CSCI-UA.0201

Computer Systems Organization

Concurrency – Condition Variables

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies
Producer/consumer based on a FIFO Queue

queue_t queue;
pthread_mutex_t mu;
...

void produce(int x) {
    pthread_mutex_lock(&mu);
    enqueue(&queue, x);
    pthread_mutex_unlock(&mu);
}
The Need for Modular Synchronization

Suppose queue is bounded:
- `enqueue` may block until queue has room
- decision whether to block depends on internal state of the queue

Multiple producers/consumers:
- every thread needs to keep track of the lock, the queue state, etc.
The Need for Modular Synchronization

Suppose queue is bounded:
• enqueue may block until queue has room
• decision whether to block depends on internal state of the queue

Multiple producers/consumers:
• every thread needs to keep track of the lock, the queue state, etc.
Modular Synchronization

Let queue handle its own synchronization

• queue has its own lock
  – acquired by each enqueue/dequeue call
  – released when the call returns

• if thread enqueues on a full queue
  – queue itself detects the problem
  – suspend the caller and resume when the queue has room
Condition Variables

• A mechanism to block a thread until some condition becomes true
• Condition variables allow a thread to
  – temporarily release the lock and suspend itself until awoken by another thread
  – awake other threads that are currently suspended waiting for that condition
Monitors

The combination of

• a data structure and its operations
• a mutual exclusion lock
• and the lock's condition variables is called a **monitor**

Monitors enable modular synchronization.
Condition Variables in the pthread lib

- `pthread_cond_t`
- `pthread_cond_wait` / `pthread_cond_timedwait`
- `pthread_cond_signal`
- `pthread_cond_broadcast`
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

• Atomically releases mutex and causes the calling thread to be put on an internal waiting queue for cond.

• On successful return, mutex is locked (which the calling thread should unlock later)
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

• Atomically releases mutex and causes the calling thread to be put on an internal waiting queue for cond.
• On successful return, mutex is locked (which the calling thread should unlock later)

No other thread can grab the released mutex before the calling thread is put in the waiting queue
int pthread_cond_signal(pthread_cond_t *cond);

• Unblock at least one of the threads waiting on cond

int pthread_cond_broadcast(pthread_cond_t *cond);

• Unblock all threads waiting on cond.
A Typical Monitor Execution

lock(lock)

lock

critical section

waiting room
A Typical Monitor Execution

lock

critical section

wait (cond, lock)

waiting room
A Typical Monitor Execution

lock

lock(lock)

critical section

waiting room
A Typical Monitor Execution

lock

waiting room

critical section
A Typical Monitor Execution

lock(lock)

lock

critical section

waiting room
A Typical Monitor Execution

lock(lock)

lock

critical section

unlock(lock)

broadcast(cond, lock)

waiting room
A Typical Monitor Execution

- lock(lock)
- critical section
- unlock(lock)
broadcast(cond, lock)

waiting room
A Typical Monitor Execution

- Lock
- Critical section
- Waiting room
Using Condition Variables

```c
pthread_mutex_t mu;
pthread_cond_t cond;
...
void foo() {
    pthread_mutex_lock(&mu);
    while (!property)
        pthread_cond_wait(&cond, &mu);
    ...
    pthread_mutex_unlock(&mu);
}
```
Using Condition Variables

```c
pthread_mutex_t mu;
pthread_cond_t cond;
...
void foo() {
    pthread_mutex_lock(&mu);
    while (!property)
        pthread_cond_wait(&cond, &mu);
    ...
    pthread_mutex_unlock(&mu);
}
```

create new condition variable
Using Condition Variables

```c
pthread_mutex_t mu;
pthread_cond_t cond;
...
void foo() {
    pthread_mutex_lock(&mu);
    acquire the lock
    while (!property)
        pthread_cond_wait(&cond, &mu);
    ...
    pthread_mutex_unlock(&mu);
}
```
Using Condition Variables

```c
pthread_mutex_t mu;
pthread_cond_t cond;
...
void foo() {
    pthread_mutex_lock(&mu); /* not happy */
    while (!property) {
        pthread_cond_wait(&cond, &mu);
    }
    ...
    pthread_mutex_unlock(&mu);
}
```
Using Condition Variables

```c
pthread_mutex_t mu;
pthread_cond_t cond;
...
void foo() {
    pthread_mutex_lock(&mu);
    while (!property)
        pthread_cond_wait(&cond, &mu);
    ...
    pthread_mutex_unlock(&mu);
}
```
release the lock and suspend until notified
Using Condition Variables

```c
pthread_mutex_t mu;
pthread_cond_t cond;
...

void foo() {
    pthread_mutex_lock(&mu);
    while (!property) {
        pthread_cond_wait(&cond, &mu);
        ...
    }
    happy: property must hold
    pthread_mutex_unlock(&mu);
}
```
Example: Blocking Queue

typedef struct {
    pthread_mutex_t mu;
    pthread_cond_t notFull;
    pthread_cond_t notEmpty;
    int items[LEN];
    int tail, head, count;
} queue_t;
Example: Blocking Queue

typedef struct {
    pthread_mutex_t mu;
    pthread_cond_t notFull;
    pthread_cond_t notEmpty;
    int items[LEN];
    int tail, head, count;
} queue_t;

mutex lock for queue
typedef struct {
    pthread_mutex_t mu;
    pthread_cond_t notFull;
    pthread_cond_t notEmpty;
    int items[LEN];
    int tail, head, count;
} queue_t;
Example: Blocking Queue

typedef struct {
    pthread_mutex_t mu;
    pthread_cond_t notFull;
    pthread_cond_t notEmpty;
    int items[LEN];
    int tail, head, count;
} queue_t;
Example: Blocking Queue

typedef struct {
    pthread_mutex_t mu;
    pthread_cond_t notFull;
    pthread_cond_t notEmpty;
    int items[LEN];
    int tail, head, count;
} queue_t;

internal queue state protected by lock
void enqueue(queue_t *queue, int x) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == LEN)
        pthread_cond_wait(&queue->notFull, &queue->mu);
    queue->items[queue->tail] = x;
    if (++queue->tail == LEN) queue->tail = 0;
    ++queue->count;
    pthread_cond_signal(&queue->notEmpty, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
}
Blocking Queue: enqueue

```c
void enqueue(queue_t *queue, int x) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == LEN)
        pthread_cond_wait(&queue->notFull, &queue->mu);
    queue->items[queue->tail] = x;
    if (++queue->tail == LEN) queue->tail = 0;
    ++queue->count;
    pthread_cond_signal(&queue->notEmpty, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
}
```

wait until queue has space
void enqueue(queue_t *queue, int x) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == LEN)
        pthread_cond_wait(&queue->notFull, &queue->mu);
    queue->items[queue->tail] = x;
    if (++queue->tail == LEN) queue->tail = 0;
    ++queue->count;
    pthread_cond_signal(&queue->notEmpty, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
}

queue has space!
insert element
void enqueue(queue_t *queue, int x) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == LEN)
        pthread_cond_wait(&queue->notFull, &queue->mu);
    queue->items[queue->tail] = x;
    if (++queue->tail == LEN) queue->tail = 0;
    ++queue->count;
    pthread_cond_signal(&queue->notEmpty, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
}
Blocking Queue: dequeue

```c
int dequeue(queue_t *queue) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == 0)
        pthread_cond_wait(&queue->notEmpty, &queue->mu);
    int x = queue->items[queue->head];
    if (++queue->head == LEN) queue->head = 0;
    --queue->count;
    pthread_cond_signal(&queue->notFull, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
    return x;
}
```
Blocking Queue: dequeue

```c
int dequeue(queue_t *queue) {
    pthread_mutex_lock(&queue->mu);

    while (queue->count == 0) {
        pthread_cond_wait(&queue->notEmpty, &queue->mu);
    }

    int x = queue->items[queue->head];
    if (++queue->head == LEN) queue->head = 0;
    --queue->count;
    pthread_cond_signal(&queue->notFull, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
    return x;
}
```
wait until queue is nonempty
Blocking Queue: dequeue

```c
int dequeue(queue_t *queue) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == 0)
        pthread_cond_wait(&queue->notEmpty, &queue->mu);
    int x = queue->items[queue->head];
    if (++queue->head == LEN) queue->head = 0;
    --queue->count;
    pthread_cond_signal(&queue->notFull, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
    return x;
}
```

Queue nonempty!
retrieve next element
Blocking Queue: dequeue

```c
int dequeue(queue_t *queue) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == 0)
        pthread_cond_wait(&queue->notEmpty, &queue->mu);
    int x = queue->items[queue->head];
    if (++queue->head == LEN) queue->head = 0;
    --queue->count;
    pthread_cond_signal(&queue->notFull, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
    return x;
}
```

wake up one waiting producer
Improved enqueue?

```c
void enqueue(queue_t *queue, int x) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == LEN)
        pthread_cond_wait(&queue->notFull, &queue->mu);
    queue->items[queue->tail] = x;
    if (++queue->tail == LEN) queue->tail = 0;
    ++queue->count;
    if (queue->count == 1)
        pthread_cond_signal(&queue->notEmpty, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
}
```
void enqueue(queue_t *queue, int x) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == LEN)
        pthread_cond_wait(&queue->notFull, &queue->mu);
    queue->items[queue->tail] = x;
    ++queue->tail = (queue->tail == LEN) ? 0 : queue->tail;
    ++queue->count;
    if (queue->count == 1)
        pthread_cond_signal(&queue->notEmpty, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
}
Lost Wakeup

lock

queue state:

[]
LEN=2

waiting room

deqeue()  dequeue()

24
Lost Wakeup

- Lock
- Queue state: []
  LEN=2
- Dequeue()
- Waiting room
Lost Wakeup

lock

queue state:

[]
LEN=2

dequeue()
Lost Wakeup

waiting room

dequeue()

queue state:
[
LEN=2

lock
Lost Wakeup

lock

queue state:
[]
LEN=2

waiting room

dqueue()
Lost Wakeup

enqueue(1)  enqueue(2)

lock

queue state: [ ] LEN=2

waiting room
dequeue()  dequeue()
Lost Wakeup

queue state:
[1]
LEN=2

enqueue(2)

depqueue()
Lost Wakeup

queue state: [1]
LEN=2

enqueue(2)

dequeue()

signal(notEmpty)
Lost Wakeup

Lock

Waiting room

Queue state:
[1]
LEN=2

eenqueue(2) → dequeue()

dequeue()
Lost Wakeup

Waiting room

dequeue()

Lock

Queue state:
[1,2]
LEN=2

dequeue()
Lost Wakeup

lock

queue state: [1, 2]
LEN=2

dequeue()

waiting room

dequeue()

no call to
signal(notEmpty)!

image
Lost Wakeup

lock

queue state: [1,2]
LEN=2

dequeue()

waiting room

dequeue()
Lost Wakeup

waiting room

dequeue()

lock

queue state:
[1]
LEN=2
Lost Wakeup

lock

queue state:
[1]
LEN=2

signal(notFull)

suspended thread waits for
signal(notEmpty)!

waiting room
dequeue()
Lost Wakeup

queue state:
[1]
LEN=2

remaining thread is stuck!
The Lost-Wakeup Problem

• Condition variables are inherently vulnerable to lost wakeups
  – one thread waits forever without realizing that its waiting condition has become true

• Programming practices
  – if in doubt, broadcast to all waiting processes
  – specify a timeout when waiting
typedef struct {
    pthread_mutex_t mu;
    pthread_cond_t cond;
    int items[LEN];
    int tail, head, count;
} queue_t;
Simplified Blocking Queue

typedef struct {
    pthread_mutex_t mu;
    pthread_cond_t cond;
    int items[LEN];
    int tail, head, count;
} queue_t;
void enqueue(queue_t *queue, int x) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == LEN) {
        pthread_cond_wait(&queue->cond, &queue->mu);
    }
    queue->items[queue->tail] = x;
    if (++queue->tail == LEN) queue->tail = 0;
    ++queue->count;
    pthread_cond_broadcast(&queue->cond, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
}
Simplified Blocking Queue: dequeue

```c
int dequeue(queue_t *queue) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == 0)
        pthread_cond_wait(&queue->cond, &queue->mu);
    int x = queue->items[queue->head];
    if (++queue->head == LEN) queue->head = 0;
    --queue->count;
    pthread_cond_broadcast(&queue->cond, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
    return x;
}
```
int dequeue(queue_t *queue) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == 0)
        pthread_cond_wait(&queue->cond, &queue->mu);
    int x = queue->items[queue->head];
    if (++queue->head == LEN) queue->head = 0;
    --queue->count;
    pthread_cond_signal(&queue->cond, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
    return x;
}
Simplified Blocking Queue: dequeue

```c
int dequeue(queue_t *queue) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == 0)
        pthread_cond_wait(&queue->cond, &queue->mu);
    int x = queue->items[queue->head];
    if (++queue->head == LEN) queue->head = 0;
    --queue->count;
    pthread_cond_signal(&queue->cond, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
    return x;
}
```

enough?
Simplified Blocking Queue: dequeue

```c
int dequeue(queue_t *queue) {
    pthread_mutex_lock(&queue->mu);
    while (queue->count == 0)
        pthread_cond_wait(&queue->cond, &queue->mu);
    int x = queue->items[queue->head];
    if (++queue->head == LEN) queue->head = 0;
    --queue->count;
    pthread_cond_signal(&queue->cond, &queue->mu);
    pthread_mutex_unlock(&queue->mu);
    return x;
}
```

enough?
Lost Wakeup in Simplified Queue with signal()

lock

queue state: []
LEN=1

waiting room

enq(0) enq(1) enq(2)
Lost Wakeup in Simplified Queue with signal()

lock

queue state: []
LEN=1

enq(1)
enq(2)

waiting room
Lost Wakeup in Simplified Queue with signal()

- lock
- queue state: [0]
  LEN=1
- waiting room
- enq(1)
- enq(2)
- signal()
Lost Wakeup in Simplified Queue with signal()
Lost Wakeup in Simplified Queue with signal()

- Lock
- Queue state: [0]
  LEN=1

waiters:
- enq(2)
- enq(1)
Lost Wakeup in Simplified Queue with signal()
Lost Wakeup in Simplified Queue with signal()

- Lock
- Queue state: [0]
  LEN=1
- Waiting room
- enq(1) → enq(2)
Lost Wakeup in Simplified Queue with signal()
Lost Wakeup in Simplified Queue with signal()

lock

queue state: [0] LEN=1

deq() deq() enq(1)

waiting room
enq(2)
Lost Wakeup in Simplified Queue with signal()
Lost Wakeup in Simplified Queue with signal()

lock

queue state: []
LEN=1

waiting room

enq(1) → deq() → deq()

enq(2)
Lost Wakeup in Simplified Queue with signal()
Lost Wakeup in Simplified Queue with signal()

queue state: []
LEN=1

enq(1)
deq()

waiting room

deq()
enq(2)
Lost Wakeup in Simplified Queue with signal()

lock

enq(1)

deq()

queue state: []
LEN=1

waiting room

ten(2)

enq()
Lost Wakeup in Simplified Queue with signal()

lock

queue state: []
LEN=1

waiting room

enq(1)

deq()

enq(2)

deq()
Lost Wakeup in Simplified Queue with signal()

lock

queue state: []
LEN=1

waiting room

deq()

tenq(2)
deq()
Lost Wakeup in Simplified Queue with signal()
Lost Wakeup in Simplified Queue with signal()

lock

queue state: [1]
LEN=1

deq(

enq(2)

deq()

waiting room

signal()
Lost Wakeup in Simplified Queue with signal()

lock

queue state: [1] LEN=1

enq(2)
deq()
deq()
Lost Wakeup in Simplified Queue with signal()
Lost Wakeup in Simplified Queue with signal()

remaining threads are stuck!
 pthread_mutex_lock(&mu);
Fairness and Starvation

pthread_mutex_lock does not guarantee fairness

Thread 0

```
pthread_mutex_lock(&mu);
processing...
pthread_mutex_unlock(&mu);
```

Thread 1

```
pthread_mutex_lock(&mu);
block and wait
```

Thread 2

```
pthread_mutex_lock(&mu);
block and wait
```
Fairness and Starvation

pthread_mutex_lock does not guarantee fairness

Thread 0

pthread_mutex_lock(&mu);

processing...

pthread_mutex_unlock(&mu);

pthread_mutex_lock(&mu);

block and wait

Thread 1

pthread_mutex_lock(&mu);

block and wait

Thread 2

pthread_mutex_lock(&mu);

block and wait

processing...
**Fairness and Starvation**

`pthread_mutex_lock` does not guarantee fairness

Thread 0

```c
pthread_mutex_lock(&mu);
processing...
pthread_mutex_unlock(&mu);
block and wait
processing...
```

Thread 1

```c
pthread_mutex_lock(&mu);
block and wait
processing...
pthread_mutex_unlock(&mu);
```

Thread 2

```c
block and wait
processing...
```

Thread 2 is starving!
Ticket Lock

typedef struct {
    pthread_mutex_t mu;
    pthread_cond_t cond;
    unsigned long queue_head, queue_tail;
} ticket_lock_t;
Ticket lock: lock

void lock(ticket_lock_t *tlock) {
  unsigned long my_ticket;
  pthread_mutex_lock(&tlock->mu);
  my_ticket = tlock->queue_tail++;
  while (my_ticket != tlock->queue_head) {
    pthread_cond_wait(&tlock->cond, &tlock->mu);
  }
  pthread_mutex_unlock(&tlock->mu);
}
void unlock(ticket_lock_t *tlock) {
    pthread_mutex_lock(&tlock->mu);
    tlock->queue_head++;
    pthread_broadcast(&tlock->cond, &tlock->mu);
    pthread_mutex_unlock(&tlock->mu);
}