CS202: Concurrency Tools

Where we are

- Threads
- Multiple simultaneously (logically) executing pieces of code can access the same memory = CONCURRENCY
- Hard to reason about what a CONCURRENT program does
  NEED TO CONSIDER ALL INTERLEAVINGS

GOAL: Figure out tools that make this easier.

\[
\begin{align*}
    x &\gets \text{acquire}(\text{mutex}) \\
    f &\begin{cases}
        x &= x + 2; \\
        x &= x \times 3;
    \end{cases} \quad \text{release}(\text{mutex}) \\
    \text{printf}(...) &\quad \text{printf}(\ldots, \text{"%d\n", } x) \\
    h &\quad x = x - 1;
\end{align*}
\]
mutex: Only one holder at a time

mutex_init

acquire

release

Things to keep in mind when using a mutex

acquire(&mutex);
if(x<5) {
    release;
    return;
} else {
    int y = exp(x, 2);
    y += 7;
    x = y;
    release;
A slightly more complex problem

\[ \text{buffer} \]

\[ \text{Producer} \]

\[ \text{Consumer} \]

Building this with a Mutex

\[ \text{mutex } m \]

Concerns with the Mutex Version

- Consumer unnecessarily acquires & releases mutex
Some General Ideas On Using Mutexes Correctly

- Make sure you always release them
  - Put critical section in its own function
  - Acquire on entry
  - Single return
- Or scoped locks

Condition Variables

```
# 1 (producer)
```

```c
// code
```

```c
signal that items available
```

```c
# 2 (consumer)
```

```c
while (...) \$
```
```plaintext
<table>
<thead>
<tr>
<th>BUFFER</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
</tr>
<tr>
<td>out</td>
</tr>
<tr>
<td>in_idx</td>
</tr>
<tr>
<td>out_idx</td>
</tr>
<tr>
<td>count</td>
</tr>
</tbody>
</table>

in = //receive from t1
expensive_op2(in);

// send out to t2
out = expensive_op0();

SIGNAL THAT SPACE IS AVAILABLE

MUTEX
COND. VAR

Condition Variable

cond_init(cv*, ...)

cond_wait(cv*c, mutex*m)

⇒ Caller must hold (have acquired) mutex m.
⇒ Unlocks/releases m & waits; blocks until c is "notified"
⇒ Locks m before returning.

cond_signal(cv*c, mutex*m)

⇒ ① Acquire m

⇒ Notify one thread waiting on c

⇒ Notify all threads

cond_broadcast(cv*c, mutex*m)
```
Observe

\[ \rightarrow \text{Mutex } m \]
\[ \text{proteces } c \]

\[ t1 \text{ decides to wait on } c \]
\[ (\text{count} == 0) \]

\[ t2 \text{ does work} \]
\[ \text{notifies } c \]

\[ t1 \text{ wait } (c) \]

\[ \times \text{BADXBADX} \]

Let us go look at condition variables in action (Handout 4)

Avoiding Bugs with Cond Vari

while loop around wait

if (!condition)
    cond-wait(...)
Semaphores

Semaphores

\[ \text{sem_init}(\text{semaphore}*, \text{int}) \]
\[ \text{sem_up}(\text{semaphore}*) \]
\[ \text{sem_down}(\text{semaphore}*) \]

Hardout 4

Please don't use semaphores in this class (or generally)

Monitors

One mutex + one or more cond vars

\[ \rightarrow \text{Jointly protect the same data structure.} \]
look at producer-consumer queue.

Rules for tools

1. Acquire mutex at the beginning of a function, release right before return
   Single return point is good!!

2. Use the same mutex \( M \) for all cond-wait/notify/boast calls that use a C.V.

3. While loop around cond-wait

4. **Don't call SLEEP!!**

Getting started

1. Identify units of concurrency
   - They should not be too large, they run
2. Identify the data that is accessed by this concurrent logic

3. Write down synchronization constraints → are they mutual exclusion or scheduling constraints?

4. Create mutexes & CVs for each constraint.

5. Write methods using mutexes & CVs.
Handout 3 gave examples of race conditions. The following panels demonstrate the use of concurrency primitives (mutexes, etc.). We are using concurrency primitives to eliminate race conditions (see items 1 and 2a) and improve scheduling (see item 2b).

1. Protecting the linked list......

```c
Mutex list_mutex;
List_elem* l = new List_elem;
l->data = data;
acquire(&list_mutex);
l->next = head;
head = l;
release(&list_mutex);
```

2. Producer/consumer revisited [also known as bounded buffer]

2a. Producer/consumer [bounded buffer] with mutexes

```c
Mutex mutex;
void producer (void *ignored) {
    while (true) {
        /* next line produces an item and puts it in nextProduced */
        nextProduced = means_of_production();
        acquire(&mutex);
        while (count == BUFFER_SIZE) {
            release(&mutex);
            yield(); /* or schedule() */
            acquire(&mutex);
        }
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        release(&mutex);
    }
}

void consumer (void *ignored) {
    while (true) {
        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
        buffer[out] = nextConsumed;
        out = (out + 1) % BUFFER_SIZE;
        count--;
        release(&mutex);
    }
}
```
2b. Producer/consumer [bounded buffer] with mutexes and condition variables

```
Mutex mutex;
Cond nonempty;
Cond nonfull;

void producer (void *ignored) {
    for (;;) {
        // next line produces an item and puts it in nextProduced */
        nextProduced = means_of_production();

        acquire(&mutex);
        while (count == BUFFER_SIZE)
            cond_wait(&nonfull, &mutex);
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal(&nonempty, &mutex);
        release(&mutex);
    }
}

void consumer (void *ignored) {
    for (;;) {
        // next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}
```

Question: why does cond_wait need to both release the mutex and
sleep? Why not:
```
while (count == BUFFER_SIZE) {
    cond_wait(&nonfull, &mutex);
    acquire(&mutex);
}
```

2c. Producer/consumer [bounded buffer] with semaphores

```
Semaphore mutex(1);        /* mutex initialized to 1 */
Semaphore empty(BUFFER_SIZE);  /* start with BUFFER_SIZE empty slots */
Semaphore full(0);        /* 0 full slots */

void producer (void *ignored) {
    for (;;) {
        // next line produces an item and puts it in nextProduced */
        nextProduced = means_of_production();

        sem_down(&empty);
        sem_down(&mutex);  /* get exclusive access */

        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        sem_up(&mutex);
        sem_up(&full);   /* we just increased the # of full slots */
    }
}

void consumer (void *ignored) {
    for (;;) {
        // next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}
```

Semaphores *can* (not always) lead to elegant solutions (notice
that the code above is fewer lines than 2b) but they are much
harder to use.

The fundamental issue is that semaphores make implicit (counts,
conditions, etc.) what is probably best left explicit. Moreover,
they *also* implement mutual exclusion.

For this reason, you should not use semaphores. This example is
here mainly for completeness and so you know what a semaphore
is. But do not code with them. Solutions that use semaphores in
this course will receive no credit.