# Concurrency

- Exam 1
- 26 - Concurrency
- 27 - Overview, and POSIX threads (pthreads)
- 28 - Locks
- 29 - Concurrent Data Structures
- 30 Condition Variables
- 31 - Semaphores
- 32 - Common Problems
- 33 - Event-Based Concurrency



# Concurrent Data *counters*

```
typedef struct __counter_t {
        int value;
} counter_t;

void init(counter_t *c) {
        c->value = 0;
}

void increment(counter_t *c) {
        c->value++;
}

void decrement(counter_t *c) {
        c->value--;
}

int get(counter_t *c) {
        return c->value;
}
```

- simple, but not correct

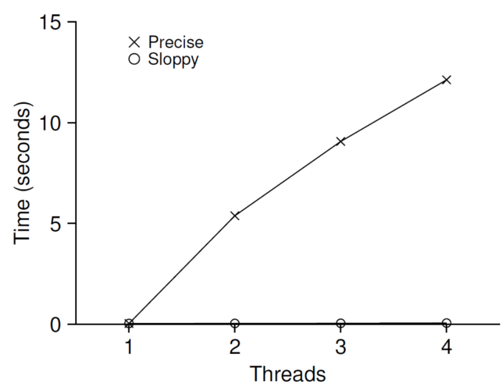
# Concurrent Data *counters*

```
typedef struct __counter_t {
        int value;
        pthread_lock_t lock;
} counter_t;

void init(counter_t *c) {
        c->value = 0;
        Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value++;
        Pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value--;
        Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        int rc = c->value;
        Pthread_mutex_unlock(&c->lock);
        return rc;
}
```

- correct, but not performant (research as late as 2010)



# Concurrent Data *counter performance*




**Performance of**
**Traditional vs. Sloppy Counters**
(Threshold of Sloppy, $S$, is set to 1024)

- sloppiness can be useful….

# Concurrent Data *sloppy (approximate) counters*

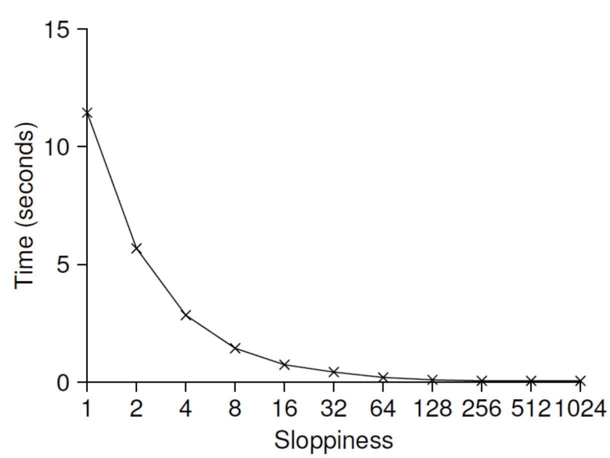
- sloppiness can be useful….
  - each core has local counter
  - threads update local counter
  - periodically transfer counts to a *global* counter

| Time | $L_1$ | $L_2$ | $L_3$ | $L_4$ | $G$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 2 | 1 | 0 | 2 | 1 | 0 |
| 3 | 2 | 0 | 3 | 1 | 0 |
| 4 | 3 | 0 | 3 | 2 | 0 |
| 5 | 4 | 1 | 3 | 3 | 0 |
| 6 | $5 \rightarrow 0$ | 1 | 3 | 4 | 5 (from $L_1$) |
| 7 | 0 | 2 | 4 | $5 \rightarrow 0$ | 10 (from $L_4$) |



# Concurrent Data *sloppy (approximate) counters*

- importance of threshold *s*
  - low *s*: poor performance, global count quite accurate
  - high *s*: good performance, global count quite inaccurate



# Concurrent Data *linked lists*

```
// basic node structure
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

// basic list structure (one used per list)
typedef struct __list_t {
        node_t *head;
        pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}

(Cont.)
```



# Concurrent Data *linked lists*

```
int List_Insert(list_t *L, int key) {
        pthread_mutex_lock(&L->lock);
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
                perror("malloc");
                pthread_mutex_unlock(&L->lock);
               return -1; // fail
        }
        new->key = key;
        new->next = L->head;
        L->head = new;
        pthread_mutex_unlock(&L->lock);
        return 0; // success
}

int List_Lookup(list_t *L, int key) {
        pthread_mutex_lock(&L->lock);
        node_t *curr = L->head;
        while (curr) {
                if (curr->key == key) {
                        pthread_mutex_unlock(&L->lock);
                        return 0; // success
                }
                curr = curr->next;
        }
        pthread_mutex_unlock(&L->lock);
        return -1; // failure
}
```

# Concurrent Data *linked lists*

- Works!
  - but slow
  - also error-prone:
    - if `malloc()` fails, code must release the lock
    - solution: lock and release *only around crit section*

```
void List_Insert(list_t *L, int key) {
        // synchronization not needed
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
                perror("malloc");
                return;
        }
        new->key = key;

        // just lock critical section
        pthread_mutex_lock(&L->lock);
        new->next = L->head;
        L->head = new;
        pthread_mutex_unlock(&L->lock);
}
```

# Concurrent Data *scaling linked lists*

- Hand-over-hand locking (*lock coupling*)
  - lock per node
  - traverse the list:
    - grab next node's lock
    - release current node's lock
  - evaluation:
    - concurrency: *great!*
    - performance: *horrible!*

- Michael and Scott *concurrent queues*:
  - one lock for head
  - one lock for tail
  - and a dummy node to separate head and tail operations

# Concurrent Data *concurrent queues*

```
typedef struct __node_t {
        int             value;
        struct __node_t  *next;
} node_t;

typedef struct __queue_t {
        node_t *head;
        node_t *tail;
        pthread_mutex_t headLock;
        pthread_mutex_t tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
        node_t *tmp = malloc(sizeof(node_t));
        tmp->next = NULL;
        q->head = q->tail = tmp;
        pthread_mutex_init(&q->headLock, NULL);
        pthread_mutex_init(&q->tailLock, NULL);
}

(Cont.)
```



# Concurrent Data *concurrent queues*

```
void Queue_Enqueue(queue_t *q, int value) {
        node_t *tmp = malloc(sizeof(node_t));
        assert(tmp != NULL);

        tmp->value = value;
        tmp->next = NULL;

        pthread_mutex_lock(&q->tailLock);
        q->tail->next = tmp;
        q->tail = tmp;
        pthread_mutex_unlock(&q->tailLock);
}

int Queue_Dequeue(queue_t *q, int *value) {
        pthread_mutex_lock(&q->headLock);
        node_t *tmp =     q->head;
        node_t *newHead = tmp->next;
        if (newHead == NULL) {
                pthread_mutex_unlock(&q->headLock);
                return -1; // queue was empty
        }
        *value = newHead->value;
        q->head = newHead;
        pthread_mutex_unlock(&q->headLock);
        free(tmp);
        return 0;
}
```

# Condition variables *one BAD implementation*

- Waiting:

```
while(initialized == 0)
        ; // spin
```

- Signaling:

```
initialized = 1;
```

  - spinlock wastes CPU
  - does not include lock synchronization
  - source of many, many bugs



# Concurrency

- Exam 1
- 26 - Concurrency
- 27 - Overview, and POSIX threads (pthreads)
- 28 - Locks
- 29 - Concurrent Data Structures
- 30 Condition Variables
- 31 - Semaphores
- 32 - Common Problems
- 33 - Event-Based Concurrency

# Condition Variables *parent waiting on child*

**A Parent Waiting For Its Child**

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

**What we would like to see here is:**

```
parent: begin
child
parent: end
```



# Condition Variables *parent waiting on child*

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

- Correct, but:
    - wildly inefficient
    - sometimes incorrect

# Condition Variables *parent waiting on child*

- Condition variable:
  - `signal()` that condition is true
  - `wait()` on condition to become true

  - `pthread_cond_wait`:
    - put the calling thread to sleep
    - releases the mutex
    - wait for signal from some other thread
    - re-acquire mutex

  - `pthread_cond_signal`:
    - unblock *at least one* of the threads blocked on the conditional variable
    - ??



# Condition Variables *parent waiting on child*

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
        Pthread_mutex_lock(&m);
        Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
        printf("child\n");
        thr_exit();
        return NULL;
}

void thr_join() {
        Pthread_mutex_lock(&m);
        Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
        printf("parent: begin\n");
        pthread_t p;
        Pthread_create(&p, NULL, child, NULL);
        thr_join();
        printf("parent: end\n");
        return 0;
}
```

*no*

# Condition Variables *parent waiting on child*

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
        Pthread_mutex_lock(&m);
        done = 1;
        Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
        printf("child\n");
        thr_exit();
        return NULL;
}

void thr_join() {
        Pthread_mutex_lock(&m);
        while (done == 0)
                Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
        printf("parent: begin\n");
        pthread_t p;
        Pthread_create(&p, NULL, child, NULL);
        thr_join();
        printf("parent: end\n");
        return 0;
}
```

*yes!*



# Condition variables *continued*

- Waiting:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
        pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

  - the wait call releases the lock when the thread sleeps
  - the wait call re-acquires the lock when thread awakened

- Signaling:

```
pthread_mutex_lock(&lock);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

# 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**

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

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

p1
p2
p4
p5 *$c_1$ unblocked*
p6
p1
p2
p3

$c_2$1
$c_2$2
$c_2$4
$c_2$5 *$c_1$ unblocked*
$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 $c_1$ becoming ready, 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 *$c_1$ unblocked*
p6
p1
p2
p3 *p blocks*

$c_1$2
$c_1$4
$c_1$5 *$c_2$ unblocked*
…
$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);            // p1
        while (count == 1)                     // p2
            Pthread_cond_wait(&empty, &mutex)  // p3
        put(i);                                // p4
        Pthread_cond_signal(&fill);            // 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(&fill, &mutex);  // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&empty);           // c5
        Pthread_mutex_unlock(&mutex);          // c6
        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 *$c_1$ unblocked*
p6
p1
p2
p3 *p blocks*

$c_1$2
$c_1$4
$c_1$5 *p unblocked*

$c_2$2
$c_2$3 *blocks*

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

`pthread_cond_broadcast()`

"covering condition"