Process Management

Synchronization

Lecture 6

Overview

- **Race Condition**
  - Problems with concurrent execution
- **Critical Section**
  - Properties of correct implementation
  - Symptoms of incorrect implementation
- **Implementations of Critical Section**
  - Low level
  - High level language
  - High level abstraction
- **Classical synchronization problems**

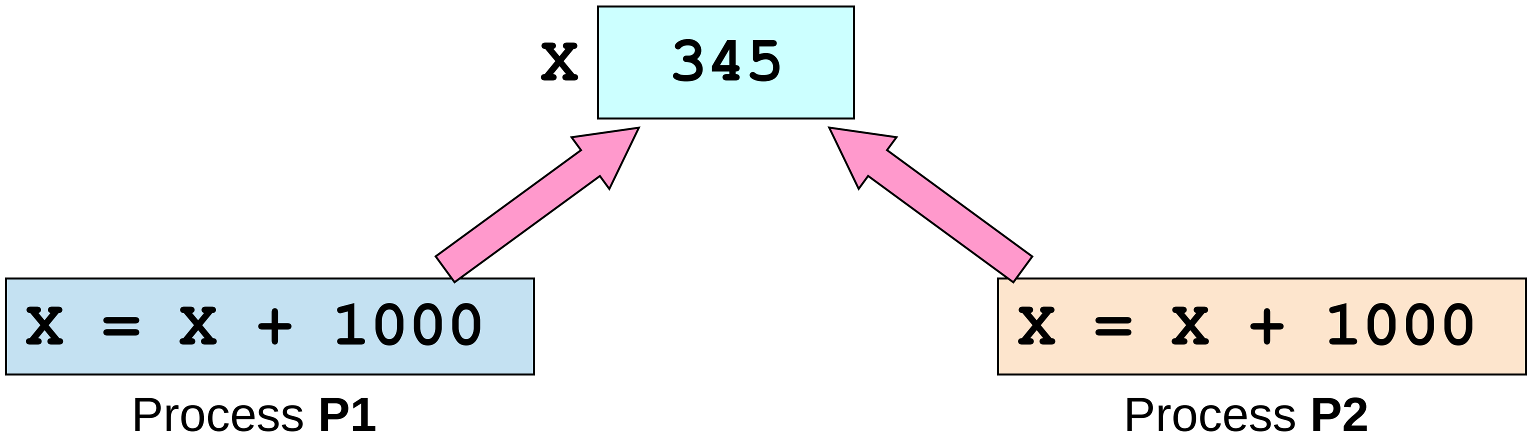
Problems with Concurrent Execution

- When two or more processes:
  - Execute concurrently in interleaving fashion AND
  - Share a modifiable resource
  - → **Can cause synchronization problems**

- Execution of a single sequential process is ***deterministic***
  - Repeated execution gives the same result

- Execution concurrent processes may be non-deterministic
  - Execution outcome depends on the order in which the shared resource is access/modified
  - known as **race conditions**

Race Condition: Illustration



- Process **`P1`** and **`P2`** shares a variable **`X`**
- The statement **`X = X + 1000`** can be roughly translated as the following machine instructions:

1. **`Load X ➔ Register1`**
2. **`Add 1000 to Register1`**
3. **`Store Register1 ➔ X`**

this will be 3 assembly instructions

Race Condition: Good behavior

Time	Value of X	P1	P2
1	345	`Load X` ➔ `Reg1`	
2	345	`Add 1000 to Reg 1`	
3	1345	`Store Reg1` ➔ `X`	
4	1345		`Load X` ➔ `Reg1'`
5	1345		`Add 1000 to Reg1'`
6	**2345**		`Store Reg1'` ➔ `X`

- The above execution order exhibits good behavior:
  - Give the desired result **`2345`**

Race Condition: Bad behavior

Time	Value of X	P1	P2
1	345	`Load X ➔ Reg1`	
2	345	`Add 1000 to Reg1`	
3	345	this might happen due to timing interrupt	`Load X ➔ Reg1'`
4	345		`Add 1000 to Reg1'`
5	1345	`Store Reg1 ➔ X`	
6	**1345**		`Store Reg1' ➔ X`

- There are many other execution sequence that exhibit good/bad behaviors!

The 2 processes execute concurrently in interleaving fashion and share the same modifiable resource

Race Condition: **Solution**

- Incorrect execution is due to the **unsynchronized access to shared modifiable resources**

- General outline of solution:
  - Designate code segment with race condition as **critical section**
  - At any point in time, only **one process** can execute in the critical section

➔Other processes are prevented from entering the same critical section

Critical Section (CS)

- Generic Skeleton of code with Critical Section(s):

idea of critical section:
- having a door in the bathroom
- so only 1 person can enter at one time
- there is a need to ensure that the
-- person inside must exit in a timely fashion

Critical Section

```
        //Normal code
Enter CS
        //Critical Work
Exit CS
        //Normal code
```

- Example:

```
Enter CS
      X = X + 1000
Exit CS
```

Process **P1**

```
Enter CS
      X = X + 1000
Exit CS
```

Process **P2**

Properties of Correct CS Implementation

Mutual Exclusion:

- If process P_i is executing in critical section, all other processes are prevented from entering the critical section.

Progress:

- If no process is in a critical section, one of the waiting processes should be granted access.

there should have nothing blocking you if there is no one in the critical section

Bounded Wait:

bounded wait will not cause starvation

- After process P_i request to enter critical section, there exists an upper bound on the number of times other processes can enter the critical section before P_i.

Independence:

- Process **not** executing in critical section should never block other process.

Symptoms of Incorrect Synchronization

- **Deadlock:**
  - All processes blocked ➔ no progress

analogy:
Deadlock - 2 ppl walk into each other on the street
Livelock - 2 ppl walking towards each other trying to avoid but keep blocking each other

- **Livelock:**
  - Usually related to deadlock avoidance mechanism
  - Processes keep changing state to avoid deadlock and make no other progress
  - Typically processes are not blocked

as a whole no progress is being made - usually happens where there is some deadlock avoidance

- **Starvation:**
  - Some processes never get to make progress in their execution because it is perpetually denied necessary resources

CS Implementations Overview

- Assembly level implementations:
  - Mechanisms provided by the processor
- High level language implementations:
  - Utilizes only normal programming constructs
- High level abstraction:
  - Provide abstracted mechanisms that provide additional useful features
  - Commonly implemented by assembly level mechanisms

Don’t worry! The processor has all the answers!

ASSEMBLY LEVEL IMPLEMENTATION

Test and Set: An Atomic Instruction

- A common machine instruction provided by processors to aid synchronization

```
TestAndSet Register, MemoryLocation
```

- **Behavior:**
  1. Load the current content at `MemoryLocation` into `Register`
  2. Stores a 1 into `MemoryLocation`

  - Important: The above is performed as a **single machine operation**, i.e., **atomic**

Using Test and Set

- For ease of discussion, assume that the ***TestAndSet*** machine instruction has an equivalent high level language version

TestAndSet() takes a memory address M:
- **Returns the current content at M**
- **Set content of M to 1**

```
void EnterCS( int* Lock )
{
    while( TestAndSet( Lock ) == 1);
}
```

```
void ExitCS( int* Lock )
{
    *Lock = 0;
}
```

Satisfy:
- Mutual exclusion
- Progress
(since both cannot keep setting lock at the same time)
-

Observations and Comments

- The implementation works!
  - However, it employs **busy waiting** (keep checking the condition until it is safe to enter critical section)
  - ➔ Wasteful use of processing power

- Variants of this instruction exists on most processors:
  - Compare and Exchange
  - Atomic Swap
  - Load Link / Store Conditional

Using only your brain power…. ☺

HIGH LEVEL LANGUAGE IMPLEMENTATION

Using HLL: **Attempt 1**

Lock: 0

```
while (Lock != 0);
Lock = 1;
   Critical Section
Lock = 0;
```

Process **P0**

```
while (Lock != 0);
Lock = 1;
   Critical Section
Lock = 0;
```

Process **P1**

- Makes intuitive sense ☺
  - But it doesn't work properly ☹

- It violates the "Mutual Exclusion" requirement!
  - How?

when the 2 checking statements in the processes run after each other
- so P0 check Lock != 0; then P1 checks, then both are able to go into the critical section - and only then they lock

Using HLL : **Attempt 1 Fixed***

Lock | 0

will not be able to stop mutual exclusion on a multi processor implementation

```
//Disable Interrupts
while (Lock != 0);
Lock = 1;
    Critical Section
Lock = 0;
//Enable Interrupts
```

Process **P0**

```
//Disable Interrupts
while (Lock != 0);
Lock = 1;
    Critical Section
Lock = 0;
//Enable Interrupts
```

Process **P1**

- Solve the problem by preventing context switch
- However:
  - Buggy critical section may stall the WHOLE system
  - Busy waiting
  - Requires permission to disable/enable interrupts

Using High Level Language: **Attempt 2**

Turn: 0

```
while (Turn != 0);
    Critical Section
Turn = 1;
```

Process **P0**

```
while (Turn != 1);
    Critical Section
Turn = 0;
```

Process **P1**

- **Assumption:**
  - P0 and P1 executes the above in loop
  - Take turn to enter **critical section**
- **Problems:**
  - Starvation:
    - e.g., If P0 never enters CS, P1 starves
  - Violate the **independence** property!

Satisfy:
mutual exclusion

Does not satisfy:
progress - if P0 does not enter, then P1 does not enter
(so even if there is no one in the critical section, P1 cannot enter)

Using High Level Language: **Attempt 3**

Want[0]	0
Want[1]	0

```
Want[0] = 1;
while (Want[1]);
   Critical Section
Want[0] = 0;
```

Process **P0**

```
Want[1] = 1;
while (Want[0]);
   Critical Section
Want[1] = 0;
```

Process **P1**

- Solve the independence problem
  - If P0 or P1 is not around, another process can still enter CS
- Problem:
  - Deadlock! Try identify the execution sequence that causes deadlock

Peterson's Algorithm

Turn	0
Want[0]	0
Want[1]	0

since Turn can only be set with 1 value, thus only 1 process is able to do at a time

```
Want[0] = 1;
Turn = 1;
while (Want[1] &&
        Turn == 1);
    Critical Section
Want[0] = 0;
```

Process **P0**

```
Want[1] = 1;
Turn = 0;
while (Want[0] &&
        Turn == 0);
    Critical Section
Want[1] = 0;
```

Process **P1**

- Assumption:
  - Writing to `Turn` is an **atomic** operation

Peterson's Algorithm: **Disadvantages**

- Busy Waiting:
  - The waiting process repeatedly test the while-loop condition instead of going into blocked state:
- Low level:
  - Higher-level programming construct is desirable
    - simplify mutual exclusion
    - less error prone
- Not general:
  - General synchronization mechanism is desirable
    - Not just mutual exclusion

Let's go meta……

HIGH LEVEL ABSTRACTION

High Level Synchronization Mechanism

- Semaphore:
  - An generalized synchronization mechanism
  - Only behaviors are specified ➔ can have different implementations
  - Provides
    - A way to block a number of processes
      - Known as **sleeping process**
    - A way to unblock/wake up one or more sleeping process

- History:
  - Proposed by **Edgar W. Dijkstra** in 1965

Semaphore: Wait( ) and Signal( )

- A semaphore **S** contains an integer value
  - Can be initialized to any non-negative values initially
- Two **atomic** semaphore operations:

Wait(S)

- If S <= 0, blocks (go to sleep)
- Decrement S

thus S will keep track of how much empty space there is left

- Also known as `P()` or `Down()`

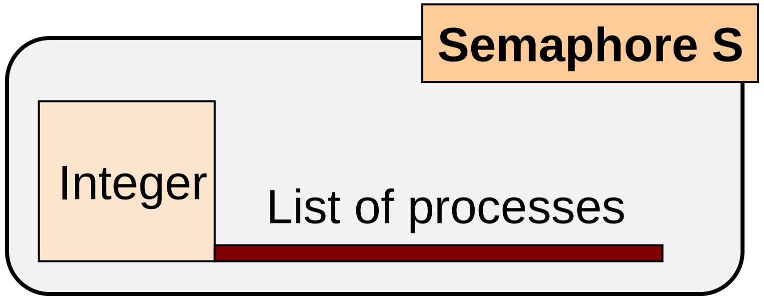
Signal(S)

- Increments S
- Wakes up one sleeping process if any
- This operation **never** blocks
- Also known as `V()` or `Up()`

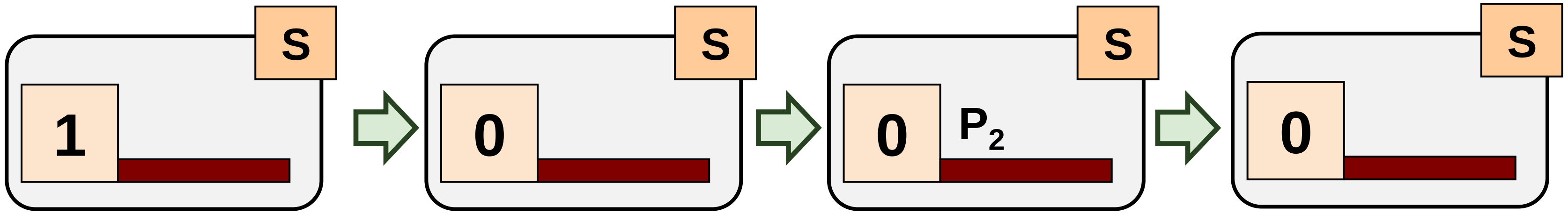
- **Reminder:** The above specifies the **behavior**, not the implementations

Semaphore: **Visualization**

- To aid understanding, you can visualize semaphore as:
  - A protected integer
  - A list to keep track of waiting processes



- Example:



After initialization

After P_1 wait(S)
P_1 proceeds

After P_2 wait(S)
P_2 suspended

After P_3 signal(S)
P_3 proceeds
P_2 wakes up, decreases S and proceeds

Semaphores: **Properties**

- Given:
  - $S_{Initial} \geq 0$

- Then, the following **invariant** must be true:

$$S_{current} = S_{Initial} + \#signal(S) - \#wait(S)$$

initial ++ -- should be true

  - `#signal(S)` :
    - number of signals() operations executed
  - `#wait(S)` :
    - number of wait() operations **completed**

General and Binary Semaphores

- **General semaphore** S:
  - S ≥ 0 (S = 0, 1, 2, 3, ...)
  - also called counting semaphores

- **Binary semaphore** S:
  - S = 0 or 1

- General semaphore is provided for convenience
  - Binary semaphore is sufficient
  - i.e., general semaphore can be mimicked by binary semaphores

since can always start from one and +- after each wait signal pair

Semaphore Example: **Critical Section**

- Binary semaphore `S = 1`
- For any process:

```
Wait( S );
    Critical Section
Signal( S );
```

- In this case, S can only be 0 or 1
  - Can be deduced by the semaphore invariant
- This usage of semaphore is commonly known as **mutex** (**mut**ual **ex**clusion)

Mutex: Correct CS - Informal Proof

- Mutual Exclusion:
  - N_{CS} = Num of processes in critical section
    = Num of processes that completed **`wait()`** but not **`signal()`**
    = **`#Wait( S ) - #Signal( S )`**
  - $S_{Initial}$ **= 1**
  - $S_{current}$ **= 1 + #Signal(S) - #Wait(S)**
  - $S_{current}$ **+** $\mathbf{N_{CS}}$ **= 1**
  - Since $S_{current} \geq 0 \rightarrow N_{CS} \leq 1$

Mutex: Correct CS - Informal Proof (cont)

- **Deadlock:**
  - Deadlock means **all** processes stuck at **`wait( S )`**

    $\Rightarrow$ $S_{curent} = 0$ and $N_{CS} = 0$
  - But S_{curent} $\mathbf{+ N_{CS} = 1}$
  - $\Rightarrow\Leftarrow$ (contradiction)

- **Starvation:**
  - Suppose **`P1`** is blocked at **`wait( S )`**
  - **`P2`** is in CS, exits CS with **`signal( S )`**
    - If no other process sleeping, **`P1`** wakes up
    - If there are other process, **`P1`** eventually wakes up (assuming fair scheduling)

Incorrect Use of Semaphore: Deadlock

- Deadlock is still possible with incorrect use of semaphore
- Example:
  - Assume semaphores **P = 1**, **Q = 1** initially

due to interleaving - there is a chance where both P and Q will be 0 when lines 1 & 2 run after each other

```
Wait( P )   1
Wait( Q )   3
Some Code
Signal( Q )
Signal( P )
```

Process **P0**

```
Wait( Q )   2
Wait( P )   4
Some Code
Signal( P )
Signal( Q )
```

Process **P1**

Other High Level Abstractions

- Semaphore is very powerful:
  - There are no known unsolvable synchronization problem with semaphore (so far ☺)
  - Other high level abstractions essentially provide extended features that are inconvenient to express using semaphore alone

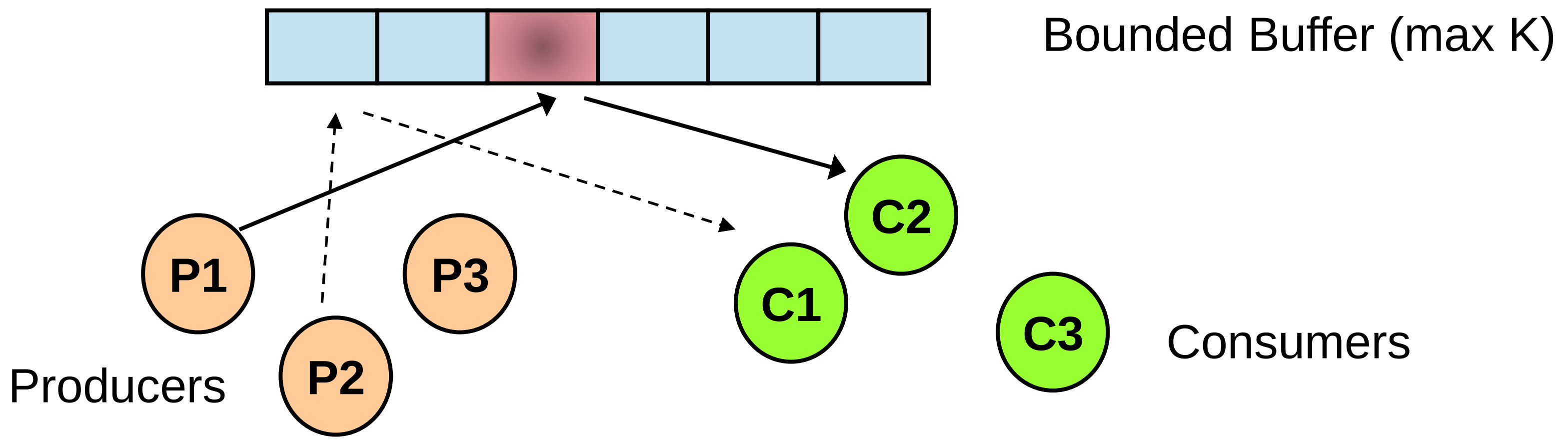
- Common alternative: **Conditional Variable**
  - Allow a task to wait for certain event to happen
  - Has the ability to ***broadcast***, i.e., wakes up all waiting tasks
  - related to **monitor**

Killing brain cells of generations of students…..

CLASSICAL SYNCHRONIZATION PROBLEMS

Producer Consumer: Specification

- Processes share a bounded buffer of size K
  - **Producers** produce items to insert in buffer
    - Only when the buffer is **not full** (< K items)
  - **Consumers** remove items from buffer
    - Only when the buffer is **not empty** (> 0 items)



using a circular buffer - with a count to check the total number in the buffer
- then(in) and (out) will specify the place

Producer Consumer: Busy Waiting

```
while (TRUE) {
    Produce Item;
    while(!canProduce);
    wait( mutex );
    if (count < K) {
        buffer[in] = item;
        in = (in+1) % K;
        count++;
        canConsume = TRUE;
    } else
        canProduce = FALSE;
    signal( mutex );
}
```

Producer Process

```
while (TRUE) {
    while (!canConsume);
    wait( mutex );
    if (count > 0) {
        item = buffer[out];
        out = (out+1) % K;
        count--;
        canProduce = TRUE;
    } else
        canConsume = FALSE;
    signal( mutex );
    Consume Item;
}
```

Consumer Process

- Initial Values:
  - **`count = in = out = 0`**
  - **`mutex = S(1)`** //semaphore with initial value 1
  - **`canProduce = TRUE`** and **`canConsume = FALSE`**;

Producer Consumer: **Busy Waiting**

- `canConsume`:
  - Triggers consumer to *try* to get item
- `canProduce`:
  - Triggers producer to *try* to produce item

- `wait(mutex) + signal(mutex)` : Creates a CS
- `in  = (in+1) % K` :

  `out = (out+1)% K` : Wraps around, circular array

- Evaluation:
  - The code **correctly solves** the problem
  - However, **busy-waiting** is used

Producer Consumer: **Blocking Version**

```
while  (TRUE)  {
        Produce Item;

        wait( notFull );
        wait( mutex );
        buffer[in] = item;
        in = (in+1) % K;
        count++;
        signal( mutex );
        signal( notEmpty );
}
```

Producer Process

```
while  (TRUE)  {

        wait( notEmpty );
        wait( mutex );
        item = buffer[out];
        out = (out+1) % K;
        count--;
        signal( mutex );
        signal( notFull );

        Consume Item;
}
```

Consumer Process

- Initial Values:
  - **count = in = out = 0**
  - **mutex = S(1), notFull = S(K), notEmpty = S(0)**

Producer Consumer: **Blocking Version**

- `wait( notFull )` : Forces producers to go to sleep
- `wait( notEmpty )` : Forces consumers to go to sleep
- `signal( notFull )` : 1 consumer wakes up 1 producer
- `signal( notEmpty )` : 1 producer wakes up 1 consumer

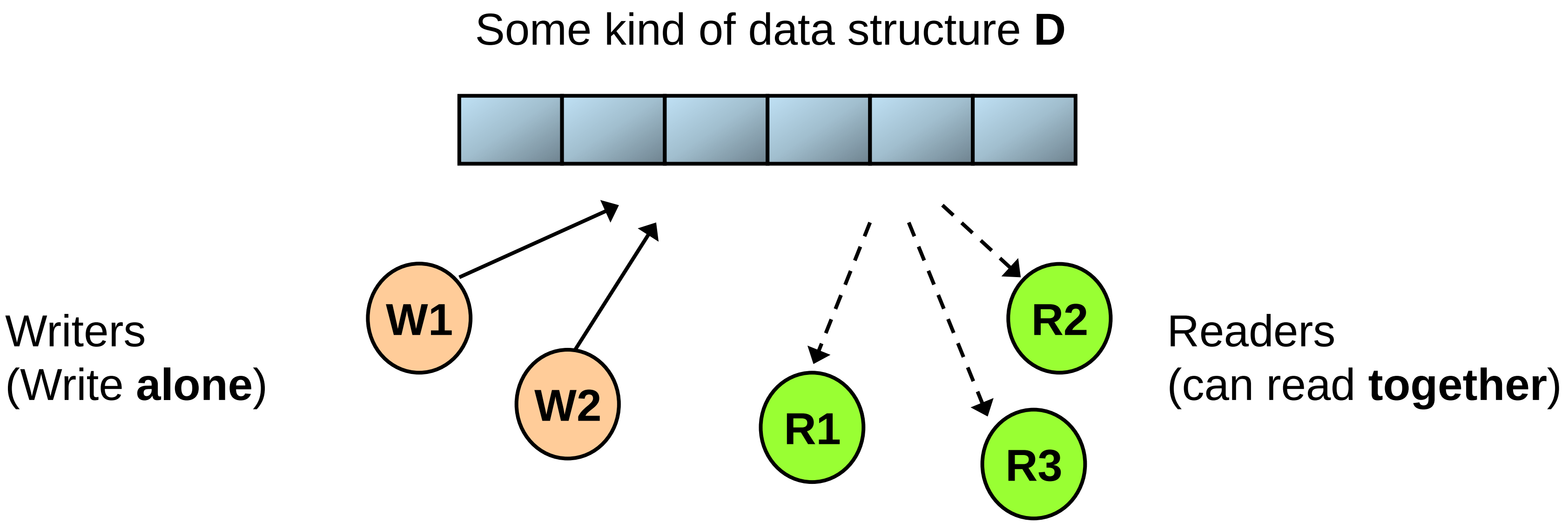
- Evaluation:
  - This code correctly solve the problem
  - No busy-waiting, “unwanted” producer/consumer will go to sleep on respective semaphores

Readers Writers: Specification

- Processes share a data structure **D**:
  - Reader: Retrieves information from **D**
  - Writer: Modifies information in **D**
- Writer must have exclusive access to D
- Reader can access with other readers

at most 1 writer on each data

but reader can have multiple accessing the data - then writers cannot write



Readers Writers: **Simple Version**

```
while (TRUE) {

    wait( roomEmpty );

    Modifies data

    signal( roomEmpty );
}
```

Writer Process

```
while (TRUE) {

    wait( mutex );
    nReader++;
    if (nReader == 1)
        wait( roomEmpty );
    signal( mutex );

    Reads data

    wait( mutex );
    nReader--;
    if (nReader == 0)
        signal( roomEmpty );
    signal( mutex );

}
```

Reader Process

- Initial Values:
  - `roomEmpty = S(1)`
  - `mutex = S(1)`
  - `nReader = 0`

Readers Writers: **Evaluation**

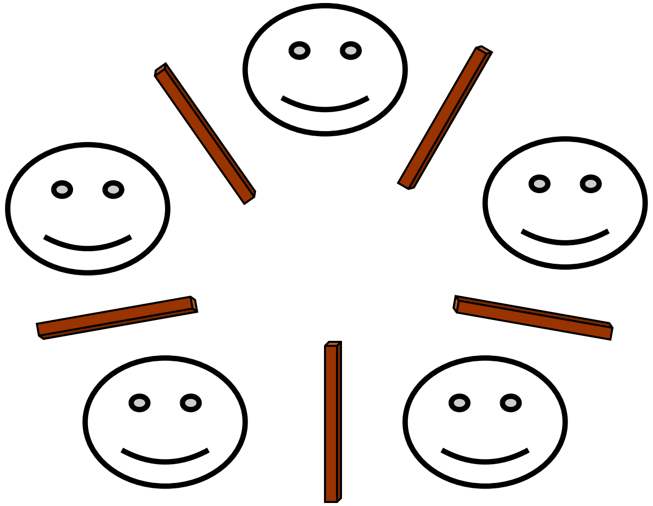
- Convince yourself that the solution satisfies the specification

- However:
  - It has one problem

  - (hint: Something to do with writer….)

writer might starve because readers can keep coming into the room - which means the writer cannot go in

Dining Philosophers: Specification



- Five philosophers are seated around a table
  - There are five single chopstick placed between each pair of philosopher
  - When any philosopher wants to eat, he/she will have to acquire both chopsticks from his/her left and right
- Devise a **deadlock-free** and **starve-free** way to allow the philosopher to eat freely

Dining Philosophers: **Attempt 1**

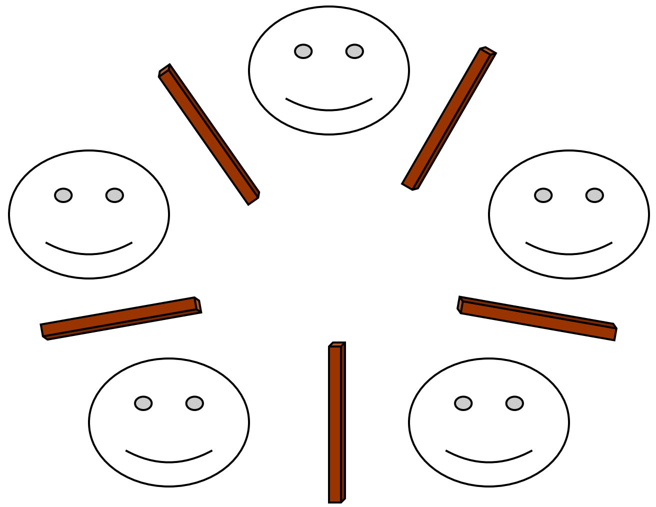
```
#define N 5
#define LEFT i
#define RIGHT ((i+1) % N)

//For philosopher i
while (TRUE){

      Think( );
      //hungry, need food!
      takeChpStk( LEFT );
      takeChpStk( RIGHT );

      Eat( );

      putChpStk( LEFT );
      putChpStk( RIGHT );
}
```



- Can you figure out the problem?

Dining Philosophers: **Attempt 1**

- Deadlock:
  - All philosopher simultaneously takes up the left chopstick, and none can proceed
- Fix attempt:
  - Make the philosopher to put down the left chopstick if right chopstick cannot be acquired
    - Try again later
  - No deadlock:
    - Livelock: All philosopher take up left chopstick, put it down, take it up, put it down, ...

Dining Philosopher: **Attempt 2**

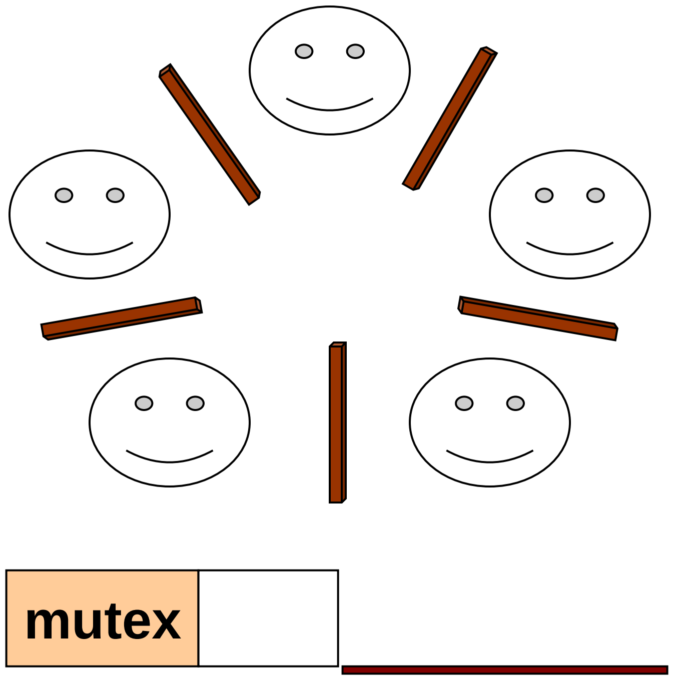
```
#define N 5
#define LEFT i
#define RIGHT ((i+1) % N)

//For philosopher i
while (TRUE){
       Think( );

       wait( mutex );

       takeChpStk( LEFT );
       takeChpStk( RIGHT );
       Eat( );
       putChpStk( LEFT );
       putChpStk( RIGHT );

       signal( mutex );
}
```



- Two questions:
  - Does it work?
  - Is it good?

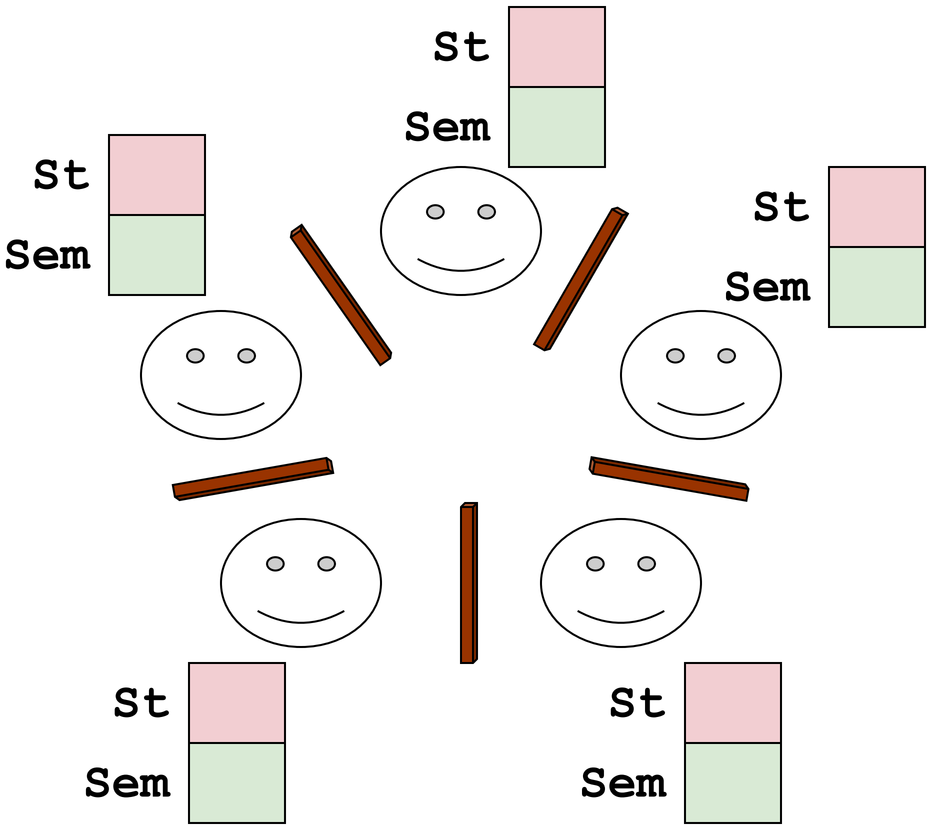
Dining Philosopher: Tanenbaum Solution

```
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)

#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher( int i ){
    while (TRUE){
        Think( );
        takeChpStcks( i );
        Eat( );
        putChpStcks( i );
    }
}
```

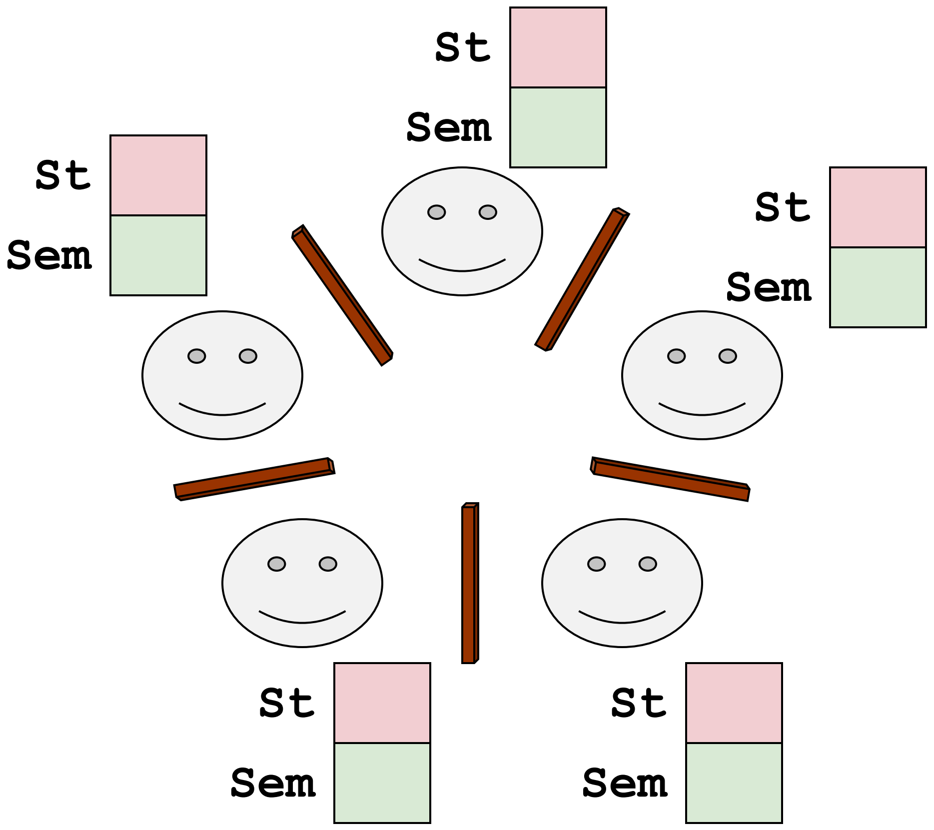


Dining Philosopher: Tanenbaum Solution

```
void takeChpStcks( i )
{
        wait( mutex );
        state[i] = HUNGRY;
        safeToEat( i );
        signal( mutex );
        wait( s[i] );
}
```

```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

         state[ i ] = EATING;
         signal( s[i] );
    }
}
```

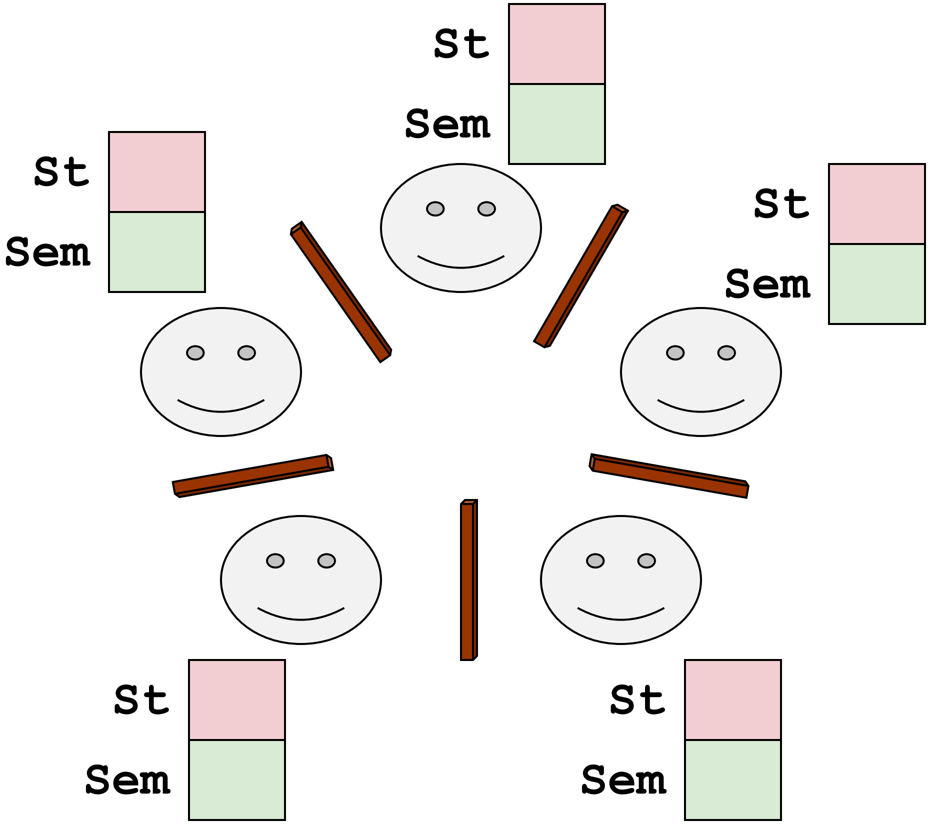


Dining Philosopher: Tanenbaum Solution

```
void putChpStcks( i )
{
        wait( mutex );

        state[i] = THINKING;
        safeToEat( LEFT );
        safeToEat( RIGHT );

        signal( mutex );
}
```

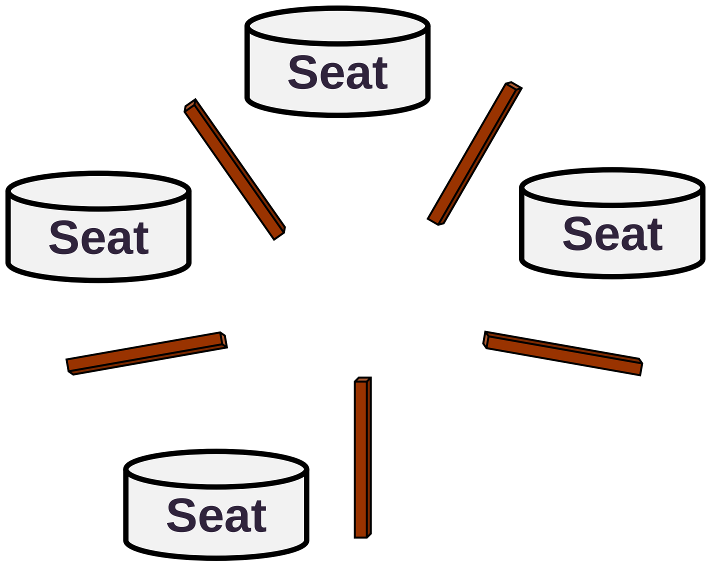


Dining Philosopher: **Limited Eater**

- If **at most 4 philosophers are allowed to sit at the table (leaving one empty seat)**

➔ **Deadlock is impossible!**

```
void philosopher( int i ){
     while (TRUE){
        Think( );
        wait( seats );
        wait( chpStk[LEFT] );
        wait( chpStk[RIGHT] );
        Eat( );
        signal( chpStk[LEFT] );
        signal( chpStk[RIGHT] );
        signal( seats );
     }
}
```



- Initial Values:
  - `seats = S(4)`
  - `chpStk = S(1)[5]`

SYNCHRONIZATION IMPLEMENTATIONS

POSIX Semaphore

- Popular implementation of semaphore under Unix
- Header File:
  - `#include <semaphore.h>`
- Compilation Flag:
  - `gcc something.c -lrt`
  - Stand for "real time library"
- Basic Usage:
  - Initialize a semaphore
  - Perform **`wait()`** or **`signal()`** on semaphore

pthread Mutex and Conditional Variables

- Synchronization mechanisms for pthreads
- Mutex (`pthread_mutex`):
  - Binary semaphore (i.e., equivalent Semaphore(1)).
  - Lock: `pthread_mutex_lock()`
  - Unlock: `pthread_mutex_unlock()`

- Conditional Variables(`pthread_cond`):
  - Wait: `pthread_cond_wait()`
  - Signal: `pthread_cond_signal()`
  - Broadcast: `pthread_cond_broadcast()`

Others

- Programming languages with thread support will have some forms of synchronization mechanisms

- Examples:
  - **Java**: all object has built-in lock (mutex), **synchronized** method access, etc.
  - **Python**: supports mutex, semaphore, conditional variable, etc.
  - **C++**: Added built-in thread in C++11; Support mutex, conditional variable

Summary

- Synchronization:
  - Problem: Race condition
  - Solution: Critical Section
  - Criteria of good solution:
    - Mutual Exclusion, progress, bounded waiting time, independence
  - Important High Level Construct: Semaphore

- Classic Synchronization problems:
  - Producer + Consumer
  - Reader + Writer
  - Dining Philosophers

Reference

- Modern Operating System (3rd Edition)
  - Chapter 2.4
- Operating System Concepts (7th Edition)
  - Chapter 5
- Edgar W. Dijkstra, "Note No.123: Cooperating Sequential Processes"
  - http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF