CONCURRENCY: SEMAPHORES

Shivaram Venkataraman
CS 537, Spring 2023
Midterm: Solutions, Grades

Project 2 grades - late, spurious. Don't panic!

Next: Spring break!

P3, P4 grades
Concurrence abstractions

How can semaphores help with producer-consumer?

How to implement semaphores?

How do you use them?

How do you implement them?
RECAP
CONCURRENCY OBJECTIVES

**Mutual exclusion** (e.g., A and B don’t run at same time)
solved with *locks*

**Ordering** (e.g., B runs after A does something)
solved with *condition variables (with state)*
CONDITION VARIABLES

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
JOIN IMPLEMENTATION

Parent:

```c
void thread_join() {
    Mutex_lock(&m);        // w
    if (done == 0)          // x
        Cond_wait(&c, &m);  // y
    Mutex_unlock(&m);      // z
}
```

Child:

```c
void thread_exit() {
    Mutex_lock(&m);        // a
    done = 1;              // b
    Cond_signal(&c);       // c
    Mutex_unlock(&m);      // d
}
```

Rule of Thumb: Keep state in addition to CV’s!
Producer/Consumer with 1 buffer

Thread 1 state:

```c
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Thread 2 state:

```c
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```
WHAT ABOUT 2 CONSUMERS?

Can you find a problematic timeline with 2 consumers (still 1 producer)?

1 producer, 1 shared buffer, 2 consumers
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&cond, &m); // p3
        do_fill(i); // p4
        Cond_signal(&cond); // p5
        Mutex_unlock(&m); // p6
    }
}

void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m); // c1
        if (numfull == 0) // c2
            Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}
HOW TO WAKE THE RIGHT THREAD?

Use two condition variables!
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i); // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); //p6
    }
}

void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if (numfull == 0) // p2
            Cond_wait(&fill, &m);
        int tmp = do_get();
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}

Producer will wait on "empty"

Consumer will wait on "fill"
PRODUCER/CONSUMER: TWO CVS

void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i); // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); //p6
    }
}

void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if (numfull == 0)
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1
3. before consumer1 runs, consumer2 runs, grabs entry, sets numfull=0.
4. consumer2 then reads bad data.
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        while (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i); // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); // p6
    }
}

void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        while (numfull == 0) // p2
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}

No concurrent access to shared state
Every time lock is acquired, assumptions are reevaluated
- A consumer will get to run after every do_fill()
- A producer will get to run after every do_get()
Whenever a lock is acquired, **recheck assumptions** about state!

Another thread could grab lock in between signal and wakeup from wait.

Note that some libraries also have “spurious wakeups” (may wake multiple waiting threads at signal or at any time).

\[ \text{signal will wake up only one thread} \]

\[ \text{signal will wake up one or more threads} \]
SUMMARY: RULES OF THUMB FOR CVS

1. Keep state in addition to CV’s

2. Always do wait/signal with lock held

3. Whenever thread wakes from waiting, recheck state
QUIZ 17

What is the sequence of execution when the producer runs for one iteration followed by the consumer?

```
P1 P2 ↓ P4 P5 P6 c1 c2 ↓ c4 c5 c6 c7
  skip wait                      skip wait
```

What is the sequence of execution if the consumer runs first?

```
c1 c2 c3 P1 P2 P4 P5 P6 c4 c5 c6 c7
  ↓ wait                            ↓ signal
```

The variable 'loops' cannot be greater than the variable 'numfull'.

False. numfull → buffer size (1 in the case)
       loops can be bigger
INTRODUCING SEMAPHORES

Condition variables have no state (other than waiting queue)
  – Programmer must track additional state

Semaphores have state: track integer value
  – State cannot be directly accessed by user program, but state determines behavior of semaphore operations
SEMAPHORE OPERATIONS

Allocate and Initialize

```c
sem_t sem;
sem_init(sem_t *s, int initval) {
    s->value = initval;
}
```

User cannot read or write value directly after initialization.
SEMAPHORE OPERATIONS

**Wait or Test: sem_wait(sem_t*)**

- Decrements sem value by 1, Waits if value of sem is negative (< 0)

**Signal or Post: sem_post(sem_t*)**

- Increment sem value by 1, then wake a single waiter if exists

Value of the semaphore, when negative = the number of waiting threads
typedef struct __lock_t {
    sem_t sem;
} lock_t;

void init(lock_t *lock) {
    sem_init(lock->sem, 1)
}

void acquire(lock_t *lock) {
    sem_wait(lock->sem)
}

void release(lock_t *lock) {
    sem_post(lock->sem)
}

sem_init(sem_t*, int initial)
sem_wait(sem_t*): Decrement, wait if value < 0
sem_post(sem_t*): Increment value
    then wake a single waiter
JOIN WITH CV VS SEMAPHORES

Parent

```c
void thread_join() {
    Mutex_lock(&m); // w
    if (done == 0) // x
        Cond_wait(&c, &m); // y
    Mutex_unlock(&m); // z
}
```

Child

```c
void thread_exit() {
    Mutex_lock(&m); // a
    done = 1; // b
    Cond_signal(&c); // c
    Mutex_unlock(&m); // d
}
```

---

```
sem_t s;
sem_init(&s, ___ -);
```

- `sem_wait()`: Decrement, wait if value < 0
- `sem_post()`: Increment value, then wake a single waiter

---

```
void thread_join() {
    sem_wait(&s); // 1->not wait 0->not wait
}
```

```
void thread_exit() {
    sem_post(&s) -> incr 0 & wake up
}
```
PRODUCER/CONSUMER: SEMAPHORES #1

Single producer thread, single consumer thread
Single shared buffer between producer and consumer

Use 2 semaphores
- emptyBuffer: Initialize to ________
- fullBuffer: Initialize to ________

Producer
while (1) {
    sem_wait(&emptyBuffer);
    Fill(&buffer);
    sem_post(&fullBuffer);
}

Consumer
while (1) {
    sem_wait(&fullBuffer);
    Use(&buffer);
    sem_post(&emptyBuffer);
}
Single producer thread, single consumer thread
Shared buffer with \textbf{N} elements between producer and consumer
Use 2 semaphores
  \begin{itemize}
  \item emptyBuffer: Initialize to \underline{N}
  \item fullBuffer: Initialize to \underline{0}
  \end{itemize}

Producer
\begin{verbatim}
i = 0;
while (1) {
  \textcolor{red}{\rightarrow sem\_wait(&emptyBuffer);}  
  Fill(&buffer[i]);
  i = (i+1)\%N;
  \textcolor{red}{\rightarrow sem\_post(&fullBuffer);}  
}
\end{verbatim}

Consumer
\begin{verbatim}
j = 0;
While (1) {
  \textcolor{red}{\rightarrow sem\_wait(&fullBuffer);}  
  Use(&buffer[j]);
  j = (j+1)\%N;
  \textcolor{red}{\rightarrow sem\_post(&emptyBuffer);}  
}
\end{verbatim}
PRODUCER/CONSUMER: SEMAPHORE #3

Final case:
  – Multiple producer threads, multiple consumer threads
  – Shared buffer with N elements between producer and consumer

Requirements
  – Each consumer must grab unique filled element
  – Each producer must grab unique empty element
PRODUCER/CONSUMER: MULTIPLE THREADS

Are my_i and my_j private or shared? Where is mutual exclusion needed??
Consider three possible locations for mutual exclusion
Which work?? Which is best??

Producer #1

\[
\text{lock}
\]
\[
\text{sem\_wait}(&\text{mutex});\]
\[
\text{sem\_wait}(&\text{emptyBuffer});
\]
\[
\text{my}_i = \text{findempty}(&\text{buffer});
\]
\[
\text{Fill}(&\text{buffer}[\text{my}_i]);
\]
\[
\text{sem\_post}(&\text{fullBuffer});
\]
\[
\text{sem\_post}(&\text{mutex});
\]

Consumer #1

\[
\text{sem\_wait}(&\text{mutex});\]
\[
\text{sem\_wait}(&\text{fullBuffer});
\]
\[
\text{my}_j = \text{findfull}(&\text{buffer});
\]
\[
\text{Use}(&\text{buffer}[\text{my}_j]);
\]
\[
\text{sem\_post}(&\text{emptyBuffer});
\]
\[
\text{sem\_post}(&\text{mutex});
\]

Dead lock → where consumer grabs mutex but is blocked on full buffer
PRODUCER/CONSUMER: MULTIPLE THREADS

Producer #2

```
sem_wait(&emptyBuffer);
sem_wait(&mutex);
myi = findempty(&buffer);
Fill(&buffer[myi]);
sem_post(&mutex);
sem_post(&fullBuffer);
```

Consumer #2

```
sem_wait(&fullBuffer);
sem_wait(&mutex);
myj = findfull(&buffer);
Use(&buffer[myj]);
sem_post(&mutex);
sem_post(&emptyBuffer);
```

Works, but limits concurrency:
Only 1 thread at a time can be using or filling different buffers
Producer/Consumer: Multiple Threads

Producer #3

```c
sem_wait(&emptyBuffer);
sem_wait(&mutex);
myi = findempty(&buffer);
sem_post(&mutex);
Fill(&buffer[myi]);
sem_post(&fullBuffer);

```

Consumer #3

```c
sem_wait(&fullBuffer);
sem_wait(&mutex);
myj = findfull(&buffer);
sem_post(&mutex);
Use(&buffer[myj]);
sem_post(&emptyBuffer);
```

Works and increases concurrency; only finding a buffer is protected by mutex; filling or using different buffers can proceed concurrently
READER/WRITER LOCKS

Let multiple reader threads grab lock (shared)
Only one writer thread can grab lock (exclusive)
  – No reader threads
  – No other writer threads

Let us see if we can understand code…
typedef struct _rwlock_t {
    sem_t lock;
    sem_t writelock;
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 1);
    sem_init(&rw->writelock, 1);
}
# Reader/Writer Locks

13  void rwlock_acquire_readlock(rwlock_t *rw) {
14      sem_wait(&rw->lock);
15      rw->readers++;
16      if (rw->readers == 1)
17         sem_wait(&rw->writelock);
18      sem_post(&rw->lock);
19  }
21  void rwlock_release_readlock(rwlock_t *rw) {
22      sem_wait(&rw->lock);
23      rw->readers--;
24      if (rw->readers == 0)
25         sem_post(&rw->writelock);
26      sem_post(&rw->lock);
27  }
29  rwlock_acquire_writelock(rwlock_t *rw) {  sem_wait(&rw->writelock);  }
31  rwlock_release_writelock(rwlock_t *rw) {  sem_post(&rw->writelock);  }

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T1: release_readlock()
T4: acquire_readlock()
T5: acquire_readlock()
T3: release_writelock()

// what happens next?
T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()

T4: acquire_writelock()
T5: acquire_writelock()
T6: acquire_readlock()

T8: acquire_writelock()
T7: acquire_readlock()
T9: acquire_readlock()
Typedef struct {
    int value;
    cond_t cond;
    lock_t lock;
} zem_t;

void zem_init(zem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    lock_init(&s->lock);
}

zem_wait(): Waits while value <= 0, Decrement
zem_post(): Increment value, then wake a single waiter
Build Zemaphore from Locks and CV

zem_wait(zem_t *s) {
    lock_acquire(&s->lock);
    while (s->value <= 0)
        cond_wait(&s->cond);
    s->value--;
    lock_release(&s->lock);
}

zem_post(zem_t *s) {
    lock_acquire(&s->lock);
    s->value++;
    cond_signal(&s->cond);
    lock_release(&s->lock);
}

zem_wait(): Waits while value <= 0, Decrement
zem_post(): Increment value, then wake a single waiter
Semaphores are equivalent to locks + condition variables
  – Can be used for both mutual exclusion and ordering
Semaphores contain **state**
  – How they are initialized depends on how they will be used
  – Init to 0: Join (1 thread must arrive first, then other)
  – Init to N: Number of available resources

Sem\_wait(): Decrement and then wait if < 0 (atomic)
Sem\_post(): Increment value, then wake a single waiter (atomic)
Can use semaphores in producer/consumer and for reader/writer locks
NEXT STEPS

Spring break!