LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

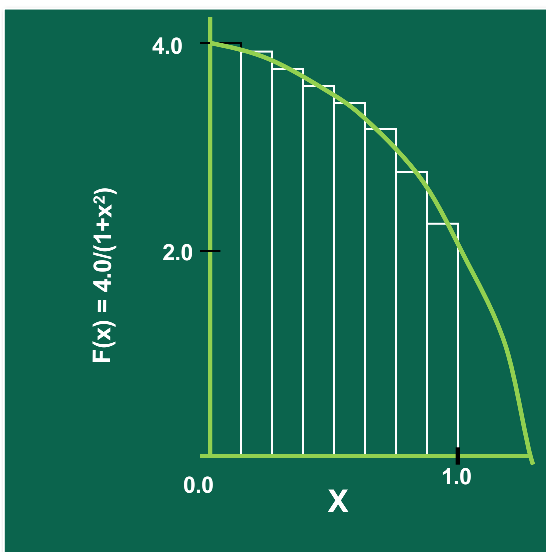
Shared Memory Programming Using POSIX Threads - Examples

Instructor: Haidar M. Harmanani

Spring 2021

Example 1: Computing π

$$\tan^{-1}(x) = \int_0^x \frac{dt}{1+t^2}$$



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Example 1: Computing π

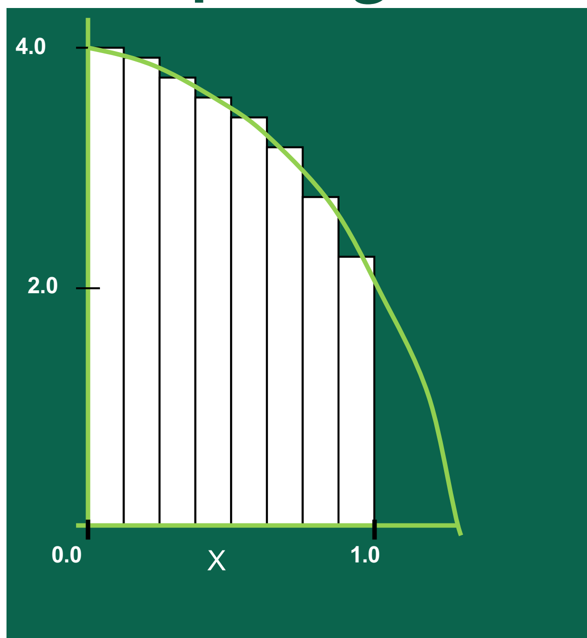
$$\tan^{-1}(x) = \int_0^x \frac{dt}{1+t^2}$$

```
static long num_steps = 100000;
double step;
void main ()
{
   int i;
   double x, pi, sum = 0.0;

   step = 1.0/(double) num_steps;

   for (i=0;i< num_steps; i++){
         x = (i+0.5)*step;
         sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```


Computing π



$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

Data Structure

```
#define STEPS 1000000000
#define STEP_SIZE 1.0/STEPS
#define THREADS 3

struct pthread_args
{
double lower;
double upper;
double local_sum;
};
```


Part I: Thread Call

```
...
pthread_t *thread;
    struct pthread_args *thread_arg;


thread = malloc((unsigned long)num_threads * sizeof(*thread));
thread_arg = malloc((unsigned long)num_threads *
sizeof(*thread_arg));


for (int i = 0; i < num_threads; i++)
{
thread_arg[i].lower = (i+0) * (1.0 / (double)num_threads);
thread_arg[i].upper = (i+1) * (1.0 / (double)num_threads);
pthread_create(thread + i, NULL, &pi_thread, thread_arg + i);
}
```

Part II: Thread Function

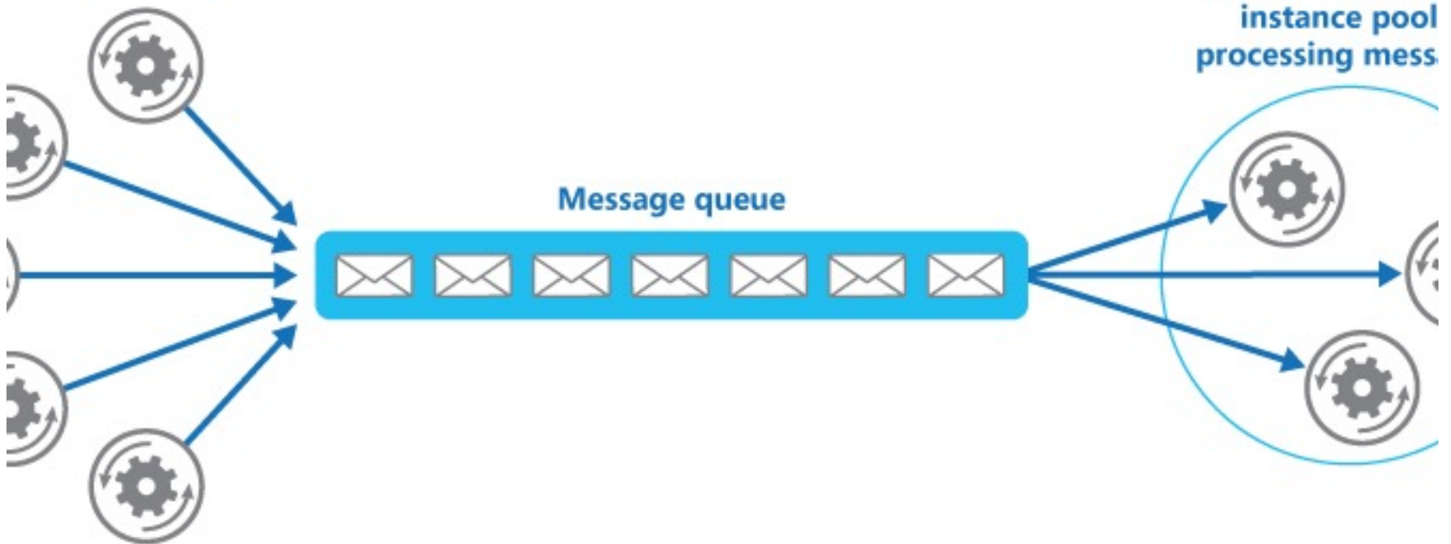
```
void * pi_thread(void *ptr)
{
double low = 0.5 * STEP_SIZE + ((struct pthread_args*)ptr)->lower;
double upp = ((struct pthread_args*)ptr)->upper;
double tsum = 0;


while(low < upp)
{
tsum += sqrt(1-low*low) * STEP_SIZE;
low += STEP_SIZE;
}
((struct pthread_args*)ptr)->local_sum = tsum;


return NULL;
}
```


Step III

- Complete the code in a breakout room.
- Compile
- Any thoughts?



Condition Variables and Consumer/Producer Problem

How to check whether a child thread has completed?

```
1  void  *child(void  *arg) {
2    printf("child\n");
3    // XXX  how  to indicate we  are done?
4    return  NULL;
5  }
6
7  int  main(int  argc,  char  *argv[])  {
8    printf("parent: begin\n");
9    pthread_t c;
10   Pthread_create(&c,  NULL,  child,  NULL);  //  create  child
11   //  XXX  how  to  wait  for  child?
12   printf("parent: end\n");
13   return  0;
```

Possible Solution: Problem?

```
1 volatile int done = 0;
2
3 void *child(void *arg) {
4   printf("child\n");
5   done = 1;
6   return NULL;
7 }
8
9 int main(int argc, char *argv[]) {
10   printf("parent: begin\n");
11   pthread_t c;
12   pthread_create(&c, NULL, child, NULL); // create child
13   while (done == 0)
14     ; // spin
15   printf("parent: end\n");
16   return 0;
17 }
```


The Crux

HOW TO WAIT FOR A CONDITION?
In multi-threaded programs, it is often useful for a thread to wait for some condition to become true before proceeding. The simple approach, of just spinning until the condition becomes true, is grossly inefficient and wastes CPU cycles, and in some cases, can be incorrect. Thus, how should a thread wait for a condition?

Condition Variables

- There are cases where a thread wishes to check whether a condition is true before continuing its execution
- Condition variables provide a mechanism for threads to synchronize.
  - mutex'es implement synchronization by controlling thread access to data
  - Condition variables allow threads to synchronize based on the actual value of data.

Condition Variables

- A condition variable is an explicit queue that threads can put themselves on when some state of execution is not as desired
  - Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met.
    - Resource consuming
- The idea goes back to Dijkstra's use of "private semaphores"
- A condition variable (cv) allows a thread to block itself until a specified condition becomes true.

Condition Variables

- A condition variable is always used in conjunction with a `mutex` lock.
- When a thread executes **`pthread_cond_wait(cv)`**, it is blocked until another thread executes **`pthread_cond_signal(cv)`** or **`pthread_cond_broadcast(cv)`**.
  - pthread_cond_signal() is used to unblock one of the threads blocked waiting on the condition variable.
  - pthread_cond_broadcast() is used to unblock all the threads blocked waiting on the condition variable.
- If no threads are waiting on the condition variable, then a pthread-cond_signal() or pthreadcond_broadcast() will have no effect.

Notes on Condition Variables

- Use a `while` loop instead of an `if` statement to check the waited for condition in order to alleviate the following problems:
  - If several threads are waiting for the same wake up signal, they will take turns acquiring the mutex, and any one of them can then modify the condition they all waited for.
- Mutex is unlocked
  - Allows other threads to acquire lock
  - When signal arrives, mutex will be reacquired before **`pthread_cond_wait`** returns

Main Thread
- Declare and initialize global data/variables that require synchronization (such as "count")
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
- Create threads A and B to do work

Thread A
- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call **`pthread_cond_wait()`** to perform a blocking wait for signal from ***Thread B.***
- A call to **`pthread_cond_wait()`** automatically and atomically unlocks the associated mutex variable so that it can be used by ***Thread B***.
- When signaled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

Thread B
- Do work
- Lock associated mutex
- Change the value of the global variable that ***Thread A*** is waiting upon.
- Check value of the global ***Thread A*** wait variable. If it fulfills the desired condition, **signal *Thread A***.
- Unlock mutex.
- Continue

Main Thread
Join / Continue

Problem
- Two of the threads perform work and update a "count" variable.
- The third thread waits until the count variable reaches a specified value.

```
int main (int argc, char *argv[])
 {
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);
 }
```

Problem

- Two of the threads perform work and update a "count" variable.
- The third thread waits until the count variable reaches a specified value.

```
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
 {
   int i;
   long my_id = (long)t;

   for (i=0; i<TCOUNT; i++) {
     pthread_mutex_lock(&count_mutex);
     count++;

     if (count == COUNT_LIMIT) {
       pthread_cond_signal(&count_threshold_cv);
       printf("inc_count(): thread %ld, count = %d  Threshold reached.\n", my_id, count);
       }
     printf("inc_count(): thread %ld, count = %d, unlocking mutex\n", my_id, count);
     pthread_mutex_unlock(&count_mutex);

       sleep(1);
     }
   pthread_exit(NULL);
 }
```


Problem

- Two of the threads perform work and update a "count" variable.
- The third thread waits until the count variable reaches a specified value.

```
void *watch_count(void *t)
 {
   long my_id = (long)t;

   printf("Starting watch_count(): thread %ld\n", my_id);

  /*
   Lock mutex and wait for signal.  Note that the pthread_cond_wait routine will automatically and
   atomically unlock mutex while it waits. Also, note that if COUNT_LIMIT is reached before this
   routine is run by   the waiting thread, the loop will be skipped to prevent pthread_cond_wait
   from never returning.
  */
   pthread_mutex_lock(&count_mutex);
   while (count<COUNT_LIMIT) {
     pthread_cond_wait(&count_threshold_cv, &count_mutex);
     printf("watch_count(): thread %ld Condition signal received.\n", my_id);
     count += 125;
     printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
    }
   pthread_mutex_unlock(&count_mutex);
   pthread_exit(NULL);
 }
```

Condition Variables: Declaration

- Condition variables must be declared with type `pthread_cond_t`, and must be initialized before they can be used.
  - `pthread cond t c` ← declares `c` as a condition variable
- Two ways to initialize a condition variable:
  - Statically, when it is declared. For example:
    `pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;`
  - Dynamically, with the `pthread_cond_init()` routine.
    - The ID of the created condition variable is returned to the calling thread through the condition parameter.
    - This method permits setting condition variable object attributes, `attr`.

pthread_cond_init

```
int pthread_cond_init( cond, attr );
```

`pthread_cond_t *cond`

– condition variable to be initialized

`const pthread_condattr_t *attr`

– attributes to be given to condition variable

`ENOMEM` - **insufficient memory for mutex**
`EAGAIN` - **insufficient resources (other than memory)**
`EBUSY` - **condition variable already intialized**
`EINVAL` - **`attr` is invalid**

Alternate Initialization

- Can also use the static initializer PTHREAD_COND_INITIALIZER

```
pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;
```

  – Uses default attributes
- Programmer must always pay attention to condition (and mutex) scope
  – Must be visible to threads

Condition Variables: Usage

- A condition variable has two operations associated with it
  – `wait()`
  – `signal()`
- The **`wait()`** call is executed when a thread wishes to put itself to sleep
- The **`signal()`** call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition

```
pthread_cond_wait(pthread_cond_t  *c,  pthread_mutex_t  *m);
pthread_cond_signal(pthread_cond_t *c);
```

Condition Variable and Mutex

- Mutex is associated with condition variables
  - Protects evaluation of the conditional expression
  - Prevents race condition between signaling thread and threads waiting on condition variable
- While the thread is waiting on a condition variable, the mutex is automatically unlocked, and when the thread is signaled, the mutex is automatically locked again

`pthread_cond_signal` Explained

- Signal condition variable, wake one waiting thread
- If no threads waiting, no action taken
  - Signal is not saved for future threads
- Signaling thread need not have mutex
  - May be more efficient
  - Problem may occur if thread priorities used

`EINVAL` **- cond is invalid**

`pthread_cond_broadcast` Explained

- Wake all threads waiting on condition variable
- If no threads waiting, no action taken
  - Broadcast is not saved for future threads
- Signaling thread need not have mutex

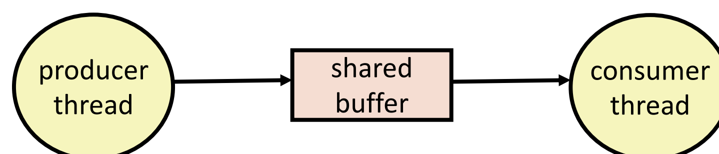
`EINVAL` - cond is invalid

Producer/Consumer Problem

Producer-Consumer Problem

- A common implementation pattern for cooperating processes or threads.
  – Producer produces information that is later consumed by a consumer
- Implementation details:
  – Use a common buffer in order to allow the producer and consumer to run concurrently
  – Synchronize processes so that the consumer does not try to consume an item that has not yet been produced
  – If the common data buffer is bounded, the consumer process must wait if the buffer is empty, and the producer process must wait if the buffer is full.

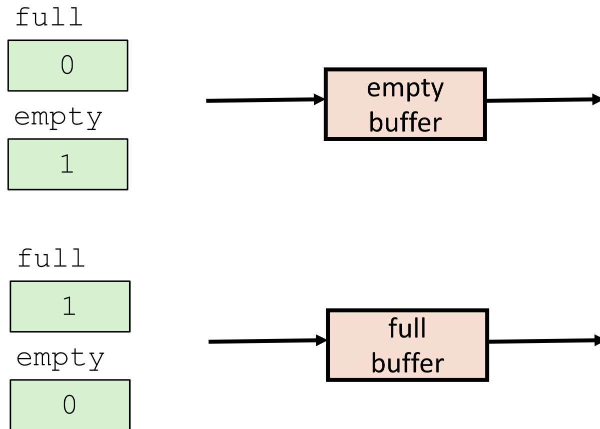
Producer-Consumer Problem



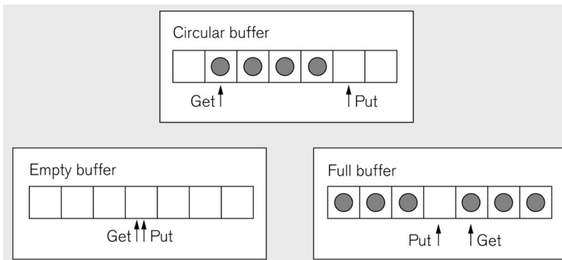
- Examples
  – Multimedia processing:
    - Producer creates video frames, consumer renders them
  – Event-driven graphical user interfaces
    - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer
    - Consumer retrieves events from buffer and paints the display

Producer-Consumer on 1-element Buffer

- Maintain two semaphores: full + empty



Condition Variables
Example




```
1   pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
2   pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
3   pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
4   Item buffer[SIZE];
5   int put=0;                         // Buff index for next insert
6   int get=0;                         // Buff index for next remove
7
8   void insert(Item x)                       // Producer thread
9   {
10    pthread_mutex_lock(&lock);
11    while((put>get&&(put-get)==SIZE-1)||   // While buffer is
12          (put<get&&(put+get)==SIZE-1))    // full
13    {
14      pthread_cond_wait(&nonfull, &lock);
15    }
16    buffer[put]=x;
17    put=(put+1)%SIZE;
18    pthread_cond_signal(&nonempty);
19    pthread_mutex_unlock(&lock);
20  }
21
22  Item remove()                             // Consumer thread
23  {
24    Item x;
25    pthread_mutex_lock(&lock);
26    while(put==get)                         // While buffer is empty
27    {
28      pthread_cond_wait(&nonempty, &lock);
29    }
30    x=buffer[get];
31    get=(get+1)%SIZE;
32    pthread_cond_signal(&nonfull);
33    pthread_mutex_unlock(&lock);
34    return x;
35  }
```

Sequential Circular Buffer Code

```
init(int v)
{
   items = front = rear = 0;
}
```

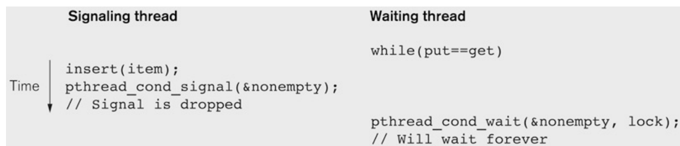
```
int remove()
{
   if (items == 0)
       error();
   if (++front >= n) front = 0;
   int v = buf[front];
   items--;
   return v;
}
```

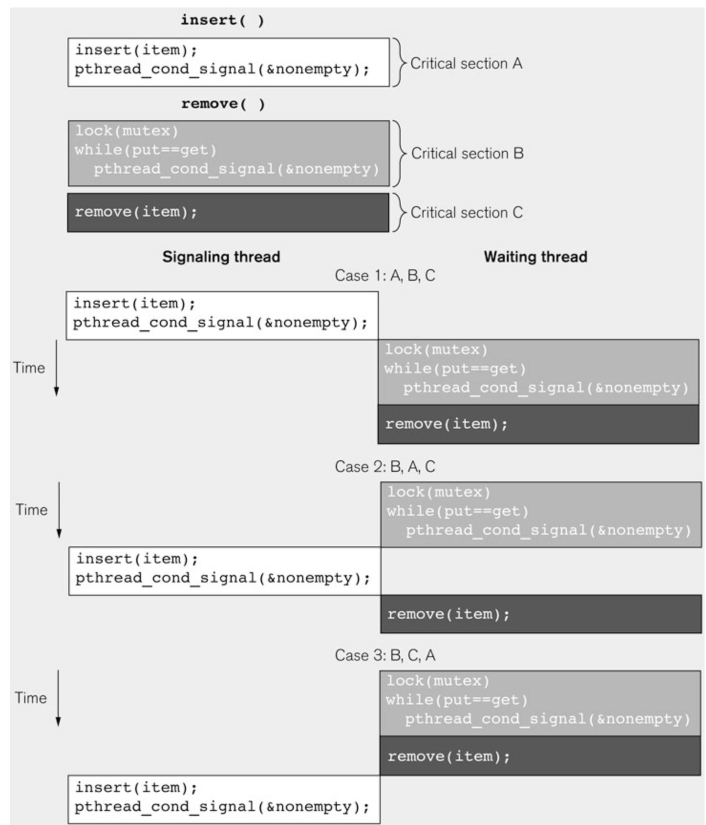
```
insert(int v)
{
   if (items >= n)
       error();
   if (++rear >= n) rear = 0;
   buf[rear] = v;
   items++;
}
```


Use of Mutex and Signals

```
1  pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
3  pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
4  Item buffer[SIZE];
5  int put=0;                         // Buff index for next insert
6  int get=0;                         // Buff index for next remove
7
8  void insert(Item x)                      // Producer thread
9  {
10   pthread_mutex_lock(&lock);
11   while((put>get&&(put-get)==SIZE-1)||   // While buffer is
12         (put<get&&(put+get)==SIZE-1))    // full
13   {
14     pthread_cond_wait(&nonfull, &lock);
15   }
16   buffer[put]=x;
17   put=(put+1)%SIZE;
18   pthread_cond_signal(&nonempty);
19   pthread_mutex_unlock(&lock);
20 }
21
22 Item remove()                            // Consumer thread
23 {
24   Item x;
25   pthread_mutex_lock(&lock);
26   while(put==get)                        // While buffer is empty
27   {
28     pthread_cond_wait(&nonempty, &lock);
29   }
30   x=buffer[get];
31   get=(get+1)%SIZE;
32   pthread_cond_signal(&nonfull);
33   pthread_mutex_unlock(&lock);
34   return x;
35 }
```

Condition Variables
Example





Producer-Consumer on 1-element Buffer

```
#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
  int buf; /* shared var */
  sem_t full; /* sems */
  sem_t empty;
} shared;
```

```
int main(int argc, char** argv) {
  pthread_t tid_producer;
  pthread_t tid_consumer;

  /* Initialize the semaphores */
  sem_init(&shared.empty, 0, 1);
  sem_init(&shared.full,  0, 0);

  /* Create threads and wait */
  pthread_create(&tid_producer, NULL,
                 producer, NULL);
  pthread_create(&tid_consumer, NULL,
                 consumer, NULL);
  pthread_join(tid_producer, NULL);
  pthread_join(tid_consumer, NULL);

  return 0;
}
```

Producer-Consumer on 1-element Buffer

Initially: `empty==1, full==0`

Producer Thread

```
void *producer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* Produce item */
    item = i;
    printf("produced %d\n",
            item);

    /* Write item to buf */
    P(&shared.empty);
    shared.buf = item;
    V(&shared.full);
  }
  return NULL;
}
```

Consumer Thread

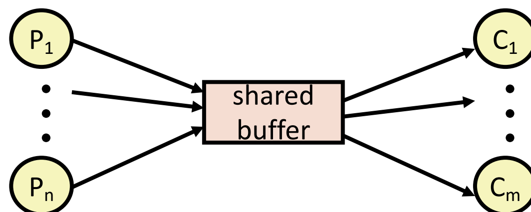
```
void *consumer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* Read item from buf */
    P(&shared.full);
    item = shared.buf;
    V(&shared.empty);

    /* Consume item */
    printf("consumed %d\n", item);
  }
  return NULL;
}
```


Why 2 Semaphores for 1-Entry Buffer?

- Consider multiple producers & multiple consumers

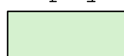


- Producers will contend with each to get empty
- Consumers will contend with each other to get full

Producers

```
P(&shared.empty);
shared.buf = item;
V(&shared.full);
```

empty



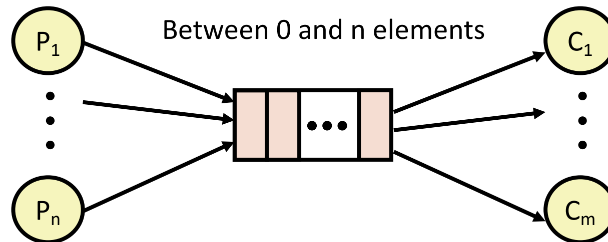
full

Consumers

```
P(&shared.full);
item =
shared.buf;
V(&shared.empty);
```

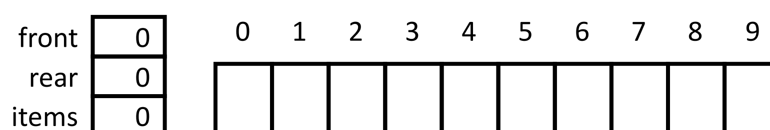
Producer-Consumer on an n-element Buffer

- Implemented using a shared buffer package called sbuf.

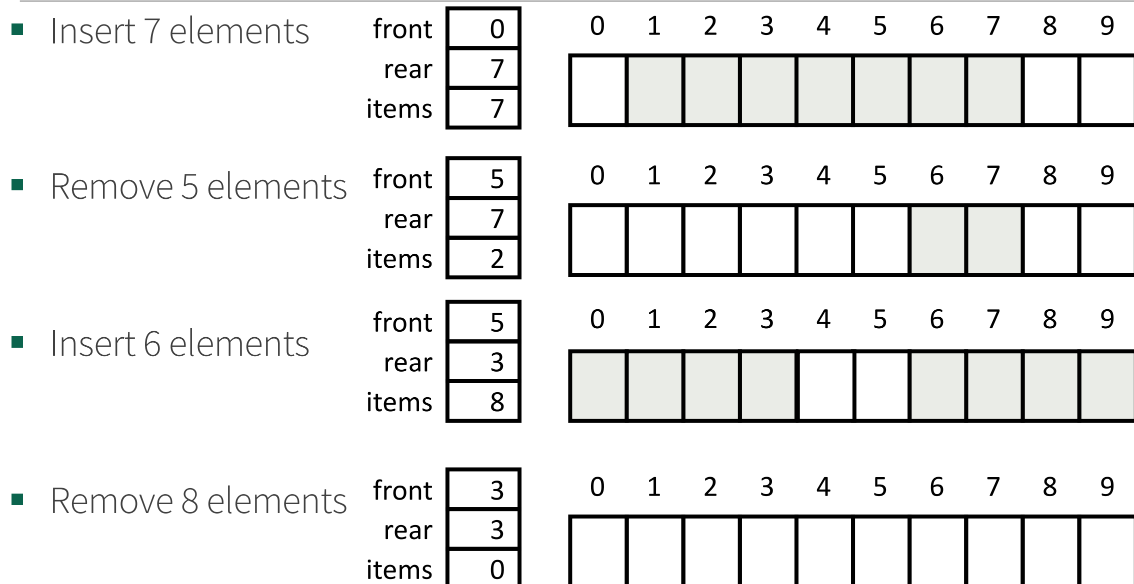


Circular Buffer (n = 10)

- Store elements in array of size n
- items: number of elements in buffer
- Empty buffer:
  - front = rear
- Nonempty buffer
  - rear: index of most recently inserted element
  - front: (index of next element to remove – 1) mod n
- Initially:

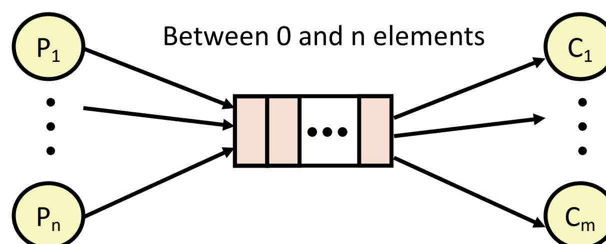


Circular Buffer Operation (n = 10)



Producer-Consumer on an n-element Buffer

- Requires a mutex and two counting semaphores:
  - mutex: enforces mutually exclusive access to the buffer and counters
  - slots: counts the available slots in the buffer
  - items: counts the available items in the buffer
- Makes use of general semaphores
  - Will range in value from 0 to n



Thread Safety

- Functions called from a thread must be thread-safe
- Def: A function is thread-safe iff it will always produce correct results when called repeatedly from multiple concurrent threads.
- Classes of thread-unsafe functions:
  - Class 1: Functions that do not protect shared variables
  - Class 2: Functions that keep state across multiple invocations
  - Class 3: Functions that return a pointer to a static variable
  - Class 4: Functions that call thread-unsafe functions

Thread-Unsafe Functions (Class 1)

- Failing to protect shared variables
  - Fix: Use P and V semaphore operations
  - Issue: Synchronization operations will slow down code

Thread-Unsafe Functions (Class 2)

- Relying on persistent state across multiple function invocations
  - Example: Random number generator that relies on static state

```
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```


Thread-Safe Random Number Generator

- Pass state as part of argument
  - and, thereby, eliminate static state

```
/* rand_r - return pseudo-random integer on 0..32767 */

int rand_r(int *nextp)
{
    *nextp = *nextp*1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

- Consequence: programmer using rand_r must maintain seed

Summary

- Create threads to execute work encapsulated within functions
- Coordinate shared access between threads to avoid race conditions
  - Local storage to avoid conflicts
  - Synchronization objects to organize use