Multi Threading and Synchronization

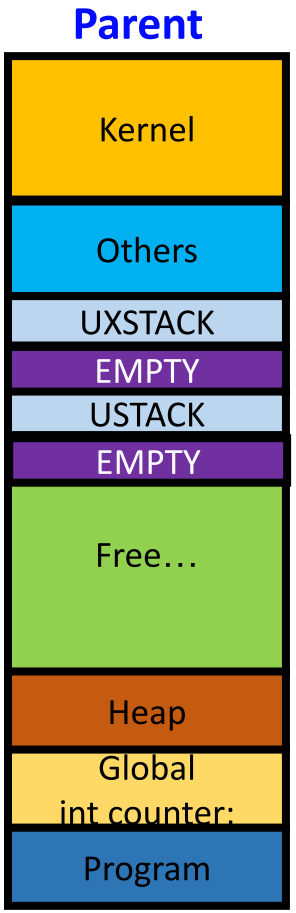
ECE 469, Feb 22

Aravind Machiry

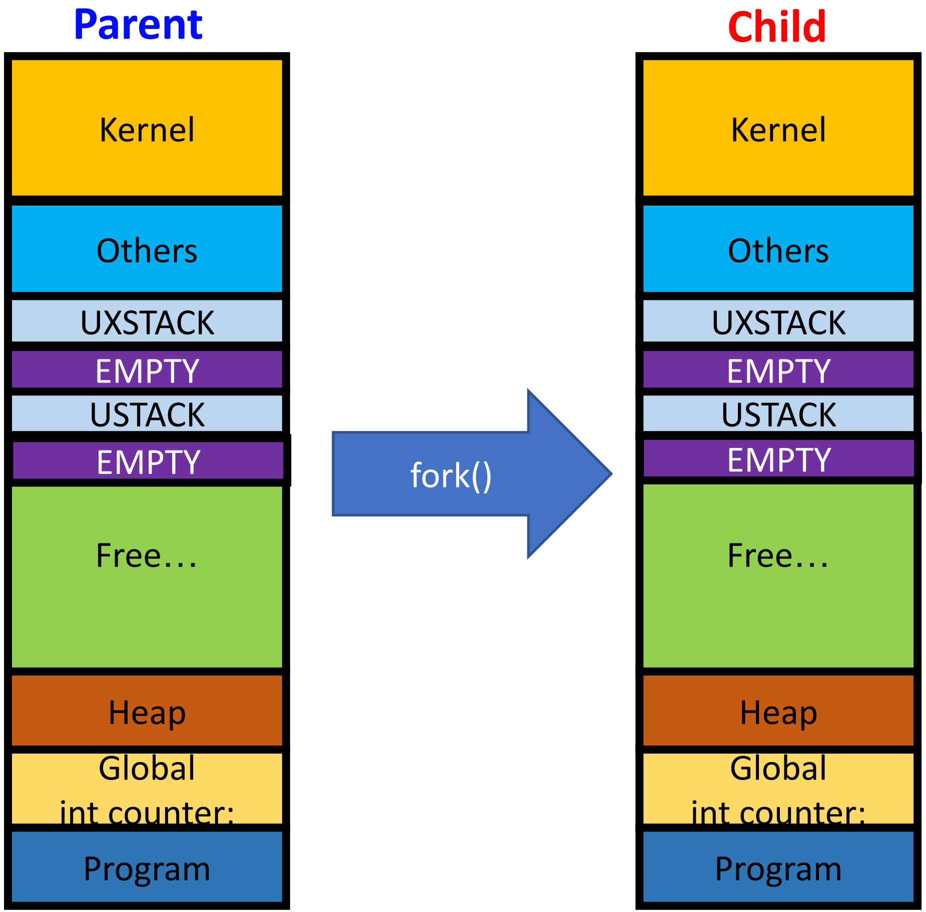
Web Server Example

- How does a web server handle 1 request?
- A web server needs to handle many concurrent requests
- Solution 1:
  - Have the parent process fork as many processes as needed
  - Processes communicate with each other via inter-process communication

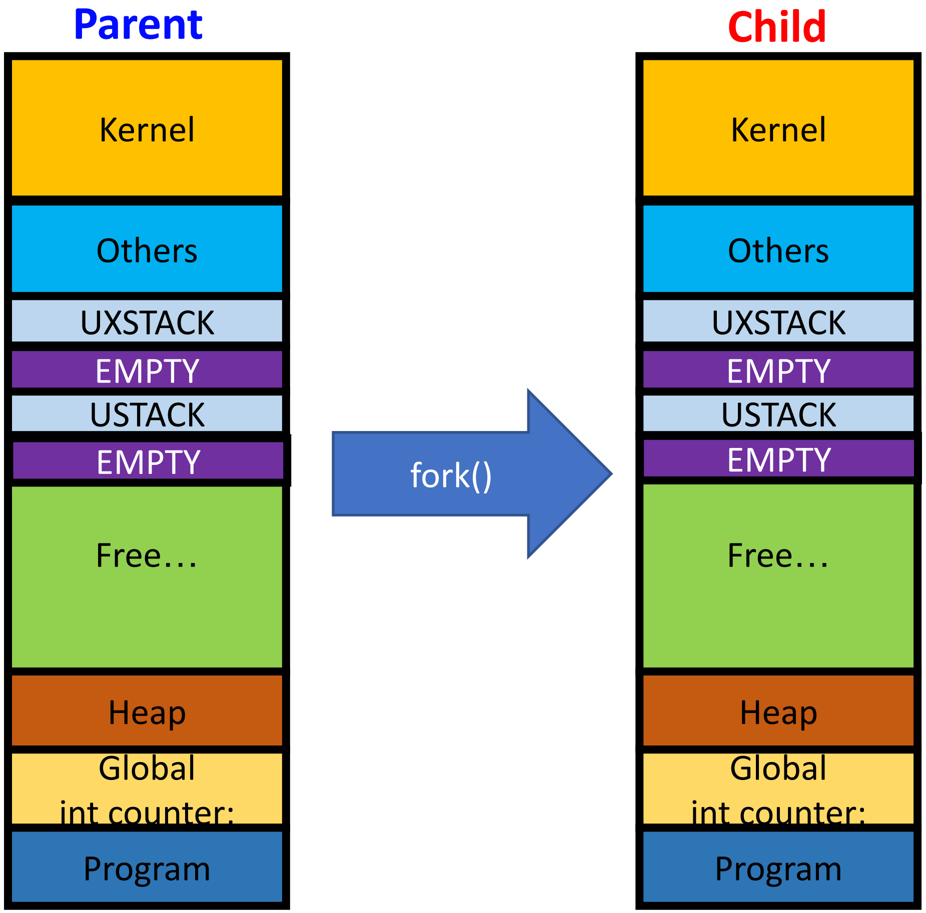
Multiple Processes



Multiple Processes



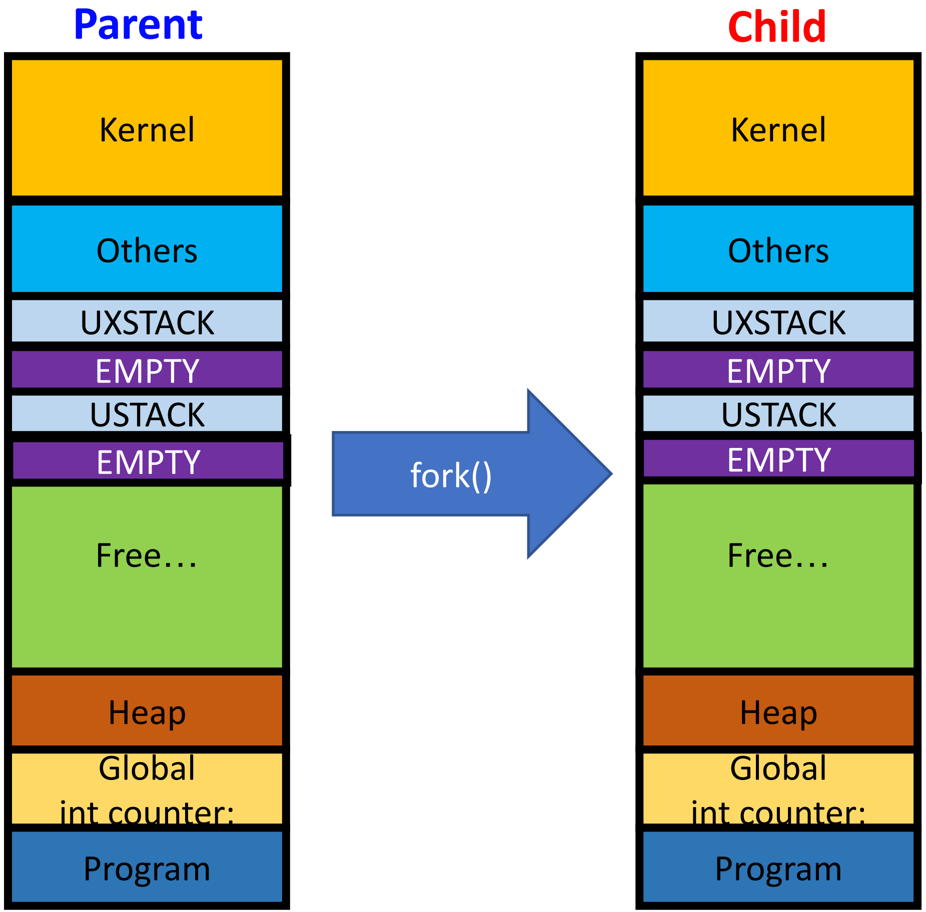
Multiple Processes



Fork() creates new process by copying memory space

Process creates a new PRIVATE memory space

Multiple Processes



Fork() creates new process by copying memory space

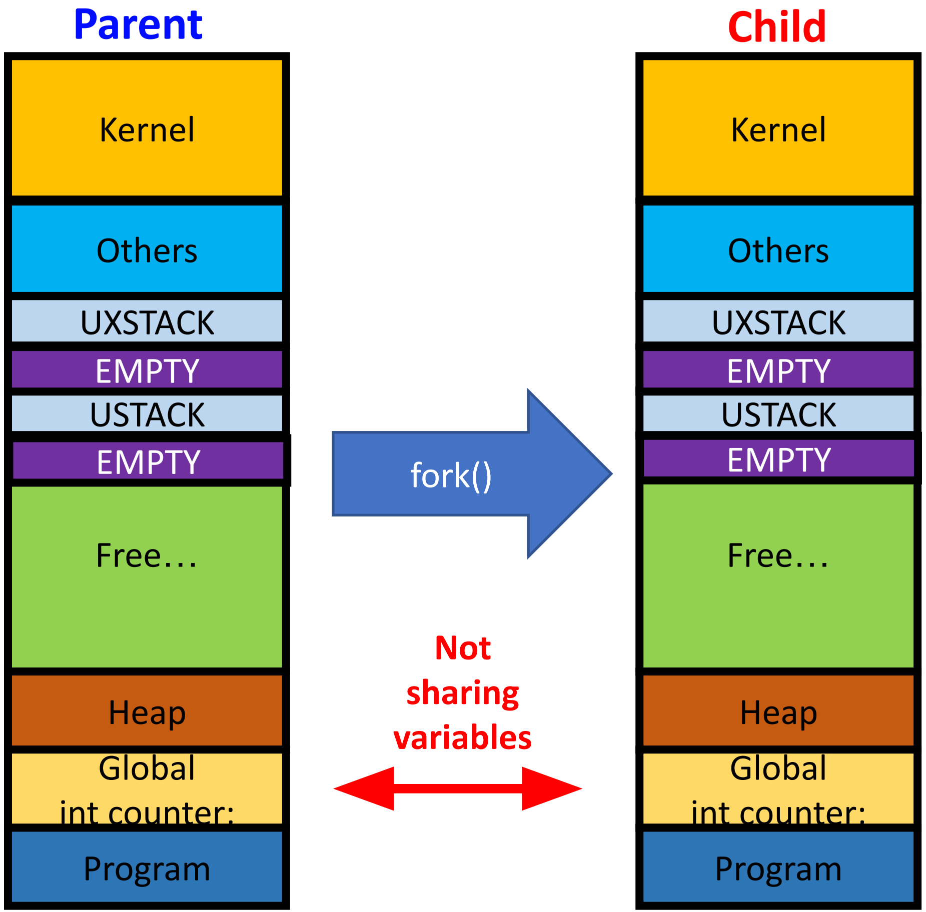
```c
#include <stdio.h>
#include <unistd.h>

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);
}
```

Multiple Processes



Fork() creates new process by copying 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
```

How do Process communicate?

- At process creation time
  - Parents get one chance to pass everything at fork()

- OS provides generic mechanisms to communicate
  - Shared Memory: multiple processes can read/write same physical portion of memory; implicit channel
    - System call to declare shared region
    - No OS mediation required once memory is mapped

  - Message Passing: explicit communication channel provided through send()/receive() system calls
    - A system call is required

How do Process communicate?

- IPC is, in general, expensive due to the need for system calls
  - Although many OSes have various forms of lightweight IPC

The Soul of a Process

- But all the processes in the web-server are **cooperating**!
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)

- What don’t they share?
  - Each has its own execution state: PC, SP, and registers

The Soul of a Process

- Key idea: Why don’t we **separate the concept of a process from its execution state**?
  - Process: address space, privileges, resources, etc.
  - Execution state: PC, SP, registers

- Exec state also called thread of control, or thread

Threads

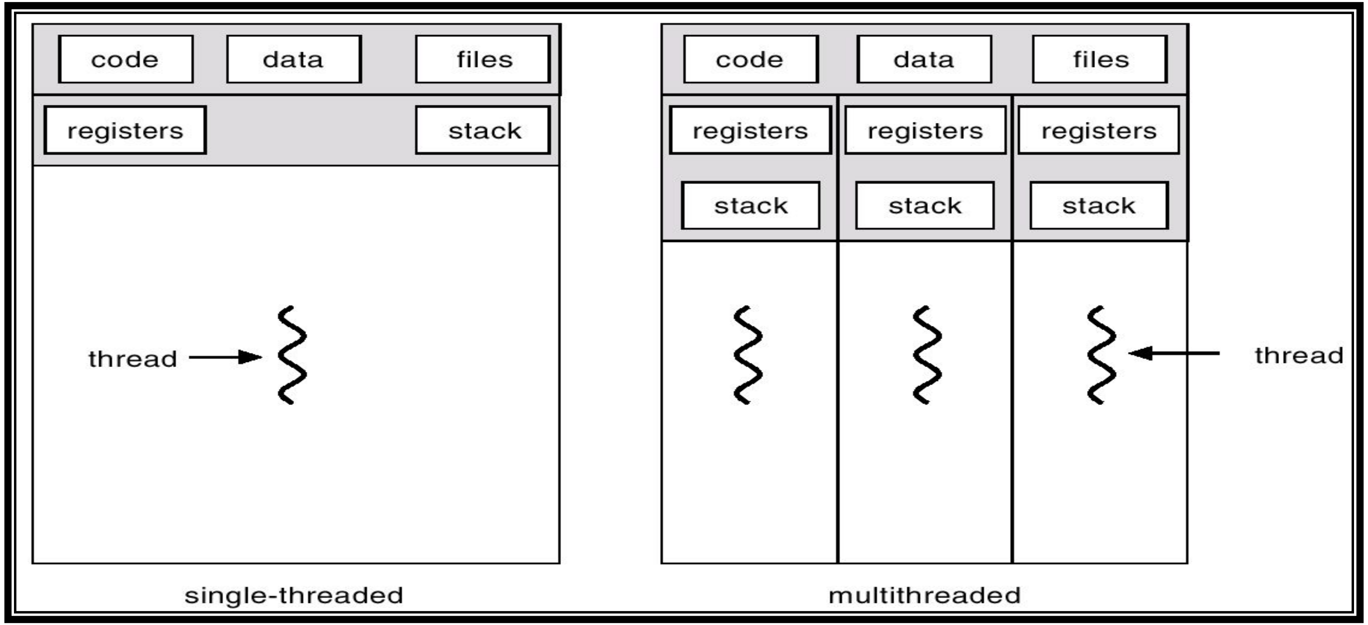
- Separate the concepts of a "thread of control" (PC, SP, registers) from the rest of the process (address space, resources, accounting, etc.)
- Modern OSes support two entities:
  - the *task* (process), which defines an address space, a resource container, accounting info
  - the *thread* (lightweight process), which defines a single sequential execution stream within a task (process)

Threads vs. Process

- There can be several threads in a single address space

- Threads are the unit of scheduling; tasks are containers (address space, other shared resources) in which threads execute

Single threaded v/s multithreaded



What differs in threads of a process?

- A.K.A User Environment (JOS)

- Process management info
  - State (ready, running, blocked)
  - PC & Registers, parents, etc
  - CPU scheduling info (priorities, etc.)

- Memory management info
  - Segments, page table, stats, etc
  - Code, data, heap, execution stack

- I/O and file management
  - Communication ports, directories, file descriptors, etc

What differs in threads of a process?

- A.K.A User Environment (JOS)

- Process management info
  - State (ready, running, blocked)
  - PC & Registers, parents, etc
  - CPU scheduling info (priorities, etc.)

- Memory management info
  - Segments, page table, stats, etc
  - Code, data, heap, execution stack

- I/O and file management
  - Communication ports, directories, file descriptors, etc

Thread Control Block

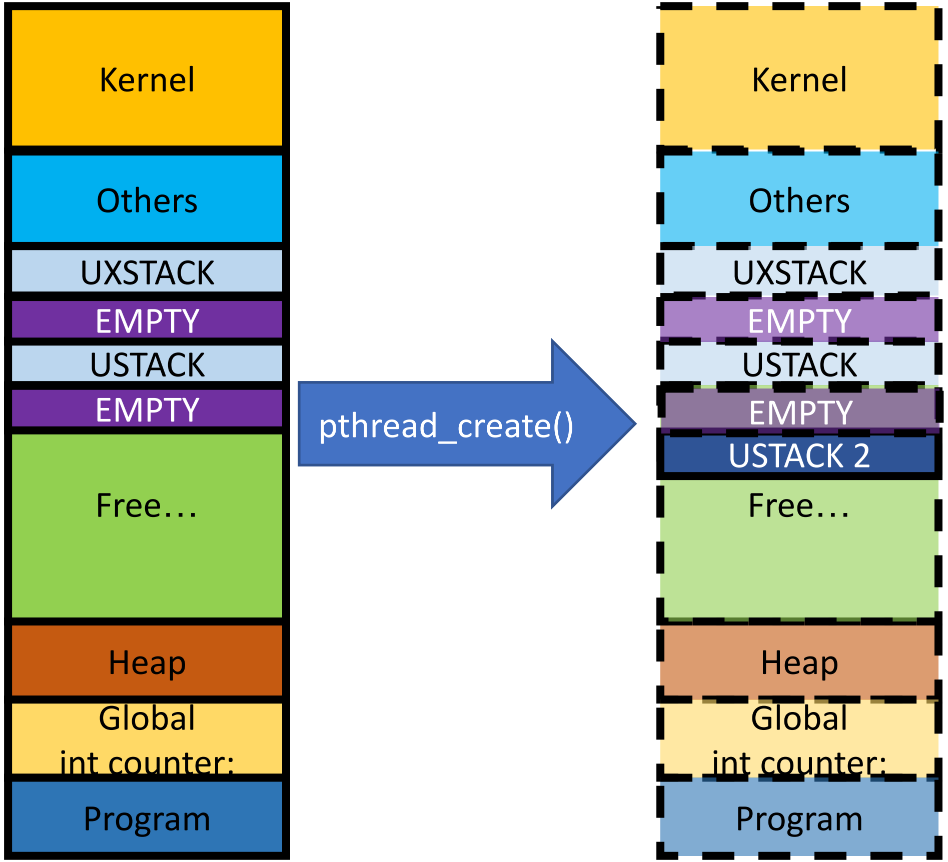
- Shared information
  - Process info: parent process
  - Memory: code/data segments, page table, and stats
  - I/O and file: comm ports, open file descriptors

- Private state
  - State (ready, running and blocked)
  - PC, Registers
  - Execution stack

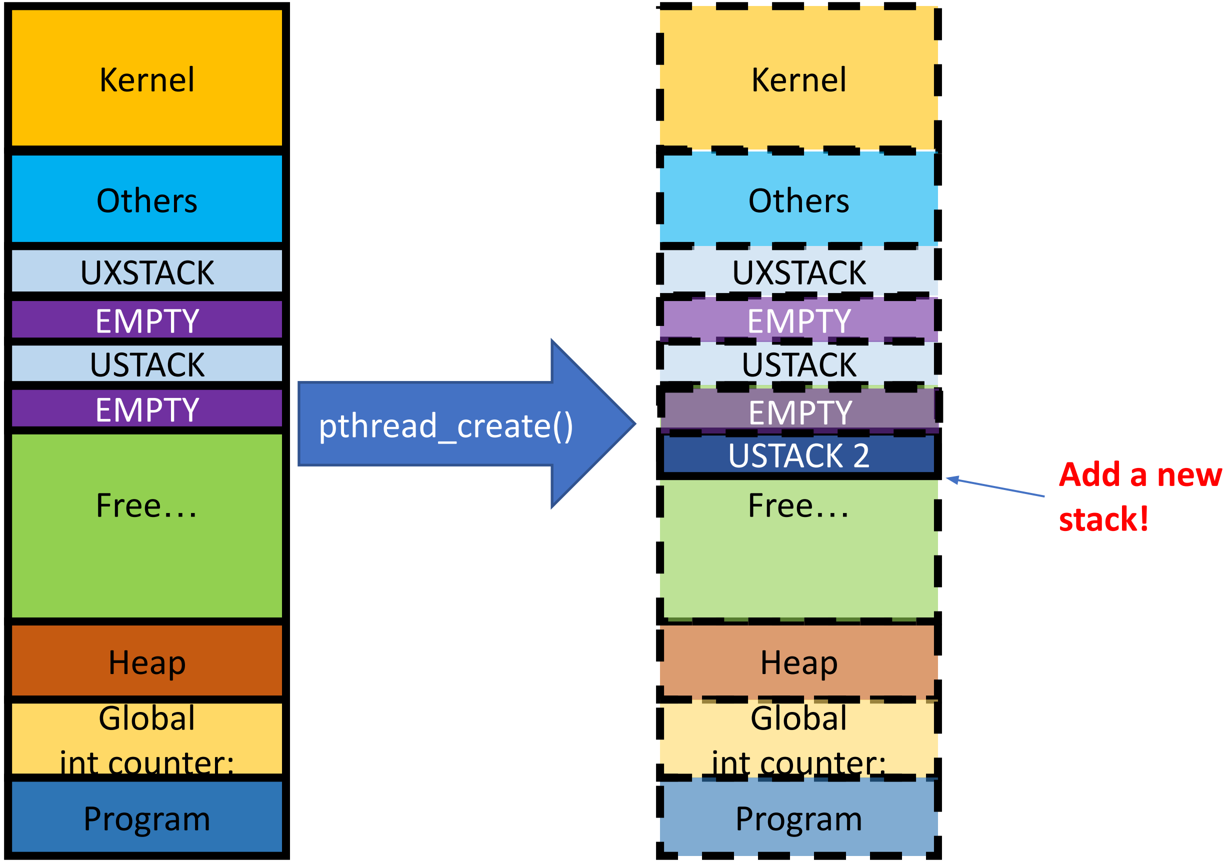
Threads


Kernel
Others
UXSTACK
EMPTY
USTACK
EMPTY
Free…
Heap
Global
int counter;
Program

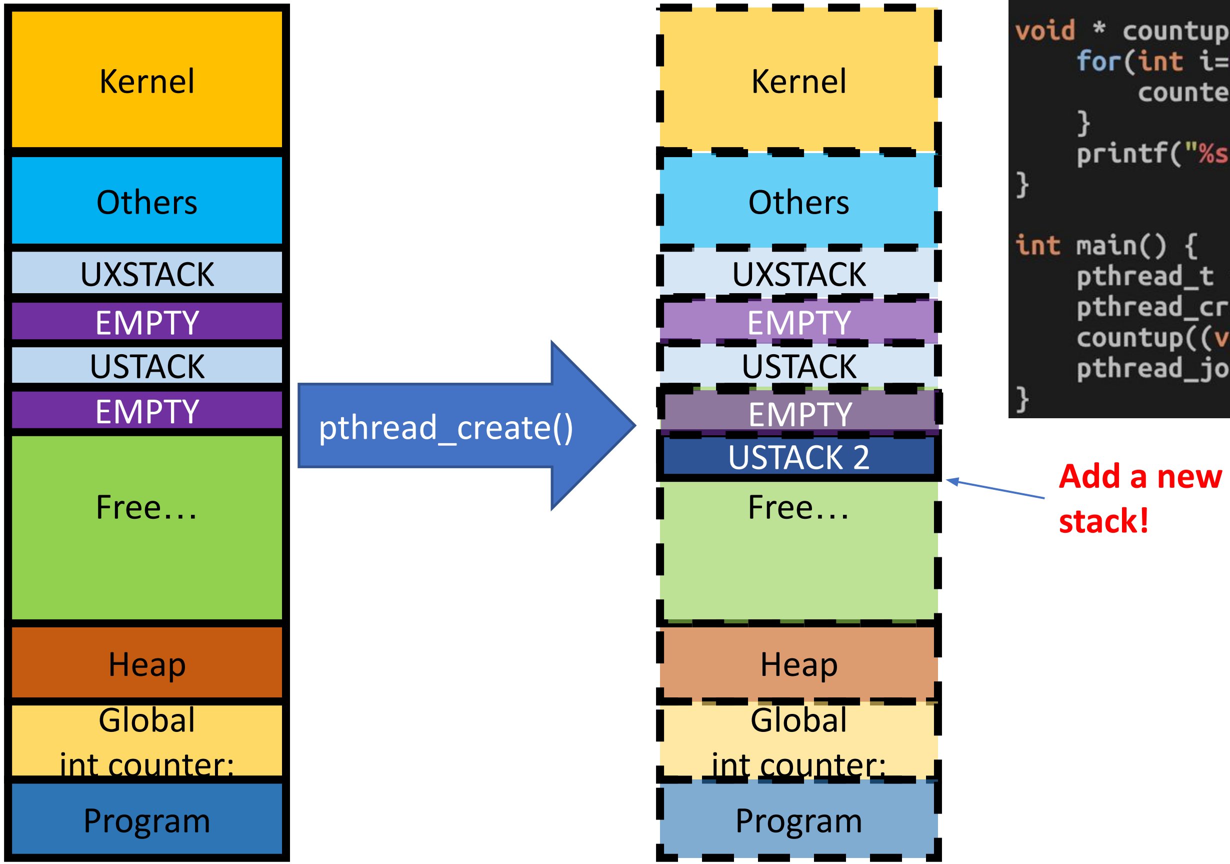
Threads


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

Threads


Kernel
Others
UXSTACK
EMPTY
USTACK
EMPTY
Free…
Heap
Global
int counter;
Program
pthread_create()
Kernel
Others
UXSTACK
EMPTY
USTACK
EMPTY
USTACK 2
Free…
Heap
Global
int counter;
Program
Add a new stack!

Threads



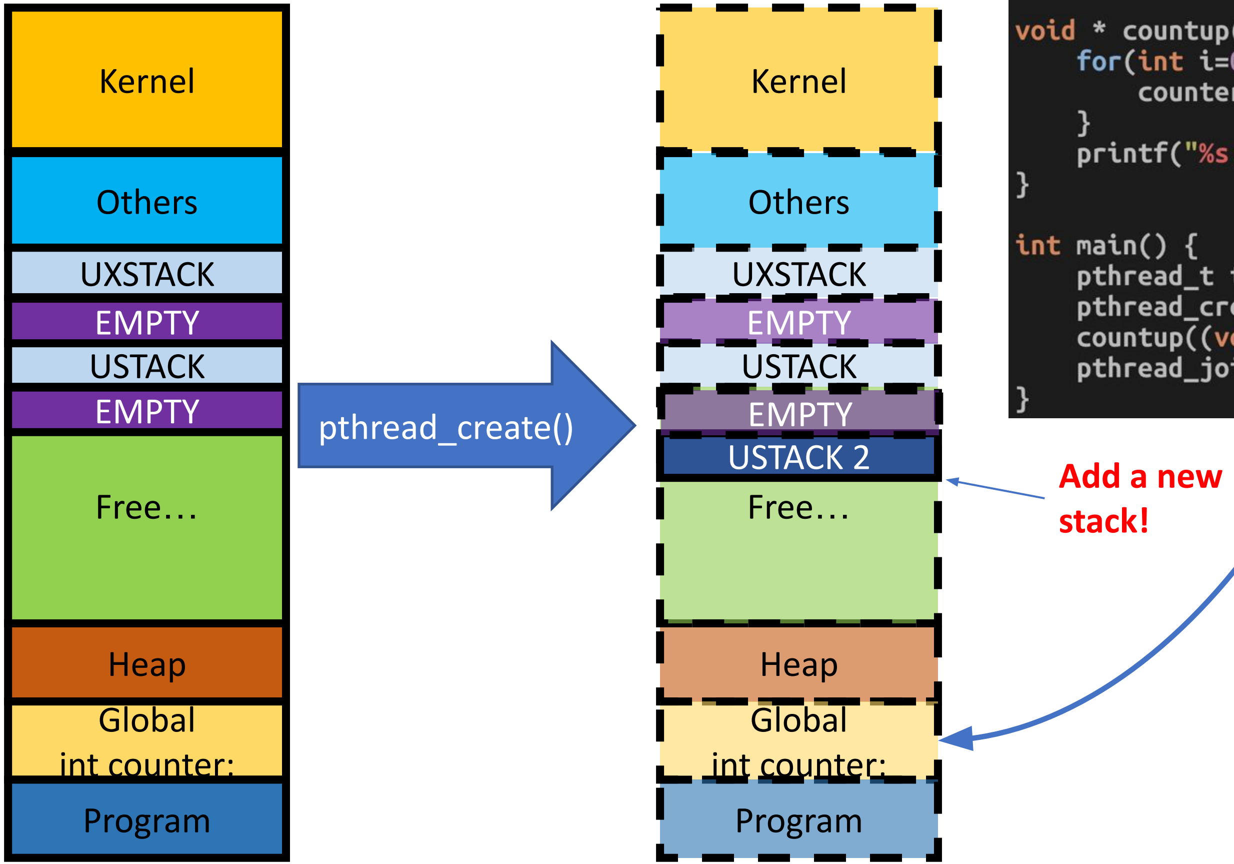

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

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);
}
```

Threads



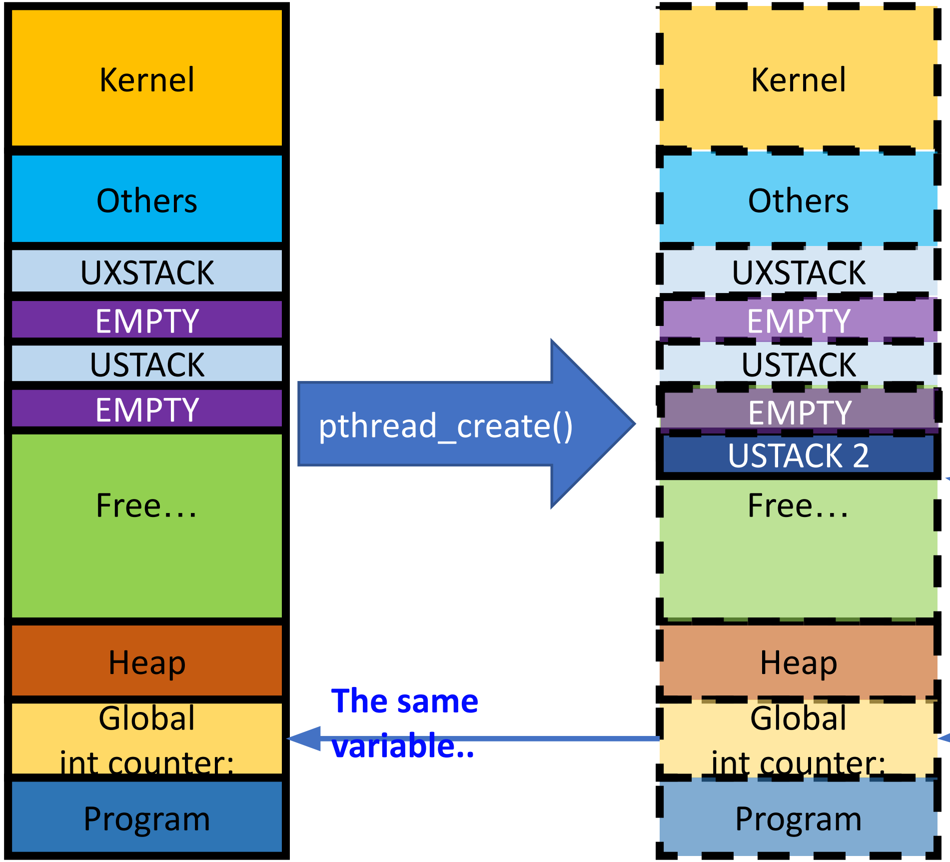

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

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);
}
```

Threads




```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);
}
```

Add a new stack!

Adding value..

Programming with Threads

- Flexible, but error-prone, since there no protection between threads
  - In C/C++,
    - automatic variables are private to each thread
    - global variables and dynamically allocated memory (malloc) are shared
- Need synchronization!

The need for synchronization!

- Cooperating processes may share data via
  - shared address space (code, data, heap) by using threads
  - Files
  - (Sending messages)
- What can happen if processes try to access shared data (address) concurrently?
  - Sharing bank account with sibling:

    At 3pm: If (balance > $10) withdraw $10
- How hard is the solution?

"Too much milk" Problem

Person A

1. Look in fridge: out of milk

2. Leave for Walmart

5. Arrive at Walmart

6. Buy milk

7. Arrive home

Person B

3. Look in fridge: out of milk

4. Leave for Walmart

8. Arrive at Walmart

9. Buy milk

10. Arrive home

- How to put in a locking mechanism?

Possible Solution 1

Person A

```
if ( noMilk ) {
  if (noNote) {
    leave note;
    buy milk;
    remove note;
  }
}
```

Person B

```
if ( noMilk ) {
  if (noNote) {
    leave note;
    buy milk;
    remove note;
  }
}
```

Will this work?

Person A

```
if ( noMilk ) {
  if (noNote) {
    leave note;
    buy milk;
    remove note;
  }
}
```

Person B

```
if ( noMilk ) {
  if (noNote) {
    leave note;
    buy milk;
    remove note;
  }
}
```

Will this work?

Person A

```
1.if ( noMilk ) {
   2.if (noNote) {
     5.leave note;
     buy milk;
     remove note;
   }
}
```

Person B

```
3.if ( noMilk ) {
   4.if (noNote) {
     6.leave note;
     buy milk;
     remove note;
   }
}
```

- Process can get context switched after checking milk and note, but before leaving note

Why does this work for humans?

- Human can perform *test* (look for other person & milk) and *set* (leave note) at the same time.

Possible Solution 2

Person A

```
leave noteA
if (no noteB) {
  if (noMilk) {
    buy milk
  }
}
remove noteA
```

Person B

```
leave noteB
if (no noteA) {
  if (noMilk) {
    buy milk
  }
}
remove noteB
```

Will this work?

Person A

```
leave noteA
if (no noteB) {
  if (noMilk) {
    buy milk
  }
}
remove noteA
```

Person B

```
leave noteB
if (no noteA) {
  if (noMilk) {
    buy milk
  }
}
remove noteB
```

Will this work?

Person A

```
leave noteA
if (no noteB) {
   if (noMilk) {
      buy milk
   }
}
remove noteA
```

Person B

```
leave noteB
if (no noteA) {
   if (noMilk) {
      buy milk
   }
}
remove noteB
```

- We may not have Milk: Both process can leave note and skip buying milk

Possible Solution 3

Process A

```
leave noteA
while (noteB)
  do nothing;
if (noMilk)
  buy milk;
remove noteA
```

Process B

```
leave noteB
if (noNoteA) {
  if (noMilk) {
    buy milk
  }
}
remove noteB
```

Will this work?

Process A

```
leave noteA
while (noteB)
   do nothing;
if (noMilk)
   buy milk;
remove noteA
```

Process B

```
leave noteB
if (noNoteA) {
   if (noMilk) {
      buy milk
   }
}
remove noteB
```

Works, but complicated!

Process A

```
leave noteA
while (noteB)
  do nothing;
if (noMilk)
  buy milk;
remove noteA
```

Process B

```
leave noteB
if (noNoteA) {
  if (noMilk) {
    buy milk
  }
}
remove noteB
```

- A's code is different from B's
- busy waiting is a waste

How can we solve this?

- Root cause: Data Race

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

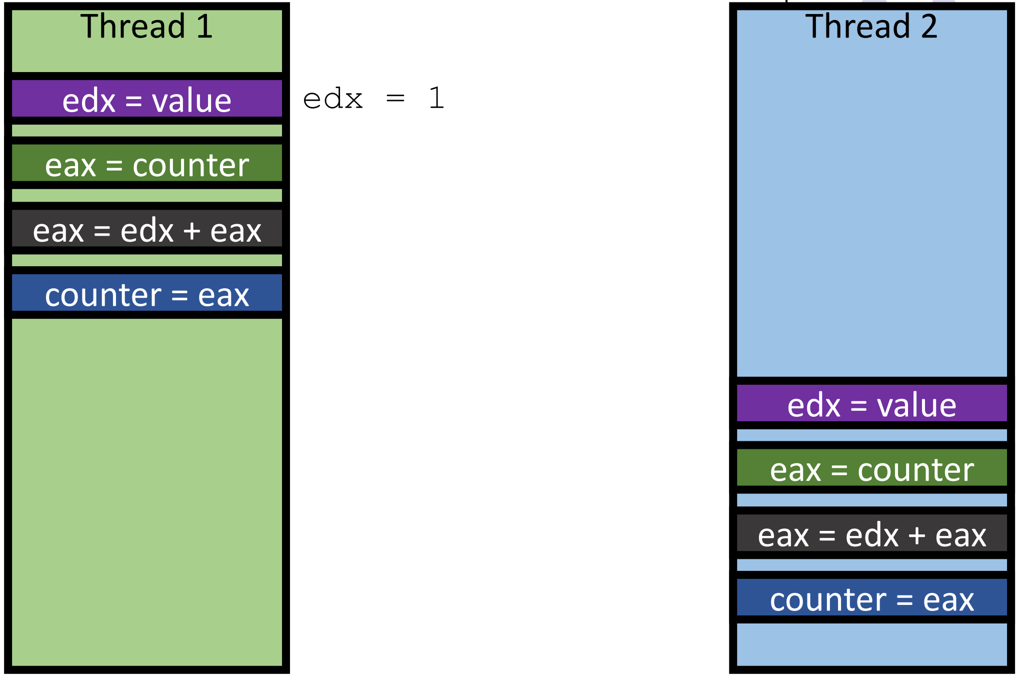
- counter += value
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`

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

Shared variable: No race

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

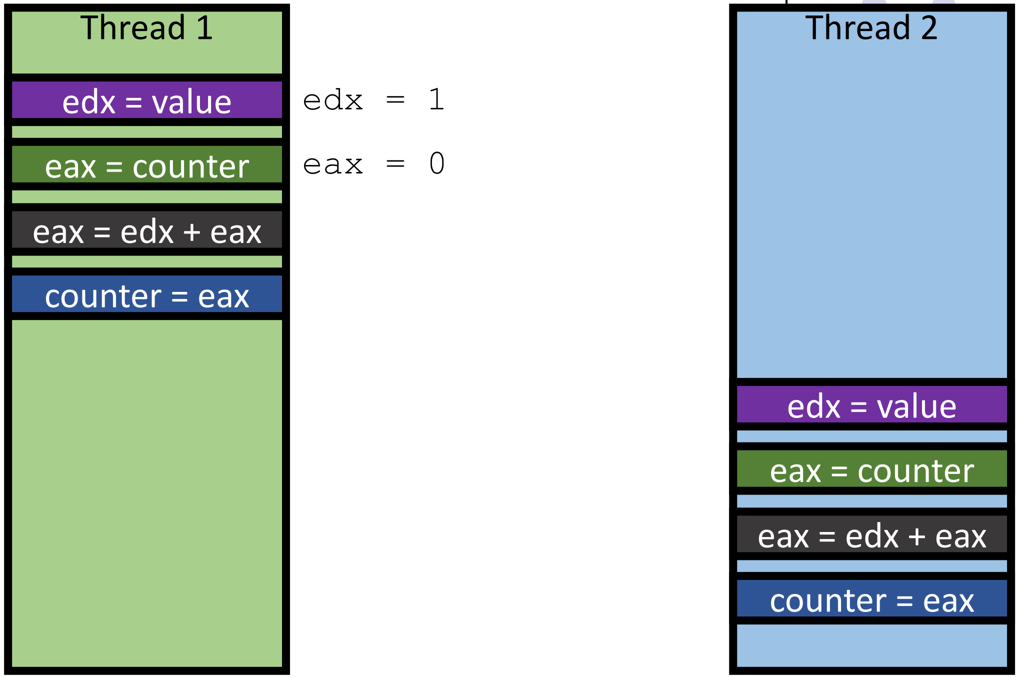
- Assume counter = 0 at start, and value = 1;



Shared variable: No race

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

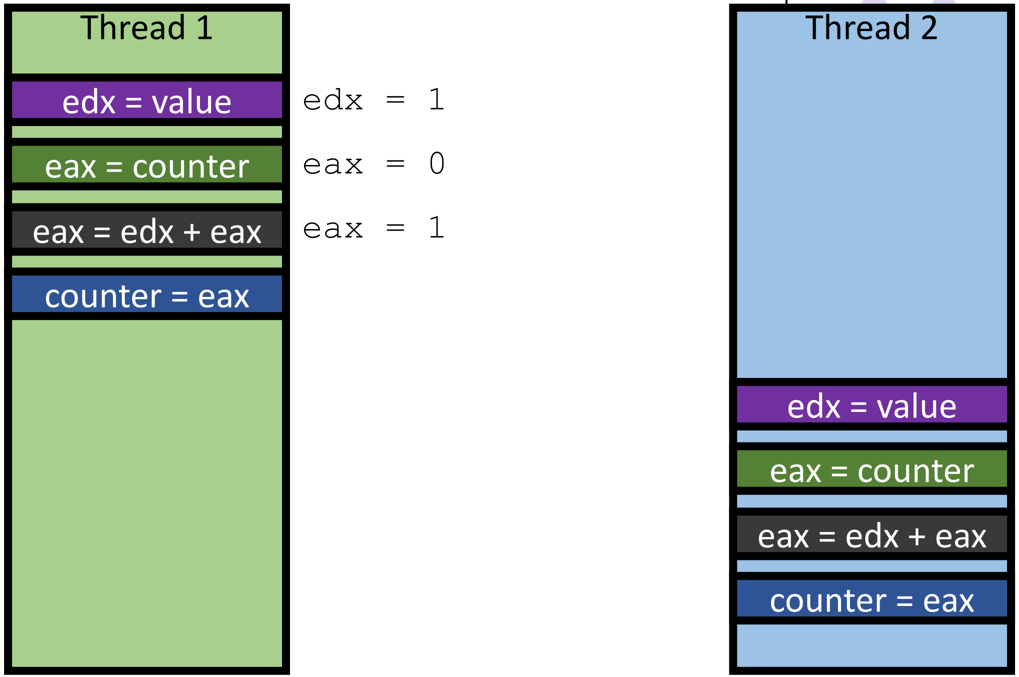
- Assume counter = 0 at start, and value = 1;



Shared variable: No race

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

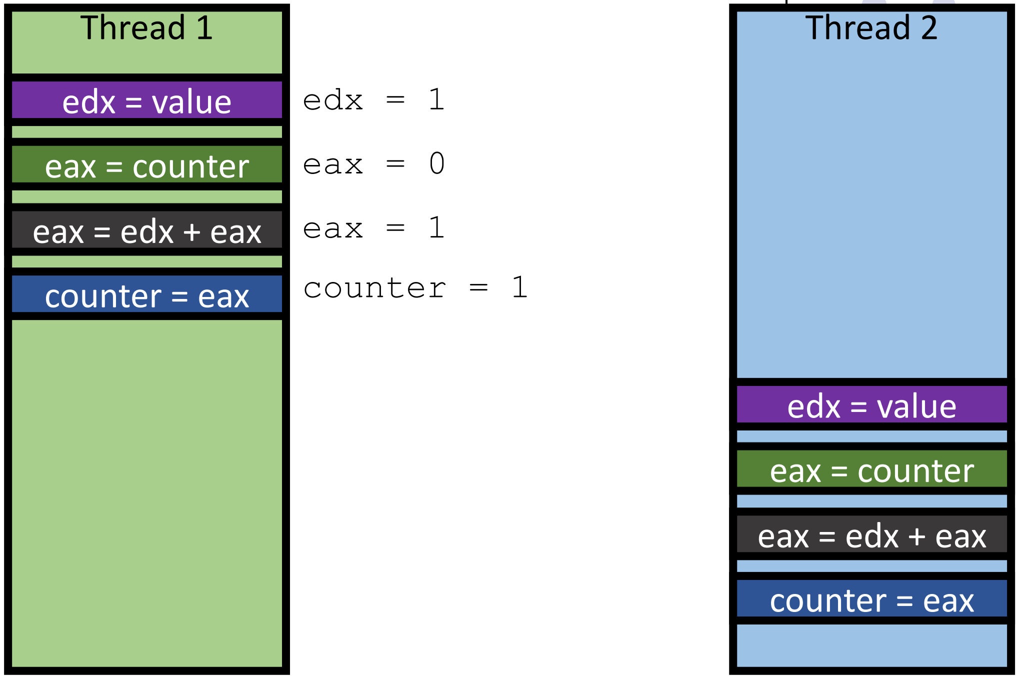
- Assume counter = 0 at start, and value = 1;



Shared variable: No race

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

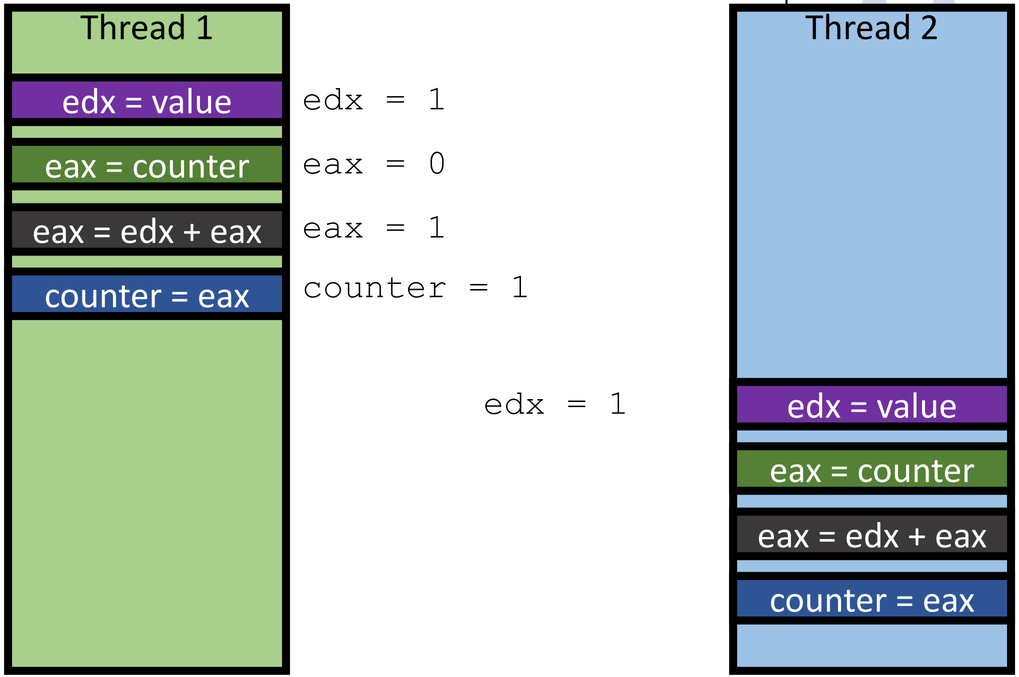
- Assume counter = 0 at start, and value = 1;



Shared variable: No race

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

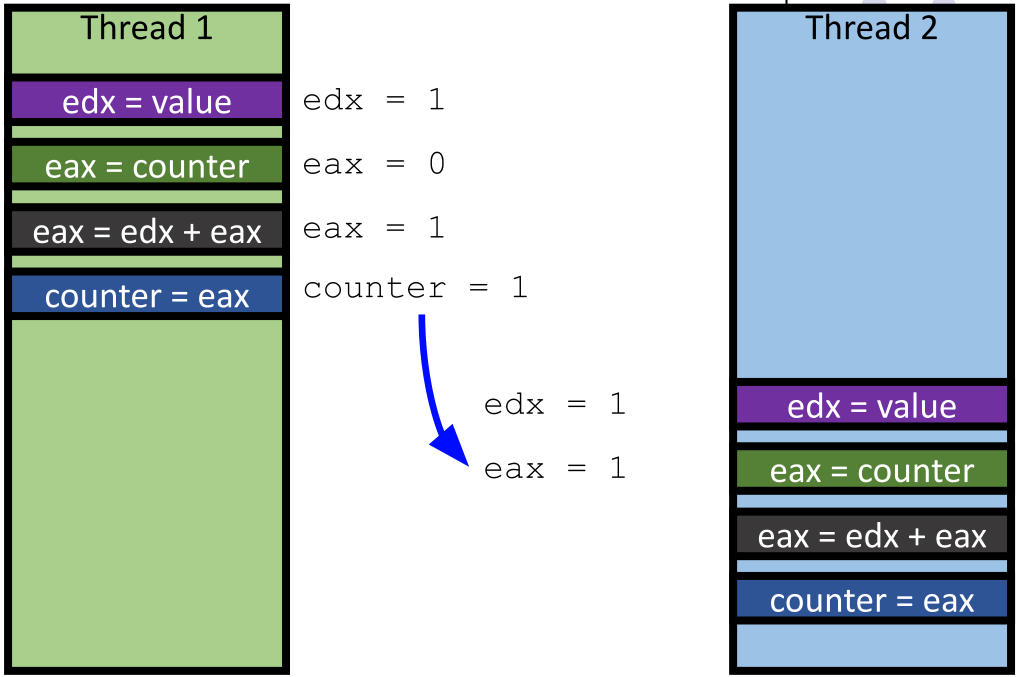
- Assume counter = 0 at start, and value = 1;



Shared variable: No race

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

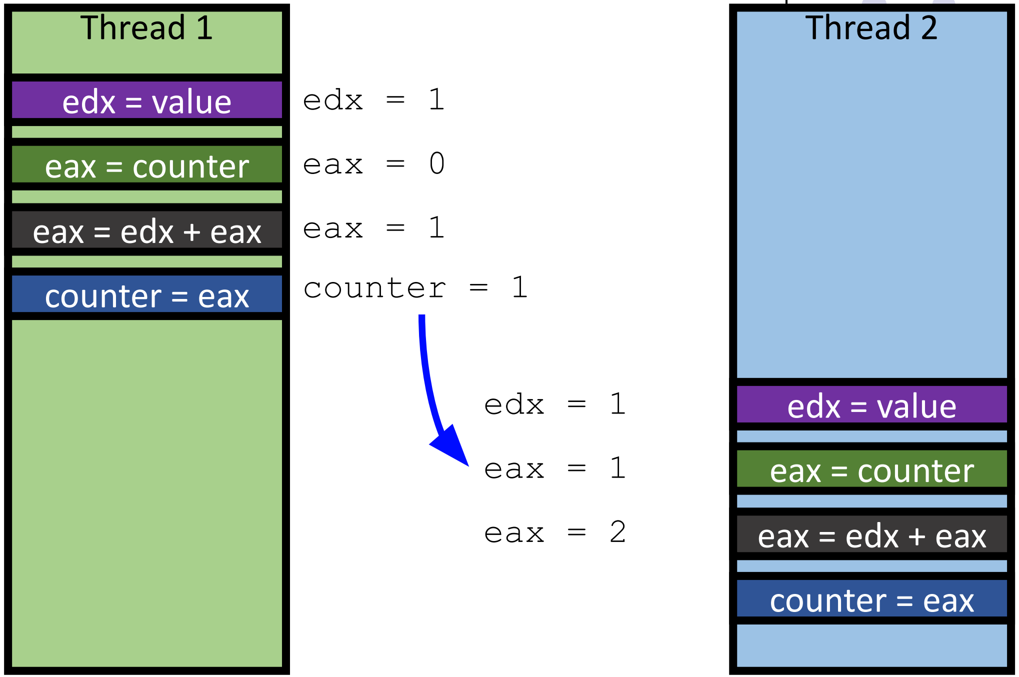
- Assume counter = 0 at start, and value = 1;



Shared variable: No race

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

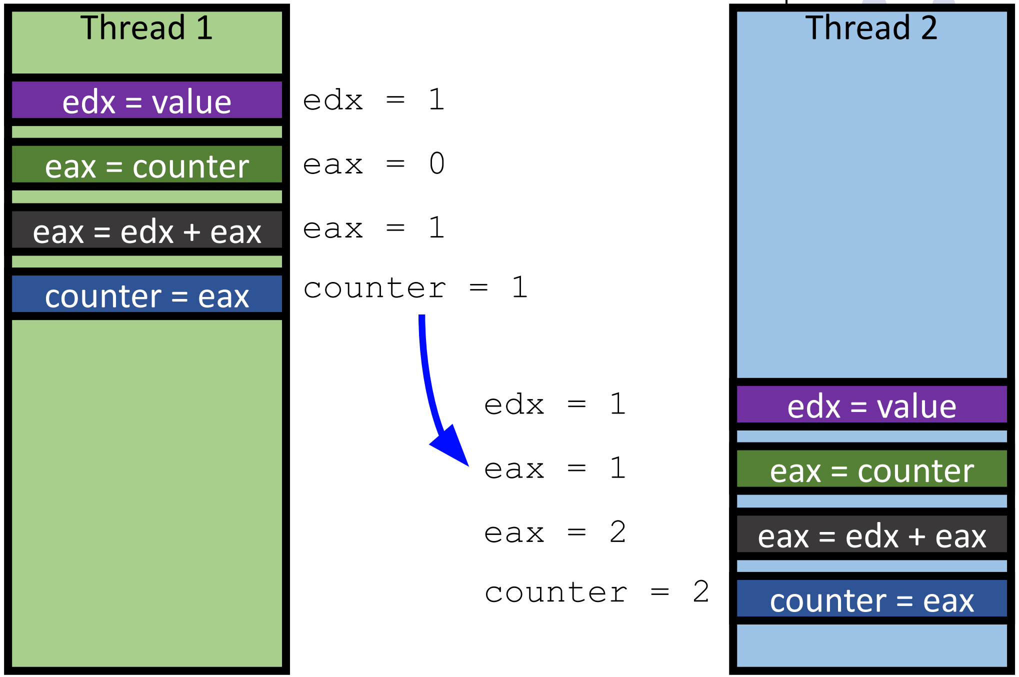
- Assume counter = 0 at start, and value = 1;



Shared variable: No race

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

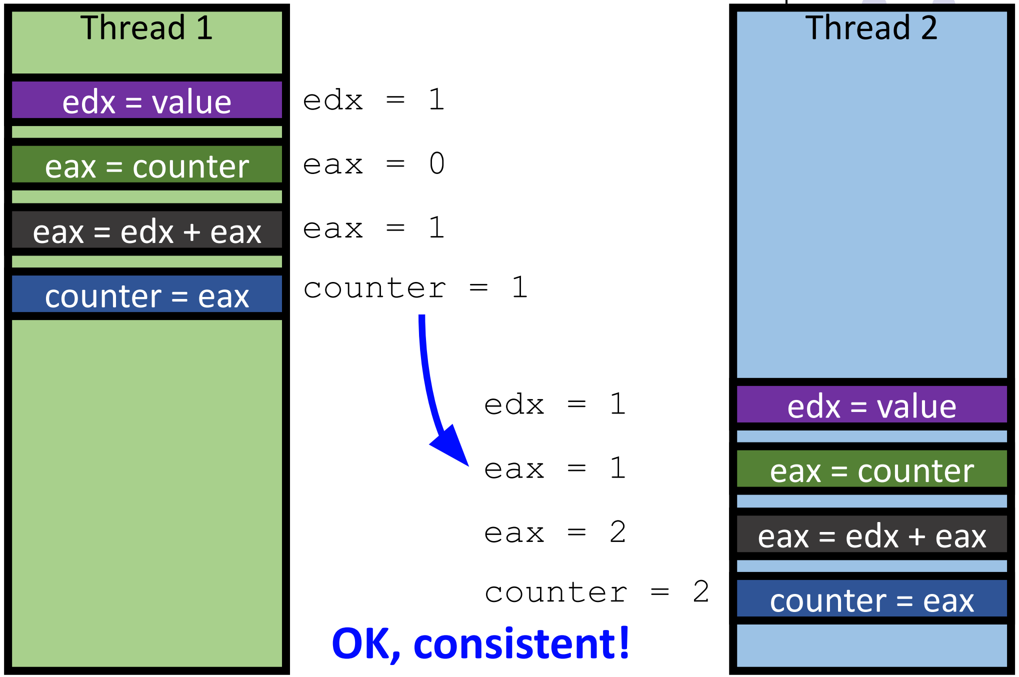
- Assume counter = 0 at start, and value = 1;



Shared variable: No race

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

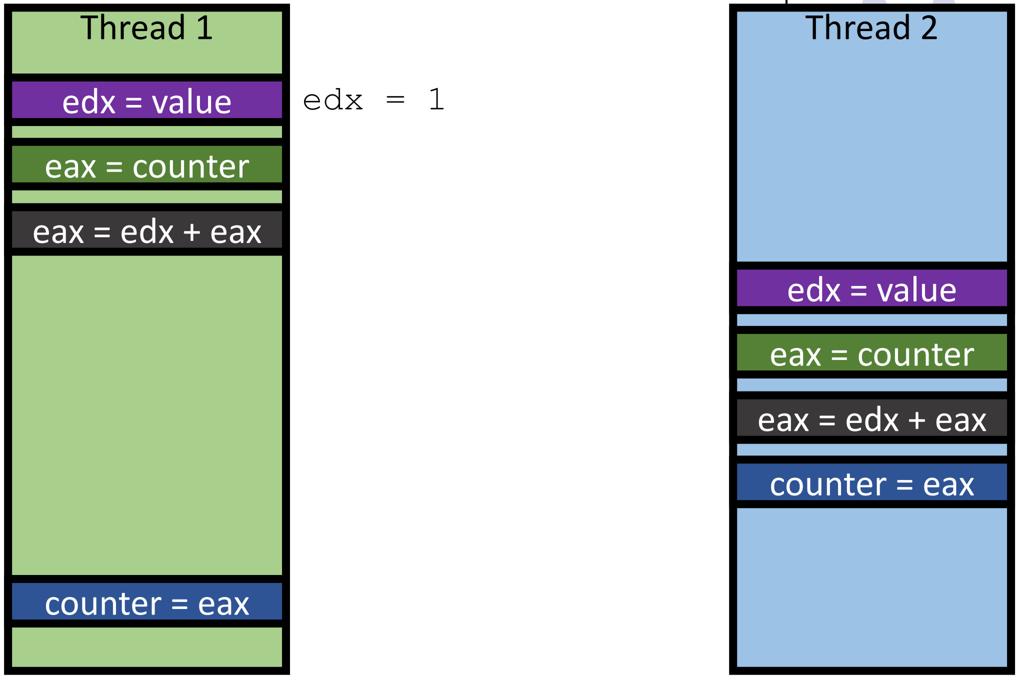
- Assume counter = 0 at start, and value = 1;



Shared variable: Data race

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

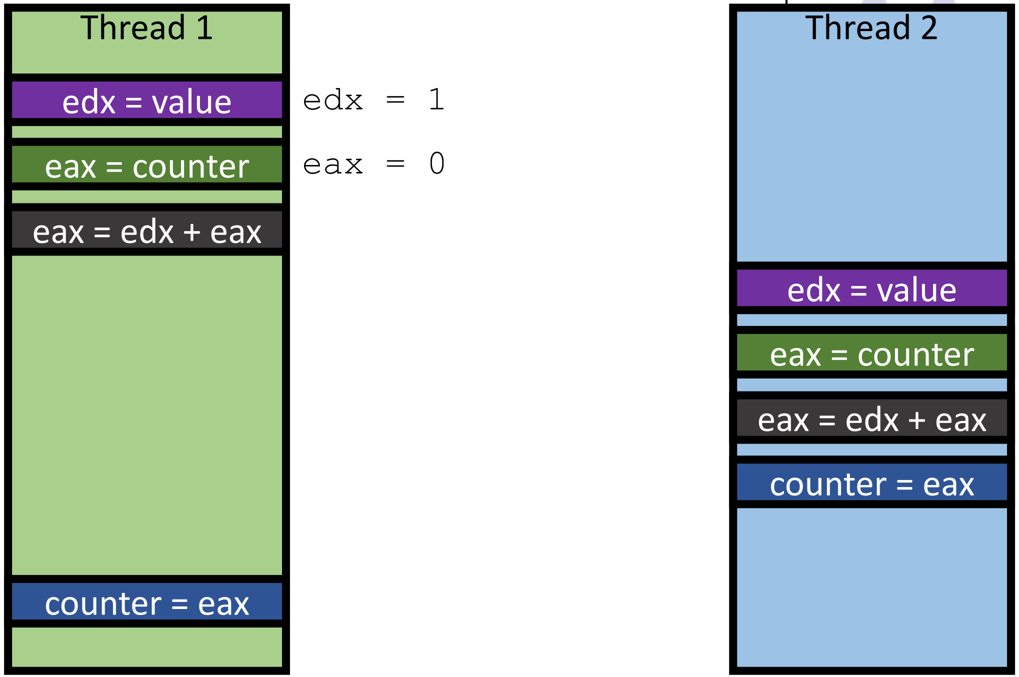
- Assume counter = 0 at start, and value = 1;



Shared variable: Data race

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

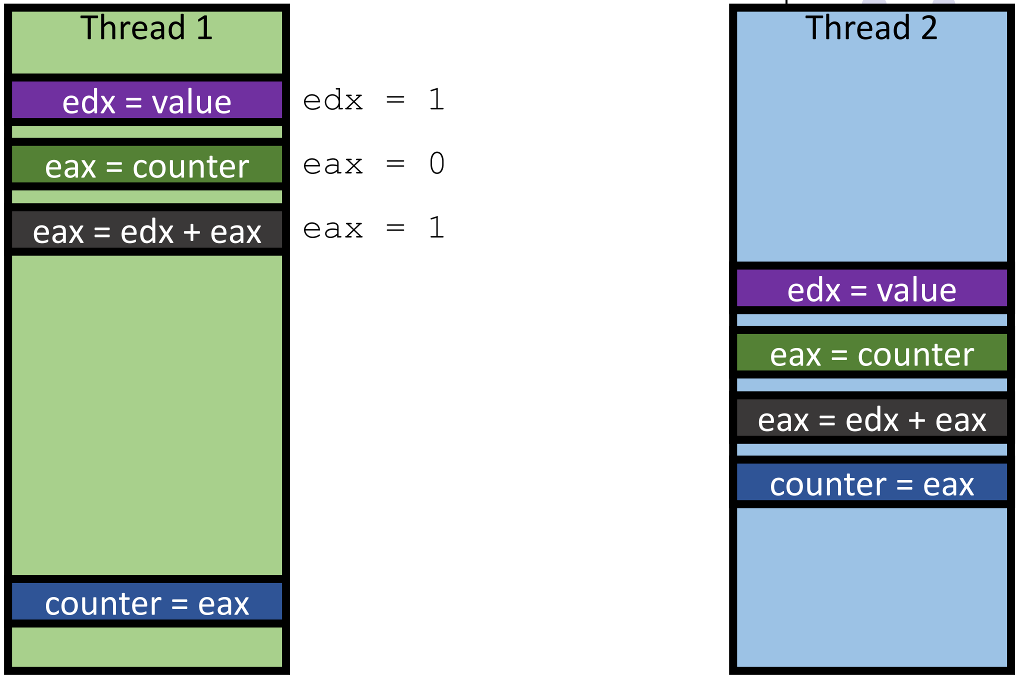
- Assume counter = 0 at start, and value = 1;



Shared variable: Data race

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

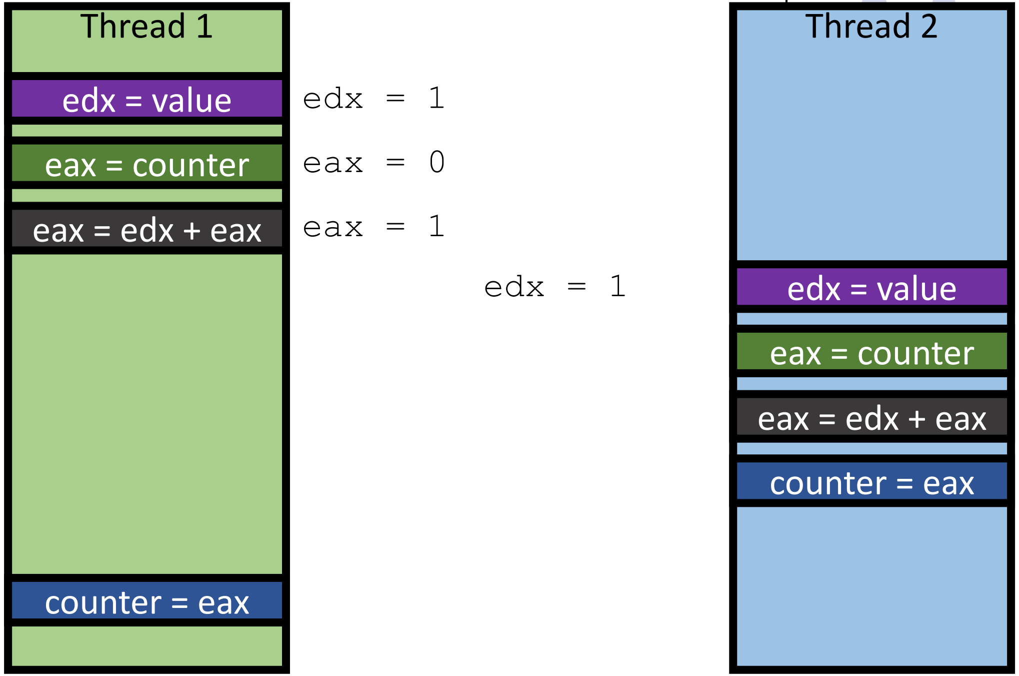
- Assume counter = 0 at start, and value = 1;



Shared variable: Data race

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

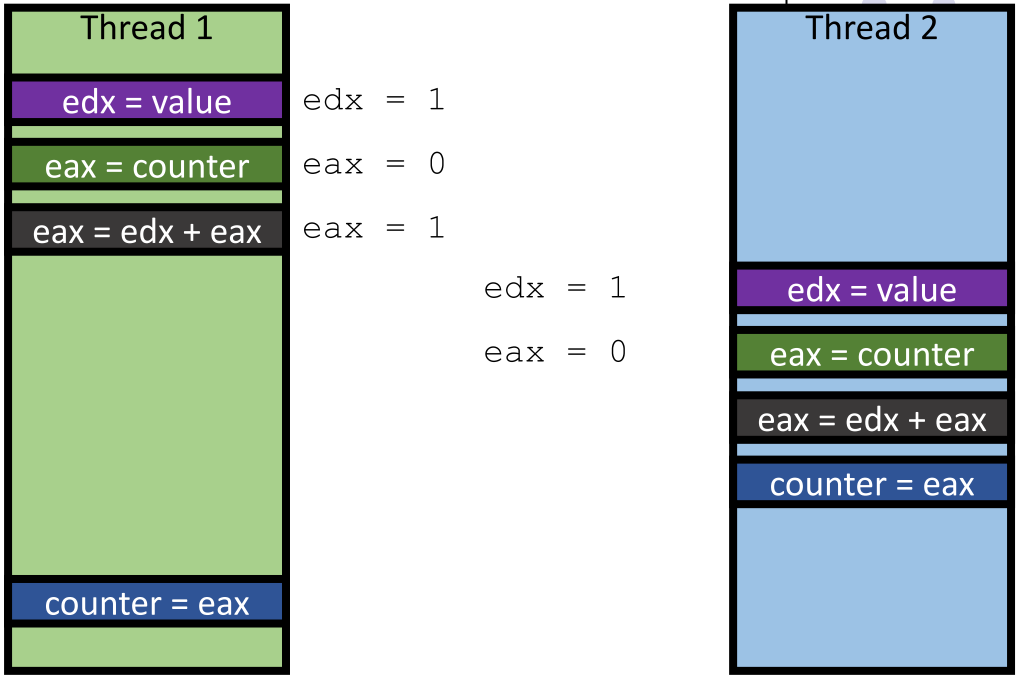
- Assume counter = 0 at start, and value = 1;



Shared variable: Data race

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

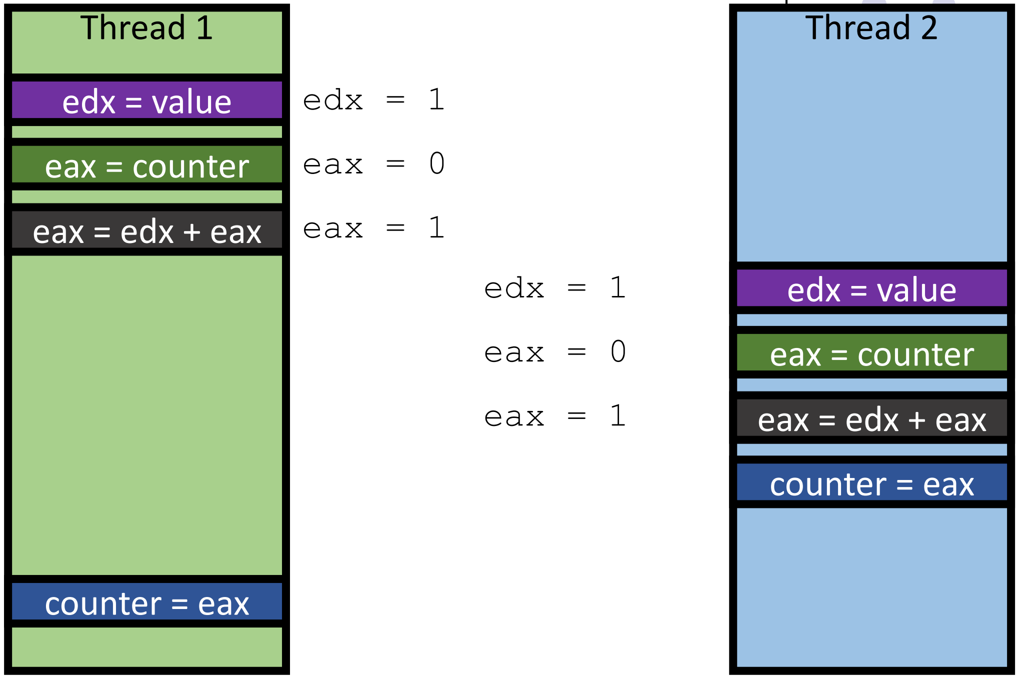
- Assume counter = 0 at start, and value = 1;



Shared variable: Data race

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

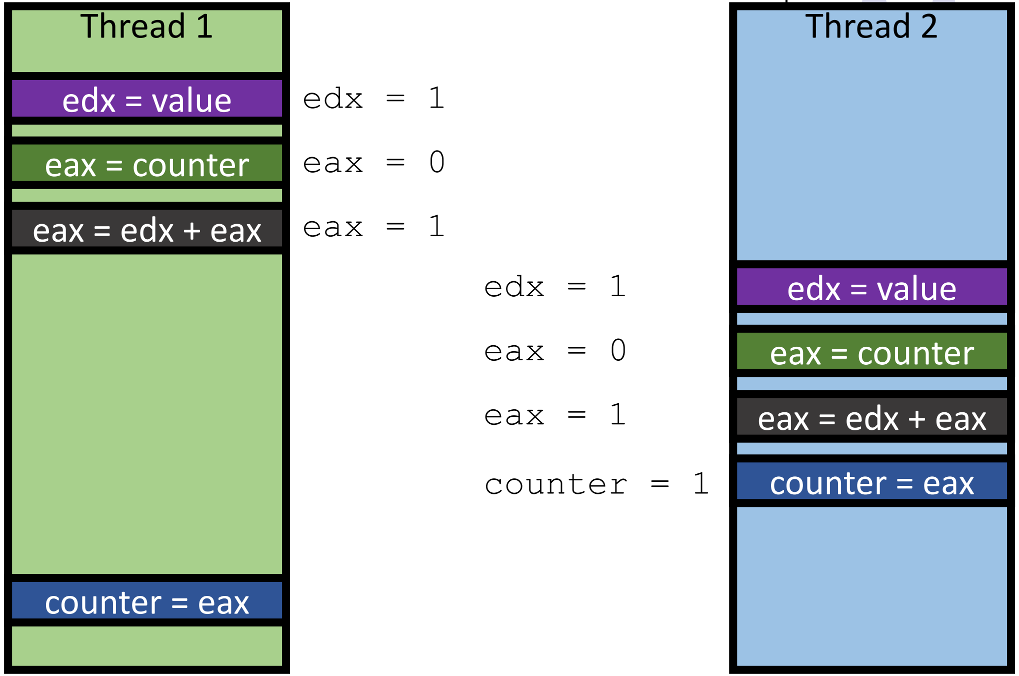
- Assume counter = 0 at start, and value = 1;



Shared variable: Data race

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

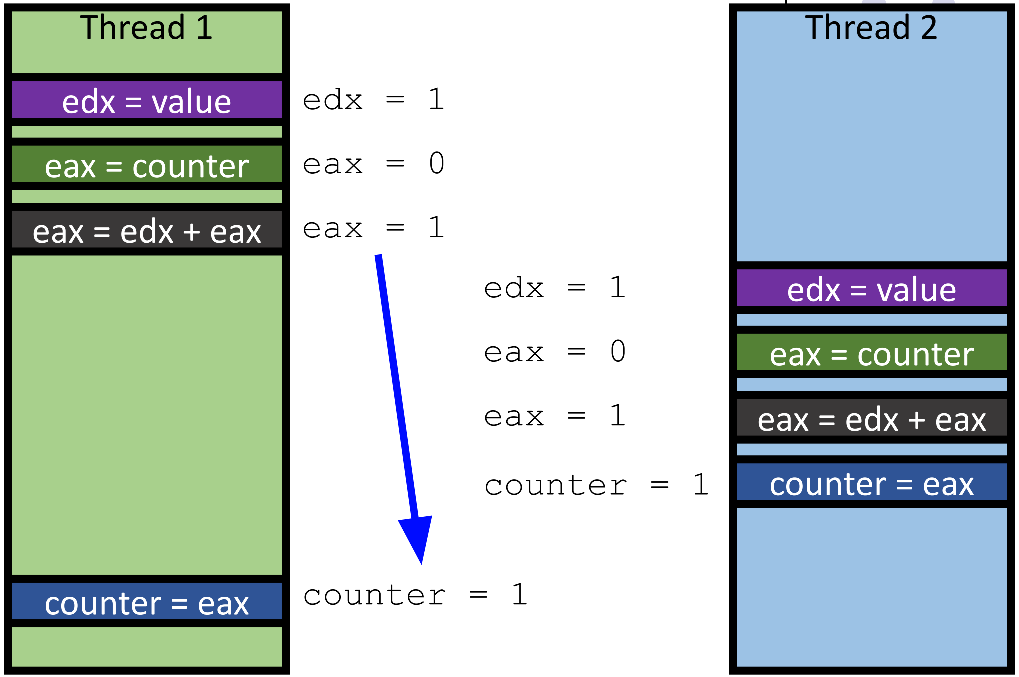
- Assume counter = 0 at start, and value = 1;



Shared variable: Data race

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

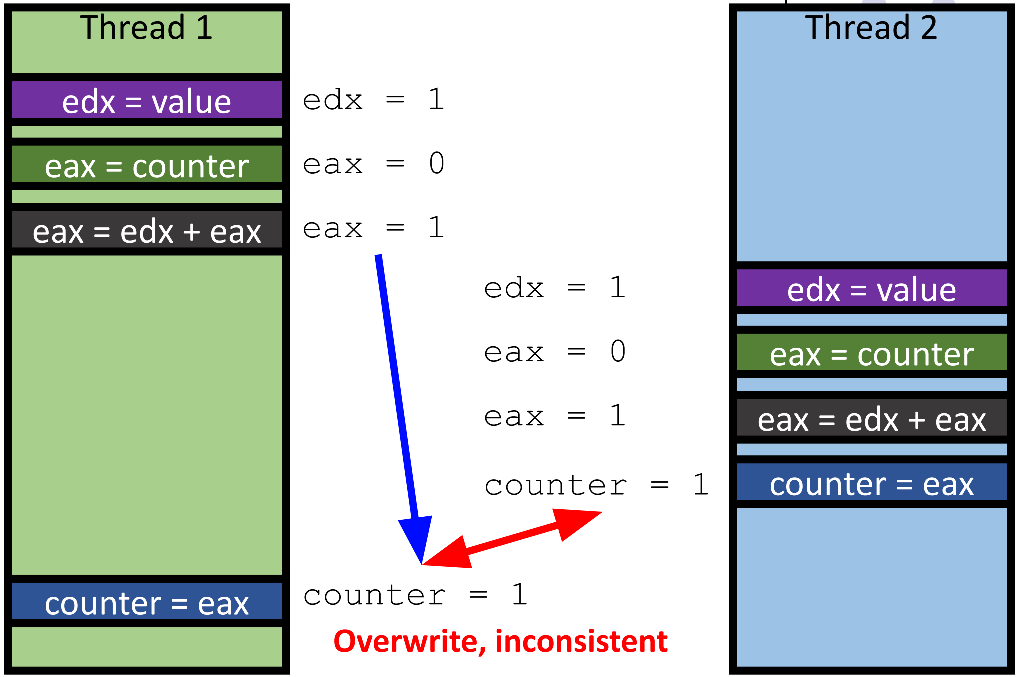
- Assume counter = 0 at start, and value = 1;



Shared variable: Data race

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

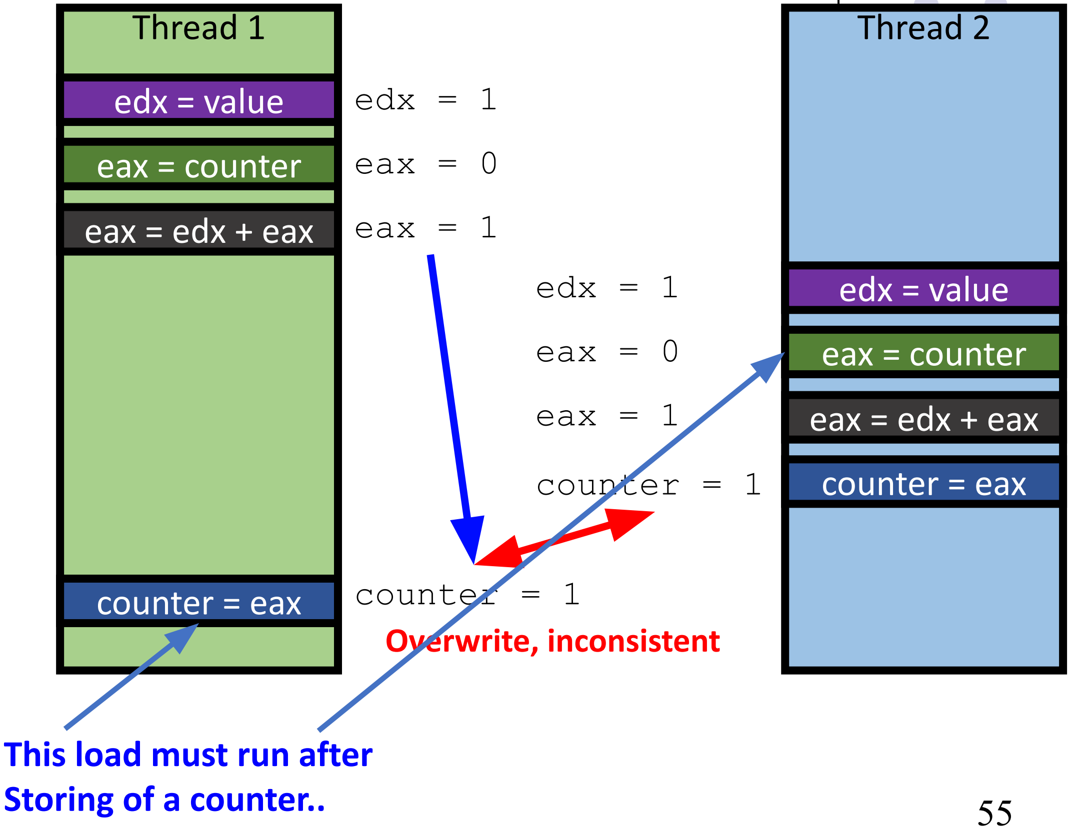
- Assume counter = 0 at start, and value = 1;



Shared variable: Data race

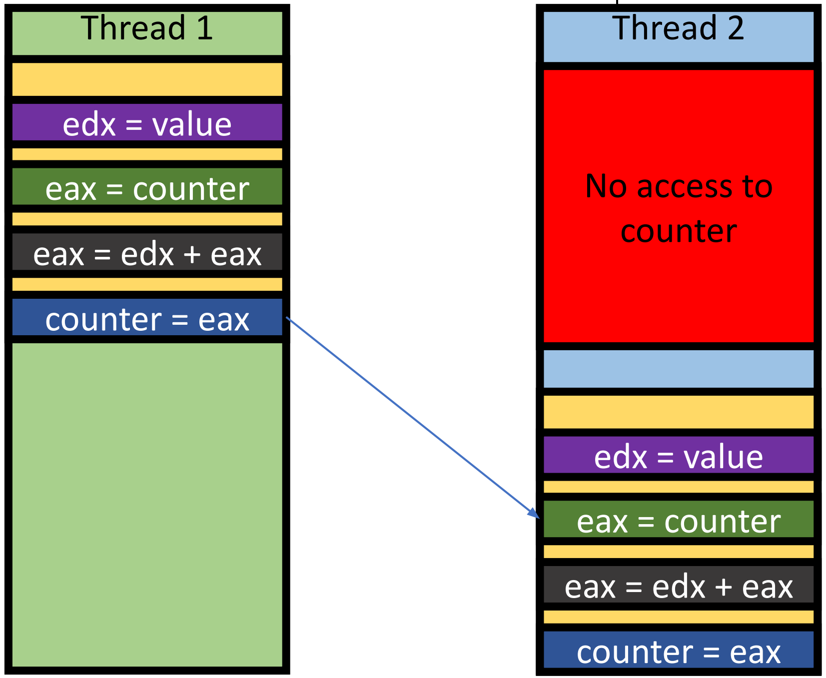
- counter += value
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`

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



How can we prevent data races?

- What we need?
  - Exclusive access to counter (shared variable)



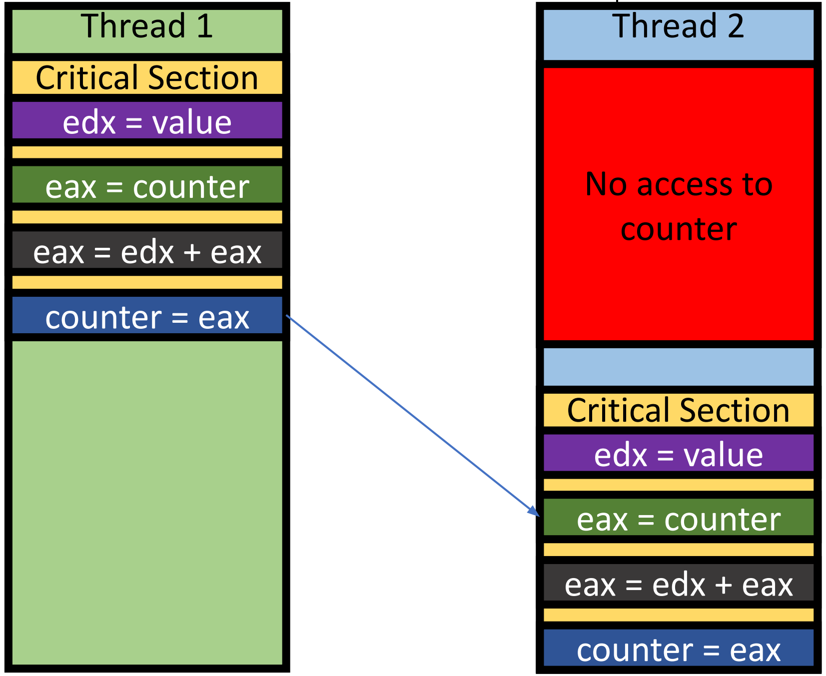
How can we prevent data races?

- *Critical section* – a section of code, or collection of operations, in which only one process shall be executing at a given time

- *Mutual exclusion (Mutex)* - mechanisms that ensure that only one person or process is doing certain things at one time (others are excluded)

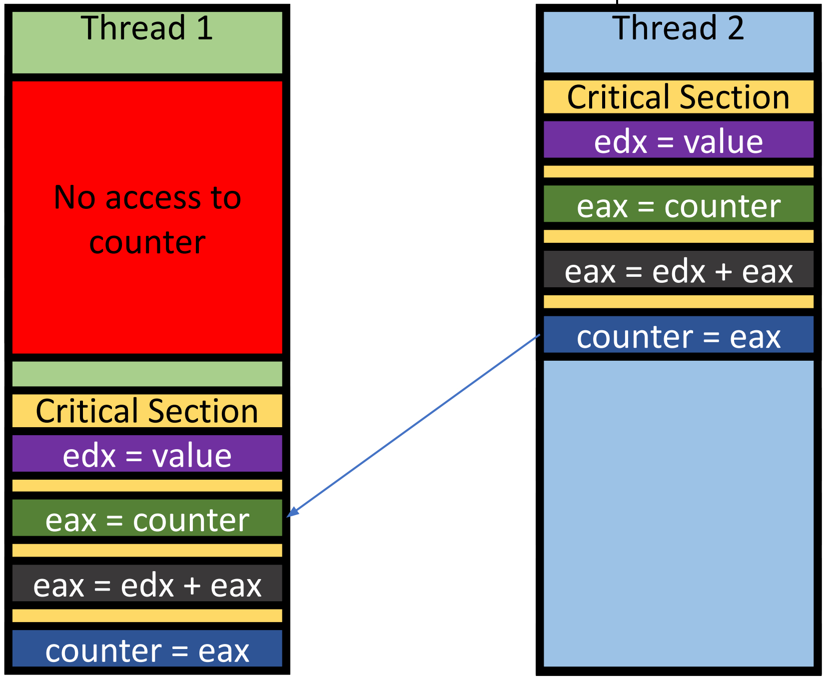
How can we prevent data races?

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

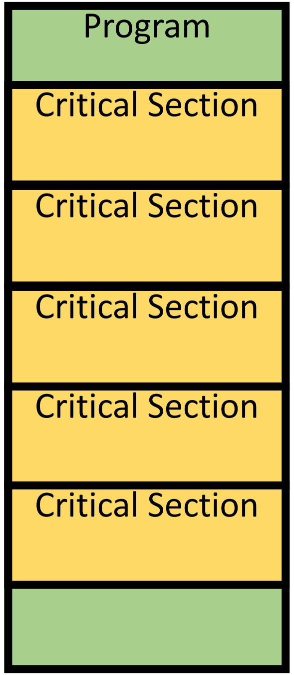


How can we prevent data races?

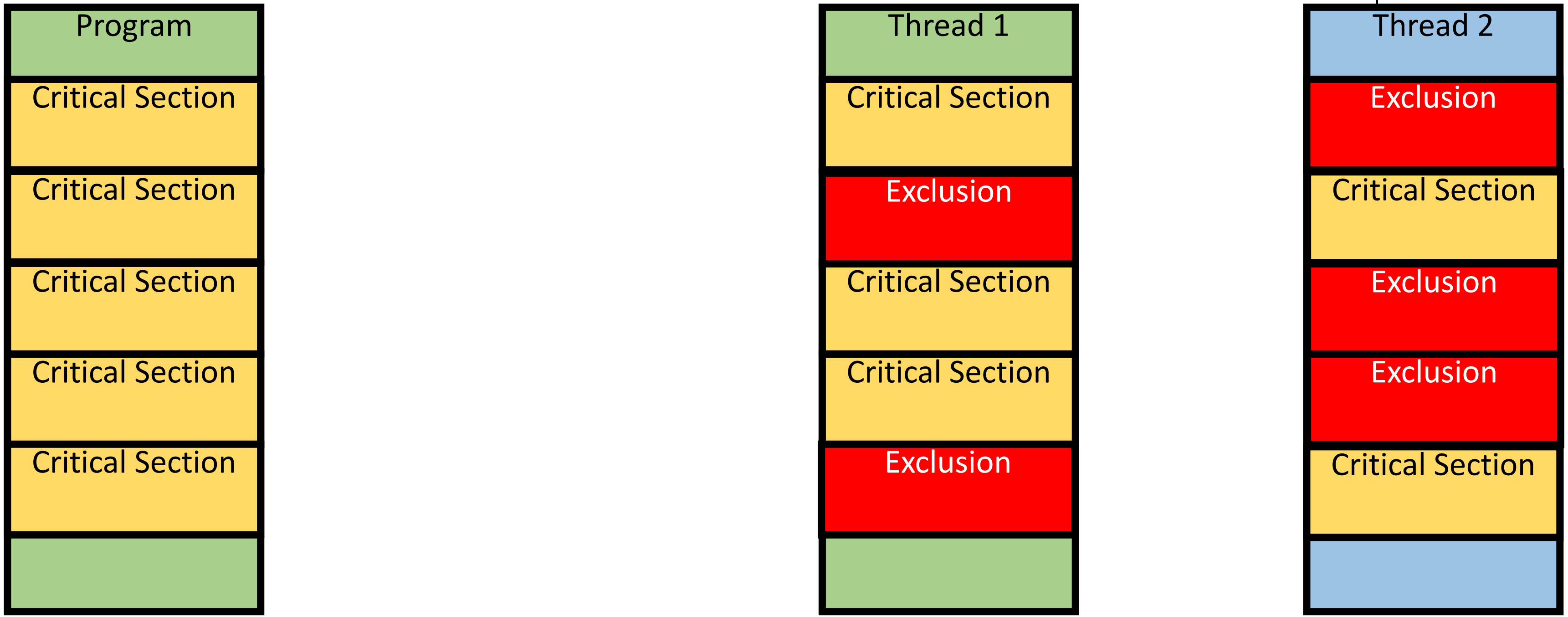
- Mutual Exclusion / Critical Section
  - Combine multiple instructions as a chunk
  - Let only one chunk execution runs
  - Block other executions



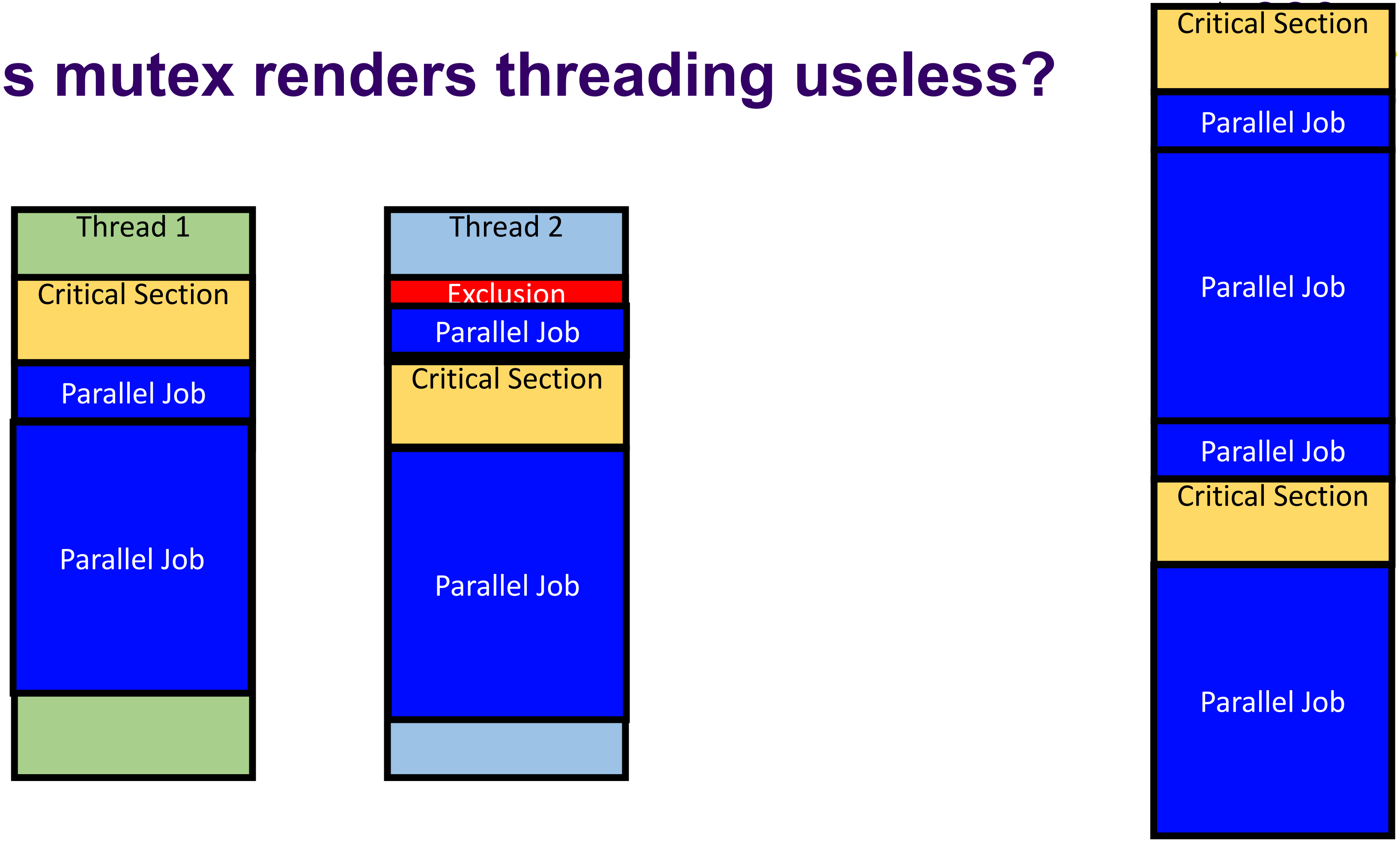
Does mutex renders threading useless?



Does mutex renders threading useless?



Does mutex renders threading useless?



Mutex Considerations

- Mutex can synchronize multiple threads and yield consistent result
  - No read before previous thread store the shared data
- Making the entire program as critical section is meaningless
  - 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!