Synchronization: Patterns

Condition Variables and Semaphores

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

Topics

- *Can we do something more powerful than just locking?*
  - **Condition variables** to “signal” other threads.
  - **Semaphores** to count how many things are available.
- *Can we allow multiple readers but only one writer?*
- *What can we solve with synchronization?*
  - How do **dining philosophers** help us with sychronization?
  - What’s a **circular buffer?**

Condition Variables

Producer-Consumer Pattern

- Producer-Consumer: a common programming pattern
  - **Producer(s):** one set of threads creating data
  - **Consumer(s):** second set of threads using the data
  - **Store data:** shared resource (e.g., variable or buffer) to hold the values that have been produced but not yet consumed
    - This is the shared resource needs protection

Audience Participation - Data Race

- Is there a data race in this code?

```
static int avail = 0;

int main() {
  pthread_t t1;
  pthread_create(&t1, NULL, thread_func,
NULL);

  for (;;) {
    while (avail > 0) {
      printf("I just consumed %d\n", avail);
      avail--;
    }
  }
  pthread_join(t1, NULL);
}
```

```
static void *thread_func(void *arg) {
  for (;;) {
    avail++;
    sleep(1);
  }

  return 0;
}
```

a) Yes, two threads change a shared variable.

b) No, one increments, the other decrements.

c) No, avail is static.

d) No, main()'s while loop prevents concurrent edits to a shared variable.

Producer-Consumer

```c
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int avail = 0;

int main() {
  pthread_t t1;
  pthread_create(&t1, NULL, thread_func, NULL);

  for (;;) {
    pthread_mutex_lock(&mtx);
    {
      while (avail > 0) {
        // Simulate "consume everything available"
        printf("I just consumed %d\n", avail);
        avail--;

      }
    }
    pthread_mutex_unlock(&mtx);
  }
  pthread_join(t1, NULL);
}
```

```c
static void *thread_func(void *arg) {
  for (;;) {
    pthread_mutex_lock(&mtx);
    {

      avail++;

    }
    pthread_mutex_unlock(&mtx);
    sleep(1);
  }

  return 0;
}
```

Simulate making something one at a time.

Simulate consuming something: decrement to 0

Use same mutex in both to serialize access to shared data.
Avoids data race problems.

Audience Participation - Efficiency

```c
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int avail = 0;

int main() {
  pthread_t t1;
  pthread_create(&t1, NULL, thread_func, NULL);

  for (;;) {
    pthread_mutex_lock(&mtx);
    {
      while (avail > 0) {
        // Simulate "consume everything available"
        printf("I just consumed %d\n", avail);
        avail--;

      }
    }
    pthread_mutex_unlock(&mtx);
  }
  pthread_join(t1, NULL);
}
```

```c
static void *thread_func(void *arg) {
  for (;;) {
    pthread_mutex_lock(&mtx);
    {
      avail++;
    }
    pthread_mutex_unlock(&mtx);
    sleep(1);
  }

  return 0;
}
```

What is the major source of inefficiency in this program?

a) Wasted space: Use of an int when a bool would be better for `avail`.

b) Wasted CPU: main keeps looping even when nothing to consume.

c) Wasted CPU: main locking & unlocking mutex when there are multiple values to consume.

d) Wasted CPU: Program will never end.

Producer-Consumer (with Error Checking)

```c
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int avail = 0;

int main() {
    pthread_t t1;
    int s = pthread_create(&t1, NULL, thread_func, NULL);
    if (s != 0) {
        perror("pthread_create");
        exit(1);
    }
    for (;;) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0) {
            perror("pthread_mutex_lock");
            exit(1);
        }
        while (avail > 0) {
            printf("I just consumed %d\n", avail);
            avail--;
        }
        s = pthread_mutex_unlock(&mtx);
        if (s != 0) {
            perror("pthread_mutex_unlock");
            exit(1);
        }
    }
    s = pthread_join(t1, NULL);
    if (s != 0) {
        perror("pthread_create");
        exit(1);
    }
}
```

```c
static void *thread_func(void *arg) {
    for (;;) {
        int s = pthread_mutex_lock(&mtx);
        if (s != 0) {
            perror("pthread_mutex_lock");
            pthread_exit((void *)1);
        }
        avail++;

        s = pthread_mutex_unlock(&mtx);
        if (s != 0) {
            perror("pthread_mutex_unlock");
            pthread_exit((void *)1);
        }
        sleep(1);
    }

    return 0;
}
```

Condition Variable

- *Condition variable purpose*
  - to signal a change in state
- *Using a condition variable*
  1) one thread sends a notification to the condition variable
  2) another thread waits until a notification is sent to the condition variable

  While waiting, the thread sleeps (no CPU use)

Integrates With Mutex

- We want to ensure that consumer(s) are thread safe
  - Expect the processing of a value to occur inside a mutex
  - But also: there is a potential notification race
  - A condition variable works closely with a mutex:

We need to hold the mutex while processing data so we don't corrupt the shared resource.

We'll wait until there is data available, but not hold the mutex while waiting!

That way the producer (or other consumers) can do work while we sleep.

pthread Condition Variables

- *Define the variable*

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- *Wait on a condition variable*

```
pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t
*mutex);
```

  - Internally, it will Atomically release the mutex and wait for cond
  - Once signalled, wakes up and *locks the mutex*
  - Why release mutex when waiting? Don't sleep while holding a lock!

- *Lock-safe sleep:* consumer can be sure that
  - Nothing added between unlocking mutex and waiting for cond (important because *a signal with no thread waiting is lost*)
  - Once woken up, it immediately holds the mutex

pthread Condition Variables (cont'd)

- *Wake up one thread waiting on cond*

```
pthread_cond_signal(pthread_cond_t *cond);
```

  - How many threads are waiting on cond?
    - 1: It wakes it up one thread
    - 2+: One wakes up, no control over which one
    - 0: Signal is lost: doesn't count how many signals are pending
- *Wake up all threads waiting on cond*

```
pthread_cond_broadcast(pthread_cond_t *cond);
```

  - All threads wake up and try to grab mutex; they compete for the lock

pthread Condition Variables (cont'd)

- *Guidelines for signaling*
  - `signal()` and `broadcast()` are similar; how to choose?
  - If *any one* of the waiting threads is sufficient to process the event, use `pthread_cond_signal()`
    - It's likely that all the threads do the same thing
  - If *all* of the waiting threads need to respond to an event, use `pthread_cond_broadcast()`
    - It's likely each thread does something different in response to the event; all need to happen
- Making sure a thread is actually responding to the condition can be tricky! Races are difficult to avoid here, check your work

Usage Pattern

Producer:

```
    pthread_mutex_lock(&mutex);

    <do some work producing an
item>


pthread_mutex_unlock(&mutex);

    pthread_cond_signal(&cond);
```

Consumer:

```
while(true) {
    pthread_mutex_lock(&mutex);

    while ( <no work to do> ) {
        pthread_cond_wait(&cond,
&mutex);
    }

    <do some work>

    pthread_mutex_unlock(&mutex);
}
```

- A condition variable must always use the same mutex
- Producer should signal after releasing mutex to avoid waking up a consumer with cond only to wait for mutex (extra context switch)
- Some systems optimize with "wait morphing" to just move process from one wait queue to another in the OS

Producer-Consumer With Condition Variable

```c
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

static int avail = 0;

int main() {
    pthread_t t1;
    pthread_create(&t1, NULL, thread_func, NULL);

    for (;;) {
        pthread_mutex_lock(&mtx);

        // This while loop is new.
        while (avail == 0) {
            pthread_cond_wait(&cond, &mtx);
        }

        while (avail > 0) {
            // Simulate "consume everything"
            printf("--> Consumer:%d.\n", avail);
            avail--;
        }

        pthread_mutex_unlock(&mtx);
    }
    pthread_join(t1, NULL);
}
```

```c
static void *thread_func(void *arg) {
    for (;;) {
        pthread_mutex_lock(&mtx);

        avail++;
        printf("Producer: %d.\n", avail);

        pthread_mutex_unlock(&mtx);

        // This signal is new.
        pthread_cond_signal(&cond);
        sleep(1);
    }
}
```

Analysis

- *Use of Condition Variables Discussion*
  - mutex still protects the shared variable avail
  - After producing an item, producer sends a signal to cond to wake up a waiting thread, if any: pthread_cond_signal(&cond)
    - This notifies other thread there is something to consume
  - At each iteration, consumer checks if there is any available item to consume (the new while loop).
    - If nothing's available (avail == 0), it sleeps: pthread_cond_wait()
    - This releases the mutex before sleeping
  - Consumer wakes up when signalled by the producer:
    - pthread_cond_wait() grabs mutex before returning

pthread_cond_wait() in loop?

- *Why put pthread_cond_wait() in a loop?*
  - Consumer only has work to do when `(avail != 0)`: called the **condition variable's predicate**
  - Consumer only waits if there is no data to process – for this, just ”`if(avail == 0)`” seems fine?
  - But, we must recheck the predicate after we are notified:
    - We were waiting on the mutex as well as cond, so another thread may have consumed the data first
    - Therefore, no guarantee after a wake-up that data is available

```
int main() {
  for (;;) {
    pthread_mutex_lock(&mtx);

    // This while loop is new.
    while (avail == 0) {
      pthread_cond_wait(&cond, &mtx);
    }

    while (avail > 0) {
      // Simulate "consume everything"
      avail--;
    }

    pthread_mutex_unlock(&mtx);
  }
}
```

Producer-Consumer with Condition Variable with Error Checking

```c
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static int avail = 0;

int main() {
    pthread_t t1;
    void *res;
    int s;

    s = pthread_create(&t1, NULL, thread_func, NULL);
    if (s != 0) {
        perror("pthread_create");
        exit(1);
    }

    for (;;) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0) {
            perror("pthread_mutex_lock");
            exit(1);
        }

        // This while loop is new.
        while (avail == 0) {
            s = pthread_cond_wait(&cond, &mtx);
            if (s != 0) {
                perror("pthread_mutex_lock");
                exit(1);
            }
        }

        while (avail > 0) {
            /* This is simulating "consume everything available" */
            printf("--> Consumer: avail at %d.\n", avail);
            avail--;
        }

        s = pthread_mutex_unlock(&mtx);
        if (s != 0) {
            perror("pthread_mutex_unlock");
            exit(1);
        }
    }
}
```

```c
static void *thread_func(void *arg) {
    for (;;) {
        int s = pthread_mutex_lock(&mtx);
        if (s != 0) {
            perror("pthread_mutex_lock");
            pthread_exit((void *)1);
        }
        avail++;
        printf("Producer: avail up to %d.\n", avail);

        s = pthread_mutex_unlock(&mtx);
        if (s != 0) {
            perror("pthread_mutex_unlock");
            pthread_exit((void *)1);
        }

        // This signal is new.
        s = pthread_cond_signal(&cond);
        if (s != 0) {
            perror("pthread_cond_signal");
            pthread_exit((void *)1);
        }
        sleep(1);
    }

    return 0;
}
```

Condition Variable Template for Consumer

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int main() {
    int s = pthread_mutex_lock(&mtx);

    if (s != 0) {
        perror("pthread_mutex_lock");
        exit(1);
    }

    while (/* Check if there is nothing to consume */) {
        /* Use while, not if, other threads might have woken
           up first and changed the shared variable. */
        pthread_cond_wait(&cond, &mtx);
    }

    // Do the necessary work with the shared variable, e.g., consume.

    s = pthread_mutex_unlock(&mtx);
    if (s != 0) {
        perror("pthread_mutex_lock");
        exit(1);
    }
}
```

Semaphores

Semaphores

- *A semaphore is a lock with a count*
  - A lock (mutex) is either available or not available, i.e., binary.
  - A semaphore is more flexible: it indicates the availability as a count, i.e., how many are available.
- *Useful when availability is not binary but a count*
  - e.g., how many items are available to consume?
  - If the *availability count* is **0,** it means the semaphore is **unavailable**
  - If the *availability count* is **greater** than 0, it means the semaphore is **available**
  - Must initialize the semaphore with an availability count!

pthread Semaphore Functions

- Create & Initialize the semaphore

```
sem_t sem;
sem_init(sem_t *sem, int pshared, unsigned int value);
```

  - Sets current # available to value for sem
  - pshared indicates if sem is for threads (0) or processes (1)

pthread Semaphore Functions

- *Wait to "acquire" one of the semaphore's count*

  ```
  sem_wait(sem_t *sem);
  ```

  - If count is 0, blocks until count > 0
  - When count is > 0, decrements count and returns immediately
  - Does not (generally) guarantee mutual exclusion: more than one might be available!

- Signal to count-up the semaphore:

  ```
  sem_post(sem_t *sem);
  ```

  - If synchronizing access to *limited resources,* then posting can be like *releasing a resource*
  - E.g., allow at most 50 students registered in a course
  - If synchronizing between different sections of code, then it might indicate a new resource produced

Audience Participation - Semaphores

- Which of these creates a semaphore which behaves the same as a mutex?

```
sem_init(sem_t *sem, int pshared, unsigned int value);
```

a) sem_init(&sem, 0, 0);

b) sem_init(&sem, 0, 1);

c) sem_init(&sem, 0, 2);

d) sem_init(&mutex, 0, 10);

Semaphore Use Ideas

- *Places to use a Semaphore*
  - Can have a single section of code wait and then post to acquire and release the resources
  - Can have different parts of the code use them, such as:
    - Producer: post when an item is ready
    - Consumer: wait until an item is ready
    - May still need a mutex to protect shared data – what are we *actually* replacing here?

Read-Write Lock

Read-Write Lock

- *Read-write lock*
  - Another synchronization primitive
  - Allows either unlimited readers, or a single writer
    - Multiple readers can all read at the same time!
    - Nobody else can access data while anyone writes
- *Acquire lock for reading*

```
pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

  - Allows any thread(s) to grab rwlock for reading as long as there is no thread that holds it for writing
- *Acquire lock for writing*

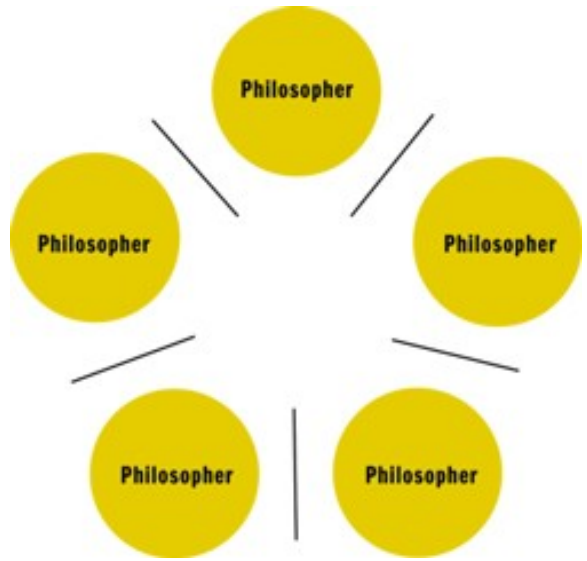
```
pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

  - This allows only one thread to grab rwlock for writing

Dining Philosophers

Dining Philosophers

- *Problem Description*
  - Philosophers sit at a round table
  - Philosophers alternate between eating and thinking
  - To eat, a philosopher needs two forks (at their left and right); to think, no forks are needed
  - One fork between adjacent philosophers: each fork is a resource shared by two adjacent philosophers
- *We can model this as a synchronization problem*
  - Each thread is a philosopher
  - A fork is a shared resource that only one should access at a time



Try 1: Big Lock!

- *Challenge*
  - come up with a solution that protects shared resources correctly and does not deadlock.
- *Try 1: One big lock (not efficient)*
  - Idea: **use one mutex to guard all forks and control access**
  - Correctly avoids deadlocks but allows only one philosopher to eat
  - Linux used to use this approach to protect kernel resource during a syscall: “the big kernel lock”

Try 2: Lock Each Fork

- *Try 2: One lock per fork*
  - Let's create a bad "solution":
  - **Have all threads grab their right fork and then their left fork**
  - But if every philosopher grabs their right fork at the same time, then no philosophers can grab their left fork
  - The result: deadlock due to hold-and-wait and circular-wait
- *Recall: deadlock conditions discussed previously*
  - We can break any of these conditions to avoid a deadlock:

1) Hold-and-wait
2) Circular wait
3) Mutual exclusion
4) No preemption

Possible Solutions

- *Solution 1*
  - **Have one philosopher grab forks in a different order**
  - E.g., designate a particular philosopher to grab right fork first, and the rest grab left fork first
  - Equivalently, label the forks, and always grab them in label order (think a minute about why this is equivalent if it’s not obvious)
  - This disrupts the circular-wait condition
- *Solution 2*
  - **Try grabbing both locks at once**
  - Grab the left lock. Try the right lock. If you can't grab it, give up the left lock, and try again
  - This breaks hold-and-wait condition, since no philosopher can both hold a fork and wait
  - This does not (deterministically) prevent starvation, and could also lead to livelock
    - But almost nothing is guaranteed anyway, and it requires less coordination

Dining Philosophers Implementation

```c
#define NUMBER 5

static pthread_mutex_t mtx[NUMBER] = {PTHREAD_MUTEX_INITIALIZER};


int main() {
  pthread_t t[NUMBER];

  for (int i = 0; i < NUMBER; ++i) {
    pthread_create(&t[i], NULL,
           thread_func, i);
  }

  for (int i = 0; i < NUMBER; ++i) {
    pthread_join(t[i], NULL);
  }
}
```

```c
static void *thread_func(void *arg) {
  int left = (int)arg;
  int right = ((int)arg + 1) % NUMBER;
  for (;;) {
    printf("Thread %d: thinking\n", (int)arg);
    sleep(5);

    pthread_mutex_lock(&mtx[left]);

    if (pthread_mutex_trylock(&mtx[right]) != 0) {
      pthread_mutex_unlock(&mtx[left]);
      continue;
    }

    printf("Thread %d: eating\n", (int)arg);

    pthread_mutex_unlock(&mtx[left]);

    pthread_mutex_unlock(&mtx[right]);
  }

  return 0;
}
```

Bounded Buffer
(Circular Buffer)

Bounded Buffer

- *Problem Description*
  - Multiple threads share a buffer
  - *Producer* threads place items *into* the buffer
    - They must wait if the buffer is *full*
  - *Consumer* threads take items *from* the buffer
    - They must wait if buffer is *empty*
- *Details*
  - Producers *place* items at head, from lower to higher indices, one at a time
  - Consumers *remove* items from tail, from lower to higher indices, one at a time – this is the *same order! (Sometimes also called a FIFO.)*
  - When you get to last element in the buffer, go back to 0
    - Think of it as a loop or ring (hence the many names: ring buffer, circular buffer)

Solution

- *Possible solution*
  - **Mutex + Condition Variable**
  - Mutex protects the data structure for all threads
  - Condition variable signals consumer (and producer?)
  - Can be inefficient, if threads are competing to check for availability
  - Worse problem: it's *easy to mess up and hard to debug*

Semaphores - More Elegant Solution

```c
#define SIZE 10
static char buf[SIZE] = {0};
static int in = 0, out = 0;
static sem_t filled_cnt;
static sem_t avail_cnt;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

int main() {
  pthread_t t1;
  sem_init(&filled_cnt, 0, 0);
  sem_init(&avail_cnt, 0, SIZE);

  pthread_create(&t1, NULL, thread_func, NULL);

  // Producer Code
  for (int i = 0;; i++) {
    sem_wait(&avail_cnt);
    pthread_mutex_lock(&mtx);

    // Produce
    buf[in] = i;
    printf("Produced: %d in %d\n", buf[in], in);
    in = (in + 1) % SIZE;

    pthread_mutex_unlock(&mtx);

    sem_post(&filled_cnt);
  }
  pthread_join(t1, NULL);
}
```

```c
static void *thread_func(void *arg) {
  for (;;) {
    sleep(1);
    sem_wait(&filled_cnt);
    pthread_mutex_lock(&mtx);

    // Consume
    printf("Consumed: %d\n", buf[out]);
    out = (out + 1) % SIZE;

    pthread_mutex_unlock(&mtx);

    sem_post(&avail_cnt);
  }

  return 0;
}
```

Summary

- *Condition Variable*
  - One thread signals another for an event.
  - Paired with a mutex for mutual exclusion.
- *Produce-Consumer Pattern*
  - Shared data structure storing waiting items.
- *Semaphore*
  - Synchronization with a count
- *Read-Write Lock*
  - Multiple readers allowed; only one writer.
- *Classing problems*
  - **Dining Philosophers:** worry about deadlock / livelock
  - **Bounded buffer:** elegant semaphore solution