CS 1550

Week 6

Project 1 Quiz & Midterm prep

Teaching Assistant

Henrique Potter

CS 1550 – Lab 2 is out

- **Due**: Monday, February 17, 2020 @11:59pm
- **Late**: Wednesday, February 19, 2020 @11:59pm
  - 10% reduction per late day

Keep in mind the different qemu

- qemu with xv6 (Labs)

Midterm is approaching

- Questions in the midterm demand the application of what was learned

Midterm is approaching

- Questions in the midterm demand the application of what was learned
  - Think about how the concepts you learned could be applied

Midterm is approaching

- Questions in the midterm demand the application of what was learned
  - Think about how the concepts you learned could be applied
  - OS exams tend to ask questions that indirectly cover what was during courses
    - Instead of asking what Race Condition is, exams will show you a code excerpt and ask what type of problems it could have
    - Real-life scenario in which a concept could also be logically applied

Project 1 – Quiz

20 min

Midterm is approaching

- **Question 6**
  - Pair up **men** and **women** as they enter a Friday night mixer.

Midterm is approaching

- **Question 6**
  - Pair up men and women as they enter a Friday night mixer
  - Each **man** and each **woman** will be represented by **one thread(Process)**

Midterm is approaching

- **Question 6**
  - Pair up men and women as they enter a Friday night mixer
  - Each man and each woman will be represented by one thread

Midterm is approaching

- **Question 6**
  - Pair up men and women as they enter a Friday night mixer
  - Each man and each woman will be represented by one thread

Midterm is approaching

- **Question 6**
  - Pair up men and women as they enter a Friday night mixer.
  - Each man and each woman will be represented by one thread
  - When the **man** or **woman** enters the **mixer**, its thread will call **one** of two procedures, ***man*** or ***woman***, depending on the **thread gender**.

Midterm is approaching

- **Question 6**
  - Pair up men and women as they enter a Friday night mixer.
  - Each man and each woman will be represented by one thread
  - When the **man** or **woman** enters the **mixer**, its thread will call **one** of two procedures, ***man*** or ***woman***, depending on the **thread gender**.

```
Man () {

}
```

```
Woman () {

}
```

Midterm is approaching

- **Question 6**
  - Pair up men and women as they enter a Friday night mixer.
  - Each man and each woman will be represented by one thread
  - When the man or woman enters the mixer, its thread will call one of two procedures, *man* or *woman*, depending on the thread gender.
  - Each procedure takes a single parameter, ***name***, which is just an integer name for the **thread**.

```
Man (name) {

}
```

```
Woman (name) {

}
```

Midterm is approaching

- **Question 6**
  - The procedure **must wait** until there is an **available thread** of the opposite **gender** and must then **exchange names** with this **thread**.

```
Man (name) {

}
```

```
Woman (name) {

}
```

Midterm is approaching

- **Question 6**
  - The procedure **must wait** until there is an **available thread** of the opposite **gender** and must then **exchange names** with this **thread**

Semaphore: sem = 0;
String: nameM, nameW;

Man (name) {
nameM = name;
}

Woman (name) {
nameW = name;
}

Midterm is approaching

- **Question 6**
  - The procedure must wait until there is an available thread of the opposite gender and must then exchange names with this thread.
  - Each procedure must **return** the integer **name** of the thread it paired up with

```
Semaphore: sem = 0;
String: nameM, nameW;
```

```
Man (name) {
   nameM = name;
   return nameW;
}
```

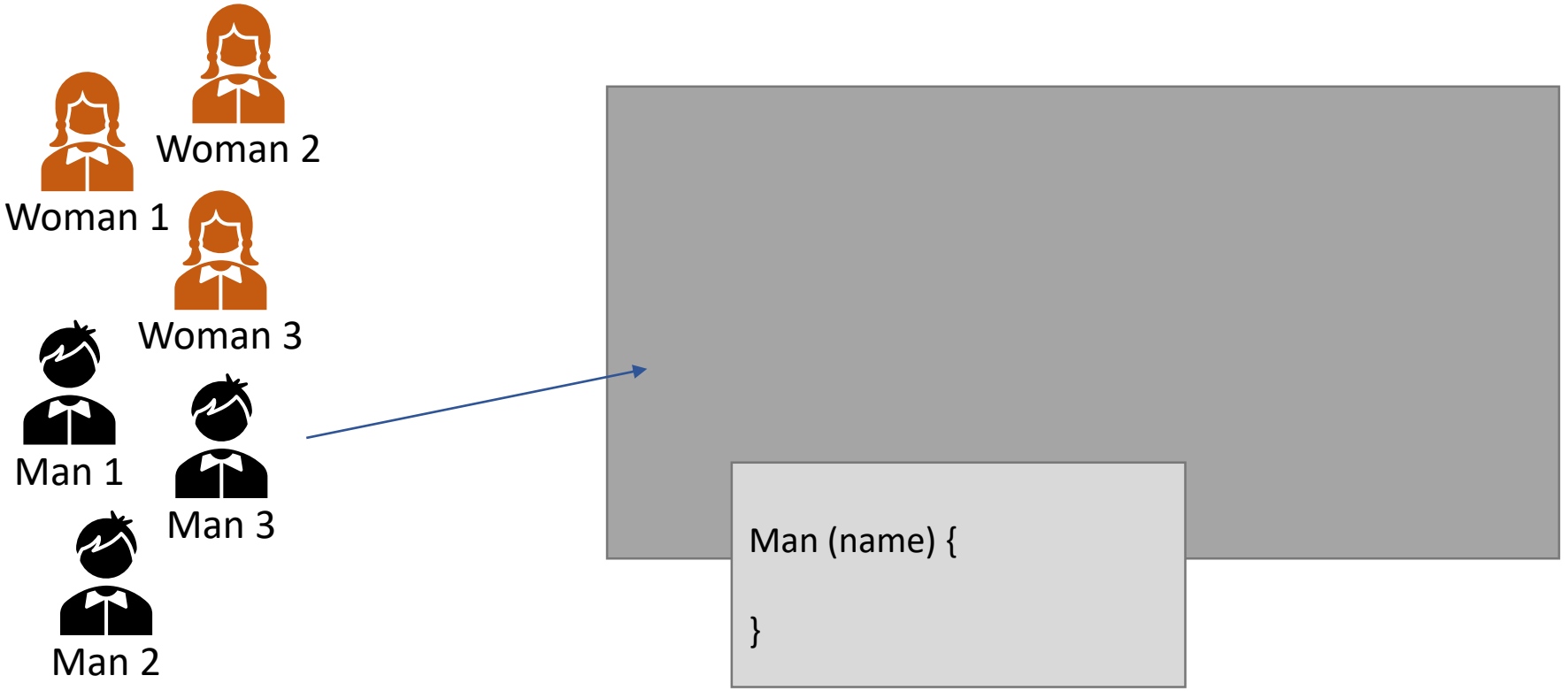
```
Woman (name) {
   nameW = name;
   return nameM;
}
```

Midterm is approaching

- **Question 6**
  - Each procedure must **return** the integer **name** of the thread it paired up with

Midterm is approaching

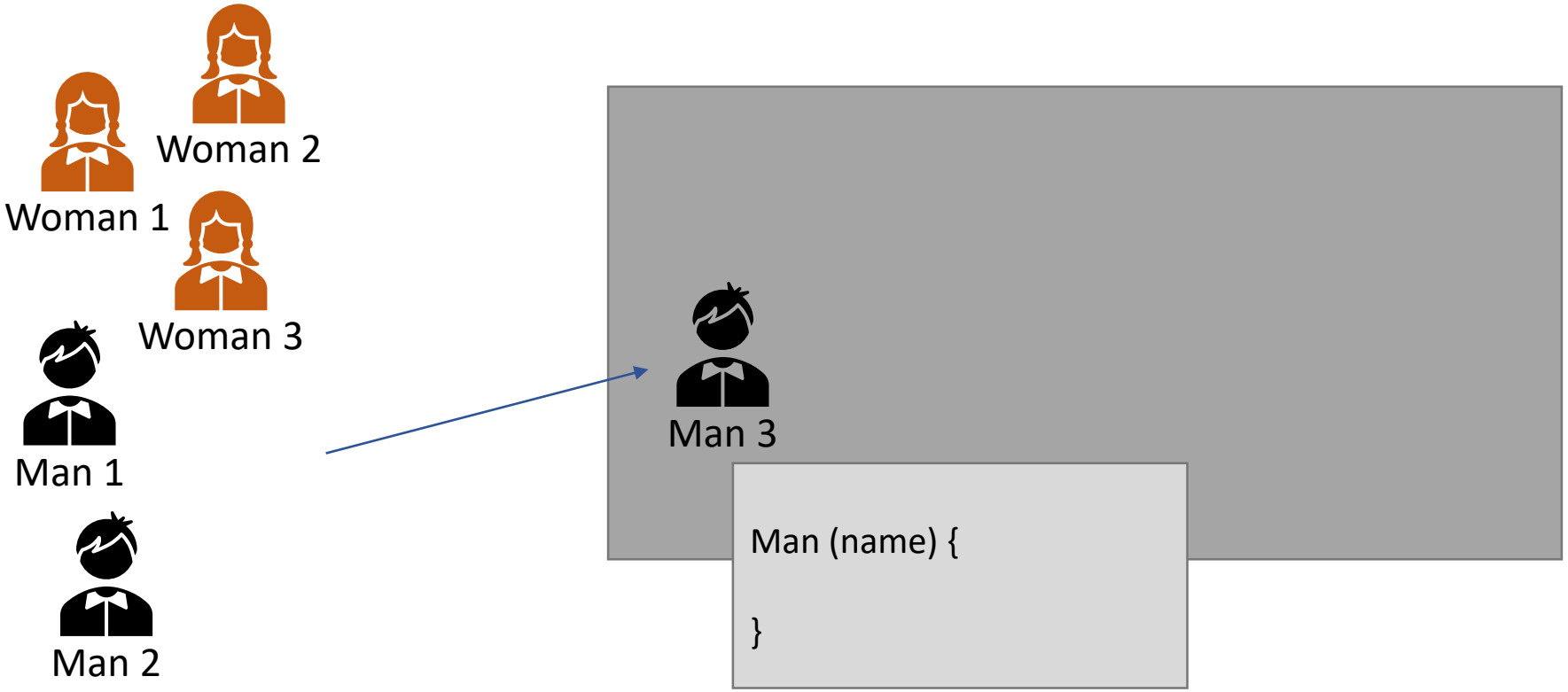
- **Question 6**
  - Each procedure must **return** the integer **name** of the thread it paired up with



When a Man attempts to enter a call to the **Man function** is done.

Midterm is approaching

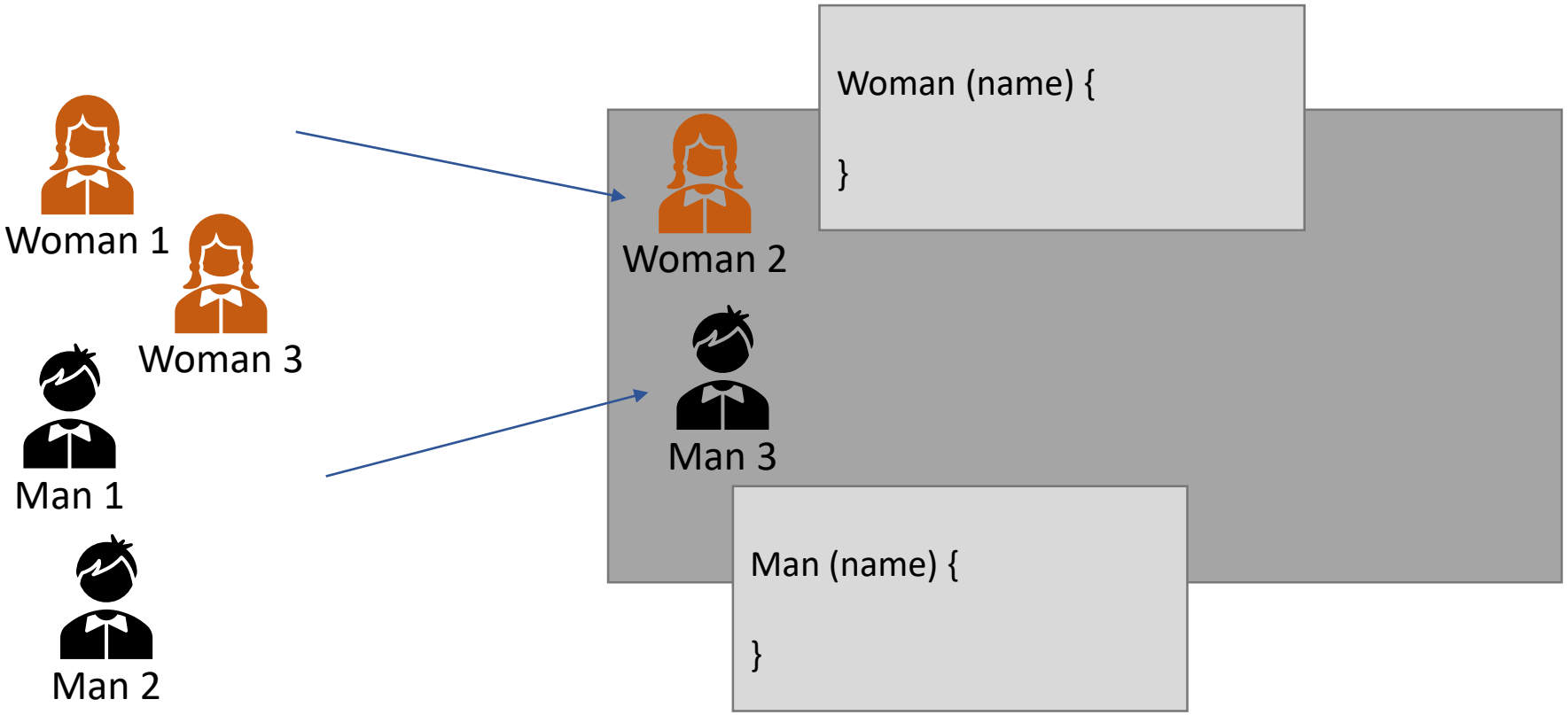
- **Question 6**
  - Each procedure must **return** the integer **name** of the thread it paired up with



He must **wait** to be paired with a Woman's name.

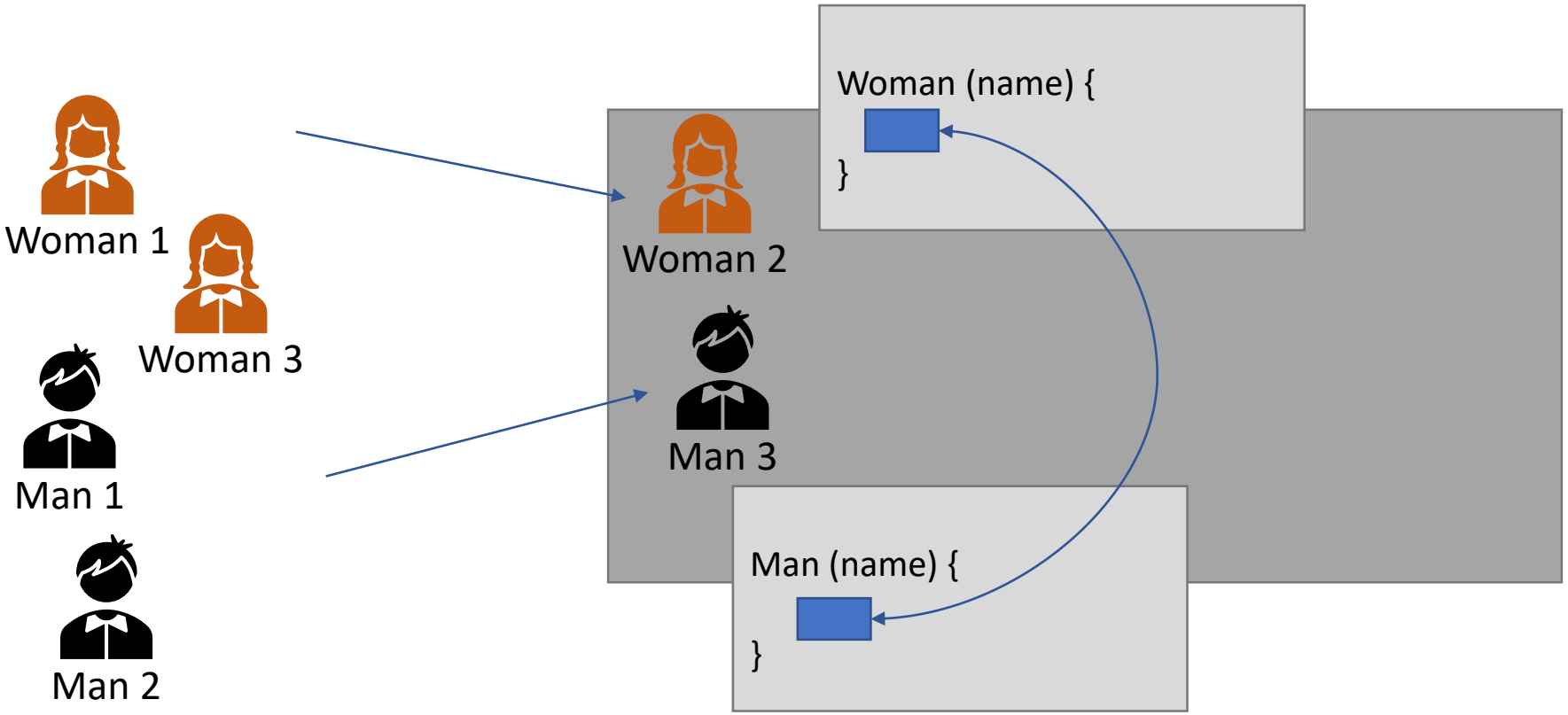
Midterm is approaching

- **Question 6**
  - Each procedure must **return** the integer **name** of the thread it paired up with



Midterm is approaching

- **Question 6**
  - Each procedure must **return** the integer **name** of the thread it paired up with



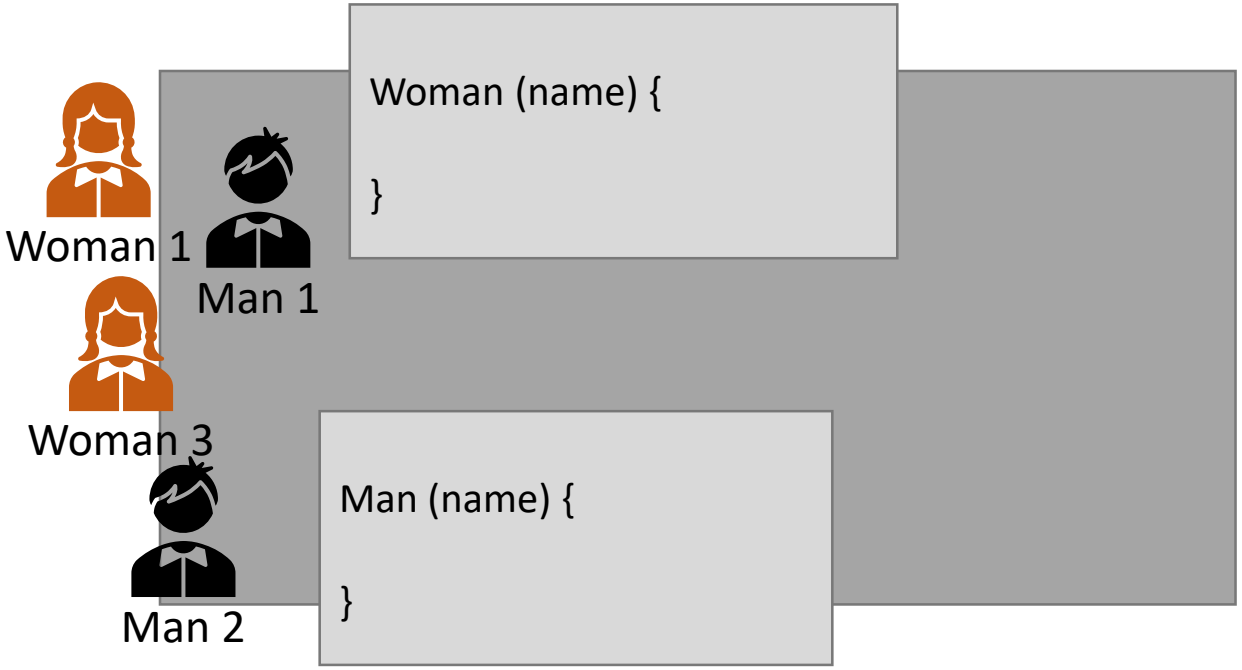
We need a **signaling mechanism** that would hold both processes/threads(Man and Woman) and only allow them to go when they are **paired**

Midterm is approaching

- **Question 6**
  - Men and women may enter the fraternity **in any order**, and many threads may **call** the ***man*** and ***woman*** **procedures simultaneously**.

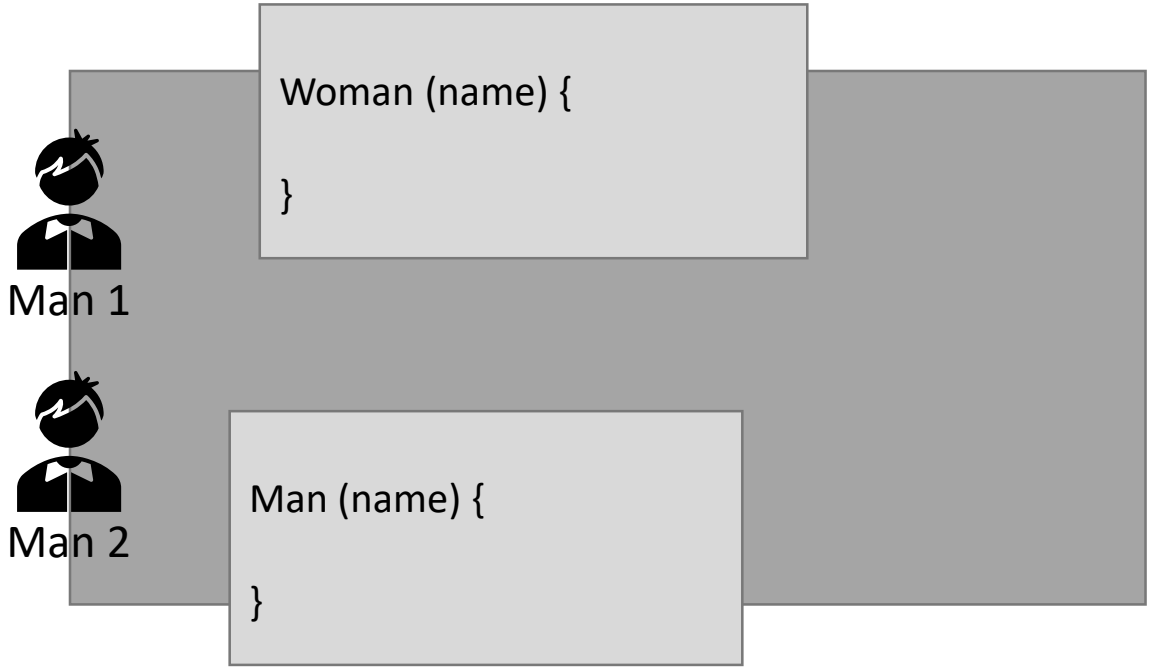
Midterm is approaching

- **Question 6**
  - Men and women may enter the fraternity in any order, and many threads may call the *man* and *woman* procedures simultaneously.



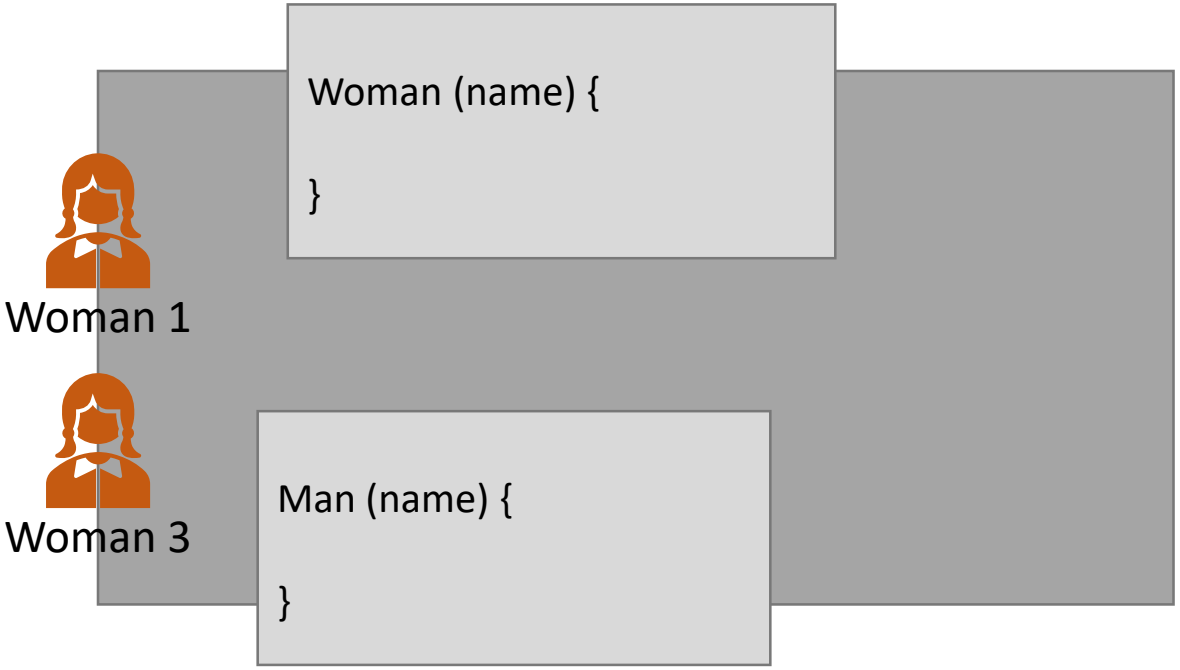
Midterm is approaching

- **Question 6**
  - Men and women may enter the fraternity in any order, and many threads may call the *man* and *woman* procedures simultaneously.



Midterm is approaching

- **Question 6**
  - Men and women may enter the fraternity in any order, and many threads may call the *man* and *woman* procedures simultaneously.



Midterm is approaching

- **Question 6**
  - Men and women may enter the fraternity in any order, and many threads may call the *man* and *woman* procedures simultaneously.
  - It doesn't **matter which man** is paired up with **which woman** (Pitt frats aren't very choosy in this exercise), as long as each pair contains one man and one woman, and each gets the other's name.
  - Use semaphores and shared variables to implement the **two procedures**.

Midterm is approaching

Man 2

Man 1

String: nameM, nameW; /* shared variables to share names */

```
wName Man (name) {
   nameM = name;
   return nameW;
}
```

```
mName Woman (name) {
   nameW = name;
   return nameM;
}
```

Woman 1

Woman 3

Midterm is approaching

Man 2

Man 1

String: nameM, nameW; /* shared variables to share names */

```
wName Man (name) {

    nameM = name;

    return nameW;
}
```

```
mName Woman (name) {

    nameW = name;

    return nameM;
}
```

Woman 1

Woman 3

Midterm is approaching

Man 2

Man 1

Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */

```
wName Man (name) {
   Down(mutexM);
   nameM = name;

   return nameW;
}
```

Only allow 1 person to enter

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;

   return nameM;
}
```

Woman 1

Woman 3

Midterm is approaching

Man 2

Man 1

Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexM);
   return nameW;
}
```

Only allow 1 person to enter

Should we allow each process to signal back to the same gender?

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;

   return nameM;
}
```

Woman 1

Woman 3

Midterm is approaching

Man 2

Man 1

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexM);
   return nameW;
}
```

Only allow 1 person to enter

Should we allow each process to signal back to the same gender?

No, multiple Mans would overwrite each others name.

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;

   return nameM;
}
```

Woman 1

Woman 3

Midterm is approaching

Man 2

Man 1

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexW);
   return nameW;
}
```

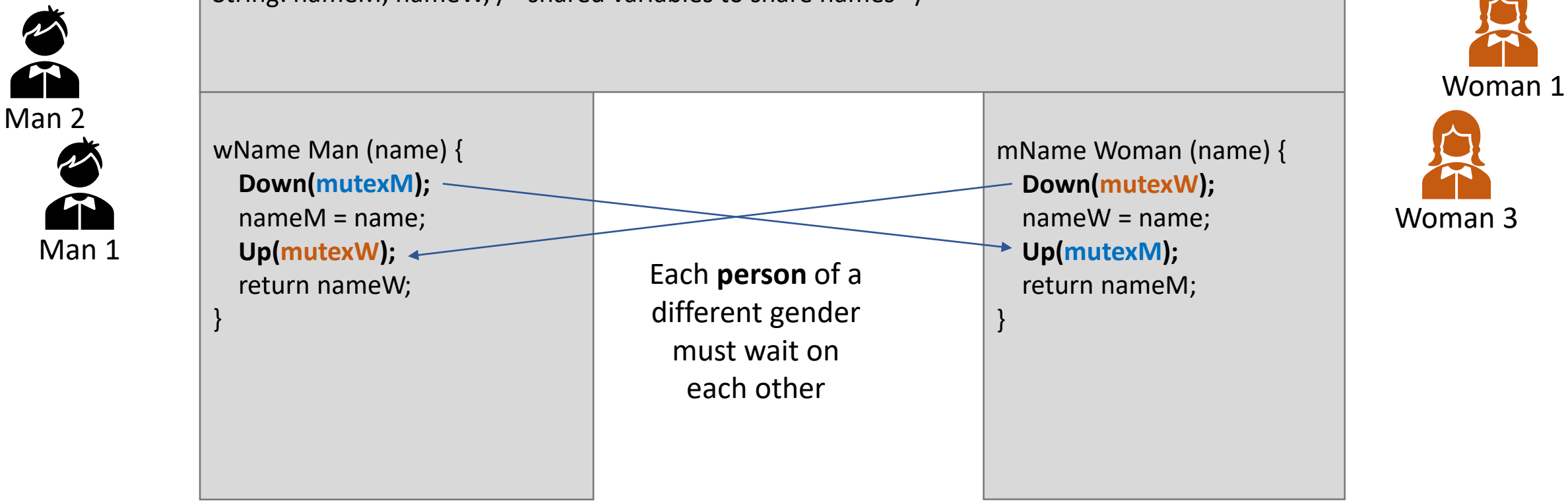
Only allow 1 person to enter

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(mutexM);
   return nameM;
}
```

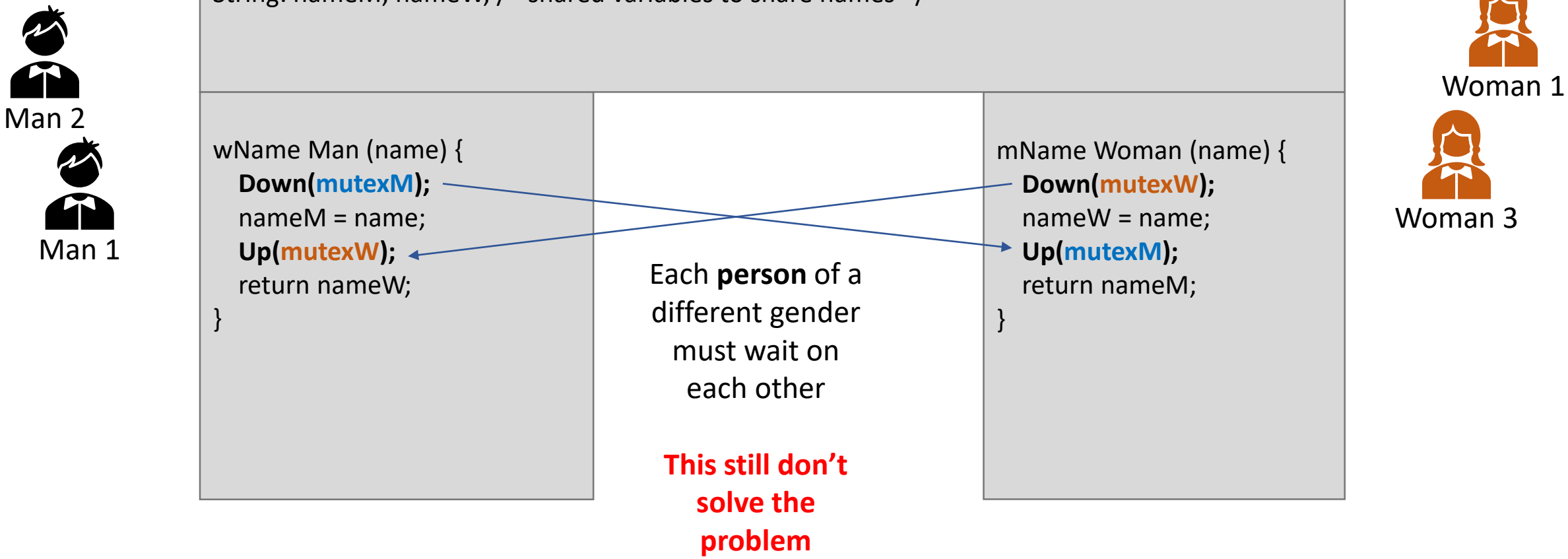
Woman 1

Woman 3

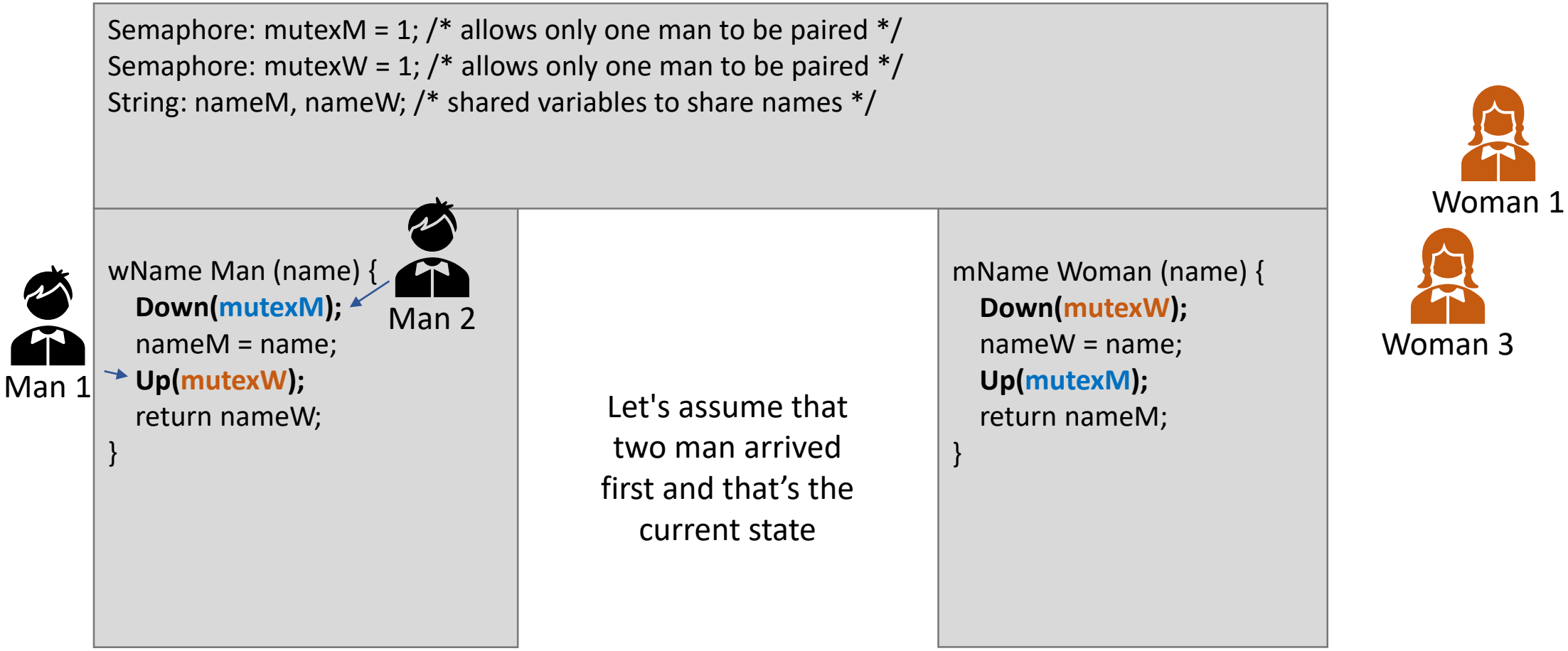
Midterm is approaching



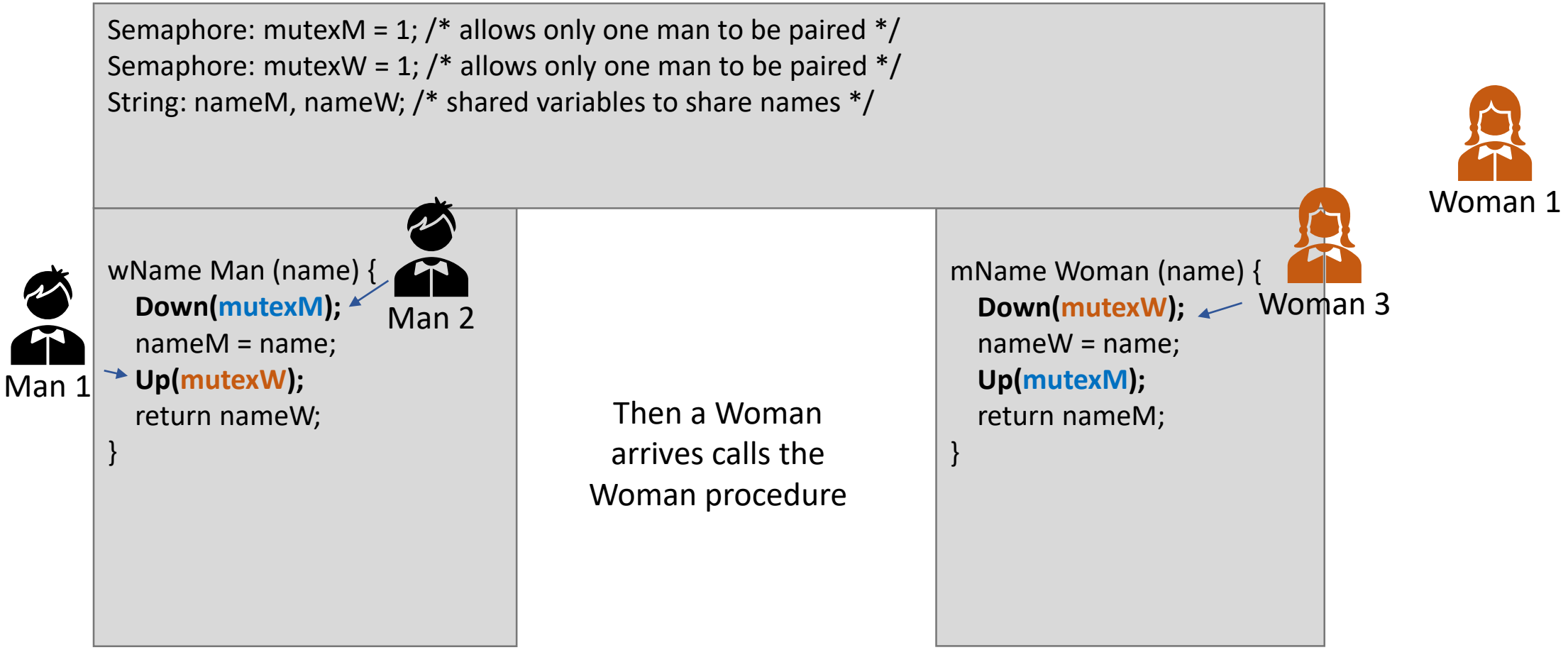
Midterm is approaching



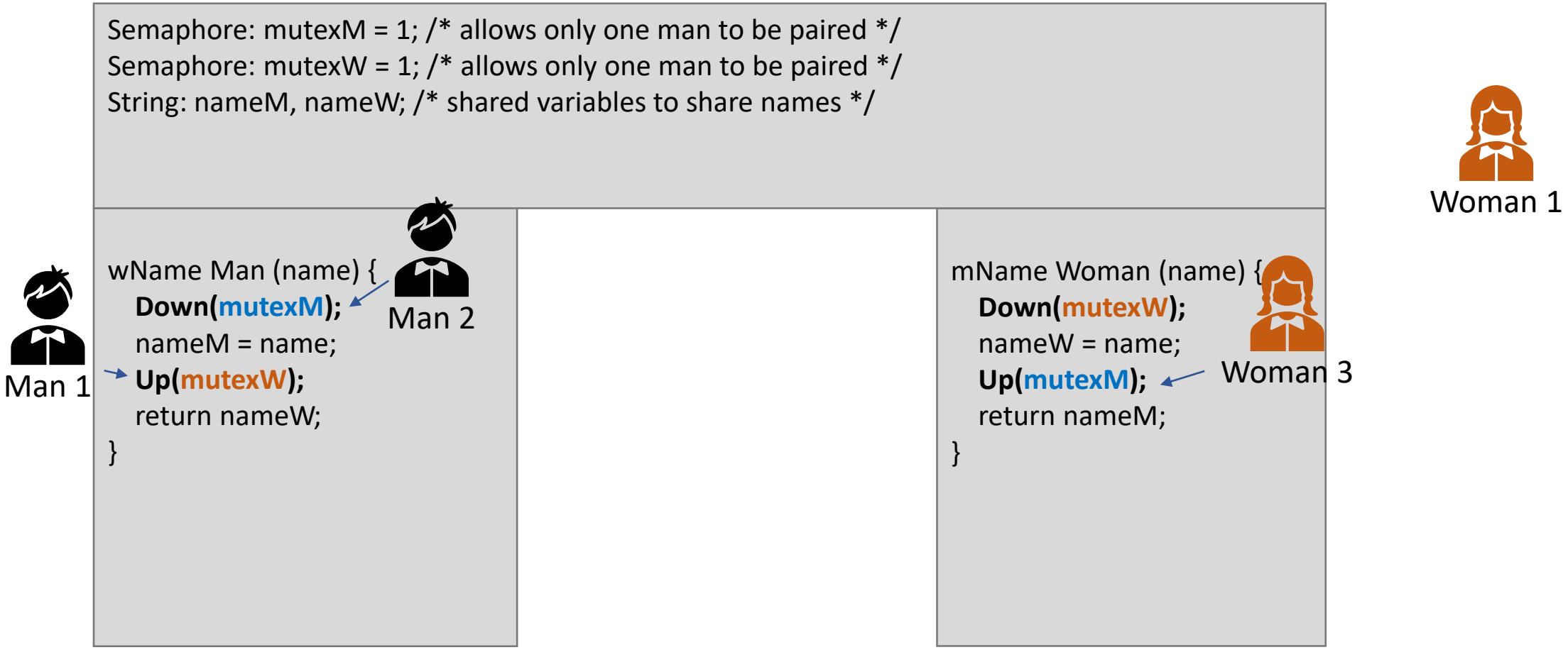
Midterm is approaching



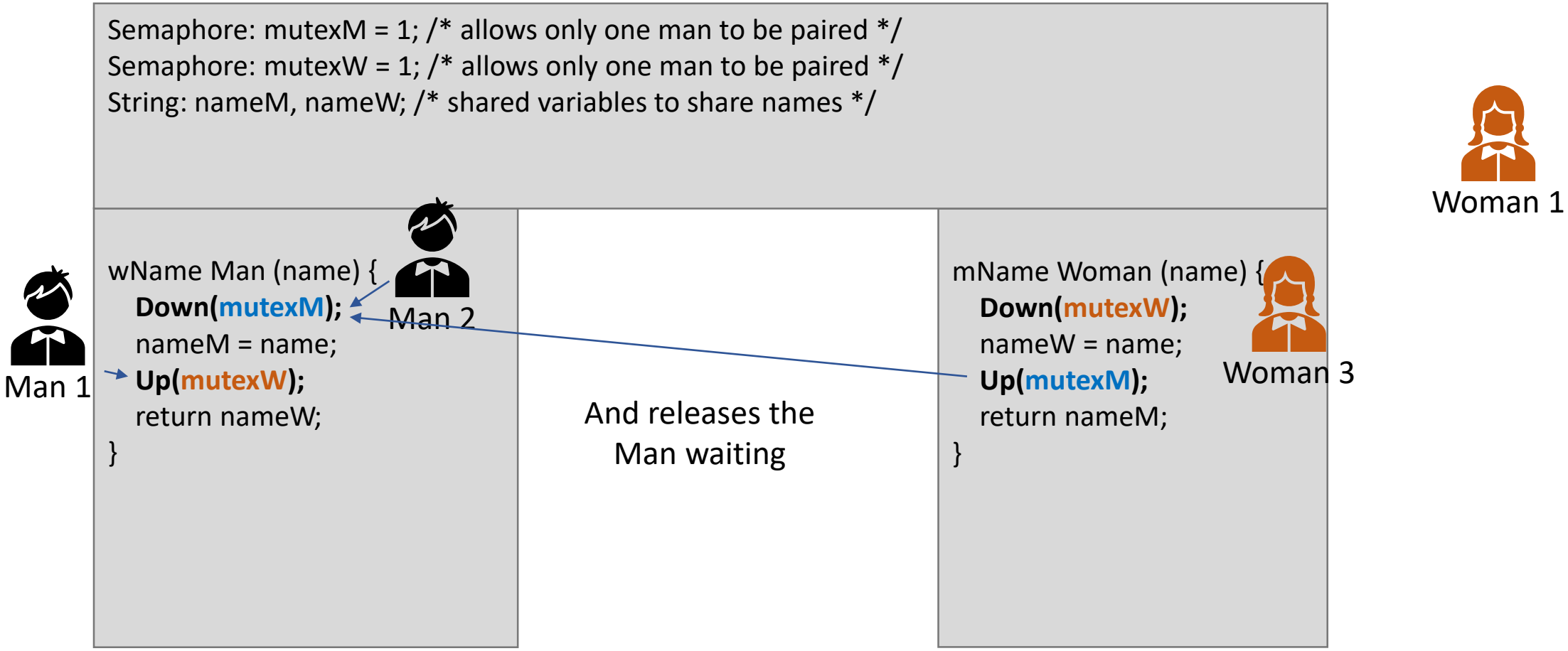
Midterm is approaching



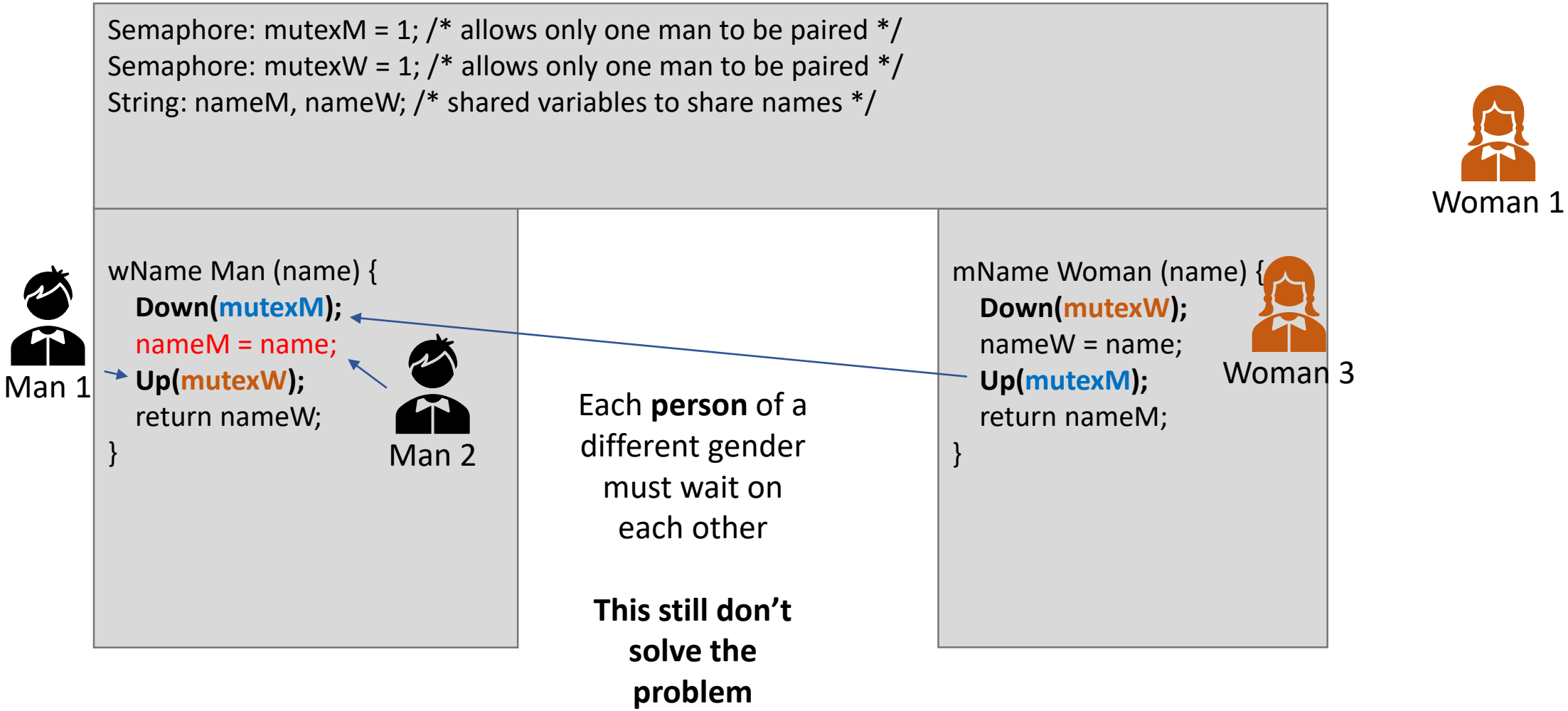
Midterm is approaching



Midterm is approaching



Midterm is approaching



Midterm is approaching

Man 2

Man 1

Woman 1

Woman 3

Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexW);
   return nameW;
}
```

We need to also that a woman can only return the name of a single man

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(mutexM);
   return nameM;
}
```

Midterm is approaching


Man 2


Woman 1

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0;/* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Down(waitW);
   Up(mutexW);
   return nameW;
}
```


Man 1

We need to also that a woman can only return the name of a single man

We needs processes to signal each other

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Down(waitM);
   Up(mutexM);
   return nameM;
}
```


Woman 3

Midterm is approaching


```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0;/* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Down(waitW);
   Up(mutexW);
   return nameW;
}
```

We need to also that a woman can only return the name of a single man

We needs processes to signal each other

Now each is waiting on each other **on deadlock**

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Down(waitM);
   Up(mutexM);
   return nameM;
}
```

Midterm is approaching


Man 2


Woman 1

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0;/* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = na;
   Up(waitM);
   Down(waitW);

   Up(mutexW);
   return nameW;
}
```


Man 1

We need to also that a woman can only return the name of a single man

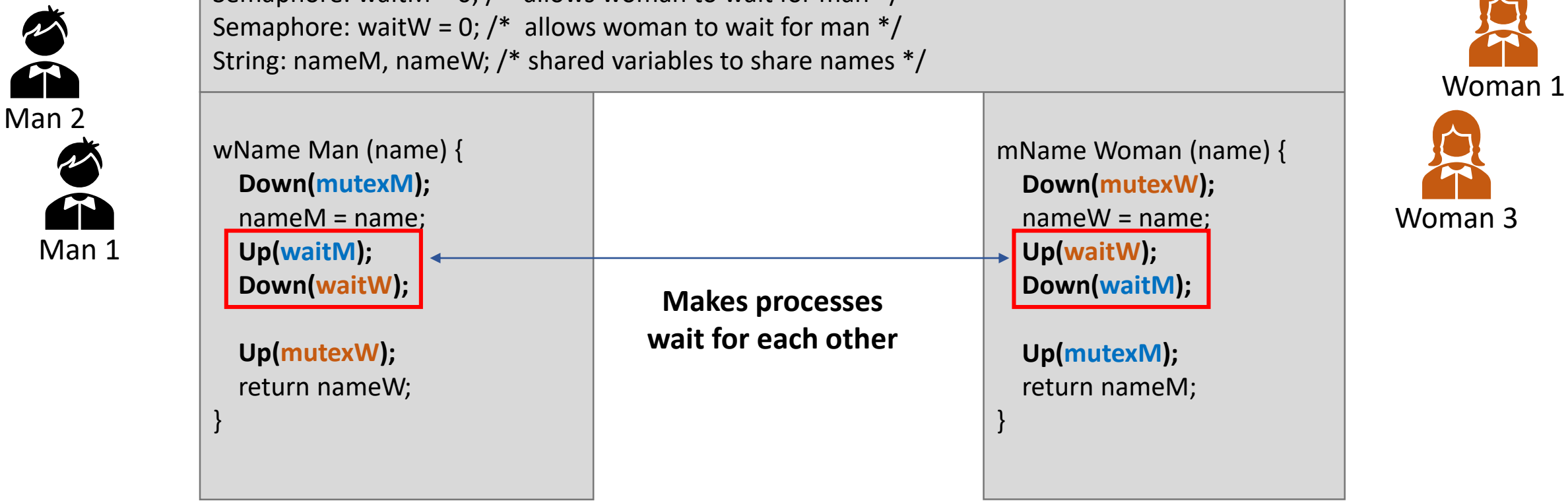
We needs processes to signal each other

Now each is waiting on each other **on deadlock**

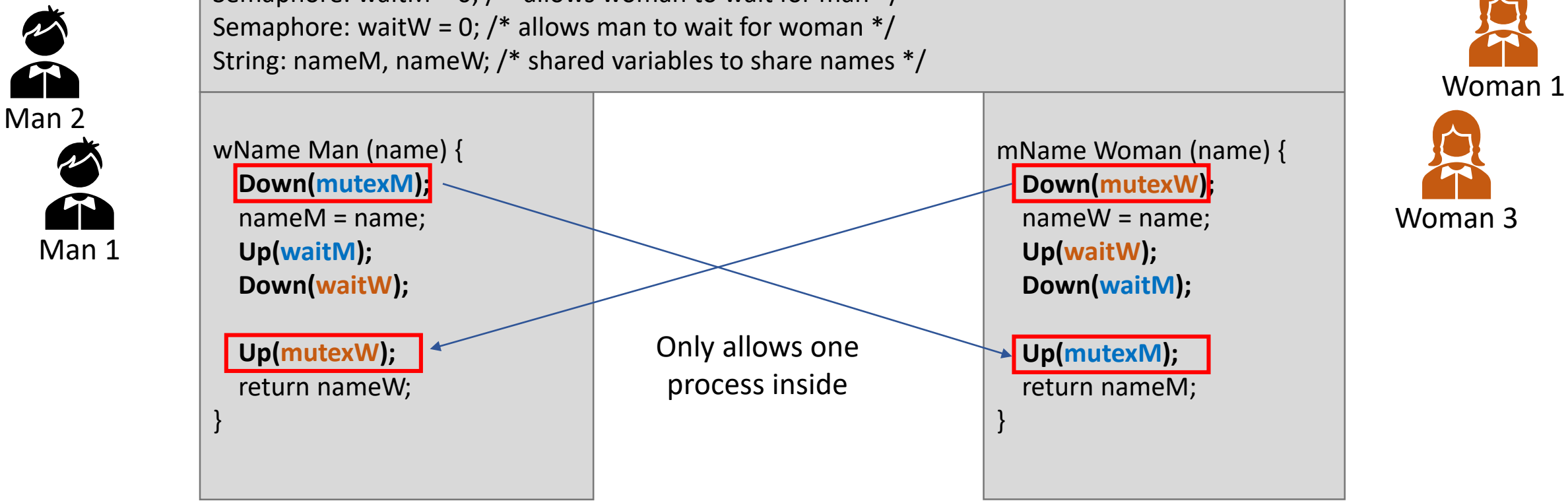
```
mName Woman (name) {
   Down(mutexW);
   nameW = n;
   Up(waitW);
   Down(waitM);

   Up(mutexM);
   return nameM;
}
```


Woman 3

Midterm is approaching



Midterm is approaching



Midterm is approaching


Man 2


Man 1

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0; /* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(waitM);
   Down(waitW);

   Up(mutexW);
   return nameW;
}
```

We still have a problem. We cannot return directly the shared **global** variable value. It mays **still be changed**.

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(waitW);
   Down(waitM);

   Up(mutexM);
   return nameM;
}
```


Woman 1


Woman 3

Midterm is approaching


Man 2


Man 1


Woman 1


Woman 3

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0; /* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   String temp;
   Down(mutexM);
   nameM = name;
   Up(waitM);
   Down(waitW);
   temp = nameW;
   Up(mutexW);
   return temp;
}
```

We still have a problem. We cannot return directly the shared **global** variable value. It mays **still be changed**.

It must be a local variable.

```
mName Woman (name) {
   String temp;
   Down(mutexW);
   nameW = name;
   Up(waitW);
   Down(waitM);
   temp = nameM;
   Up(mutexM);
   return temp;
}
```

Midterm is approaching

Man 2

Man 1

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0; /* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   String temp;
   Down(mutexM);
   nameM = name;
   Up(waitM);
   Down(waitW);
   temp = nameW;
   Up(mutexW);
   return temp;
}
```

Finally we have the solution!

```
mName Woman (name) {
   String temp;
   Down(mutexW);
   nameW = name;
   Up(waitW);
   Down(waitM);
   temp = nameM;
   Up(mutexM);
   return temp;
}
```

Woman 1

Woman 3

Midterm is approaching

- **1.** Implement a **binary semaphore** using the atomic exchange instruction (**XCHG**). Give the code for both **P**(mutex) and **V**(mutex).

Midterm is approaching

- **1.** Implement a **binary semaphore** using the atomic exchange instruction (**XCHG**). Give the code for both **P**(mutex) and **V**(mutex).

```
BinarySemaphore {
    Bit b;
} S;
```

Midterm is approaching

- **1.** Implement a **binary semaphore** using the atomic exchange instruction (**XCHG**). Give the code for both **P**(mutex) and **V**(mutex).

```
BinarySemaphore {
    Bit b;
} S;

P(S){
    Bit temp = 1;
    While (temp) XCHNG(temp, S.b);
}
```

Midterm is approaching

- **1.** Implement a **binary semaphore** using the atomic exchange instruction (**XCHG**). Give the code for both **P**(mutex) and **V**(mutex).

```
BinarySemaphore {
    Bit b;
} S;

P(S){
    Bit temp = 1;
    While (temp) XCHNG(temp, S.b);
}
V(S) {
    S.b = 0;
}
```

SpinLocks are Busy waiting

```
void lock(struct spinlock * lk) {
    while(xchg(&lk->locked, 1) != 0)
      ;
}
```

Locks – Processes sharing CPU

```
void
acquiresleep(struct sleeplock *lk)
{
}
```

Locks – Processes sharing CPU

```
void
acquiresleep(struct sleeplock *lk)
{
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
}
```

Locks – Processes sharing CPU

```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    release(&lk->lk);
}
```

Locks – Processes sharing CPU

```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    release(&lk->lk);
}
```

```
void
releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);

    wakeup(lk);

    release(&lk->lk);
}
```

Locks – Processes sharing CPU

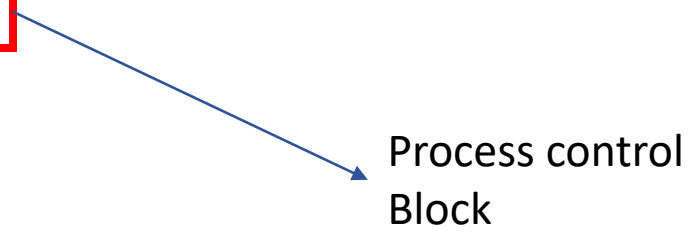
```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    release(&lk->lk);
}
```

Locks – Processes sharing CPU

```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    release(&lk->lk);
}
```

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();
    …
  p->chan = chan;
  p->state = SLEEPING;

  sched();
    …
}
```

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  …
}
```



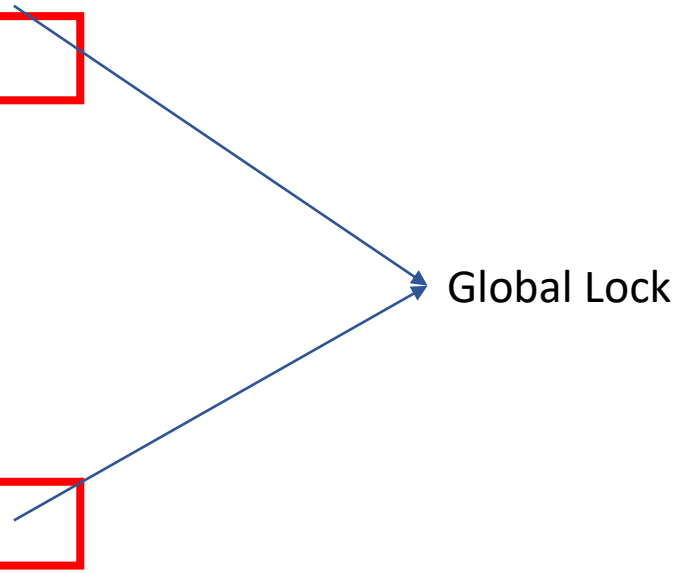
```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");
  if(lk == 0)
    panic("sleep without lk");

  …
}
```

Control checks. This should be impossible

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");
  if(lk == 0)
    panic("sleep without lk");

  …
  p->chan = chan;
  p->state = SLEEPING;

  sched();
  …
}
```

Change process state to sleep. Call scheduler

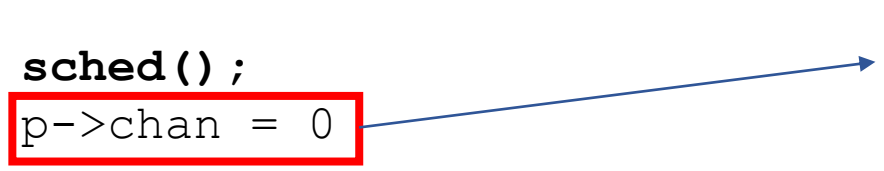
```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");
  if(lk == 0)
    panic("sleep without lk");


  acquire(&ptable.lock);

  p->chan = chan;
  p->state = SLEEPING;

  sched();
  p->chan = 0

  release(&ptable.lock);

}
```



```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");
  if(lk == 0)
    panic("sleep without lk");


  acquire(&ptable.lock);

  p->chan = chan;
  p->state = SLEEPING;

  sched();
  p->chan = 0

  release(&ptable.lock);

}
```

Once sched() is called this process execution is held “at this line”

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");
  if(lk == 0)
    panic("sleep without lk");


  acquire(&ptable.lock);

  p->chan = chan;
  p->state = SLEEPING;

  sched();
  p->chan = 0

  release(&ptable.lock);

}
```



When process awakes, he is removed from the sleep channel

Locks – Processes sharing CPU

```
void
releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);

    wakeup(lk);

    release(&lk->lk);
}
```

Locks – Processes sharing CPU

```
void
releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);

    wakeup(lk);

    release(&lk->lk);
}
```

```
void
wakeup(void *chan)
{
    acquire(&ptable.lock);

    wakeup_channel(chan);

    release(&ptable.lock);
}
```

Locks – Processes sharing CPU

```
static void
wakeup_channel(void *chan)
{
    struct proc *p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

}
```

A processes is awaken from the sleeping channel

Locks – Processes sharing CPU

```
static void
wakeup_channel(void *chan)
{
    struct proc *p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)

}
```

Locks – Processes sharing CPU

```
static void
wakeup_channel(void *chan)
{
    struct proc *p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
}
```

Locks – Processes sharing CPU

```
static void
wakeup_channel(void *chan)
{
    struct proc *p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
}
```

Locks – Processes sharing CPU

- Who needs to be a syscall?
  - SpinLocks
  - Sleep/Wakeup

CS 1550 – Lab exercise 2

- **PROCESS SYNCHRONIZATION IN XV6**
  - **Due**: Monday, February 17, 2018 @11:59pm

  - Part 2 - step 5: user.h
    - Add declaration for init_lock()
      - void init_lock(struct spinlock *);
    - struct condvar;
    - struct spinlock;

  - Part 3 - step 8: defs.h
    - Add declaration for sleep1()

CS 1550

Week 6

Project 1 Quiz & Midterm prep

Teaching Assistant

Henrique Potter