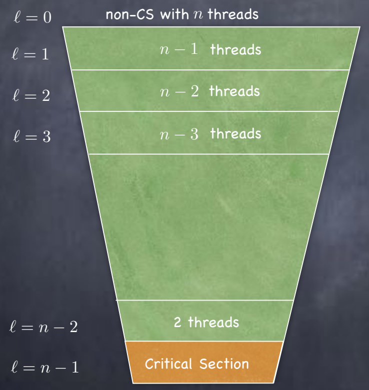
Pheeeeeeewwww...

but what if we have more than 2 threads?

Filter lock:

when 2 threads aren't enough



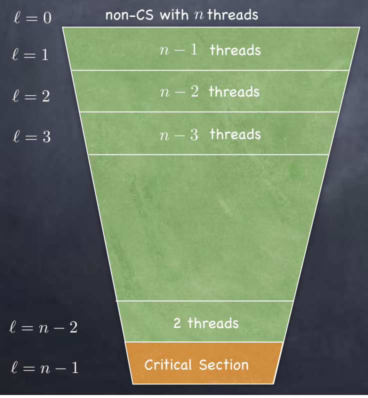
- n -level Peterson
  - level 0: non CS
  - level $1 \ldots n-2$: waiting rooms
  - level $n-1$: CS
- Each level leaves one process (the "victim") in its waiting room

---

Filter lock:

when 2 threads aren't enough

```
class Filter implements lock {
  int[] level;
  int[] victim;
  ...
}
```



```
public Filter(int n) {
  level := new int[n];
 victim := new int[n];
  for (int i := 0; i < n; i++) {
   level[i] := 0;
   }
}

public void acquire(){
 int me := ThreadID.get();
 for (int  i := 1; i < n; i++){
    level[me] := i;
    victim[i] := me;
    while ((∃k ≠ me) (level[k] ≥ i ∧ victim[i] = me)) {};
 }
}

public void release() {
 int me := ThreadID.get();
 level[me] := 0;
 }
```

Fairness

- Threads have no guarantees of entering CS in the order they called $acquire()$
- Towards that goal, we split $acquire()$ in two sections:
  - doorway: an interval D consisting of a bounded number of steps
  - waiting: an interval W that may take an unbounded number of steps

FIFO lock:	if T_1 finishes doorway before T_2, then T_1 acquires CS before T_2

Lamport's Bakery algorithm

- Each thread that wants to enter CS, acquires a ticket
- New ticket number is higher than that any ticket previously acquired
- Threads enter CS in increasing ticket number
- Acquiring a ticket is not an atomic action...

The Bakery lock

```
class Bakery implements lock{
  boolean[]  flag;
  Ticket[] ticket;

 public Bakery(int n) {
    flag :=  new boolean[n];
    ticket := new Ticket [n];
    for (int i := 0;  i < n;  i++) {
        flag[i] := false;  ticket[i] := 0;
    }
 }

 public void  acquire(){
   int  i := ThreadID.get();
   flag[i] := true;
   ticket[i] := max (ticket[0], . . . , ticket[n − 1]) + 1;
   while ((∃k ≠ i) (flag[k] ∧ (ticket[k], k ≪ ticket[i], i)))  {};
 }

 public void release(){
  flag[ThreadID.get()] := false;
 }
}
```

The Bakery lock

```
class Bakery implements lock{
  boolean[]  flag;
  Ticket[] ticket;

 public Bakery(int n) {
    flag :=  new boolean[n];
    ticket := new Ticket [n];
    for (int i := 0;  i < n;  i++) {
        flag[i] := false;  ticket[i] := 0;
    }
 }

 public void  acquire(){
   int  i := ThreadID.get();
   flag[i] := true;
   ticket[i] := max (ticket[0], . . . , ticket[n − 1]) + 1;
   while ((∃k ≠ i) (flag[k] ∧ (ticket[k], k ≪ ticket[i], i)))  {};
 }

 public void release(){
  flag[ThreadID.get()] := false;
 }
}
```

Lemma 1

Bakery-lock is deadlock free

Proof Some waiting thread has the lowest (id, ticket) combination

Lemma 2

Bakery-lock satisfies mutual exclusion

Proof Suppose T_1 and T_2 in mutual exclusion, and that

$(ticket[T_1], T_1) \ll ticket[T_2], T_2)$ ●

- when T_2 entered CS, $flag[T_1]$ must have been false.
- T_1 computed its ticket after T_2
- contradiction with ●

Lemma 3

Bakery-lock is a FIFO lock

Proof If $D_{T_1} \rightarrow D_{T_2}$, $ticket[T_2] > ticket[T_1]$ so T_2 cannot enter CS while $flag[T_1]$.

Corollary

Bakery-lock is starvation-free

Why isn't everyone using the Bakery lock?

- Elegant
- Concise
- Fair

Requires to read N distinct variables

Surely we can do better...

Theorem Deadlock-free mutual exclusion among N threads requires at least N multi-reader/single-writer (MRSW) registers.

Theorem Deadlock-free mutual exclusion among N threads requires at least N multi-reader/multi-writer (MRMW) registers.

A New Hope

- How can we do better?
  - Use hardware to support atomic operations beyond load and store
  - Define higher-level programming abstractions that leverage hardware support

---

Only on uni-processors

Disabling Interrupts for Mutual Exclusion

lock.acquire()	{ disable interrupts}
lock.release()	{ enable interrupts}

- Simple, but flawed
  - thread may never give up CPU!
  - even if it does, it could it take too long to respond to an interrupt

---

Only on uni-processors

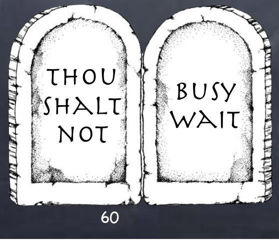
Disabling Interrupts: A Refinement

- Use a variable to implement the lock; enforce mutual exclusion only on the operations that test and modify that variable

```
class lock { int value := FREE }

lock.acquire() {
  disableInterrupts();
  while (value == BUSY) {
    enableInterrupts();
    disableInterrupts();
  }
  value := BUSY;
  enableInterrupts();
}

lock.release() {
  disableInterrupts();
  value := FREE;
  enableInterrupts();
}
```



Lock Implementation: Uniprocessor

Only on uni-processors

- If lock is BUSY, wait on a queue and switch to another process

```
class lock { int value := FREE }
```

```
lock.acquire() {
   disableInterrupts();
   if (value == BUSY) {
      current->state = WAITING
      waiting.Add(current);
      next = scheduler();
      next->state = RUNNING;
      ctx_switch(&current->sp, next->sp);
      current = next;
   } else {
      value := BUSY;
   }
   enableInterrupts();
}
```

who's returning?

```
lock.release() {
   disableInterrupts;
   if (!waiting.Empty()) {
      next = waiting.Remove();
      next->state = READY;
      readyQueue.add(next);
   } else {
      value := FREE;
   }
   enableInterrupts();
}
```

Atomic Read/Modify/Write

Also for multi-processors

- On a multiprocessor, disabling interrupts does not ensure atomicity
  - other CPUs could still enter the critical section
  - costly to disable interrupts on all CPUs
- Hardware provides special machine instructions
  - Test-and-Set (TAS)
    - reads in a register the value of a memory location, writes back TRUE in its place
    - TAS (value, r): ⟨r := value; value := TRUE⟩ (r is usually not explicit)
  - Compare-and-Swap (CAS)
    - compares contents of a memory location to given value; if same, sets memory location to a new given value
  - Many others (e.g. Load Link Store Conditional)

```
bool CAS (*int p, int old, int new) {
   if *p ≠ old return FALSE;
   *p := new;
    return TRUE
  }
```

Spinlocks

- A lock where the processor waits in a tight loop for the lock to become free
  - lock should be held for a short time
  - used to protect CPU scheduler and implement more general locks

```
lockValue := FREE

spinLock.acquire() {
   while (TAS(lockValue) ==BUSY
   ;
}
spinLock.release() {
   lockValue := FREE;
}
```

* well, certainly not for the time needed to complete an arbitrarily long CS!

How Many Spinlocks?

- Various data structures need safe concurrent access, e.g.,
  - list of threads waiting on lock I
  - list of threads waiting on lock J
  - ready queue
- One spinlock for the entire kernel? Bottleneck!
- Instead
  - one spinlock per lock
  - one spinlock for ready queue
    - Per-core ready list: one spinlock per core

Lock Implementation: Multiprocessor

```
lock.acquire() {
  disableInterrupts();
  spinLock.acquire();
  if (value == BUSY) {
     waiting.Add(current);
     suspend(&spinlock);
  } else {
     value = BUSY;
  }
  spinLock.release();
  enableInterrupts();
}
```

```
lock.release() {
  disableInterrupts();
  spinLock.acquire();
  if (!waiting.Empty()) {
     next = waiting.Remove()
     makeReady(next);
  } else {
     value := FREE;
  }
  spinLock.release();
  enableInterrupts;
}
```

Lock Implementation: Multiprocessor

```
suspend(SpinLock *lock) {
  struct PCB *next;

  disableInterrupts();
  schedSpinLock.acquire();
  lock->release();
  current->state = WAITING;
  next = scheduler();
  next->state = RUNNING:
  ctx_switch(current, next);
  current = next;
  schedSpinLock.release();
  enableInterrupts();
}
```

```
makeReady(struct PCB *thread) {
     disableInterrupts();
     schedSpinLock.acquire();
     readyQueue.add(thread);
     thread->state = READY;
     schedSpinLock.release();
     enableInterrupts();
}
```

SEMAPHORES

Semaphores (Dijkstra, 1962)

- Introduced in THE Operating System
  - catchy name...
- Stateful
  - a non-negative integer (count)
  - a lock
  - a queue
- Interface
  - Init (starting value)
  - P(): decrement Probeer ("Try")
    - procure
  - V(): increment Verhoog ("+1")
    - vacate

No operation to read the semaphore's value

NONE!

Semantics of P and V

- P():
  - wait until count > 0
  - when so, decrement count by 1

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

- V():
  - increment count by 1

```
V() {
  n := n+1;
}
```

Binary Semaphores: count can be either 0 or 1

Semaphore's count

- Must be initialized
- Maintains the semaphore's state
  - Reflects sequence of past P, V operations
  - Positive value indicates how many future P operations will succeed
- Important
  - It is not possible to read the count
  - It is not possible to increase or decrease the count but through P and V
  - It is not possible to increment/decrement by more than 1

Implementing semaphores

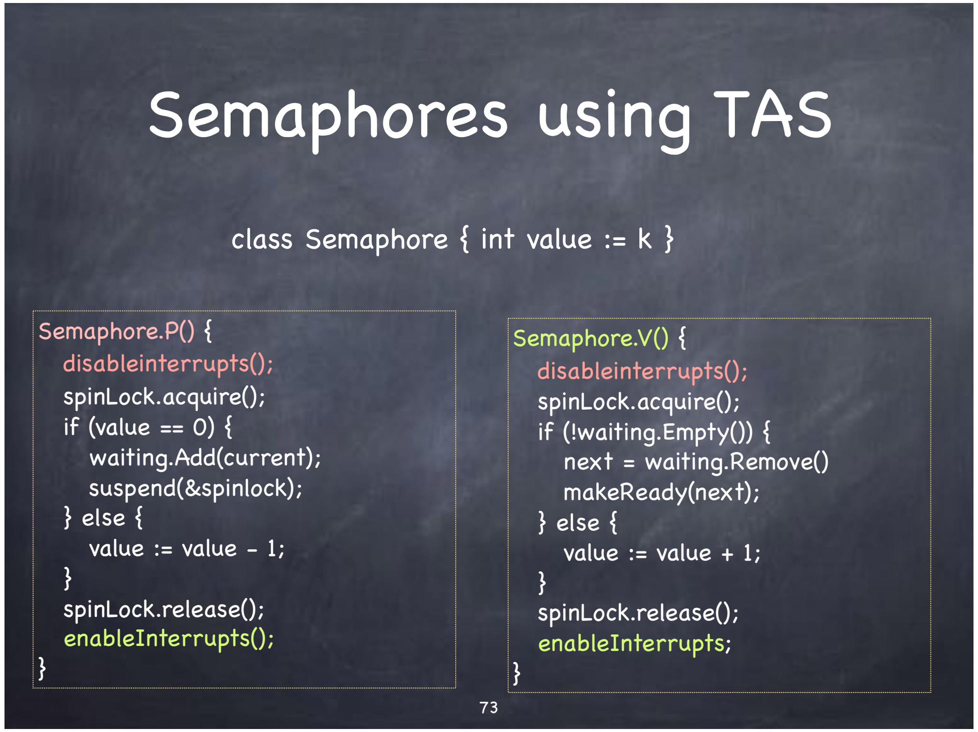
- Been there, done that:
  - by enabling/disabling interrupts
  - by using TAS
    - with a queue, to avoid busy waiting

Semaphores with interrupts

class Semaphore { int value := k }

```
Semaphore.P() {
  Disable interrupts;
  while (value == 0) {
    Enable interrupts;
    Disable interrupts;
  }
  value := value - 1;
  Enable interrupts;
}
```

```
Semaphore.V() {
  Disable interrupts;
  value := value + 1;
  Enable interrupts;
}
```

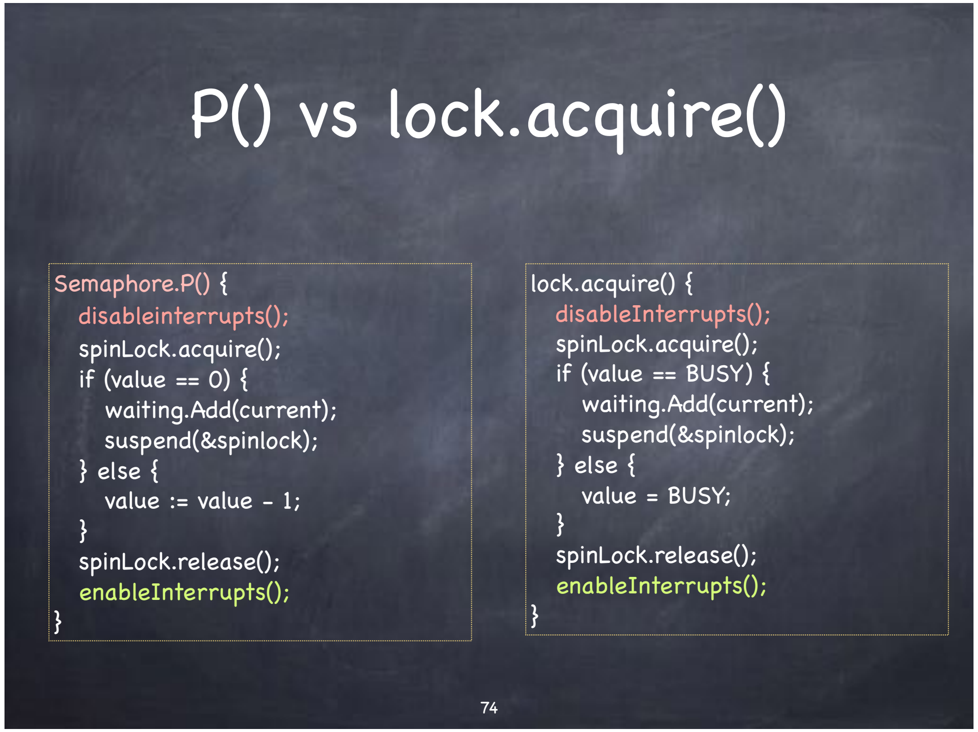


Semaphores using TAS

class Semaphore { int value := k }

```
Semaphore.P() {
   disableinterrupts();
   spinLock.acquire();
   if (value == 0) {
      waiting.Add(current);
      suspend(&spinlock);
   } else {
      value := value - 1;
   }
   spinLock.release();
   enableInterrupts();
}
```

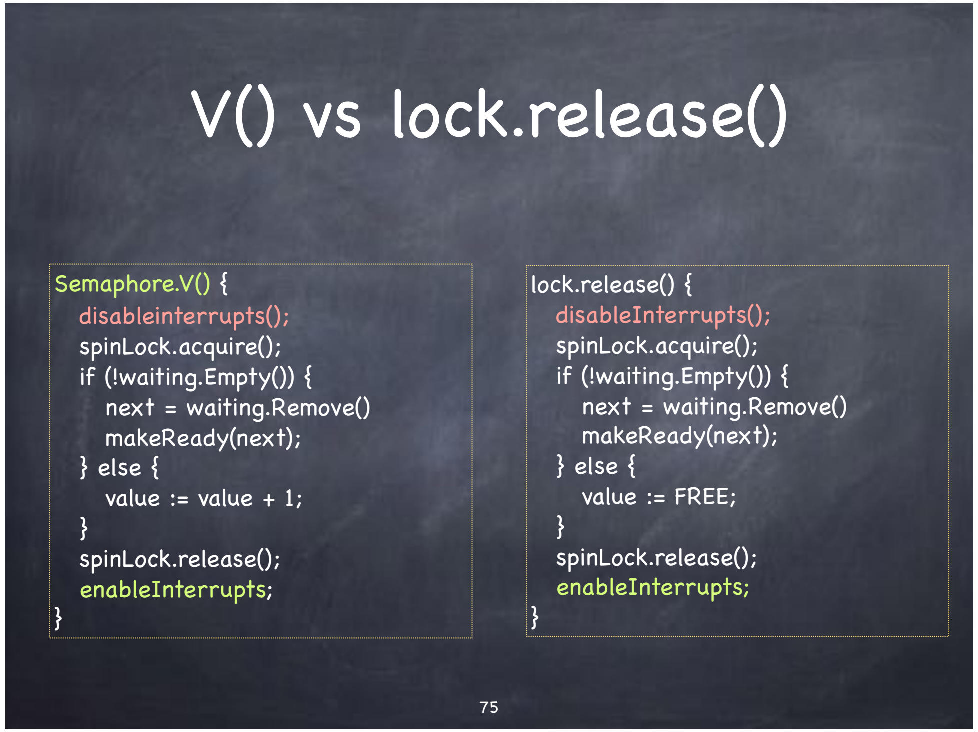
```
Semaphore.V() {
   disableinterrupts();
   spinLock.acquire();
   if (!waiting.Empty()) {
      next = waiting.Remove()
      makeReady(next);
   } else {
      value := value + 1;
   }
   spinLock.release();
   enableInterrupts;
}
```



P() vs lock.acquire()

```
Semaphore.P() {
   disableinterrupts();
   spinLock.acquire();
   if (value == 0) {
      waiting.Add(current);
      suspend(&spinlock);
   } else {
      value := value - 1;
   }
   spinLock.release();
   enableInterrupts();
}
```

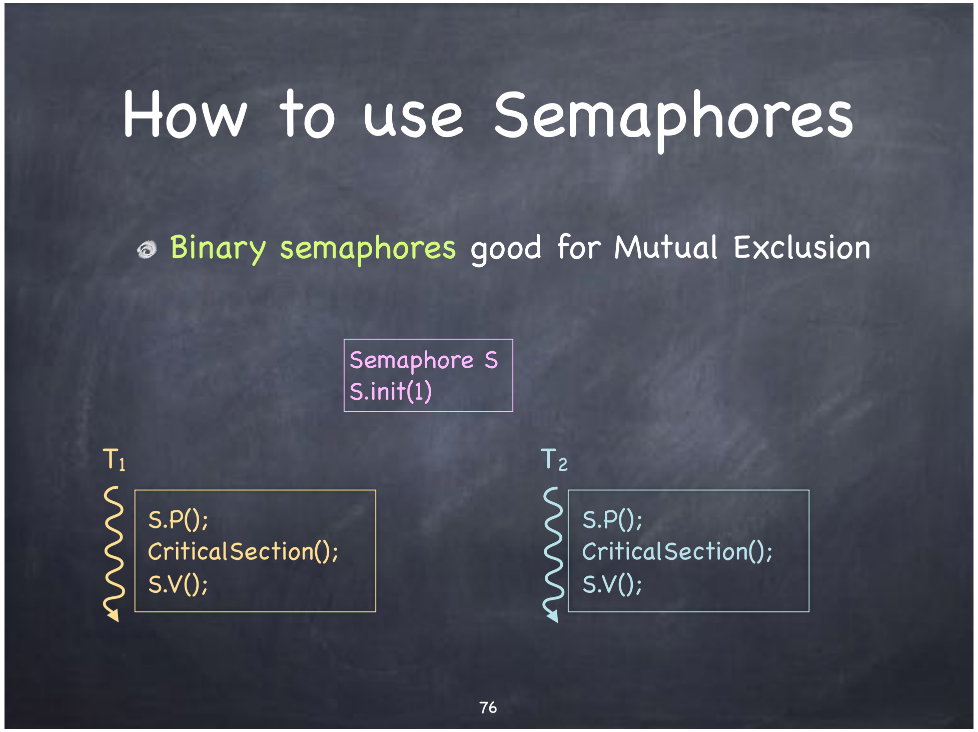
```
lock.acquire() {
   disableInterrupts();
   spinLock.acquire();
   if (value == BUSY) {
      waiting.Add(current);
      suspend(&spinlock);
   } else {
      value = BUSY;
   }
   spinLock.release();
   enableInterrupts();
}
```



V() vs lock.release()

```
Semaphore.V() {
   disableinterrupts();
   spinLock.acquire();
   if (!waiting.Empty()) {
      next = waiting.Remove()
      makeReady(next);
   } else {
      value := value + 1;
   }
   spinLock.release();
   enableInterrupts;
}
```

```
lock.release() {
   disableInterrupts();
   spinLock.acquire();
   if (!waiting.Empty()) {
      next = waiting.Remove()
      makeReady(next);
   } else {
      value := FREE;
   }
   spinLock.release();
   enableInterrupts;
}
```



How to use Semaphores

- Binary semaphores good for Mutual Exclusion

```
Semaphore S
S.init(1)
```

T_1

```
S.P();
CriticalSection();
S.V();
```

T_2

```
S.P();
CriticalSection();
S.V();
```

How to use Semaphores

- Counting semaphores good for signaling or counting resources
  - One thread performs P() to await an event
  - Another thread performs V() to inform waiting thread that event has occurred

```
Semaphore packetarrived
packetarrived.init(0)
```

T1

```
pkt := getpacket();
enqueue(pcketq, pkt);
packjetarrived.V()
```

T2

```
packetarrived.P();
pkt := dequeue(packetq)
print(pkt);
```

Producer-Consumer with Bounded Buffer

- A set of producer and consumer threads communicate through a buffer of size N
  - producer inserts resources into the buffer (writes to "in" and moves right)
    - disk blocks, output, memory pages, characters...
  - consumer removes resources from the buffer (reads from out and moves right)
- Producer and consumer execute at different rates

Safety

- Sequence of consumed values is a prefix of the sequence of produced values
- Let
  - nc = number consumed
  - np = number produced
  - N = size of buffer, then maintain the following invariant:

$$0 \leq np - nc \leq \text{N}$$

How to go about this problem

- Are there shared variables? If so, we'll need to make sure the code accessing them is in a critical section
  - variable in (shared by producers)
  - variable out (shared by consumers)
  - the buffer (shared by all)
- How many locks we need?

Step 1: Guard Shared Resources

```
Shared:
int buf[N];
int in := 0, out := 0;
lock: in_lock, out_lock
```

Invariant

$0 \le np - nc \le N$

```
// add item to buffer
void produce(int item) {
  in_lock.acquire();
  buf[in] := item;
  in := (in+1)%N
  in_lock.release();
}
```

```
// remove item from buffer
int consume() {
  out_lock.acquire();
  int item := buf[out];
  out := (out+1)%N;
  out_lock.release();
  return(item);
}
```

Step 1: Guard Shared Resources*

*with Semaphores

```
Shared:
int buf[N];
int in := 0, out := 0;
lock: in_lock, out_lock
```

Implement mutual exclusion with a binary semaphore initialized to 1

```
// add item to buffer
void produce(int item) {
  in_lock.acquire();
  buf[in%N] := item;
  in := in+1;
  in_lock.release();
}
```

```
// remove item from buffer
int consume() {
  out_lock.acquire();
  int item := buf[out%N];
  out := out+1;
  out_lock.release();
  return(item);
}
```

Step 1: Guard Shared Resources*

*with Semaphores

```
Shared:
int buf[N];
int in := 0, out := 0;
Semaphore mutex_in(1), mutex_out(1);
```

Implement mutual exclusion with a binary semaphore initialized to 1

```
// add item to buffer
void produce(int item) {
  in_lock.acquire();
  buf[in%N] := item;
  in := in+1;
  in_lock.release();
}
```

```
// remove item from buffer
int consume() {
  out_lock.acquire();
  int item := buf[out%N];
  out := out+1;
  out_lock.release();
  return(item);
}
```

Step 1: Guard Shared Resources*

*with Semaphores

```
Shared:
int buf[N];
int in := 0, out := 0;
Semaphore mutex_in(1), mutex_out(1);
```

Implement mutual exclusion with a binary semaphore initialized to 1

```
// add item to buffer
void produce(int item) {
  mutex_in.P();
  buf[in%N] := item;
  in := in+1;
  mutex_in.V();
}
```

```
// remove item from buffer
int consume() {
  out_lock.acquire();
  int item := buf[out%N];
  out := out+1;
  out_lock.release();
  return(item);
}
```

Step 1: Guard Shared Resources*

*with Semaphores

```
Shared:
int buf[N];
int in := 0, out := 0;
Semaphore mutex_in(1), mutex_out(1);
```

Implement mutual exclusion with a binary semaphore initialized to 1

```
// add item to buffer
void produce(int item) {
  mutex_in.P();
  buf[in%N] := item;
  in := in+1;
  mutex_in.V();
}
```

```
// remove item from buffer
int consume() {
  mutex_out.P();
  int item := buf[out%N];
  out := out+1;
  mutex_out.V();
  return(item);
}
```

Step 1: Coordinate Actions

```
Shared:
int buf[N];
int in := 0, out := 0;
Semaphore mutex_in(1), mutex_out(1);
```

Need a full buffer entry to remove an item; and an empty one to add an item

```
// add item to buffer
void produce(int item) {
  mutex_in.P();
  buf[in%N] := item;
  in := in+1;
  mutex_in.V();
}
```

```
// remove item from buffer
int consume() {
  mutex_out.P();
  int item := buf[out%N];
  out := out+1;
  mutex_out.V();
  return(item);
}
```

Step 1: Coordinate Actions

```
Shared:
int buf[N];
int in := 0, out := 0;
Semaphore mutex_in(1), mutex_out(1);
Condition empty, full;
```

Need a full buffer entry to remove an item; and an empty one to add an item

```
// add item to buffer
void produce(int item) {
  wait(empty);
  mutex_in.P();
  buf[in%N] := item;
  in := in+1;
  mutex_in.V();
  signal(full);
}
```

```
// remove item from buffer
int consume() {
  wait(full);
  mutex_out.P();
  int item := buf[out%N];
  out := out+1;
  mutex_out.V();
  signal(empty);
  return(item);
}
```

Step 1: Coordinate Actions*

*with Semaphores

```
Shared:
int buf[N];
int in := 0, out := 0;
Semaphore mutex_in(1), mutex_out(1);
Condition empty, full;
```

Use two counting semaphores: one to count empty entries, one to count full

```
// add item to buffer
void produce(int item) {
  wait(empty);
  mutex_in.P();
  buf[in%N] := item;
  in := in+1;
  mutex_in.V();
  signal(full);
}
```

```
// remove item from buffer
int consume() {
  wait(full);
  mutex_out.P();
  int item := buf[out%N];
  out := out+1;
  mutex_out.V();
  signal(empty);
  return(item);
}
```

Step 1: Coordinate Actions*

*with Semaphores

```
Shared:
int buf[N];
int in := 0, out := 0;
Semaphore mutex_in(1), mutex_out(1);
Semaphore empty(N), full(0);
```

Use two counting semaphores: one to count empty entries, one to count full

```
// add item to buffer
void produce(int item) {
  wait(empty);
  mutex_in.P();
  buf[in%N] := item;
  in := in+1;
  mutex_in.V();
  signal(full);
}
```

```
// remove item from buffer
int consume() {
  wait(full);
  mutex_out.P();
  int item := buf[out%N];
  out := out+1;
  mutex_out.V();
  signal(empty);
  return(item);
}
```

Step 1: Coordinate Actions*

*with Semaphores

```
Shared:
int buf[N];
int in := 0, out := 0;
Semaphore mutex_in(1), mutex_out(1);
Semaphore empty(N), full(0);
```

Use two counting semaphores: one to count empty entries, one to count full

```
// add item to buffer
void produce(int item) {
  empty.P();
  mutex_in.P();
  buf[in%N] := item;
  in := in+1;
  mutex_in.V();
  signal(full);
}
```

```
// remove item from buffer
int consume() {
  wait(full)
  mutex_out.P();
  int item := buf[out%N];
  out := out+1;
  mutex_out.V();
  signal(empty);
  return(item);
}
```

Step 1: Coordinate Actions*

*with Semaphores

```
Shared:
int buf[N];
int in := 0, out := 0;
Semaphore mutex_in(1), mutex_out(1);
Semaphore empty(N), full(0);
```

Use two counting semaphores: one to count empty entries, one to count full

```
// add item to buffer
void produce(int item) {
  empty.P();
  mutex_in.P();
  buf[in%N] := item;
  in := in+1;
  mutex_in.V();
  full.V();
}
```

```
// remove item from buffer
int consume() {
  wait(full)
  mutex_out.P();
  int item := buf[out%N];
  out := out+1;
  mutex_out.V();
  signal(empty);
  return(item);
}
```

Step 1: Coordinate Actions*

*with Semaphores

```
Shared:
int buf[N];
int in := 0, out := 0;
Semaphore mutex_in(1), mutex_out(1);
Semaphore empty(N), full(0);
```

Use two counting semaphores: one to count empty entries, one to count full

```
// add item to buffer
void produce(int item) {
  empty.P();
  mutex_in.P();
  buf[in%N] := item;
  in := in+1;
  mutex_in.V();
  full.V();
}
```

```
// remove item from buffer
int consume() {
  full.P();
  mutex_out.P();
  int item := buf[out%N];
  out := out+1;
  mutex_out.V();
  empty.V();
  return(item);
}
```

Musings on Producer/Consumer

- We used two semaphores because we used two different variables (in & out) accessed solely by producers and consumers respectively
  - if we used variables changed by both producers and consumers, we would have have had to use a single semaphore
    - sacrificing concurrency
- Extracting more concurrency increases complexity
  - only do so if the return in performance is worth it!

```
Shared:
int buf[N];
int in := 0, out := 0;
Semaphore mutex_in(1), mutex_out(1);
Semaphore empty(N), full(0);
```

```
// remove item from buffer
int consume() {
  full.P();
  mutex_out.P();
  int item := buf[out%N];
  out := out+1;
  mutex_out.V();
  empty.V();
  return(item);
}
```

```
// add item to buffer
void produce(int item) {
  empty.P();
  mutex_in.P();
  buf[in%N] := item;
  in := in+1;
  mutex_in.V();
  full.V();
}
```

Step 1: Coordinate Actions*

*with Semaphores

Is there a V for every P?

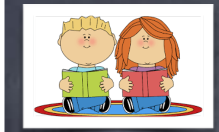
```
Shared:
int buf[N];
int in := 0, out := 0;
Semaphore mutex_in(1), mutex_out(1);
Semaphore empty(N), full(0);
```

Are mutexes initialized to 1?

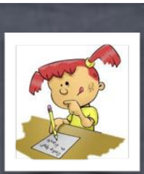
```
// add item to buffer
void produce(int item) {
  empty.P();
  mutex_in.P();
  buf[in] := item;
  in := (in+1)%N;
  mutex_in.V();
  full.V();
}
```

```
// remove item from buffer
int consume() {
  full.P();
  mutex_out.P();
  item := buf[out];
  out := (out+1)%N;
  mutex_out.V();
  empty.V();
  return(item);
}
```

Do mutexes P&V in the same thread?



Readers-Writers



- Models access to an object (e.g., a database), shared among several threads
  - some threads only read the object
  - others only write it
- Safety

$$(\#r \geq 0) \wedge (0 \leq \#w \leq 1) \wedge (\#r > 0) \Rightarrow (\#w = 0))$$

Fairness questions

- Suppose a writer is active, and a combination of readers and writers arrive
  - Who should get in next?
- Suppose that a writer is waiting, and an endless stream of readers arrives
  - Who should get in next?

Readers-Writers Solution

```
Shared:
int rcount = 0;
Semaphore rcount_mutex (1);
Semaphore rOw_lock(1);
```

```
void write() {
  rOw_lock.P();
  …
  /* Perform write */
  …
  rOw_lock.V();
 }
```

```
int read() {
  rcount_mutex.P();
  rcount := rcount+1;
  if (rcount = 1) then
    rOw_lock.P();
  rcount_mutex.V();
  …
  /* Perform read */
  …
  rcount_mutex.P();
  rcount := rcount-1;
  if (rcount = 0) then
    rOw.lock.V();
  rcount_mutex.V();
}
```

if I am the first reader, P() to enforce invariant

if I am the last reader, V() to indicate CS is empty

Musings on Readers/Writers

- Semaphore rOw provides mutex between readers and writers
  - writers always rOw.P() / rOw.V()
  - readers do so only when rcount transitions from 0 to 1 or from 1 to 0
- If a writer is writing, where are readers waiting?
- Once a writer exits, all readers can fall through
  - Which reader gets to go first?
  - Are all readers guaranteed to fall through?

```
int read() {
  rcount_mutex.P();
  rcount := rcount+1;
  if (rcount == 1) then
    rOw_lock.P();
  rcount_mutex.V();
  …
  /* Perform read */
  …
  rcount_mutex.P();
  rcount := rcount-1;
  if (rcount == 0) then
    rOw.lock.V();
  rcount_mutex.V();
}
```

```
void write() {
  rOw_lock.P();
  …
  /* Perform write */
  …
  rOw_lock.V();
}
```

```
Shared:
int rcount = 0;
Semaphore rcount_mutex (1)
Semaphore rOw_lock(1);
```

More Musings on Readers/Writers

- If readers and writers are waiting, and a writer exits, who goes first?
- Why do readers use a mutex?
- Why don't writers use a mutex?

```
int read() {
  rcount_mutex.P();
  rcount := rcount+1;
  if (rcount == 1) then
    rOw_lock.P();
  rcount_mutex.V();
  …
  /* Perform read */
  …
  rcount_mutex.P();
  rcount := rcount-1;
  if (rcount == 0) then
    rOw.lock.V();
  rcount_mutex.V();
}
```

```
void write() {
  rOw_lock.P();
  …
  /* Perform write */
  …
  rOw_lock.V();
}
```

```
Shared:
int rcount = 0;
Semaphore rcount_mutex (1)
Semaphore rOw_lock(1);
```

Classic Mistakes with Semaphores

T_i
```
P(S)
CS
P(S)
```

T_i stuck on 2nd P(). Subsequent processes hopelessly pile on 1st P()

T_j
```
V(S)
CS
V(S)
```

Undermines mutex:
- T_j does not get permission via P()
- "extra" V() allows other processes into CS inappropriately

T_l
```
P(S)
if (x) return;
CS
V(S)
```

Conditional code can change code flow in the CS. Caused by code updates (bug fixes, etc.) by someone other than original author of code.