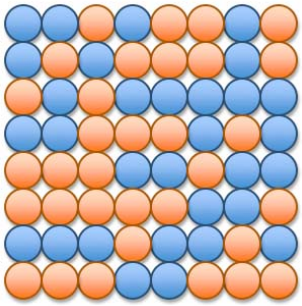


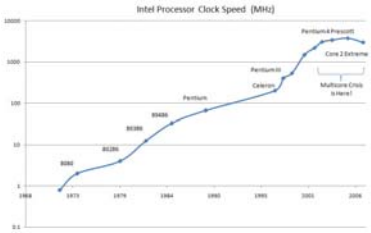
UNIT 10A
Multiprocessing & Deadlock

Why Multiprocessing ?

- Everything happens at once in the world. Inevitably, computers must deal with that world.
  - Traffic control, process control, banking, fly by wire, etc.
- It is essential to future speed-up of any computing process.
  - Google, Yahoo, etc. use thousands of small computers, even when a job could be done with one big computer.
  - Chips can't run any faster because they would generate too much heat.
  - Moore's law will allow many processors per chip.

Moore's Law vs. Clock Speed

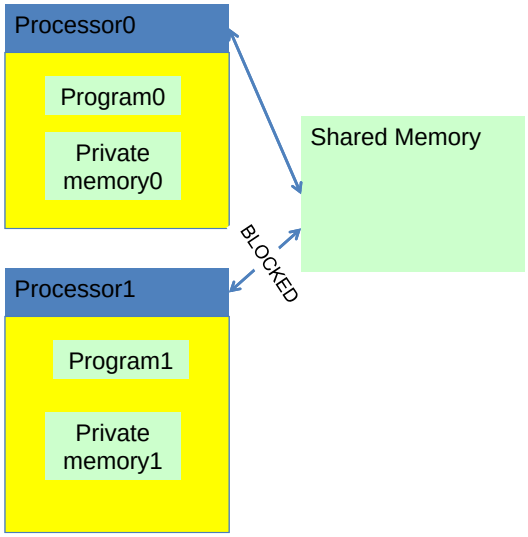
Source: Wikimedia Commons http://tinyurl.com/3d7qf3m



Source: Bob Warfield http://tinyurl.com/3pt6we9

A Ruby Multiprocessor Model

- The processors run independently.
- The shared memory is used for communication.
- Only one processor at a time may execute a line of Ruby that touches the shared memory. The memory hardware makes the others wait.

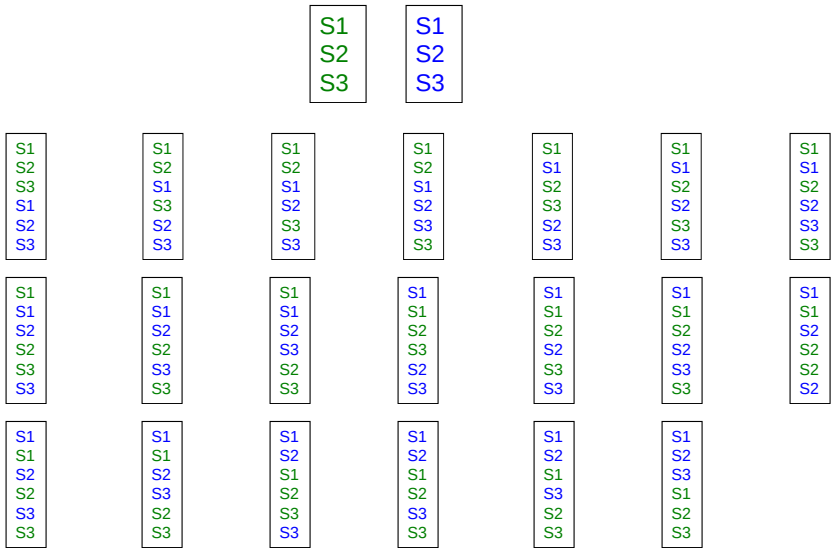


Multiprocessing is *very* hard.

- Only a tiny percentage of practicing programmers can do it.
- It requires art and mathematics.
  - It's like digital hardware design.
  - It needs proofs.
- Conventional debugging doesn't work.
  - If you stop the program to observe, you change the behavior.
  - Testing is futile because the number of possible execution sequences for the same input explodes.

5

There are many ways to execute two sequences in parallel.



6

Streams: One process sends, another receives.

```
# Shared
@full = false
@box = nil
```

```
#   Producer 0
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

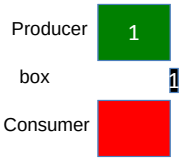
```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```


A Typical Execution Pattern

```
# Shared
@full = false
@box = nil
```

```
#   Producer 0
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```



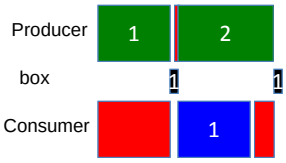
The producer creates item 1 and puts in it in the box while the consumer waits.

A Typical Execution Pattern

```
# Shared
@full = false
@box = nil
```

```
#   Producer 0
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```



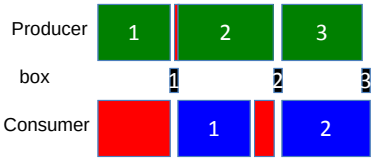
The producer creates item 2 while the consumer processes item1. The consumer happens to finish first so it waits until producer puts item 2 into the box.

A Typical Execution Pattern

```
# Shared
@full = false
@box = nil
```

```
#   Producer 0
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```

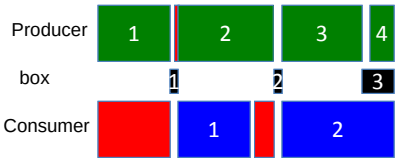


The producer finishes item 3 and puts it 3 into the box while the consumer still processes item 2 …

```
# Shared
@full = false
@box = nil
```

```
#   Producer 0
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```

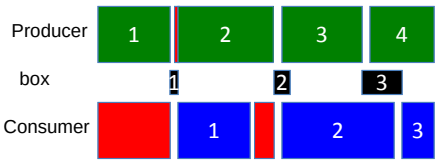


The producer starts on item 4 while the consumer still works on item 2. Item 3 waits in the box.


```
# Shared
@full = false
@box = nil
```

```
#   Producer 0
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```

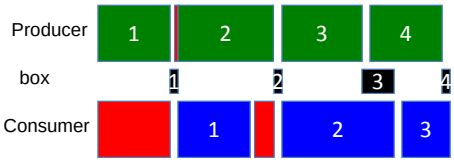



The consumer finishes item 2, picks up item 3 and starts working on it while the producer continues to work on item 4.

```
# Shared
@full = false
@box = nil
```

```
#   Producer 0
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```

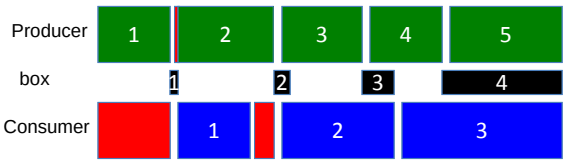


The producer finishes item 4 and puts it in the box while the consumer continues to work on item 3.


```
# Shared
@full = false
@box = nil
```

```
#   Producer 0
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```

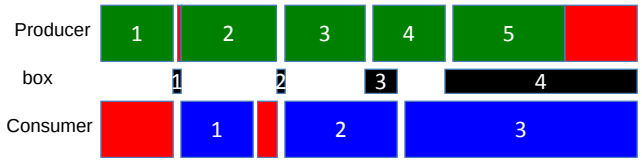


The producer starts on item 5 while the consumer continues to work on item 3. Item 4 waits in the box.

```
# Shared
@full = false
@box = nil
```

```
#   Producer 0
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```



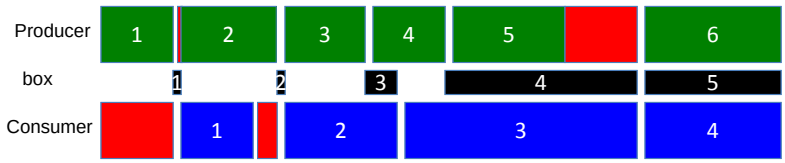
The producer finishes item 5, but the box is still full so it waits while the consumer continues to work on item 3.


```
# Shared
@full = false
@box = nil
```

```
#   Producer 0
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```

The consumer finishes item 3, takes item 4 from the box, and starts working on it. The producer puts item 5 in the box and starts working on item 6.

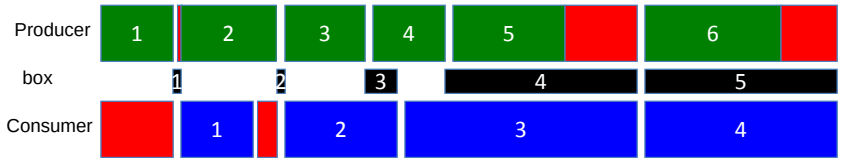


```
# Shared
@full = false
@box = nil
```

```
#   Producer 0
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```

The producer finishes working on item 6, but item 5 is still in the box so it waits while the consumer continues to work on item 4.

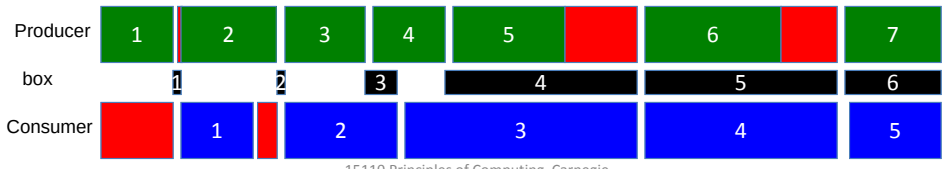



```
# Shared
@full = false
@box = nil
```

```
#   Producer 0
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```

The consumer finishes item 4, picks up item 5 from the box, and starts working on it. The producer puts item 6 in the box and starts working on item 7. And they live happily ever after!



Streams with a *Race Condition*

The order of accesses to @box and @full is very important!

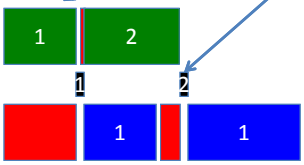
```
#   Producer
    while true do
      mail0 = whatever()
      while @full do #nothing
      end
      @full = true
      @box = mail0
end
```

BUG!

```
#   Consumer
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```

```
@full = true
@box = mail0 = 1
while !@full
mail1 = @box = 1
@full = false
```

```
@full = true
while !@full
mail1 = @box = 1
@full = false
@box = mail0 = 2
```



An unfortunate interleaving of process steps leads to a mistake.

Item 1 is processed twice!

19

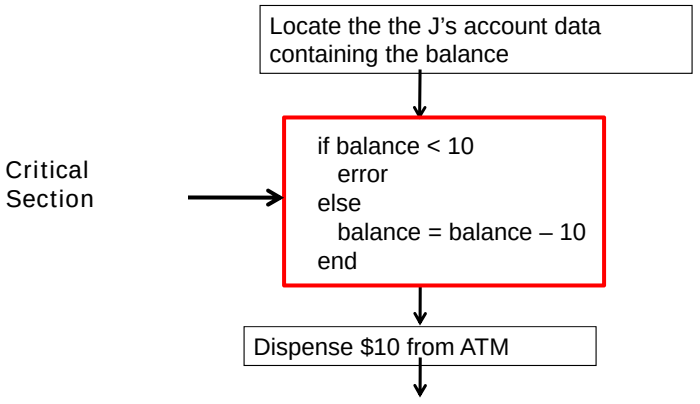
Critical Sections

- Often, a process really needs exclusive access to some data for more than one line.
- A **critical section** is a sequence of two or more lines that need exclusive access to the shared memory.
- Real Life Examples
  - Crossing a traffic intersection
  - A bank with many ATMs
  - Making a ticket reservation

Critical Section Example

- Consider a bank with multiple ATM's.
- At one, Mr. J requests a withdrawal of $10.
- At another, Ms. J requests a withdrawal of $10 from the same account.
- The bank's computer executes:
  1. For Mr. J, verify that the balance is big enough.
  2. For Ms. J, verify that the balance is big enough.
  3. Subtract 10 from the balance for Mr. J.
  4. Subtract 10 from the balance for Ms. J.
- The balance went negative if it was less than $20!

Critical Sections in Ruby



What can we do to prevent one processor from entering the critical section while another is in it?

Careful Driver Method

Don't enter the intersection unless it's empty.

In shared memory: `@free = true` `#initially unlocked`

```
#Process 1
while true do
  Non-Critical_Section
  while !@free do #nothing
   end
  @free = false
  Critical_Section
  @free = true
end
```

```
#Process 2
while true do
  Non-Critical_Section
  while !@free do #nothing
   end
  @free = false
  Critical_Section
  free = true
@end
```

Interference is possible!

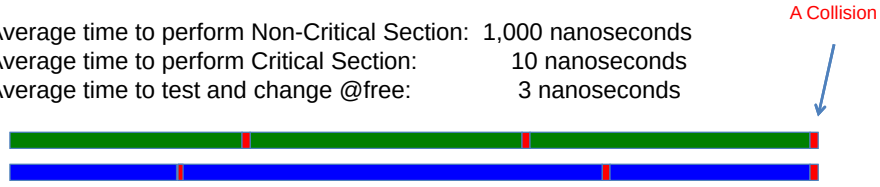
Computers vs. Real Life

- The careful driver method works in real life because
  - The number of times in your life you cross the intersection is low. Twice a day for forty years is about 29,000.
  - The chance of two drivers arriving at the intersection simultaneously is low.
  - Cars move slowly enough that if you don't see anyone coming, you'll get across before anyone comes.

The Probability of a Collision

```
while true do
  Non-Critical_Section
  while !@free do #nothing
   end
  @free = false
  Critical_Section
  @free = true
end
```

Average time to perform Non-Critical Section: 1,000 nanoseconds
Average time to perform Critical Section: 10 nanoseconds
Average time to test and change @free: 3 nanoseconds



Probability of one collision: 1/1,000 = 0.001
Iterations of outer loop in one second: 1,000,000,000/1,013 = 987,166
Probability of *no* collisions in 1 second: $(1\text{-}0.001)^{987,166} = (0.999)^{987,166} = 0$

The Stop Sign Method

1. Signal your intention (by stopping).
2. Wait until cross road has no one waiting or crossing.
3. Cross intersection.
4. Renounce intention (by leaving intersection).

The Stop and Look Method

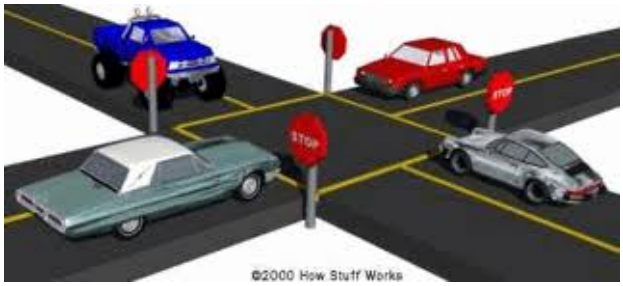
```
        # Shared Memory
         @free[0] = true    #P0 is not stopped at
        sign
         @free[1] = true    #P1 is not stopped at
# Process 0                    # Process 1
while true do                  while true do
   Non-Critical_Section           Non-Critical_Section
   @free[0] = false               @free[1] = false
   while !@free[1] do             while !@free[0] do
   end                            end
  Critical_Section               Critical_Section
  @free[0] = true                @free[1] = true
end                            end
```

Deadlock is possible!

Deadlock

- Deadlock is the condition when two or more processes are all waiting for some shared resource, but no process actually has it to release, so all processes to wait forever without proceeding.
- It's like gridlock in real traffic.

The Stop Sign Method with Tie Breaking



1. Signal your intention (by stopping).
2. Wait until cross road has no one else waiting or crossing.
3. If two of you are both waiting, yield to the car to your right.
4. Cross intersection.
5. Renounce intention (by leaving intersection).

Stop Sign with Tie Breaking

```
@free[1] = true
@free[2] = true
```

```
# Process 1
   while true
      Non-Critical_Section1
      @free[1] = false
      while !@free[2] do
      end
      Critical_Section1
      @free[1] = true
   end
```

```
#  Polite-Process 2
 while true do
  Non-Critical_Section2
  @free[2] = false
  while !@free[1] do
    @free[2] = true
    while !@free[1] do
    end
    @free[2] = false
  end
  Critical_Section2
  @free[2] = true
end
```

Process 2 backs off when it detects a conflict.

30

Types of Race Condition Bugs

In decreasing order of seriousness:

1. Interference: multiple process in critical section.
2. Deadlock: two processes idle forever, neither entering their critical or non-critical sections.
3. Starvation: one process needlessly idles forever while the other stays in its non-critical section.
4. Unfairness: a process has lower priority for no reason.

31

Peterson's algorithm avoids all bugs!

```
@free[0] = true
@free[1] = false
priority = 0
```

```
# Process 0
  while true do
    Non-Critical_Section0
    @free[0] = false
    priority = 1
    while !@free[1] and
          priority==1 do
    end
    Critical_Section0
    @free[0] = true
 end
```

```
#  Process 1
  while true do
    Non-Critical_Section1
    @free[1] = false
    priority = 0
    while !@free[0] and
          priority==0 do
    end
    Critical_Section1
    @free[1] = true
  end
```

A Probabilistic Approach

```
# Process 1
while true
  Non_Critical_Section1
  n1 = 0.000001 #microsecond
  @free[1] = false
  while !@free[2] do
      @free[1] = true
      sleep(rand(n1))
      n1 = 2 * n1
      @free[1] = false
  end
  Critical_Section1
  @free[1] = true
end
```

```
# Process 2
while true
   Non_Critical_Section2
   n2 = 0.000001
   @free[2] = false
   while !@free[1] do
       @free[2] = true
       sleep(rand(n2))
       n2 = 2 * n2
       @free[2] = false
   end
   Critical_Section2
  @free[2] = true
end
```

A Probabilistic Approach

```
# Process 1
while true
  Non_Critical_Section1
  n1 = 0.000001 #microsecond
  @free[1] = false
  while !@free[2] do
      @free[1] = true
      sleep(rand(n1))
      n1 = 2 * n1
      @free[1] = false
  end
  Critical_Section1
  @free[1] = true
end
```

```
# Process 2
while true
   Non_Critical_Section2
   n2 = 0.000001
   @free[2] = false
   while !@free[1] do
       @free[2] = true
       sleep(rand(n2))
       n2 = 2 * n2
       @free[2] = false
   end
   Critical_Section2
  @free[2] = true
end
```

A Probabilistic Approach

```
# Process 1
while true
  Non_Critical_Section1
  n1 = 0.000001 #microsecond
  @free[1] = false
  while !@free[2] do
      @free[1] = true
      sleep(rand(n1))
      n1 = 2 * n1
      @free[1] = false
  end
  Critical_Section1
  @free[1] = true
end
```

```
# Process 2
while true
   Non_Critical_Section2
   n2 = 0.000001
   @free[2] = false
   while !@free[1] do
       @free[2] = true
       sleep(rand(n2))
       n2 = 2 * n2
       @free[2] = false
   end
   Critical_Section2
  @free[2] = true
end
```

A Probabilistic Approach

```
# Process 1
while true
  Non_Critical_Section1
  n1 = 0.000001 #microsecond
  @free[1] = false
  while !@free[2] do
      @free[1] = true
      sleep(rand(n1))
      n1 = 2 * n1
      @free[1] = false
  end
  Critical_Section1
  @free[1] = true
end
```

```
# Process 2
while true
   Non_Critical_Section2
   n2 = 0.000001
   @free[2] = false
   while !@free[1] do
       @free[2] = true
       sleep(rand(n2))
       n2 = 2 * n2
       @free[2] = false
   end
   Critical_Section2
  @free[2] = true
end
```

A Probabilistic Approach

```
# Process 1
while true
  Non_Critical_Section1
  n1 = 0.000001 #microsecond
  @free[1] = false
  while !@free[2] do
      @free[1] = true
      sleep(rand(n1))
      n1 = 2 * n1
      @free[1] = false
  end
  Critical_Section1
  @free[1] = true
end
```

```
# Process 2
while true
   Non_Critical_Section2
   n2 = 0.000001
   @free[2] = false
   while !@free[1] do
       @free[2] = true
       sleep(rand(n2))
       n2 = 2 * n2
       @free[2] = false
   end
   Critical_Section2
  @free[2] = true
end
```

A Probabilistic Approach

```
# Process 1
while true
  Non_Critical_Section1
  n1 = 0.000001 #microsecond
  @free[1] = false
  while !@free[2] do
      @free[1] = true
      sleep(rand(n1))
      n1 = 2 * n1
      @free[1] = false
  end
  Critical_Section1
  @free[1] = true
end
```

```
# Process 2
while true
   Non_Critical_Section2
   n2 = 0.000001
   @free[2] = false
   while !@free[1] do
       @free[2] = true
       sleep(rand(n2))
       n2 = 2 * n2
       @free[2] = false
   end
   Critical_Section2
  @free[2] = true
end
```

A Probabilistic Approach

```
# Process 1
while true
  Non_Critical_Section1
  n1 = 0.000001 #microsecond
  @free[1] = false
  while !@free[2] do
      @free[1] = true
      sleep(rand(n1))
      n1 = 2 * n1
      @free[1] = false
  end
  Critical_Section1
  @free[1] = true
end
```

```
# Process 2
while true
   Non_Critical_Section2
   n2 = 0.000001
   @free[2] = false
   while !@free[1] do
       @free[2] = true
       sleep(rand(n2))
       n2 = 2 * n2
       @free[2] = false
   end
   Critical_Section2
  @free[2] = true
end
```

A Probabilistic Approach

```
# Process 1
while true
  Non_Critical_Section1
  n1 = 0.000001 #microsecond
  @free[1] = false
  while !@free[2] do
      @free[1] = true
      sleep(rand(n1))
      n1 = 2 * n1
      @free[1] = false
  end
  Critical_Section1
  @free[1] = true
end
```

```
# Process 2
while true
   Non_Critical_Section2
   n2 = 0.000001
   @free[2] = false
   while !@free[1] do
       @free[2] = true
       sleep(rand(n2))
       n2 = 2 * n2
       @free[2] = false
   end
   Critical_Section2
  @free[2] = true
end
```

A Probabilistic Approach

```
# Process 1
while true
  Non_Critical_Section1
  n1 = 0.000001 #microsecond
  @free[1] = false
  while !@free[2] do
      @free[1] = true
      sleep(rand(n1))
      n1 = 2 * n1
      @free[1] = false
  end
  Critical_Section1
  @free[1] = true
end
```

```
# Process 2
while true
   Non_Critical_Section2
   n2 = 0.000001
   @free[2] = false
   while !@free[1] do
       @free[2] = true
       sleep(rand(n2))
       n2 = 2 * n2
       @free[2] = false
   end
   Critical_Section2
  @free[2] = true
end
```

The probability of dithering is vanishingly small, proportional to $1/2^N$ for N collisions.

New Vocabulary

- Stream: A programming pattern in which one process sends data to another process sequentially
- Race Condition: A multiprocessing bug in which proper functioning depends upon luck
- Deadlock: A condition in which all processes are stalled waiting for each other
- Starvation: A condition in which a process is needlessly stalled

Takeaways

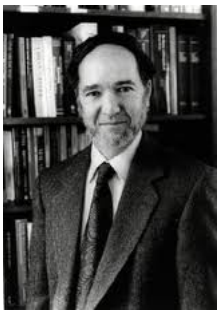
- Multiprocessing is very hard because controlling events in the real world is very hard.
- Sequential programming was a great invention because it made controlling simple things very easy.
- Leave it to the Engineers and hope they get it right.

Afterthoughts

Some counter-intuitive ideas about bugs and risks.

This man removed all the traffic lights and signs!

Why did Jared Diamond sleep under a tree when his aborigine companion wouldn't?



Why is a 1% chance of a bug biting better than a 0.1% chance?

- If there is a 1% chance of error, the bug will show up during 100 days of testing.
- If there is a 0. 1% chance, the bug will show up after three years when the system is deployed.

Economics as Multiprocessing

A national economy could be looked at a system with 1B independent processes representing buyers and sellers of goods. Consider the following economic maladies:

A. Depression
B. Bubbles
C. Income Inequality
D. Wasted productive resources

How do these problems correspond to the four multiprocessing problems?

1. Interference
2. Deadlock
3. Starvation
4. Unfairness

Hint: Think of entering a critical section as buying a good.