Synchronization

Prof. Bracy and Van Renesse

CS 4410

Cornell University

based on slides designed by Prof. Sirer

Announcements

- If you are on the fence about this class
  - Today is a good day (today is Add Deadline)
- Not in CMS?
- 4411 Projects **must be done in pairs**
  - According to CMS, many of you are not paired up.
    - Please pair up on CMS
    - Please meet at the blackboard after class to find a partner or post on 4411 piazza.

Threads share memory

Threads have:

- Private registers
  - *context switching* saves and restores registers when switching from thread to thread
- Shared “global” memory
  - *global* means not stack memory
- Usually private stack
  - pointers into stacks across threads frowned upon

Two threads, one variable

Two threads updating a single shared variable “amount”

- One thread wants to decrement amount by $10K
- The other thread wants to decrement amount by 50%

```
amount = 100,000;
```

```
...
amount = amount - 10,000;
...
```

```
...
amount = 0.50 * amount;
...
```

What happens when two threads execute concurrently?

Two threads, one variable

amount = 100,000;

```
...
r1 = load from amount
r1 = r1 - 10,000;
store r1 to amount
...
```

```
...
r2 = load from amount
r2 = 0.5 * r2
store r2 to amount
...
```

amount= ?

Two threads, one variable

amount = 100,000;

```
...
r2 = load from amount
r2 = 0.5 * r2
store r2 to amount
...
```

```
...
r1 = load from amount
r1 = r1 - 10,000;
store r1 to amount
...
```

amount= ?

Two threads, one variable

amount = 100,000;

```
…
r1 = load from amount
r1 = r1 - 10,000;
store r1 to amount
…
```

```
…
r2 = load from amount

r2 = 0.5 * r2
store r2 to amount
…
```

amount= ?

Shared counters

- One possible result: everything works!
  - although different, either order is correct
- Another possible result: lost update!
  - ⇒ Wrong
  - ⇒ Difficult to debug

Called a "race condition"

Race conditions

- **Definition:** *timing dependent error involving shared state*
  - Once thread A starts, it needs to "race" to finish
  - Whether RC happens depends on thread schedule
    - different "*schedules*" or "*interleavings*" (total order on machine instructions)
- All possible interleavings should be *safe*
  - Correspond to some sequential order of user-defined "operations" (here: withdraw, pay-taxes, *etc.*)

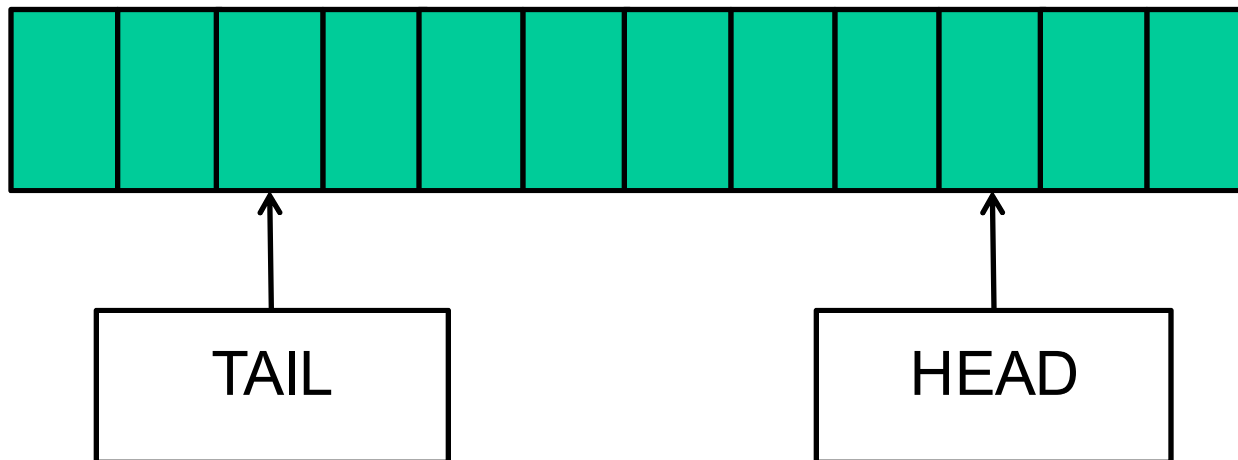
Race conditions…

…are hard to detect and debug:

- Number of possible interleavings is huge
- Some interleavings are good
- Some interleavings are bad:
  - But bad interleavings may rarely happen!
  - **Works 100x ≠ no race condition**
- Timing dependent = small changes can hide bug

Example: races with queues

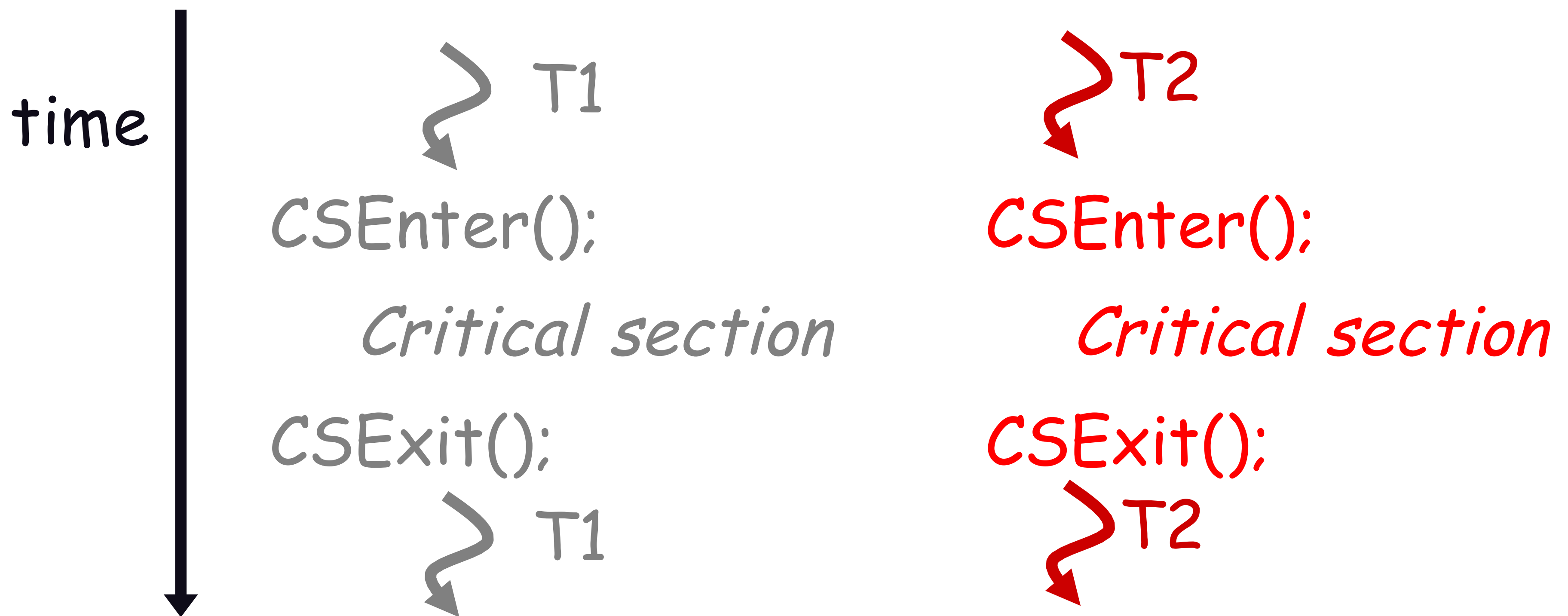
- 2 concurrent enqueue() operations?
- 2 concurrent dequeue() operations?



What could possibly go wrong?

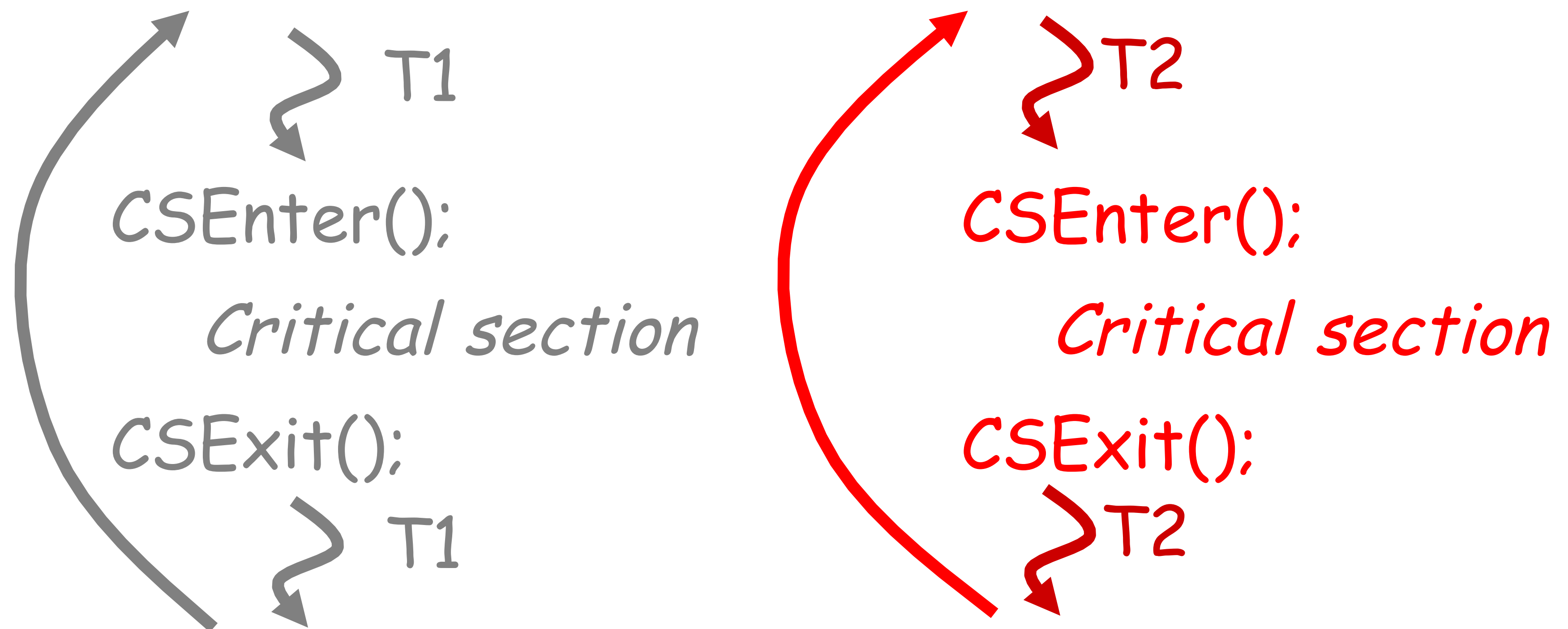
Critical Section

Code that can be executed by only one thread at a time



Critical Section

Perhaps the threads loop (perhaps not!)



Critical Section Goals

- We would like
  - Safety: No more than one thread can be in a critical section at any time
  - Liveness: A thread that is seeking to enter the critical section will eventually succeed
  - Fairness: If two threads are both trying to enter a critical section, they have equal chances of success
- … in practice, fairness is rarely guaranteed

Too much milk problem

- Two roommates want to ensure that the fridge is always stocked with milk
  - If the fridge is empty → need to restock it
  - But they don't want to buy too much milk
- Caveats
  - Can only communicate by reading and writing onto a notepad on the fridge
  - Notepad can have different cells, labeled by a string (just like variables)
- Write the pseudo-code to ensure that at most one roommate goes to buy milk

Solving the problem

A first idea: no protection

```
if fridge_empty():
    buy_milk()
```

```
if fridge_empty():
    buy_milk()
```

- Is this Safe? Live? Fair?

Solving the problem

A second idea:

- Have a boolean flag, *out-to-buy-milk*. Initially false.

```
while(outtobuymilk)
    do_nothing();
if fridge_empty():
    outtobuymilk :=true
    buy_milk()
    outtobuymilk := false
```

```
while(outtobuymilk)
    do_nothing();
if fridge_empty():
    outtobuymilk :=true
    buy_milk()
    outtobuymilk := false
```

– Is this Safe? Live? Fair?

Solving the problem

A third idea:

- Have two boolean flags, one for each roommate. Initially false.

```
greenbusy := true
if not redbusy and
      fridge_empty():
   buy_milk()
greenbusy := false
```

```
redbusy := true
if not greenbusy and
      fridge_empty():
   buy_milk()
redbusy := false
```

– Is this Safe? Live? Fair?

Solving the problem

A fourth idea:

- Have two boolean flags, one for each roommate. Initially false. *Asymmetric*

```
greenbusy = true
while redbusy:
    do_nothing()
if fridge_empty():
    buy_milk()
greenbusy = false
```

```
redbusy = true
if not greenbusy and
    fridge_empty():
    buy_milk()
redbusy = false
```

– Is this Safe? Live? Fair?

Solving the problem

A fourth idea:

- Have two boolean flags, one for each roommate. Initially false. *Asymmetric*

```
greenbusy = true
while redbusy:
    do_nothing()
if fridge_empty():
    buy_milk()
greenbusy = false
```

```
redbusy = true
if not greenbusy and
    fridge_empty():
    buy_milk()
redbusy = false
```

- Really complicated, even for a simple example, hard to ascertain that it is correct
- Asymmetric code, hard to generalize, unfair

Solving the problem, really

The final attempt (Peterson's solution):

- Adding another binary variable: *turn*: { red, green }

```
greenbusy := true
turn := red
while redbusy and
        turn == red:
    do_nothing()
if fridge_empty():
    buy_milk()
greenbusy := false
```

```
redbusy := true
turn := green
while greenbusy and
        turn == green:
    do_nothing()
if fridge_empty():
    buy_milk()
redbusy := false
```

- Really complicated, even for a simple example, hard to ascertain that it is correct
- Hard to generalize, inefficient, …

Hardware Solution

- Use more powerful hardware primitives to provide a mutual exclusion primitive
- Typically relies on a multi-cycle bus operation that atomically reads and updates a memory location
- Example Conceptual Spec of *Test-And-Set*:

```
ATOMIC  int TestAndSet(int *var) {
    int oldVal := *var;
    *var := 1;
    return oldVal;
}
```

Buying Milk Solved with TAS

Shared variable: int outtobuymilk, initially 0

```
while(TAS(&outtobuymilk) == 1)
    do_nothing();
 if fridge_empty():
     buy_milk()
outtobuymilk := 0
```

```
while(TAS(&outtobuymilk) == 1)
    do_nothing();
 if fridge_empty():
     buy_milk()
outtobuymilk := 0
```

Spinlocks

```
spinlock_acquire(int *lock) {
   while(test_and_set(lock) == 1)
      /* do nothing */;
}
spinlock_release(int *lock) {
   *lock = 0;
}
```

Buying Milk with Spinlock

Shared spinlock: int outtobuymilk, initially 0

```
spinlock_acquire(&outtobuymilk);
if fridge_empty():
     buy_milk()
spinlock_release(outtobuymilk);
```

```
spinlock_acquire(&outtobuymilk);
 if fridge_empty():
     buy_milk()
spinlock_release(outtobuymilk);
```

Spinlock Issues

- Participants not in critical section must spin → wasting CPU cycles
  - Replace the "do nothing" loop with a "yield()" ? Processes would still be scheduled and descheduled

- Need better primitive:
  - allows one process to pass through
  - all others to sleep until they can be executed again

Semaphore

Dijkstra 1962

- Non-negative integer with atomic increment and decrement
  - S := new Semaphore(initial_value)   *// must initialize!*
- Can only be modified by:
  - P(S): decrement or block if already 0
  - V(S): increment and wake up waiting thread if any
  - *No interface to read the value*
- These operations have the following semantics

```
P(S){
    while(S == 0)
        ;
    S -= 1;
}
```

```
V(S){
    S += 1;
}
```

Semaphore implementation for true parallelism

```
struct Sema { int lock = 0; int count; };

P(Sema *s) {
    for ever
        if (test_and_set(&s->lock) == 0) {
            if (s->count > 0) break;
            else s->lock := 0;      // OUCH --- busy waiting until count > 0
        }
    s->count -= 1;
    s->lock := 0;
}
V(Sema *s) {
    while (test_and_set(&s->lock) == 1)
        /* do nothing */;
    s->count += 1;
    s->lock := 0;
}
```

Semaphore implementation for non-preemptive threading

```
struct Sema { Queue waitQ; int count; };

P(Sema *s) {
    if (s->count > 0) s->count -= 1;
    else {
        s->waitQ.enq(curThread);
        thread_stop();      // sets status to WAITING and runs another thread
        // continues here after thread is restarted using thread_start()
    }
}
V(Sema *s) {
    if (s->waitQ.empty()) s->count += 1
    else {
        assert(s->count == 0);
        Thread t = s->waitQ.deq();
        thread_start(t);        // sets status to RUNNABLE
    }
}
```

can be made to work for pre-emptive threading on a uniprocessor by disabling interrupts

Binary Semaphore

- Semaphore value is either 0 or 1
  - Used for mutual exclusion (sema as a more efficient lock)
  - Initially 1 in that case:

```
semaphore S
S.init(1);

Thread1():                    Thread2():
  P(S);                         P(S);
  CriticalSection();            CriticalSection();
  V(S);                         V(S);
```

Counting Semaphores

- Sema count can be any integer
  - Used for signaling, or counting resources
  - Typically: one thread performs P() to wait for event, another thread performs V() to alert waiting thread that an event occurred

```
semaphore packetarrived
packetarrived.init(0);

PacketProcessor():
    x = retrieve_packet_from_card();
    enqueue(packetq, x);
    V(packetarrived);

NetworkingThread():
    P(packetarrived);
    x = dequeue(packetq);
    print_contents(x);
```

Classical Synchronization
Problems

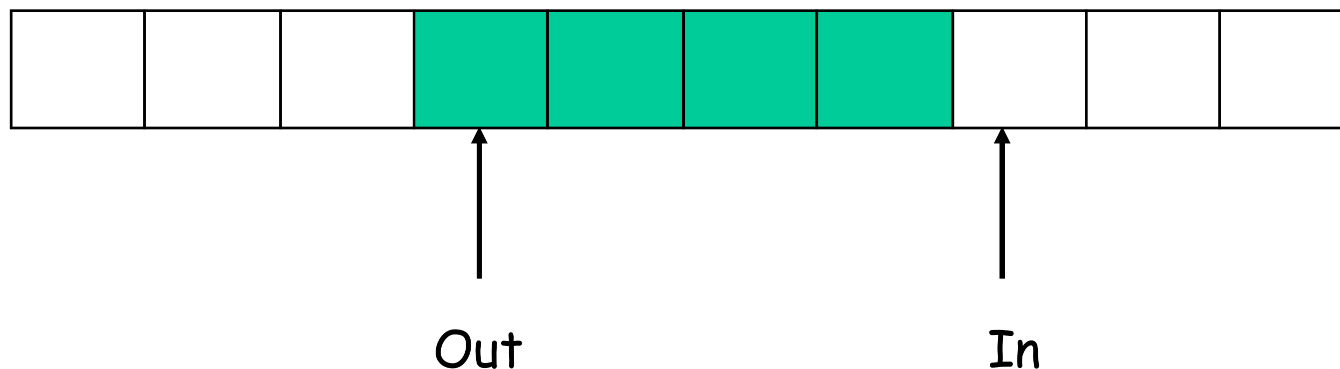
Bounded Buffer

- 2+ threads communicate with some threads *producing* data that others *consume*

- **Example:** compiler preprocessor produces a source file that compiler's parser consumes

Producer-Consumer Problem

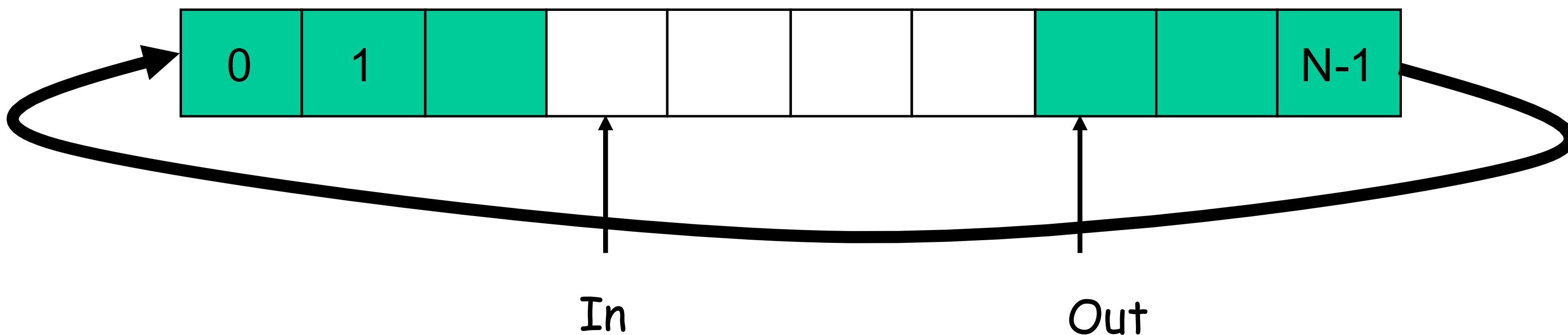
- Imagine an unbounded (infinite) buffer
- Producer process writes data to buffer
  - Writes to In and moves rightwards
- Consumer process reads data from buffer
  - Reads from Out and moves rightwards
  - Should not try to consume if there is no data



Need an infinite buffer

Producer-Consumer Problem

- Bounded buffer: size **N**
  - Access entry 0… N-1, then "wrap around" to 0 again
- Producer process writes data to buffer
  - Don't write more than N "un-eaten" items!
- Consumer process reads data from buffer
  - Don't consume if there is no data!



Producer-Consumer Code, v1

```
int array[N];
int in, out;
```

```
void produce (int item) {
  // add item to buffer
   array[in] = item;
   in++;
}
```

[x = (x +1)%N or wrap-around inc. of your choice]

```
int consume() {
    // remove item
    int item = array[out];
    out++;
    return item;
}
```

Problems:

- Consumer could consume when nothing is there!
- Producer could overwrite not-yet-consumed data!

Solving with semaphores

Use of 2 Semaphores offers a **clean & simple** solution

nFilled: keeps track of buffer entries *in use*; *ensures consumer only consumes when something is there*

- initialized to 0,
- incremented by producer
- decremented by consumer

nEmpty: keeps track of *empty* buffer entries; *ensures producer only produces when there is room in the buffer*

- initialized to N
- decremented by producer
- incremented by consumer

Producer-Consumer Code, v2

```
Shared: Semaphores nEmpty , nFilled;
Init: nEmpty = N;     /* # empty buffer entries */
      nFilled = 0;    /*    #  full buffer entries */
```

```
int array[N];
int in, out;
```

```
void produce (int item) {
  P(nEmpty);  // verify room for item
  // add item to buffer
  array[in] = item;
  in++;
  V(nFilled);  // "new item!"
}
```

```
int consume() {
    P(nFilled);  // verify item there
    // remove item
    int item = array[out];
    out++;
    V(nEmpty);  // "more room!"
    return item;
}
```

Does v2 work?

Observation:

- Producer & consumer each have own indices (in,out)
- Semaphores prevent concurrent reading/writing of same buffer entry

→ **Works!** *But only if there is only 1 producer and 1 consumer*

What if there are multiple producers or consumers?

- Multiple threads using and modifying in & out
- Particularly bad if a thread gets interrupted…

```
produce:
    ...
    // add it to buffer
    array[in] = item;     <- here
    in++;
    ...
```

```
consume:
    ...
    // remove item
    int item = array[out];     <- or here
    out++;
    ...
```

Mutex

- **Mutex:** implemented using a Binary semaphore that is initialized to 1
- *Provides **mut**ual **ex**clusion to the critical section of code*

```
produce:
   P(mutex_p);
   // add it to buffer
   array[in] = item;
   in++;
   V(mutex_p);
```

```
consume:
   P(mutex_c);
   // remove item
   int item = array[out];
   out++;
   V(mutex_c);
```

Intuition: effectively makes these 2 lines of code atomic.

Producer-Consumer Code, v3

```
Shared: Semaphores mutex_p , mutex_c , nEmpty , nFilled
Init: mutex_p  = 1;    /* for mutual exclusion */
      mutex_c  = 1;
      nEmpty = N;     /* # empty buffer entries */
      nFilled = 0;    /*    #  full buffer entries */
```

```
int array[N];
int in, out;
```

```
void produce (int item) {
   P(nEmpty);  // verify room for item
   P(mutex_p);
   // add item to buffer
   array[in] = item;
   in++;
   V(mutex_p);
   V(nFilled);  // "new item!"
}
```

```
int consume() {
   P(nFilled);  // verify item there
   P(mutex_c);
   // remove item
   int item = array[out];
   out++;
   V(mutex_c);
   V(nEmpty);  // "more room!"
   return item;
}
```

Busy Waiting considered Harmful

```
mutex = Semaphore(1)
...

for ever:
    P(mutex)
    if buffer is empty:
        V(mutex)
        continue
    get item from buffer
    V(mutex)
    process item
```

- This solution works, but it loops continuously until there is an item in the buffer
- This wasted valuable CPU cycles
- In this case, you need a semaphore for waiting and signaling
- You may also need a mutex semaphore for updating the buffer

Producer-Consumer Applications

- Applications:
  - Data from bar-code reader consumed by device driver
  - File data: computer → printer spooler → line printer device driver
  - Web server produces data consumed by client's web browser
- Example: "pipe" (|) in Unix

  > cat file | sort | uniq | more

  > prog | sort
- **Thought questions:**
  - where's the bounded buffer?
  - how "big" should the buffer be, in an ideal world?

Readers and Writers

- In this problem, threads share data that some threads “read” and other threads “write”
- **Goal:**
  - Allow:
    - multiple concurrent readers
    - only a single writer at a time
  - Constraint: if a writer is active, readers must wait

Readers-Writers Problem

- Courtois et al 1971
- Models access to a database
  - **Reader:** thread that looks at the database, but won't change it
  - **Writer:** thread that modifies the database
- **Example:** making an airline reservation
  - *When you browse* to look at flight schedules the web site is acting as a reader on your behalf
  - *When you reserve a seat*, the web site has to write into the database to make the reservation

Readers-Writers Problem

- N threads share 1 object in memory
  - Some write: **1** writer active at a time
  - Some read: **n** readers active simultaneously
- *Insight:* generalizes the critical section concept

- Need to clarify:
  - Writer is active & a combo of readers/writers show up: Who should get in next?
  - Writer is waiting & endless of stream of readers comes. Fair for them to become active?
- For now: back-and-forth turn-taking:
  - If a reader is waiting, *readers* get in next
  - If a writer is waiting, *one* writer gets in next

Readers-Writers

```
mutex = Semaphore(1)
wrl = Semaphore(1)
rcount = 0;

write() {
  wrl.P();
   . . .
   /*perform write */
   . . .
   wrl.V();
}
```

```
read () {
   mutex.P();
   rcount++;
   if (rcount == 1)
      wrl.P();
   mutex.V();
   . . .
   /* perform read */
   . . .
   mutex.P();
   rcount--;
   if (rcount == 0)
      wrl.V();
   mutex.V();
}
```

Readers-Writers Notes

- If there is a writer
  - First reader blocks on **wrl**
  - Other readers block on **mutex**
- Once a reader is active, all readers get to go through
  - Which reader gets in first?
- The last reader to exit signals a writer
  - If no writer, then readers can continue
- If readers and writers waiting on **wrl**, and writer exits
  - Who gets to go in first?
- Why doesn't a writer need to use **mutex**?

Does this work as we hoped?

- When readers active → no writer can enter
  - Writers wait @ P(wrl)
- When writer is active → nobody can enter
  - Any other reader or writer will wait (where?)
- Back-and-forth isn't so fair:
  - Any number of readers can enter in a row
  - Readers can "starve" writers
- A fair back-and-forth solution with semaphores is really tricky!
  - Try it! (don't spend too much time…)

Common programming errors

Process I

P(S)

CS

P(S)

Process j

V(S)

CS

V(S)

Process k

P(S)

CS

Typo: Process I stuck forever on 2nd P(S).

Every *other subsequent* process freezes up on 1st P(s).

Typo: Process J undermines mutual exclusion:

(1) by not checking for permission via P(S)

(2) "extra" V() operations → allows other processes into the CS inappropriately

Omission: Whoever next calls P() will freeze up. Confusing because that *other* process could be correct, but *it's* the one that hangs when you use a debugger to look at its state!

More common mistakes

- Conditional code that can change code flow in the critical section
- Usual causes: code updates (bug fixes, added functionality) by someone *other* than the original author of the code

```
P(S)
if(something or other)
   return;
CS
V(S)
```

Language Support for
Concurrency

Revisiting semaphores!

- Semaphores are "low-level" primitives
  - Small errors:
    - → Easily bring system to grinding halt
    - → Very difficult to debug

- Two usage models:
  - **Mutual exclusion:** the "real" abstraction is a critical section
  - **Communication:** threads use semaphores to communicate (*e.g.,* bounded buffer example)

- Simplification: Provide concurrency support in compiler
  → Enter **Monitors**

Monitors

- Hoare 1974
- Abstract Data Type for handling/defining shared resources
- Comprises:
  - Shared Private Data
    - The resource
    - Cannot be accessed from outside
  - Procedures that operate on the data
    - Gateway to the resource
    - Can only act on data local to the monitor
  - Synchronization primitives
    - Among threads that access the procedures

Monitor Semantics

- Monitors guarantee mutual exclusion
  - Only one thread can execute monitor procedure at any time
    - "in the monitor"

Structure of a Monitor

```
Monitor monitor_name
{
    // shared variable declarations

    procedure P1(. . . .) {
        . . . .
    }
    procedure P2(. . . .) {
        . . . .
    }
    .
    .
    procedure PN(. . . .) {
        . . . .
    }

    initialization_code(. . . .) {
        . . . .
    }
}
```

For example:

```
Monitor stack
{
    int top;
    void push(any_t *) {
        . . . .
    }

    any_t * pop() {
        . . . .
    }

    initialization_code() {
        . . . .
    }
}
```

Only one operation can execute at a time

Condition Variables

- Monitors can define *Condition Variables*:
  - `Condition x;`
  - Provides a mechanism to wait for events
    - Example events: resources available, any writers , …
- 3 operations on Condition Variables
  - `x.wait()`: release monitor lock, sleep until woken up (*or you wake up on your own*)
  - `x.signal()`: wake at least one process waiting on condition (if there is one)
    - No history associated with signal
  - `x.broadcast()`: wake **all** processes waiting on condition
    - Useful for resource manager

Using Condition Variables

- To wait for some condition:

  while not *some_predicate*():

  CV.wait()

  - this releases the monitor lock and allows another thread to enter
  - as CV.wait() returns, lock is automatically reacquired

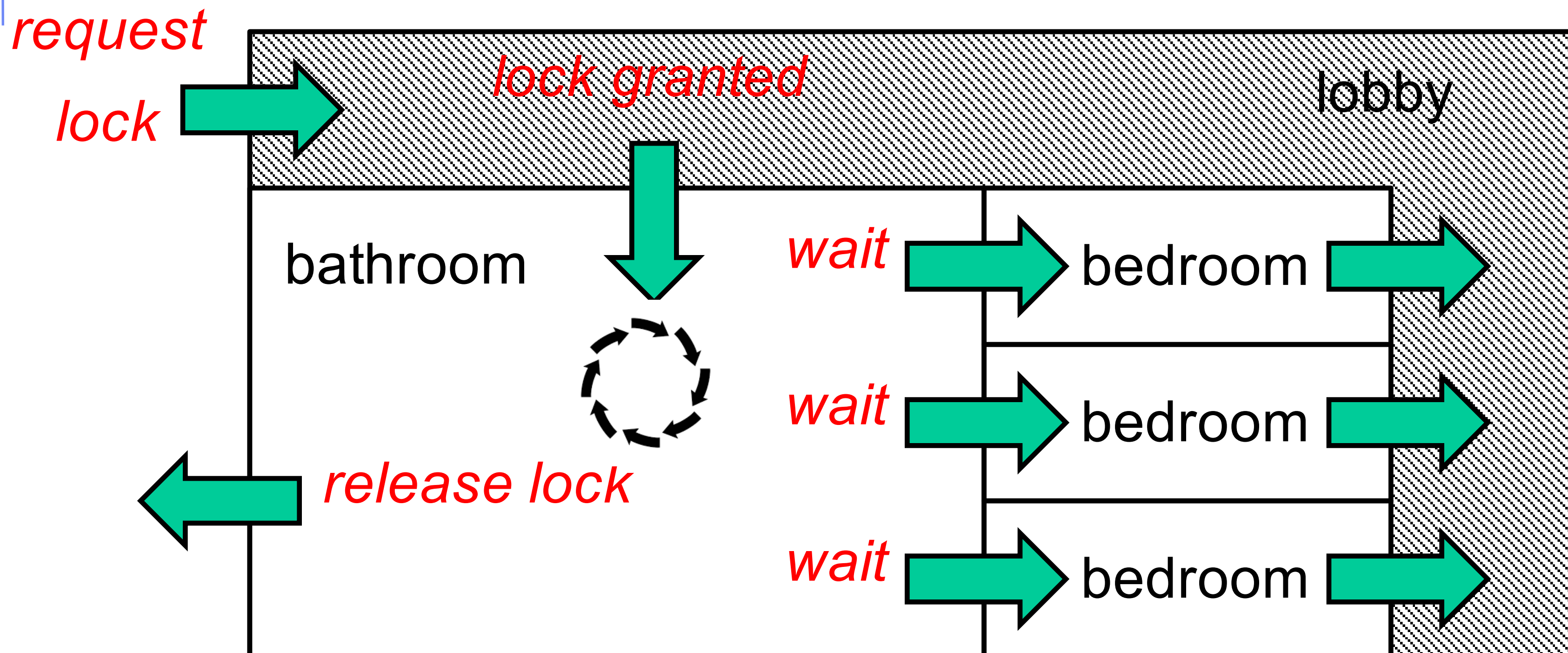
- When the condition becomes satisfied:

  CV.broadcast(): wakes up all threads

  or CV.signal(): wakes up at least one

Types of wait queues

- Monitors have two kinds of “wait” queues
  - Entry to the monitor (“the lobby”): has a queue of threads waiting to obtain mutual exclusion so they can enter
  - Condition variables (“the bedrooms”): each condition variable has a queue of threads waiting on the associated condition



Condition Variables ≠ Semaphores

- Access to monitor is controlled by a lock
  - Wait: blocks thread and gives up the monitor lock
    - To call wait, thread has to be in monitor, hence the lock
    - Semaphore P() blocks thread only if value less than 0
  - Signal: causes waiting thread to wake up
    - If there is no waiting thread, the signal is lost
    - V() increments value, so future threads need not wait on P()
    - Condition variables have no history!

- However they can be used to implement each other

Hoare vs. Mesa Semantics

- **Hoare Semantics:** monitor lock is transferred directly from the signaling thread to the newly woken up thread
  - But it is typically not desirable to force the signaling thread to relinquish the monitor lock immediately to a woken up thread
  - Confounds scheduling with synchronization, penalizes threads
- **Mesa Semantics:** Every real system simply puts a woken up thread on the monitor entry queue ("the lobby"), but does not immediately run that thread, or transfer the monitor lock

Language Support

- Can be embedded in programming language:
  - Synchronization code added by compiler, enforced at runtime
  - Mesa/Cedar from Xerox PARC
  - **Java: synchronized, wait, notify, notifyall**
  - **C#: lock, wait (with timeouts) , pulse, pulseall**
  - **Python: acquire, release, wait, notify, notifyAll**
- Monitors easier and safer than semaphores
  - Compiler can check
  - Lock acquire and release are implicit and cannot be forgotten

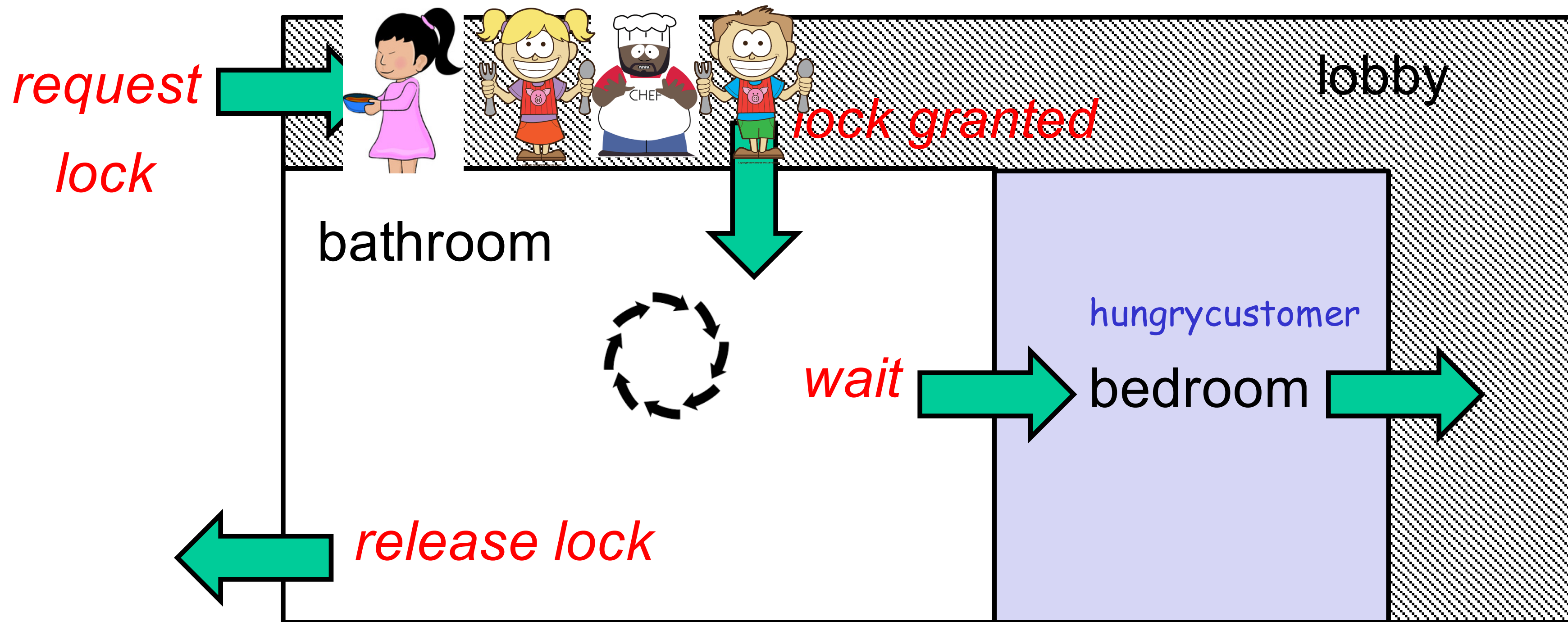
Monitor Solutions to Classical Problems

A Simple Monitor

```
Monitor EventTracker {
    int numburgers = 0;
    condition hungrycustomer;

    void customerenter() {
        while (numburgers == 0)
                hungrycustomer.wait()
        numburgers -= 1
    }
    void produceburger() {
        ++numburger;
        hungrycustomer.signal();
    }
}
```

- Because condition variables lack state, all state must be kept in the monitor
- The condition for which the threads are waiting is necessarily made explicit in the code
  - numburgers > 0
- Hoare vs. Mesa semantics
  - What happens if there are lots of customers?

```
int numburgers = 0;
condition hungrycustomer;

void customerenter() {
    while (numburgers == 0)
        hungrycustomer.wait()
    numburgers -= 1
}

void produceburger() {
    ++numburger;
    hungrycustomer.signal();
    printf();
}
```



Producer Consumer using Monitors

```
Monitor Producer_Consumer {
    char buf[SIZE];
    int n = 0, tail = 0, head = 0;
    condition not_empty, not_full;

    void produce(char ch) {
      while(n == SIZE)
            wait(not_full);
      buf[head%SIZE] = ch;
      head++;
      n++;
      notify(not_empty);
    }

    char consume() {
        while(n == 0)
            wait(not_empty);
        ch = buf[tail%SIZE];
        tail++;
        n--;
        notify(not_full);
        return ch;
    }
}
```

What if no thread is waiting when notify() called?

Then signal is a "no-op". Very different from calling V() on a semaphore – semaphores remember how many times V() was called!

Readers and Writers

```
Monitor ReadersNWriters
 int WaitingWriters = 0, WaitingReaders = 0, NReaders = 0, NWriters = 0;
 Condition CanRead, CanWrite;

 void BeginWrite()
        assert NReaders == 0 or NWriters == 0
        ++WaitingWriters;
        while NWriters > 0 or NReaders > 0
               CanWrite.wait()
        --WaitingWriters;
        NWriters = 1;

 void EndWrite()
        assert NWriters == 1 and NReaders == 0
        NWriters := 0;
        if WaitingWriters > 0
            CanWrite.signal();
        else if WaitingReaders > 0
            CanRead.broadcast();

 void BeginRead()
        assert NReaders == 0 or NWriters == 0;
        ++WaitingReaders;
        while NWriters > 0 or WaitingWriters > 0
               CanRead.wait();
        --WaitingReaders;
        ++NReaders;

 void EndRead()
        assert NReaders > 0 and NWriters == 0;
        --NReaders;
        if NReaders == 0 and WaitingWriters > 0
            CanWrite.signal();
```

Understanding the Solution

- A writer can enter if there is no other active writer and no readers are waiting
- A reader can enter if there is no active writer and no writers are waiting

Understanding the Solution

- When a writer finishes, it checks to see if any readers are waiting
  - If so, it lets all of them enter
  - If not, and there is a writer waiting, it lets one of them enter
- When the last reader finishes, it lets a writer in (if any is there)

Understanding the Solution

- It wants to be fair
  - If a writer is waiting, readers queue up
  - If a reader (or another writer) is active or waiting, writers queue up
  - … this is mostly fair, although once it lets a reader in, it lets ALL waiting readers in all at once, even if some showed up “after” other waiting writers

Subtle aspects?

- Condition variables force the actual conditions that a thread is waiting for to be made explicit in the code
  - The comparison preceding the “wait()” call concisely specifies what the thread is waiting for

- The fact that condition variables themselves have no state forces the monitor to explicitly keep the state that is important for synchronization
  - This is a good thing

Barbershop Problem

- One possible version:
  - A barbershop holds up to k clients
  - N barbers work on clients
  - M clients total want their hair cut
  - Each client will have their hair cut by the first barber available

Implementing the Barbershop

(1) Identify the waits

- Customers?
- Barbers?

(2) Create condition variables for each

(3) Create counters to trigger the waiting

(4) Create signals for the waits

Barrier Synchronization

- Important synchronization primitive in high-performance parallel programs
- nThreads threads divvy up work and run rounds of computations separated by *barriers*
- Implementing barriers is not easy. The solution to the right uses a “double-turnstile”.
- Can you see why a single “turnstile” would not work?

```
def barrier():
    assert nLeaving == 0 and nArrived < nThreads
    nArrived++
    if nArrived == nThreads:
        nLeaving = nThreads
        cond1.broadcast()
    else:
        while nArrived < nThreads:
            cond1.wait()

    assert nArrived == nThreads and nLeaving > 0
    nLeaving--
    if nLeaving == 0:
        nArrived = 0
        cond2.broadcast()
    else:
        while nLeaving > 0:
            cond2.wait()
```

Mapping to Real Languages

```
class RWlock:
  def __init__(self):
     self.lock = Lock()
     self.readCond = Condition(self.lock)
     self.writeCond = Condition(self.lock)
     self.nActiveReaders = 0
     self.nActiveWriters = 0
     self.nWaitingReaders = 0
     self.nWaitingWriters = 0
```

signal() == notify()

broadcast) == notifyAll()

```
def readAcquire(self):
    with self.lock:
       self.nWaitingReaders += 1
       while self.nWaitingWriters > 0 or self.nActiveWriters > 0:
          self.readCond.wait()
       self.nWaitingReaders -= 1
       self.nActiveReaders += 1

  def readRelease(self):
    with self.lock:
       self.nActiveReaders -= 1
       if self.nActiveReaders == 0 and self.nWaitingWriters > 0:
          self.writeCond.notify()
```

- Python monitors are simulated by explicitly allocating a lock and acquiring and releasing it (with the "with" statement) when necessary
  - More flexible than Hoare's approach

To conclude

- Race conditions are a pain!
- We studied several ways to handle them
  - Each has its own pros and cons
- Support in Python, Java, C# has simplified writing multithreaded applications
  - Java and C# support at most one condition variable per object, so are slightly more limited
- Some new program analysis tools automate checking to make sure your code is using synchronization correctly
  - The hard part for these is to figure out what “correct” means!