Outline

▶ Announcements / Questions
▶ Race Conditions
  ▶ Abstract Discussion
  ▶ Examples
▶ Race Conditions in Tshlab
  ▶ waitpid()
  ▶ Solution attempts
Public Service Announcements

- Tshlab due on Thursday (10/22).
  - With late days, on Saturday (10/24).
- Malloclab released Thursday night.
- Exam 2 is next Thursday (10/29).
Any questions (unrelated to tshlab)?
This recitation is very dense. The material is not simple. Please ask questions as we go through it.
What is a race condition?

Imperative Language State

- Imperative language programmers make assumptions about execution ordering.
  - `foo(); bar(); baz();` has a known ordering.
- Each command modifies the state of the program.
  - Programmers also assume that the program is in a certain state before a command executes.
  - Each state assumption is called an invariant.
- This abstraction is a gigantic finite state machine.
- This abstraction is what is implicitly taught in 15-123.
An Extremely Abstract State Example

- Let $S(A)$ mean "Global State A".
- Suppose we have the following functions:
  - $\text{foo}() : S(A) \rightarrow S(B)$
  - $\text{bar}() : S(B) \rightarrow S(C)$
  - $\text{baz}() : S(C) \rightarrow S(A)$
- What is the end state of this code?
  - $\text{foo}(); \text{bar}(); \text{baz}();$
- What is the end state of this code?
  - $\text{foo}(); \text{foo}(); \text{bar}(); \text{baz}();$
- What invariants are violated?
What is a race condition, then?

- A race condition is when **concurrent** code asynchronously changes the program state in an unhandled, unexpected, or undesired manner.
  - We’ll have a few examples, of course.

- Concurrent code is code executing in any one of:
  - Another thread (in the same process).
  - Another process (that can change the current process).
  - Signal handlers (ah-hah!).
  - ...

- On the previous slide, the second `foo();` could have come from another thread, or in a signal handler!
What is a race condition?

That was the best complete definition I could think of.

- But it’s very abstract.
  - It may be useful to you when reviewing this recitation later.
- So we’ll go through some simple examples of race conditions.
  - Think about how the program state has been unwantedly changed.
Consider the following threads, which modify a global int sum=0.

<table>
<thead>
<tr>
<th>Adder Thread 1</th>
<th>Adder Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>add1():</td>
<td>add2():</td>
</tr>
<tr>
<td>int local = sum;</td>
<td>int local = sum;</td>
</tr>
<tr>
<td>local += 3;</td>
<td>local += 17;</td>
</tr>
<tr>
<td>sum = local;</td>
<td>sum = local;</td>
</tr>
</tbody>
</table>

We run both these threads at the same time, and then print out the value of sum. What are possible values for sum?
What we want to have happen:

<table>
<thead>
<tr>
<th>Time</th>
<th>Adder Thread 1</th>
<th>Adder Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int local = sum; // 3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>local += 3; // 3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>sum = local; // 3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>int local = sum; // 3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>local += 17; // 20</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>sum = local; // 20</td>
</tr>
</tbody>
</table>

At the end, sum == 20.
What could go wrong (1):

<table>
<thead>
<tr>
<th>Time</th>
<th>Adder Thread 1</th>
<th>Adder Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int local = sum; // 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>local += 3; // 3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>int local = sum; // 0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>local += 17; // 17</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>sum = local; // 17</td>
</tr>
<tr>
<td>5</td>
<td>sum = local; // 3</td>
<td></td>
</tr>
</tbody>
</table>

At the end, sum == 3! Hm.
What could go wrong (2):

<table>
<thead>
<tr>
<th>Time</th>
<th>Adder Thread 1</th>
<th>Adder Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>int local = sum; // 0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>local += 17; // 17</td>
</tr>
<tr>
<td>2</td>
<td>int local = sum; // 0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>local += 3; // 3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>sum = local; // 3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>sum = local; // 17</td>
</tr>
</tbody>
</table>

With this ordering, sum == 17.
How does this apply to Tshlab?

- We don’t use threads in Tshlab.
  - But we do use signal handlers...
- Consider the code on the next slide, which deals with reaping finished child processes.
  - Imagine that we want to wait for the foreground process.
  - Why is the following approach wrong?
Consider this method of waiting for a foreground process.

<table>
<thead>
<tr>
<th>eval(), Parent</th>
<th>SIGCHLD Handler</th>
</tr>
</thead>
</table>
| if (fg) {
  waitpid(child, ...);
} | pid = waitpid(.., WNOHANG ..);
  if (pid > 0) {
    print_status(pid);
  } |

- Sometimes this works, and the child status is printed.
- Sometimes \( \text{pid} = -1 \), and no status is ever printed.
  - What invariant has been violated?
  - How can we make sure that the status is always printed?
Well, let's try without `waitpid()`.

<table>
<thead>
<tr>
<th>eval(), Parent</th>
<th>SIGCHLD Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (fg) {</td>
<td>pid = waitpid(., WNOHANG .)</td>
</tr>
<tr>
<td>while(joblist(child));</td>
<td>if (pid &gt; 0) {</td>
</tr>
<tr>
<td>}</td>
<td>print_status(pid);</td>
</tr>
<tr>
<td></td>
<td>remove_from_joblist(pid);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

- Does this fix the race condition?
- Is this a good idea?
Maybe this fixes the problem:

```c
eval(), Parent
if (fg) {
    while(joblist(child)) {
        sleep(100);
    }
}
```

```c
SIGCHLD Handler
pid = waitpid(.., WNOHANG ..);
if (pid > 0) {
    print_status(pid);
    remove_from_joblist(pid);
}
```

- Does this work?
- Is this a good idea?
Questions?