Karlsruhe Institute of Technology

Operating Systems

13. Condition Variables and Semaphores

Prof. Dr. Frank Bellosa | WT 2020/2021

Concurrency Objectives

- Mutal Exclusion (e.g., thread A and B don't run at the same time)
  - solved with locks (mutex)
- Ordering (e.g., thread B runs after thread A did something)
  - solved with condition variables and semaphores

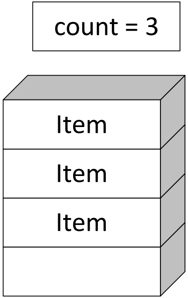
- Condition Variable: queue of waiting threads
  - B waits for a signal on CV before running **`wait(CV, ...)`**
  - A sends signal to CV when time for B to run **`signal(CV, ...)`**

Condition Variable

- **`wait(cond_t *cv, mutex_t *lock)`**
  - assumes the lock is held when wait() is called
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning
- **`signal(cond_t *cv)`**
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, just return, doing nothing
- **`broadcast(cond_t *cv)`**
  - wake all waiting threads (if >= 1 thread is waiting)
  - if there are no waiting thread, just return, doing nothing
- **Keep state in addition to CV's**
  - CV's are used to signal threads when state changes
  - If state is already as needed, thread doesn't wait for a signal!

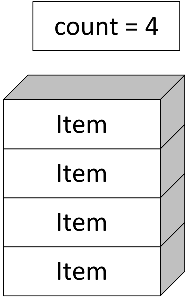
Producer-Consumer Problem

- Consider the producer-consumer problem (also known as bounded-buffer problem)
  - A buffer is shared between a producer and a consumer (here: LIFO)
  - An integer **`count`** keeps track of the number of currently available (previously produced) items
  - Every time, the producer produces an item, it places it in the buffer and increments **`count`**
  - When the buffer is full, the producer needs to sleep until the consumer consumed an item
  - When the consumer consumes an item, it removes the item from the buffer and decrements **`count`**
  - When the buffer is empty, the consumer needs to sleep until the producer produces an item



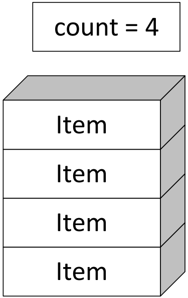
Producer-Consumer Problem

- Consider the producer-consumer problem (also known as bounded-buffer problem)
  - A buffer is shared between a producer and a consumer (here: LIFO)
  - An integer **`count`** keeps track of the number of currently available (previously produced) items
  - Every time, the producer produces an item, it places it in the buffer and increments **`count`**
  - When the buffer is full, the producer needs to sleep until the consumer consumed an item
  - When the consumer consumes an item, it removes the item from the buffer and decrements **`count`**
  - When the buffer is empty, the consumer needs to sleep until the producer produces an item



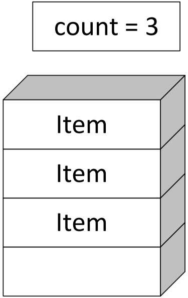
Producer-Consumer Problem

- Consider the producer-consumer problem (also known as bounded-buffer problem)
  - A buffer is shared between a producer and a consumer (here: LIFO)
  - An integer **`count`** keeps track of the number of currently available (previously produced) items
  - Every time, the producer produces an item, it places it in the buffer and increments **`count`**
  - When the buffer is full, the producer needs to sleep until the consumer consumed an item
  - When the consumer consumes an item, it removes the item from the buffer and decrements **`count`**
  - When the buffer is empty, the consumer needs to sleep until the producer produces an item



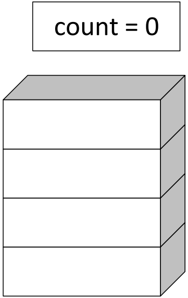
Producer-Consumer Problem

- Consider the producer-consumer problem (also known as bounded-buffer problem)
  - A buffer is shared between a producer and a consumer (here: LIFO)
  - An integer **`count`** keeps track of the number of currently available (previously produced) items
  - Every time, the producer produces an item, it places it in the buffer and increments **`count`**
  - When the buffer is full, the producer needs to sleep until the consumer consumed an item
  - When the consumer consumes an item, it removes the item from the buffer and decrements **`count`**
  - When the buffer is empty, the consumer needs to sleep until the producer produces an item



Producer-Consumer Problem

- Consider the producer-consumer problem (also known as bounded-buffer problem)
  - A buffer is shared between a producer and a consumer (here: LIFO)
  - An integer **`count`** keeps track of the number of currently available (previously produced) items
  - Every time, the producer produces an item, it places it in the buffer and increments **`count`**
  - When the buffer is full, the producer needs to sleep until the consumer consumed an item
  - When the consumer consumes an item, it removes the item from the buffer and decrements **`count`**
  - When the buffer is empty, the consumer needs to sleep until the producer produces an item



Producer-Consumer Problem

```
void producer()
{
   Item newItem;

   for(;;) // ever
   {
      newItem = produce();

      if( count == MAX_ITEMS )
         sleep();

      insert( newItem );
      count++;

      if( count == 1 )
         wake_up( consumer );
   }
}
```

```
void consumer()
{
   Item item;

   for(;;) // ever
   {
      if( count == 0 )
         sleep();

      item = remove();
      count--;

      if( count == MAX_ITEMS - 1 )
         wake_up( producer );

      consume( item );
   }
}
```

- Race condition on `count`

Non-Solution with mutex

```
void producer()
{
   Item newItem;

   for(;;) // ever
   {
      newItem = produce();

      if( count == MAX_ITEMS )
         sleep();
      mutex_lock( &lock );
      insert( newItem );
      count++;
      mutex_unlock( &lock );
      if( count == 1 )
         wake_up( consumer );
   }
}
```

```
void consumer()
{
   Item item;

   for(;;) // ever
   {
      if( count == 0 )
         sleep();
      mutex_lock( &lock );
      item = remove();
      count--;
      mutex_unlock( &lock );
      if( count == MAX_ITEMS - 1 )
         wake_up( producer );

      consume( item );
   }
}
```

- **`if`** statements can still be racy

Another non-Solution with mutex

```
void producer()
{
    Item newItem;

    for(;;) // ever
    {
        newItem = produce();
        mutex_lock( &lock );
        if( count == MAX_ITEMS )
            sleep();

        insert( newItem );
        count++;
        if( count == 1 )
            wake_up( consumer );
        mutex_unlock( &lock );
    }
}
```

```
void consumer()
{
    Item item;

    for(;;) // ever
    {   mutex_lock( &lock );
        if( count == 0 )
            sleep();

        item = remove();
        count--;
        if( count == MAX_ITEMS - 1 )
            wake_up( producer );
        mutex_unlock( &lock );
        consume( item );
    }
}
```

- One cannot work while the other sleeps with lock held (deadlock)

Final non-Solution with mutex

```
void producer()
{   [...]
    for(;;) // ever
    {
       newItem = produce();
       mutex_lock( &lock );
       if( count == MAX_ITEMS ) {
          mutex_unlock( &lock );
          sleep();
          mutex_lock( &lock );
       }
       insert( newItem );
       count++;
       if( count == 1 )
          wake_up( consumer );
       mutex_unlock( &lock );
    }
}
```

```
void consumer()
{   [...]
    for(;;) // ever
    {  mutex_lock( &lock );
       if( count == 0 )
       {
          mutex_unlock( &lock );
          sleep();
          mutex_lock( &lock );
       }
       item = remove();
       count--;
       if( count == MAX_ITEMS - 1 )
          wake_up( producer );
       mutex_unlock( &lock );
       consume( item );
    }
}
```

- Still racy and can cause wakeup loss

Solution with 2 Condition Variables

- Two condition variables: **`empty`** and **`fill`**

```
void producer()
{
  Item newItem;

  for(;;) // ever
  {
    newItem = produce();

    mutex_lock( &lock );
    while( count == MAX_ITEMS )
      cond_wait( &empty, &lock );

    insert( newItem );
    count++;
    cond_signal( &fill );
    mutex_unlock( &lock );
  }
}
```

```
void consumer()
{
  Item item;

  for(;;) // ever
  {
    mutex_lock( &lock );
    while( count == 0 )
      cond_wait( &fill, &lock );

    item = remove();
    count--;

    cond_signal( &empty );
    mutex_unlock( &lock );
    consume( item );
  }
}
```

Alloc-Free with cond_broadcast

```
void *allocate(int size) {
      mutex_lock(&m);
      while (bytesLeft < size)
       cond_wait(&c, &m);
      ...
  }
```

```
void free(void *ptr, int size) {
    ...
    cond_broadcast(&c)
    ...
   }
```

Rules of Thumb for CV's

- **Keep state in addition to CV's**
- **Always do wait/signal with lock held**
- **Whenever thread wakes from waiting, recheck state**
  - Use "while" instead of "if"
  - Some implementations also have "spurious wakeups" (may wake multiple waiting threads at signal or at any time)

Semaphore

- Introduce two syscalls that operate on data structure with integer element that we call semaphore
  - `sem_wait( &s )`: if `s` > 0: `s--` and continue. Otherwise let caller sleep.
  - `sem_post( &s )`: if no thread is waiting: `s++`. Otherwise wake one up.
- Initialize `s` to the maximum number of threads that may enter the CS at any given time
  - `sem_init( &s, initval)` user cannot write value after initialization
  - `sem_wait` corresponds to `enter_critical_section()`
  - `sem_post` corresponds to `leave_critical_section()`
  - If you want to be specific about your semaphore allowing more than one thread in the CS, you can call it counting semaphore
- A semaphore that is initialized to 1 is called binary semaphore, mutex semaphore or just mutex
  - A mutex only admits one thread into the CS at a time

Condition Variables vs. Semaphores

- Condition variables have no state (other than waiting queue)
  - Programmer must track additional state
- Semaphors have state: track integer value
  - State cannot be directly accessed by user program,
    but state determines behavior of semaphore operations
  - Each semaphore is also associated with a wake-up queue
    - Weak semaphores Wake up a random waiting thread on **`post`**
    - Strong semaphores Wake up thread strictly in the order in which they started **`wait`**ing

Equivalence Claim

- Locks can be built from semaphores
- Condition Variables can be built from semaphores
- Semaphores can be built from locks + condition variables

Build Lock from Semaphore

```
typedef struct __lock_t {
    sem_t sem;
} lock_t;

void lock_init(lock_t *lock) {
    sem_init(&lock->sem, 1); // 1 thread can grab lock
}

void lock_acquire(lock_t *lock) {
    sem_wait(&lock->sem);
}

void lock_release(lock_t *lock) {
    sem_post(&lock->sem);
}
```

Building CV's over Semaphores

- Possible, but really hard to do right
- Read about Microsoft Research's attempts:
  http://research.microsoft.com/pubs/64242/ImplementingCVs.pdf

Build Semaphore from Lock and CV

```
typedef struct {
    int value;
    cond_t cond;
    lock_t lock;
} sem_t;

void sem_init(sem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    lock_init(&s->lock);
}
```

Build Semaphore from Lock and CV

```
sem_wait(sem_t *s) {
    lock_acquire(&s->lock);

    while (s->value <= 0)
        cond_wait(&s->cond, &s->lock);

    s->value--;

    lock_release(&s->lock);
}
```

```
sem_post(sem_t *s) {
    lock_acquire(&s->lock);

    s->value++;

    cond_signal(&s->cond);

    lock_release(&s->lock);
}
```

Bounded Buffer with Semaphores

```
sem_t emptyBuffer, fullBuffer; // counting semaphores
sem_t mutex; // binary semaphore

init() {
    sem_init(&emptyBuffer, ITEMS); // all slots empty
    sem_init(&fullBuffer, 0);      // no filled slots
    sem_init(&mutex, 1);           // mutex
}
```

Bounded Buffer with Semaphores

```
void producer() {

    for(;;) { // for ever
        sem_wait(&emptyBuffer);
        // wait for free slot
        sem_wait(&mutex);
        slot = findempty(&buffer);
        sem_signal(&mutex);

        fill(&buffer[slot]);
        sem_signal(&fullBuffer);
        // signal filled slot
    }
}
```

```
void consumer() {

    for(;;) { // for ever
        sem_wait(&fullBuffer);
        // wait for filled slot
        sem_wait(&mutex);
        slot = findfull(&buffer);
        sem_signal(&mutex);

        use(&buffer[slot]);
        sem_signal(&emptyBuffer);
        // signal use of slot
    }
}
```

Readers-Writers Problem

- Problem: Model access to shared data structures
  - Many threads compete to read or write the same data
  - Readers only read the data set; they do not perform any updates
  - Writers can both read and write
- Using a single mutex for read and write operations is not a good solution, as it unnecessarily blocks out multiple readers while no writer is present
- Idea: Locking should reflect different semantics for reading data and for writing data
  - If no thread writes, multiple readers may be present
  - If a thread writes, no other readers and writers are allowed

1^{st} Readers-Writers Problem: Readers Preference

- No reader should have to wait if other readers are already present

```
typedef struct _rwlock_t{
    int reader;
    sem_t lock;
    sem_t writelock;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 1);
    sem_init(&rw->writelock, 1);
}
```

1^{st} Readers-Writers Problem: Readers Preference

- No reader should have to wait if other readers are already present

```
void writer() {
   for(;;)  { // ever
      // generate data to write
      sem_wait(&rw->writelock);
      // write data
      sem_post(&rw->writelock);
   }
}
```

- Writers cannot acquire **`writelock`** until the last reader leaves the critical section

```
void reader() {
   for(;;) { // ever
      sem_wait(&rw->lock);
      rw->readers++ ;
      if (rw->readers == 1)
         sem_wait(&rw->writelock);
      sem_post(&rw->lock);
      // read data
      sem_wait(&rw->lock);
      rw->readers--;
      if (rw->readers == 0)
         sem_post(&rw->writelock);
      sem_post(&rw->lock);
   }
}
```

2^{nd} Readers-Writers Problem: Writers Preference

- No writer shall be kept waiting longer than absolutely necessary
- Code is analogous to 1^{st} readers-writers problem but with separate readers- and writers-counts
- Read “Concurrent Control with Readers and Writers” [CHP71]
- 1^{st} and 2^{nd} readers-writers problem have the same issue:
  - Readers preference → writers can starve
  - Writers preference → readers can starve

3^{rd} Readers-Writers Problem: Bounded Waiting

- No thread shall starve
- POSIX threads contains readers-writers locks to address this issue
- Multiple readers but only a single writer are let into the CS
- If readers are present while a writer tries to enter the CS then
  - don't let further readers in
  - block until readers finish
  - let writer in

References I

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Commun. ACM*, 14(10):667–668, October 1971.