

# CS 3502

# Operating Systems

## Lock

Kun Suo

Computer Science, Kennesaw State University

<https://kevinsuo.github.io/>

# Outline

---

- Start from examples
- Concurrency and synchronization
  - Race condition
  - Critical section
- Mutual exclusion
  - Spinlock
  - Mutex lock
  - Semaphore
  - Deadlock and priority inversion



# Concurrency Example

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

- **thread.c** (What does this program do?)

Expected output?

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```



# Concurrency Example

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

Expected output?

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

Counter value: before

Two threads increase a counter

Counter value: after



# Concurrency Example



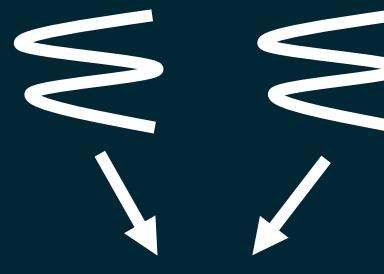
./cpu.o /home/pi

● #1

fish /home/pi

#2

```
pi@raspberrypi ~> gcc threads.c -pthread -o threads.o
pi@raspberrypi ~> ./threads.o 1
Initial value : 0
Final value   : 2
pi@raspberrypi ~> ./threads.o 10
Initial value : 0
Final value   : 20
pi@raspberrypi ~> ./threads.o 100
Initial value : 0
Final value   : 200
pi@raspberrypi ~> ./threads.o 1000
Initial value : 0
Final value   : 2000
pi@raspberrypi ~> ./threads.o 10000
Initial value : 0
Final value   : 20000
pi@raspberrypi ~> ./threads.o 100000
Initial value : 0
Final value   : 200000
pi@raspberrypi ~> ./threads.o 1000000
Initial value : 0
Final value   : 2000000
pi@raspberrypi ~> ./threads.o 10000000
Initial value : 0
Final value   : 20000000
pi@raspberrypi ~> ./threads.o 100000000
Initial value : 0
Final value   : 200000000
pi@raspberrypi ~> ./threads.o 1000000000
Initial value : 0
Final value   : 2000000000
pi@raspberrypi ~> ./threads.o 10000000000
Initial value : 0
Final value   : 20000000000
pi@raspberrypi ~> ./threads.o 100000000000
Initial value : 0
Final value   : 200000000000
pi@raspberrypi ~> ./threads.o 1000000000000
Initial value : 0
Final value   : 2000000000000
pi@raspberrypi ~> ./threads.o 10000000000000
Initial value : 0
Final value   : 20000000000000
pi@raspberrypi ~> ./threads.o 100000000000000
Initial value : 0
Final value   : 200000000000000
pi@raspberrypi ~> ./threads.o 1000000000000000
Initial value : 0
Final value   : 2000000000000000
pi@raspberrypi ~> ./threads.o 10000000000000000
Initial value : 0
Final value   : 20000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000
Initial value : 0
Final value   : 200000000000000000
pi@raspberrypi ~> ./threads.o 1000000000000000000
Initial value : 0
Final value   : 2000000000000000000
pi@raspberrypi ~> ./threads.o 10000000000000000000
Initial value : 0
Final value   : 20000000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000000
Initial value : 0
Final value   : 200000000000000000000
pi@raspberrypi ~> ./threads.o 1000000000000000000000
Initial value : 0
Final value   : 2000000000000000000000
pi@raspberrypi ~> ./threads.o 10000000000000000000000
Initial value : 0
Final value   : 20000000000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000000000
Initial value : 0
Final value   : 200000000000000000000000
pi@raspberrypi ~> ./threads.o 1000000000000000000000000
Initial value : 0
Final value   : 2000000000000000000000000
pi@raspberrypi ~> ./threads.o 10000000000000000000000000
Initial value : 0
Final value   : 20000000000000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000000000000
Initial value : 0
Final value   : 200000000000000000000000000
pi@raspberrypi ~> ./threads.o 1000000000000000000000000000
Initial value : 0
Final value   : 2000000000000000000000000000
pi@raspberrypi ~> ./threads.o 10000000000000000000000000000
Initial value : 0
Final value   : 20000000000000000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000000000000000
Initial value : 0
Final value   : 200000000000000000000000000000
pi@raspberrypi ~> ./threads.o 1000000000000000000000000000000
Initial value : 0
Final value   : 2000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 10000000000000000000000000000000
Initial value : 0
Final value   : 20000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000000000000000000
Initial value : 0
Final value   : 200000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 1000000000000000000000000000000000
Initial value : 0
Final value   : 2000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 10000000000000000000000000000000000
Initial value : 0
Final value   : 20000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000000000000000000000
Initial value : 0
Final value   : 200000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 1000000000000000000000000000000000000
Initial value : 0
Final value   : 2000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 10000000000000000000000000000000000000
Initial value : 0
Final value   : 20000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000000000000000000000000
Initial value : 0
Final value   : 200000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 1000000000000000000000000000000000000000
Initial value : 0
Final value   : 2000000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 10000000000000000000000000000000000000000
Initial value : 0
Final value   : 20000000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000000000000000000000000000
Initial value : 0
Final value   : 200000000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 1000000000000000000000000000000000000000000
Initial value : 0
Final value   : 2000000000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 10000000000000000000000000000000000000000000
Initial value : 0
Final value   : 20000000000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000000000000000000000000000000
Initial value : 0
Final value   : 200000000000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 1000000000000000000000000000000000000000000000
Initial value : 0
Final value   : 2000000000000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 10000000000000000000000000000000000000000000000
Initial value : 0
Final value   : 20000000000000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000000000000000000000000000000000
Initial value : 0
Final value   : 200000000000000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 1000000000000000000000000000000000000000000000000
Initial value : 0
Final value   : 2000000000000000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 10000000000000000000000000000000000000000000000000
Initial value : 0
Final value   : 20000000000000000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000000000000000000000000000000000000
Initial value : 0
Final value   : 200000000000000000000000000000000000000000000000000
pi@raspberrypi ~> ./threads.o 100000000000000000000000000000000000000000000000000
Initial value : 0
Final value   : 200000000000000000000000000000000000000000000000000
```



[https://youtu.be/8SDd\\_I92hUI](https://youtu.be/8SDd_I92hUI)

# Concurrency Example

Reality?

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012 // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298 // what the??
```

A key part of the program above, where the shared counter is incremented, takes three instructions:

- one to load the value of the counter from memory into a register,
- one to increment it, and
- one to store it back into memory.

Because these three instructions do not execute **atomically** (all at once), strange things can happen.



```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "common_threads.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i, j;
    for (i = 0; i < loops; i++) {
        for (j = 0; j < 1000; j++) {
            counter++;
            counter--;
        }
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <loops>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

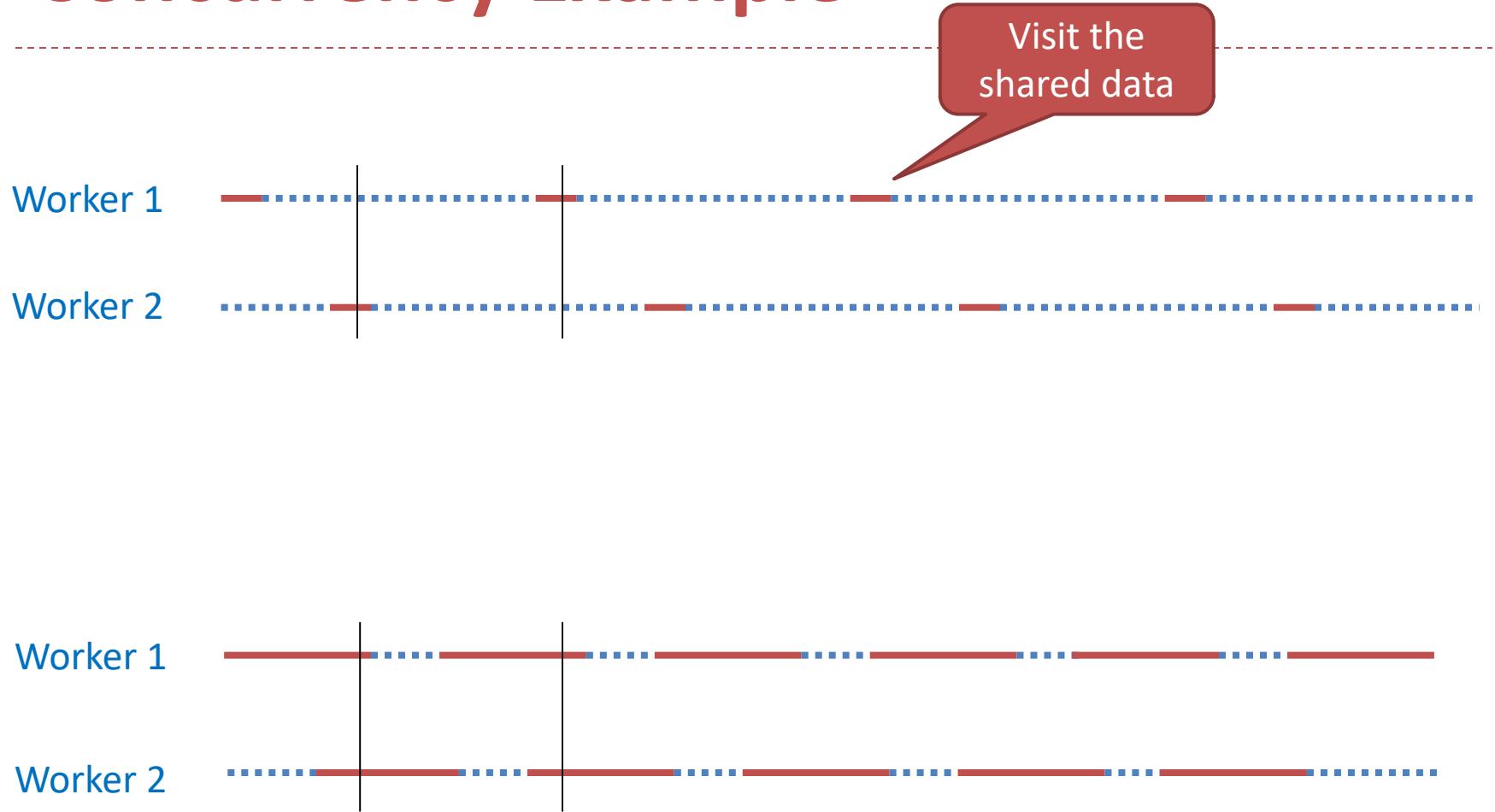
# Concurrency

## Example

---

For each iteration of i, it takes more time each round

# Concurrency Example



# Concurrency Example

```
fish /home/ksuo
ksuo@ksuo-VirtualBox ~> gcc threads.c -o threads.o -pthread
ksuo@ksuo-VirtualBox ~> ./threads.o 10
Initial value : 0
Final value   : 20
ksuo@ksuo-VirtualBox ~> ./threads.o 10
Initial value : 0
Final value   : 9
ksuo@ksuo-VirtualBox ~> ./threads.o 10
Initial value : 0
Final value   : 20
ksuo@ksuo-VirtualBox ~> ./threads.o 10
Initial value : 0
Final value   : 15
ksuo@ksuo-VirtualBox ~> ./threads.o 10
Initial value : 0
Final value   : 20
ksuo@ksuo-VirtualBox ~> ./threads.o 10
Initial value : 0
Final value   : 10
ksuo@ksuo-VirtualBox ~> ./threads.o 100
Initial value : 0
Final value   : 127
ksuo@ksuo-VirtualBox ~> ./threads.o 100
Initial value : 0
Final value   : 100
ksuo@ksuo-VirtualBox ~>
```

# Outline

---

- Start from examples
- Concurrency and synchronization
  - Race condition
  - Critical section
- Mutual exclusion
  - Spinlock
  - Mutex lock
  - Semaphore
  - Deadlock and priority inversion



# Race condition

[https://github.com/kevinsuo/CS3502/blob/master/race\\_condition.c](https://github.com/kevinsuo/CS3502/blob/master/race_condition.c)

- A race condition occurs when two or more threads access shared data and they try to change it at the same time.
- The order in which the threads attempt to access the shared data makes the results unpredictable

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

int counter = 0;

void *compute()
{
    int i = 0;
    while (i < 100) {
        counter = counter + 1;
        i++;
    }
    printf("Counter value: %d\n", counter);
}

int main()
{
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, compute, (void *)&thread1);
    pthread_create(&thread2, NULL, compute, (void *)&thread2);

    pthread_exit(NULL);
    exit(0);
}
```

Race condition occurs for variable counter

```
pi@raspberrypi ~/Downloads> ./race_condition.o
Counter value: 100
Counter value: 200
```

Seem nothing wrong?

# Race condition example

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

int counter = 0;

void *compute()
{
    int i = 0;
    while (i < 10000) {
        counter = counter + 1;
        i++;
    }
    printf("Counter value: %d\n", counter);
}

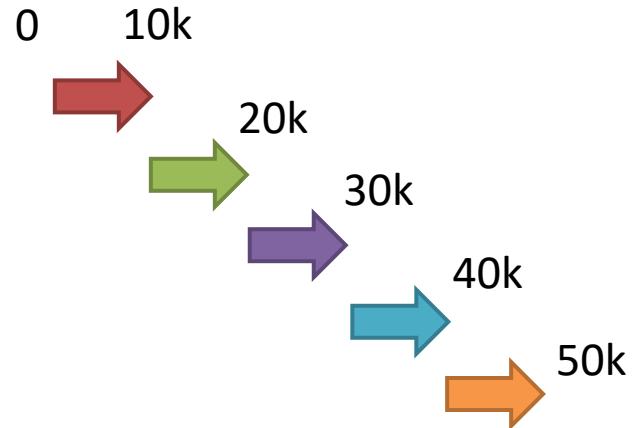
int main()
{
    pthread_t thread1, thread2, thread3, thread4, thread5;

    pthread_create(&thread1, NULL, compute, (void *)&thread1);
    pthread_create(&thread2, NULL, compute, (void *)&thread2);
    pthread_create(&thread3, NULL, compute, (void *)&thread3);
    pthread_create(&thread4, NULL, compute, (void *)&thread4);
    pthread_create(&thread5, NULL, compute, (void *)&thread5);

    pthread_exit(NULL);
    exit(0);
}
```

Increase the loop number

Add more threads



```
pi@raspberrypi ~/Downloads> ./race_condition.o
Counter value: 14467
Counter value: 10410
Counter value: 12080
Counter value: 22745
Counter value: 32725
```

Weird results!



# Critical section

- A section of code in a concurrent task that **modifies or accesses** a resource shared with another task.
- Examples
  - A piece of code that reads from or writes to a shared memory region
  - Or a code that modifies or traverses a shared linked list.

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```

# Critical section example

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

int counter = 0;

void *compute()
{
    int i = 0;
    while (i < 100) {
        counter = counter + 1;
        i++;
    }
    printf("Counter value: %d\n", counter);
}

int main()
{
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, compute, (void *)&thread1);
    pthread_create(&thread2, NULL, compute, (void *)&thread2);

    pthread_exit(NULL);
    exit(0);
}
```

Critical section: All threads read and write the shared counter



# Critical section vs. Race condition

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

int counter = 0;

void *compute()
{
    int i = 0;
    while (i < 100) {
        counter = counter + 1;
        i++;
    }
    printf("Counter value: %d\n", counter);
}

int main()
{
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, compute, (void *)&thread1);
    pthread_create(&thread2, NULL, compute, (void *)&thread2);

    pthread_exit(NULL);
    exit(0);
}
```

Critical section is where the race condition happens.

When multiple threads visit the critical section, race condition problem appears!



# Outline

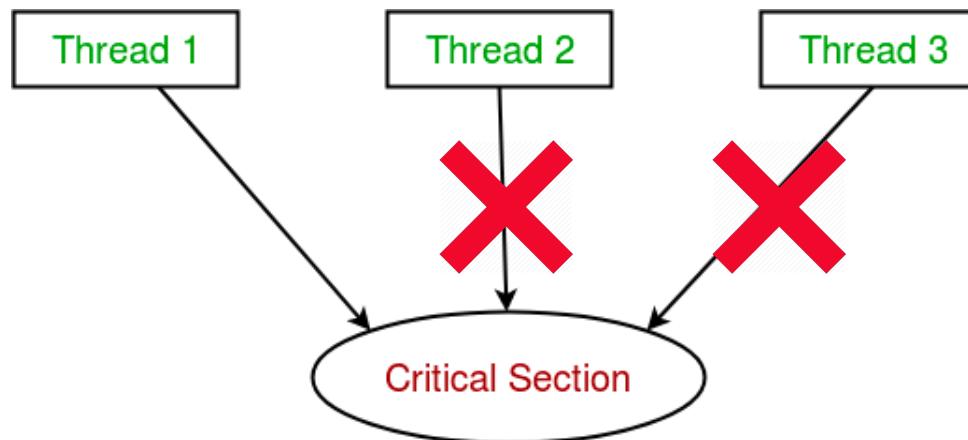
---

- Start from examples
- Concurrency and synchronization
  - Race condition
  - Critical section
- Mutual exclusion
  - Spinlock
  - Mutex lock
  - Semaphore
  - Deadlock and priority inversion



# To avoid race condition

- Principles:
  1. No two processes are simultaneously in the critical region

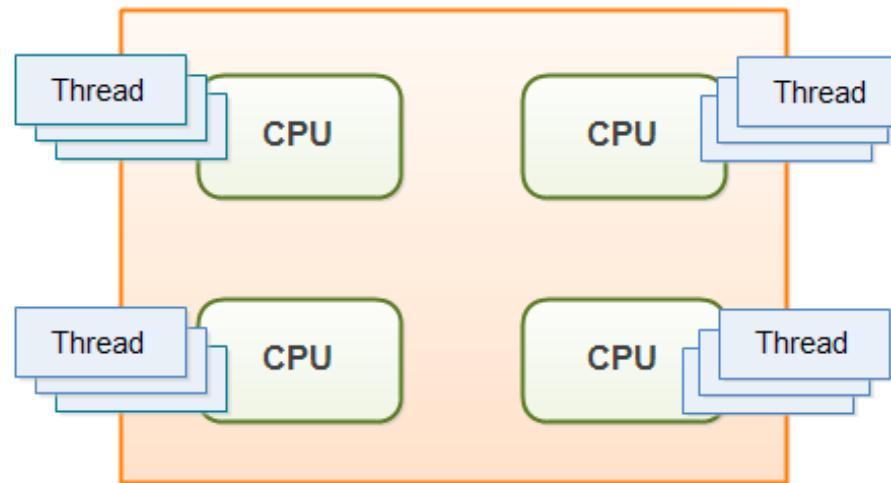


# To avoid race condition

- Principles:

2. No assumptions are made about speeds or numbers of CPUs

Thread could have varied speeds

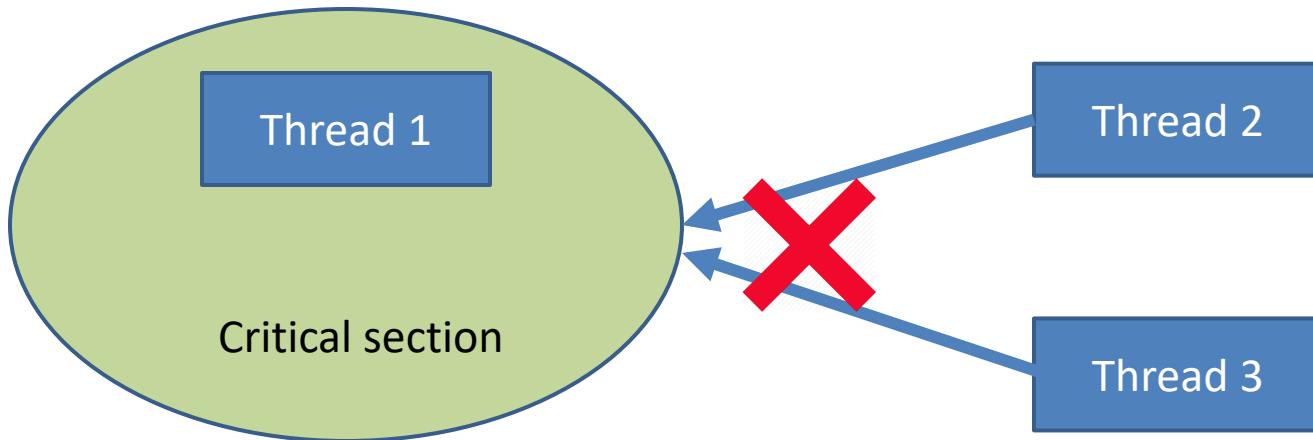


Thread could exist at each core

# To avoid race condition

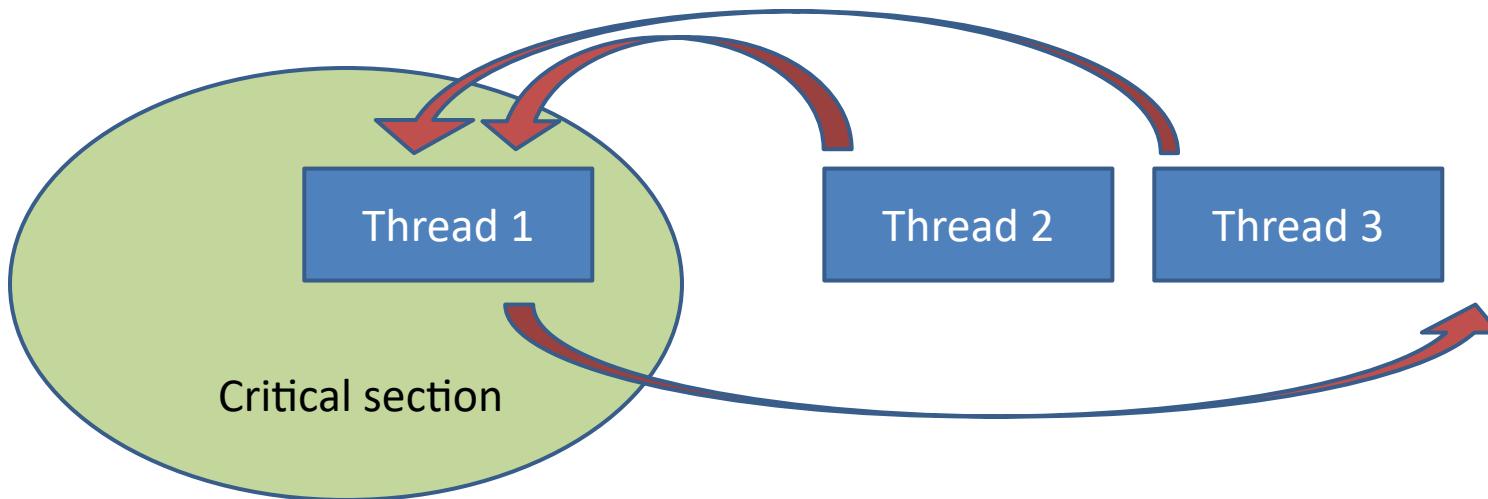
- Principles:

3. No process running outside its critical region may block another process running in the critical region



# To avoid race condition

- Principles:
  4. No process must wait forever to enter its critical region
    - ▶ Waiting forever indicates a deadlock



# To avoid race condition

---

- Principles:

1. No two processes are simultaneously in the critical region
2. No assumptions are made about speeds or numbers of CPUs

OS lock

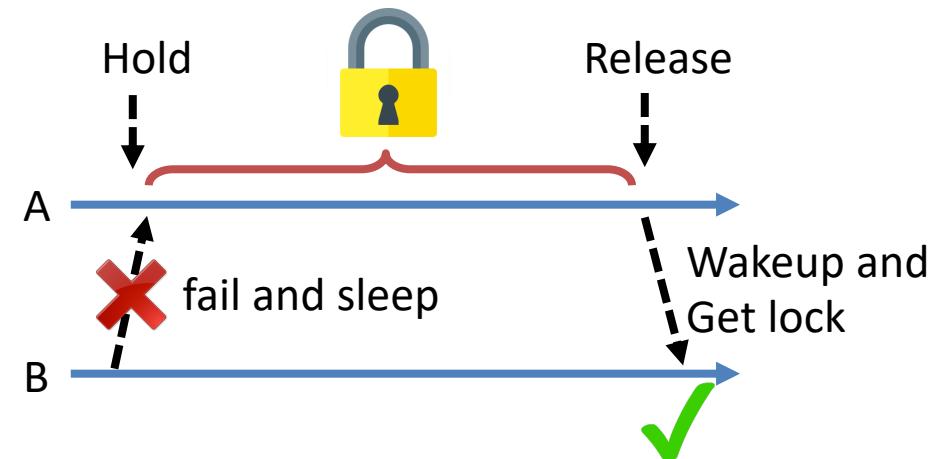
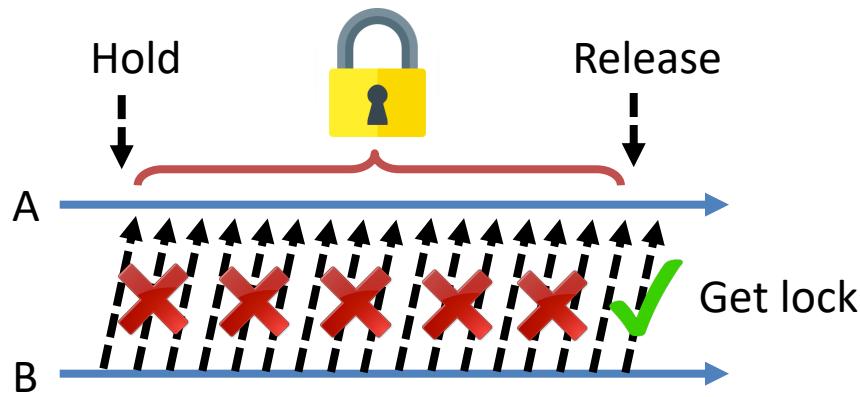
3. No process running outside its critical region may block another process running in the critical region
4. No process must wait forever to enter its critical region
  - ▶ Waiting forever indicates a deadlock

- (1) and (2) are enforced by the operating system's implementation of locks
  - Programmers assume that locks satisfy (1) and (2)
- (3) and (4) must be ensured by the programmer using the locks.
  - OS cannot enforce these.



# Lock (mutual exclusion)

- A lock (mutual exclusion) is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution
- Types of mutual mechanism:
  - Busy-waiting, e.g., spinlock
  - Sleep and wakeup

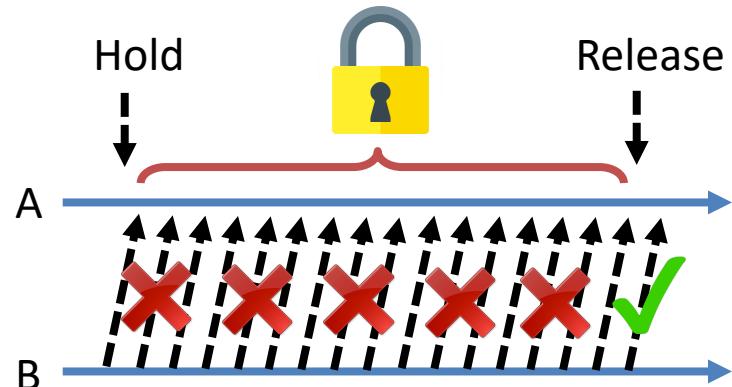


# 1, Spinlock: A busy-waiting lock implementation

- Don't block. Instead, constantly poll the lock for availability.
- Usage: small critical region
- Advantage
  - Very efficient with short critical sections
    - ▶ if you expect a lock to be released quickly
- Disadvantage
  - Doesn't yield the CPU and burns CPU cycles
    - ▶ Bad if critical sections are long.
  - Efficient only if machine has multiple CPUs.
    - ▶ Counterproductive on uniprocessor machines

```
while (lock is unavailable)
    continue; // try again
return success;
```

```
SpinLock(resource);
Execute Critical Section;
SpinUnlock(resource);
```



<https://github.com/kevinsuo/CS3502/blob/master/spinlock.c>

# Without Spinlock example

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

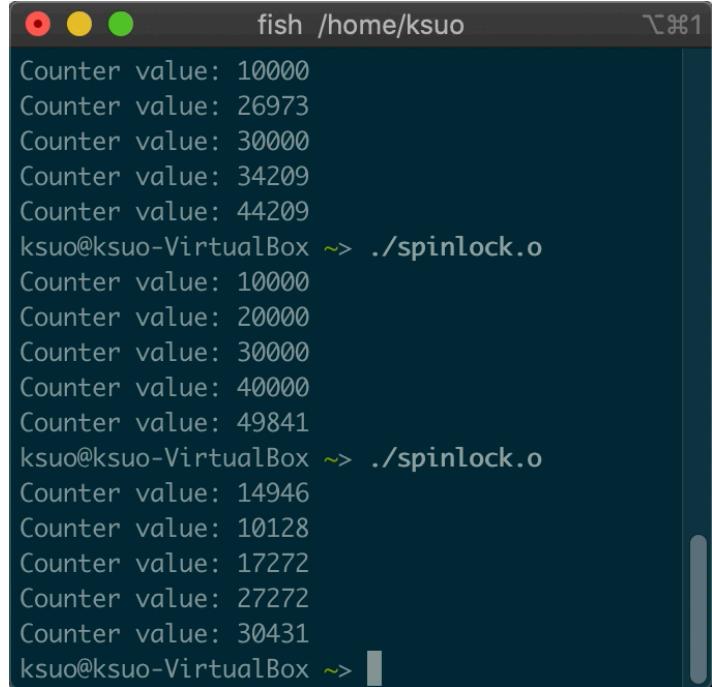
int counter = 0;
static pthread_spinlock_t slock;

void *compute()
{
    int i = 0;
    while (i < 10000) {
        counter = counter + 1;
        i++;
    }
    printf("Counter value: %d\n", counter);
}

int main()
{
    pthread_t thread1, thread2, thread3, thread4, thread5;

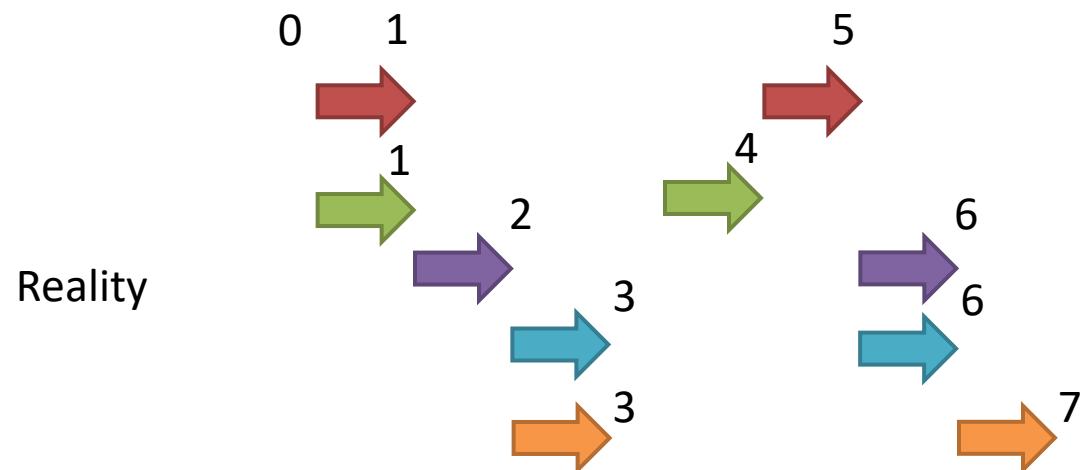
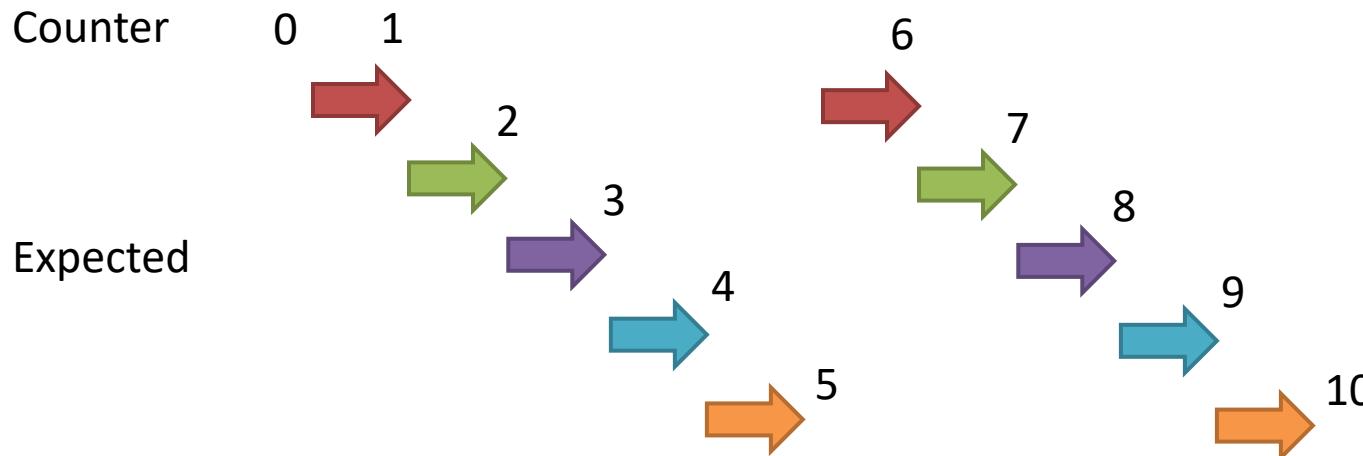
    pthread_create(&thread1, NULL, compute, (void *)&thread1);
    pthread_create(&thread2, NULL, compute, (void *)&thread2);
    pthread_create(&thread3, NULL, compute, (void *)&thread3);
    pthread_create(&thread4, NULL, compute, (void *)&thread4);
    pthread_create(&thread5, NULL, compute, (void *)&thread5);

    pthread_exit(NULL);
    exit(0);
}
```



```
Counter value: 10000
Counter value: 26973
Counter value: 30000
Counter value: 34209
Counter value: 44209
ksuo@ksuo-VirtualBox ~> ./spinlock.o
Counter value: 10000
Counter value: 20000
Counter value: 30000
Counter value: 40000
Counter value: 49841
ksuo@ksuo-VirtualBox ~> ./spinlock.o
Counter value: 14946
Counter value: 10128
Counter value: 17272
Counter value: 27272
Counter value: 30431
ksuo@ksuo-VirtualBox ~>
```

# Without Spinlock example



# Spinlock example

<https://github.com/kevinsuo/CS3502/blob/master/spinlock.c>

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

int counter = 0;
static pthread_spinlock_t slock;

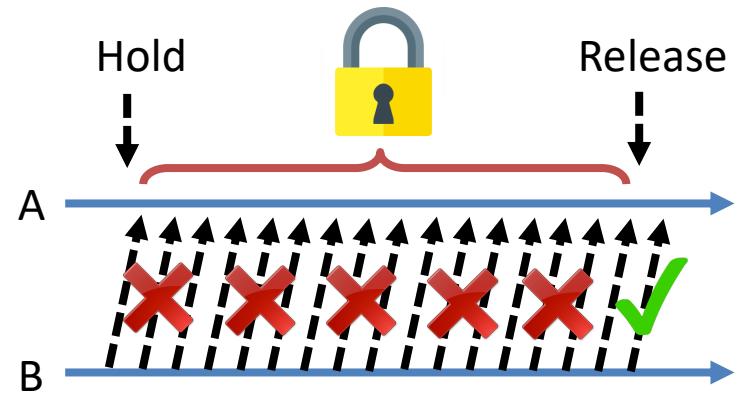
void *compute()
{
    int i = 0;
    pthread_spin_lock(&slock);
    while (i < 10000) {
        counter = counter + 1;
        i++;
    }
    printf("Counter value: %d\n", counter);
    pthread_spin_unlock(&slock);
}

int main()
{
    pthread_t thread1, thread2, thread3, thread4, thread5;

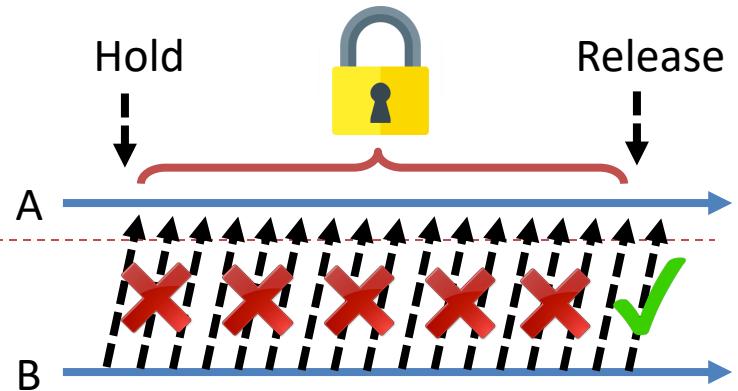
    pthread_create(&thread1, NULL, compute, (void *)&thread1);
    pthread_create(&thread2, NULL, compute, (void *)&thread2);
    pthread_create(&thread3, NULL, compute, (void *)&thread3);
    pthread_create(&thread4, NULL, compute, (void *)&thread4);
    pthread_create(&thread5, NULL, compute, (void *)&thread5);

    pthread_exit(NULL);
    exit(0);
}
```

```
ksuo@ksuo-VirtualBox ~/Desktop> ./spinlock.o
Counter value: 10000
Counter value: 20000
Counter value: 30000
Counter value: 40000
Counter value: 50000
```



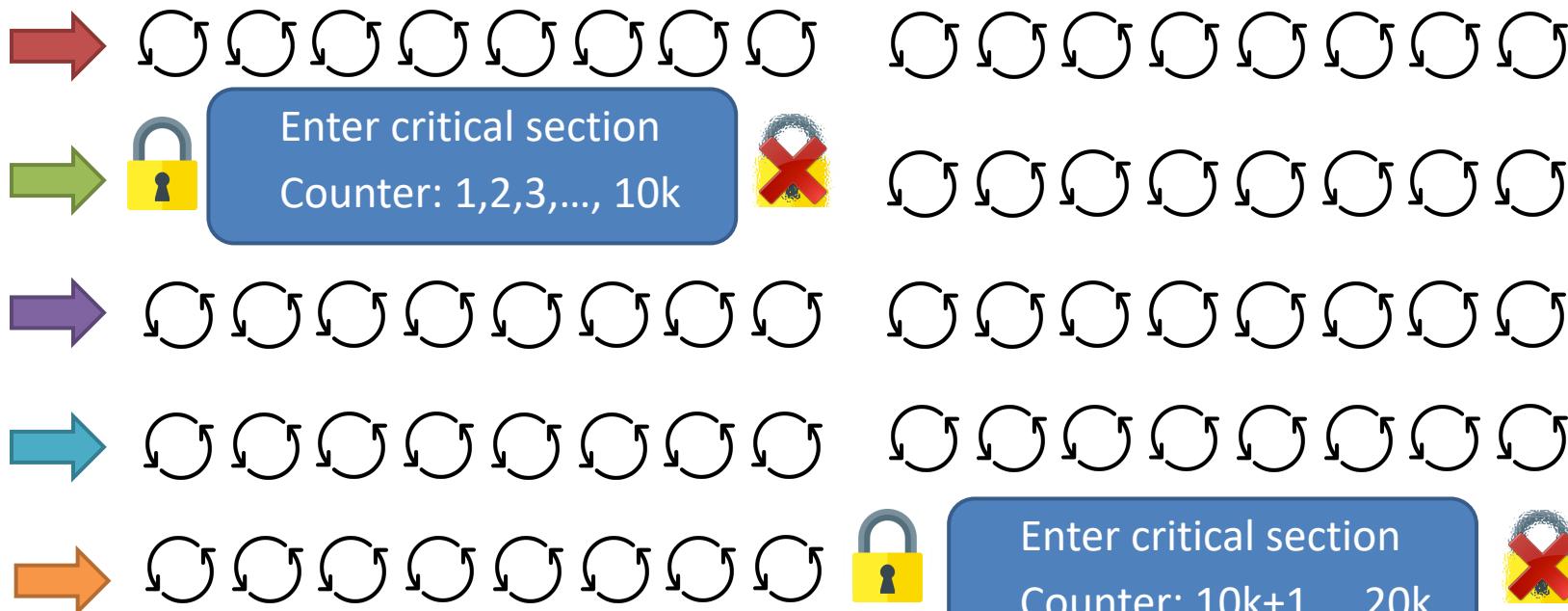
# Spinlock example



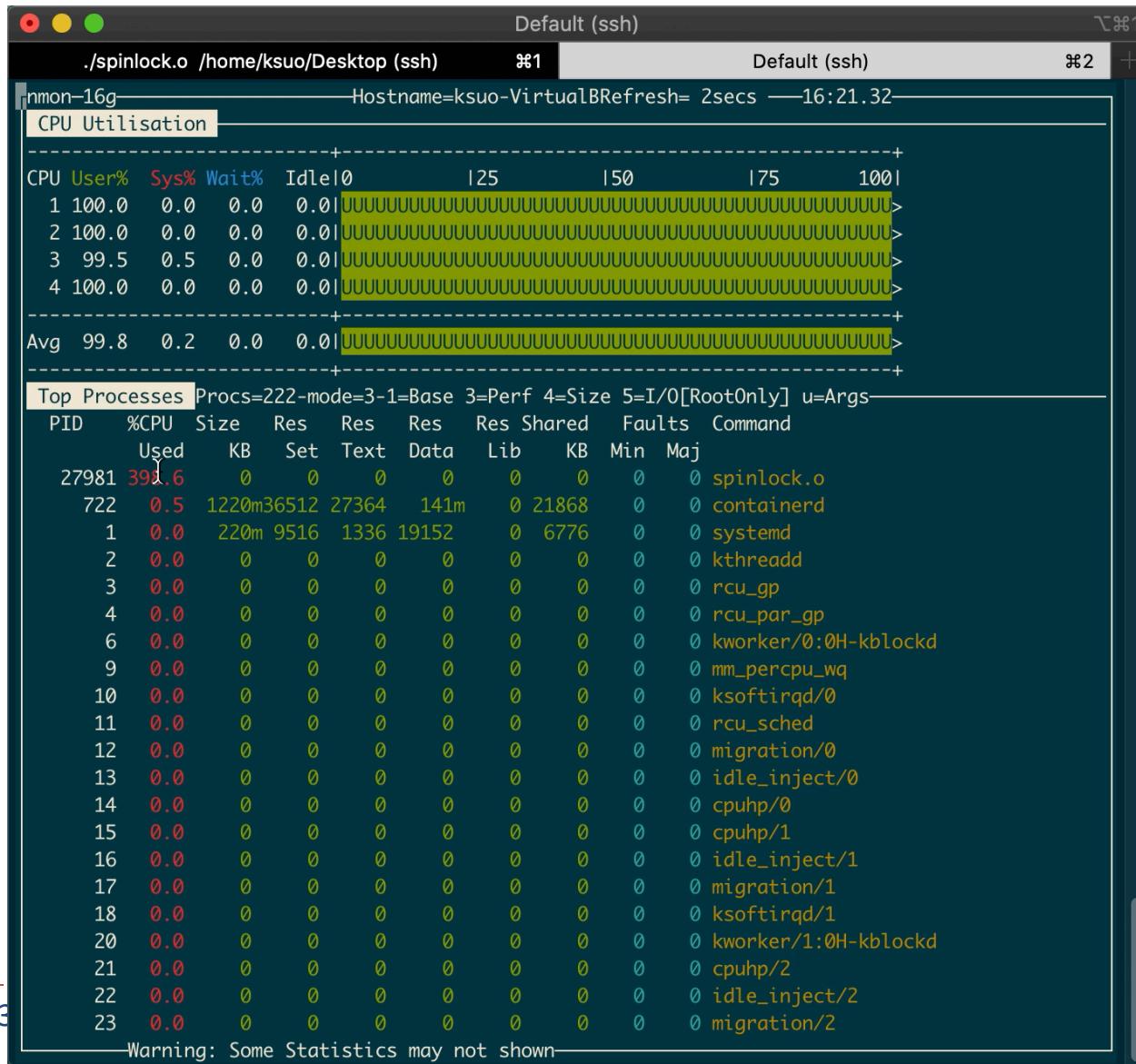
Counter  
is 0

Counter  
is 10k

Counter  
is 20k



# Spinlock example: CPU utilization



CS 3

ms

# Other mutual exclusion similar as busy waiting (spinlock)

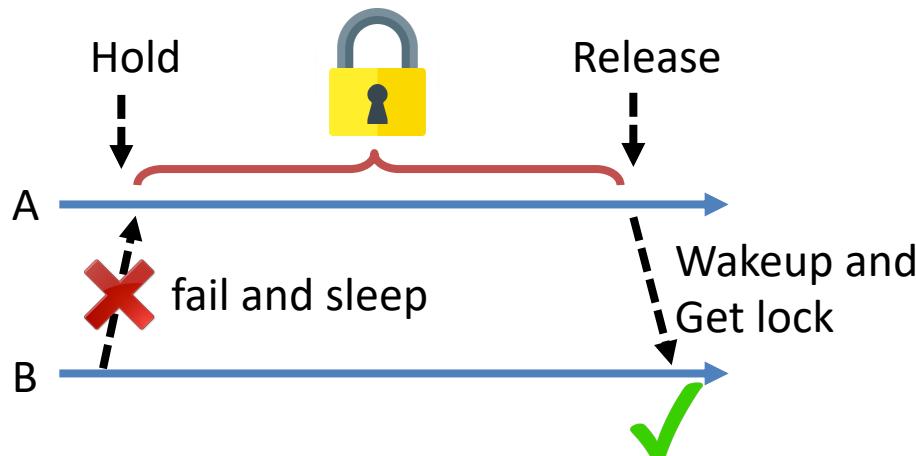
---

- Disabling interrupts:
  - OS technique, not users'
- Lock variables:
  - Test-and-set lock (TSL) is a two-step process, not atomic
- Peterson's algorithm
  - Does not need atomic operation and mainly used in user space application



## 2, Mutex lock: A sleep-and-wakeup lock implementation

- A variable that can be in one of two states: unlocked or locked
- Mutex is used as a LOCK around critical sections



Example:  
Lock(mutex)  
CriticalSection...  
Unlock(mutex)

### Pro:

Better cpu utilization

### Con:

Overhead on entering sleep or wake up  
Not suited for short duration of lock acquisition

# Mutex lock example

<https://github.com/kevinsuo/CS3502/blob/master/mutexlock.c>

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

int counter = 0;
static pthread_mutex_t mlock;

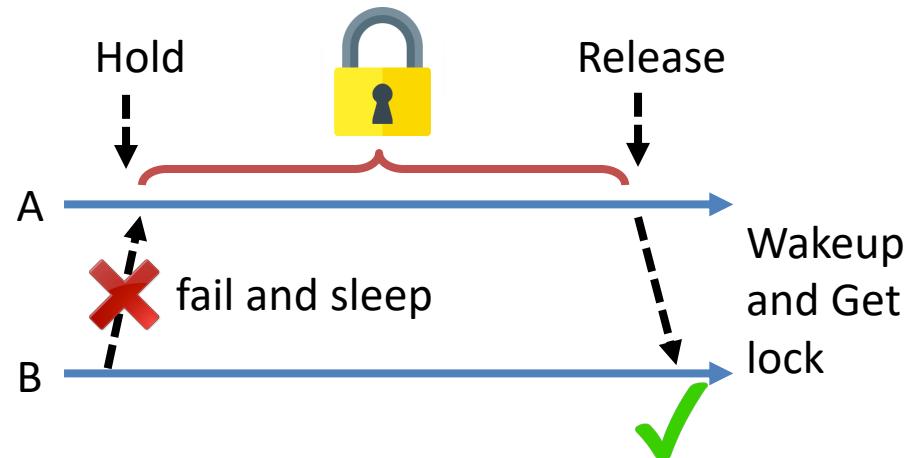
void *compute()
{
    int i = 0;
    pthread_mutex_lock(&mlock);
    while (i < 10000) {
        counter = counter + 1;
        i++;
    }
    printf("Counter value: %d\n", counter);
    pthread_mutex_unlock(&mlock);
}

int main()
{
    pthread_t thread1, thread2, thread3, thread4, thread5;

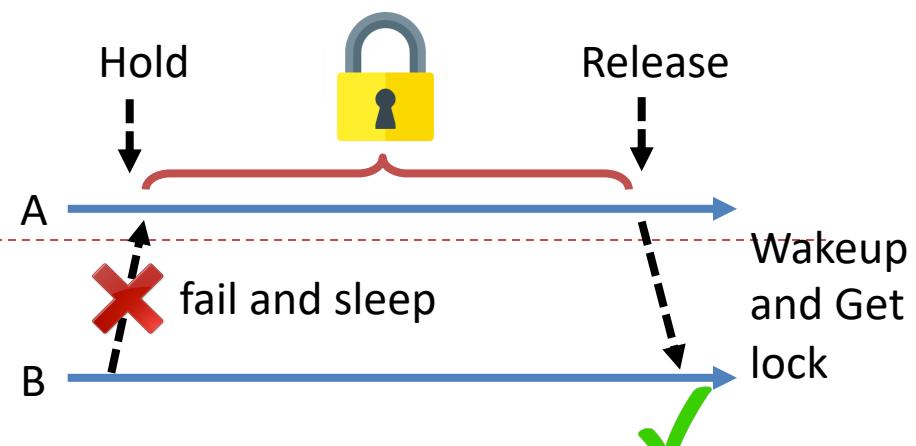
    pthread_create(&thread1, NULL, compute, (void *)&thread1);
    pthread_create(&thread2, NULL, compute, (void *)&thread2);
    pthread_create(&thread3, NULL, compute, (void *)&thread3);
    pthread_create(&thread4, NULL, compute, (void *)&thread4);
    pthread_create(&thread5, NULL, compute, (void *)&thread5);

    pthread_exit(NULL);
    exit(0);    CS 3502
}
```

```
pi@raspberrypi ~/Downloads> ./mutexlock.o
Counter value: 10000
Counter value: 20000
Counter value: 30000
Counter value: 40000
Counter value: 50000
```



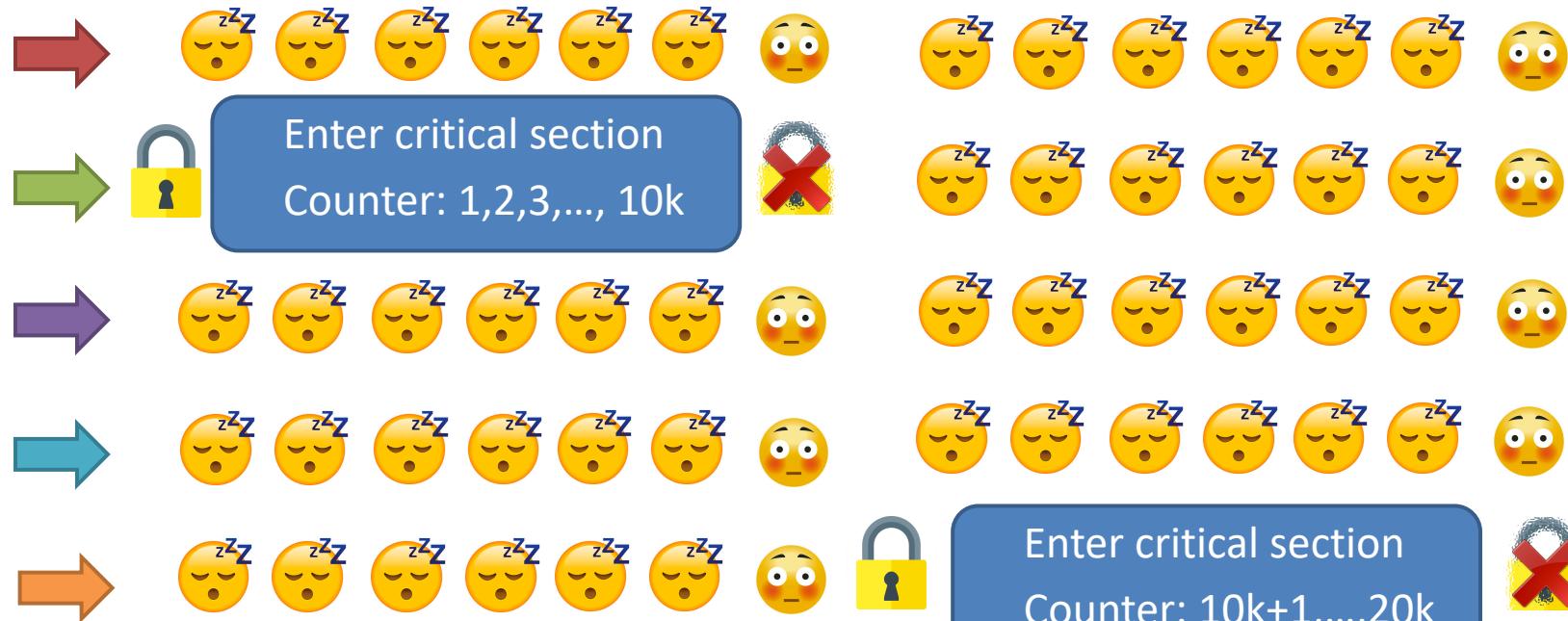
# Mutex example



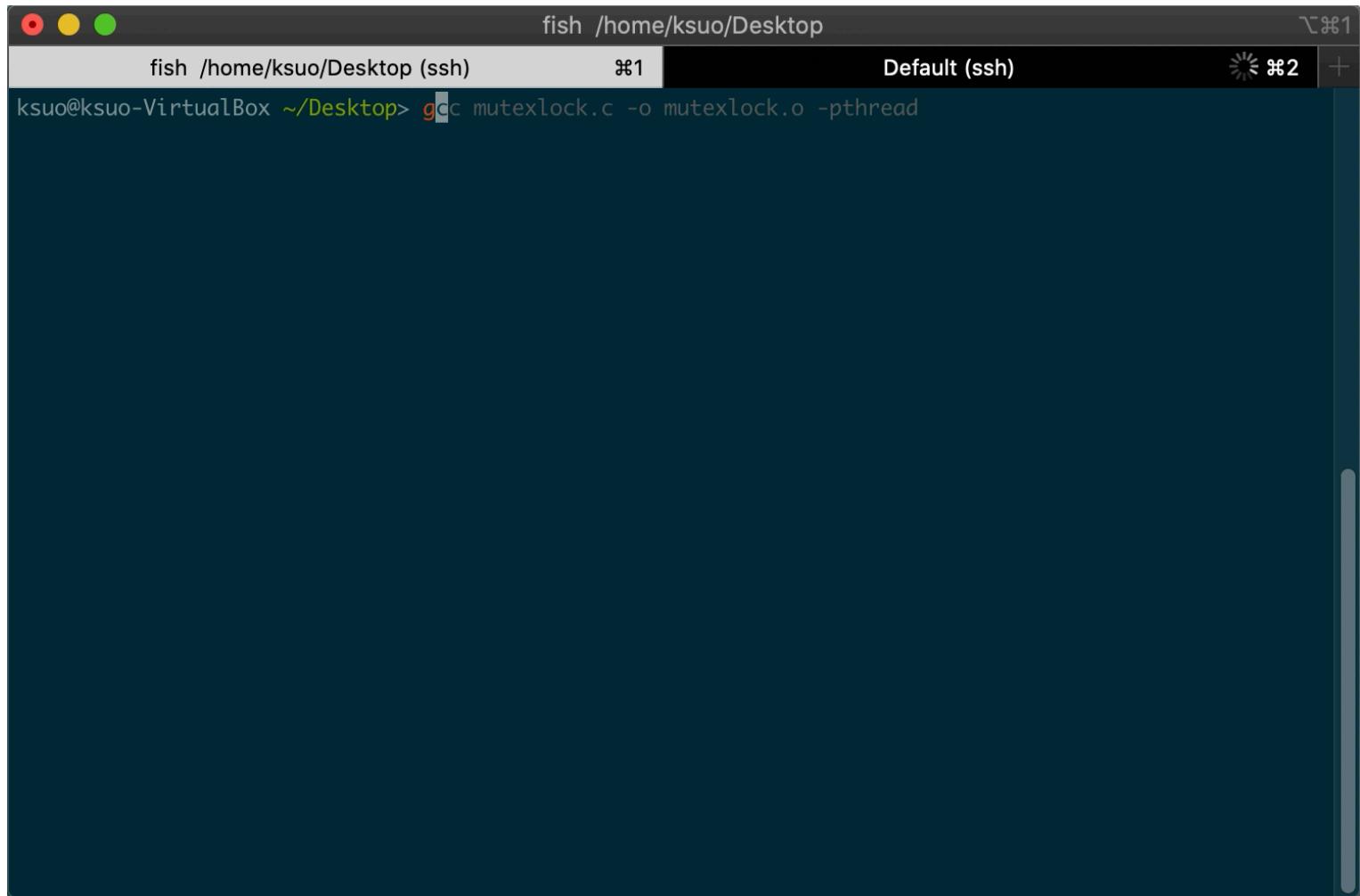
Counter  
is 0

Counter  
is 10k

Counter  
is 20k



# Mutex lock CPU utilization



A screenshot of a terminal window titled "fish /home/ksuo/Desktop". The window has two tabs: "fish /home/ksuo/Desktop (ssh)" (selected) and "Default (ssh)". The command entered is "gcc mutexlock.c -o mutexlock.o -pthread". The terminal is dark-themed.

```
fish /home/ksuo/Desktop
fish /home/ksuo/Desktop (ssh) ⌘1 Default (ssh) ⌘2 +
ksuo@ksuo-VirtualBox ~/Desktop> gcc mutexlock.c -o mutexlock.o -pthread
```

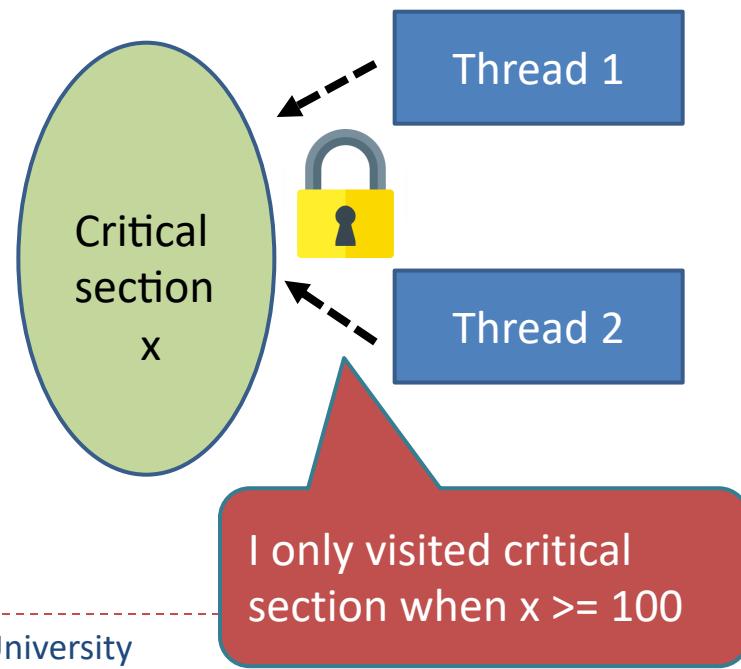
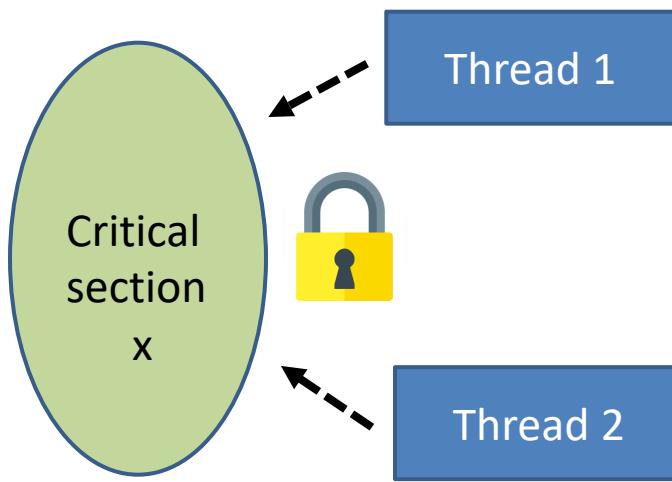
# Busy waiting lock vs Sleep wake up lock

	Mechanism	Use case	Implementation	Other examples
Busy waiting lock	constantly poll the lock for availability	When the waiting time is short	Spin lock	Disabling interrupts; Lock variables; Peterson's algorithm
Sleep wake up lock	Sleep if lock not available; wake up if available	When the waiting time is long	Mutex lock	Semaphore



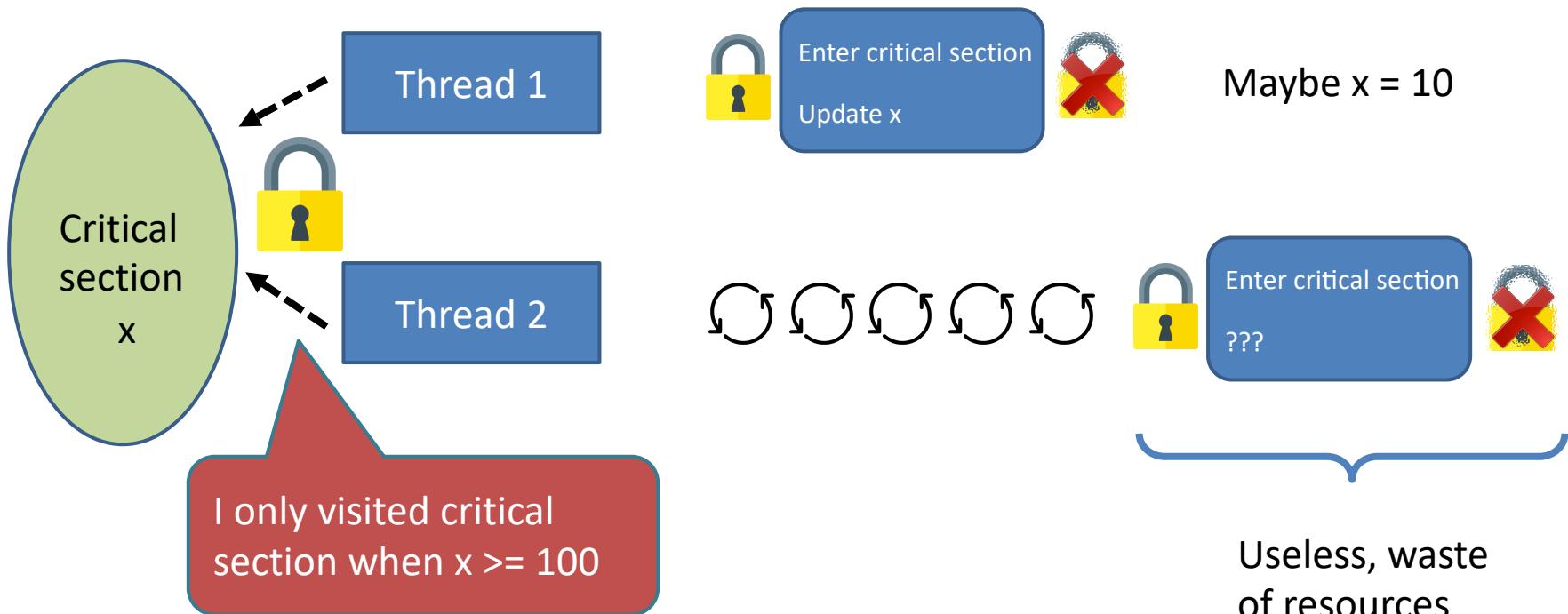
# 3. Mutex lock with conditions

- Mutex locks solve the competition problem of multiple threads accessing the same global variable under the shared memory space. (**without conditions**)
- How about competition **with condition** variables?



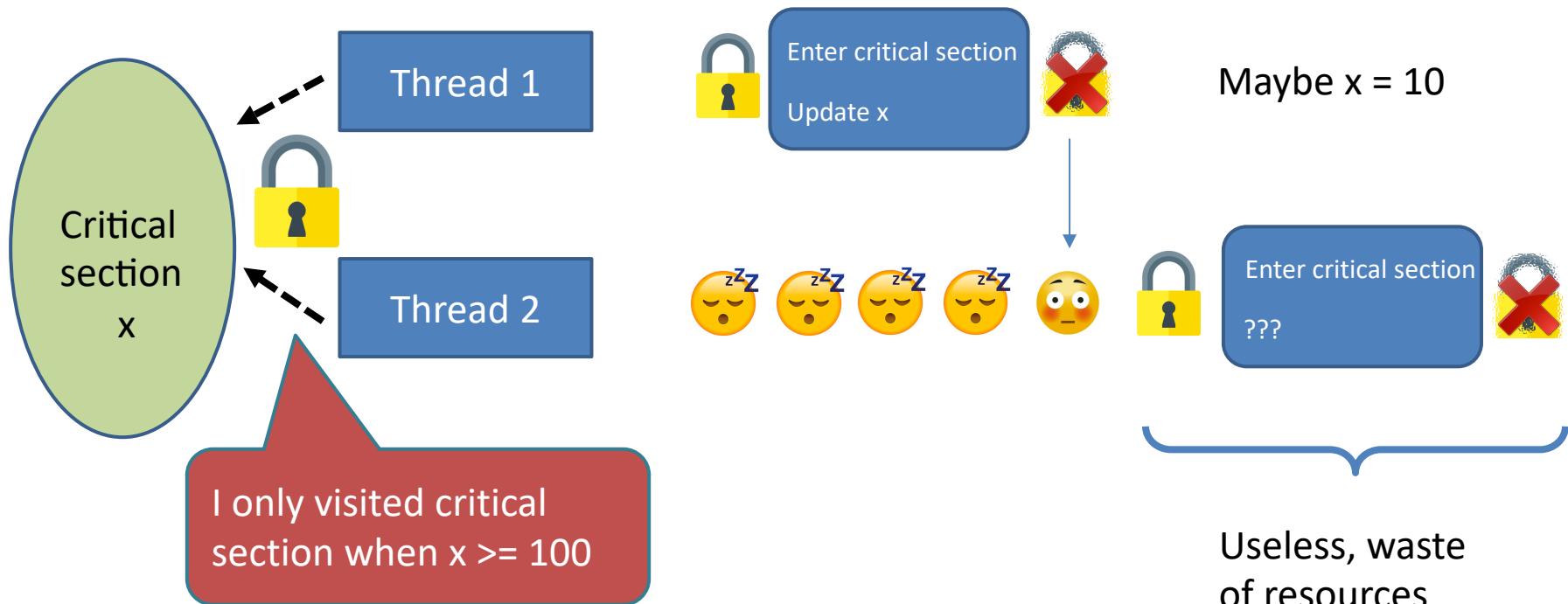
# 3. Mutex lock with conditions

- Can we use busy-waiting lock?



# 3. Mutex lock with conditions

- Can we use sleep-and-wakeup lock?



# Mutex lock with conditions example

- How about competition **with** condition variables?
  - Example: T1: increase x every time;
  - T2: when x is larger than 99, then set x to 0;

```
//thread 1:  
  
while(true)  
{  
  
    iCount++;  
  
}
```

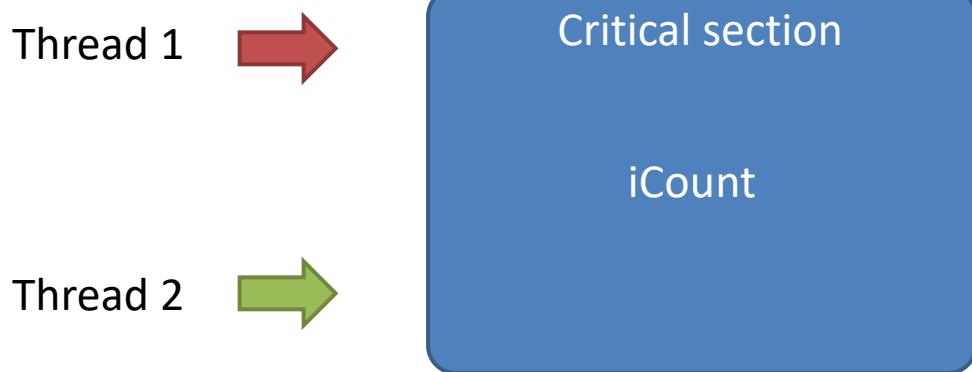
```
//thread 2:  
while(true)  
{  
  
    if(iCount >= 100)  
    {  
        iCount = 0;  
    }  
  
}
```

T1 and T2 compete  
for variable iCount!

# Mutex lock with conditions example

```
//thread 1:  
  
while(true)  
{  
  
    iCount++;  
  
}
```

```
//thread 2:  
while(true)  
{  
  
    if(iCount >= 100)  
    {  
        iCount = 0;  
    }  
  
}
```



# Mutex lock with conditions

- How about competition **with condition variables?**
  - Example: T1: increase x every time;
  - T2: when x is larger than 99, then set x to 0;

```
//thread 1:  
  
while(true)  
{  
    pthread_mutex_lock(&mutex);  
    iCount++;  
    pthread_mutex_unlock(&mutex);  
}
```

```
//thread 2:  
while(true)  
{  
    pthread_mutex_lock(&mutex);  
    if(iCount >= 100)  
    {  
        iCount = 0;  
    }  
    pthread_mutex_unlock(&mutex);  
}
```

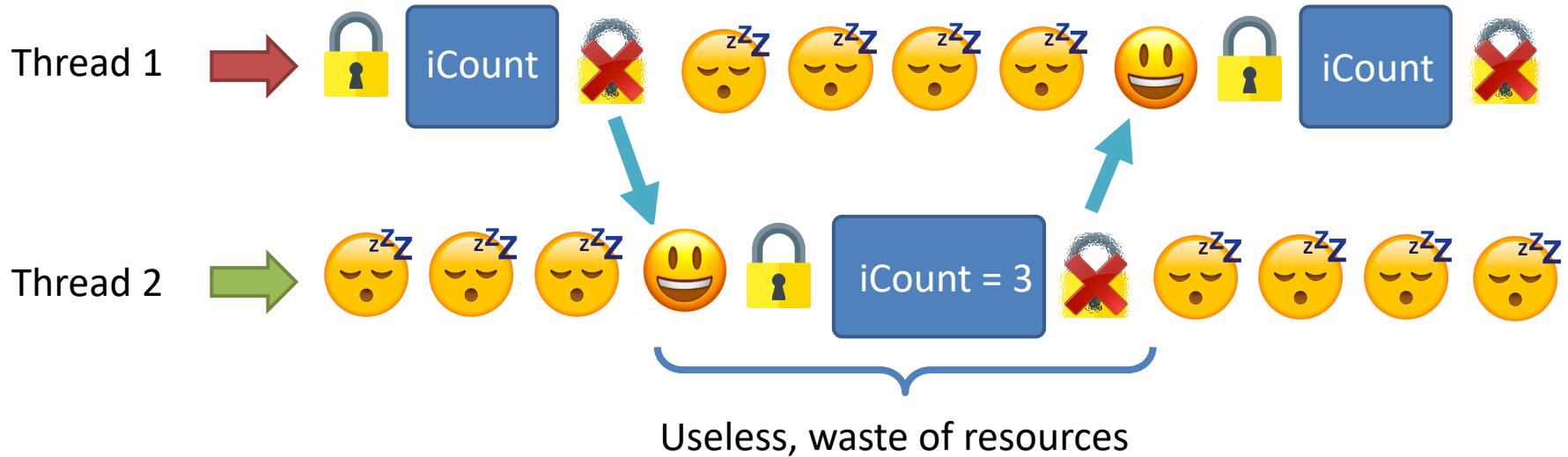
T2 needs to:  
lock;  
determine;  
unlock;  
every time to check

# Mutex lock with conditions

```
//thread 1:  
  
while(true)  
{  
    pthread_mutex_lock(&mutex);  
    iCount++;  
    pthread_mutex_unlock(&mutex);  
}
```

```
//thread 2:  
while(true)  
{  
    pthread_mutex_lock(&mutex);  
    if(iCount >= 100)  
    {  
        iCount = 0;  
    }  
    pthread_mutex_unlock(&mutex);  
}
```

Critical section



# Pthread\_cond\_signal and Pthread\_cond\_wait

Release the lock  
Sleep here until condition is reached

- Pthread\_cond\_wait(&condition, &lock)



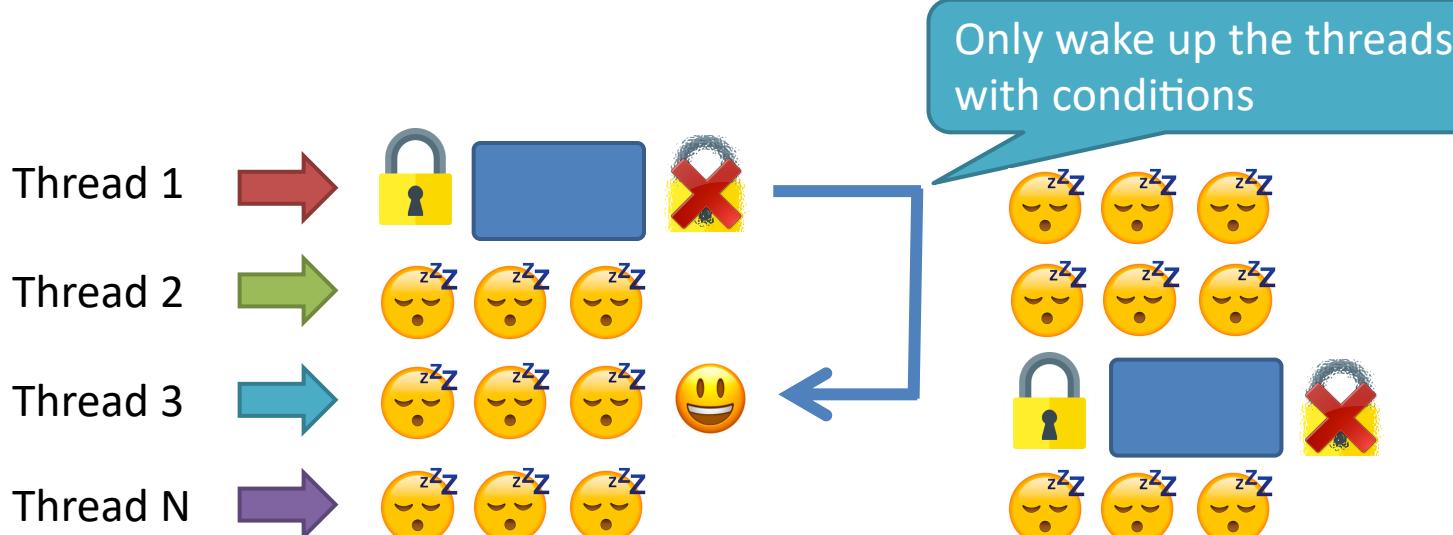
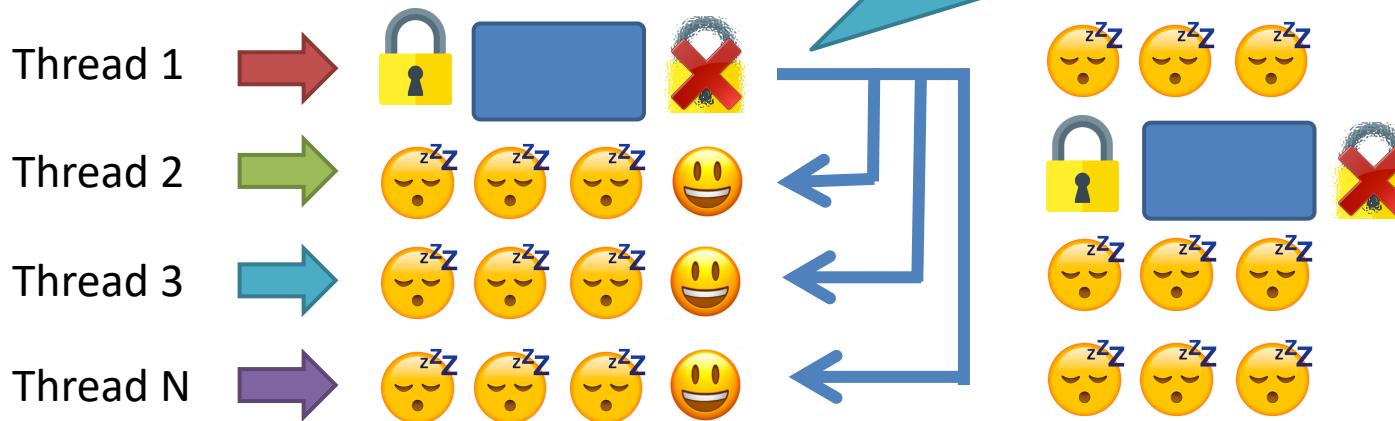
- Pthread\_cond\_signal(&condition)



When condition is reached,  
notify all threads waiting for it

# Pthread\_cond\_signal/Pthread\_cond\_wait v.s. Pthread\_mutex\_lock

All threads sleeping will be  
wakened up when lock is released



# Condition variable

- How about competition **with condition variables?**
  - Example: T1: increase x every time;
  - T2: when x is larger than 99, then set x to 0;

```
//thread1 :  
while(true)  
{  
    pthread_mutex_lock(&mutex);  
    iCount++;  
    pthread_mutex_unlock(&mutex);  
  
    pthread_mutex_lock(&mutex);  
    if(iCount >= 100)  
    {  
        pthread_cond_signal(&cond);  
    }  
    pthread_mutex_unlock(&mutex);  
}
```

```
//thread2:  
while(1)  
{  
    pthread_mutex_lock(&mutex);  
    while(iCount < 100)  
    {  
        pthread_cond_wait(&cond, &mutex);  
    }  
    printf("iCount >= 100\r\n");  
    iCount = 0;  
    pthread_mutex_unlock(&mutex);  
}
```

When T2 executes here:

- 1 : release mutex
- 2 : blocked here
- 3 : when waked, get mutex and execute

```
//thread1 :
while(true)
{
    pthread_mutex_lock(&mutex);
    iCount++;
    pthread_mutex_unlock(&mutex);

    pthread_mutex_lock(&mutex);
    if(iCount >= 100)
    {
        pthread_cond_signal(&cond);
    }
    pthread_mutex_unlock(&mutex);
}
```

1. Get the 
2. release the 
3. Get the 
4. release the 

```
//thread2:
while(1)
{
    pthread_mutex_lock(&mutex);
    while(iCount < 100)
    {
        pthread_cond_wait(&cond, &mutex);
    }
    printf("iCount >= 100\r\n");
    iCount = 0;
    pthread_mutex_unlock(&mutex);
}
```

1. Get the 
2. release the 
3. Get the 
4. release the 

### 3. Wake up

```
//thread 1:
while(true)
{
    pthread_mutex_lock(&mutex);
    iCount++;
    pthread_mutex_unlock(&mutex);
}
```

1. 
2. Release the 

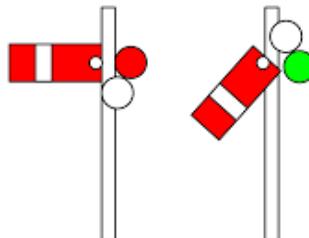


T2 needs to:  
lock;  
determine;  
unlock;  
every time to check

```
//thread 2:
while(true)
{
    pthread_mutex_lock(&mutex);
    if(iCount >= 100)
    {
        iCount = 0;
    }
    pthread_mutex_unlock(&mutex);
}
```

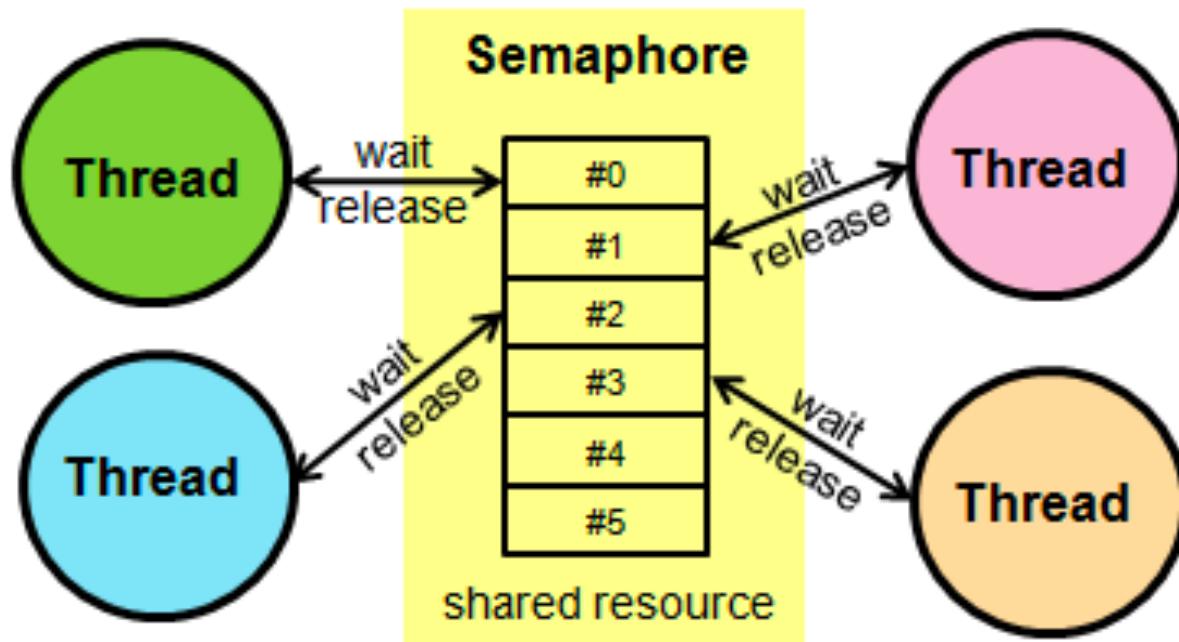
# 4. Semaphore

a system of sending messages by holding the arms or two flags in certain positions



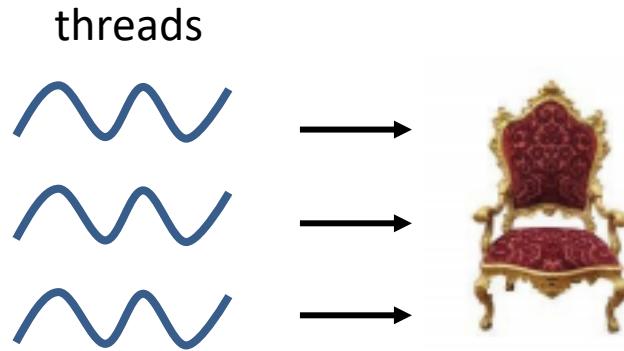
# 4. Semaphore

- Semaphore is a variable used to control access to shared resources by multiple processes/threads



# Mutex lock and Semaphore

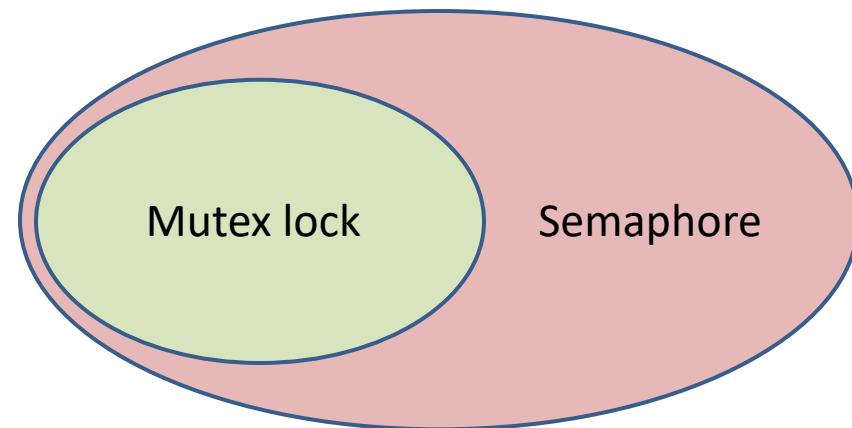
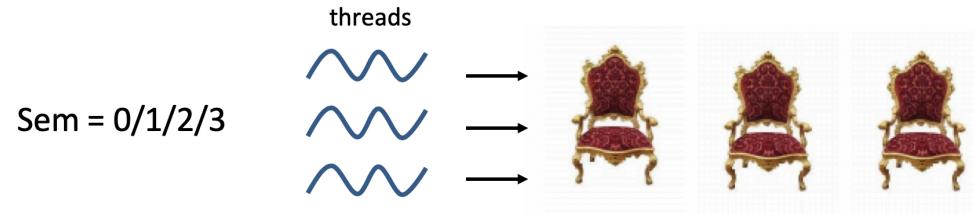
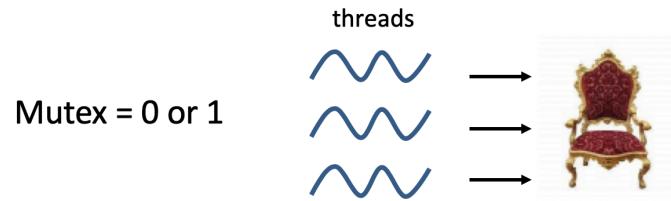
$\text{Mutex} = 0 \text{ or } 1$



$\text{Sem} = 0/1/2/3$



# Mutex lock and Semaphore

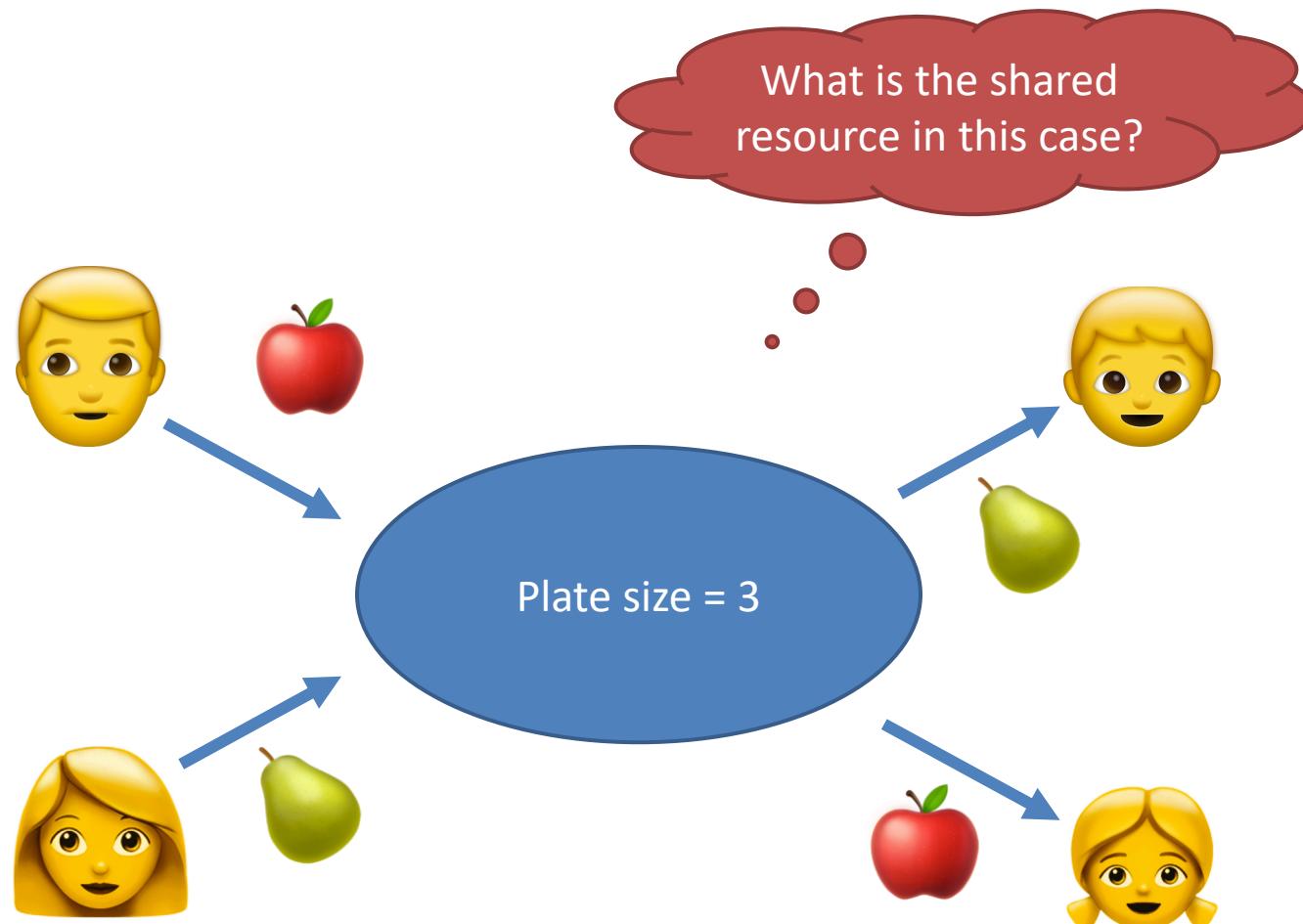


# Semaphore

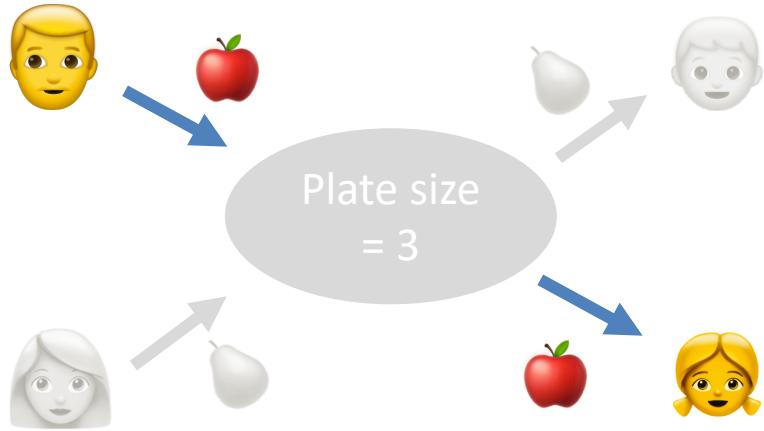
- A semaphore “sem” is a special integer on which only two operations can be performed.
  - DOWN(sem)
  - UP(sem)
- Down operation (P; request):
  - Checks if a semaphore is  $> 0$ ,  $\text{sem--}$ 
    - ▶ Request one-unit resource and one process enters
  - if a semaphore  $\leq 0$ , wait and sleep
- Up operation (V; release)
  - $\text{sem}++$ 
    - ▶ Release one-unit resource and one process leaves



# Semaphore example



# Semaphore example



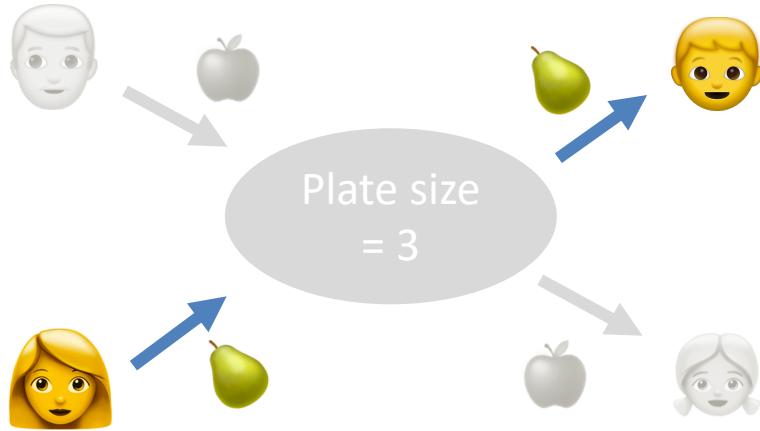
- Semaphore of apple ( $s_2$ ):

- Daughter: request apple
  - Father: release the apple

Father thread:  
peel apple  
put apple  
 $V(s_2)$

Daughter thread:  
 $P(s_2)$   
get apple  
eat apple

# Semaphore example



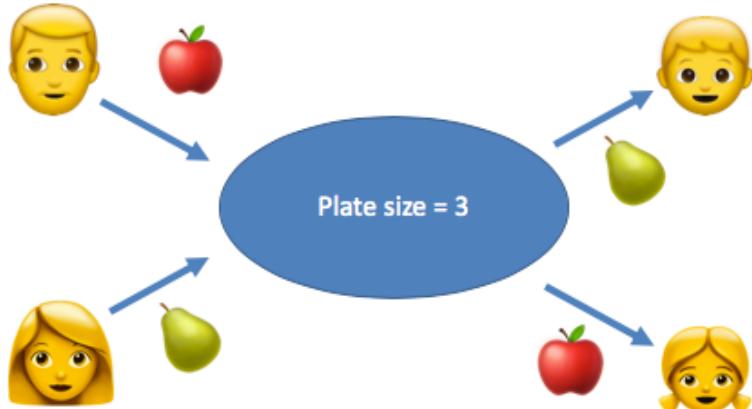
- Semaphore of pear ( $s1$ ):

- Son: request pear
  - Mather: release the pear

Mother thread:  
peel pear  
put apple  
 $V(s1)$

Son thread:  
 $P(s1)$   
get pear  
eat pear

# Semaphore example



- Semaphore of plate ( $s_3$ ):
  - Son/Daughter: release the space
  - Father/Mother: request the space

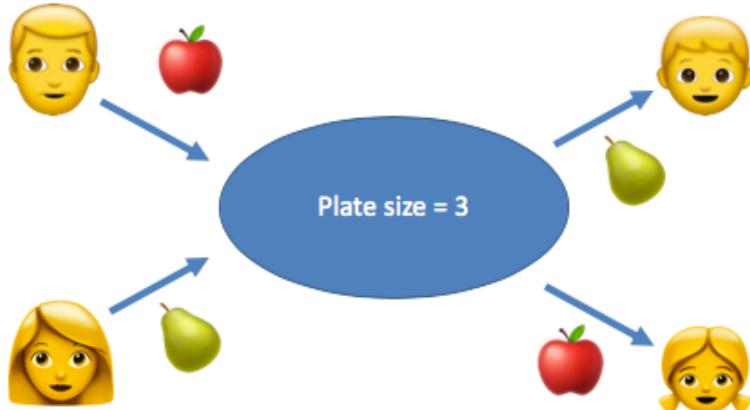
Father thread:  
peel apple  
 $P(s_3)$   
put apple

Mother thread:  
peel pear  
 $P(s_3)$   
put apple

Son thread:  
get pear  
 $V(s_3)$   
eat pear

Daughter thread:  
get apple  
 $V(s_3)$   
eat apple

# Semaphore example



- **Semaphore:**

- Son: whether there is pear,  $s_1$
- Daughter: whether there is apple,  $s_2$
- Father/Mother: whether there is space,  $s_3$

Father thread:  
peel apple  
 $P(s_3)$   
put apple  
 $V(s_2)$

Mother thread:  
peel pear  
 $P(s_3)$   
put apple  
 $V(s_1)$

Son thread:  
 $P(s_1)$   
get pear  
 $V(s_3)$   
eat pear

Daughter thread:  
 $P(s_2)$   
get apple  
 $V(s_3)$   
eat apple

# Semaphore example

- Semaphore:
  - Son: whether there is pear, **pear**
  - Daughter: whether there is apple, **apple**
  - Father/Mother: whether there is space, **remain**

Father thread:

    peel apple  
    P(remain)  
    put apple  
    V(apple)

Daughter thread:

    P(**apple**)  
    get apple  
    V(remain)  
    eat apple

```
void *father(void *arg) {
    while(1) {
        sleep(5); //simulate peel apple
        P(s3) [sem_wait(&remain);
        sem_wait(&mutex);
        nremain--;
        napple++;
        sem_post(&mutex);
        V(s2) [sem_post(&apple);
    }
}
```

```
void *daughter(void *arg) {
    while(1) {
        P(s2) [sem_wait(&apple);
        sem_wait(&mutex);
        nremain++;
        napple--;
        sem_post(&mutex);
        V(s3) [sem_post(&remain);
        sleep(10); //simulate eat apple
    }
}
```

[https://github.com/kevinsuo/CS3502/  
blob/master/semaphore.c](https://github.com/kevinsuo/CS3502/blob/master/semaphore.c)

# Semaphore example

```
pi@raspberrypi ~/Downloads> ./semaphore.o
father 🧑 before put apple, remain=3, apple🍎=0, pear🍐=0
father 🧑 after put apple, remain=2, apple🍎=1, pear🍐=0

daughter👩 before eat apple, remain=2, apple🍎=1, pear🍐=0
daughter👩 after eat apple, remain=3, apple🍎=0, pear🍐=0

mother 🧑 before put pear , remain=3, apple🍎=0, pear🍐=0
mother 🧑 after put pear , remain=2, apple🍎=0, pear🍐=1

son 🧑 before eat pear , remain=2, apple🍎=0, pear🍐=1
son 🧑 after eat pear , remain=3, apple🍎=0, pear🍐=0

father 🧑 before put apple, remain=3, apple🍎=0, pear🍐=0
father 🧑 after put apple, remain=2, apple🍎=1, pear🍐=0

mother 🧑 before put pear , remain=2, apple🍎=1, pear🍐=0
mother 🧑 after put pear , remain=1, apple🍎=1, pear🍐=1

daughter👩 before eat apple, remain=1, apple🍎=1, pear🍐=1
daughter👩 after eat apple, remain=2, apple🍎=0, pear🍐=1

father 🧑 before put apple, remain=2, apple🍎=0, pear🍐=1
father 🧑 after put apple, remain=1, apple🍎=1, pear🍐=1

son 🧑 before eat pear , remain=1, apple🍎=1, pear🍐=1
son 🧑 after eat pear , remain=2, apple🍎=1, pear🍐=0

mother 🧑 before put pear , remain=2, apple🍎=1, pear🍐=0
mother 🧑 after put pear , remain=1, apple🍎=1, pear🍐=1

father 🧑 before put apple, remain=1, apple🍎=1, pear🍐=1
father 🧑 after put apple, remain=0, apple🍎=2, pear🍐=1

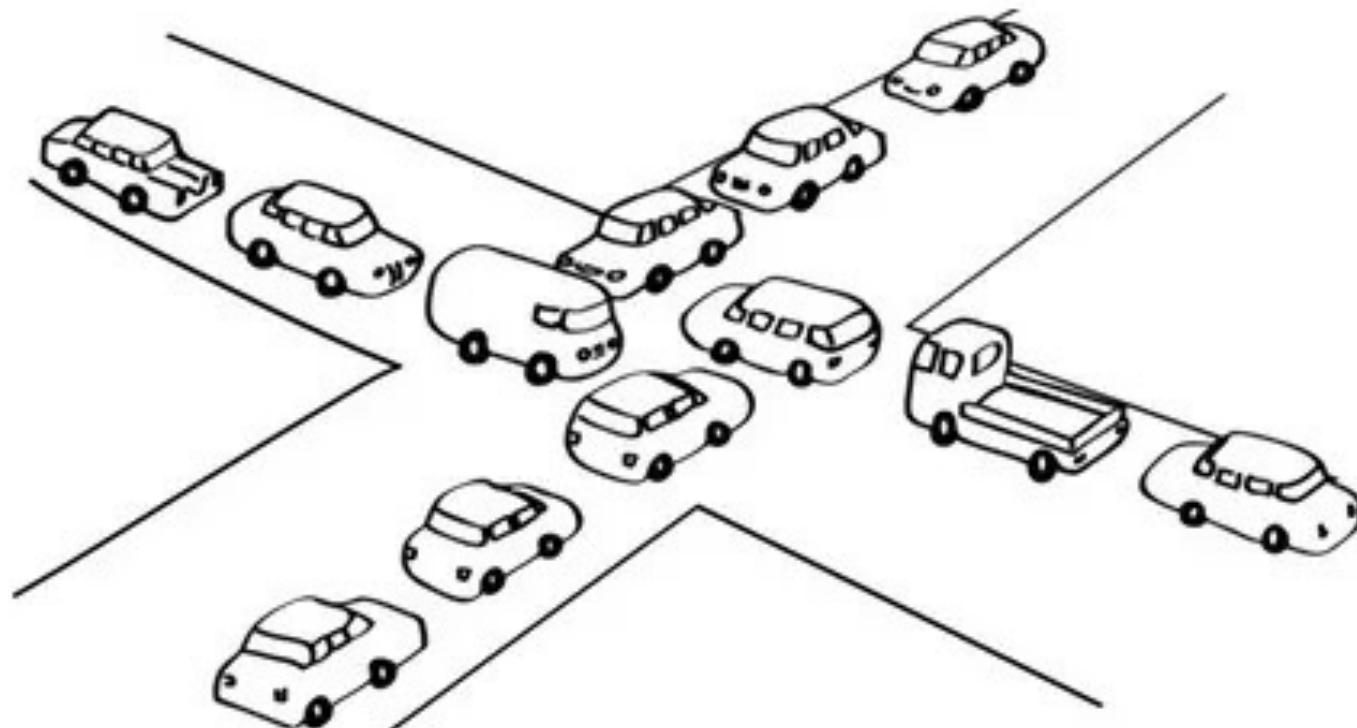
daughter👩 before eat apple, remain=0, apple🍎=2, pear🍐=1
daughter👩 after eat apple, remain=1, apple🍎=1, pear🍐=1
```

gcc -pthread semaphore.c  
-o semaphore.o

[https://youtu.be/ZIW  
wvcuROME](https://youtu.be/ZIWwvcuROME)

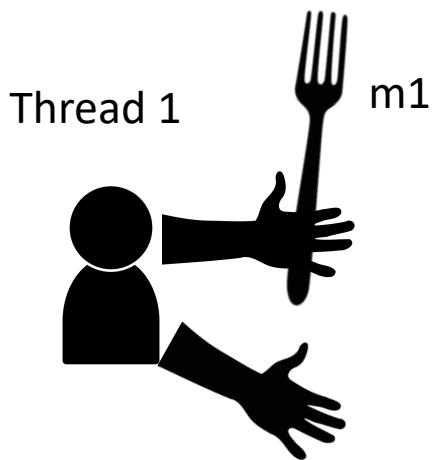
# Deadlocks

---



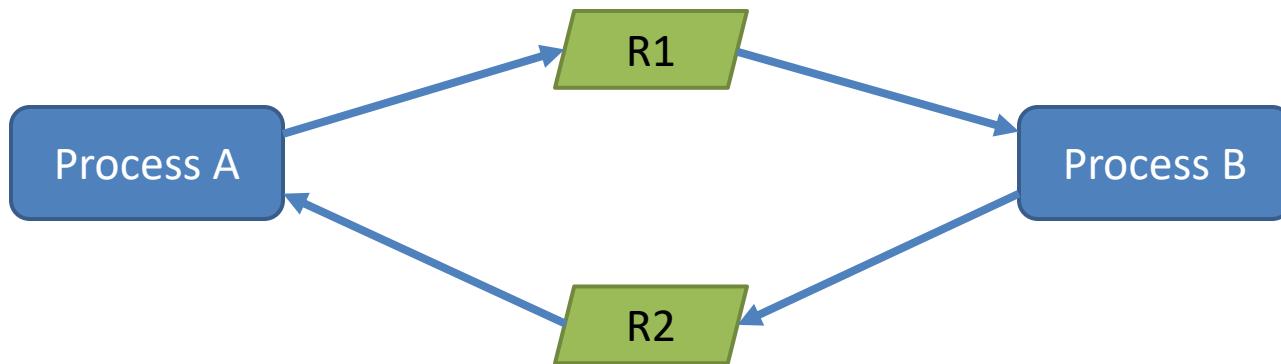
# Deadlocks: philosopher dining

- Six people
- Three folks
- Three knives

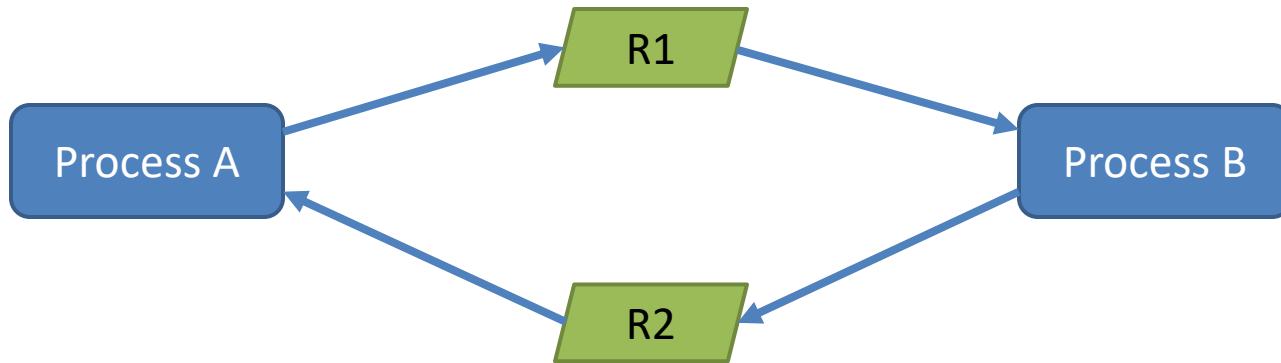


# Deadlocks

- When two or more threads stop making progress indefinitely because they are **all waiting for each other** to do something.
  - If process A waits for process B to release a resource, and
  - Process B is waiting for process A to release another resource at the same time.
  - In this case, neither A nor B can proceed because both are waiting for the other to proceed.



# Deadlock example



Thread 1

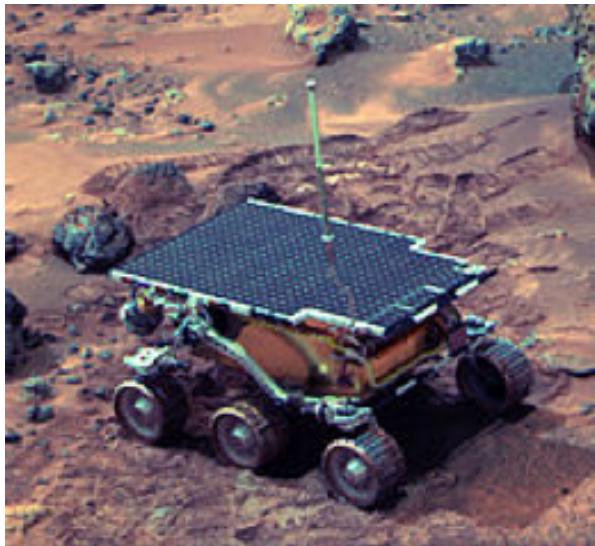
```
pthread_mutex_lock(&R1);
/* use resource 1 */
pthread_mutex_lock(&R2);
/* use resources 1 and 2 */
do_something();
pthread_mutex_unlock(&R2);
pthread_mutex_unlock(&R1);
```



Thread 2

```
pthread_mutex_lock(&R2);
/* use resource 2 */
pthread_mutex_lock(&R1);
/* use resources 1 and 2 */
do_something();
pthread_mutex_unlock(&R1);
pthread_mutex_unlock(&R2);
```

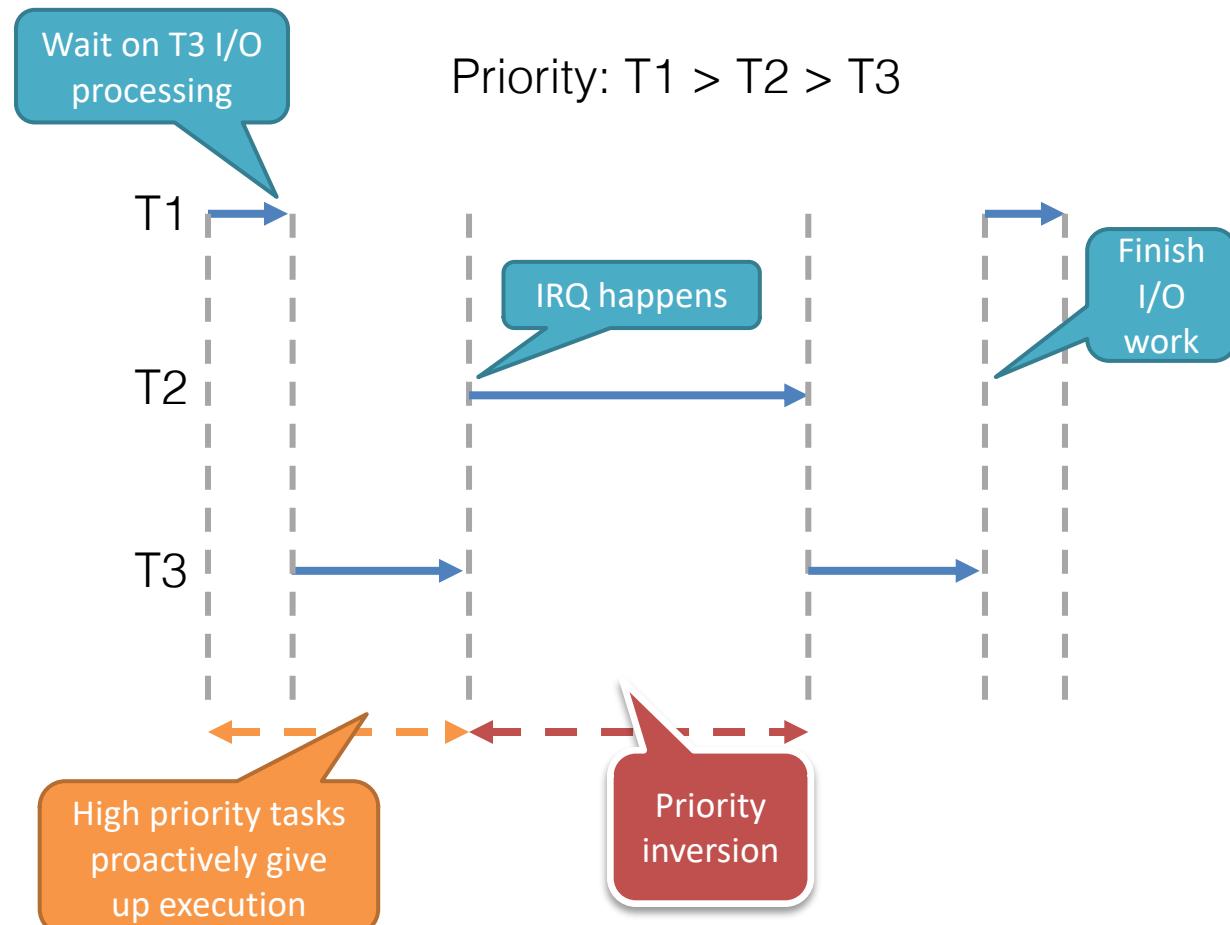
# Deadlock example: Priority Inversion



1997/07/04 Pathfinder  
—> Mars



<https://www.youtube.com/watch?v=lyx7kARrGeM>  
<https://www.youtube.com/watch?v=t9RM5xcNUak>  
<https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft>



# Conclusion

---

- Concurrency and synchronization
  - Execution models
  - Race condition
  - Critical section
- Mutual exclusion
  - Spinlock
  - Mutex lock
  - Semaphore
  - Deadlock and priority inversion

