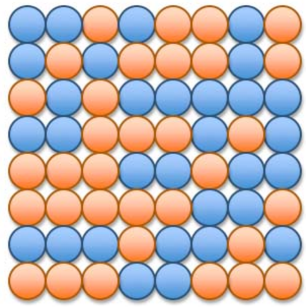
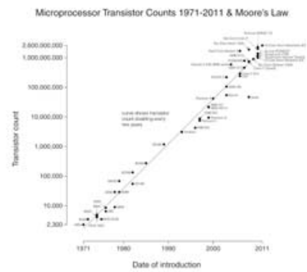


UNIT 10A
Multiprocessing & Deadlock

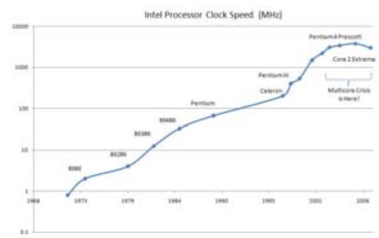
Why Multiprocessing ?

- 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.
- Even if your computer has one processor, a convenient way to cope with different external processes is to devote different internal, computer processes to each.

Moore's Law vs. Clock Speed



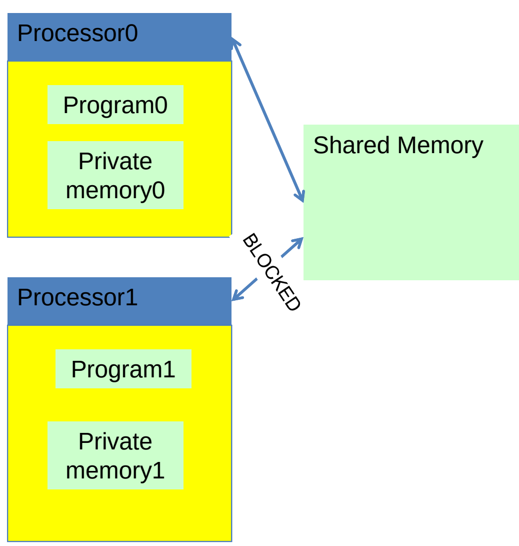
Source: Wikimedia Commons http://tinyurl.com/3d7qf3m



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

A Multiprocessor Model

- The processors run independently..
- The shared memory is used for communication.
- Only one processor at a time may execute an line of Ruby touching the shared memory. The memory hardware makes the other ones wait.



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
```

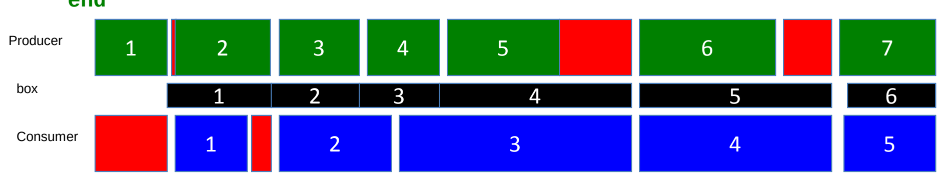
```
#   Consumer 1
    while true do
      while !@full do #nothing
      end
      mail1 = @box
      @full = false
      process(mail1)
    end
```


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
```



Streams with a *Race Condition*

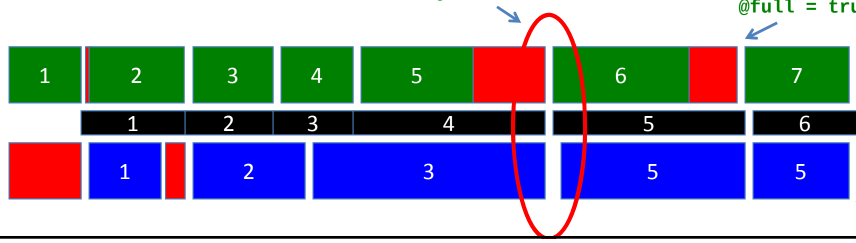
```
#   Producer 0
    while true do
      mail0 = whatever
      while @full do #nothing
      end
      @box = mail0
      @full = true
    end
```

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

The order of accesses to @box and @full is very important. Suppose the order of execution is

```
@full = false
while @full
@box = 5
mail1 = 5
@full = true
```

```
while !@full
@full = false
mail1 = 5
while @full
@box = 6
@full = true
```



Critical Sections

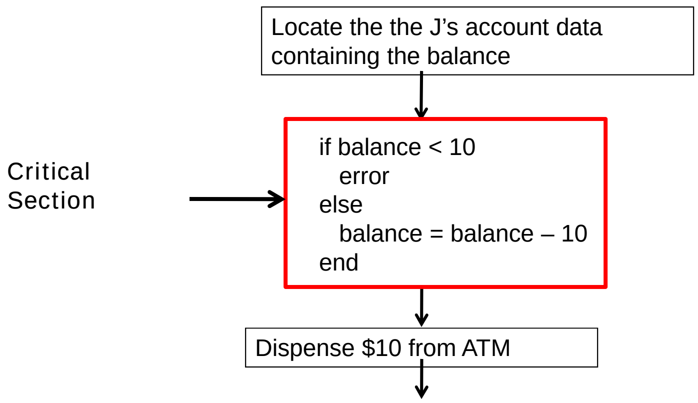
- Often, a process really needs exclusive access to some data for more than one line.
- A **critical section** is a sequence of two or more lines that need exclusive access to the shared memory.
- Real Life Examples
  - Crossing a traffic intersection
  - A bank with many ATMs
  - Making a ticket reservation

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?

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. (Not a bad bug.)

11

Careful Driver Method

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

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

```
#Process 1
while true do
  NonCriticalSection
  while !free do #nothing
   end
  free = false
  CriticalSection
  free = true
end
```

```
#Process 2
while true do
  NonCriticalSection
  while !free do #nothing
   end
  free = false
  CriticalSection
  free = true
end
```

Interference is possible!

15110 Principles of Computing, Carnegie Mellon University - MORRIS

12

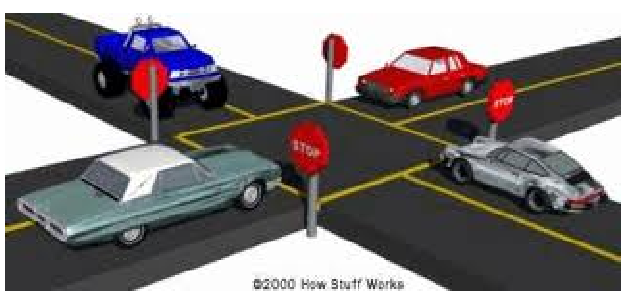
The Probability of a Collision

```
while true do
  NonCriticalSection
  while !free do #nothing
   end
  free = false
  CriticalSection
  free = true
end
```

Average time to perform Noncritical Section: 1,000 nanoseconds
Average time to perform CriticalSection: 10 nanoseconds
Average time to execute tests: 2 nanoseconds

Probability of one collision 1/1,000 = .001
Iterations of outer loop in one second: 10,000,000/1,012 = 9891
Probability of no collisions in 1 second: $(1-0.001)^{9891} = 0.00005$

The Stop and Look 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).

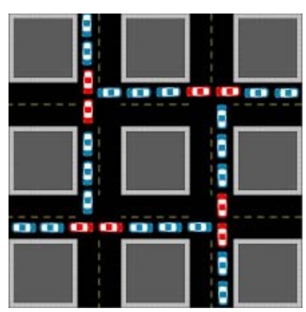
The Stop and Look Method

```
# Shared Memory
 free[0] = true   #P0 is not stopped at sign
 free[1] = true   #P1 is not stopped at sign

# Process 0
while true do
A  nonCriticalSection
B  free[0] = false
C  while !free[1] do
   end
D criticalSection
E free[0] = true
end

# Process 1
while true do
A  nonCriticalSection
B  free[1] = false
C  while !free[0] do
   end
D criticalSection
E free[1] = true
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.



A Stop Light Solution

```
owner = 1
```

```
# Process 1
   while true
A    nonCriticalSection1
B    while owner == 2 do
     end
C    criticalSection1
D    owner = 2
   end
```

```
# Process 2
   while true
A    nonCriticalSection2
B    while owner == 1 do
     end
C    criticalSection2
D    owner = 1
   end
```


Check a Multiprocess by Filling *State Table*

```
# Process 1
   while true
A    nonCriticalSection1
B    while owner == 2 do
     end
C    criticalSection1
D    owner = 2
   end
```

```
# Process 2
   while true
A    nonCriticalSection2
B    while owner == 1 do
     end
C    criticalSection2
D    owner = 1
   end
```

A state is described by the values of control variables (in this case just owner) and the line that is about to be executed. The three characters. “dXY” means “owner contains d, P1 is about to execute X, P2 is about to execute Y”.

Initial state: both processes are in their non-critical section and owner = 1.

State: owner P1at P2at	P1 steps	P2 steps
1AA		

A process exits non-critical section

```
# Process 1
    while true
A      nonCriticalSection1
B      while owner == 2 do
       end
C      criticalSection1
D      owner = 2
    end
```

```
# Process 2
    while true
A      nonCriticalSection2
B      while owner == 1 do
       end
C      criticalSection2
     owner = 1
    end
```

If it was P1, next state is

If it was P2, next state is

State: owner P1at P2at	P1 steps	P2 steps
1AA	1BA	1AB

Enter the two new states and explore one.

```
# Process 1
    while true
A      nonCriticalSection1
B      while owner == 2 do
       end
C      criticalSection1
D      owner = 2
    end
```

```
# Process 2
    while true
A      nonCriticalSection2
B      while owner == 1 do
       end
C      criticalSection2
D      owner = 1
    end
```

P1 enters critical section.

State: owner P1at P2at	P1 steps	P2 steps
1AA	1BA	1AB
1BA	1CA	1BB
1AB		

Enter the two new states and explore one.

```
# Process 1
   while true
A     nonCriticalSection1
B     while owner == 2 do
      end
C     criticalSection1
D     owner = 2
   end
```

```
# Process 2
   while true
A     nonCriticalSection2
B     while owner == 1 do
      end
C     criticalSection2
D     owner = 1
   end
```

P2 stalls.

State: owner P1at P2at	P1 steps	P2 steps
1AA	1BA	1AB
1BA	1CA	1BB
1AB	1BB	1AB
1CA		
1BB		

The Complete Stop Light State Table

owner = 1

owner-P1-P2	P1 steps	P2 steps
1AA	1BA	1AB
1BA	1CA	1BB
1AB	1BB	1AB
1CA	1DA	1CB
1BB	1CB	1BB
1DA	2AA	1DB
1CB	1DB	1BB
2AA	2BA	2AB
1DB	2AB	1DB
2BA	2BA	2BB
2AB	2BB	2AC
2BB	2BB	2BC
2AC	2BC	2AD
2BC	2BC	2BD
2AD	2BD	1AA
2BD	2BD	1BA

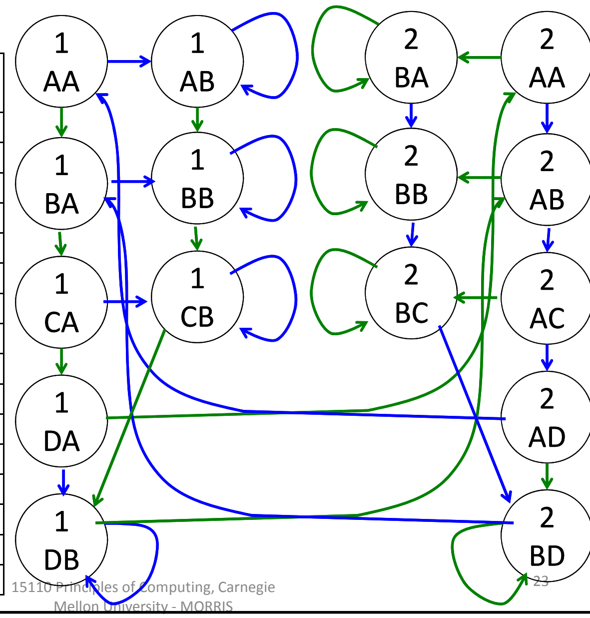
```
# Process 1
   while true
A     nonCriticalSection1
B     while owner == 2 do
      end
C     criticalSection1
D     owner = 2
   end
```

```
# Process 2
   while true
A     nonCriticalSection2
B     while owner == 1 do
      end
C     criticalSection2
D     owner = 1
   end
```

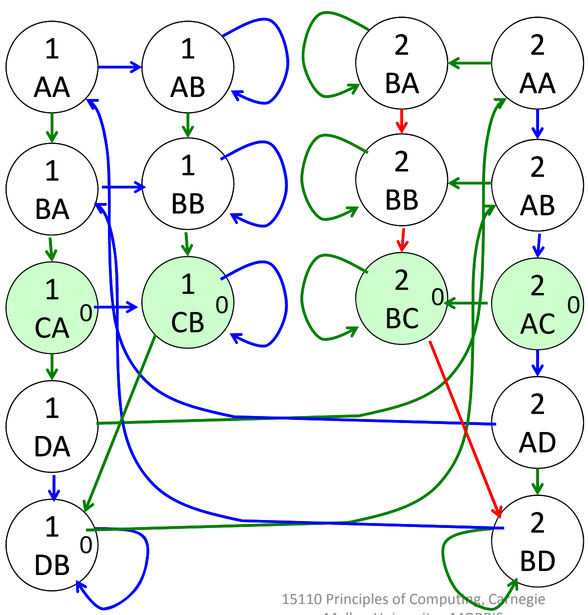
Here is the complete state table. You can tell there is no interference because there is no state with CC in it. To check deadlock, we better draw the picture.

The Complete Stop Light State Graph

owner-P1-P2	P1 steps	P2 steps
1AA	1BA	1AB
1BA	1CA	1BB
1AB	1BB	1AB
1CA	1DA	1CB
1BB	1CB	1BB
1DA	2AA	1DB
1CB	1DB	1BB
2AA	2BA	2AB
1DB	2AB	1DB
2BA	2BA	2BB
2AB	2BB	2AC
2BB	2BB	2BC
2AC	2BC	2AD
2BC	2BC	2BD
2AD	2BD	1AA
2BD	2BD	1BA

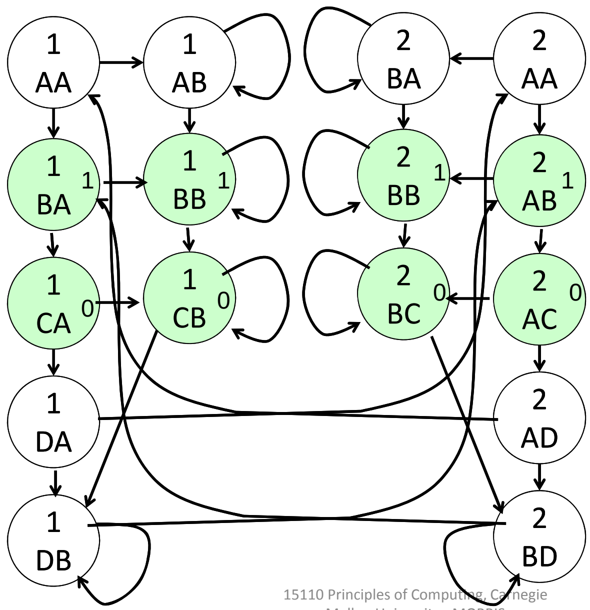


Looking for Deadlock



Color the states that include the critical section, C, and mark them with a 0.

Looking for Deadlock

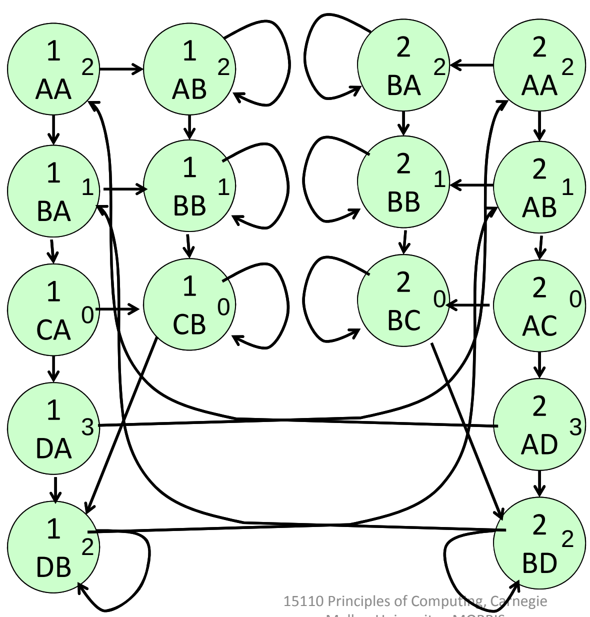


for n=0,1,2,3,…
Mark with n+1 all the nodes that have a transition to one marked with n.

15110 Principles of Computing, Carnegie Mellon University - MORRIS

25

Looking for Deadlock



for n=0,2,3,…
Mark with n+1 all the nodes that have a transition to one marked with n.

There is no deadlock, but….

15110 Principles of Computing, Carnegie Mellon University - MORRIS

26

There can be *Starvation*.

```
owner = 1
```

```
# Process 1                          # Process 2
   while true                           while true
A    nonCriticalSection1            A    nonCriticalSection2
B    while owner == 2 do            B    while owner == 1 do
     end                                 end
C    criticalSection1               C    criticalSection2
D    owner = 2                      D    owner = 1
   end                                  end
```

Starvation when one process can stay in its non-critical section forever, preventing the other one from getting into the critical section.

If Process 1 stays at A forever, Process 2 can't get into its critical section, even if it wants to.

An Asymmetric Solution

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

```
# Process 1
   while true
      nonCriticalSection1
A     free[1] = false
B     while !free[2] do
      end
      criticalSection1
C     free[1] = true
   end
```

```
#  Polite-Process 2
 while true do
  nonCriticalSection2
A free[2] = false
B while !free[1] do
C   free[2] = true
D   while !free[1] do
    end
E   free[2] = false
  end
  criticalSection2
F free[2] = true
end
```

Process 2 backs off when it detects a conflict. This one has no major flaws, but it takes a huge state table to show it!

State Table for Asymmetric Solution

P1-P2-free[1]-free[2]	P1	P2
AAtt	BAft	ABtf
BAft	CAft	BBff
ABtf	BBff	Attf
CAft	AAtt	CBff
BBff	BBff	BCff
AFtf	BFff	AAtt
CBff	ABtf	CCff
BCff	BCff	BDft
Btff	Btff	BAft
CCff	ACtf	CDft
BDft	CDft	BDft
ACtf	BCff	ADtt
CDft	ADtt	CDft
ADtt	BDft	AEtt
AEtt	BEft	ABtf
BEft	CEft	BBff
CEft	AEtt	CBff

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

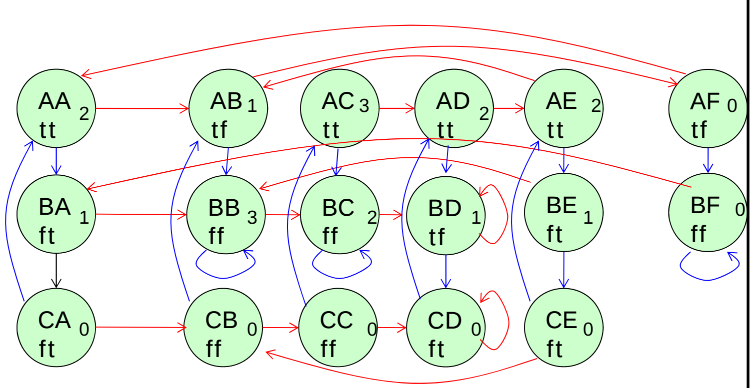
```
# Process 1
  while true
     nonCriticalSection1
A    free[1] = false
B    while !free[2] do
     end
     criticalSection1
C    free[1] = true
  end
```

```
#  Polite-Process 2
 while true do
  nonCriticalSection2
A free[2] = false
B while !free[1] do
C   free[2] = true
D   while !free[1] do
    end
E   free[2] = false
  end
  criticalSection2
F free[2] = true
end
```

There is no interference because a state starting with CF doesn't occur.

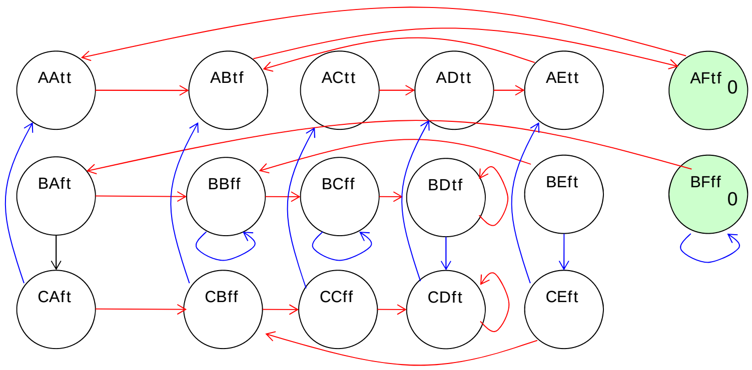
Graph for Asymmetric Solution

P1-P2-free[1]-free[2]	P1	P2
AAtt	BAft	ABtf
BAft	CAft	BBff
ABtf	BBff	Attf
CAft	AAtt	CBff
BBff	BBff	BCff
AFtf	BFff	AAtt
CBff	ABtf	CCff
BCff	BCff	BDft
Btff	Btff	BAft
CCff	ACtf	CDft
BDft	CDft	BDft
ACtf	BCff	ADtt
CDft	ADtt	CDft
ADtt	BDft	AEtt
AEtt	BEft	ABtf
BEft	CEft	BBff
CEft	AEtt	CBff



No Deadlock

Checking for Starvation of P2

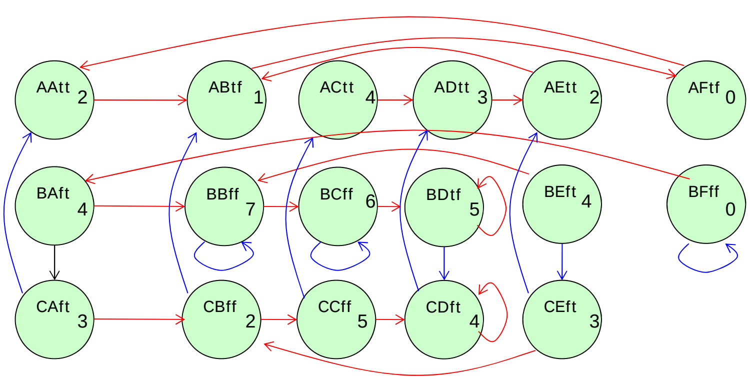


Erase blue exits from Axxx states and mark Process 2's critical sections.

15110 Principles of Computing, Carnegie Mellon University - MORRIS

31

Checking for Starvation of P2



Number all the other states by distance from critical section. Process 2 can't be starved.

15110 Principles of Computing, Carnegie Mellon University - MORRIS

32

How to select states to consider

- If you think you see a potential problem, choose states that lead to it.
- Otherwise, just do them all and look for problems.
- You don't have to label lines that don't touch shared memory.

Peterson's algorithm is symmetric and works!

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

```
# Process 0
  while true do
    nonCritcalSection0
    free[0] = false
    priority = 1
    while !free[1] and
          priority==1 do
    end
    criticalSection0
    free[0] = true
 end
```

```
#  Process 1
  while true do
    nonCritcalSection1
    free[1] = false
    priority = 0
    while !free[0] and
          priority==0 do
    end
    criticalSection1
    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
```

Probability collision will occur on Nth iteration = $1/2^N$

Multiprocessing is *very* hard.

- Conventional debugging doesn't work.
  - You can't step the program to investigate where something goes wrong.
  - Testing is futile. If there are N labeled lines, there are 2^N different execution sequences to test.
- It requires more art and mathematics.
  - It's like digital hardware design.
  - It needs proofs.
- The state table method becomes unwieldy.
  - The potential number of states is the product of the numbers of values of all the control variable and the numbers of labeled lines in all the processes.
  - Computer Scientists invent programs to test large tables.
- Only a tiny percentage of practicing programmers can do it.

When is a 1% chance of error in a day 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 when the system is in operation and the programmer has moved on.
- If there is a 0.01% chance of error, the bug will show up after a human generation has seen no error and depends upon the code to run a vital service.

This man removed all the traffic lights and signs!

Homework

1. An Asymmetric Solution

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

```
# Process 1
    while true
A       nonCriticalSection1
B       free[1] = false
C       while !free[2] do
        end
D       criticalSection1
E       free[1] = true
    end
```

```
# Process 2
    while true
A       nonCriticalSection2
B       while !free[1] do
        end
C       free[2] = false
D       criticalSection2
E       free[2] = true
    end
```

It is OK to have the two processes run different programs. Here we switch statements B and C in Process 2 to bias things in favor of Process 1 and break the ties that seem to cause problems. Use the table below to analyze the possible sequences and discover a problem: interference, deadlock, or starvation. You only need to show enough states to demonstrate a problem.

State Table

free[1]-free[2]-P1-P2	P1 moves	P2 moves
TTAA	TTBA	TTAB
TTAB		
TTBA		

2. A national economy could be looked at a system with 100,000 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.