Semaphores

Mythili Vutukuru

CSE, IIT Bombay

Semaphore

- Another synchronization primitive like condition variables
  - Can be used to achieve similar synchronization between threads
- Semaphore is a variable with an underlying counter
  - Semaphore counter can be initialized to a suitable value
- Two functions on a semaphore variable
  - Down/wait decrements the counter by one, blocks the calling thread if the resulting value is negative
  - Up/post increments the counter by one, wakes up any one thread that is blocked on the semaphore
- Not possible to access counter value in any other way
  - For example, cannot check if counter is positive and only then call down

Example: use semaphore as a lock

- Consider semaphore variable “sem” initialized to value 1
- Multiple threads in a program can use semaphore for mutual exclusion
  - down(sem) //counter is 0, any further downs will wait here
  - critical section //accessed with mutual exclusion
  - up(sem) //counter incremented to 1, waiting thread woken up
- Such semaphore used as locks are called binary semaphores

T1

down(sem) //counter=0
critical section
….
…
up(sem)

T2

down(sem) //counter -1
(BLOCKED)

(woken by T1’s call to up)
critical section
up(sem)

T1

down(sem) //counter 0
critical section
up(sem) //counter 1

T2

down(sem) //counter 0
critical section
up(sem) //counter 1

The semaphore value tracks the available resources or slots, and up() restores one unit of availability.

Example: T1→T2

- Suppose we want two threads to synchronize as follows: T1→T2
  - T1 does some work and only then T2 runs
  - If T2 starts before T1, it must wait until T1 finishes its task
- We can achieve this using a semaphore: “sem” initialized to 0
  - T1 does its work, calls up(sem)
  - T2 calls down(sem), then does its work

T2

down(sem) //counter -1
(BLOCKED)

(woken by T1’s up)
T2 does its task

T1

T1 does its task
up(sem) //counter is 0

T1

T1 does its task
up(sem) //counter 1

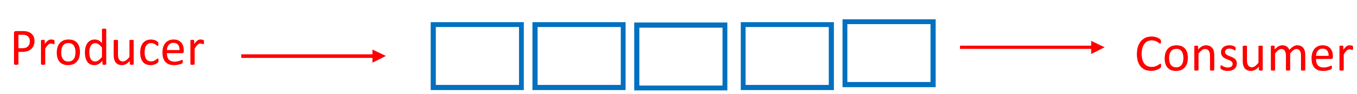
down(sem) //counter 0
//no need to block
T2 does its task

Semaphore implementation

- You may assume that up and down operations are implemented atomically (using locking internally as needed)
  - Counter accessed and updated with mutual exclusion
  - Need not worry about race conditions between up and down
  - No need to use extra locks to protect atomicity of wait/down
- But need to use separate locks or binary semaphores to access shared data in a program
  - Semaphore used for signaling doesn’t provide mutual exclusion

Recap: Producer-consumer problem

- Producer and consumer threads, sharing data via a buffer of size N
  - Producers produce items, add into a shared buffer
  - Consumers consume item from shared buffer
- What kind of coordination is needed between threads?
  - Producer thread produces and places items into buffer, waits if the buffer is full → Consumer signals after making space in the buffer
  - Consumer thread consumes items from buffer, waits if the buffer is empty → Producer signals after producing items
- We have studied implementation with condition variables, similar implementation possible with semaphores



Producer-consumer using semaphores (1)

- One semaphore "sem_empty" is initialized to N, indicates number of empty slots in buffer available for producers to use
  - Producer does down every time it produces an item
  - Once all slots are filled, down operation blocks
  - Sleeping producer woken up by consumer that calls "up" after consuming
- Another semaphore "sem_filled" is initialized to 0, indicates number of filled slots in buffer that are ready to be consumed
- Easier solution than CV, no need to keep separate counter

```
//Producer
down(sem_empty) //blocks if buffer full
produce item
up(sem_filled)//wakeup consumer
```

```
//Consumer
down(sem_filled)//blocks if buffer empty
consume item
up(sem_empty)//wake up producer
```

Producer-consumer using semaphores (2)

- Note: semaphore solution does not use any locks by default
  - Condition variables had associated locks
- If buffer needs to be accessed correctly, needs extra locks via binary semaphores for mutual exclusion
- Use another semaphore mutex (initialized to 1) for locking

```
//Producer
down(sem_empty) //blocks if buffer full
down(mutex)
produce item and add to buffer
up(mutex)
up(sem_filled)//wakeup consumer
```

```
//Consumer
down(sem_filled)//blocks if buffer empty
down(mutex)
consume item and remove from buffer
up(mutex)
up(sem_empty)//wake up producer
```

Producer-consumer using semaphores: deadlock

- With condition variables, lock given to sleep/wait is released after the thread is safely put to sleep
  - No such concept of releasing any locks with semaphores
- With semaphore, if you do down with another binary semaphore/lock held, the lock will not be released on its own
- The solution shown below leads to deadlock: why?

If the consumer runs first then it get's blocked in the first down(sem_filled) because there's no filled slot but it isn't able to release the mutex, hence producer never get's to run it'll block in down(mutex).

Make sure to signal before blocking.

```
//Producer
down(mutex)
down(sem_empty) //blocks if buffer full
produce item
up(sem_filled)//wakeup consumer
up(mutex)
```

```
//Consumer
down(mutex)
down(sem_filled)//blocks if buffer empty
consume item
up(sem_empty)//wake up producer
up(mutex)
```

Guidelines for using semaphores

- Semaphores can be used to do similar thread synchronization as CV
  - Waiting on CV ~ down operation on semaphore
  - Signaling on CV ~ up operation on semaphore
  - No equivalent of signal broadcast with semaphores
- Separate semaphores needed for signaling and mutual exclusion

Signaling should be out of mutual exclusion place to avoid deadlock, Principle: signal shouldn't be blocked.

- Semaphore counter can replace some integer/bool variables
  - But cannot access or change semaphore counter separately
- Careful with deadlocks
  - Ensure that "up" can run every time a thread blocks due to "down"
  - Note: no locks released when a thread blocks due to down operation
- Pay attention to initial value of semaphore

  What value the semaphore is starting with.

Example: Batched processing (1)

- Two kinds of threads in an application
  - Request threads, each containing an application request
  - Batch processor thread processes N requests at a time in a batch
- What kind of synchronization do we need?
  - Batch processing thread must wait until N requests arrive, then start batch
  - Request thread must wait until batch starts, then get processed and finish
- Example: suppose Covid-19 vaccination vial has 10 doses. Nurse waits for 10 patients to arrive, then opens the vial and vaccinates all 10

Example: Batched processing (2)

- Solution using two CVs: one for requests to wait, one for batch processor to wait
  - Other integer and boolean variables, mutex/lock for atomicity

```
//Request thread
lock(mutex)
count++
if(count == N)
   signal(cv_batch_processor)
while(not batch_started)
   wait(cv_request, mutex)
unlock(mutex)
```

```
//Batch processor thread
lock(mutex)
while(count < N)
   wait(cv_batch_processor, mutex)
batch_started = true
signal_broadcast(cv_request)
unlock(mutex)
```

Example: Batched processing (3)

- Semaphore mutex initialized to 1, acts as lock to update count
- Semaphore sem_batch_processor, initialized to 0
  - Batch processor waits, until Nth request unblocks it
- Semaphore sem_request, initialized to 0
  - All N request threads wait on it (until batch starts)
  - When batch starts, batch processor thread does up N times to unblock all

```
//Request thread
down(mutex)
count++
if(count == N)
   up(sem_batch_processor)
up(mutex)
down(sem_request)
```

```
//Batch processor thread
down(sem_batch_processor)
//ready to start batch
do N times: up(sem_request)
```

wake up all the 'n' request threads.

Example: Batched processing (4)

- Alternate pattern of solution
- Semaphore sem_request, initialized to 0
  - All N request threads wait on it (until batch starts)
  - When batch starts, batch processor does up once, unblocks only one thread
  - Each woken up request thread wakes up one other thread

```
//Request thread
down(mutex)
count++
if(count == N)
   up(sem_batch_processor)
up(mutex)
down(sem_request)
up(sem_request)
```

```
//Batch processor thread
down(sem_batch_processor)

up(sem_request)
```