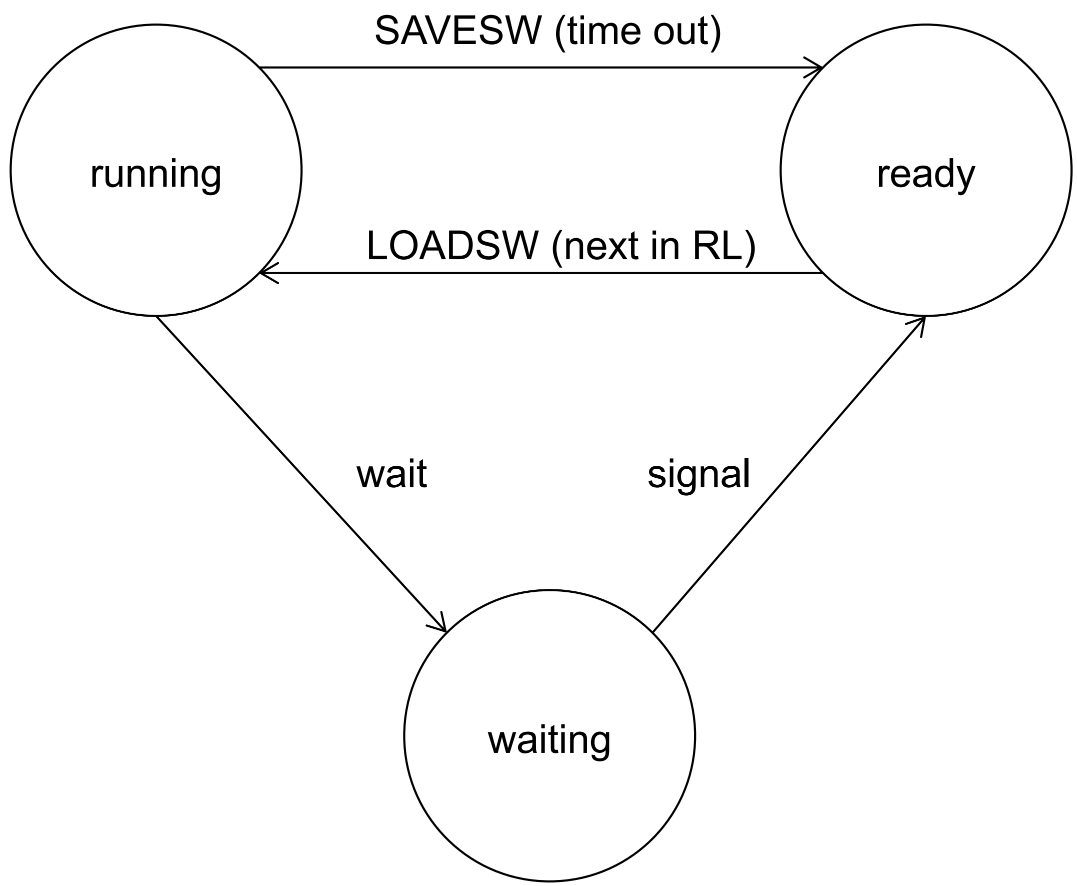
Semaphores

Peter J. Denning

Process Waiting

- Most processes must coordinate with one another and with the user.
- Signals and message exchange are the basic tools for coordination among processes.
- Need method to make process wait until a signal (or message) comes from another process.

Process States



Examples

- Performer waiting for customer to make request
- ATM waiting for someone to make transaction
- Traffic signal waiting for car to arrive
- SETI central waiting SETI slaves to send results
- Consumer waiting for next items to appear in the buffer (or pipe)
- Producer waiting for next empty space in the buffer (or pipe)

Semaphore

- An object used to allow a process to wait on a condition, represented by a semaphore s.
- Two operations:
  - WAIT(s):
    - caller asks if condition s is true
    - if condition true, pass without waiting
    - If condition false, go to sleep
  - SIGNAL(s):
    - caller signals that the condition s has come true
    - awakens a sleeping process, if any
    - does not wait

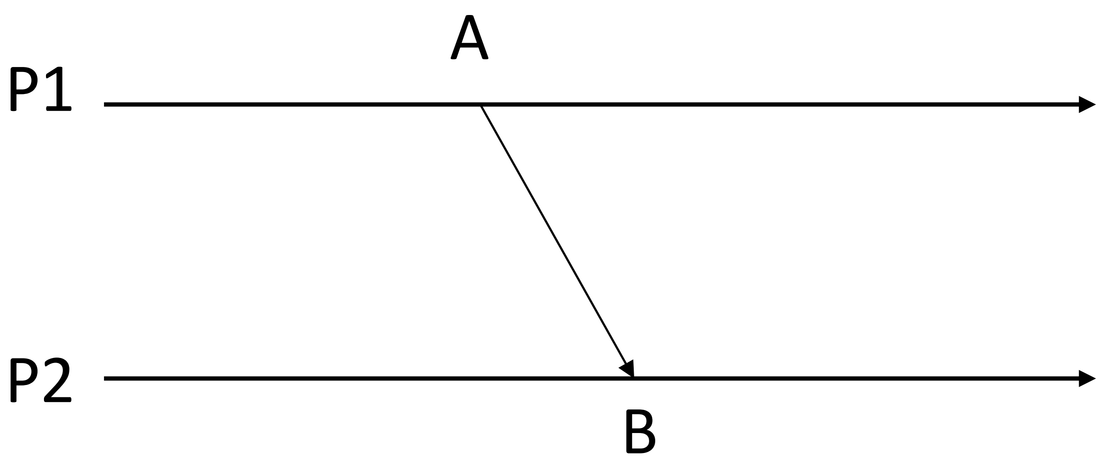
Semaphore Mechanics

- Semaphore s contains a counter and a queue.
  - Queue is FIFO list of processes waiting on semaphore
    - Head, tail, links threaded through PCBs
  - Count means
    - If <0, (absolute) count is number waiting
    - If ≥0, count is number of free passes
  - Initially counter has value ≥0, queue empty

Semaphore -- Notation

- Semaphore s [k]
- [k] means initial count is k ($k \geq 0$)
- s.c = counter
- s.q = queue
- WAIT(s)
- SIGNAL(s)

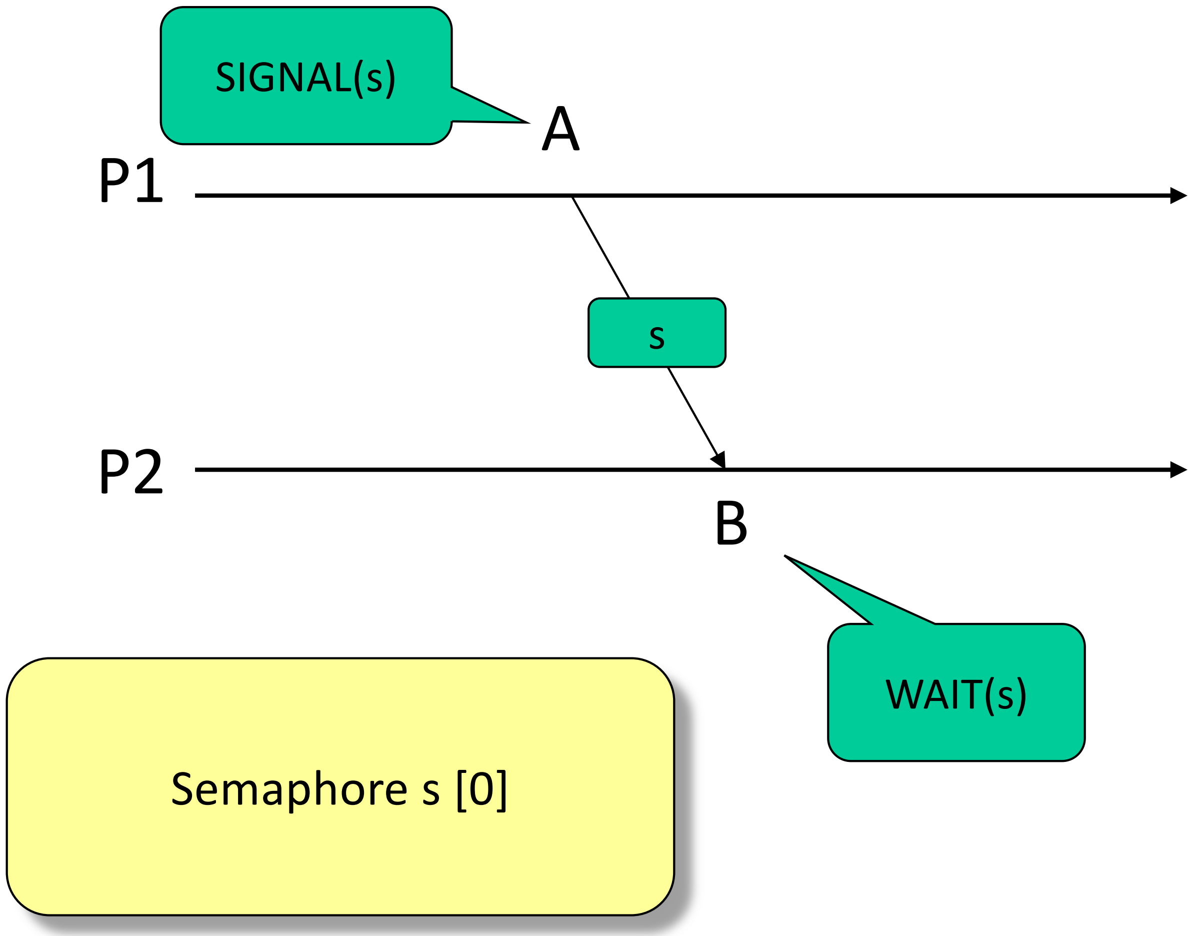
Basic Idea: synchronization “A before B”



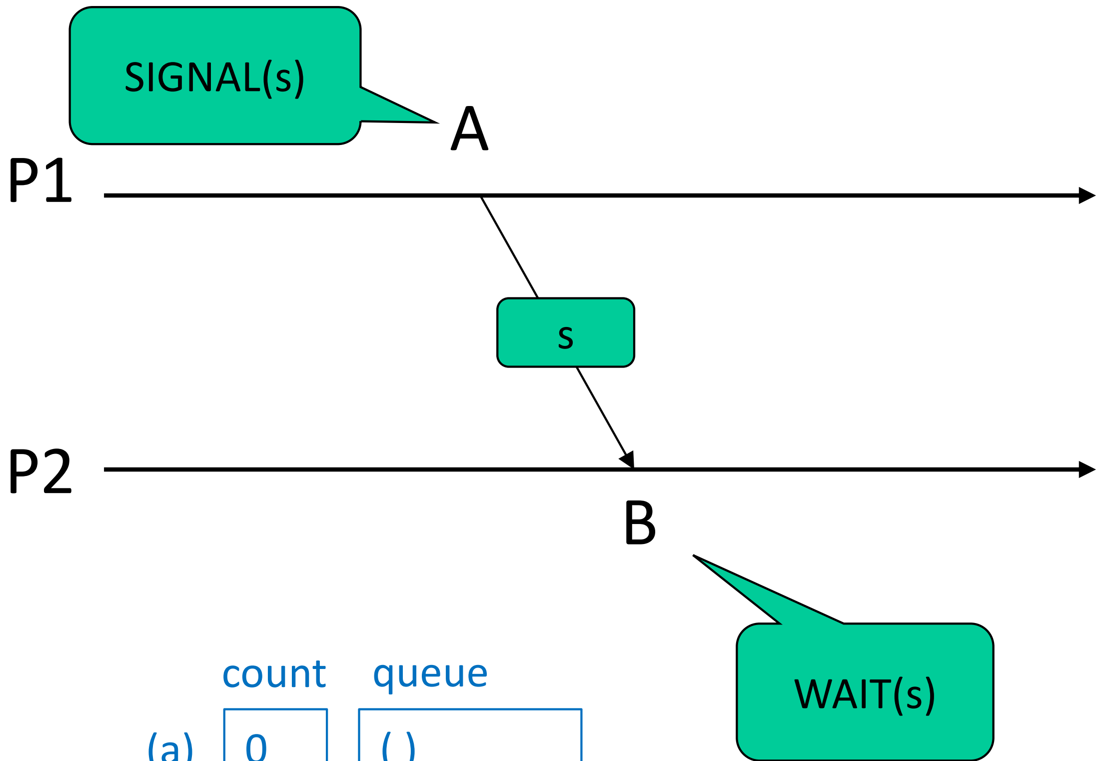
“P2 cannot pass B until P1 passes A”

Commonly, A is the EXIT of P1
and B is the ENTRY of P2:
sequencing processes

Semaphores can implement basic synchronization



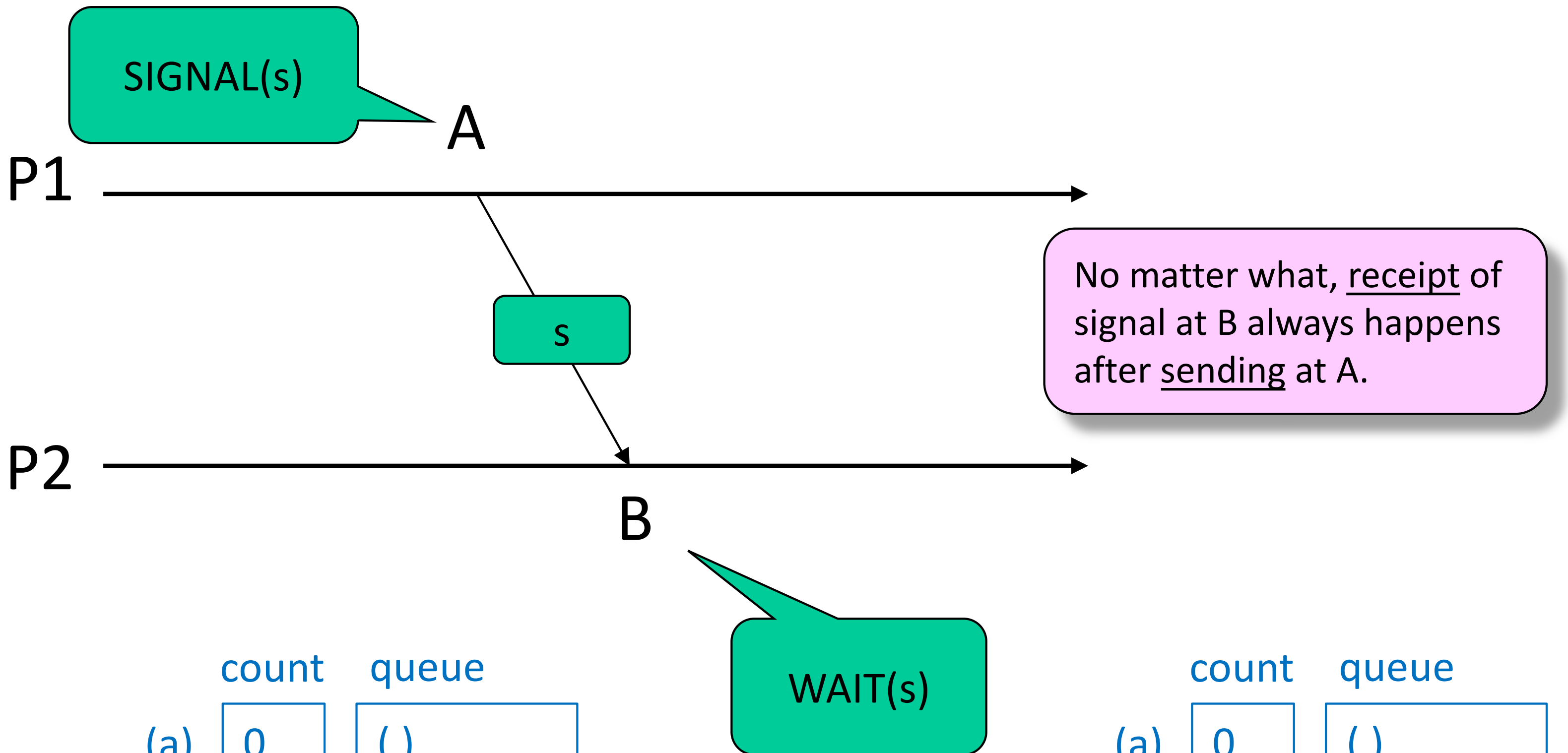
Semaphores can implement basic synchronization



		count	queue
	(a)	0	( )
P2 reaches B	(b)	-1	(P2)
P1 reaches A	(c)	0	( )

		count	queue
	(a)	0	( )
P1 reaches A	(b)	1	( )
P2 reaches B	(c)	0	( )

Semaphores can implement basic synchronization



		count	queue
	(a)	0	( )
P2 reaches B	(b)	-1	(P2)
P1 reaches A	(c)	0	( )

		count	queue
	(a)	0	( )
P1 reaches A	(b)	1	( )
P2 reaches B	(c)	0	( )

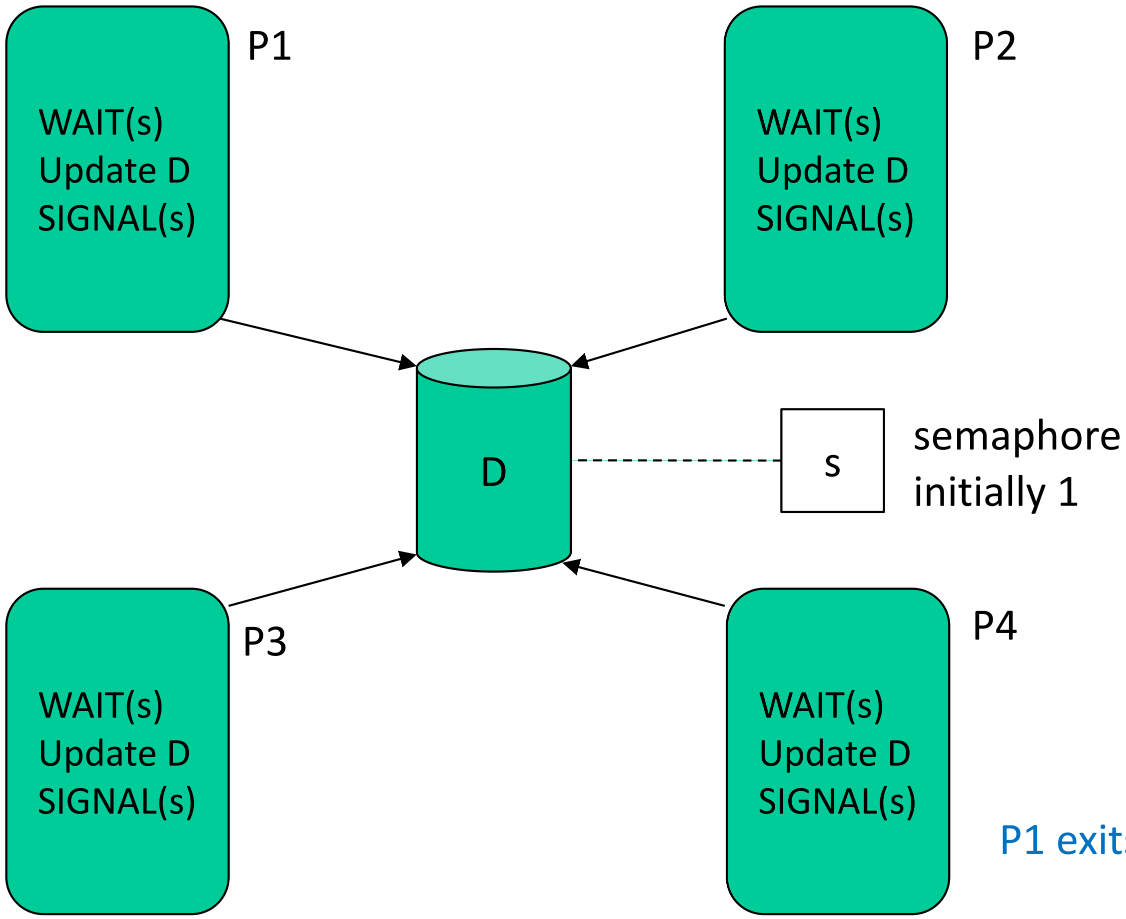
Mutual Exclusion

- Critical section of code: accesses shared data.
- Requirement that a critical section be executed by at most one process at a time: avoid race conditions on the shared data.
- Semaphore `s [1]` locks the critical section.

```
WAIT(s)
<Critical Section>
SIGNAL(s)
```

- s stands for condition "critical section free"

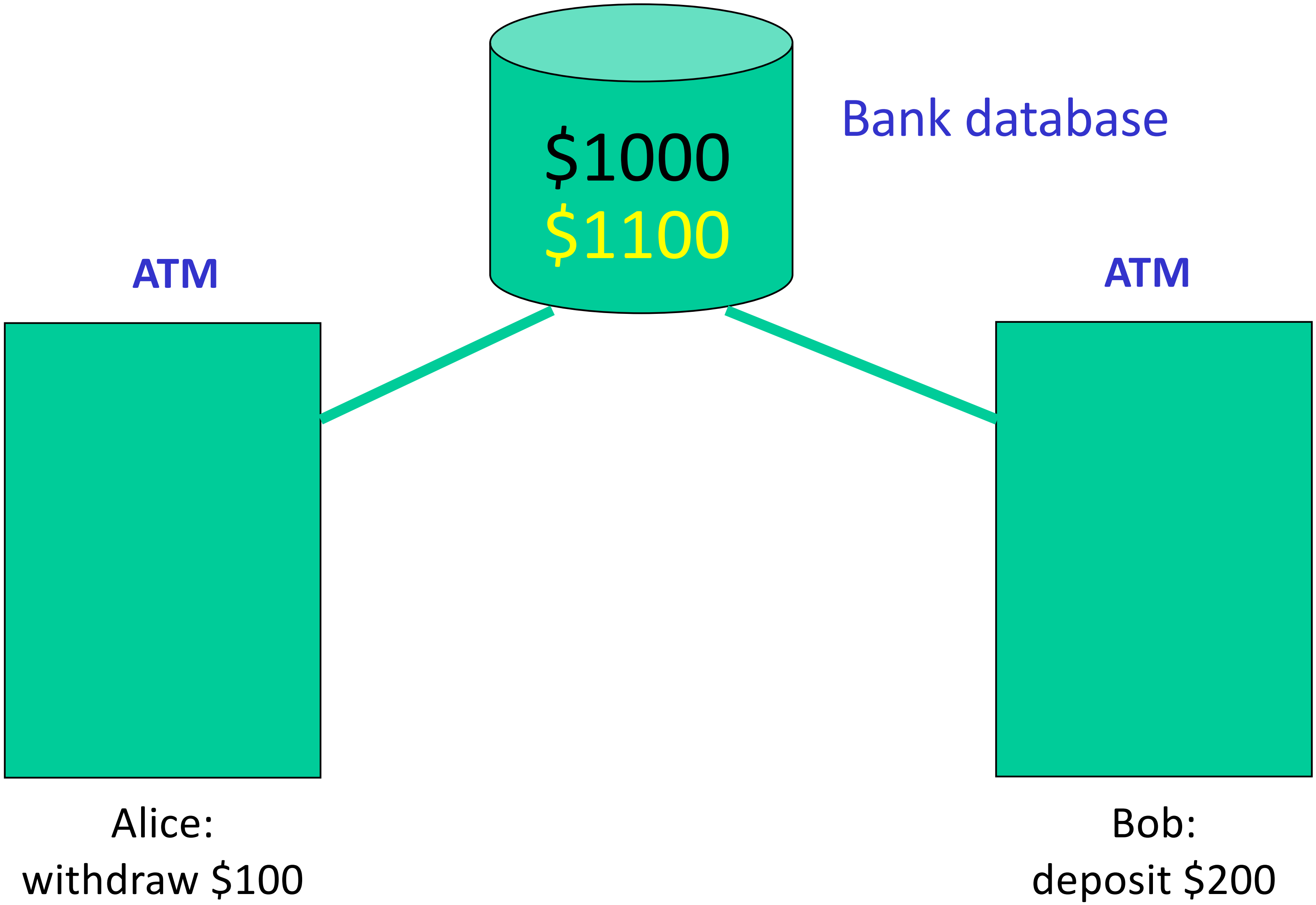
Mutual Exclusion

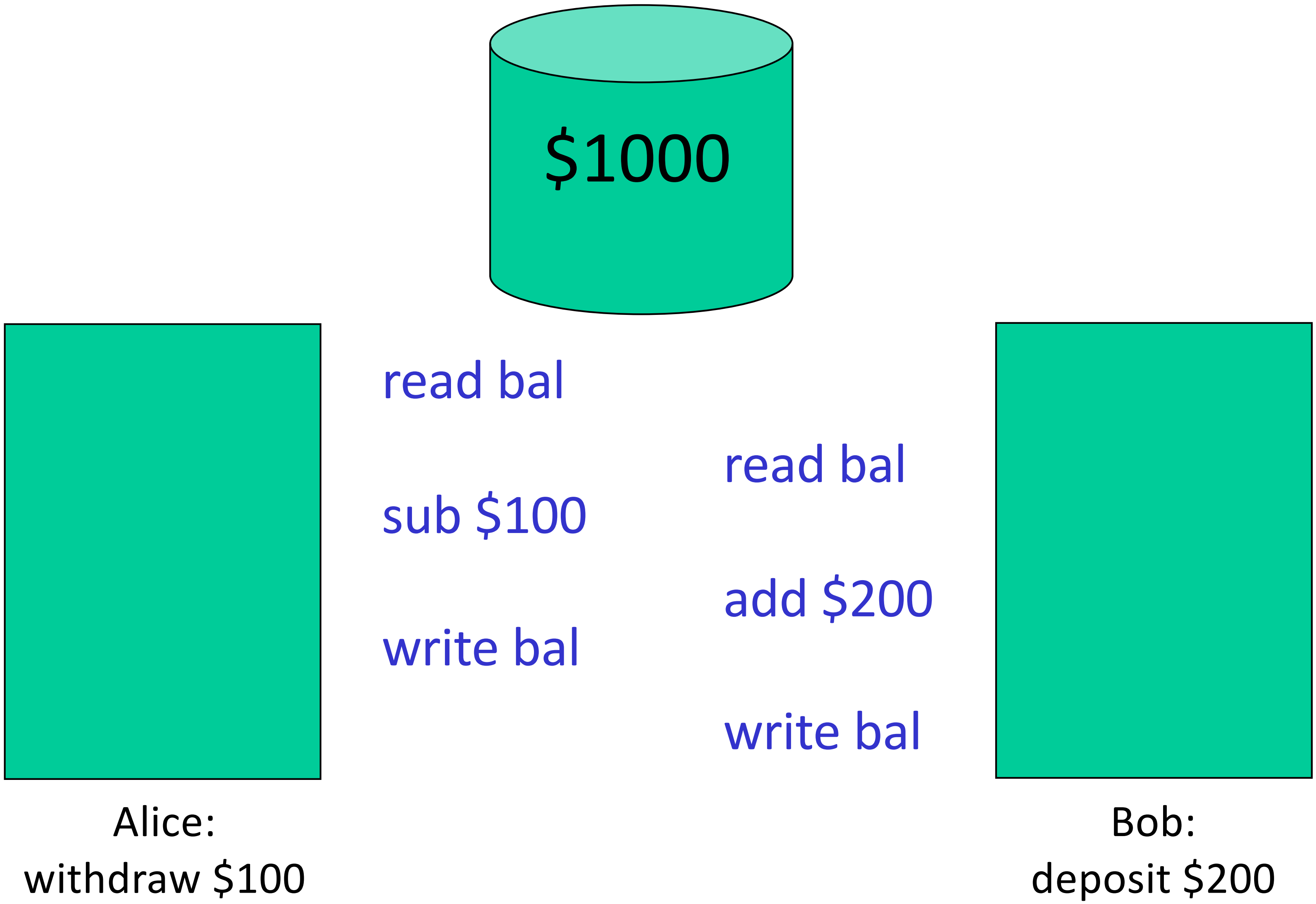


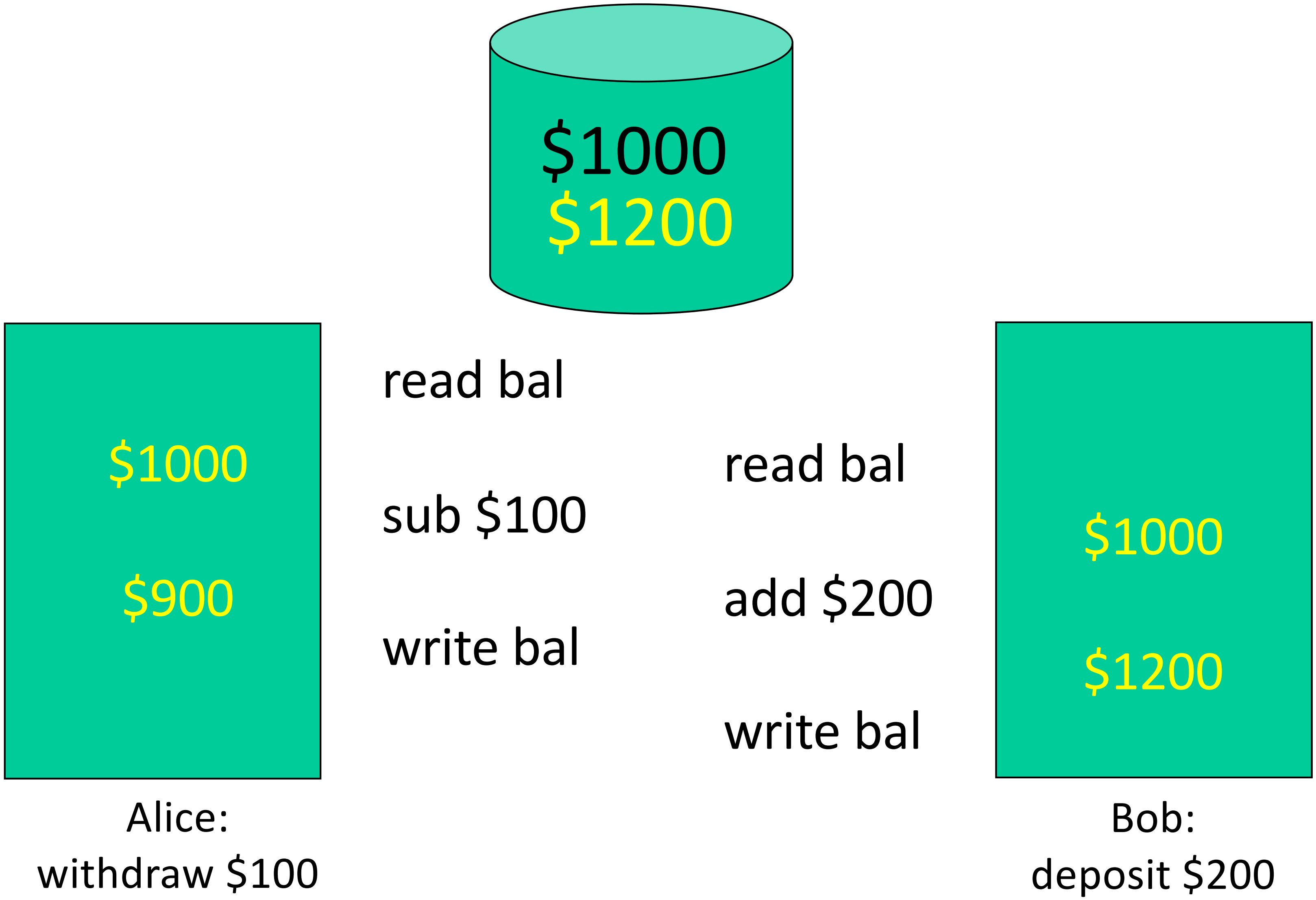
		count	queue
initial	(a)	1	( )
P1 in D	(b)	0	( )
P2 requests D	(c)	-1	(P2)
P3 requests D	(d)	-2	(P2,P3)
P1 exits; P2 enters D	(e)	-1	(P3)
P4 requests D	(e)	-2	(P3,P4)
P2 exits, P3 enters D	(e)	-1	(P4)
P3 exits, P4 enters D	(e)	0	( )
P4 exits D	(e)	1	( )

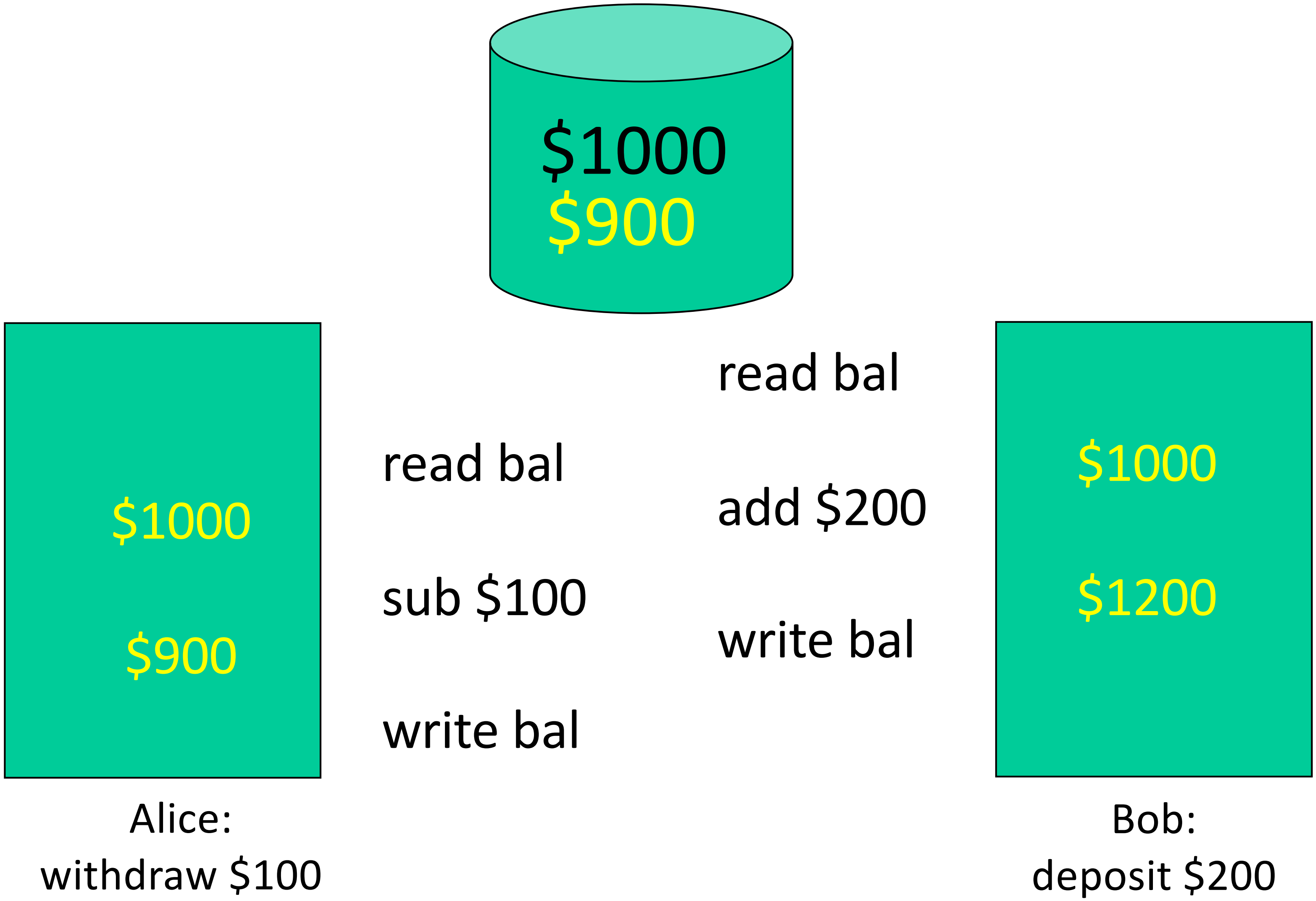
Race Conditions

- One of the most common coordination problems is preventing races among two or more processes.
- When two processes share data, the final result may depend on their speeds and order, giving random and unpredictable results.
- ATM example illustrates. Alice and Bob share an account at the bank and can perform transactions from ATMs at any time.


$1000
$1100
Bank database
ATM
ATM
Alice:
withdraw $100
Bob:
deposit $200


$1000
read bal
sub $100
write bal
read bal
add $200
write bal
Alice:
withdraw $100
Bob:
deposit $200


$1000
$1200
read bal
sub $100
write bal
read bal
add $200
write bal
$1000
$900
$1000
$1200
Alice:
withdraw $100
Bob:
deposit $200


$1000
$900
read bal
add $200
write bal
$1000
$1200
read bal
sub $100
write bal
$1000
$900
Alice:
withdraw $100
Bob:
deposit $200

Critical Section

Segment of program that must be executed as a unit in order to avoid race conditions.

Atomicity

Executions must be one after the other, not at the same time – order matters but no internal races.

Serialization

read bal
sub $100
write bal

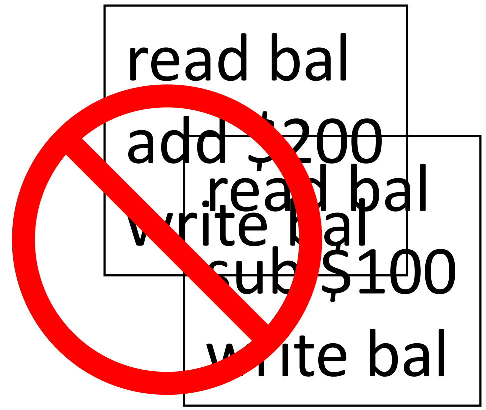
or

read bal
add $200
write bal

read bal
add $200
write bal

read bal
sub $100
write bal

but not



Spin Locks - 1

- It is not always feasible to implement “wait for a lock” by a queue.
- Sometimes cycling on testing a lock until someone unlocks it is the only way.

Spin Locks - 2

- Multiple independent CPUs accessing TCBs and RL is a common example.
  - Cannot allow race conditions updating TCBs-RL
  - Need to mutually exclude multiple CPUs accessing these shared structures.
  - Cannot implement a queue – need to lock the TCB-RL structures to do that (circularity).

Spin Locks – 3

```
L: if x=1 then goto L
   x=1
   <Critical Section>
   x=0


while(x=1) do { }
  x=1
  <Critical Section>
  X=0
```

These are two equivalent ways to try to program a lock to protect a critical section of code.

Do you see the bug?

Spin Locks - 4

- Test and Set lock instruction, TSL(x)
- x is a memory location used for the lock
  - 0 means unlocked
  - 1 means locked
- TSL(x) means: in one (uninterruptable) instruction cycle, read and return the contents of x and replace them with 1.

Spin Locks - 5

```
while(TSL(x)) do { }
<Critical Section>
X=0
```

Now the testing-setting is atomic.

Do you see how to prove that two CPUs cannot be in the CS at once?

Hint: assume they are, find a contraction to the definition of TSL.

```
with lock {
   <Critical Section>
   }
```

Simplified way to do the above spin lock in a programming language.

Compiler translates "with" statement to the above code.

PSEUDOCODE FOR WAIT AND SIGNAL KERNEL CALLS

```
sem s: structure with
       c: counter
       q: queue
    lock: lock
```

```
attach(i, queue): link i to tail of queue
i =  detach(queue): unlink and return head of queue
RL:  Ready List
PID: CPU register holding ID of running process
```

```
WAIT(s):
     with s.lock:
          s.c--
          if s.c<0 then
             SAVESW
             attach(PID, s.q)
             set PID = detach(RL)
             LOADSW
     return
```

```
SIGNAL(s):
     with s.lock:
          s.c++
          if s.c≤0 then
             P = detach(s.q)
             attach(P, RL)
     return
```

PSEUDO CODE FOR SEMAPHORE CREATE AND DELETE

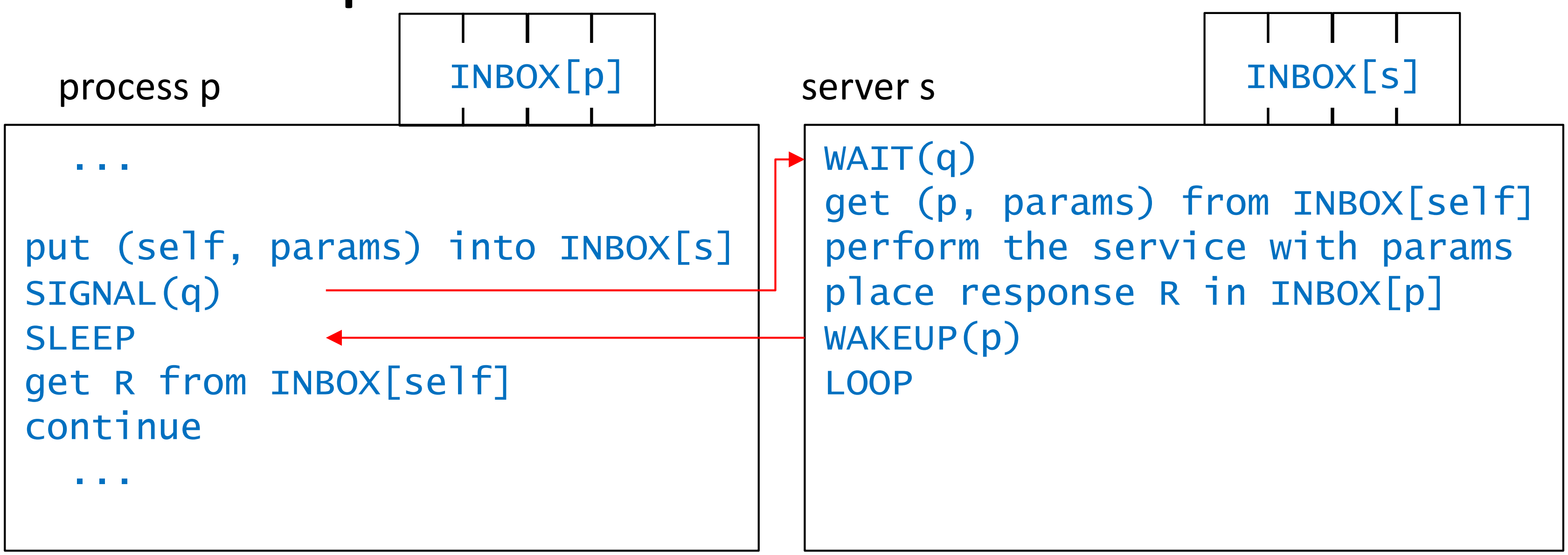
```
SCB: array of M control blocks (M>N, number of processes) each with
       lock          :lock with TSL during "with" statements
       counter       :counter
       queue         :(head, tail) descriptor of queue
       link          :next SCB in queue

Kernel provides two more operations:

     s = CREATE_SEM(I≥0), return index s of a new SCB with initial count I
     DELETE_SEM(s), DELETE_SCB[s]

There are many ways to implement CREATE and DELETE, but those details
are not important here.
```

Suspend and Resume

- Suspended is a process state in which the process may not use the CPU or memory
- A process can be suspended or resumed only by one of its parents
  - SUSPEND(p) is kernel call to suspend process p.
  - RESUME(p) is kernel call to restore process p to normal operation with CPU and memory.
- Not a semaphore operation

Sleep and Wakeup

- A common synchronization pattern occurs with requests to service processes and their responses.



- SLEEP means: put the caller from running into the “sleeping” state
- WAKEUP(p) means: move p from “sleeping” to ready state
- Can implement with private semaphore: psem[p] can be waited on only by p. Initial value of counter is 0.
  - SLEEP = WAIT(psem[self])
  - WAKEUP(p) = SIGNAL(psem[p])

- Much more efficient implementation.
  - If p is not waiting, count=0 and queue=( ).
  - If p is waiting, count=-1 and queue=(p)
- Represent these two states with “waiting” bit.
- However, in the request-response protocol, the responder may execute WAKEUP before the requester SLEEPs. Corresponds semaphore count =1.
- Represent this with “wakeup waiting” bit.

Pseudocodes

SLEEP

```
with TCB[self]
  if wakeupwaiting=0
     wait=1
     SAVESW
     LOADSW
  else
     wakeupwaiting=0
  return
```

WAKEUP(p)

```
with TCB[p]
  if waiting=0
     wakeupwaiting=1
  else
     attach(p,RL)
     waiting=0
  return
```