8/30/10

Reminder

Unix / Nachos Tutorials

SAL 109	Tuesday	5:30 - 7:00
SAL 126		7:00 - 8:30
SAL 109	Wednesday	5:30 - 7:00
		7:00 - 8:30

End of last lecture:

Application programmers want to be able to build programs using multiple execution streams and have them work together

This creates the possibility of Race Conditions

Key Question: How does the O.S. allow multiple execution streams to share data?

Mechanisms for sharing data between execution streams (between processes)

1. Message Passing

Requires a system call so the OS can can perform the data sharing transfer between 2 processes.

Disadv: The O.S. must run code for every sharing event.

2. Global Memory

Processes can request a "special" block of memory that can be shared. All authorized processes share this block of memory

Disadv: Still requires O.S. involvement at startup time.

Better Solution: Have a way for multiple execution streams to share data requiring "no" extra work by O.S.

⇓

Allow multiple execution streams inside a single process

⇓

Threads

To share data between threads we use global variables.

How does the O.S. track threads?

- Each thread has its own CPU state
  - in their process

CPU Scheduler now schedules threads

Restaurant (single execution stream)

Process

1. Customer arrives
2. Employee takes order
3. " cooks food
4. " bag food
5. " take money
6. Customer gets food & leaves

Multithreading is like having multiple, specialized employees

- Each thread performs its own task(s)
- Cooperation occurs through data sharing

Why are multiple execution streams, in a process, a better solution?

```
int i;

void myThread1(){
    ~~# do some th~~        int j;
    i=0;
    i= i+1;
}

void myThread2(){
    i=10;
    i=i-1;
}

void main(){
    // Make a thread using myThread1
    // Make a thread using myThread2
}
```

Result: A Process contains

1. A grouping of resources
2. One, or more, execution streams ↗ thread
3. CPU state saved separately for each thread

About Race Conditions:

Rule: Sequential execution within an execution stream

No restrictions on execution order between execution streams.

3 Issues

1. How to share data? already answered

2. How to ensure threads, in a process, execute one at a time?

3. How to ensure proper sequencing of events?

→ Big Problem

2 Ways to use Threads

① Can be independent of each other

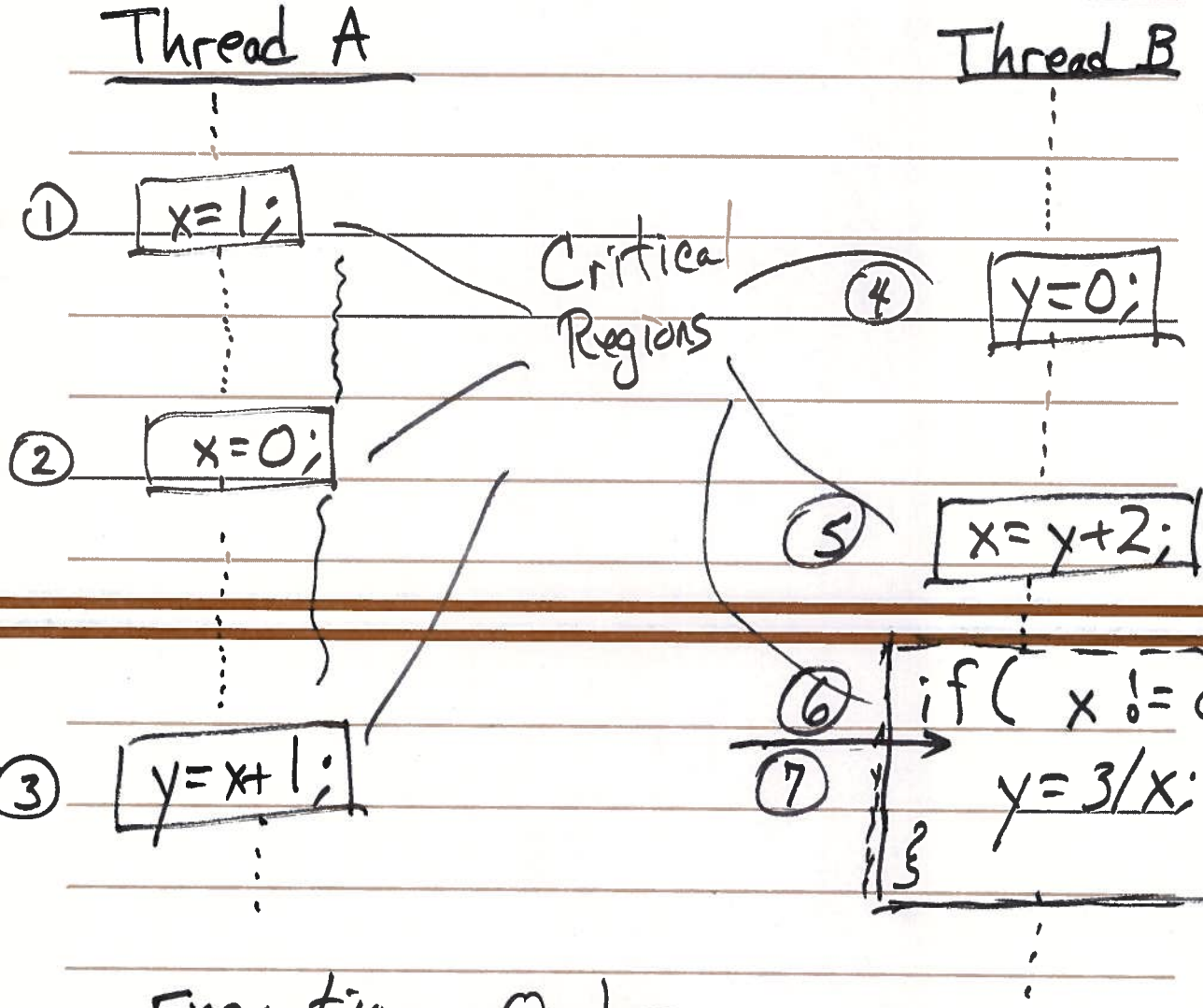
- no shared data
- no race conditions

② Cooperate Together

- share data
- Race conditions can occur

Simple Concurrent Problem

```
int x=0, y=0;      //global
```



Execution Order

..... 6, 2, 7 .....

Atomic Operation

One, or more, tasks that execute as a "single operation"

⇓

can't be split up

↙

No interference with shared data

Critical Region (Critical Section)

A section of code that utilizes shared resources

We use atomic operations to control access to critical regions.

If we do this properly, we can solve issues 2 & 3.