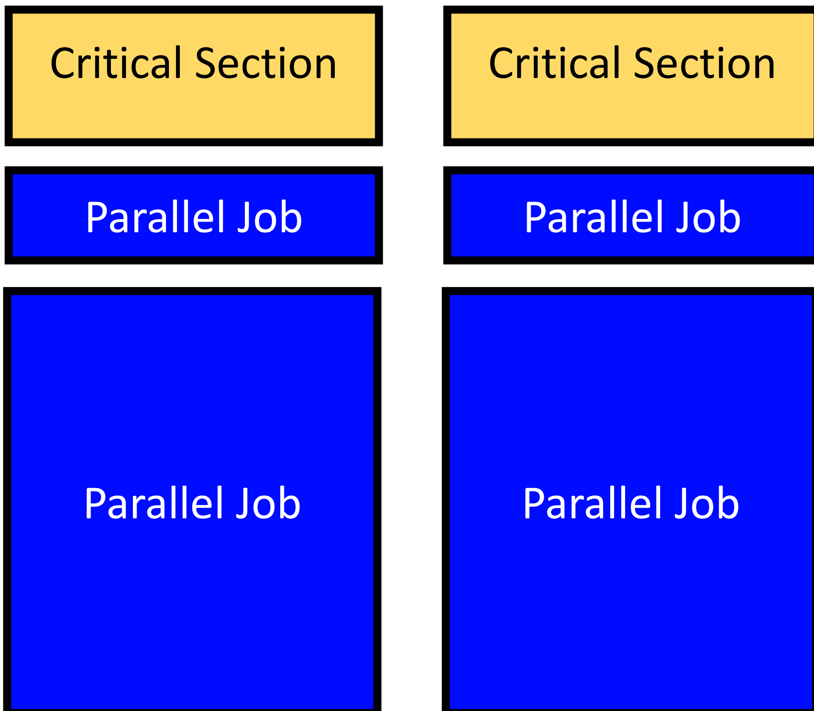


Would Mutex Render Threading Useless?

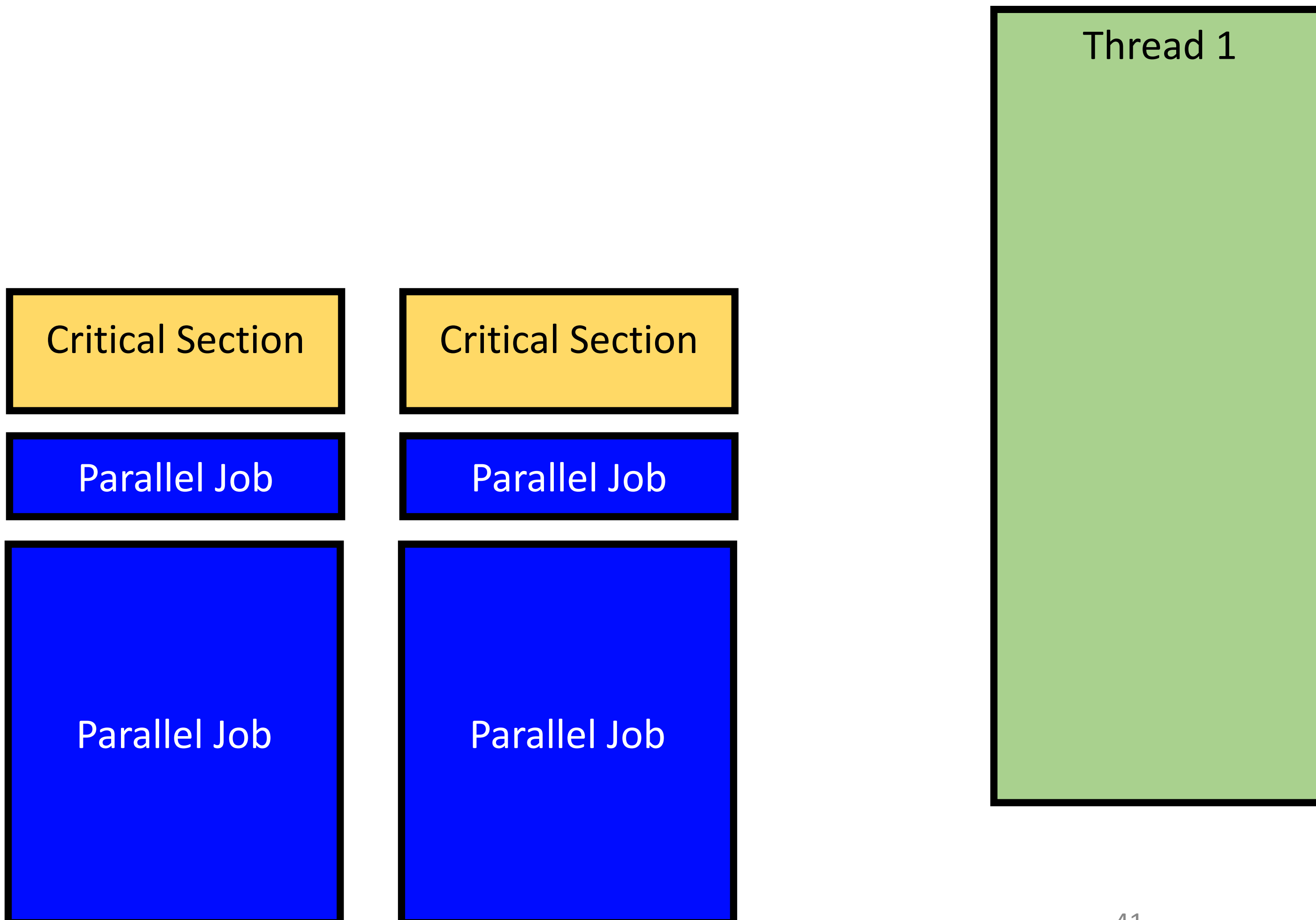
Thread 1
Critical Section
Critical Section
Critical Section
Critical Section
Critical Section

Thread 1	Thread 2
Critical Section	Exclusion
Exclusion	Critical Section
Critical Section	Exclusion
Critical Section	Exclusion
Exclusion	Critical Section

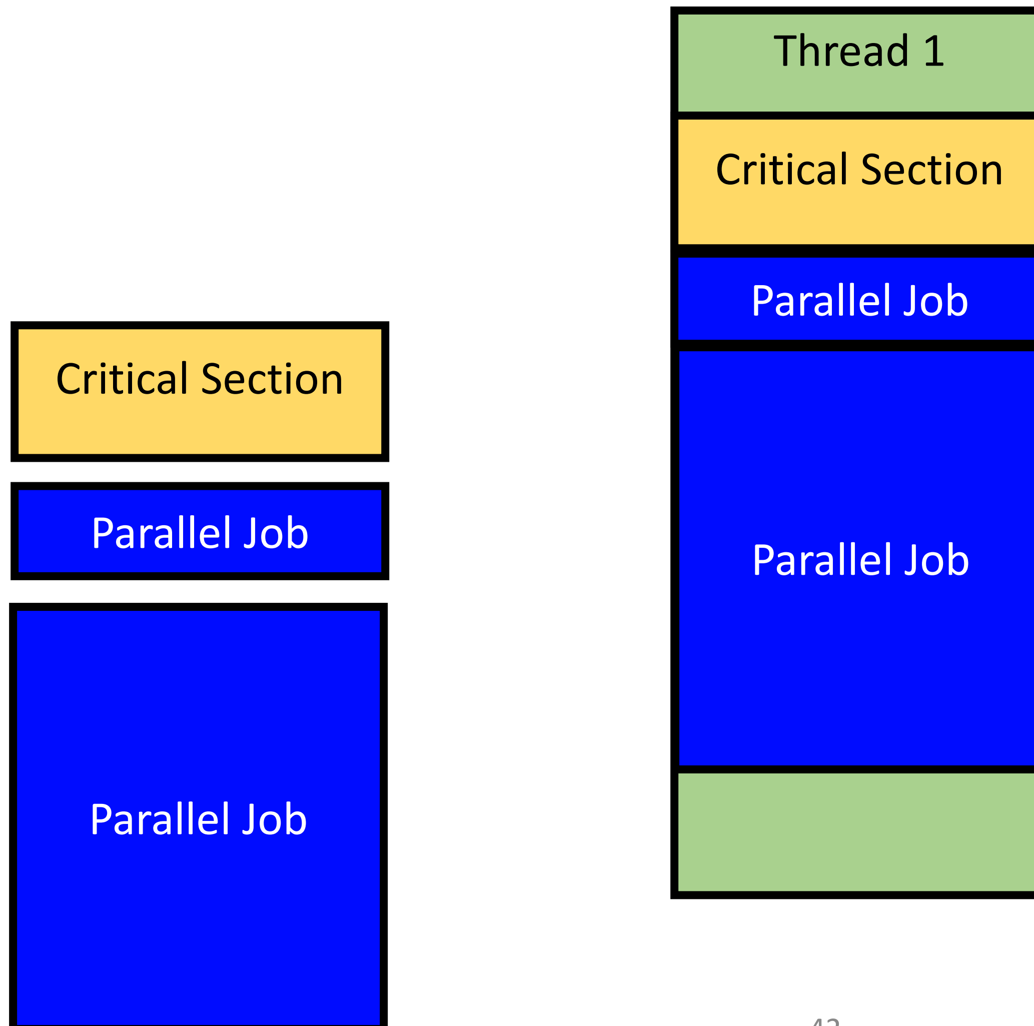
Use Critical Section **Only If Required**



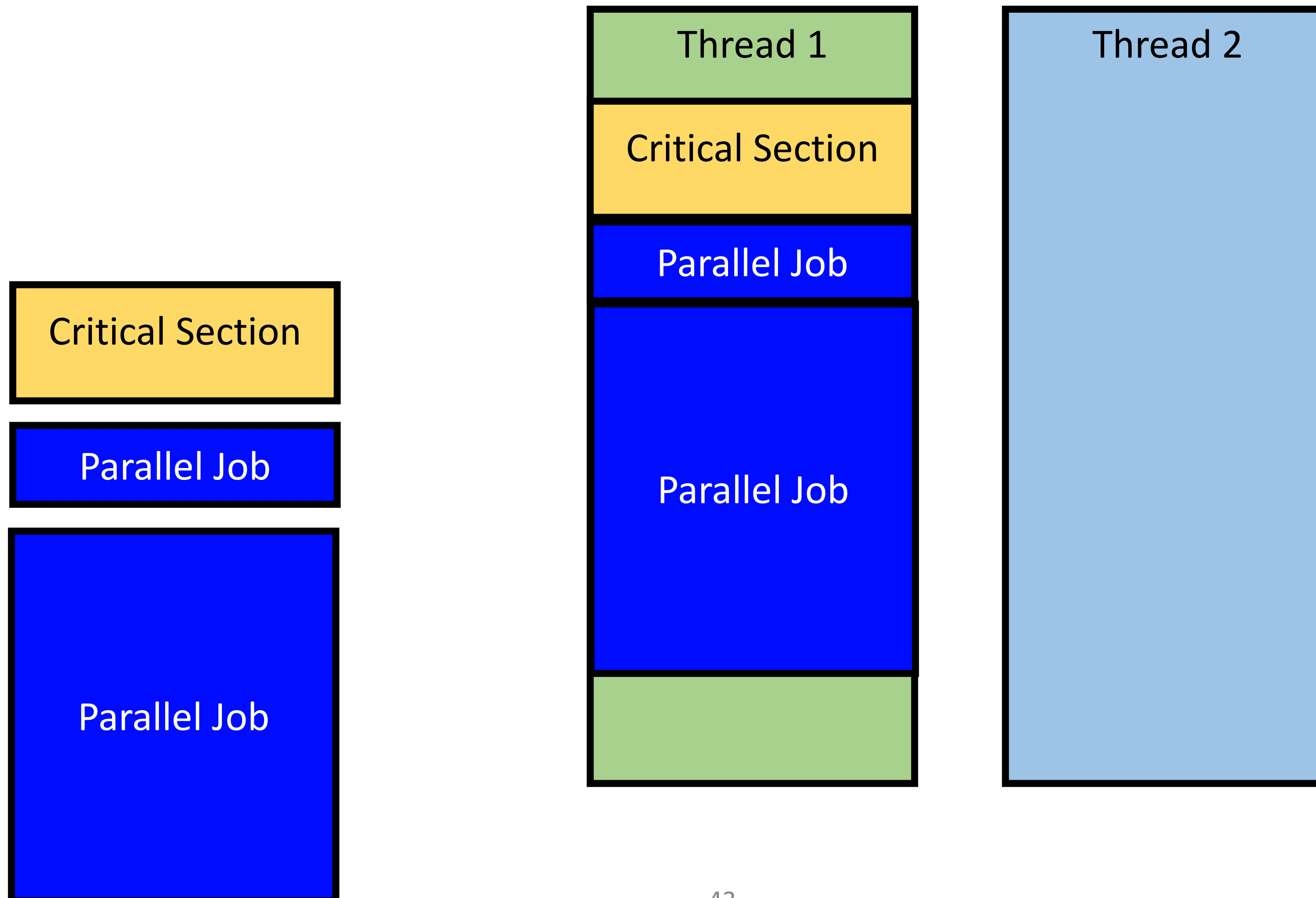
Use Critical Section Only If Required



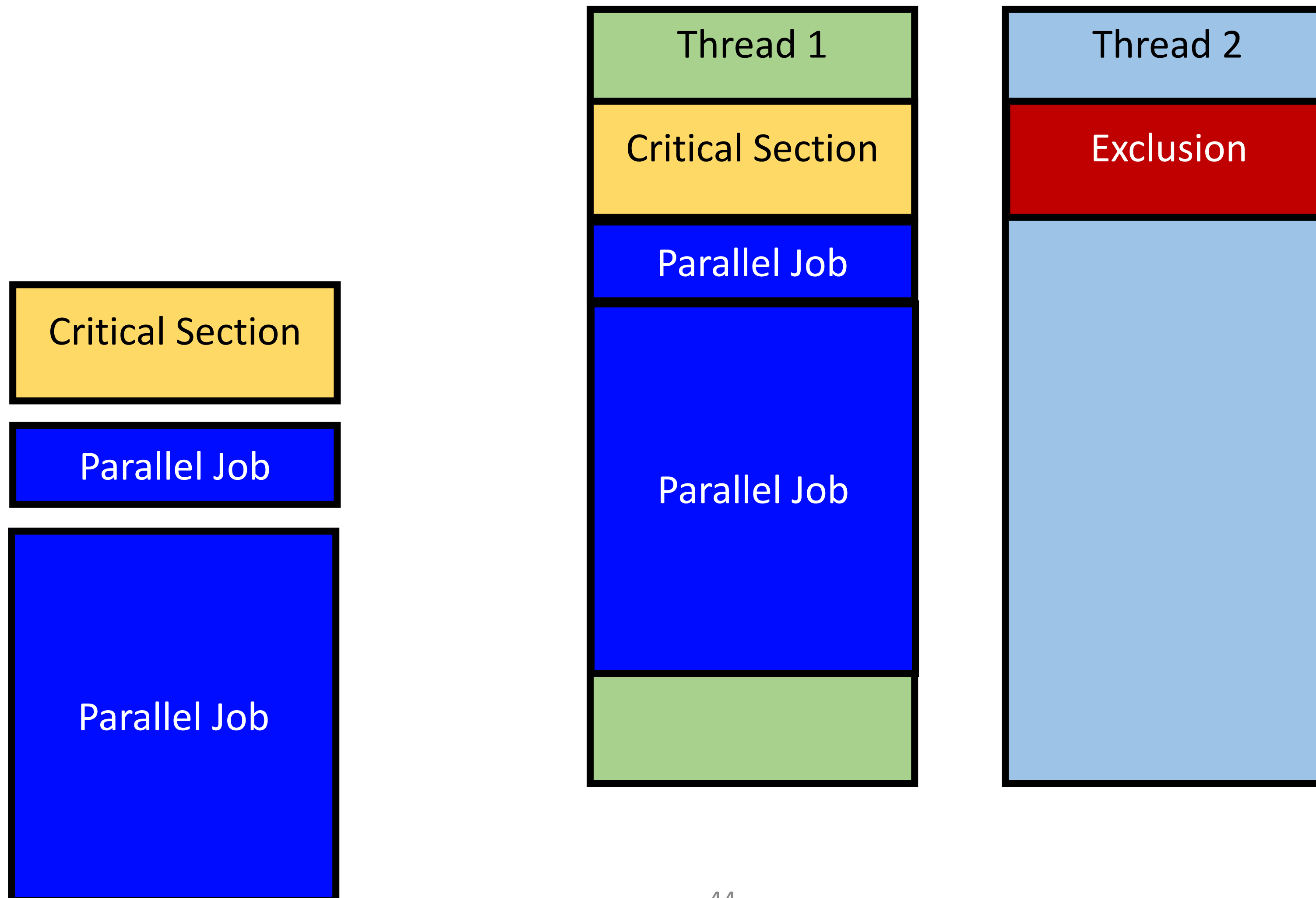
Use Critical Section Only If Required



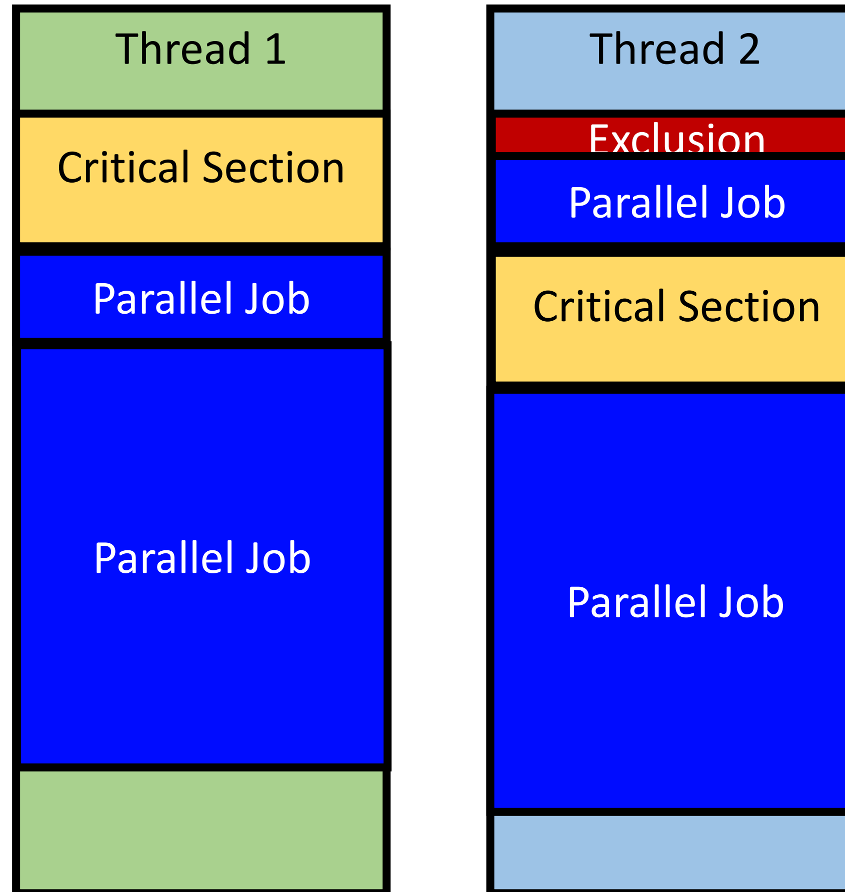
Use Critical Section Only If Required



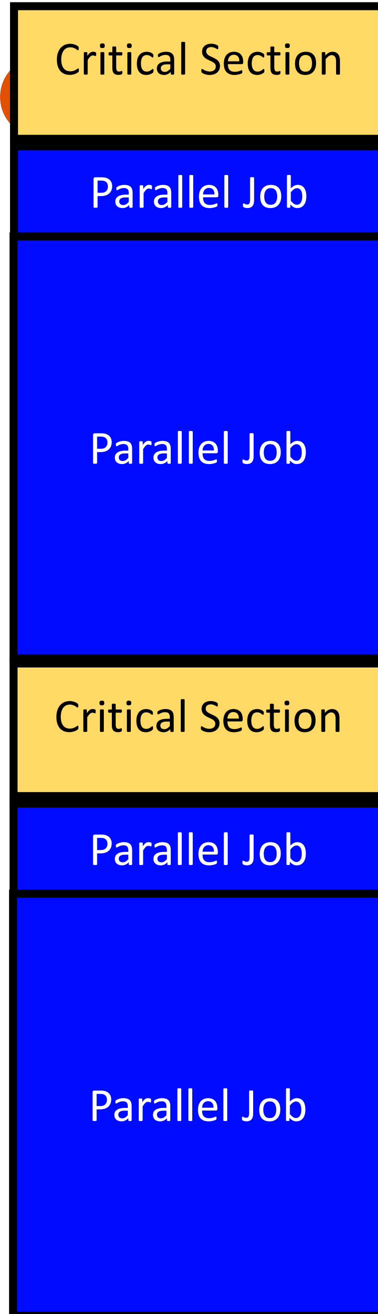
Use Critical Section Only If Required

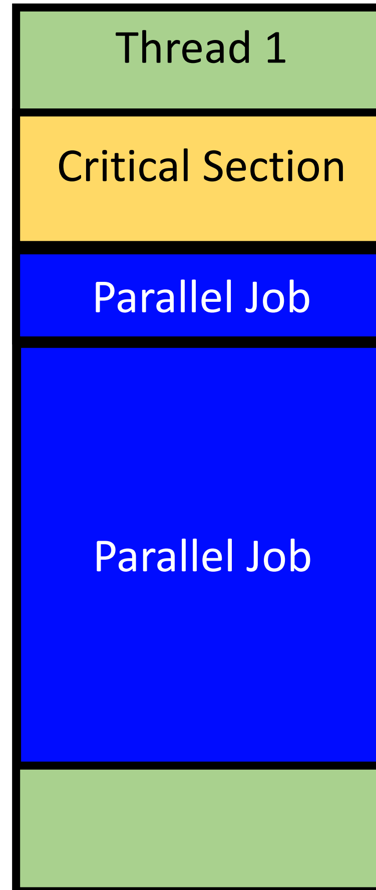


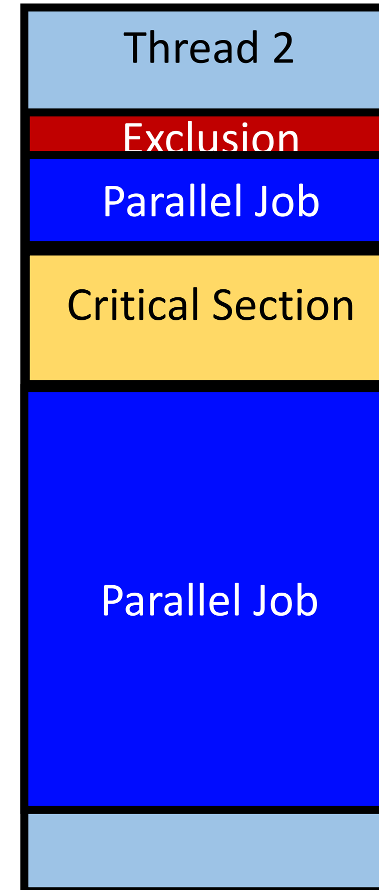
Use Critical Section Only If Required



Use Critical Section Only If Required







Caveat: Apply Mutex only if required

Mutex can synchronize multiple threads and yield **consistent results**	Making entire program a critical section is **meaningless for parallelism**	Apply critical section **as short as possible**
• No read before previous thread stores shared data	• Running time will be same as single-threaded execution	• 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()`**

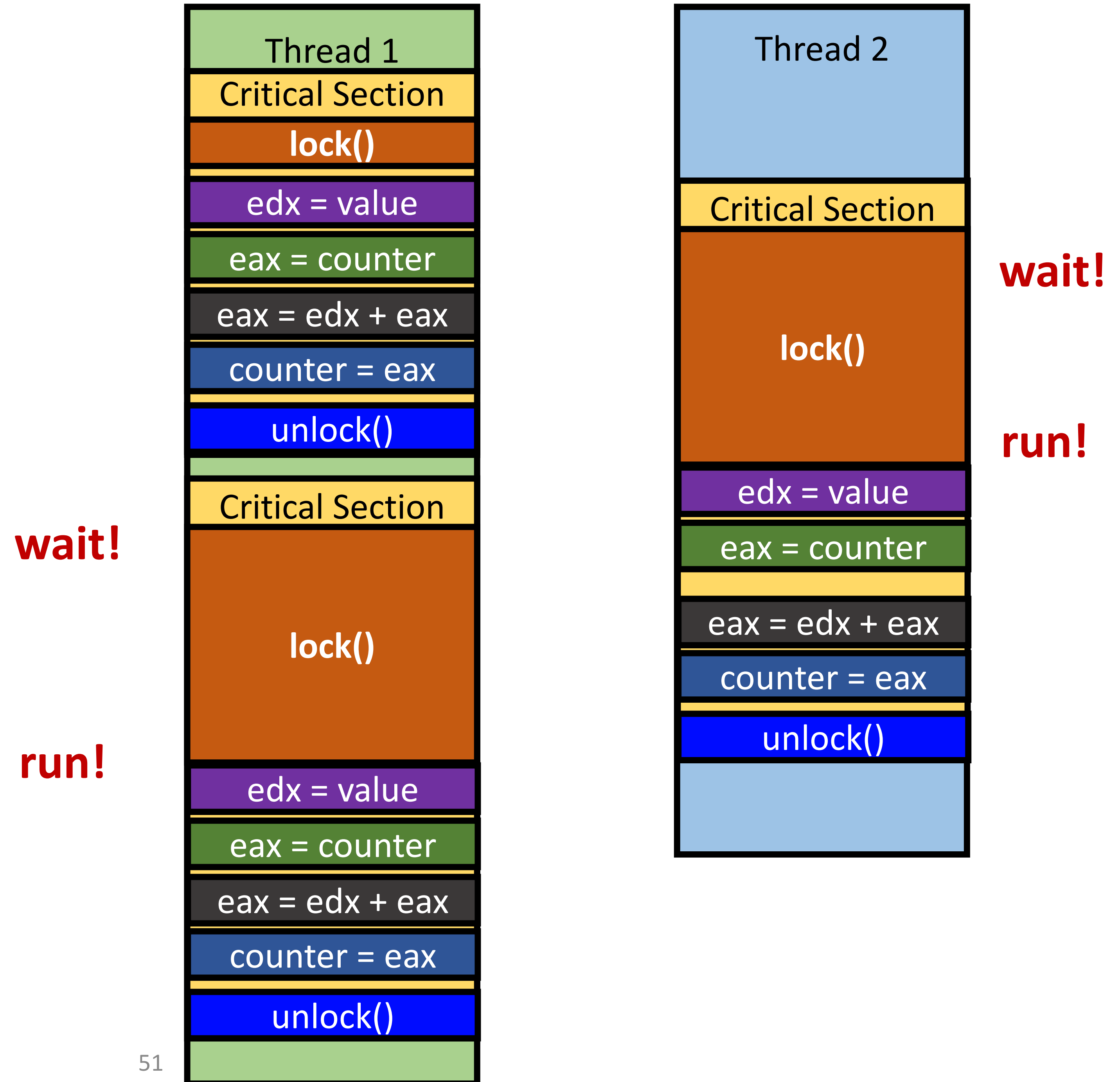
Mutex (Mutual Exclusion)

- **Lock** → prevent others from entering 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

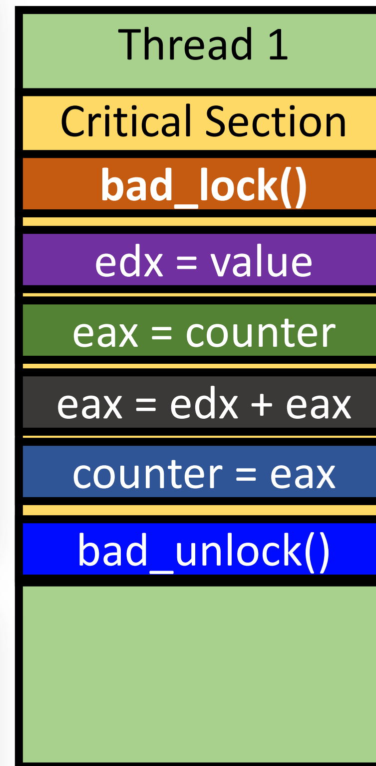


How Can We Implement Locks?

```
void
bad_lock(volatile uint32_t *lock) {
    while (*lock == 1);
    *lock = 1;
}

void
bad_unlock(volatile uint32_t *lock) {
    *lock = 0;
}
```

```
void *
count_bad_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        bad_lock(&lock);
        count += 1;
        bad_unlock(&lock);
    }
}
```

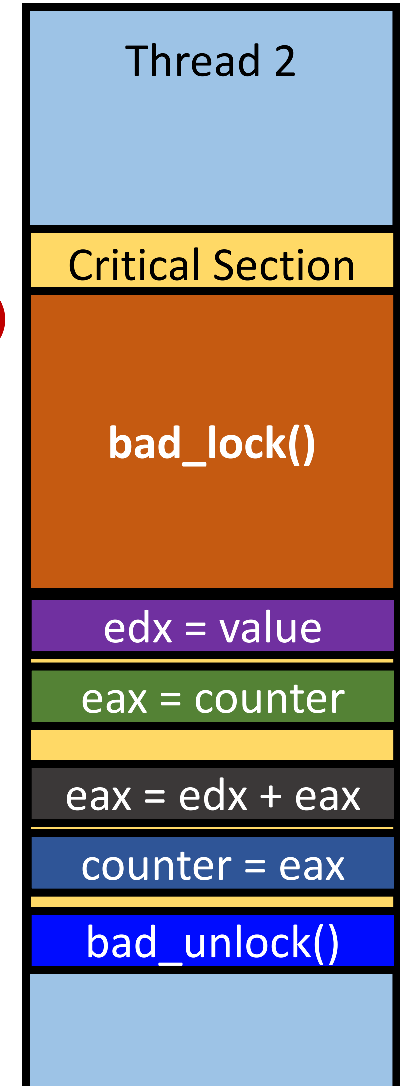
Critical Section



*lock == 0, pass while
*lock = 1 (T1)

*lock == 1, stay in while (T2)

*lock = 0 (T1)
*lock == 0, break while (T2)
*lock = 1 (T2)



Unfortunately, only works in single CPU environment

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