OPERATING SYSTEMS

SYNCHRONIZATION

Producer: Thread A

i1: count = count + 1

Consumer: Thread B

i2: count = count - 1

CRITICAL SECTIONS

- Counter increment/decrement should happen without interruption/thread switching.
- We call this 'Atomic' execution.
- The section that require atomic execution are called "critical sections".
- It doesn't have to be one instruction, we could have multiple instructions that need to be executed atomically.

```
in = out = 0;
```

Thread A

```
while (true) {
item = produce_item;
while
(counter == BUFFER_SIZE) {}/* do nothing */;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
counter1++;
counter2++;
}
```

Critical Section

Thread B

```
while (true) {
while (counter == 0) {}/* do nothing
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter1--;
Counter2--
consume_item(item);
```

Critical Section

SYNCHRONIZATION

Producer: Thread A		Consumer: Thread B
i1: count = count + 1		i2: count = count - 1
a1: load count a2: add 1 a3: store count	Register/ALU "view"	b1: load count b2: subtract 1 b3: store count

Remember that "executing" an instruction involves multiple architecture-level steps, including loading registers, loading ALUs, executing ALUs, fetching results from ALU, etc.

SYNCHRONIZATION

Producer: Thread A

i1: count = count + 1

a1: load count a2: add 1 a3: store count

Register/ALU "view"

Consumer: Thread B

i2: count = count - 1

b1: load count b2: subtract 1 b3: store count

Remember that "executing" an instruction involves multiple architecture-level steps, including loading registers, loading ALUs, executing ALUs, fetching results from ALU, etc.

Assume initial value of count = 4

SYNCHRONIZATION

Producer: Thread A

i1: count = count + 1

a1: load count
a2: add 1
a3: store count

Register/ALU "view"

Consumer: Thread B

i2: count = count - 1

b1: load count
b2: subtract 1
b3: store count

Remember that "executing" an instruction involves multiple architecture-level steps, including loading registers, loading ALUs, executing ALUs, fetching results from ALU, etc.

Assume initial value of count = 4

Notice that all of the instructions in both threads are executed sequentially … a1<a2<a3, and b1<b2<b3

Worksheet Q1

What are the final values of count for these 4 histories?

a1 < b1 < a2 < a3 < b2 < b3

a1 < a2 < b1 < b2 < b3 < a3

a1 < a2 < a3 < b1 < b2 < b3

b1 < b2 < a1 < b3 < a2 < a3

SYNCHRONIZATION

Thread A

i1: count = count + 1
a1: load count a2: add 1 a3: store count

Register/ALU "view"

Thread B

i2: count = count - 1
b1: load count b2: subtract 1 b3: store count

Remember that "executing" an instruction involves multiple architecture-level steps, including loading registers, loading ALUs, executing ALUs, fetching results from ALU, etc.

Assume initial value of count = 4

Which is the desired final value of `count`?

a1 < b1 < a2 < a3 < b2 < b3 — count = 3

a1 < a2 < b1 < b2 < b3 < a3 — count = 5

a1 < a2 < a3 < b1 < b2 < b3 — count = 4

b1 < b2 < a1 < b3 < a2 < a3 — count = 5

SYNCHRONIZATION

Thread A		Thread B
i1: count = count + 1		i2: count = count - 1
a1: load count a2: add 1 a3: store count	Register/ALU "view"	b1: load count b2: subtract 1 b3: store count

Remember that "executing" an instruction involves multiple architecture-level steps, including loading registers, loading ALUs, executing ALUs, fetching results from ALU, etc.

Assume initial value of count = 4

What is the one property of the desired history that is different from the non-desirable histories?

a1 < b1 < a2 < a3 < b2 < b3 — count = 3

a1 < a2 < b1 < b2 < b3 < a3 — count = 5

a1 < a2 < a3 < b1 < b2 < b3 — count = 4

b1 < b2 < a1 < b3 < a2 < a3 — count = 5

SYNCHRONIZATION

Thread A		Thread B
i1: count = count + 1		i2: count = count - 1
a1: load count a2: add 1 a3: store count	Register/ALU "view"	b1: load count b2: subtract 1 b3: store count

Remember that "executing" an instruction involves multiple architecture-level steps, including loading registers, loading ALUs, executing ALUs, fetching results from ALU, etc.

Assume initial value of count = 4

What is the one property of the desired history that is different from the non-desirable histories?

History	Result
a1 < b1 < a2 < a3 < b2 < b3	count = 3
a1 < a2 < b1 < b2 < b3 < a3	count = 5
a1 < a2 < a3 < b1 < b2 < b3	count = 4
b1 < b2 < a1 < b3 < a2 < a3	count = 5

SYNCHRONIZATION PRIMITIVES

- System may provide mechanisms for atomic execution …
- OS should provide synchronization primitives that use atomic execution and offer the user more “abstract” solution:
  - Mutex Locks
  - Semaphores
  - Monitors

MUTEX

Most operating systems allow, via a system call, the use of a **<u>mutex</u>** … this allows the application programmer to solve a critical section problem

Mutex
+acquire()
+release()

This high level idea is the following : **have the OS provide system calls for a lock that can be used to control access to a critical section.**

```
do {
   // acquire lock
     Critical section
   // release lock
   // other stuff (remainder)
} while(true);
```

```
do {
   // acquire lock
     Critical section
   // release lock
   // other stuff (remainder)
} while(true);
```

MUTEX LOCKS

- Calls to either acquire() or release() must be performed atomically.
- What does that mean?
  - Thread should not be interrupted while performing the locking/releasing sequence:
  - disable interrupt OR
  - use a single instruction for locking/releasing:
    - test and set()
    - compare and swap()

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;;
}
```

```
release() {
   available = true;
}
```

CRITICAL SECTIONS

- Worksheet Q2: Add mutex code to ensure atomicity of critical section.

Mutex
+acquire()
+release()

```
in = out = 0;
```

Thread A

```
while (true) {
item = produce_item;
while
(counter == BUFFER_SIZE) {}/* do nothing */;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
```

Critical Section

Thread B

```
while (true) {
while (counter == 0) {}/* do nothing
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
consume_item(item);
```

Critical Section

WESTERN
WASHINGTON UNIVERSITY

MUTEX SOLUTION FOR PRODUCER/CONSUMER

```
in = out = 0;
Mutex mutex = Mutex.Init();
```

Thread A

```
while (true) {
item = produce_item;
while
(counter == BUFFER_SIZE) {}/* do nothing */;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
```

Thread B

```
while (true) {
while (counter == 0) {}/* do nothing
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
consume_item(item);
```

MUTEX SOLUTION FOR PRODUCER/CONSUMER

Initialize mutex object by main thread.

Use the mutex by both threads

```
in = out = 0;
Mutex mutex=Mutex.Init();
```

Thread A

```
while (true) {
item = produce_item;
while
(counter == BUFFER_SIZE) {}/* do nothing */;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
mutex.acquire()
counter++;
mutex.release()
}
```

Thread B

```
while (true) {
while (counter == 0) {}/* do nothing
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
mutex.acquire()
counter--;
mutex.release()
consume_item(item);
```

SEMAPHORES: MORE CONTROL

- Mutex lock allow only one thread to use critical section.

T1 *T2* *T3*

Semaphore
2

critical 1 critical 2 critical 3

Semaphore
int value
+ Semaphore(int) + increment signal + decrement wait

SEMAPHORES: MORE CONTROL

- Mutex lock allow only one thread to use critical section.
- Solution: semaphores
  - Can only access critical section if S>0.
  - Can control number of threads running critical section by initializing S (and capping it) to an integer value.

```
wait(S) {
    while (S < 0) {
    //busy wait
    }
    S--;
}
```

Use wait() instead of acquire().

Critical Section

```
singal(S) {
    S++;
}
```

Use signal() instead of release().

SEMAPHORES

- Uses a counter S that can only be accesses through atomic operations wait() and signal ().
- Must use wait () before accessing critical section
- Can only access critical section if S>0.
- Can control number of threads running critical section
- Can control order of threads.

```
wait(S) {
    while (S < 0) {
    //busy wait
    }
    S--;
}
```

Critical Section

```
singal(S) {
    S++;
}
```

Semaphore
int value
Thread* list_of_waiting
+wait()
+signal()

SEMAPHORES: AVOIDING BUSY WAITING

- Busy waiting wastes CPU cycles.
- When it expected time to wait is short, it is rather beneficial compared to expensive context switch.
- Otherwise, CPU time is lost.

```
wait(S) {
    while (S < 0) {
    //busy wait
    }
    S--;
}
```

Critical Section

```
singal(S) {
    S++;
}
```

SEMAPHORES: AVOIDING BUSY WAITING

- Busy waiting wastes CPU cycles.
- When it expected time to wait is short, it is rather beneficial compared to expensive context switch.
- Otherwise, CPU time is lost.
- Solution: block() and wakeup()

```
wait(S) {
    while (S < 0) {
    //busy wait
    }
    S--;
}
```

Critical Section

```
singal(S) {
    S++;
}
```

THREAD BLOCKING

- To avoid wasting CPU time while waiting on a semaphore (or a lock), threads can block themselves using a block() call.
- In linux, block() is implemented using 'sleep()' system call.
- Threads can place themselves in a waiting queue for the specific semaphore.
- And *afterwards* block themselves.

sleep(3) - Linux man page

Name

sleep - sleep for the specified number of seconds

Synopsis

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

Description

sleep() makes the calling thread sleep until *seconds* seconds have elapsed or a signal arrives which is not ignored.

SEMAPHORE WITH WAITING QUEUE

SEMAPHORE WITH WAITING QUEUE

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Add list of waiting threads/processes.

SEMAPHORE WITH WAITING QUEUE

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Add list of waiting threads/processes.

SEMAPHORE WITH WAITING QUEUE

```
typedef struct {
     int value;
     struct process *list;
} semaphore;

wait(semaphore *S) {
          S->value--;
          if (S->value < 0) {
               add this process to S->list;
               block();
          }
}
```

SEMAPHORE WITH WAITING QUEUE

```
typedef struct {
    int value;
    struct process *list;
} semaphore;

wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
}
```

SEMAPHORE WITH WAITING QUEUE

```
typedef struct {
     int value;
     struct process *list;
} semaphore;

wait(semaphore *S) {
          S->value--;
          if (S->value < 0) {
                add this process to S->list;
                block();
          }
}

signal(semaphore *S) {
          S->value++;
          if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
          }
}
```

SEMAPHORE WITH WAITING QUEUE

```
typedef struct {
    int value;
    struct process *list;
} semaphore;

wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
}

signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

SEMAPHORE WITH WAITING QUEUE

```
typedef struct {
     int value;
     struct process *list;
} semaphore;

wait(semaphore *S) {
          S->value--;
          if (S->value < 0) {
                add this process to S->list;
                block();
          }
}

signal(semaphore *S) {
          S->value++;
          if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
          }
}
```

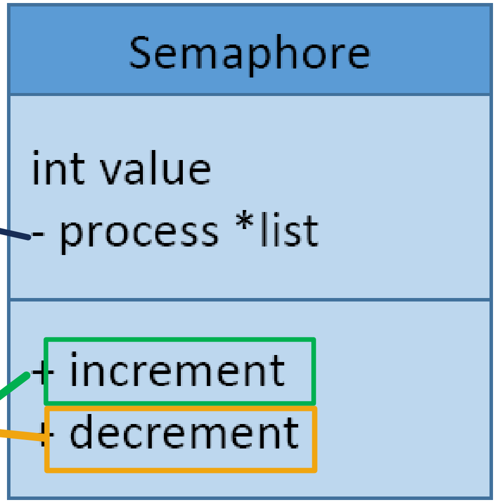
Semaphore
int value - process *list
+ increment + decrement

SEMAPHORE WITH WAITING QUEUE

```
typedef struct {
     int value;
     struct process *list;
} semaphore;

wait(semaphore *S) {
          S->value--;
          if (S->value < 0) {
               add this process to S->list;
               block();
          }
}

signal(semaphore *S) {
          S->value++;
          if (S->value <= 0) {
               remove a process P from S->list;
               wakeup(P);
          }
}
```



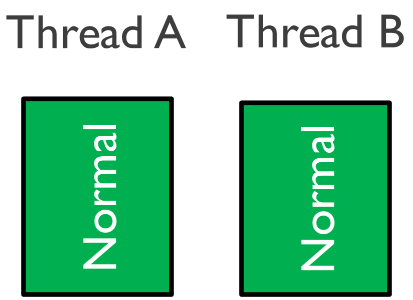
CRITICAL SECTION

No Execution

Normal Execution

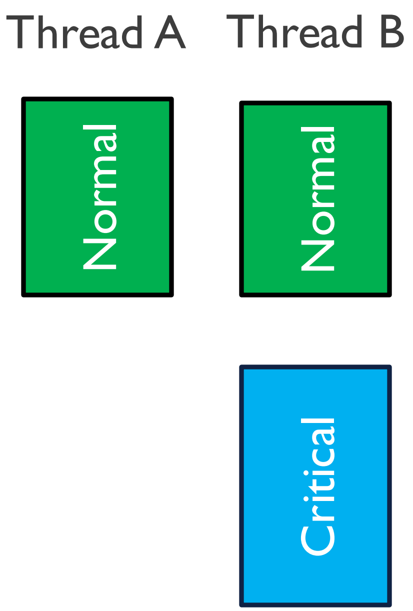
Atomic Execution

Critical Section Execution



CRITICAL SECTION

- No Execution
- Normal Execution
- Atomic Execution
- Critical Section Execution



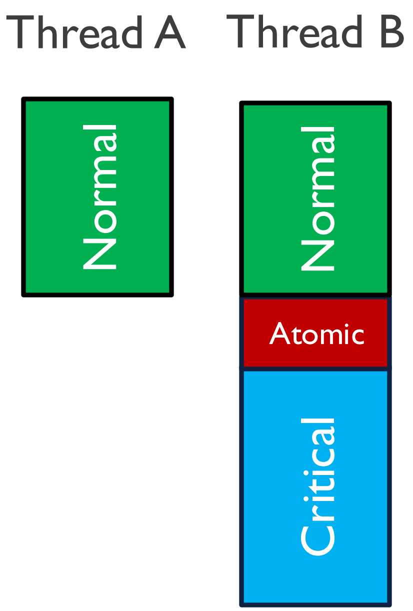
CRITICAL SECTION

No Execution

Normal Execution

Atomic Execution

Critical Section Execution



acquire(), decrement(), wait() ..

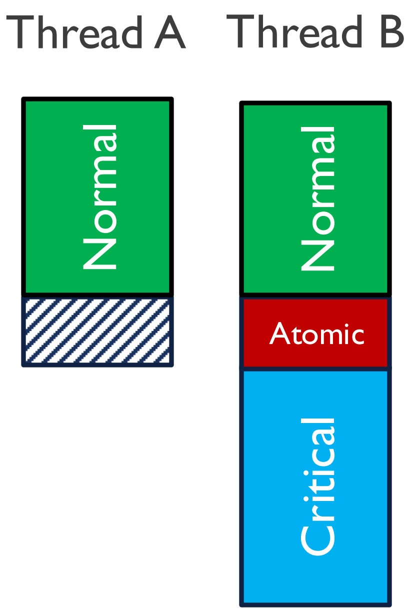
CRITICAL SECTION

No Execution

Normal Execution

Atomic Execution

Critical Section Execution



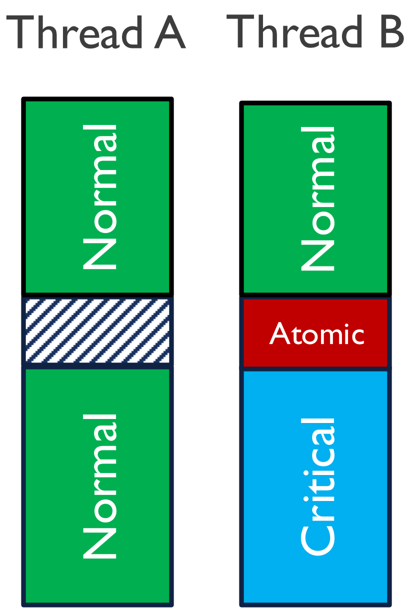
CRITICAL SECTION

No Execution

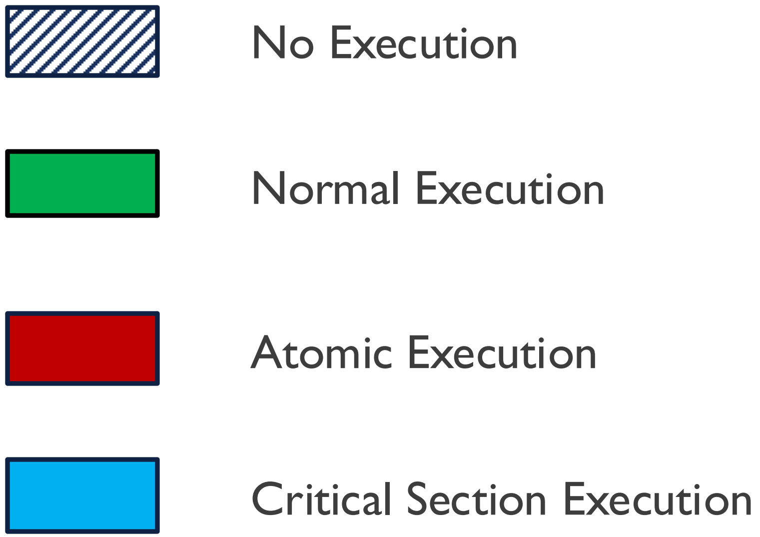
Normal Execution

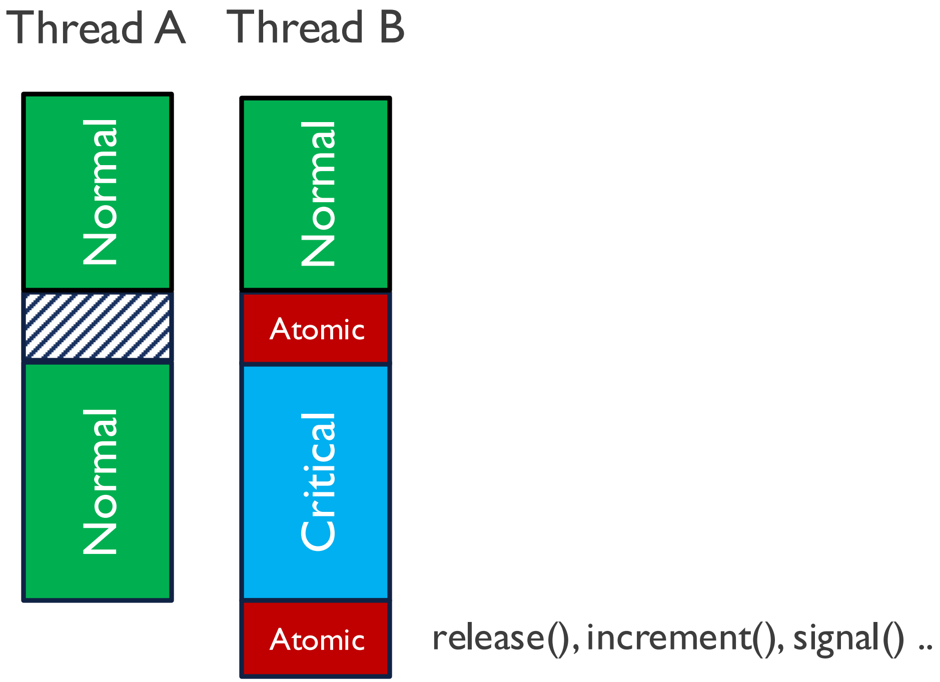
Atomic Execution

Critical Section Execution



CRITICAL SECTION





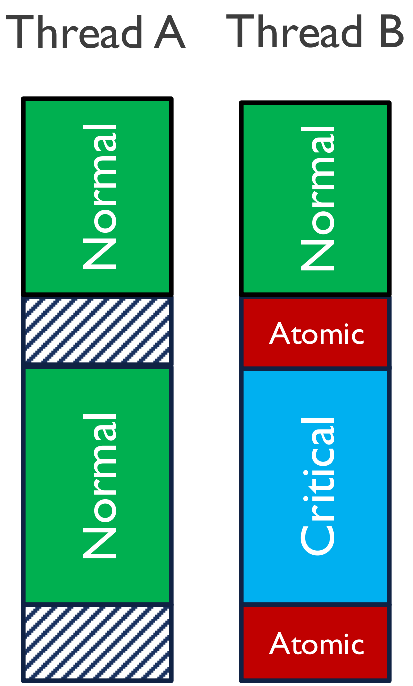
CRITICAL SECTION

No Execution

Normal Execution

Atomic Execution

Critical Section Execution



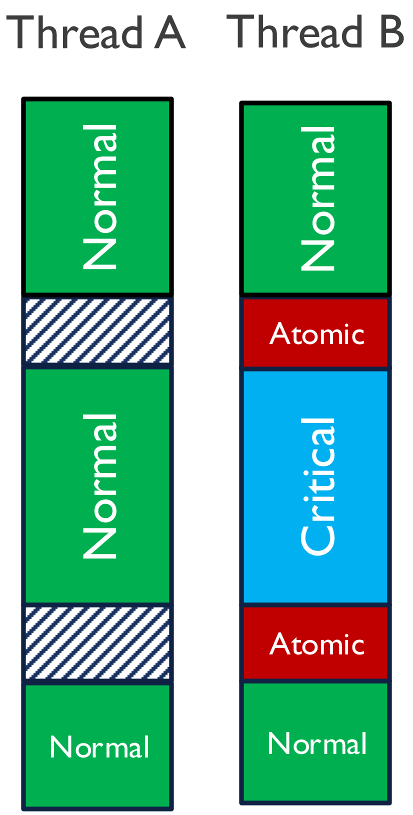
CRITICAL SECTION

No Execution

Normal Execution

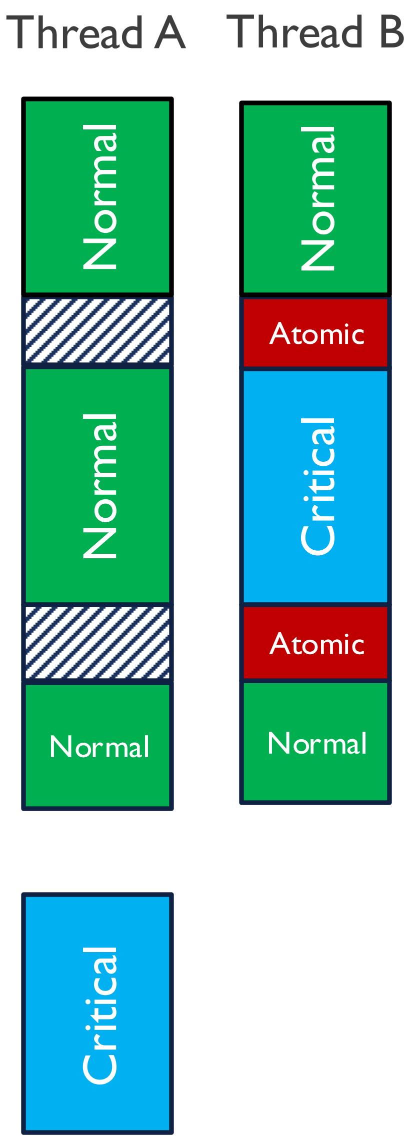
Atomic Execution

Critical Section Execution

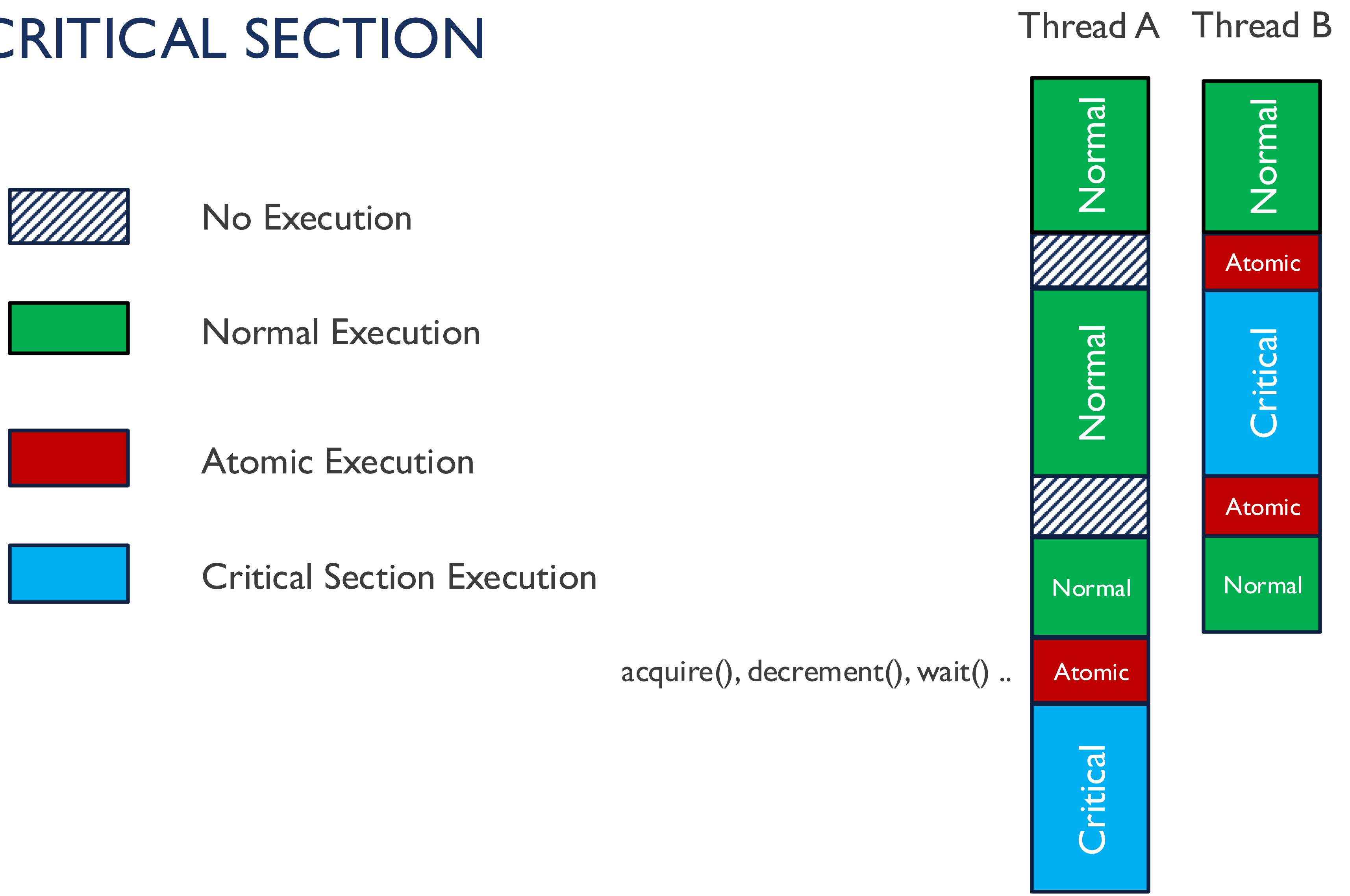


CRITICAL SECTION

(hatched)	No Execution
(green)	Normal Execution
(red)	Atomic Execution
(blue)	Critical Section Execution



CRITICAL SECTION



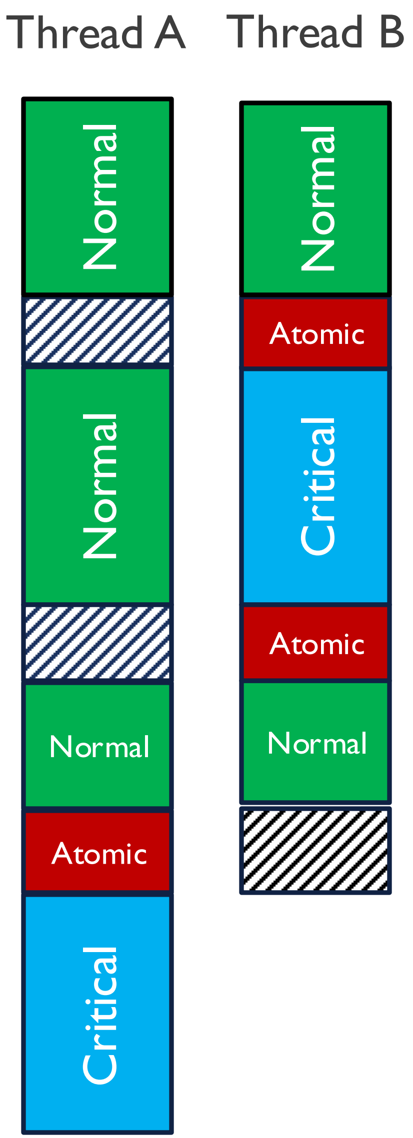
CRITICAL SECTION

No Execution

Normal Execution

Atomic Execution

Critical Section Execution



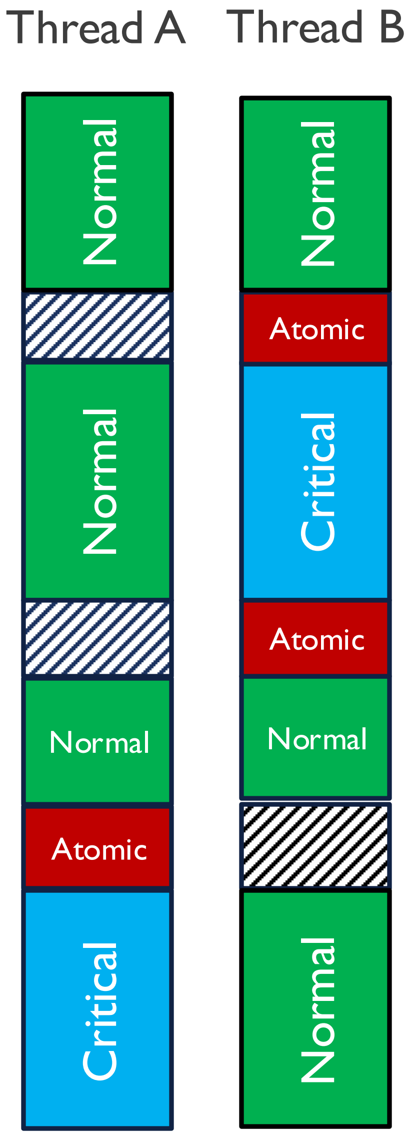
CRITICAL SECTION

No Execution

Normal Execution

Atomic Execution

Critical Section Execution



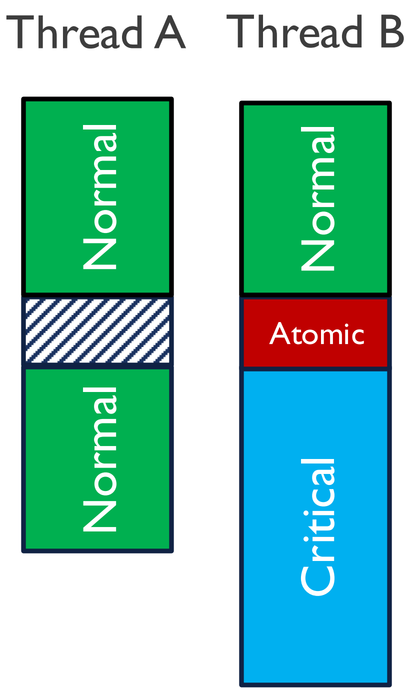
CRITICAL SECTION

No Execution

Normal Execution

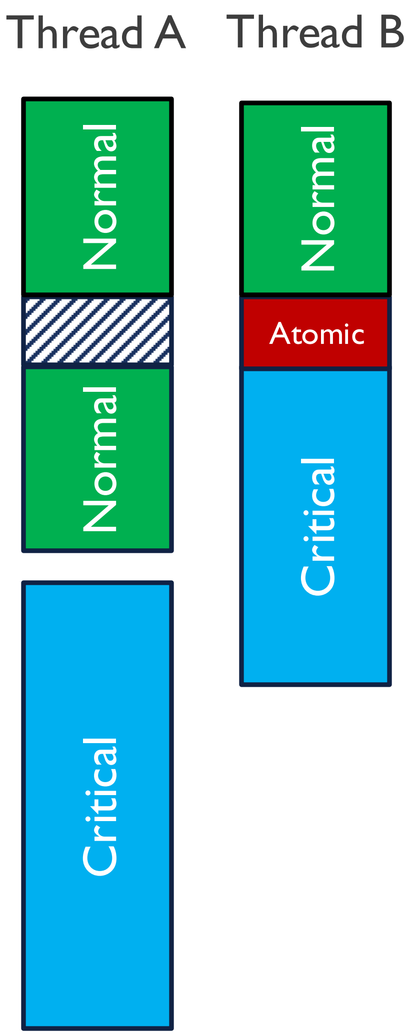
Atomic Execution

Critical Section Execution



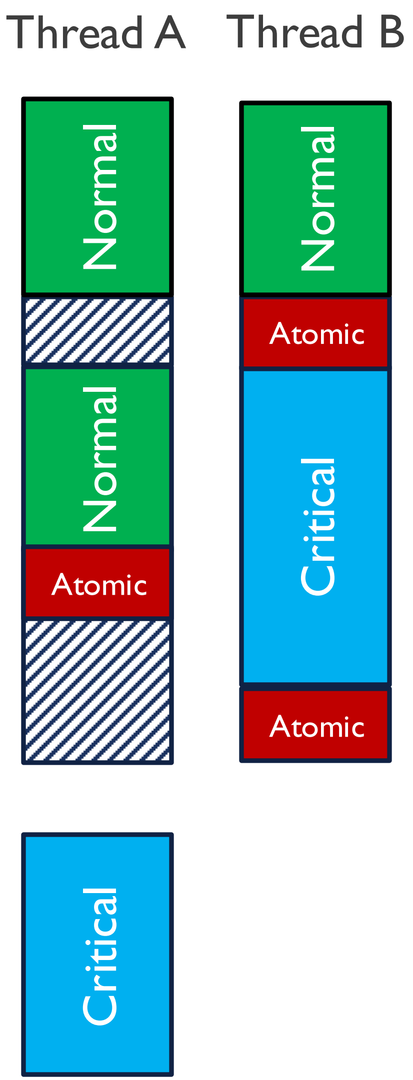
CRITICAL SECTION

(hatched box)	No Execution
(green box)	Normal Execution
(red box)	Atomic Execution
(blue box)	Critical Section Execution



CRITICAL SECTION

(hatched)	No Execution
(green)	Normal Execution
(red)	Atomic Execution
(blue)	Critical Section Execution



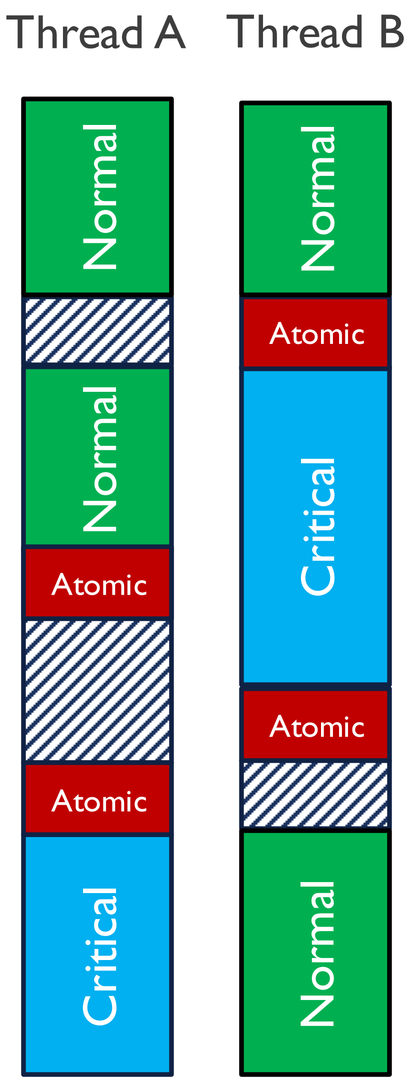
CRITICAL SECTION

No Execution

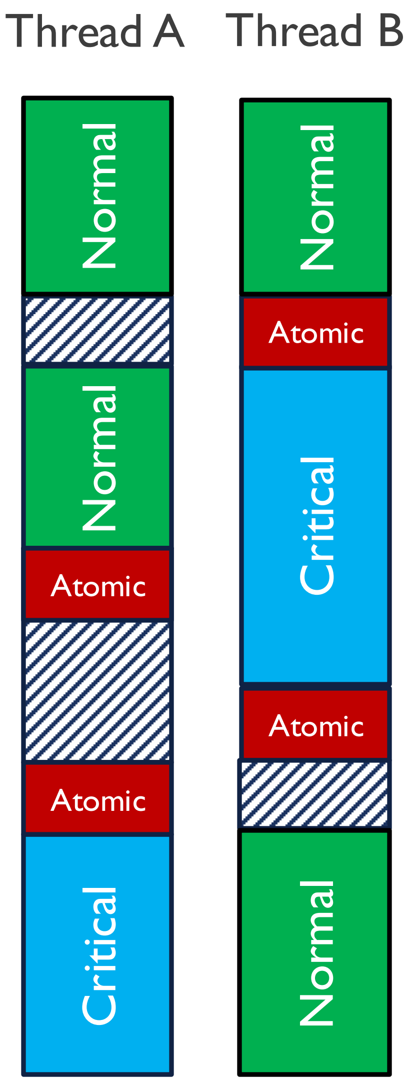
Normal Execution

Atomic Execution

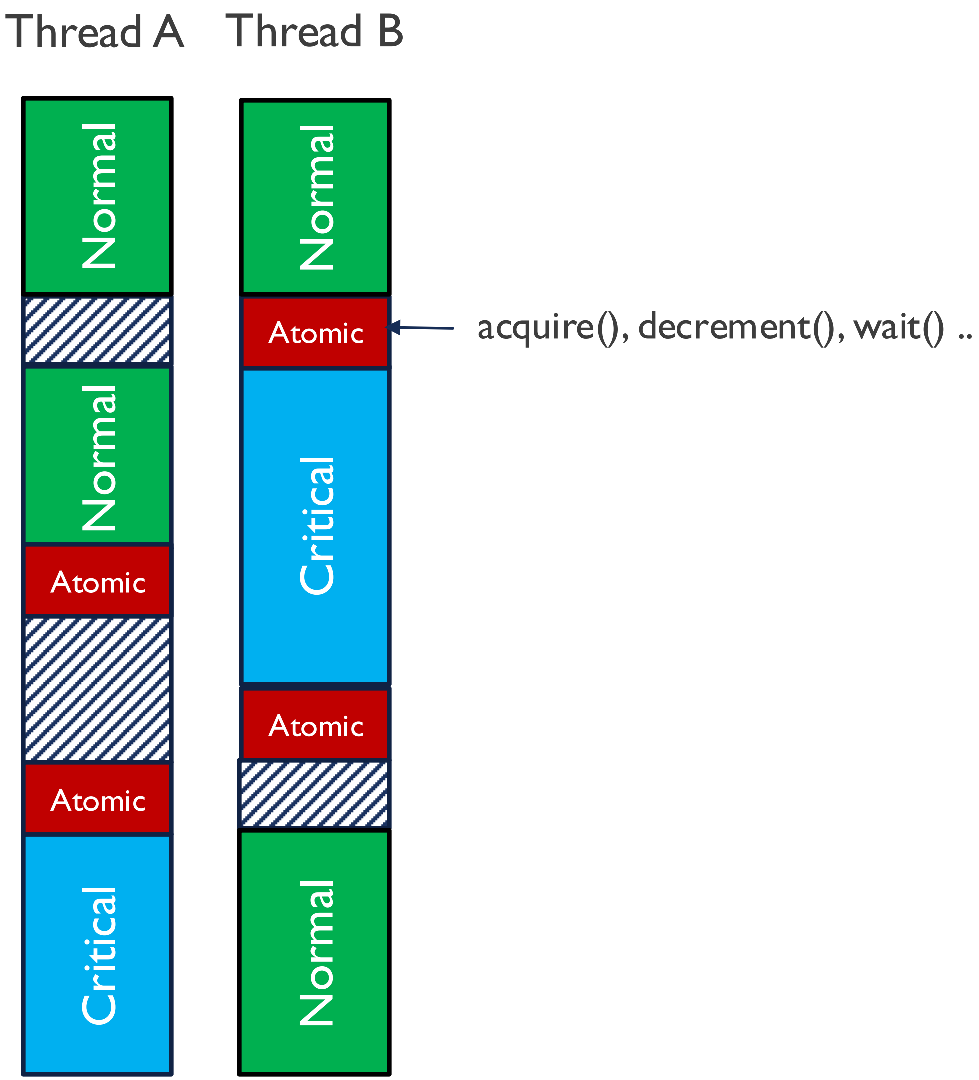
Critical Section Execution



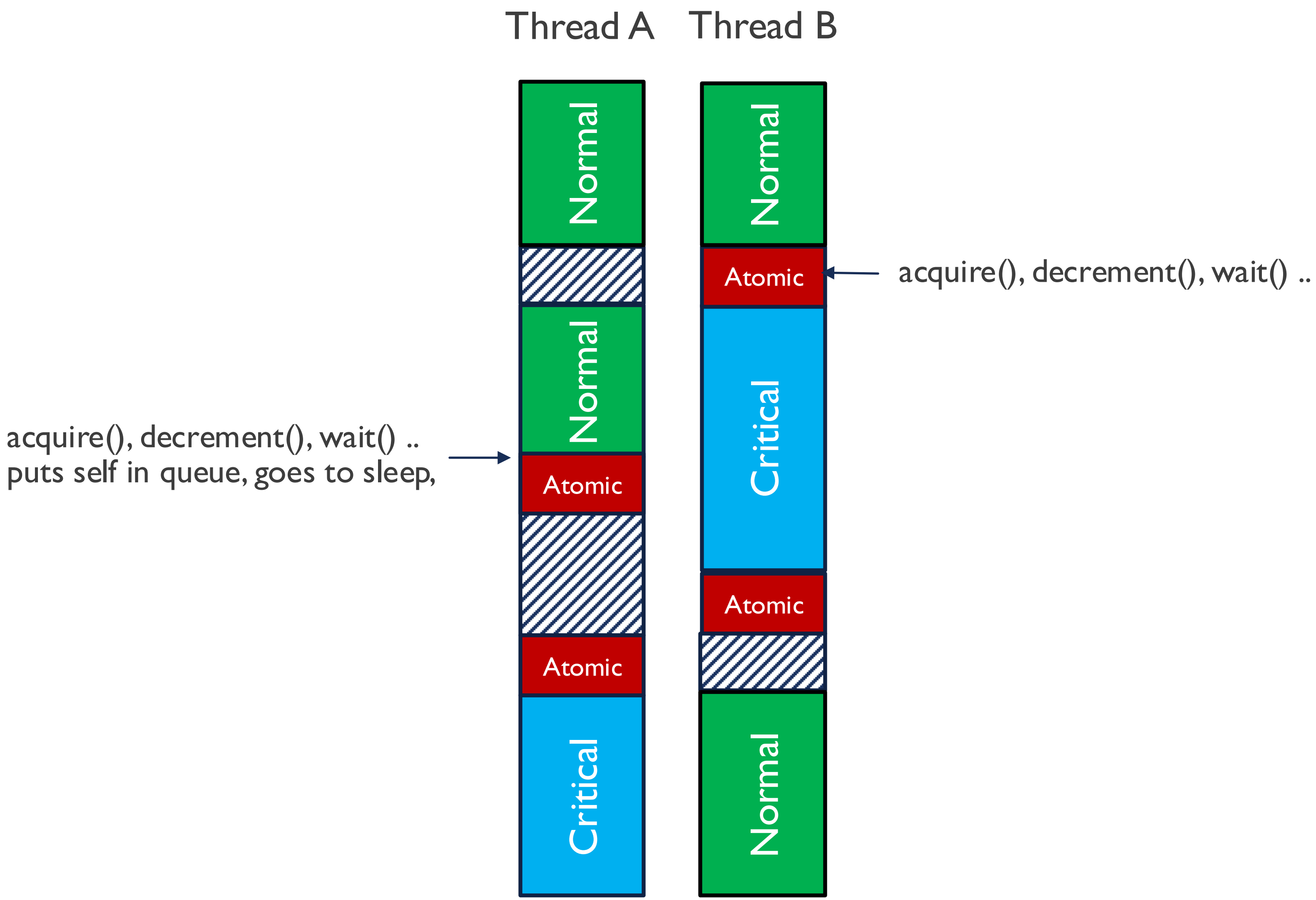
CRITICAL SECTION



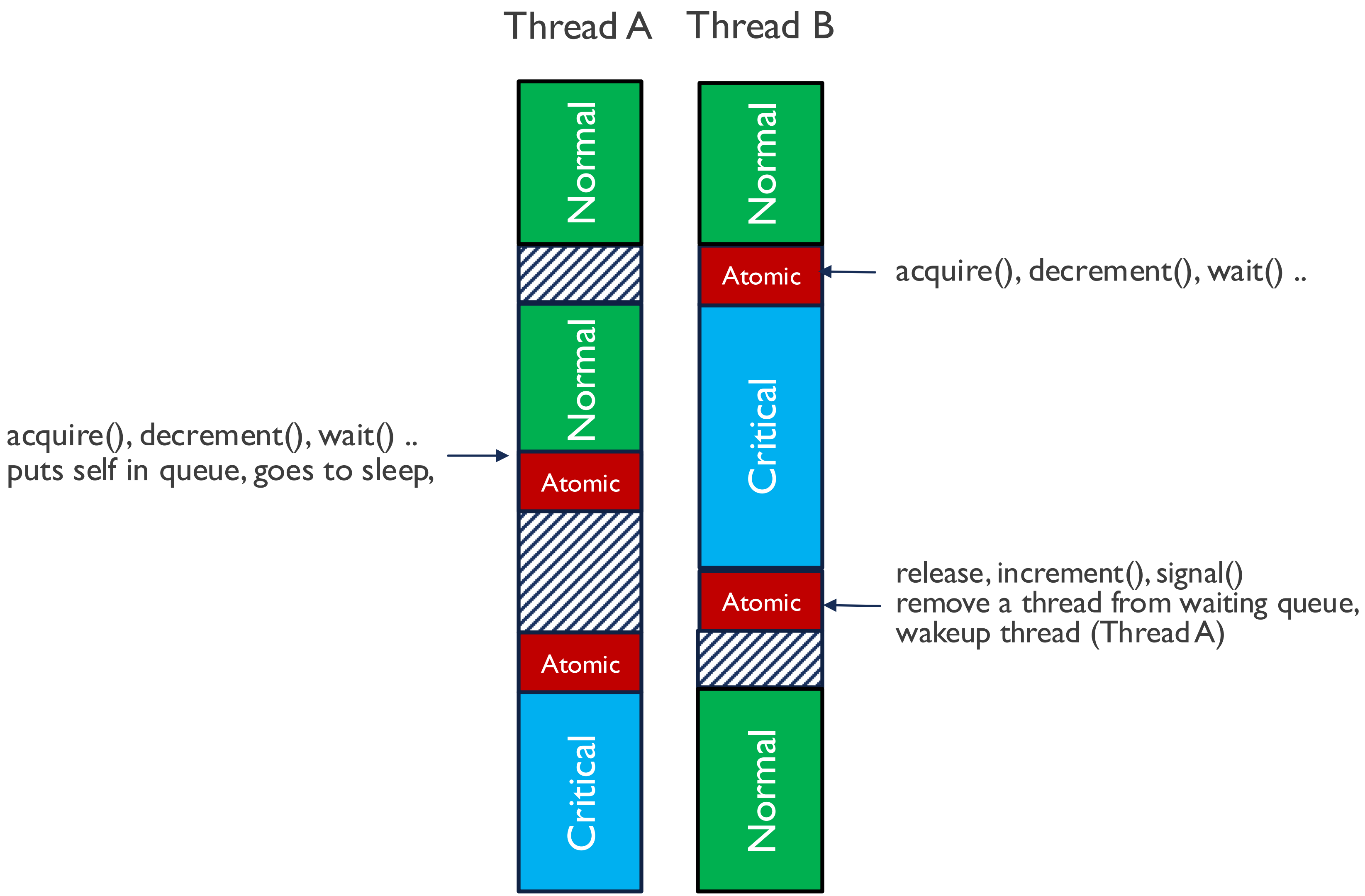
CRITICAL SECTION



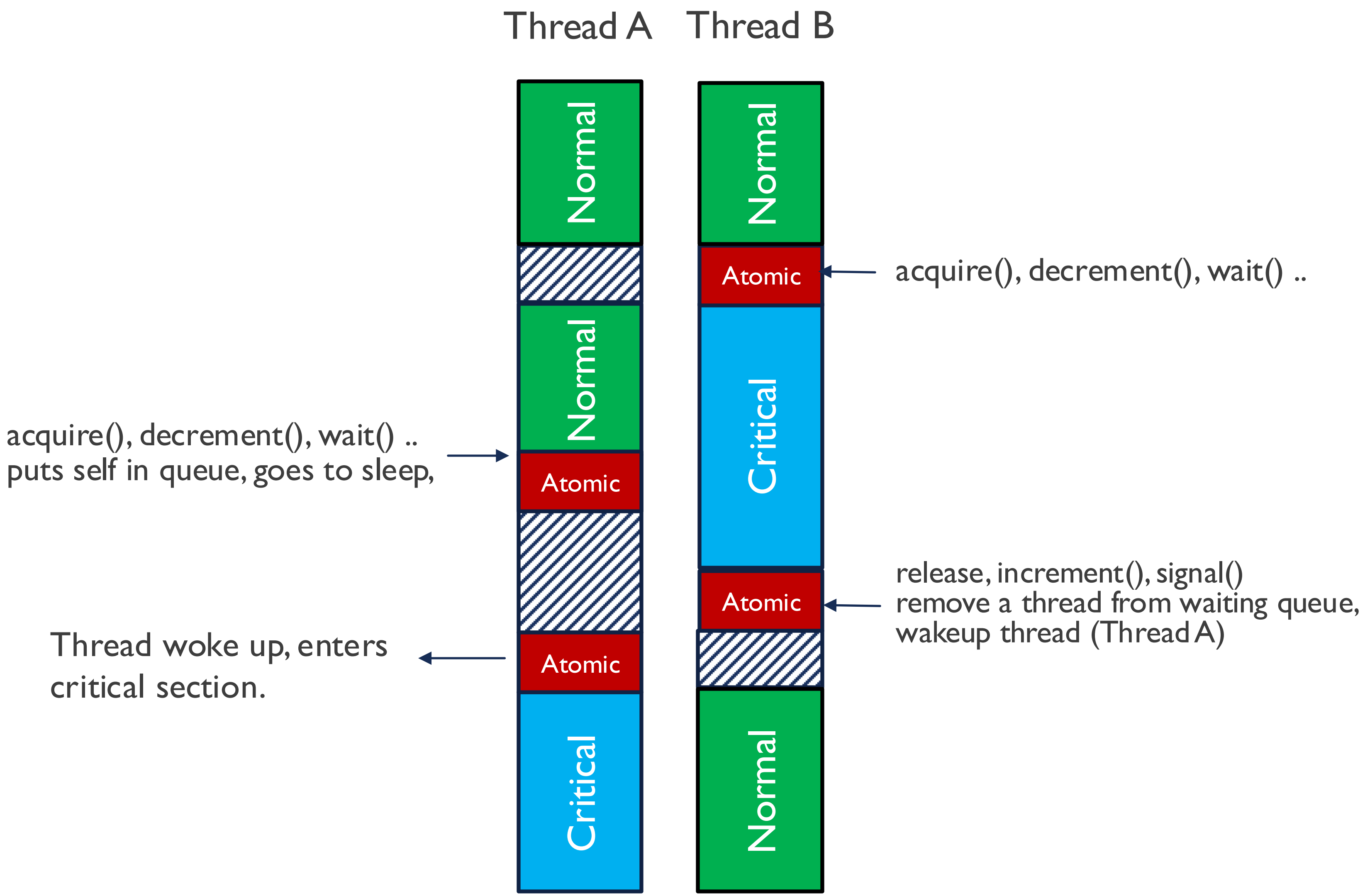
CRITICAL SECTION



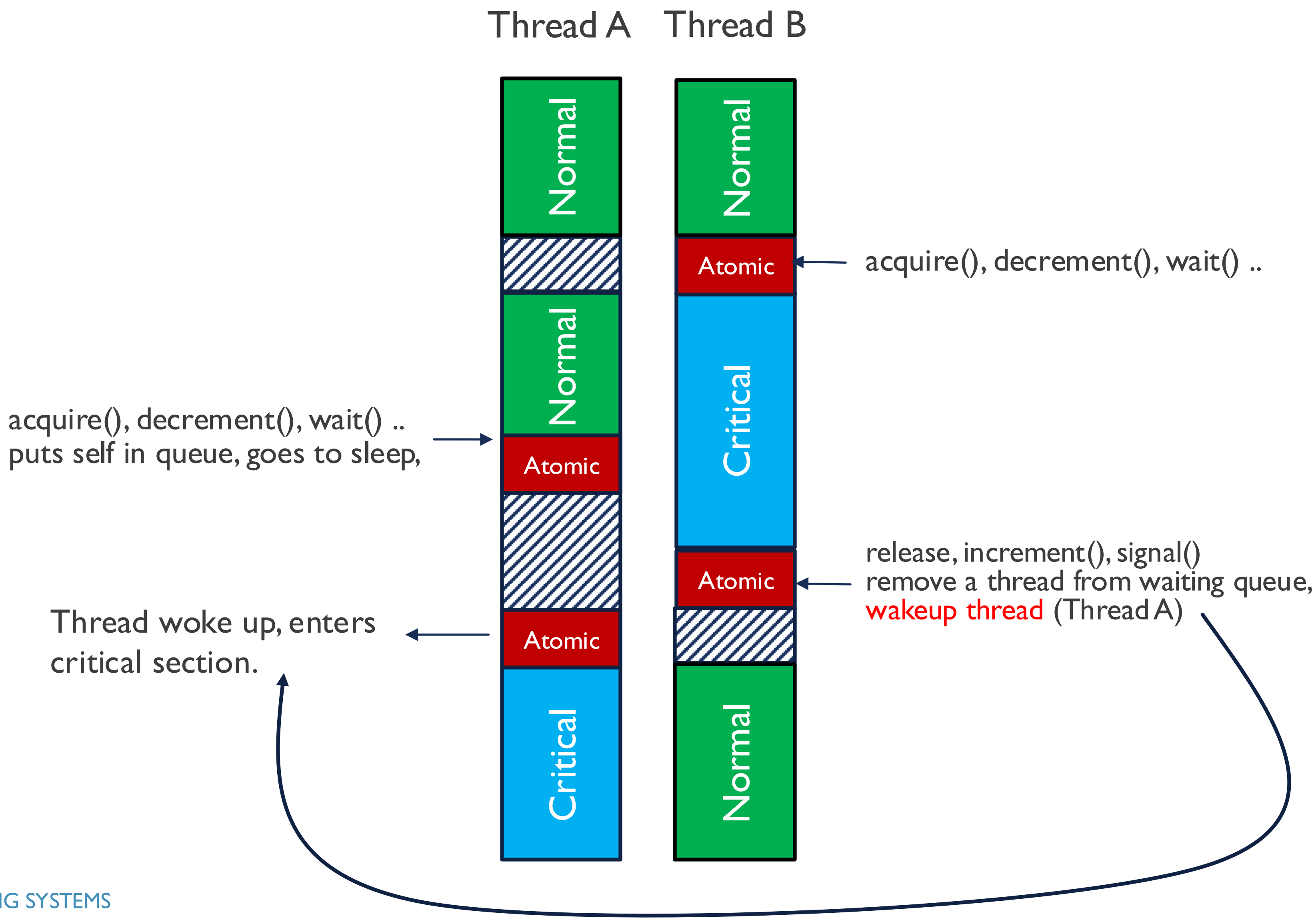
CRITICAL SECTION



CRITICAL SECTION



CRITICAL SECTION



CRITICAL SECTION