Problems with Semaphores

- Used for 2 independent purposes
  - Mutual exclusion
  - Condition synchronization
- Hard to get right
  - Small mistake easily leads to incorrect solutions, including deadlock

May want to separate mutual exclusion, condition synchronization

Monitors (Hoare)

- Abstract Data Type
  - a class (as are locks and semaphores)
  - 3 key differences from a regular class:
    - **only one thread in monitor at a time (mutual exclusion is automatic)**
    - special type of variable allowed, called "**condition variable**"
      - 4 special ops allowed only on condition variables: wait, signal, broadcast, notempty
    - no public data allowed (must call methods to effect any change)

Wait, Signal, Broadcast, Notempty

- Given a condition variable *c*
  - Wait(*c*):
    - Thread is put on queue for *c*, goes to sleep
    - Releases control of the monitor
  - Signal(*c*):
    - If queue for *c* not empty, wake up one thread
    - Has no effect if no threads are waiting
  - Broadcast(*c*):
    - Wake up all threads waiting on queue for *c*
  - Notempty(*c*):
    - Return false if queue for c is empty; true otherwise

The Multiple Semantics of Signal

- Signal and Urgent Wait (Hoare) (SU)
  - signaler immediately gives up control
  - thread that was waiting executes in monitor
  - signaler executes before new threads
- Signal and Continue (Mesa, Java) (SC)
  - **will be used in this class** unless otherwise stated
  - signaler continues executing
  - thread that was waiting put on ready queue
  - when thread is scheduled and re-enters monitor:
    - **state may have changed!** use “while”, not “if”

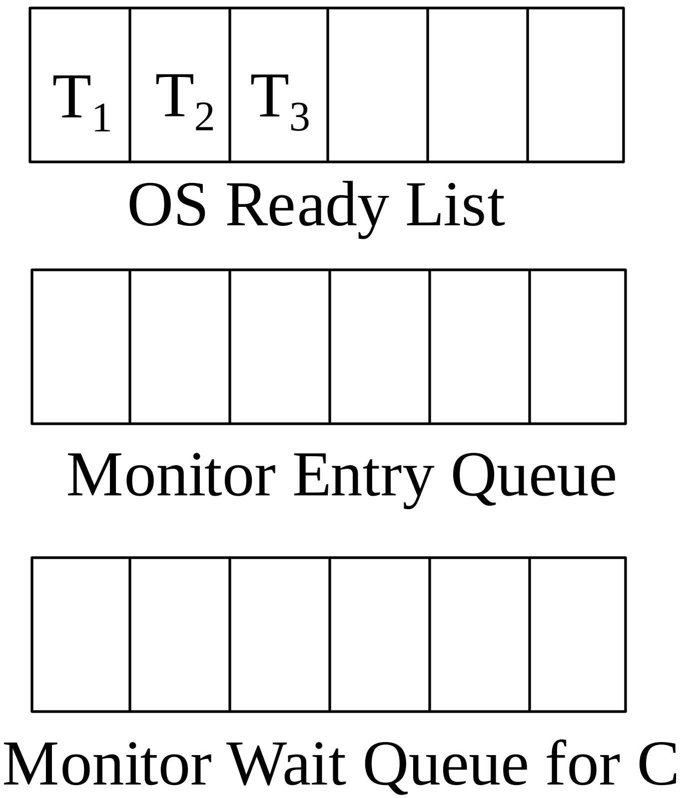
The Multiple Semantics of Signal

- Signal and Wait (SW)
  - Same as Signal and Urgent Wait, except that signaler has no priority over new threads trying to enter
- Signal and Exit (SX)
  - Signaler exits monitor
  - Means that signal must be the last operation done in each monitor function

SU and SW can cause programming difficulty:

Example: an alarm--cannot broadcast

Animation of Monitor Operation (Uses Signal and Continue)

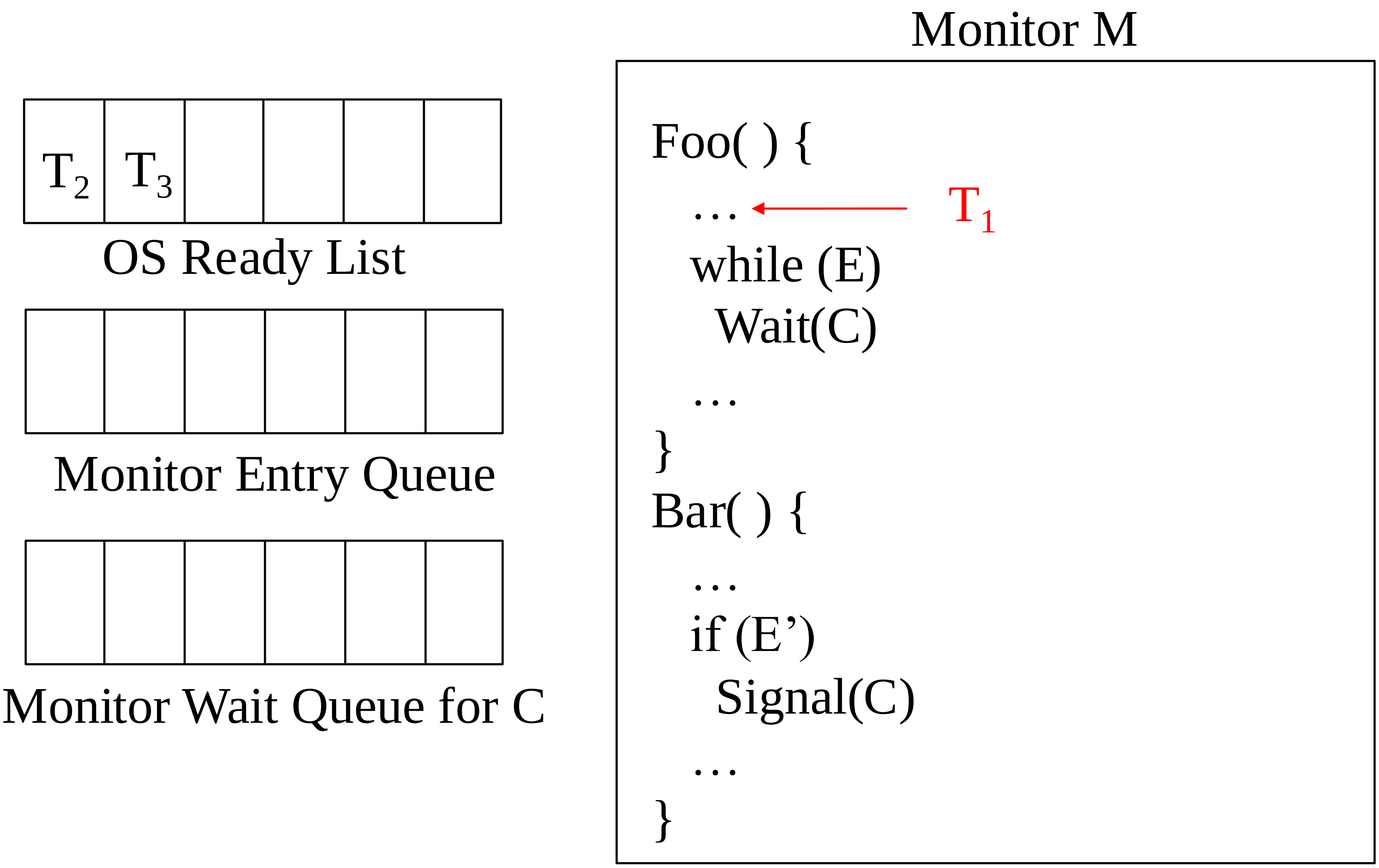


Monitor M

```
Foo( ) {
   …
   while (E)
     Wait(C)
   …
}
Bar( ) {
   …
   if (E')
     Signal(C)
   …
}
```

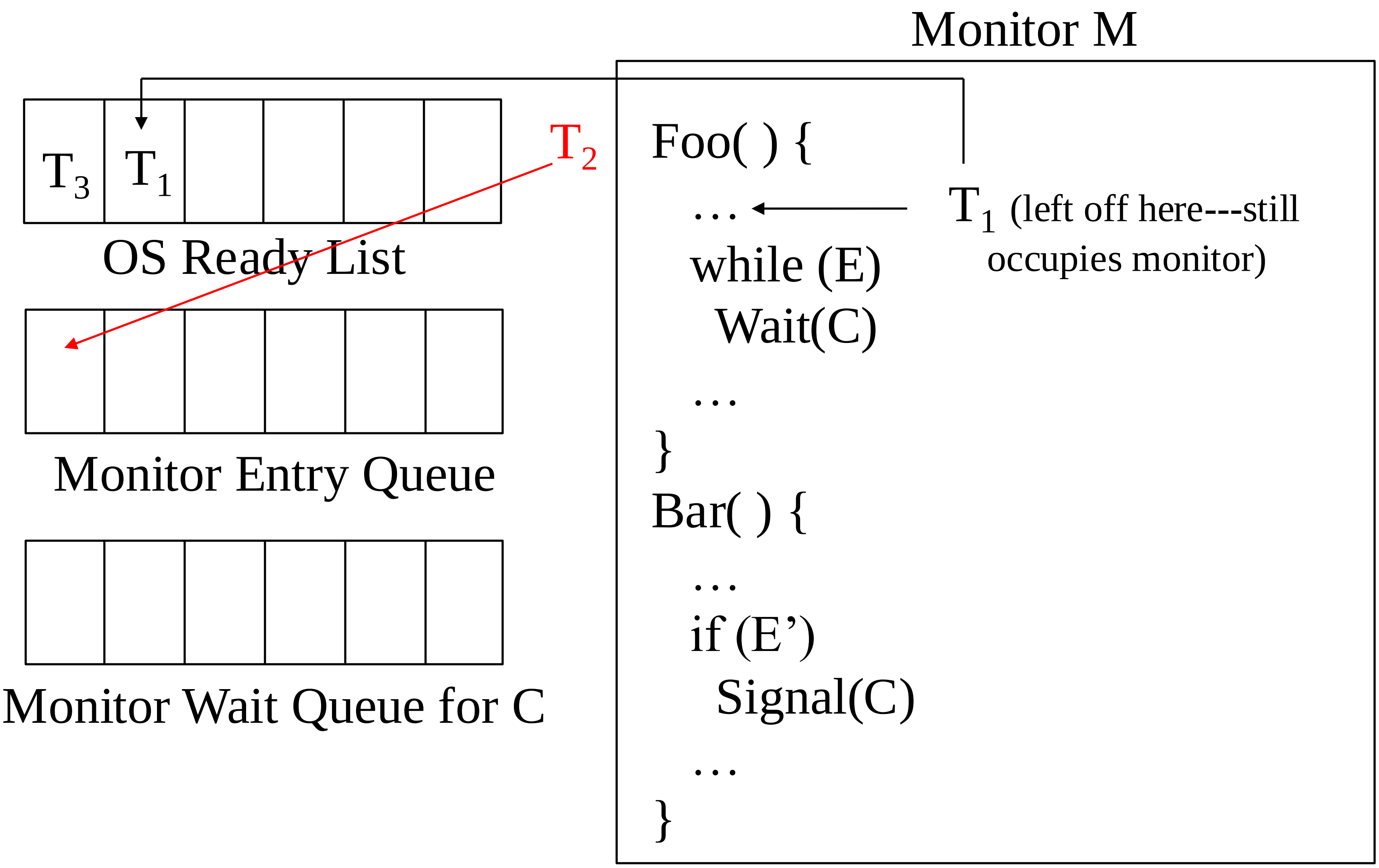
Initial state: T_1, T_2, T_3 on ready list; T_1,T_2 will call Foo; T_3 will call Bar

Animation of Monitor Operation (Uses Signal and Continue)



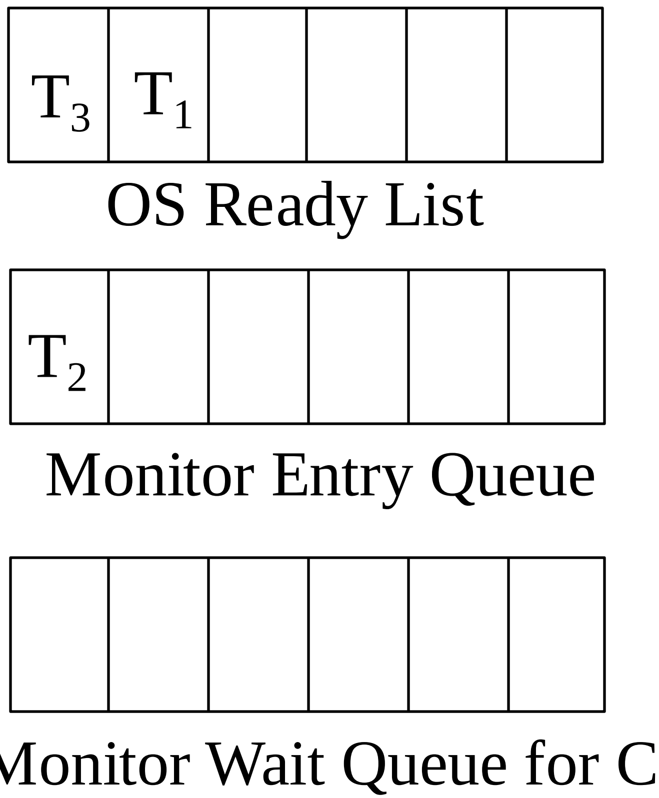
T_1 selected to run and invokes M.Foo( ); enters M

Animation of Monitor Operation (Uses Signal and Continue)



Context switch to T_2,which invokes M.Foo( ); goes to entry queue and blocks because T_1 still occupies M. Note that the Entry Queue is **outside** M.

Animation of Monitor Operation (Uses Signal and Continue)



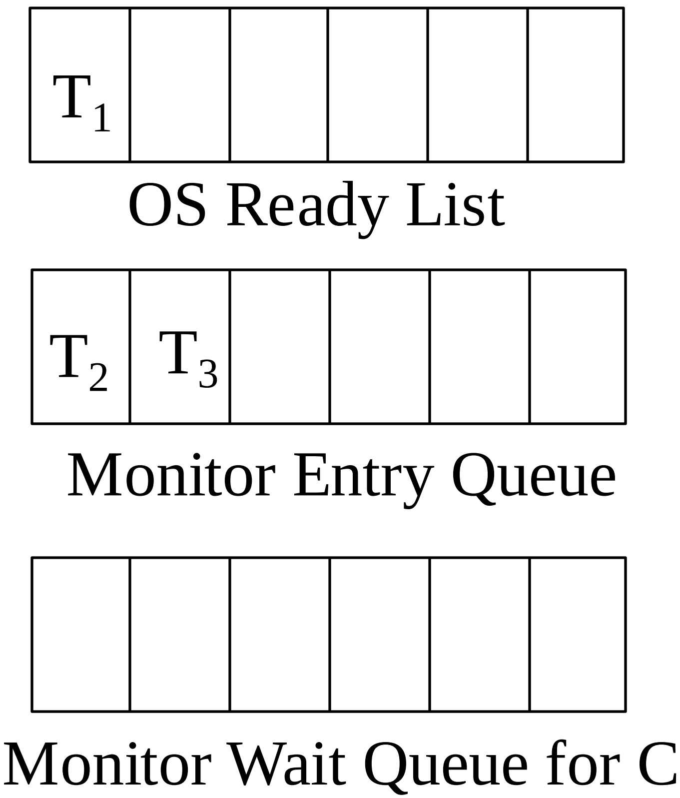
Monitor M

```
Foo( ) {
   …
   while (E)
     Wait(C)
   …
}
Bar( ) {
   …
   if (E')
     Signal(C)
   …
}
```

T_1 (left off here---still occupies monitor)

T_1 is both on ready list **and** occupying the monitor.

Animation of Monitor Operation (Uses Signal and Continue)

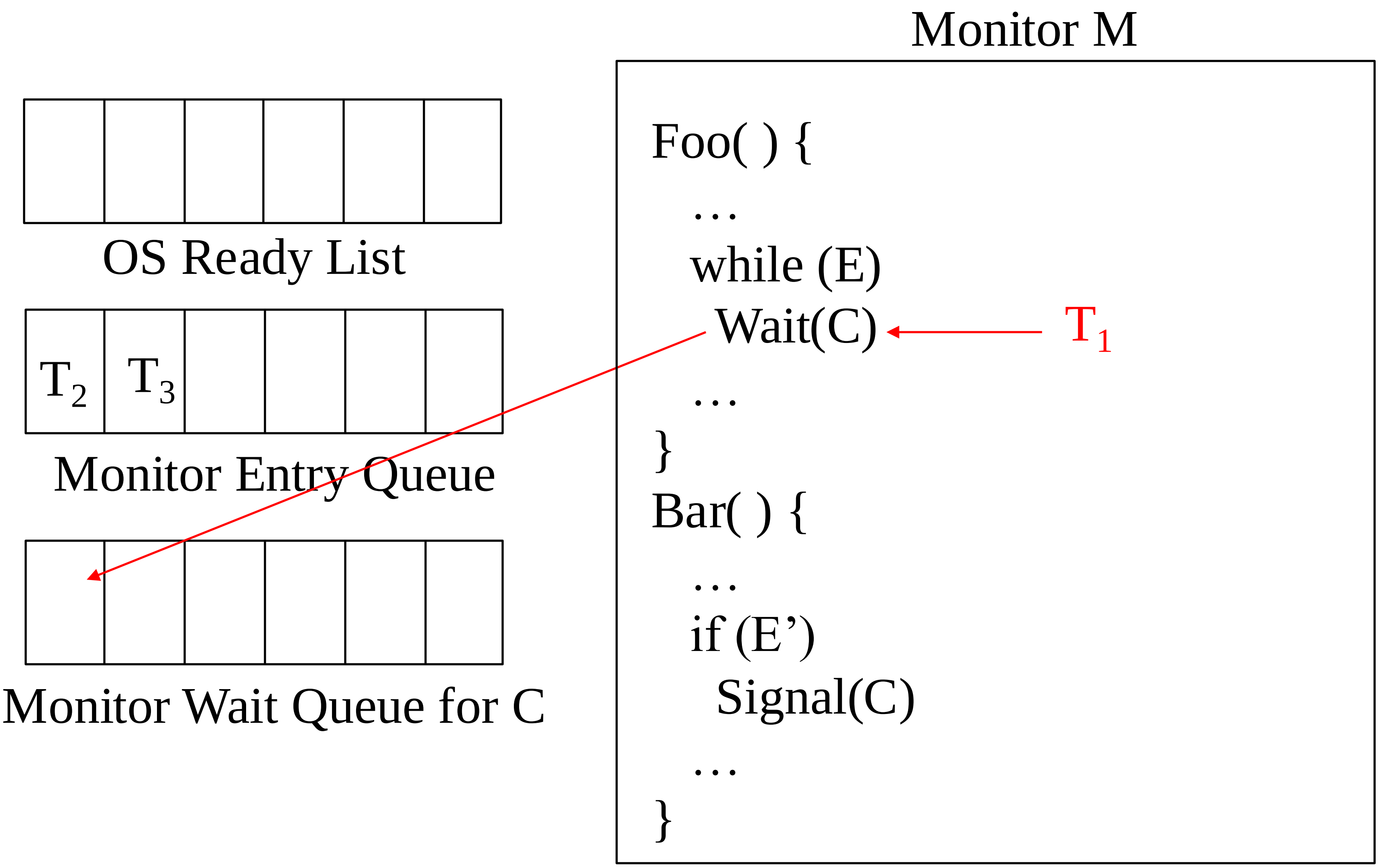


Monitor M

```
Foo( ) {
   …  <-- T1 (left off here---still occupies monitor)
   while (E)
    Wait(C)
   …
}
Bar( ) {
   …
   if (E')
    Signal(C)
   …
}
```

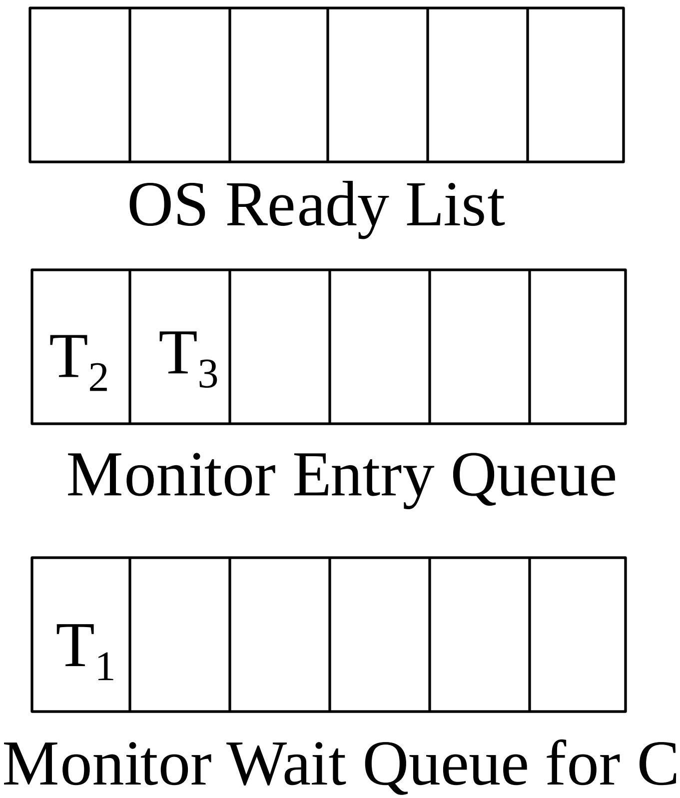
Context switch to T_3, which executes M.Bar( ); but, it can't gain control of the monitor and is also placed on monitor entry queue

Animation of Monitor Operation (Uses Signal and Continue)



Context switch to T_1; invokes Wait(C); goes to wait queue and blocks

Animation of Monitor Operation (Uses Signal and Continue)

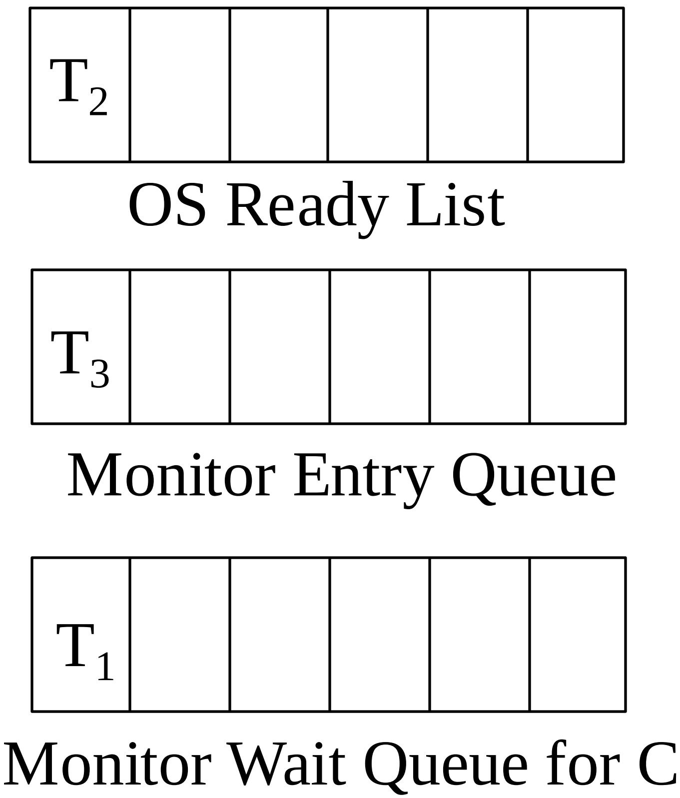


Monitor M

```
Foo( ) {
   …
   while (E)
     Wait(C)
   …
}
Bar( ) {
   …
   if (E’)
     Signal(C)
   …
}
```

Note that the Wait Queue is **outside** M.

Animation of Monitor Operation (Uses Signal and Continue)

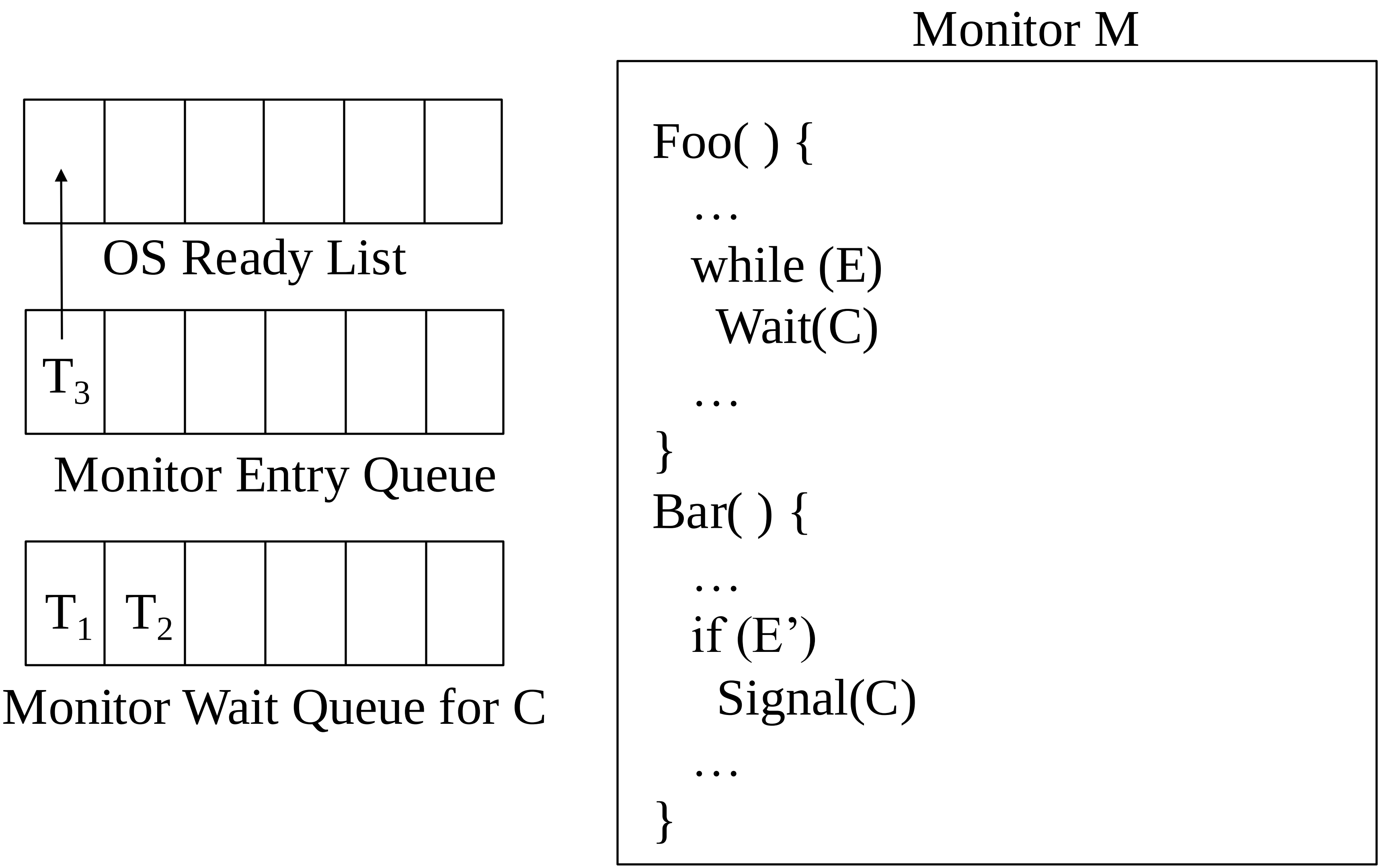


Monitor M

```
Foo( ) {
   …
   while (E)
     Wait(C)
   …
}
Bar( ) {
   …
   if (E')
     Signal(C)
   …
}
```

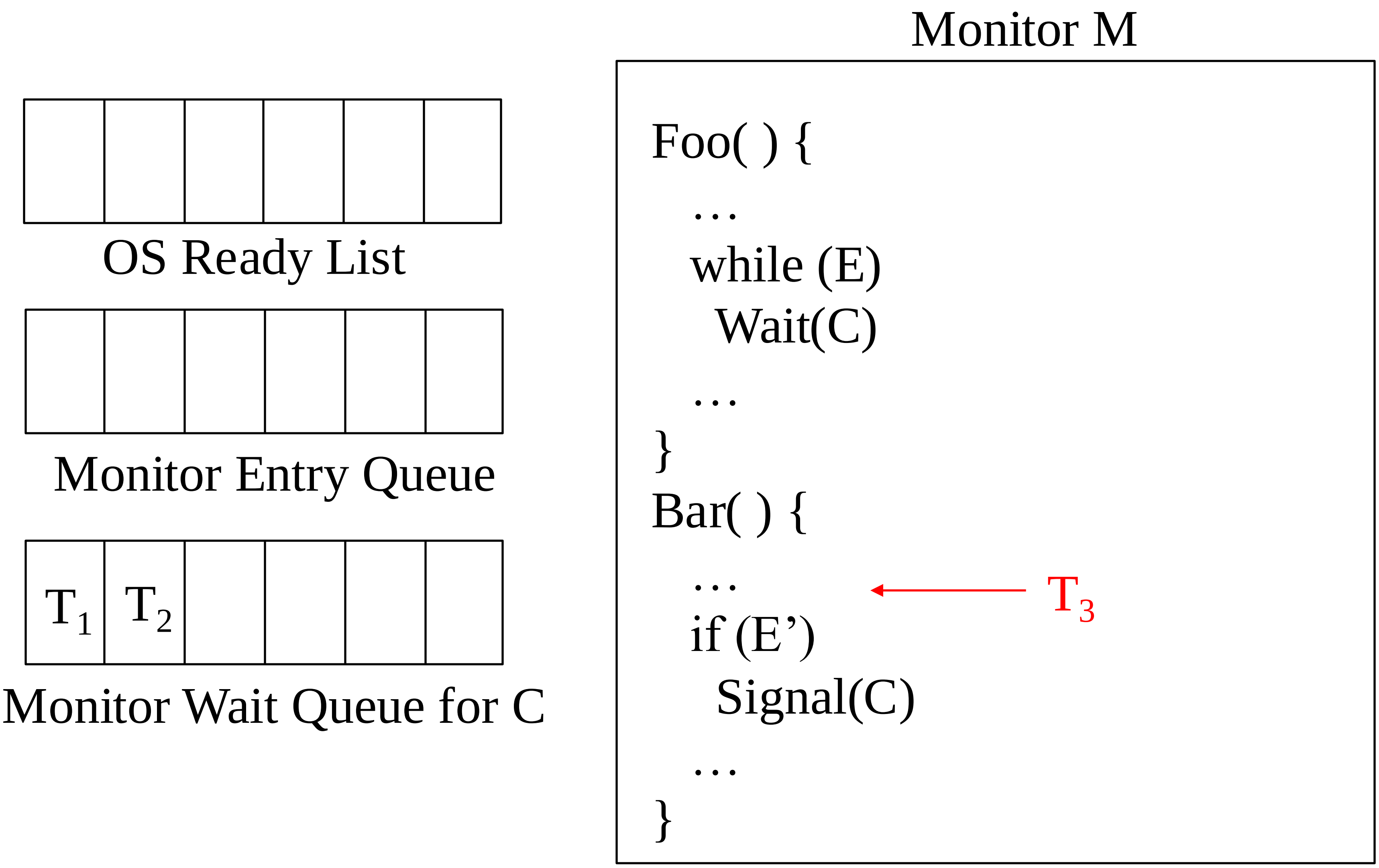
T_1 leaves monitor, so T_2 moved to OS ready list (and will run)

Animation of Monitor Operation (Uses Signal and Continue)



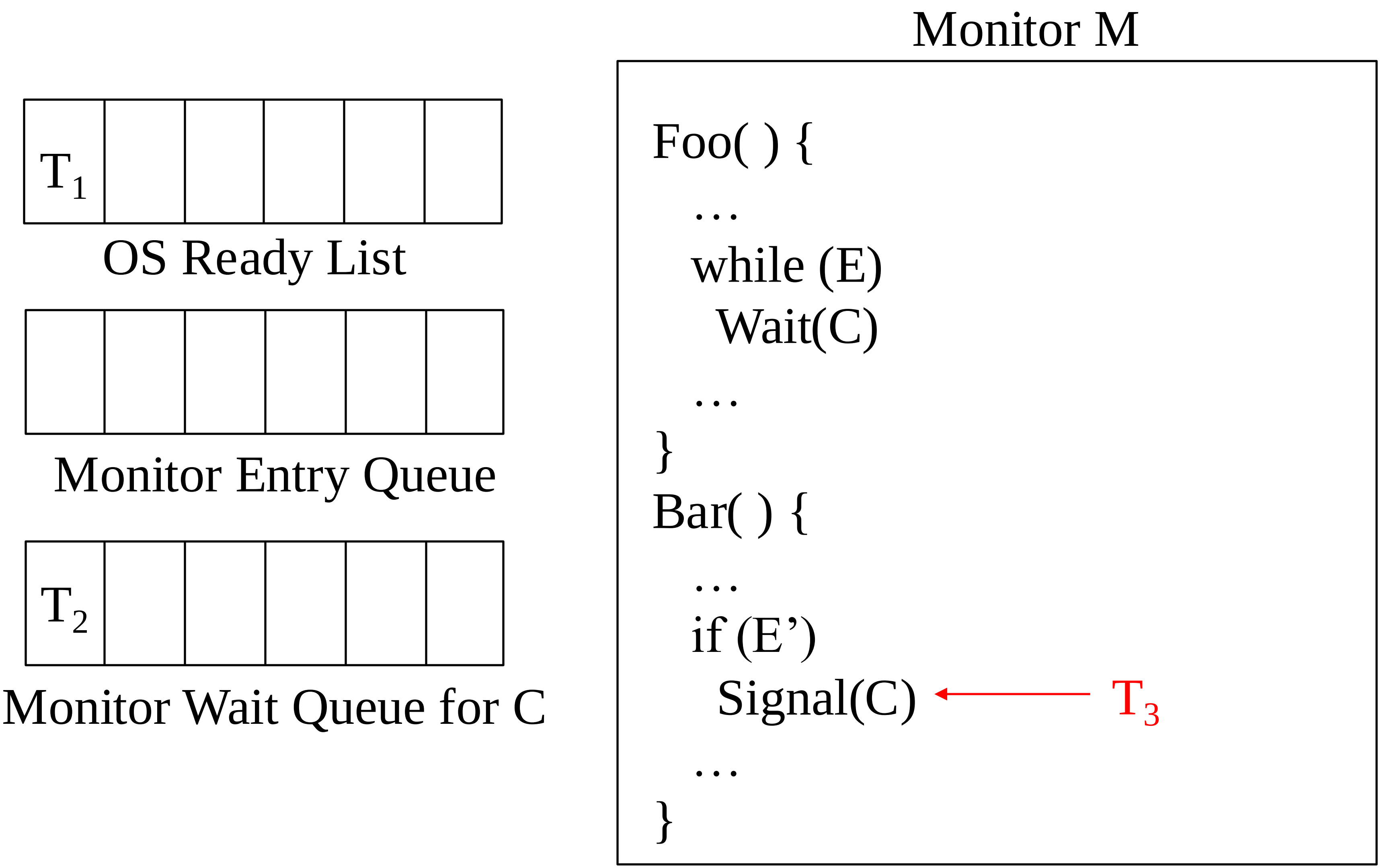
T_2 is now granted access to M, executes Foo, hits Wait(C) and blocks

Animation of Monitor Operation (Uses Signal and Continue)



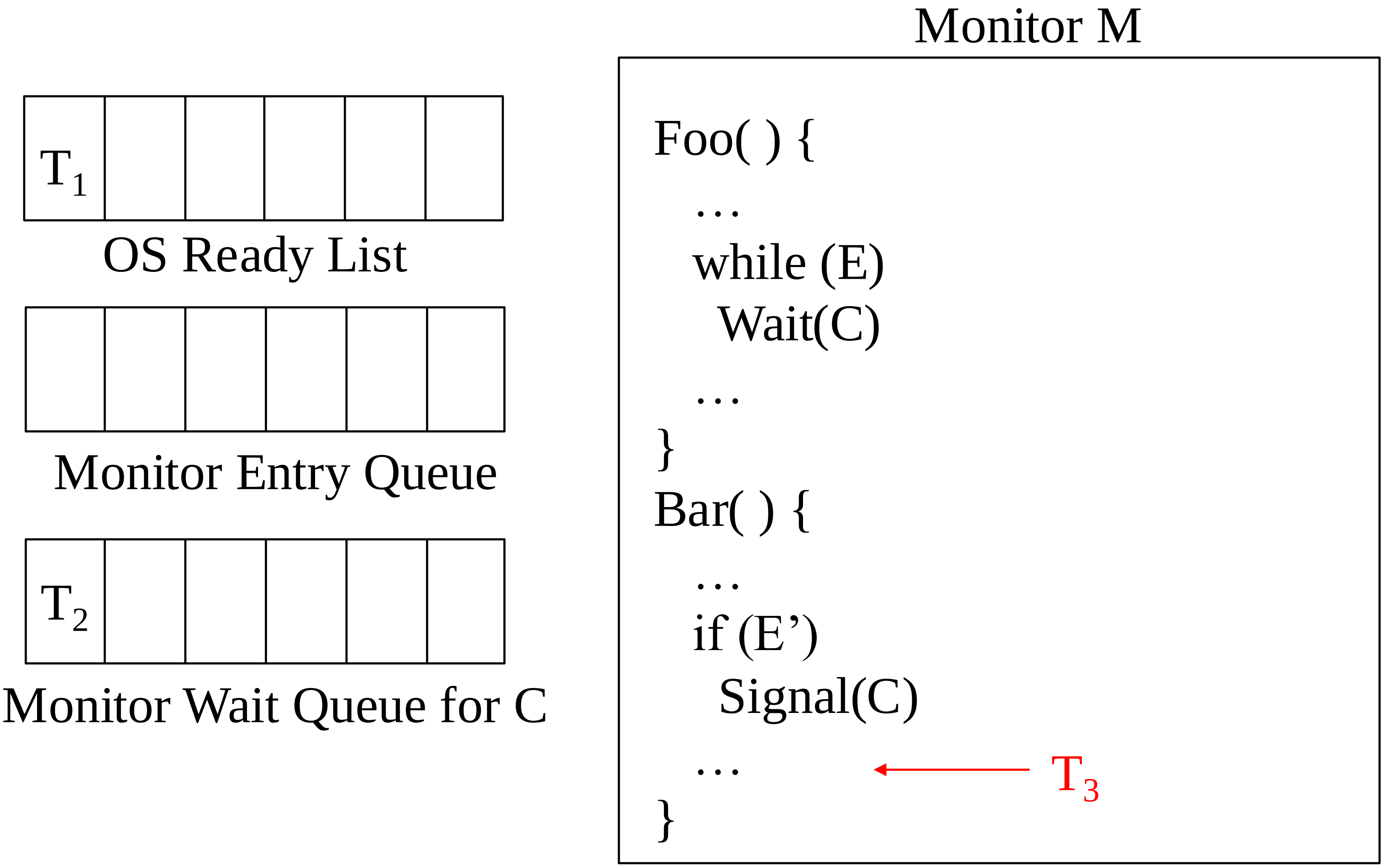
T_3 selected to run and invokes M.Bar( ); enters M

Animation of Monitor Operation (Uses Signal and Continue)



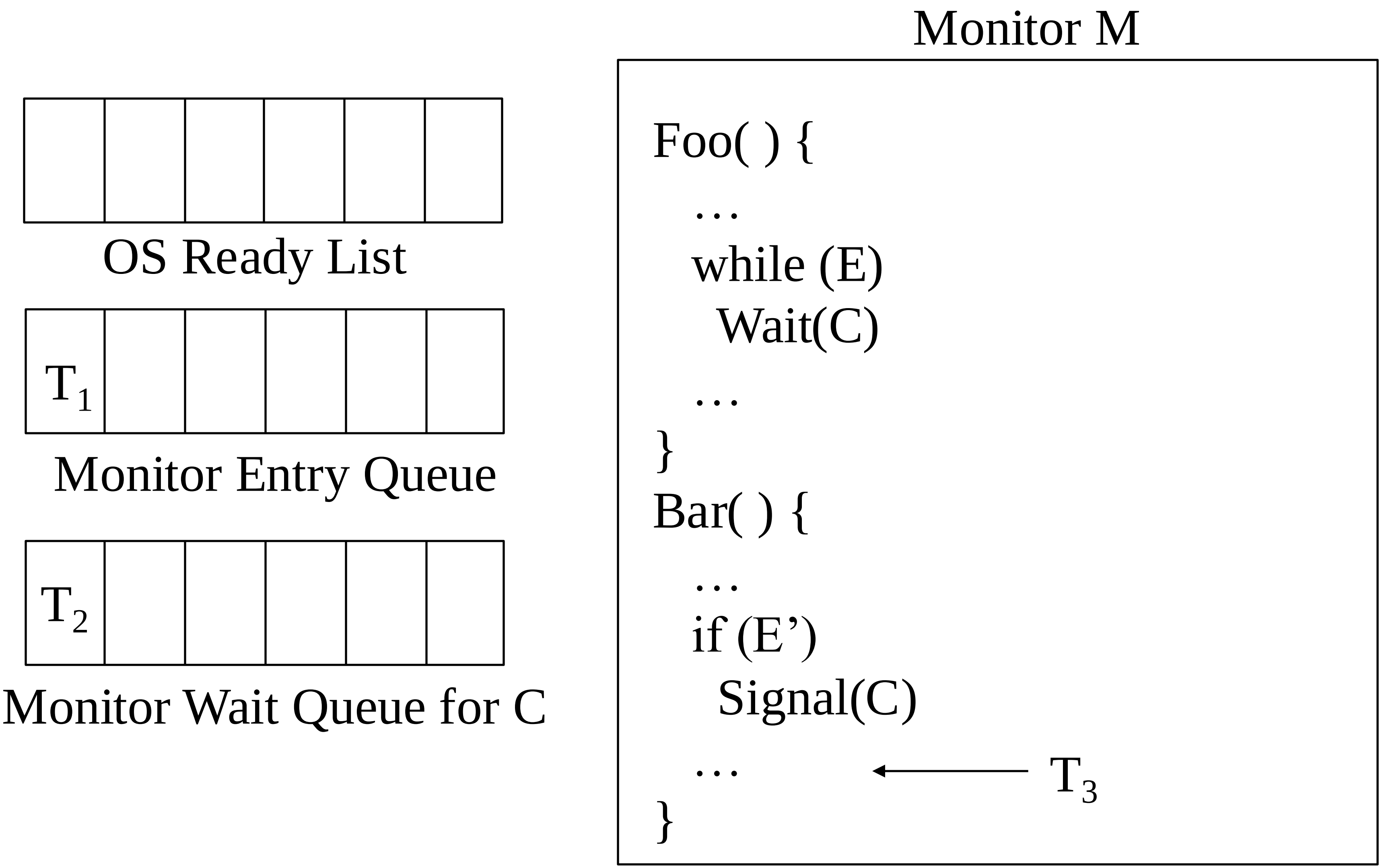
T_3 invokes Signal(C), which moves T_1 to the ready list

Animation of Monitor Operation (Uses Signal and Continue)



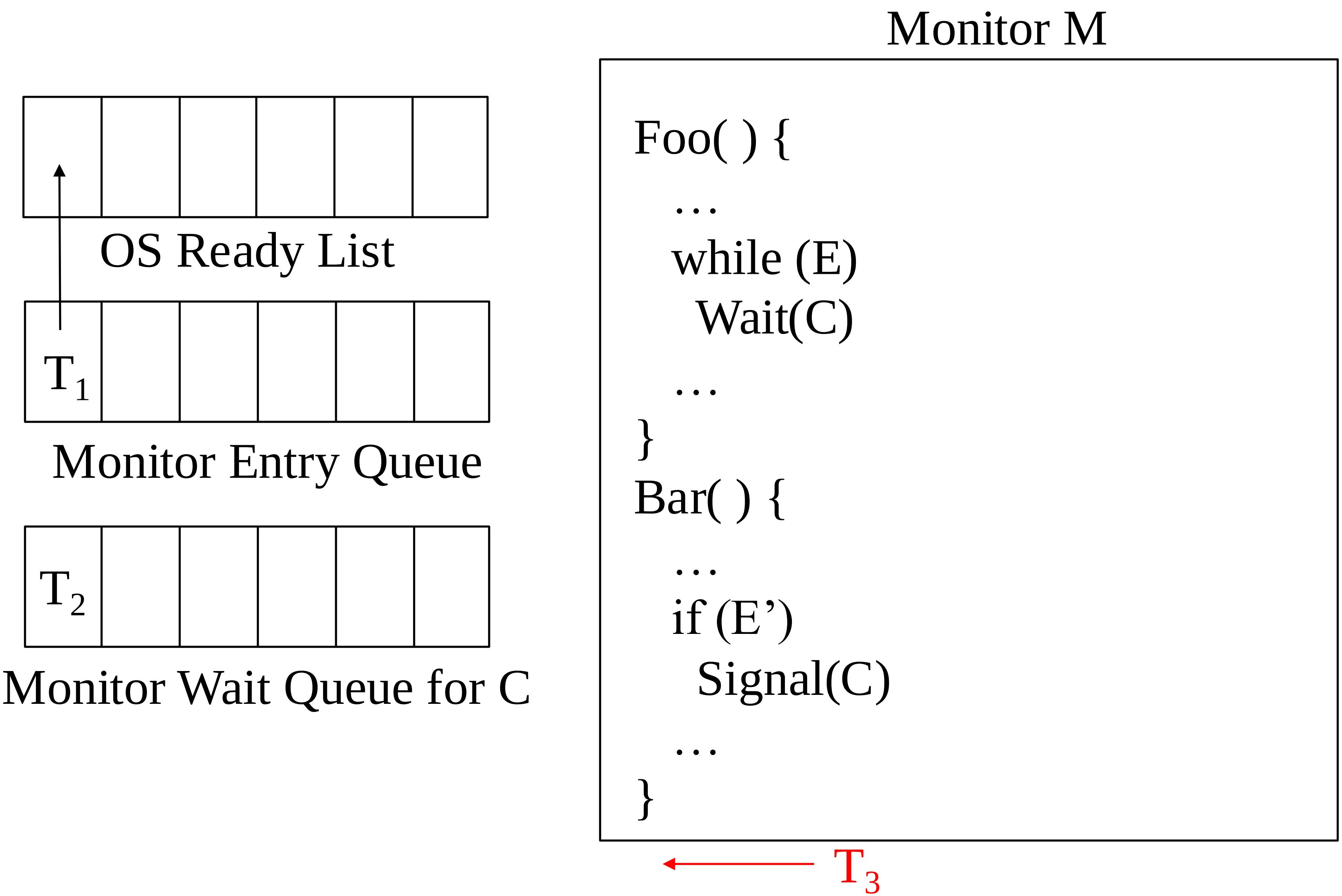
T_3 retains control of M because of Signal and Continue semantics

Animation of Monitor Operation (Uses Signal and Continue)



Context switch to T_1 but it cannot enter M; goes to entry queue and blocks

Animation of Monitor Operation (Uses Signal and Continue)



T_3 completes M.Bar( ); exits monitor (completed function); T_1 moved to Ready List

Animation of Monitor Operation (Uses Signal and Continue)

OS Ready List

Monitor Entry Queue

T_2

Monitor Wait Queue for C

Monitor M

```
Foo( ) {
   …
   while (E)          ← T1
     Wait(C)
   …
}
Bar( ) {
   …
   if (E’)
     Signal(C)
   …
}
```

T_1 restarts in M.Foo( ) at line after Wait(C), which is rechecking E

Monitor Solution to Critical Section

- Just make the critical section a monitor routine!

Differences between Monitors and Semaphores

- Monitors enforce mutual exclusion
- P( ) vs Wait
  - P blocks if value is 0, Wait always blocks
- V( ) vs Signal
  - V either wakes up a thread or increments value
  - Signal only has effect if a thread waiting
- Semaphores have “memory”

Readers/Writers Solution using Monitors

- Similar idea to semaphore solution
  - simpler, because don't worry about mutex
- When can't get into database, wait on appropriate condition variable
- When done with database, signal others

Note: can't just put code for "reading database" and code for "writing database" in the monitor (couldn't have >1 reader)

Implementing semaphores using monitors---unfair (but correct) solution

```
monitor SemaphoreImplementation
  int s = INIT_VAL; cond c
  P( ): while (s == 0)
             wait(c)
           s--
  V( ): s++; signal(c)
end SemaphoreImplementation
```

Implementing semaphores using monitors---fair solution

```
monitor SemaphoreImplementation
  int s = INIT_VAL; cond c
  P( ): if (s == 0) wait(c)
           else s--
  V( ): if (empty(c)) s++
           else signal(c)
end SemaphoreImplementation
```

Solution style known as *passing the condition*

First Attempt: Implementing Monitors using Semaphores

Shared vars:

- sem mutex := 1 (one per monitor)
- sem c := 0; int nc := 0 (both *c*, *nc* are per condition var)

Monitor entry: P(mutex)

Wait(c, mutex):

- nc++; V(mutex); P(c); P(mutex)

Signal(c, mutex):

- if (nc > 0) then {nc--; V(c);}

Monitor exit: V(mutex)

Correct Implementation of Monitors using Semaphores (Assume that "tid" is the id of a thread)

Shared vars:

- sem mutex := 1; (one per monitor)
- int nc := 0; List delayQ (one per condition var)
- sem c[NumThreads] := 0; (one entry per thread; one entry per thread per condition works also)

Monitor entry: P(mutex)

Wait(c, mutex):

```
nc++; Append(delayQ, tid); V(mutex);  P(c[tid]); P(mutex)
```

Signal(c, mutex):

```
if (nc > 0) then {nc--; id = Remove(delayQ); V(c[id]);}
```

Monitor exit: V(mutex);

Semaphores and Monitors Have Equal Power

- We just showed (on the last four slides) that:
  - Semaphores can be implemented using monitors
  - Monitors can be implemented using semaphores

Java-style monitors

- Integrated into the class mechanism
  - annotation “synchronized” can be applied to a member function
    - this function executes with implicit mutual exclusion with respect to all other functions annotated with “synchronized”
  - “synchronized” can also refer to a block
  - Wait and Signal are called Wait and Notify, respectively
    - Java’s Notify uses Signal and Continue semantics

Differences between traditional monitors and Java-style monitors

- Traditional
  - all functions synchronized
  - no public data
  - separate construct
    - simpler to implement (i.e. no inheritance)
  - safer
    - e.g., can statically guarantee no race conditions, because no public data
  - less flexible
- Java-style
  - can mix and match
  - public data allowed
  - integrated with class
    - interaction with rest of language, i.e. inheritance?
  - riskier
    - can circumvent the monitor idea by using and modifying public data
  - more flexible

Rendezvous (two-thread barrier) with semaphores

sem a = 0, b = 0

Thread 1:	Thread 2:
V(a)	V(b)
P(b)	P(a)

Can the V and P operations be inverted?

In-Class Exercise

- Implement a two-thread barrier using monitors
  - Hint: use the *notempty* function

Rendezvous with monitors---Attempt

cond a, b

Thread 1:

Signal(a)

Wait(b)

Thread 2:

Signal(b)

Wait(a)

(Assume the above code is in a monitor, and each thread is calling a unique function)

What's wrong with this?

Rendezvous with monitors---correct

cond a, b

Thread 1:

```
if (!empty(a))
  Signal(a)
else
  Wait(b)
```

Thread 2:

```
if (!empty(b))
  Signal(b)
else
  Wait(a)
```

(Assume the above code is in a monitor, and each thread is calling a unique function)

Tricky---easier to program rendezvous with semaphores

Alternate rendezvous with monitors

```
        cond a, b
        int ar1 = 0, ar2 = 0
Thread 1:                         Thread 2:
  ar1 = 1                           ar2 = 1
  Signal(a)                         Signal(b)
  while (!ar2)                      while (!ar1)
    Wait(b)                           Wait(a)
  ar2 = 0                           ar1 = 0
```

(Assume the above code is in a monitor, and each thread is calling a unique function)

Even less intuitive than the previous slide's solution