

# Agenda

Thread concurrency and synchronization

- race conditions

- critical section

Tools for synchronization

- mutex

- barrier

- conditional variable

Deadlock

Concurrency examples: Producer/Consumer and Dining Philosophers

Thread Safety

# Thread concurrency and synchronization

- Need to think about memory and scope
  - Share: global & heap space
  - Own: stack & copy of local variables and parameters (but via pointers could share)
- Need to think about which parts of code can be executed simultaneously by multiple threads
  - May need to coordinate their access to the things they share (e.g. global variables)
  - May only want 1 thread to execute some parts (e.g. printing out final result)

# Exercise

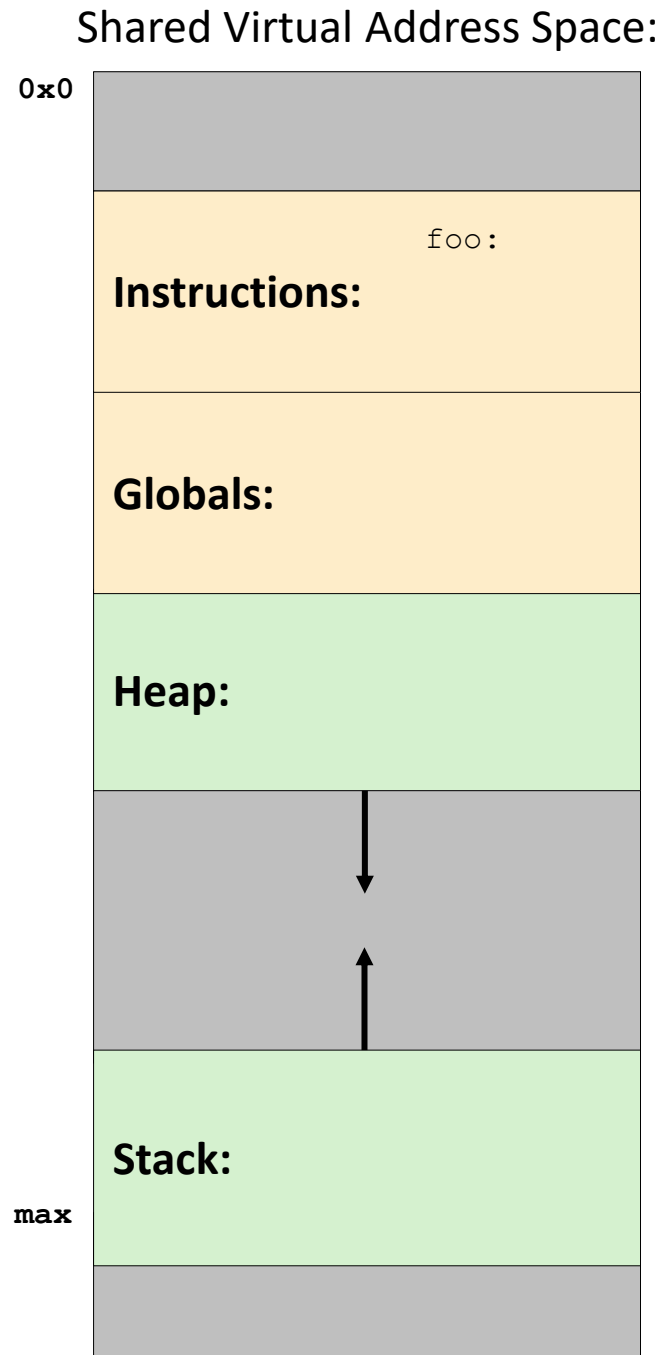
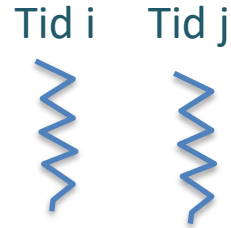
```
static int x = 0;

void* foo(void *usr) {
    int* p = (int*) usr;
    int y = *p;
    *p = *p + 1;
    x += y;
}
```

If threads i and j both execute function foo code:

Q1: which variables do they each get own copy of? which do they share?

Q2: which stmts can affect values seen by the other thread?



# Exercise: Draw a stack diagram

```
static int x = 0;

void* foo(void *usr) {
    int* p = (int*) usr;
    int y = *p;
    *p = *p + 1;
    x += y;
}

int main() {
    int pvalue = 35;
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, foo, &pvalue);
    pthread_create(&thread2, NULL, foo, &pvalue);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```

# Exercise: Draw a stack diagram (Possibility #1)

```
static int x = 0;

void* foo(void *usr) {
    int* p = (int*) usr;
    int y = *p;
    *p = *p + 1;
    x += y;
}
```

# Exercise: Draw a stack diagram (Possibility #2)

```
static int x = 0;

void* foo(void *usr) {
    int* p = (int*) usr;
    int y = *p;
    *p = *p + 1;
    x += y;
}
```

# Thread Synchronization

It is up to the Programmer to ensure that threads don't interfere in undesired ways

OS provides synchronization primitives that threads can use to coordinate their execution: mutex, barrier

Synchronizing access to shared variables:

- **block** thread(s) to wait until it is safe for them to execute a section of code that manipulates shared state
- **notify** blocked thread(s) that it is safe to proceed

# Thread synchronization

- A **race condition** occurs when a program's outcome differs based on CPU scheduling
- A **critical section** is a portion of code in which only one thread at a time can be executing

## Tools for synchronizing threads:

- **mutex** (mutual exclusion): guard that ensures only one thread executes a section of code
- **barrier**: guard that waits for all threads before continuing
- **semaphore**: maintains a count that is associated with a resource. Users decrement semaphore to “check out” resource and increment it when finished
- **condition variable**: a shared variable for coordinating the actions between threads



# Race Condition Example

```
main() {  
    int i;  
    for (i = 0; i < 100; i++) {  
        pthread_create(&tid, NULL, thread_main, &i);  
    }  
}
```

```
void *thread_main(void *arg) {  
    int i = *((int *)arg);  
    printf("%d: i=%d\n", pthread_self(), i);  
    ...  
}
```

Race Condition: outcome depends on scheduling  
order of concurrent threads on CPUs

If no race:

- each thread would ALWAYS get a unique value of i
- set of printed values would be 0 through 99 for every run  
(not necessarily printed in order, but each printed exactly once by a unique thread)

# Race condition example

hello I'm thread 61 with pthread\_id 140566268266240  
hello I'm thread 62 with pthread\_id 140566259873536  
hello I'm thread 29 with pthread\_id 140566545225472  
hello I'm thread 64 with pthread\_id 140566243088128  
hello I'm thread 67 with pthread\_id 140566217910016  
hello I'm thread 29 with pthread\_id 140566536832768  
hello I'm thread 30 with pthread\_id 140566587188992  
hello I'm thread 30 with pthread\_id 140566578796288  
hello I'm thread 63 with pthread\_id 140566251480832

# A Race Condition

In mult-threaded process, race conditions occur when multiple threads' **access to shared state is interleaved** and (could) affect program results/output/behavior

(ex) Sum of values from 1 to 100 is 5050

if one run of threaded Sum computes 5050, other 5040

An **atomic** action is a task that cannot be interrupted (no risk of a race condition)

(ex) single read or write of up to some (small) HW defined size

```
int x = 6; // x gets all bits of 6 or none but not partial  
movl $6, -8(%ebp)
```

# Atomic instructions (all or nothing?)

- Single Instruction either executed or not
  - OS can context switch *at any time (at any clock cycle)*
  - OS restarts thread at interrupted instruction(s)
    - All instructions in the pipeline that did not complete their STORE phase (F,D,E,S) are restarted
- But actions of a single instruction may not be atomic  
`addl $20, (%eax) # not atomic: a Read-Modify-Write op.`
- Atomic action: single read (load) or write (store) of up to some max size (typically 4 or 8 bytes)  
`movl $20, (%eax) # atomic: all or none of 20 is written`

# pthread Synchronization

## Mutex: mutually exclusive access

```
pthread_mutex_t mutex;  
pthread_mutex_init(&mutex, NULL);
```

```
pthread_mutex_lock(&mutex);  
pthread_mutex_unlock(&mutex);  
pthread_mutex_destroy(&mutex);
```

## Condition variable

```
pthread_condition_t shared;  
pthread_condition_init(&shared, NULL);
```

```
pthread_condition_signal(&shared);  
pthread_condition_wait(&shared);  
pthread_condition_destroy(& shared);
```

## Barrier : block until all threads have called wait “wait until we all get to this point”

```
pthread_barrier_t mybarrier;  
pthread_barrier_init(&mybarrier, NULL, numtids);
```

```
pthread_barrier_wait(&mybarrier);  
pthread_barrier_destroy(&mybarrier);
```

# Demo: Updating a counter

```
static unsigned long long count = 0; // global variable

int main(int argc, char *argv) {
    pthread_t *tids;
    int ntids, i, *tid_args;

    ntids = atoi(argv[1]);
    tids = (pthread_t *)malloc(sizeof(pthread_t)*ntids);
    tid_args = (int *)malloc(sizeof(int)*ntids);

    for (i=0; i < ntids; i++) {
        tid_args[i] = i;
        pthread_create(&tids[i], 0, thread_hello, &tid_args[i]);
    }
    for (i=0; i < ntids; i++) {
        pthread_join(tids[i], 0);
    }
    ...
}
```

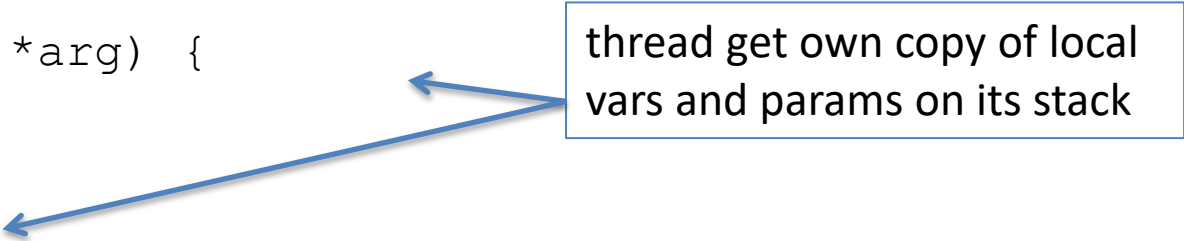
|                       |  |
|-----------------------|--|
| <b>thread_hello</b> : | // each spawned thread's "main" function |
| <b>tid_args[i]</b> :  | // each spawned thread's logical tid     |
| <b>tids[i]</b> :      | // each spawned thread's pthread tid     |

# Demo: Updating a counter (thread main function)

```
static unsigned long long count =0; // global variable
```

```
void *thread_count(void *arg) {
```

thread get own copy of local  
vars and params on its stack



```
int myid, i;  
myid = *((int *)arg);
```


```
printf("hello I'm thread %d with pthread_id %lu\n",  
      myid, pthread_self());
```

```
for(i = 0; i < 100000; i++) {
```

```
    count += i;
```

```
}
```

all threads can access global variable (shared access)



```
printf("goodbye I'm thread %d\n",myid);  
return (void *)0; // recast 0 to return type (void *)
```

```
}
```

# Some runs with 4 threads:

result with 4 threads should be 400000

```
hello I'm thread 0 with pthread_id 140449083262720
hello I'm thread 1 with pthread_id 140449074870016
goodbye I'm thread 0
goodbye I'm thread 1
hello I'm thread 2 with pthread_id 140448987346688
hello I'm thread 3 with pthread_id 140449066477312
goodbye I'm thread 3
goodbye I'm thread 2
count = 263742
count time is 0.010365
```



# Critical Section

Let's put `count++` in a critical section:

```
pthread_mutex_t mutex;  
  
pthread_mutex_lock(&mutex);  
count++;  
pthread_mutex_unlock(&mutex);
```

# Visualizing our counter program: flow diagram

# Visualizing our counter program: timeline

# Aside: thread performance with mutex

Threads incur overhead

more threads not necessarily faster!

mutex is expensive

“embarrassingly parallel”: algorithms in which the same work is done for lots of data

# Compare the performance

```
void *thread_count(void* args) {
    int myid, i;
    myid = *((int*) args);
    printf("hello I'm thread %d (%lu)\n",
        myid, pthread_self());

    for(i = 0; i < 100000; i++) {
        pthread_mutex_lock(&mutex);
        count += 1;
        pthread_mutex_unlock(&mutex);
    }

    printf("goodbye I'm thread %d\n",myid);
    return (void *)0;
}
```

```
void *thread_count(void* args) {
    int myid, i;
    myid = *((int*) args);
    printf("hello I'm thread %d (%lu)\n",
        myid, pthread_self());

    int total = 0;
    for(i = 0; i < 100000; i++) {
        total += 1;
    }

    pthread_mutex_lock(&mutex);
    count += total;
    pthread_mutex_unlock(&mutex);

    printf("goodbye I'm thread %d\n",myid);
    return (void *)0;
}
```

# Exercise: mutex

Sketch a program where two threads count the number of letters in `tolstoy.txt`:

letter count = 2513005

What data needs to be shared?

# Deadlock

Idea: All threads are simultaneously blocked, waiting for other threads!

- synchronization problem!

Potential problem with all distributed systems, not just threads

# Deadlock: Example

Idea: Transferring money from A to B  
and from B to A simultaneously

```
void *Transfer(void *args){  
    //argument passing removed to increase readability  
    //...  
    pthread_mutex_lock(&fromAcct->lock);  
    pthread_mutex_lock(&toAcct->lock);  
  
    fromAcct->balance -= amt;  
    toAcct->balance += amt;  
  
    pthread_mutex_unlock(&fromAcct->lock);  
    pthread_mutex_unlock(&toAcct->lock);  
    return NULL;  
}
```



# Deadlock: example

## Thread 0

```
Transfer(...) {  
    //acctA is fromAcct  
    //acctB is toAcct  
    pthread_mutex_lock(&acctA->lock);  
    Thread 0 gets here  
    pthread_mutex_lock(&acctB->lock);
```

## Thread 1

```
Transfer(...) {  
    //acctB is fromAcct  
    //acctA is toAcct  
    pthread_mutex_lock(&acctB->lock);  
    Thread 1 gets here  
    pthread_mutex_lock(&acctA->lock);
```

*Figure 1. An example of deadlock*

Suppose that Threads 0 and 1 are executing concurrently and represent users A and B, respectively. Now consider the situation in which A and B want to transfer money to each other: A wants to transfer 20 dollars to B, while B wants to transfer 40 to A.

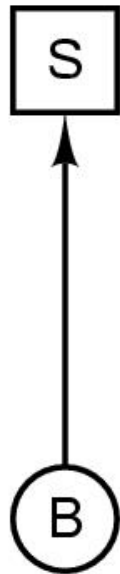
# Deadlock: Visualization

Circles represent processes/threads  
Squares represent resources



(a)

A **holds** resource R



(b)

B **requests** resource R

Resource Allocation Graph (RAG)

If our graph has a cycle, then we have deadlock

# Deadlock: Visualization

## Thread 0

```
Transfer(...) {  
    //acctA is fromAcct  
    //acctB is toAcct  
    pthread_mutex_lock(&acctA->lock);  
    Thread 0 gets here  
    pthread_mutex_lock(&acctB->lock);
```

## Thread 1

```
Transfer(...) {  
    //acctB is fromAcct  
    //acctA is toAcct  
    pthread_mutex_lock(&acctB->lock);  
    Thread 1 gets here  
    pthread_mutex_lock(&acctA->lock);
```

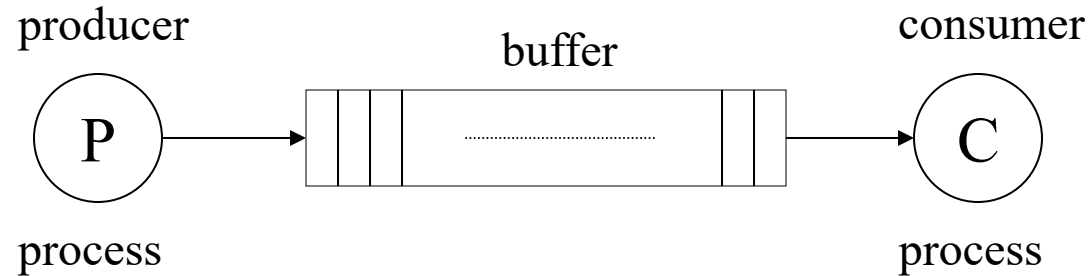
# Deadlock: Fixed

```
void *Transfer(void *args){
    //argument passing removed to increase readability
    //...

    pthread_mutex_lock(&fromAcct->lock);
    fromAcct->balance -= amt;
    pthread_mutex_unlock(&fromAcct->lock);

    pthread_mutex_lock(&toAcct->lock);
    toAcct->balance += amt;
    pthread_mutex_unlock(&toAcct->lock);
    return NULL;
}
```

# The Producer/Consumer Problem



A shared fixed-size buffer (e.g. an array)

Two types of threads

1. Producer: creates items and adds them to the buffer. Must wait if the buffer is full
2. Consumer: removes items. Must wait if the buffer is empty.

Many real-world examples:

Printer queue

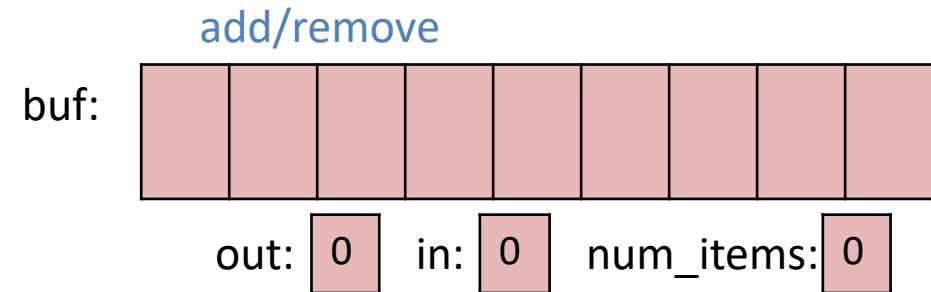
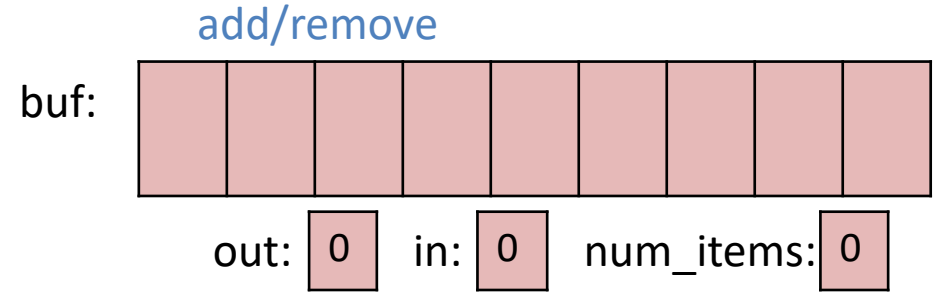
CPU queue of ready threads/processes

Network messaging

# Example: Producer/Consumer

Producer: Add 2,4,6,8

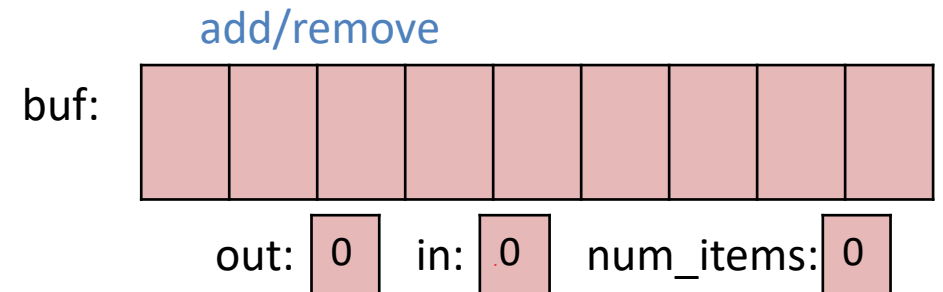
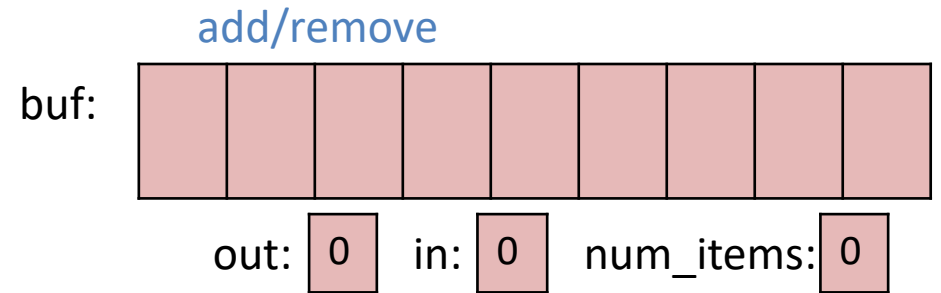
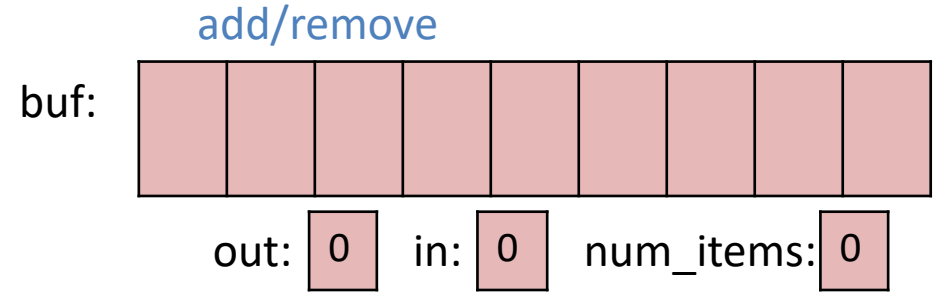
Consumer: Read 2



# Example: Producer/Consumer

Producer: Add 10,12,14,16,18

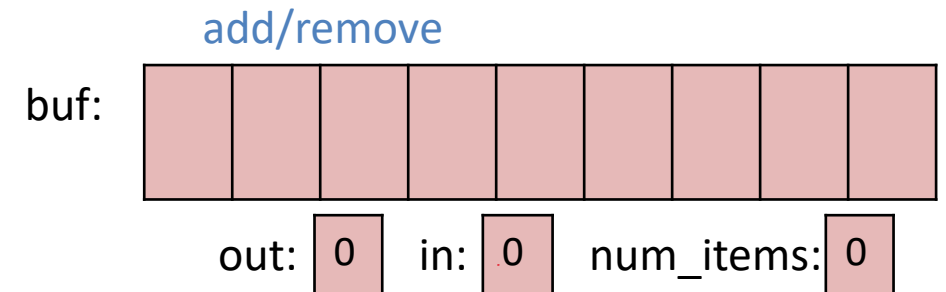
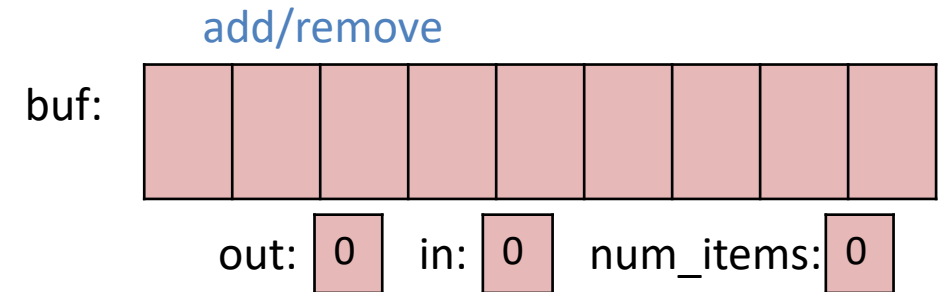
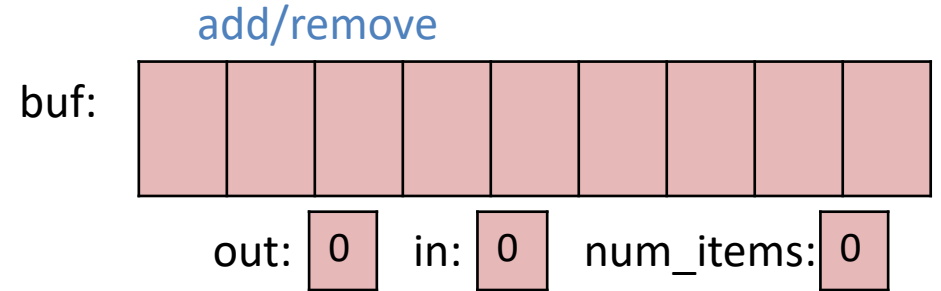
Consumer: Read 4



# Example: Producer/Consumer

Producer: 20,22,24,26

Consumer: Read 4

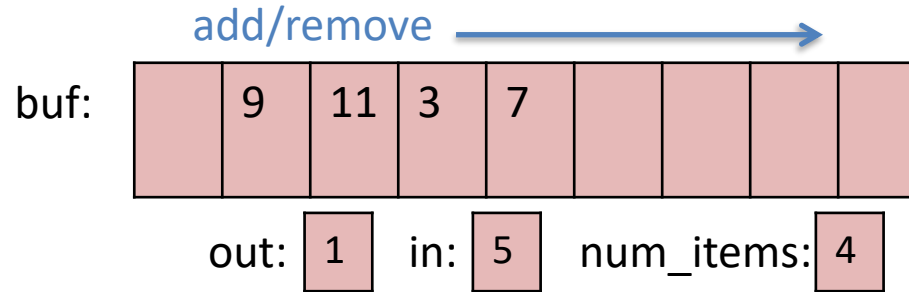




# Producer/Consumer Synchronization?

Circular Queue Buffer: add to one end (in), remove from other (out)

```
int buf[N];  
int in, out;  
int num_items;
```



Assume Producers & Consumers forever produce & consume

Q: Where is Synchronization Needed in Producer & Consumer?

Producer:

Consumer:

# Producer/Consumer Synchronization?

## Producer:

- Needs to wait if there is no space to put a new item in the buffer (**Scheduling**)
- Needs to wait to have mutually exclusive access to shared state associated with the buffer (**Atomic**):
  - Size of the buffer (num\_items)
  - Next spot to insert into (in)

## Consumer:

- Needs to wait if there is nothing in the buffer to consume (**Scheduling**)
- Needs to wait to have mutually exclusive access to shared state associated with the buffer (**Atomic**):
  - Size of the buffer (num\_items)
  - Next spot to remove from (out)

Assume: Producer & Consumer thread execute forever

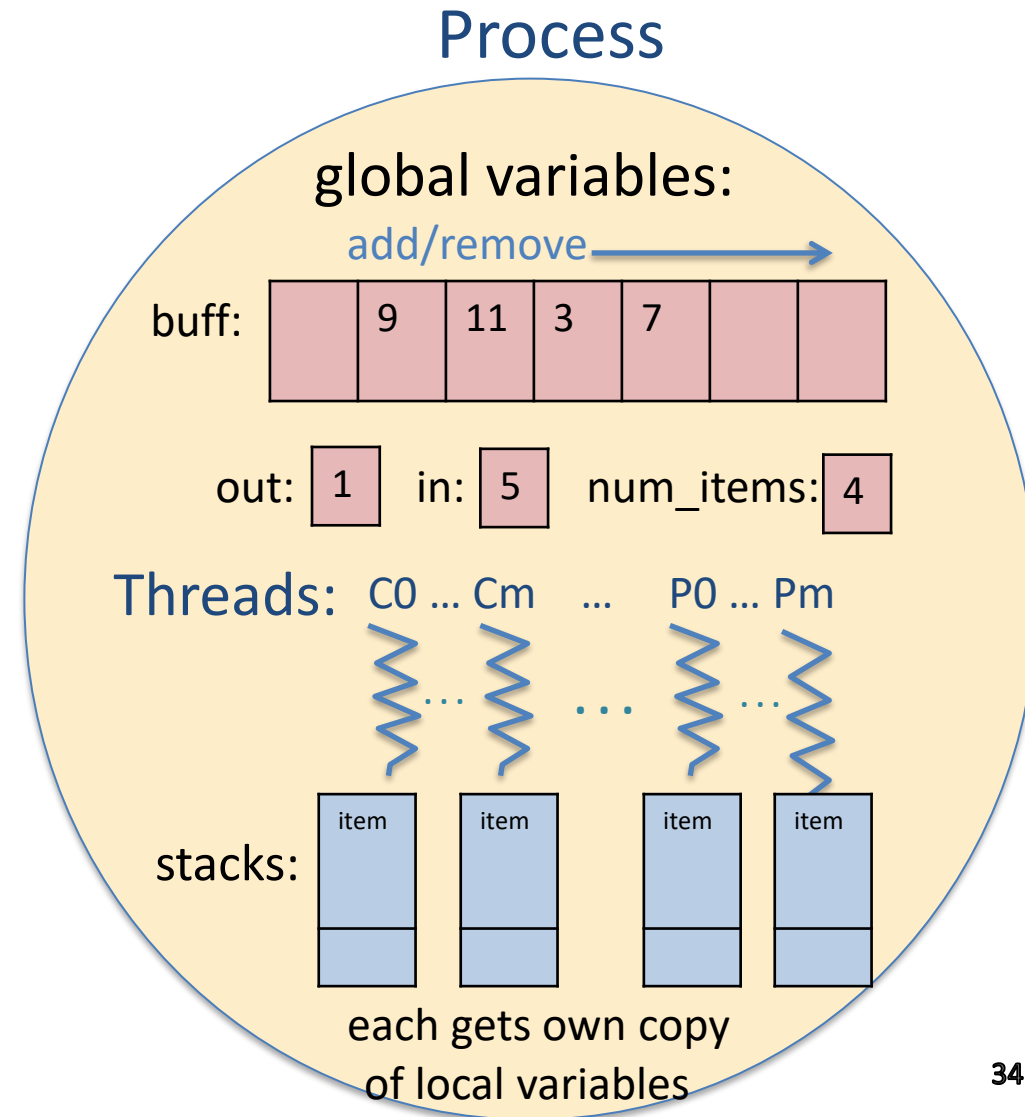
Circular Queue Buffer: add to one end, remove from another

# Producer/Consumer

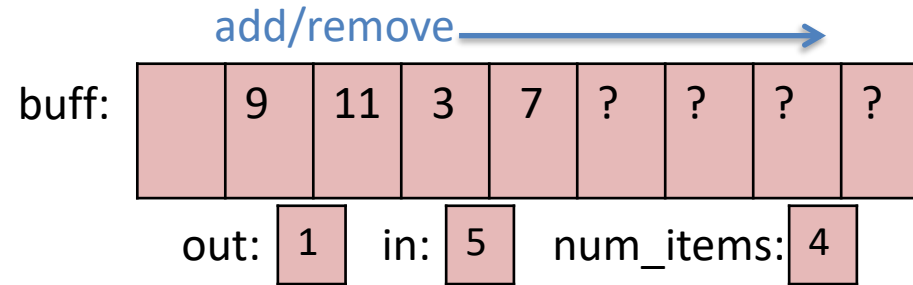
```
int  num_items=0, in=0, out=0, buff[N];
```

```
void producer_threads() {  
    int item;  
    while(1) {  
        item = produce_item();  
        //add to queue  
        buff[in] = item;  
        in = (in+1)%N;  
        num_items++;  
    }  
}
```

```
void consumer_threads() {  
    int item;  
    while(1) {  
        //remove from queue  
        item = buff[out];  
        out = (out+1)%N;  
        num_items--;  
        consume_item(item);  
    }  
}
```



# Producer/Consumer Solution



// Global Variables:

```
pthread_mutex_t mux = PTHREAD_MUTEX_INITIALIZER;  
int num_items=0, in=0, out=0, buff[N];
```

## Producer Threads:

```
int item;  
while(1) {  
    item = produce_item();  
    pthread_mutex_lock(&mux);  
    buff[in] = item;  
    in = (in+1)%N;  
    num_items++;  
    pthread_mutex_unlock(&mux);  
}
```

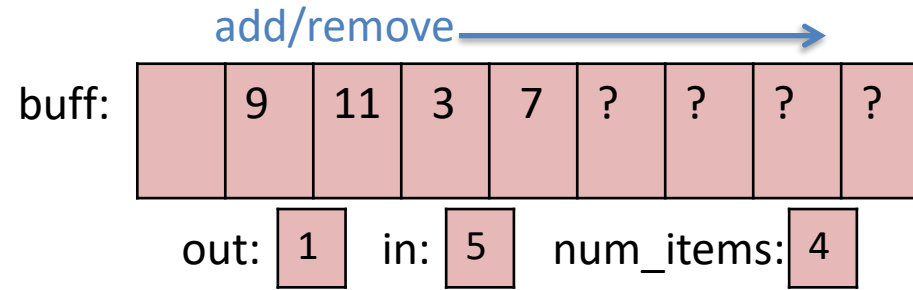
## Consumer Threads:

```
int item;  
while(1) {  
    pthread_mutex_lock(&mux);  
    item = buff[out];  
    out = (out+1)%N;  
    num_items--;  
    pthread_mutex_unlock(&mux);  
    consume_item(item);  
}
```

Need Atomic Access to buff state: num\_items, buff, in, out

- Want num\_items++ and num\_items-- to be atomic
- Don't want multiple consumer threads simultaneously using or updating out
- Don't want multiple producer threads simultaneously using or updating in

# Producer/Consumer Solution



// Global Variables:

```
pthread_mutex_t mux = PTHREAD_MUTEX_INITIALIZER;  
int num_items=0, in=0, out=0, buff[N];
```

## Producer Threads:

```
int item;  
while(1) {  
    item = produce_item();  
    pthread_mutex_lock(&mux);  
    buff[in] = item;  
    in = (in+1)%N;  
    num_items++;  
    pthread_mutex_unlock(&mux);  
}
```

## Consumer Threads:

```
int item;  
while(1) {  
    pthread_mutex_lock(&mux);  
    item = buff[out];  
    out = (out+1)%N;  
    num_items--;  
    pthread_mutex_unlock(&mux);  
    consume_item(item);  
}
```

## Still need some Scheduling Type of Synchronization:

- Consumer threads must wait when there are no items in buff to consume
- Producer threads must wait when there is no space in buff to put item

# Producer/Consumer Solution

// Global Variables:

```
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mux = PTHREAD_MUTEX_INITIALIZER;
int num_items=0, in=0, out=0, buff[N];
```

## Producer Threads:

```
int item;
while(1) {
    item = produce_item();
    pthread_mutex_lock(&mux);
    while(num_items >= N) {
        pthread_cond_wait(&full,
                          &mux);
    }
    buff[in] = item;
    in = (in+1)%N;
    num_items++;
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&mux);
}
```

## Consumer Threads:

```
int item;
while(1) {
    pthread_mutex_lock(&mux);
    while(num_items == 0) {
        pthread_cond_wait(&empty,
                          &mux);
    }
    item = buff[out];
    out = (out+1)%N;
    num_items--;
    pthread_cond_signal(&full);
    pthread_mutex_unlock(&mux);
    consume_item(item);
}
```

# Synchronization Can be Tricky

- Race Condition: missing synchronization

```
if(num_items > 0) {  
    pthread_mutex_lock(&mutex);  
    item = buff[out];  
    out = (out+1)%N;  
    num_items--;  
    pthread_mutex_unlock(&mutex);  
}  
...  
if(num_items > 0) {  
    pthread_mutex_lock(&mutex);  
    item = buff[out];  
    out = (out+1)%N;  
    num_items--;  
    pthread_mutex_unlock(&mutex);  
}
```

Q: Where is the Race Condition?

# Race Condition:

Reading value of `num_items` is NOT Atomic

```
if(num_items > 0) { ←  
    pthread_mutex_lock(&mutex);  
    item = buff[out];  
    out = (out+1)%N;  
    num_items--;  
    pthread_mutex_unlock(&mutex);  
}
```

```
...  
if(num_items > 0) { ←  
    pthread_mutex_lock(&mutex);  
    item = buff[out];  
    out = (out+1)%N;  
    num_items--;  
    pthread_mutex_unlock(&mutex);  
}
```

num\_items: 1



Ci



Ck

- (1) threads both read 1 simultaneously, and evaluate  $(1 > 0)$  is TRUE
- (2) One gets mutex first, gets valid item from buff
- (3) Other gets mutex second, gets garbage item from buff



# Synchronization Can be Tricky

- Deadlock: tid(s) blocked waiting forever
  - bad ordering of concurrent synch calls:

Tid 1:

```
pthread_mutex_lock(&mux1);  
pthread_mutex_lock(&mux2);
```

Tid 2:

```
pthread_mutex_lock(&mux2);  
Pthread_mutex_lock(&mux1);
```

- Forgetting to unlock (must be on every path):

```
pthread_mutex_lock(&mux1);  
if(some bool expr) {  
    // do some atomic stuff  
    pthread_mutex_unlock(&mux1);  
}
```

If cond isn't true, then no call to unlock mutex -> next call to lock will block forever

# Dining Philosophers

5 Philosophers

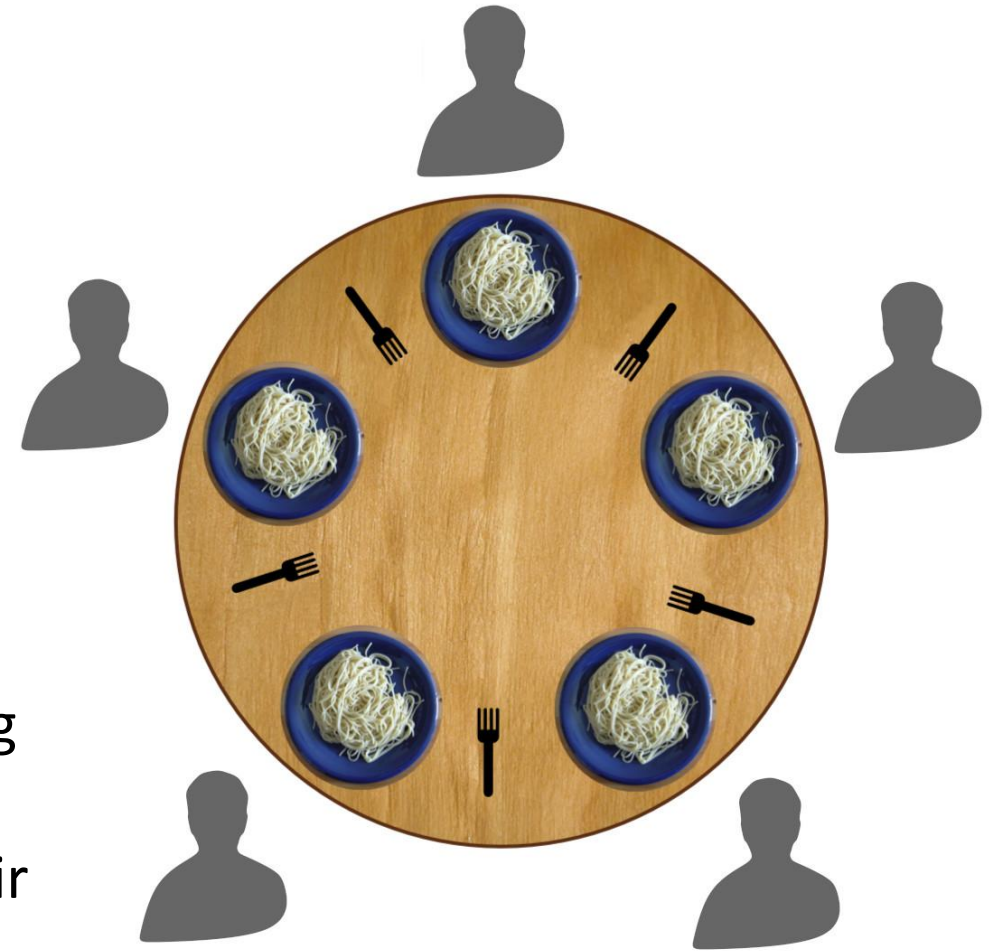
5 plates of spaghetti

5 forks

Philosophers alternate between eating and thinking

A philosopher can only eat if they can use both their left and right forks

Simulate the philosophers using a thread for each philosopher, such that no philosopher starves!



# Dining Philosophers: Can this deadlock?

```
printf("%d is hungry\n", data->id);
```

```
int leftFork = data->id;
```

```
int rightFork = (data->id + 1) % N;
```

```
pthread_mutex_lock(&mutex[rightFork]);
```

```
pthread_mutex_lock(&mutex[leftFork]);
```

```
printf("%d is eating\n", data->id);
```

```
sleep(duration);
```

```
pthread_mutex_unlock(&mutex[rightFork]);
```

```
pthread_mutex_unlock(&mutex[leftFork]);
```

```
pthread_mutex_t mutex[N]
```

```
struct thread_data {
```

```
    int id;
```

```
};
```

# Dining Philosophers: Can this deadlock?

```
printf("%d is hungry\n", data->id);
```

```
int leftFork = data->id;
```

```
int rightFork = (data->id + 1) % N;
```

```
pthread_mutex_lock(&mutex[leftFork]);
```

```
pthread_mutex_lock(&mutex[rightFork]);
```

```
printf("%d is eating\n", data->id);
```

```
sleep(duration);
```

```
pthread_mutex_unlock(&mutex[rightFork]);
```

```
pthread_mutex_unlock(&mutex[leftFork]);
```

```
pthread_mutex_t mutex[N]
```

```
struct thread_data {
```

```
    int id;
```

```
};
```

# Dining Philosophers: Can this deadlock?

```
printf("%d is hungry\n", data->id);
```

```
int fork1 = (data->id); // left
```

```
int fork2 = (data->id + 1) % N; // right
```

```
if (fork1 > fork2) {
```

```
    int tmp = fork2;
```

```
    fork2 = fork1;
```

```
    fork1 = tmp;
```

```
}
```

```
pthread_mutex_lock(&mutex[fork1]);
```

```
pthread_mutex_lock(&mutex[fork2]);
```

```
printf("%d is eating\n", data->id);
```

```
sleep(duration);
```

```
pthread_mutex_unlock(&mutex[fork2]);
```

```
pthread_mutex_unlock(&mutex[fork1]);
```

# Thread safety

A function is **thread safe** if it can be used by multiple threads with reliable behavior

Example: Most C library function **are not** thread safe, such as strtok

Example: When we do not properly protect critical sections of code, our thread routines are not thread safe either!

# Thread safety: Demo

```
void countElemsStr(int *counts, char *input_str) {
    int val, i;
    char *token;
    token = strtok(input_str, " ");
    while (token != NULL) {
        val = atoi(token);
        counts[val] = counts[val] + 1;
        token = strtok(NULL, " ");
    }
}

int main( int argc, char **argv ) {
    //lines omitted for brevity, but gets user defined length of string

    char *inputString = calloc(length * 2, sizeof(char));
    fillString(inputString, length * 2);
    countElemsStr(counts, inputString);

    return 0;
}
```

# Thread safety: Demo

```
void *countElemsStr(void *args) {
    struct t_arg *myargs = (struct t_arg *)args;
    // get thread parameters from myargs (omitted for brevity)

    //tokenize values
    char* saveptr;
    token = strtok_r(input_str, " ", &saveptr);
    while (token != NULL) {
        val = atoi(token); //convert to an int
        local_counts[val] = local_counts[val] + 1; //update associated counts
        token = strtok_r(NULL, " ", &saveptr);
    }

    pthread_mutex_lock(&mutex);
    for (i = 0; i < MAX; i++) {
        counts[i] += local_counts[i];
    }
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```