Concurrency

Mendel Rosenblum

Concurrency

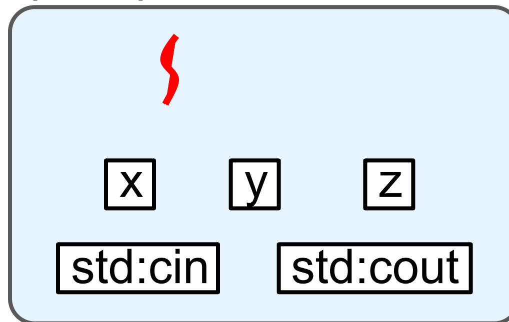
Optional readings:

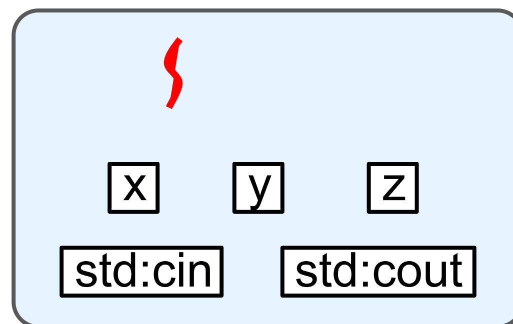
Operating Systems: Principles and Practice:

Chapter 5 up through Section 5.1

Independent Threads

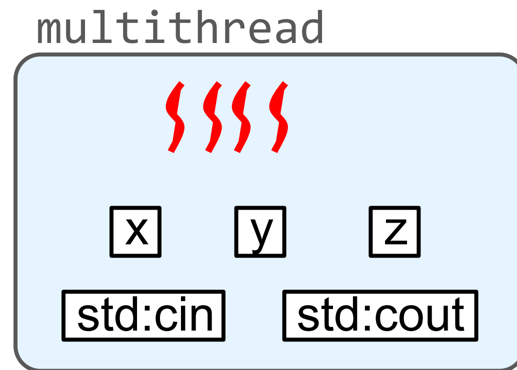
- No shared state
  - Thread cannot affect other threads
  - Thread cannot be affected by other threads
- Attractive properties
  - Deterministic
    - Depends only on input
  - Reproducible
  - Scheduling order doesn't matter
- Give examples of independent threads
  - Are they common?





Cooperating threads

- Share state
- Undesirable properties:
  - Behavior is nondeterministic
  - May be irreproducible
  - Depends on execution order
  - Scheduling matters



Cooperating Threads Example

Thread #1

```
cout << 'A' << 'B' << 'C';
```

Thread #2

```
cout << 'C' << 'B' << 'A';
```

- What outputs could we get?
- Can get different outputs, depending on execution order:
  ABCCBA
  ABCBCA
- Is this possible?
  AABBCC

Why permit threads to cooperate?

- Resource sharing
  - One disk, many files
  - One bank account file, many ATMs
- Performance
  - Parallel execution on multiple cores
  - Overlap I/O and computation
- Concurrency is necessary

When Does Order Matter?

Thread #1

```
A = 1;
```

Thread #2

```
B = 1;
```

- What are possible end values of A and B?
- Not all operations conflict
- Order of Thread #1 and #2 irrelevant

When Does Order Matter?

Thread #1

```
A = 1

A = B + 1;
```

Thread #2

```
B = 2;

B = 2 * B;
```

- Order of Thread #1 and #2 matters
  - What values are possible for A after both threads complete?
- Call a **race condition** (**race** for short)
- Need something to control order of threads
  - Atomic operations

Atomic operation

Before reasoning about cooperating threads: We need atomic operations.

Atomic = Appears to execute instantaneous to other threads

- Cannot be observed mid-execution
- Indivisible

What is Atomic?

Usually atomic:

- Single-word reads
- Single-word writes

Example:

```
int A, B
A = B;
```

You get:

- Either old value or new value
- Never partial bits

What Is Not Atomic?

Not usually atomic:

- x++
- struct copy
- multi-step operations

If no atomic operation exists, you cannot create one.

Hardware must provide atomic primitives. (Can't build one)

Building Higher-Level Constructs

If you have at least one atomic primitive

- You can build: Higher-level synchronization constructs

This is our strategy

- Build simple atomic operations, work but hard to use
- Build higher level atomic operations

Annoucements

Too Much Milk

	Roommate A
3:00	Arrive home: no milk
3:05	Leave for store
3:10	Arrive at store
3:15	Leave store
3:20	Arrive home, put milk away
3:25	
3:30	

Too Much Milk

	Roommate A	Roommate B
3:00	Arrive home: no milk	
3:05	Leave for store	
3:10	Arrive at store	Arrive home: no milk
3:15	Leave store	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Leave store
3:30		Arrive home: too much milk!

What do we want to happen?

We want:

- Safety: Avoid bad things
    - Never buy too much milk
- Liveness (availablibity): Do what it is supposed to do
    - Someone buys milk if needed

Must ensure both properties

Key Definitions

Synchronization

- Using atomic ops to ensure correctness of cooperating threads

Critical section

- Code only one thread executes at a time

Mutual exclusion

- Mechanism to enforce critical sections

  Commonly implemented via some locking mechanism

Computerized Milk Purchase

Process A

```
if (milk == 0) {
  buy_milk();
  milk = 1
}
```

Process B

```
if (milk == 0) {
  buy_milk();
  milk = 1
}
```

Error: too much milk

Computerized Milk Purchase

Process A

```
1 if (milk == 0) {
2   buy_milk();
3   milk = 1
4 }
```

Process B

```
1
2 if (milk == 0) {
3   buy_milk();
4   milk = 1
5 }
```

Error: too much milk

Computerized Milk Purchase (add note)

```
1 if (milk == 0) {
2  if (note == 0) {
3    note = 1;
4    buy_milk();
5    note = 0;
6  }
7 }
```

Does this work?

Still Too Much Milk

Thread A:

```
1  if (milk == 0) {
2   if (note == 0) {
3
4
5
6
7
8
9
10    note = 1;
11    buy_milk();
12    note = 0;
13  }
14 }
```

Need atomic op (lines 3–10)

Thread B:

```
if (milk == 0) {
  if (note == 0) {
    note = 1;
    buy_milk();
    note = 0;
  }
}
```

Less likely but not fixed

Second Attempt (note whose turns)

Thread A:

```
1 if (note == 0) {
2  if (milk == 0) {
3   buy_milk();
4  }
5  note = 1;
6 }
```

Thread B:

```
1 if (note == 1) {
2  if (milk == 0) {
3   buy_milk();
4  }
5  note = 0;
6 }
```

Does this work? (Safety and Liveness)

Safe but starve

Third Attempt (Notes for each thread)

Thread A:

```
1 noteA = 1;
2 if (noteB == 0) {
3  if (milk == 0) {
4    buy_milk();
5   }
6 }
7 noteA = 0;
```

Thread B:

```
1 noteB = 1;
2 if (noteA == 0) {
3  if (milk == 0) {
4    buy_milk();
5   }
6 }
7 noteB = 0;
```

Does this work?

Fourth Attempt

Thread A:

```
1 noteA = 1;
2 if (noteB == 0) {
3  if (milk == 0) {
4   buy_milk();
5  }
6 }
7 noteA = 0;
```

Thread B:

```
1 noteB = 1;
2 while (noteA == 1) {
3  // do nothing
4 }
5 if (milk == 0) {
6  buy_milk();
7 }
8 noteB = 0;
```

Fourth Attempt Evaluation

- Solution works but has two disadvantages:
  - Asymmetric
  - Busy waiting
    - Consumes CPU doing nothing
- A symmetric solution: **Peterson's Algorithm**

http://en.wikipedia.org/wiki/Peterson%27s_algorithm.

Where This Leads

The milk problem shows:

- Races are subtle
- Atomic reads/writes are insufficient
- Synchronization is tricky

We need better primitives:

- Locks
- Condition variables

Next lecture:
→ Proper synchronization constructs