2 problems. 40 points. 30 minutes *Closed book. Closed notes. No electronic device.* *Write your name above.*

Program BB models a "bounded-buffer" of size N.

Awaits are weak (i.e., a thread passes await(B) S if B holds continuously).

Parameter j is an integer in 1..N.

```
program BB():
  N:  positive integer
  num ← 0

  function cAdd(j):
     await (num ≤ N – j)
        num ← num + j

  function cRmv(j):
     await (num ≥ j)
        num ← num – j
```

1. [25 points] Implement program BB (including its progress) using locks and condition variables as the *only* synchronization constructs. Your answer will consist of

- Definitions of additional variables (e.g., locks, condition variables).
- Pseudocode bodies of functions cAdd(j) and cRmv(j). *Each function must be less than 12 lines.*

Solution

Shared variables:

```
   Lock lck                         [1 pt]
   Condition(lck) cvAdd, cvRmv      [4 pt]

cAdd(j):
   lck.acq()                        [1 pt]
   while (num > N - j)              [4 pt]
      cvAdd.wait()
   num ← num + j
   cvRmv.signal()                   [2 pt]
   if (num < N)                  ** [1 pt]
      cvAdd.signal()             ** [1 pt]
   lck.rel()                        [1 pt]

cRmv(j):
   lck.acq()                        [1 pt]
   while (num < j)                  [4 pt]
      cvRmv.wait()
   num ← num - j
   cvAdd.signal()                   [2 pt]
   if (num > 0)                  ** [1 pt]
      cvRmv.signal()             ** [1 pt]
   lck.rel()                        [1 pt]
```

Note: The ** lines are needed. Otherwise the following can happen, which voilates BB's progress:

initially	num is 0
thread u calls cRmv(1)	num is 0; u stuck at cvRmv
thread v calls cRmv(1)	num is 0; u, v stuck at cvRmv
thread w calls cAdd(2), returns	num is 2; u unstuck, v stuck at cvRmv
thread u returns	num is 1; v stuck at cvRmv

End of solution

2. [15 points] Implement program `BB` using semaphores as the *only* synchronization constructs. *Your solution must ensure priority for awakened threads*, i.e., if a thread is awakened at a gate, it must not get blocked again.

Your answer will consist of

- Definitions of additional variables (e.g., semaphores).
- Brief description of function bodies. No need for pseudocode.

Solution

Suppose thread u is blocked in `cRmv(j)`. It should be awakened, say by thread v, only if `num` $\geq$ `j`; otherwise, u would get blocked again. So v has to know the value of u's parameter `j`. Here are two ways:

- v reads u's `j` (requires new functions)
- u waits on a gate specific to `j` (requires new variables)

Let's do the second option here.

Shared variables:

```
Semaphore(1) mutex                                                          [1 pt]
Semaphore(0) gateAdd[1..N]       // thread stuck in cvAdd(j) waits on gateAdd[j]   [2 pt]
int nwAdd[1..N]          // nwAdd[j] is # threads waiting on gateAdd[j]; initially 0   [2 pt]
Semaphore(0) gateRmv[1..N]       // thread stuck in cvRmv(j) waits on gateRmv[j]   [2 pt]
int nwRmv[1..N]          // nwRmv[j] is # threads waiting on gateRmv[j]; initially 0   [2 pt]
```

Function `cAdd(j)`:

1. do `mutex.P()`
   if guard does not hold, do `nwAdd[j]++`, `mutex.V()`, `gateAdd[j].P()`, `nwAdd[j]--` [3 pt]
2. do action
   if there is a `k` such that `nwAdd[k] > 0` and `num` $\leq$ `N - k`, do `gateAdd[k].V()` and return
   or if there is a `k` such that `nwRmv[k] > 0` and `num` $\leq$ `N - k`, do `gateRmv[k].V()` and return
   or if there is no such `k` do `mutex.V()` and return [3 pt]

Function `cRmv(j)` is symmetric (step 3 is exactly the same).

[6 pt] max for solution that uses one gate (instead of `N` gates) and works for the case `N = 1`

[7 pt] max for solution that uses memory proportional to the max # of threads (= max # of ongoing calls).

[−1 pt] for not using gate counters

[−1 pt] for doing `mutex.P()` after `gate.P()` in step 1.

[−2 pt] for not doing the selection in step 2, e.g., waking up more than one thread and/or releasing `mutex`.

End of solution