Concurrency

- Semaphores, Condition Variables, Producer Consumer Problem

Kartik Gopalan

Chapters 2 (2.3) and 6
Tanenbaum's Modern OS

Semaphore

- Semaphore is a fundamental synchronization primitive used for
  - Locking around critical regions
  - Inter-process synchronization

- A semaphore “sem” is a special integer on which only two operations can be performed.
  - DOWN(sem)
  - UP(sem)

The DOWN(sem) Operation

- If (sem > 0) then
  - Decrements sem by 1
  - The caller continues executing.
  - This is a "successful" down operation.

- If (sem == 0) then
  - Block the caller
  - The caller blocks until another process calls an UP.
  - The blocked process wakes up and tries DOWN again.
  - If it succeeds, then it moves to "ready" state
  - Otherwise it is blocked again till someone calls UP.
  - And so on.

The UP(sem) Operation

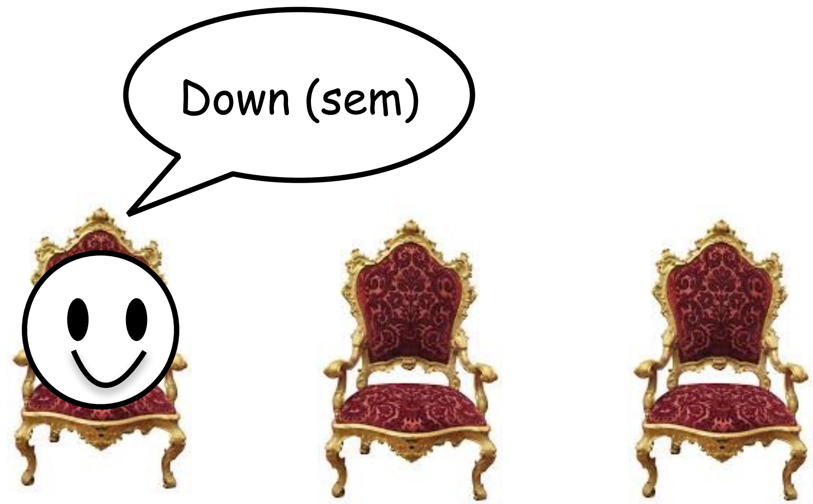
- This operation increments the semaphore sem by 1.
- If the original value of the semaphore was 0, then UP operation wakes up any process that was sleeping on the DOWN(sem) operation.
- All woken up processes compete to perform DOWN(sem) again.
  - Only one of them succeeds and the rest are blocked again.

Semaphore Example — “Chair is taken”

sem = 3

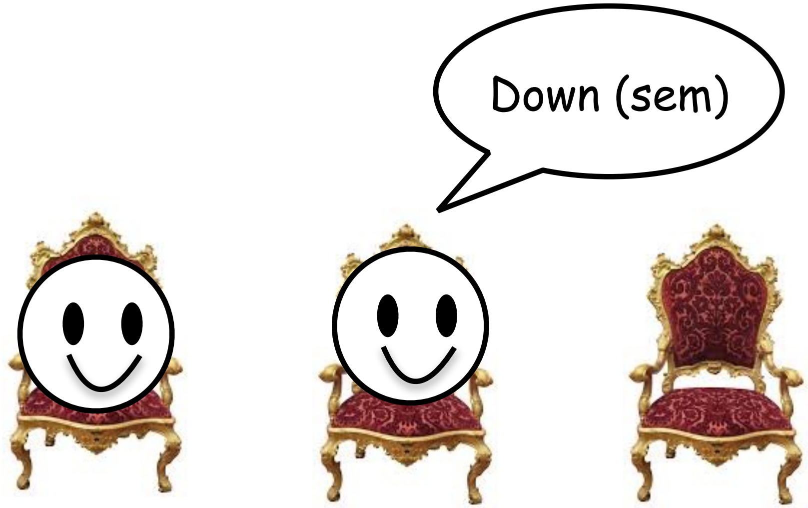
Semaphore Example — “Chair is taken”

sem = 2



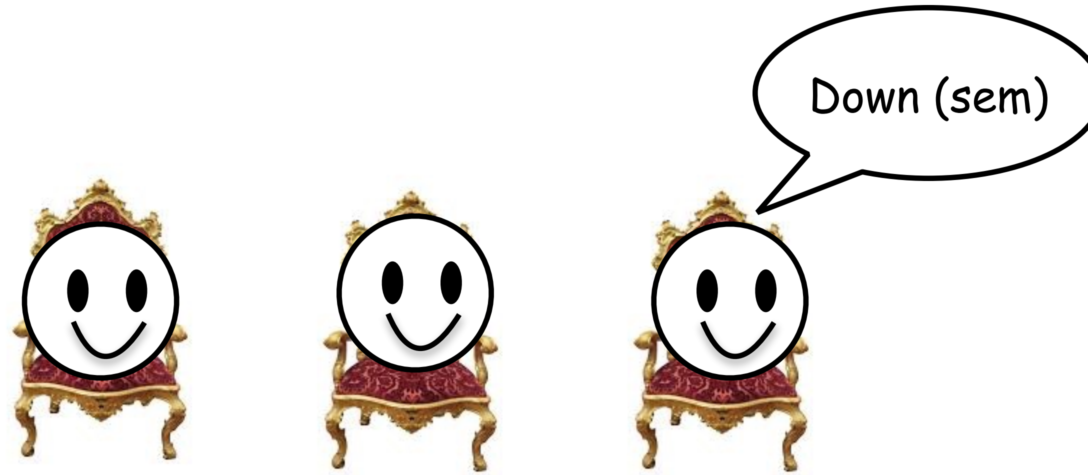
Semaphore Example — “Chair is taken”

sem = 1



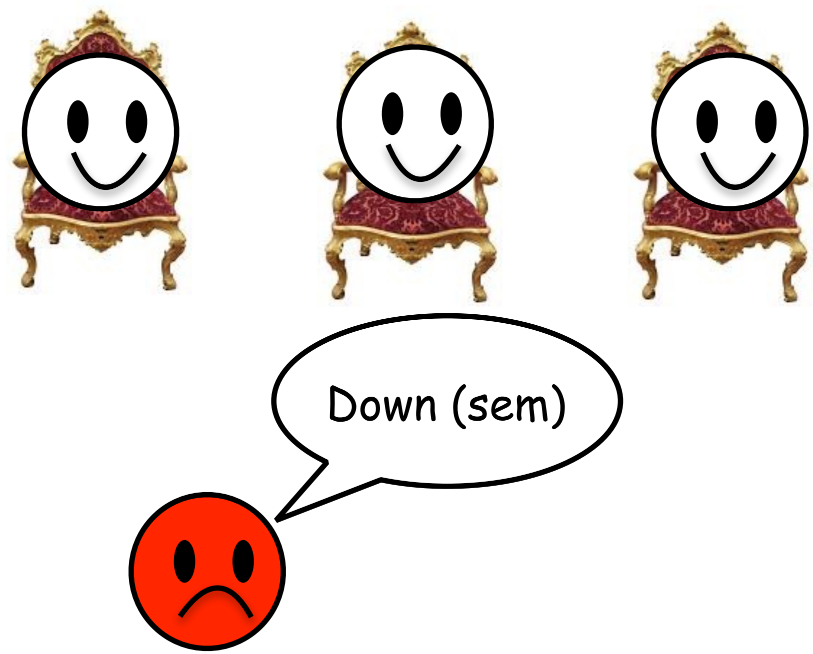
Semaphore Example — “Chair is taken”

sem = 0



Semaphore Example — “Chair is taken”

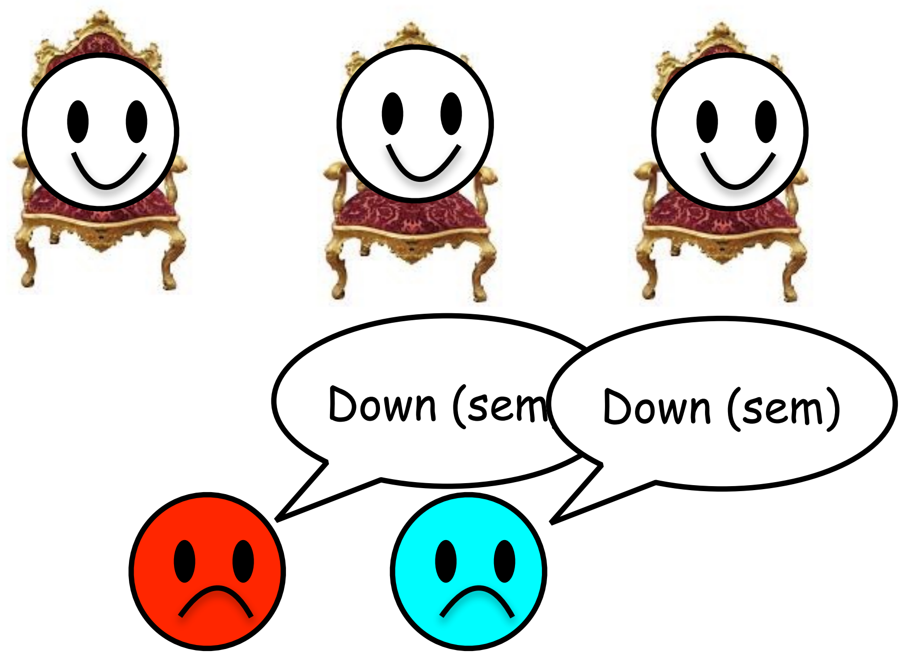
sem = 0



Blocked

Semaphore Example — “Chair is taken”

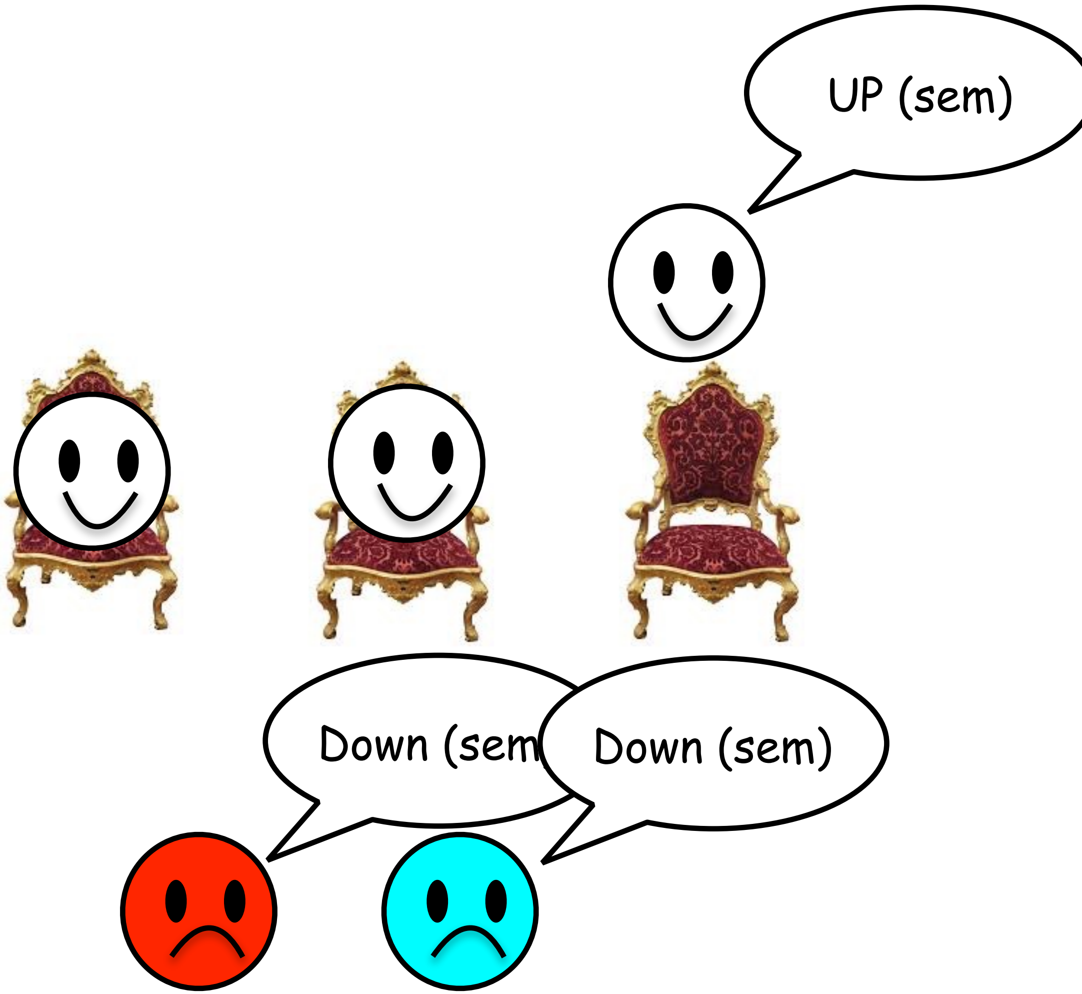
sem = 0



Blocked

Semaphore Example — “Chair is taken”

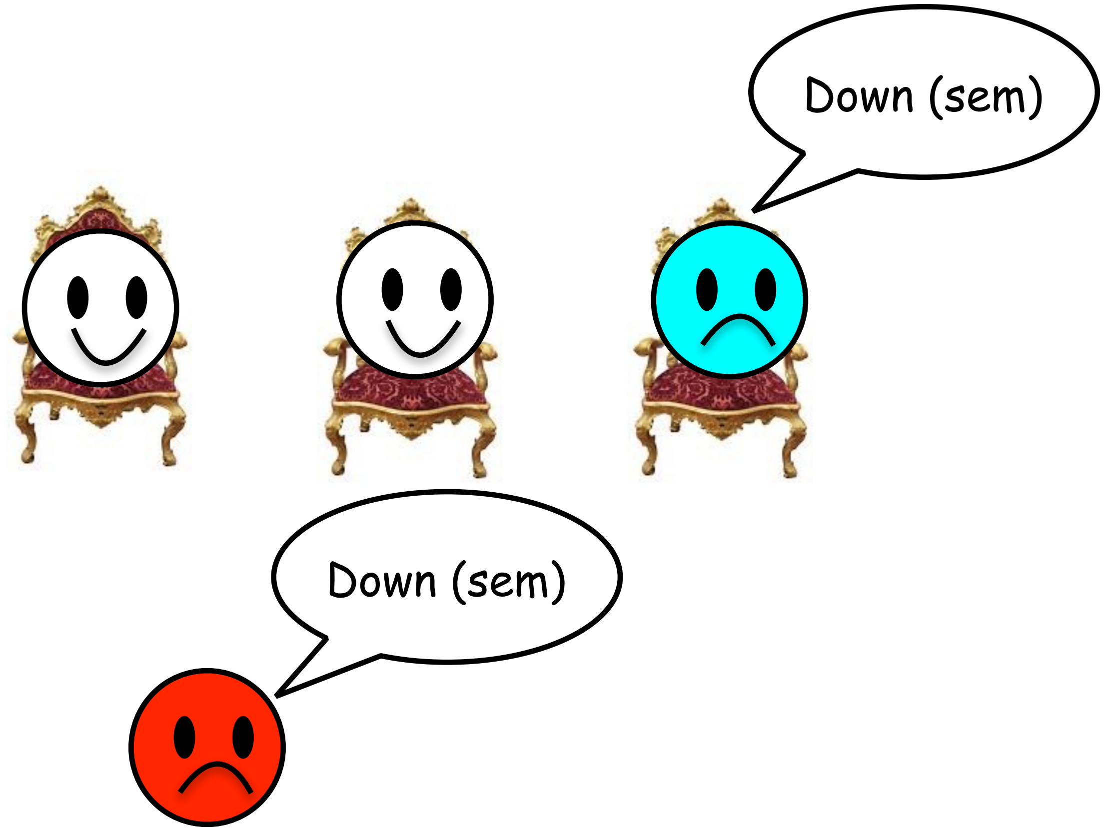
sem = 1



Blocked

Semaphore Example — “Chair is taken”

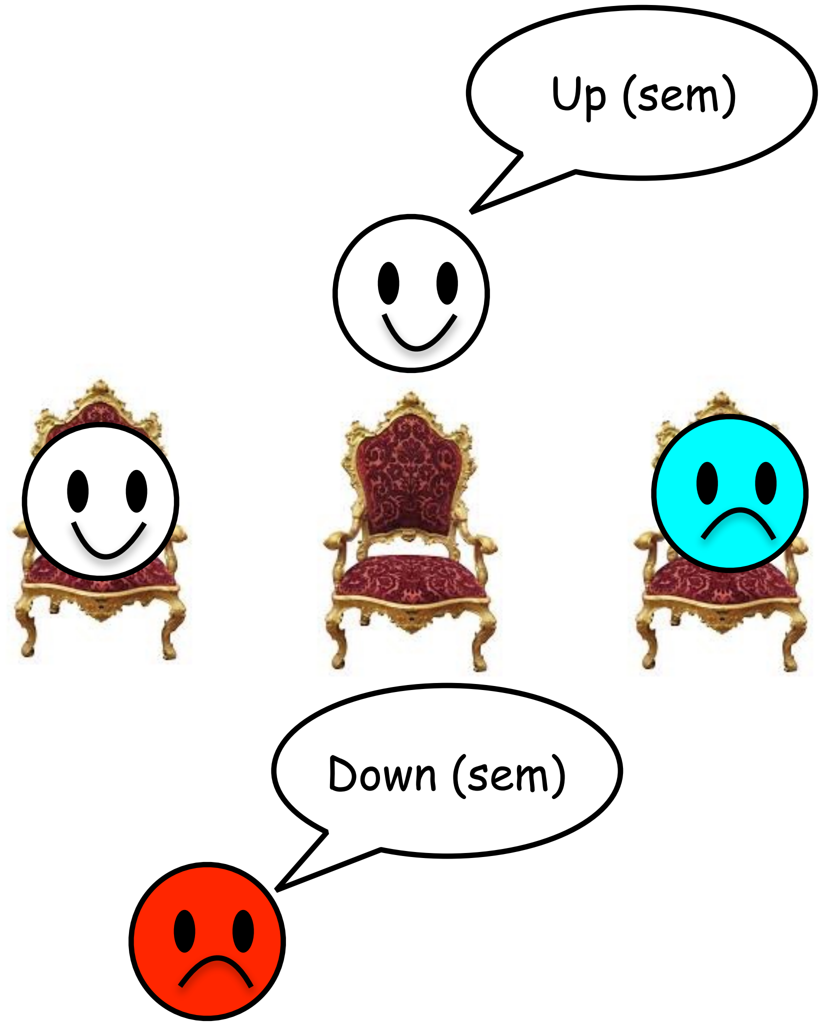
sem = 0



Blocked

Semaphore Example — “Chair is taken”

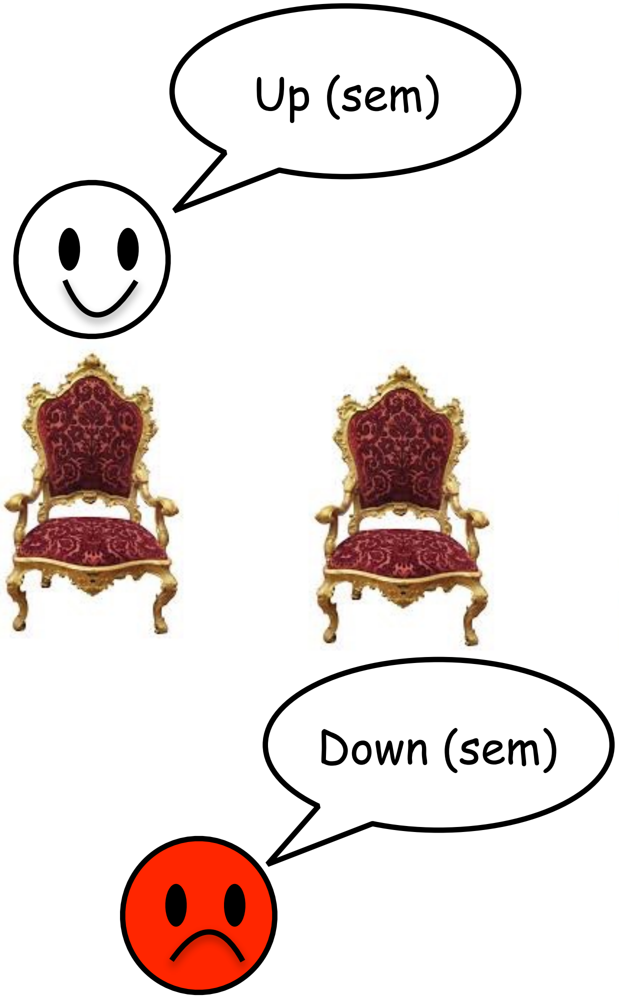
sem = 1



Blocked

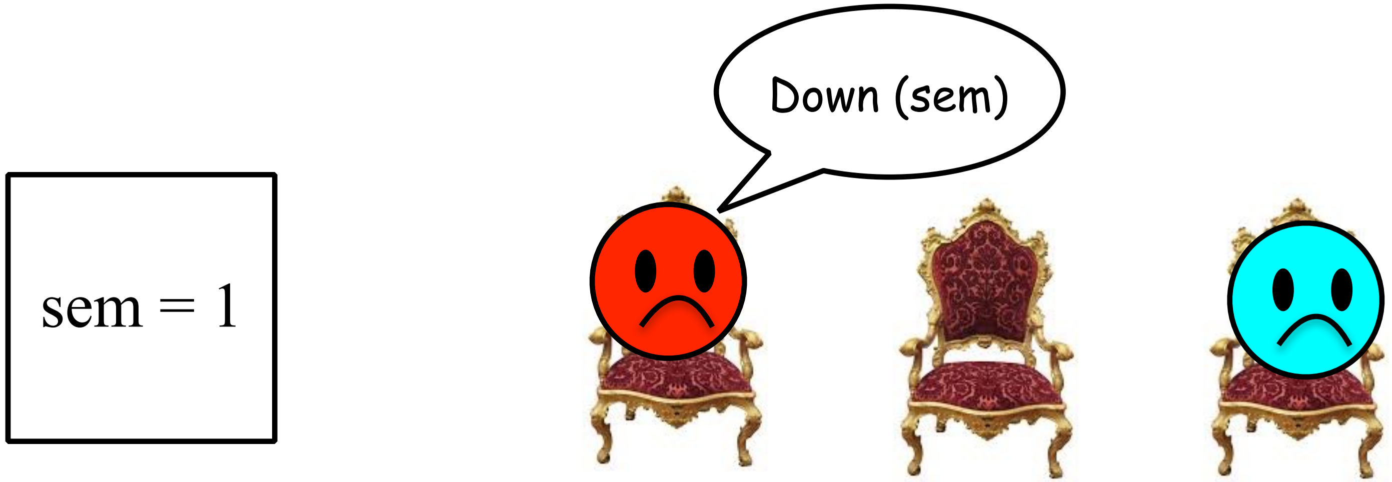
Semaphore Example — "Chair is taken"

sem = 2



Blocked

Semaphore Example — “Chair is taken”



Mutex

- Mutex is simply a binary semaphore
  - It can have a value of either 0 or 1

- Mutex is used as a LOCK around critical sections

- Locking a mutex means calling Down(mutex)
  - If mutex==1, decrement mutex value to 0
  - Else, sleep until someone performs an UP

- Unlocking a semaphore means calling UP(mutex)
  - Increment mutex value to 1
  - Wake up all sleepers on DOWN(mutex)
  - Only one sleeper succeeds in acquiring the mutex. Rest go back to sleep.

- For example:

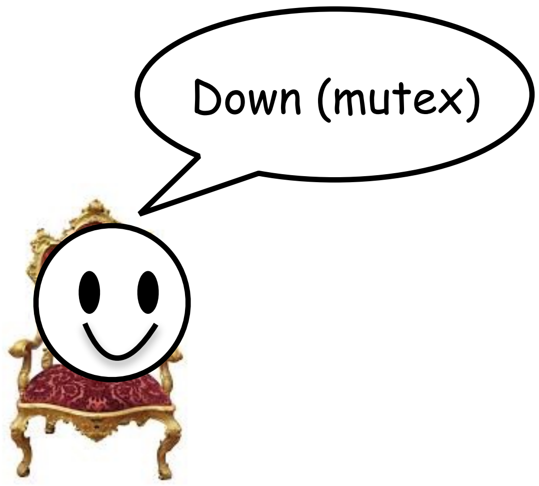
```
Down(mutex) // Acquire the lock, sleep if mutex is 0
Critical Section…
Up(mutex) // release the lock, wake up sleepers
```

Mutex Example — “Chair is taken”

mutex = 1

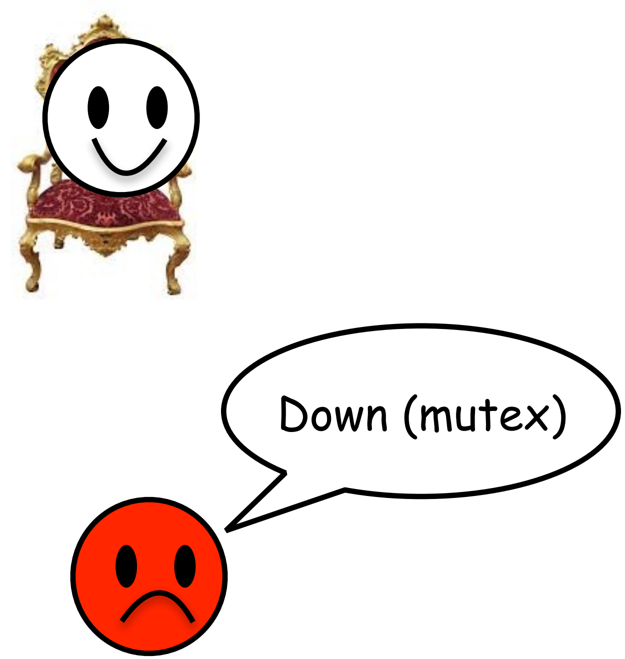
Mutex Example — “Chair is taken”

mutex = 0



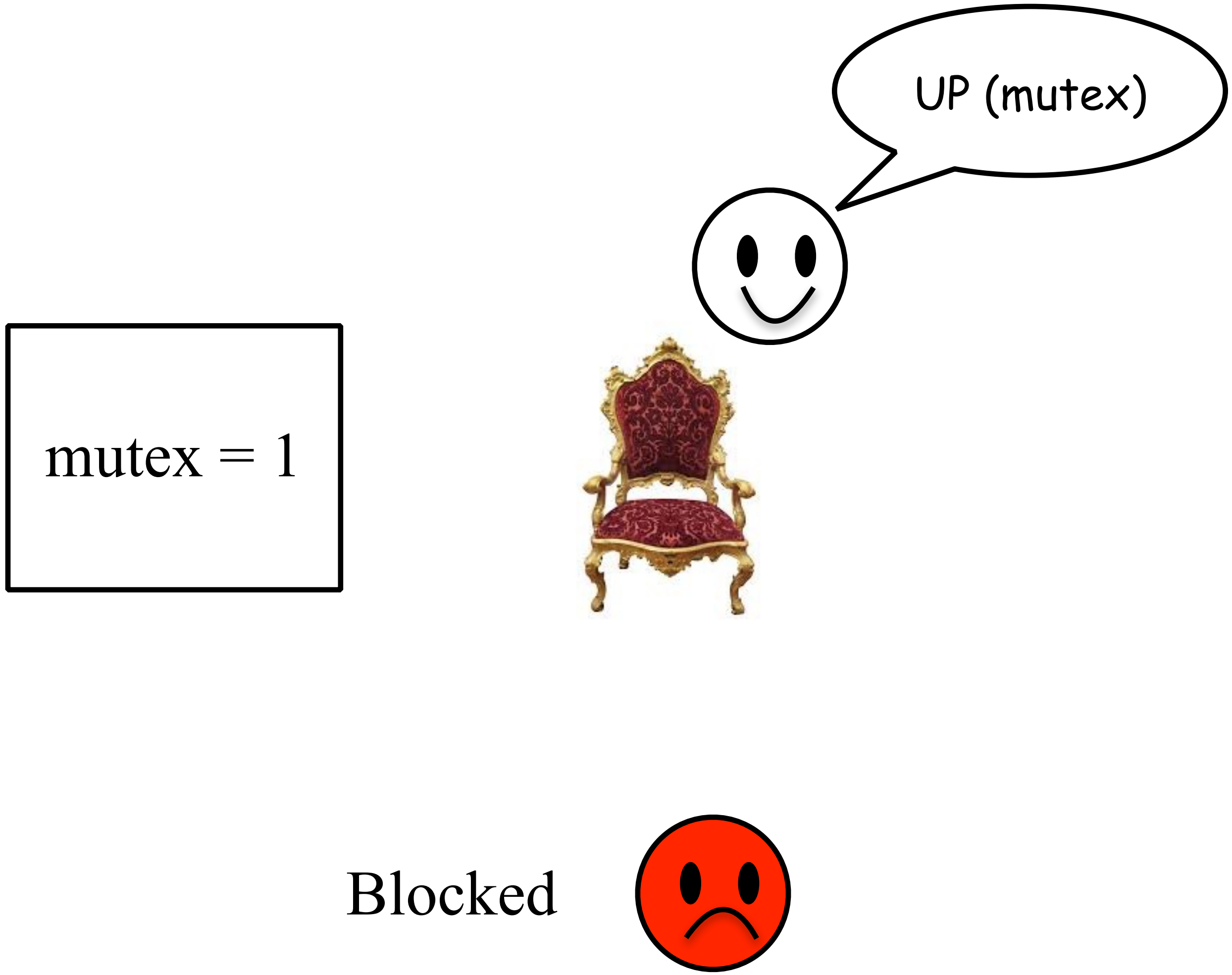
Mutex Example — “Chair is taken”

mutex = 0



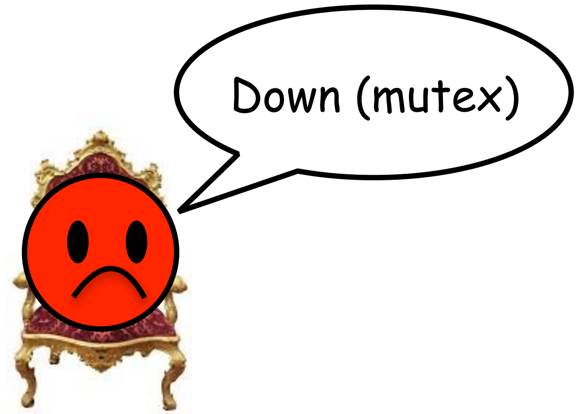
Blocked

Mutex Example — “Chair is taken”

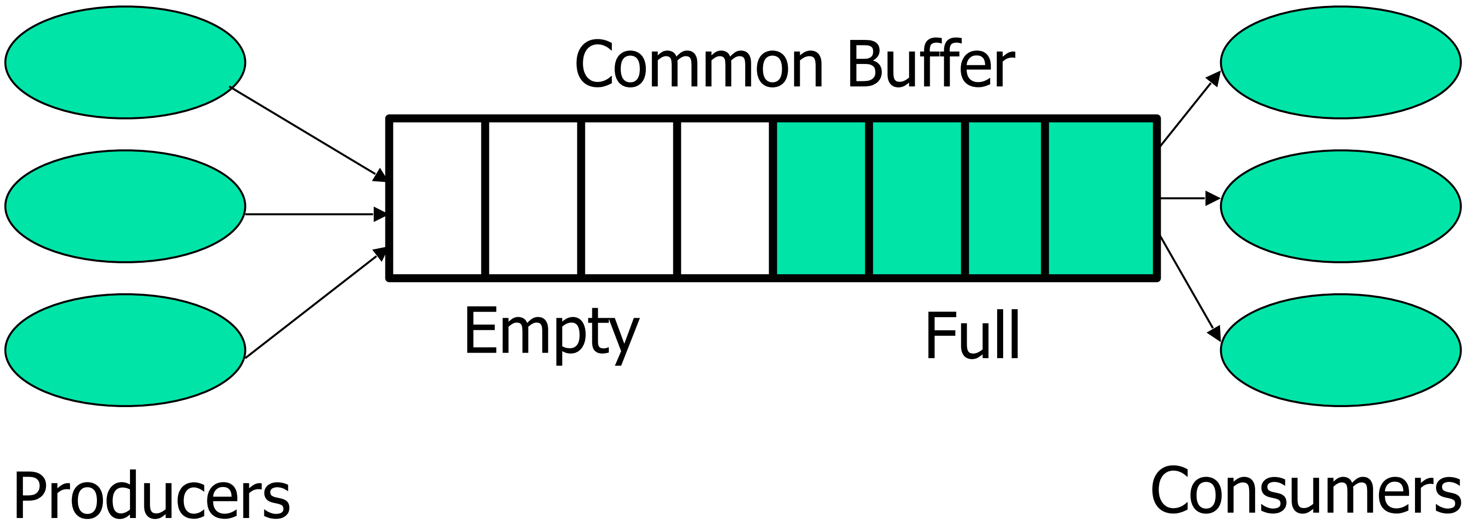


Mutex Example — “Chair is taken”

mutex = 0



Example: Producer-Consumer Problem



- Producers and consumers run in concurrent processes.
- Producers produce data and consumers consume data.
- Producer informs consumers when data is available
- Consumer informs producers when a buffer is empty.
- Two types of synchronization needed
  - Locking the buffer to prevent concurrent modification
  - Informing the other side that data/buffer is available

Using Semaphores for the P-C problem

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

Note: Two types of semaphores used here.
A binary semaphore to lock the queue (mutex)
Regular sems to block on event (empty and full).

Up: Increments the value of semaphore, wakes up sleepers to compete on sem
Down: Decrements semaphore, but blocks the caller if sem value is 0

Using Semaphores – POSIX interface

- sem_open() -- Connects to, and optionally creates, a named semaphore
- sem_init() -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).
- sem_wait(), sem_trywait() -- Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process.
- sem_post() -- Increments the count of the semaphore.
- sem_close() -- Ends the connection to an open semaphore.
- sem_unlink() -- Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it.
- sem_destroy() -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).
- sem_getvalue() -- Copies the value of the semaphore into the specified integer.
- Semaphore overview : Do “man sem_overview” on any linux machine

Another way for using Semaphores - System V interface

- Creation
  - int semget(key_t key, int nsems, int semflg);
  - Sets sem values to zero.

- Initialization (NOT atomic with creation!)

```
union semun arg;
arg.val = 1;
if (semctl(semid, 0, SETVAL, arg) == -1) {
    perror("semctl"); exit(1);
}
```

- Incr/Decr/Test-and-set
  - int semop(int semid ,struct sembuf *sops, unsigned int nsops);
- Deletion
  - `semctl(semid, 0, IPC_RMID, 0);`

Examples:

seminit.c

semdemo.c

semrm.c

Monitors and Condition Variables

Monitors and condition variables

```
monitor example
    integer i;
    condition c;

    procedure Function1()
    .
    .        wait(c);
    .
    end;

    procedure Function2()
    .
    .        signal(c);
    .
    end;
end monitor;
```

- Monitor is a collection of critical section procedures (functions)
  - i.e. functions that operate on shared resources
- There's one global lock on all procedures in the monitor.
  - Only one procedure can be executed at any time
- wait(c) : releases the lock on monitor and puts the calling process to sleep.
  ALSO: Automatically re-acquires the lock upon return from wait(c).
- signal(c): wakes up all the processes sleeping on c; the woken processes then compete to obtain lock on the monitor.

P-C problem with monitors and condition variables

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;
    count := 0;
end monitor;
```

Atomic Locking – TSL Instruction

Test-and-Set Lock (TSL) Instruction

- Instruction format: **TSL Register, Lock**

- Lock
  - Located in memory.
  - Has a value of 0 or 1

- Register
  - One of CPU registers

- TSL does the following two operations **atomically** (as one step)
  1. Register := Lock; // Copy the old value of Lock to Register
  2. Lock := 1; // Set the new value of Lock to 1

- **Atomic**: means that the caller cannot be preempted between the two operations

- TSL is a basic primitive using which other more complex locking mechanisms can be implemented.

Implementation of Mutex Using TSL

```
mutex_lock:
      TSL REGISTER,MUTEX                  | copy mutex to register and set mutex to 1
      CMP REGISTER,#0                     | was mutex zero?
      JZE ok                              | if it was zero, mutex was unlocked, so return
      CALL thread_yield                   | mutex is busy; schedule another thread
      JMP mutex_lock                      | try again later
ok:   RET | return to caller; critical region entered


mutex_unlock:
      MOVE MUTEX,#0                       | store a 0 in mutex
      RET | return to caller
```

In C-syntax:

```
void Lock(boolean *lock) {
      while (test_and_set(lock) == true);
}
```

Compare and Set Instruction

- Atomic Operation:
  - If a memory location equals a "given" value, then assign a "new" value to the memory location. Else return the old value of the memory location.
- Useful for lock-free synchronization
- bool compare_and_set(mem, old, new)

```
{
   if mem ≠ old
      return false;
   else
      mem = new;
   return true
}
```

- Ref: https://en.wikipedia.org/wiki/Compare-and-swap

- x86 instruction:
  - CMPXCHG NEWVAL, MEMORY
  - NEWVAL: Explicit operand. A register.
  - MEMORY: Explicit operand. A memory location (or a register).
  - Plus two implicit operands:
    - EAX register : contains the "given" value and returns the final value of MEMORY
    - EFLAGS.ZF bit: Indicates if exchange was successful or not.
- IF (%EAX == MEMORY) THEN
  - EFLAGS.ZF := 1
  - MEMORY := NEWVAL
- ELSE
  - EFLAGS.ZF := 0
  - %EAX := MEMORY