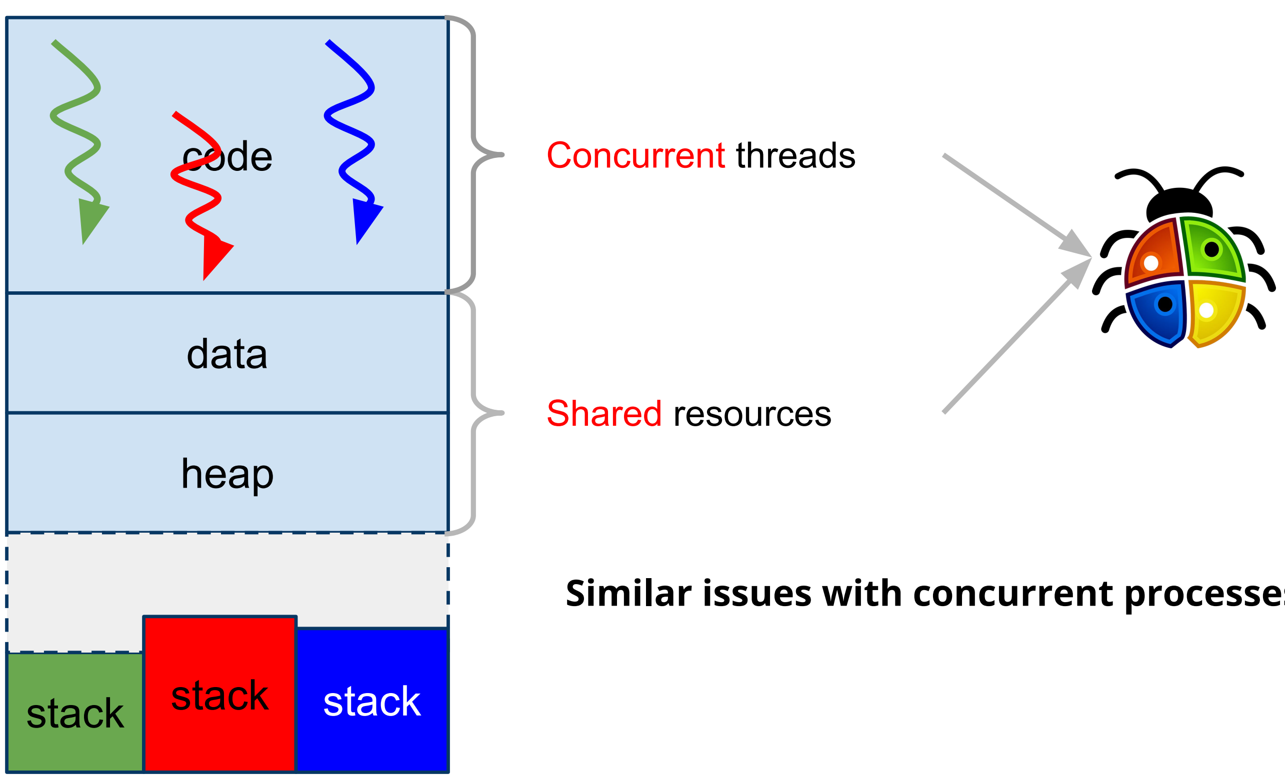
Process/Thread Synchronization



Similar issues with concurrent processes!!!

Interruptable Points ⟶ Race Condition

```
void husband() {
  // more code

  acct -= 20;

}
```

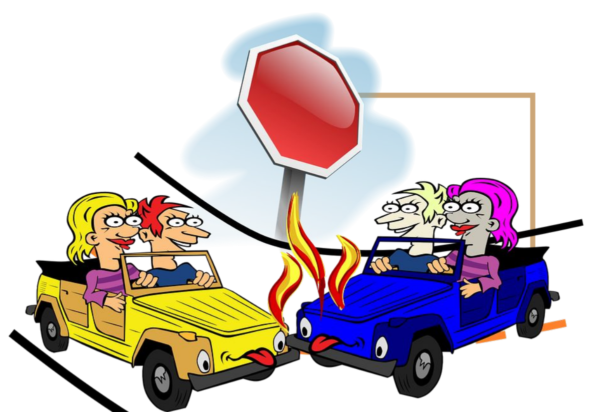
```
void wife() {
  // more code

  acct -= 500;

}
```

Context switching may cause issues

```
// in assembly
I.P.
  mov   %r3,acct
I.P.
  sub   %r3,20
I.P.
  mov   acct,%r3
I.P.
```

```
            // in assembly
                       I.P.
  mov   %r4,acct
                       I.P.
  sub   %r4,500
                       I.P.
  mov   acct,%r4
                       I.P.
```

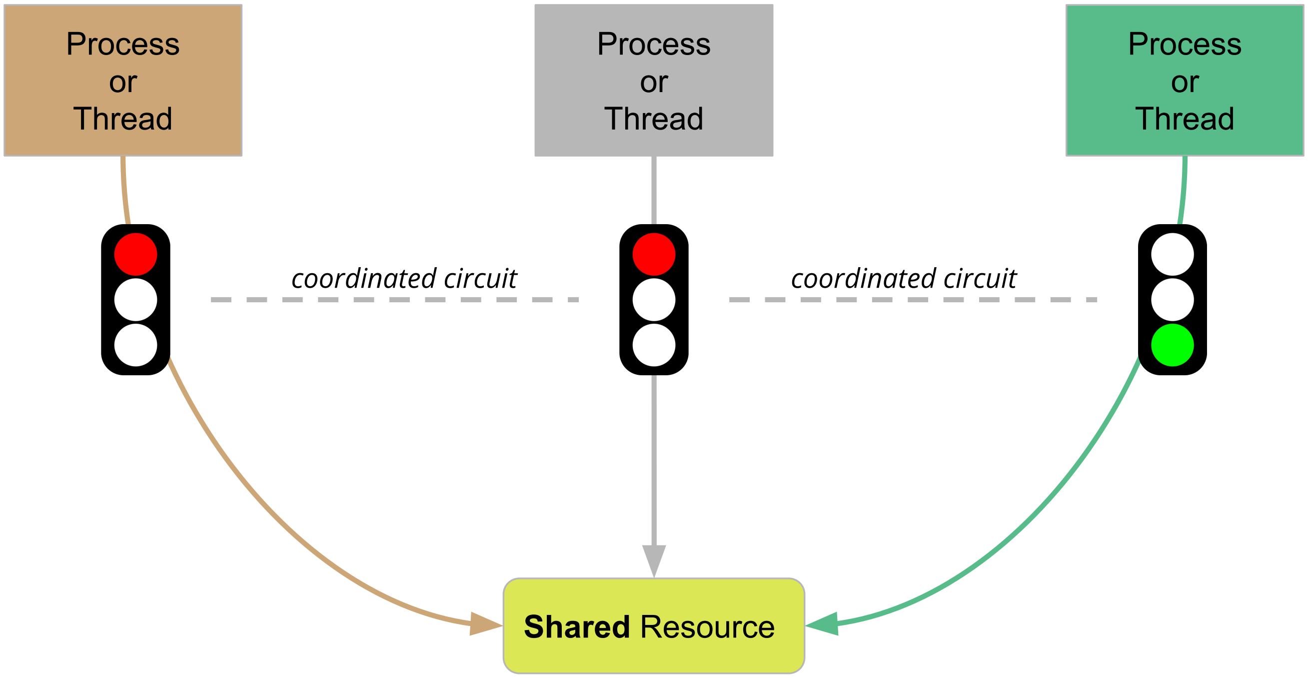

How to avoid “Race Condition”



Synchronization Mechanism

Concurrent Walk by N people
Sharing Common Floor Space

Concurrent Process Synchronization Model



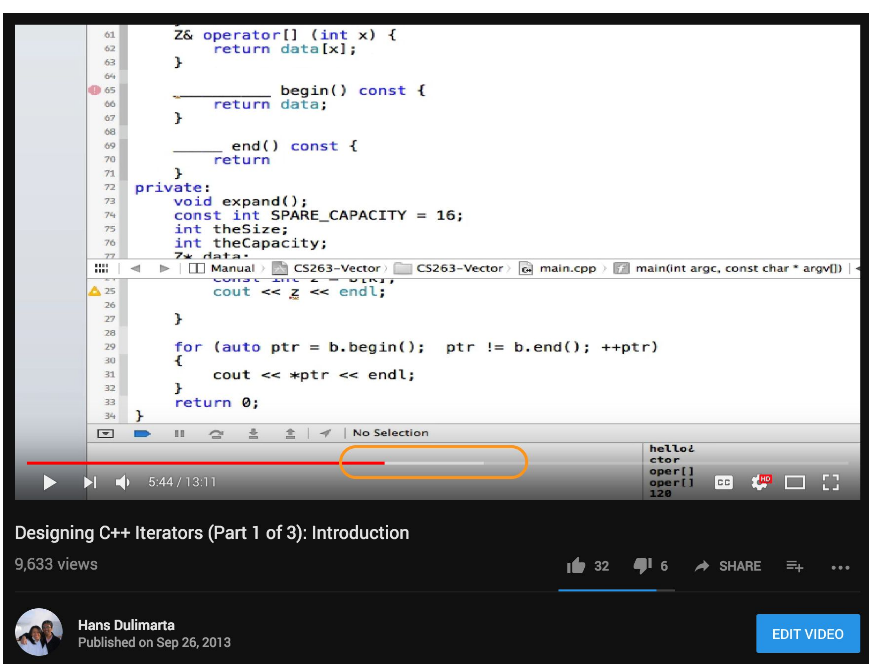
Goal: develop synchronization mechanism

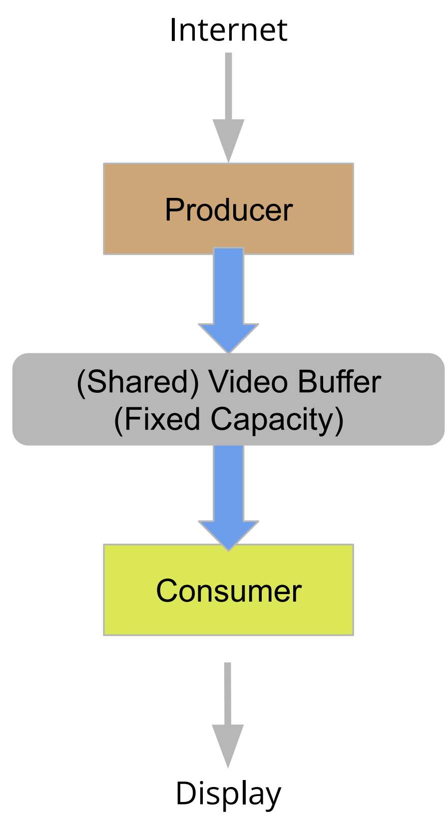
- Implement the *coordinated* “traffic light” paradigm (“STOP” and “GO”)
- Use software solution
  - Design “STOP” and “GO” mechanisms using *ordinary program variables* (integer counters, boolean flags, etc.) entirely in user space (**without OS assistance**)
  - Design them **with OS assistance**
- Use hardware solution
- Combination and software and hardware solution

Producer - Consumer Problem

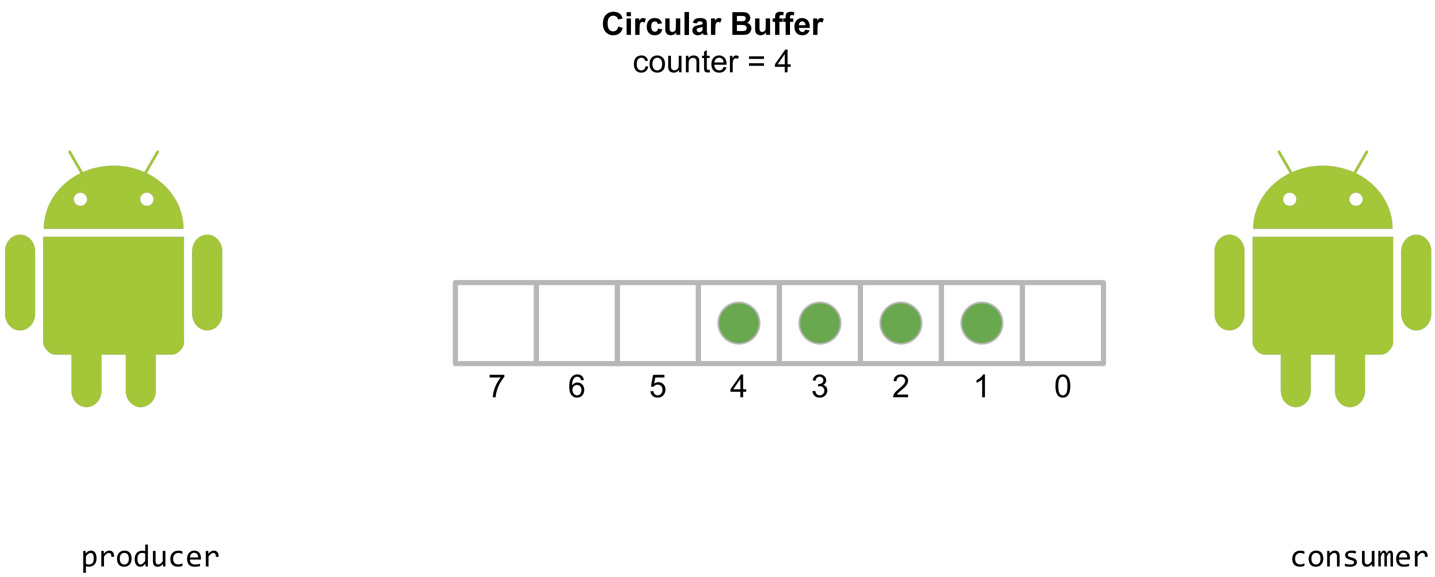
Two Processes & One Shared Buffer

Real Example: Video Streaming

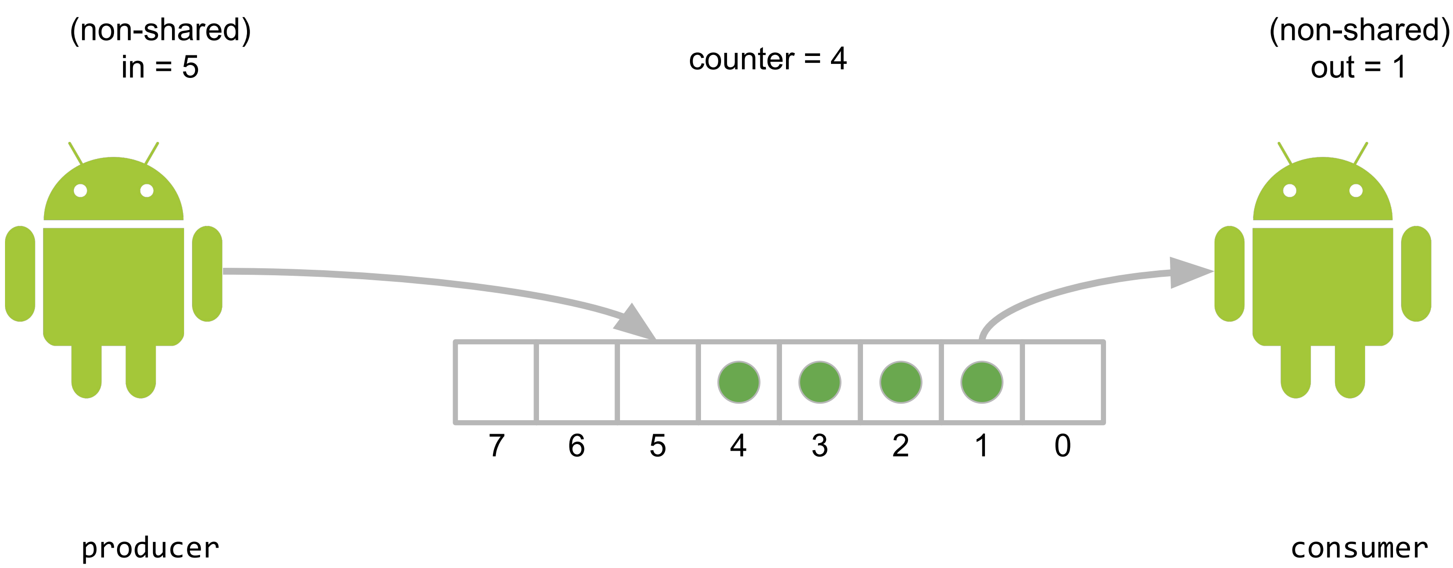




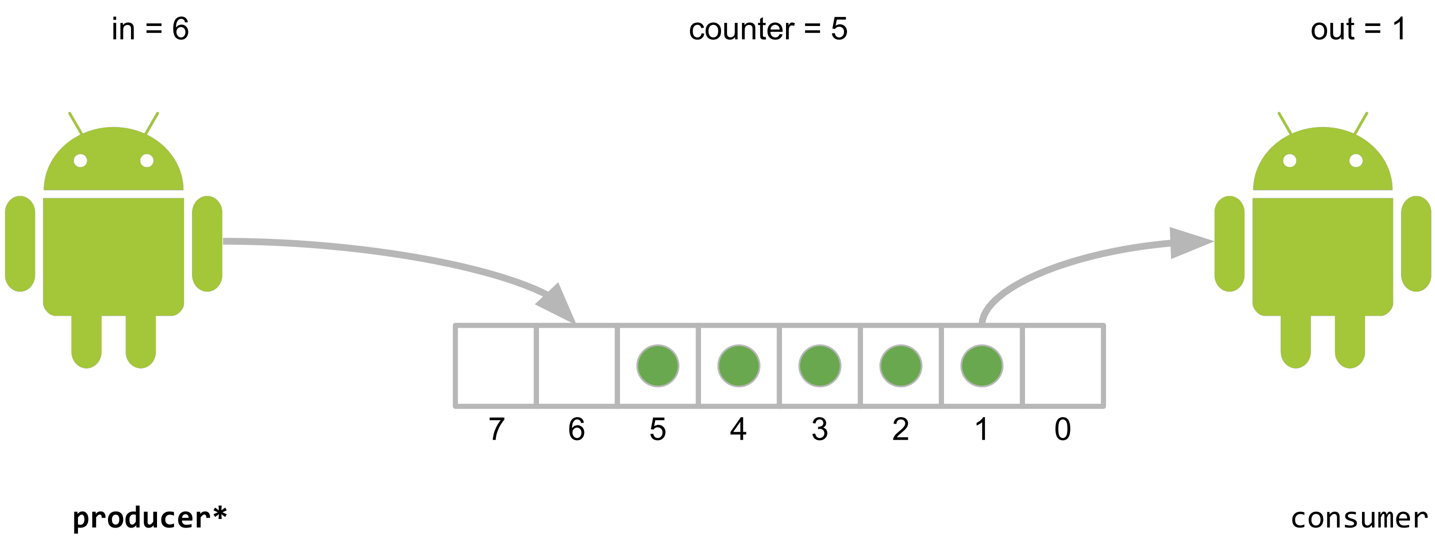
Producer/Consumer: **shared buffer & counter**



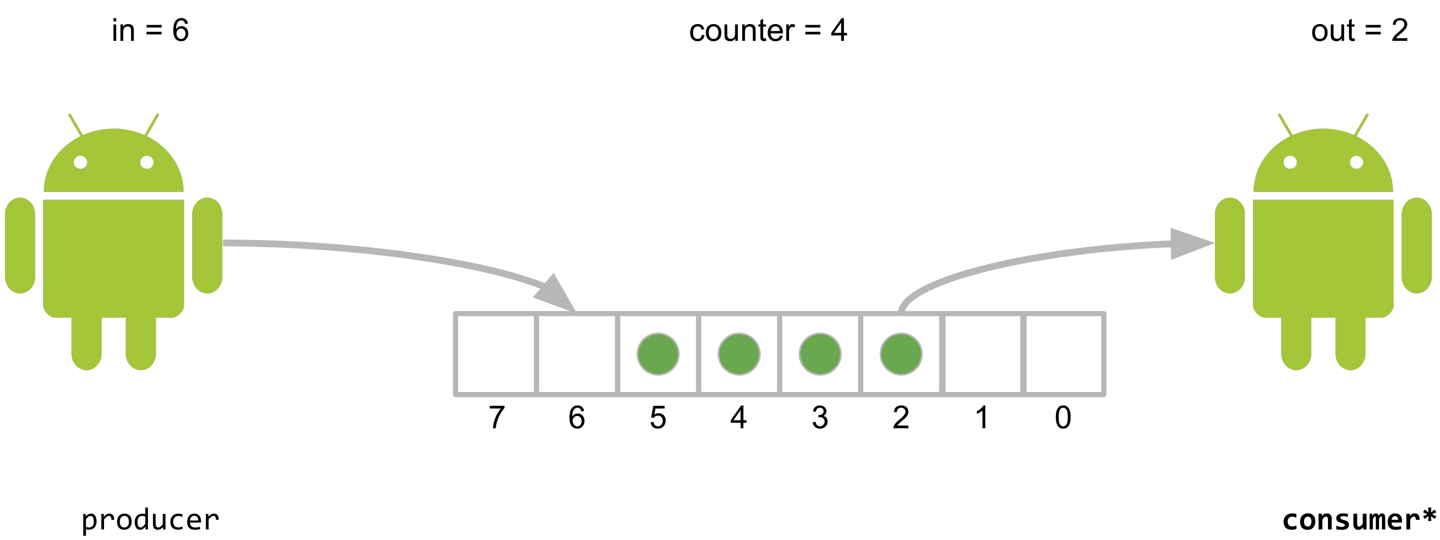
Producer/Consumer: **shared buffer & counter**



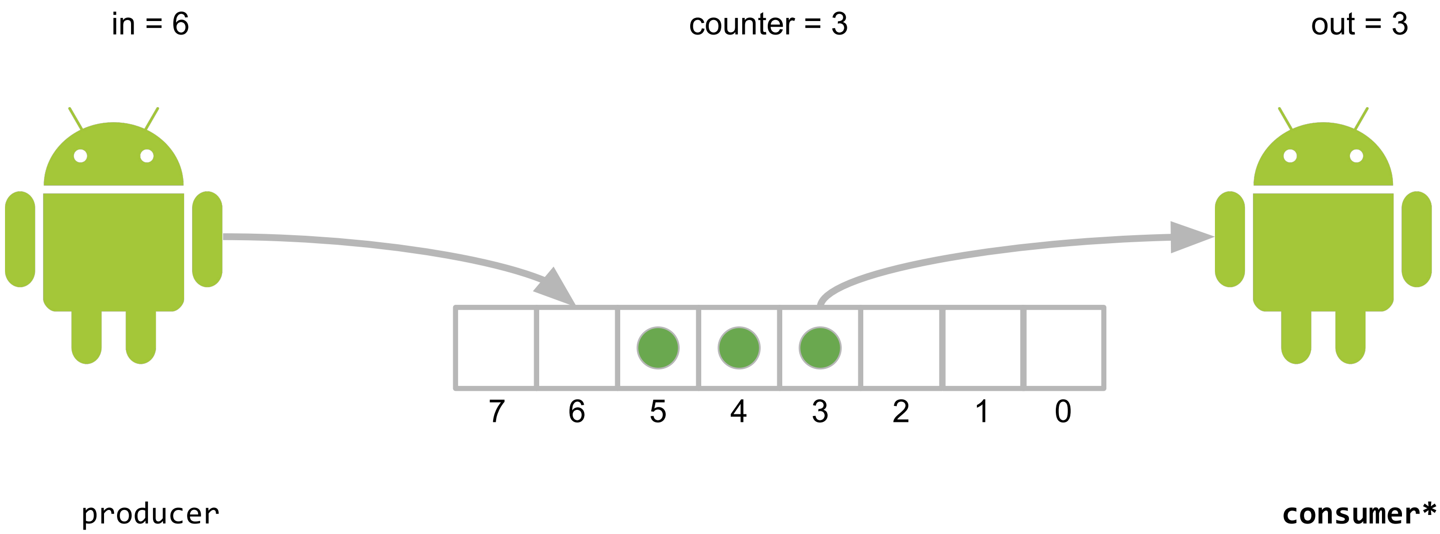
Producer/Consumer: **shared buffer & counter**



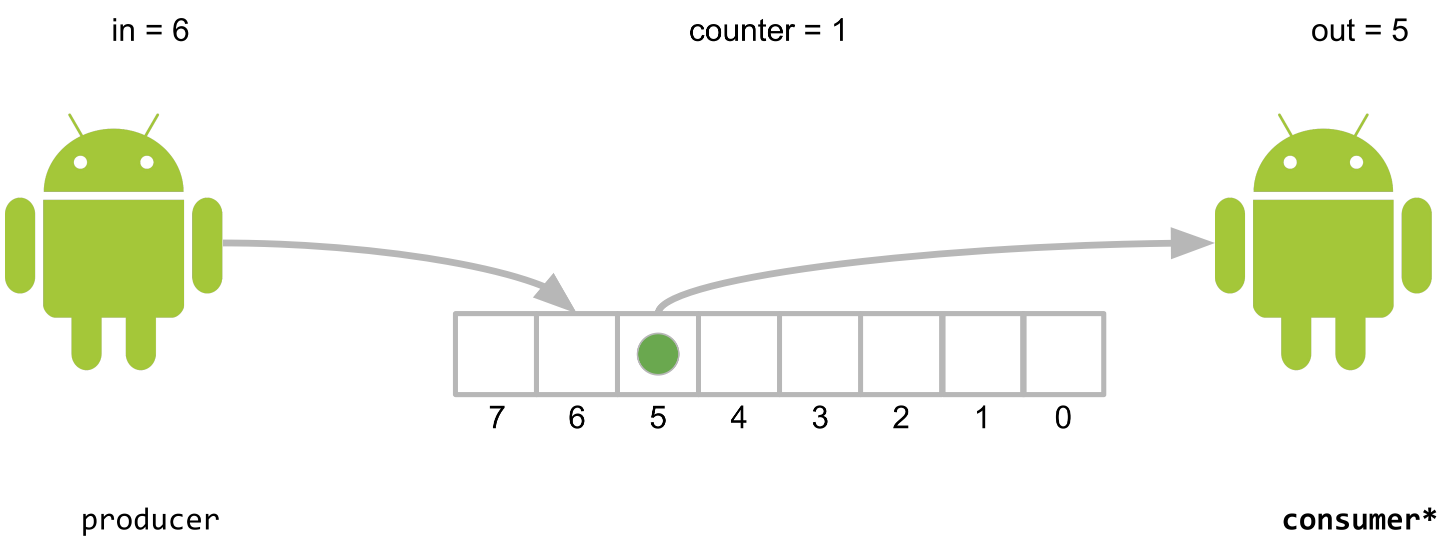
Producer/Consumer: **shared buffer & counter**



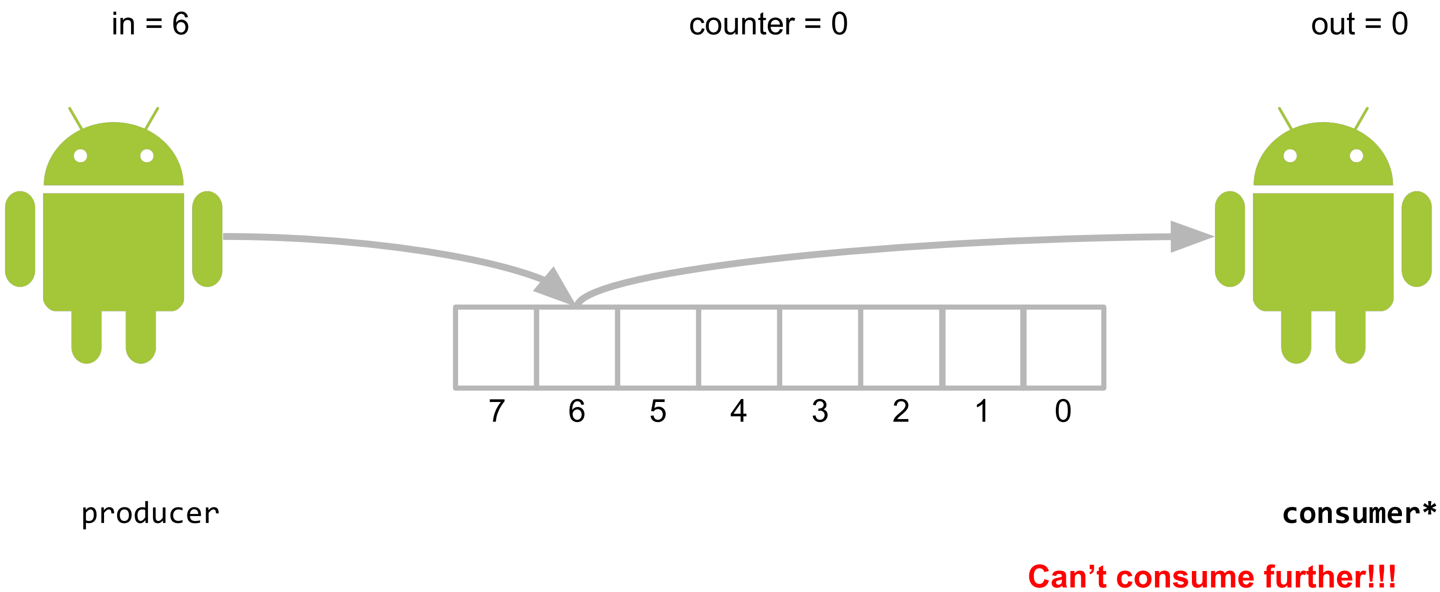
Producer/Consumer: **shared buffer & counter**



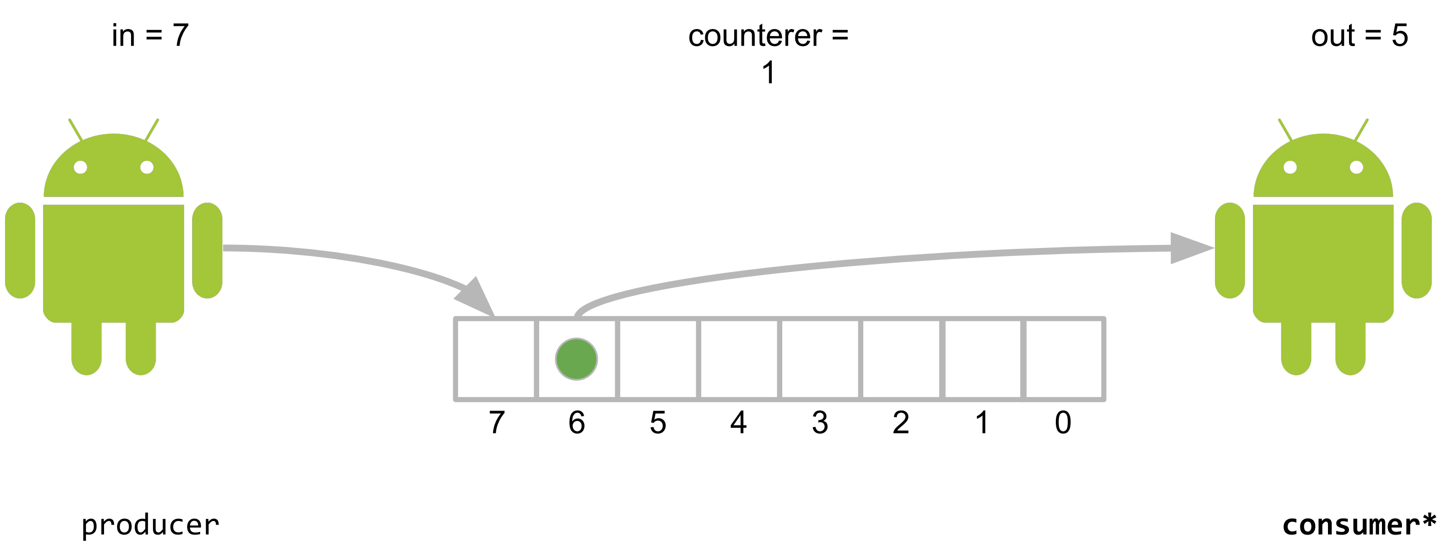
Producer/Consumer: **shared buffer & counter**



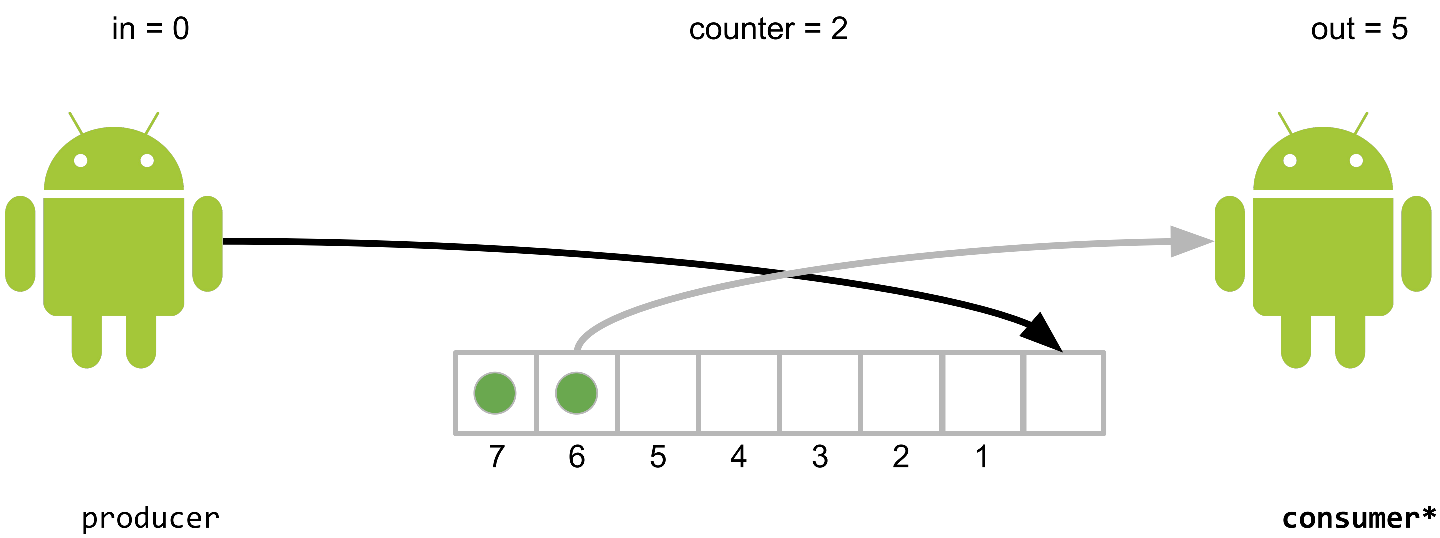
Producer/Consumer: **shared buffer & counter**



Producer/Consumer: **shared buffer & counter**

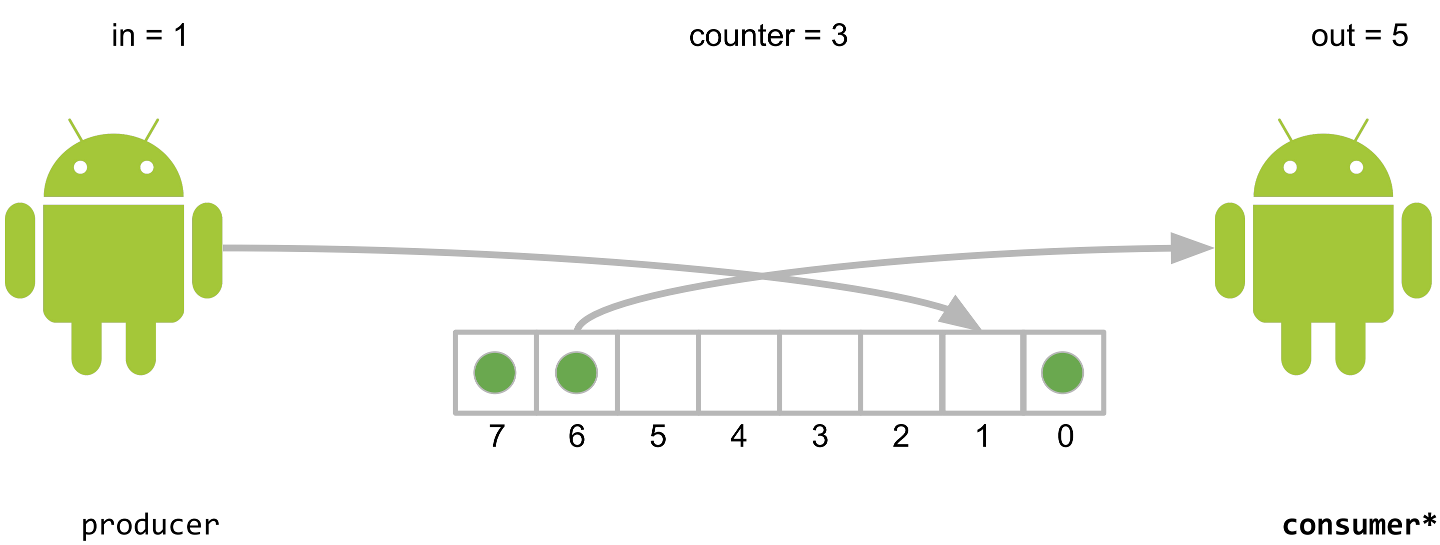


Producer/Consumer: **shared buffer & counter**

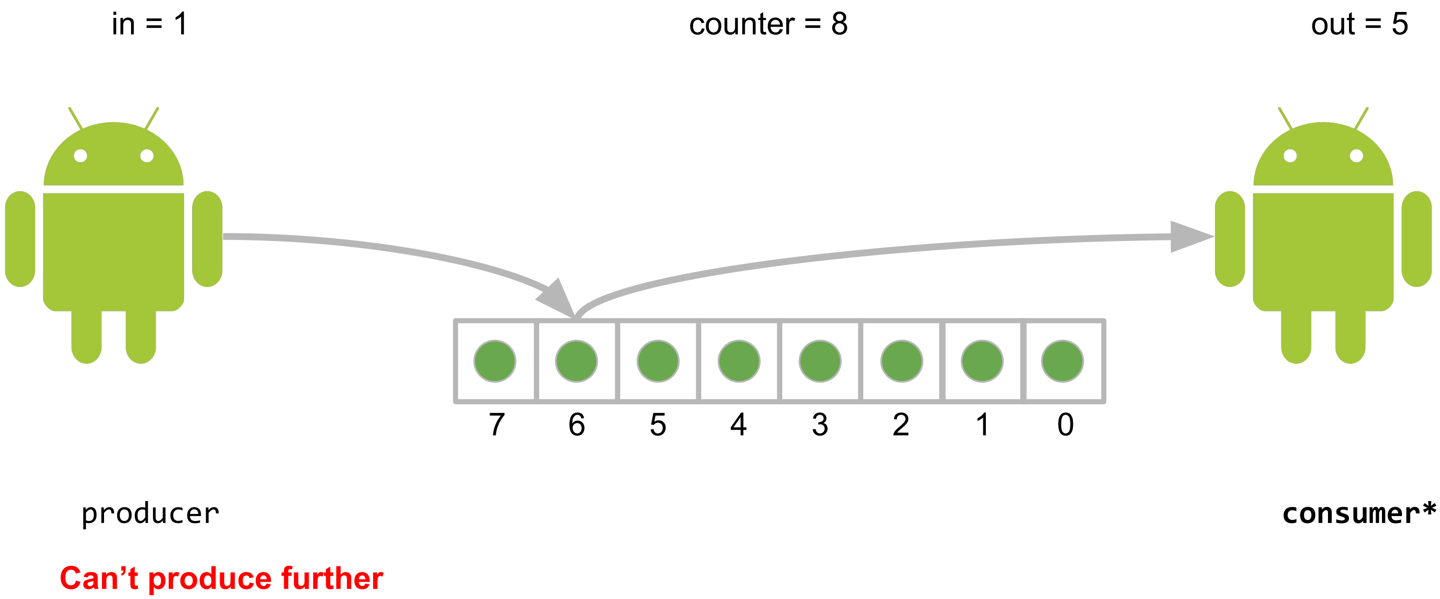


the buffer is circular

Producer/Consumer: **shared buffer & counter**



Producer/Consumer: **shared buffer & counter**



Producer/Consumer with Bounded Buffer

- Consumer has to wait/**block when the buffer is empty**
- Producer has to wait/**block when the buffer is full**
- Can't assume strict alternation
- Can't assume relative speed between producer/consumer

***For now**: use **busy wait** to block a process/thread*

```
while (some_condition) {
  // do nothing
}
```

```
// without curly brackets
// put a semicolon
while (some_condition) ;
```

Group Exercise: Write Producer/Consumer Code

Shared variables

```
int counter;      /* item count */
Item buff[N];     /* buffer for storing items */
```

```
/* producer */

while (true) {
  p_item = produce_item();
  // put p_item into buffer




}
```

```
/* consumer */

while (true) {




  // get c_item from buffer
  consume_item(c_item);
}
```

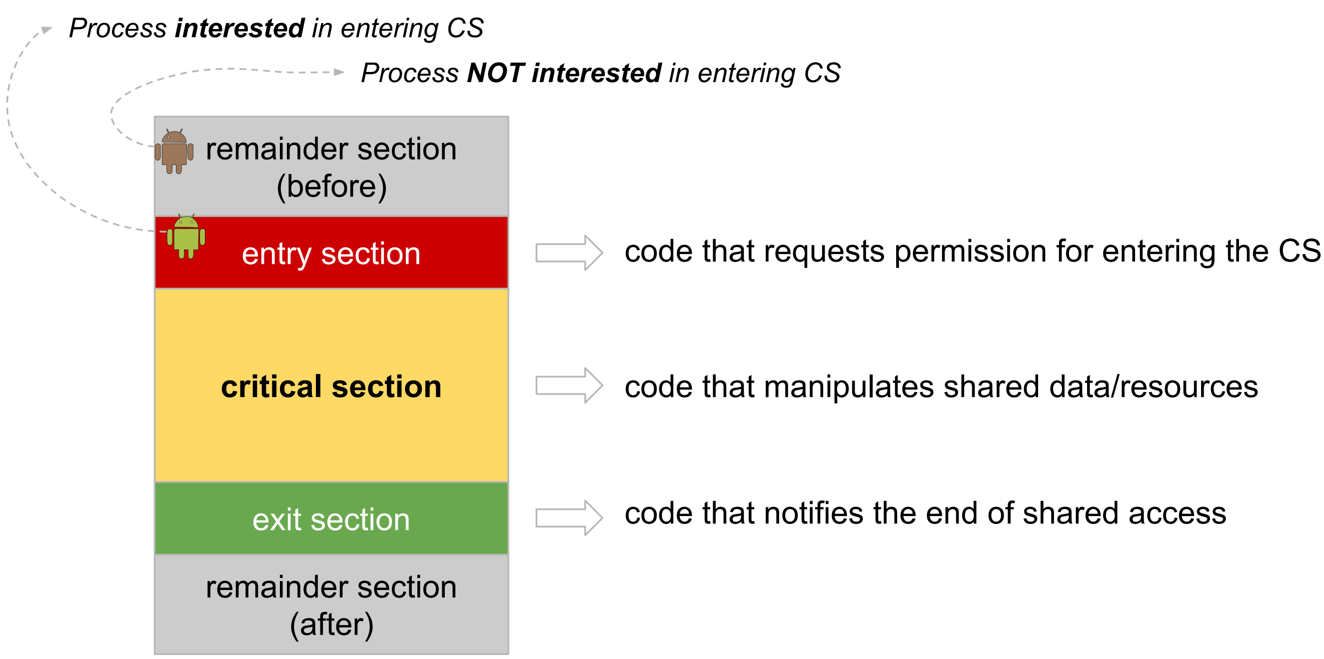

Producer/Consumer (*almost* a solution)

```
/* producer */
in = 0;
while (true) {
  p_item = produce_item();

  while (counter == BUFF_SIZE)
    /* do nothing */;
  buff[in] = p_item;
  counter++;
  in++;
  in %= BUFF_SIZE;
}
```

```
/* consumer */
out = 0;
while (true) {


  while (counter == 0)
    /* do nothing */;
  c_item = buff[out];
  counter--;
  out++;
  out %= BUFF_SIZE;
  consume_item (c_item);
}
```

Critical Section: Model & Formalism

Model for Shared Access



Producer/Consumer Code

```
/* producer */
in = 0;
while (true) {
  p_item = produce_item();

  while (counter == BUFF_SIZE)
    /* do nothing */;

  buff[in] = p_item;

  counter++;

  in++;
  in %= BUFF_SIZE;
}
```

remainder section (before)
entry section
critical section
exit section
remainder section (after)

```
/* consumer */
out = -1;
while (true) {

  while (counter == 0)
    /* do nothing */;

  c_item = buff[out];

  counter--;

  out++;
  out %= BUFF_SIZE;
  consume_item (c_item);

}
```


Requirements for Solution to CS Problems

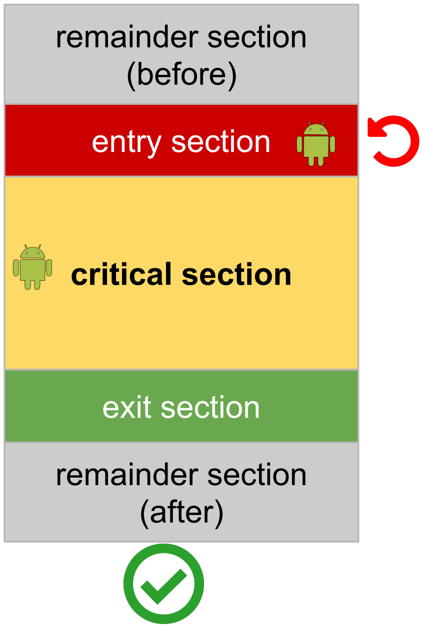
A good solution must guarantee

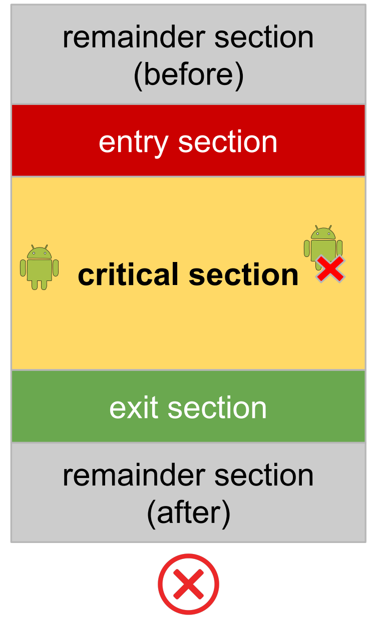
- Mutual Exclusion (**only one** may gain entry)
- Progress case I (gain entry **without** other contenders present)
- Progress case II (gain entry **with** other contenders present)
  - Neither Deadlock, Nor Livelock
- No Starvation/Bounded Waiting (no indefinite **re-entry**)

Limit Entry – Make Progress – Fair Progress

Mutual Exclusion (ME)

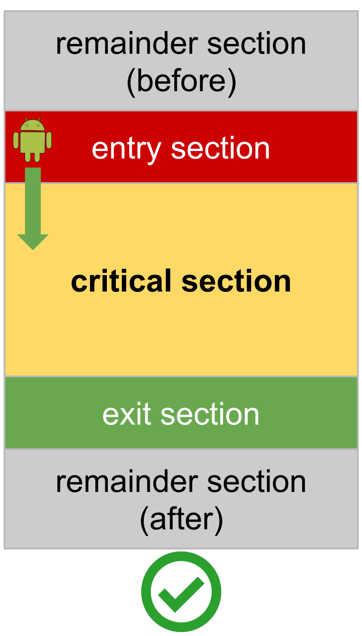
*At most one process should be allowed to be in its **critical section***

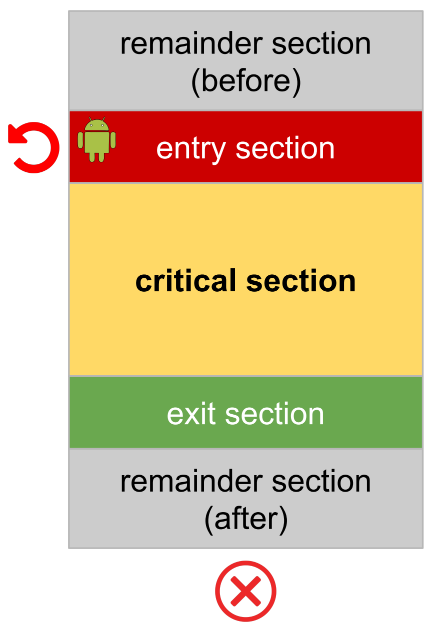




Progress: Case I (PC1)

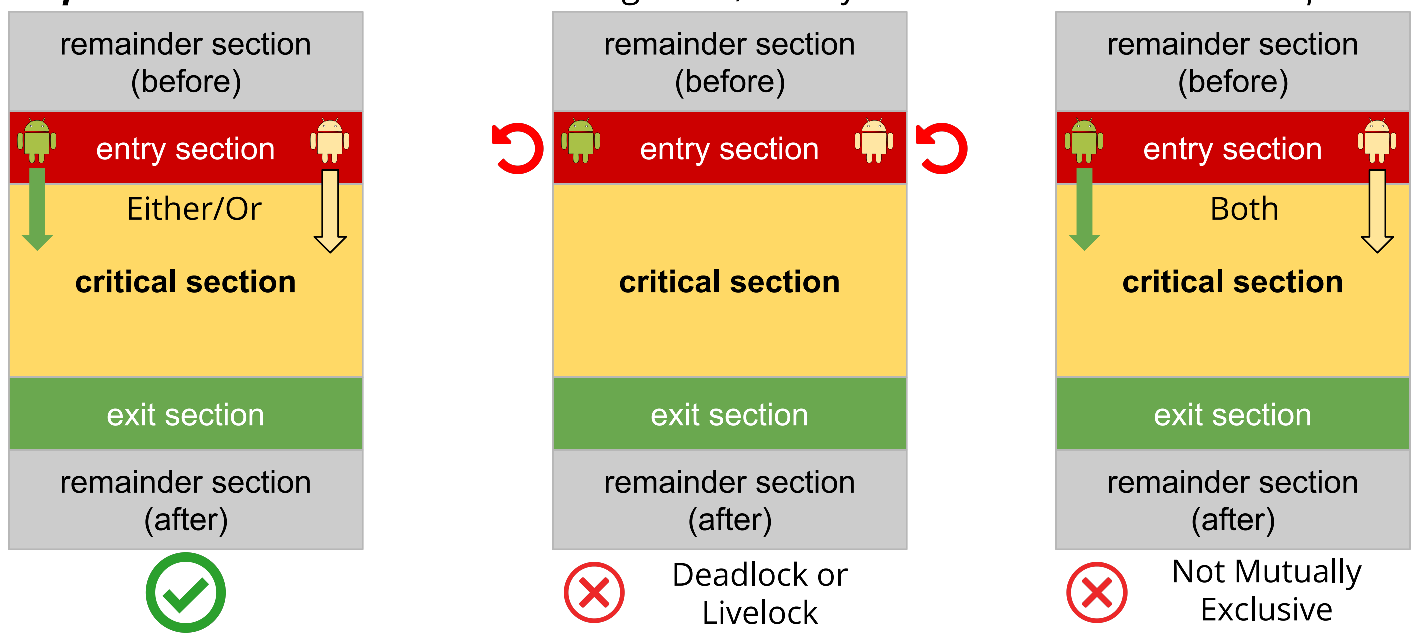
*If only **one process** is interested in entering its CS, that process should be allowed to proceed*





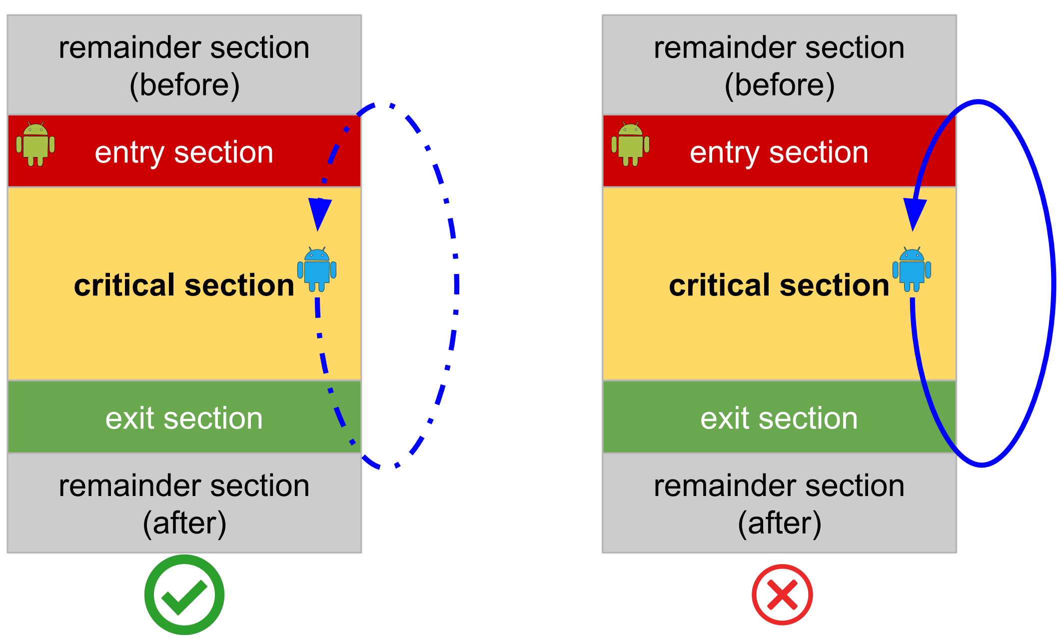
Progress: Case II (PC2)

*If **two processes** are interested in entering its CS, one of them should be allowed to proceed*



Bounded Waiting (BW)

A process should not be allowed to <u>reenter indefinitely</u> starving others



Prove (Direct or By Contradiction) Disprove & Counter Example(s)

Prove or Disprove?

- *First **try hard** to break the code by considering all possible cases of context switching, i.e. find a counterexample to disprove*
- *In the process (of trying to break the code, **but you can't find one**), you usually find an insight how to prove the correctness*

General Approach of Proving/Disproving

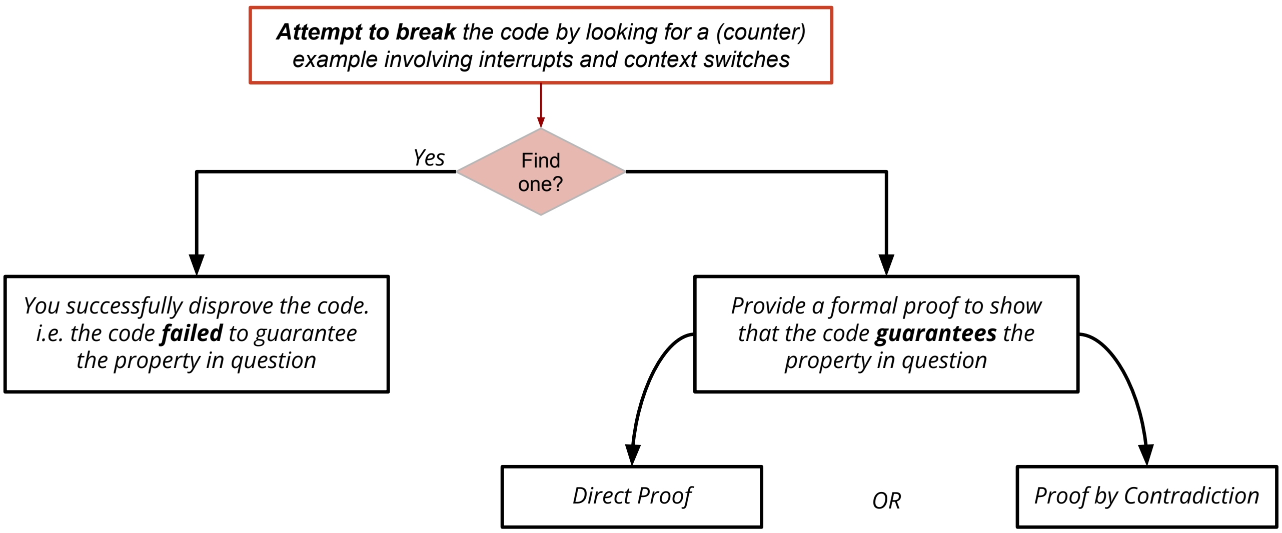
- Approach your code analysis as if your are *debugging* a program
  - Place breakpoints
  - Inspect all variables
  - Analyze what can/will happen to the process(es) based on the values of their variables
  - Incorporate context switching
- Breakpoint locations (refer to the illustrations in previous pages)
  - ME: **freeze** one process inside its Critical Section, **freeze** the other in its Entry Section
  - PC1: **freeze** one process inside its Entry Section
  - PC2: **freeze** both processes inside their respective Entry Section
  - BW: **freeze** one process inside its Entry Section, place (**don't freeze**) the other inside its Critical Section and move it through the rest of the code and reenter

Disproving (Showing that Code is Poorly Design)

- Disproving XYZ means showing that a code does not guarantee XYZ
  - Disproving ME means showing that a code does not guarantee Mutual Exclusion
- Disproving progress case I is generally easier (it involves only ONE process)
- General approach (for disproving mutual exclusion, progress case II, and bounding waiting)
  - look for **ONE context switching scenario** that would fail the code

Proof Guidelines

- First, try to break the code (**disprove**) for the property in question (*mutual exclusion*, *progress I*, *progress II*, or *bounded waiting*) by looking for a scenario of (multiple) interruptible points and (multiple) context switching
- Next (after unable to break the code), come up with a formal proof
  - (either) Direct proof technique
  - (or) Proof by contradiction
  - In both routes of proof: analyze the value of all the variables (as if you are **debugging** the code)



Proof by Contradiction

- Begin by **claiming the opposite** of the statement you attempt to prove
  - To proof "**the earth is round**" you begin by claiming "supposed **the earth is NOT round**"
- Analyze all the logical consequences from the supposition.
  - In code: analyze the value of all the variables when the supposition is true
- Look for a contradiction among all the logical consequences
  - In code: the variables may show impossible/contradicting values.
    - One logical consequence requires a particular variable to have value X
    - Another logical consequence requires that variable **at the same time** to have value Y

Summary of Proving CS Solution

- Prove (or disprove) Mutual Exclusion
- Prove (or disprove) Progress Case I: only one process is interested
- Prove (or disprove) Progress Case II: both processes are interested
  - Disprove of progress may also lead to demonstration of deadlock
- Prove (or disprove) Bounded Waiting: one process is blocked in its entry section, the other process is inside its CS and finishes, loops back and attempts to re-enter

Disproving Mutual Exclusion

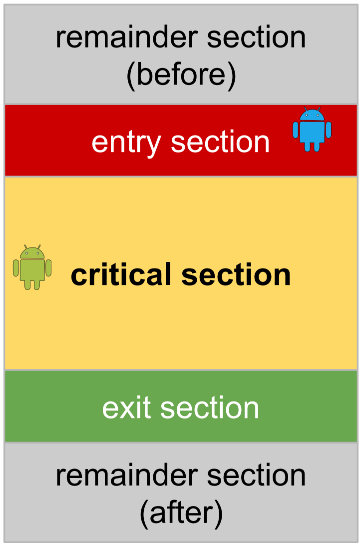
Initial setup

- *Place both processes in their respective entry section*

Goal

- *Find a context switching scenario that will allow both processes in their CS at the same time*

Proving Mutual Exclusion (Direct Proof)



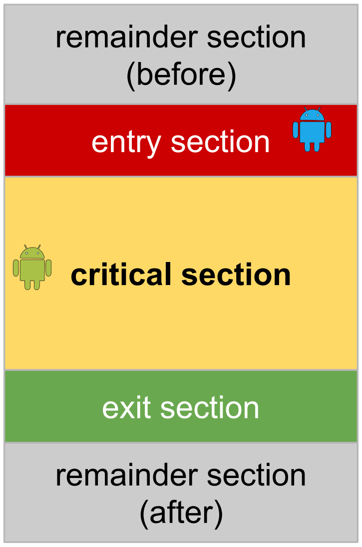
Initial setup

- *Green inside its critical section, inspect its vars*
- *Blue in its entry section, inspect its vars*

Goal

- *Analyze the variable values to show / prove that Blue will (busy) wait*

Proving Mutual Exclusion (By Contradiction)



*Initial setup (**assume** mutual exclusion is not guaranteed)*

- *Place both Green and Blue inside their respective critical section*
- *Inspect the variables from each process perspective*

Goal

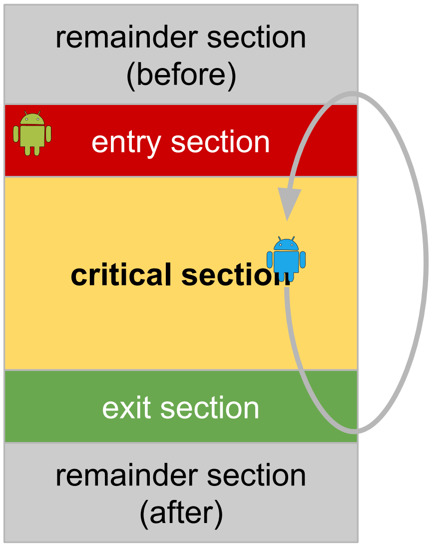
- *Find at least one contradicting fact*

Bounded Waiting

Bounded Waiting: there exists a bound, or limit, on the **number of times** that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- if a process is (waiting) inside its **entry section**, the must be a limit on the **number of times** other processes are allowed to reenter their critical section

Disproving Bounded Waiting



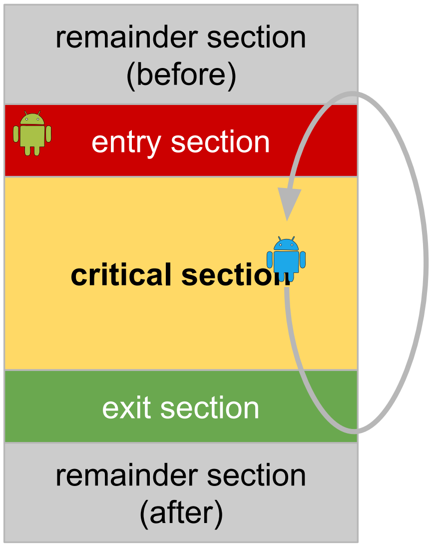
Initial Setup

- *Green wants to enter*
- *Blue is in critical section and <u>repeatedly</u> attempts to (re)enter.*

Goal

- *Find ONE context switching scenario which would allow Blue to re-enter its critical section indefinite number of times*

Proving Bounded Waiting (Direct Proof)



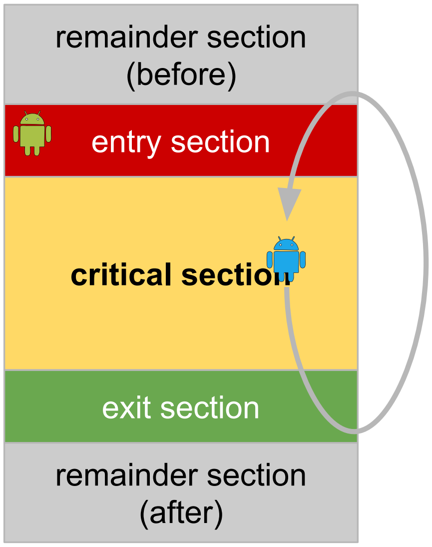
Initial Setup

- *Green wants to enter*
- *Blue is in critical section and <u>repeatedly</u> attempts to (re)enter.*
- *Inspect the variables from each process perspective*

Goal

- *Use the values of these variables to show that Blue will not be allowed to re-enter indefinitely*

Proving Bounded Waiting (by Contradiction)



Initial Setup (assume NO bounded waiting)

- *Green wants to enter*
- *Blue is in critical section and is able to to (re)enter indefinite number of times*
- *Inspect the variables from each process perspective*

Goal

- *Find a contradicting fact based on the value(s) of these variables*

Software Solution: User Space (using program variables)

Analyze Proposed Solutions

(class handout)

Dekker's Solution (for two processes) [1963]

Code for P0

```
while (true) {
    remainder section (before)

    flag[0] = true;
    while (flag[1]) {
      if (turn == 1) {
        flag[0] = false;
        while (turn == 1);
        flag[0] = true
      }
    }

    critical section

    turn = 1;
    flag[0] = false;

    remainder section (after)
}
```

Code for P1

```
while (true) {
    remainder section (before)

    flag[1] = true;
    while (flag[0]) {
      if (turn == 0) {
        flag[1] = false;
        while (turn == 0);
        flag[1] = true
      }
    }

    critical section

    turn = 0;
    flag[1] = false;

    remainder section (after)
}
```

Shared vars: `int turn = 0, bool flag[2] = {false, false}`

Peterson's Solution (for **two** processes) [1981]

Code for P0

```
while (true) {
    remainder section (before)
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
    critical section
    flag[0] = false;
    remainder section (after)
}
```

Code for P1

```
while (true) {
    remainder section (before)
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
    critical section
    flag[1] = false;
    remainder section (after)
}
```

Shared vars: `int turn = 0, bool flag[2] = {false, false}`

Hardware Solution

Hardware Solution

- Entry sections/exit sections typically requires **a sequence of machine instructions** that allow *interruptible points* in between
- Hardware Solutions
  - Disable Interrupt?
  - Implement the entry/exit section using **ONLY ONE** machine instruction
    - TestAndSet: return the *old value* (of a variable) and set it to a *new value*
    - CmpAndSwap: return the *old value* (of a variable) and **conditionally** set it to a *new value*

TestAndSet/TS[L] and CmpAndSwap/CAS

- The "C" functions below describe only the semantic of the TSL and CAS **assembly instructions**
- TSL: update a "lock" and set the CPU status register using the old value of the lock
- CAS: similar to TSL, but update the lock only if a condition is met

```
bool test_n_set (bool *lok)
{
  bool old = *lok;


  *lok = true;
  return old;
}
```

```
int cmp_n_swap (int *lok, int expected, int new_val)
{
  int old = *lok;

  if (*lok == expected)
     *lok = new_val;
  return old;
}
```

Quick Review of Loops in Assembly (Compiler Explorer)

Assembly instructions: Test and Set

lock is 0 when the "room" is NOT locked

```
# Entry code
spin: ts   lock   # copy old value of lock to accumulator, then set lock to 1
      jnz  spin   # if accumulator WAS NOT zero, try again


# Exit code
      sub  lock, lock
```

Intel x86

```
# Entry code
spin: bts  lock,0   # copy bit-0 of lock to Carry Flag before setting the bit
      jc   spin     # if lock WAS non-zero, try again

# Exit code
      sub  lock,lock
```

Assembly instructions: Compare & Swap

```
# Entry Code
Spin: cas  lock,0,1  # if lock == 0 set lock to 1, evaluate its old content
      jnz  spin

# Exit code
      sub  lock,lock
```

Intel x86

```
# Entry Code
      mov  edx,1
spin: mov  eax,lock
      test eax,eax
      jnz  spin           # if lock is not zero, try again
      cmpxchg lock,edx    # IF lock == eax, set lock to edx ELSE eax = edx
      test eax,eax
      jnz  spin

# Exit code
    sub  lock,lock
```


Spinlock with TS

```
while (true) {
    remainder section (before)
    again:
            ts  lock
            jnz again
    critical section
            sub lock,lock
    remainder section (after)
}
```

Spinlock with CAS

```
while (true) {
    remainder section (before)
    again:
            cas lock,0,1
            jnz again
    critical section
    of process B
            sub lock,lock
    remainder section (after)
}
```

lock is a shared variable

Generalized “Peterson’s Solution” (for N tasks)

```
while (true) {
    remainder section (before)
    waiting[i] = true;
    test = true;
    while (waiting[i] && test)
      test = test_and_set(&lock);
    waiting[i] = false;

    critical section of Pi

    p = (i + 1) mod N
    while (p != i && !waiting[p])
      p++;   // mod N
    if (p == i)
      lock = false;
    else
      waiting[p] = false;
    remainder section (after)
}
```

Code for Pi

```
while (true) {
    remainder section (before)
    waiting[j] = true;
    test = true;
    while (waiting[j] && test)
      test = test_and_set(&lock);
    waiting[j] = false;

    critical section of Pj

    p = (j + 1) mod N
    while (p != j && !waiting[p])
      p++;   // mod N
    if (p == j)
      lock = false;
    else
      waiting[p] = false;
    remainder section (after)
}
```

Code for Pj

`waiting[]` and `lock` are **shared** (initialized to false). Other vars local