Concurrency (ch. 26)

Operating Systems
Based on: Three Easy Pieces by Arpaci-Dusseaux

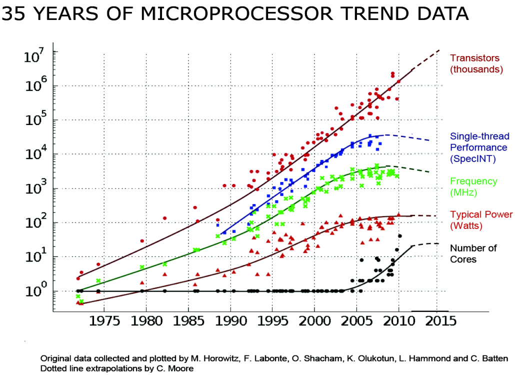
Moshe Sulamy

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**

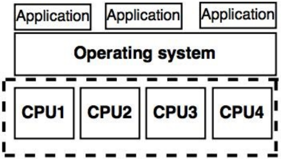
CPU Trends



- 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 $\longrightarrow$ 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 $\rightarrow$ process)

- Existing mechanism, but...

Processes

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

Processes

- Use processes for concurrency
  - Communicate via pipes or signals
  - e.g., Chrome (tab $\rightarrow$ process)
- Existing mechanism, but...
  - Cumbersome
  - Copy overhead
  - Expensive context switch
- New abstraction: **thread**

Thread

- Light-weight process
- Shared address space

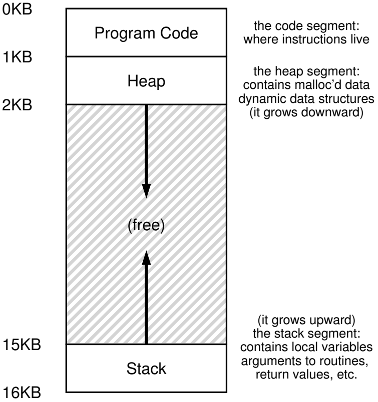
- **Multi-threaded program**
  - More than one point of execution

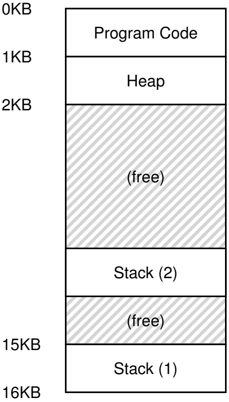
Thread

- 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?

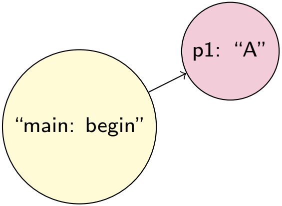
- **Parallelism**
  - Transform **single-threaded** program to utilize multiple CPUs
  - e.g., perform operation on each item of a large array
- **Overlap**
  - Avoid getting stuck when issuing I/O
  - Perform other activities withing the same program

An Example

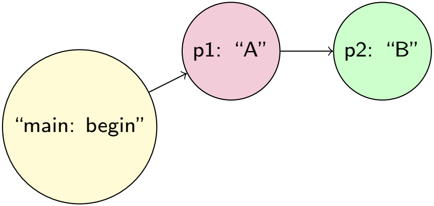
```c
void* mythread(void *arg) {
    printf("%s\n", (char*) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin\n");
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join waits for threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: end\n");
}
```

An Example

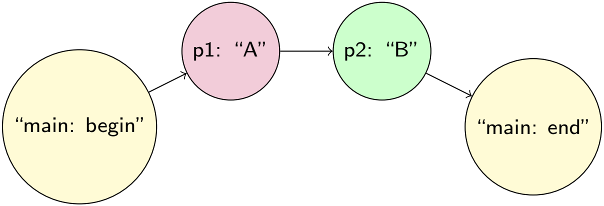
An Example



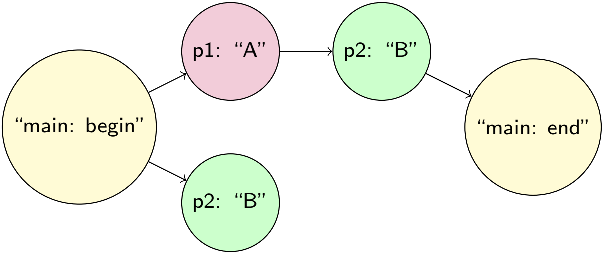
An Example



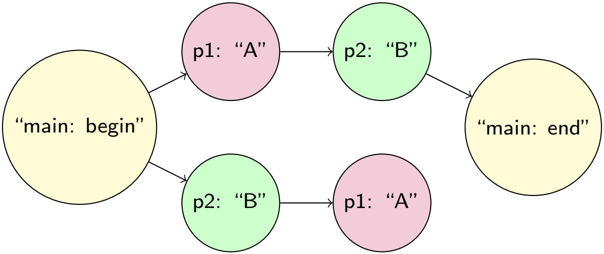
An Example



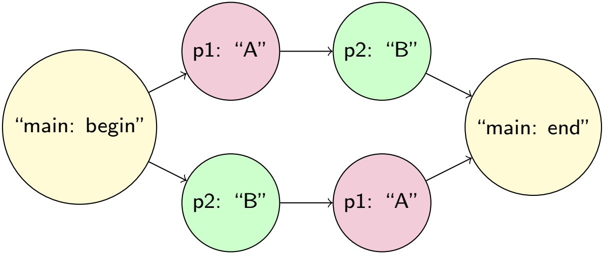
An Example



An Example

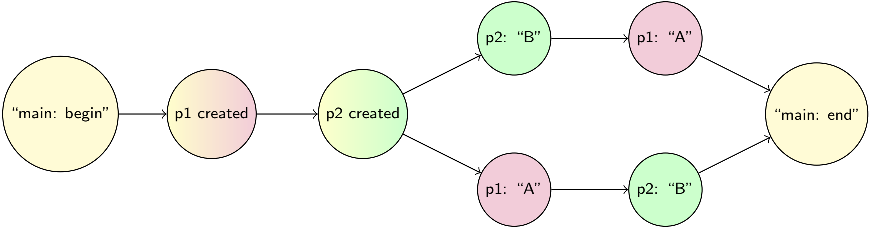


An Example



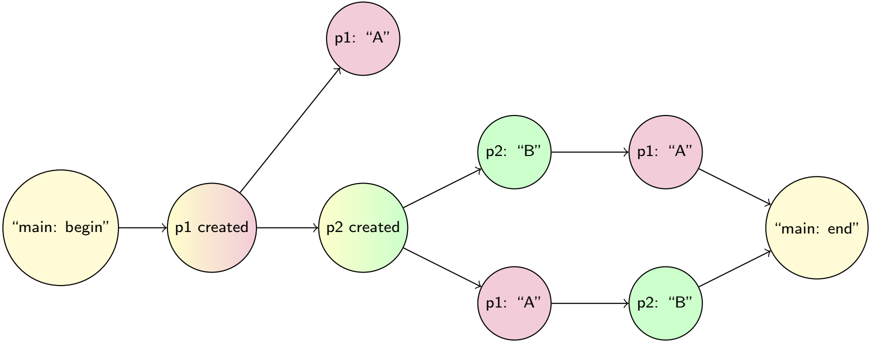
An Example

- It gets worse



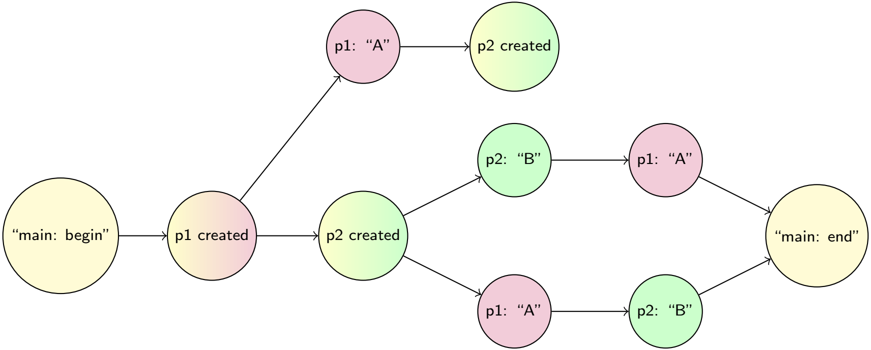
An Example

- It gets worse



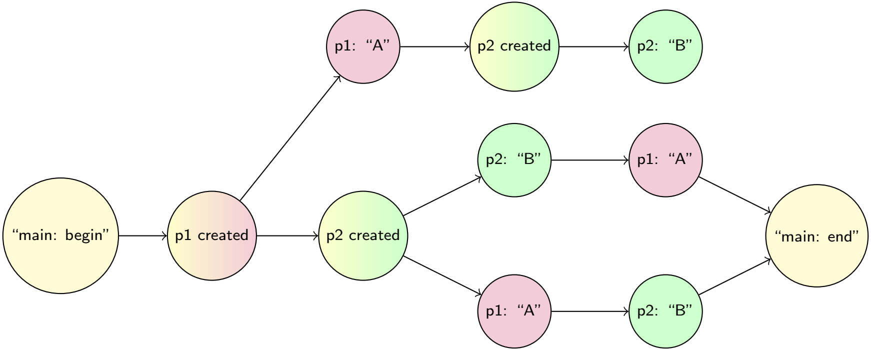
An Example

- It gets worse



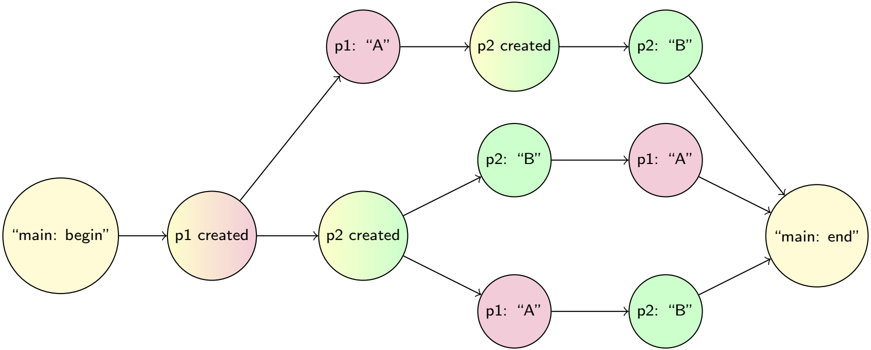
An Example

- It gets worse



An Example

- It gets worse



Shared Data

- Order depends on **scheduler**

```c
static volatile int counter = 0;

void* mythread(void *arg) {
    printf("%s: begin\n", (char*) arg);
    for (int i = 0; i < 1e7; ++i) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char*) arg);
    return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join waits for threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: end (counter = %d)\n", counter);
}
```

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;
```

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

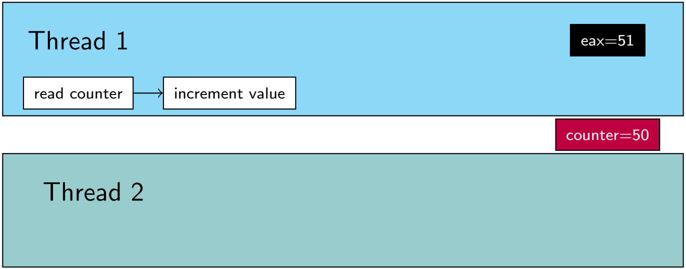
- Generated code sequence:
  - (assuming `counter` is at `0x8049a1c`)

Machine code:

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

Pseudo-code:

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



Uncontrolled Scheduling

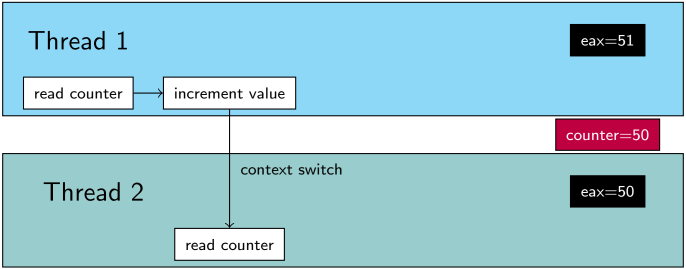
- Generated code sequence:
  - (assuming `counter` is at `0x8049a1c`)

Machine code:

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

Pseudo-code:

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



Uncontrolled Scheduling

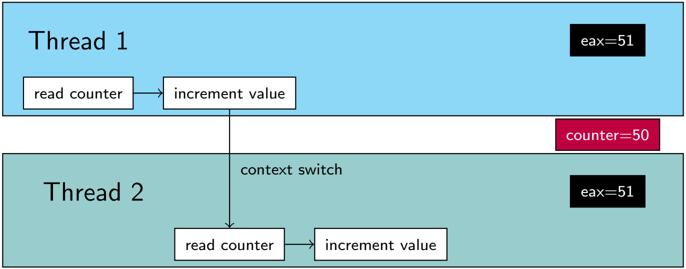
- Generated code sequence:
  - (assuming `counter` is at `0x8049a1c`)

Machine code:

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

Pseudo-code:

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



Uncontrolled Scheduling

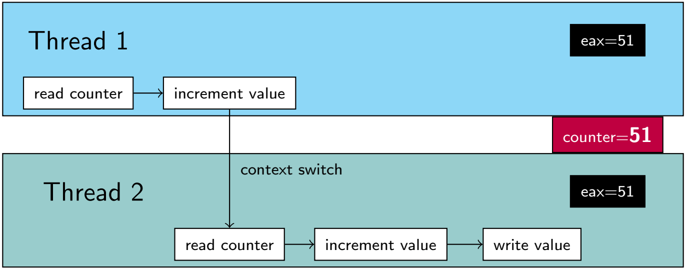
- Generated code sequence:
  - (assuming `counter` is at `0x8049a1c`)

Machine code:

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

Pseudo-code:

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



Uncontrolled Scheduling

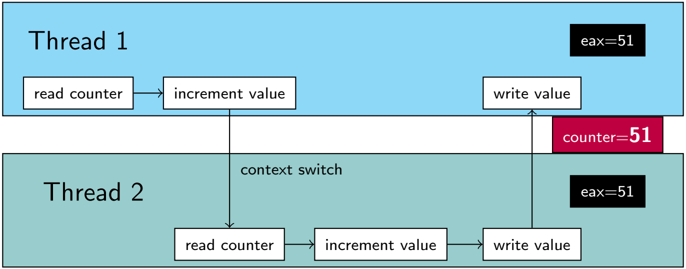
- Generated code sequence:
  - (assuming `counter` is at `0x8049a1c`)

Machine code:

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

Pseudo-code:

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



Uncontrolled Scheduling

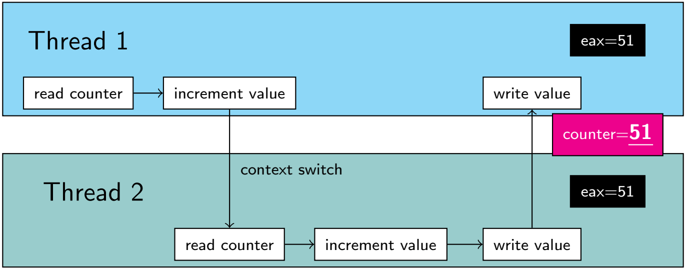
- Generated code sequence:
  - (assuming `counter` is at `0x8049a1c`)

Machine code:

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

Pseudo-code:

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



Uncontrolled Scheduling

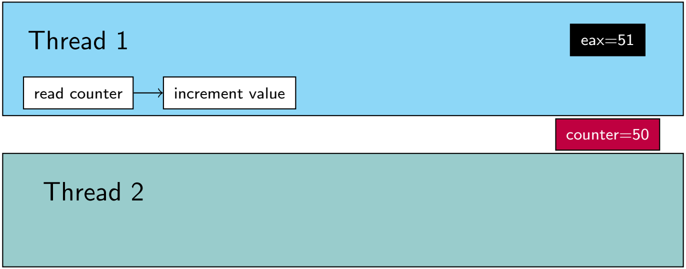
- It gets worse

Uncontrolled Scheduling

- It gets worse

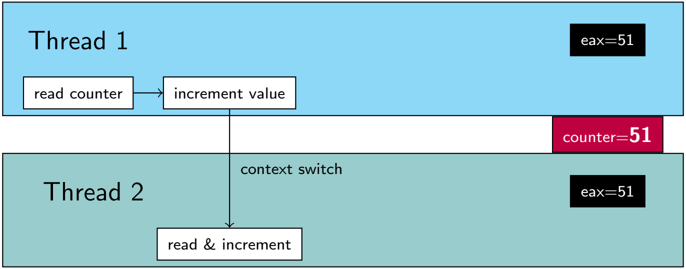
Uncontrolled Scheduling

- It gets worse



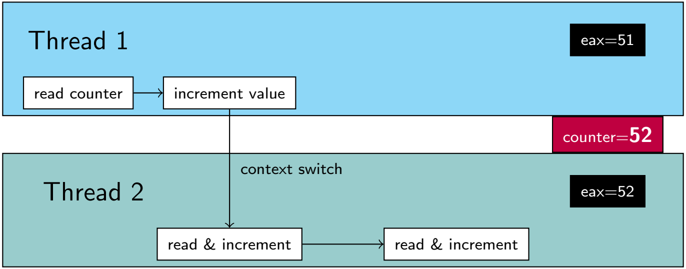
Uncontrolled Scheduling

- It gets worse



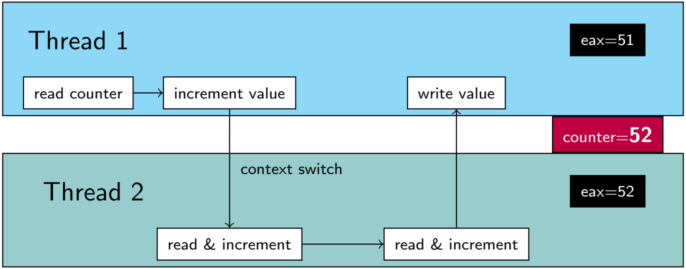
Uncontrolled Scheduling

- It gets worse



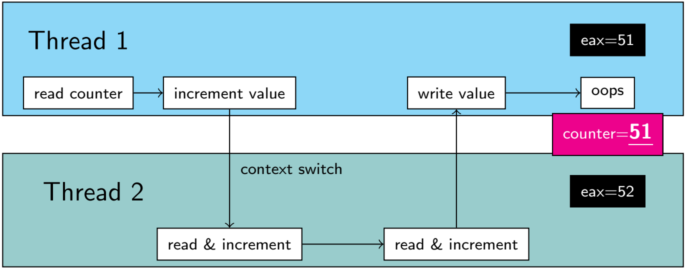
Uncontrolled Scheduling

- 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**