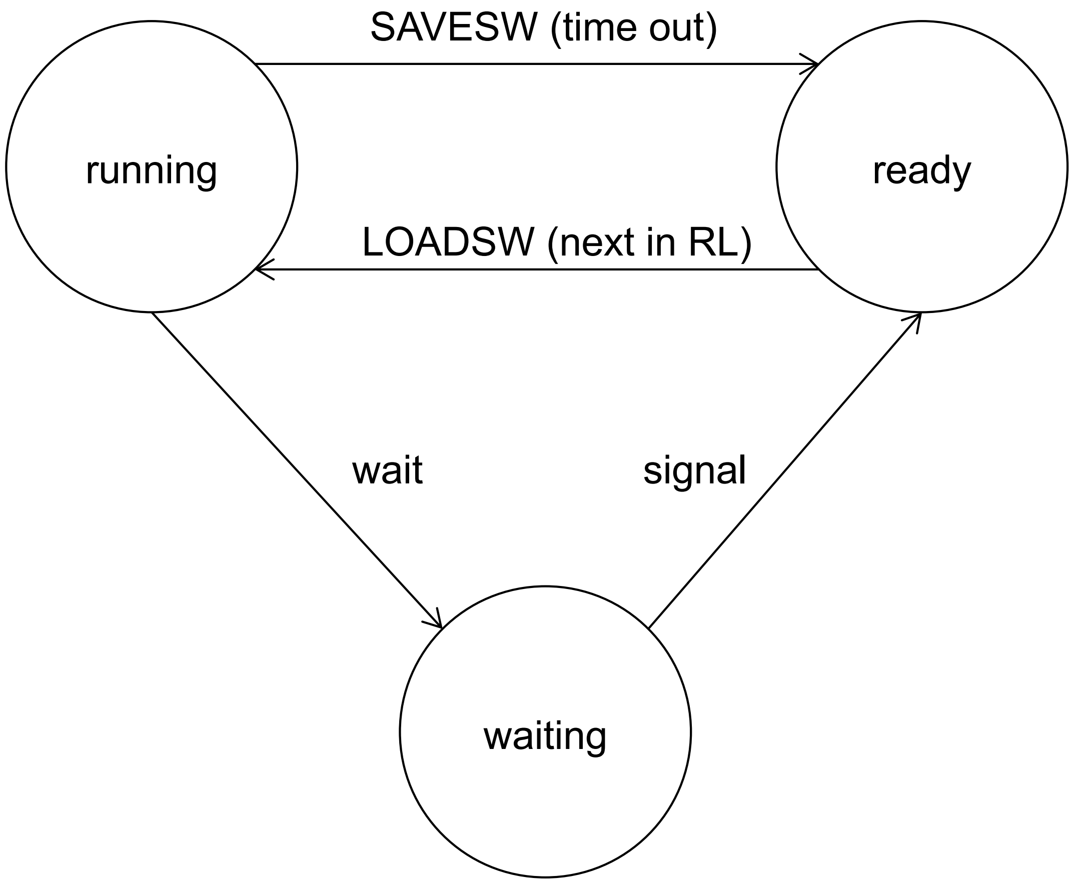
Wait-Signal Pseudocode

Peter J. Denning

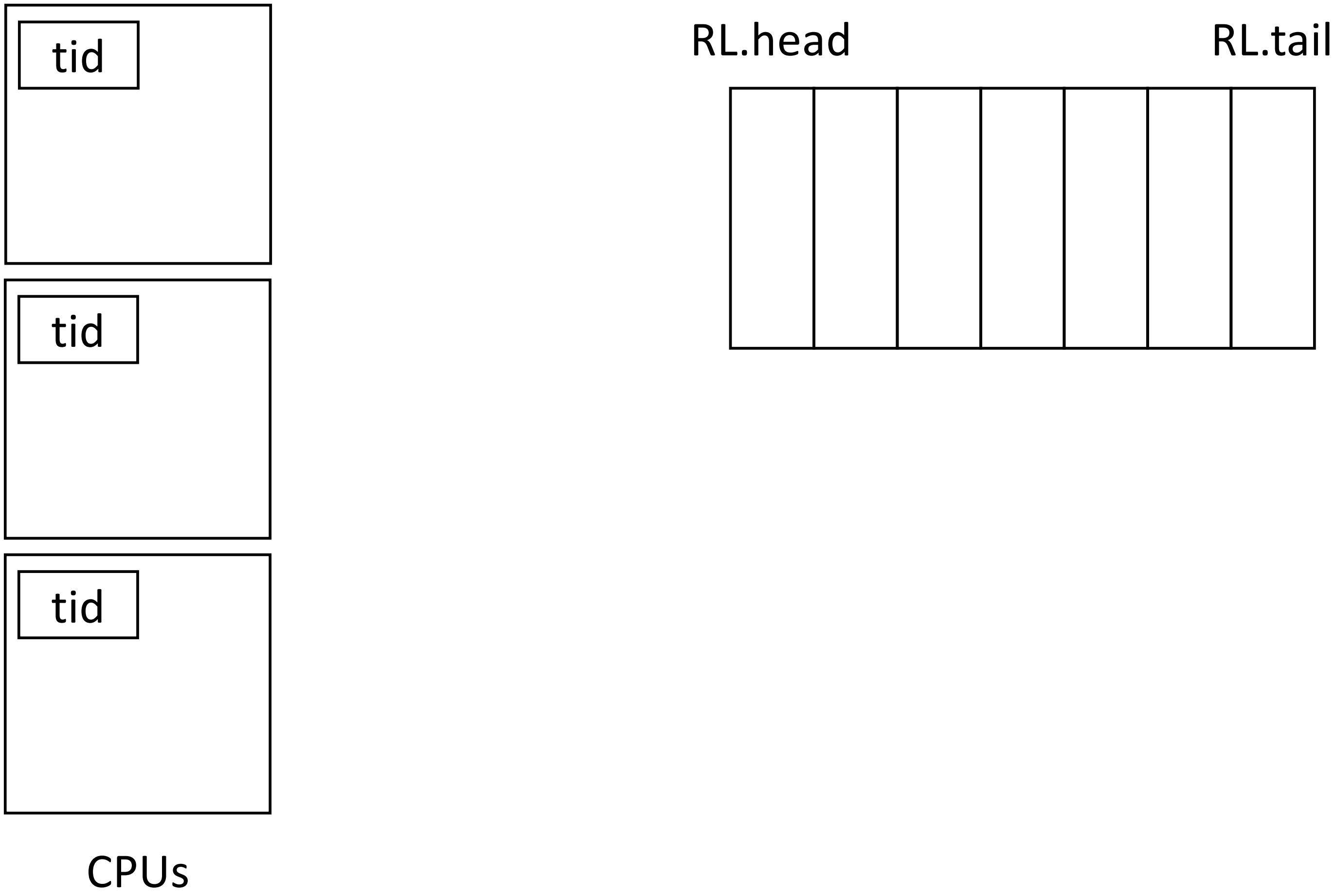
Implementing semaphores

- Objective is to implement the wait state in the thread-state diagram.
- Each state is presented by a FIFO queue listing all the threads in that state.
- Every thread is always in exactly one queue.
- State changes only when threads change queues, are created, or are terminated.
- The wait state is actually a set of queues, one for each semaphore; each semaphore represents a particular reason for waiting.

Process States



Queues for Running and Ready States



tid

tid

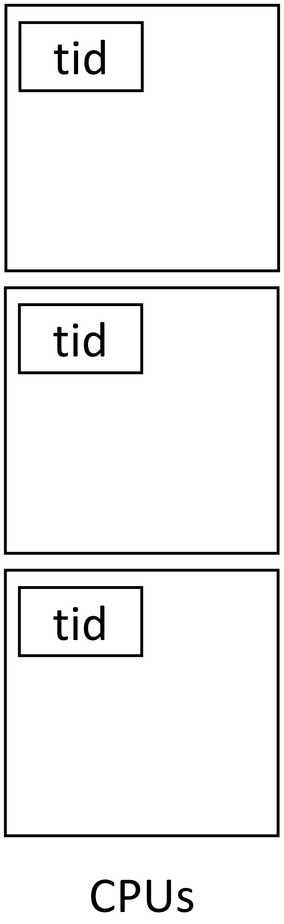
tid

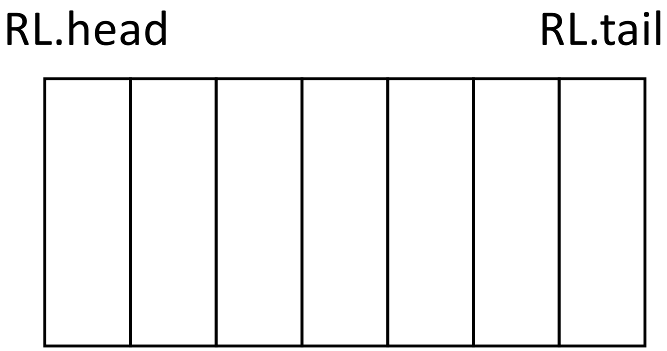
CPUs

RL.head

RL.tail

Queues for Running and Ready States

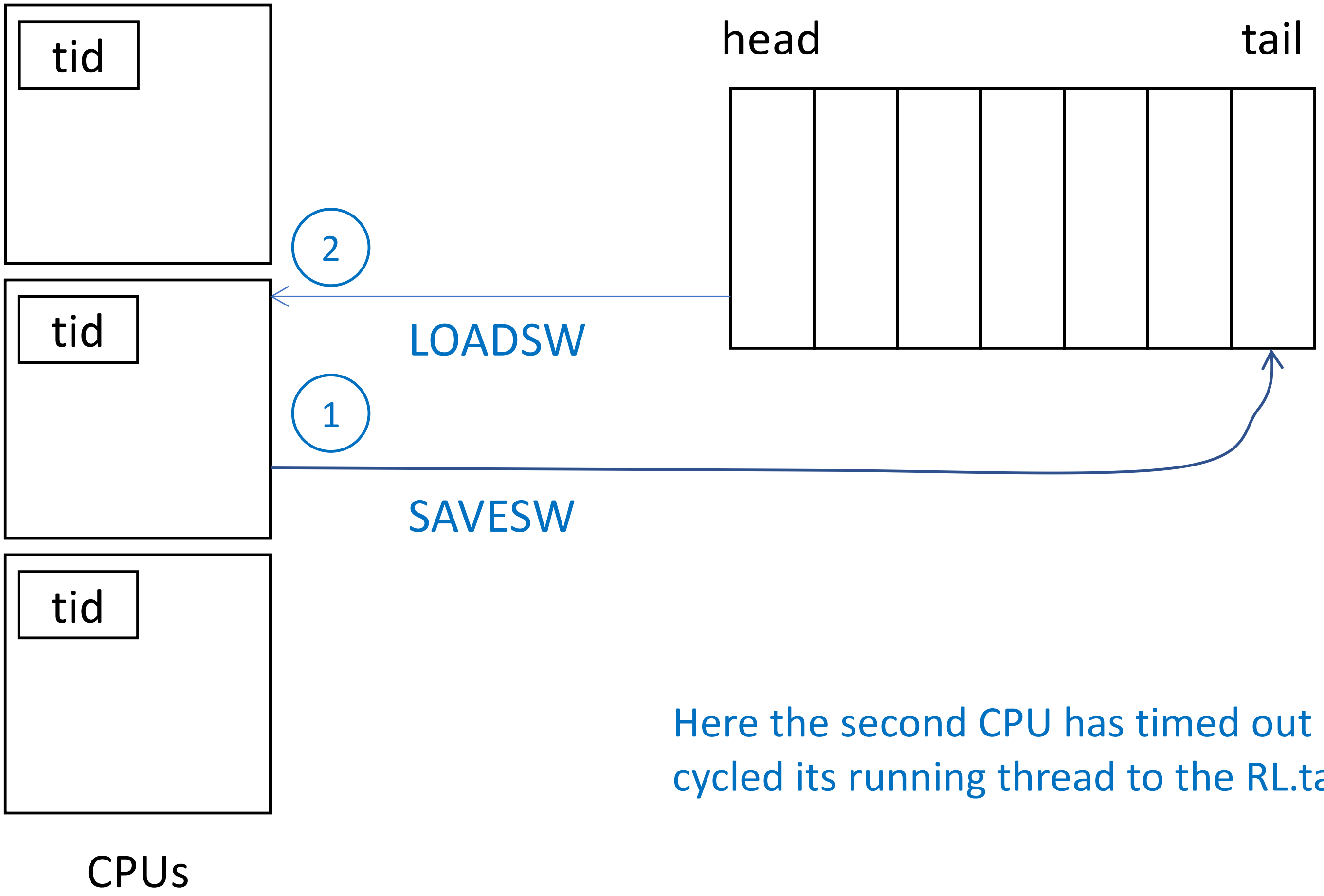




Here 3 CPUs are available to run separate threads. Each CPU is a “running queue” of length 1.

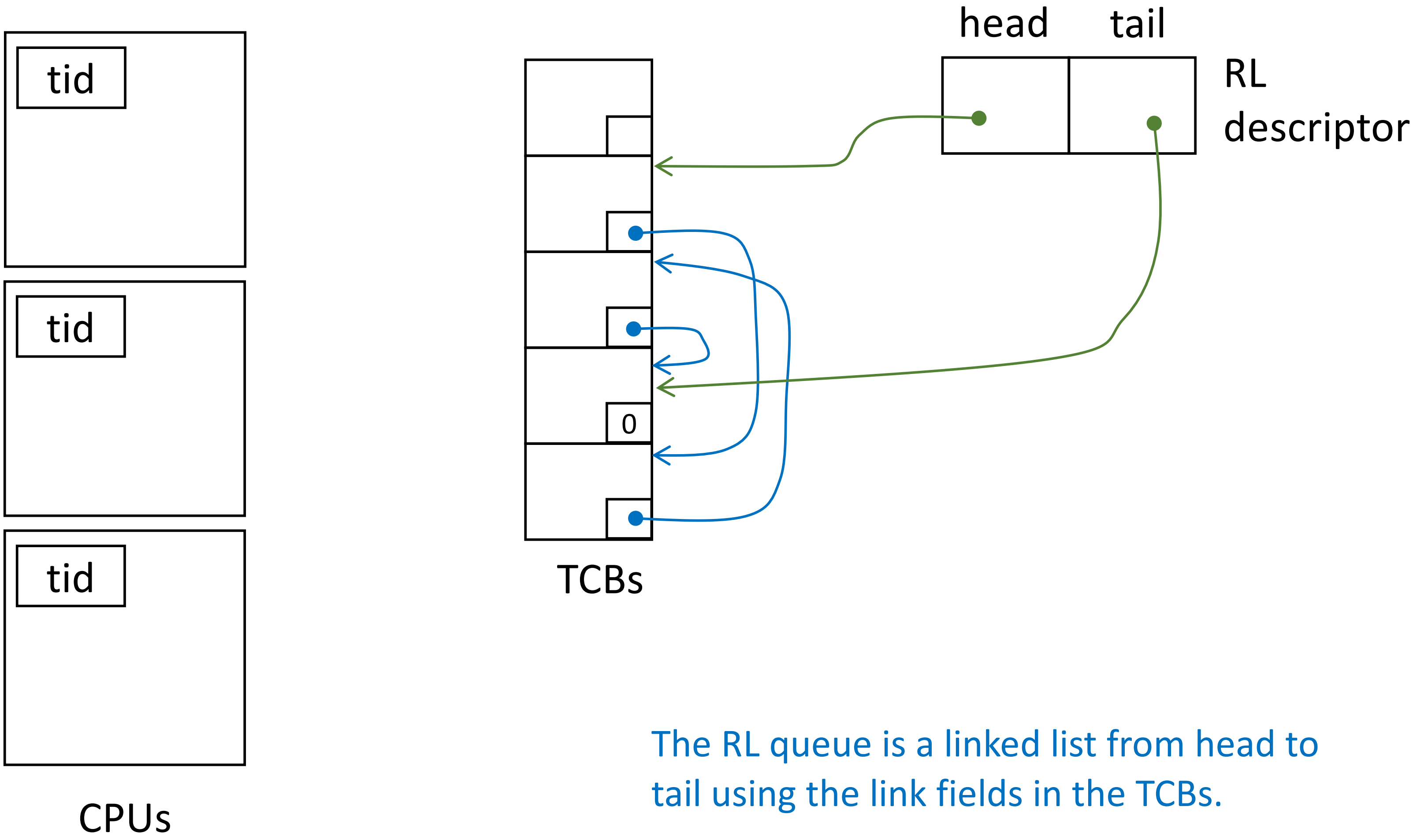
When a CPU times out, SAVESW saves its stateword and moves its tid to tail of the RL queue. Then it moves the RL.head tid into the CPU and LOADSW retrieves its stateword from that thread’s TCB.

Queues for Running and Ready States



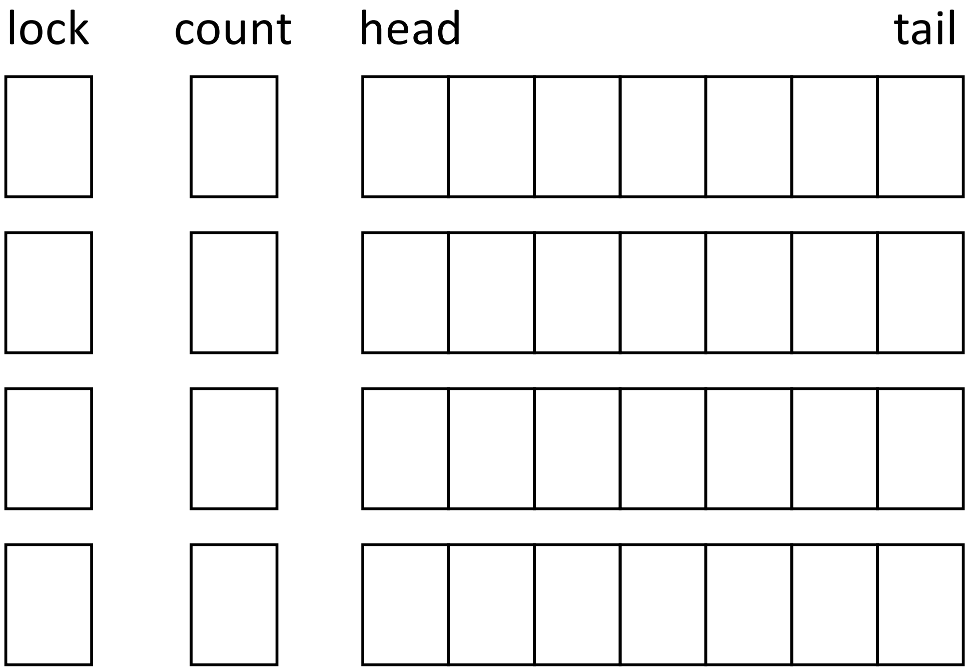
Here the second CPU has timed out and cycled its running thread to the RL.tail.

Queues for Running and Ready States



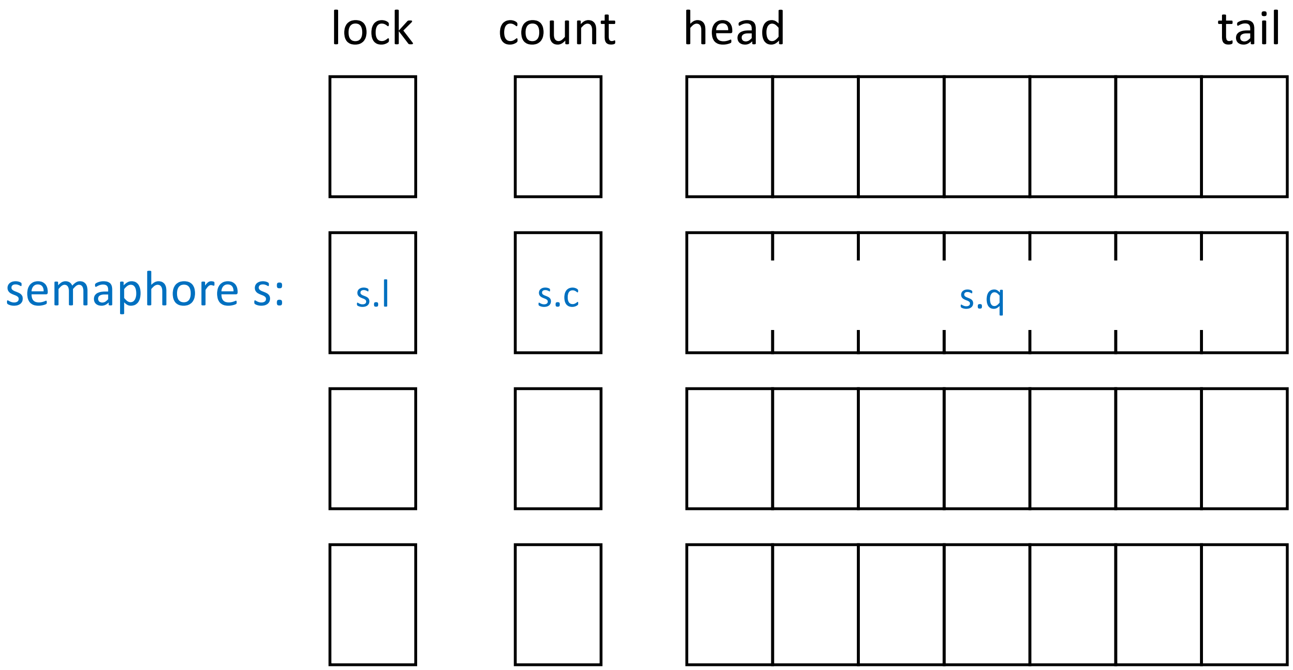
The RL queue is a linked list from head to tail using the link fields in the TCBs.

Queues for Wait States



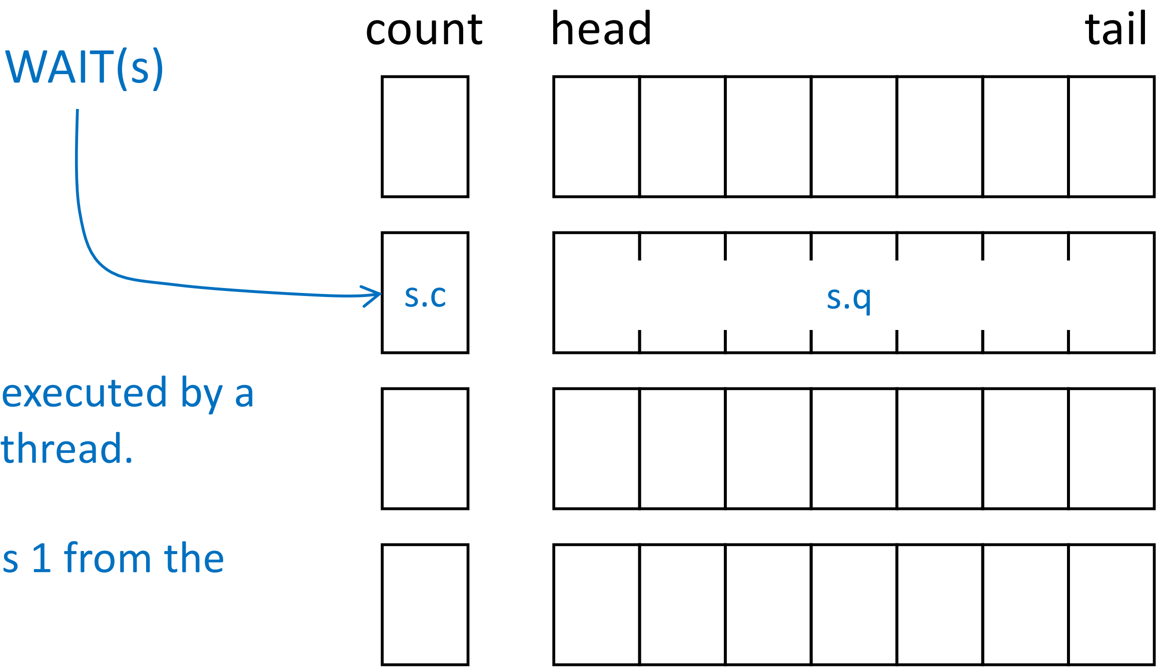
Each semaphore is a condition that can be waited for.   Has its own lock, count, and queue – all stored in a semaphore control block (SCB). Each line above is the content of a SCB.

Queues for Wait States



Lock of semaphore s is s.l
Count of semaphore s is s.c
Queue of semaphore s is s.q

Queues for Wait States



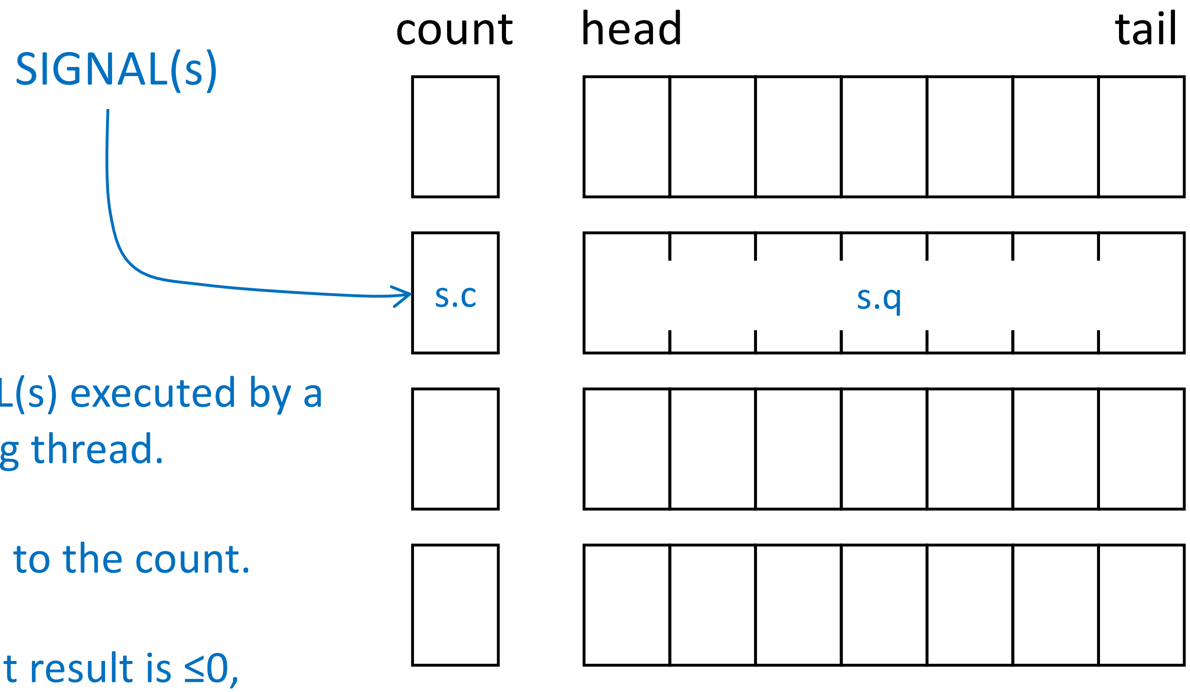
WAIT(s) executed by a running thread.

Subtracts 1 from the count.

If count result is <0, puts tid at tail of queue and with LOADSW starts next ready thread (RL.head).

If count result is ≥ 0, WAIT returns without waiting.

Queues for Wait States

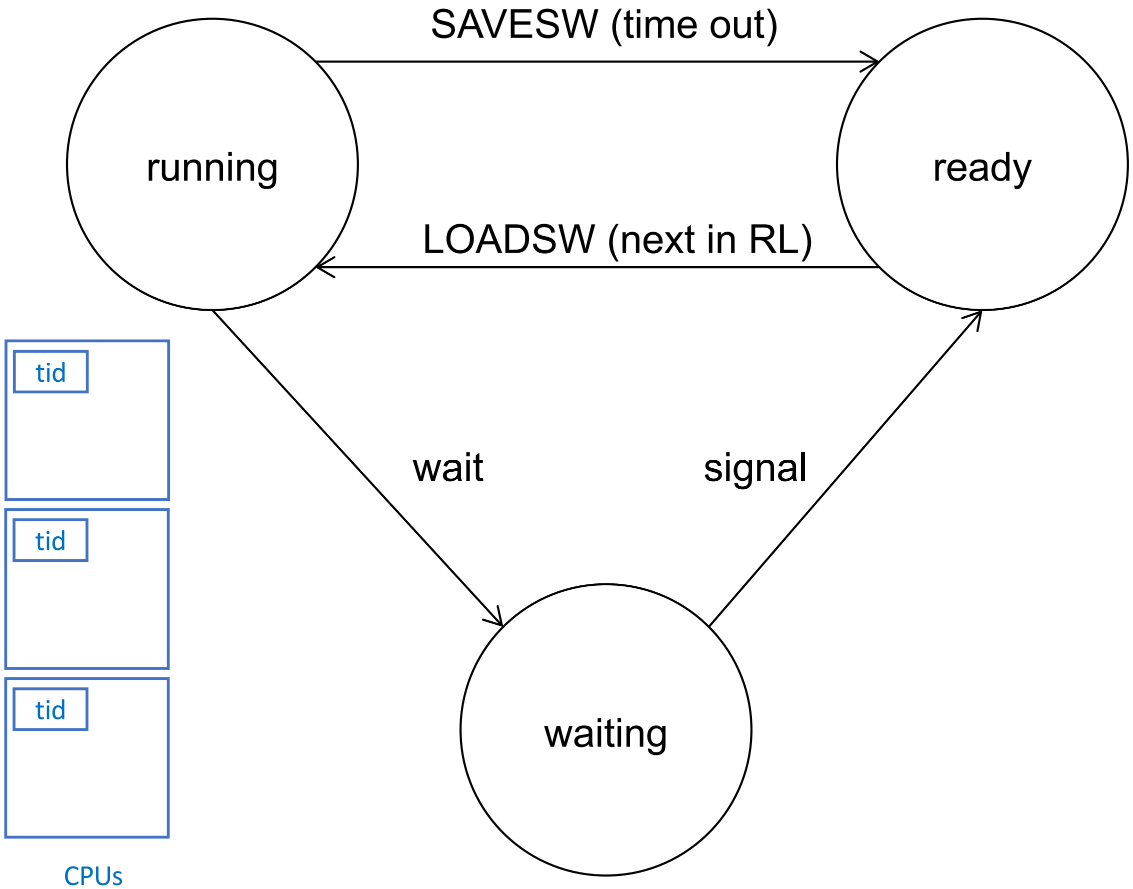


SIGNAL(s) executed by a running thread.

Adds 1 to the count.

If count result is ≤0, move tid from s.q.head to RL.tail.

Returns without waiting



SAVESW (time out)

running

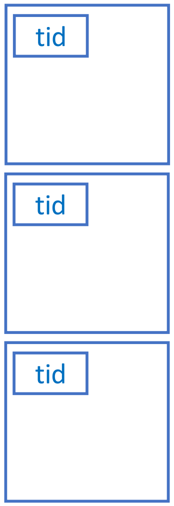
ready

LOADSW (next in RL)

wait

signal

waiting

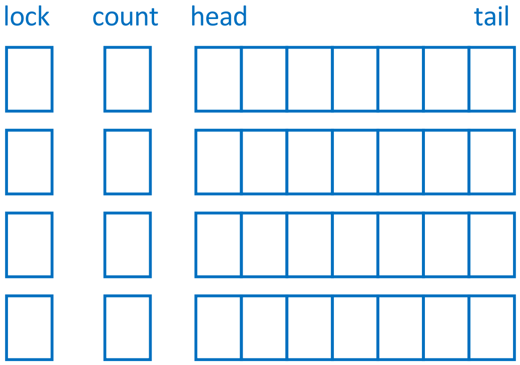


tid

tid

tid

CPUs



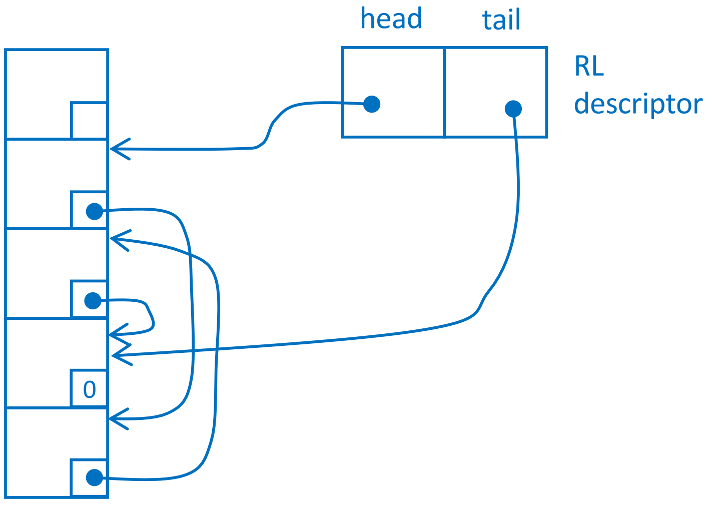
lock

count

head

tail

SCBs



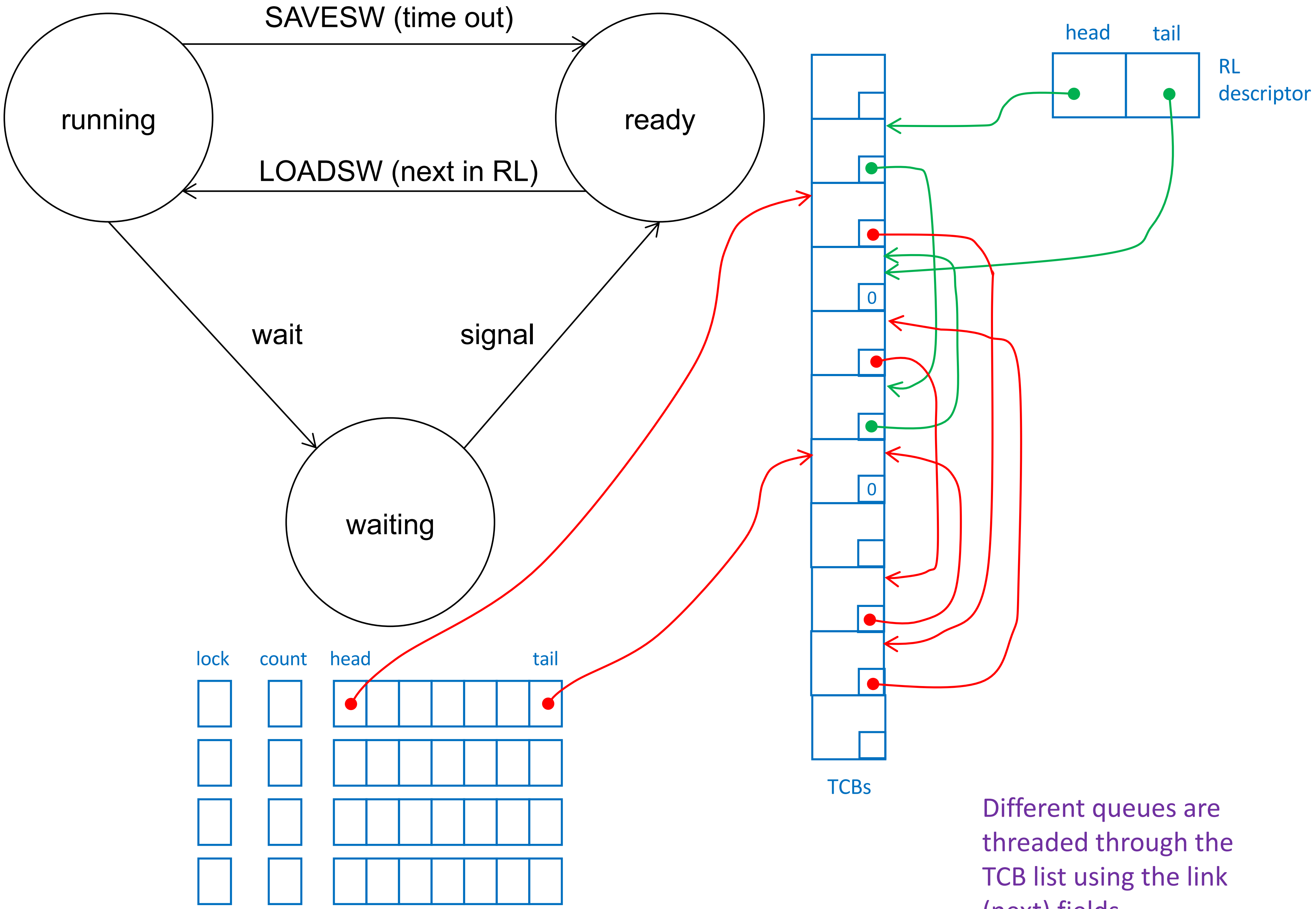
head

tail

RL descriptor

0

TCBs



Different queues are threaded through the TCB list using the link (next) fields.

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
NOTE: attach and detach lock RL during access
RL:  Ready List
tid: CPU register holding ID of running thread
```

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

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

PSEUDOCODE 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 system free list
```

Initially (boot time) all SCBs linked on a system free list

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 the details are not important here.