Synchronization

Process Coordination

Race Condition

- A **race condition** is when multiple processes concurrently read and write to a shared memory location and the result depends on the order of execution

- Practical example: Account Holder 1 (Husband) and Holder 2 (Wife) withdraw ₹20000 cash simultaneously from different ATMs, when account had only ₹30000 to begin with.

Race Condition in Programming

- Parallel execution of count++ and count--
- count++ implementation

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- implementation

```
register2 = count
register2 = register2 - 1
count = register2
```

Execution order differences

Initial value = 5, Final Result = 5

```
register1 = count
register1 = register1 + 1
count = register1
```

```
                                register2 = count
                                register2 = register2 - 1
                                count = register2
```

Initial value = 5, Final Result = 4

```
register1 = count
                                register2 = count
                                register2 = register2 - 1
register1 = register1 + 1
count = register1
                                count = register2
```

Initial value = 5, Final Result = 6

```
register1 = count
                                register2 = count
                                register2 = register2 - 1
                                count = register2
register1 = register1 + 1
count = register1
```

Solutions?

- Atomic Operations
- Critical Section Selection

Atomics

- Group of operations is called **atomic** if all of them must be conducted as if it were a single operation
- Atomic operations **cannot** be interrupted by other operations.
- If due to any reason atomic operation set is not fully executed, entire atomic block is rolled back

Atomics

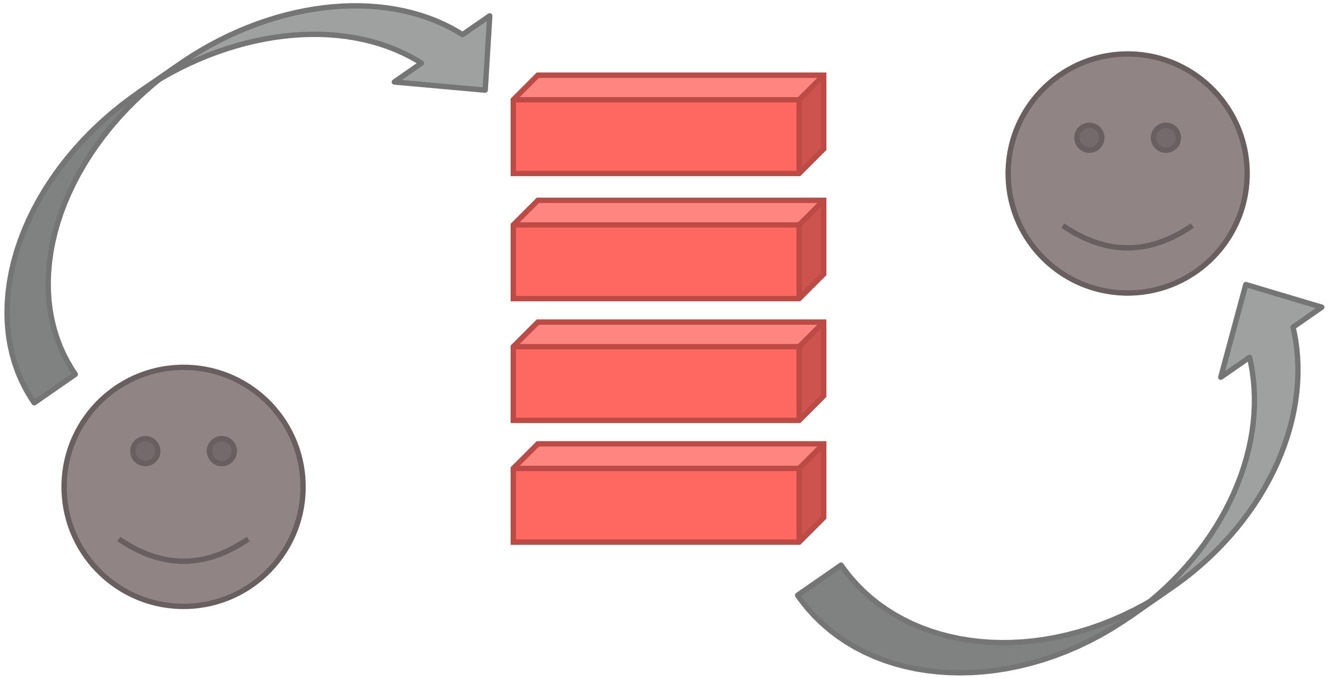
```
atomic {
      register1 = count
      register1 = register1 + 1
      count = register 1
}
```

Producer – Consumer Problem

A generic form of previously stated problem

Considered as a classical computer science problem, and significant in the history of multi-process operating systems

Producer and Consumer



Producer

```
while (true) {
  nextProduced = produce();
  while (counter == BUFFER_SIZE) {
    // just wait when all slots are full
  }
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}
```

Consumer

```
while (true) {
  while (counter == 0) {
    // just wait when all slots are empty
  }
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;
  consume(nextConsumed)
}
```

Critical Section Selection

Shared Resources

- Resources ?
  - Computer components that a process needs to execute itself
    - memory space
    - filesystem access
    - I/O device access
- Shared Resource
  - When the same resource needs to be made available to **2 or more** processes
  - There is a probability of more than 1 process simultaneously accessing and modifying the quantity/quality of the resource

Critical Section

- A **critical section** is part of program that accesses shared resource that cannot be concurrently accessed (without adverse effect)
- For example, during count++ or count--, reading and writing the ‘count’ variable in memory are critical sections.
- Only **one instance** of a critical section must execute at a time on a shared resource
- Critical section bugs are the hardest to solve in OS

Typical program with critical section

```
do {
        entry section

                critical section

        exit section

                remainder section

} while (true);
```

Critical Section Solutions

The solution **must enforce all three** required conditions -

- **MUTUAL EXCLUSION –** If process Pi is executing in its critical section, then *no other processes can be executing* in their critical sections
- **PROGRESS** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next *cannot be postponed indefinitely*
- **BOUNDED WAITING** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Peterson’s Solution

- Gary L Peterson; 1981
- Original solution: Works on only 2 processes
- Space required: 2 x booleans, 1 x integer

Initial condition

```
bool flag[2] = {false, false};
int turn;
```

Critical section selection

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j) {
        // wait
    }
    // critical section
    flag[i] = false;
    //remainder section
} while (true)
```

Bakery Algorithm

- Developed by Leslie Lamport
- Scales Peterson’s Solution to N processes
- Uses tokens at a bakery counter for analogy (hence the name)

Variables required

- Given, **N** number of processes; we need the following –
- An integer array of length N (initially all 0)
    - `int number[N]`
- A Boolean array of length N (initially all false)
    - `boolean choosing[N]`

The steps for critical section entry

- When process wants to enter critical section, indicate that it is about to get a choosing number
- It then receives a number. (number[i] for P_i)
- It indicates then that it has picked up a number
- Process with smallest choosing number enters critical section
- *If two processes have same token number*, the one with lower index goes in.
- i.e., `if number[i] == number[j],` compare i and j. Lower wins

Utilities and Assumptions

- maxInArray() finds largest number in a an array
- When comparing a,b with c,d we are check –
    - Is **A < C** ? If yes, then A,B < C,D
    - If **A == C**, then is **B < D** ? If yes, then A,B < C,D

Code

```
do {
      choosing[i] = true;
      number[i] = maxInArray(number) + 1;
      choosing[i] = false;
      for (j = 0; j < n; j++) {
            while (choosing[j]); //wait
            while ((number[j] != 0) &&
                  (number[j],j < number[i],i)); //wait
      }
                  //critical section
      number[i] = 0;
                  //remainder section
} while (true)
```

Hardware based solutions

- Modern systems have hardware support for synchronizations
  - **Uniprocessors** – disables interrupts
  - **Atomics** – has atomic operation support
  - **Test and Set** – instruction support

Uniprocessor

- Currently running code executes without preemption
- Too inefficient for multitasking systems
- Not scalable beyond application specific processers

Test and Set (Hardware Atomic Instruction)

```
bool testAndSet (bool *target) {
    bool origVal = *target;
    *target = TRUE;
    return origVal;
}
```

Critical Section Entry via TestAndSet

```
do {
    while(testAndSet(&lock)); //wait

    //critical section

    lock = FALSE;

    //remainder section
} while (TRUE)
```

Swap

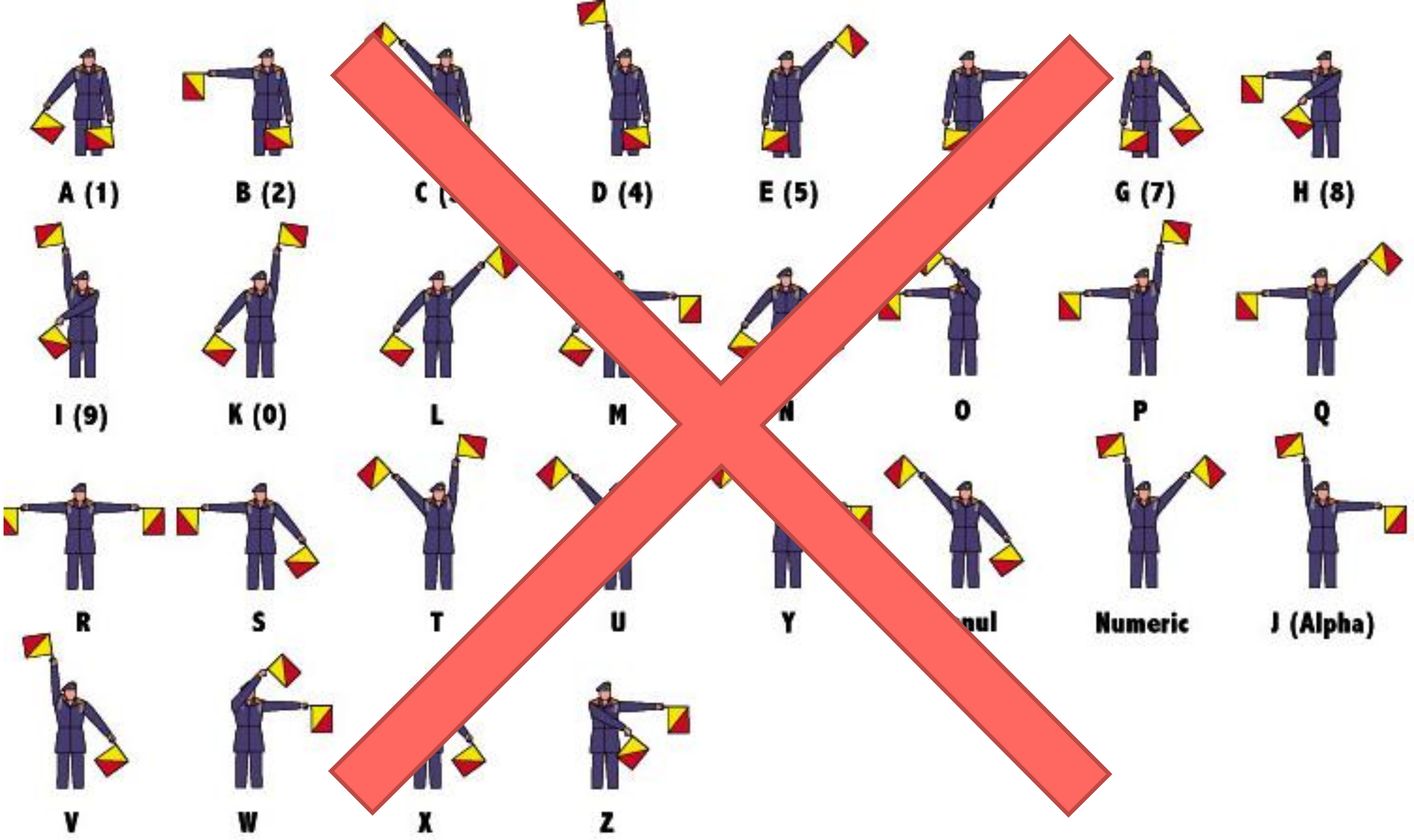
```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Critical Section Entry via Swap

```
do {
    key = TRUE;
    while (key == TRUE)
        swap(&lock, &key);

    //critical section
    lock = FALSE;
    //remainder section
} while (TRUE)
```

Semaphores

- A hand signal using flags


A (1)
B (2)
D (4)
E (5)
G (7)
H (8)
I (9)
K (0)
L
M
N
O
P
Q
R
S
T
U
Y
Numeric
J (Alpha)
V
W
X
Z

Semaphores

- ~~Magical integers that makes OS run 5x faster~~
- An integer that can be modified by ***wait()*** an ***signal()*** operations only
- Helps achieve mutual exclusion

Semaphore Operations (both atomic)

```
wait(S) {
    while (S <= 0); //do nothing
    S--;
}

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

Semaphores

- Two types –
  - counting (0 → N)
  - binary (true / false)
- Binary Semaphores == Mutex Locks
  - They provide mutual exclusion support
- Counting semaphores control access to limited instance resource.

Critical Section for N Processes

- Shared Data –
  - `semaphore mutex = 1;`
- Process P_i

```
do {
    wait(mutex);
    //critical section
    signal(mutex);
    //remainder section
} while (TRUE);
```

Producer Consumer – using Semaphores

- Shared Data –
  - `semaphore full, empty; //counting semaphores`
  - `semaphore mutex; //binary`
- Initial State
  - `full = 0`
  - `empty = N`
  - `mutex = 1`

Producer Process

```
do {
     // produce nextItem (remainder)
     wait(empty);                // reserved empty slot first
     wait(mutex);                // blocks the buffer for operation
     // add nextItem to buffer (critical)
     signal(mutex);              // free up buffer
     signal(full);               // notify that one more slot is full
} while (TRUE);
```

Consumer Process

```
do {
     wait(full);                 // stake claim to an item first
     wait(mutex);                // blocks the buffer for operation
     // pick nextItem from buffer (critical)
     signal(mutex);              // free up buffer
     signal(empty);              // notify that one more slot is empty
     // consume nextItem (remainder)
} while (TRUE);
```

The Reader-Writer Problem

- Readers read data from a file
- Multiple readers can read a single file simultaneously
- If a writer is writing to file, no reader can start reading

- Writers write data to a file
- Only one writer can be writing to a file at a time
- If 1 or more readers are reading a file, a writer cannot start writing to the file

Reader-Writer Solution, using Semaphores

- Shared data –

  -
    ```
    semaphore mutex;              // binary
    semaphore wrt;                // counting
    int readcount;
    ```
- Initial Data –

  -
    ```
    wrt = 1;
    mutex = 1;
    readcount = 0;
    ```

Writer Process

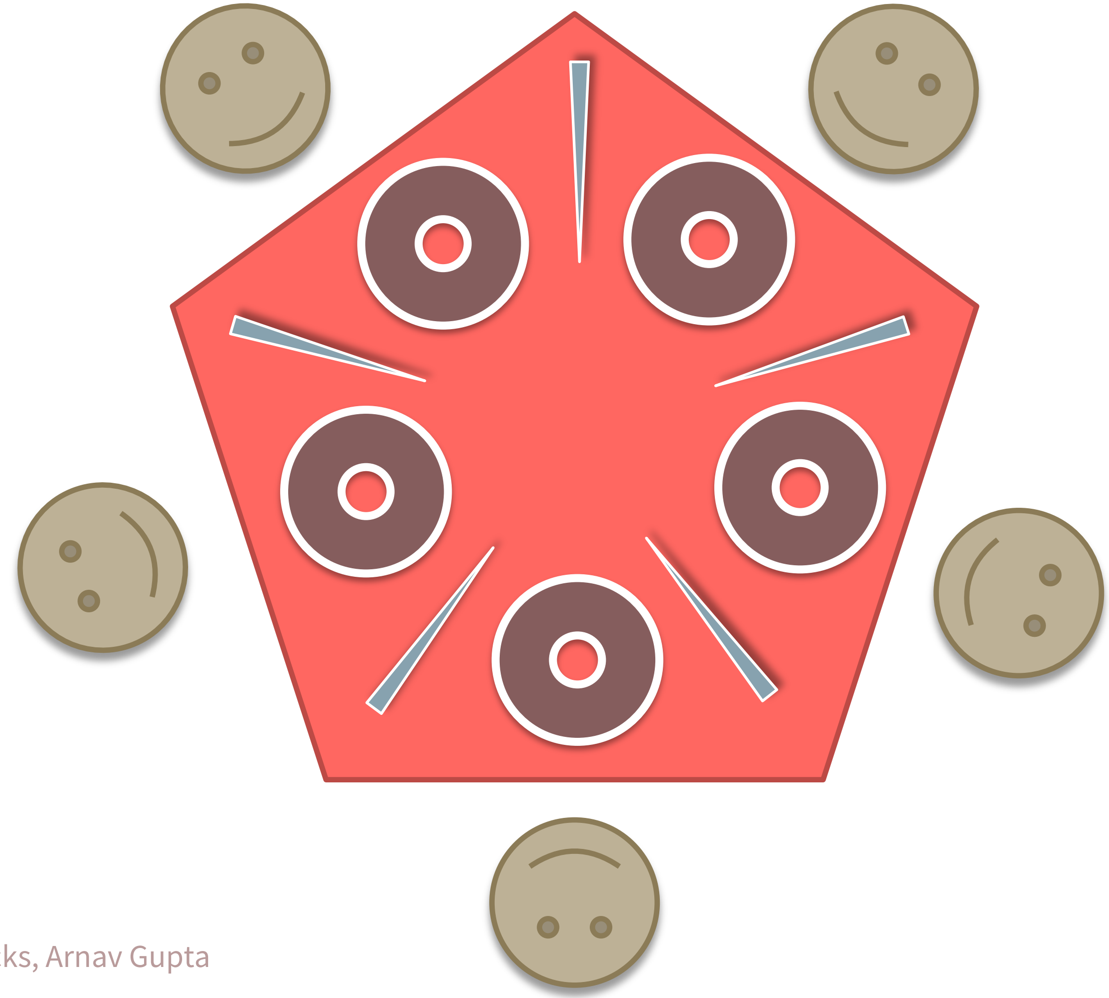
```
void write () {
    wait(wrt);          // make sure no one else is writing

    // writing performed

    signal(wrt);        // free lock so others can write
}
```

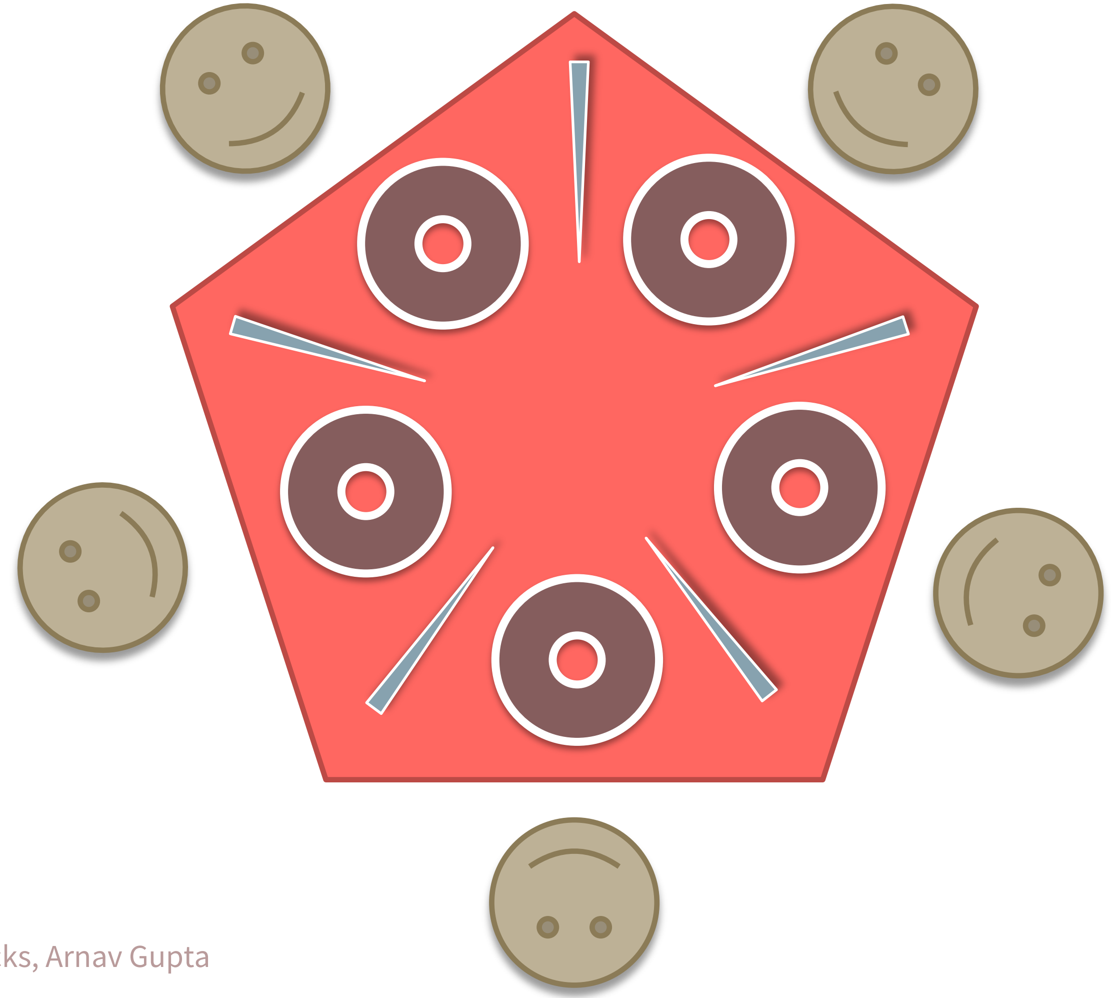
Reader Process

```
void read () {
    wait(mutex);                        // mutex guards readcount editing
    readcount++;
    if (readcount == 1) wait(wrt);      // if readers >= 1, prevent write
    signal(mutex);

    // reading performed

    wait(mutex);                        // mutex guards readcount editing
    readcount--;
    if (readcount == 0) signal(wrt);    // if no more readers, free to write
    signal(mutex);
}
```

Dining Philosophers Problem

- 5 philosophers sitting around a table
- 5 bowls of noodles in front of them (1 each)
- 5 chopsticks (alternating in between each bowl)
- Philosophers sometimes think, sometimes eat
- To eat, each philosopher needs 2 chopsticks





Dining Philosophers – Semaphore Solution

- Shared Data –
    - `semaphore chopstick[5]`
- Initially all chopstick[i] values are **1**

i^{th} Philosopher (P_i) process

```
do {
    wait(chopstick[i]);
    wait(chopstick[i+1] % 5);

    // eat (critical)

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

    //think (remainder)
} while (TRUE);
```

Discuss . . .

- What are any potential problems (if any) with the solution ?
- How can the issue be solved ?

The Cigarette-Smokers Problem

- To roll and smoke a cigarette, we need –
    - tobacco
    - matches
    - paper
- There are **3** smokers, sitting in a circle
- There is agent in the center
- Agent has infinite supply of material
- One smoker (**S1**) has paper, other (**S2**) has tobacco and last one (**S3**) has matches
- If **S1** gets ***tobacco + matches***, he can smoke. **S2** can smoke if he gets ***paper + matches*** and **S3** needs ***paper + tobacco*** to smoke
- Agent places *2 items* at a time on the table
- The smoker who has the remaining item picks these, makes cigarette and smokes
- After smoking, he signals agent, who puts another 2 items on table