Done with Processes for now

Today: Threads + Concurrency

Threads

More precisely:
- Each thread gets its own set of registers
- All threads in a process share the same memory.

thread-create

```c
void a(...) {
    tid = thread_create(void(*fn)(void*), void* arg);
    thread_join(tid);
    thread_exit();
    printf("NT: ");
    thread_create(&a, NULL);
}
```
**Contrast With Processes**

Two threads can **concurrently**

<table>
<thead>
<tr>
<th>Thread</th>
<th>$x=0$</th>
<th>$x=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y=1, y=0$</td>
<td>$y=1$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread</th>
<th>$x=1$</th>
<th>$x=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y=3$</td>
<td>$y=2$</td>
<td></td>
</tr>
</tbody>
</table>

**Concurrency**

- Each thread has its own copy of registers.
Isolated: One thread cannot manipulate another's registers

- All threads in a process have access to the same memory
  - Multiple changes to memory at a time
  - Many things happening at once

→ Goal: Find tools to build correct concurrent programs

Why?

Some assumptions

- Thread stacks isolated

Consistency model
Unless otherwise specified we will adopt **Sequential Consistency**:
- All threads agree on the order of operations (GLOBAL ORDER)
- The order of operations performed by a thread is preserved

**Critical Sections**

Things that should appear to occur atomically

Some concerns

Mutual exclusion
Implementing Critical Sections

- Critical section

- Single processor: Prevent anything else from running

How?

Multiple Processors
- Trackers
- Mutexes
Data

Critical Section

Code


Condition Variable

Data

Critical Section

Wait when condition holds

→ Mutual Exclusion

lock

unlock

cond-init (cond*)

cond-wait (Mutex*, Cond*)

cond-signal

cond-broadcast
1. Example to illustrate interleavings: say that thread tid1 executes f() and thread tid2 executes g(). (Here, we are using the term "thread" abstractly. This example applies to any of the approaches that fall under the word "thread").

   a. [this is pseudocode]

   ```c
   int x;
   int main(int argc, char** argv) {
      tid tid1 = thread_create(f, NULL);
      tid tid2 = thread_create(g, NULL);
      thread_join(tid1);
      thread_join(tid2);
      printf("%d\n", x);
   }
   void f() {
      x = 1;
      thread_exit();
   }
   void g() {
      x = 2;
      thread_exit();
   }
   ```

   What are possible values of x after tid1 has executed f() and tid2 has executed g()? In other words, what are possible outputs of the program above?

   b. Same question as above, but f() and g() are now defined as follows:

   ```c
   int y = 12;
   f() { x = y + 1; }
   g() { y = y * 2; }
   ```

   What are the possible values of x?

   c. Same question as above, but f() and g() are now defined as follows:

   ```c
   int x = 0;
   f() { x = x + 1; }
   g() { x = x + 2; }
   ```

   What are the possible values of x?
3. Producer/consumer example:

```c
/*
"buffer" stores BUFFER_SIZE items
"count" is number of used slots. a variable that lives in memory
"out" is next empty buffer slot to fill (if any)
"in" is oldest filled slot to consume (if any)
*/

void producer (void *ignored) {
    for (;;) {
        /* next line produces an item and puts it in nextProduced */
        nextProduced = means_of_production();
        while (count == BUFFER_SIZE);
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
    }
}

``` 

```c
void consumer (void *ignored) {
    for (;;) {
        while (count == 0);
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}
```

4. Some other examples. What is the point of these?

a. Can both "critical sections" run?
```
int flag1 = 0, flag2 = 0;

int main () {
    tid id = thread_create (p1, NULL);
    p2 (); thread_join (id);
}
```
```
void p1 (void *ignored) {
    flag1 = 1;
    if (!flag2) {
        critical_section_1 ();
    }
}
```
```
void p2 (void *ignored) {
    flag2 = 1;
    if (!flag1) {
        critical_section_2 ();
    }
}
```

b. Can use() be called with value 0, if p2 and p1 run concurrently?
```
int data = 0, ready = 0;

void p1 () {
    data = 2000;
    ready = 1;
}
```
```
void p2 () {
    while (!ready) {}
    use(data);
}
```

```c
c. Can use() be called with value 0?
```
```
int a = 0, b = 0;

void p1 (void *ignored) { a = 1; }
```
```
void p2 (void *ignored) {
    if (a == 1)
        b = 1;
}
```
```
void p3 (void *ignored) {
    if (b == 1)
        use(a);
}  ```
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 2) and improve scheduling (see item 2b).

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

```c
Mutex list_mutex;
insert(int data) {
    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) {
    for (;;) {
        /* 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) {
    for (;;) {
        acquire(&mutex);
        while (count == 0) {
            release(&mutex);
            yield(); /* or schedule() */
            acquire(&mutex);
        }
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        release(&mutex);
        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}
```
2b. Producer/consumer [bounded buffer] with mutexes and condition variables

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

void producer (void *ignored) {
    for (;;) {
        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 (;;) {
        acquire(&mutex);
        while (count == 0)
            cond_wait(&nonempty, &mutex);
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal(&nonfull, &mutex);
        release(&mutex);
        sem_up(&mutex);
        sem_up(&empty);   /* we just increased the # of full slots */
        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}
```

Question: why does cond_wait need to both release the mutex and sleep? Why not:

```c
while (count == BUFFER_SIZE) {
    cond_wait(&nonempty, &mutex);
    acquire(&mutex);
}
```

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.

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

```c
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 (;;) {
        nextProduced = means_of_production();
        sem_down(&empty);
        sem_down(&mutex);  /* get exclusive access */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        sem_up(&mutex);
        sem_up(&full);   /* we just increased the # of full slots */
        /* next line diminishes the count of empty slots and
        * waits if there are no empty slots */
        sem_down(&empty);
        sem_down(&mutex);
        /* exclusive access */
        buffer[out] = nextConsumed;
        out = (out + 1) % BUFFER_SIZE;
        sem_up(&mutex);
        sem_up(&empty);   /* one further empty slot */
        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}
```
kid a = thread - create ( )

kid b - create ( )

main ( )

if kid a = thread - create ( NULL)