TAS behavior: surprise!

TAS

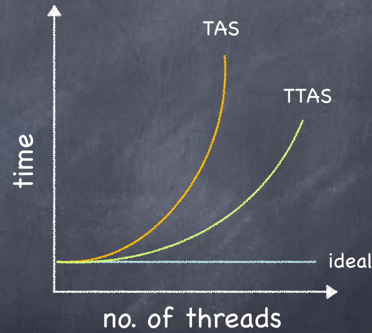
```
lock.acquire() { while (TAS(value)); }

lock.release() { value := 0 }
```

TTAS

```
lock.acquire() {
    while (true) {
      while (value);
      if (!TAS(value))
        return;
    }
}

lock.release() { value := 0 }
```



no. of threads

Why are TAS and TTAS performing poorly?

Why is TTAS performing much better than TAS?

Cache effects

- Simple TAS
  - all TAS go to bus
    - everybody waits
  - each TAS invalidates cached copies of lock
    - every TAS writes "1" blindly
    - every thread must use bus to retrieve new lock value even if unchanged
    - lots of bus traffic!
  - lock release may be delayed
    - by spinner hogging the bus

Cache effects

- TTAS
  - Suppose T_2 reads a lock held by T_1
    - cache miss the first time
    - but, as long as T_1 has the lock, all reads are from the cache
  - When T_1 releases lock
    - all cached copies of lock are invalidated
    - all spinning thread reread lock value
    - call TAS at about same time
    - burst of bus traffic, before quiescence

SEMAPHORES

N U J V

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")
  - V(): increment Verhoog ("+1")

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
int guard := 0 }

```
Semaphore.P() {
   while (TAS(guard)) ;
   if (value == 0) {
      Put on queue of threads
      waiting for lock;
      Set guard to 0;
      Go to sleep;
   } else {
      value := value - 1;
      guard := 0
   }
}
```

```
Semaphore.V() {
   while (TAS(guard)) ;
   if anyone on wait queue {
      Take a waiting thread off lock's
      wait queue and put it at the
      front of ready queue;
   } else {
      value := value + 1;
   }
   guard := 0
}
```


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)
```

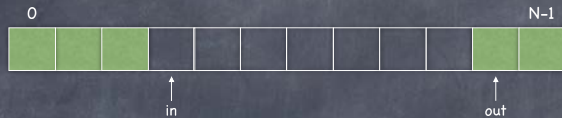
T_1

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

T_2

```
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:

$$0 \leq np - nc \leq N$$

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

$$0 \leq np - nc \leq N$$

```
// 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

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

Invariant

$0 \leq np - nc \leq N$

```
// 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;
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);
}
```

Step 1: Coordinate Actions*

*with Semaphores

Is there a V for every P?

Are mutexes initialized to 1?

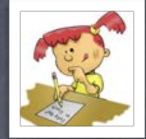
Do mutexes P&V in the same thread?

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

```
// 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);
}
```

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();
}
```

Musings on Readers/Writers

- Semaphore rOw provides mutex between readers and writers
  - writers always rOw.P() / rOw.V()
  - readers 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?
- What if we move rcount_mutex.V() just above if (rcount = 1)?

```
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);
```