# Operating Systems 412

Pete Keleher

# Locks! and synchronization in general

- But what are our goals?
  - mutual exclusion
    - on one section of code
    - on multiple sections of code that access the same state
  - fairness
    - fair share
    - free of starvation
    - deadlock-free
  - performance
    - wait time
    - aggregate overhead of synchronization

# Locks! and synchronization in general

- What are our mechanisms?
  - disabling interrupts
    - pretty much all we need if single core
    - but
      - privileged instruction
      - need to *trust* thread
      - not efficient
      - doesn’t work on multiprocessors
  - atomic instructions
    - test-and-set
      - set memory location to value, returning old value
    - compare-and-swap
      - store at memory location only if it equals specific value
    - load-linked store
      - load from memory location
      - store new value to same location (only if it has not been updated)

# Producer-Consumer flawed take 1

```
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // p1
        if (count == 1)                       // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i);                               // p4
        Pthread_cond_signal(&cond);           // p5
        Pthread_mutex_unlock(&mutex);         // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // c1
        if (count == 0)                       // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get();                      // c4
        Pthread_cond_signal(&cond);           // c5
        Pthread_mutex_unlock(&mutex);         // c6
        printf("%d\n", tmp);
    }
}
```

**Figure 30.8: Producer/Consumer: Single CV And If Statement**

$c_1$1
$c_1$2
$c_1$3 *block*

p1
p2
p4
p5 *$c_1$ ready Q!*
p6
p1
p2
p3

$c_2$1
$c_2$2
$c_2$4
$c_2$5 *$c_1$ ready Q!*
$c_2$6
$c_2$1
$c_2$2
$c_2$3

$c_1$4 *crash*

# Producer-Consumer flawed take 1

- What was the problem?
  - between $c_1$ adding to ready Q and calling get(), the world changed
- Getting *signaled()* is only a hint that the world has changed
  - need to check again
  - and do so atomically w/ the get()
- Semantics
  - this is *Mesa* semantics
  - *Hoare* semantics imply a signaled thread runs immediately

Most systems assume Mesa semantics. You should too. Even if not strictly necessary.

# Producer-Consumer flawed take 2

Assume buffer size 1, initially empty, 2 consumers, 1 producer

*But there's still a bug….*

```
int loops;
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // p1
        while (count == 1)                    // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i);                               // p4
        Pthread_cond_signal(&cond);           // p5
        Pthread_mutex_unlock(&mutex);         // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // c1
        while (count == 0)                    // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get();                      // c4
        Pthread_cond_signal(&cond);           // c5
        Pthread_mutex_unlock(&mutex);         // c6
        printf("%d\n", tmp);
    }
}
```

Figure 30.10: **Producer/Consumer: Single CV And While**

$c_1$1
$c_1$2
$c_1$3 *blocks*

$c_2$1
$c_2$2
$c_2$3 *blocks*

p1, p2, p4
p5 *$c_1$ ready Q!*
p6
p1
p2
p3 *p blocks*

$c_1$2
$c_1$4
$c_1$5 *$c_2$ ready Q!!!*
…
$c_1$3 *$c_1$ blocks*

$c_2$2
$c_2$3 *$c_2$ blocks*

everyone blocked!

# Producer-Consumer correct take 3

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Figure 30.12: **Producer/Consumer: Two CVs And While**

$c_1$1
$c_1$2
$c_1$3 *blocks*

$c_2$1
$c_2$2
$c_2$3 *blocks*

p1, p2, p4
p5 *$c_1$ ready Q!*
p6
p1
p2
p3 *p blocks*

$c_1$2
$c_1$4
$c_1$5 ***p ready Q!***

all good!

Assume initially no memory available.

# Memory allocation covering condition

```
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t  c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...; // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c); // whom to signal??
    Pthread_mutex_unlock(&m);
}
```

Figure 30.15: **Covering Conditions: An Example**

$t_a$ alloc(100) *blocks*
$t_b$ alloc(10) *blocks*
$t_c$ free(50)

*Which thread to wake?*
- wake 'em all!
    - might be inefficient
    - but correct

"covering condition"

# Semaphores

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

- wait()
  - decrement value by one
  - wait if value is negative
- post()
  - increment value by one
  - if one or more threads waiting: wake one

The value, when negative, is equal to the number of waiting threads.

# Semantics

- mutex locks
  - “binary semaphore”
  - lock by calling wait()
  - unlock by calling post()
  - initial value of 1
- ordering primitive
  - “counting semaphore”
  - parent waiting for child, sharing a semaphore
    - parent calls wait()
    - child calls post()
    - initial value? 0

In general, how to determine the initial value?

- how many of your resources you are willing to give out?

# Credits

All figures from Arpaci-Dusseau and Arpaci-Dusseau.