1. Producer/consumer example [also known as bounded buffer]

1a. Buggy implementation

```c
/*
  buffer stores BUFFER_SIZE items
  count is number of used slots, a variable that lives in memory
  out is next empty buffer slot to fill (if any)
  in is oldest filled slot to consume (if any)
*/

void producer (void *ignored) {
    for (;;) {
        /* next line produces an item and puts it in nextProduced */
        nextProduced = means_of_production();
        while (count == BUFFER_SIZE)
            ; // do nothing
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
    }
}

void consumer (void *ignored) {
    for (;;) {
        while (count == 0)
            ; // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--; /* next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}
```

--Review: what’s the problem?
--Answer: count++ and count-- might compile to, respectively:

reg1 <−− count      # load
reg1 <−− reg1 + 1   # increment register
count <−− reg1      # store
reg2 <−− count      # load
reg2 <−− reg2 - 1   # decrement register
count <−− reg2      # store

and then if we get the following interleaving, *count* is incorrect:

reg1 <−− count
reg1 <−− reg1 + 1
reg2 <−− count
reg2 <−− reg2 - 1
count <−− reg1
count <−− reg2

--Review: why not use instructions like "addl $0x1, _count"?
--Answer: not atomic if there are multiple CPUs.

--Review: so why not use "LOCK addl $0x1, _count"?
--Answer: we could do that here, but LOCK won’t save us every time

--Review: recall that a more general-purpose approach to protecting critical sections is to use locks. What is the interface to locks?
--Answer: lock.acquire() and lock.release()

[or mutex.acquire() and mutex.release()]

1b. Producer/consumer [bounded buffer] using mutexes

Mutex mutex;

void producer (void *ignored) {
    for (;;) {
        /* next line produces an item and puts it in nextProduced */
        nextProduced = means_of_production();
        acquire(&mutex);
        while (count == BUFFER_SIZE) {
            release(&mutex);
            yield(); /* or schedule() */
            acquire(&mutex);
        }
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        release(&mutex);
    }
}

void consumer (void *ignored) {
    for (;;) {
        acquire(&mutex);
        while (count == 0)
            ; // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--; /* next line abstractly consumes the item */
        consume_item(nextConsumed);
        release(&mutex);
    }
}
lc. Producer/consumer [bounded buffer] using mutexes and condition variables

```c
Mutex mutex;
Cond nonempty;
Cond nonfull;

void producer (void *ignored) {
    for (;;) {
        nextProduced = means_of_production();
        acquire(&mutex);
        while (count == BUFFER_SIZE)
            cond_wait(&nonfull, &mutex);
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal(&nonempty);
        release(&mutex);
    }
}

void consumer (void *ignored) {
    for (;;) {
        acquire(&mutex);
        while (count == 0)
            cond_wait(&nonempty, &mutex);
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal(&nonfull);
        release(&mutex);
    }
}
```

Question: why does cond_wait need to both release the mutex and sleep? Why not:

```c
    while (count == BUFFER_SIZE) {
        release(&mutex);
        cond_wait(&nonfull);
        acquire(&mutex);
    }
```

Id. Producer/consumer [bounded buffer] with semaphores

```c
Semaphore mutex(1); /* mutex initialized to 1 */
Semaphore empty(BUFFER_SIZE); /* start with BUFFER_SIZE empty slots */
Semaphore full(0); /* 0 full slots */

void producer (void *ignored) {
    for (;;) {
        nextProduced = means_of_production();
        /* next line produces an item and puts it in nextProduced */
        sem_down(&empty);
        sem_down(&mutex);  /* get exclusive access */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        sem_up(&mutex);
        sem_up(&full);   /* we just increased the # of full slots */
    }
}

void consumer (void *ignored) {
    for (;;) {
        /* next line diminishes the count of empty slots and */
        /* waits if there are no empty slots */
        sem_down(&empty);
        sem_down(&mutex);
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_up(&mutex);
        sem_up(&empty);   /* one further empty slot */
    }
}
```

Semaphores *can* (not always) lead to elegant solutions (notice that the code above is fewer lines than lc) but they are much harder to use.

The fundamental issue is that semaphores make implicit (counts, conditions, etc.) what is probably best left explicit. Moreover, they *also* implement mutual exclusion.

For this reason, you should not use semaphores. This example is here mainly for completeness and so you know what a semaphore is. But do not code with them. Solutions that use semaphores in this course will receive no credit.
### Example of a monitor: MyBuffer

```cpp
// This is pseudocode that is inspired by C++.
// Don’t take it literally.

class MyBuffer {
  public:
    MyBuffer();
    ~MyBuffer();
    void Enqueue(Item);
    Item Dequeue();
  private:
    int count;
    int in;
    int out;
    Item buffer[BUFFER_SIZE];
    Mutex* mutex;
    Cond* nonempty;
    Cond* nonfull;
};

void MyBuffer::MyBuffer()
{
  in = out = count = 0;
  mutex = new Mutex;
  nonempty = new Cond;
  nonfull = new Cond;
}

void MyBuffer::Enqueue(Item item)
{
  mutex.acquire();
  while (count == BUFFER_SIZE)
    cond_wait(&nonfull, &mutex);
  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
  ++count;
  cond_signal(&nonempty, &mutex);
  mutex.release();
}

Item MyBuffer::Dequeue()
{
  mutex.acquire();
  while (count == 0)
    cond_wait(&nonempty, &mutex);
  Item ret = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  --count;
  cond_signal(&nonfull, &mutex);
  mutex.release();
}

int main(int, char**)
{
  MyBuffer buf;
  int dummy;
  tid1 = thread_create(consumer, &buf);
  tid2 = thread_create(consumer, &buf);
  thread_join(tid1);
  // never reach this point
  return -1;
}

void producer(void* buf)
{
  MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
  for (;;) {
    Item nextProduced = means_of_production();
    sharedbuf->Enqueue(nextProduced);
  }
}

void consumer(void* buf)
{
  MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
  for (;;) {
    Item nextConsumed = sharedbuf->Dequeue();
    consume_item(nextConsumed);
  }
}

Key point: *Threads* (the producer and consumer) are separate from
*shared object* (MyBuffer). The synchronization happens in the
shared object.
```
This file handles the generic file mmap semantics used by
most "normal" filesystems (but you don't have to use this:
the NFS filesystem used to do this differently, for example)

#include <linux/config.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/compiler.h>
#include <linux/fs.h>
#include <linux/aio.h>
#include <linux/capability.h>
#include <linux/kernel_stat.h>
#include <linux/mm.h>
#include <linux/swap.h>
#include <linux/mman.h>
#include <linux/pagemap.h>
#include <linux/file.h>
#include <linux/mm.h>
#include <linux/hash.h>
#include <linux/writeback.h>
#include <linux/pagemap.h>
#include <linux/file.h>
#include <linux/slab.h>
#include <linux/capability.h>
#include <linux/vmalloc.h>
#include <asm/uaccess.h>
#include <asm/mman.h>

static ssize_t
generic_file_direct_IO(int rw, struct kiocb *iocb, const struct iovec *iov,
loff_t offset, unsigned long nr_segs);

/*
 * FIXME: remove all knowledge of the buffer layer from the core VM
 */
#include <linux/buffer_head.h> /* for generic_osync_inode */

static
/*
 * FILEME: remove all knowledge of the buffer layer from the core VM
 */
#include <linux/buffer_head.h> /* for generic_osync_inode */
#include <asm/uaccess.h>
#include <asm/mman.h>

static ssize_t
generic_file_direct_IO(int rw, struct kiocb *iocb, const struct iovec *iov,
loff_t offset, unsigned long nr_segs);

/*
 * Shared mappings implemented 30.11.1994. It's not fully working yet,
 * though.
 */

static

/*
 * Lock ordering:
 */

void __remove_from_page_cache(struct page *page)