Agenda: Synchronization

Challenges of process/thread coordination

Terminology

Methods of mutual exclusion

Classical problems

- Producer/Consumer problem
- Dining philosophers

Types of process/thread coordination

Passing information from one process to another (pipes)

Making sure processes do not “get in each other’s way” when sharing access to data (mutexes, semaphores)

Sequencing actions when processes have dependencies on each other (mutexes, semaphores, conditionals, barrier)

consumer/producer problems

Demo: Shared variable problem

Main problem: operations of different processes/threads are interleaved in an unpredictable manner

Example: N threads increment a counter X times

During testing it is observed that the final count is less than NX

What happened?

Code Sketch

```
int count=0;

void interleave() {
  pthread_t th0, th1;
  pthread_create(&th0,0,test,0);
  pthread_create(&th1,0,test,0);
  pthread_join(th0,0);
  pthread_join(th1,0);
  printf("%d\n", count);
}
void test() {
  for(int j=0;j<MAX;j++) count=count+1;
}
```

What value should count have after running this program?

Demo: threads-count.c

$ **./threads-count**
hello I'm thread 1
hello I'm thread 2
hello I'm thread 3
hello I'm thread 0
goodbye I'm thread 1
goodbye I'm thread 3
goodbye I'm thread 2
goodbye I'm thread 0
count = 196274
count time is 0.002927

$ **./threads-count**
hello I'm thread 0
hello I'm thread 1
hello I'm thread 2
hello I'm thread 3
goodbye I'm thread 2
goodbye I'm thread 1
goodbye I'm thread 0
goodbye I'm thread 3
count = 272560
count time is 0.004197

We get a different (wrong) answer every time.

Analysis of the problem

Add is not **atomic**

e.g. it takes multiple instructions to complete the operation

These commands might be scheduled in *any* order

```
P1. MOVE V, r0          Q1. MOVE V, r1
P2. INCR r0             Q2. INCR r1
P3. MOVE r0, V          Q3. MOVE r1, V
```

The interleaving P1, Q1, P2, Q2, P3, Q3 increments V only by 1

Compare the computation of v

Suppose v = 2

```
P1. MOVE V, r0
Q1. MOVE V, r1
P2. INCR r0
Q2. INCR r1
P3. MOVE r0, V
Q3. MOVE r1, V
```

VS.

```
P1. MOVE V, r0
P2. INCR r0
P3. MOVE r0, V
Q1. MOVE V, r1
Q2. INCR r1
Q3. MOVE r1, V
```

Exercise: Push and Pop example

```
struct stacknode {
   int data;
   struct stacknode *nextptr;
};
typedef struct stacknode STACKNODE;

void push (STACKNODE **topptr, int info) {
   STACKNODE *newptr;
   newptr = malloc (sizeof (STACKNODE));
   newptr->data = info;          /* Push 1 */
   newptr->nextptr = *topptr; /* Push 2 */
   *topptr = newptr;             /* Push 3 */
}
```

Pop

```
int pop (STACKNODE **topptr) {
    STACKNODE *tempptr;
    int popvalue;
    tempptr = *topptr;                  /* Pop 1 */
    popvalue = (*topptr)->data;         /* Pop 2 */
    *topptr = (*topptr)->nextptr; /* Pop 3 */
    free(tempptr);
    return popvalue;
}
```

Question: Is it possible to find an interleaved execution of Push 1, Push 2, …, Pop 3 such that the resulting data structure becomes inconsistent?

Exercise: Push and pop example

```
// push
newptr->data = info;         /* Push 1 */
newptr->nextptr = *topptr; /* Push 2 */
*topptr = newptr;            /* Push 3 */
```

```
// pop
tempptr = *topptr;              /* Pop 1 */
popvalue = (*topptr)->data;   /* Pop 2 */
*topptr = (*topptr)->nextptr; /* Pop 3 */
```

Question:

Is it possible to find an interleaved execution of Push 1, Push 2, …, Pop 3 such that the resulting data structure becomes inconsistent?

Visualizing push/pop race condition

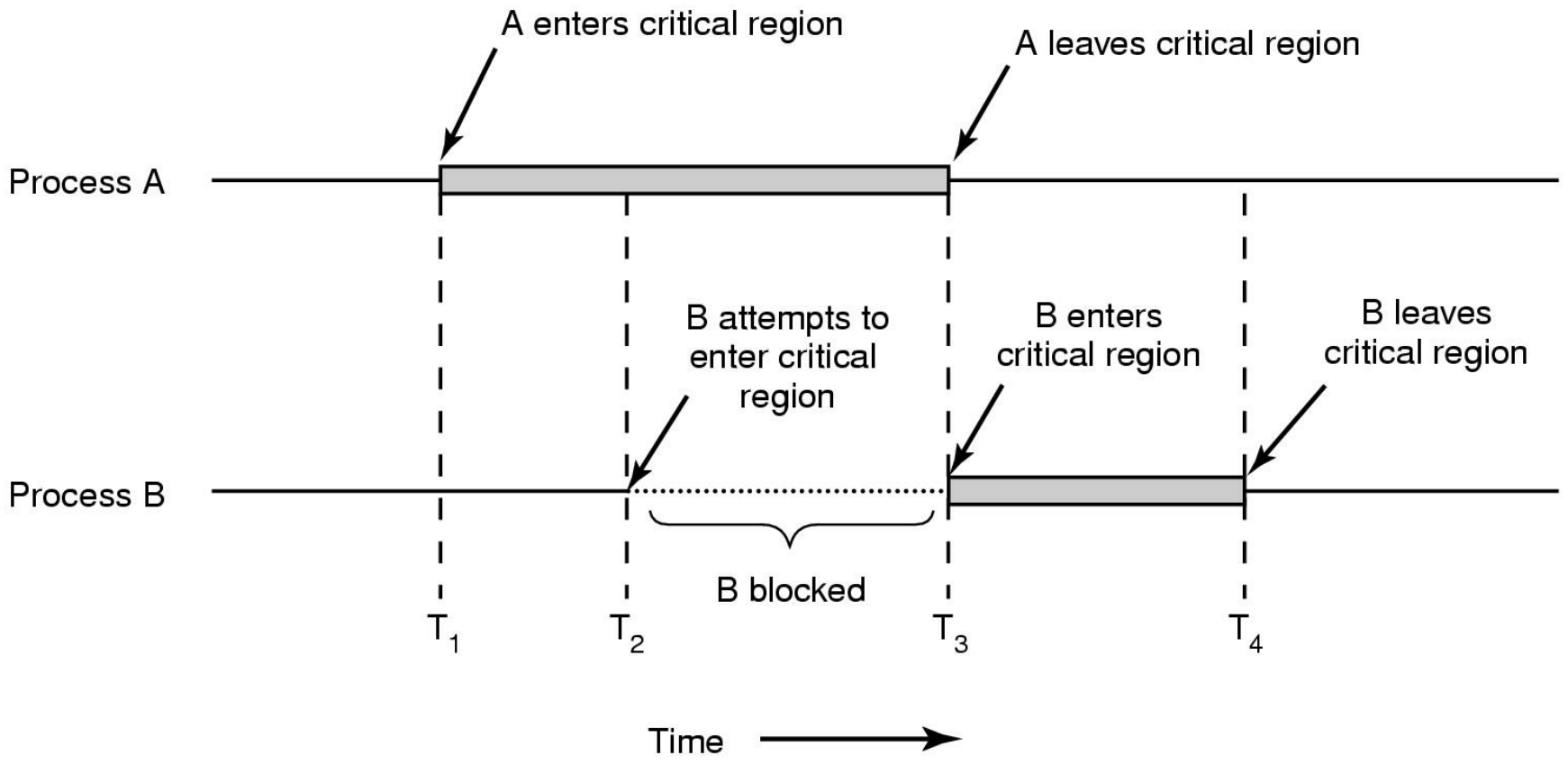
Terminology

When the behavior of code varies based on how it is scheduled, we have a **race condition.**

Portions of code that can lead to race conditions are called **critical sections.**

To avoid **race conditions**, we can protect **critical sections** of code using mechanisms that enforce **mutual exclusion**. **Mutual exclusion** ensures that only one thread/process executes in the critical section at the same time.

Critical Regions



Terminology

Deadlock occurs when coordinating processes are stopped waiting for resources that will never become available

Livelock occurs when scheduling prevents processes from making progress, yet they continue to run

Starvation occurs when scheduling omits a process from ever accessing their critical section

Fairness occurs when all processes have roughly equal access to their critical sections

Mutual Exclusion

Often abbreviated as **mutex**

Idea: protect critical sections of code using “sentinel” mechanisms that ensure only one process can enter at a time.

Analogy: A bouncer for the VIP room

Demo: Counting with mutual exclusion

```
// code sketch
void test() {
  for(int j=0;j<MAX;j++) {
     mutual_exclusion_lock();
     count=count+1; // critical section
     mutual_exclusion_unlock();
  }
}
```

Now, we get the right answer every time.

```
$ ./threads-count-mutex
hello I'm thread 0 with pthread_id 140164471678528
hello I'm thread 1 with pthread_id 140164463285824
hello I'm thread 2 with pthread_id 140164454893120
hello I'm thread 3 with pthread_id 140164378654272
goodbye I'm thread 3
goodbye I'm thread 2
goodbye I'm thread 0
goodbye I'm thread 1
count = 400000
count time is 0.035381
$ ./threads-count-mutex
hello I'm thread 2 with pthread_id 140171899594304
hello I'm thread 3 with pthread_id 140171891201600
hello I'm thread 1 with pthread_id 140171907987008
hello I'm thread 0 with pthread_id 140171916379712
goodbye I'm thread 1
goodbye I'm thread 3
goodbye I'm thread 0
goodbye I'm thread 2
count = 400000
count time is 0.033861
```

Requirements for solutions to Mutual Exclusion Problem

1. **Safety:** No two processes should be simultaneously in their critical regions
2. **Generality:** No assumptions should be made about speeds or numbers of CPUs (i.e., it should work in the worst case scenario)
3. **Absence of deadlocks:** Should not reach a state where each process is waiting for the other, and nobody gets to enter
4. **Bounded liveness (or fairness):** If a process wants to enter a critical section then it should eventually get a chance

Synchronization Overview

Idealized Problems
- 5. Producer-Consumer
- 6. Dining Philosophers

High-level Synchronization Primitives
- Monitors (Hoare, Brinch-Hansen)
- Synchronized method in Java

OS-level support (mutual exclusion and synchronization)
- 4. Special variables: Semaphores, Mutexes, etc
- Message passing primitives (send and receive)

Low-level (for mutual exclusion)
- 1. Interrupt disabling
- 2. Shared variable solutions (Using atomic read/write instructions)
- 3. Hardware instructions (Test-and-set, Compare-and Swap…)

1. Low-level solution: Disable interrupts

```
process A                    process B
  ...                        ...
  disable interrupts         disable interrupts
    CS                         CS
  enable interrupts          enable interrupts
```

- Prevents context-switch during execution of CS
- This is sometimes necessary (to prevent further interrupts during interrupt handling)
- Not a good solution for user programs (too powerful and not flexible)

2. Shared Variable Solutions: General Skeleton

Two processes with shared variables
Assumption: Reads and Writes are atomic
Each process P0 and P1 executes

```
/* Initialization */
while (TRUE) {
  /* entry code */
    CS() /* critical section */
  /* exit code */
   Non_CS()/* non-critical section */
}
```

No assumption about how often the critical section is accessed

Wrapper code

1st Attempt for Mutual Exclusion

```
Shared variable: turn :{0,1}
while (TRUE) {
  while (turn != 0);  /* busy waiting */
  CS();
  turn = 1; /* let the other enter */
  Non_CS();
}
```
P0

```
Shared variable: turn :{0,1}
while (TRUE) {
  while (turn != 1);  /* busy waiting */
  CS();
  turn = 0; /* let the other enter */
  Non_CS();
}
```
P1

turn==i means process Pi is allowed to enter
Initial value of turn doesn't matter

Problems:
1) Requires strict alternation
2) A process cannot enter its CS twice in succession

2nd Attempt for mutual exclusion

```
Shared variable: interested[i] : boolean, init FALSE
while (TRUE) {
  while (interested[1]); /* wait if P1 is trying */
  interested[0] = TRUE;  /* declare your entry */
  CS();
  interested[0] = FALSE; /* unblock P1 */
  Non_CS();
}
```

P0

```
Shared variable: interested[i] : boolean, init FALSE
while (TRUE) {
  while (interested[0]); /* wait if P0 is trying */
  interested[1] = TRUE;  /* declare your entry */
  CS();
  interested[1] = FALSE; /* unblock P0 */
  Non_CS();
}
```

P1

Does this ensure mutual exclusion of CS()?

Can the two processes become deadlocked?

3rd Attempt for mutual exclusion

```
Shared variable: interested[i] : boolean, init FALSE
while (TRUE) {
  interested[0] = TRUE;  /* declare entry first */
  while (interested[1]); /* wait if P1 is trying */
  CS();
  interested[0] = FALSE; /* unblock P1 */
  Non_CS();
}
```

P0

```
Shared variable: interested[i] : boolean, init FALSE
while (TRUE) {
  interested[1] = TRUE;  /* declare entry first */
  while (interested[0]); /* wait if P0 is trying */
  CS();
  interested[1] = FALSE; /* unblock P0 */
  Non_CS();
}
```

P1

Does this ensure mutual exclusion of CS()?

Can the two processes become deadlocked?

Peterson's Solution

```
Shared variables: interested[i] :boolean;
                  turn :{0,1}
interested[0] = FALSE;
while (TRUE) {
  interested[0] = TRUE;  /* declare interest */
  turn = 0; /* takes care of race condition */
  repeat if /* busy wait */
   (interested[1] == TRUE && turn == 0);
  CS();
  interested[0] = FALSE; /* unblock P1 */
  Non_CS();
}
```

P0

Peterson's Solution

```
Shared variables: interested[i] :boolean;
                  turn :{0,1}
interested[1] = FALSE;
while (TRUE) {
  interested[1] = TRUE;  /* declare interest */
  turn = 1; /* takes care of race condition */
  repeat if /* busy wait */
   (interested[0] == TRUE && turn == 1);
  CS();
  interested[1] = FALSE; /* unblock P1 */
  Non_CS();
}
```

P1

The case for P1 is symmetric to P0.

Left, circle the modifications from P0

Proof of mutual exclusion

```
Shared variables: interested[i] :boolean;
                  turn :{0,1}
interested[0] = FALSE;
while (TRUE) {
  interested[0] = TRUE;  /* declare interest */
  turn = 0; /* takes care of race condition */
  repeat if /* busy wait */
   (interested[1] == TRUE && turn == 0);
  CS();
  interested[0] = FALSE; /* unblock P1 */
  Non_CS();
}
```

P0

Proof of Mutual Exclusion

To prove: P0 and P1 can never be simultaneously in CS

Observation: Once P0 sets **`interested[0]`**, it stays true until P0 leaves the critical section (same for P1)

Proof by contradiction

- Suppose at time *t* both P0 and P1 are in CS
- Let *t0* be the time of the most recent execution of the assignment **`turn = 0`** by P0
- Let *t1* be the times of the most recent executions of the assignments **`turn = 1`** by P1
- Suppose *t0* < *t1*
- During the period *t0* to *t*, **`interested[0]`** equals **`TRUE`**
- During the period from *t1* to *t*, **`turn`** equals to 1
- Hence, during the period *t1* to *t*, P0 is blocked from entering its CS; a contradiction!

3. Low-level solutions: Hardware instructions

Idea: Perform read/write with a single assembly instruction

Test-and-set-lock, e.g.

`TSL X, L` X: register, **`L`** : memory loc (bit)

`L`'s content are loaded into **`X`**, and **`L`** is set to 1

x86 assembly

```
q_atomic_increment:
   movl 4(%esp), %ecx
   lock
   incl (%ecx)
   mov $0,%eax
   setne %al
   ret
```

4. OS solutions for synchronization

OS can block processes, so no more need to busy wait

Mutex
- enforces mutual exclusion of critical sections with blocking

Barrier
- forces *all* threads to reach a common point in execution before re-enabling concurrent execution

Condition variables
- force a thread to block until a particular condition is reached

Semaphores
- Used to count how many resources are available
- **Counting semaphores** take on a value from 0 to num_resources

pthread synchronization APIs

Mutex : mutually exclusive access

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
pthread_mutex_[un]lock(&mutex);
```

Condition variables: block if condition is true

```
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
pthread_cond_wait(&cond, &mux);
pthread_cond_signal(&cond);
```

Barrier mutex: block until all threads have called wait

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

Example: pthread mutex

```
static unsigned long long count = 0;
pthread_mutex_t mutex;

void *thread_count(void* args) {
  int myid, i;
  myid = *((int*) args);
  printf("hello I'm thread %d with pthread_id %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;
}
```

```
int main(int argc, char *argv[]) {
  pthread_t threads[4];
  int ids[4];
  int i = 0;

  pthread_mutex_init(&mutex, NULL);
  for (i = 0; i < 4; i++) {
    ids[i] = i;
    pthread_create(&threads[i], NULL, thread_count, &ids[i]);
  }
  for (i = 0; i < 4; i++) {
    pthread_join(threads[i], NULL);
  }
  printf("count = %llu\n", count);
  pthread_mutex_destroy(&mutex);
  return 0;
}
```

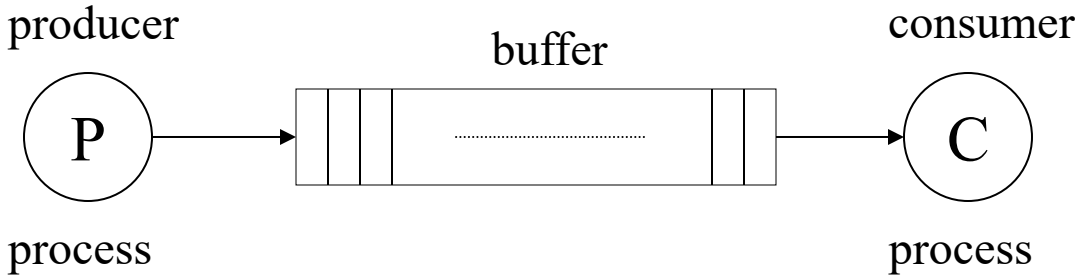
Classical Problems

So far, we have talked about protecting critical sections of code with mutual exclusion

More complicated cases need to coordinate actions of multiple threads, e.g. SCHEDULING types of synchronization

- Need shared state that can be used to test if a thread needs to wait or signal another
- Need multiple mutex locks & maybe condition variables
- May be able to use barrier synchronization instead of mutex and condition variables for some types

5. The Producer/Consumer Problem



A shared fixed-size buffer

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

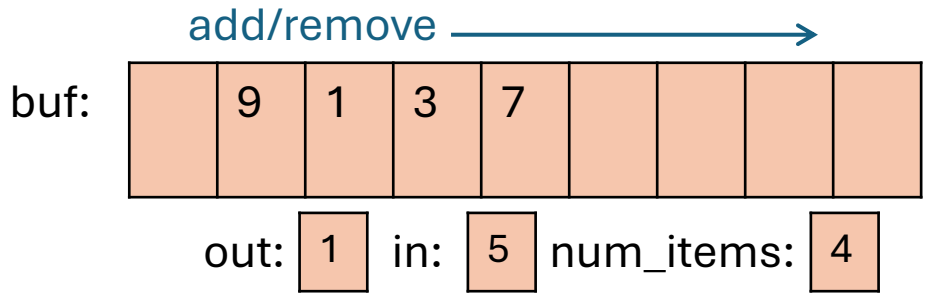
CPU queue of ready threads/processes

Network messaging

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?

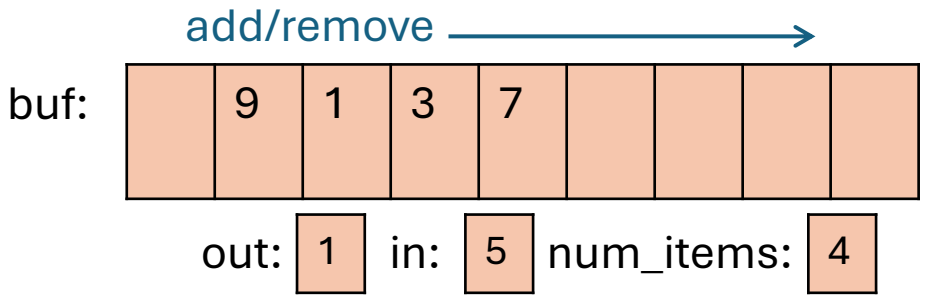
Producer:

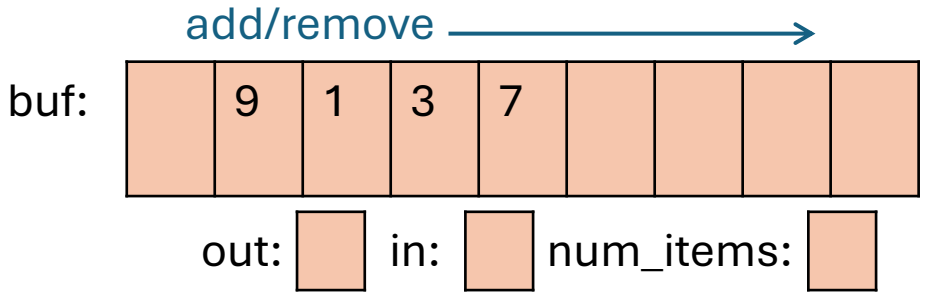
Consumer:

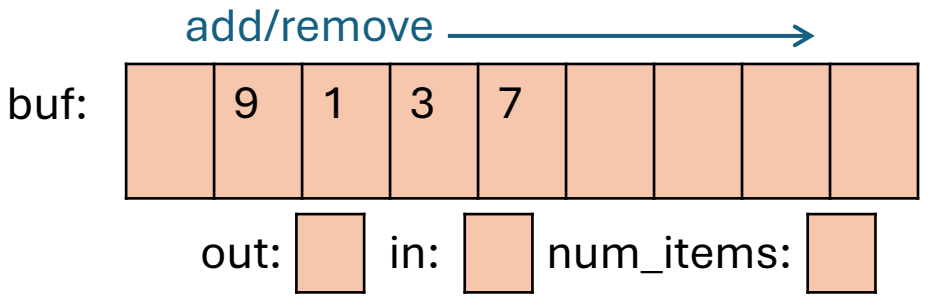
Aside: Circular buffers

1. Show how the buffer updates when we add the value 11

2. Show how the buffer updates when we remove the last two elements





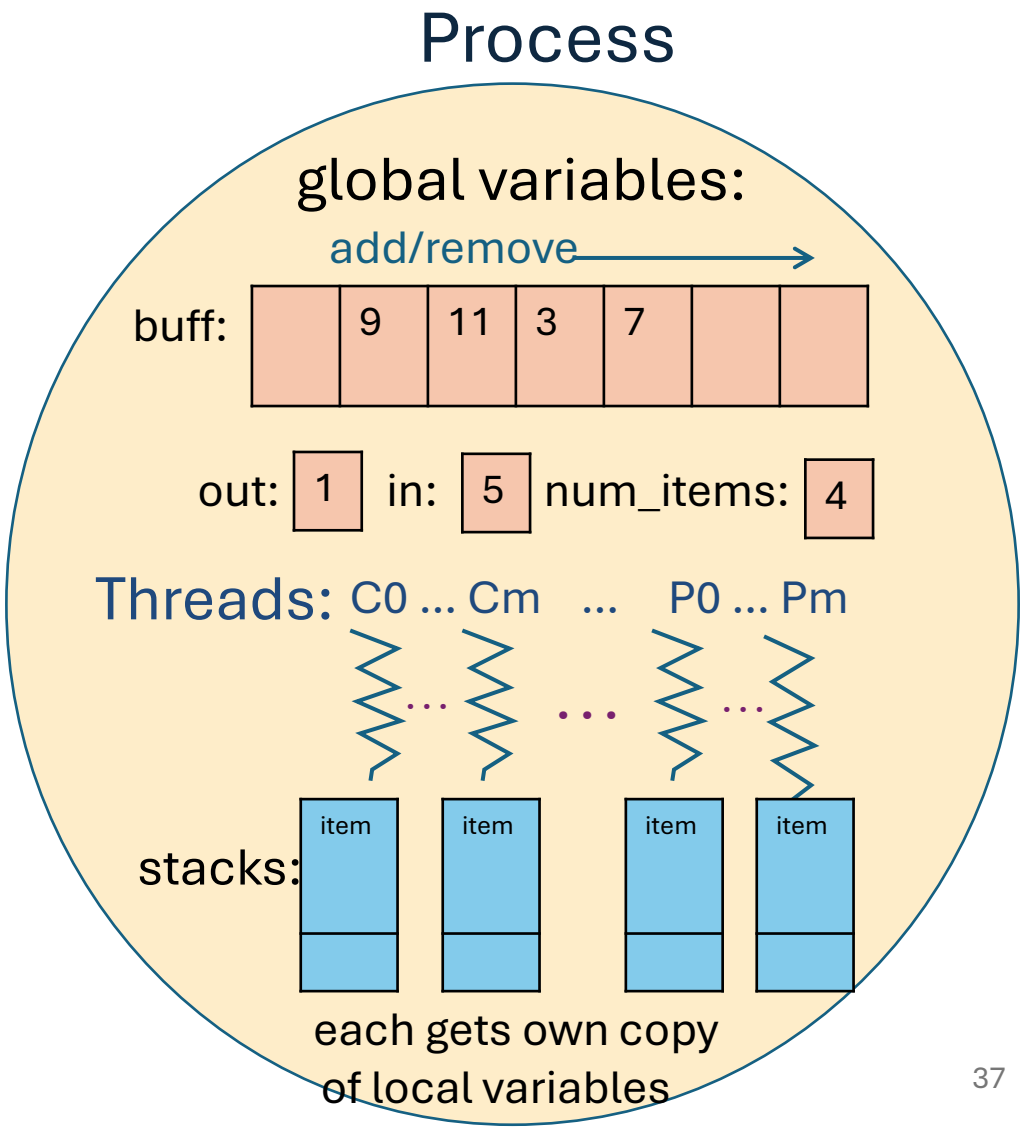


Producer/Consumer

Where are the critical sections?

```
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_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);
}
```

Exercise: Is the following code safe?

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

// Producer thread code
if(num_items < N) {
  pthread_mutex_lock(&mutex);
  buff[in] = item;
  in = (in+1)%N;
  num_items++;
  pthread_mutex_unlock(&mutex);
}
```

6. Dining Philosophers

N Philosophers

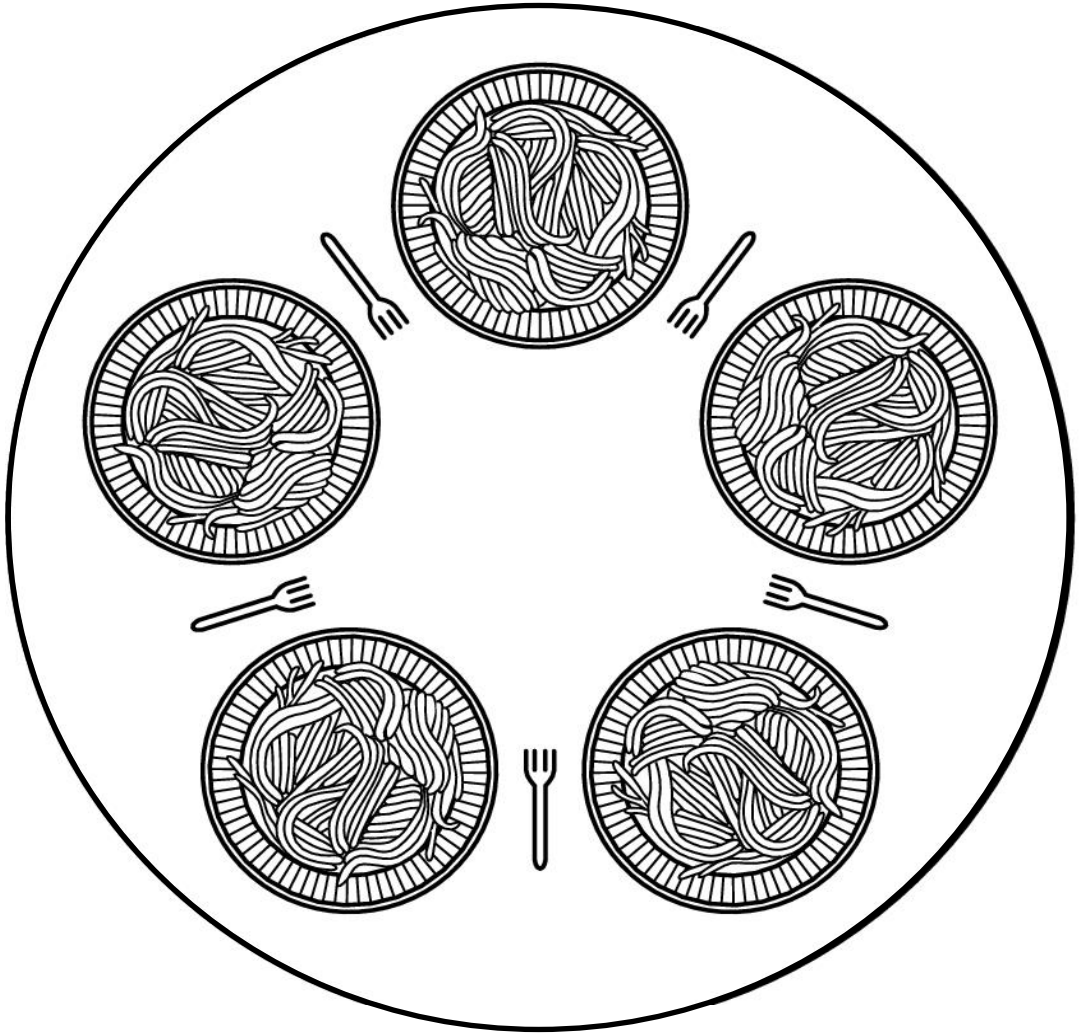
N plates of spaghetti

N 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!



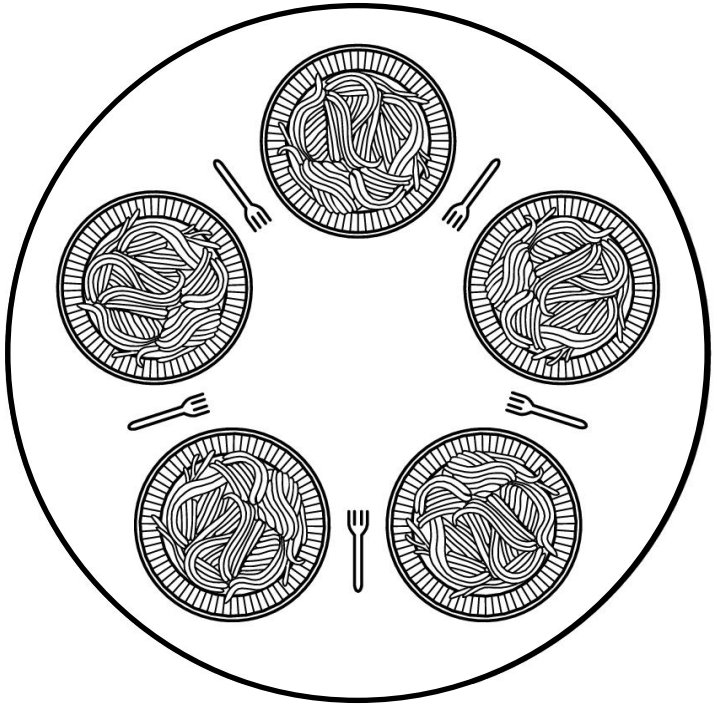
Dining Philosophers

Can this deadlock?

```
#define N 5

philosopher i
  repeat
    think();
    take_fork(i);
    take_fork((i+1)%N);
       eat()
    take_fork(i);
    take_fork((i+1)%N);

  forever
```

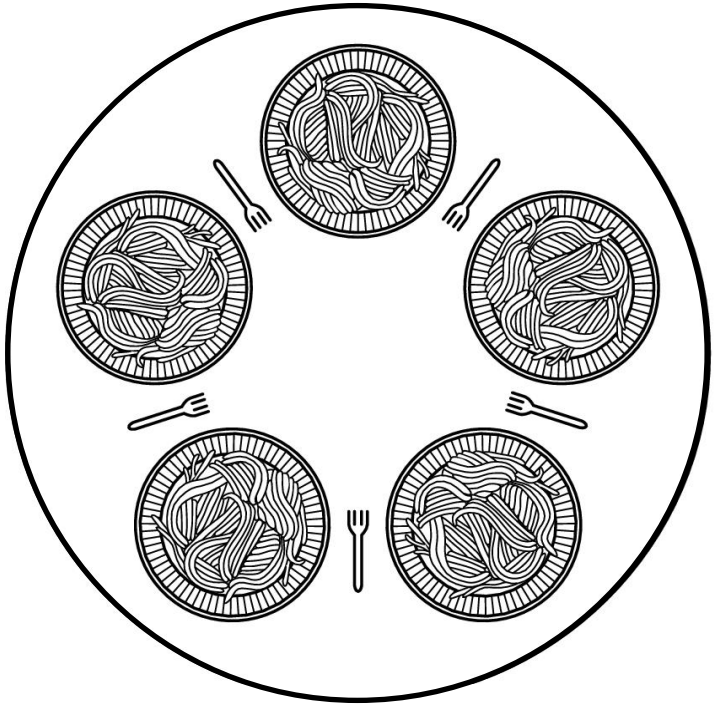


Dining Philosophers

Can this deadlock?

```
#define N 5

philosopher i
  repeat
    mutex_lock();
    think();
    take_fork(i);
    take_fork((i+1)%N);
       eat()
    take_fork(i);
    take_fork((i+1)%N);
    mutex_unlock();
  forever
```



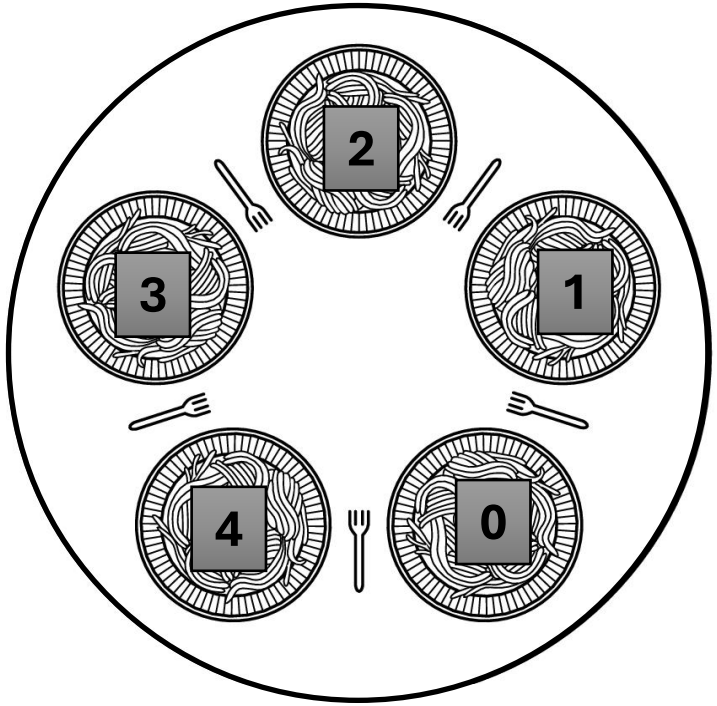
Dining Philosophers: Can this deadlock?

```
#define N 5

philosopher i
  repeat
    fork1 = i
    fork2 = (i+1)%N
    if (fork1 > fork2)
      swap(fork1, fork2)

    think();
    take_fork(fork1);
    take_fork(fork2);
       eat()
    take_fork(fork1);
    take_fork(fork2);

  forever
```



Summary of synchronization

- Two key issues:
  - Mutual exclusion while accessing shared data
  - Synchronization (sleep/wake-up) to avoid busy waiting
- Solutions at many levels
  - Low-level (Peterson's, test-and-set)
  - System calls (semaphores, message passing)
  - Programming language level (monitors)
- Solutions to classical problems
  - Correct operation in worst-case
  - As much concurrency as possible
  - Avoid busy-waiting
  - Avoid deadlocks