

# PRINCIPLES OF COMPUTER SYSTEMS DESIGN

## CSE130

Winter 2020

Concurrency III

Locks, Condition Variables, Monitors



2

### Today's Lecture

- Concurrency Recap
- Critical Sections Revisited
- Semaphore Problems
- Locks
- Condition Variables
- Monitors
- Concurrency Primitives in Pintos

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

3

### Notices

- Assignment 1 due 23:59 **Sunday January 26**
- Lab 2 due 23:59 **Sunday February 9**
- New Section Times
  - Thursday 9am to 10:30am is moved to Monday 6:30pm to 8:00pm
  - Full Schedule is now
    - Monday 1-2:30pm & 6:30-8pm
    - Tuesday 5-6:30pm
    - Wednesday 2-3:30pm
    - Friday 11am-12:30pm & 12:30-2pm
- Quiz - Possible Midterm Cancellation
  - On Canvas now
  - Take it, have your say!

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

2

### Concurrency in Operating Systems

- **"The actual or apparent simultaneous execution of threads"**
  - Actual = multicore and/or multiprocessors, Apparent = time slicing, Today's Reality = both
- Threads running concurrently in the same process typically want to share data, but uncontrolled access to this data can result in it becoming corrupt
- Remember :
  - CPUs execute one **machine instruction** at a time
  - The operating system can turn interrupts off to ensure multiple machine instructions are executed in sequence, but...
  - **User-level code can not turn interrupts off**
  - **Hence user-level code has no way to guarantee a sequence of instructions will execute without interruption ☺**
  - Which is why we have OS synchronization primitives ☺

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

4

1

## Critical Sections Revisited

- Critical Section: **Any section of a program that should be executed by only one thread at a time**
- Consider a doubly-linked list shared by many threads:
  - Insert operation has a critical section

```

1 void list_insert_after(
2     list_item *existing,
3     list_item *new)
4 {
5     new->prev = existing;
6     new->next = existing->next;
7     new->next->prev = new;
8     existing->next = new;
9 }
```

State after line 7 but before line 8

Not critical, does not impact other threads  
(unless they're trying to insert the same item)

*Critical, both statements must run to completion for the list to be consistent*

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

5

## Critical Sections

- Formalization:
  - **Entry Section** ( negotiate entry to the critical section )
  - **Critical Section** ( do critical task )
  - **Exit Section** ( signal critical task completed )
- Requirements:
  - **Mutual Exclusion**:
    - Only one thread in critical section at a time
  - **Progress**:
    - If a thread wishes to enter an idle critical section, only threads in entry or exit section may help decide if it is the one allowed to enter
    - e.g. if a thread doesn't want access to a CS at the moment, it shouldn't stop other threads from accessing it
  - **Bounded Wait**:
    - There exists a finite bound on the number of times any waiting thread must "stand-aside" whilst other threads get access to the critical section

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

6

## The Progress Requirement

If thread A wishes to enter an idle critical section, only threads in the entry or exit sections may help decide if A is the one allowed to enter.

```

void put(int x) {
    P(vacant);
    P(mutex);
    buffer[nextIn] = x;
    nextIn = (nextIn++) % N;
    V(mutex);
    V(occupied);
}

```

( see lecture recording for animation )

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

7

## Semaphore Synchronisation

Consider a solution to the **producer-consumer** problem using counting semaphores to allocate slots in a circular buffer of integers of size N ...

Binary Semaphore:	mutex = 1;
Counting Semaphores:	vacant = N, occupied = 0; // no. of slots in each state
Buffer Indexes:	nextIn = 0, nextOut = 0;

```

void put(int x) {
    P(vacant); // are there some vacant slots?
    P(mutex); // am I allowed to change the buffer?
    buffer[nextIn] = x;
    nextIn = (nextIn++) % N;
    V(mutex); // allow others to change the buffer
    V(occupied); // increment the no. of used slots
}

int get() {
    P(occupied); // is there at least one piece of data?
    P(mutex); // am I allowed to change the buffer?
    int x = buffer[nextOut];
    nextOut = (nextOut++) % N;
    V(mutex); // allow others to change the buffer
    V(vacant); // increment the no. of available slots
    return x;
}

```

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

8

## Semaphore Circular Buffer N =2

Binary Semaphore: mutex = 1  
 Counting Semaphores: vacant = 1  
 occupied = 1  
 Buffer Indexes: nextIn = 1  
 nextOut = 0



Two threads want to put data  
at the same time



```
void put(int x) {
    P(vacant);
    P(mutex);
    buffer[nextIn] = x;
    nextIn = (nextIn++) % N;
    V(mutex);
    V(occupied);
}
```

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.
( see lecture recording for animation )
9

## Semaphore Problems

- The use of three separate semaphores for such a simple task suggests a need for better synchronization primitives:
  - Condition Variables
  - Monitors
- These constructs can ( but don't always ) lead to neater code with fewer detectable bugs

26

## Lock == Mutex

- Lock** and **Mutex** are interchangeable terms
- From Wikipedia:
  - "In computer science, a **lock** or **mutex** (from mutual exclusion) is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution."
- As we've seen, locks can be implemented as binary semaphores
  - In Pintos, that's exactly how they are implemented
- Two functions:
  - lock(lock)**
  - unlock(lock)**

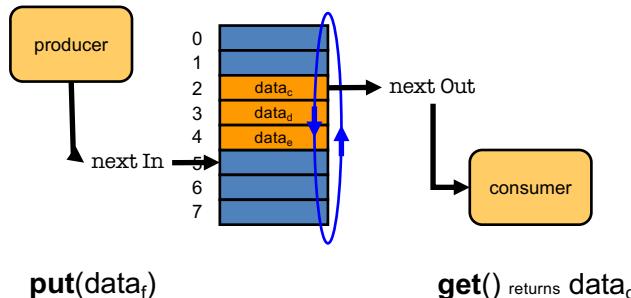
UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.
27

## Condition Variables

- A higher level concurrency primitive
- Makes coding of thread synchronization:
  - More structured
  - Easier to follow
  - Easier to debug
- Condition variables do not enforce mutual-exclusion, they require a lock or semaphore as a mutex**

28
3

## Circular Bounded Buffers



UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

29

## Condition Variables

- The general problem is the need to wait for some condition to become true within a critical section
- For the **producer** the condition is that the buffer is **not full**
- For the **consumer** the condition is that the buffer is **not empty**

30

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

## Condition Variables

- Used to wait for a condition to be true**
  - Essentially just a container for waiting threads (**remember this for Lab 2**)
- Functions:
  - wait(condition, lock)**  
release the lock  
put thread to sleep until condition becomes true  
re-acquired before exit
  - signal(condition)**  
wake one thread waiting for the condition
  - broadcast(condition)**  
same as signal, but wakes all threads waiting for the condition
- In all cases, caller must hold the lock

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

31

## Condition Variable Synchronization

Consider a solution to the producer-consumer problem using a **lock** and a **condition variable** to allocate slots in a circular buffer of integers of size **N** ...

Lock:              mutex = true; Condition Variable: modified = false; // condition = "the buffer contents have changed" Counter:            occupied = 0; // no. of occupied slots Buffer Indexes:    nextIn = 0, nextOut = 0;	<pre> void put(int x) {     lock(mutex); // request exclusive access to the buffer     while (occupied == N)         wait(modified, mutex); // wait for free space     buffer[nextIn] = x;     nextIn = (nextIn + 1) % N;     occupied++;     broadcast(modified); // let others know buffer changed     unlock(mutex); // release the buffer } </pre>	<pre> int get() {     lock(mutex); // request exclusive access to the buffer     while (occupied == 0)         wait(modified, mutex); // wait for data     int x = buffer[nextOut];     nextOut = (nextOut + 1) % N;     occupied--;     broadcast(modified); // let others know buffer changed     unlock(mutex); // release the buffer     return x; } </pre>
--	--	---

32

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

```

/* buffer-semaphore.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <semaphore.h>
#include <pthread.h>
#include <unwind.h>

#define PRODUCERS 2
#define CONSUMERS 5

#define MAX_SLEEP_MILLISECONDS 3000
#define MAX_PUTS_PER_PRODUCER 1000

#define BUFFER_SIZE 10

typedef struct buffer_t {
    int buf[BUFFER_SIZE];
    int next_in;
    int next_out;
    sem_t occupied;
    sem_t vacant;
    sem_t mutex;
} Buffer;

Buffer buffer;

```

```

/* buffer-condvar.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <semaphore.h>
#include <pthread.h>

#define PRODUCERS 2
#define CONSUMERS 5

#define MAX_SLEEP_MILLISECONDS 3000
#define MAX_PUTS_PER_PRODUCER 1000

#define BUFFER_SIZE 10

typedef struct buffer_t {
    int buf[BUFFER_SIZE];
    int next_in; /* next slot to add an element to (may not currently be vacant) */
    int next_out; /* next slot to get an element from (may not currently be occupied) */
    int occupied_slots; /* number of slots with a valid entry */
    pthread_mutex_t mutex; /* buffer access mutex (lock) */
    pthread_cond_t buffer_modified; /* signal whenever buffer modified */
} Buffer;

Buffer buffer;

```

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

33

```

void buffer_init(buffer *buffer) {
    sem_init(&buffer->occupied, 0, 0);
    sem_init(&buffer->vacant, 0, BUFFER_SIZE);
    sem_init(&buffer->mutex, 0, 1);
}

void buffer_put(buffer *buffer, int value) {
    sem_wait(buffer->vacant);
    sem_wait(buffer->mutex);

    buffer->buf[buffer->next_in] = value;
    buffer->next_in = (buffer->next_in+1) % BUFFER_SIZE;
}

sem_post(buffer->mutex);
sem_post(buffer->occupied);

int buffer_get(buffer *buffer) {
    sem_wait(buffer->occupied);
    sem_wait(buffer->mutex);

    int value = buffer->buf[buffer->next_out];
    buffer->next_out = (buffer->next_out+1) % BUFFER_SIZE;

    sem_post(buffer->mutex);
    sem_post(buffer->vacant);
    return value;
}

void buffer_init(buffer *buffer) {
    pthread_mutex_init(&buffer->mutex, NULL);
    pthread_cond_init(&buffer->buffer_modified, NULL);
}

void buffer_put(buffer *buffer, int value) {
    pthread_mutex_lock(&buffer->mutex);
    while (buffer->occupied_slots == BUFFER_SIZE)
        pthread_cond_wait(&buffer->buffer_modified, &buffer->mutex);

    buffer->buf[buffer->next_in] = value;
    buffer->next_in = (buffer->next_in+1) % BUFFER_SIZE;
    buffer->occupied_slots += 1;

    pthread_cond_broadcast(&buffer->buffer_modified);
    pthread_mutex_unlock(&buffer->mutex);
}

int buffer_get(buffer *buffer) {
    pthread_mutex_lock(&buffer->mutex);
    while (buffer->occupied_slots == 0)
        pthread_cond_wait(&buffer->buffer_modified, &buffer->mutex);

    int value = buffer->buf[buffer->next_out];
    buffer->next_out = (buffer->next_out+1) % BUFFER_SIZE;
    buffer->occupied_slots -= 1;

    pthread_cond_broadcast(&buffer->buffer_modified);
    pthread_mutex_unlock(&buffer->mutex);
    return value;
}

```

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

34

Exactly the same!

```

static void *produce(void *arg) {
    long tid = (long) arg;
    int value;
    printf("P %d running\n", tid);

    for (int i = 0; i < MAX_PUTS_PER_PRODUCER; i++) {
        value = tid*1000+i;
        printf("=> P %d produced %04d\n", tid, value);
        buffer_put(&buffer, value);
    }

    printf("P %d ### finished ##\n", tid);
    pthread_exit(NULL);
    return NULL;
}

static void *consume(void *arg) {
    long tid = (long) arg;
    int value, millis;
    printf("C %d running\n", tid);

    for (;;) {
        millis = rand() % MAX_SLEEP_MILLISECONDS;
        value = buffer_get(&buffer);
        printf("<- C %d consumed %04d - sleeping for %dns\n",
              tid, value, millis);
        sleep_ms(millis);
    }

    // Will never return, consumers are in an infinite loop
    return NULL;
}

```

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

35

## Self Study ( totally not graded, just do it for fun! )

- Download the source code from lecture handouts
  - Build it, run it, study it
- Change it so producers are slow and consumers are fast
  - Q: What happens?
- Increase no. of producers & decrease no. of consumers
  - Q: What happens?

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

37

## Monitors

- A synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true
- Has a mechanism for signaling other threads that their condition has been met
- Consists of a mutex (lock) and condition variables
- Provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

38

## Monitors in C

- What do you notice about our Buffer type?
  - It ( kind of ) is a monitor
  - It maintains a lock and condition variable, but...
- No ‘private’ concept in C
- What we’d like is a buffer.h something like this:

```
typedef struct buffer_t { /* nothing */ } Buffer;

Buffer *buffer_init(int *buf, int size);
void buffer_destroy(Buffer *buffer);
void buffer_put(Buffer *buffer, int value);
int buffer_get(Buffer *buffer);
```

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

39

## Concurrency Primitives in Pintos

- Semaphore
 

```
void semaphore_init(struct semaphore *, unsigned value);
void semaphore_down(struct semaphore *); // Dijkstra's "P"
void semaphore_up(struct semaphore *); // Dijkstra's "V"
```
- Lock
 

```
void lock_init(struct lock *);
void lock_acquire(struct lock *);
void lock_release(struct lock *);
```
- Condition Variable
 

```
void condvar_init(struct condvar *);
void condvar_wait(struct condvar *, struct lock *);
void condvar_signal(struct condvar *, struct lock *);
void condvar_broadcast(struct condvar *, struct lock *);
```

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

40



## Assignment 1 - Secret Sauce

- Watch your processes!!
  - See all the processes you have and kill ‘em:
 

```
$ ps aux | grep <cruzid>
$ killall -u <cruzid>
```
- Watch your shared memory segments!!
  - Update to the latest assignment installation
  - Use you cruzid as the segment name
 

```
key_t key = ftok("<cruzid>", 65);
```
  - See all the shared memory segments you have
 

```
$ ipcs -m | grep <cruzid>
0x000000607 264699941 <cruzid> 666 160 0
0x000000000 264667172 <cruzid> 666 128 0
```
  - Remove one
 

```
$ ipcrm shm 264699941
```

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

41

## Assignment 1 - Basic Pseudo Code

```

fork
if error
  exit
if child
  sort one side
  exit
if parent
  sort the other side
  wait for child to finish
merge

```



UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

42

## Assignment 1 - Pseudo Code 1

```

create shared memory
attach to shared memory
copy all local memory into shared memory
fork
if error
  exit
if child
  attach to shared memory
  sort one side of shared memory
  detach from shared memory
  exit
if parent
  sort the other side of shared memory
  wait for child to finish
  merge shared memory
  copy shared memory to local memory
  detach from shared memory
  destroy shared memory

```



UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

43

## Assignment 1 - Pseudo Code 2

```

create shared memory
attach to shared memory
copy right side of local memory into shared memory
fork
if error
  exit
if child
  attach to shared memory
  sort shared memory
  detach from shared memory
  exit
if parent
  sort left side of local memory
  wait for child to finish
  copy shared memory to right side of local memory
  detach from shared memory
  destroy shared memory
  merge local memory

```



UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

44

CSE130 Winter 2020 Assignment 1

Fri Jan 24 07:59:42 PST 2020

Multi Process: PASS  
 Multi Process: PASS

Functional: 10/10 100.0%

Speedup: PASS (1.80 times faster)  
 Speedup: PASS (1.82 times faster)  
 Speedup: PASS (1.80 times faster)  
 Speedup: PASS (1.79 times faster)  
 Speedup: PASS (1.80 times faster)  
 Speedup: PASS (1.81 times faster)  
 Speedup: PASS (1.81 times faster)

Non-Functional: 10/10 100.0%

Tests: 20/20 100.0% of 90%

C-Code: 100.0% of 10%

Total: 100.0%

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.

45

## Next Lecture

- Deadlocks

46

UCSC BSOE CSE130 Winter 2020. Copyright © 2017-2020 David C. Harrison. All rights reserved.