# Semaphores and Condition Variables

*Modern Operating Systems*, by Andrew Tanenbaum Chap 2.3 & 6

Operating Systems: Three Easy Pieces (a.k.a. the OSTEP book) Chap 30&31

# Semaphore

- A fundamental synchronization primitive used for:
  - Locking critical regions
  - Inter-process synchronization

- A special integer, "sem", on which only two operations can be performed
  - A "DOWN(sem)" operation
  - An "UP(sem)" operation

# The DOWN(sem) Operation

- If (sem>0)…
  - Subtract 1 from sem
  - Continue processing (this is a "successful") down operation

- while (sem==0) …
  - Block the caller until sem>0 (someone else did an "UP(sem)")
  - try DOWN(sem) again.. if successful, break
    - Might fail because other processes might do a DOWN first

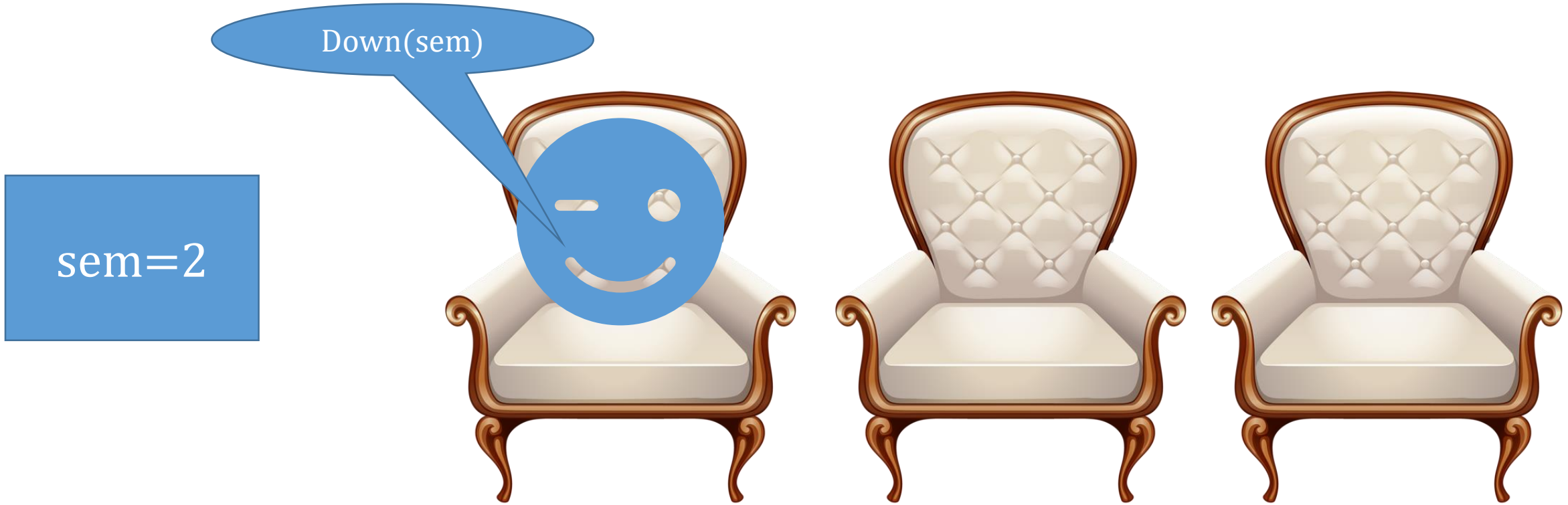
# The UP(sem) Operation

- If (sem==0)
  - add 1 to sem
  - wake up all processes blocked on DOWN(sem)
  - All woken up processes compete to perform DOWN(sem)
    - Only one can succeed… the rest are blocked again
- else
  - add 1 to sem

# Semaphore Example – "Chair is taken"

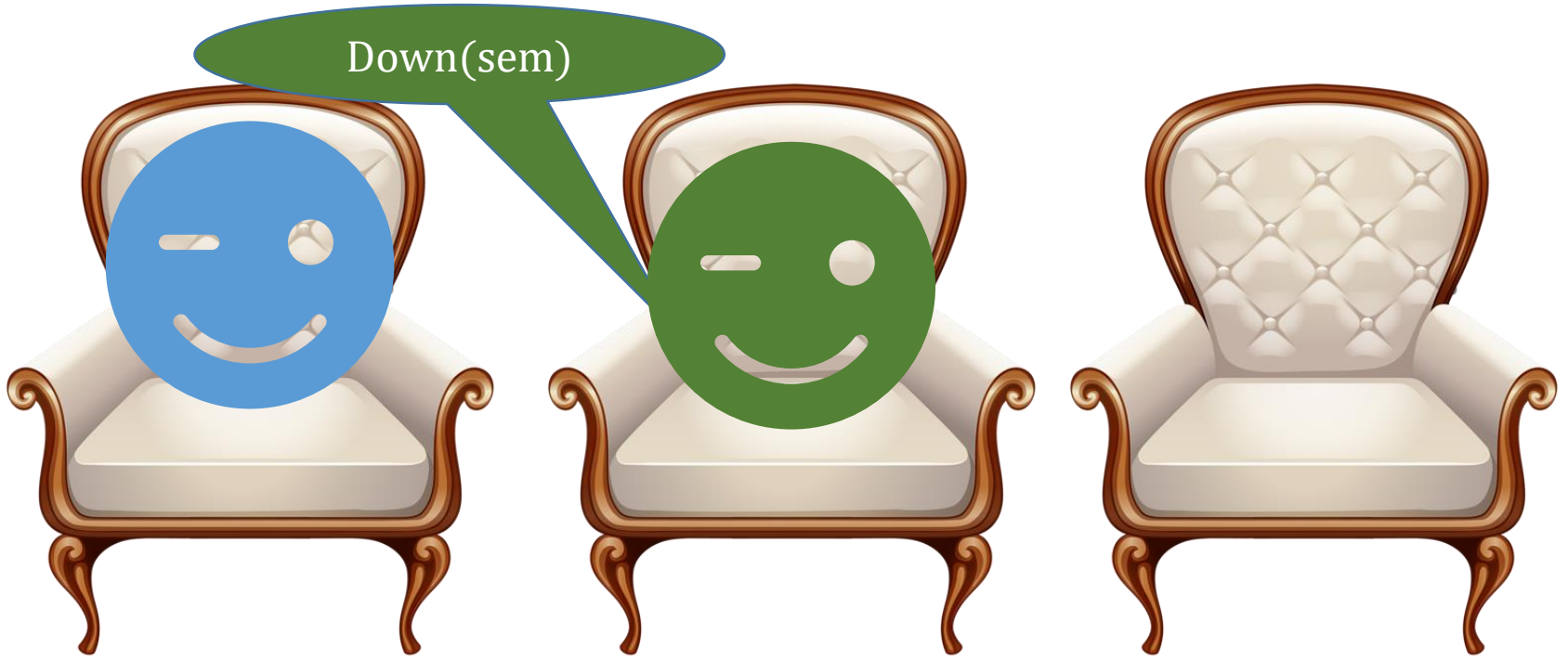
sem=3

# Semaphore Example – "Chair is taken"



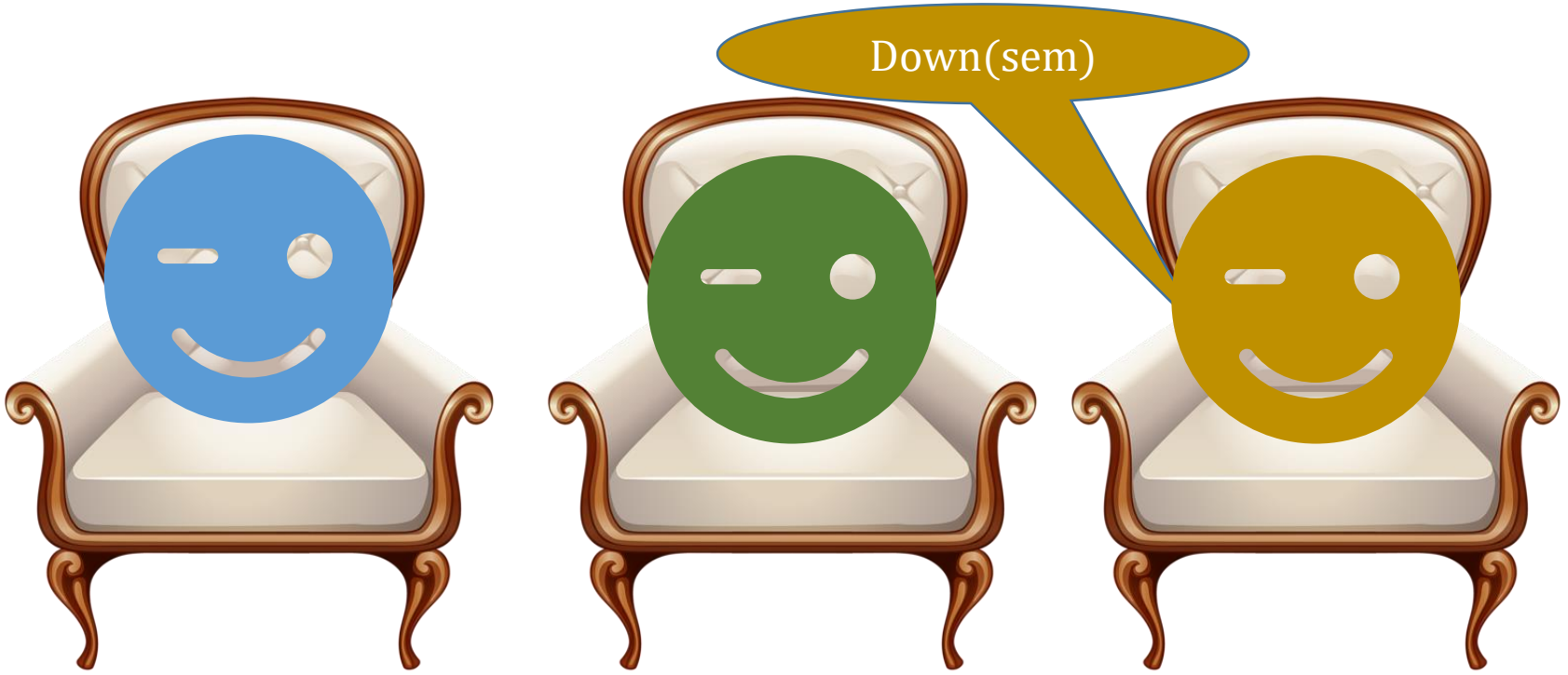
# Semaphore Example – "Chair is taken"

sem=1

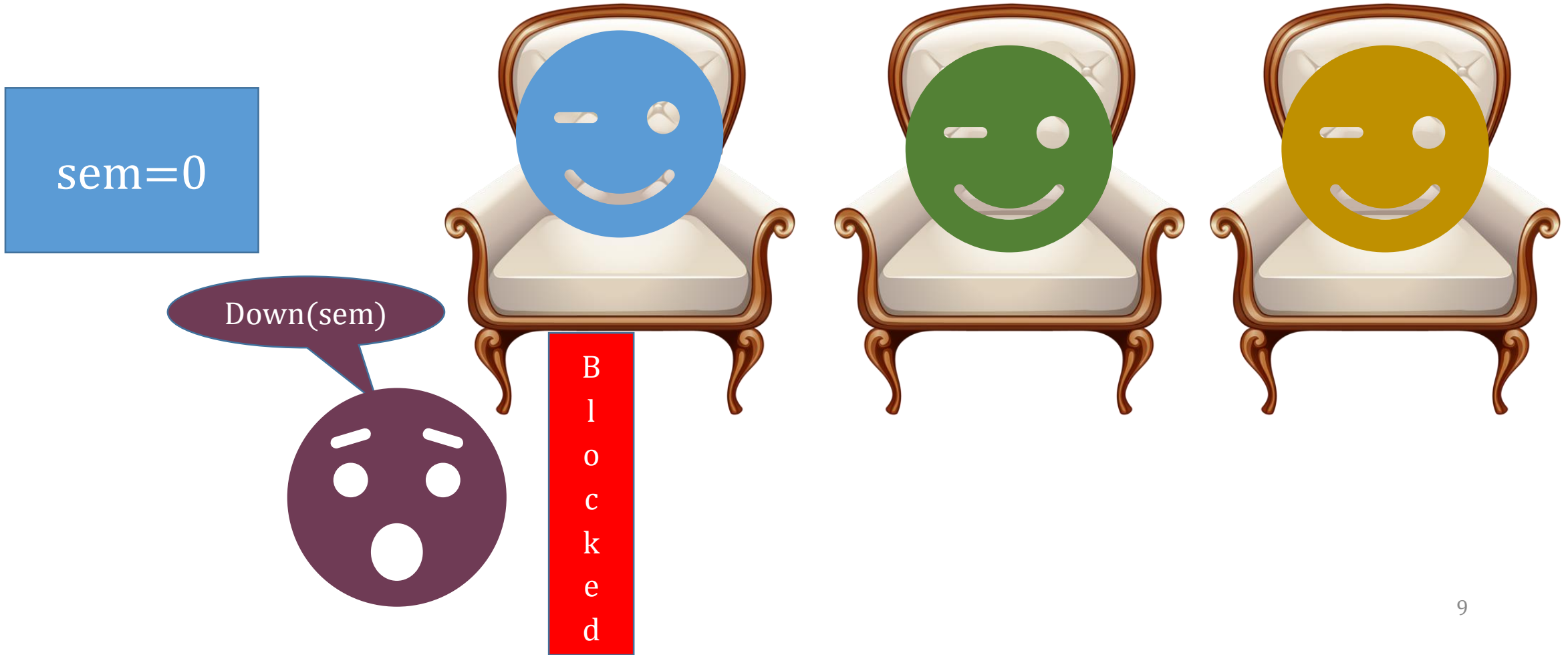


# Semaphore Example – "Chair is taken"

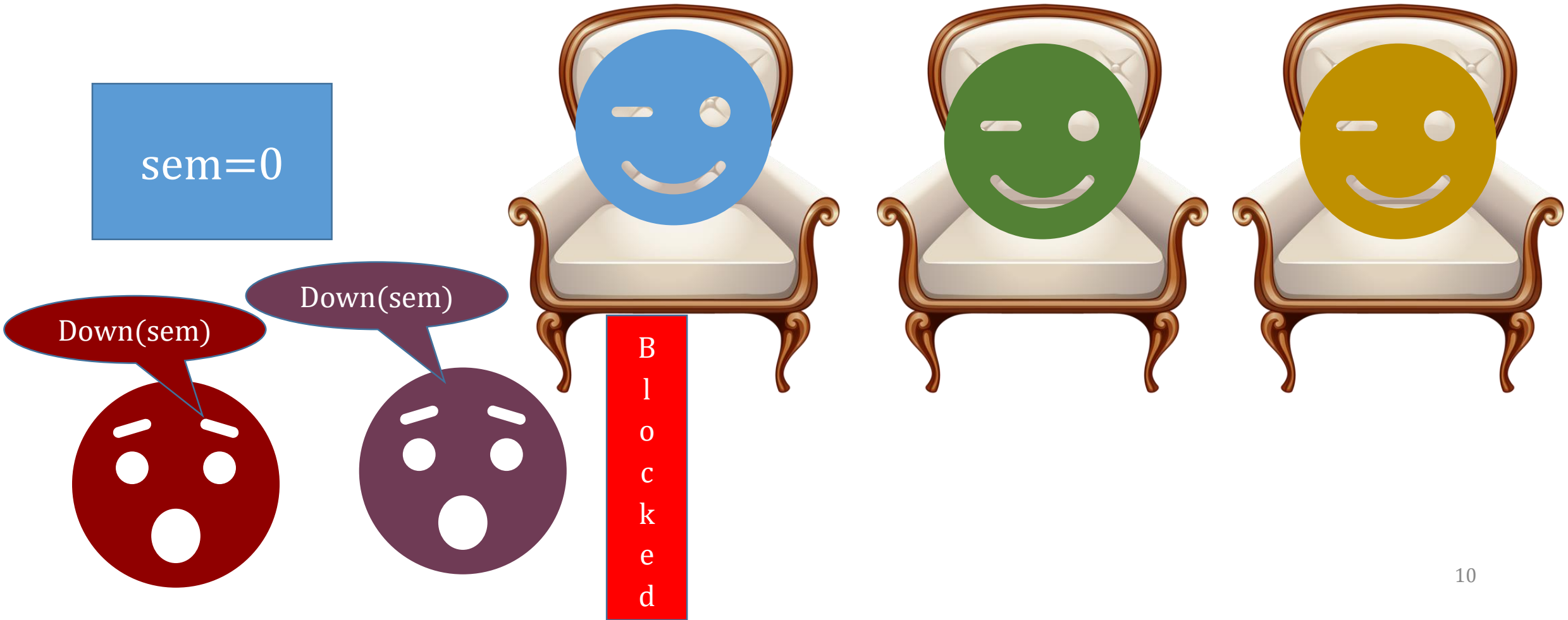
sem=0



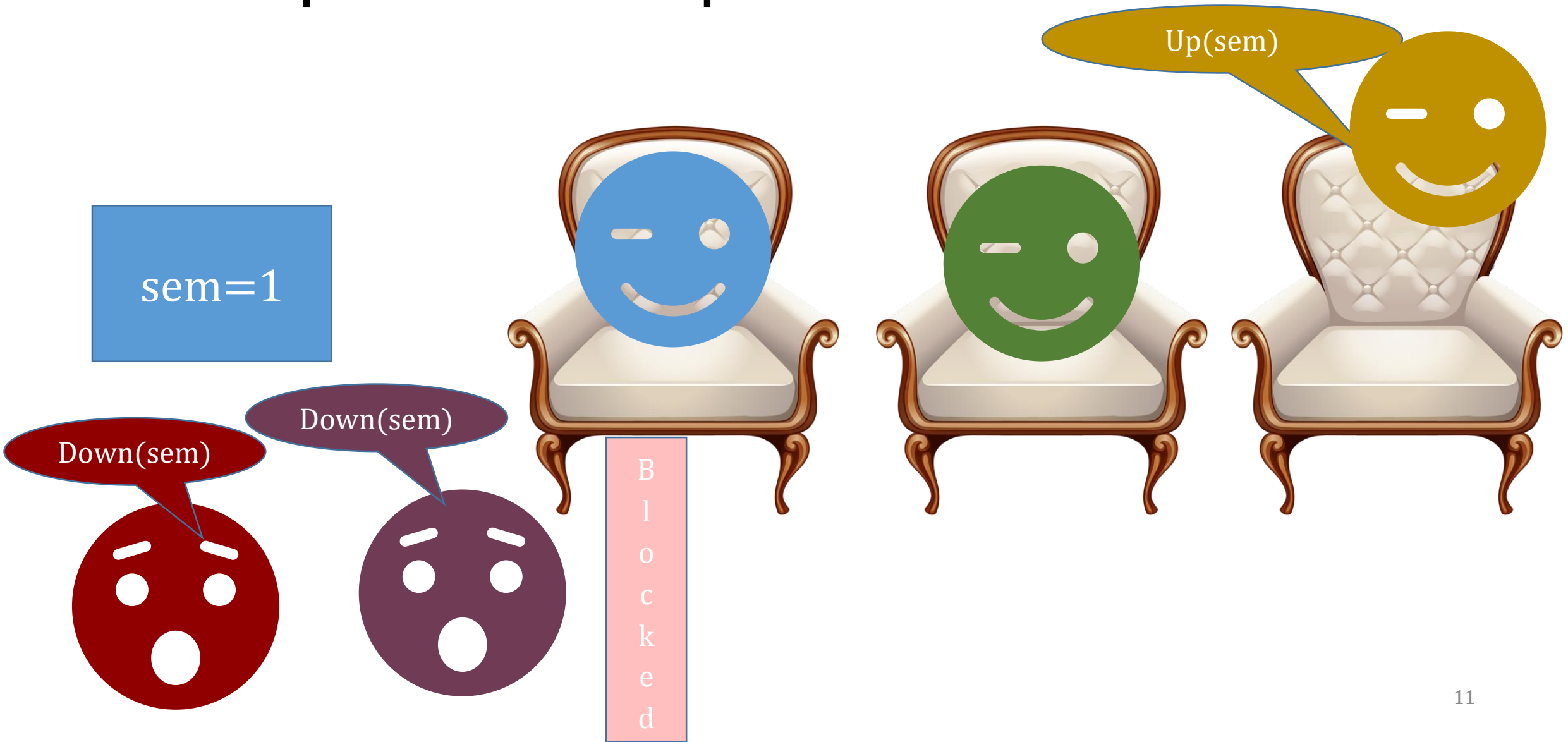
# Semaphore Example – "Chair is taken"



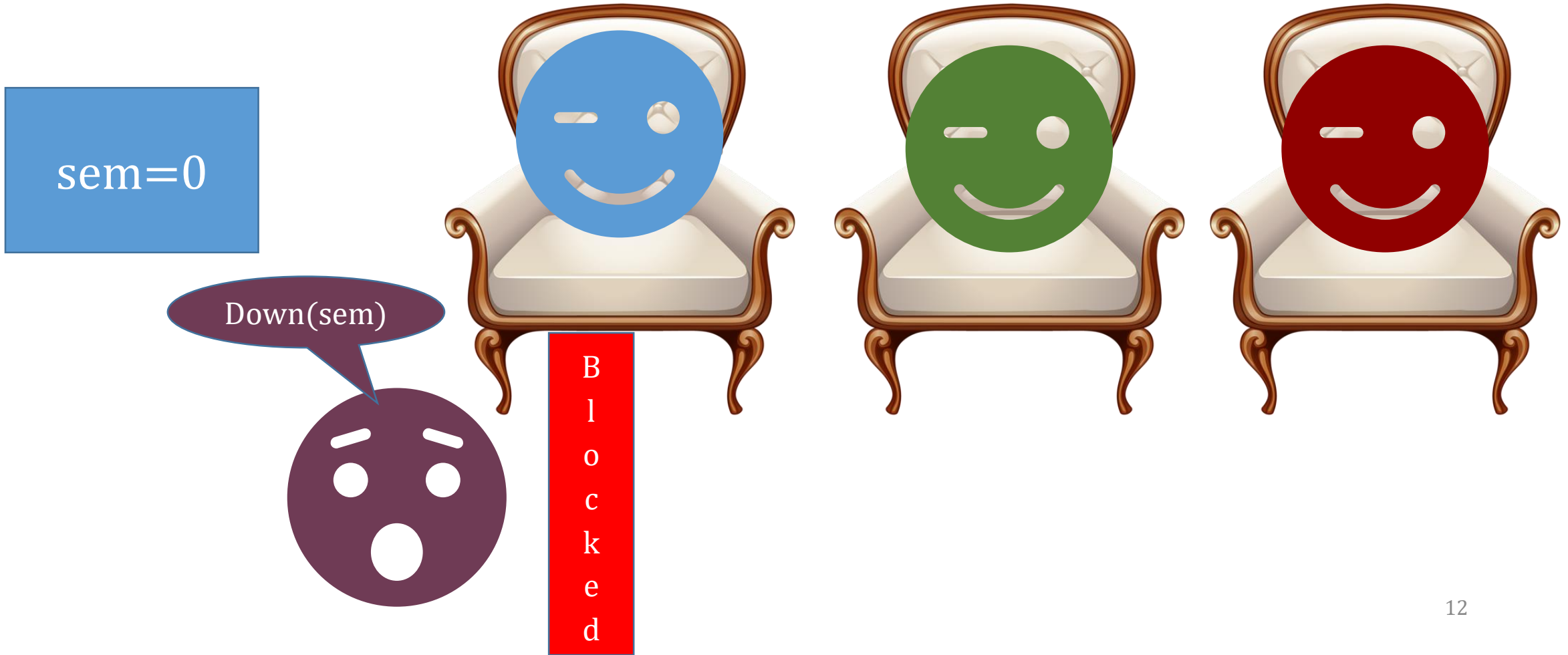
# Semaphore Example – "Chair is taken"



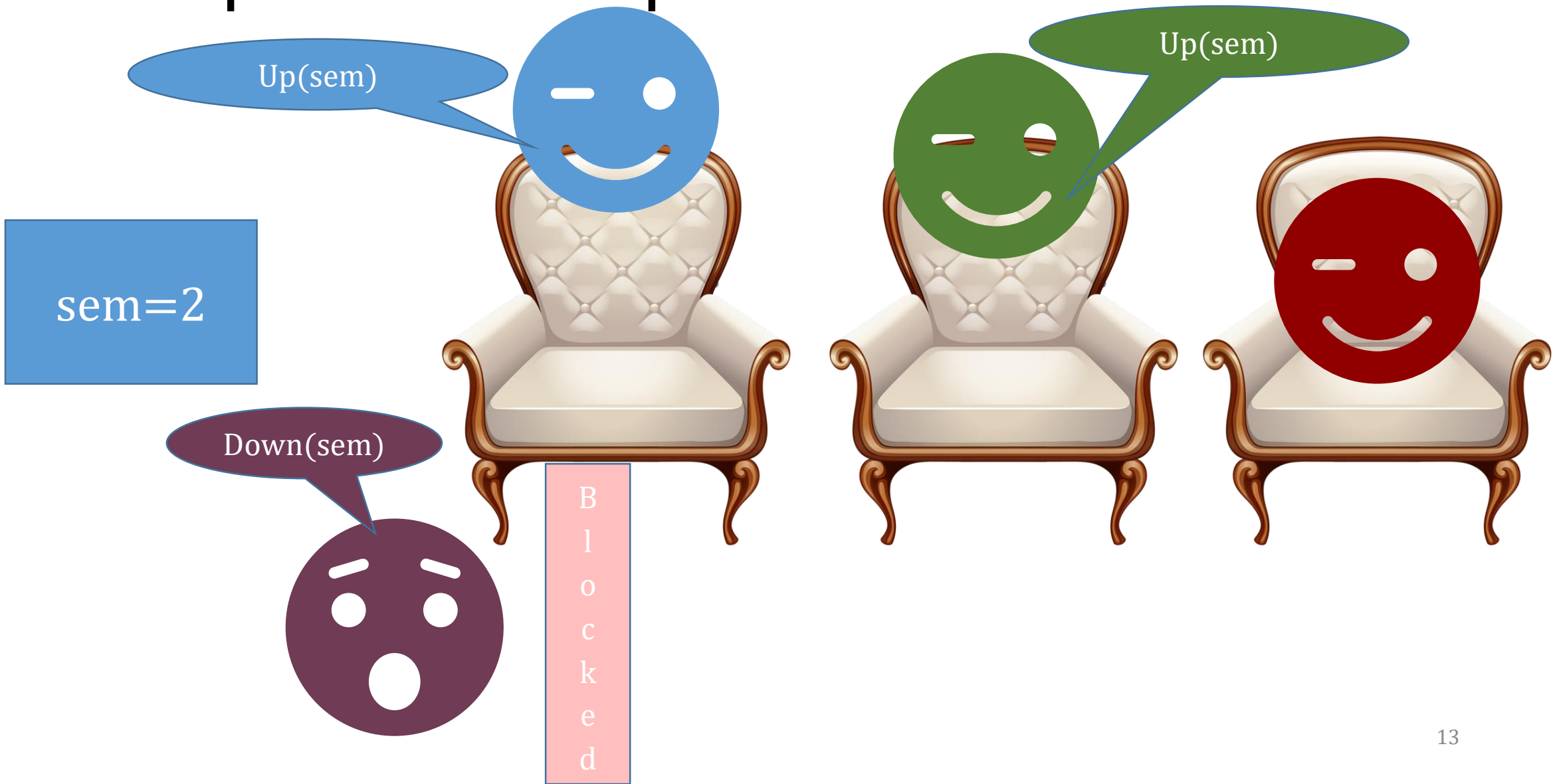
# Semaphore Example – "Chair is taken"



# Semaphore Example – "Chair is taken"



# Semaphore Example – "Chair is taken"



# Semaphore Example – "Chair is taken"

sem=1

# Mutex

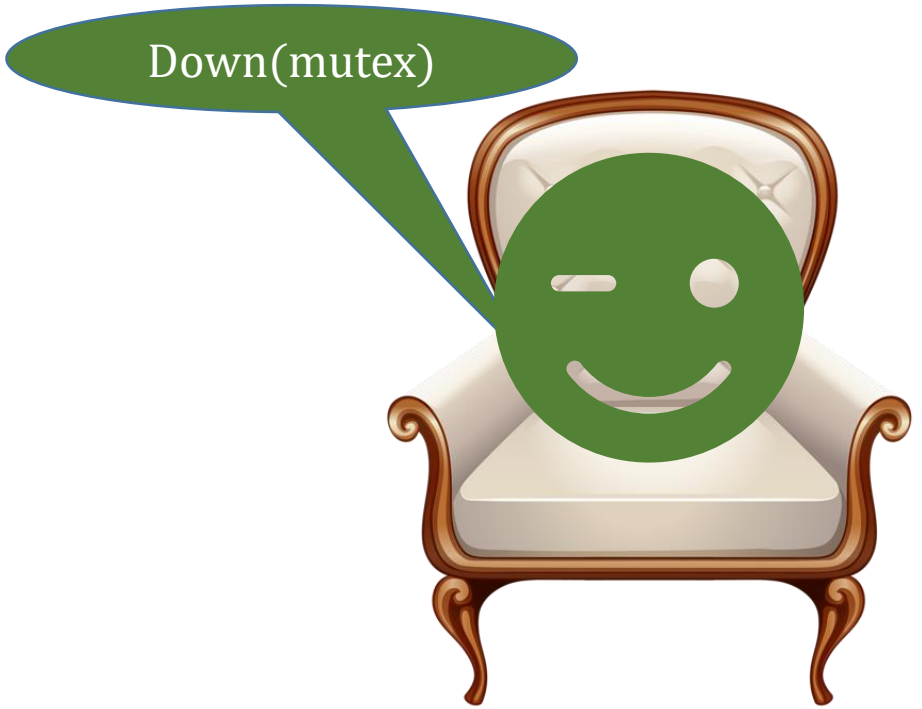
- A binary semaphore – Can have a value of 0 (blocking) or 1 (open)
- Used as a lock around critical sections
- To enter a critical section (lock a resource) use "Down(mutex)"
  - If mutex==1 (open), decrement mutex value to 0 and continue
  - If mutex==0 (blocking) blocks until someone else does an "Up(mutex)"
- To exit a critical section (unlock resource) use "Up(mutex)"
  - Increment mutex value to 1
  - Wake up all sleepers on Down(mutex) – one gets the lock, the others go back to sleep

# Mutex Example – "Chair is taken"

mutex=1

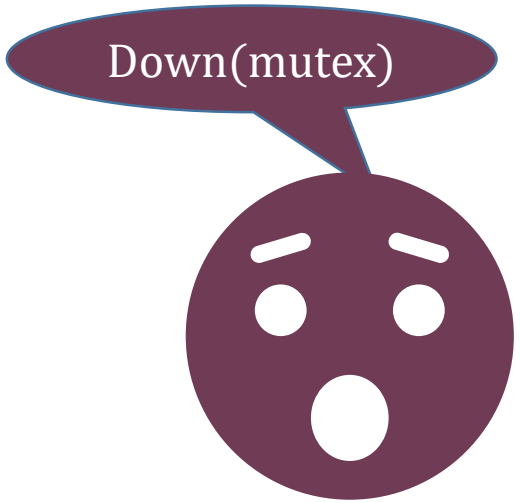
# Mutex Example – "Chair is taken"

mutex=0



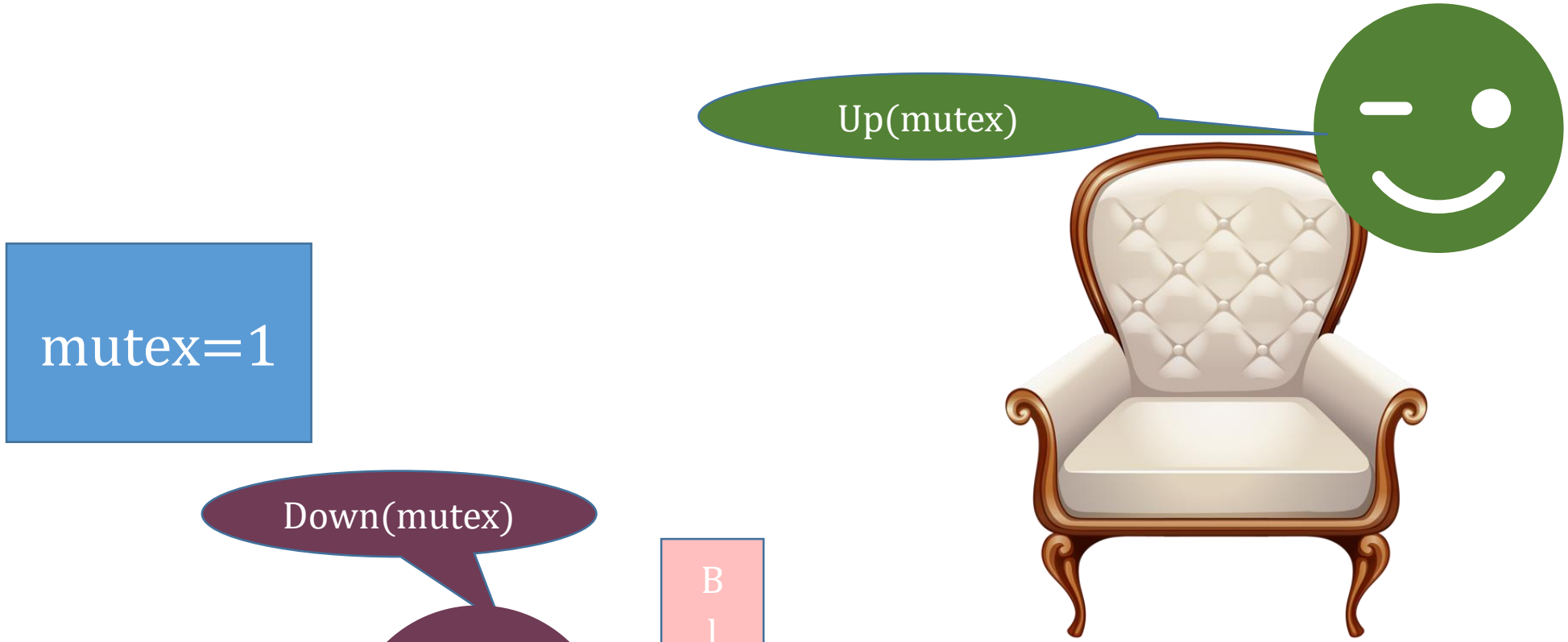
# Mutex Example – "Chair is taken"

mutex=0



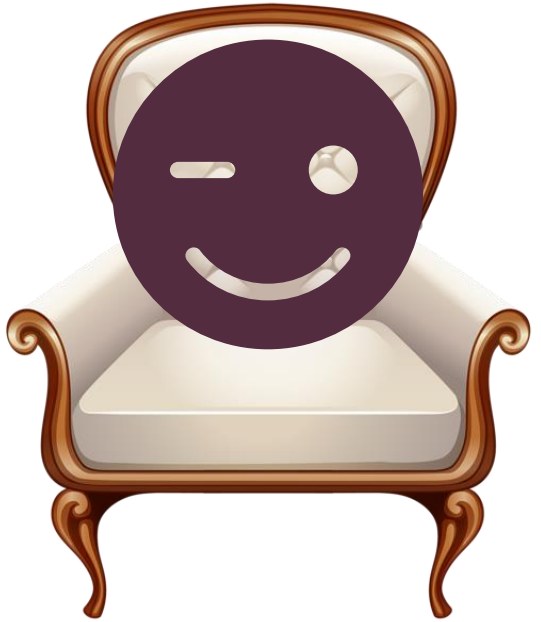

Blocked

# Mutex Example – "Chair is taken"



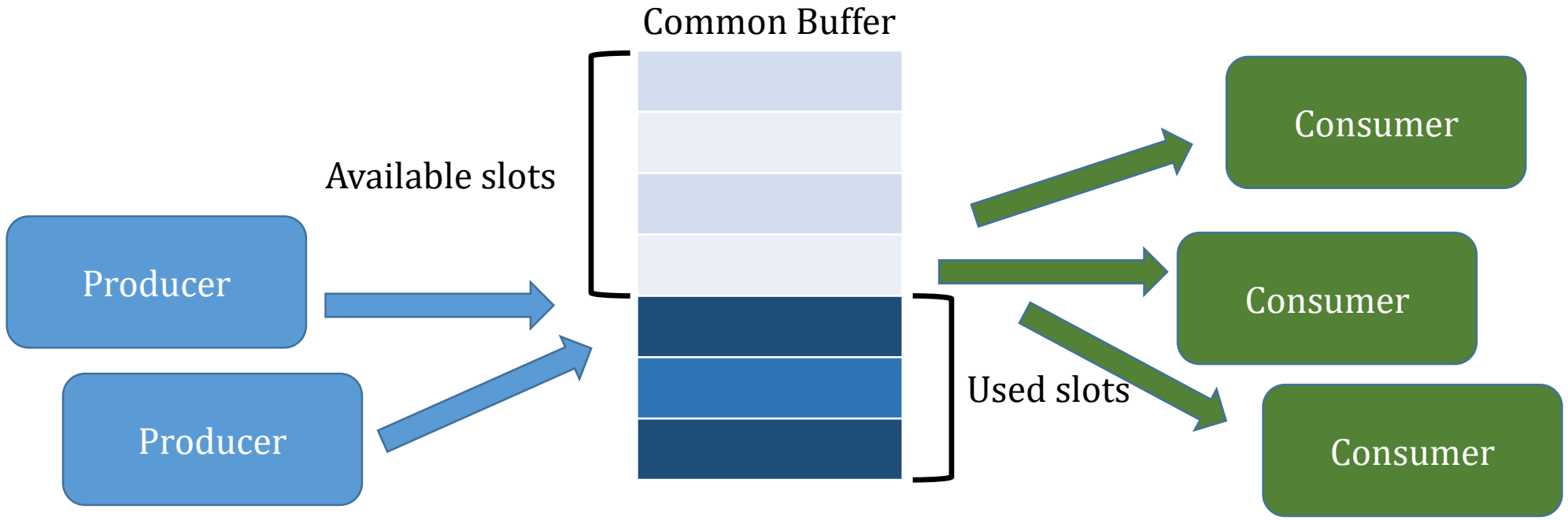
# Mutex Example – "Chair is taken"

mutex=0



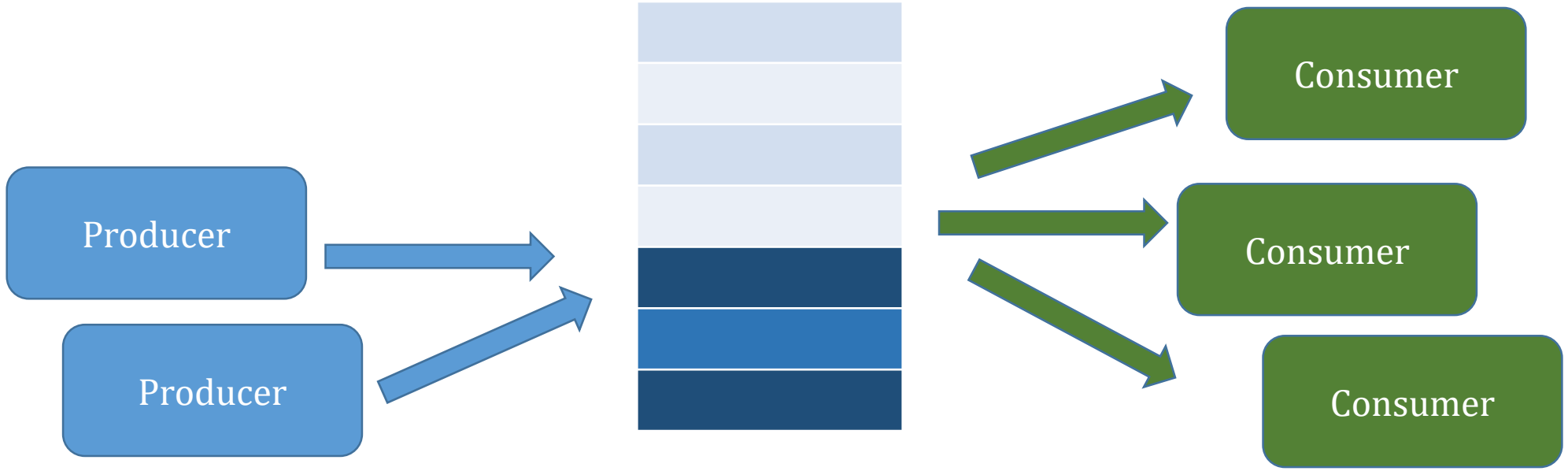
# Example: Producer/Consumer Problem

- Producers and Consumers runs concurrently
  - Producers add items to a common buffer
  - Consumers take items from a common buffer



# Producer/Consumer - Coordination

- Consumers should sleep when buffer is empty
- Producers should sleep when buffer is full
- No two processes should work on the buffer at the same time



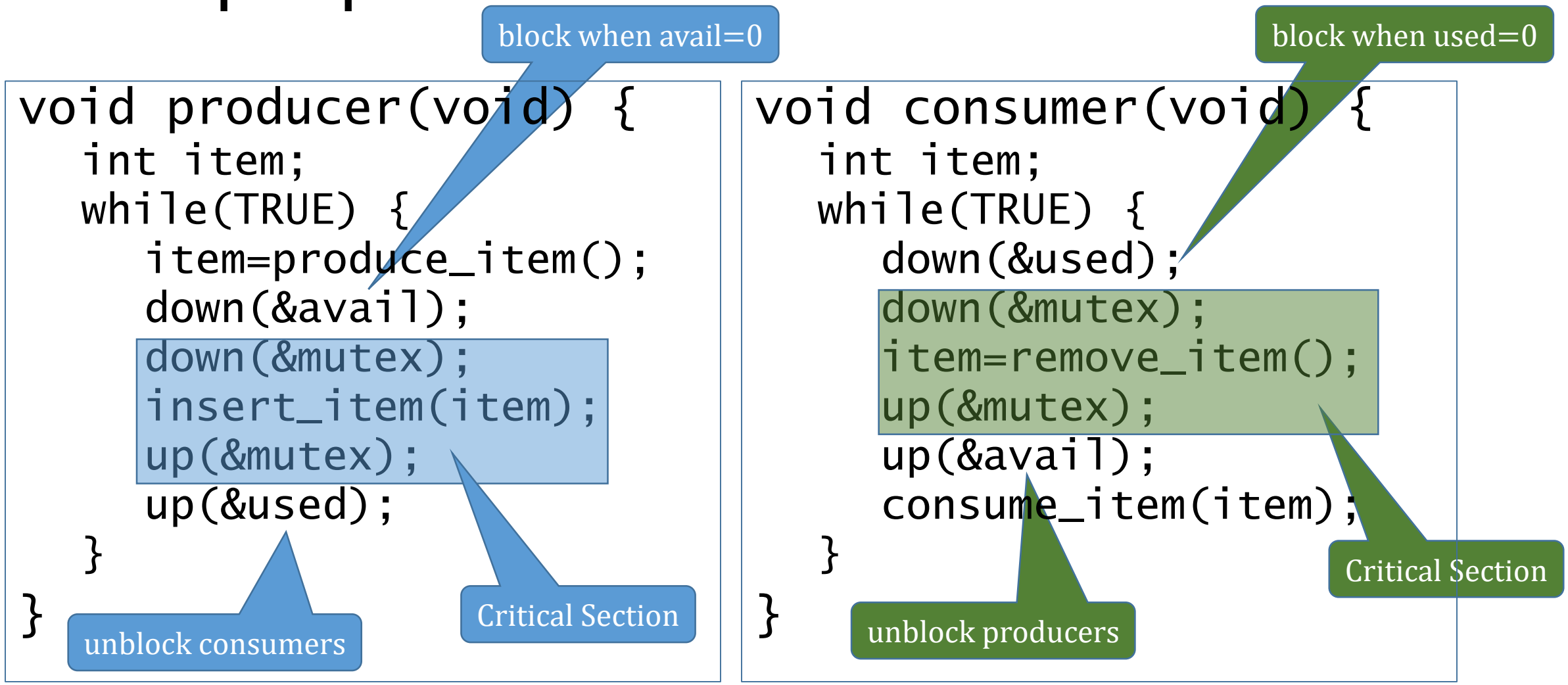
# Using Semaphores for Prod/Cons

```
#define N 100 // Number of slots in the common buffer
typedef int semaphore; // Semaphore is a special kind of int
semaphore mutex=1; // Control access to the common buffer
semaphore avail=N; // The number of available slots in buffer
semaphore used=0; // The number of used slots in buffer
```

Note: Two types of semaphores used here…

- A binary semaphore (mutex) to lock the buffer
- Regular semaphores to count avail and used slots

# Example producer/consumer code



# Semaphores – POSIX interface

- sem_open() – Optionally creates and/or connects to a named semaphore
- sem_init() – Initializes an un-named semaphore in (shared) memory
- sem_wait() – blocks while semaphore is held by other processes, then decrements (down)
- sem_trywait() – returns an error if semaphore is held by other processes
- sem_post() – Increments the count of the semaphore (up)
- sem_getvalue() – Returns the current value of the semaphore
- sem_close() – Ends the connection to a named semaphore
- sem_unlink() – Ends the connection to a named semaphore and removes it when all connections are ended
- sem_destroy() – Cleans up an unnamed semaphore

# Semaphores – System V (older) interface

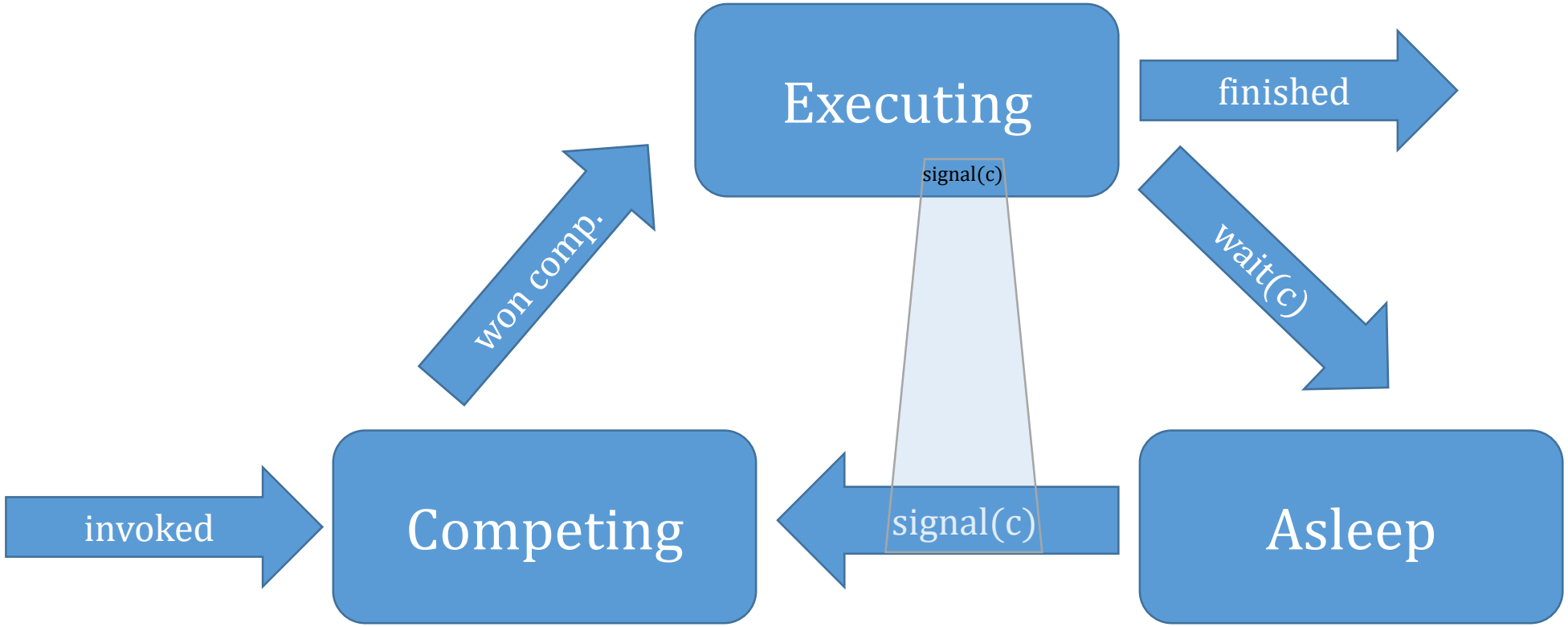
- semget(key, nsems, semflg) – creation (sem value=0)
- semctl(semid,0,SETVAL,arg) – Initialization
  - union semun arg; arg.val=1;
  - Not atomic w/ creation
- semop(semid, &sops, nsops) - Incr/Decr/Test-and-Set
  - struct sembuf sops;
- semctl(semid,0,IPC_RMID, 0) - Deletion

# Monitors with Condition Variables

- Object oriented view of contention used for thread-safe objects
- Invented by Per Brinch Hansen and C.A.R. Hoare for Concurrent Pascal
- Monitor: A collection of critical section functions that operate on shared resources

- One global lock for all procedures… only one procedure can be executing at any given time
- One or more "condition variables", c
- wait(c) – Releases the global lock, and puts the calling function to sleep. Does not return until it re-acquires the monitor lock
- signal(c) – Wakes all functions waiting on c, who then compete for the monitor lock

# Monitor Function State Graph



# Example Monitor: Producer/Consumer

```
procedure producer;
begin
   while true do
   begin
      item=produce_item;
      PCmon.insert(item)
   end
end;
```

```
procedure consumer;
begin
   while true do
   begin
      item=Pcmon.remove;
      consume_item(item)
   end
end;
```

```
monitor Pcmon
   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 – Test and Set Lock

- Hardware building block for locks
- TSL Instruction: `TSL Register,Lock`
  - `Lock` – Address in memory with a value of 0 or 1
  - `Register` – One of the CPU General Purpose Registers

- The TSL instruction does TWO operations **atomically** (as one)

```
1. Register := Lock; // Copy value of Lock to Register
2. Lock := 1; // Set the Lock value to 1
```

- **Atomic**: Entire instruction must complete without pre-emption

# Implementing Mutex using TSL

```
mutex_lock:
     TSL REG,MUTEX
     CMP REG,#0
     JZE ok
     CALL thread_yield
     JMP mutex_lock
ok:  RET
```

```
mutex_unlock:
     MOVE MUTEX,#0
     RET
```

Set mutex to 1 (locked) if it wasn't already

If it wasn't locked, good to go

If it was locked, let another thread work

...and try again when scheduler runs you again

In C Syntax:

```
void lock(boolean *mutex) {
  while(test_and_set(lock)==true) {};
}
void unlock(boolean *mutex) {
  mutex=0;
}
```

# Compare and Set (CAS)

- Atomic Operation useful for "lock-free" synchronization

```
bool compare_and_set(target,old,new) {
      if (target!=old) return false; // somebody touched target
      target=new; // Nobody touched target… OK to update
      return true;
}
```

Note: This is atomic!

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

# Example CAS: race-free adder

```
int add(int *counter, int increment) {
     int old; int new;
     do {
          old=*counter;
          new=old+increment;
     } while (compare_and_swap(counter,old,new)==false);
     return new;
}
```

Note: These do not need to be atomic!

# Compare and Swap Instruction

- Alternate x86 Instruction for locking
- Instruction Format: CMPXCHG NEWVAL,TARGET
  - NEWVAL – Any CPU General Purpose Register
  - TARGET – Memory location or Register
  - EAX implicit operand – Contains the "compare to" (old) value of TARGET
  - EAX result contains the actual old value of TARGET
  - EFLAGS.ZF – indicates if exchange was successful

if (%EAX==TARGET) then { EFLAGS.ZF:=1; TARGET:=NEWVAL }
else { EFLAGS.ZF:=0; %EAX:=TARGET; }