COMPS267F Chapter 5

Process Synchronization

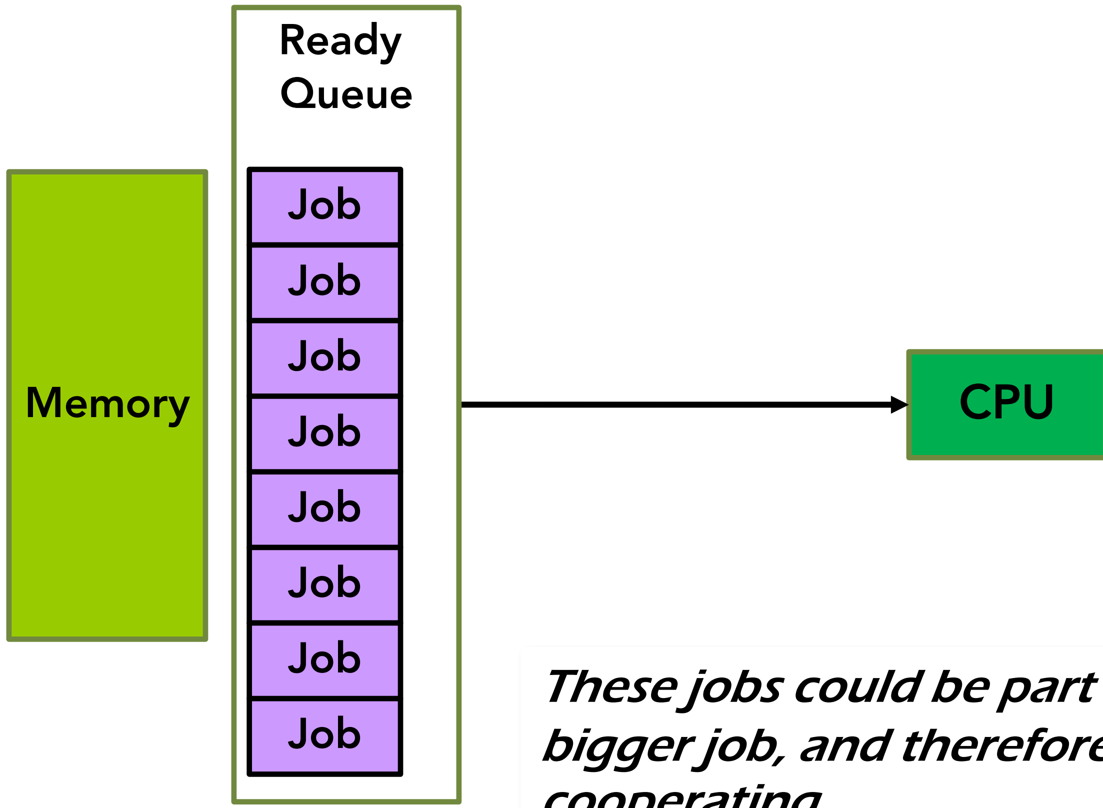
Dr. Andrew Kwok-Fai LUI

Aim of the Chapter

- We have multi-programming computer systems
  - Many programs are running at the same time
  - They use and may share data
  - Data integrity problems may emerge
    - Two processes/threads are reading/writing same data container at the same time
- Concurrent access can cause data integrity problems
- Process synchronization is the study of how to manage process activities to keep data integrity

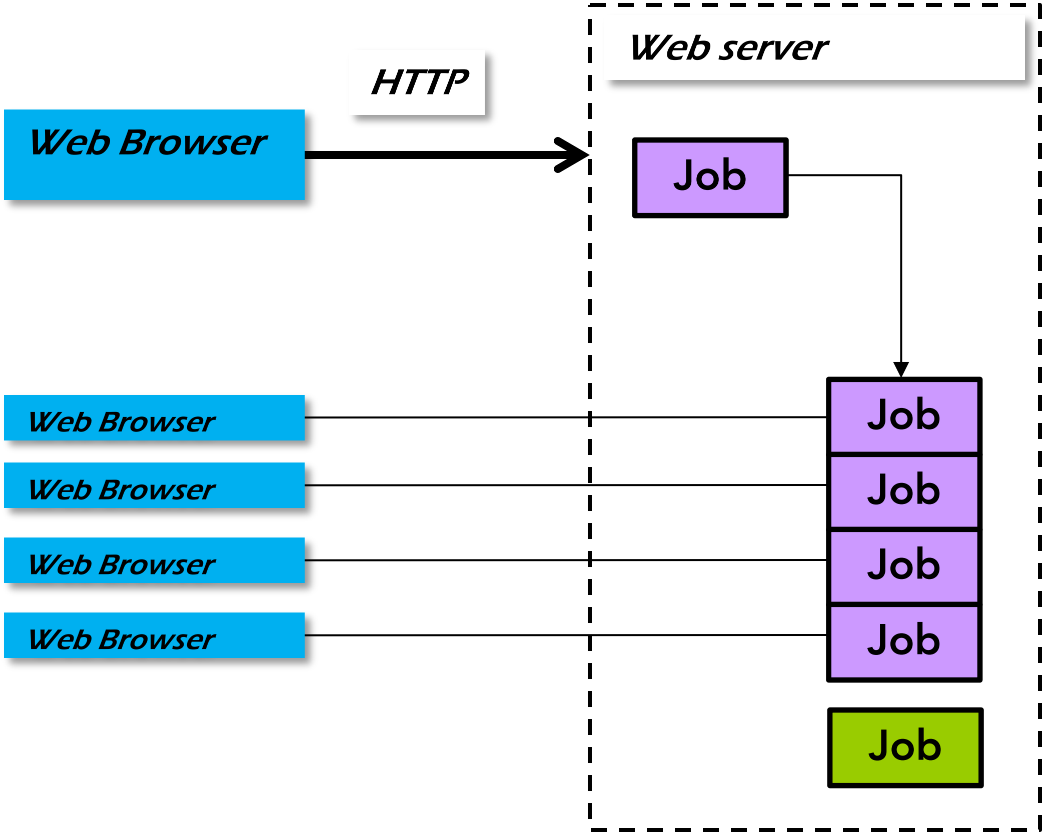
concurrent access

Multi-Programming



These jobs could be part of a bigger job, and therefore they are cooperating

Multi-Programming with Cooperation



A new job or process is created for each new HTTP connection

Example: Multi-programming Web Server

Server Information					
Tomcat Version	**JVM Version**	**JVM Vendor**	**OS Name**	**OS Version**	**(**
Apache Tomcat/5.5.23	1.5.0_12-b04	Sun Microsystems Inc.	Linux	2.6.18-128.4.1.el5	

JVM

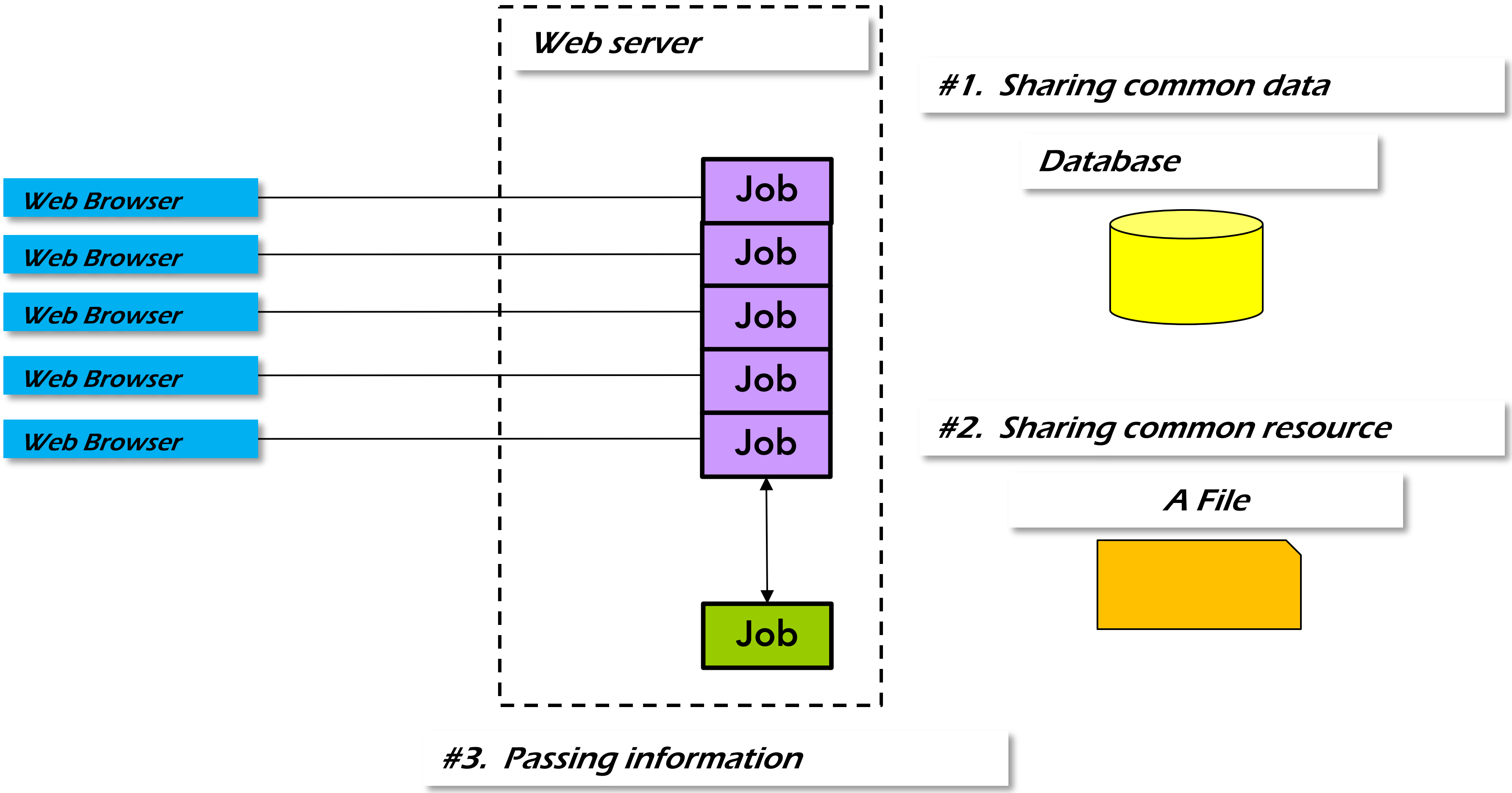
Free memory: 38.21 MB Total memory: 75.37 MB Max memory: 489.18 MB

http-80

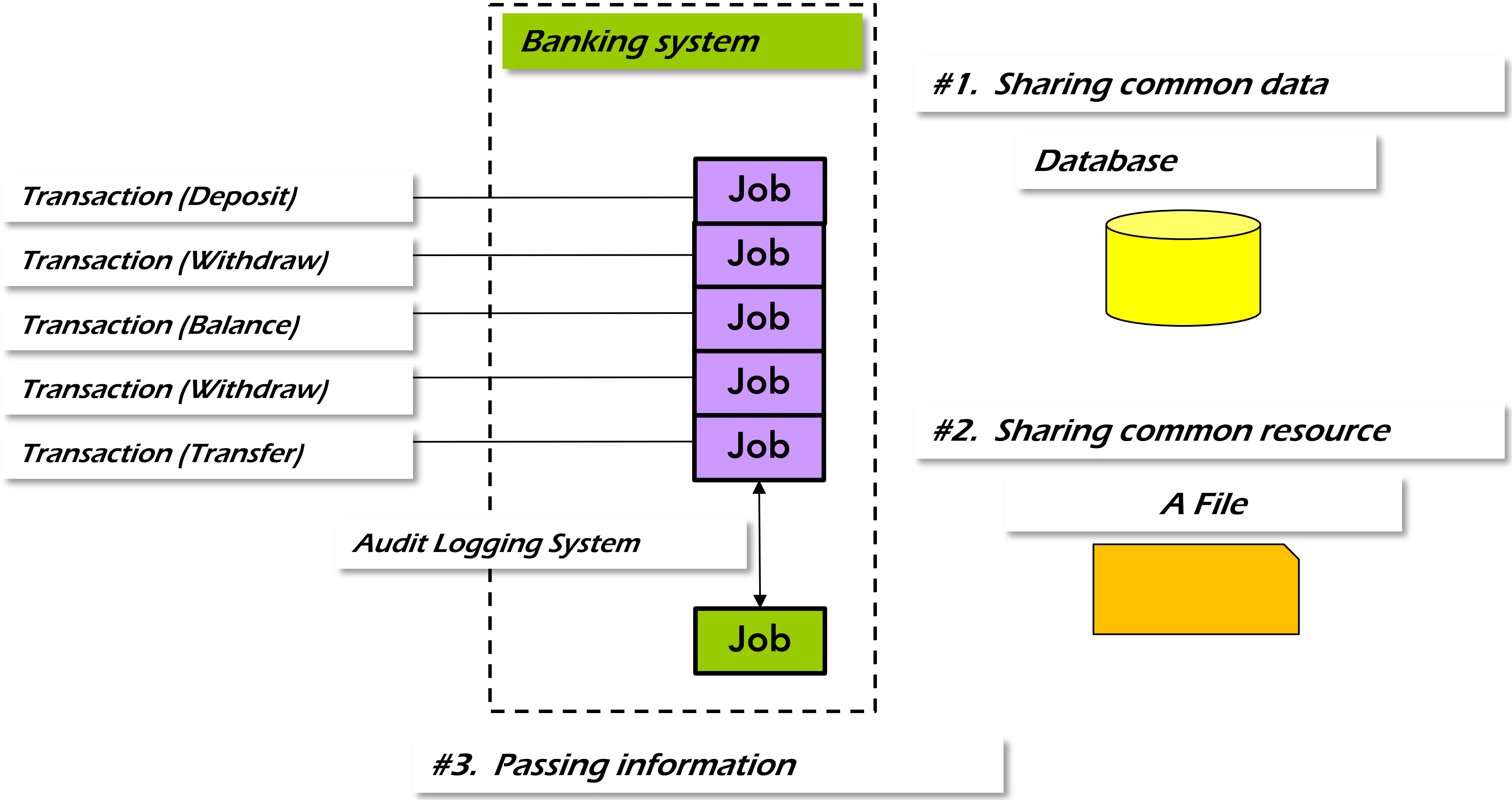
Max threads: 150 Min spare threads: 25 Max spare threads: 75 Current thread count: 76 Current thread busy: 4
Max processing time: 43763 ms Processing time: 3995.215 s Request count: 160944 Error count: 726 Bytes received: 6.80 MB Bytes sent: 248.12 MB

Stage	Time	B Sent	B Recv	Client	VHost	Request
R	?	?	?	?	?	?
R	?	?	?	?	?	?

Concurrent Access in Multi-Programming



Concurrent Access in Multi-Programming

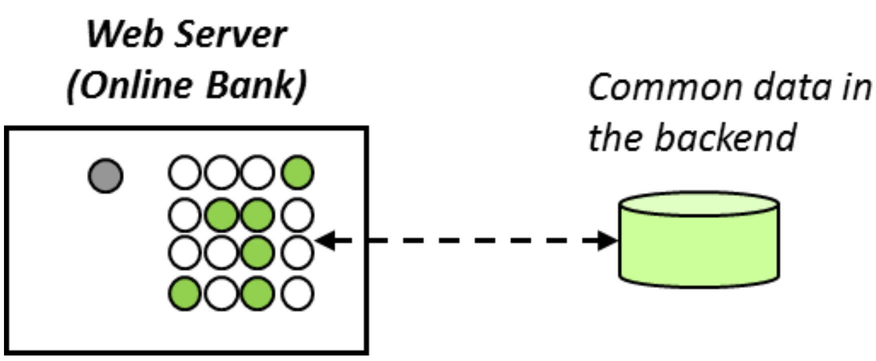


Concurrent Access

- Concurrent access is necessary.
  - Reasons for relation or cooperation
    - Information Sharing: access to the same database
    - Computation Speedup: parallel processing of data
    - Modularity: result of software engineering
  - Two or more programs can cooperate, and the following happen
    - Accessing the same data (one is reading while another is writing)
    - Passing information
    - Using the same resource type

Concurrent Access in Thread Pools

- Web servers are often running on thread pools
- There should be one single data storage
  - Physical or logical
  - Multiple threads have concurrent access to the same data



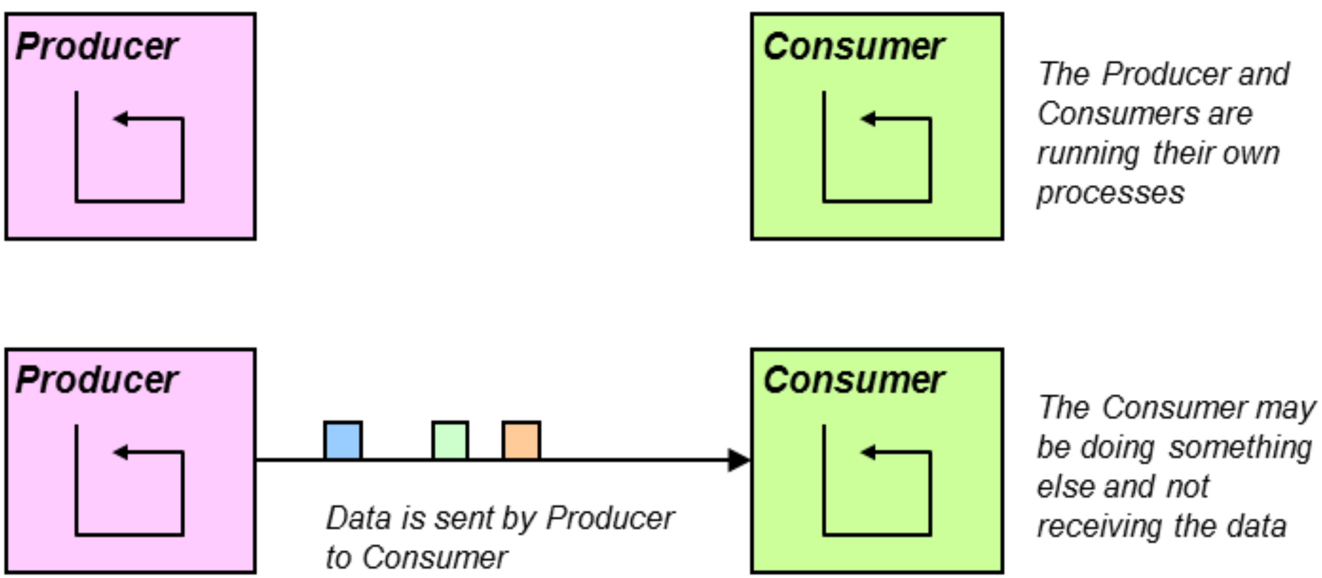
Demo

- Download Netbeans project
  - RaceCondition.zip
  - Run the class RaceCondition under the default package

race conditions

A Data Integrity Problem

- A case study of concurrent access
  - One process/thread writing data to a buffer
  - Another process/thread reading data from the buffer

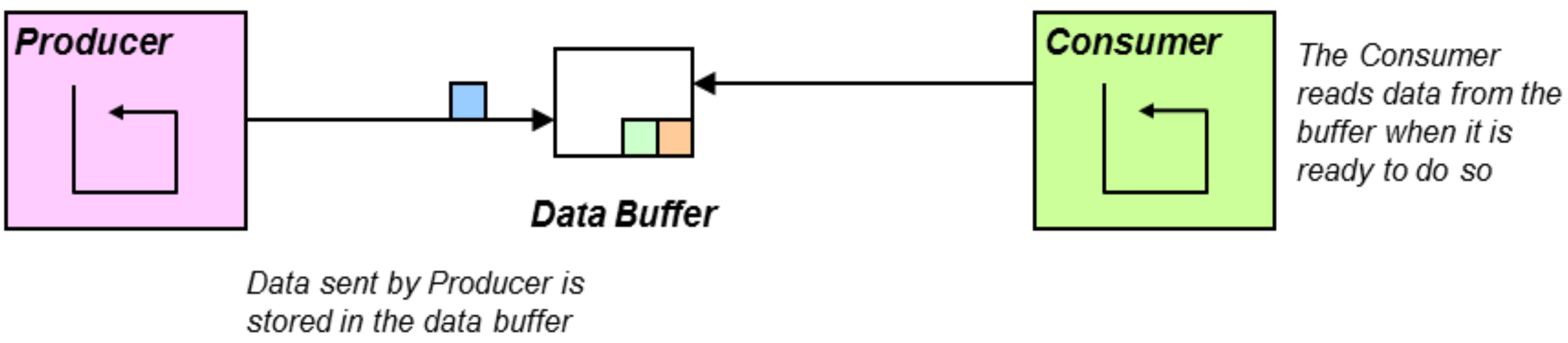


Data Integrity Problem

- The two processes/threads are Producer and Consumer
  - The producer generates data
  - The producer sends the generated data to the consumer
  - The consumer receives the data and performs further processing on the data

Data Integrity Problem

- A buffer is essential
  - Data lost when the consumer is busy
  - A buffer collects data sent out by the Producer
  - The buffer waits for the Consumer to collect data
  - The buffer is bounded



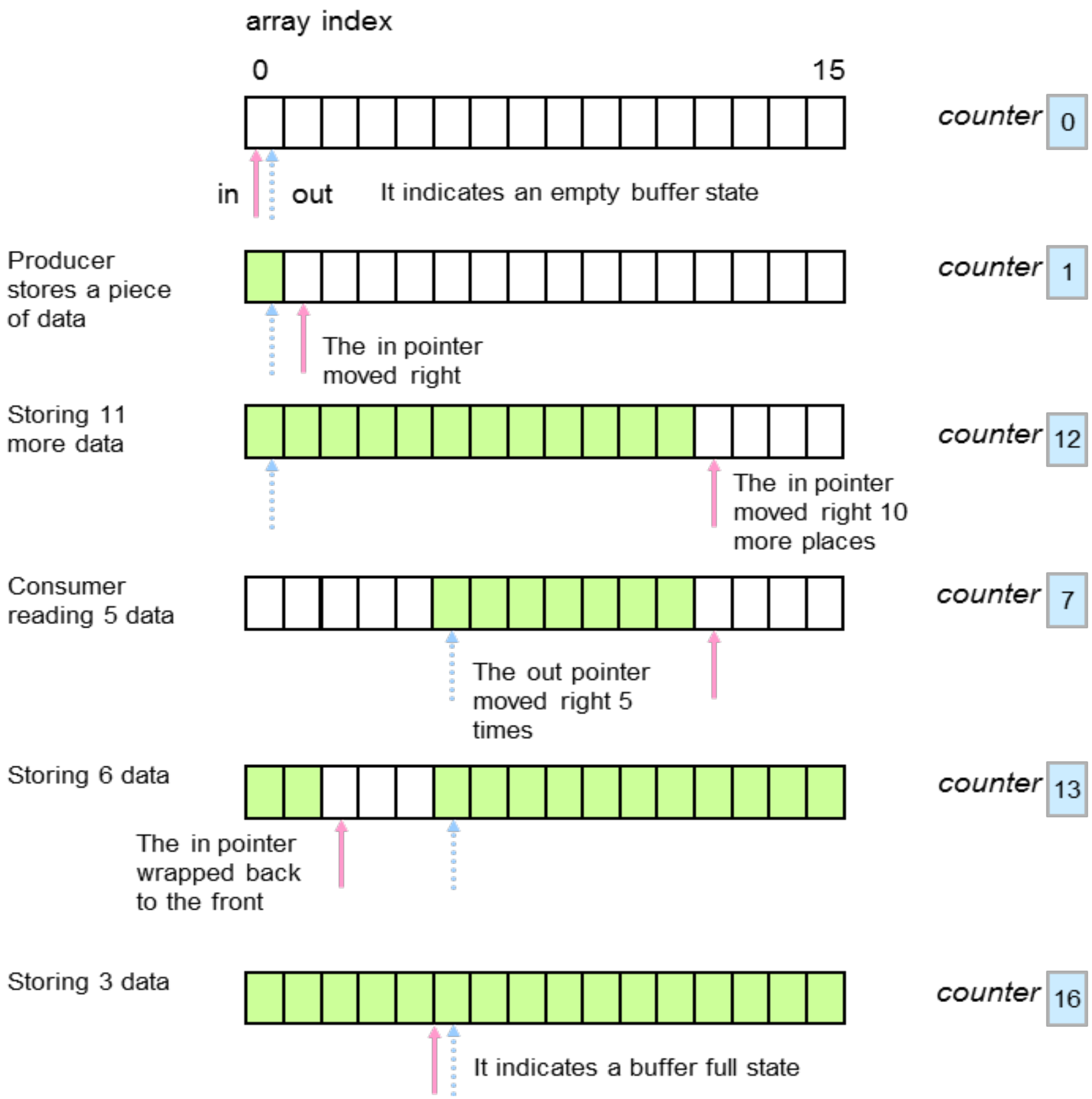
Bounded Buffer Implementation

- A bounded buffer implementation
  - Array with size of 16
  - Circular wrap around manner
  - Pointers pointing to the used portion of the array
  - The in pointer is where the new data is placed (by producer)
  - The out pointer is where the data is removed (by consumer)
  - The counter variable keeps track of number of data stored

```
static class Buffer {
    static final int BUFFERSIZE = 16;
    int data[] = new int[BUFFERSIZE];
    int in = 0;
    int out = 0;
    int counter = 0; /* how many data is stored in the array */
}
```

Bounded Buffer Implementation

Operations of Bounded Buffer Implementation



The in and out pointers define used portion of the array

The counter variable keeps track of how many data is stored

Dr. Andrew Kwok-Fai Lui

Bounded Buffer Implementation

```
static class Producer implements Runnable {
...
    public void run() {
      int nextProduced = 0;
      while (true) {
        if (toStop) {
          break;
        }

      nextProduced++;

      try {
          while (buffer.counter == Buffer.BUFFERSIZE) {
            Thread.currentThread().sleep(resttime);
          }
          buffer.data[buffer.in] = nextProduced;
          buffer.in = (buffer.in + 1) % Buffer.BUFFERSIZE;
          buffer.counter = buffer.counter + 1;
          Thread.currentThread().sleep(rate);
        } catch (InterruptedException ex) {
        }
     }
   }
...
```

Bounded Buffer Implementation

```
 static class Consumer implements Runnable {
...
    public void run() {
      int nextConsumed = 0;
      while (true) {
        try {
          while (buffer.counter == 0) {
            Thread.currentThread().sleep(resttime);
          }
          lastConsumed = nextConsumed;
          nextConsumed = buffer.data[buffer.out];
          buffer.out = (buffer.out + 1) % Buffer.BUFFERSIZE;
          buffer.counter = buffer.counter - 1;
        } catch (InterruptedException ex) {
        }
}
...
```

Race Condition

- Concurrent access situation
  - Two update operations
    - The Producer adds one to the counter variable
    - The Consumer subtract one from the counter variable

Producer: `buffer.counter = buffer.counter + 1;`

Consumer: `buffer.counter = buffer.counter - 1;`

  - The variable counter can become incorrect
    - The outcome will not remain the same as some execution sequence of Producer and Consumer will cause error
    - The counter may become incorrect at some point in the future!

Race Condition

- The problem statements in machine code

Producer	***Remarks***
1. LDA RA, counter	Loading counter from memory into CPU register RA
2. ADD RA, 1	RA = RA + 1
3. STO counter, RA	Storing the value in RA to the counter in memory

Consumer	***Remarks***
1. LDA RB, counter	Loading counter from memory into CPU register RB
2. SUB RB, 1	RA = RA - 1
3. STO counter, RB	Storing the value in RB to the counter in memory

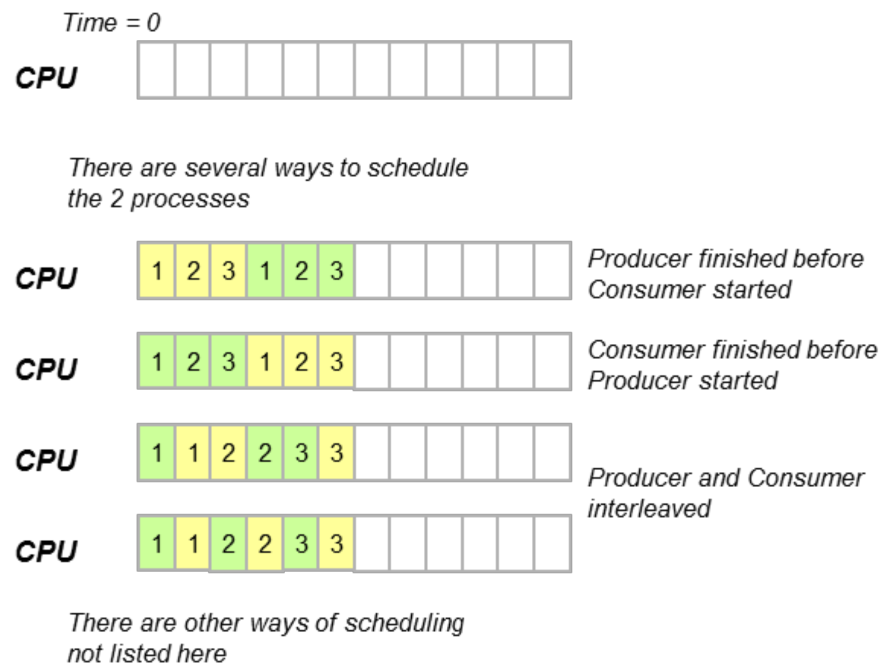
Race Condition

- Summary of the situation
  - There is one variable counter
  - There are two threads running concurrently
  - There is one processor (actually irrelevant)
  - The Process Scheduler decides which and when the Producer and Consumer control the CPU
  - The order of execution is unpredictable

Race Condition

- An example of Producer and Consumer running their code once
  - There are 6 instructions
  - Different ways to schedule the two processes/threads to run the 6 instructions
  - 20 valid permutations of the order of execution

Process	*Instructions*
Producer	1. LDA RA, counter 2. ADD RA, 1 3. STO counter, RA
Consumer	1. LDA RB, counter 2. SUB RB, 1 3. STO counter, RB



Race Condition

- The order of execution of the instructions can change the outcome of execution
  - Most patterns of execution orders produce correct outcome
  - At least one pattern produces a wrong outcome

Race Condition

Process	Instructions
Producer	1. LDA RA, counter 2. ADD RA, 1 3. STO counter, RA
Consumer	1. LDA RB, counter 2. SUB RB, 1 3. STO counter, RB

The CPU has lots of general purpose registers. Producer will use Register RA to perform the calculation, and Consumer will use Register RB.

CPU Scheduling Scenario

1	2	3	1	2	3						

	CPU RA	***CPU*** RB	***counter***
	?	?	5
1. LDA RA, counter	5	?	5
2. ADD RA, 1	6	?	5
3. STO counter, RA	6	?	6
1. LDA RB, counter	6	6	6
2. SUB RB, 1	6	5	6
3. STO counter, RB	6	5	5 *Correct Result*

Correct Outcome

Order of Execution of Instructions

Process	*Instructions*
Producer	1. LDA RA, counter 2. ADD RA, 1 3. STO counter, RA
Consumer	1. LDA RB, counter 2. SUB RB, 1 3. STO counter, RB

CPU Scheduling Scenario

1	1	2	2	3	3						

Pre-emption happens after execution of just one instruction

Instruction	***CPU*** RA	***CPU*** RB	***counter***	
	?	?	5	
1. LDA RA, counter	5	?	5	
1. LDA RB, counter	5	5	5	
2. ADD RA, 1	6	5	5	
2. SUB RB, 1	6	4	5	
3. STO counter, RA	6	4	6	
3. STO counter, RB	6	4	4	*Error*

Wrong Outcome

Race Condition

- Only 2 out of 20 permutations produce correct outcome
- The outcome of processes depending on the execution order of the instructions is known as <u>race condition</u>
  - Multiple processes are reading/writing to a shared data
  - The processes' execution order depends on the CPU scheduler/timing
  - The outcome depends on the order of execution

Race Condition

Process	Instructions
Producer	1. LDA RA, counter 2. ADD RA, 1 3. STO counter, RA
Consumer	1. LDA RB, counter 2. SUB RB, 1 3. STO counter, RB

The CPU has lots of general purpose registers. Producer will use Register RA to perform the calculation, and Consumer will use Register RB.

CPU Scheduling Scenario

1	2	3	1	2	3						

Instruction	CPU: RA	CPU: RB	counter
	?	?	5
1. LDA RA, counter	5	?	5
2. ADD RA, 1	6	?	5
3. STO counter, RA	6	?	6
1. LDA RB, counter	6	6	6
2. SUB RB, 1	6	5	6
3. STO counter, RB	6	5	5 *Correct Result*

Correct Outcome

Order of Execution of Instructions

Process	*Instructions*
Producer	1. LDA RA, counter 2. ADD RA, 1 3. STO counter, RA
Consumer	1. LDA RB, counter 2. SUB RB, 1 3. STO counter, RB

CPU Scheduling Scenario

1	1	2	2	3	3						

Pre-emption happens after execution of just one instruction

	CPU ***RA***	***CPU*** ***RB***	***counter***	
	?	?	5	
1. LDA RA, counter	5	?	5	
1. LDA RB, counter	5	5	5	
2. ADD RA, 1	6	5	5	
2. SUB RB, 1	6	4	5	
3. STO counter, RA	6	4	6	
3. STO counter, RB	6	4	4	*Error*

Wrong Outcome

solution for race condition and critical section

Solution for the Race Condition

- Allow only the scheduling patterns leading to correct outcomes
  - Which patterns will produce correct outcomes?

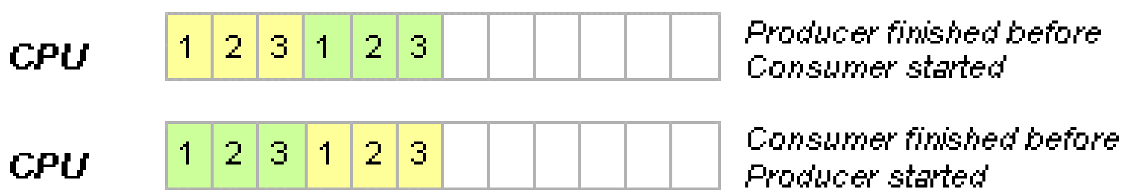
Solution for the Race Condition

Example 1: Scheduling Patterns that lead to Correct Calculation

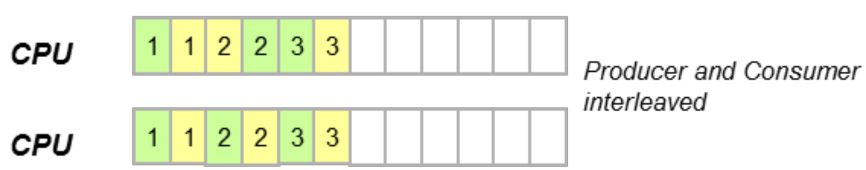
The first example of scheduling pattern allowed correct calculation. Find out the only other scheduling pattern leading to correct calculation.

Answer:

The first example is non-interleaved. The Producer executes all 3 instructions together and then the Consumer executes all 3 instructions. The other pattern is also non-interleaved. The Consumer executes all 3 instructions together and then the Producer executes all 3 instructions.



This leads to the observation that interleaving the instructions in a calculation procedure can cause errors. Allowing interleaved scheduling means that two processes are interfering each other with the reading/writing of the variable counter.



Solution for the Race Condition

- Some instructions should not be executed in an interleaved manner
  - The outcome is error free if each of the following code is allowed to complete execution without interleave or pre-emption

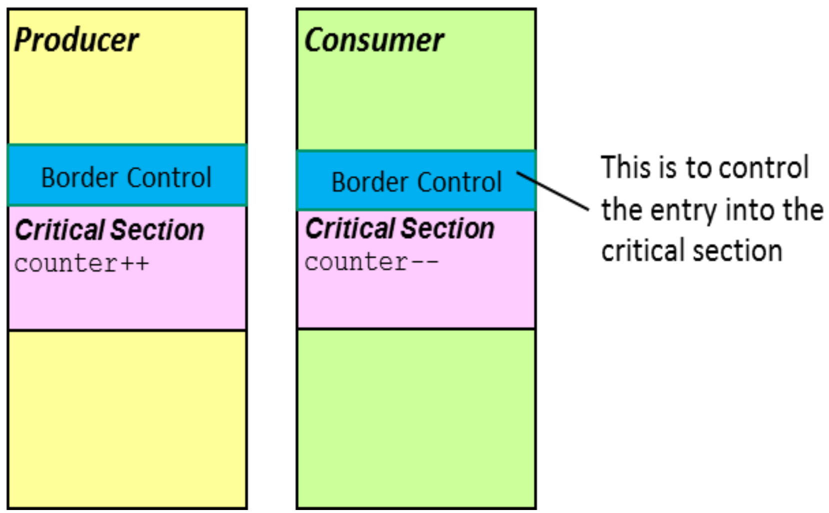
```
Producer: buffer.counter = buffer.counter + 1;
Consumer: buffer.counter = buffer.counter - 1;
```

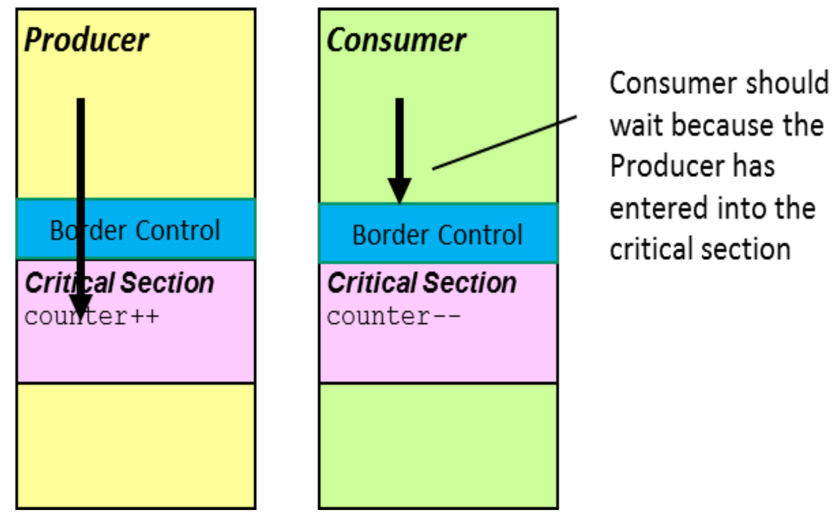
- We must define which instructions are to be controlled
  - Define <u>Critical Section (CS)</u> to include such instructions
  - No pre-emption is allowed in the middle of execution of the critical section

Critical Section

- A section of code
- Processes are updating a shared data or manipulating a common resource
  - A place where race condition can occur
- A solution is needed for synchronizing processes' cooperation
  - A mechanism for control processes to enter into a critical section
    - Like an immigration border control for processes/threads
    - The border control is placed just before every critical section
  - One process is allowed inside the critical section at a time
    - Other processes have to wait at the border control

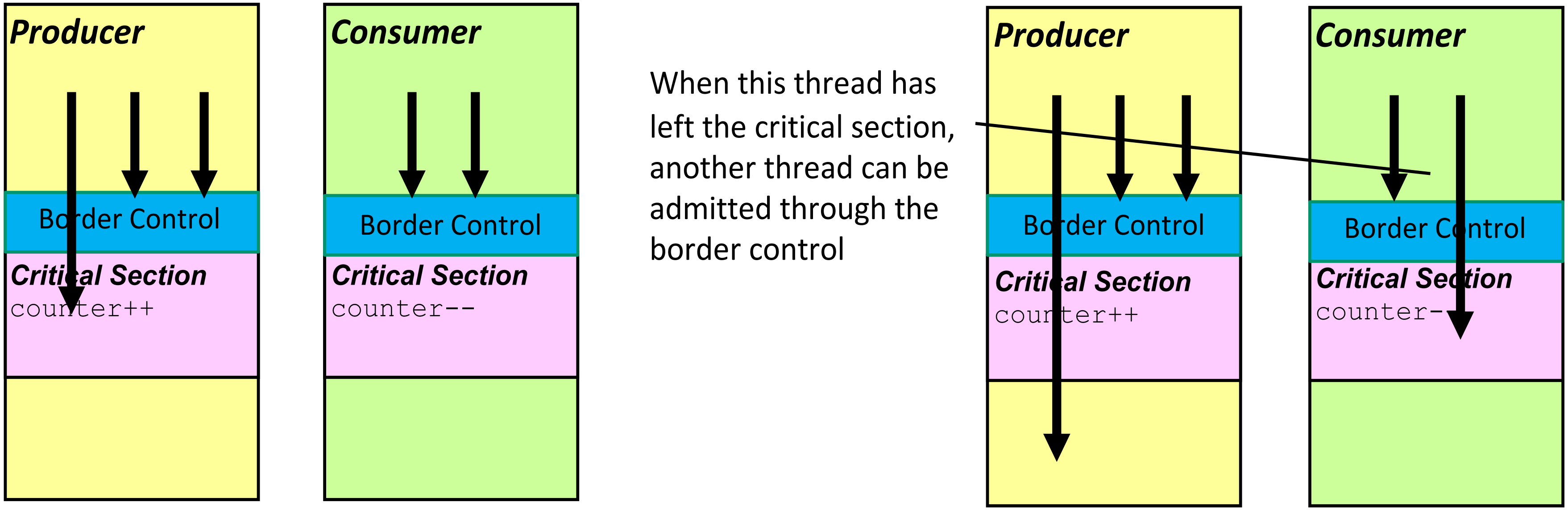
Critical Section



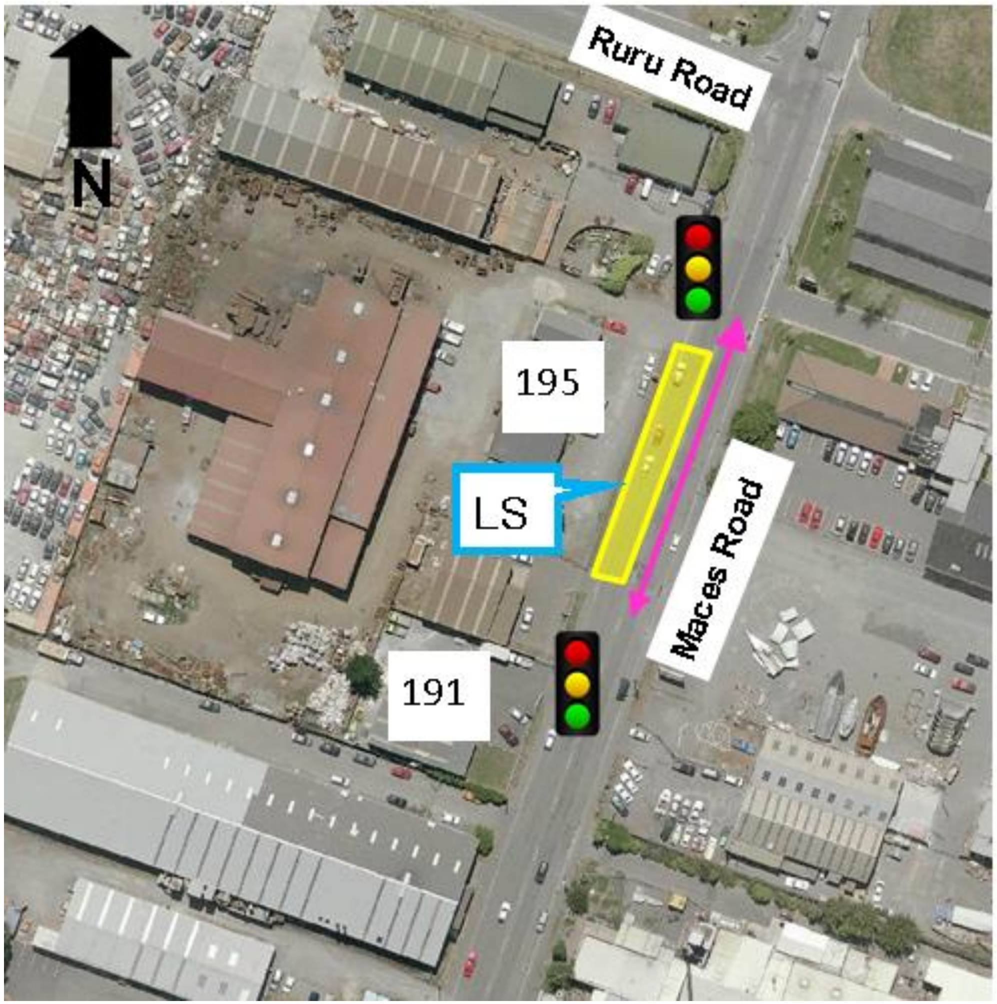


Critical Section

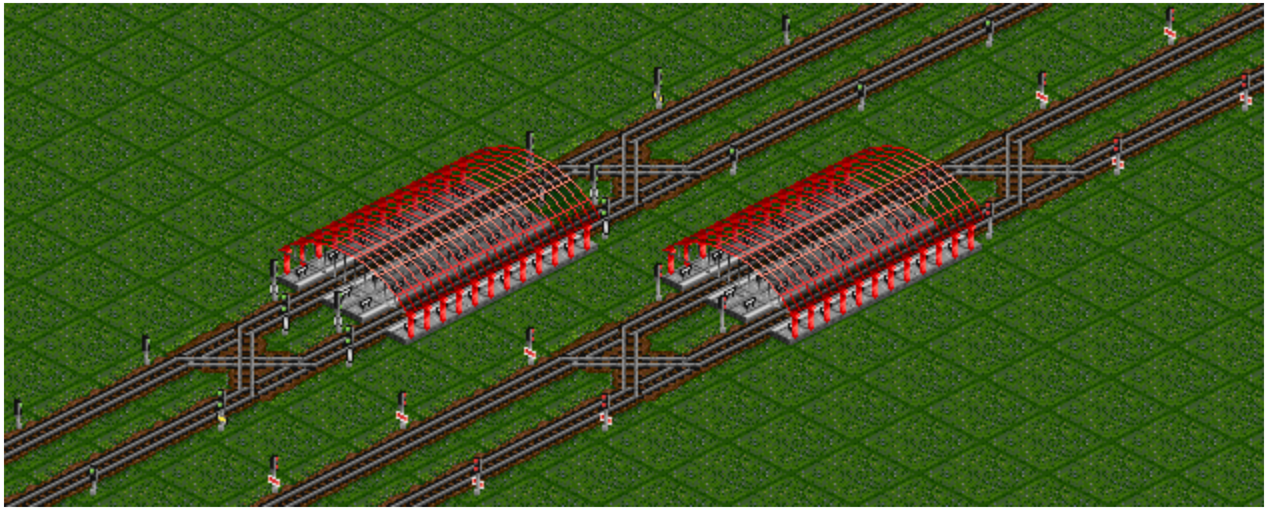
- In case of multiple threads running Producers and Consumers



Closed Road



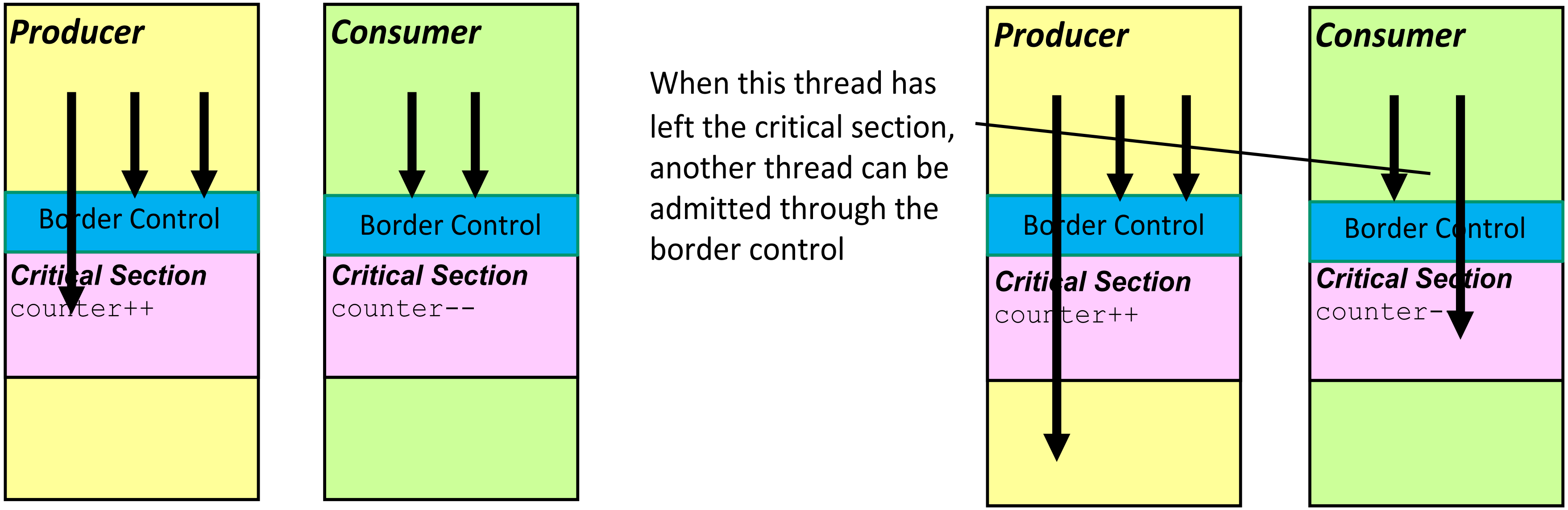
Rail Merge



implementations of critical section

Critical Section

- In case of multiple threads running Producers and Consumers

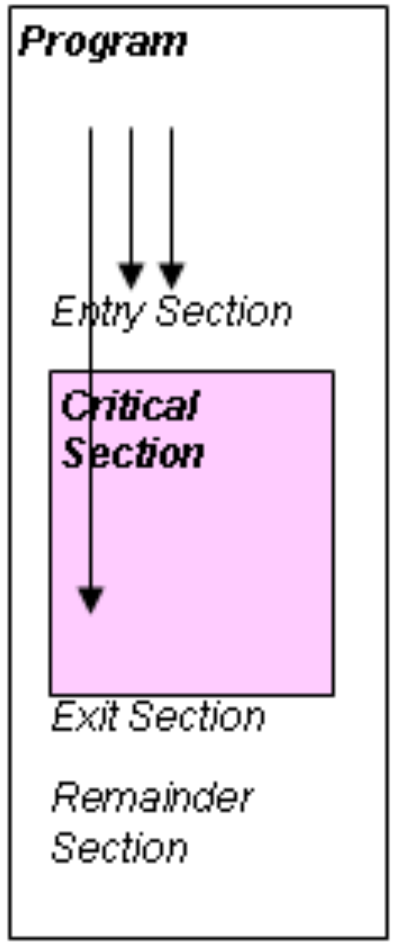


Dr. Andrew Kwok-Fai Lui

Solution to Critical Section Problem

- Programming construct to control the entry to CS
  - A loop to trap the process execution
    - Until the process is allowed to progress into the critical section.
    - Like a lock

Critical Section



Mutual Exclusion
One process in the critical section at a time

Progress
One of the processes waiting to enter into critical section would eventually be selected to progress

Bounded Waiting
The waiting time of a process wanting to enter into the critical section is bounded

Critical Section

- Structure of a critical section
  - Entry section
    - Where a process is making a request to enter in the critical section
    - Waiting for entry would stay in this section
  - Exit section
    - Where a process has indicated it has left the critical section
    - This section is often followed by a remainder section
  - Critical section
    - Contains concurrent access program code prone to race conditions
  - Remainder section
    - Normal code irrelevant to the problem

Requirements of Critical Section Solution

- A solution to the critical section problem must satisfy the following requirements
  - Mutual exclusion (One Process)
  - Progress (At Least One Process is Working)
  - Bounded waiting (Waiting time for any Process is Bounded)
- Mutual exclusion is not sufficient
  - The waiting process cannot wait forever

Solution #1

```
/* PROCESS 1 */

while (true) {
...
  while (turn != 1)
    ; /* busy wait */

  **Critical Section;**

  turn = 2;
  Remainder Section;
}
```

```
/* PROCESS 2 */

while (true) {
...
  while (turn != 2)
    ; /* busy wait */

  **Critical Section;**

  turn = 1;
  Remainder Section;
}
```

Solution #1

process 1

```
while (true) {

  while (turn != 1)
; /* busy wait */

  Critical Section;

  turn = 2;

  Remainder Section;

}
```

i

turn

process 2

```
while (true) {

  while (turn != 2)
; /* busy wait */

  Critical Section;

  turn = 1;

  Remainder Section;

}
```

Solution #1

process 1

```
while (true) {

  while (turn != 1)
; /* busy wait */

  Critical Section;

  turn = 2;

  Remainder Section;

}
```

1

turn

process 2

```
while (true) {

  while (turn != 2)
; /* busy wait */

  Critical Section;

  turn = 1;

  Remainder Section;

}
```

Solution #1

```
while (true) {

  while (turn != 1)
; /* busy wait */

  Critical Section;

  turn = 2;

  Remainder Section;

}
```

1

turn

process 2

```
while (true) {

  while (turn != 2)
; /* busy wait */

  Critical Section;

  turn = 1;

  Remainder Section;

}
```

Verdict to Solution #1

- Solution fails the requirement of Progress
  - If turn is initialized to 1 and Process 2 arrives at the entry section, then Process 2 will wait in the loop
  - The wait is ended when Process 1 arrived at the scene, entered and left the critical section, and sets the turn to 2
  - The problem is Process 2 will wait forever if Process 1 never comes
- The process cannot take initiative to enter the CS

Solution #2

```
/* PROCESS 1 */

while (true) {
  flag[1] = true; /* indicating
the desire to go into cs */

  while (flag[2] == true)
    ; /* busy wait */

 **Critical Section;**

  flag[1] = false;
  Remainder Section;
}
```

```
/* PROCESS 2 */

while (true) {
  flag[2] = true; /* indicating
the desire to go into cs */

  while (flag[1] == true)
    ; /* busy wait */

  **Critical Section;**

  flag[2] = false;
  Remainder Section;
}
```

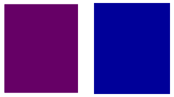
Solution #2

- The two variables flag indicate if a process is in the entry section and wish to enter into the critical section

Dr. Andrew Kwok-Fai Lui

Solution #2

```
while (true) {

  flag[1] = true;
  while (flag[2] == true)
; /* busy wait */


  Critical Section;


  flag[1] = false;


  Remainder Section;

}
```



flag

```
while (true) {

  flag[2] = true;
  while (flag[1] == true)
; /* busy wait */


  Critical Section;


  flag[2] = false;


  Remainder Section;

}
```

Solution #2

process 1

```
while (true) {

  flag[1] = true;

  while (flag[2] == true)
; /* busy wait */

  Critical Section;

  flag[1] = false;

  Remainder Section;
}
```

flag

process 2

```
while (true) {

  flag[2] = true;

  while (flag[1] == true)
; /* busy wait */

  Critical Section;

  flag[2] = false;

  Remainder Section;

}
```

Solution #2

process 1

```
while (true) {

  flag[1] = true;


  while (flag[2] == true)
; /* busy wait */


  Critical Section;

  flag[1] = false;

  Remainder Section;
}
```

flag

process 2

```
while (true) {

  flag[2] = true;


  while (flag[1] == true)
; /* busy wait */


  Critical Section;

  flag[2] = false;

  Remainder Section;
}
```

Solution #2

process 1

```
while (true) {

  flag[1] = true;


  while (flag[2] == true)
; /* busy wait */


  Critical Section;

  flag[1] = false;

  Remainder Section;
}
```

flag

```
while (true) {

  flag[2] = true;


  while (flag[1] == true)
; /* busy wait */


  Critical Section;

  flag[2] = false;

  Remainder Section;
}
```


Dr. Andrew Kwok-Fai Lui

Solution #2

process 1

```
while (true) {

  flag[1] = true;

  while (flag[2] == true)
; /* busy wait */

  Critical Section;

  flag[1] = false;

  Remainder Section;
}
```

flag

process 2

```
while (true) {

  flag[2] = true;

  while (flag[1] == true)
; /* busy wait */

  Critical Section;

  flag[2] = false;

  Remainder Section;
}
```

Solution #2

process 1

```
while (true) {

  flag[1] = true;


  while (flag[2] == true)
; /* busy wait */


  Critical Section;

  flag[1] = false;

  Remainder Section;
}
```

flag

process 2

```
while (true) {

  flag[2] = true;


  while (flag[1] == true)
; /* busy wait */


  Critical Section;

  flag[2] = false;

  Remainder Section;
}
```

Solution #2

process 1

```
while (true) {

  flag[1] = true;


  while (flag[2] == true)
; /* busy wait */


  Critical Section;

  flag[1] = false;

  Remainder Section;
}
```

flag

process 2

```
while (true) {

  flag[2] = true;


  while (flag[1] == true)
; /* busy wait */


  Critical Section;

  flag[2] = false;

  Remainder Section;
}
```

Verdict to Solution #2

- Solution fails the requirement of Progress
  - Process 1 executed: flag[1] = true;
  - Process 2 executed: flag[2] = true;
  - Process 1 will enter into the while loop because flag[2] is true
  - Process 2 will enter into the while loop because flag[1] is true
- The process cannot coordinate between them the entry into CS

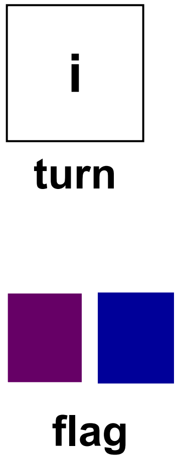
Solution #3: Peterson's Solution

```
/* PROCESS 1 */

while (true) {

flag[1] = true;
turn = 2;

while (flag[2] == true && turn == 2)
 ; /* busy wait */

Critical Section;

flag[1] = false;

Remainder Section;

}
```

```
/* PROCESS 2 */

while (true) {

flag[2] = true;
turn = 1;

while (flag[1] == true && turn == 1)
 ; /* busy wait */

Critical Section;

flag[2] = false;

Remainder Section;

}
```

Solution #3

```
while (true) {

  flag[1] = true;

  turn = 2;

  while (flag[2] == true
    && turn == 2)
; /* busy wait */

  Critical Section;

  flag[1] = false;

  Remainder Section;
}
```




```
while (true) {

  flag[2] = true;

  turn = 1;

  while (flag[1] == true
    && turn == 1)
; /* busy wait */

  Critical Section;

  flag[2] = false;

  Remainder Section;
}
```

Solution #3

```
while (true) {

  flag[1] = true;

  turn = 2;

  while (flag[2] == true
   && turn == 2)
; /* busy wait */

  Critical Section;

  flag[1] = false;

  Remainder Section;
}
```

process 1

1

turn

flag

```
while (true) {

  flag[2] = true;

  turn = 1;

  while (flag[1] == true
   && turn == 1)
; /* busy wait */

  Critical Section;

  flag[2] = false;

  Remainder Section;
}
```

Solution #3

```
while (true) {

  flag[1] = true;

  turn = 2;

  while (flag[2] == true
   && turn == 2)
; /* busy wait */

  Critical Section;

  flag[1] = false;

  Remainder Section;
}
```

process 1

2

turn

flag

```
while (true) {

  flag[2] = true;

  turn = 1;

  while (flag[1] == true
   && turn == 1)
; /* busy wait */

  Critical Section;

  flag[2] = false;

  Remainder Section;
}
```

Solution #3

process 1

```
while (true) {

  flag[1] = true;

  turn = 2;

  while (flag[2] == true
    && turn == 2)
; /* busy wait */

  Critical Section;

  flag[1] = false;

  Remainder Section;
}
```

2

turn

flag

```
while (true) {

  flag[2] = true;

  turn = 1;

  while (flag[1] == true
    && turn == 1)
; /* busy wait */

  Critical Section;

  flag[2] = false;

  Remainder Section;
}
```

Verdict to Solution #3

- Satisfied all three requirements of a critical section solution
  - Mutual exclusion
  - Progress
  - Bounded waiting

Atomic Execution

- The solution assumes the operations on shared variables are <u>atomic</u>
  - An atomic execution: execution is completed without the interruption of another process
    - Otherwise race condition would still occur
  - Atomic operations are achieved with computer hardware support

semaphores

Semaphore

- A semaphore is a tool for synchronization
  - Eases the implementation of critical section solutions
  - Supports other patterns of process synchronization
- Three components of a semaphore
  - A semaphore variable.
    - integer represents the number of available resource.
  - The wait function
    - Atomic operation accepts a semaphore variable as a parameter
  - The signal function
    - Atomic operation that accepts a semaphore variable as a parameter

CS Solution based on Semaphore

```
while (true) {
  S = 1; /* initialization */

  wait(S);
  Critical Section;
  signal(S);

  Remainder Section;
}
```

```
wait (S) {
   while (S <= 0)
         ; /* busy wait */
   S--;
}

signal (S) {
  S++;
}
```

Semaphore

- Initial value of the semaphore variable: maximum number of process that can enter into the critical section together
  - Set to 1 to allow at most one process to enter
  - Could be understood as the number of resources
- The wait function plays the role of a gate
  - It stops processes if the semaphore variable is not positive
- The signal function plays the role of a notifier
  - It notifies if there is any process waiting for entry into the CS

Semaphore Implementation

```
wait (S) {
  while (S <= 0)
        ; /* busy wait */
  S--;
}

signal (S) {
 S++;
}
```

Busy Wait wastes CPU cycles

Semaphore Implementation

```
wait (S) {
  S--;
  if (S < 0) {
add this process to a queue
block();
  }
}

signal (S) {
 S++;
 if (S <= 0) {
   P = a process removed from the queue wakeup(P);
 }
}
```

Using a queue to manage the waiting processes

Put the waiting processes to sleep to save CPU cycles

Example

Example 2: Using Semaphore for General Synchronization Tasks

Consider how to make process P1 executing a statement S1 before another process P2 executing another statement S2.

Answer:

There are two statements S1 and S2. S1 belongs to P1 and S2 belongs to P2. To make P1 executing S1 before P2 executing S2, a way is need to stop P2 from executing S2 first. We need a gate before S2 to stop P2. A semaphore is used for the gate.

- Wait(R) is called before S2. R is a semaphore variable. For the gate to stop P2, the semaphore variable R must be initialized to 0.
- Signal(R) is called after S1. P1 has already executed S1 and so we should open the gate that stops P2. Signal(R) notifies the gate to open.

Sem R = 0;

P1	***P2***
S1; Signal (R);	Wait (R); S2;

Semaphore and Deadlocks

- Improper use of semaphores can cause deadlocks
  - Deadlock means that a set of processes are unable to make any progress
  - Processes are waiting for a signal that never comes

Example: Deadlock

S and Q initialized to 1

P1	***P2***
wait(S);	wait(Q);
wait(Q);	**wait(S);**
...	...
signal(S);	signal(Q);
signal(Q);	signal(S);

S becomes 0 after wait(S)

Q becomes 0 after wait(Q)

Both P1 and P2 may get trapped here if S and Q are both 0

Example: Correct Solution

Example 3: Fix the Deadlock Problem

Fix the deadlock problem.

Answer:

The order of waiting is important. If the processes are calling wait in the same order, then the deadlock problem is resolved. The deadlock problem is caused by P1 holding the resource S and waiting for Q, and P2 holding Q and waiting for S. If the resources S and Q are always acquired in the same order, then this hold-and-wait situation will not occur.

P1	*P2*
wait(S);	wait(S);
wait(Q);	wait(Q);
…	…
signal(Q);	signal(Q);
signal(S);	signal(S);

Binary and Counter Semaphore

- Binary semaphore can support range of 0 and 1 only
  - Binary semaphores are also called mutex locks
  - Supported by computer hardware
- Counting semaphores basically have no restriction on the range
  - Counting semaphore can be developed from binary semaphores

Binary Semaphore to Counting Semaphore

```
Binary semaphores S1 and S2
Counting semaphore S
An integer variable C
```

```
wait (S) {
  wait(S1);
  C--;

  if (C < 0) {
    signal(S1);
    wait(S2);
  }
  else
    signal(S1);
}
```

```
signal (S) {
  wait(S1);
  C++;

  if (C <= 0)
    signal(S2);

  signal(S1);
}
```

Monitors an alternative

Monitors

- Monitors are easier to use than semaphores
  - Semaphores need correctly placed wait() and signal() as well as the initial value
- Monitor locks is a mutex lock applicable to any object in a program
  - Object oriented approach to synchronization
  - The Monitor is a module/class containing one or more functions

Monitors

Source Code

```
monitor class Counter {
  int count = 0;

  public method int add(int howmany) {
    count = count + howmany;
    return count;
  }
  public method int sub(int howmany){
    count = count - howmany;
    return count;
  }
}
```

Implemented Code

```
class Counter {
  Lock thisLock = new Lock();
  int count = 0;

  public method int add(int howmany) {
    thisLock.acquire();
    try {
      count = count + howmany;
      return count;
    } finally {
       thisLock.release();
    }
  }
  public method int sub(int howmany){
    thisLock.acquire();
    try {
      count = count - howmany;
      return count;
    } finally {
       thisLock.release();
    }
  }
}
```

Monitors

Syntactic sugar

Source Code

```
monitor class Counter {
  int count = 0;

  public method int add(int howmany) {
    count = count + howmany;
    return count;
  }
  public method int sub(int howmany){
    count = count - howmany;
    return count;
  }
}
```

Each object of this class becomes a critical section, and mutex applied

Implemented Code

```
class Counter {
  Lock thisLock = new Lock();
  int count = 0;

  public method int add(int howmany) {
    thisLock.acquire();
    try {
      count = count + howmany;
      return count;
    } finally {
       thisLock.release();
    }
  }
  public method int sub(int howmany){
    thisLock.acquire();
    try {
      count = count - howmany;
      return count;
    } finally {
       thisLock.release();
    }
  }
}
```

Monitor locks are acquired and released

case study: synchronization in java

Synchronization in Java

- Offer both semaphores and monitors
  - For simple mutex tasks
  - For sophisticated multiple threads coordination
- Three main methods
  - Using the `java.util.concurrent.Semaphore` class
    - Classical semaphores
  - The synchronized keyword
    - Monitor locks for simple mutex tasks
  - Inter-thread signaling
    - Threads to wait and signal each other

The synchronized Keyword

- Define a critical section in Java programs
- The critical section may be one of three scopes
  - Methods of an object as a critical section
  - The class as a critical section
  - Code block as a critical section

The synchronized Keyword

- Methods of an object as a critical section

```
public class Counter {
  private int count = 0;

  public synchronized int add(int howmany) {
    count = count + howmany;
    return count;
  }
  public synchronized int sub(int howmany){
    count = count - howmany;
    return count;
  }
}
```

Mutex applied to each object of this class

Critical section

No two threads can enter into either methods of an object at the same time

But two threads can enter into two different objects each at the same time

The synchronized Keyword

- The class as a critical section

```
public class Counter {
  private int count = 0;

  public static synchronized int add(int howmany) {
    count = count + howmany;
    return count;
  }
  public static synchronized int sub(int howmany){
    count = count - howmany;
    return count;
  }
}
```

Critical section

Mutex applied to the whole class

The synchronized Keyword

- A code block as a critical section

```
...
  AnObject lock = new AnObject();
  // AnObject is a class defined by user or use Object
...
  synchronized (lock) {
     // critical section under mutex
  }
```

Critical section

Mutex applied to this code block

Java Inter-Thread Signalling

- Java threads provide signalling support
  - A thread can wait and sleep
  - A thread can notify other threads

Methods	***Remarks***
`wait`	The thread calling wait will become inactive (or sleep) until it receives a notify signal.
`notify`	The thread calling notify will send a signal to one of the waiting threads (on the same synchronized object) and wake it up.
`notifyAll`	The thread calling notify will send a signal to all waiting threads (on the same synchronized object).

Java Inter-Thread Signalling

- Multiple threads can use wait and notify to synchronize their actions

```
public class Counter {
  AnObject obj = new AnObject();
  private int count = 0;

  public synchronized void method1() {
    ...
    obj.wait();  // this thread waits here
    ...
  }
  public synchronized void method2(){
    ...
    obj.signal();  // this thread sends signal to a waiting thread
    ...
  }
}
```

classic problems of synchronization

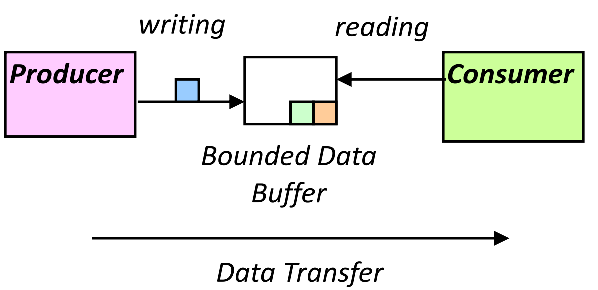
Classic Problems of Synchronization

- Some classic problems have been investigated in depth by computer scientists
  - Abstract the essence of a large range of general problems
- Abstraction is a powerful tool in computing
  - Simplify complex problems to their core
  - Scale of analysis becomes feasible
  - Real world problems are complicated
    - Simplified to save time and effort
  - Value of abstraction lies in generalization
    - Applicable to similar systems?

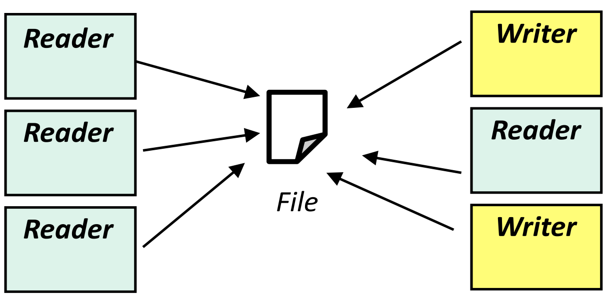
Classic Problems of Synchronization

- Three problems will be discussed

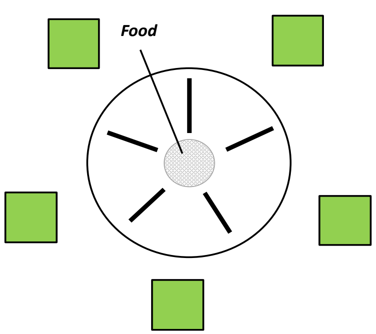
Producer Consumer Problem



Readers Writer Problem



Dining Philosophers Problem



Bounded Buffer Problem

- Producer-consumer problem
  - Two processes/threads
  - Producer generates data
  - Data sent to Consumer through a bounded buffer
  - Consumer receives data
- Abstracted many systems
  - Web browsers reading data
  - Word processor reading data
  - Sending emails from client to server

Bounded Buffer Problem

- There are two conditions that could cause a process unable to proceed – empty buffer and full buffer
- A solution based on semaphores
  - Semaphore mutex: CS solution for updating the shared data buffer
  - Semaphore full: stop the Producer from sending data to the buffer if the buffer is full
  - Semaphore empty: stop the Consumer from retrieving data from the buffer if the buffer is empty

Bounded Buffer Problem

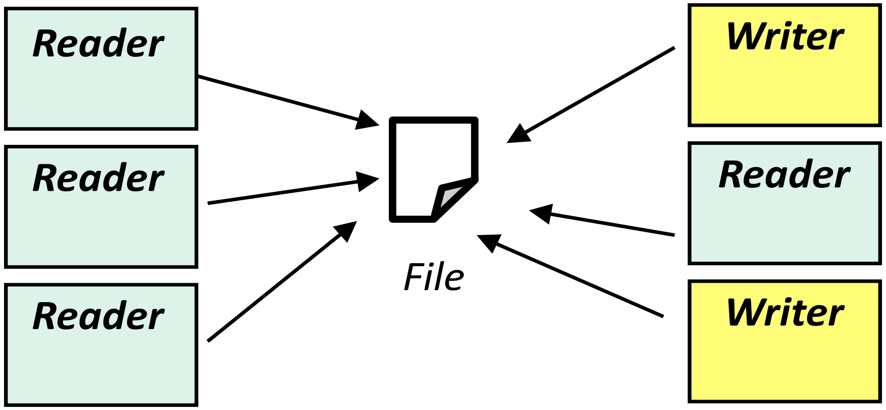
Producer	Consumer

`while (true) {`
  `wait(full);`
  `wait(mutex);`
 	
  `add a datam to buffer`
 	
  `signal(mutex);`
  `signal(empty);`
`}`
	`while (true) {`
	  `wait(empty);`
	  `wait(mutex);`
	
	  `remove a datum from buffer`
	
	  `signal(mutex);`
	  `signal(full);`
	`}`

Sem full = size of buffer;
Sem empty = 0;
Sem mutex = 1;

Classic Problems of Synchronization

Readers Writer Problem



Readers Writers Problem

- It is also known as file sharing problem
- There are certain rules concerning read/write file access permission
  - A file can allow many readers to access at a time
  - A file can only allow one writer
  - When a writer is allowed, no reader is allowed
- Problem analysis
  - At most only one writer is allowed
  - If there is no writer, then any number of readers is allowed

Readers Writers Problem

- A solution based on semaphores
  - The semaphore mutex: ensuring mutual exclusion over the shared variable readercount
    - This variable is shared between readers.
  - The semaphore w: stopping writer if there is already at least one reader, or stopping a reader if there is already one writer in CS

Readers Writers Problem

Reader	Writer

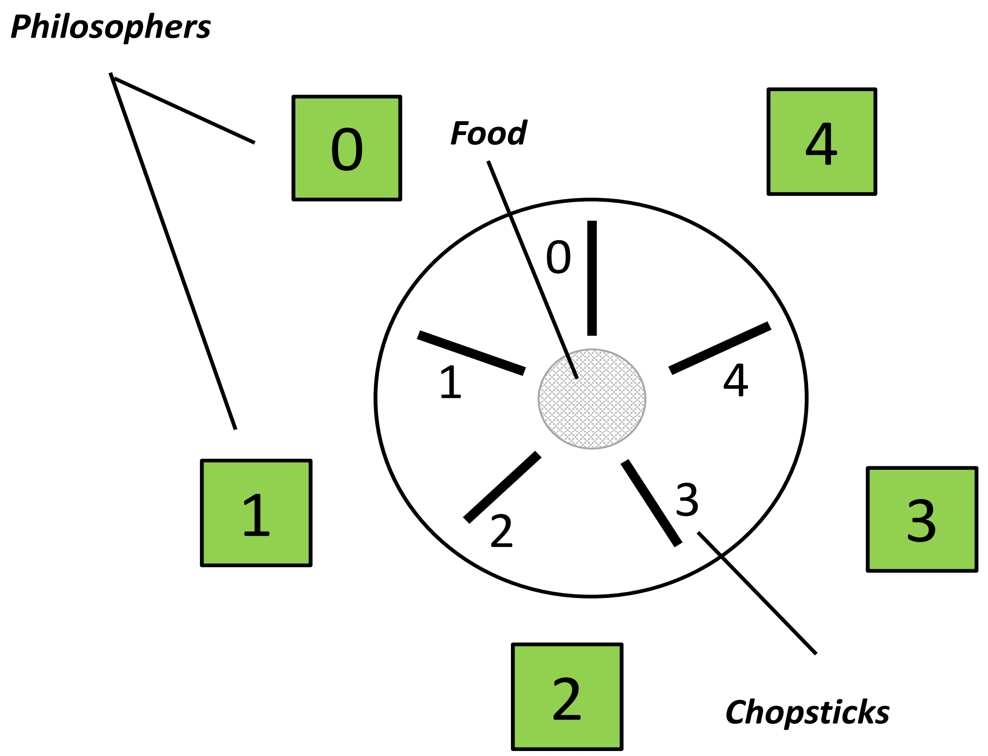
`wait(mutex);`
`readercount++;`
`if (readercount == 1)`
`  wait(w);`
`signal(mutex);`
 	
`/* critical section */`
 	
`wait(mutex);`
`readercount--;`
`if (readercount == 0)`
`  signal(w);`
`signal(mutex);`
	`wait(w);`
	
	`/* critical section */`
	
	`signal(w);`

Sem mutex = 1;
Sem w = 1;

Dining Philosophers Problem

- A famous problem because it abstracts problems involving sharing a number of resources between multiple processes
  - N philosophers at a dining table
  - Each philosopher has a plate
  - N chopsticks on the table, one between each pair of philosopher
  - To eat, a philosopher must have acquired the two chopsticks on each side
    - A philosopher must take one chopstick at one time
  - After eaten, the philosopher replaces the chopsticks

Dining Philosophers Problem



Dining Philosophers Problem

http://www.doc.ic.ac.uk/~jnm/book/book_applets/Diners.html

Dining Philosophers Problem

- The chopsticks are shared resources
  - The philosophers are numbered 0 to 4
  - The chopsticks are also numbered 0 to 4
  - Chopstick 0 is on the left of philosopher 0 and on the right of philosopher 4
- Multiple processes sharing multiple resource instances
  - Philosopher 0 needs chopsticks 0 and 1
  - Philosopher 1 needs chopsticks 1 and 2
  - Philosopher 2 needs chopsticks 2 and 3

Dining Philosophers Problem

Philosopher

```
/* PHILOSOPHER I */

while (true) {
  wait(chopstick [I]); /* get the left chopstick */
  wait(chopstick [(I+1)%5]);  /* get the right chopstick */

... eat

  signal(chopstick [I]);
  signal(chopstick [(I+1)%5]);

}
```

Sem chopstick[0] ... chopstick[4] = 1;

Verdict of the Solution

- The solution can cause deadlock
  - Each philosopher happens to obtain the left chopstick only

Example: Fix the Deadlock

Example 4: Fix the Deadlock Problem

Fix the deadlock problem of the above dining philosopher solution.

Answer:

There are three possible solutions:

1. Allow at most 4 philosophers to share five chopsticks. One philosopher will have a pair of chopsticks to get food and will not have to wait.

2. A philosopher is not allowed to hold one chopstick and wait for another. The philosopher can only hold chopsticks if both are available.

3. Impose an order of picking up the chopsticks. Philosophers with odd ID always pick the left chopstick before the right. Philosophers with even ID always pick the right chopstick before the left.

Example: Fix the Deadlock

Philosopher

```
/* PHILOSOPHER I */

while (true) {
if (I % 2 == 1) {
  wait(chopstick [I]); /* get the left chopstick */
  wait(chopstick [(I+1)%5]);  /* get the right chopstick */
} else {
  wait(chopstick [(I+1)%5]);  /* get the right chopstick */
  wait(chopstick [I]); /* get the left chopstick */
}

... eat

if (I % 2 == 1) {
  signal(chopstick [I]);
  signal(chopstick [(I+1)%5]);
} else {
  signal(chopstick [(I+1)%5]);
  signal(chopstick [I]);
}

}
```

Sem chopstick[0] ... chopstick[4] = 1;