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

https://github.com/kevinsuo/CS3502/blob/master/threads.c

```
#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
```

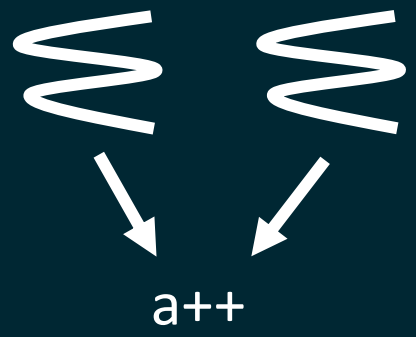
Counter value: before

Two threads increase a counter

Counter value: after

Concurrency Example

```
⌥⌘1  ./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   : 13787
pi@raspberrypi ~> ./threads.o 100000
Initial value : 0
Final value   : 121949
pi@raspberrypi ~> ./threads.o 1000000
Initial value : 0
Final value   : 1151319
pi@raspberrypi ~>
```



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.

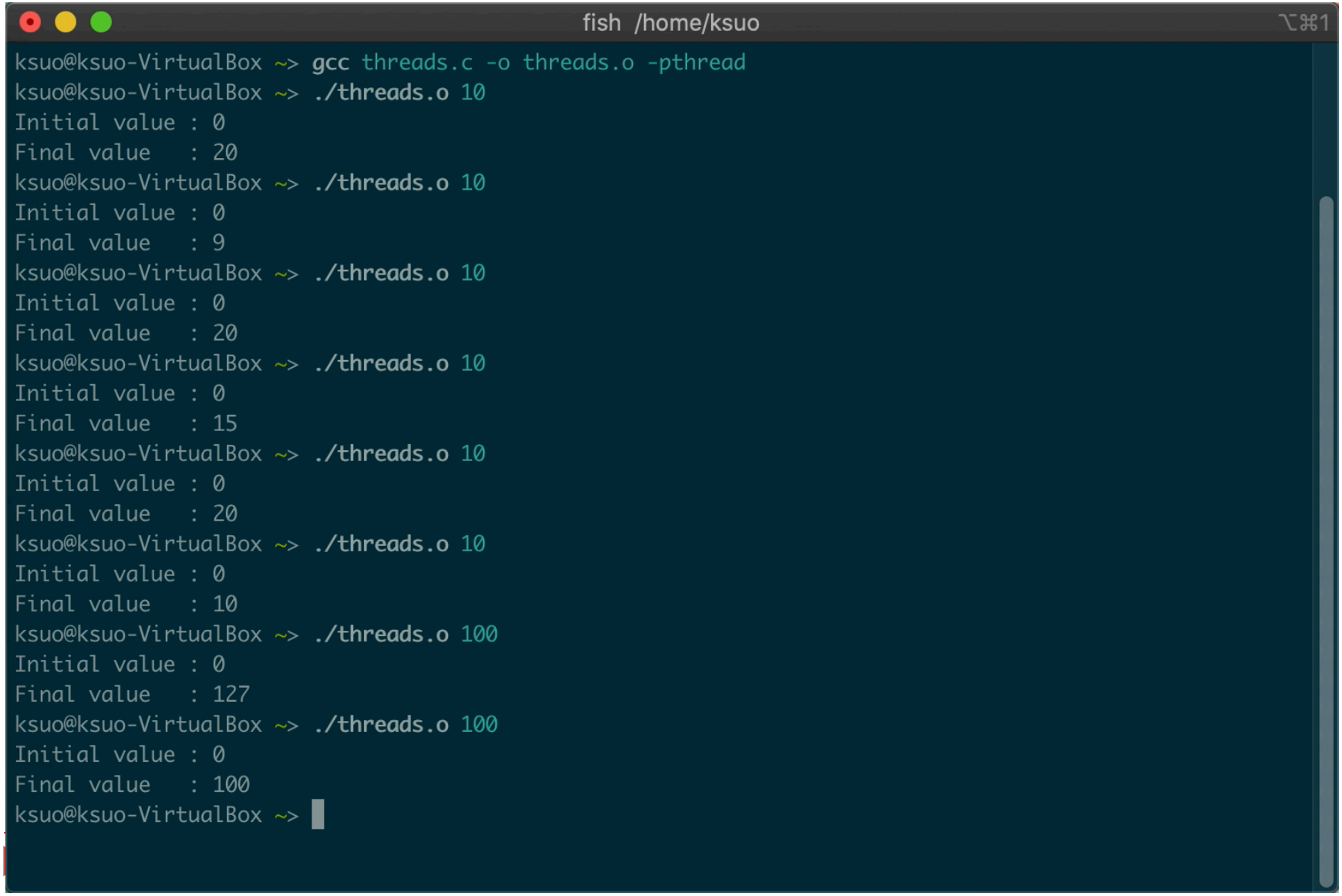
Concurrency Example

```c
#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;
}
```

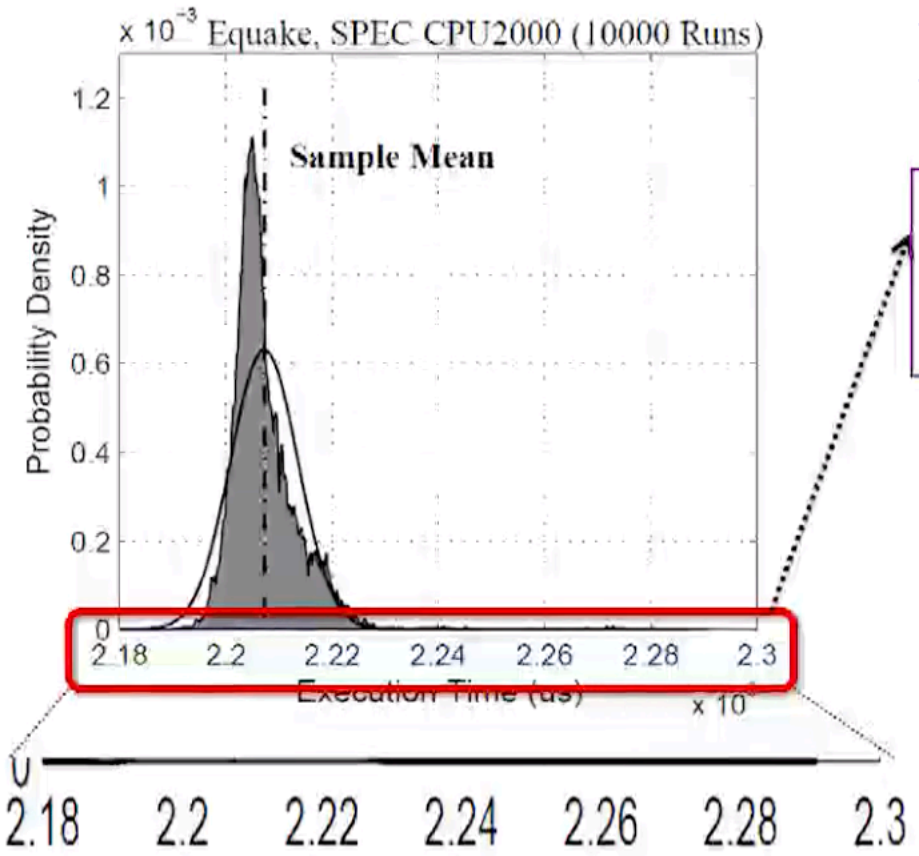
For each iteration of i, it takes more time each round

Concurrency Example

Concurrency Example



Performance variation: run a program on a single server



https://parsec.cs.princeton.edu/publications/chen14ieeetc.pdf

Parameters: avg = 2.16, max=2.3, min=2.18

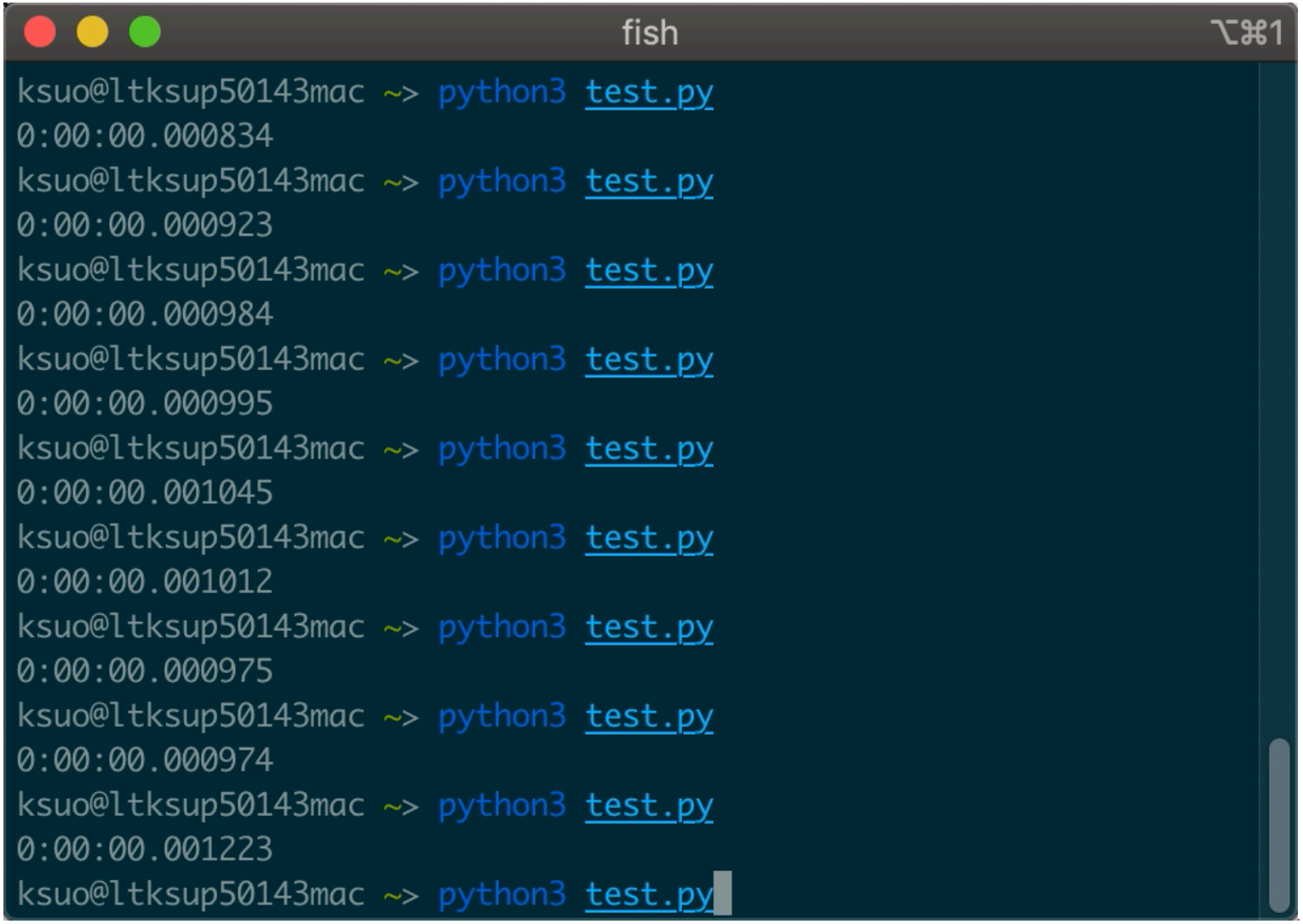
Variations (%): $\frac{2.3-2.18}{2.16} * 100\% = 0.9\%$

Performance variation on a single server: 1%

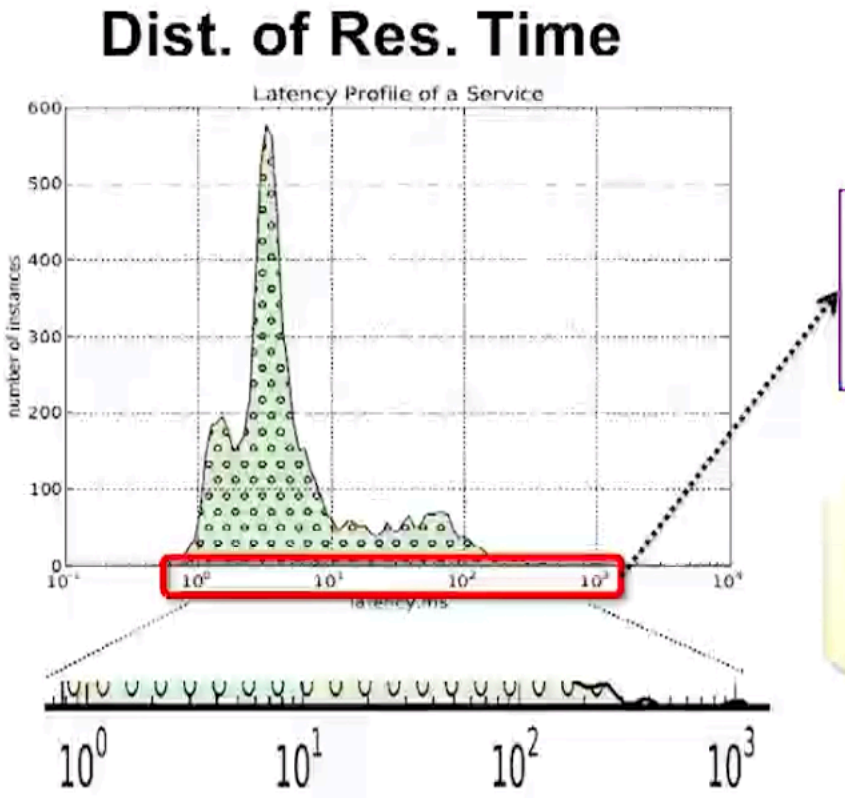
T. Chen et al., Statistical Performance Comparisons of Computers, HPCA 2012.

Statistical Performance Comparisons of Computers

Performance variation: run a program on a single server



Performance variation: run a service on Google datacenter for 10k times



http://www2013.w3c.br/proceedings/p703.pdf

Parameters: avg ≈ 5, max > 300, min < 1

Variations (%): $\frac{300-1}{5} * 100\% = 600\%$

Performance variation in a datacenter: ~10x

D. Krushevskaja and M. Sandler, Understanding Latency Variations of Black Box Services, WWW 2013.

Lambda Management Console

https://us-east-2.console.aws.amazon.com/lambda/home?region=us-east-2#/functions/test?newFu...

aws Services Resource Groups tonys Ohio Support

test

Throttle | Qualifiers | Actions | test | Test | Save

The section below shows the result returned by your function execution.

```
{
  "statusCode": 200,
  "number": 92
}
```

Summary

Code SHA-256
sGK910fBuTtqRilDvyMmBh5WRbACHBR3lwm4fY/EVEQ=

Request ID
85bd5671-6a88-4efb-ba3a-4d677a1007df

Duration
0.38 ms

Billed duration
100 ms

Resources configured
128 MB

Max memory used
23 MB

Log output

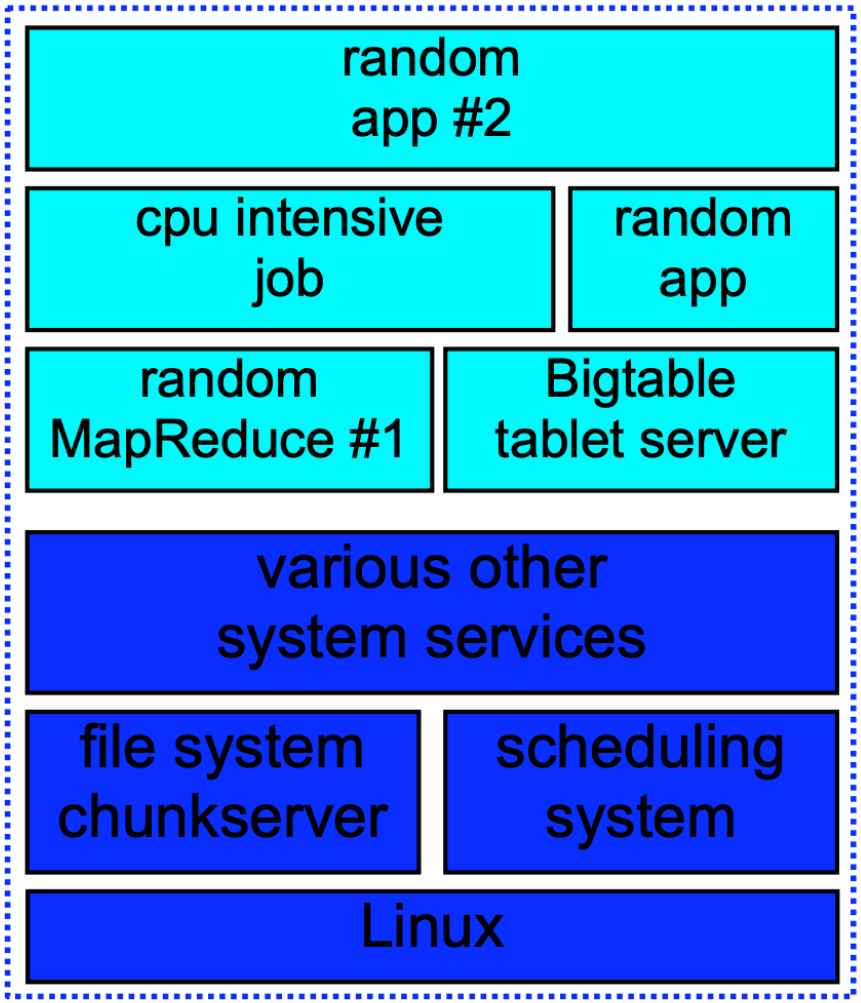
The section below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.

```
START RequestId: 85bd5671-6a88-4efb-ba3a-4d677a1007df Version: $LATEST
END RequestId: 85bd5671-6a88-4efb-ba3a-4d677a1007df
REPORT RequestId: 85bd5671-6a88-4efb-ba3a-4d677a1007df  Duration: 0.38 ms       Billed
Duration: 100 ms        Memory Size: 128 MB     Max Memory Used: 23 MB
```

00:01 Finish

Feedback English (US)

A typical Google server



Achieving Rapid Response Times in Large Online Services

Sharing!

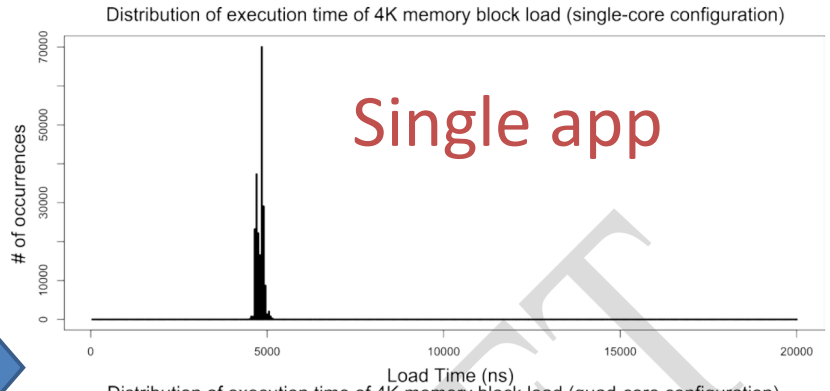
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44875.pdf

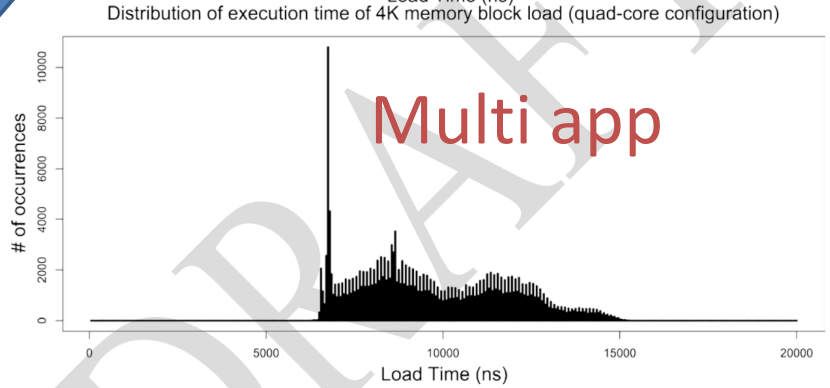
J. Dean, Achieving Rapid Response Times in Large Online Services, talk at Berkeley, 2012.

Unmanaged sharing

Typical unmanaged sharing

Read 4KB data from DRAM

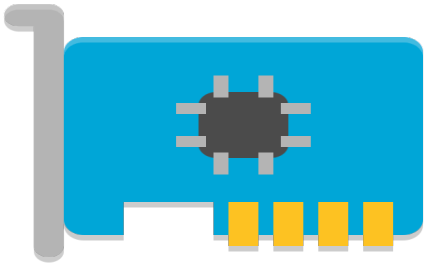




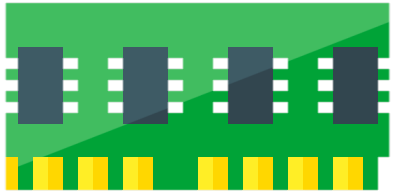
https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/SDS_DO005_White_Paper.pdf

FAA. White Paper on Issues Associated with Interference Applied to Multicore Processors. 2016

Sharing is everywhere

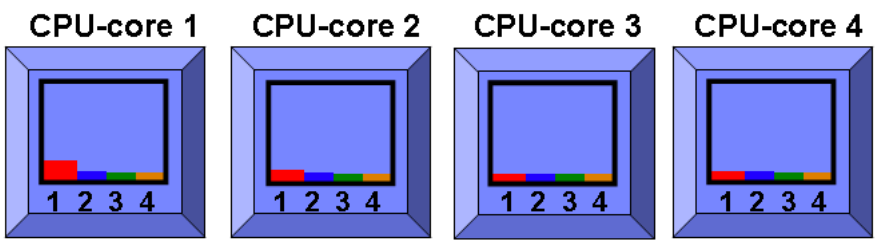


Network



DRAM

Cache



CPU

Sharing is everywhere: web search

websearch

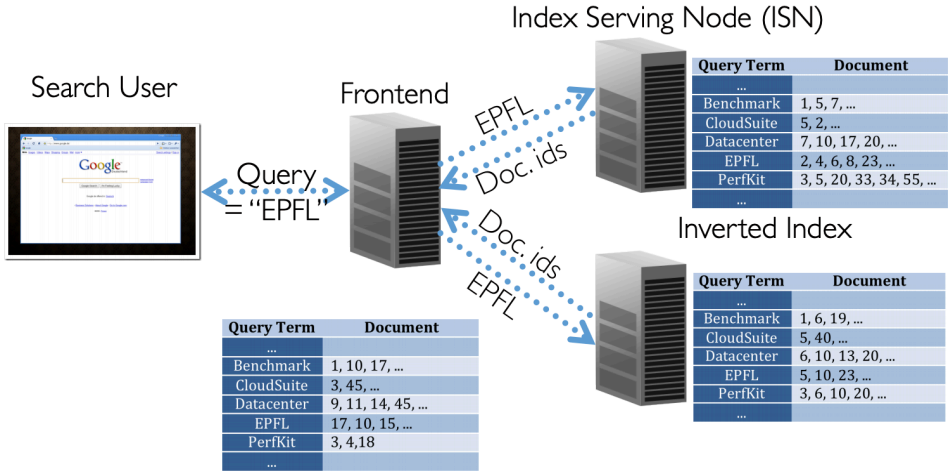
	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
LLC (small)	134%	103%	96%	96%	109%	102%	100%	96%	96%	104%	99%	100%	101%	100%	104%	103%	104%	103%	99%
LLC (med)	152%	106%	99%	99%	116%	111%	109%	103%	105%	116%	109%	108%	107%	110%	123%	125%	114%	111%	101%
LLC (big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	264%	222%	123%	102%
DRAM	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	270%	228%	122%	103%
HyperThread	81%	109%	106%	106%	104%	113%	106%	114%	113%	105%	114%	117%	118%	119%	122%	136%	>300%	>300%	>300%
CPU power	190%	124%	110%	107%	134%	115%	106%	108%	102%	114%	107%	105%	104%	101%	105%	100%	98%	99%	97%
Network	35%	35%	36%	36%	36%	36%	36%	37%	37%	38%	39%	41%	44%	48%	51%	55%	58%	64%	95%
brain	158%	165%	157%	173%	160%	168%	180%	230%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%

Each entry is color-coded as follows: 140% is ≥120%, 110% is between 100% and 120%, and 65% is ≤100%.

Bing

Google

Yandex

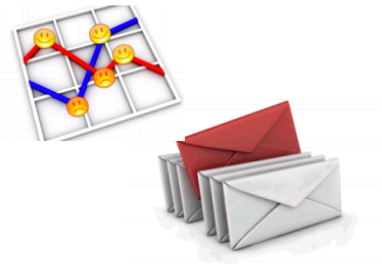


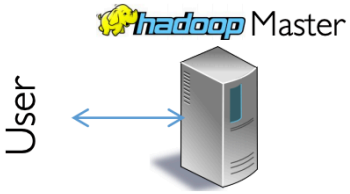
Sharing is everywhere: machine learning

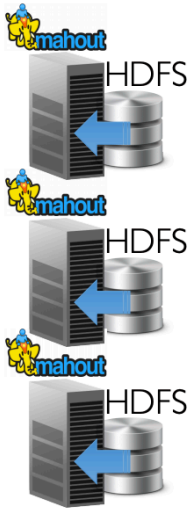
ml_cluster

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
LLC (small)	101%	88%	99%	84%	91%	110%	96%	93%	100%	216%	117%	106%	119%	105%	182%	206%	109%	202%	203%
LLC (med)	98%	88%	102%	91%	112%	115%	105%	104%	111%	>300%	282%	212%	237%	220%	220%	212%	215%	205%	201%
LLC (big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	276%	250%	223%	214%	206%
DRAM	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	287%	230%	223%	211%
HyperThread	113%	109%	110%	111%	104%	100%	97%	107%	111%	112%	114%	114%	114%	119%	121%	130%	259%	262%	262%
CPU power	112%	101%	97%	89%	91%	86%	89%	90%	89%	92%	91%	90%	89%	89%	90%	92%	94%	97%	106%
Network	57%	56%	58%	60%	58%	58%	58%	58%	59%	59%	59%	59%	59%	63%	63%	67%	76%	89%	113%
brain	151%	149%	174%	189%	193%	202%	209%	217%	225%	239%	>300%	>300%	279%	>300%	>300%	>300%	>300%	>300%	>300%

Each entry is color-coded as follows: 140% is ≥120%, 110% is between 100% and 120%, and 65% is ≤100%.





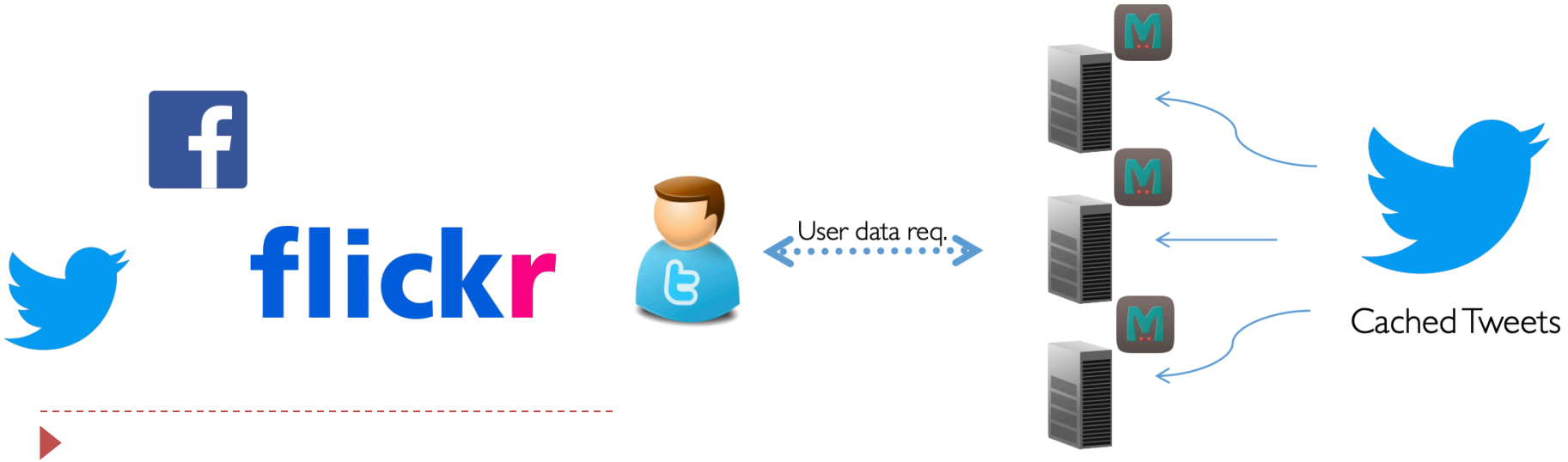


Sharing is everywhere: data caching

memkeyval

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
LLC (small)	115%	88%	88%	91%	99%	101%	79%	91%	97%	101%	135%	138%	148%	140%	134%	150%	114%	78%	70%
LLC (med)	209%	148%	159%	107%	207%	119%	96%	108%	117%	138%	170%	230%	182%	181%	167%	162%	144%	100%	104%
LLC (big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	280%	225%	222%	170%	79%	85%
DRAM	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	252%	234%	199%	103%	100%
HyperThread	26%	31%	32%	32%	32%	32%	33%	35%	39%	43%	48%	51%	56%	62%	81%	119%	116%	153%	>300%
CPU power	192%	277%	237%	294%	>300%	>300%	219%	>300%	292%	224%	>300%	252%	227%	193%	163%	167%	122%	82%	123%
Network	27%	28%	28%	29%	29%	27%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
brain	197%	232%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%

Each entry is color-coded as follows: 140% is ≥120%, 110% is between 100% and 120%, and 65% is ≤100%.



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

- 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

```c
#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

```c
#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

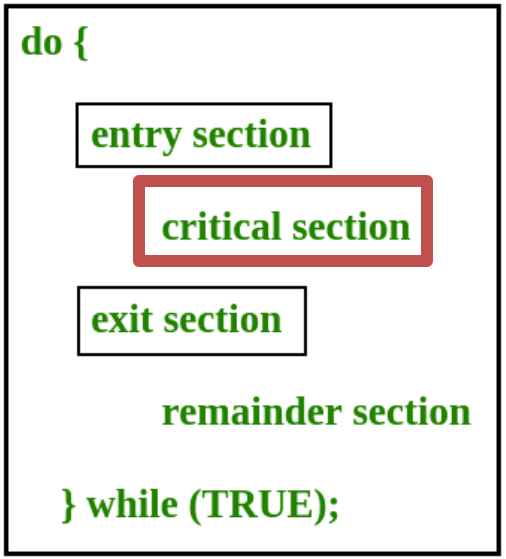
Add more threads

```
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.

Critical section example

```c
#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

```c
#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!

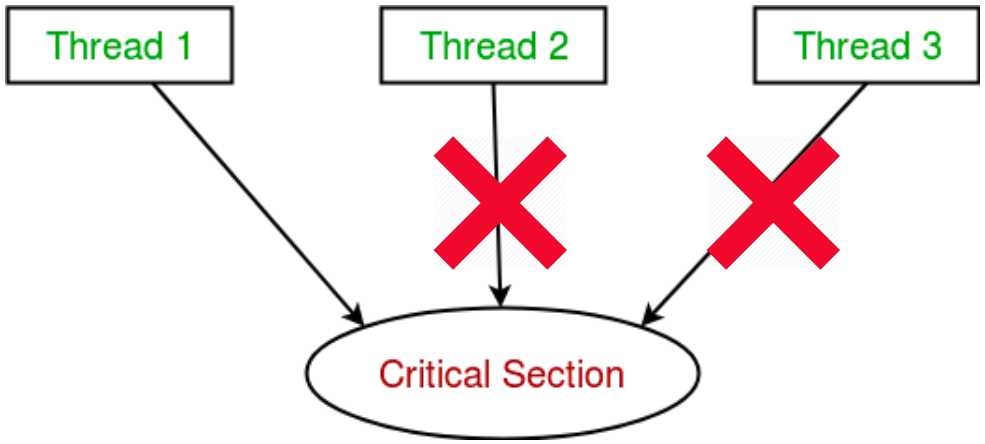
Outline

- 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

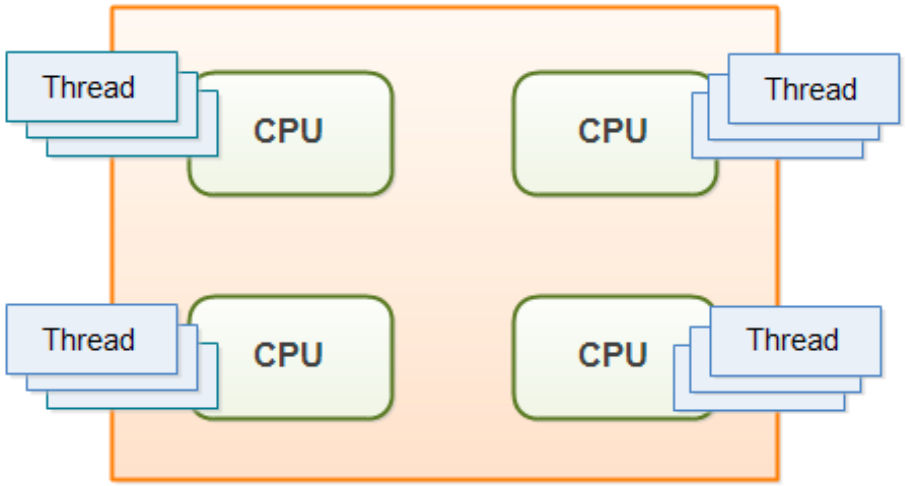


To avoid race condition

- Principles:

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

Thread could have varied speeds

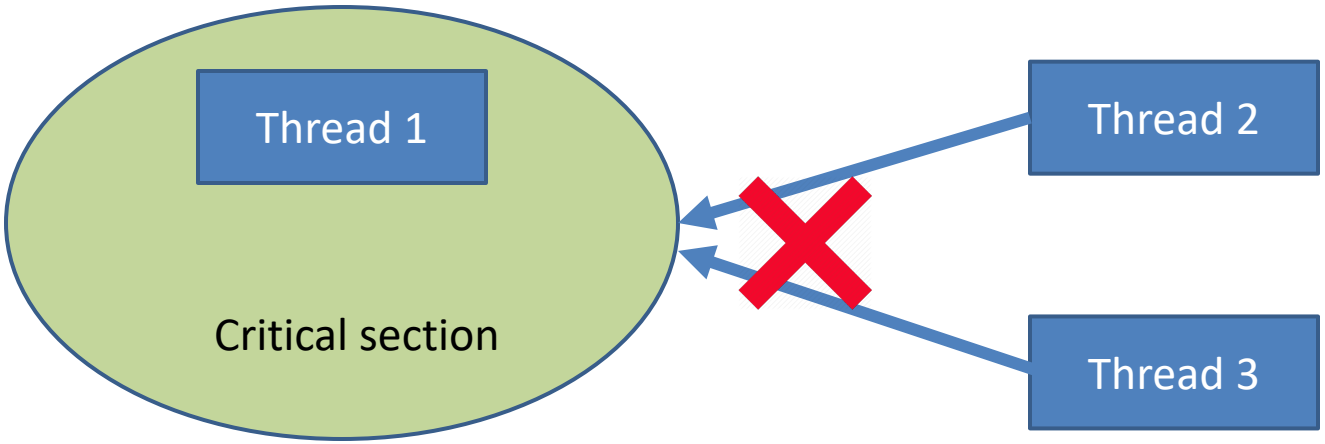


Thread could exist at each core

To avoid race condition

- Principles:

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

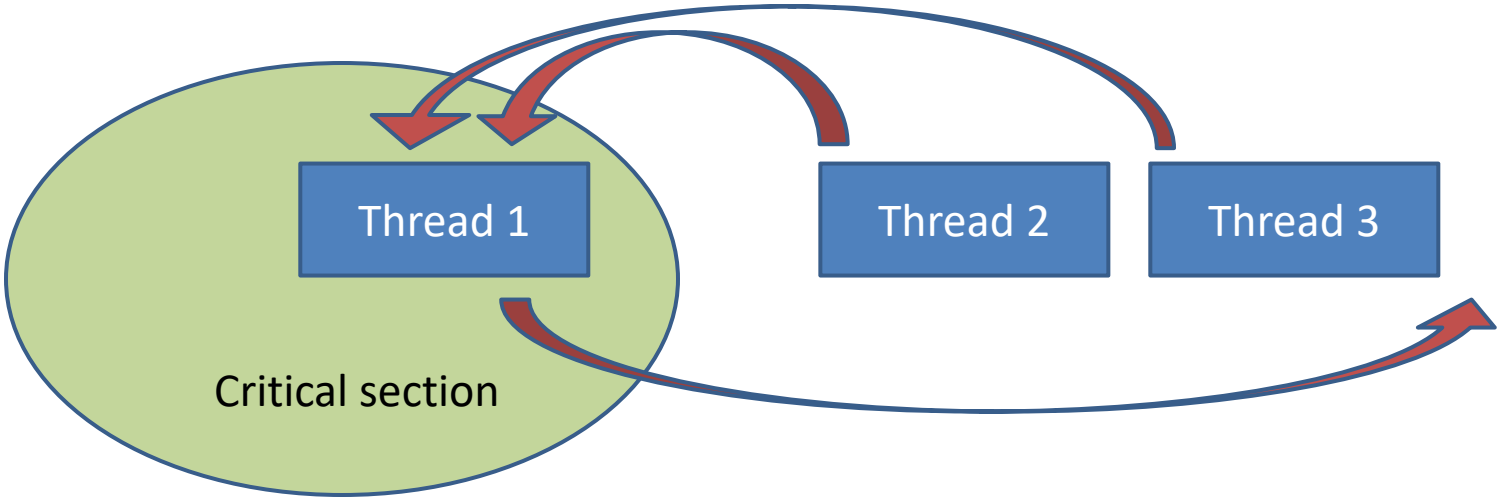


To avoid race condition

- 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

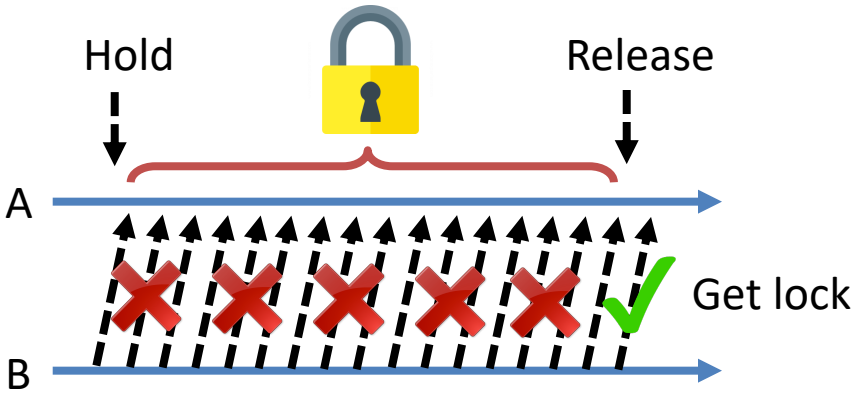
3. No process running outside its critical region may block another process running in the critical region
4. No process must wait forever to enter its critical region
   - Waiting forever indicates a deadlock

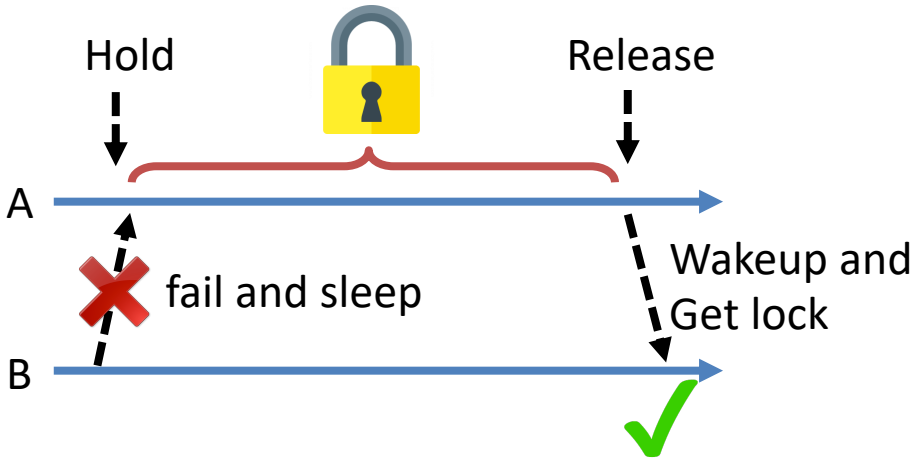
- (1) and (2) are enforced by the operating system's implementation of locks
  - Programmers assume that locks satisfy (1) and (2)
- (3) and (4) must be ensured by the programmer using the locks.
  - OS cannot enforce these.

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





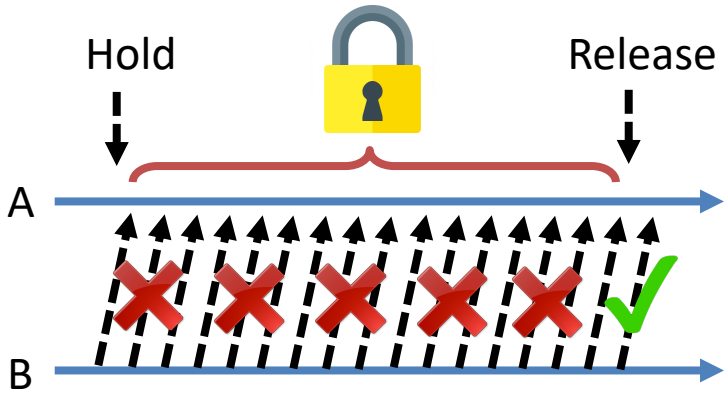
1, Spinlock: A busy-waiting lock implementation

- Don’t block. Instead, constantly poll the lock for availability.
- Usage: small critical region

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

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

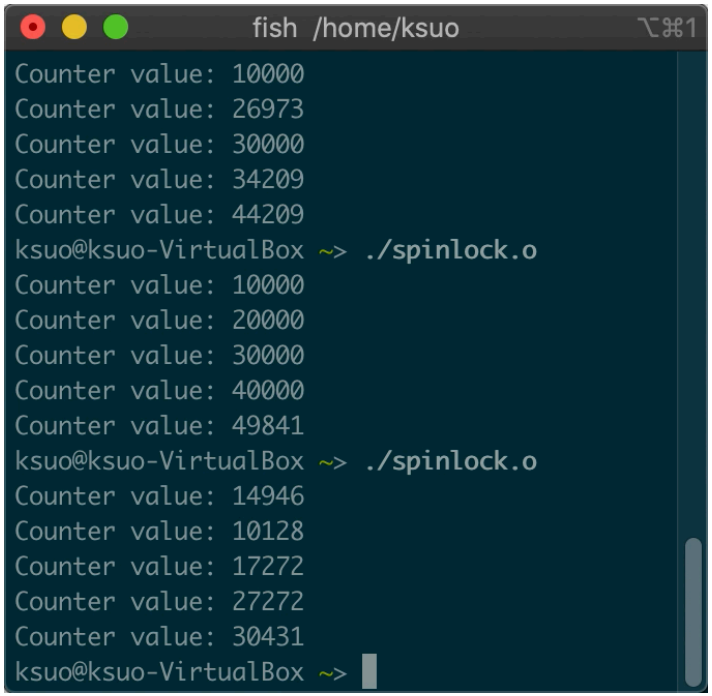
- Advantage
  - Very efficient with short critical sections
    - if you expect a lock to be released quickly
- Disadvantage
  - Doesn’t yield the CPU and burns CPU cycles
    - Bad if critical sections are long.
  - Efficient only if machine has multiple CPUs.
    - Counterproductive on uniprocessor machines



Spinlock example

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

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

int counter = 0;
static pthread_spinlock_t splock;

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);
}
```

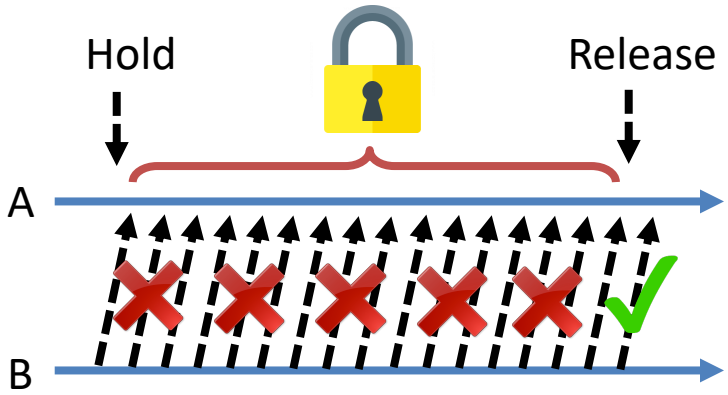


Spinlock example

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

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

int counter = 0;
static pthread_spinlock_t splock;

void *compute()
{
        int i = 0;
        pthread_spin_lock(&splock);
        while (i < 10000) {
                counter = counter + 1;
                i++;
        }
        pthread_spin_unlock(&splock);
        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);
}
```

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

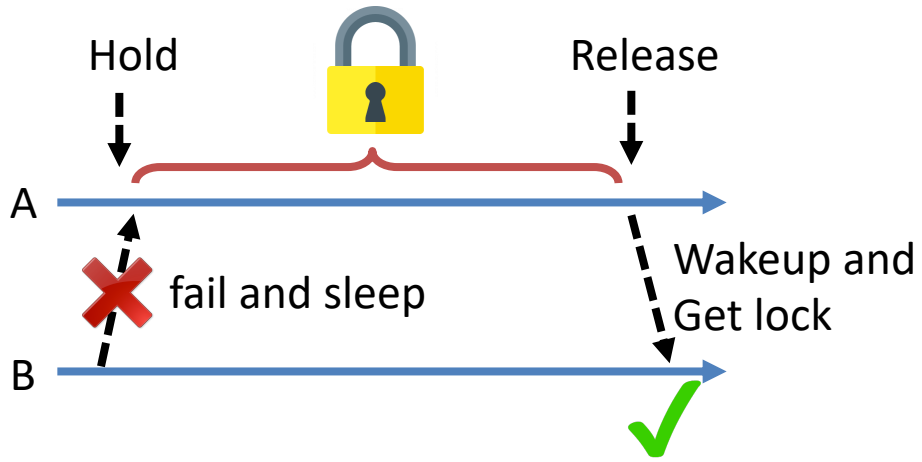


Other mutual exclusion similar as busy waiting (spinlock)

- 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)
Critical Section…
Unlock(mutex)
```

Pro:

Better cpu utilization

Con:

Overhead on entering sleep or wake up

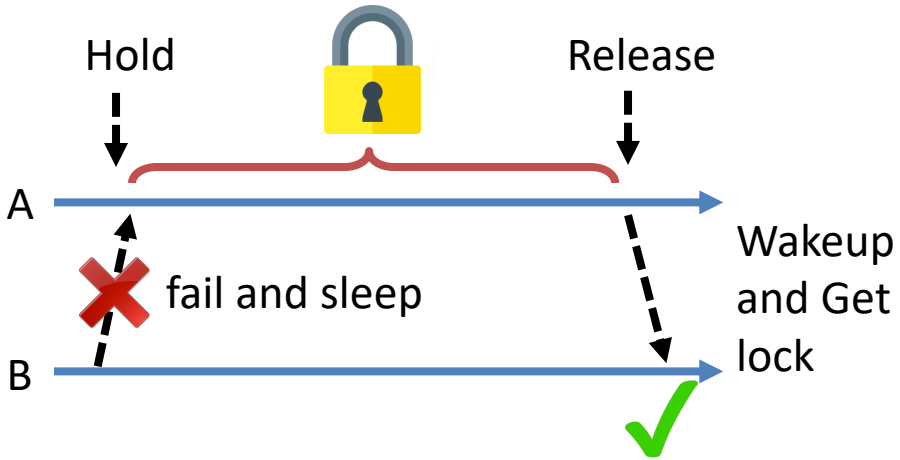
Not suited for short duration of lock acquisition

Mutex lock example

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

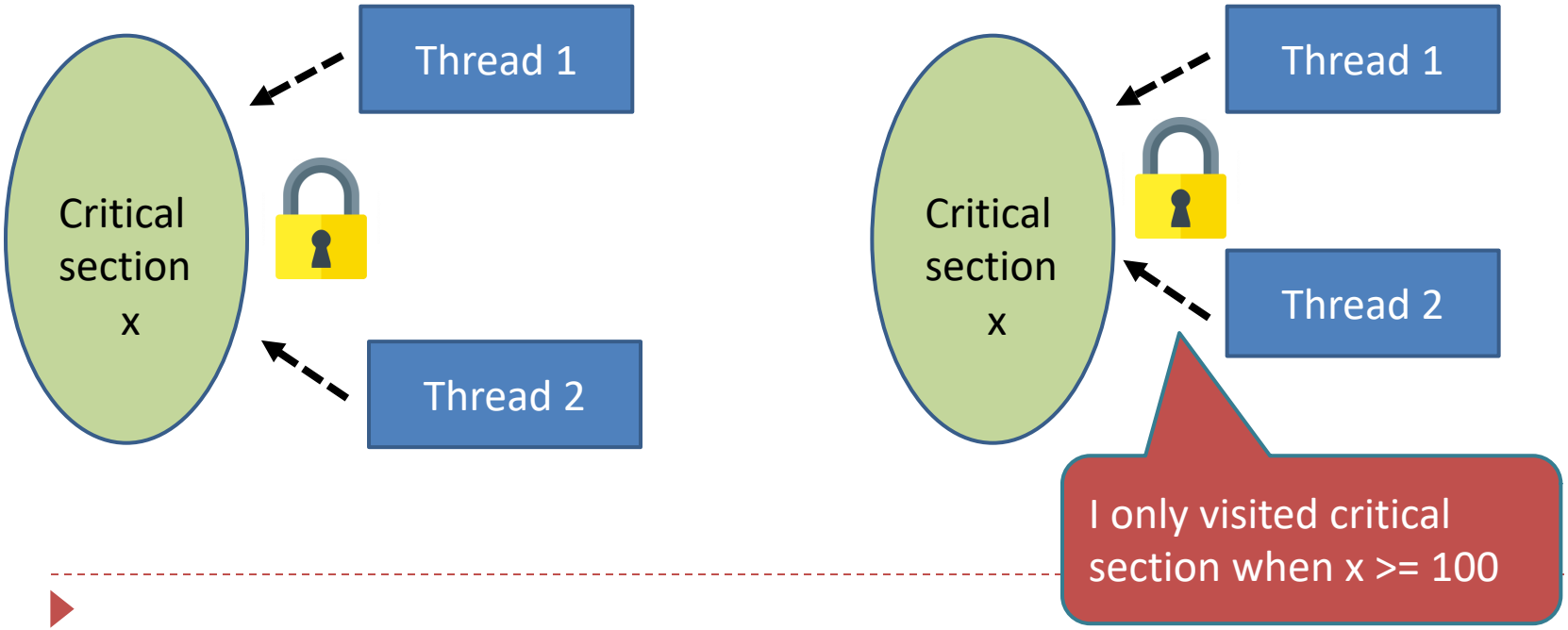
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int counter = 0;
static pthread_mutex_t mlock;

void *compute()
{
        int i = 0;
        pthread_mutex_lock(&mlock);
        while (i < 10000) {
                counter = counter + 1;
                i++;
        }
        pthread_mutex_unlock(&mlock);
        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);
}
```

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



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?



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)
{

    iCount++;

}
```

```
//thread 2:
while(true)
{

    if(iCount >= 100)
    {
        iCount = 0;
    }


}
```

T1 and T2 compete for variable iCount!

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;

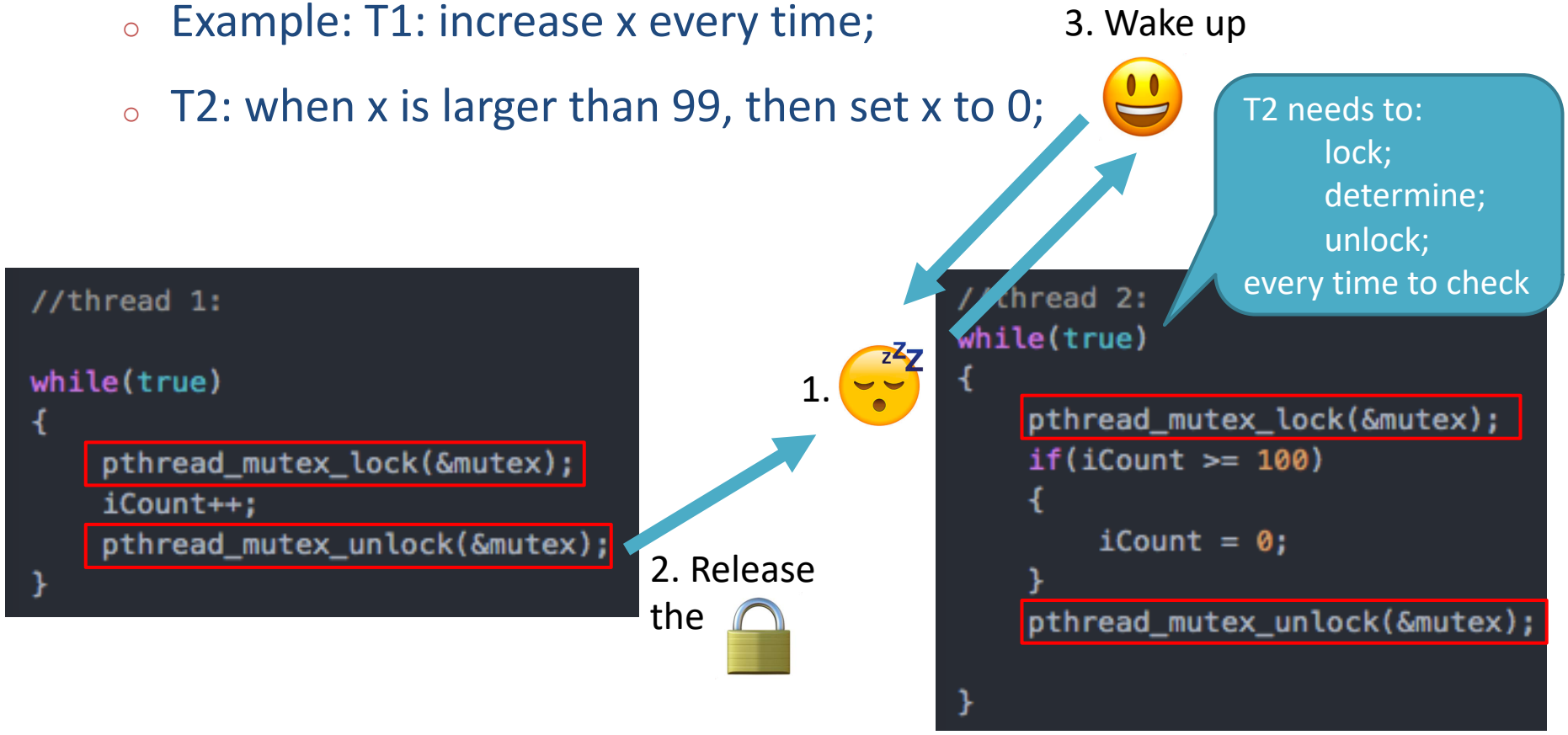
```
//thread 1:

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

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



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
4. release the

3. Wake up

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

```
//thread 1:

while(true)
{
    pthread_mutex_lock(&mutex);
    iCount++;
    pthread_mutex_unlock(&mutex);
}
```

2. Release the

1.

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

}
```

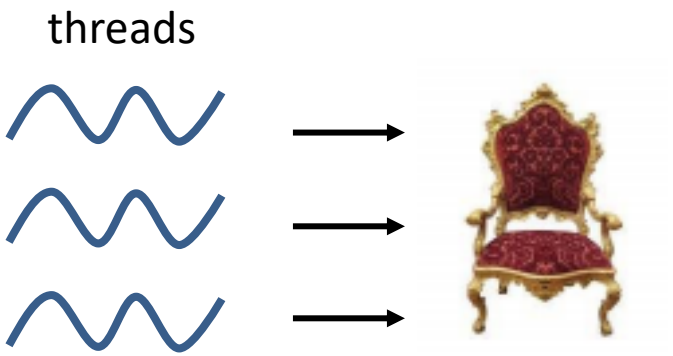
4. Semaphore

- Semaphore is a fundamental synchronization primitive used for locking of critical sections

- A semaphore “sem” is a special integer on which only two operations can be performed.
  - DOWN(sem)
  - UP(sem)

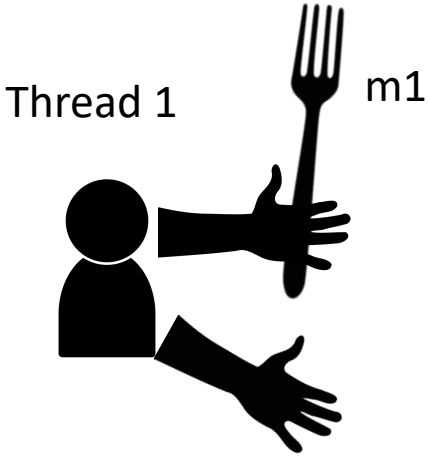
Mutex and semaphore

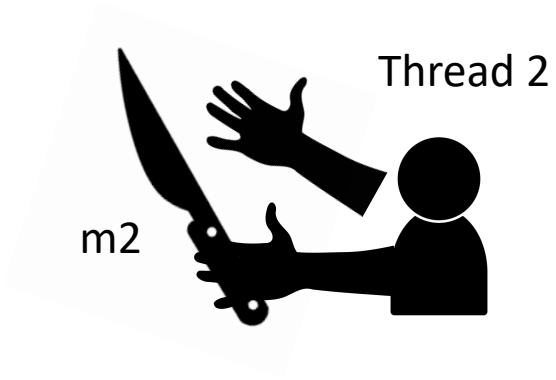
Mutex = 0 or 1



Sem = 0/1/2/3

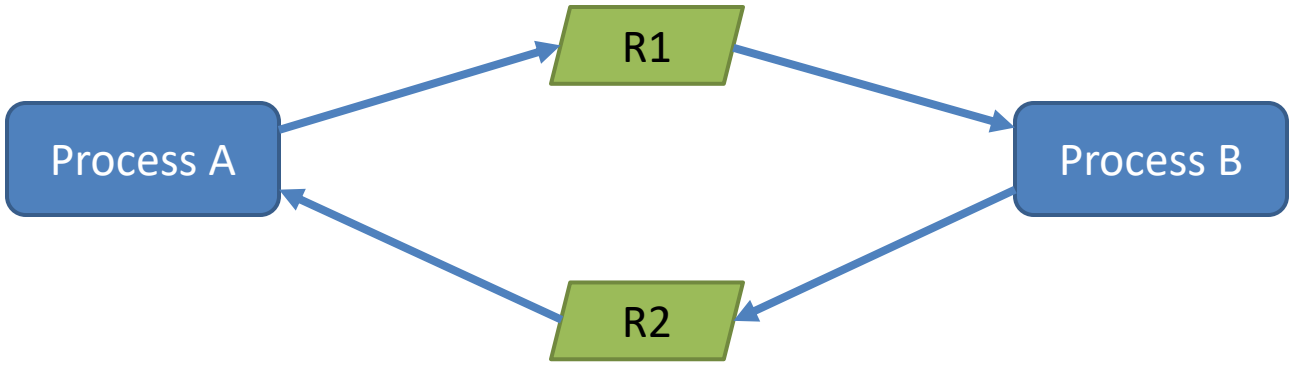
Deadlocks



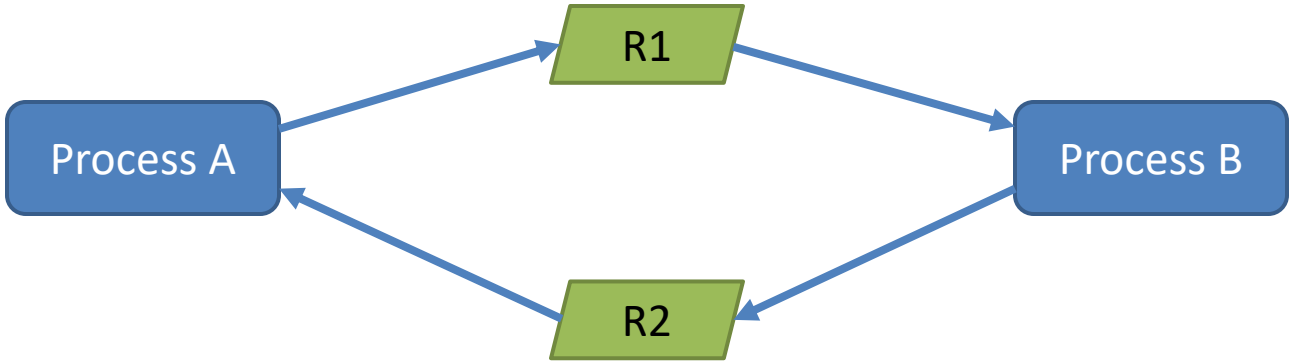


Deadlocks

- When two or more processes 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 not B can proceed because both are waiting for the other to proceed.



Deadlock example



Thread 1

```
pthread_mutex_lock(&m1);
/* use resource 1 */
pthread_mutex_lock(&m2);
/* use resources 1 and 2 */
do_something();
pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
```

Stuck right here

Thread 2

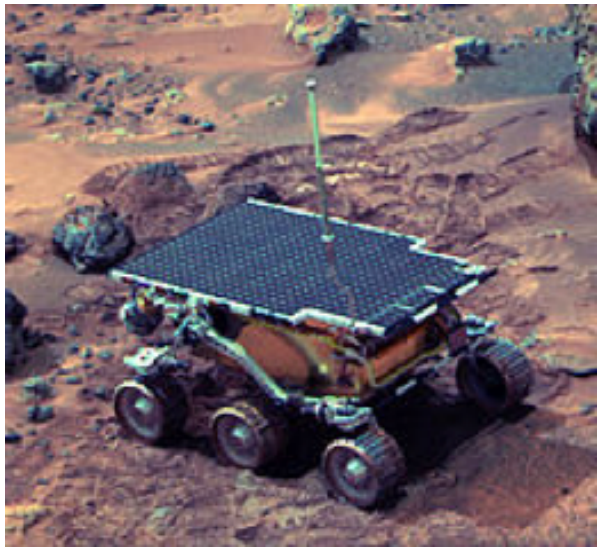
```
pthread_mutex_lock(&m2);
/* use resource 2 */
pthread_mutex_lock(&m1);
/* use resources 1 and 2 */
do_something();
pthread_mutex_unlock(&m1);
pthread_mutex_unlock(&m2);
```

Priority Inversion

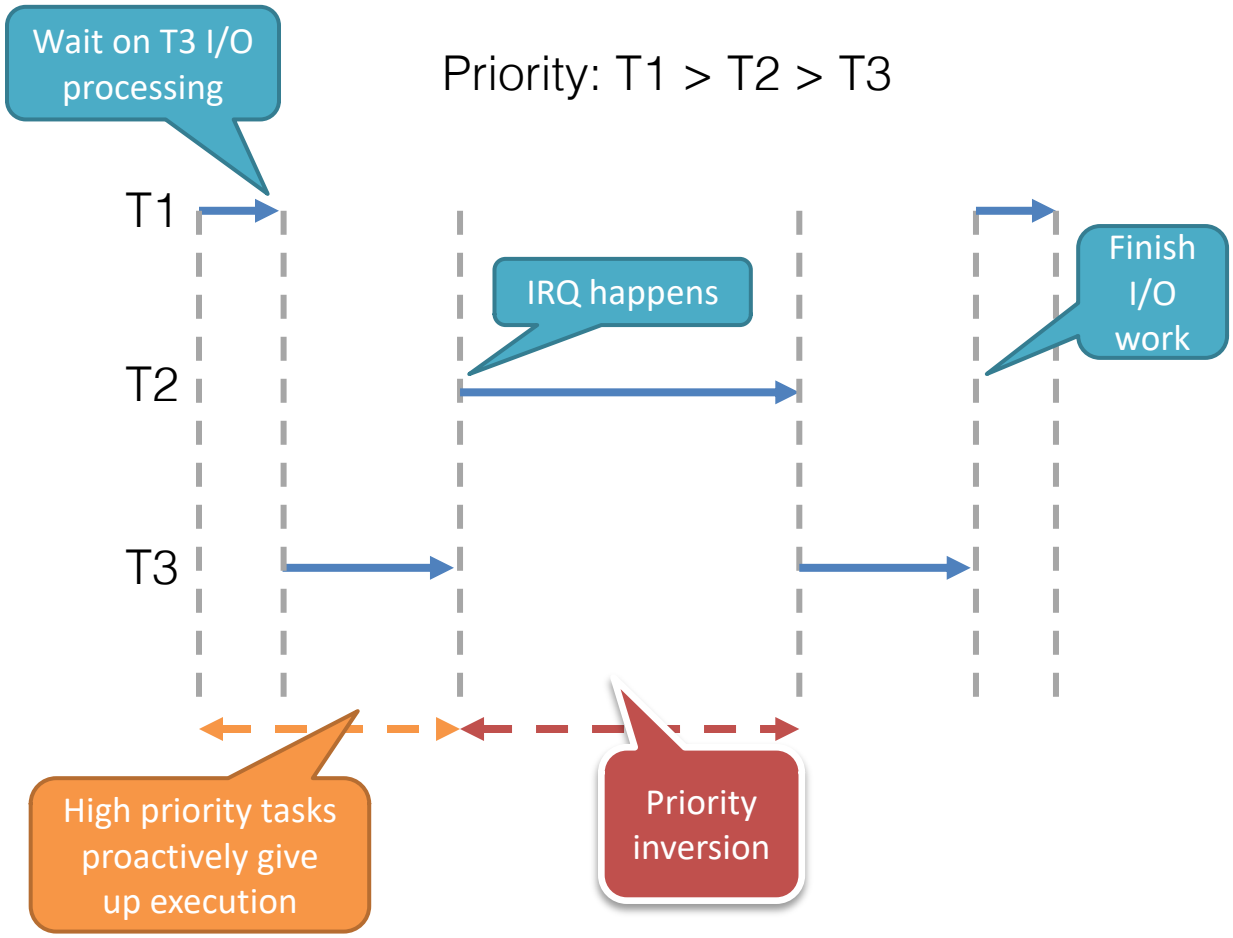
https://www.youtube.com/watch?v=lyx7kARrGeM

https://www.youtube.com/watch?v=t9RM5xcNUak

https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft



1997/07/04 Pathfinder
—> Mars



Conclusion

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