Concurrency and Process Synchronization: Synchronizing Access to Shared Objects

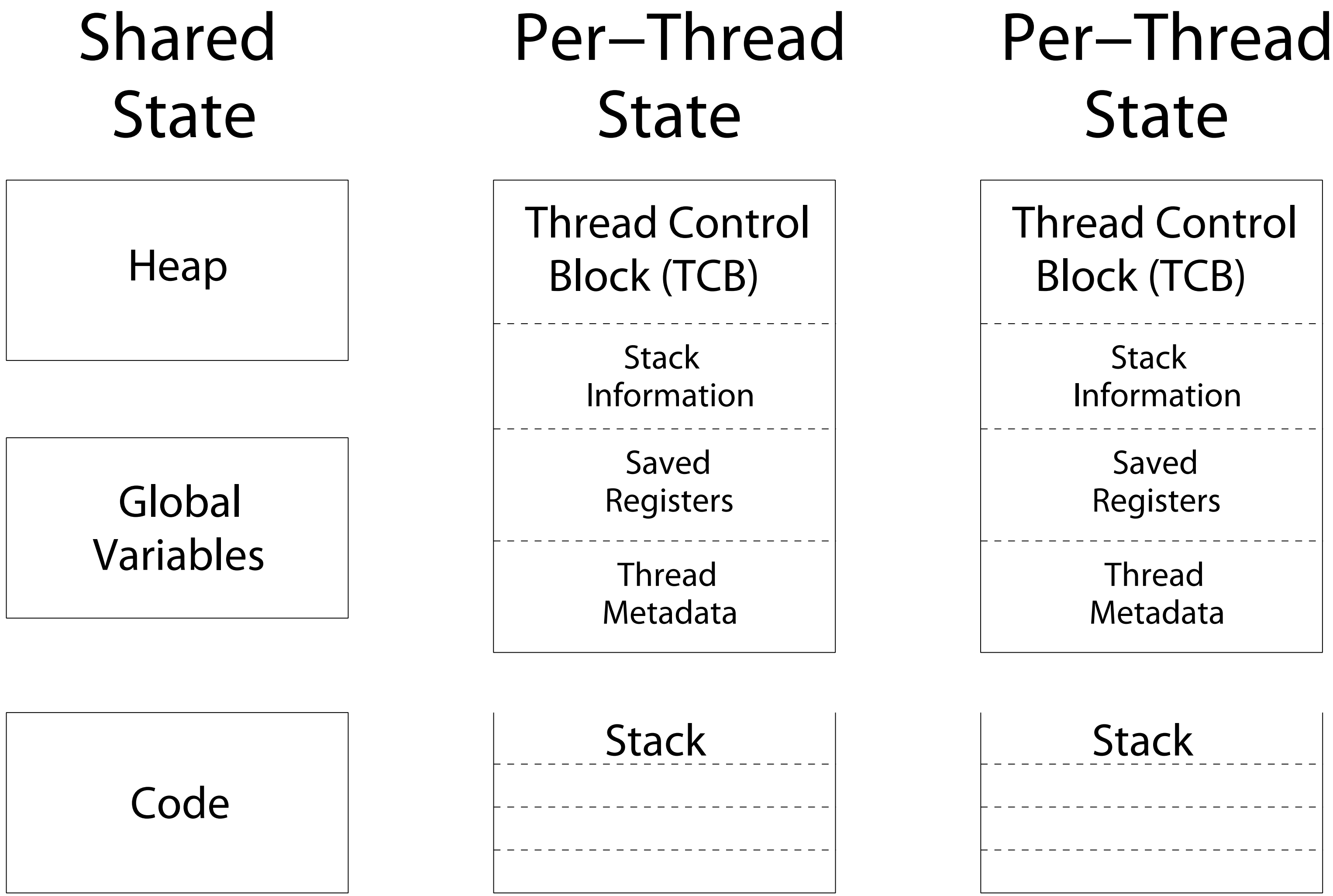
Outline

- Background
- Classical Producer-Consumer Problem
  - Basic solutions
- Race Conditions
- The Critical-Section Problem
- Peterson's Solution, Bakery Solution
- Hardware Support
- Mutex locks
- Semaphores
- Solution to P-C problem using Semaphores

Thread vs Process State

- Process-wide state:
  - Memory contents (global variables, heap)
  - I/O bookkeeping
  - Kept in **Process Control Block (PCB)**
- Thread-”local" state:
  - CPU registers including program counter (PC)
  - Execution stack
  - Kept in **Thread Control Block (TCB)**

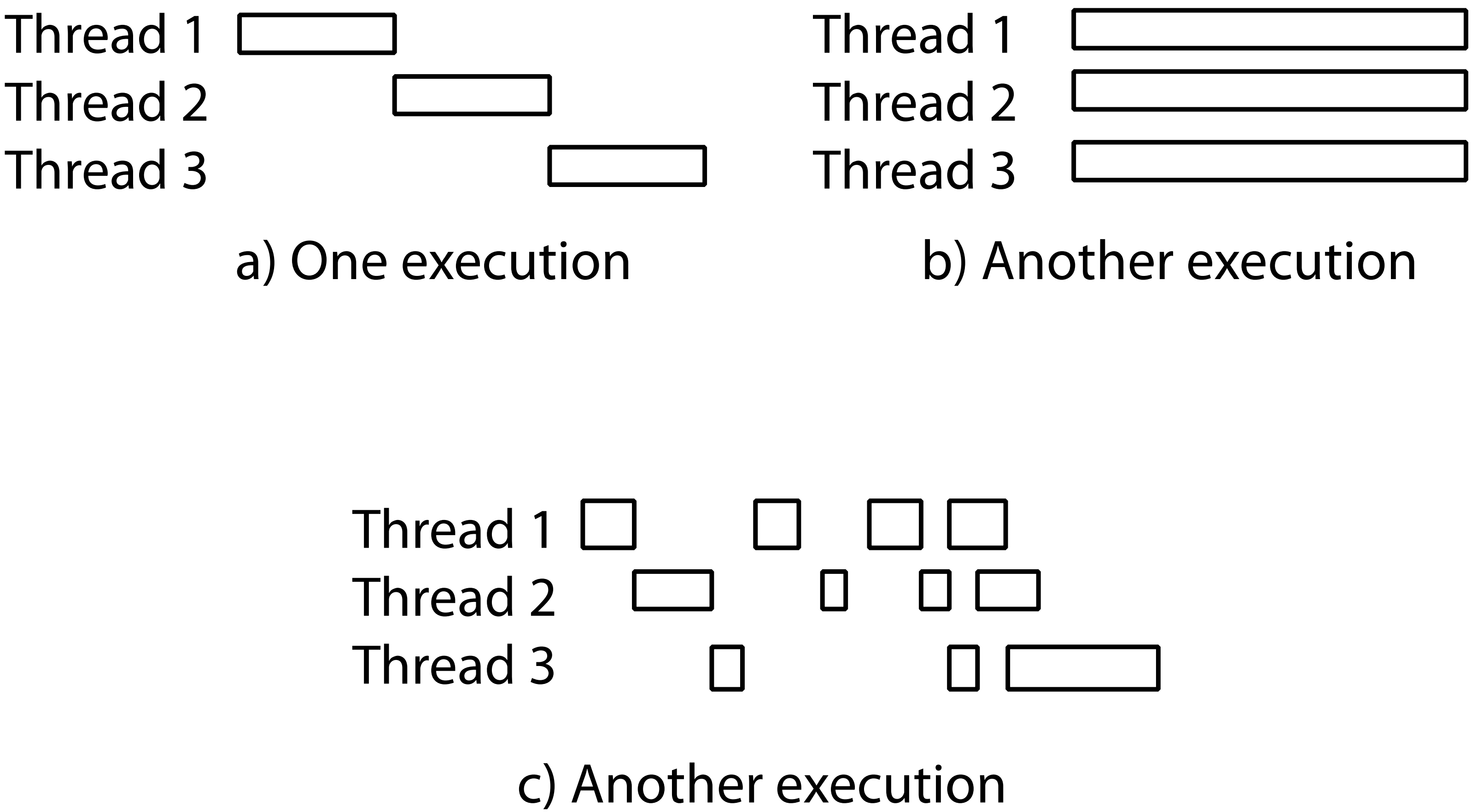
Shared vs Per-Thread State



Programmer vs. Processor View

Programmer's View	Possible Execution #1
.	.
.	.
.	.
x = x + 1;	x = x + 1;
y = y + x;	y = y + x;
z = x +5y;	z = x + 5y;
.	.
.	.
.	.

Possible Executions



Correctness with Concurrent Threads

- Non-determinism:
  - Scheduler can run threads in **any order**
  - Scheduler can switch threads **at any time**
  - This can make testing very difficult
- *Independent Threads*
  - No state shared with other threads
  - Deterministic, reproducible conditions
- *Cooperating Threads*
  - Shared state between multiple threads
- **Goal: Correctness by Design**

Shared Objects

- Concurrent access to shared (data) objects may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes/threads
- First look at some toy examples to illustrate thread hazards
- Then look at the classical consumer-producer problem
- *Producer* process produces information consumed by *Consumer* process
  - Very common paradigm for cooperating processes/threads
  - Shared memory (IPC) is used in case of processes
  - Heap/Global variables are used in case of threads

Race Conditions

- What are the possible final values of **x** below?

Thread A	Thread B
`x = 1;`	**`x = 2;`**

- It can be x = 1 or 2 depending on which thread wins or loses the race to set x → Race condition
- Definition: *a timing dependent error involving shared state*
  - Whether it happens depends on how threads scheduled or interleaved while accessing/manipulating shared objects
  - In effect, once thread A starts doing something, it needs to “race” to finish it because if thread B looks at the shared object before A is done, it may see something inconsistent

Race Conditions

- What are the possible values of **x** below?
- Initially **`y = 12;`**

Thread A	Thread B
`x = y + 1;`	`y = y * 2;`

- 13 or 25 (non-deterministic)
- Race Condition

Two threads, one counter

Popular web server

- Uses multiple threads to speed things up.
- Simple shared state error:
  - each thread increments a shared counter to track number of hits

```
...
hits = hits + 1;
...
```

- What happens when two threads execute concurrently?

Shared counters

- Possible result: lost update!

hits = 0

time

T1

read hits (0)

hits = 0 + 1

T2

read hits (0)

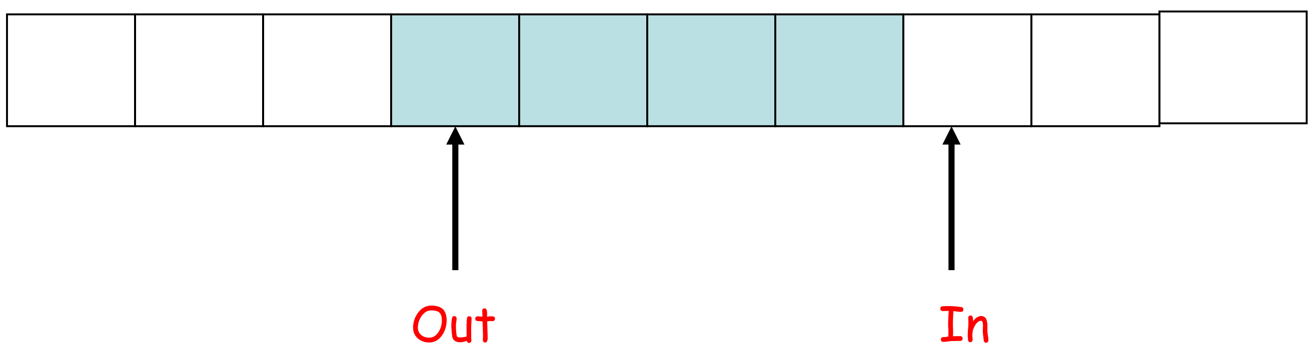
hits = 0 + 1

hits = 1

- One other possible result: everything works.
  - ⇒ Difficult to debug
- Rrace condition

Producer-Consumer Problem

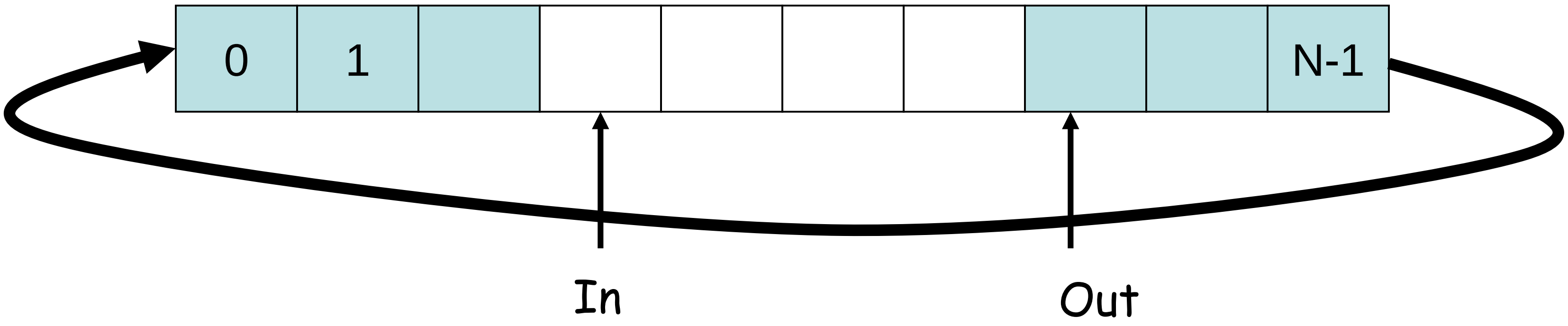
- Start by imagining an unbounded (infinite) buffer
- Producer process writes data to buffer
  - Writes to In and moves rightwards
- Consumer process reads data from buffer
  - Reads from Out and moves rightwards
  - Should not try to consume if there is no data



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
  - Must not write more than ‘N’ items more than consumer “ate”
- Consumer process reads data from buffer
  - Should not try to consume if there is no data



Solution #1: Producer/Consumer Problem

Bounded Buffer Case:

```
#define BUFFER_SIZE 10

typedef struct{
   ..some stuff..
}item;

item buffer[BUFFER_SIZE];
int in = 0
int out = 0;
```

Threads: Keep above variables in Data segment;
Processes: Use shared memory;

Solution #1: Producer/Consumer (1/2)

- Producer process/thread:

```
item nextProduced;

while(true)
{
   /*Produce an item in nextProduced*/
   while(((in + 1) % BUFFER_SIZE) == out)
     continue; //do nothing…

   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Solution #1: Producer/Consumer (2/2)

- Consumer process/thread:

```
item nextConsumed;

while(true)
{
   while(in == out)
     continue; //do nothing..

/* Consume item in nextConsumed */
   nextConsumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
}
```

Solution #1: Producer/Consumer

Producer:

```
item nextProduced;

while(true)
{
while(((in+1)%BUFFER_SIZE)==out)
      continue; //do nothing…

buffer[in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
}
```

Consumer:

```
item nextConsumed;

while(true)
{
   while(in==out)
      continue; //do nothing..
   nextConsumed=buffer[out];
   out=(out+1)%BUFFER_SIZE;
}
```

Solution #1: Analysis

- Solution #1 does not suffer from any data inconsistency issues for 1 producer and 1 consumer scenario
  - Reason: In is manipulated (write operation) only by Producer
  - Reason: Out is manipulated only by Consumer
- But, it fails in the case of multiple producers and/or multiple consumers scenarios
  - Reason: In is now manipulated by more than one Producer, similarly Out is by more than one Consumer process/thread
- In any case, solution #1 only allows BUFFER_SIZE-1 items filled-up at the same time
- To remedy this, the processes would need to *synchronize* their access to the shared buffer

Solution #2: Producer/Consumer Problem

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills in all the buffer items.
  - Assume an integer count keeps track of the number of full buffers.
  - Initially, count is set to 0.
  - It is incremented by the producer after it produces a new buffer item
  - It is decremented by the consumer after it consumes a buffer item
- Any issues with above solution?
  - Scenario 1: 1 Producer and 1 Consumer?
  - Scenario 2: More than 1 Producer and/or More than 1 Consumer?
  - Same variable is being manipulated by more than one process/thread which could be either Producer/Consumer.
    - Recall thread hazards discussed earlier in OS-1 course!

Sol #2: Producer-Consumer

- Producer

```
while (true) {

  /*  produce an item and */
  /* put in nextProduced  */

  while (count == BUFFER_SIZE)
          ; // do nothing b/c full

  buffer [in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  count++;
}
```

- Consumer

```
while (true)  {

  while (count == 0)
          ; // do nothing b/c empty

  nextConsumed =  buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  count--;

  /*  consume the item */
  /*  in nextConsumed  */
}
```

Sol #2: Analysis

- count++ *not* atomic operation. Could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- *not* atomic operation. Could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2

- Consider this execution interleaving with “count = 5” initially:
  - S0: producer executes register1 = count {register1 = 5}
  - S1: producer executes register1 = register1 + 1 {register1 = 6}
  - S2: consumer executes register2 = count {register2 = 5}
  - S3: consumer executes register2 = register2 - 1 {register2 = 4}
  - S4: producer executes count = register1 {count = 6 }
  - S5: consumer executes count = register2 {count = 4}

What just happened?

- Incorrect state of “count” as both processes manipulating this shared variable concurrently
- This is an example of race condition: When multiple jobs share & manipulate the same data concurrently, the outcome depends on the particular order in which the access takes place for the shared data (critical section)
- Threads share global memory (data segment) and many other except stack segment
  - This can result in *race conditions*
- Race conditions can arise even for processes that use IPC to coordinate
- To prevent race conditions, concurrent tasks must be synchronized to ensure that only one task at a time can manipulate shared structures/variables in critical section

Race conditions

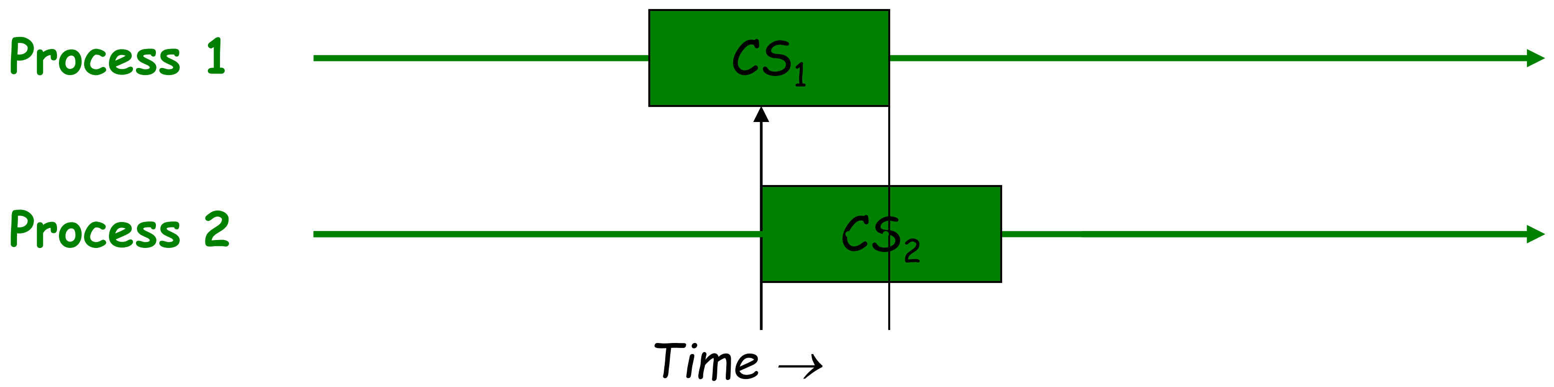
- Hard to detect:
  - All possible schedules/interleavings have to be safe
    - Number of possible schedule permutations is huge
    - Some bad schedules? Some that will work sometimes?
  - they are intermittent
    - Timing dependent = small changes can hide bug
  - Debugging is very hard as errors due to race conditions might surface very rarely
    - Celebrate if bug is deterministic and repeatable!

Race Conditions in Kernel

- OS is subject to several possible race conditions as it is implemented as multi-threaded process
- E.g., Kernel Data Structures like
  - List of Open files
  - Process table
  - DS for maintaining memory allocation
  - DS for interrupt handling
- Kernel developers have to ensure that OS is free from race conditions!
- Two general approaches to handle CSs in OS
  - Non-preemptive kernels → only one job is active in kernel at a time
  - Preemptive kernels → high priority job can preemptive low priority one during its execution in kernel
    - More responsive
    - Needed for RT scheduling of jobs

Critical Section Problem (CSP)

- Problem: Design a protocol for processes to cooperate, such that only one process is in its critical section at a time to avoid race conditions
  - How to make multiple instructions of CS seem like one atomic instruction irrespective of underlying scheduling decisions?



Processes progress with non-zero speed, no assumption on clock speed

Used extensively in operating systems:
Queues, shared variables, interrupt handlers, etc.

Scheduler assumptions

```
Thread a:
  while(i < 10)
    i = i +1;
  print "A won!";
```

```
Thread b:
  while(i > -10)
    i = i - 1;
  print "B won!";
```

If var i is shared, and initialized to 0

- Who wins on a single-CPU system?
- Is it guaranteed that someone wins?

Scheduler Assumptions

- Normally we assume that
  - A scheduler always gives every executable thread opportunities to run
    - In effect, each thread makes *finite progress*
  - But schedulers aren't always fair
    - Some threads may get more chances than others
  - To reason about worst case behavior we sometimes think of the scheduler as an adversary trying to "mess up" the algorithm

CSP Solution Structure

Shared vars:

Initialization:

Process/Thread:

. . .

. . .

Entry Section

Critical Section

Exit Section

Added to solve the CS problem

Example Problem Setting

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
    CS entry
critical section
    CS exit
reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

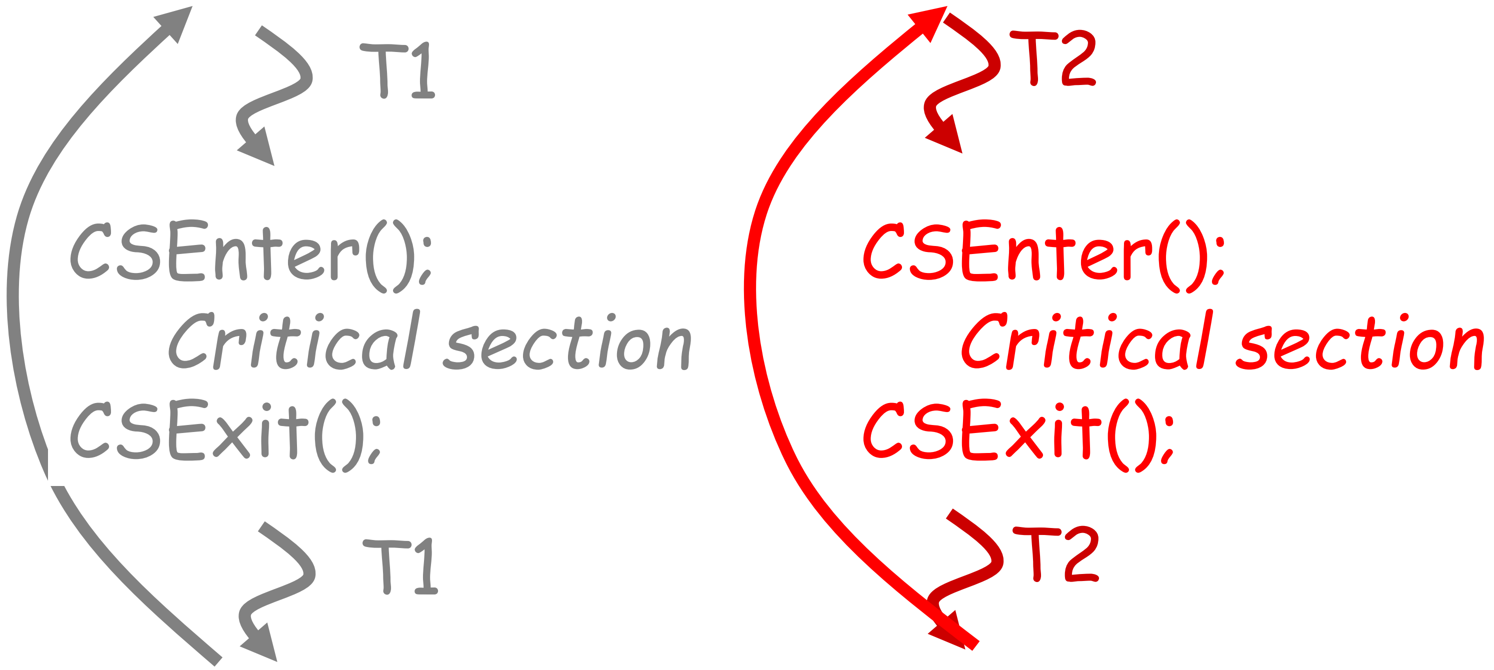
Critical Section Assumptions

- Tasks do some stuff but eventually ***might*** try to access shared data

time	T1	T2
	CSEnter();	CSEnter();
	Critical section	*Critical section*
	CSExit();	CSExit();
	T1	T2

Critical Section Assumptions

- Perhaps they loop (perhaps not!)



Critical Section Goals

- We would like
  1. Safety (aka mutual exclusion)
     - No more than one thread can be in a critical section (CS) at any time.
  2. Liveness (aka progress)
     - Only threads that are not executing in Remainder Sections can participate in deciding which will enter its CS next
     - A thread that is seeking to enter the CS will eventually succeed (i.e., selection can't be postponed indefinitely)
  3. Bounded waiting
     - A bound must exist on the number of times that other threads are allowed to enter their CSs after a thread has made a request to enter its CSand before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the N processes
- Ideally we would like fairness as well
  - If two threads are both trying to enter a critical section, they have equal chances of success
  - … in practice, fairness is rarely guaranteed!

Solving the problem

- A first idea:
  – Have a boolean flag, *inside*. Initially false.

```
CSEnter()
{
    while(inside) continue;
    inside = true;
}
```

```
    inside = false;
}
```

Code is unsafe: thread 0 could finish the while test when inside is false, but then 1 might call CSEnter() before 0 can set inside to true!

Race condition inside CSEnter code!!

- Now ask:
  – Is this Safe? Live? Bounded waiting?

Solving the problem: Take 2

- A different idea (assumes just two threads):
  - Have a boolean flag array, *inside[i]*. Initially false.

Code isn't live: with bad luck, both threads could be looping, with 0 looking at 1, and 1 looking at 0

```
CSEnter(int i)
{
    inside[i] = true;
    while(inside[j]) continue;
}
```

```
{
    Inside[i] = false;
}
```

- Now ask:
  - Is this Safe? Live? Bounded waiting?

Solving the problem: Take 3

- Another broken solution, for two threads
  - Have a turn variab

Code isn't live: thread 1 can't enter unless thread 0 did first, and vice-versa. But perhaps one thread needs to enter many times and the other fewer times, or not at all

```
CSEnter(int i)
{
    while(turn != i) continue;
}
```

```
    turn = j;
}
```

- Now ask:
  - Is this Safe? Live? Bounded waiting?

Real-Life Analogy: Too Much Milk

Time	**Person A (Alice)**	**Person B (Bob)**
7:00	Look in Fridge. Out of milk	
7:05	Leave for store A	
7:10	Arrive at store A	Look in Fridge. Out of milk
7:15	Buy milk	Leave for store B
7:20	Arrive home, put milk away	Arrive at store B
7:25		Buy milk
7:30		Arrive home, put milk away

Too Much Milk: Correctness

1. At most one person buys milk

2. At least one person buys milk if needed

Solution Attempt #1

- Leave a note
  - Place on fridge before buying
  - Remove after buying
  - Don’t go to store if there’s already a note

- Leaving/checking a note is atomic (~ load/store)

```
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove Note;
  }
}
```

Attempt #1in Action

Alice

```
if (noMilk) {
  if (noNote) {


    leave Note;
    buy milk;
    remove Note;
  }
}
```

Bob

```


if (noMilk) {
  if (noNote) {






    leave Note;
    buy milk;
    remove note;
  }
}
```

Solution Attempt #2

```
leave Note;
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
   }
}
remove Note;
```

But there's always a note – you just left one!

At least you don't buy milk twice…

Solution Attempt #3

- Leave a named note – each person ignores their own

Alice

```
leave note Alice
if (noMilk) {
  if (noNote Bob) {
    buy milk
  }
}
remove note Alice;
```

Bob

```
leave note Bob
if (noMilk) {
  if (noNote Alice) {
    buy milk
  }
}
remove note Bob;
```

Attempt #3 in Action

Alice

```
leave note Alice
if (noMilk) {

  if (noNote Bob) {
    buy milk
  }
}
```

Bob

```
leave note Bob

if (noMilk) {
  if (noNote Alice) {
    buy milk
  }
remove note Bob
```

```
remove note Alice
```

Solution Attempt #4

Alice

```
leave note Alice
while (note Bob) {
  do nothing
}
if (noMilk) {
  buy milk
}
remove note Alice;
```

Bob

```
leave note Bob
if (noNote Alice) {
  if (noMilk) {
    buy milk
  }
}
remove note Bob;
```

- This is a correct solution, but …

Issues with Solution #4

- Complexity
  - Proving that it works is hard
  - How do you add another thread?

- Busy-waiting
  - Alice **consumes CPU time to wait**
- Fairness
  - Who is more likely to buy milk?

Peterson's Solution

- Process P_i

```
do {
    inside [i]:= true;
    turn = j;
while (inside [j] && turn == j)  continue;
        critical section
    inside [i] = false;
        remainder section
} while (true);
```

- Meets all three requirements; solves the critical-section problem for two processes.

Analysis of Peterson’s algorithm:

- Safety (by contradiction):
  - Assume that both processes (P1 and P2) are in their critical section (and thus have their *inside* flags set). Since only one, say P1, can have the *turn*, the other (P2) **must** have reached the while() test before P1 set his *inside* flag.
  - However, after setting his *inside* flag, P1 gave away the *turn* to P2. P2 has already changed the turn and **cannot** change it again, contradicting our assumption.

Liveness & Bounded waiting => the turn variable.

Issues with Peterson's algorithm

- Complexity: 3 variables: turn and a two-element array
- Limited for two processes, though it can be extended to any N processes with more complexity
- Relies on busy waiting (aka spin locks)
- It may not work on modern systems
  – load and store may not be atomic instructions
    - Ex: double-precision floating point store instruction
    - **store** mem, R1 && **store** mem, R2
    - int I = 0; //Atomic
    - long long a =0; // Not atomic, 2 cycles as memory bus is 32-bit
  – memory reference ordering may not be preserved at the memory controller in multi-core/CPU systems

Dekker's Solution

```
CSEnter(int i)
{
  inside[i] = true;
  while(inside[J])
  {
    if (turn == J)
    {
       inside[i] = false;
       while(turn == J) continue;
       inside[i] = true;
     }
  }}
```

```
CSExit(int i)
{
  turn = J;
  inside[i] = false;
  }
```

Napkin analysis of Dekker's algorithm:

- Safety: No process will enter its CS without setting its *inside* flag. Every process checks the other process *inside* flag after setting its own. If both are set, the *turn* variable is used to allow only one process to proceed.
- Liveness: The *turn* variable is only considered when both processes are using, or trying to use, the resource
- Bounded waiting: The *turn* variable ensures alternate access to the resource when both are competing for access

Why does it work?

- Safety: Suppose thread 0 is in the CS.
    - Then inside[0] is true.
    - If thread 1 was simultaneously trying to enter, then turn must equal 0 and thread 1 waits
    - If thread 1 tries to enter “now”, it *sets* turn to 0 and waits
- Liveness: Suppose thread 1 wants to enter and can’t (stuck in while loop)
  – Thread 0 will eventually exit the CS
  – When inside[0] becomes false, thread 1 can enter
  – If thread 0 tries to reenter immediately, it sets turn=1 and hence will wait politely for thread 1 to go first!

Summary

- Dekker's algorithm does not provide *strict* alternation
  - Initially, a thread can enter critical section without accessing *turn*

- Dekker's algorithm will not work with many modern CPUs
  - CPUs execute their instructions in an out-of-order (OOO) fashion
  - This algorithm won't work on Symmetric MultiProcessors (SMP) CPUs equipped with OOO without the use of *memory barriers*

- Additionally, Dekker's algorithm can fail regardless of platform due to many optimizing compilers
  - Compiler may remove writes to *flag* since never accessed in loop
  - Further, compiler may remove *turn* since never accessed in loop
    - Creating an infinite loop!

Can we generalize Peterson's solution to more than 2 processes/threads?

- Obvious approach won't work:

```
CSEnter(int i)
{
    inside[i] = true;
    for(J = 0; J < N && J!=i; J++)
        while(inside[J] && turn == J)
            continue;
}
```

```
CSExit(int i)
{
    inside[i] = false;
}
```

- Issue: Who's turn next?

Bakery “concept”

- Described by Leslie Lamport
- Think of a popular store with a crowded counter
  - People take a ticket from a machine
  - If nobody is waiting, tickets don’t matter
  - When several people are waiting, ticket order determines order in which they can make purchases

Bakery Algorithm: "Take 1"

- int ticket[n]; // initialized to 0s
- int next_ticket; //initialized to 0

```
CSEnter(int i)
{
    ticket[i] = ++next_ticket;
    for(J = 0; J < n && J!=i; J++)
        while(ticket[J] && ticket[J] < ticket[i])
            continue;
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

- Oops… access to next_ticket is a problem!
- ++next_ticket is not atomic

Bakery Algorithm: “Take 2”

- int ticket[n];

Just add 1 to the max!

```
CSEnter(int i)
{
    ticket[i] = max(ticket[0], … ticket[N-1])+1;
    for(J = 0; J < n && J!=i; J++)
        while(ticket[J] && ticket[j] < ticket[i])
            continue;
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

- Clever idea: just add one to the max.
- Oops… two could pick the same value!

Bakery Algorithm: “Take 3”

If i, j pick same ticket value, id’s break tie:

(ticket[J] < ticket[i]) || (ticket[J]==ticket[i] && J<i)

Notation: (B,J) < (A,i) to simplify the code:

(B<A || (B==A && J<i)), e.g.:

(ticket[J],J) < (ticket[i],i)

Bakery Algorithm: “Take 4”

- int ticket[N];
- boolean picking[N] = false;

```
CSEnter(int i)
{
    ticket[i] = max(ticket[0], … ticket[N-1])+1;
    for(J = 0; J < N && J!=i; J++)
        while(ticket[J] && (ticket[J],J) < (ticket[i],i))
            continue;
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

- Both i & J enter into max( ) at the same time (i.e, get same ticket number), but i goes fast and enters while
- Oops… i could look at J when J is still storing its ticket, and yet J could have a lower id than me (i)!
  - So, both enter CS!!

Bakery Algorithm: Almost final

- int ticket[N];
- boolean choosing[N] = false;

```
CSEnter(int i)
{
    choosing[i] = true;
    ticket[i] = max(ticket[0], … ticket[N-1])+1;
    choosing[i] = false;
    for(J = 0; J < N && J!=i; J++) {
        while(choosing[J]) continue;
        while(ticket[J] && (ticket[J],J) < (ticket[i],i))
            continue;
    }
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

Bakery Algorithm: Final

- int ticket[N];
- boolean choosing[N] = false;

```
CSEnter(int i)
{
    do {
        ticket[i] = 0;
        choosing[i] = true;
        ticket[i] = max(ticket[0], … ticket[N-1])+1;
        choosing[i] = false;
    } while(ticket[i] >= MAXIMUM);
    for(J = 0; J < N && J!=i; J++) {
        while(choosing[J]) continue;
        while(ticket[J] && (ticket[J],J) < (ticket[i],i))
            continue;
    }
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

Bakery Algorithm: Issues?

- What if we don’t know how many threads might be running?
  - The algorithm depends on having an agreed upon value for N
  - Somehow would need a way to adjust N when a thread is created or one goes away
- Also, technically speaking, ticket can overflow!
  - Solution: Change code so that if ticket is “too big”, set it back to zero and try again.

How do real systems do it?

- Some real systems actually use algorithms such as the bakery algorithm
  - A good choice where busy-waiting isn't going to be super-inefficient
  - For example, if you have enough CPUs so each thread has a CPU of its own
- Some systems disable interrupts briefly when entering Critical Sections
  - Time consuming in multi-CPU systems for msg passing others about interrupt disabling
- Some systems take hardware "help": Special *atomic instructions*
  - *Test-and-Set, Swap, Read-Modify-Write, etc*
  - *They guarantee exclusive access to memory locations (aka lock variables)*

Concurrent Applications

Shared Objects

Bounded Buffer Barrier

Synchronization Variables

Semaphores Locks Condition Variables

Atomic Instructions

Interrupt Disable Test-and-Set

Hardware

Multiple Processors Hardware Interrupts

Lock Variables

- Lock variable is a shared object that is guaranteed to be accessed by a single process at any given time
- Lock variables are used to achieve mutual exclusion

Process/Thread:

. . .

Acquire Lock

Critical Section

Release Lock

. . .

Test_and_Set Instruction

- Definition:

```
boolean test_and_set (boolean *target)
 {
     boolean retValue = *target;
     *target = TRUE;
     return retValue;
 }
```

Critical Sections with Atomic Hardware Primitives

Share: int lock;
Initialize: lock = false;

Process i

```
While(test_and_set(&lock))
   CONTINUE;
```

Assumes that test_and_set is compiled to a special hardware instruction that sets the lock and returns the OLD value (true: locked; false: unlocked)

Critical Section

```
lock = false;
```

Problem: Does not satisfy bounded waiting)
(see Galvin 9th Ed for correct solution, Figure 5.7)

Solution using TestAndSet

- Shared boolean variable *lock* initialized to FALSE.
- Solution:

```
while (true) {
        while ( test_and_set (&lock ))
                Continue;   /* do nothing

                //    critical section

        lock = FALSE;

                //      remainder section
}
```

Compare_and_Swap Instruction

- Definition:

```
int compare_and_swap (int *value, int expected, int new_value)
    {
        int temp = *value;
        if (*value == expected)
            *value = new_value;

        return temp;
    }
```

Solution using Compare_and_Swap

- Shared Boolean variable lock initialized to FALSE (0)
- Solution:

```
while (true)  {
     while (compare_and_swap(&lock, 0, 1) !=0)
continue;

          //    critical section

          lock = 0;

          //      remainder section
}
```

H/W Implementation of Test_and_Set and Compare_and_Swap Instructions

- Test_and_Set is implemented using TSL h/winstruction
  - TSL register, lock
  - It loads the content of the variable named lock to register and writes a not NULL value into the lock variable atomically
- Compare_and_Swap is implemented using CMPXCHG h/w instruction
  - CMPXCHG register, lock
  - Swaps the contents of the register andthe lock variable atomically
- H/W guarantees atomicity by disabling the access to the memory bus to the other processors
- Nowadays the x86 instruction set allows all instructions to be atomic
  - The lock prefix makes atomic any instruction following it
    - **lock store register, memLoc**
  - **This lock prefix can be used with all the memory access instructions**
    - **INC, DEC, ADD, SUB, XOR, etc**

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is the mutex lock
- Protect critical regions with it by first **`acquire()`** a lock then **`release()`** it
  - Boolean variable indicating if lock is available or not

- Calls to **`acquire()`** and **`release()`** must be atomic
  - CSEntry and CSExit facing race conditions
  - To solve this CS problem in CSEntry and CSExit method, they are usually implemented via hardware atomic instructions!!
    - Test_and_Set and Compare_and_Swap

- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**
- **Even hardware instructions** Test_and_Set and Compare_and_Swap suffer from busy waiting!

acquire() and release()

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;;
}
release() {
   available = true;
}
do {
   acquire()
      critical section

 release()
      remainder section
} while (true);
```

```
while(TestAndSet(&lock));
while (!available);
   available = false;
lock=false;
```

- Code for release()?
- Is preemption possible in critical section?

Presenting critical sections to users

- CSEnter and CSExit are possibilities
- But more commonly, operating systems have offered a kind of locking primitive
- We call these <u>*semaphores*</u>

Semaphores

- Non-negative integer with atomic increment and decrement
- Integer ‘S’ that (besides initialization) can only be modified by:
  - P(S) or S.wait(): decrement or block if already 0
  - V(S) or S.signal(): increment and wake up process if any
- These operations are *atomic* (indivisible)

These systems use the operation **signal()** instead of **V()**

```
sema…

P(S) {                          V(S) {
   while(S ≤ 0)                    S++;
      ;                         }
   S--;
}
```

Semaphore Types

- Counting Semaphores:
  - Any integer
  - Used for synchronization
- Binary Semaphores
  - Value is limited to 0 or 1
  - Used for mutual exclusion (mutex)

Shared: semaphore S

Init: S = 1;

Process i

P(S);

Critical Section

V(S);

Semaphore Implementation

- Must guarantee that no two processes can execute P () and V () on the same semaphore at the same time
  - No process may be interrupted in the middle of these operations

- Thus, implementation becomes the critical section problem where the P and V codes are placed in the critical section.
  - Could now have busy waiting in critical section of critical section implementation!
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.
- Busy waiting (spinlocks)
  - Consumes CPU resources
  - No context switch overhead, so OK n multi-CPU systems

Semaphore Implementation with no Busy waiting

- Alternative approach: Blocking
- With each semaphore there is an associated waiting queue. Each entry/node in a waiting queue (linked list) has two data items:
  - pointer to PCB of a blocked process
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.
- Should spin or block?
  - Less time $\Rightarrow$ spin
  - More time $\Rightarrow$ block
  - A theory result:
- Spin for as long as block cost
- If lock is still not available, then block
- Shown factor of 2-optimal!

Implementing Semaphores

```
typedef struct semaphore {
    int value:
    ProcessList L;
} Semaphore;

void P(Semaphore *S) {
    S->value = S->value - 1;
    if (S.value < 0) {
        add this process P to S.L;
        block(P); //moved to
    waiting Queue of Semaphore
    }
}
```

```
void V(Semaphore *S) {
    S->value = S->value + 1;
    if (S->value <= 0) {
        remove a process Q from
    S.L;
        wakeup(Q); //moved from
    waiting to Ready Queue of CPU
    }
}
```

Implementing Semaphores

- Per-semaphore list of processes
  - Implemented using PCB link field
  - Queuing Strategy: FIFO works fine
    - Will LIFO work?

Common programming errors

Whoever next calls P() will freeze up. The bug might be confusing because that other process could be perfectly correct code, yet that's the one you'll see hung when you use the debugger to look at its state!

A typ
(foreve
P() ope
process will freeze up too wh
to enter the critical se

ss J won't respect
n even if the other
the rules correctly.
ce we've done two
"ext operations this way, other
proce might get into the CS
propriately!

Process

P(S)
CS
P(S)

V(S)
CS
V(S)

P(S)
CS

More common mistakes

- Conditional code that can break the normal top-to-bottom flow of code in the critical section
- Often a result of someone trying to maintain a program, e.g. to fix a bug or add functionality in code written by someone else

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

Producer-Consumer Problem

- Solving with semaphores
  - We'll use two kinds of semaphores
  - We'll use *counters* to track how much data is in the buffer
    - One counter counts as we add data and stops the producer if there are N objects in the buffer
    - A second counter counts as we remove data and stops a consumer if there are 0 in the buffer
  - Idea: since general semaphores can count for us, we don't need a separate counter variable
- Why do we need a second kind of semaphore?
  - We'll also need a mutex semaphore

Producer-consumer with a bounded buffer

- Problem Definition
  - Producer puts things into a shared buffer (wait if full)
  - Consumer takes them out (wait if empty)
  - Use a fixed-size buffer between them to avoid lockstep
    - Need to synchronize access to this buffer
- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full
    - scheduling constraint
  - Producer must wait for consumer to empty buffers, if all full
    - scheduling constraint
  - Only one thread can manipulate buffer queue at a time
    - mutual exclusion
- Remember why we need mutual exclusion

- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore full;  // consumer's constraint`
  - `Semaphore empty; // producer's constraint`
  - `Semaphore mutex; // mutual exclusion`

Producer-Consumer Problem

```
Init: Semaphore mutex = 1;  /* for mutual exclusion*/
      Semaphore empty = N; /* number empty buf entries */
      Semaphore full = 0;     /* number full buf entries */
      any_t buf[N];
      int tail = 0, head = 0;
```

Producer

```
void put(char ch)  {

   P(empty);
   P(mutex);

   // add ch to buffer
   buf[head%N] = ch;
   head++;

   V(mutex);
   V(full); //not V(empty)
}
```

Consumer

```
char get() {

   P(full);
   P(mutex);

   // remove ch from buffer
   ch = buf[tail%N];
   tail++;

   V(mutex);
   V(empty); //not V(full)

   return ch;
}
```

What's wrong?

```
Init: Semaphore mutex = 1;  /* for mutual exclusion*/
      Semaphore empty = N;  /* number empty buf entries */
      Semaphore full = 0;   /* number full buf entries */
      any_t buf[N];
      i
```

Oops! Even if you do the correct operations, the *order* in which you do semaphore operations can have an incredible impact on correctness

What if buffer is full?

Producer

```
void put(char ch

    P(mutex);
    P(empty);

    // add ch to buffer
    buf[head%N] = ch;
    head++;

    V(mutex);
    V(full);
}
```

mer

```
get() {

    ull);
    P(mutex);

    // remove ch from buffer
    ch = buf[tail%N];
    tail++;

    V(mutex);
    V(empty);

    return ch;
}
```

What's wrong?

```
Init: Semaphore mutex = 1;  /* for mutual exclusion*/
      Semaphore empty = N;  /* number empty buf entries */
      Semaphore full = 0;   /* number full buf entries */
      any_t buf[N];
      int tail = 0, head = 0;
```

Producer

```
void put(char ch)  {

   P(empty);
   P(mutex);

   // add ch to buffer
   buf[head%N] = ch;
   head++;

  V(full);
  V(mutex);
}
```

Consumer

```
char get() {

   P(full);
   P(mutex);

   // remove ch from buffer
   ch = buf[tail%N];
   tail++;

   V(mutex);
   V(empty);

    return ch;
}
```

What's wrong?

```
Init: Semaphore mutex = 1;  /* for mutual exclusion*/
      Semaphore empty = N;  /* number empty buf entries */
      Semaphore full = 0;   /* number full buf entries */
      any_t buf[N];
      int tail = 0, head = 0;
```

Producer

```
void put(char ch)  {

   P(empty);
   P(mutex);

   // add ch to buffer
   buf[head%N] = ch;
   head++;

  V(mutex);
  V(full);
}
```

Consumer

```
char get() {

   P(full);
   P(mutex);

   // remove ch from buffer
   ch = buf[tail%N];
   tail++;

   return ch;

   V(mutex);
   V(empty);
}
```

Discussion about Bounded Buffer Solution

- Why asymmetry?
  - Producer does: `P(empty), V(full)`
  - Consumer does: `P(full), V(empty)`
- Is order of P’s important?
  - Yes! Can cause deadlock
- Is order of V’s important?
  - No, except that it might affect scheduling efficiency
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?

In a Nut Shell…

- Fundamental Issue
  - Programmers *atomic* operation is not done atomically
  - Atomic Unit: instruction sequence guaranteed to execute indivisibly
  - Also called “critical section” (CS)
- Critical Section Implementation
  - Software: Dekker’s, Peterson’s, Baker’s algorithm
  - Hardware: test_and_set, swap
    - Hard for programmers to use
  - Operating System: semaphores
- Implementing Semaphores
  - Multithread synchronization algorithms shown earlier
  - Could have a thread disable interrupts, put itself on a “wait queue”, then context switch to some other thread (an “idle thread” if needed)
  - The O/S designer makes these decisions and the end user shouldn’t need to know

Reading Assignment

- Chapter 5 from OSC by Galvin et al 9th Edition
- Chapter 2 from MOS by Tanenbaum et al

References

- "Numbers Everyone Should Know" from Jeff Dean:http://brenocon.com/dean_perf.html
- Peterson’s algorithm:
- https://en.wikipedia.org/wiki/Peterson%27s_algorithm
- Bakery algorithm: http://www.cs.umd.edu/~shankar/412-S99/note-7.html
- Intel 64 and IA-32 Archs: Chapter 8, Multi-Processor Mgmt