Synchronization

The Race Condition Problem

Critical Sections

Mutex Locks

Semaphores

Condition Variables

Barriers

Producer-Consumer Problem with Bounded Buffer

Bibliography

- [Pacheco]: Peter Pacheco, Matthew Malensek, Introduction to Parallel Programming, 2nd Edition, Morgan Kaufmann Publisher, March 2020, Chapter 4.4, 4.6, 4.7
- [Illinois]: System Programming @ University of Illinois, Chapter 7 https://cs341.cs.illinois.edu/coursebook/index.html
- [UPB]: Curs Algoritmi Paraleli si Distribuiti, Universitatea Politehnica Bucuresti, https://mobylab.docs.crescdi.pub.ro/docs/parallelAndDistributed/introduction/

Problems when multiple threads update a memory location

- **Simple example: shared counter:**
  - `Shared memory (global variable):` **`int x=0`**
  - `Thread 1:` **`x++`**
  - `Thread 2:` **`x++`**
  - **`Final value of x: uncertain!`**

- Cause of problem: incrementing a variable is not an ***atomic*** operation! Its implementation is by a sequence of atomic machine operations:
  1. Load value of variable x from memory into register
  2. Increment register
  3. Store register into memory in location of variable x

- If we have 2 threads, ***the result depends on how the atomic machine operations happen to be interleaved***!

Thinking Question

For the shared counter example:

- Which is *the minimum possible value of x* at the end, when both threads have finished execution?
- Which is *the maximum possible value of x* at the end, when both threads have finished execution?

Justify/explain your answer.

Possible good result…

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
`x = r1`		1		1
	`r2 = x`		1	1
	`r2 = r2+1`		2	1
	`x = r2`		2	2

Possible wrong result!

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
	`r2 = x`	1	0	0
	`r2 = r2+1`	1	1	0
`x = r1`		1	1	1
	`x = r2`	1	1	1

Race Conditions and Critical Sections

- **Race conditions**:
  - When multiple threads attempt to update a shared variable the result may be unpredictable.
  - When multiple threads attempt to access a shared variable, and at least one of the accesses is an update, the accesses can result in an error
- A **critical section**: a block of code that updates a shared variable that should only be updated by one thread at time.
- To avoid race conditions, threads need **mutual exclusive** access to critical sections
  - once one of the threads starts executing the critical section, it finishes executing it *before* any other thread enters the critical section.
  - only one thread can execute code from the critical section at one moment (all other threads must be outside the critical section)

Implementing Critical Sections

- There are different ways to implement mutual exclusion for critical sections
- Threaded APIs provide ***mutex-locks*** (mutual exclusion locks) as the most basic support for implementing **critical sections** and **atomic operations**

Critical section

Mutex lock

Mutex locks

- A Mutex-lock (Mutex) has two states: locked and unlocked.
- A Mutex is associated with a piece of code that manipulates shared data (a critical section).
- At any point of time, only one thread can lock a Mutex.
- To access the shared data, a thread must first try to acquire the Mutex. If the Mutex is already locked, the thread trying to acquire the lock is blocked. This is because a locked Mutex implies that there is another thread currently in the critical section
- When a thread leaves a critical section, it must unlock the Mutex so that other threads can enter the critical section.
- A Mutex must be initialized to the unlocked state at the beginning of the program.

Mutex

- The Pthreads standard includes a special type for mutex locks: pthread_mutex_t.
- A variable of type pthread_mutex_t needs to be initialized by the system before it's used:

```
int pthread_mutex_init (
      pthread_mutex_t * mutex_p /* out */ ,
     const pthread_mutexattr_t* attr_p /* in */ ) ;
```

- When a program finishes using a mutex:

```
int pthread_mutex_destroy (
      pthread_mutex_t * mutex_p /* in / out */ ) ;
```

All synchronization mechanisms in pthreads have their special defined types.
All of them have similar init and destroy functions – will not repeat these in lecture

Mutex lock and unlock operations

- To lock the mutex and gain exclusive access to the critical section, a thread calls:

```
int pthread_mutex_lock(pthread_mutex_t * mutex_p);
```

- A call to this function attempts a lock on the mutex. If the mutex is already locked, the calling thread blocks; otherwise the mutex is locked and the calling thread returns.
- A successful return from the function returns a value 0. Other values indicate error conditions.

- When a thread is finished executing the code in a critical section, it should call:

```
int pthread_mutex_unlock(pthread_mutex_t * mutex_p);
```

- On calling this function the lock is relinquished and one of the blocked threads is scheduled to enter the critical section.
- Only the thread that locked the mutex can unlock it!

Mutex Example – Shared Counter

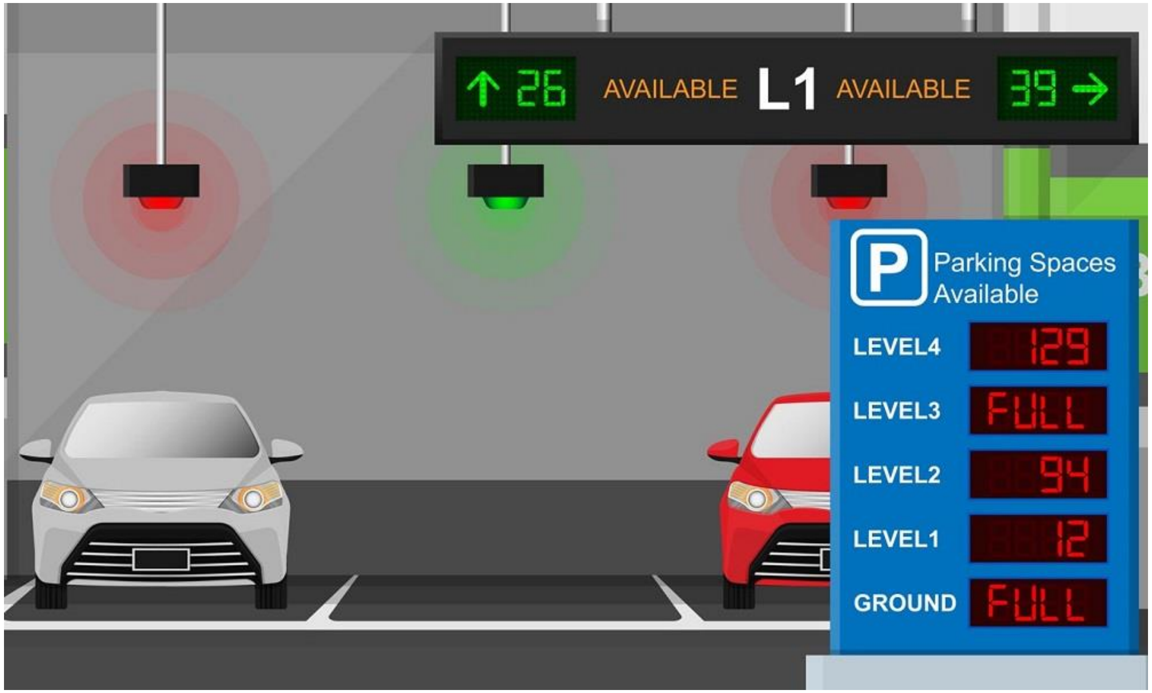
```
#define NUM_THREADS 2

/* shared variable */
int count = 0;

/* mutex to protect critical regions
when threads are updating count */
pthread_mutex_t count_mutex;

/* thread function */
void *inc_count(void *t)
{
  int i;
  int my_id = *(int *)t;
  pthread_mutex_lock(&count_mutex);
  count++;
  pthread_mutex_unlock(&count_mutex);
  return NULL;
}
```

Mutex Example (contd.)

```c
int main(int argc, char *argv[]) {

  pthread_t threads[NUM_THREADS];
  int ids[NUM_THREADS];

   pthread_mutex_init(&count_mutex, NULL);

  for (int i = 0; i < NUM_THREADS; i++) {
    ids[i] = i;
    pthread_create(&threads[i], NULL, inc_count, (void *)&ids[i]);
  }

  for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }

  printf("Final value of count = %d \n", count);

  pthread_mutex_destroy(&count_mutex);
}
```

Semaphores

- Semaphore: another synchronization primitive, ***similar to a generalization of a Mutex.***
- **A semaphore is initialized to some value.** That value represents the ***number of threads allowed inside the protected region***. A Mutex is similar to a binary semaphore (value initialized with 1)



Semaphores Operations

- **sem_wait** (similar to mutex_lock): lowers the value; If the value reaches zero and a sem_wait is called, the thread will be blocked until a sem_post is called.
- **sem_post (**similar to mutex_unlock**):** increases the value.
- Differences:
  - When using a Mutex, lock and unlock must be executed from **the same thread (only the thread that locked a mutex can unlock it)**
  - When using a semaphore, sem_wait and sem_post can be called from **different threads**! (a thread can block a semaphore and another thread can unblock it)
    - Mutex locks can be used only for critical regions
    - Semaphores can be used for critical regions and also for signaling
  - Semaphores can also be used for synchronization **between processes** (not only between threads)

Semaphore functions

Semaphores are not part of <pthread.h>

```
#include <semaphore.h>

int sem_init(
      sem_t*     semaphore_p   /* out */,
      int        shared        /* in  */,
      unsigned   initial_val   /* in  */);



int sem_destroy(sem_t*  semaphore_p  /* in/out */);
int sem_post(sem_t*     semaphore_p  /* in/out */);
int sem_wait(sem_t*     semaphore_p  /* in/out */);
```

Condition Variables

- Another synchronization problem (**producer–consumer synchronization**): ***a thread A can't proceed until another thread B has taken some action.***
  - *Thread A **waits** (is blocked) until it is **signaled (notified)** by thread B*

B

A

Condition Variables

- A **condition variable in pthreads** is a data object:
- pthread_cond_t
- A thread can suspend its execution until a certain event or *condition* occurs pthread_cond_wait
- When the event or condition occurs another thread can *signal* the thread to “wake up.” pthread_cond_signal
- Condition variables are not for mutual exclusion, they deal with another form of synchronization - signaling

Condition Variables

- Simple example:
- a set of threads increment a *shared variable x*
- another thread A must wait until the *shared variable x* reaches a certain value in order to start performing his task.
  - Without condition variables: thread A must continuously check the value of x (polling). Thread A is busy and running all the time.
  - With condition variables: thread A waits until it is signaled (notified) by one of the incrementer threads that x reached the value

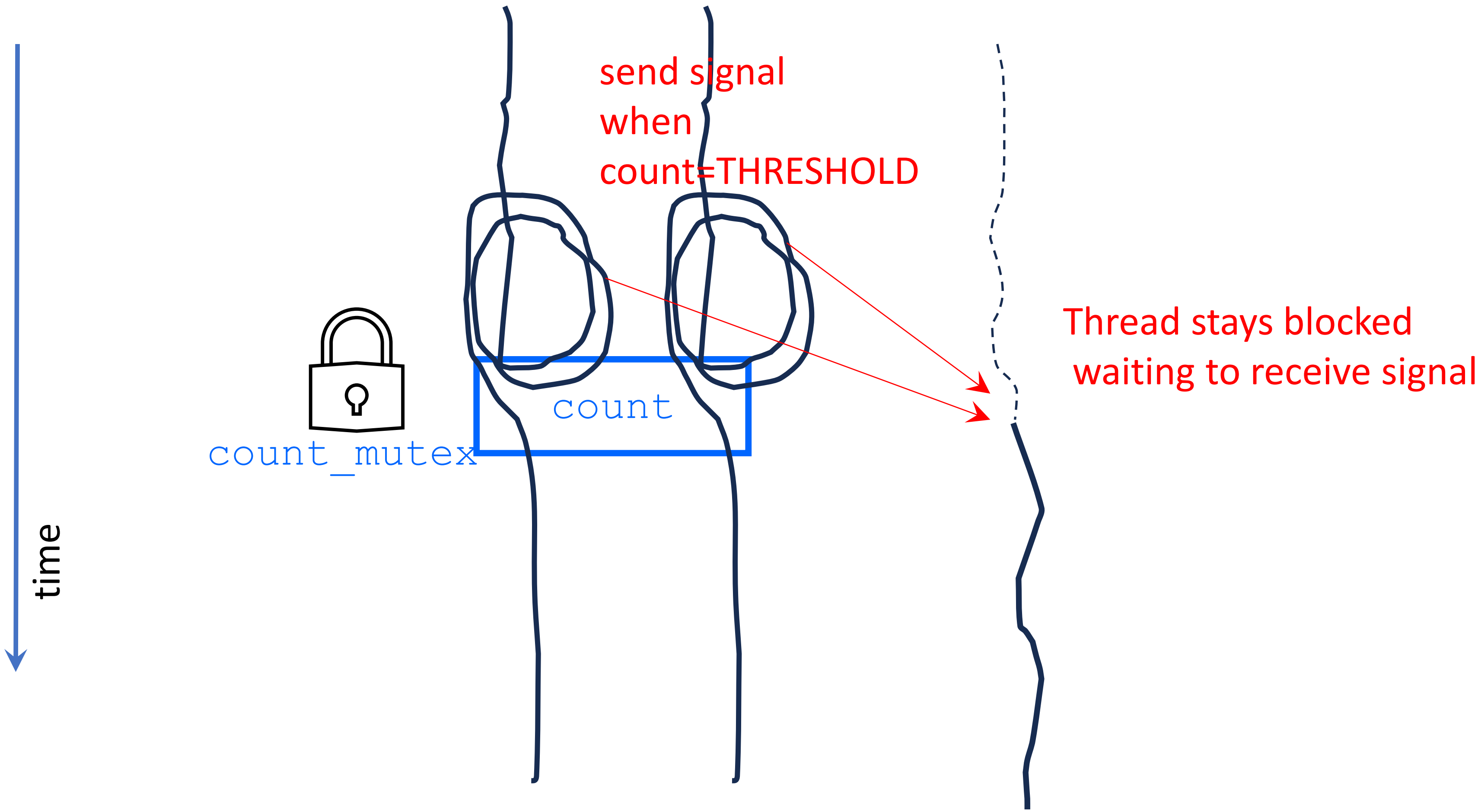
Condition Variables

- ***A condition variable is associated with a predicate (Boolean condition on a shared variable)***
- ***A condition variable always <u>has a mutex associated</u> with it – because the Boolean condition is given by a <u>shared</u> variable***
- The thread A locks this mutex and tests the predicate defined on the shared variable; if the predicate is not true, the thread waits on the condition variable associated with the predicate using the function pthread_cond_wait.
- When another thread knows that the predicate becomes true, it uses the condition variable to send signal to threads waiting on the condition - function pthread_cond_signal

Condition Variables Example

- A simple scenario:
  - A shared variable int count
  - Two threads increment count, each thread performs a number of REPEAT increment operations
  - A third thread waits to be notified when the value of count passes over a THRESHOLD value in order to start doing some work

Condition signaling



Condition Variables Example (1)

```c
#define NUM_THREADS 3
#define REPEAT 15
#define THRESHOLD 10

/* shared variable */
int count = 0;

/* mutex to protect critical regions
when threads are updating count */
pthread_mutex_t count_mutex;

/* condition variable to signal
when count reaches threshold value*/
pthread_cond_t count_threshold_cv;
```

Condition Variables Example (2):

```
/* thread function for the two incrementer threads */
void *increment_count(void *t)
{
  int i;
  int my_id = *(int *)t;
  for (i = 0; i < REPEAT; i++)
  {
    pthread_mutex_lock(&count_mutex);
    count++;
    if (count == THRESHOLD)
    {
      /* if condition reached send signal to waiting thread  */
      pthread_cond_signal(&count_threshold_cv);
    }
    pthread_mutex_unlock(&count_mutex);
    sleep(1); /* simulate some activity */
  }
  pthread_exit(NULL);
}
```

Condition Variables Example (3):

```
/* thread function for the wait thread */
void *wait_count(void *t)
{
  inr my_id = *(int *)t;

  /*
  Lock mutex and wait for signal.
  Function pthread_cond_wait will automatically unlock mutex while it waits.
  While is needed in case the thread gets unblocked by some other OS event
  */
  pthread_mutex_lock(&count_mutex);
  while (count < THRESHOLD)
  {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
  }
  pthread_mutex_unlock(&count_mutex);

  pthread_exit(NULL);
}
```

Use while, NOT if !!!!

Condition Variables Example (4)

```c
int main(int argc, char *argv[]) {
  int i;
  pthread_t threads[NUM_THREADS];
  int ids[NUM_THREADS];

  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init(&count_threshold_cv, NULL);

  ids[0] = 0;
  pthread_create(&threads[0], NULL, wait_count, (int *)&ids[0]);
  for (i = 1; i < NUM_THREADS; i++) {
    ids[i] = i;
    pthread_create(&threads[i], NULL, increment_count, (int *)&ids[i]);
  }
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);
}
```

pthread_cond_signal

- When a thread sees that a predicate has been satisfied, it signals the condition using the function pthread_cond_signal
- int pthread_cond_signal(pthread_cond_t *cond);
- When the condition is signaled (using pthread_cond_signal), **one** of the waiting threads is unblocked, and when the mutex becomes available, it is handed to this thread (and the thread becomes runnable).

- It is possible to wake **all** threads that are waiting on the condition variable as opposed to a single thread. This can be done using the function pthread_cond_broadcast.
- int pthread_cond_broadcast(pthread_cond_t *cond);

pthread_cond_wait

- int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
- A call to this function blocks the execution of the thread until it receives a signal from another thread or is interrupted by an OS signal.
- **In addition to blocking the thread, the pthread_cond_wait function releases the lock on mutex, for the duration of its blocked status**
  - This is important because otherwise no other thread will be able to work on the shared variable and the predicate would never be satisfied.
- When the thread is released on a signal, it waits to reacquire the lock on mutex before returning. It always returns with the mutex locked.
- **Programming Note:** A call to pthread_cond_wait **must be put in a loop checking the predicate associated with the condition variable**, *because the thread might be woken up also due to some* ***other*** *reasons* (such as an OS signal).

Barriers

- Another synchronization problem: We need to perform a multi-threaded computation that has two stages (several threads execute each stage), but we don’t want to advance to the second stage until all threads finished the first stage.

  - Dinner table manners: do not start eating second course until everyone (including the slowest eater) finished eating the first course!

Barriers

- **Barrier:** introduces a point of synchronization where no thread can proceed beyond the barrier until all the threads have reached the barrier.
  - A Barrier object is initialized with the number of threads that want to participate
  - pthread_barrier_t
  - Each participating thread executes a call to a barrier function pthread_barrier_wait in the place where it wants to synchronize with the others
  - When a thread reaches a barrier (executes the call to the barrier function), it will ***wait until all the threads participating in the barrier reach the barrier***, and then they'll all proceed together.

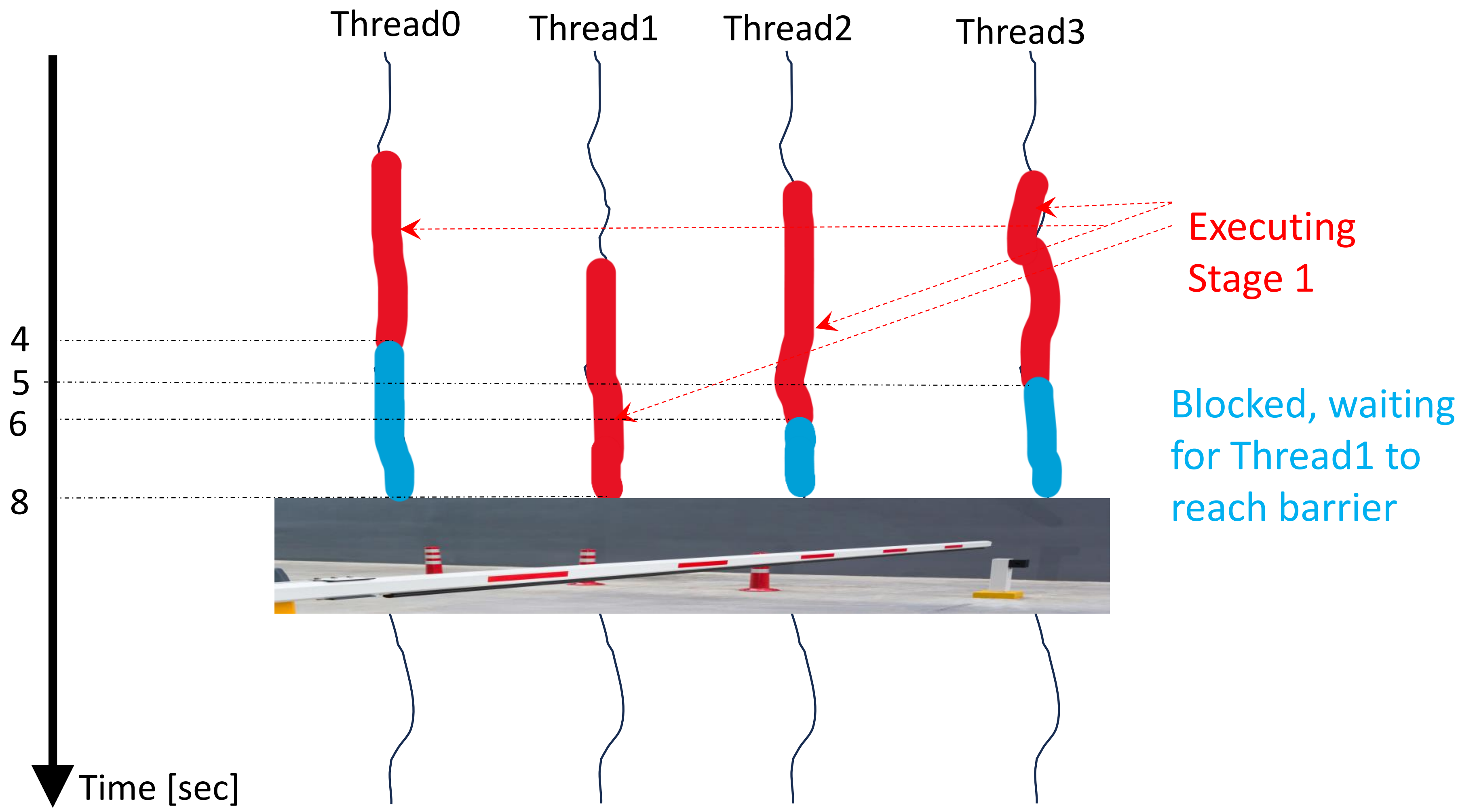
Barrier Example(1)

```
/* the barrier */
pthread_barrier_t mybarrier;

/* thread function for all threads*/
void *threadFn(void *id_ptr)
{
    int thread_id = *(int *)id_ptr;

    printf("thread %d: starting Stage 1.\n", thread_id);
    sleep(thread_id); /* simulate work in stage 1*/
    printf("thread %d: Stage 1 finished ...\n", thread_id);

    pthread_barrier_wait(&mybarrier);

    printf("thread %d: going to Stage 2!\n", thread_id);

    return NULL;
}
```

Barrier Example(2)

```c
int main(int argc, char *argv[]){

    int i;
    pthread_t threads[THREAD_COUNT];
    int ids[THREAD_COUNT];

    pthread_barrier_init(&mybarrier, NULL, THREAD_COUNT);

    for (i = 0; i < THREAD_COUNT; i++){
        ids[i]=i;
        pthread_create(&threads[i], NULL, threadFn, (void *)&ids[i]);
    }

    for (i = 0; i < THREAD_COUNT; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_barrier_destroy(&mybarrier);
    return 0;
}
```

Barrier Example



Source Code of of Examples

- mutex.c
- condvar.c
- barrier.c

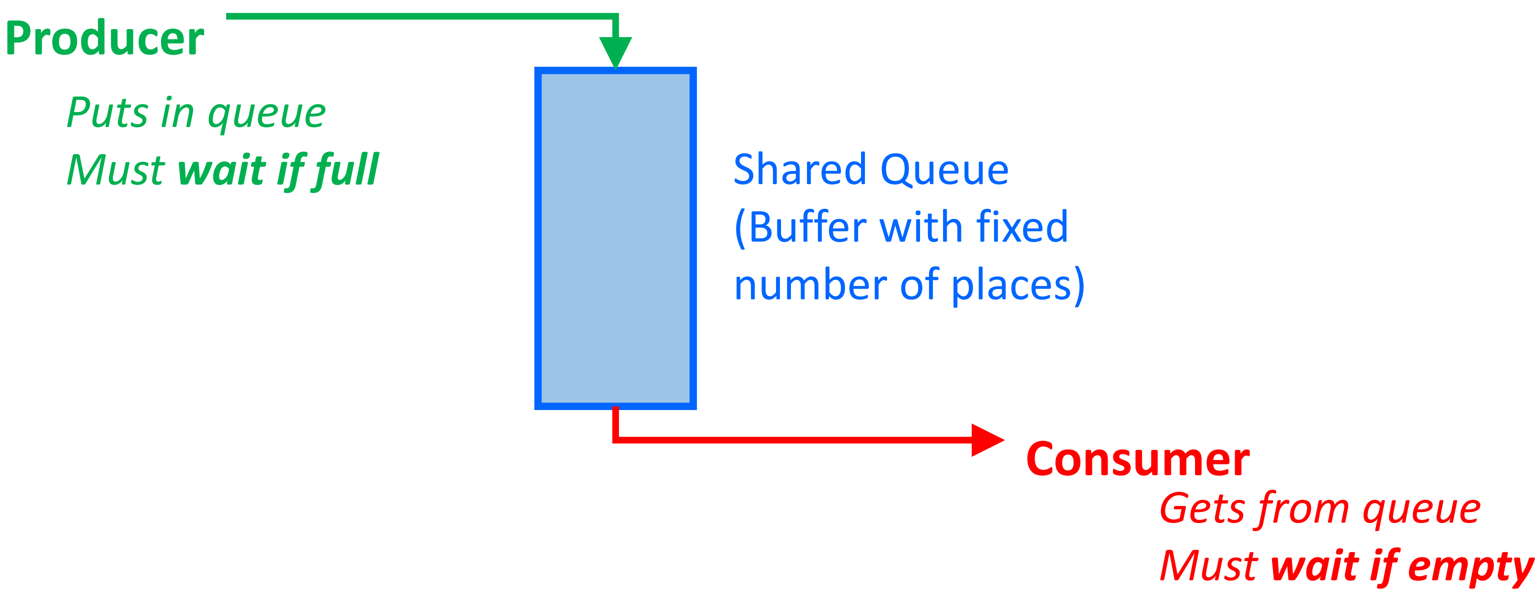
The Producer-Consumer Problem

The Producer-Consumer Problem

- The most classic problem for synchronization
- Producer-consumer relationships occur frequently in various applications
  - The producer threads create tasks and insert them into a work-queue. The consumer threads pick up tasks from the task queue and executes them. Producers and consumers may work at different speed!
- Variants:
  - The shared queue between producers and consumers has a fixed size (Producer-Consumer with Bounded Buffer)
  - The shared queue between producers and consumers has an unlimited size (Producer-Consumer with Unlimited Buffer or Infinite Buffer)

Producer-Consumer with Bounded Buffer

- A number of **Producers** put items into a Shared Queue (a Buffer)
- A number of **Consumers** get items out of the Shared Queue
- All Producers and Consumers work concurrently
- The size of the Queue is **fixed (there are a limited number of places in the Queue = a Bounded Buffer)**

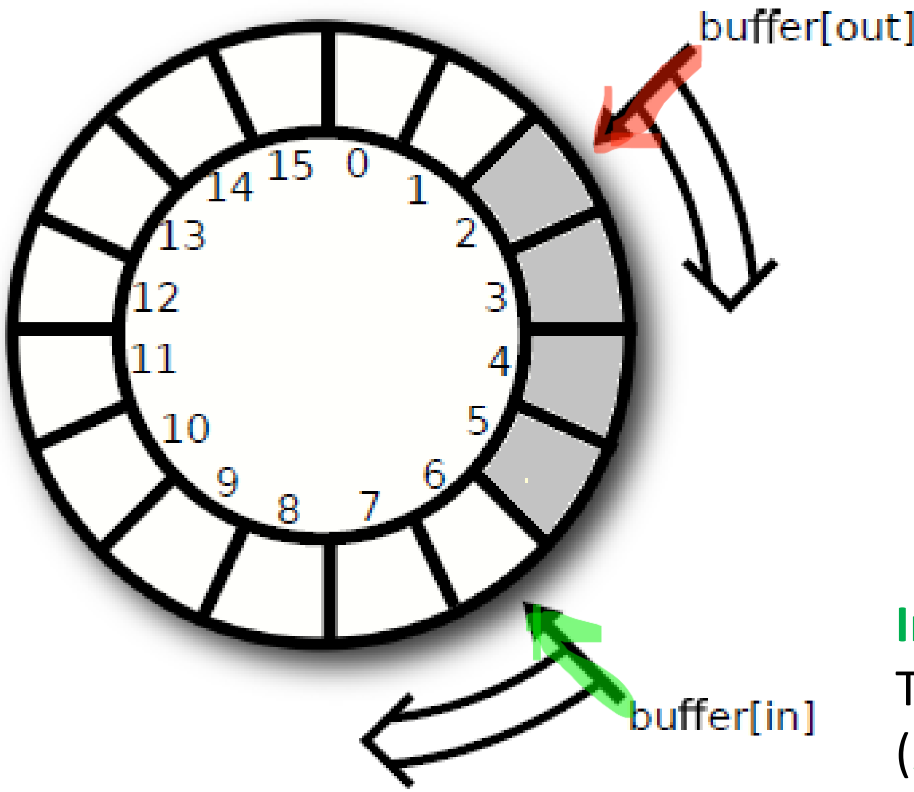


Producer-Consumer with Bounded Buffer

- Problems:
  - If the Buffer is **empty**, *Consumers must block* until some item appear in queue
  - If the Buffer is **full**, *Producers must block* until some item is removed from queue
  - Several Producers or Consumers must not attempt to put or get items from the queue at the same time (the **classical mutual exclusion** – cannot have 2 threads increment the same head or tail index at the same time)

Bunded Buffer Queue Implementation

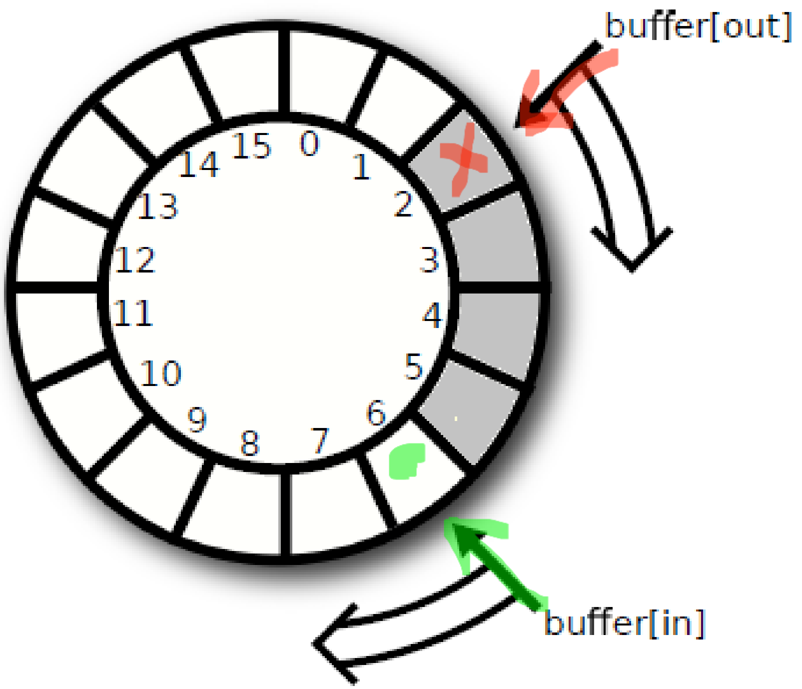
- **Circular Array** (**Ring Buffer**): an array exploited in a circular way: after the last index, we consider that the next element follows at the first index



out: the index of the Head of the Queue (*Consumers get this element – dequeue*)

In: the index of the Tail of the Queue (*Producers put here - enqueue*)

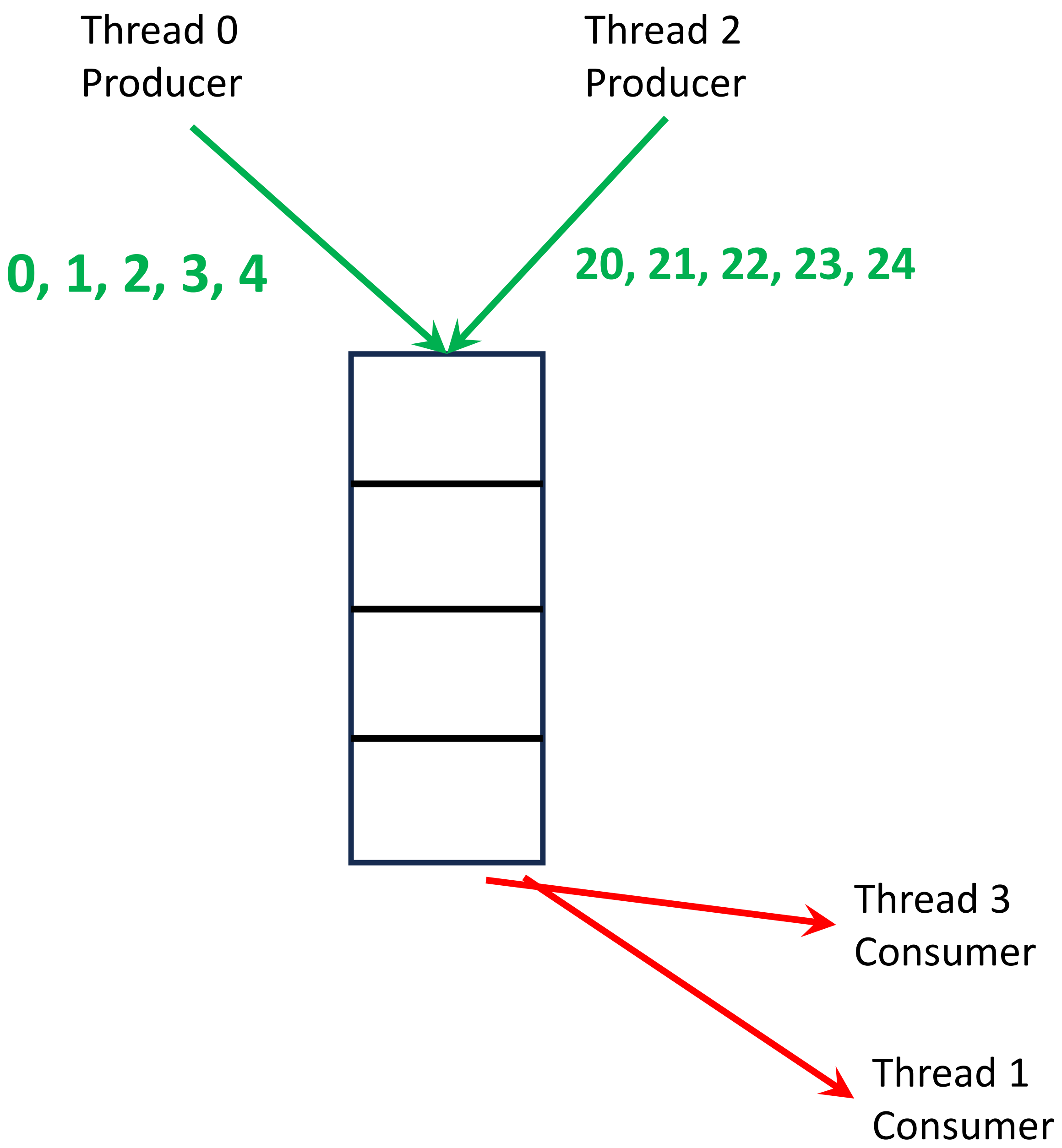
Bunded Buffer Queue Implementation



- Initially: Buffer empty, in=out=0
- Enqueue: Put in buffer:
  - b[in] = value;
  - in = (in + 1) % BUFFER_SIZE;
- Test Buffer is Full:
  - If ((in + 1) % BUFFER_SIZE == out)
- Dequeue: Get from buffer:
  - value = b[out];
  - out = (out + 1) % BUFFER_SIZE;
- Test Buffer is Empty:
  - If (out == in)

Producer and Consumer Threads

```
/* thread function for producer threads */
void *producer(void *t) {
    int i;
    int my_id = *(int *)t;
    for (i = 0; i < REPEAT; i++) {
        enqueue(i + my_id * 10);
    }
    pthread_exit(NULL);
}

/* thread function for consumer threads */
void *consumer(void *t) {
    int i;
    int my_id = *(int *)t;
    for (i = 0; i < REPEAT; i++)  {
        int rez = dequeue();
        printf("Consumer thread %d got %d \n", my_id, rez);
    }
    pthread_exit(NULL);
}
```


Thread 0
Producer
Thread 2
Producer
0, 1, 2, 3, 4
20, 21, 22, 23, 24
Thread 3
Consumer
Thread 1
Consumer

Enqueue and Dequeue

- **Producer: Enqueue**
- If *queue is full*:
  - thread must **wait** until a place gets empty (buffer not full condition)
- If *queue is not full*:
  - *put element at tail*
  - **notify** other threads(consumers) that an element is there = buffer not empty condition true

- **Consumer: Dequeue**
- If *queue is empty*:
  - thread must **wait** until at least an element is there (buffer not empty condition)
- If *queue is not empty*:
  - *removes head element*
  - **notify** other threads (producers) that a place got empty = buffer not full condition true

Enqueue and Dequeue

- **Producer: Enqueue**
- If *queue is full*:
  - thread must **wait** until a place gets empty (buffer not full condition)
- If *queue is not full*:
  - *put element at tail*
  - **notify** other threads(c
    that an e
    = buffer n
    conditio

- **Consumer: Dequeue**
- If *queue is empty*:
  - thread must **wait** until at least an element is there (buffer not empty condition)
- If *queue is not empty*:
  - *removes head element*
  - **notify** other threads
) that a place
    = buffer not
    on true

Every operation that handles the buffer in-out indexes must obtain exclusive access to these variables

Producer-Consumer with Bounded Buffer – Solution with Mutex locks and Condition Variables

Producer-Consumer with Bounded Buffer- Solution with Mutex locks and Condition Variables

```
/* condition variable to signal when buffer
is not empty - wakes up a waiting consumer  */
pthread_cond_t not_empty_cv;

/* condition variable to signal when buffer
is not full - wakes up a waiting producer */
pthread_cond_t not_full_cv;


/* mutex lock protecting access to the buffer */
pthread_mutex_t lock;
```

```c
#define REPEAT 5
#define NUM_THREADS 4

#define BUFFER_SIZE 4

/* The buffer containing the Shared Queue */
int b[BUFFER_SIZE];

/* Index of queue tail (where to put next) – shared variable */
int in = 0;
/* Index of queue head (where to get next) – shared variable */
int out = 0;

/* mutex lock protecting access to the buffer */
pthread_mutex_t lock;

/* condition variable to signal when buffer
is not empty - wakes up a waiting consumer  */
pthread_cond_t not_empty_cv;

/* condition variable to signal when buffer
is not full - wakes up a waiting producer */
pthread_cond_t not_full_cv;
```

```c
int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int ids[NUM_THREADS];

    /* Init cond var and mutex */
    init_synchro();

    /* Create threads */
    /* odd thread ranks are consumers, even ranks are producers */
    for (int i = 0; i < NUM_THREADS; i++) {
        ids[i] = i;
        if (i % 2 ==0)
            pthread_create(&threads[i], NULL, producer, (void *)&ids[i]);
        else
            pthread_create(&threads[i], NULL, consumer, (void *)&ids[i]);
    }

    /* Wait for all threads to complete */
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    /* Clean up */
    destroy_synchro();
}
```

Init and cleanup synchro

```c
void init_synchro()
{
    /* Initialize mutex */
    pthread_mutex_init(&lock, NULL);
    /* Initialize cond vars */
    pthread_cond_init(&not_empty_cv, NULL);
    pthread_cond_init(&not_full_cv, NULL);
}


void destroy_synchro()
{
    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&not_empty_cv);
    pthread_cond_destroy(&not_full_cv);
}
```

Producer and Consumer Threads

```
/* thread function for producer threads */
void *producer(void *t) {
    int i;
    int my_id = *(int *)t;
    for (i = 0; i < REPEAT; i++) {
        enqueue(i + my_id * 10);
    }
    pthread_exit(NULL);
}

/* thread function for consumer threads */
void *consumer(void *t) {
    int i;
    int my_id = *(int *)t;
    for (i = 0; i < REPEAT; i++)  {
        int rez = dequeue();
        printf("Consumer thread %d got %d \n", my_id, rez);
    }
    pthread_exit(NULL);
}
```

```c
/* Add one value into the Queue.
If Queue is full, wait */
void enqueue(int value)
{
    pthread_mutex_lock(&lock);

    /* while queue is full, wait */
    /* need a while loop, not a simple if !!! */
    while ((in + 1) % BUFFER_SIZE == out)
        pthread_cond_wait(&not_full_cv, &lock);

    /* put in queue */
    b[in] = value;
    in = (in + 1) % BUFFER_SIZE;

    /* signal that buffer is not empty */
    pthread_cond_signal(&not_empty_cv);
    pthread_mutex_unlock(&lock);
}
```

```c
/* Pop one value from the Queue.
If Queue isempty, wait*/
int dequeue()
{
    pthread_mutex_lock(&lock);

    /* while queue is empty, wait */
     /* need a while loop, not a simple if !!! */
    while (out == in)
        pthread_cond_wait(&not_empty_cv, &lock);

    /* take out an element */
    int tmp = b[out];
    out = (out + 1) % BUFFER_SIZE;

    /* signal that buffer is not full   */
    pthread_cond_signal(&not_full_cv);

    /* exit critical section */
    pthread_mutex_unlock(&lock);

    return tmp;
}
```

Thinking question (1)

- WHY should we use while and not just a simple if in this sequence?

```
/* while queue is empty, wait */
 /* need a while loop, not a simple if !!! */
while (out == in)
    pthread_cond_wait(&not_empty_cv, &lock);
```

Thinking question (2)

- In a sequence like the one below, if the queue is empty, the thread waits (is blocked) in the condition variable. What happens with the mutex lock held by this thread?

```
/* Pop one value from the Queue.
If Queue isempty, wait*/
int dequeue()
{
    pthread_mutex_lock(&lock);

    /* while queue is empty, wait */
    while (out == in)
        pthread_cond_wait(&not_empty_cv, &lock);
    …
    …
```

Thinking question (3)

- In a sequence like the one below, the thread acquires lock1 but gets blocked when trying to acquire lock2.
- The thread waits (is blocked) at lock2. What happens with the lock1 held by this thread?

```
pthread_mutex_lock(&lock1);

pthread_mutex_lock(&lock2);

…
…
```

Producer-Consumer with Bounded Buffer

- Solution with Mutex Locks and Condition variables: bounded_buff_condvar.c

Producer-Consumer with Bounded Buffer – Solution with Semaphores

Producer-Consumer with Bounded Buffer – Solution with Semaphores

```
#define BUFFER_SIZE 16

/* The buffer containing the Queue */
int b[BUFFER_SIZE];

/* Index of queue tail (where to add next item) */
int in = 0;
/* Index of queue head (where to remove item) */
int out = 0;

/* mutual exclusion on shared buffer and in out indexes */
pthread_mutex_t lock;

/* semaphore will block consumers if buffer is empty */
/* initialized on value 0 */
sem_t empty;

/* semaphore will block producers if buffer is full */
/* initialized on value BUFFER_SIZE */
sem_t full;
```

```c
/* Add one value into the Queue.
If Queue is full, wait */
void enqueue(int value)
{
    /* wait if the buffer is full */
    sem_wait(&full);

    pthread_mutex_lock(&lock);
    b[(in++) % BUFFER_SIZE] = value;
    pthread_mutex_unlock(&lock);

    /* signal that buffer is not empty
    - increment semaphore of used entries*/
    sem_post(&empty);
}
```

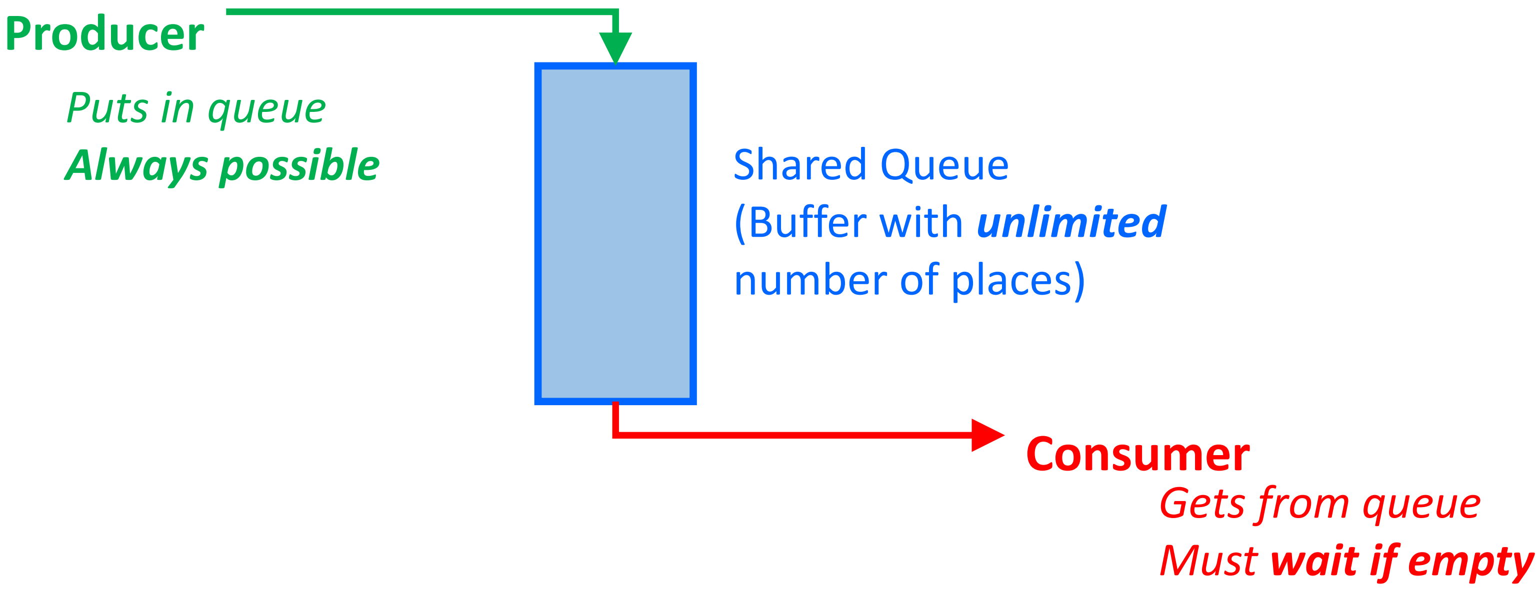
```c
/* Pop one value from the Queue.
If Queue isempty, wait*/
int dequeue()
{
    /* wait if the buffer is empty */
    sem_wait(&empty);

    pthread_mutex_lock(&lock);
    int result = b[(out++) % (BUFFER_SIZE)];
    pthread_mutex_unlock(&lock);

    /* signal that buffer is not full
    - increment semaphore of free space */
    sem_post(&full);
    return result;
}
```

Producer-Consumer with Bounded Buffer

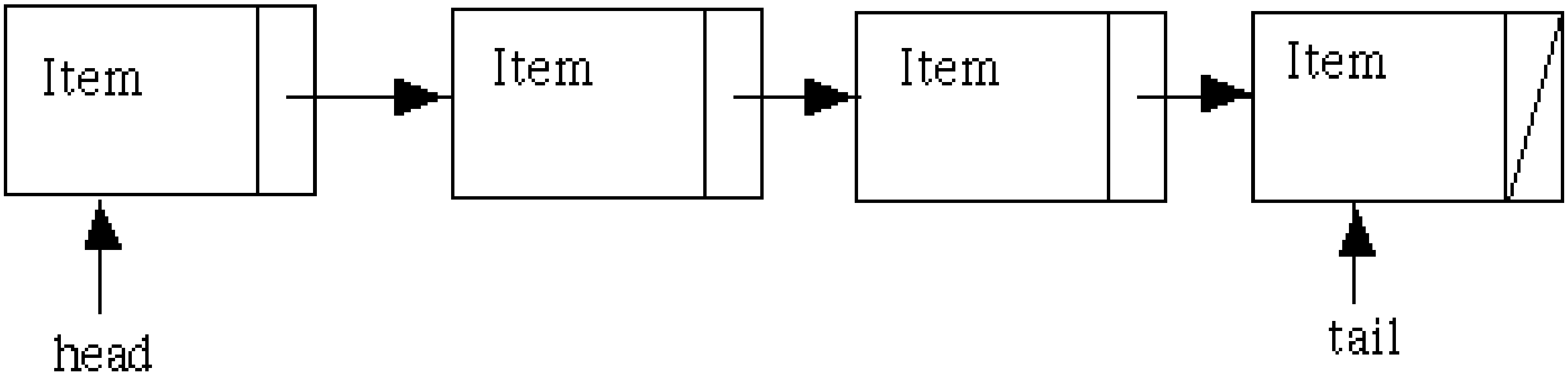
- Source code for solution with Semaphores: [bounded_buff_semaph.c]

Producer-Consumer with Unbounded Buffer

- A number of **Producers** put items into a Shared Queue (a Buffer)
- A number of **Consumers** get items out of the Shared Queue
- All Producers and Consumers work concurrently
- The size of the Queue is **unbounded (infinite buffer)**



Unbounded Buffer Queue Implementation



head: *Consumers get (dequeue) from here*

tail: *Producers put (enqueue) here*