CMP325
Operating Systems
Lecture 15

Synchronization using Semaphore

Fall 2021

Arif Butt (PUCIT)

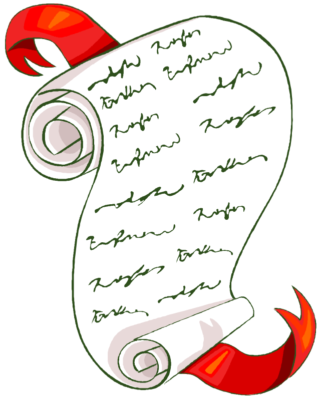
Note:

Some slides and/or pictures are adapted from course text book and Lecture slides of

- Dr Syed Mansoor Sarwar
- Dr Kubiatowicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

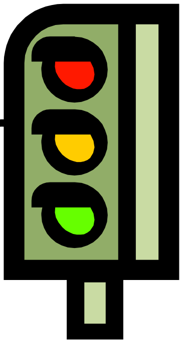
For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice video lectures on the subject of **OS with Linux** by Arif Butt available on the following link:
http://www.arifbutt.me/category/os-with-linux/

Today's Agenda



- Review of Previous Lecture
- Semaphores **(OS Based solution)**
  - Introduction to semaphores
  - Mutual Exclusion using Binary and Counting Semaphores
  - Serialization using Semaphores
  - Problems with Semaphores
  - Solving Standard Synchronization Problems using Semaphores
    - Producer Consumer Problem
    - Dinning Philosopher Problem
    - Readers Writers Problem
    - Sleeping Barber Problem
    - Smokers Problem

Introduction to Semaphores



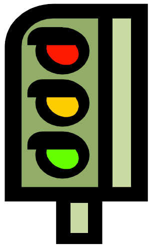
- Till date, we have been designing protocols that processes /threads can use to coordinate their activities when one wants to go to its CS
- Semaphores are a kind of generalized lock, first defined by Dijkstra in late 60s
- **Real Life Definition.** *A* semaphore is a system of signals used to communicate visually using flags lights or some other mechanism
- **CS Definition.** *A* data structure that is useful for solving a variety of synchronization problem

Introduction to Semaphores (…)

A semaphore is an integer variable, with three differences

- When you create a semaphore, you can initialize it to any integer value, but after that you can perform only two operations on it namely increment and decrement
- When a process/thread decrements the semaphore, if the result is negative, the thread blocks itself (notifies the scheduler that it cannot proceed) and cannot continue until another thread increments the semaphore
- When a process increments the semaphore, if there are other threads waiting , one of the waiting threads gets unblocked. Which one?
  - Strong semaphore
  - Weak Semaphores

Introduction to Semaphores (…)



Value of Semaphore

- **Zero** value means there are no threads waiting, but if a thread tries to decrement, it will block
- **Positive** value represents the number of threads that can decrement without blocking
- **Negative** value represents the number of threads that have blocked and are waiting

Introduction to Semaphores (…)

- Implementation of semaphores is normally available as part of Operating System's "System calls". That's why it is known as OS based solution to CSP.
- **Semaphore Initialization**
  - To achieve mutual exclusion we initialize semaphore with 1 and for synchronization it is initialized to zero.

```
semaphore    s = 1;
```

- **Semaphore Operations**
  - **signal()**. An **atomic operation** that increments the semaphore by 1 and wake up a waiting Process, if any

```
s.increment();          s.signal();          s.V();
    s.increment_and_wake_a_waiting_process_if_any();
```

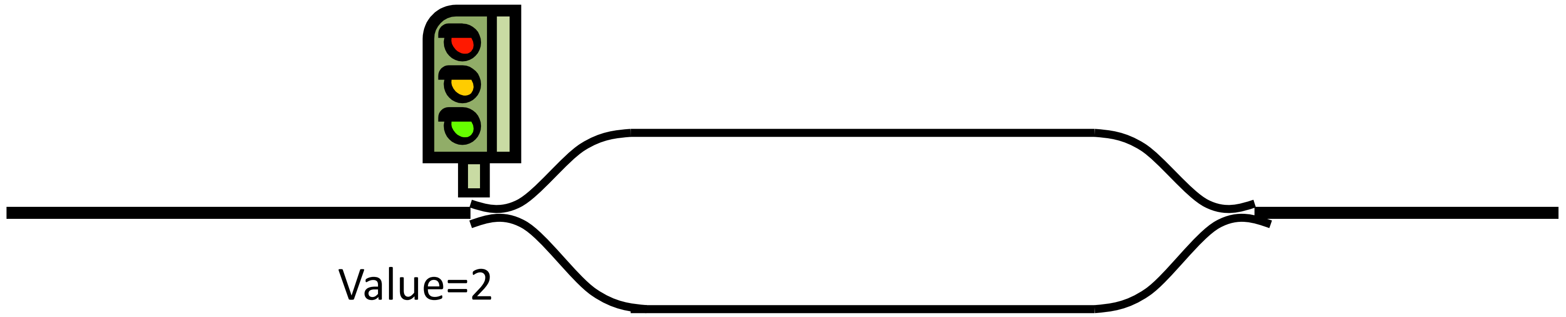
  - **wait()**. An **atomic operation** that decrements the semaphore by 1 and blocks the calling process if the result is negative.

```
s.decrement();          s.wait();          s.P();
    s.decrement_and_block_if_the_result_is_negative();
```

- Note that **P()** stands for "proberen" (to test) and **V()** stands for "verhogen" (to increment) in Dutch

Introduction to Semaphore(…)

- **Semaphore from railway analogy**
  - Here is a semaphore initialized to 2 for resource control:



N Process CSP Solution using Semaphores

Binary Semaphores.

- Mutex will either be 0 (means there is a process inside its CS)
- Mutex will either be 1 (means no process is inside its CS.
- Mutex will never be negative or greater than 1

```
semaphore mutex = 1;

            Process Pi

do {
     wait (mutex) ;
   <CS>
   signal (mutex);
   <RS>
} while (1);
```

```
//entry to CS of a process is
controlled by wait operation

wait(semaphore s) {
    while(s<=0) ;
        s--;
}
//exit from CS is signaled by signal
operation

signal(semaphore s){
    s++;
}
```

Atomic Execution of wait and signal operation

- **wait()** and **signal()** operations should be indivisible /executed atomically; i.e. when one process modifies the semaphore value, no other process can simultaneously modify that semaphore value.
- **wait()** and **signal()** are not machine instructions, Operating Systems ensure that they are executed atomically. HOW?
  - In a uni-processor environment, disable interrupts (MI bit) before executing code for wait() and signal().
  - In a bus based multiprocessor environment, lock the data bus before executing wait and signal and release the bus afterwards.
  - Since wait() and signal() are just like CSs, so another option is to use the software solution, i.e. put the code of Bakery algorithm before the wait and signal operation.

New Definition of Semaphores

Semaphore is not a simple integer variable, rather is an integer variable with a list of processes associated with it.

```
typedef struct{
    int value;
    struct process * L;
}semaphore;
```

Two simple ops are used:

- **block()** is a system call to kernel that will place the calling process in a waiting queue.
- **wakeup(P)** is a system call to kernel that will remove the process P from the waiting queue and place it in the ready queue.

```
void wait(semaphore s) {
    s.value--;
    if(s.value < 0) {
        add caller process to list L;
        block();
    }
}
```

```
void signal(semaphore s) {
    s.value++;
    if(s.value <= 0) {
        remove process P from list;
        wakeup(P);
    }
}
```

Counting Semaphores

- Binary semaphores allows only one process at a time to access the shared resource.
- Counting semaphores allows N > 1 processes to access the resource simultaneously.
- Lets implement a counting semaphore S using two binary semaphores s_1 and s_2.

```
counting semaphore S = 5; //max 5 processes can enter into a particular piece of code
int c = 5; //c will contain the current value of S at any instant of time
semaphore s1 = 1; //A b.s used to get hold of c mutually exclusively
semaphore s2 = 0; //A b.s used to achieve synchronization. A process Pi waiting on S
will actually be waiting on b.s s2
```

```
void wait(semaphore S) {
    wait(s1);
    c--;
    if (c < 0) {
        signal(s1);
        wait(s2);   }
     else
     signal(s1);
}
```

```
void signal(semaphore S) {
    wait(s1);
    c++;
    if (c <= 0)
        signal(s2);
    signal(s1);
}
```

Uses of Semaphores

- To ensure Mutual Exclusion of a Critical Section (as locks)
- To control access to a shared pool of resources (using counting semaphores)
- To cause a thread/process to wait for a specific action to be signaled by another thread/process (serialization)

Synchronization using Semaphores

- Semaphores provide a powerful OS tool that are used to achieve synchronization between processes, other than achieving mutual exclusion.
- **Example 1**. Consider two processes P_i and P_j with statements A and B in them respectively. We want that statement B in P_j should be executed after statement A in P_i is executed. Give a semaphore based solution.

```
P_i                 P_j
 ⋮                   ⋮
A                   wait(s);
signal(s);          B
 ⋮                   ⋮
```

- **Example 2**. Consider three processes P_1, P_2 and P_3. Instruction A in P_1 executes after instruction B in P_2 is executed. Instruction B in P_2 executes after instruction C in P3 has executed. Give a semaphore based solution. $(C < B < A)$

Problems with Semaphores

- Semaphores provide a powerful synchronization tool, but:
  - wait() and signal() operations are scattered among several processes. It is difficult to understand their effects
  - Usage must be correct in all the processes
  - One bad process (i.e. one programming error) can kill whole system
- Wrong initialization or placement of wait and signal may cause following problems:
  - Violation of M.E
  - Dead Lock
  - Starvation

Problems with Semaphores (…)

- **Violation of Mutual Exclusion.**
  - By mistake the programmer has placed the signal operation before the wait operation in P_0.
  - S is initialized to 1.
  - Let P1 executes first, decrements s to 0 and enter its CS.
  - Let P0 now executes, and instead of a wait gives a signal to s, increments s to 1 and enter it Cs.
  - Both p0 and p1 are in Cs (M.E violated)

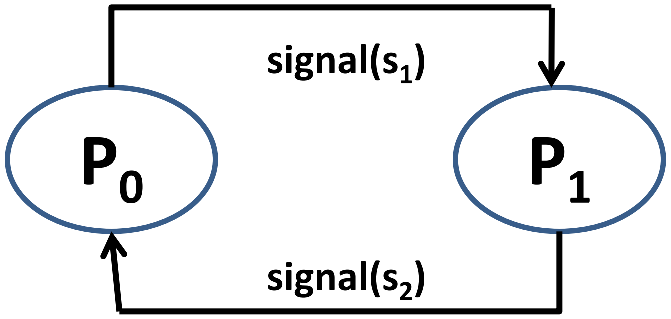
```
P0                P1
signal(s);        wait(s);

<CS>              <CS>
wait(s);          signal(s);
⋮                 ⋮
```

Problems with Semaphores (…)

- **Dead Lock.**
  - A set of processes are said to be in a state of dead lock, if every process is waiting for an event that can be caused only by another process in the set.

```
P0                        P1
wait(s1);                 wait(s2);
wait(s2);                 wait(s1);
⋮                         ⋮
signal(s1);               signal(s2);
signal(s2);               signal(s1);
⋮                         ⋮
```



Problems with Semaphores (…)

- **Starvation.**
  - Indefinite blocking due to unavailability of a resource

```
P0                      P1
wait(s);                wait(s);
⋮                       ⋮
nothing/wait(s);        signal(s);
⋮                       ⋮
```

Solution of Wrong use of Semaphores

- **Problem.** Using semaphores for achieving serialization and M.E is error prone due to tandem / wrong use of wait and signal operations by programmer.
- **Solution.** Use high level language constructs, i.e. shift the responsibility of enforcing M.E / serialization form the programmer to the compiler. These constructs will be available in compilers and will release the application programmer of the hustle of using semaphores to achieve synchronization. Examples of such constructs are:
  - Critical Regions
  - Monitors

Producer Consumer (Unbounded Buffer)

```
binary semaphore mutex = 1; //To access buffer mutually exclusively
counting semaphore full = 0; //counts the no of slots that are full
int in = 0, out = 0;
```

Producer

```
do{
    item = produceItem();
    wait(mutex);
    placeItem(buffer,item);
    signal(mutex);
    signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    item = takeItem(buffer);
    signal(mutex);
    consumeItem(item);
}while(1);
```

```
void placeItem(buffer, item){
   buffer[in] = item;
   in = in + 1;}
item takeItem(buffer){
   item = buffer[out];
   out = out + 1;
   return item;}
```

Producer Consumer (Bounded Buffer)

```
#define N 5
item buffer[N];
binary semaphore mutex = 1; //To access buffer mutually exclusively
counting semaphore full = 0; //counts the no of slots that are full
counting semaphore empty = N; //counts the no of slots that are empty
int in = 0, out = 0;
```

Producer

```
do{
    item = produceItem();
    wait(empty);
    wait(mutex);
    placeItem(buffer,item);
    signal(mutex);
    signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    item = takeItem(buffer);
    signal(mutex);
    signal(empty);
    consumeItem(item);
}while(1);
```

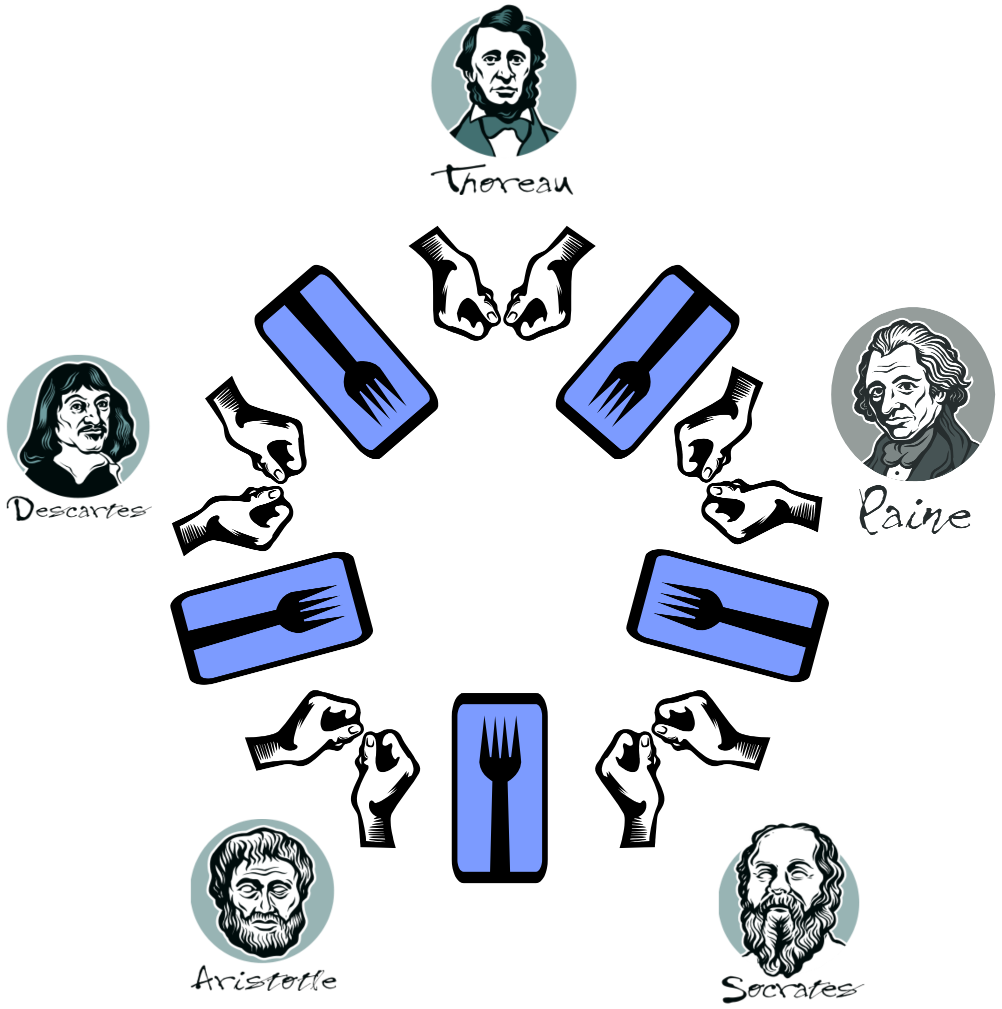
```
void placeItem(buffer, item){
   buffer[in] = item;
   in = (in + 1) % N;}
item takeItem(buffer){
   item = buffer[out];
   out = (out + 1) % N;
   return item;}
```

Dining Philosopher Problem

- Five Chinese philosophers, who spend their lives just thinking and eating
- Sit on a round table with five plates of rice and five chopsticks
- A philosopher requires two chopsticks to eat (so at a time a maximum of two philosophers can eat)
- **Protocol used for eating:**
  - Picks up left chopstick and then right chopstick, one at a time in either sequence
  - If successful in acquiring two chopsticks, the philosopher eats for a while, then puts down the chopstick and continues to think
- One fine day all became hungry at a time. All pick up the left chopstick first and then look for the right chopstick, which was not there. They did not fight like us but waited and waited and waited and finally starved to death. Sad day in China….

"Allocate several resources among processes in deadlock free, starvation free manner"

Dining Philosopher Problem



Dining Philosopher Problem

Lets code the protocol using which Chinese Philosopher starved to death

```
semaphore chopstick[5];  //all initialized to 1
                    Pi
do{
   think();
   wait(chopstick[i]);
   wait(chopstick[(i+1)%5]);
   eat;
   signal(chopstick[(i+1)%5]);
   signal(chopstick[i]);

}while(1);
```

Limitation: *All* philosophers start simultaneously, picking up their left chopstick; look for the right, which is not available, so started waiting and finally starved to death

Dining Philosopher Problem (Solution-A)

```
semaphore chopstick[5];  //all initialized to 1
binary semaphore mutex = 1;
```

P_i

```
do{
    think();
    wait(mutex);
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);
    eat;
    signal(chopstick[(i+1)%5]);
    signal(chopstick[i]);
    signal(mutex);
}while(1);
```

Dining Philosopher Problem (Solution-B)

```
semaphore chopstick[5];  //all initialized to 1
counting semaphore table = 4;
                              Pi

do{
    think();
    wait(table);
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);
    eat;
    signal(chopstick[(i+1)%5]);
    signal(chopstick[i]);
    signal(table);
}while(1);
```

Dining Philosopher Problem (Solution-C)

```
semaphore chopstick[5];  //all initialized to 1
                         Pi
do{
   think();
   if ((i % 2) == 1){
       wait(chopstick[(i+1)%5]);
       wait(chopstick[i]);
   }
   else
       wait(chopstick[i]);
       wait(chopstick[(i+1)%5]);
   }
   EAT;
   signal(chopstick[(i+1)%5]);
   signal(chopstick[i]);

}while(1);
```

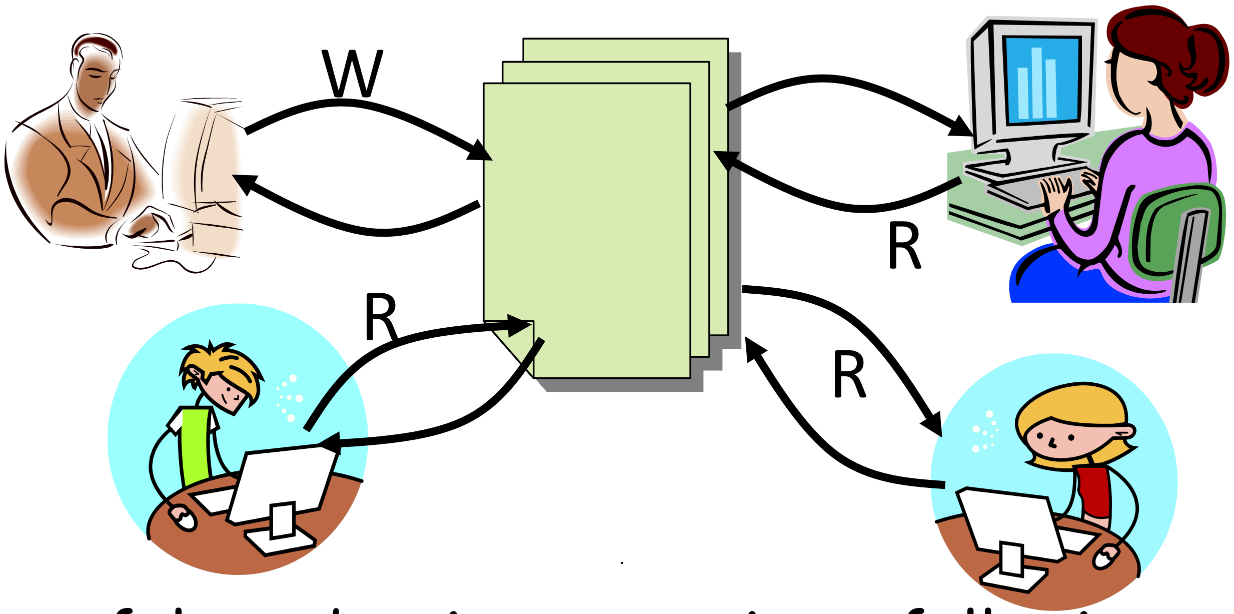
Dining Philosopher Problem (Solution-D)

```
semaphore chopstick[5];  //all initialized to 1
boolean flag[5]; //initialized to false; i.e. all are available
```

P$_i$

```
do{
   if(!(flag[i] OR flag[(i+1)%5])){
      flag[i] = true;
      flag(i+1)%5] = true;
      wait(chopstick[i]);
      wait(chopstick[(i+1)%5]);
      EAT;
      signal(chopstick[(i+1)%5]);
      flag(i+1)%5] = false;
      signal(chopstick[i]);
      flag[i] = false;
   }
   else
       think();
}while(1);
```

Reader Writer Problem



- For successful read-write operations, following conditions must be satisfied:
  - Two or more readers can access shared data simultaneously
  - Only one writer can access it at a time
  - If a writer is writing to the file, no reader may read it

Reader Writer Problem

Readers have Priority

- If one or more readers are reading a shared resource and some other readers and writers also want to access that shared resource; we will let the readers read and writers wait until there is no reader reading the shared resource
- If a reader want to read, it wait for a minimum amount of time

Writers have Priority

- If one or more readers are reading a shared resource and some other readers and writers also want to access that shared resource; we will NOT let any further readers to come in and read, rather let the old readers finish reading and let a writer write
- If a writer wants to write, it waits for minimum amount of time

Readers have priority

```
int readCount = 0;
binary semaphore mutex = 1; //To access readCount mutually exclusively
binary semaphore wrt  = 1; //Used by writer process so that only
//one writer process can enter the writing phase at a time
```

Writer

```
do{
   wait(wrt);
   WRITING IS PERFORMED
   signal(wrt);
}while(1);
```

Reader

```
do{
   wait(mutex);
   readCount++;
   if (readCount == 1)
       wait(wrt);
   signal(mutex);
   READING IS PERFORMED
   wait(mutex);
   readCount--;
   if (readCount == 0)
       signal(wrt);
   signal(mutex);
}while(1);
```

Writers have priority

```
int readCount, writeCount = 0;
binary semaphore x,y,z = 1; //y,x is for writecount,readcount
binary semaphore wrt, read  = 1;
```

Writer

```
do{
   wait(y);
   writeCount++;
   if (writeCount==1)
      wait(read);
   signal(y);
   wait(write);
   <DO WRITING>;
   signal(write);
   wait(y);
   writeCount--;
   if (writeCount==0)
      signal(read);
   signal(y);
}while(1);
```

Reader

```
do{
   wait(z);
   wait(read);
   wait(x);
   readCount++;
   if (readCount==1)
      wait(write);
   signal(x);
   signal(read);
   signal(z);
   <DO READING>;
   wait(x);
   readcount--;
   if (readcount==0)
      signal(write);
   signal(x);
}while(1);
```

Sleeping Barber Problem

- A barber shop consists of a room with **n** waiting chairs and **one** barber chair
    - If there are no customers to be served the barber goes to sleep
    - If a customer arrives and the barber is asleep, the customer wakes up the barber
    - If the barber is busy, but chairs are available, then the customer sits on one of the free chairs
    - If a customer enters the barber shop and all chairs are occupied, then the customer leaves the shop

Customer

- Check if chair available, if not leave
- Inform barber that I have arrived
- Wait until barber cuts his hair

Barber

- Sleep until a cust wakes him up
- Service cust (during that remember to update available chairs
- Tell cust to leave after finished
- Repeat for other customers

Sleeping Barber Problem (Solution)

```
const int N = 5
int chairs_occupied = 0;
binary semaphore mutex = 1; //To access chairs_occupied mutually exclusively
counting semaphore barber_finished = 0;
counting semaphore cust_arrived = 0;
```

Customer

```
do{
   wait(mutex);
   if (chairs_occuupied < N)
       chairs_occupied ++;
   else
   {
       signal(mutex);
       exit(0);
   }
   signal(mutex);
   signal(cust_arrived);
   wait(barber_finished);
}while(1);
```

Barber

```
do{
   wait(cust_arrived);
   wait(mutex);
   chairs_occupied --;
   signal(mutex);
   cut_hair();
   signal(barber_finished);
}while(1);
```

Smokers Problem

- **Ingredients (Resources)** Tobacco, Paper, Match Box.
- **3 x Smokers (Applications)** sits around a table. Each one has one of the three ingredients/resources (tobacco, paper, match box)
- **1 x Agent (Operating System)** has infinite supply of all three materials.
- **Protocol**
  - Agent places two of the ingredients (at random) on the table.
  - The smoker who has the remaining third ingredient makes and smokes the cigarette.
  - After smoking/completion signals the agent.
  - The agent then puts out another two of the three ingredients and cycle repeats.

Smokers Problem (…)

```
binary semaphore tobacco = 0;
binary semaphore match = 0;
binary semaphore paper = 0;
binary semaphore agent = 1;
```

Smoker with match box

```
do{
    wait(tobacco);
    wait(paper);
    SMOKE
    signal(agent);
}
```

Smoker with tobacco

```
do{
    wait(paper);
    wait(match);
    SMOKE
    signal(agent);
}
```

Smoker with paper

```
do{
    wait(tobacco);
    wait(match);
    SMOKE
    signal(agent);
}
```

Smokers Problem

Agent Process

```
do{
wait(agentSem);
//Pick two ingredients, if ingredients are tobacco and paper then
signal(tobacco);
signal(paper);
// else if ingredients are paper and match then
signal(paper);
signal(match);
// else if ingredients are tobacco and match then
signal(tobacco);
signal(match);
}
```

SUMMARY

We're done for now, but Todo's for you after this lecture…

- Go through the slides and Book Sections: 6.5, 6.6
- Google the standard synchronization problems discussed in the slides and understand the pros and cons of the pseudo code/algorithm.
- Google out the system calls available in POSIX and System-V for the use of semaphores to achieve synchronization