Part 7: Parallel Computation with POSIX Threads

Contents

[DOCUMENT FINALIZED]

Sources: `http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html`, `https://computing.llnl.gov/tutorials/pthreads`, Wikipedia

Parallel Computation

In this age of multi-processor and mulit-core computer architectures it is important to utilize the gained computational power by parallelizing programs.

The programs we have seen thus far are single-threaded — there is only one flow of execution that manipulates data in memory and connected devices. By contrast, multi-threaded applications use multiple execution flows that are independently run on multiple CPUs or cores.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes:

Multi-Core Processors

A multi-core processor is a processor that includes multiple execution units ("cores") on the same chip. A multi-core processor can issue multiple instructions per cycle from multiple instruction streams.

Simultaneous multi-threading (of which Intel's Hyper-

Threading is the best known) was an early form of pseudo-multi-coreism. A processor capable of simultaneous multi-threading has only one execution unit, but when that execution unit is idling (such as during a cache miss), it uses that execution unit to process a second thread. IBM's Cell microprocessor, designed for use in the Sony PlayStation 3, is another prominent multi-core processor.

Symmetric Multiprocessing

A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and communicate via a bus. Bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more than 32 processors.

Distributed Computers

A distributed computer is a distributed memory computer system in which the processing elements are connected by a network. Distributed computers are highly scalable.

A cluster is a group of loosely coupled computers that

work together closely, so that in some respects they can be regarded as a single computer. Clusters are composed of multiple standalone machines connected by a network. The vast majority of the fastest 500 supercomputers are clusters.

In this course we will study the basics of multi-threaded programming. In particular, we will work with the POSIX thread library and discuss how single-threaded programs can be parallelized using threads, how threads can communicate with each other, and how we can protect data from being corrupted by multiple threads trying to access the data concurrently.

POSIX Threads

The POSIX thread library is a standardized thread application program interface for C and C++.

It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing.

Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process.

While most effective on a multiprocessor system, gains are also found on uni-processor systems which exploit latency in I/O and other system functions which may halt process execution. (One thread may execute while another is waiting for I/O or some other system latency.)

Parallel programming technologies such as MPI and PVM are used in a distributed computing environment

while threads are limited to a single computer system.

All threads within a process share the same address space. A thread is spawned by defining a function and its arguments which will be processed in the thread. The purpose of using the POSIX thread library in your software is to execute software faster.

All running threads are executed simultaneously on all available CPU cores. Hundreds of threads my run at the same time, but if your system only has 4 cores, the operating system will schedule all threads to take turns, and at any given time only 4 threads will be active.

If you achieve perfect parallelization of your computation task, you can expect your program to run roughly k times faster on a k-core computer, compared to a single-threaded implementation.

Some tasks such as matrix multiplication are very easy to parallelize — conceptually you just let each core compute a fraction of the entries of the result matrix. Other algorithms are harder to parallelize. For instance, it has been shown that even with n computation cores, one needs at least logarithmic time to add n numbers each

consisting of n bits.

Thread Basics

Thread operations include thread creation, termination, synchronization (join, blocking), scheduling, data management and thread interaction.

A thread does not maintain a list of created threads, nor does it know the thread that created it.

All threads within a process share the same address space, i.e. code, global and heap variables have the same address in all such threads.

Threads in the same process share:

- Code
- Most data (heap and global variables)
- Open files (descriptors)
- Signals and signal handlers
- Current working directory
- User and group id

Each thread has a unique:

- Thread ID
- Set of registers, including stack pointer
- Stack for local variables, return addresses
- Signal mask
- Priority
- Return value: errno

pthread functions return 0 when successful.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_func(void *ptr)
{
  const char *msg = (char *)ptr;
  printf("%s\n", msg);
  // return pointer to thread result, can't be pointing to
  // local variable
  return ptr;
}

int main()
{
  pthread_t t1, t2;
  const char *msg1 = "I am Thread 1";
  const char *msg2 = "I am Thread 2";
  void *ret1, *ret2;

  // Create independent threads each of which will
  // execute thread_func
  pthread_create(&t1, 0, thread_func, (void*)msg1);
  pthread_create(&t2, 0, thread_func, (void*)msg2);
```


Example (Continued)

```
  // Wait till threads are complete before main
  // continues. Unless we wait we run the risk of
  // executing an exit which will terminate the
  // process and all threads before the threads
  // have completed. Thread results are stored in
  // ret1 and ret2.
  pthread_join(t1, &ret1);
  pthread_join(t2, &ret2);

  printf("Thread 1 returns: %s\n", (char*)ret1);
  printf("Thread 2 returns: %s\n", (char*)ret2);
  return 0;
}
```

Compile with: `g++ -lpthread ex1.c`

(`-lpthread` links with pthread library)

Run: `./a.out`

Output:

```
I am Thread 1
I am Thread 2
Thread 1 returns: I am Thread 1
Thread 2 returns: I am Thread 2
```


Details

In this example the same function is used in each thread. The arguments are different. The functions need not be the same.

Threads terminate by explicitly calling `pthread_exit`, by letting the function return, or by a call to the function exit which will terminate the process including any threads.

Threads should not be killed by other threads directly because this can possibly leak resources like memory, file descriptors, and mutexes. Instead, a kill-flag can be used which threads frequently check. If set by a master thread, threads can then clean up and exit voluntarily.

```
bool kill_threads = false;

void *thread(void *data) {
  ...
  while (work_left) {
    if (kill_threads) {
      return retval; // exit thread voluntarily
    }
    ...
  }
}
```

Creating, Joining, and Exiting PThreads

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*function)(void *),
                   void *arg);
```

creates a new thread. Arguments:

- `thread` - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
- `attr` - set to 0 if default thread attributes are used (this is all we need in this course)
- `function` - pointer to the function to be threaded. It has a single argument, which points to data we want the thread to process.
- `arg` - pointer to argument of function. Multiple arguments can be used by passing the address of a struct or class object.


```
int pthread_join(pthread_t th, void **retval);
```

waits for termination of another thread. Arguments:

- `th` - The current thread is suspended until the thread identified by th terminates
- `retval` - If `retval` is not 0, the return value of th is stored in the location pointed to by `retval`.

```
void pthread_exit(void *retval);
```

terminates the calling thread. Arguments:

`retval` - Return value of thread.

This function kills the thread. The `pthread_exit` function never returns.

Note: `*retval` must not point to local data because such data ceases to exist once the thread terminates. Return values can also be stored in the data structure whose address is passed on to `pthread_create`.

Thread Synchronization

The threads library provides three synchronization mechanisms:

- Mutexes — Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.
- Joins — Make a thread wait till others are complete (terminated).
- Condition Variables — Used for waiting until signalled to continue by another thread.

Mutexes

Mutexes are used to prevent data inconsistencies due to operations by multiple threads upon the same memory area performed at the same time or to prevent conditions in which the order of memory operations matters.

A contention or race condition can occur when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed.

Mutexes are used for serializing shared resources such as memory or files.

Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it. One can apply a mutex to protect a segment of memory ("critical region") from other threads. Mutexes can be applied only to threads in a single process and do not work between processes.

Example

```
// without mutex

int counter = 0;

void increment() // possible machine code:
{                 // 1. load counter into register
  counter++       // 2. increment register
}                 // 3. store register to counter
```

```
// with mutex guarding the critical section

pthread_mutex_t counter_mutex =
                      PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

void increment()
{
  pthread_mutex_lock(&counter_mutex);
  counter++
  pthread_mutex_unlock(&counter_mutex);
}
```


Data Inconsistency Example

Suppose `counter=0` and two threads are calling `increment` at virtually the same time. Also suppose that memory accesses are serialized, i.e. only one CPU core can actually access memory at any given time.

Still, without locking, `counter` can be 1 or 2 after finishing both calls. How can this happen?

```
 Thread 1                 Thread 2
----------------------------------------
1. Load to Reg. (0)
                         1. Load to Reg. (0)
2. Incr. Reg. (1)
                         2. Incr. Reg. (1)
3. Store Reg. (1)
                         3. Store Reg. (1)

Result:  counter = 1
```


Good Case

```
 Thread 1                 Thread 2
----------------------------------------
1. Load to Reg. (0)
2. Incr. Reg. (1)       doing something
3. Store Reg. (1)            else

                        1. Load to Reg. (1)
  doing something       2. Incr. Reg. (2)
      else              3. Store Reg. (2)

Result:  counter = 2, as expected!
```

To avoid data corruption or inconsistencies when threads share data we need to ensure that at any given time at most one thread is in the critical section, which in this case is the `counter++` instruction.

Using mutexes accomplishes this goal. When locking a mutex, the runtime system checks whether another thread already locked it. If so, the current thread is suspended and added to a queue of waiting threads.

When unlocking a mutex, a waiting thread is signalled to continue and enter the critical section. In this case the mutex stays locked. If no thread is waiting the mutex is unlocked.

This way, at most one thread executes code in the critical section between `pthread_mutex_lock()` and `pthread_mutex_unlock()`.

```
 Thread 1                 Thread 2
----------------------------------------
   lock mutex            do something
1. Load to Reg. (0)       lock mutex
2. Incr. Reg. (1)            wait
3. Store Reg. (1)            wait
   unlock mutex        continue/still locked
                         1. Load to Reg. (1)
   do something          2. Incr. Reg. (2)
       else              3. Store Reg. (2)
                           unlock mutex
Result:  counter = 2
```

Complete Example

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t counter_mutex =
                      PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

void *increment(void*)
{
  pthread_mutex_lock(&counter_mutex);
  counter++;
  printf("Counter value: %d\n", counter);
  pthread_mutex_unlock(&counter_mutex);
}

int main()
{
  const int TN = 10; // thread number
  pthread_t threads[TN];

  // create TN threads
  for (int i=0; i < TN; i++) {
    pthread_create(&threads[i], 0, increment, 0);
  }
```


Complete Example (Continued)

```
  // wait until all threads are complete before
  // main continues
  for (int i=0; i < TN; i++) {
    pthread_join(threads[i], 0);
  }
  return 0;
}
```

Compile with `g++ -lpthread ex2.c`

Run: `./a.out`

Output:

```
Counter value: 1
Counter value: 2
...
Counter value: 10
```

Everytime, because `counter` is protected by a mutex.

Condition Variables

Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.

Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming because the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.

A condition variable is always used in conjunction with a mutex lock.

A representative sequence for using condition variables is shown below:


```
Main Thread
- Declare and initialize global variables that require
  synchronization (such as "count")
- Declare and initialize a condition variable
- Declare and initialize the associated mutex
- Create threads 1 and 2 to do work
- Join and continue

Thread 1
- Do work up to the point where a certain condition must
  hold (such as "count" reaching a specified value)
- Lock associated mutex and check condition
- If condition doesn't hold call pthread_cond_wait() to
  perform a blocking wait for signal from thread 2. Note
  that a call to pthread_cond_wait() unlocks the associated
  mutex variable so that it can be used by thread 2.
- When signalled, wake up. Mutex is locked. Do some work.
- Explicitly unlock mutex
- Continue and return when done

Thread 2
- Do work
- Lock associated mutex
- Change the value of the global variable that thread 1 is
  waiting upon.
- Check value of the global thread 1 wait variable. If it
  fulfills the desired condition, signal thread 1.
- Unlock mutex.
- Continue and return when done
```

Using Condition Variables

This example code (cond.c) demonstrates the use of condition variables we just described. The main function creates three threads. Two of the threads perform work and update a "count" variable. The third thread waits until the count variable reaches a specified value.

```
Output:
inc_count: thread 1, count = 1, unlocking mutex
Starting watch_count: thread 0
inc_count: thread 2, count = 2, unlocking mutex
inc_count: thread 1, count = 3, unlocking mutex
inc_count: thread 2, count = 4, unlocking mutex
inc_count: thread 1, count = 5, unlocking mutex
inc_count: thread 2, count = 6, unlocking mutex
inc_count: thread 1, count = 7, unlocking mutex
inc_count: thread 2, count = 8, unlocking mutex
inc_count: thread 1, count = 9, unlocking mutex
inc_count: thread 2, count = 10, unlocking mutex
inc_count: thread 1, count = 11, unlocking mutex
inc_count: thread 2, count = 12  Threshold reached.
inc_count: thread 2, count = 12, unlocking mutex
watch_count: thread 0 signal received.
watch_count: thread 0 count now = 137.
inc_count: thread 1, count = 138, unlocking mutex
inc_count: thread 2, count = 139, unlocking mutex
inc_count: thread 1, count = 140, unlocking mutex
inc_count: thread 2, count = 141, unlocking mutex
inc_count: thread 1, count = 142, unlocking mutex
inc_count: thread 2, count = 143, unlocking mutex
inc_count: thread 1, count = 144, unlocking mutex
inc_count: thread 2, count = 145, unlocking mutex
Main(): Waited on 3 threads. Done.
```



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

const int NUM_THREADS = 3;
const int NUM_INC = 10;     // how often inc_count increments
const int COUNT_LIMIT = 12; // when to wake up watch_count
int count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_cond;

// increment counter a few times
// wake up watch_count thread when reaching COUNT_LIMIT
void *inc_count(void *t)
{
  int my_id = *(int *)t;

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

    // check the value of count and signal waiting thread when
    // condition is reached. This occurs while mutex is locked.
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_cond);
      printf("inc_count: thread %d, count = %d  Threshold reached.\n",
             my_id, count);
    }
    printf("inc_count: thread %d, count = %d, unlocking mutex\n",
           my_id, count);
    pthread_mutex_unlock(&count_mutex);

    // do some "work" so threads can alternate on mutex lock
    sleep(1);
  }
  return 0;
}
```



```c
// wait until signalled, then add 125
void *watch_count(void *t)
{
  int my_id = *(int*)t;
  printf("Starting watch_count: thread %d\n", my_id);

  /*
    Lock mutex and wait for signal. pthread_cond_wait will unlock
    mutex while it waits. Also, if COUNT_LIMIT is reached before
    this function 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_cond, &count_mutex);
    printf("watch_count: thread %d signal received.\n", my_id);
    count += 125;
    printf("watch_count: thread %d count now = %d.\n", my_id, count);
  }
  pthread_mutex_unlock(&count_mutex);
  return 0;
}
```



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

  // initialize mutex and condition variable objects
  pthread_mutex_init(&count_mutex, 0);
  pthread_cond_init(&count_cond, 0);

  // create threads
  for (int i=0; i < NUM_THREADS; i++) {
    ids[i] = i;
    if (!i) {
      pthread_create(&threads[i], 0, watch_count, (void *)&ids[i]);
    } else {
      pthread_create(&threads[i], 0, inc_count, (void *)&ids[i]);
    }
  }

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

  // clean up and exit
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_cond);
  return 0;
}
```

Condition Variable Details

Functions for creating and destroying condition variables:

```
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *cond,
             const pthread_condattr_t *attr);
```

Condition variables must be defined with type `pthread_cond_t`, and must be initialized before they can be used. There are two ways to initialize a condition variable:

1. Statically, when it is defined. For example:

```
pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
```

2. Dynamically, with the `pthread_cond_init()` function. 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. We ignore attributes in this course, and pass on 0 instead.

`pthread_cond_destroy()` should be used to free a condition variable that is no longer needed.

Functions for waiting and signalling on condition variables:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

`pthread_cond_wait()` blocks the calling thread until the specified condition is signalled. This function should be called while mutex is locked, and it will automatically release the mutex while it waits. After a signal is received and the thread is awakened, mutex will be automatically locked for use by the thread. The programmer is then responsible for unlocking mutex when the thread is finished with it.

The `pthread_cond_signal()` function is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked. The mutex must be unlocked after, for `pthread_cond_wait()` to complete.

The `pthread_cond_broadcast()` function should be used instead of `pthread_cond_signal()` if more than one thread is in a blocking wait state. All waiting threads will be woken up.

It is a logical error to call `pthread_cond_signal()` before calling `pthread_cond_wait()`, because in this case the signal will not be received — it's lost.

Proper locking and unlocking of the associated mutex variable is essential when using these functions. For example:

- Failing to lock the mutex before calling `pthread_cond_wait()` may cause it NOT to block.
- Failing to unlock the mutex after calling `pthread_cond_signal()` may not allow a matching `pthread_cond_wait()` function to complete (it will remain blocked).

Producer/Consumer

When the processing time for individual work items varies or work items become available only one after another, using a producer/consumer (or writer/reader) threading framework can improve CPU utilization.

A possible implementation spawns writer threads that generate work items and add them to a queue (a dynamic first-in-first-out data structure), from which reader threads remove work items and process them.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <pthread.h>
#include "Queue.h"

pthread_mutex_t mutex;
pthread_cond_t item_arrived_cond;
pthread_cond_t item_read_cond;

// data guarded by mutex
Queue queue(100);    // contains list of work items
bool kill_all;       // true => ask readers to quit
int items_to_be_processed; // how many items in total
int items_processed; // number of processed items
```

```
// writer thread code that produces work items
void *writer(void *job)
{
  // how many items to create?
  int num_items = *(int*)job;

  for (int i=0; i < num_items; i++) {
    pthread_mutex_lock(&mutex);
    // wait until there is space in queue
    while (queue.full()) {
      pthread_cond_wait(&item_read_cond, &mutex);
    }
    queue.add(i);
    items_to_be_processed++;
    // signal to one waiting reader that work
    // has arrived
    pthread_cond_signal(&item_arrived_cond);
    pthread_mutex_unlock(&mutex);
  }
  return 0;
}
```



```
// reader thread code that consumes work items
// and processes them
void *reader(void *)
{
  for (;;) {
    pthread_mutex_lock(&mutex);
    while (queue.empty() && !kill_all) {
      // wait here when queue is empty and
      // termination not requested
      pthread_cond_wait(&item_arrived_cond,
                                           &mutex);
    }

    if (kill_all) {
      // work done - quit thread
      pthread_mutex_unlock(&mutex);
      break;
    }

    // retrieve one work item
    int work = queue.remove();

    // let others access the queue
    pthread_mutex_unlock(&mutex);
```



```
    // process work item
    {
      // be busy for a while ...
      volatile int a = work;
      // volatile: a not loaded into register
      // and compiler can't remove the following
      // useless code
      for (int i=0; i < 1000000; i++) {
        a += i*3 + i*i;
      }
      // in production code, the result would
      // now be stored somewhere ...
    }

    // signal to a waiting writer that item
    // has been processed
    pthread_mutex_lock(&mutex);
    items_processed++;
    pthread_cond_signal(&item_read_cond);
    pthread_mutex_unlock(&mutex);
  }

  return 0;
}
```



```
int main(int argc, char *argv[])
{
  if (argc != 4) {
    fprintf(stderr, "call: %s #writers #readers #items\n",
                    argv[0]);
    exit(10);
  }

  int num_writers = 2;
  int num_readers = 8;
  int num_items   = 1000;
  num_writers = atoi(argv[1]);
  num_readers = atoi(argv[2]);
  num_items   = atoi(argv[3]);
  printf("writers:%d, readers:%d items:%d\n",
         num_writers, num_readers, num_items);
  pthread_cond_init(&item_arrived_cond, 0);
  pthread_cond_init(&item_read_cond, 0);
  pthread_t writers[num_writers], readers[num_readers];
  int jobs[num_writers];

  queue.reset();
  items_processed = items_to_be_processed = 0;
  kill_all = false;

  // spawn reader threads
  for (int i=0; i < num_readers; i++) {
    pthread_create(&readers[i], 0, reader, 0);
  }
```

```
  // spawn writer threads
  int items_per_writer = num_items / num_writers;
  int rem = num_items % num_writers;
  for (int i=0; i < num_writers; i++) {
    jobs[i] = items_per_writer;
    if (rem-- > 0) { // distribute remaining items
      jobs[i]++;
    }
    pthread_create(&writers[i], 0, writer, &jobs[i]);
  }

  // wait for all writers to finish
  for (int i=0; i < num_writers; i++) {
    pthread_join(writers[i], 0);
  }

  // wait for all readers to finish their work
  for (;;) {
    pthread_mutex_lock(&mutex);
    if (items_processed == items_to_be_processed) {
      // done
      pthread_mutex_unlock(&mutex);
      break;
    }
    // wait for one reader to read item
    pthread_cond_wait(&item_read_cond, &mutex);
    pthread_mutex_unlock(&mutex);
  }
```



```
  // readers are now wating for new work, but we are done
  // so we ask them to quit
  printf("terminate readers\n");

  // wake up readers to make them see that kill_all
  // has changed
  pthread_mutex_lock(&mutex);
  kill_all = true;
  pthread_cond_broadcast(&item_arrived_cond);
  pthread_mutex_unlock(&mutex);

  // wait for all reader threads
  for (int i=0; i < num_readers; i++) {
    pthread_join(readers[i], 0);
  }

  printf("processed: %d, to_be_processed: %d\n",
         items_processed, items_to_be_processed);
}
```



```
// simple integer queue data structure supporting
//
//  constructor(capacity) allocates capacity elements
//  reset()               empties queue
//  add(x)                adds element x at tail
//  remove()              removes and returns head element
//  empty()               true iff empty
//  full()                true iff full

class Queue
{
public:
  // initializes empty queue with maximal c elements
  Queue(int c) {
    capacity = c;
    data = new int[capacity];
    reset();
  }

  ~Queue() { delete [] data; }

  // empties queue
  void reset() { head = tail = n = 0; }

  // return true iff queue is empty
  bool empty() { return n == 0; }

  // return true iff queue is full
  bool full() { return n >= capacity; }
```



```
  // add element to queue (at tail)
  // pre-condition: not full
  void add(int x) {
    assert(!full());
    data[tail++] = x;
    if (tail >= capacity)
      tail = 0;
    n++;
  }

  // remove and return head element
  // pre-condition: not empty
  int remove() {
    assert(!empty());
    int x = data[head++];
    if (head >= capacity)
      head = 0;
    n--;
    return x;
  }

private:
  int capacity;   // maximum number of elements
  int *data;      // pointer to element array
  int head, tail; // current remove/add locations
  int n;          // actual number of elements stored
};
```

Code available in `lec-week11/writer_reader`

Deadlocks and Livelocks

A deadlock is a situation which occurs when a process or thread enters a waiting state because a resource requested by it is being held by another waiting process, which in turn is waiting for another resource. If a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process, then the system is said to be in a deadlock.

As an example, suppose a computer has three CD drives and three processes. Each of the three processes holds one of the drives. If each process now requests another drive, the three processes will be in a deadlock. Each process will be waiting for the "CD drive released" event, which can only be caused by one of the other waiting processes. Thus, it results in a circular chain.

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific

process is not progressing.

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

Livelock is a risk with some algorithms that detect and recover from a deadlock. If more than one process takes action, the deadlock detection algorithm can be repeatedly triggered. This can be avoided by ensuring that only one process (chosen randomly or by priority) takes action.

Deadlock Example

```
#include <pthread.h>
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void *simple_thread(void *)
{
  pthread_mutex_lock(&mutex2);   // lock mutex2
  pthread_mutex_lock(&mutex1);   // lock mutex1
  pthread_mutex_unlock(&mutex1); // unlock mutex1
  pthread_mutex_unlock(&mutex2); // unlock mutex2
  return 0;
}

int main()
{
  pthread_t tid;

  // create a thread
  pthread_create(&tid, 0, &simple_thread, 0);

  pthread_mutex_lock(&mutex1);   // lock mutex1
  pthread_mutex_lock(&mutex2);   // lock mutex2
  pthread_mutex_unlock(&mutex2); // unlock mutex2
  pthread_mutex_unlock(&mutex1); // unlock mutex1

  // wait for thread to finish
  pthread_join(tid, 0);
}
```


In this example, two threads lock two mutexes in different orders.

This creates a deadlock because both threads will be successful acquiring their first respective mutexes, but then block on the second call, where both will wait for the other thread to unlock the mutex — which will never happen.

This situation could have been prevented by locking mutexes in the same order:

```
void *simple_thread(void *)
{
  pthread_mutex_lock(&mutex1);   // lock mutex1
  pthread_mutex_lock(&mutex2);   // lock mutex2
  ...

int main()
{
  ...
  pthread_mutex_lock(&mutex1);   // lock mutex1
  pthread_mutex_lock(&mutex2);   // lock mutex2
  ...
```

Now one of the threads will block on the first call, and proceed once the other thread is done!

In general, deadlocks can be prevented by imposing an order on mutexes and only locking mutexes in that order.

Only rarely will mutexes be locked in consecutive lines of code, and debugging deadlocks may therefore be much harder in more complex projects.

Unlike "silent" data corruption which may happen in data race situations, deadlocks manifest themselves explicitly: the program just freezes. But of course, like data races, deadlocks may not happen in any particular program test run, thread scheduling depends on external factors such as other programs running at the same time.

The `valgrind` tool `helgrind` can help identify the lines of code which may potentially cause deadlocks, e.g. by running

```
valgrind --tool=helgrind a.out
```

Your program should be compiled with `-g` to give `valgrind` access to source code information. Try it with the deadlock example mentioned earlier!