CSC 369

Operating Systems

Lecture 4:

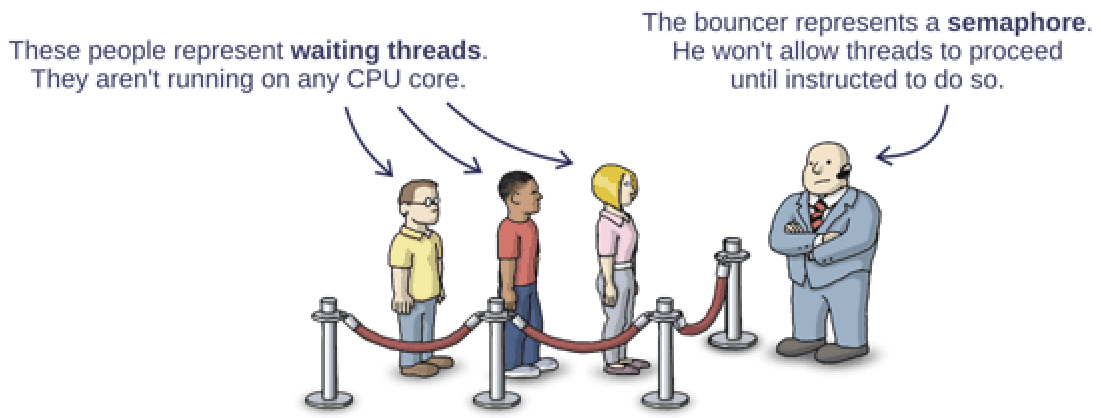
Synchronization problems

Condition variables, Monitors

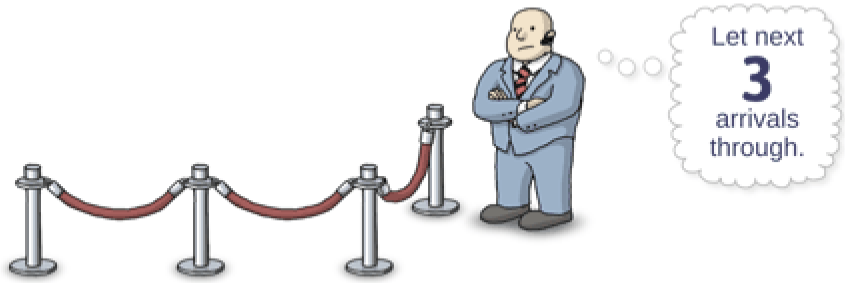
University of Toronto, Department of Computer Science

Semaphores reminder

- How does it work?


These people represent **waiting threads**. They aren't running on any CPU core.
The bouncer represents a **semaphore**. He won't allow threads to proceed until instructed to do so.

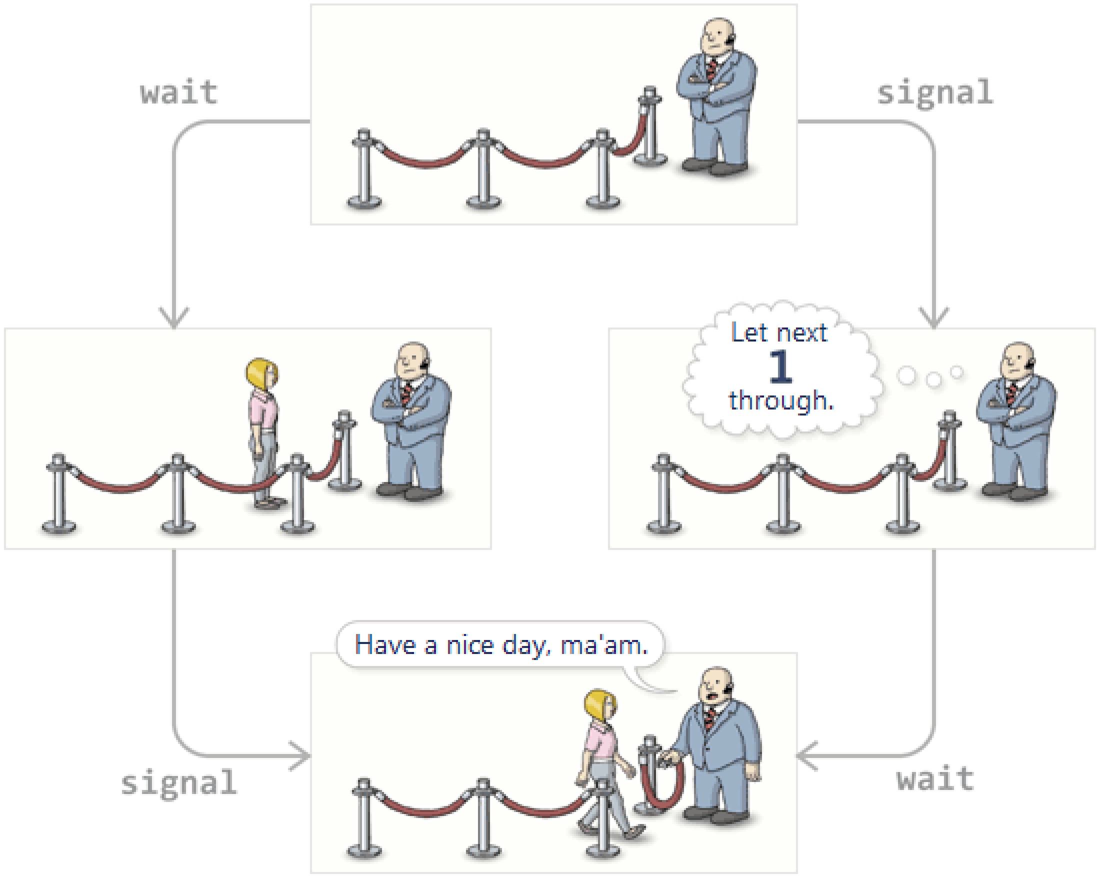
- What if signal()/post() an empty queue:



Source: preshing.com

More synchronization

- If wait and signal are both called some number of times, outcome is the same:



Source: preshing.com

Semaphore Limitations

- Can be hard to reason about synchronization
- Reason for waiting is embedded in the semaphore's P() / wait() operation
  - E.g. "if count == 0, then sleep"
  - Sometimes you want a more complex wait condition
  - E.g. "if x == 0 and (y > 0 or z > 0), then sleep"
  - That condition must be checked outside the P()
  - But checking requires mutual exclusion
    - Easy to get stuck this way

Abstract Example

```
//shared state variables
int x,y,z; // some initial values
//mutual exclusion for shared vars
Semaphore mutex = 1;
//semaphore to wait if necessary
Semaphore no_go = 0;

compute_a_thing {
   P(mutex);  //lock out others
   x = f1(x); //compute new x
   y = f2(y); //compute new y
   z = f3(z); //compute new z
   if (x != 0 || (y <= 0 && z <= 0))
        V(no_go);
   V(mutex); //up for grabs
}
```

```
use_a_thing {
   P(mutex); //lock out others
   if(x == 0 && (y > 0 || z > 0))
        P(no_go);
   // Now either x is non-zero
   // or y and z are non-positive
   // In this state, it is safe
   // to run “process” on x,y,z,
   // which may also change them.
   process(x,y,z);
   V(mutex);
}
```

- What can go wrong? How do we fix it?

What can go wrong?

- Deadlock.
  - If a *use_a_thing* thread has to wait on *no_go*, then no *compute_a_thing* thread can get *mutex* => no signal
- Changing state can "invalidate" previously sent signal (but signal can't be revoked), for example:
  - T1: *compute_a_thing*, new x=1, y=1, z=1, good => V(no_go)
  - T2: *compute_a_thing,* new x=0, y=1, z=0, bad => just skip "if"
  - T3: *use_a_thing,* sees "bad" values for x,y,z, goes to P(no_go)
    - P() returns immediately (count was 1)
    - Even though x,y,z are not in usable state.

count=1

count=1

Must always re-check state of x,y,z before using them.

Even after P() succeeds.

What can go wrong?

- Changing state can “invalidate” previously sent signal (but signal can’t be revoked) - other scenario:
  - T1: *compute_a_thing*, new x=1, y=1, z=1, V(no_go)

    count=1
  - T2: *compute_a_thing*, new x=2, y=2, z=1, V(no_go)

    count=2
  - T3: *use_a_thing,* sees x != 0 => good => proceed to run process(x,y,z)
    - Results in x=0, y=1, z=0 (“bad” values)
  - T4: *use_a_thing,* calls P() and returns immediately

    count=1
    - Even though x,y,z are not in usable state.

Again, must always re-check state of x,y,z before using them. Even after P() succeeds.

Fixed Example?

```
//shared state variables
int x,y,z; // some initial values
//mutual exclusion for shared vars
Semaphore mutex = 1;
//semaphore to wait if necessary
Semaphore no_go = 0;

compute_a_thing {
   P(mutex);  //lock out others
   x = f1(x);
   y = f2(y);
   z = f3(z);
   if (x != 0 || (y <= 0 && z <= 0))
        V(no_go);
   V(mutex); //up for grabs
}
```

```
use_a_thing {
   P(mutex); //lock out others
   while(x==0 && (y>0 || z>0)) {
        V(mutex); // no deadlock
        P(no_go);
        P(mutex);
   }
   // Now either x is non-zero
   // or y and z are non-positive
   process(x,y,z);
   V(mutex);
}
```

NOTE:

- Not very pretty, hard to read
- Semaphore count is not relevant
- May want to wake ALL waiters

- This is a very common pattern .. “Check condition, block and release mutex”

Condition Variables

- Abstract data type that encapsulates pattern of “release mutex, sleep, re-acquire mutex”
- Internal data is just a queue of waiting threads
    - Recall we didn’t need the semaphore count
- Operations are (each of these is atomic) – in pseudocode:
    - cv_wait(struct cv *cv, struct mtx *mutex)
        - Releases lock, waits, re-acquires mutex before return
    - cv_signal(struct cv *cv)
        - Wake one enqueued thread
    - cv_broadcast(struct cv *cv)
        - Wakes all enqueued threads

CAUTION: if no one is waiting, signal or broadcast has no effect.

Not recorded for later use, as with semaphore V().

Using Condition Variables

- Always used together with locks
- The lock protects the shared data that is modified and tested when deciding whether to wait or signal/broadcast
- General Usage:

```
lock_acquire(lock);
while(condition not true) {
      cv_wait(cond, lock);
}
...    // do stuff
cv_signal(cond); //or cv_broadcast(cond)
lock_release(lock);
```

Pthreads Condition Variables API

```
pthread_mutex_t mutex;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
(or pthread_cond_init() alternative)

pthread_cond_wait(&cv, &mutex);
pthread_cond_signal(&cv);
pthread_cond_broadcast(&cv);
```

Producer-Consumer revisited

- Why do we need synchronization?
- How do we solve the problem?
  - We could use semaphores
  - Or condition variables

Readers-Writers Problem

```
// Only one writer allowed.
// No reader while writer
// is writing

Writer() {




    Write(); // crit. sect.



}
```

```
// Multiple readers
// Readers don’t modify
// any data

Reader() {




     Read(); // crit. sect.



}
```

Readers-Writers Problem

```
// Number of readers
int readcount = 0;

//mutual exclusion to readcount
Semaphore mutex = 1;

//exclusive writer or reading
Semaphore w_or_r = 1;

Writer() {
  P(w_or_r);//lock out others
  Write();
  V(w_or_r);//up for grabs
}
```

```
Reader() {

  P(mutex); //protect readcount
  // one more reader
  readcount += 1;
  // is this the first reader?
  if(readcount == 1)
     //synch w/ writers
     P(w_or_r);
  V(mutex); //unlock readcount
  Read();
  P(mutex); //protect readcount
  readcount -= 1;
  //is this the last reader?
  if(readcount == 0)
     V(w_or_r);
  V(mutex);
}
```

Locks & Condition Variables

- Convenient way to
  - Provide mutual exclusion for critical sections
  - Block a thread that has to wait for something to happen
- What if the locking part was automatic?

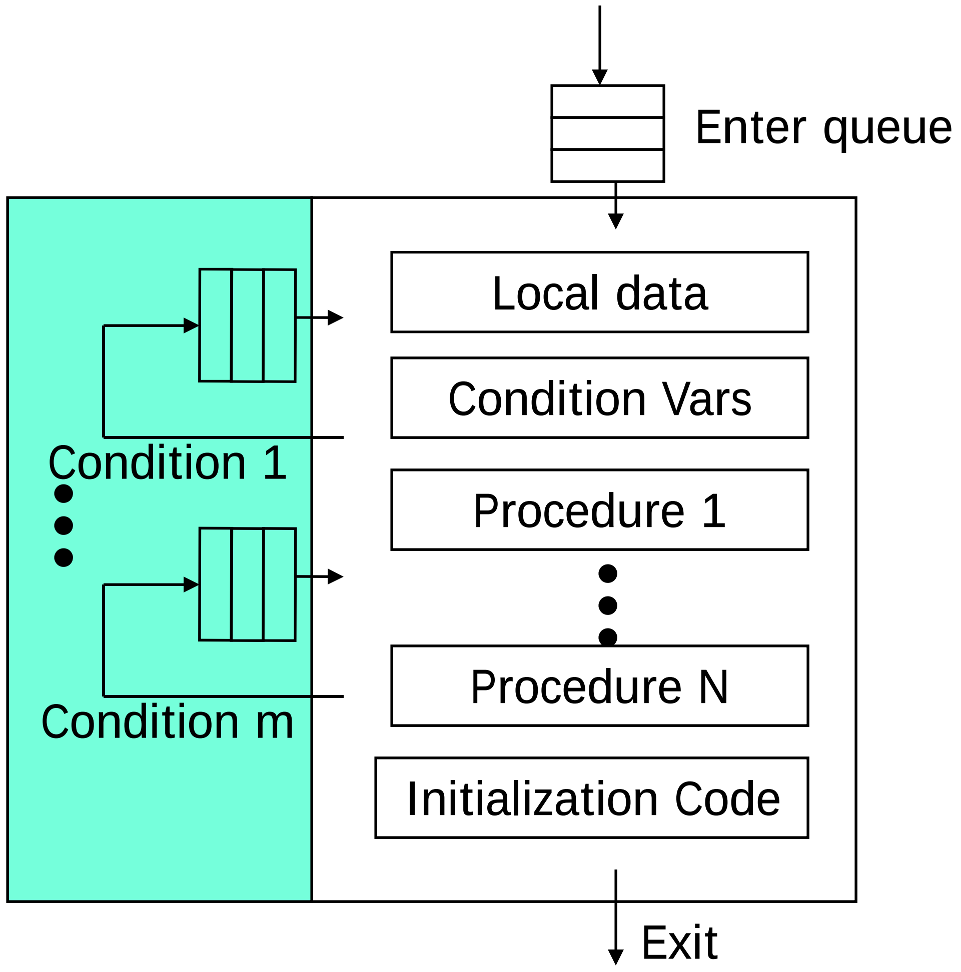
Monitors

- an *abstract data type* (data and operations on the data) with the restriction that only one process at a time can be active within the monitor
    - Monitor enforces mutual exclusion
    - Local data accessed only by the monitor's procedures (not by any external procedure)
    - A process *enters* the monitor by invoking one of its procedures
    - Other processes that attempt to enter monitor are blocked

Enforcing single access

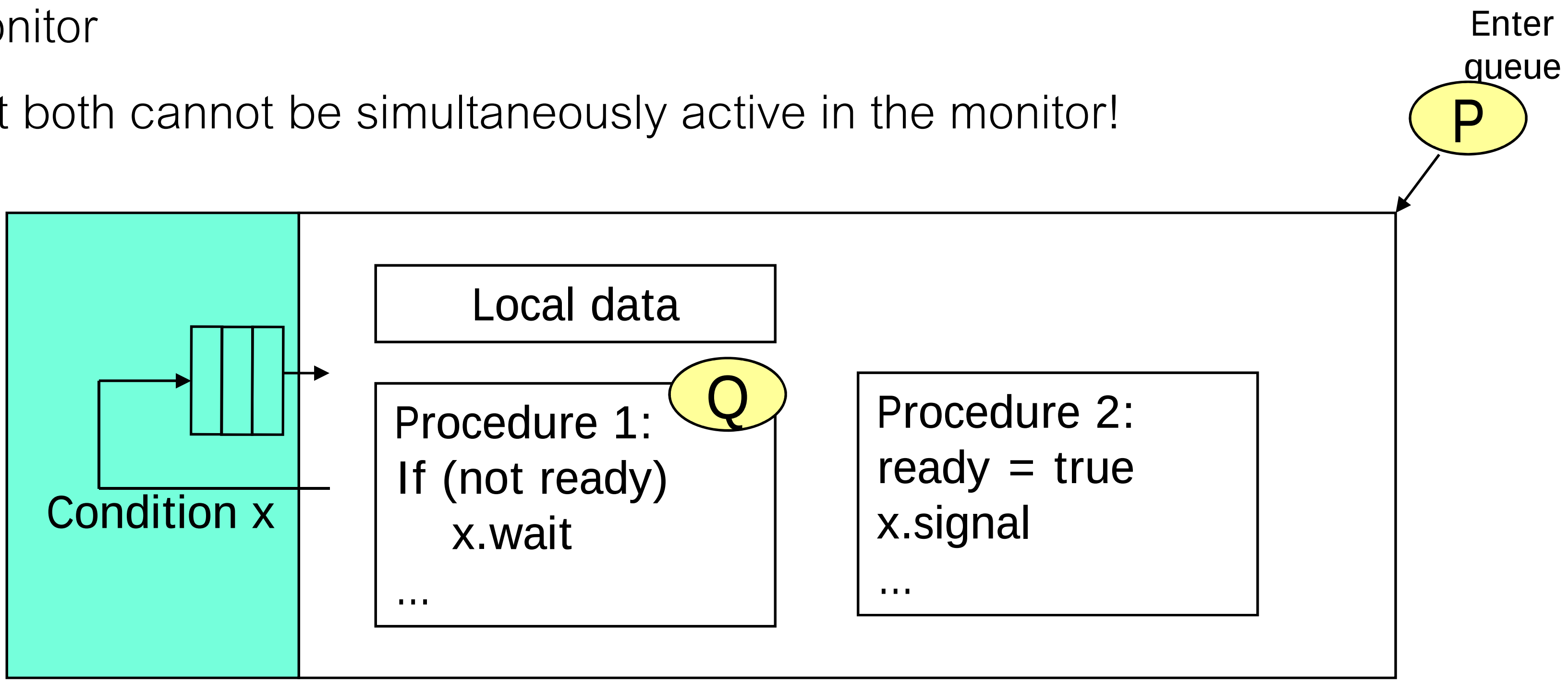
- A process in the monitor may need to wait for something to happen
  - May need to allow another process to use the monitor
  - Condition variables can be used for this
  - Operations on a *condition* variable are:
    - *wait* (suspend the invoking process)
    - *signal* (resume exactly one suspended process)
      - if no process is suspended, a *signal* has no effect (to be contrasted with the *V()* operation on semaphores, which always affects the state of the semaphore)

Monitor Diagram



More on Monitors

- If process *P* executes an *x.signal* operation and ∃ a suspended process *Q* associated with condition *x*, then we have a problem:
  - P is already “in the monitor”, does not need to block
  - Q becomes unblocked by the signal, and wants to resume execution in the monitor
  - but both cannot be simultaneously active in the monitor!



Monitor Semantics for Signal

- 2 possibilities exist
  - Hoare monitors (original)
    - signal() immediately switches from the caller to a waiting thread
    - The condition that the waiter was blocked on is guaranteed to hold when the waiter resumes
    - Need another queue for the signaler, if signaler was not done using the monitor
  - Mesa monitors (Mesa, Java, Nachos, OS/161)
    - Signal() places a waiter on the ready queue, but signaler continues inside monitor
    - Condition is not necessarily true when waiter resumes
    - Must check condition again

Hoare vs. Mesa Semantics

- Hoare

```
 if (empty)

      wait(condition);
```

Condition guaranteed to hold after signal. No need to recheck.

- Mesa

```
while(empty)

        wait(condition)
```

Condition *may not* be true when thread runs again. Must recheck.

- Tradeoffs
  - Hoare monitors make it easier to reason about program
  - Mesa monitors are easier to implement, more efficient, can support additional ops like broadcast

Using Monitors in C

- Not integrated with the language (as in Java)
- Bounded buffer: Want a monitor to control access to a buffer of limited size, N
  - Producers add to the buffer if it is not full
  - Consumers remove from the buffer if it is not empty
- Need two functions – add_to_buffer() and remove_from_buffer()
- Need one lock – only lock holder is allowed to be active in one of the monitor's functions
- Need two conditions – one to make producers wait, one to make consumers wait

Bounded Buffer Monitor – Variables

```
#define N 100
typedef struct buf_s {
  int data[N];     /* buffer storage */
  int inpos;       /* producer inserts here */
  int outpos;      /* consumer removes from here */
  int numelements; /* # items in buffer */
  struct lock *buflock;/* access to monitor */
  struct cv *notFull;  /* for producers to wait */
  struct cv *notEmpty; /* for consumers to wait */
} buf_t;

buf_t buffer;

void add_to_buff(buf_t *b, int value);
int remove_from_buff(buf_t *b);
```

Note: add/remove functions take a pointer to the buffer to operate on, so we can have multiple buffers.
Producers would call “add_to_buff(&buffer, newvalue);”
Consumers would call “remove_from_buff(&buffer);”

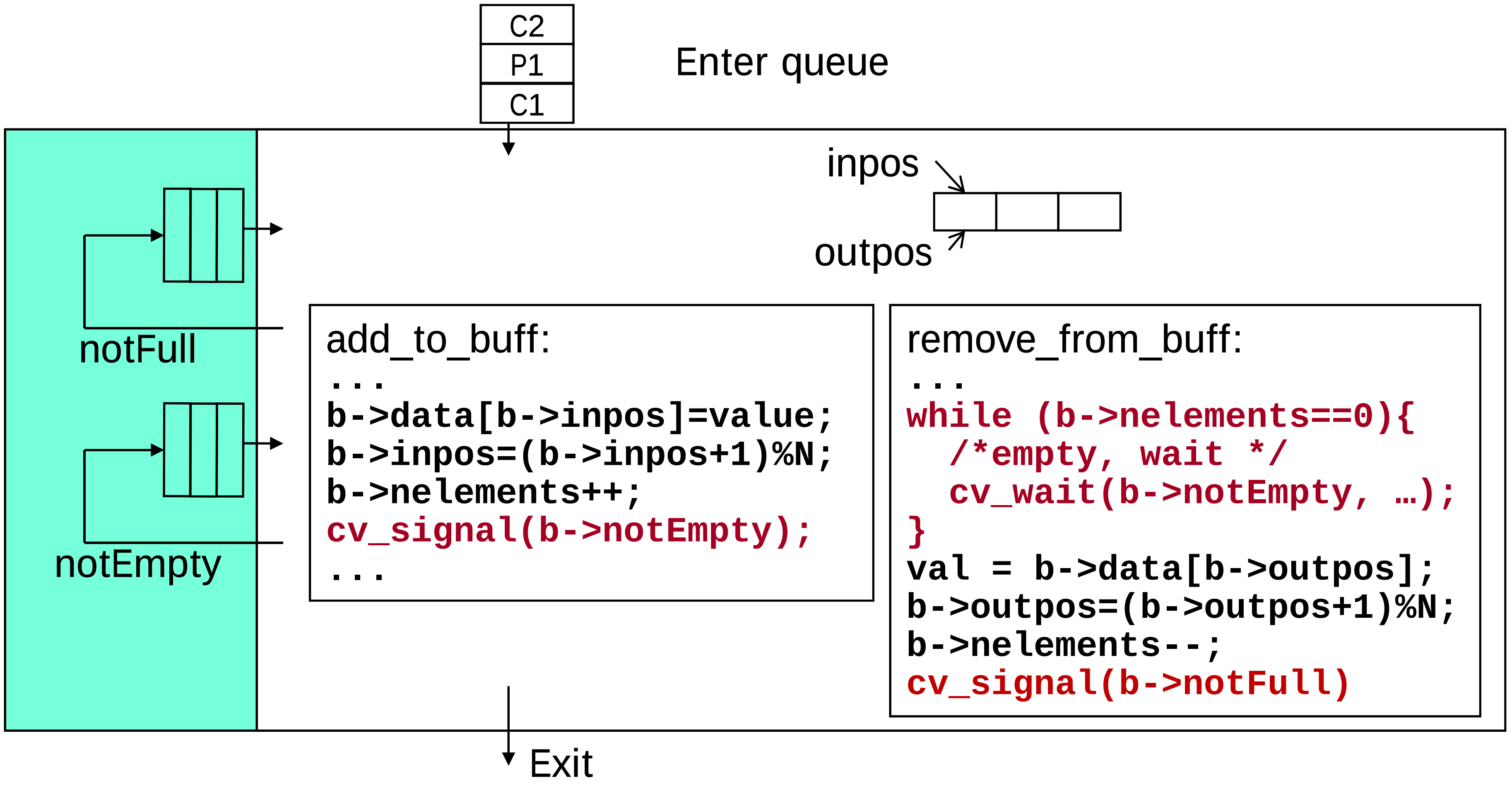
Bounded Buffer Monitor – Add Function

```
void add_to_buff(buf_t *b, int value) {
  lock_acquire(b->buflock);
    while (b->nelements == N) {
      /* buffer b is full, wait */
      cv_wait(b->notFull, b->buflock);
    }
    b->data[b->inpos] = value;
    b->inpos = (b->inpos + 1) % N;
    b->nelements++;
    cv_signal(b->notEmpty);
  lock_release(b->buflock);
}
```

Bounded Buffer Monitor - Remove

```
int remove_from_buff(buf_t *b) {
  int val;
  lock_acquire(b->buflock);
    while (b->nelements == 0) {
      /* buffer b is empty, wait */
      cv_wait(b->notEmpty, b->buflock);
    }
    val = b->data[b->outpos];
    b->outpos = (b->outpos + 1) % N;
    b->nelements--;
    cv_signal(b->notFull);
  lock_release(b->buflock);
  return val;
}
```

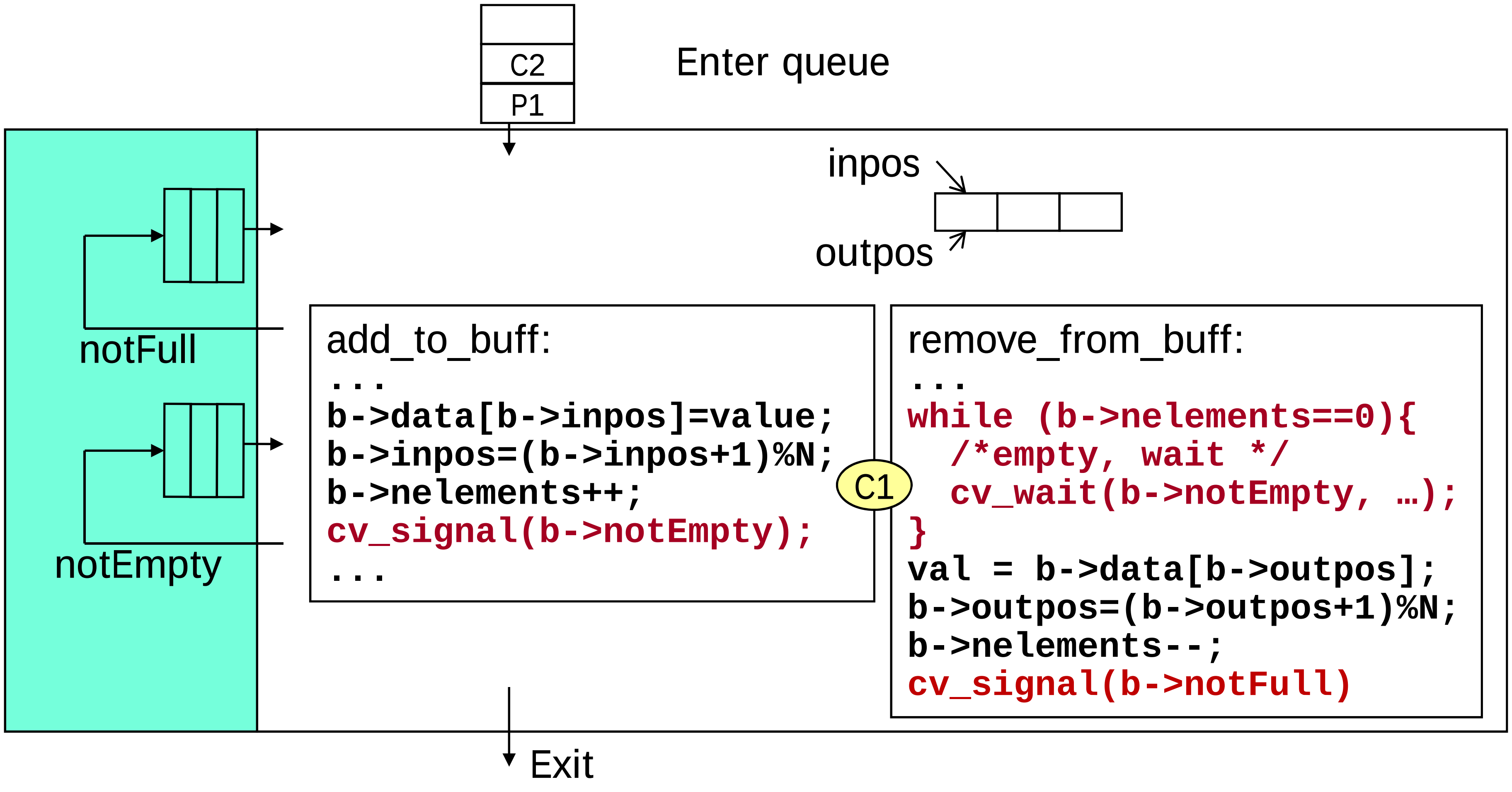
Possible Execution (MESA)



```
add_to_buff:
...
b->data[b->inpos]=value;
b->inpos=(b->inpos+1)%N;
b->nelements++;
cv_signal(b->notEmpty);
...
```

```
remove_from_buff:
...
while (b->nelements==0){
  /*empty, wait */
  cv_wait(b->notEmpty, …);
}
val = b->data[b->outpos];
b->outpos=(b->outpos+1)%N;
b->nelements--;
cv_signal(b->notFull)
```

Possible Execution (MESA)



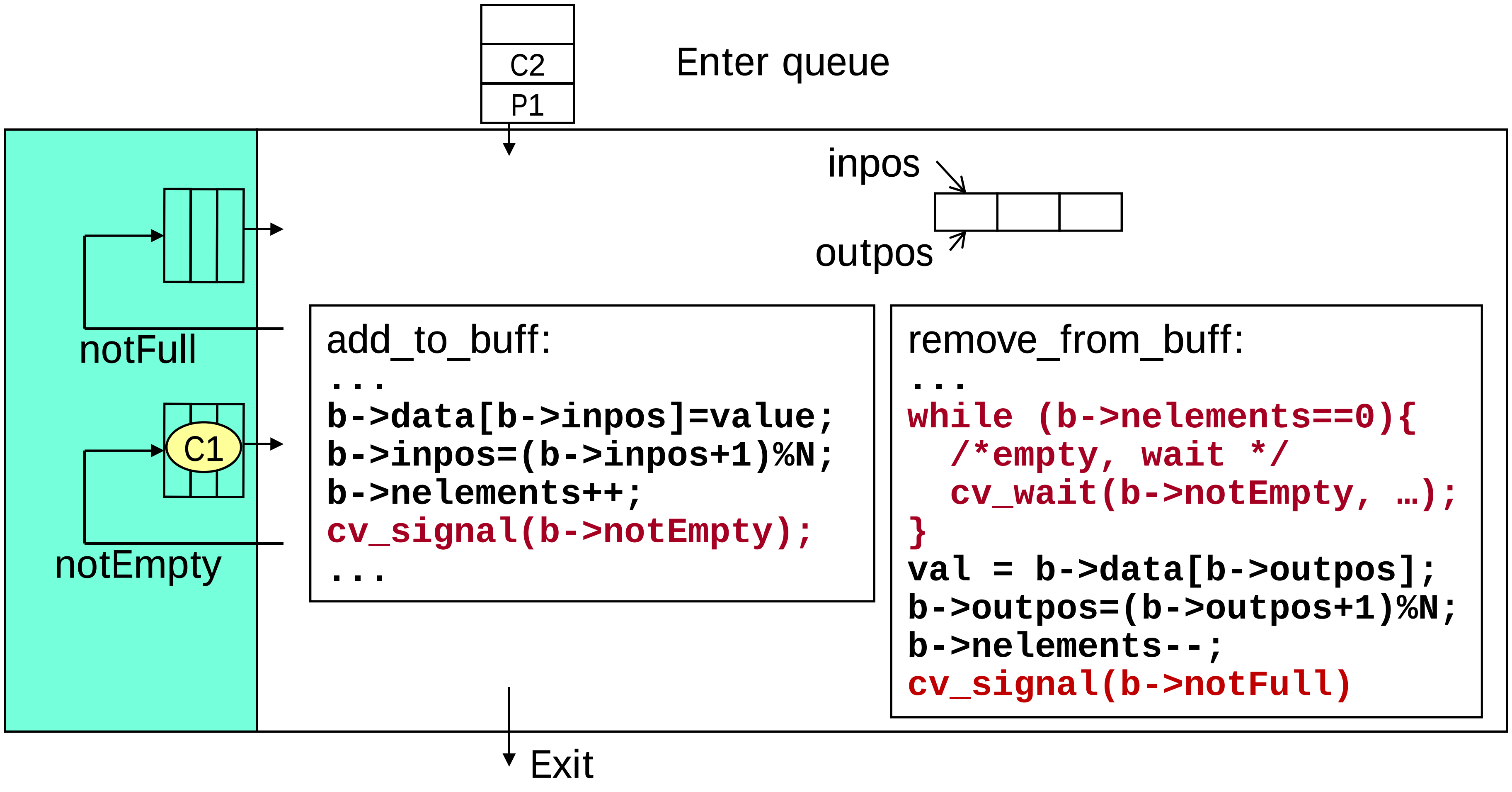
add_to_buff:

```
...
b->data[b->inpos]=value;
b->inpos=(b->inpos+1)%N;
b->nelements++;
cv_signal(b->notEmpty);
...
```

remove_from_buff:

```
...
while (b->nelements==0){
  /*empty, wait */
  cv_wait(b->notEmpty, …);
}
val = b->data[b->outpos];
b->outpos=(b->outpos+1)%N;
b->nelements--;
cv_signal(b->notFull)
```

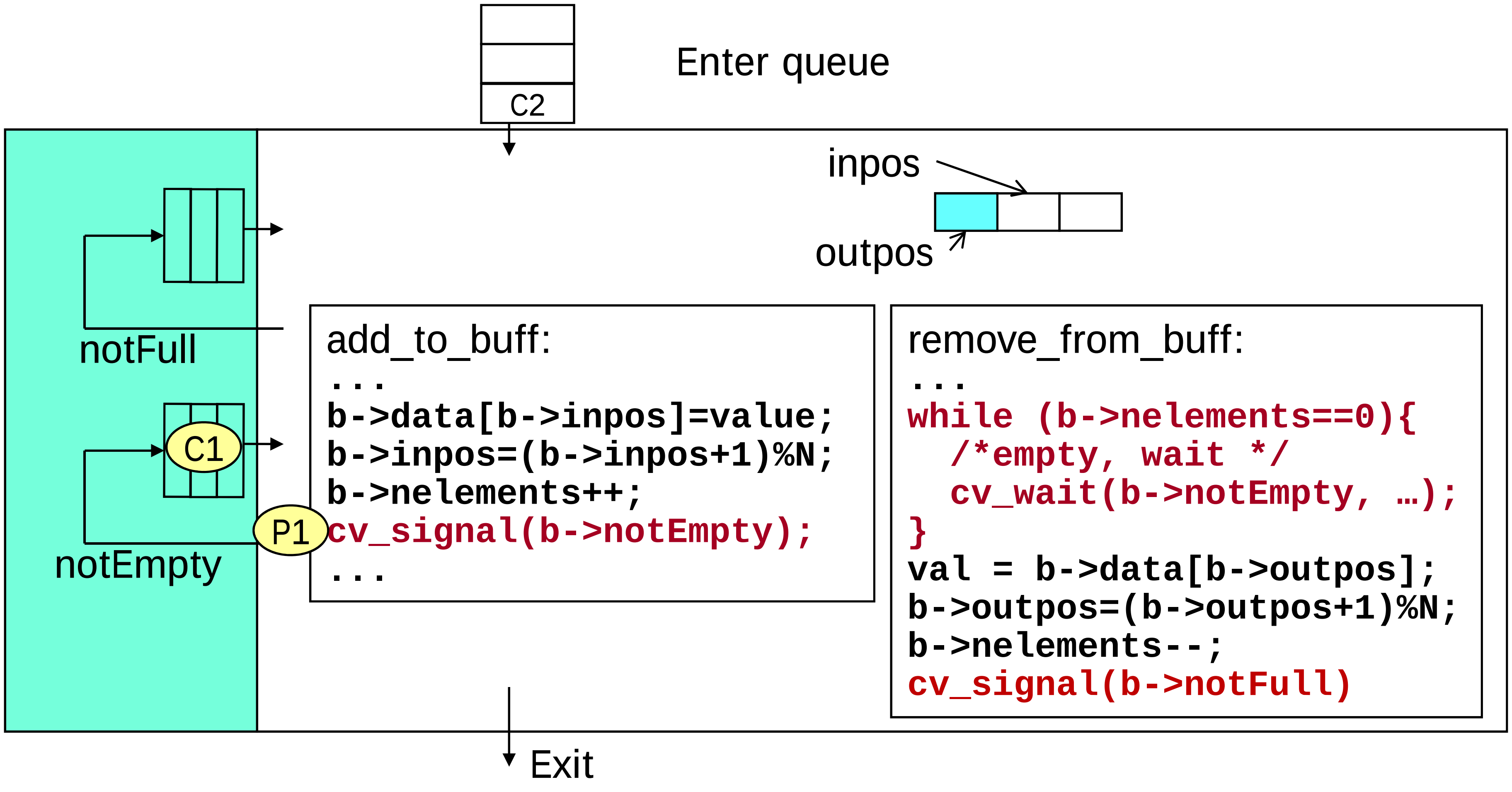
Possible Execution (MESA)



```
add_to_buff:
...
b->data[b->inpos]=value;
b->inpos=(b->inpos+1)%N;
b->nelements++;
cv_signal(b->notEmpty);
...
```

```
remove_from_buff:
...
while (b->nelements==0){
  /*empty, wait */
  cv_wait(b->notEmpty, …);
}
val = b->data[b->outpos];
b->outpos=(b->outpos+1)%N;
b->nelements--;
cv_signal(b->notFull)
```

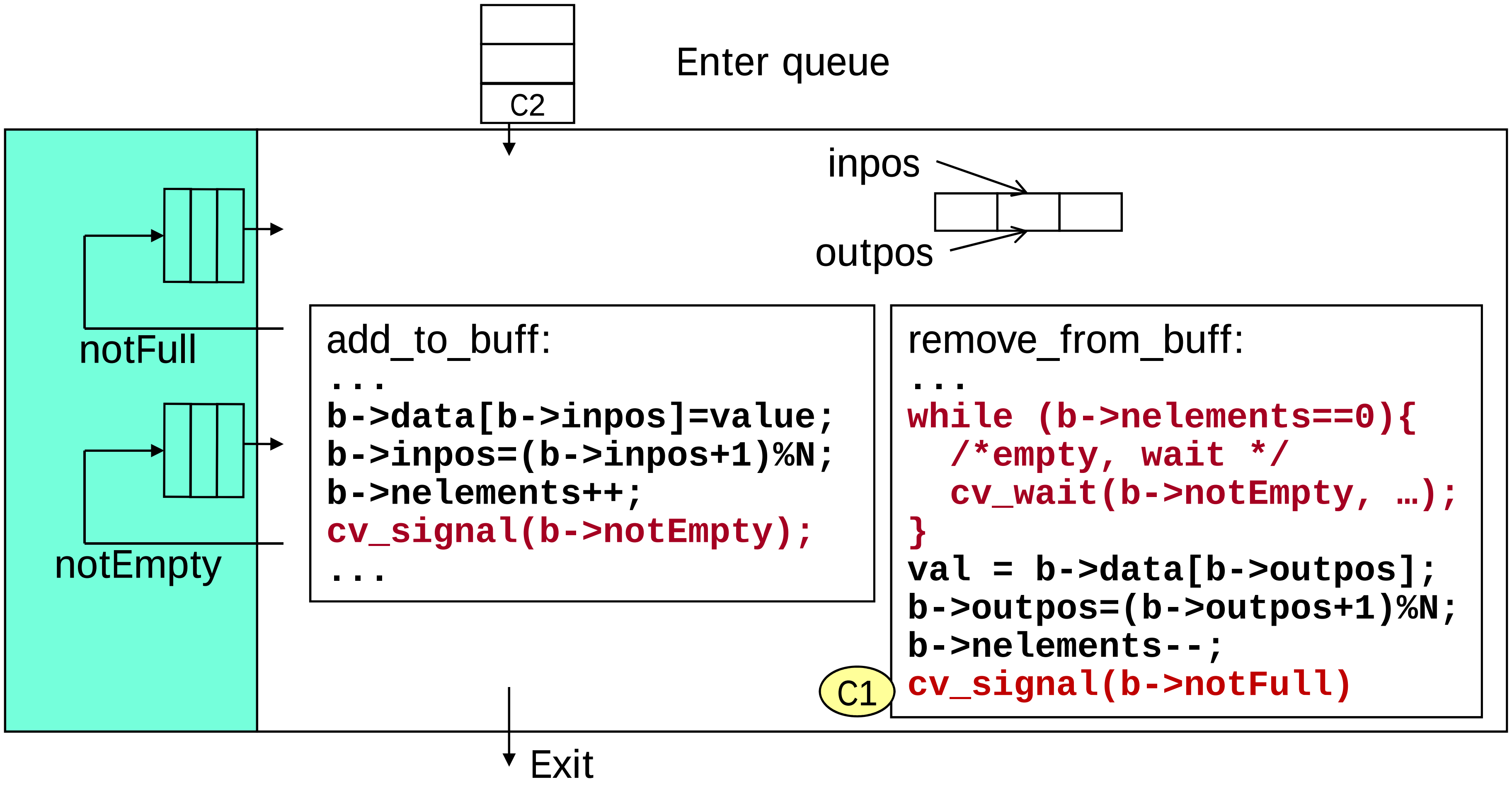
Possible Execution (MESA)



```
add_to_buff:
...
b->data[b->inpos]=value;
b->inpos=(b->inpos+1)%N;
b->nelements++;
cv_signal(b->notEmpty);
...
```

```
remove_from_buff:
...
while (b->nelements==0){
  /*empty, wait */
  cv_wait(b->notEmpty, …);
}
val = b->data[b->outpos];
b->outpos=(b->outpos+1)%N;
b->nelements--;
cv_signal(b->notFull)
```

Possible Execution (MESA)



add_to_buff:

```
...
b->data[b->inpos]=value;
b->inpos=(b->inpos+1)%N;
b->nelements++;
cv_signal(b->notEmpty);
...
```

remove_from_buff:

```
...
while (b->nelements==0){
  /*empty, wait */
  cv_wait(b->notEmpty, …);
}
val = b->data[b->outpos];
b->outpos=(b->outpos+1)%N;
b->nelements--;
cv_signal(b->notFull)
```

Notes on Monitors

- Bounded buffer example illustrates pattern for using locks and condition variables in monitors:
  1. The first step in any function is to acquire the lock on the monitor
  2. The last step before returning is to release the lock on the monitor
     - Be careful with early returns due to errors!
  3. Call `cv_wait()` only inside `while` loops (Mesa)
  4. Whenever one of the conditions being waited on *might* have changed from FALSE to TRUE, signal the corresponding condition variable
     - Signaller need not know previous state, only that current state is TRUE

More monitor examples

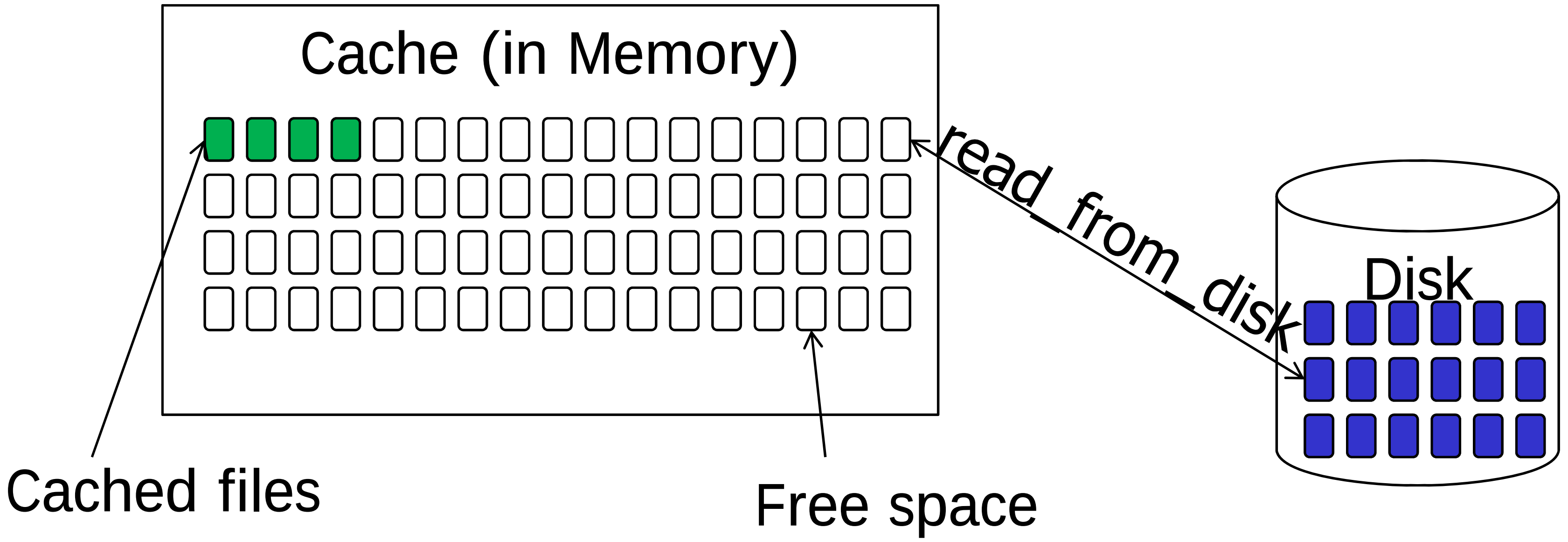
Optional reading, just to see how monitors are used in some OS contexts

Exercise: Cache Synchronization

- Files are stored on disk, but are copied into memory when some process issues a read system call
- Popular files may be read by many processes in a short period of time
- OS will *cache* recently used files in memory
  - To handle read system call, first check cache
    - If present, copy to process-supplied address from cache
    - If missing, get space in cache, read from disk, copy to process-supplied address from cache
  - Subsequent reads can access in-memory copy instead of waiting for the disk
- To simplify, assume files are all the same size, cache is unlimited (no replacement)

Cache Setup

- Read from cache if present, else read from disk



Synchronization Requirement

- No file should be in the cache more than once
  - i.e. If Process A and Process B issue reads for file F concurrently, and F is not already cached, then either A or B (but not both) will read F from disk into the cache, and the other will read F from the cache.
  - The second reader must wait for the first read to complete, so that it can read from the cache
- Implement cache as a monitor with single external function, `get_file_from_cache(char *filename)`
  - Assume some internal functions:
    - `char *cache_lookup(char *filename)`
    - `char *cache_get_space(char *filename)`
  - And `read_from_disk(char *buf, char *filename)`

Notes on assumed functions

- `char *cache_lookup(char *filename)`
  - Returns a pointer to the file data in memory, if the file with name “filename” is found in the cache.
  - Returns NULL if the named file is not in the cache
- `char *cache_get_space(char *filename)`
  - Returns a pointer to the space in the cache allocated to hold the file data and associates “filename” with this space so that future calls to cache_lookup can find it.
  - Cannot fail, since we are assuming the cache does not run out of space.
- Also assume a single global cache structure, with a global lock variable, `cache_lock`

Cache Synchronization Solution

```
/* Returns pointer to file data in memory */
char *get_file_from_cache(char *filename) {
    char *fdata;
    lock_acquire(cache_lock);
    fdata = cache_lookup(filename);
    if (!fdata) {
        fdata = cache_get_space(filename);
        read_from_disk(fdata, filename);
    }
    lock_release(cache_lock);
    return fdata;
}
```

What’s wrong with our solution?

- One answer: Nothing. The lock ensures that only 1 reader can find the file missing from cache and read it from disk. Additional readers must block on entry to the monitor function. When the lock is released, the file is present in the cache.
- But...
  - Reading from disk is slow
  - the cache holds many files
  - *no process can read a file from the cache if any process is reading from the disk!*
- We should release the lock before reading from disk
  - But we still need to make other readers of the same file wait

Solution, Take 2

- Need a struct to represent a file in cache, including a pointer to its data in memory, and info about its status:

```
struct cache_block {
    char *fdata; /* ptr to file data */
    char *fname; /* file name, for lookup */
    int status;  /* state of cache block */
}
```

- What are the different states that `status` represents?
    - File is already in cache – PRESENT_CACHE_ENTRY
    - Space has been allocated for file, but not yet read – NEW_CACHE_ENTRY
- Can we just rewrite `get_file_from_cache()`?
    - Not really. Releasing lock means we are leaving the monitor

The view from outside

- Cache monitor now provides 2 functions, used like this:

```
 ...
char *fdata;
result = cache_find(&fdata, filename);
if (result == NEW_CACHE_ENTRY) {
    read_from_disk(fdata, filename);
    cache_set_present(filename);
}
/* now copy to user address from fdata */
...
```

- How do we make other readers of the same file wait?
  - We need a condition variable – can make it an element of the

    `struct cache_block`   e.g. `struct cv *cb_cv`

New monitor function: cache_find

```
int cache_find(char **fdata, char *filename) {
    struct cache_block *cb;
    lock_acquire(cache_lock);
    cb = cache_lookup(filename);
    if (!cb) {
        cb = cache_get_space(filename); //sets cb->status to NEW
        goto out;
    }
    // get here if cache block for filename already exists. Now what?
   while (cb->status == NEW_CACHE_ENTRY)
       cv_wait(cache_lock, cb->cb_cv);
out:
    *fdata = cb->fdata;
    lock_release(cache_lock);
    return cb->status;
}
```

New monitor function: cache_set_present

```
int cache_set_present(char *filename) {
    int result = 0;
    struct cache_block *cb;
    lock_acquire(cache_lock);
    cb = cache_lookup(filename);
    if (!cb) {
        result = EINVAL; /* should not fail, no replacement */
    } else {
        cb->status = PRESENT_CACHE_ENTRY; /* set present */
        /* wake waiters – cv_broadcast is used because multiple
          * readers could be waiting and all can continue now. */
        cv_broadcast(cache_lock, cb->cb_cv);
    }
    lock_release(cache_lock);
    return result;
}
```

Deadlock and Starvation

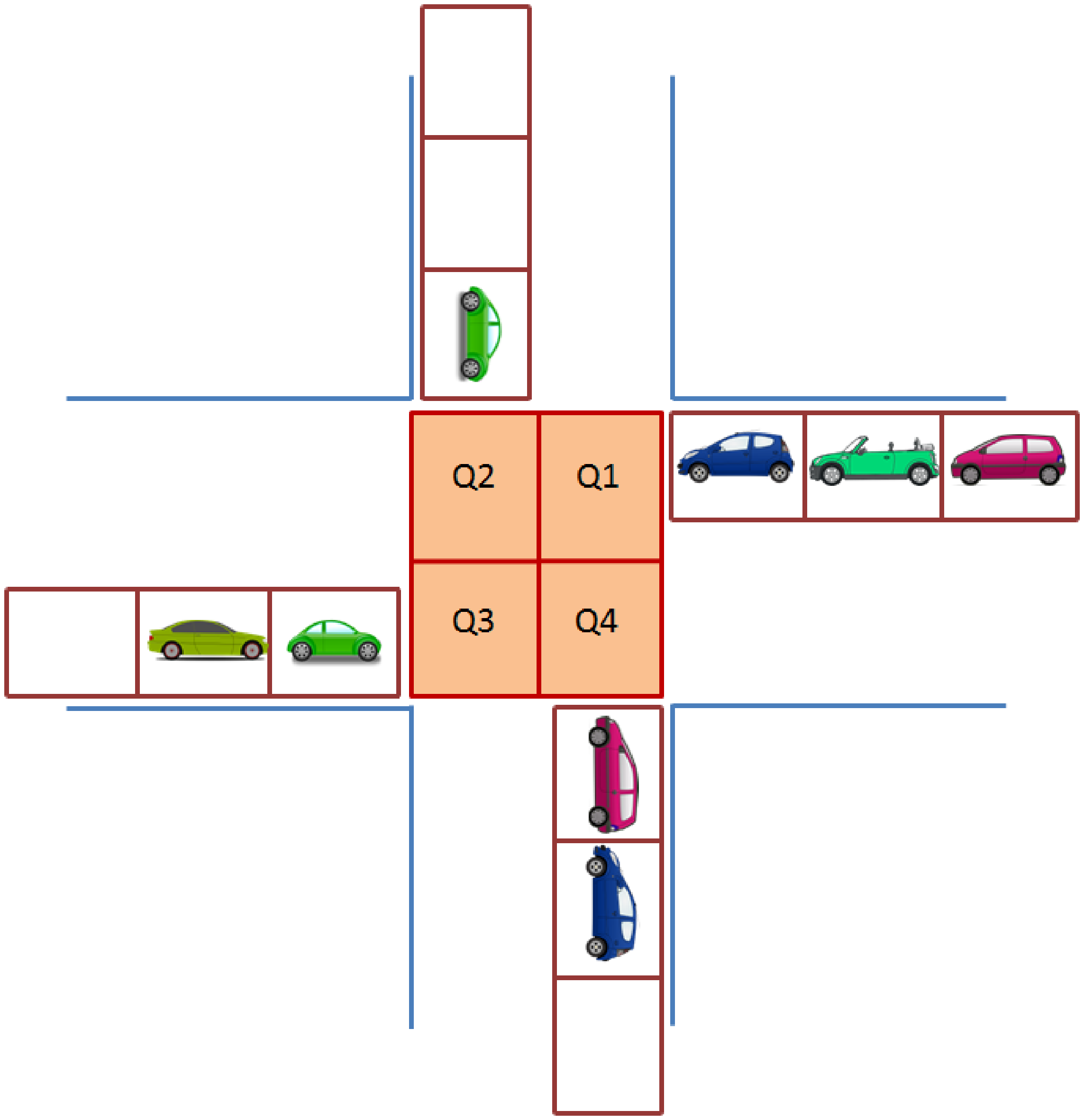
- a set of threads is in a *deadlocked* state when every process in the set is waiting for an event that can be caused only by another process in the set
  - in the case of semaphores, the event is the execution of the *signal* operation

$\neq$

- *starvation* (or indefinite postponement) happens if a thread is waiting indefinitely because other threads are in some way preferred
  - This is often a scheduling problem

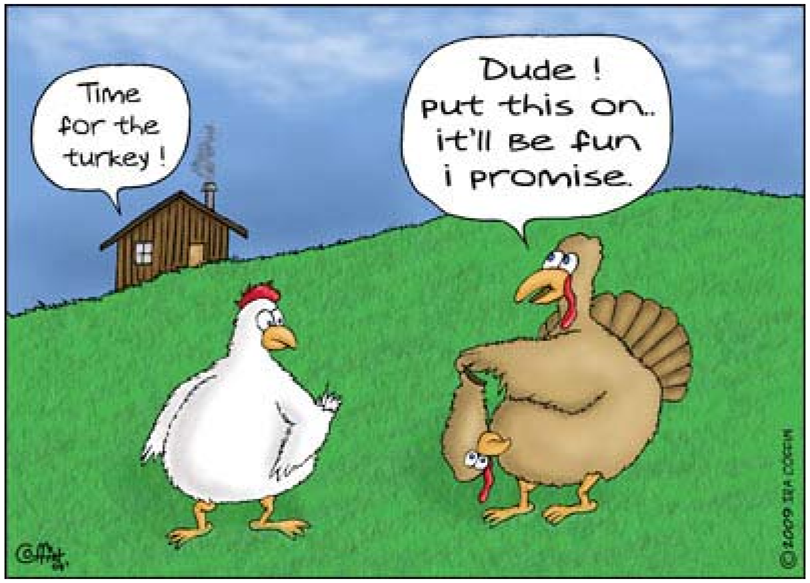
Assignment 2

- To be posted
- Implement traffic monitor
  - Cars arrive at an intersection
  - Lanes are represented as buffers, cars cannot “skip”
  - Once at the front of the lane, cars may proceed through the intersection IF they are allowed to occupy all the corresponding quadrants (depending on their outbound direction)



Next Time…

- No class on Monday – happy Thanksgiving!



- Wednesday: Introduction to scheduling
- A2 to be released ...