

Concurrency

Operating Systems

Moshe Sulamy

Tel-Aviv Academic College

Thus Far... Virtualization

- **Virtual CPU**

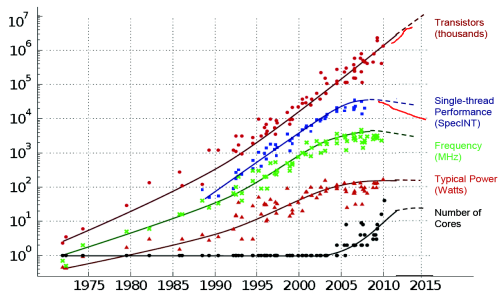
- Illusion of multiple programs at the same time
- **Context switch**
- **Scheduler**

- **Virtual Memory**

- Private **address space**
- **Segmentation**
- **Paging**
 - **TLB**
 - **Swapping**

CPU Trends

35 YEARS OF MICROPROCESSOR TREND DATA

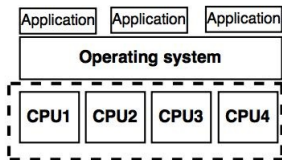


Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

- Transistor count doubles every 2 years (*Moore's law*)
- Clock speed tapering off (*Dennard scaling*)

CPU Trends

- Uniprocessor: nearly extinct
- New boss: multicore processor



- Faster programs → concurrent execution
 - Utilize many CPUs

Concurrency

- Faster: multiple activities at once
 - For example, array of 10^7 random integers
 - Output boolean array, true for each item > 50
 - **Sequential**: iterate entire array, 10^7 steps
 - **Concurrent**: 10 cores \rightarrow tenth the time

Concurrency

- Faster: multiple activities at once
 - For example, array of 10^7 random integers
 - Output boolean array, true for each item > 50
 - **Sequential**: iterate entire array, 10^7 steps
 - **Concurrent**: 10 cores \rightarrow tenth the time
- More responsive: perform one activity while another blocks
 - e.g., handle concurrent client requests in server
 - e.g., responsive GUI while background operation is running

Processes

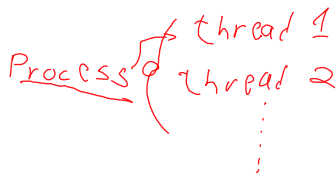
- Use processes for concurrency
 - Communicate via pipes or signals
 - e.g., Chrome (tab → process)
- Existing mechanism, but...

Processes

- Use processes for concurrency
 - Communicate via pipes or signals
 - e.g., Chrome (tab → process)
- Existing mechanism, but...
 - Cumbersome
 - Copy overhead
 - Expensive context switch


Processes

- Use processes for concurrency
 - Communicate via pipes or signals
 - e.g., Chrome (tab → process)
- Existing mechanism, but...
 - Cumbersome
 - Copy overhead
 - Expensive context switch



- New abstraction: thread

Thread

- Light-weight process
- Shared address space 
- **Multi-threaded program**
 - More than one point of execution

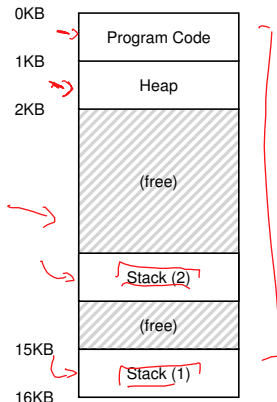
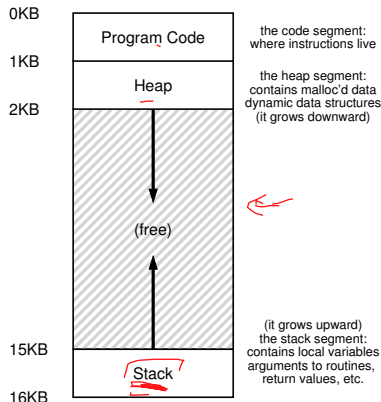


Thread

- Each thread:
 - Program counter (PC)
 - Set of registers
 - Stored in **thread control block (TCB)**
- On context switch:
 - Save register state of T1
 - Restore register state of T2
 - Address space remains the same

Thread

- One stack per thread
- Thread-local storage



Thread Models

- User threads

- Thread support in user space
- OS is not aware (user-managed)
- Fast thread creation and switching
- Willingly give up CPU, no actual concurrency

- Kernel threads

- Kernel provides structures and functionality
- Different cores, no forced cooperation
- Higher overhead to create and context switch

Why Use Threads?



- **Parallelism**

- Transform **single-threaded** program to utilize multiple CPUs
- e.g., perform operation on each item of a large array

- **Overlap**

- Avoid getting stuck when issuing I/O
- Perform other activities withing the same program

An Example

```

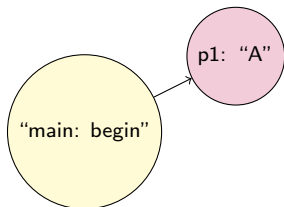
1 void* mythread(void *arg) {
2     printf("%s\n", (char*) arg);
3     return NULL;
4 }
5
6 int main(int argc, char *argv[]) {
7     pthread_t p1, p2;
8     printf("main: begin\n");
9     + (pthread_create(&p1, NULL, mythread, "A"));
10    + (pthread_create(&p2, NULL, mythread, "B"));
11    // join waits for threads to finish
12    pthread_join(p1, NULL);
13    pthread_join(p2, NULL);
14    printf("main: end\n");
15 }

```

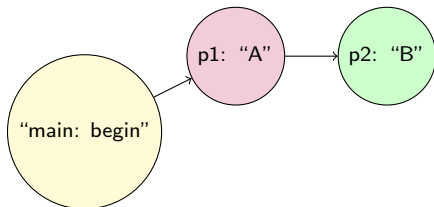
An Example



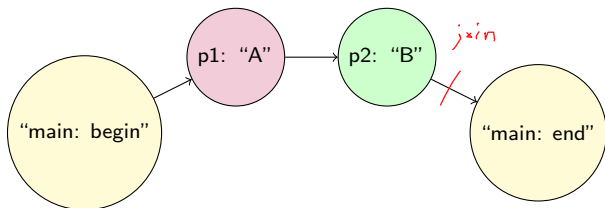
An Example



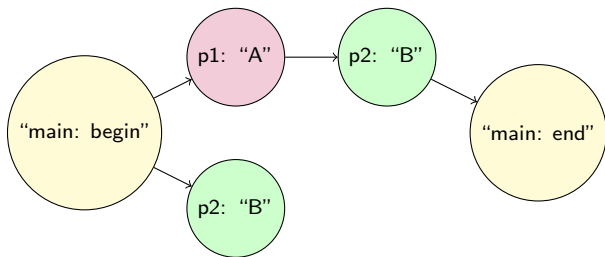
An Example



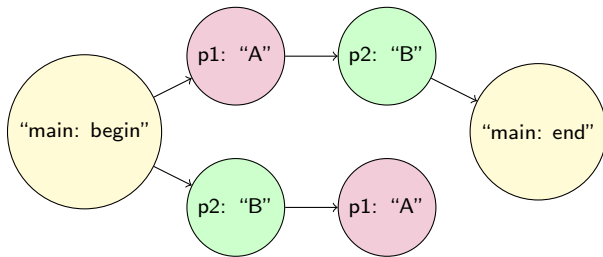
An Example



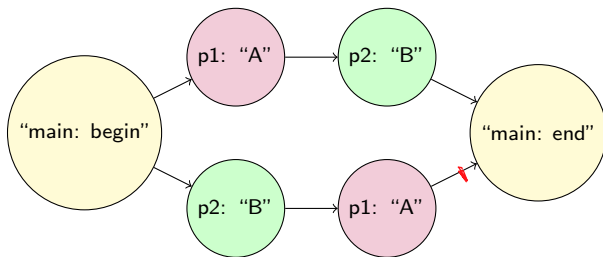
An Example



An Example

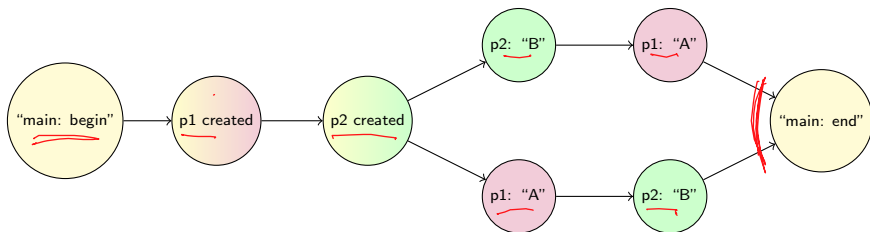


An Example



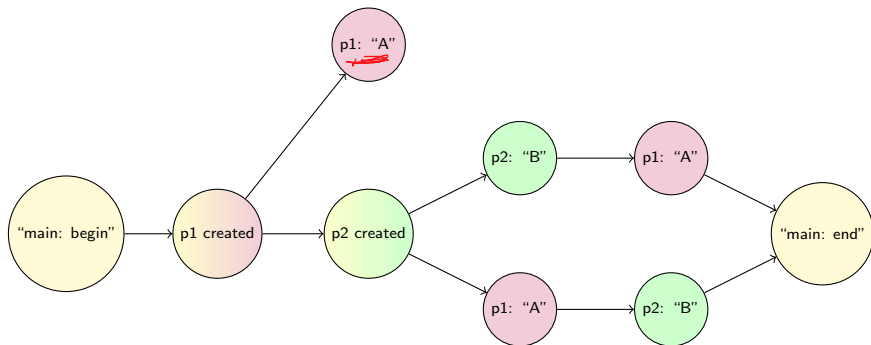
An Example

- It gets worse



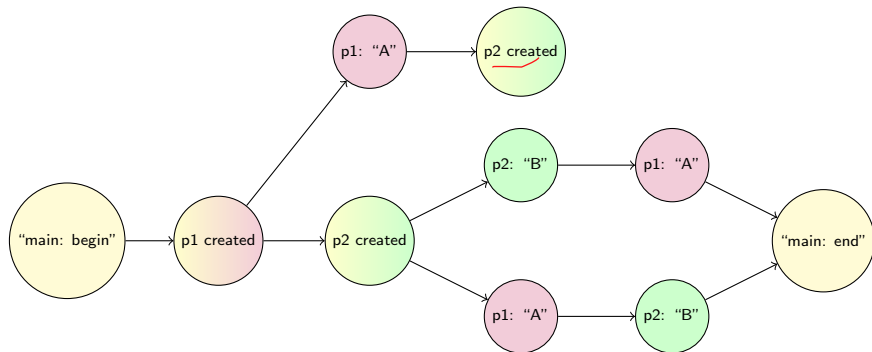
An Example

- It gets worse



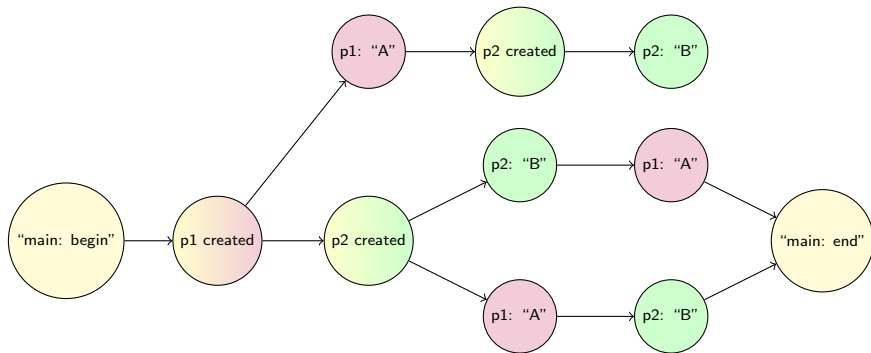
An Example

- It gets worse



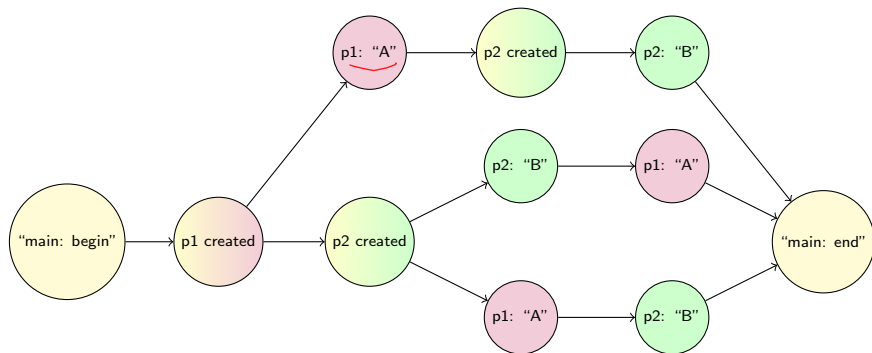
An Example

- It gets worse



An Example

- It gets worse



Shared Data

- Order depends on **scheduler**

```
1  static volatile int counter = 0;
2
3  void* mythread(void *arg) {
4      printf("%s: begin\n", (char*) arg);
5      for (int i = 0; i < 1e7; ++i) {
6          counter = counter + 1;
7      }
8      printf("%s: done\n", (char*) arg);
9      return NULL;
10 }
11 int main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     printf("main: begin (counter = %d)\n", counter);
14     pthread_create(&p1, NULL, mythread, "A");
15     pthread_create(&p2, NULL, mythread, "B");
16     // join waits for threads to finish
17     pthread_join(p1, NULL);
18     pthread_join(p2, NULL);
19     printf("main: end (counter = %d)\n", counter);
20 }
```

Shared Data

```
prompt> gcc main.c -o main -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: end (counter = 20000000)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: end (counter = 19345221)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: end (counter = 19221041)
```

Uncontrolled Scheduling

- Generated code sequence:
 - (assuming counter is at 0x8049a1c)

Machine code:

```
mov 0x8049a1c, %eax  
add $0x1, %eax  
mov %eax, 0x8049a1c
```

Pseudo-code:

```
int temp = counter;  
temp = temp + 1;  
counter = temp;
```

Thread 1

counter=50

Thread 2

Uncontrolled Scheduling

- Generated code sequence:
 - (assuming `counter` is at `0x8049a1c`)

Machine code:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Pseudo-code:

```
int temp = counter;
temp = temp + 1;
counter = temp;
```



Uncontrolled Scheduling

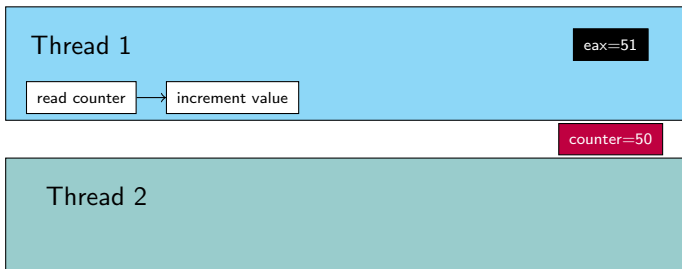
- Generated code sequence:
 - (assuming `counter` is at `0x8049a1c`)

Machine code:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Pseudo-code:

```
int temp = counter;
temp = temp + 1;
counter = temp;
```



Uncontrolled Scheduling

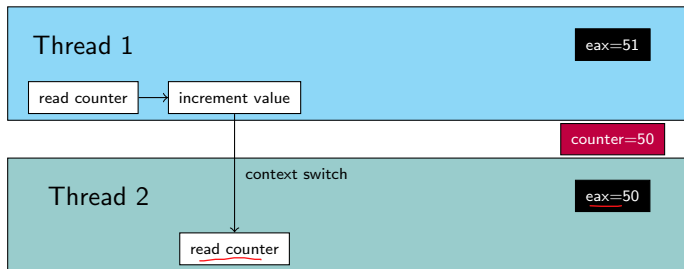
- Generated code sequence:
 - (assuming counter is at 0x8049a1c)

Machine code:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Pseudo-code:

```
int temp = counter;
temp = temp + 1;
counter = temp;
```



Uncontrolled Scheduling

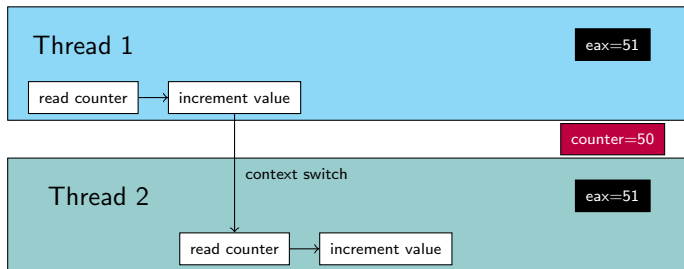
- Generated code sequence:
 - (assuming counter is at 0x8049a1c)

Machine code:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Pseudo-code:

```
int temp = counter;
temp = temp + 1;
counter = temp;
```



Uncontrolled Scheduling

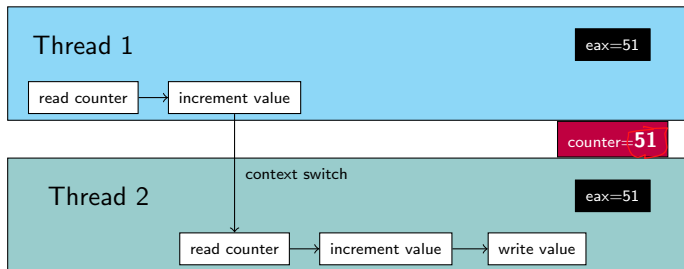
- Generated code sequence:
 - (assuming counter is at 0x8049a1c)

Machine code:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Pseudo-code:

```
int temp = counter;
temp = temp + 1;
counter = temp;
```



Uncontrolled Scheduling

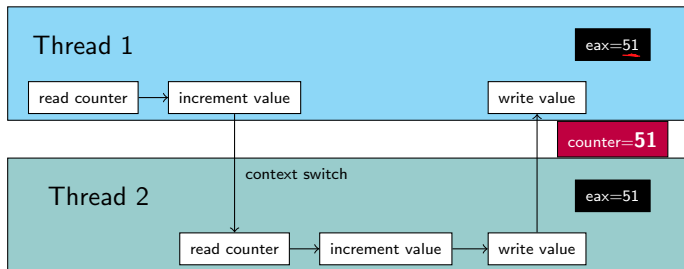
- Generated code sequence:
 - (assuming counter is at 0x8049a1c)

Machine code:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Pseudo-code:

```
int temp = counter;
temp = temp + 1;
counter = temp;
```



Uncontrolled Scheduling

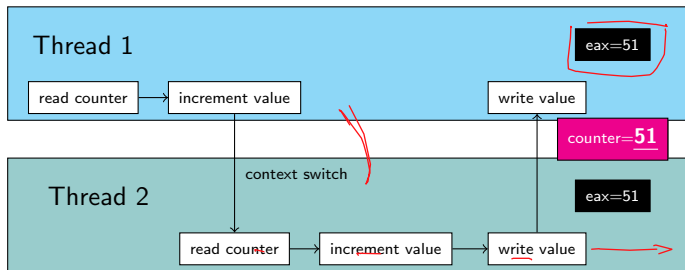
- Generated code sequence:
 - (assuming counter is at 0x8049a1c)

Machine code:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Pseudo-code:

```
int temp = counter;
temp = temp + 1;
counter = temp;
```



Uncontrolled Scheduling

- It gets worse

Thread 1

counter=50

Thread 2

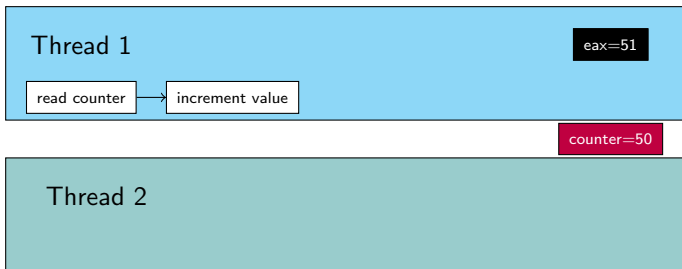
Uncontrolled Scheduling

- It gets worse



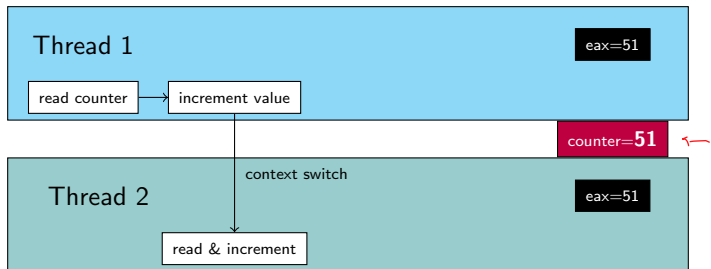
Uncontrolled Scheduling

- It gets worse



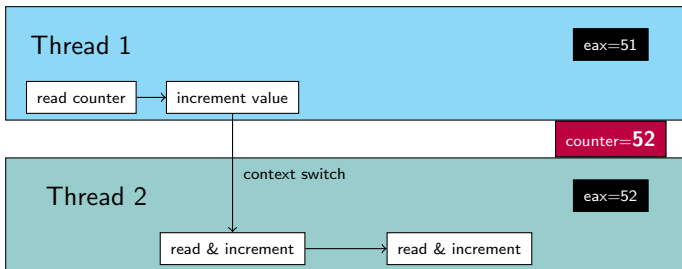
Uncontrolled Scheduling

- It gets worse



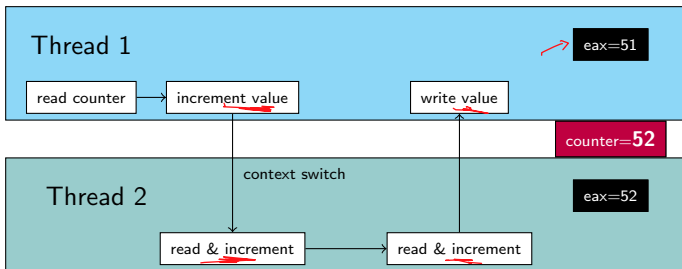
Uncontrolled Scheduling

- It gets worse



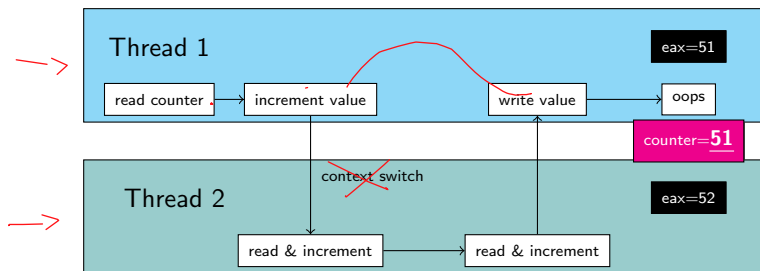
Uncontrolled Scheduling

- It gets worse



Uncontrolled Scheduling

- It gets worse



Uncontrolled Scheduling

- We demonstrated a **race condition**
 - Outcome is **indeterminate**, depends on scheduler
- Increment code is a **critical section**
 - Piece of code that accesses a shared resource
 - Must not be concurrently executed
- Our goal? ~~mutual exclusion~~
 - Only one thread may execute the critical section at a time

Atomicity

- **Atomic:** all or nothing
 - Either entire series of actions occur, or none of it
 - Sometimes called **transaction**
- **Synchronization primitives**
 - Atomic operations (e.g., increment)
 - Hardware and OS support

Atomicity

- **Atomic:** all or nothing
 - Either entire series of actions occur, or none of it
 - Sometimes called **transaction**
- **Synchronization primitives**
 - Atomic operations (e.g., increment)
 - Hardware and OS support

What support is needed to build useful synchronization primitives? How can we build these correctly and efficiently?

Summary

- Concurrency is everywhere
 - Leads to **indeterminate** execution, e.g., **race condition**
 - Depends on scheduler: correct execution means little!
- Solution?
 - Assume scheduler is **malicious**
 - Identify each **critical section** in your code
 - Piece of code that accesses a shared resource
 - Avoid with **mutual exclusion**
 - When T1 runs critical section, T2 doesn't, and vice versa
 - Using various **synchronization primitives**