CONCURRENCY:
LOCKS AND CONDITION VARIABLES

Kai Mast

CS 537

Fall 2022

Announcements

- Consider using git for your projects
  - Much easier to understand what broke your tests
  - Allows backing up your code easily
- Midterm grades and solutions should (hopefully) be released today

RECAP: CONCURRENCY BASICS

Thread

- Separate string of execution within the same process
- All threads of a process share the same address space

Race Condition

- Multiple threads competing for the data
- Outcome is non-deterministic and depends on the scheduler

Critical Section

- Code that must be executed atomically
- Only one thread at a time should be in the critical section

Mutex

- Provides **mut**ual **excl**usion between threads
- Can be used to protect a critical section

RECAP: LOCK IMPLEMENTATION GOALS

Correctness

- *Mutual exclusion:* Only one thread in critical section at a time
- *Progress:* If several simultaneous requests, must allow one to proceed

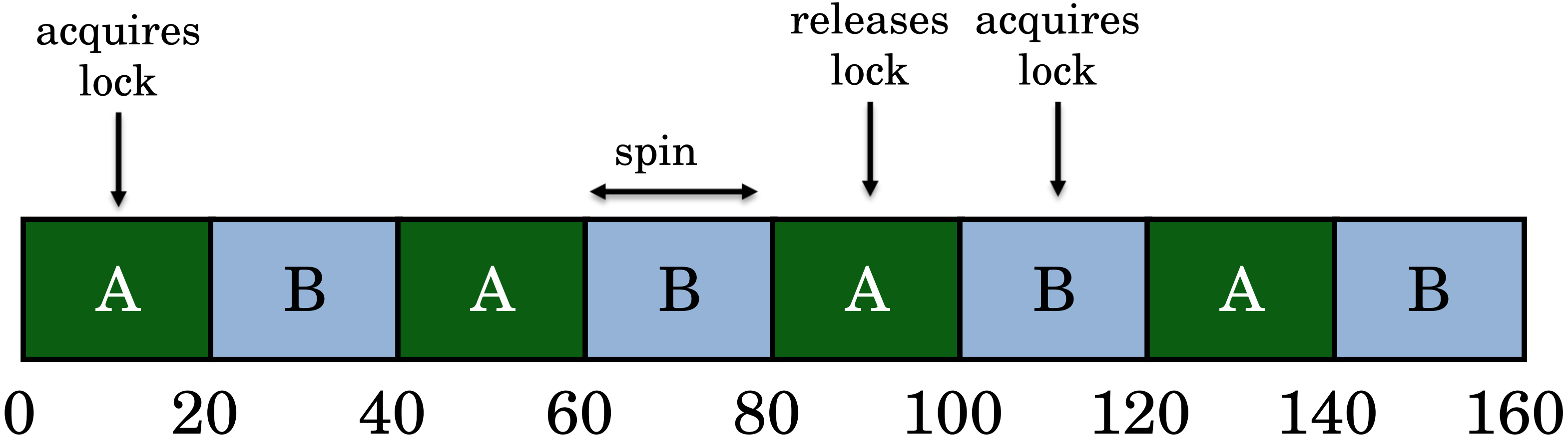
Performance:

- CPU is not used unnecessarily (e.g., no spin-waiting or busy-waiting)
- Fast to acquire lock if no contention with other threads (common case!)

Fairness: Each thread eventually gets a turn

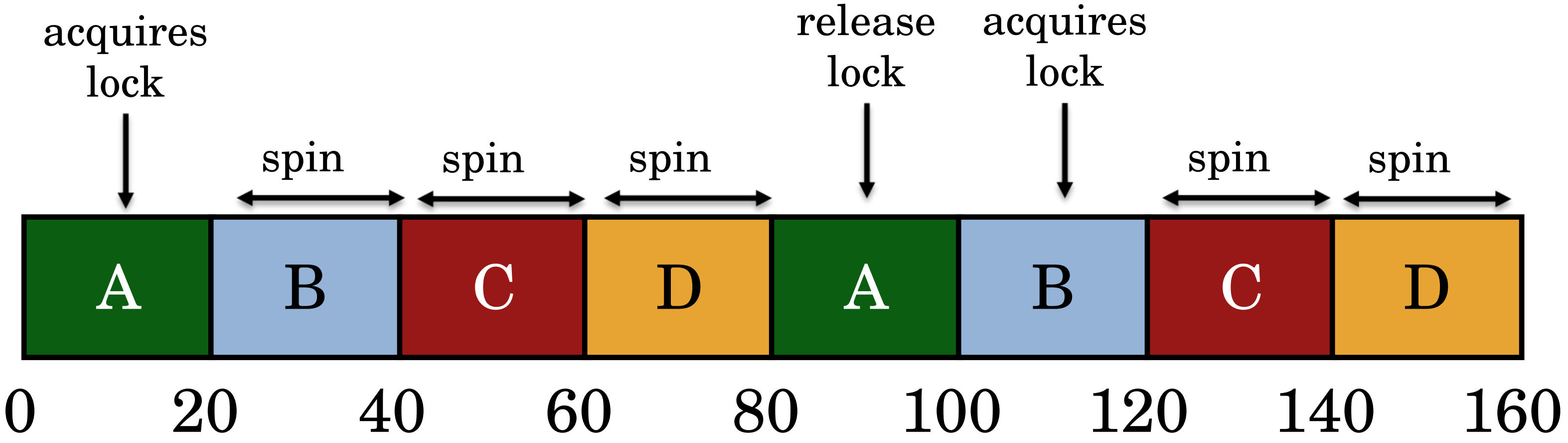
- Wait times are *bounded* (starvation-free)
- Must eventually allow each waiting thread to enter (assuming others eventually exit)

BASIC SPINLOCKS



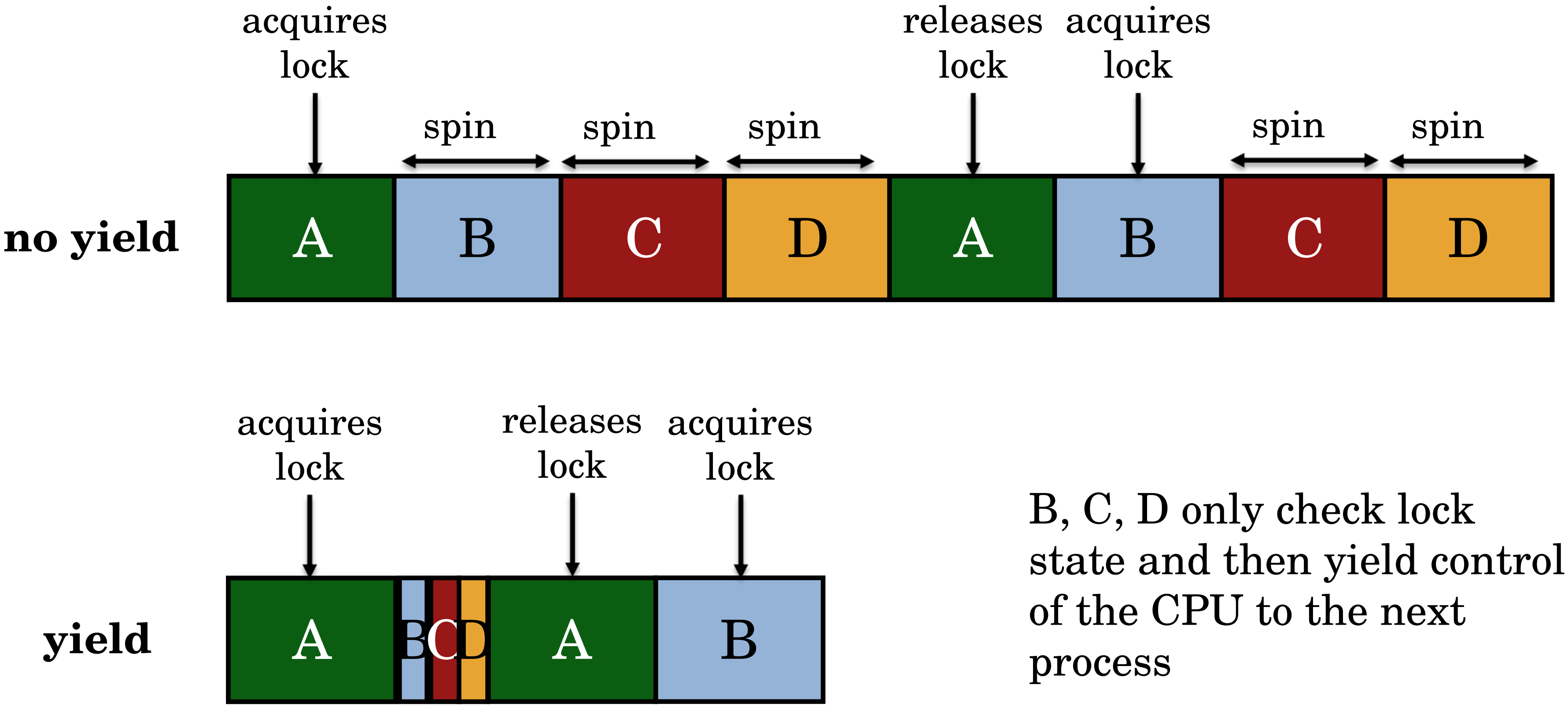
Scheduler is unaware of locks/unlocks!
B may be unlucky and never acquire lock

CPU SCHEDULER IS IGNORANT



CPU scheduler may run **B, C, D** instead of **A**
even though **B, C, D** are waiting for **A**

YIELD TO IMPROVE SPINNING



B, C, D only check lock state and then yield control of the CPU to the next process

SPINLOCK PERFORMANCE

Waste of CPU cycles?

Without yield: O(num_threads * **time_slice**)

With yield: O(num_threads * **context_switch_cost**)

Even with yield spinning is

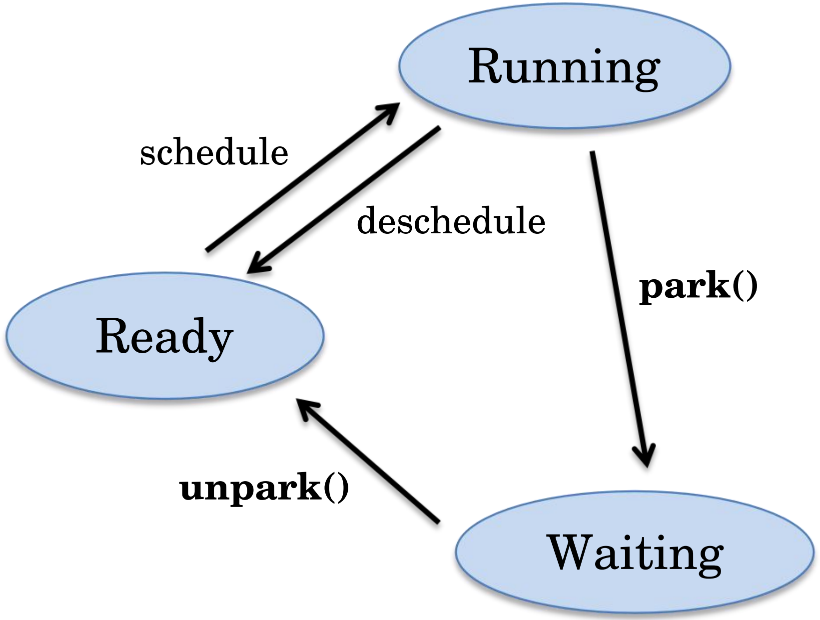
- Slow with high thread contention
- Unfair as it may starve threads

LOCK IMPLEMENTATION: PARK WHEN WAITING

Remove waiting threads from scheduler ready queue

Scheduler may run any thread that is **ready**

Good separation of concerns between lock and scheduler



REASONING ABOUT LOCKS

- When using locks, do not assume any implementation details are necessary for correctness of the calling process
- Do not assume any particular ordering for which process acquires the lock next
- Application code must work correctly regardless of which process acquires the lock next

LOCK USAGE
AND CONCURRENT DATA STRUCTURES

(Book Chapter 29)

REDUCING LOCK CONTENTION

- Locks ensure correct execution of a concurrent program
- But can also slow down the execution
  - Why? Only one thread can be in the critical section

Idea: Replace one "big" lock with many "small" locks

What could go wrong?

- Deadlocks: threads might wait for each other
- More potential bugs: Multiple locks are much harder to reason about

CONCURRENT COUNTERS

Goal: Increment a counter from multiple threads in a multi-core system

e.g., to log some system-wide metric (number of file accesses)

Naive Implementation

```
void increment(counter_t *c) {
    pthread_mutex_lock(&c->lock);
    c->value += 1;
    pthread_mutex_unlock(&c->lock);
}
```

Problem? Only one thread can increment the counter at a time

Possible Solution?

CONCURRENT COUNTERS

Idea: Have per-thread counters; periodically merge counter values

```
typedef struct __counter_t {
    int global; // global count
    pthread_mutex_t glock; // global lock
    int local[NUMCPUS]; // per-CPU count
    pthread_mutex_t llock[NUMCPUS]; // ... and locks
    int threshold; // update frequency
} counter_t;
```

CONCURRENT COUNTERS: INCREMENT

```
void update(counter_t *c, int threadID, int amt) {
    int cpu = threadID % NUMCPUS;
    pthread_mutex_lock(&c->llock[cpu]);
    c->local[cpu] += amt;
    if (c->local[cpu] >= c->threshold) {
        pthread_mutex_lock(&c->glock);
        c->global += c->local[cpu];
        pthread_mutex_unlock(&c->glock);
        c->local[cpu] = 0;
    }
    pthread_mutex_unlock(&c->llock[cpu]);
}
```

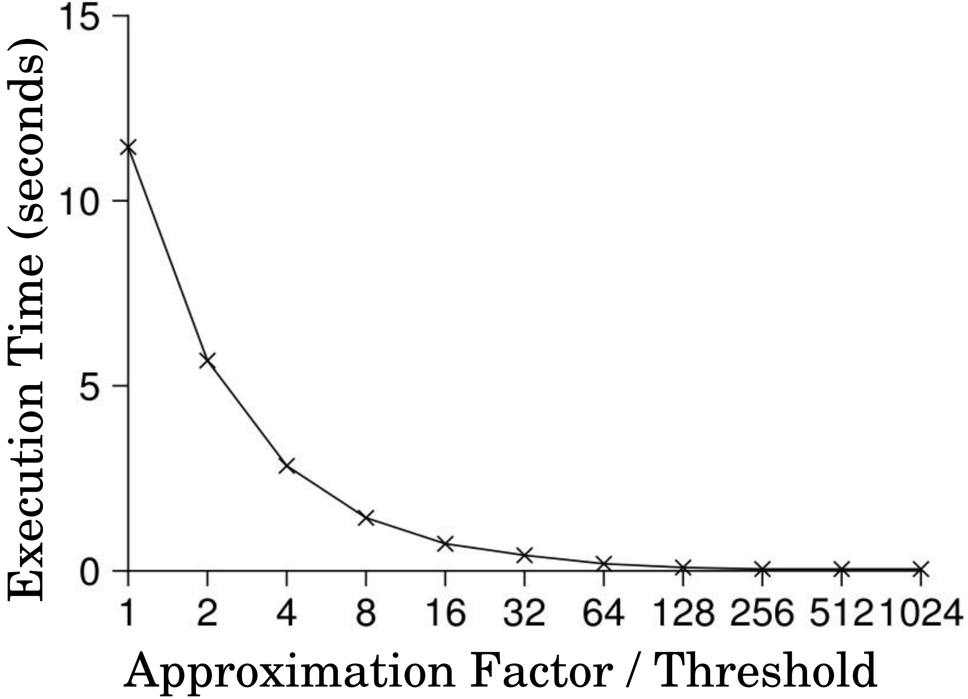
Concurrent Counters: Get

```
int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);
    int val = c->global;
    pthread_mutex_unlock(&c->glock);
    return val;
}
```

Limitations?

- Only returns an approximate value
- Read-heavy workloads can still cause lock contention

CONCURRENT COUNTERS: PRECISION



Higher threshold reduces lock contention, but reduces precision

CONCURRENT QUEUE

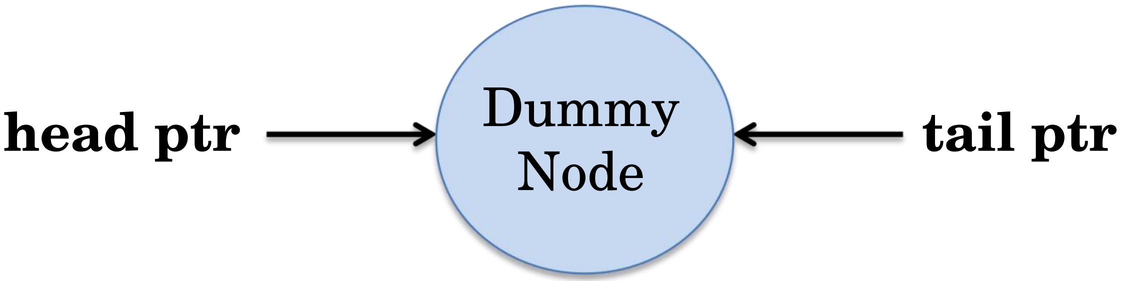
```c
typedef struct __node_t {
  int value;
  struct __node_t *next;
} node_t;

typedef struct __queue_t {
  node_t *head;
  node_t *tail;
  pthread_mutex_t head_lock
  pthread_mutex_t tail_lock;
} queue_t;
```

```c
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->head_lock,NULL);
  pthread_mutex_init(
       &q->tail_lock,NULL);
}
```

Two “small” locks instead one “big” lock

CONCURRENT QUEUE: INITIAL STATE



Why do we need the dummy node?
To make accesses to head and tail pointers independent

What value does the dummy node hold?
Undefined (we never set it explicitly)

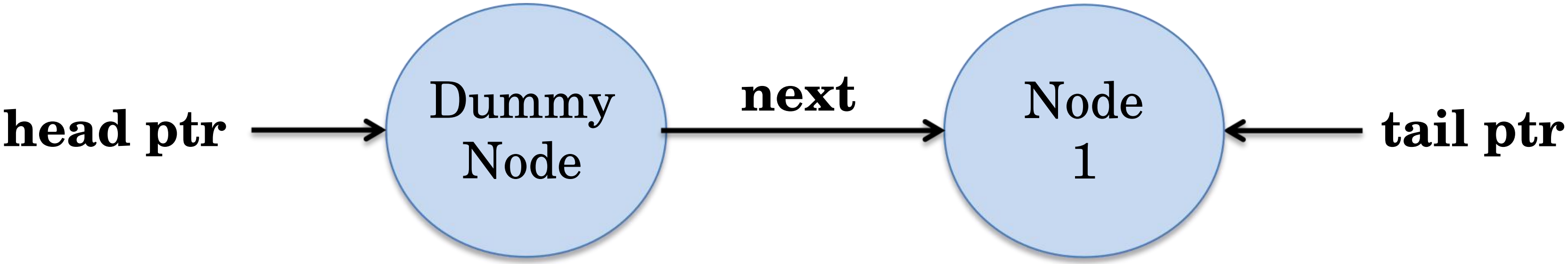
CONCURRENT QUEUES: ENQUEUE

```c
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 head_lock
  pthread_mutex_t tail_lock;
} queue_t;
```

```c
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->tail_lock);
  q->tail->next = tmp;
  q->tail = tmp;
  pthread_mutex_unlock(&q->tail_lock);
}
```

Concurrent Queue: After 1 Enqueue



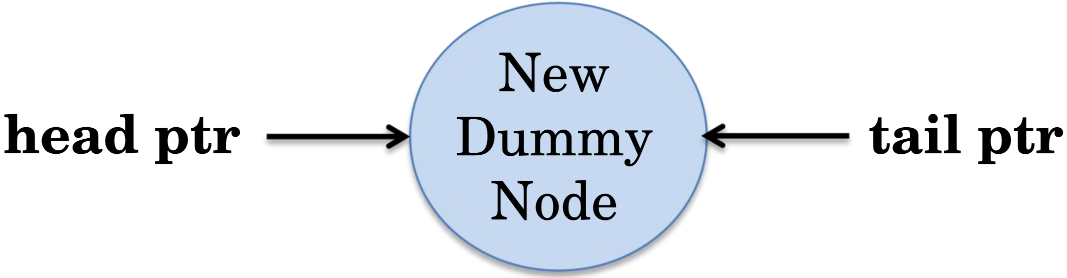
CONCURRENT QUEUES: DEQUEUE

```c
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 head_lock
  pthread_mutex_t tail_lock;
} queue_t;
```

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

Concurrent Queue: After Dequeue



Where did the dummy node come from?
It is what we previously called “Node 1”

What value does the new dummy node hold?
- Whatever “Node 1” previously held
- Does not matter, because we will never access the value again

DESIGNING CONCURRENT DATA STRUCTURES

Is the data structure the bottleneck of your application?

- Profile your code before starting to optimize, e.g., using valgrind
- If it is not the bottlenec, optimizing the data structure and its locking behavior might not affect performance much
  - Keep the DS and locking mechanism simple

What is the expected workload?

- How many readers/writers (or producers/consumers)?
- Is the workload mostly updates, mostly reads, or mixed?
- Build the locking mechanism around this access pattern

CONDITION VARIABLES

(Book Chapter 30)

Synchronization 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*

ORDERING EXAMPLE: JOIN

```
pthread_t p1, p2;
pthread_create(&p1, NULL, mythread, "A");
pthread_create(&p2, NULL, mythread, "B");

// join waits for specified thread to finish
pthread_join(p1, NULL);
pthread_join(p2, NULL);
```

How to implement join()?

CONDITION VARIABLES

Condition Variable: queue of waiting threads

X waits for a signal on particular CV before running
Y sends signal to particular CV when time for ***X*** to run

`wait``(cond_t *cv,  mutex_t *lock)`

- assumes the specified lock is held when wait() is called
- moves thread to blocked state + releases the lock (atomically)
- when awoken, reacquires lock before returning (Mesa semantics)

`signal``(cond_t *cv)`

- wake a single waiting thread (if more than one thread is waiting)
- if there is no waiting thread, just return doing nothing

JOIN IMPLEMENTATION: ATTEMPT 1

Shared State (in kernel)

```
typedef struct __thread {
    mutex_t mutex;
    condvar_t cond;
} thread_t;
```

Child

```
void thread_exit(thread_t *t) {
    cond_signal(&t->cond);
}
```

Parent

```
void thread_join(thread_t *t) {
    mutex_lock(&t->mutex);
    cond_wait(&t->cond, &t->mutex);
    mutex_unlock(&t->mutex);
}
```

Works!?

JOIN IMPLEMENTATION: PROBLEM 1

Parent

```
void thread_join(thread_t *t) {
    mutex_lock(&t->mutex);
    cond_wait(&t->cond, &t->mutex);
    mutex_unlock(&t->mutex);
}
```

Child

```
void thread_exit(thread_t *t) {
    cond_signal(&t->cond);
}
```

Child might exit before `thread_join` is called

- Remember, signal does nothing if there are no waiters

CV RULE #1

Keep state in addition to condition variables!

CV's are used to signal (wake up) threads when state changes

If state is already as needed, thread does not call wait on CV

JOIN IMPLEMENTATION: ATTEMPT 2

Shared State (in kernel)

```
typedef struct __thread {
    int done;
    mutex_t mutex;
    condvar_t cond;
} thread_t;
```

Child

```
void thread_exit(thread_t *t) {
    t->done = 1;
    cond_signal(&t->cond);
}
```

Parent

```
void thread_join(thread_t *t) {
    mutex_lock(&t->mutex);
    if (t->done == 0) {
        cond_wait(&t->cond, &t->mutex);
    }
    mutex_unlock(&t->mutex);
}
```

Works!?

JOIN IMPLEMENTATION: PROBLEM 2

Parent

```
void thread_join(thread_t *t) {
    mutex_lock(&t->mutex);
    if (t->done == 0) {
        cond_wait(&t->cond, &t->mutex);
    }
    mutex_unlock(&t->mutex);
}
```

Child

```
void thread_exit(thread_t *t) {
    t->done = 1;
    cond_signal(&t->cond);
}
```

Child sets done to 1 without holding a lock

CV RULE #2

Protect shared state in concurrent programs

- Hold the lock while changing the shared variable and calling `signal` or `broadcast` to avoid race conditions

JOIN IMPLEMENTATION: ATTEMPT 3

Shared State (in kernel)

```
typedef struct __thread {
    int done;
    mutex_t mutex;
    condvar_t cond;
} thread_t;
```

Child

```
void thread_exit(thread_t *t) {
    mutex_lock(&t->mutex);
    t->done = 1;
    cond_signal(&t->cond);
    mutex_unlock(&t->mutex);
}
```

Parent

```
void thread_join(thread_t *t) {
    mutex_lock(&t->mutex);
    if (t->done == 0) {
        cond_wait(&t->cond, &t->mutex);
    }
    mutex_unlock(&t->mutex);
}
```

Works!?

CV RULE #3

Always check state after waking up

Problem: Spurious wake-ups

- On some systems threads might be woken up even if `cond_signal/cond_broadcast` was not called
- Similarly, sometimes `cond_signal` will wake up more than one thread

Solution?

- We need to verify the state has changed as expected before continuing
- Use `while`, not `if` when waiting on a condition variable
  - Calls `cond_wait` again if state has not changed

JOIN IMPLEMENTATION: ATTEMPT 4

Shared State (in kernel)

```
typedef struct __thread {
    int done;
    mutex_t mutex;
    condvar_t cond;
} thread_t;
```

Child

```
void thread_exit(thread_t *t) {
    mutex_lock(&t->mutex);
    t->done = 1;
    cond_signal(&t->cond);
    mutex_unlock(&t->mutex);
}
```

Parent

```
void thread_join(thread_t *t) {
    mutex_lock(&t->mutex);
    while (t->done == 0) {
        cond_wait(&t->cond, &t->mutex);
    }
    mutex_unlock(&t->mutex);
}
```

Works? Yes!

SUMMARY

- Reducing lock granularity can improve performance but also increases likelihood of bugs
- Condition variables are an important primitive to order execution and detect state changes
- Always use condition variables in conjunction with locks; never have unprotected shared state

Next time: More fun with locks and condition variables!