# CISC 372: Parallel Computing
# Threads, part 3: condition variables

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

# Example: bank account

```
const int max = 10; // keep bal in 0..max
int bal = 0;
pthread_mutex_t mutex;
```

```
void * deposit_thread(void * arg) {
  while (1) {
    // WAIT UNTIL bal<10 ...
    pthread_mutex_lock(&mutex);
    bal++;
    pthread_mutex_unlock(&mutex);
  }
}
```

```
void * withdraw_thread(void * arg) {
  while (1) {
    // WAIT UNTIL bal>0 ...
    pthread_mutex_lock(&mutex);
    bal--;
    pthread_mutex_unlock(&mutex);
  }
}
```

- only want depositor to take the lock if `bal<10`
- only want withdrawer to take the lock if `bal>0`
- an example of the producer-consumer pattern

# Bad solution

```
while (true) {
  pthread_mutex_lock(&mutex);
  if (bal > 0) break;
  pthread_mutex_unlock(&mutex);
}
bal--;
pthread_mutex_unlock(&mutex);
```

- functionally, this is correct
- performance-wise: disaster
  - thread is constantly spinning, rechecking `bal` repeatedly, unnecessarily
  - …and taking and releasing lock
  - a thread that should be quietly waiting is instead constantly consuming resources (CPU)
  - if many threads do this: lock contention
  - performance grinds to a halt

# Monitors

- the "monitor" is a standard solution to this problem
- a concurrency concept introduced by Per Brinch Hansen and C.A.R. Hoare in early 1970s
- used in many programming languages/APIs
- Concurrent Pascal (1974, Hansen)
- Java: `synchronize`, `wait()`, `notify()`, `notifyAll()`
- Pthreads: condition variables and mutexes
- monitor = condition variable + mutex

# Condition variables

- a condition variable $c$ is used with a mutex
- when a thread owns the mutex, it may want to wait until some condition holds (due to actions of other threads)
- it can do this by waiting on $c$
- this reliquishes the locks and the thread goes to sleep
- other threads run
- at some point in future, another thread can issue a notification on $c$
- the thread that is asleep may be notified
  - it wakes up and has the opportunity to regain the lock once the thread owning the lock relinquishes it
  - typically, after the thread wakes up, it will check some condition
  - if the condition holds, great, it continues running with the lock
  - otherwise, it waits again (loops are good for this)

# Condition variables in Pthreads

- `pthread_cond_init(pthread_cond_t * cond, NULL)`
  - initialize a condition variable
- `int pthread_cond_destroy(pthread_cond_t * cond);`
  - destroy the previously initialized condition variable
- `int pthread_cond_signal(pthread_cond_t * cond);`
  - wake up one or more threads waiting on `cond`
- `int pthread_cond_broadcast(pthread_cond_t * cond);`
  - wake up all threads waiting on `cond`
- `int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mutex);`
  1. release lock on `mutex`
  2. go to sleep
  3. when woken up: try to regain lock on `mutex`

# Semantics of a condition variable `c`

- every thread is either running, blocked waiting for lock, or asleep
  - a sleeping thread is not contending for resources/consuming CPU cycles
  - note: I am using "asleep" here in a non-standard sense
- state of `c`: set of waiting threads ("wait-set")
- `wait` involves 3 atomic operations:
  1. release lock on mutex, state changes from running to asleep, thread added to `c`'s wait-set
  2. when signaled: state changes from asleep to blocked, thread removed from `c`'s wait-set
  3. later the thread may regain the lock on `mutex`
     - just like any thread trying to unlock `mutex`
     - once lock has been obtained, the call to `wait` returns
- `signal`
  - changes state of one or more waiting threads as above, removes them from `c`'s wait-set
  - usually called from thread that owns lock on `mutex`, but not required by Pthreads
- `broadcast`: signals all waiting threads, `c`'s wait-set become empty

# Typical pattern for using condition variables

```
obtain lock on mutex;
...
while (!expr) {
  wait on cond;
}
// at this point you know expr holds
// assuming expr can only be changed
// by a thread holding lock on mutex!
...
release lock on mutex;
```

# Bank account: bank1.c

```
const int max = 10; // keep bal in 0..max
int bal = 0;
pthread_mutex_t mutex;
pthread_cond_t balLT10, balGT0;
```

```
void * deposit_thread(void * arg) {
  while (1) {
    pthread_mutex_lock(&mutex);
    while (!(bal<max))
      pthread_cond_wait(&balLT10, &mutex);
    // now I know bal<10 and I have the lock
    bal++;
    pthread_cond_signal(&balGT0);
    pthread_mutex_unlock(&mutex);
  }
}
```

```
void * withdraw_thread(void * arg) {
  while (1) {
    pthread_mutex_lock(&mutex);
    while (!(bal>0))
      pthread_cond_wait(&balGT0, &mutex);
    // now I know bal>0 and I have the lock
    bal--;
    pthread_cond_signal(&balLT10);
    pthread_mutex_unlock(&mutex);
  }
}
```

# Generalized: `bank2.c`

- now allow multiple accounts, multiple depositors, multiple withdrawers
- a depositor randomly chooses an account and an amount
  - deposits the amount to the account (no waiting)
  - repeat forever
- a withdrawer randomly chooses an account and an amount
  - waits for the balance to be at least the amount
  - withdraws the amount from the account
  - repeat forever
- command line args: number of accounts, number of depositors, number of withdrawers
- solution
  - one mutex and one condition variable for each account
  - mutex guards all accesses to the account balance
  - condition variable signals whenever a deposit is made to the account
  - depositor signals every time it makes a deposit to the account
  - withdrawer waits, and upon being signaled, checks the balance

# Application: concurrency flags

- a flag is a boolean variable
- a concurrency flag is a shared boolean variable used in a particular disciplined way
  - also known as a "binary semaphore"
- concurrency `flags` are basic concurrency building blocks
- can be used to construct all kinds of complex synchronization patterns and data structures
  - mutual exclusion protocols, barriers, reductions, . . .
- state: a flag has two values, 0 and 1
- atomic operations
  - `raise`
    - can only be invoked when value is 0, otherwise error
    - sets value to 1
  - `lower`
    - blocks until value is 1, then sets value to 0 in one atomic step
    - no other thread can perform any operation on flag between check that value is 1 and set to 0

# Interface for flags: `flag.h`

```
typedef ... flag_t;

/* Initializes the flag with the given value.  Must be called before
   the first time the flag is used. */
void flag_init(flag_t * f, _Bool val);

/* Destroys the flag */
void flag_destroy(flag_t * f);

/* Increments f atomically, and returns the result.  Notifies threads
   waiting for a change on f. An assertion is violated if f is 1 when
   this function is called. */
void flag_raise(flag_t * f);

/* Waits for f to be 1, then sets it to 0, all atomically. */
void flag_lower(flag_t * f);
```

# Implementation of flags: `flags.h` and `flags.c`

```
typedef struct flag {
  _Bool val;
  pthread_mutex_t mutex;
  pthread_cond_t condition_var;
} flag_t;

void flag_init(flag_t * f, _Bool val) {
  f->val = val;
  pthread_mutex_init(&f->mutex, NULL);
  pthread_cond_init(&f->condition_var, NULL);
}

void flag_destroy(flag_t * f) {
  pthread_mutex_destroy(&f->mutex);
  pthread_cond_destroy(&f->condition_var);
}
```

# Implementation of flags: raise and lower

```
void flag_raise(flag_t * f) {
  pthread_mutex_lock(&f->mutex);
  assert(!f->val);
  f->val = 1;
  pthread_cond_broadcast(&f->condition_var);
  pthread_mutex_unlock(&f->mutex);
}

void flag_lower(flag_t * f) {
  pthread_mutex_lock(&f->mutex);
  while (f->val == 0)
    pthread_cond_wait(&f->condition_var, &f->mutex);
  f->val = 0;
  pthread_mutex_unlock(&f->mutex);
}
```

# Application of flags: barrier implementations

A very common pattern in multi-threaded programs:

```
while (true) {
  compute something;
  barrier();
}
```

- barrier(): no thread can leave until every thread has entered
- thread 1 needs to read something produced by thread 2 in previous iteration
- how to construct a “barrier” for threads?
- many ways, using synchronization primitives we have already learned
- solutions differ in their performance charactertistics
- desired characteristics of barriers:
  1. no one leaves until everyone enters
  2. no unnecessary delay: after last thread enters, everyone can leave without further delay
  3. reuseable : need to use the same barrier object over and over

# A 2-thread barrier using flags

- two flags are used `f1` and `f2`
  - `f1` is used by Thread 1 to send a signal to Thread 2 saying "I have arrived at barrier"
  - `f2` is used by Thread 2 to send a signal to Thread 1 saying "I have arrived at barrier"
- Thread 1
  1. raises `f1`
  2. lowers `f2`
- Thread 2
  1. lowers `f1`
  2. raises `f2`

Is it a correct, re-useable barrier with no unnecessary delay?

See `2barrier.c`.