```
Handout for CS 372H
Class 10
18 February 2010

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

    1a. Buggy implementation

    /*
    "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) {
                release(&mutex);
                yield(); /* or schedule() */
                acquire(&mutex);
             }

             nextConsumed = buffer[out];
             out = (out + 1) % BUFFER_SIZE;
             count−−;
             release(&mutex);

             /* next line abstractly consumes the item */
             consume_item(nextConsumed);
         }
     }
```

```
    1c. Producer/consumer [bounded buffer] using mutexes and condition
    variables

         Mutex mutex;
         Cond nonempty;
         Cond nonfull;

         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)
                    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);

                 /* next line abstractly consumes the item */
                 consume_item(nextConsumed);
             }
         }


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

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



```
    1d.  Producer/consumer [bounded buffer] with semaphores

        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 (;;) {
                 /* next line produces an item and puts it in nextProduced */
                 nextProduced = means_of_production();

                /*
                 * next line diminishes the count of empty slots and
                 * waits if there are no empty slots
                 */
                 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 full slots and
                  * waits if there are no full slots
                  */
                 sem_down(&full);
                 sem_down(&mutex);

                 nextConsumed = buffer[out];
                 out = (out + 1) % BUFFER_SIZE;

                 sem_up(&mutex);
                 sem_up(&empty);   /* one further empty slot */

                 /* next line abstractly consumes the item */
                 consume_item(nextConsumed);
             }
         }

        Semaphores *can* (not always) lead to elegant solutions (notice
        that the code above is fewer lines than 1c) 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.
```

2. Example of a monitor: MyBuffer

```
    // 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(producer, &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 (;;) {
            /* next line produces an item and puts it in nextProduced */
            Item nextProduced = means_of_production();
            sharedbuf−>Enqueue(nextProduced);
        }
    }

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

            /* next line abstractly consumes the item */
            consume_item(nextConsumed);
        }
    }
```

Key point: *Threads* (the producer and consumer) are separate from *shared object* (MyBuffer). The synchronization happens in the shared object.

```c
/*
 *      linux/mm/filemap.c
 *
 * Copyright (C) 1994-1999  Linus Torvalds
 */

/*
 * 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/uio.h>
#include <linux/hash.h>
#include <linux/writeback.h>
#include <linux/pagevec.h>
#include <linux/blkdev.h>
#include <linux/security.h>
#include <linux/syscalls.h>
#include "filemap.h"
/*
 * FIXME: 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.
 *
 * Shared mappings now work. 15.8.1995  Bruno.
 *
 * finished 'unifying' the page and buffer cache and SMP-threaded the
 * page-cache, 21.05.1999, Ingo Molnar <mingo@redhat.com>
 *
 * SMP-threaded pagemap-LRU 1999, Andrea Arcangeli <andrea@suse.de>
 */

/*
 * Lock ordering:
 *
 *  ->i_mmap_lock               (vmtruncate)
 *    ->private_lock            (__free_pte->__set_page_dirty_buffers)
 *      ->swap_lock             (exclusive_swap_page, others)
 *        ->mapping->tree_lock
 *
 *  ->i_mutex
 *    ->i_mmap_lock             (truncate->unmap_mapping_range)
 *
 *  ->mmap_sem
 *    ->i_mmap_lock
 *      ->page_table_lock or pte_lock   (various, mainly in memory.c)
 *        ->mapping->tree_lock  (arch-dependent flush_dcache_mmap_lock)
 *
 *  ->mmap_sem
```



```c
 *    ->lock_page               (access_process_vm)
 *
 *  ->mmap_sem
 *    ->i_mutex                 (msync)
 *
 *  ->i_mutex
 *    ->i_alloc_sem             (various)
 *
 *  ->inode_lock
 *    ->sb_lock                 (fs/fs-writeback.c)
 *    ->mapping->tree_lock      (__sync_single_inode)
 *
 *  ->i_mmap_lock
 *    ->anon_vma.lock           (vma_adjust)
 *
 *  ->anon_vma.lock
 *    ->page_table_lock or pte_lock     (anon_vma_prepare and various)
 *
 *  ->page_table_lock or pte_lock
 *    ->swap_lock               (try_to_unmap_one)
 *    ->private_lock            (try_to_unmap_one)
 *    ->tree_lock               (try_to_unmap_one)
 *    ->zone.lru_lock           (follow_page->mark_page_accessed)
 *    ->zone.lru_lock           (check_pte_range->isolate_lru_page)
 *    ->private_lock            (page_remove_rmap->set_page_dirty)
 *    ->tree_lock               (page_remove_rmap->set_page_dirty)
 *    ->inode_lock              (page_remove_rmap->set_page_dirty)
 *    ->inode_lock              (zap_pte_range->set_page_dirty)
 *    ->private_lock            (zap_pte_range->__set_page_dirty_buffers)
 *
 *  ->task->proc_lock
 *    ->dcache_lock             (proc_pid_lookup)
 */

/*
 * Remove a page from the page cache and free it. Caller has to make
 * sure the page is locked and that nobody else uses it - or that usage
 * is safe.  The caller must hold a write_lock on the mapping's tree_lock.
 */
void __remove_from_page_cache(struct page *page)
{
        struct address_space *mapping = page->mapping;

```