

Feb 18, 10 15:10

I10-handout.txt

Page 1/8

```

1 Handout for CS 372H
2 Class 10
3 18 February 2010
4
5 1. Producer/consumer example [also known as bounded buffer]
6
7     1a. Buggy implementation
8
9     /*
10    "buffer" stores BUFFER_SIZE items
11    "count" is number of used slots. a variable that lives in memory
12    "out" is next empty buffer slot to fill (if any)
13    "in" is oldest filled slot to consume (if any)
14    */
15
16    void producer (void *ignored) {
17        for (;;) {
18            /* next line produces an item and puts it in nextProduced */
19            nextProduced = means_of_production();
20            while (count == BUFFER_SIZE)
21                ; // do nothing
22            buffer [in] = nextProduced;
23            in = (in + 1) % BUFFER_SIZE;
24            count++;
25        }
26    }
27
28    void consumer (void *ignored) {
29        for (;;) {
30            while (count == 0)
31                ; // do nothing
32            nextConsumed = buffer[out];
33            out = (out + 1) % BUFFER_SIZE;
34            count--;
35            /* next line abstractly consumes the item */
36            consume_item(nextConsumed);
37        }
38    }
39
40    --Review: what's the problem?
41    --Answer: count++ and count-- might compile to, respectively:
42
43        reg1 <-- count      # load
44        reg1 <-- reg1 + 1   # increment register
45        count <-- reg1     # store
46
47        reg2 <-- count      # load
48        reg2 <-- reg2 - 1   # decrement register
49        count <-- reg2     # store
50
51    and then if we get the following interleaving, "count" is
52    incorrect:
53
54        reg1 <-- count
55        reg1 <-- reg1 + 1
56        reg2 <-- count
57        reg2 <-- reg2 - 1
58        count <-- reg1
59        count <-- reg2
60
61    --Review: why not use instructions like "addl $0x1, _count"?
62    --Answer: not atomic if there are multiple CPUs.
63
64    --Review: so why not use "LOCK addl $0x1, _count"?
65    --Answer: we could do that here, but LOCK won't save us every time
66
67    --Review: recall that a more general-purpose approach to protecting
68    critical sections is to use locks. What is the interface to locks?
69    --Answer: lock.acquire() and lock.release()
70            [or mutex.acquire() and mutex.release()]
71

```

Feb 18, 10 15:10

I10-handout.txt

Page 2/8

```

72
73     1b. Producer/consumer [bounded buffer] using mutexes
74
75     Mutex mutex;
76
77     void producer (void *ignored) {
78         for (;;) {
79             /* next line produces an item and puts it in nextProduced */
80             nextProduced = means_of_production();
81
82             acquire(&mutex);
83             while (count == BUFFER_SIZE) {
84                 release(&mutex);
85                 yield(); /* or schedule() */
86                 acquire(&mutex);
87             }
88
89             buffer [in] = nextProduced;
90             in = (in + 1) % BUFFER_SIZE;
91             count++;
92             release(&mutex);
93         }
94     }
95
96     void consumer (void *ignored) {
97         for (;;) {
98
99             acquire(&mutex);
100            while (count == 0) {
101                release(&mutex);
102                yield(); /* or schedule() */
103                acquire(&mutex);
104            }
105
106            nextConsumed = buffer[out];
107            out = (out + 1) % BUFFER_SIZE;
108            count--;
109            release(&mutex);
110
111            /* next line abstractly consumes the item */
112            consume_item(nextConsumed);
113        }
114    }
115

```

Feb 18, 10 15:10

I10-handout.txt

Page 3/8

```

116
117 1c. Producer/consumer [bounded buffer] using mutexes and condition
118 variables
119
120     Mutex mutex;
121     Cond nonempty;
122     Cond nonfull;
123
124     void producer (void *ignored) {
125         for (;;) {
126             /* next line produces an item and puts it in nextProduced */
127             nextProduced = means_of_production();
128
129             acquire(&mutex);
130             while (count == BUFFER_SIZE)
131                 cond_wait(&nonfull, &mutex);
132
133             buffer [in] = nextProduced;
134             in = (in + 1) % BUFFER_SIZE;
135             count++;
136             cond_signal(&nonempty);
137             release(&mutex);
138         }
139     }
140
141     void consumer (void *ignored) {
142         for (;;) {
143
144             acquire(&mutex);
145             while (count == 0)
146                 cond_wait(&nonempty, &mutex);
147
148             nextConsumed = buffer[out];
149             out = (out + 1) % BUFFER_SIZE;
150             count--;
151             cond_signal(&nonfull);
152             release(&mutex);
153
154             /* next line abstractly consumes the item */
155             consume_item(nextConsumed);
156         }
157     }
158
159     Question: why does cond_wait need to both release the mutex and
160     sleep? Why not:
161
162         while (count == BUFFER_SIZE) {
163             release(&mutex);
164             cond_wait(&nonfull);
165             acquire(&mutex);
166         }
167
168

```

Feb 18, 10 15:10

I10-handout.txt

Page 4/8

```

169 1d. Producer/consumer [bounded buffer] with semaphores
170
171     Semaphore mutex(1); /* mutex initialized to 1 */
172     Semaphore empty(BUFFER_SIZE); /* start with BUFFER_SIZE empty slots */
173     Semaphore full(0); /* 0 full slots */
174
175     void producer (void *ignored) {
176         for (;;) {
177             /* next line produces an item and puts it in nextProduced */
178             nextProduced = means_of_production();
179
180             /*
181              * next line diminishes the count of empty slots and
182              * waits if there are no empty slots
183              */
184             sem_down(&empty);
185             sem_down(&mutex); /* get exclusive access */
186
187             buffer [in] = nextProduced;
188             in = (in + 1) % BUFFER_SIZE;
189
190             sem_up(&mutex);
191             sem_up(&full); /* we just increased the # of full slots */
192         }
193     }
194
195     void consumer (void *ignored) {
196         for (;;) {
197
198             /*
199              * next line diminishes the count of full slots and
200              * waits if there are no full slots
201              */
202             sem_down(&full);
203             sem_down(&mutex);
204
205             nextConsumed = buffer[out];
206             out = (out + 1) % BUFFER_SIZE;
207
208             sem_up(&mutex);
209             sem_up(&empty); /* one further empty slot */
210
211             /* next line abstractly consumes the item */
212             consume_item(nextConsumed);
213         }
214     }
215
216     Semaphores *can* (not always) lead to elegant solutions (notice
217     that the code above is fewer lines than 1c) but they are much
218     harder to use.
219
220     The fundamental issue is that semaphores make implicit (counts,
221     conditions, etc.) what is probably best left explicit. Moreover,
222     they *also* implement mutual exclusion.
223
224     For this reason, you should not use semaphores. This example is
225     here mainly for completeness and so you know what a semaphore
226     is. But do not code with them. Solutions that use semaphores in
227     this course will receive no credit.
228

```

Feb 18, 10 15:10

I10-handout.txt

Page 5/8

```

229 2. Example of a monitor: MyBuffer
230
231 // This is pseudocode that is inspired by C++.
232 // Don't take it literally.
233
234 class MyBuffer {
235     public:
236         MyBuffer();
237         ~MyBuffer();
238         void Enqueue(Item);
239         Item = Dequeue();
240     private:
241         int count;
242         int in;
243         int out;
244         Item buffer[BUFFER_SIZE];
245         Mutex* mutex;
246         Cond* nonempty;
247         Cond* nonfull;
248     }
249
250 void
251 MyBuffer::MyBuffer()
252 {
253     in = out = count = 0;
254     mutex = new Mutex;
255     nonempty = new Cond;
256     nonfull = new Cond;
257 }
258
259 void
260 MyBuffer::Enqueue(Item item)
261 {
262     mutex.acquire();
263     while (count == BUFFER_SIZE)
264         cond_wait(&nonfull, &mutex);
265
266     buffer[in] = item;
267     in = (in + 1) % BUFFER_SIZE;
268     ++count;
269     cond_signal(&nonempty, &mutex);
270     mutex.release();
271 }
272
273 Item
274 MyBuffer::Dequeue()
275 {
276     mutex.acquire();
277     while (count == 0)
278         cond_wait(&nonempty, &mutex);
279
280     Item ret = buffer[out];
281     out = (out + 1) % BUFFER_SIZE;
282     --count;
283     cond_signal(&nonfull, &mutex);
284     mutex.release();
285 }
286

```

Feb 18, 10 15:10

I10-handout.txt

Page 6/8

```

287 int main(int, char**)
288 {
289     MyBuffer buf;
290     int dummy;
291     tid1 = thread_create(producer, &buf);
292     tid2 = thread_create(consumer, &buf);
293     thread_join(tid1);
294
295     // never reach this point
296     return -1;
297 }
298
299 void producer(void* buf)
300 {
301     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
302     for (;;) {
303         /* next line produces an item and puts it in nextProduced */
304         Item nextProduced = means_of_production();
305         sharedbuf->Enqueue(nextProduced);
306     }
307 }
308
309 void consumer(void* buf)
310 {
311     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
312     for (;;) {
313         Item nextConsumed = sharedbuf->Dequeue();
314
315         /* next line abstractly consumes the item */
316         consume_item(nextConsumed);
317     }
318 }
319
320 Key point: *Threads* (the producer and consumer) are separate from
321 *shared object* (MyBuffer). The synchronization happens in the
322 shared object.
323

```

Feb 18, 10 15:10 I10-handout.txt Page 7/8

```

324 /*
325 *      linux/mm/filemap.c
326 *
327 * Copyright (C) 1994-1999 Linus Torvalds
328 */
329
330 /*
331 * This file handles the generic file mmap semantics used by
332 * most "normal" filesystems (but you don't /have/ to use this:
333 * the NFS filesystem used to do this differently, for example)
334 */
335 #include <linux/config.h>
336 #include <linux/module.h>
337 #include <linux/slab.h>
338 #include <linux/compiler.h>
339 #include <linux/fs.h>
340 #include <linux/aio.h>
341 #include <linux/capability.h>
342 #include <linux/kernel_stat.h>
343 #include <linux/mm.h>
344 #include <linux/swap.h>
345 #include <linux/mman.h>
346 #include <linux/pagemap.h>
347 #include <linux/file.h>
348 #include <linux/uio.h>
349 #include <linux/hash.h>
350 #include <linux/writeback.h>
351 #include <linux/pagevec.h>
352 #include <linux/blkdev.h>
353 #include <linux/security.h>
354 #include <linux/syscalls.h>
355 #include "filemap.h"
356 /*
357 * FIXME: remove all knowledge of the buffer layer from the core VM
358 */
359 #include <linux/buffer_head.h> /* for generic_osync_inode */
360
361 #include <asm/uaccess.h>
362 #include <asm/mman.h>
363
364 static ssize_t
365 generic_file_direct_IO(int rw, struct kiocb *iocb, const struct iovec *iov,
366                       loff_t offset, unsigned long nr_segs);
367
368 /*
369 * Shared mappings implemented 30.11.1994. It's not fully working yet,
370 * though.
371 *
372 * Shared mappings now work. 15.8.1995 Bruno.
373 *
374 * finished 'unifying' the page and buffer cache and SMP-threaded the
375 * page-cache, 21.05.1999, Ingo Molnar <mingo@redhat.com>
376 *
377 * SMP-threaded pagemap-LRU 1999, Andrea Arcangeli <andrea@suse.de>
378 */
379
380 /*
381 * Lock ordering:
382 *
383 * ->i_mmap_lock          (vmtruncate)
384 * ->private_lock        (__free_pte->__set_page_dirty_buffers)
385 * ->swap_lock           (exclusive_swap_page, others)
386 * ->mapping->tree_lock
387 *
388 * ->i_mutex
389 * ->i_mmap_lock          (truncate->unmap_mapping_range)
390 *
391 * ->mmap_sem
392 * ->i_mmap_lock
393 * ->page_table_lock or pte_lock (various, mainly in memory.c)
394 * ->mapping->tree_lock (arch-dependent flush_dcache_mmap_lock)
395 *
396 * ->mmap_sem

```

Feb 18, 10 15:10 I10-handout.txt Page 8/8

```

397 * ->lock_page          (access_process_vm)
398 *
399 * ->mmap_sem
400 * ->i_mutex            (msync)
401 *
402 * ->i_mutex
403 * ->i_alloc_sem        (various)
404 *
405 * ->inode_lock
406 * ->sb_lock            (fs/fs-writeback.c)
407 * ->mapping->tree_lock (__sync_single_inode)
408 *
409 * ->i_mmap_lock
410 * ->anon_vma.lock      (vma_adjust)
411 *
412 * ->anon_vma.lock
413 * ->page_table_lock or pte_lock (anon_vma_prepare and various)
414 *
415 * ->page_table_lock or pte_lock
416 * ->swap_lock          (try_to_unmap_one)
417 * ->private_lock       (try_to_unmap_one)
418 * ->tree_lock          (try_to_unmap_one)
419 * ->zone.lru_lock      (follow_page->mark_page_accessed)
420 * ->zone.lru_lock      (check_pte_range->isolate_lru_page)
421 * ->private_lock       (page_remove_rmap->set_page_dirty)
422 * ->tree_lock          (page_remove_rmap->set_page_dirty)
423 * ->inode_lock         (page_remove_rmap->set_page_dirty)
424 * ->inode_lock         (zap_pte_range->set_page_dirty)
425 * ->private_lock       (zap_pte_range->__set_page_dirty_buffers)
426 *
427 * ->task->proc_lock
428 * ->dcache_lock        (proc_pid_lookup)
429 */
430
431 /*
432 * Remove a page from the page cache and free it. Caller has to make
433 * sure the page is locked and that nobody else uses it - or that usage
434 * is safe. The caller must hold a write_lock on the mapping's tree_lock.
435 */
436 void __remove_from_page_cache(struct page *page)
437 {
438     struct address_space *mapping = page->mapping;
439 }
440

```