Concurrent Examples

- Program a:
  ```sh
  #!/usr/bin/sh
  count=1
  while [ $count -le 20 ]
  do
    echo -n "a"
    count=`expr $count + 1`
  done
  ```
- Program b:
  ```sh
  #!/usr/bin/sh
  count=1
  while [ $count -le 20 ]
  do
    echo -n "b"
    count=`expr $count + 1`
  done
  ```

- When run sequentially (a; b) output is sequential.
- When run concurrently (a&; b&) output is interspersed and different from run to run.

Race Conditions

- A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.
  - E.g., If any code after a fork depends on whether the parent or child runs first.
- A parent process can call wait() to wait for termination (may block)
- A child process can wait for parent to terminate by polling (wasteful) (How would you do this?)
- Standard solution is to use signals.
**Example 1**

Process A

\[
x = \text{get(count)} \\
\text{write}(x + 1)
\]

```
x = 1 \\
\text{write}(2)
```

Process B

\[
y = \text{get(count)} \\
\text{write}(y + 1)
\]

```
y = 2 \\
\text{write}(3)
```

The value of count is what we expect.

---

**Example 2**

Process A

\[
x = \text{get(count)} \\
\text{write}(x + 1)
\]

```
x = 1 \\
\text{write}(2)
```

Process B

\[
y = \text{get(count)} \\
\text{write}(y + 1)
\]

```
y = 1 \\
\text{write}(2)
```

```
y = 2 \\
\text{write}(3)
```

```
\text{write}(2)
```

Not what we wanted!

---

**Example: Race Conditions**

```bash
#!/bin/sh
c=1
while [ $c -le 10 ]
do
    sd=`cat sharedData`
    sd=`expr $sd + 1`
    echo $sd > sharedData
    c=`expr $c + 1`
    echo d = $sd
done
#file sharedData must exist and hold
#one integer
```

---

**Producer/Consumer Problem**

- Simple example: `who | wc -l`
- Both the writing process (`who`) and the reading process (`wc`) of a pipeline execute concurrently.
- A pipe is usually implemented as an internal OS buffer.
- It is a resource that is concurrently accessed by the reader and the writer, so it must be managed carefully.
Producer/Consumer

- consumer should be blocked when buffer is empty
- producer should be blocked when buffer is full
- producer and consumer should run independently as far as buffer capacity and contents permit
- producer and consumer should never be updating the buffer at the same instant (otherwise data integrity cannot be guaranteed)
- producer/consumer is a harder problem if there are more than one consumer and/or more than one producer.

Protecting shared resources

- Programs that manage shared resources must protect the integrity of the shared resources.
- Operations that modify the shared resource are called critical sections.
- Critical section must be executed in a mutually exclusive manner.
- Semaphores are commonly used to protect critical sections.

Semaphores

- Code that modifies shared data usually has the following parts:
  - Entry section: The code that requests permission to modify the shared data.
  - Critical Section: The code that modifies the shared variable.
  - Exit Section: The code that releases access to the shared data.
  - Remainder: The remaining code.

- acquire(v)
  - block until the value of the semaphore variable v is greater than 0
  - then decrement v

- release(v)
  - increment v