Semaphores and other Wait-and-Signal mechanisms

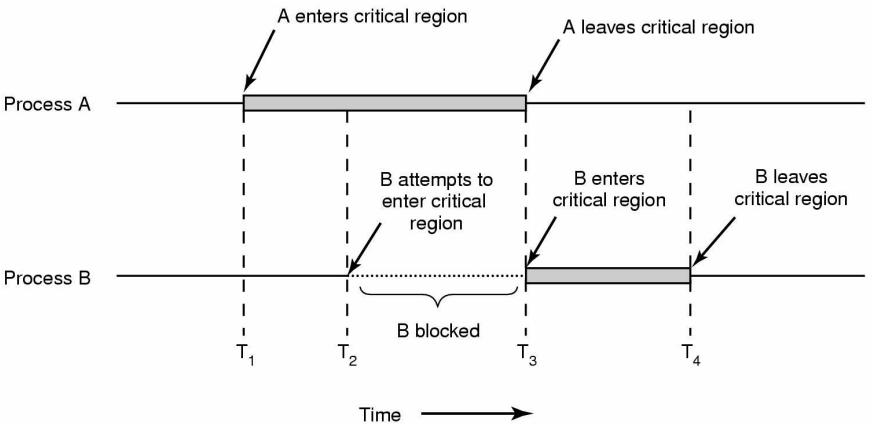
Carsten Griwodz
University of Oslo
(including slides by Otto Anshus and Kai Li)

Critical Regions

Four conditions to provide mutual exclusion

1. No two threads simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No thread running outside its critical region may block another thread
4. No thread must wait forever to enter its critical region

Critical Regions



Mutual exclusion using critical regions

Recall Processes and Threads

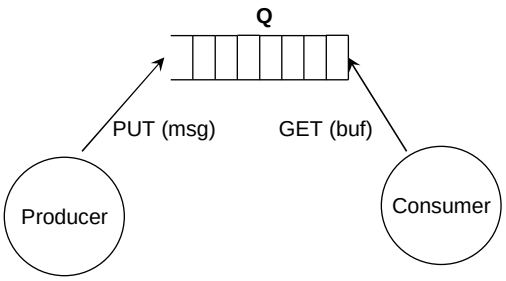
- Process
  - Address space
  - Program text and data
  - Open files
  - Child process IDs
  - Alarms
  - Signal handlers
  - Accounting information
- Implemented in kernel

- Thread
  - Program counter
  - Registers
  - Stack
- Implemented in kernel or in user space
- Threads are the scheduled entities

Producer-Consumer Problem

- Main problem description
  - Two threads
  - Different actions in the critical region
  - The consumer can not enter the CR more often than the producer
- Two sub-problems
  - Unbounded PCP: the producer can enter the CR as often as it wants
  - Bounded PCP: the producer can enter the CR only *N* times more often than the consumer

Unbounded PCP



Rules for the queue Q:

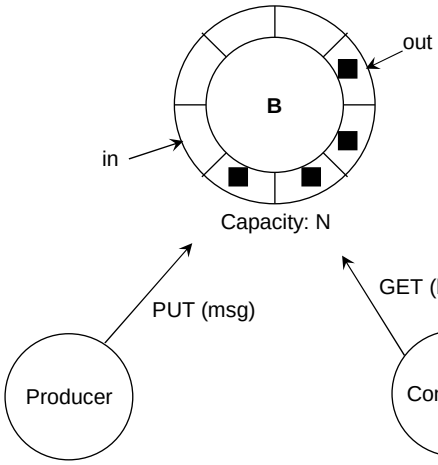
•No Get when empty

•Q shared, so must have mutex between Put and Get

Recall Mutexes

- Can be acquired and released
  - Only one thread can hold one mutex at a time
  - A second thread trying to acquire must wait
- Mutexes
  - Can be implemented using busy waiting
  - Simpler with advanced atomic operations
    - Disable interrupts, TSL, XCHG, …
  - Still many approaches using busy waiting
  - Better implemented using system calls block & unblock

Bounded PCP



Rules for the buffer B:

- No Get when empty
- No Put when full
- B shared, so must have mutex between Put and Get

Mutex Solution

Unbounded PCP

```
Put(msg) {
  acquire(mutex);
  <put>
  release(mutex);
}
```

```
Get(msg) {
  acquire(mutex);
  while(empty) {
    release(mutex);
    acquire(mutex);
  }
  <get>
  release(mutex);
}
```

Busy waiting

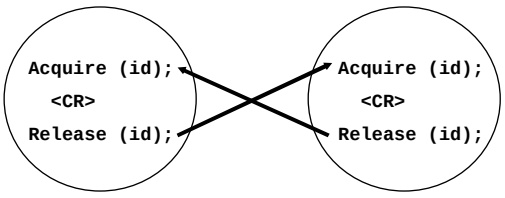
Bounded PCP

```
Put(msg) {
  acquire(mutex);
  while(full) {
    release(mutex);
    acquire(mutex);
  }
  <put>
  release(mutex);
}
```

```
Get(msg) {
  acquire(mutex);
  while(empty) {
    release(mutex);
    acquire(mutex);
  }
  <get>
  release(mutex);
}
```

Two Kinds of Synchronization

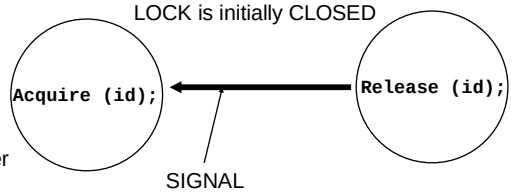
LOCK is initially OPEN



MUTEX

Acquire will let first caller through, and then block next until Release

LOCK is initially CLOSED



CONDITION SYNCHRONIZATION

Acquire will block first caller until Release

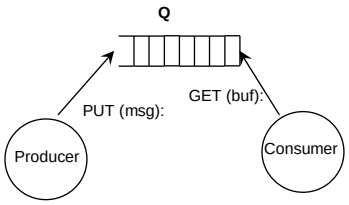
Sleep and Wakeup / Signal and Wait

- Wait (cond)
  - Insert(caller, cond_queue)
  - Block this thread
- Signal (cond)
  - Unblock first in cond_queue, or just return if empty

No counting, unused signals are ignored

Wait is atomic

Unbounded PCP using Signal and Wait



Rules for the queue Q:

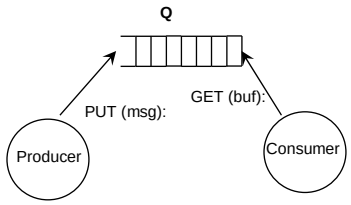
•No Get when empty

•Q shared, so must have mutex between Put and Get

```
while(1) {
  <process>
  acquire(mutex);
   <insert>
  release(mutex);
  signal(cond);
}
```

```
while(1) {
  if(empty)
    wait(cond);
  acquire(mutex);
   <remove>
  release(mutex);
  <process>
}
```

Unbounded PCP using Signal and Wait

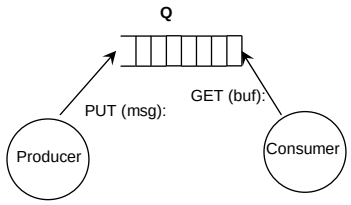



```
while(1) {
  <process>
  acquire(mutex);
    <insert>
  release(mutex);
  signal(cond);
}
```

```
while(1) {
  if(empty)
    wait(cond);
  acquire(mutex);
    <remove>
  release(mutex);
  <process>
}
```

Lost signal

Unbounded PCP using Signal and Wait

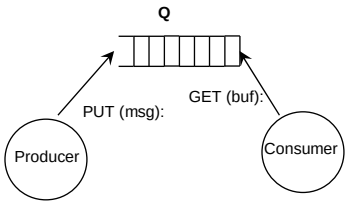



```
while(1) {
  <process>
  acquire(mutex);
   <insert>
  signal(cond);
  release(mutex);
}
```

Producer can't enter

```
while(1) {
  acquire(mutex);
   while(empty) {
     wait(cond);
     release(mutex);
     acquire(mutex);
   }
   <remove>
  release(mutex);
  <process>
}
```

Unbounded PCP using Signal and Wait




```
while(1) {
  <process>
  acquire(mutex);
   <insert>
  signal(cond);
  release(mutex);
}
```

```
while(1) {
  acquire(mutex);
   while(empty) {
     release(mutex);
     wait(cond);
     acquire(mutex);
   }
   <remove>
  release(mutex);
  <process>
}
```

Lost signal

Threads wait for …

- Access to a critical region
  - Mutex
  - Semaphore
- A condition to be fulfilled
  - Condition variable
  - Barrier
  - Semaphore

Semaphores

Semaphores (Dijkstra, 1965)

- Down or Wait or “P” (prolaag)
  - Atomic
  - Decrement semaphore value by 1
  - Block if not positive
- Up or Signal or “V” (verhoog)
  - Atomic
  - Increment semaphore by 1
  - Wake up a waiting thread if any

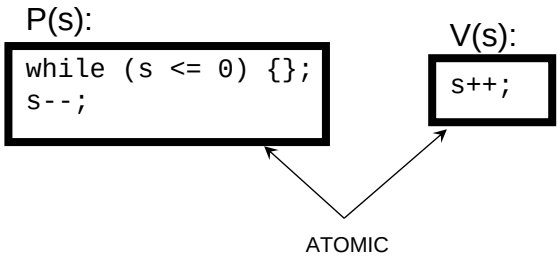
```
P(s) {
  if (--s < 0)
    Block(s);
}
```

```
V(s) {
  if (++s <= 0)
    Unblock(s);
}
```

Can get negative s: counts number of waiting threads

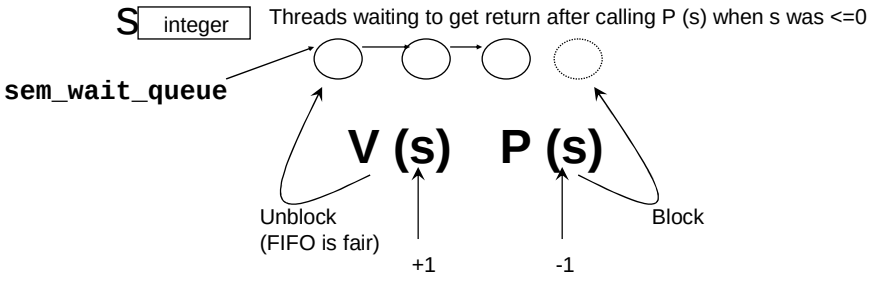
s is NOT accessible through other means than calling P and V

Semaphores w/Busy Wait



- Starvation possible?
- Does it matter in practice?

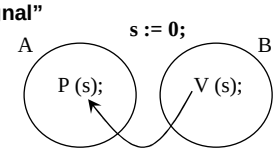
The Structure of a Semaphore



- Atomic: Disable interrupts
- Atomic: P() and V() as System calls
- Atomic: Entry-Exit protocols

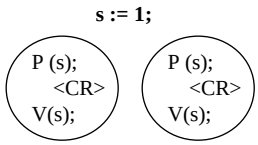
Using Semaphores

"The Signal"



A blocks until B says V

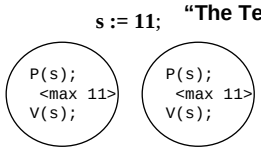
"The Mutex"



One thread gets in, next blocks until V is executed

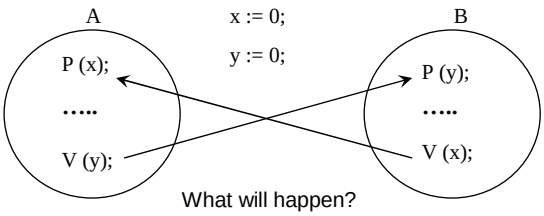
NB: remember to set the initial semaphore value!

"The Team"



Up to 11 threads can pass P, the ninth will block until V is said by one of the eight already in there

Simple to debug?

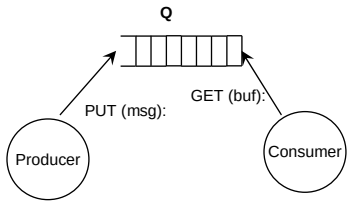


What will happen?

THEY ARE FOREVER WAITING FOR EACH OTHERS SIGNAL
("No milk")

Examples

Unbounded PCP using Semaphores



Rules for the queue Q:

- No Get when empty
- Q shared, so must have mutex between Put and Get

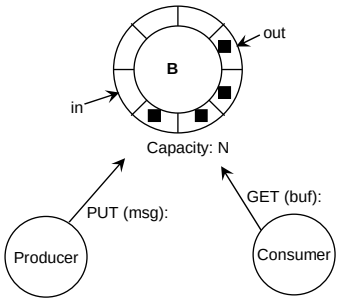
One semaphore for each condition we must wait for to become TRUE:

- Q empty: nonempty:=0;
- Q mutex: mutex:=1;

```
PUT (msg):              GET (buf):
  P(mutex);               P(nonempty);
   <insert>                P(mutex);
  V(mutex);                 <remove>
 V(nonempty);             V(mutex);
```

- Is Mutex needed when only 1 P and 1 C?
- PUT at one end, GET at other end

Bounded PCP using Semaphores



Rules for the buffer B:

- No Get when empty
- No Put when full
- B shared, so must have mutex between Put and Get

One semaphore for each condition we must wait for to become TRUE:

- B empty: nonempty:=0;
- B full: nonfull:=N
- B mutex: mutex:=1;

```
PUT (msg):
 P(nonfull);
  P(mutex);
   <insert>
  V(mutex);
 V(nonempty);
```

```
GET (buf):
 P(nonempty);
  P(mutex);
   <remove>
  V(mutex);
 V(nonfull);
```

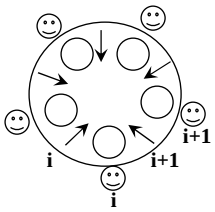
- PUT at one end, GET at other end

Dining Philosophers Problem

- Five philosopher
- Five dishes
- Five forks
- But a philosopher needs two forks for eating

- Usually the philosophers think, when they are hungry the try to eat
- How to prevent all philosophers from starving

Dining Philosophers



•Each: 2 forks to eat

•5 philosophers: 10 forks to let all eat concurrently

•5 forks: 2 can eat concurrently

Mutex on whole table:
•1 can eat at a time

```
P(mutex);
  eat;
V(mutex);
```

T_i

Get L; Get R;
•Deadlock possible

S(i) = 1 initially

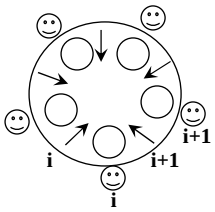
```
P(s(i));
  P(s(i+1));
    eat;
  V(s(i+1));
V(s(i));
```

T_i

Get L; Get R if free else Put L;
•Starvation possible

T_i

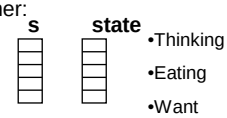
Dining Philosophers



To avoid starvation they could look after each other:

•**Entry**: If L and R is not eating we can

•**Exit**: If L (R) wants to eat and L.L (R.R) is not eating we start him eating



S() = 0 initially

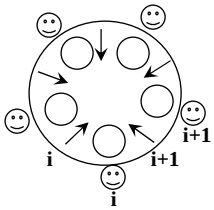
T_i

```
While (1) {
  <think>
  ENTRY;
    <eat>
  EXIT;
}
```

```
P(mutex);
  state(i):=Want;
  if (state(i-1) !=Eating AND state(i+1) != Eating)
  {/*Safe to eat*/
    state(i):=Eating;
    V(s(i));   /*Because */    }
V(mutex);
P(s(i)); /* Init was 0!!
            We or neighbor must say V(i) to us!*/
```

```
P(mutex);
  state(i):=Thinking;
  if (state(i-1)=Want AND state(i-2)
!=Eating)
  {
    state(i-1):=Eating;
    V(s(i-1));  /*Start Left neighbor*/
  }
/*Analogue for Right neighbor*/
V(mutex);
```

Dining Philosophers



S(i) = 1 initially

Can we in a simple way do better than this one?

Get L; Get R;
•Deadlock possible

```
P(s(i));
  P(s(i+1));
    eat;
  V(s(i+1));
V(s(i));
```

•Non-symmetric solution. Still quite elegant

•Remove the danger of circular waiting (deadlock)

•T1-T4: Get L; Get R;

•T5: Get R; Get L;

T_1, T_2, T_3, T_4:

```
P(s(i)):
  P(s(i+1));
    <eat>
  V(s(i+1));
V(s(i));
```

T_5

```
P(s(1));
  P(s(5));
    <eat>
  V(s(5));
V(s((1));
```

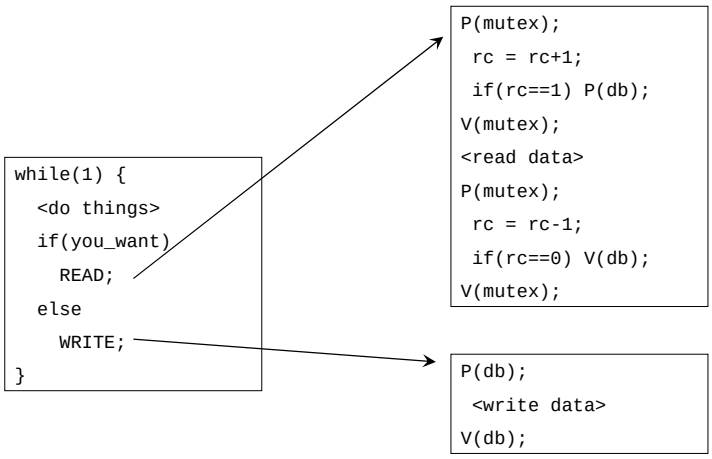
Readers and Writers Problem

- Several threads
- Shared data in a critical region

- Sometimes a thread wants to read the data
- Sometimes a thread wants to change the data

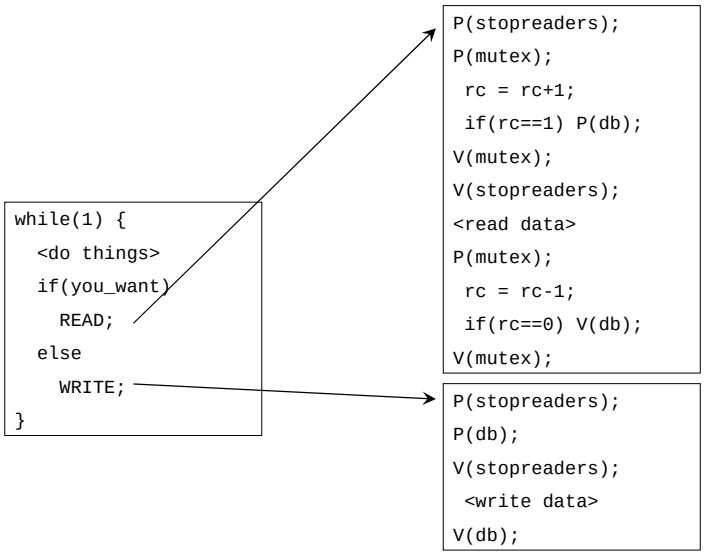
- Readers can enter a critical region together
- Writers can not enter a critical region together

The Readers and Writers Problem



One solution to the readers and writers problem
But too many readers can starve writers

The Readers and Writers Problem



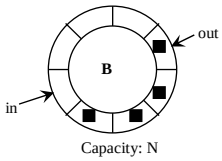
Another solution to the readers and writers problem

Other wait-and-signal mechanisms

Event Count (Reed 1977)

- **`Init( ec )`**
  - Set the eventcount to 0
- **`Read( ec )`**
  - Return the value of eventcount **`ec`**
- **`Advance( ec )`**
  - Atomically increment **`ec`** by 1
- **`Await( ec, v )`**
  - Wait until **`ec >= v`**

Bounded PCP with Event Count



in=out=0;

```
producer() {
  int next = 0;

  while (1) {
    produce an item
    next++;
    await(out, next - N);
    put the item in buffer;
    advance(in);
  }
}
```

```
consumer() {
  int next = 0;

  while ( 1 ) {
    next++;
    await(in, next);
    take an item from buffer;
    advance(out);
    consume the item;
  }
}
```

- Does this work for more than one producer and consumer?
- No, we will get multiple events happening, need a sequencer

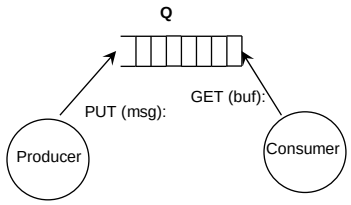
Condition Variables

- Wait (cond, mutex)
  - Insert(caller, cond_queue)
  - V(mutex)
  - Block this thread
  - When unblocked, P(mutex)
- Signal (cond)
  - Unblock first in cond_queue, or just return if empty

No counting, unused signals are ignored

Insert, Unlock and Block not interrupted

Unbounded PCP using Condition Variable

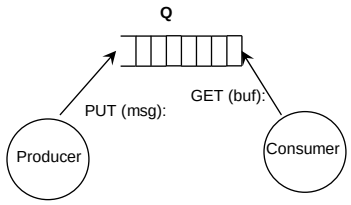



```
while(1) {
  <process>
  P(mutex);
   <insert>
  signal(cond);
  V(mutex);
}
```

No problems

```
while(1) {
  P(mutex);
   while(empty) {
     wait(cond,mutex);
   }
   <remove>
  V(mutex);
  <process>
}
```

Unbounded PCP using Condition Variable




```
while(1) {
  <process>
  P(mutex);
   <insert>
  signal(cond);
  V(mutex);
}
```

No problems either

```
while(1) {
  P(mutex);
   while(empty) {
     wait(cond,mutex);
   }
   <remove>
  V(mutex);
  <process>
}
```

```
while(1) {
  P(mutex);
   while(empty) {
     wait(cond,mutex);
   }
   <remove>
  V(mutex);
  <process>
}
```

Emulations

- Not all wait-and-signal mechanisms exist in all operating systems or thread packages
- Windows has no native condition variables
  - But semaphores (and mutexes)
- Some Unix-like systems have no native semaphores
  - But condition variables and mutexes
- Emulations

Building Condition Variables using Semaphores

```
cond:
  semaphore lock    = 1
  semaphore signal  = 0
  int       waiters = 0
```

```
wait(cond,mutex) {
  P(cond.lock);
  cond.waiters+=1;
  V(cond.lock);
  V(mutex);
  P(cond.signal);
  P(cond.lock);
  cond.waiters-=1;
  V(cond.lock);
  P(mutex);
}
```

```
signal(cond) {
  P(cond.lock);
  if(cond.waiters>0) {
    V(cond.signal);
    V(cond.unlock);
  } else {
    V(cond.unlock);
  }
```

But no lost signal because of cond.waiters & counting in semaphores

Looks like lost-signal situation in signal-and-wait

Condition Variables Extension

- Wait (cond, mutex)
  - Insert(caller, cond_queue)
  - V(mutex)
  - Block this thread
  - When unblocked, P(mutex)
- Signal (cond)
  - Unblock first in cond_queue, or just return if empty
- Broadcast (cond)
  - Unblock all in cond_queue, or just return if empty

Building Condition Variables using Semaphores

```
cond:
  semaphore lock    = 1
  semaphore signal  = 0
  int       waiters = 0
```

```
wait(cond,mutex) {
  P(cond.lock);
  cond.waiters+=1;
  V(cond.lock);
  V(mutex);
  P(cond.signal);
  P(cond.lock);
  cond.waiters-=1;
  V(cond.lock);
  P(mutex);
}
```

```
broadcast(cond) {
  P(cond.lock);
  if(cond.waiters>0) {
    for(i=0;i<cond.waiters;i++)
      V(cond.signal);
    V(cond.unlock);
  } else {
    V(cond.unlock);
  }
}
```

Condition Variables Extension II

- Wait (cond, mutex)
  - Insert(caller, cond_queue)
  - V(mutex)
  - Block this thread
  - When unblocked, P(mutex)
- Wait(cond,mutex,timeout)
  - Wait no longer than timeout
- Signal (cond)
  - Unblock first in cond_queue, or just return if empty
- Broadcast (cond)
  - Unblock all in cond_queue, or just return if empty

This needs additional scheduler support

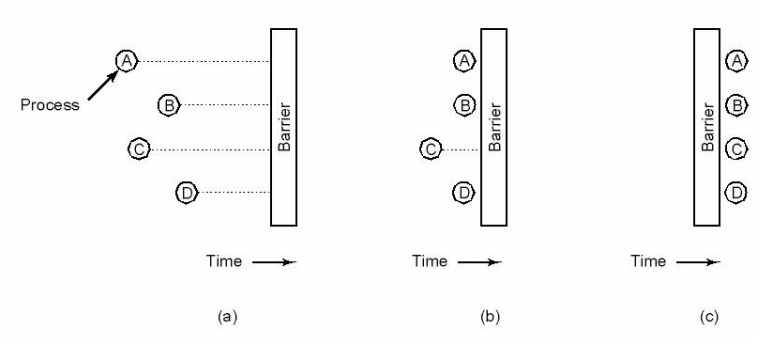
Building Semaphores using Condition Variables and Mutexes

```
semaphore:
  mutex mutex
  cond  cond
  int   val   = <initial sempahore value>
```

```
V(sem) {
  acquire(sem.mutex);
  sem.val += 1;
  if(sem.val <= 0)
    signal(sem.cond);
  release(sem.mutex);
}
```

```
P(sem) {
  acquire(sem.mutex);
  sem.val -= 1;
  if(sem.val < 0)
    wait(sem.cond,sem.mutex);
  release(sem.mutex);
}
```

Barriers



- Use of a barrier
  - threads approaching a barrier
  - all threads but one blocked at barrier
  - last thread arrives, all are let through

Threads wait for …

- Access to a critical region
  - Mutex
  - Semaphore
- A condition to be fulfilled
  - Condition variable
  - Barrier
  - Semaphore