CONCURRENCY: CONDITION VARIABLES

Shivaram Venkataraman
CS 537, Spring 2023
Midterm 1: later this week

Project 4:

after spring break: next project.

RECAP
Build higher-level synchronization primitives in OS.
Operations that ensure correct ordering of instructions across threads.
Use help from hardware.

Motivation: Build them once and get them right.

- Monitors
- Locks
- Semaphores
- Condition Variables

- Loads
- Stores
- Test&Set
- Disable Interrupts
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 and semaphores
typedef struct __lock_t {
    int flag;  // 0 unlocked
} lock_t;

void init(lock_t *lock) {
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    while (xchg(&lock->flag, 1) == 1);
    // spin-wait (do nothing)
}

void release(lock_t *lock) {
    lock->flag = 0;
}

int xchg(int *addr, int newval) {
    return *addr = newval;
    return old val
    // check if it was 1
}
typedef struct __lock_t {
    int ticket;
    int turn;
} __lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    // spin
    while (lock->turn != myturn);
}

void release(lock_t *lock) {
    FAA(&lock->turn);
}

Fairness

Fetch And Add

thread grabs a ticket

myturn

myturn = 0
get lock

myturn = 1
when turn = 1
SPINLOCK PERFORMANCE

Waste of CPU cycles?

- Without yield: \(O(\text{threads} \times \text{time_slice})\)
- With yield: \(O(\text{threads} \times \text{context_switch})\)

Even with yield, spinning is slow with high thread contention

Next improvement: Block and put thread on waiting queue instead of spinning

```c
write ( xchg(....) ) {
  if do nothing
  3
} 
```

New system calls telling scheduler that I am blocked on this lock.
typedef struct {
    bool lock = false;
    bool guard = false;
    queue_t q;
} LockT;

void acquire(LockT *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        qadd(l->q, tid);
        setpark(); // notify of plan
        l->guard = false;
        park(); // unless unpark()
    } else {
        l->lock = true;
        l->guard = false;
    }
}

void release(LockT *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else unpark(qremove(l->q));
    l->guard = false;
}
SPIN-WAITING VS BLOCKING

Each approach is better under different circumstances.

Uniprocessor
- Waiting process is scheduled → Process holding lock isn’t.
- Waiting process should always relinquish processor.
- Associate queue of waiters with each lock (as in previous implementation).

Multiprocessor
- Waiting process is scheduled → Process holding lock might be
- Spin or block depends on how long, \( t \), before lock is released
  - Lock released quickly → Spin-wait
  - Lock released slowly → Block
- Quick and slow are relative to context-switch cost, \( C \).
When to Spin-Wait? When to Block?

If you know how long, \( t \), before lock released, you can determine optimal behavior.

How much CPU time is wasted when spin-waiting?

\[
\text{waiting till lock is released} \quad t
\]

How much is wasted when blocking?

\[
\text{time taken for context switch} \quad C
\]

What is the best action when \( t < C \)?

Spin locks

When \( t > C \)?

Block lock

Problem:

Requires knowledge of future; too much overhead to do any special prediction.
Two-Phase Waiting

Theory: Bound worst-case performance; ratio of actual/optimal

When does worst-possible performance occur?

Spin for very long time $t >> C$

Ratio: $t/C$ (unbounded)

Algorithm: Spin-wait for $C$ then block $\rightarrow$ Factor of 2 of optimal

Two cases:

1. $t < C$: optimal spin-waits for $t$; we spin-wait $t$ too
2. $t > C$: optimal blocks immediately (cost of $C$);
   - we pay spin $C$ then block (cost of 2 $C$);
   - $2C / C = 2$-competitive algorithm
pthread_t p1, p2;

Pthread_create(&p1, NULL, mythread, "A");
Pthread_create(&p2, NULL, mythread, "B");

// join waits for the threads to finish
Pthread_join(p1, NULL);
Pthread_join(p2, NULL);

printf("main: done
[balance: %d]
[should: %d]", balance, max*2);

return 0;
Condition Variables

Condition Variable: queue of waiting threads

**B** waits for a signal on **CV** before running
  - \texttt{wait(CV, ...)}

**A** sends signal to **CV** when time for **B** to run
  - \texttt{signal(CV, ...)}

Ordering *between \texttt{A} and \texttt{B}*

\texttt{wait(CV, ...)}  \rightarrow  \texttt{block until some condition in \texttt{A} is true}

\texttt{signal(CV, ...)}
**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: ATTEMPT 1

Parent:

```c
void thread_join() {
    Mutex_lock(&m); // x
    Cond_wait(&c, &m); // y
    Mutex_unlock(&m); // z
}
```

Child:

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

Example schedule:

Parent:   wait until child exits

Child:    a    b    c

Parent:   x    y

Child:    a    b    c
JOIN IMPLEMENTATION: ATTEMPT 1

Parent:

```c
void thread_join() {
    Mutex_lock(&m); // x
    Cond_wait(&c, &m); // y
    Mutex_unlock(&m); // z
}
```

Child:

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

Example broken schedule:

Parent: `wail` Parent thread stuck!

Child: `a b c`
RULE OF THUMB: 1

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!
Join Implementation: Attempt 2

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() {
    done = 1;           // a
    Cond_signal(&c);    // b
}
```
JOIN IMPLEMENTATION: ATTEMPT 2

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() {
    done = 1;             // a
    Cond_signal(&c);      // b
}
void thread_join() {
    Mutex_lock(&m); // w
    if (done == 0) // x
        Cond_wait(&c, &m); // y
    Mutex_unlock(&m); // z
}

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

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

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

Use mutex to ensure no race between interacting with state and wait/signal
Possible outputs

1. Child thread executes first → Crash
2. Print 3. Parent \( p = \&x \)
   
   ```c
   Child printf();
   ```
   → \( \rightarrow \) \( \rightarrow \) \( \rightarrow \) \( \rightarrow \) NULL

3. Process exit before child executes

   ```c
   int *p = NULL; // global
   void child(void *arg) {
     printf("%d\n", *p);
   }
   ```

   ```c
   int main(int argc, char *argv[]) {
     thread_t p1;
     int x = 3;
     thread_create(&p1, child, NULL);
     p = &x;
     return 0;
   }
   ```

Fix the code?

- Move \( p = \&x \) before thread creation
- Process exit → terminates all threads
PRODUCER/CONSUMER PROBLEM
EXAMPLE: UNIX PIPES

A pipe may have many writers and readers.

Internally, there is a finite-sized buffer.

Writers add data to the buffer:
- Writers have to wait if buffer is full.

Readers remove data from the buffer:
- Readers have to wait if buffer is empty.

```
Cat test.txt | grep foo
```

```
producer / writer
```

```
fixed size = 10 entries = 40 KB
```

```
consumer
```
EXAMPLE: UNIX PIPES

start

Buf: [diagram of buffer]

end
EXAMPLE: UNIX PIPES

Implementation:
- reads/writes to buffer require locking
- when buffers are full, writers must wait
- when buffers are empty, readers must wait
PRODUCER/CONSUMER PROBLEM

Producers generate data (like pipe writers)

Consumers grab data and process it (like pipe readers)

Producer/consumer problems are frequent in systems (e.g. web servers)

General strategy use condition variables to:
  make producers wait when buffers are full
  make consumers wait when there is nothing to consume
Start with easy case:
- 1 producer thread
- 1 consumer thread
- 1 shared buffer to fill/consume (max = 1)

Numfull = number of buffers currently filled
# 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)?
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?

Wake all the threads!?  
Better solution (usually): use two condition variables
Producer/Consumer: Two CVs

```c
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
    }
}
```

```c
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);
    }
}
```
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)
            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)
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
NEXT STEPS

Next class: Semaphores