CS61 Section 9 December 2

Problem Set

1. Get rid of data race

violates fundamental law of synchronization

T1	T2
X X = 0	y = x
X X = 1	x = 2
✓ y1 = x	y2 = x
✓ x = 2	read(pfd[0], ..., 1)
write(pfd[1], &ch, 1);	y = x

} can read and write at the same time

read blocks until there is something to read

int pfd[2];
int x;
(pipe empty)

same dependency item

there's a dependency here, read must finish

Data Race: ≥ 2 accesses from concurrent threads to the same object where ≥ 1 is a write

↑ fundamental data type

Mutex:

T1	T2
✓ m.lock()	m.lock()
x = 2	(A) y = x
m.unlock()	(B) m.unlock()
m.lock()	
X x = 2	y = x
m.unlock()	

std::mutex m
int x

either A or B dependency

↑ no ordering guarantee

Inducing ordering:

- mutex
- system calls
- thread creation
- thread join

```
main()                T1()          T2()
std::thread t1(T1);   X = 2         y = x
t1.join();
std::thread t2(T2);
t2.join();
```

Grain of Synchronization

What if T1 and T2 are running and some parts require mutual exclusion and others don't?

```
T1                          T2
m.lock()                    m.lock()
expensive_fnct1();          expensive_fnct2();
X = 2                       y = x
expensive_fnct3();          expensive_fnct4();
m.unlock()                  m.unlock()
```

fine-grained

coarse-grained
↳ small # of mutexes
large blocks of code

(2.) Run w/ sanitizer

- will tell specific lines of code
- put std::mutex m into the pong board, global var

More Threads

How threads work

- run strace
- clone() syscall when thread is created
  - ↳ same as fork but different args

share everything except thread id
share fd table
share addr space

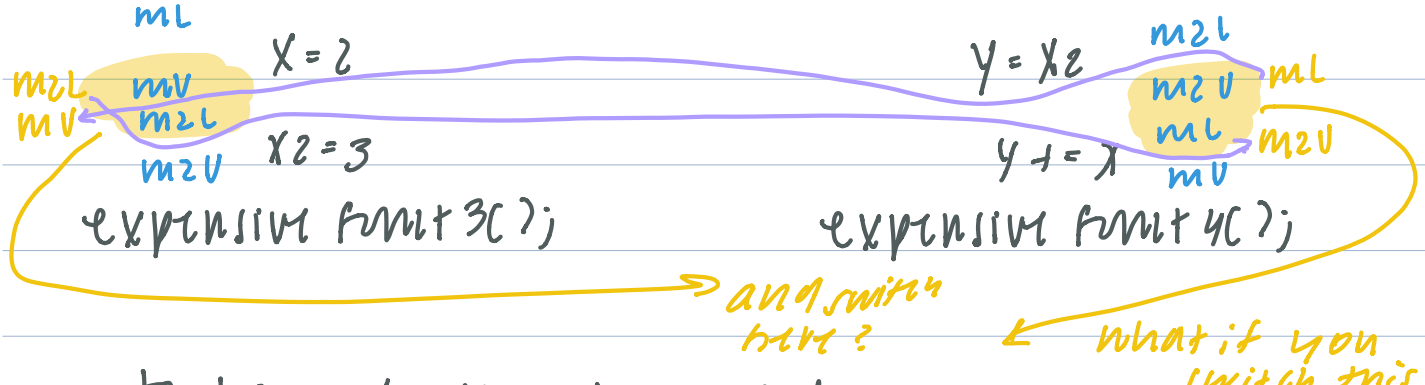
new addr space
new fd table
new pid

- mutual exclusion btwn child and parent, no data race b/c independent memory

Multiple mutexes lead to deadlock

std::mutex m, m2 (x, x2)

```
T1()                          T2()
expensive_fnct1();            expensive_fnct2();
```



to have deadlock, both need to have one lock and be waiting for another

solution: all locks in system are in order and only obtain locks in increasing order

Bounded Buffers

- structure meant to help threads communicate

writing thread ⟶ messages ⟶ reading thread

components:

- integer head, tail;
- infinite length array of messages m[∞];
- initially head = tail = 0;
- single char messages

write(ch)

① blocks for some amount of time (maybe)

② m[tail] = ch;

③ ++tail;

read(ch)

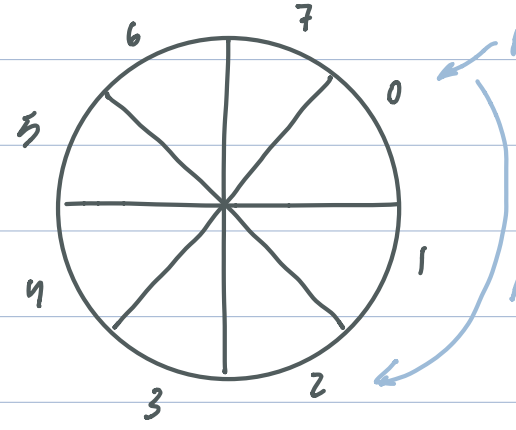
① block until head != tail

② ch = m[head];

③ ++head;

head ↓ tail ↓

A B C D E F G H I J K L M N O P Q R S T U V W X



8-char bounded buffer

pos = head % 8

len = tail - head

Why 8 slots?

· allow 2 ind thread entities to communicate

What about a rendezvous?

- only way to communicate is at same time
- bounded buffer len 0

```
struct rzvous {
    char* rd_ = nullptr;
    std::mutex m_;          std::cond_var cv_;
    void write(char ch) {
L →     while (!rd_) { ut }   cv_.wait(m_);
            *rd_ = ch;
U →         rd_ = nullptr;
            cv_.notify_all()
    }
    void read(char* ch) {
L →     while (rd_) { ut }   cv_.wait(m_);
          rd_ = ch;
          cv.notify_all();
U →     while (rd_ == ch) { ut }   cv_.wait(m_);
    }
```

polling to blocking until notified

never spin {} when lock is held

T1

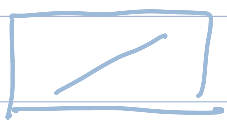
write('A')

... while ...

T2

read(&ch)

rd_



A