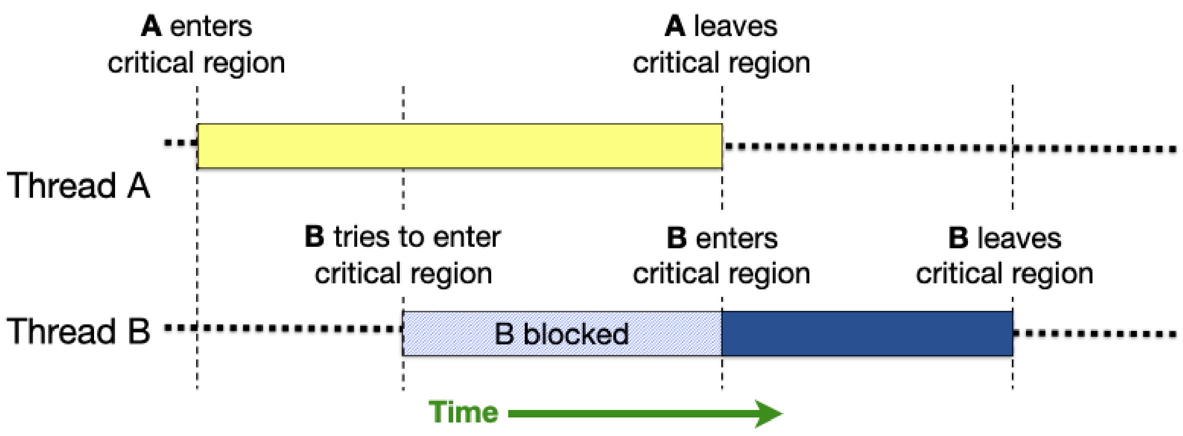
Synchronization

Learning Objectives:

1. Introduce race conditions and timing-based bugs
2. Introduce the mutual exclusion problem
3. Discuss solutions to gain familiarity with 'concurrent thinking'

Race Conditions:

1. A systems behavior is dependent on the timing or order of uncontrollable events
2. Some of your httpservers are exhibiting these!
   a. (OK) The behavior of read/recv depends upon how much data already arrived
   b. (buggy) The behavior of your application relies upon reading everything in one try
3. Can be caused by **Atomicity violation**: When some compound operation should be atomic but is not.
4. Can be caused by **Data Race**: When there are two or more unordered memory operations to the same variable , one of which is a write.
5. How can we prevent Race Conditions?
   a. Identify **critical sections** (also called, **critical regions**):
      i. The parts of a program that must be executed by only one thread at a time
      ii. Typically, these are all of the operations that access a shared resource
      iii. Conceptually, these are the parts of the program that must be atomic



   b. Supporting critical sections in general involves solving the **mutual exclusion** problem:

Mutual Exclusion:

1. How can we ensure that there is only ever one thread in a critical section at a time?
2. Four things to guarantee (according to xyz):
   a. No two threads may be in the critical region at the same time
   b. There can be any number of threads
   c. No thread that is outside of the critical region can block another thread
   d. No thread can wait forever to enter its critical region

The Strict Alternation Algorithm

```
// Strict Alternation algorithm
Globals:
 int turn;   // whose turn is it?
```

//Thread 0:	//Thread 1:

`while (1) {`
`  while (turn != 0) {}`
 	
`  critical_region();`
`  turn = 1;`
`  noncritical_region();`
`}`
	`while (1) {`
	`  while (turn != 1) {}`
	
	`  critical_region();`
	`  turn = 0;`
	`  noncritical_region();`
	`}`

1. Use single shared variable to determine whose runs next
2. Requires an atomic assignment
3. Does this provide mutual exclusion?
    a. Can there be multiple threads in the same region? No!
    b. Can we extend this to n threads? Yes!
    c. Can a thread block from outside the critical region? Yes :/
    d. Can a thread wait forever? Sorta? (Carshes)

Peterson’s Algorithm

```
// Peterson’s algorithm-``busy waiting’’
Globals:
 int turn;         // whose turn is it?
 int interest[2]; // who wants to run? Initialized to all 0’s.
```

```
//Thread i:

while (1) {

    // their index:
    int other = 1 - i;

    //I wanna run
    interest[i] = 1;

    //They get to run
    turn = other;

    //wait for them to have no interest or no turn
    while (interest[other] && turn==other) {}

    critical_section();
    interest[i] = 0;
    non_critical_section();
}
```

1. Solves the blocking issue
2. Requires an atomic assignment
3. Does this provide mutual exclusion?
    a. Can there be multiple threads in the same region?

Proof by contradiction:
Suppose that T0 is in the critical section and T1 enters the critical section (i.e., it breaks out of the while() loop on line 13). Since T0 is in the critical section, we know that when T1 enteres, turn is 1. If turn is 1, then we know that T1:10 happened before T0:10. Since we know that T0 is currently in the critical section, we know that T0:13 happened before T1:13. This gives us the following ordering chart:

T0:
interest[0] = 1
	T1:
	interest[1] = 0
	turn = 0

turn = 1
while (interest[1] && turn == 1) {}
criticail_region()	

	while (interest[1] && turn == 0) {}

This is a contradiction, since the condition in the while loop for T0 does not hold

b. Can we extend this to n threads? It's very difficult.
c. Can a thread block from outside the critical region? No, you can only block with interest[i] = 1
d. Can a thread wait forever?

Proof by contradiction (of a stronger property, that each thread will wait for at most one critical_region()):
Suppose that T1 waits for two T0 ciritical_regions(). This gives us the following ordering:

critical_region()	

	interest[1] =1
	turn=0
	while(interest[1] && turn == 0) {}

interest[0] =1
turn=1
while(interest[1] && turn == 0) {}
critical_region()	

	while(interest[1] && turn == 0) {}

This is a contradiction in that T0's while loop condition does not hold.

The Bakery Algorithm

1. Yet another mutual exclusion problem
2. **Relies on an analogy: A bakery’s ticket counter.**
    a. Thread “enters” the “bakery” (or, critical section)
    b. Thread “gets” the next ticket
    c. Thread waits for its turn
    d. Problem: what if two threads get the same number!?!?!
        i. Solution: treat a thread’s ID as a priority (a lower thread id will break ties).

```
Globals:
  int choosing[N]; // identifies if a thread is choosing a number
  int number[N]; // identifies the ticket of the current thread
```

```
//called before critical section, i is the current thread’s id
void lock(int i) {

  //get a number:
  choosing[i] = 1;
  number[i] = max(number[0], number[1],...,number[n-1]) + 1;
  choosing[i] = 0;

  //Wait for our turn to go
  for (int j = 0; j < N; ++j) {
    while(choosing[j]) {}
    while (number[j] != 0 && (number[j], j) < (number[i], i)) {}
  }
}

void unlock(int i) {
   number[i] = 0
}

void thread(int i) {
  lock(i);
  critical_section();
  unlock(i);
  non_critical_section();
}
```

3. Why do we need the choosing variable??
    a. Otherwise, the assignment to number is not atomic!
4. Does this provide mutual exclusion?
    a. Can there be multiple threads in the same region?

The proof is quite complex so I won't go through it in this class. I encourage you to read the paper if you're interested.

    b. N threads? Yep!
    c. Can a thread block from outside the critical region? No, you can only block if choosing or a lower number[]
    d. Can a thread wait forever? No, eventually the thread will have the smallest number.
5. Insanely insane property: Suppose that there are concurrent read and write operations, the algorithm does not depend upon a correct read, *the read can return any arbitrary value!*