Semaphores

Silvia Lizeth Tapia Tarifa, Riccardo Sieve

September 1st, 2026

University of Oslo

Overview

Last lecture: Locks and Barriers

- Complex techniques
- No clear separation between variables for synchronization and variables for computation
- Busy waiting

Overview

Last lecture: Locks and Barriers

- Complex techniques
- No clear separation between variables for synchronization and variables for computation
- Busy waiting

This lecture: Semaphores

- Synchronization tool
- Used easily for mutual exclusion and condition synchronization
- A way to implement signaling and scheduling
- Implementable in many ways on hardware (`CMPXCHG`)
- Available in programming language libraries and OS

Outline

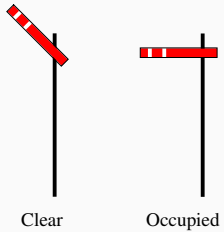
- Semaphores: Syntax and semantics
- Synchronization examples:
  - Mutual exclusion (critical sections)
  - Barriers (signaling events)
  - Producers and consumers (split binary semaphores)
  - Bounded buffer: resource counting
  - Variation of the Dining philosophers: mutual exclusion – deadlock
  - Readers and writers:
    - condition synchronization
    - passing the baton

Semaphores

Semaphores

Origins of Term

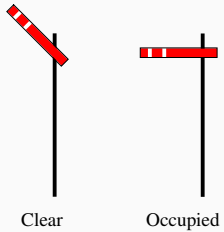
- Introduced by Dijkstra in 1968
- Inspired by railroad traffic synchronization
- Railroad semaphore indicates whether the track ahead is clear or occupied by another train



Semaphores

Origins of Term

- Introduced by Dijkstra in 1968
- Inspired by railroad traffic synchronization
- Railroad semaphore indicates whether the track ahead is clear or occupied by another train



Properties

- Semaphores in concurrent programs: work similarly
- Used to implement
  - *mutex* and
  - *condition synchronization*
- Included in most standard libraries for concurrent programming
- Also *system calls* in, e.g., Linux kernel, Windows etc.

Concept

Concept of a Semaphore

- *Semaphore*: special kind of shared program variable (with built-in sync. power)
- value of a semaphore: a *non-negative* integer
- can *only* be manipulated by two *atomic* operations:

Concept

Concept of a Semaphore

- *Semaphore*: special kind of shared program variable (with built-in sync. power)
- value of a semaphore: a *non-negative* integer
- can *only* be manipulated by two *atomic* operations:

The Semaphore Operations: P and V

- **P:** (Passeren) Wait for signal – want to *pass*
  Wait until value is greater than zero, and *decrease* value by one
- **V:** (Vrijgeven) Signal an event – *release*
  Increase the value by one

Concept

Concept of a Semaphore

- *Semaphore*: special kind of shared program variable (with built-in sync. power)
- value of a semaphore: a *non-negative* integer
- can *only* be manipulated by two *atomic* operations:

The Semaphore Operations: P and V

- **P:** (Passeren) Wait for signal – want to *pass*
  Wait until value is greater than zero, and *decrease* value by one
- **V:** (Vrijgeven) Signal an event – *release*
  Increase the value by one

- Today, libraries and sys-calls prefer other names: up/down, wait/signal, acquire/release
- Different flavors of semaphores: binary vs. counting
- Most common: mutex as a synonym for binary semaphores

Syntax and Semantics

Declaration

- sem s; default initial value is zero
- sem s := 1;
- sem s[4] := ([4] 1);

Syntax and Semantics

Declaration

- sem s; default initial value is zero
- sem s := 1;
- sem s[4] := ([4] 1);

Operations and Semantics

P-operation P(s)
$\langle \text{await } (s > 0) s := s - 1 \rangle$

V-operation V(s)
$\langle s := s + 1 \rangle$

Processes waiting on a semaphore are woken up by the op. system.

Remarks on Semaphores

Remark 1

Important: No *direct* access to the value of a semaphore.

For example, a test like `if (s = 1) then ... else` is *forbidden*!

Remarks on Semaphores

Remark 1

Important: No *direct* access to the value of a semaphore.

For example, a test like `if (s = 1) then ... else` is *forbidden*!

Kinds of semaphores

General semaphore: Possible values: *all non-negative integers*

Binary semaphore: Possible values: 0 and 1

Remarks on Semaphores

Remark 1

Important: No *direct* access to the value of a semaphore.

For example, a test like `if (s = 1) then ... else` is *forbidden*!

Kinds of semaphores

General semaphore: Possible values: *all non-negative integers*

Binary semaphore: Possible values: 0 and 1

Fairness

- As for await-statements.
- In most languages: *FIFO* ("waiting queue"): processes delayed while executing P-operations are *awoken* in the *order* they where delayed

Example: Mutual Exclusion (critical section)

Mutex implemented by a *binary semaphore*

Await

```
sem mutex := 1;
process CS[i = 1 to n] {
 while (true) {
  P(mutex);
  # critical section
  V(mutex);
  # noncritical section
 }
}
```

- The semaphore is *initially 1*
- Always P before V $\rightarrow$ (used as) binary semaphore

Example: Barrier Synchronization

Semaphores may be used for *signaling events*

Await

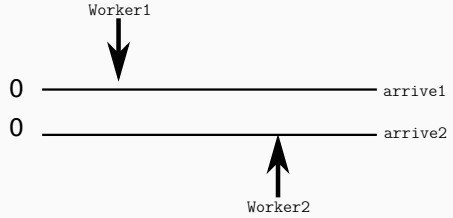
```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
    ...
    V(arrive1); # reach barrier
    P(arrive2); # wait for other
    ...
}
process Worker2 {
    ...
    V(arrive2); # reach barrier
    P(arrive1); # wait for other
    ...
}
```

Example: Barrier Synchronization

- *Signalling* semaphores: usually *initialized* to *0* and
- *Signal* with a V and then *wait* with a P

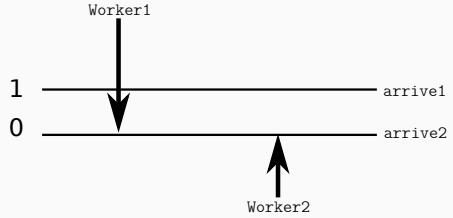
Example: Barrier Synchronization

- *Signalling* semaphores: usually *initialized* to *0* and
- *Signal* with a V and then *wait* with a P



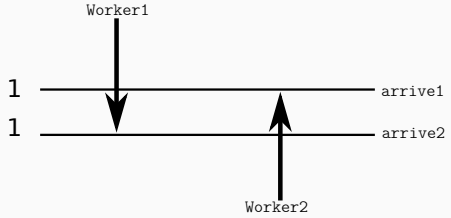
Example: Barrier Synchronization

- *Signalling* semaphores: usually *initialized* to *0* and
- *Signal* with a V and then *wait* with a P



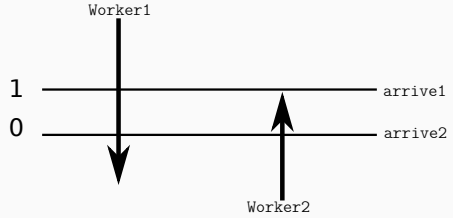
Example: Barrier Synchronization

- *Signalling* semaphores: usually *initialized* to *0* and
- *Signal* with a V and then *wait* with a P



Example: Barrier Synchronization

- *Signalling* semaphores: usually *initialized* to *0* and
- *Signal* with a V and then *wait* with a P



Split Binary Semaphores

Split binary semaphore

A *set* of semaphores, whose *sum* ≤ 1

Mutex by split binary semaphores

- Initialization: *one* of the semaphores =1, all others = 0
- Discipline: all processes call P on a semaphore, *before* calling V on (*another*) semaphore

$\Rightarrow$ Code between the P and the V

- All semaphores = 0
- Code executed *in mutex*

Example: Producer/Consumer with Split Binary Semaphores

Await

```
T buff; # one element buffer, some type T
sem empty := 1;
sem full := 0;
```

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```

Example: Producer/Consumer with Split Binary Semaphores

Await

```
T buff; # one element buffer, some type T
sem empty := 1;
sem full := 0;
```

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```

- `empty` and `full` are both *binary semaphores*, *together* they form a split binary semaphore.
- Solution works with *several* producers/consumers

Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```

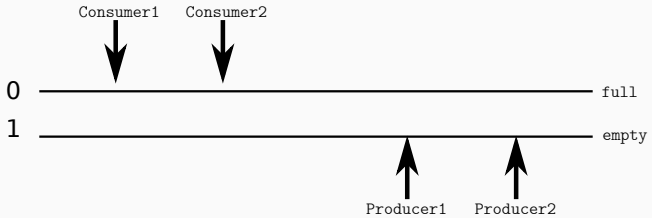
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```



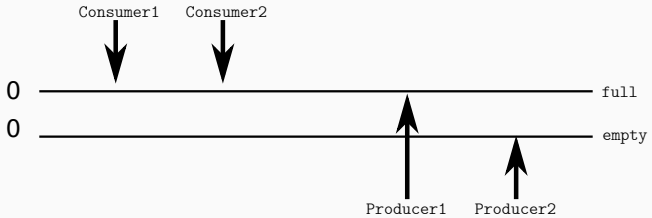
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```



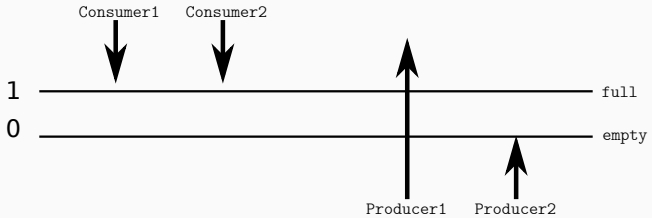
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```



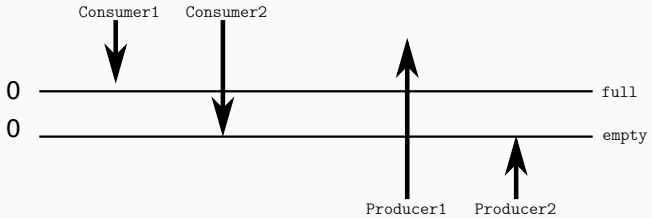
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```



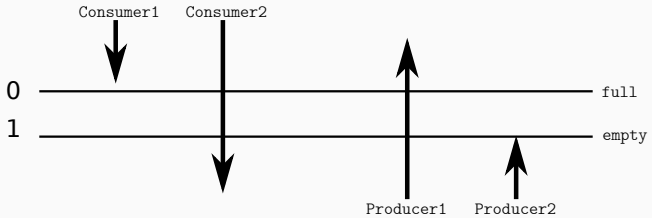
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

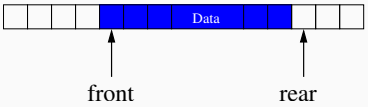
Await

```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```



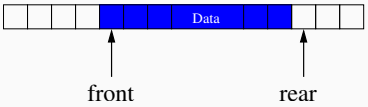
Producer/Consumer: Increasing Buffer Capacity

- Previously: tight coupling, the producer must wait for the consumer to empty the buffer before it can produce a new entry.
- Easy *generalization:* buffer of size n.
- Loose coupling/asynchronous communication $\Rightarrow$ "buffering"
  - *Ring-buffer*, typically represented
    - by an array
    - $+$ two integers `rear` and `front`.
  - Semaphores to *keep track* of the number of free/used slots



Producer/Consumer: Increasing Buffer Capacity

- Previously: tight coupling, the producer must wait for the consumer to empty the buffer before it can produce a new entry.
- Easy *generalization:* buffer of size n.
- Loose coupling/asynchronous communication $\Rightarrow$ "buffering"
  - *Ring-buffer*, typically represented
    - by an array
    - + two integers `rear` and `front`.
  - Semaphores to *keep track* of the number of free/used slots $\Rightarrow$*general* semaphore



Producer/Consumer: Increased Buffer Capacity

Await

```
T buff[n] # array, elements of type T
int front := 0, rear := 0; # ''pointers''
sem empty := n; # number of empty slots
sem full := 0; # number of filled slots
```

Await

```
process Producer {
  while (true) {
    P(empty);
    buff[rear] := data;
    rear := (rear + 1);
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    result := buff[front];
    front := (front + 1);
    V(empty);
  }
}
```

Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
  while (true) {
    P(empty);
    buff[rear] := data;
    rear := (rear + 1);
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    result := buff[front];
    front := (front + 1);
    V(empty);
  }
}
```

`full` 0

`empty` 3

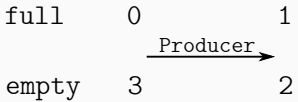
Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
  while (true) {
    P(empty);
    buff[rear] := data;
    rear := (rear + 1);
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    result := buff[front];
    front := (front + 1);
    V(empty);
  }
}
```



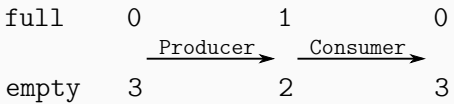
Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
  while (true) {
    P(empty);
    buff[rear] := data;
    rear := (rear + 1);
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    result := buff[front];
    front := (front + 1);
    V(empty);
  }
}
```



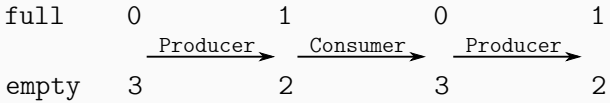
Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
  while (true) {
    P(empty);
    buff[rear] := data;
    rear := (rear + 1);
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    result := buff[front];
    front := (front + 1);
    V(empty);
  }
}
```



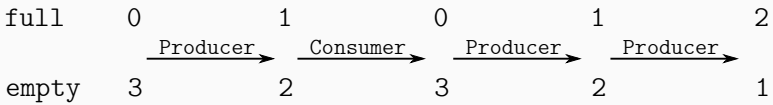
Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
  while (true) {
    P(empty);
    buff[rear] := data;
    rear := (rear + 1);
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    result := buff[front];
    front := (front + 1);
    V(empty);
  }
}
```



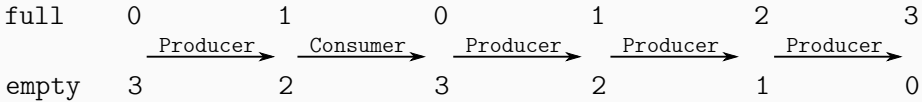
Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
  while (true) {
    P(empty);
    buff[rear] := data;
    rear := (rear + 1);
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    result := buff[front];
    front := (front + 1);
    V(empty);
  }
}
```



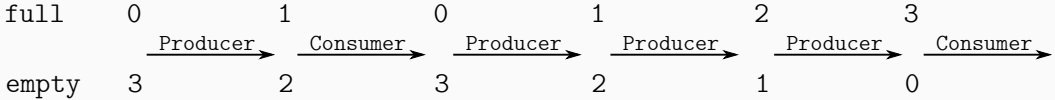
Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
  while (true) {
    P(empty);
    buff[rear] := data;
    rear := (rear + 1);
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    result := buff[front];
    front := (front + 1);
    V(empty);
  }
}
```



Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
  while (true) {
    P(empty);
    buff[rear] := data;
    rear := (rear + 1);
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    result := buff[front];
    front := (front + 1);
    V(empty);
  }
}
```

- Important: there are no critical sections!

Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
  while (true) {
    P(empty);
    buff[rear] := data;
    rear := (rear + 1);
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    result := buff[front];
    front := (front + 1);
    V(empty);
  }
}
```

- Important: there are no critical sections!
- How to enable several producers and consumers?

Increasing the Number of Processes

How to enable several producers and consumers?

New synchronization problems

- *Avoid* that two producers *deposit* to `buff[rear]` before `rear` is updated
- *Avoid* that two consumers *fetch* from `buff[front]` before `front` is updated.

Increasing the Number of Processes

How to enable several producers and consumers?

New synchronization problems

- *Avoid* that two producers *deposit* to `buff[rear]` before `rear` is updated
- *Avoid* that two consumers *fetch* from `buff[front]` before `front` is updated.

Solution

Add 2 extra binary semaphores for protection:

- `mutexDeposit` to deny two producers to deposit to the buffer at the same time.
- `mutexFetch` to deny two consumers to fetch from the buffer at the same time.

Example: Producer/Consumer with Several Processes

Await

```
T buff[n] # array, elements of type T
int front := 0, rear := 0; # ‘‘pointers’’
sem empty := n; # number of empty slots
sem full := 0; # number of filled slots
sem mutexDeposit; mutexFetch := 1; # protect the data stuct.
```

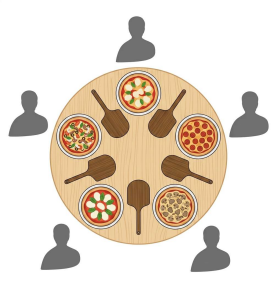
Await

```
process Producer {
  while (true) {
    P(empty);
    P(mutexDeposit);
    buff[rear] := data;
    rear := (rear + 1);
    V(mutexDeposit);
    V(full);
  }
}
```

Await

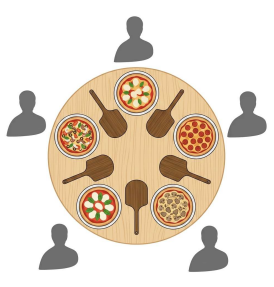
```
process Consumer {
  while (true) {
    P(full);
    P(mutexFetch);
    result := buff[front];
    front := (front + 1);
    V(mutexFetch);
    V(empty);
  }
}
```

Problem: Pizza Makers (Dining Philosophers)



source:Google Gemini

Problem: Pizza Makers (Dining Philosophers)



source:Google Gemini

- Dining Philosophers is a famous sync. problem (Dijkstra)
- Five pizza makers (philosophers) around a circular table.
- One tool placed between each pair of pizza makers
- Each pizza maker alternates between complaining and making pizza
- A pizza maker needs two tools to prepare pizza (and none for complaining)

Pizza Makers: Sketch

Await

```
process PizzaMaker [i = 0 to 4] {
    while true {
        # complain
        acquire tools;
        # make pizza
        release tools;
    }
}
```

Task:

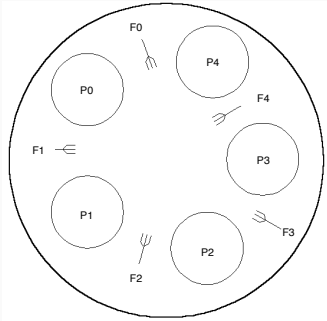
Program the actions `acquire tools` and `release tools`

Pizza Makers: 1st attempt

- Tools as *semaphores*
- Pizza makers: pick up left tool first

Await

```
sem tool[5] := ([5] 1);
 process PizzaMaker [i = 0 to 4] {
   while true {
     # complain
     P(tool[i]);
     P(tool[(i+1)%5]);
     # make pizza
     V(tool[i]);
     V(tool[(i+1)%5]);
   }
}
```



Pizza Makers: 2nd attempt

Breaking the symmetry

To avoid *deadlock*, let 1 pizza maker (say 4) grab the *right* tool first

Await

```
process PizzaMaker [i = 0 to 3] {
    while true {
        complain;
        P(tool[i]);
        P(tool[(i+1)%5]);
        make pizza;
        V(tool[i]);
        V(tool[(i+1)%5]);
    }
}
```

Await

```
process PizzaMaker4 {
    while true {
        complain;
        P(tool[4]); #!
        P(tool[0]); #!
        make pizza;
        V(tool[4]);
        V(tool[0]);
    }
}
```

Pizza Makers: Summary

- Important illustration of problems with concurrency:
  - Deadlocks,
  - Other aspects: liveness, fairness, etc.
- Resource access
- Connection to mutex/critical sections

Invariants and Condition Synchronization

Readers/Writers: Overview

- Classic synchronization problem
- *Reader* and *writer* processes, share access to a database/shared data structure
  - Readers only read from the database
  - Writers update (and read from) the database

Readers/Writers: Overview

- Classic synchronization problem
- *Reader* and *writer* processes, share access to a database/shared data structure
  - Readers only read from the database
  - Writers update (and read from) the database
- As soon as one writer is included, read and write accesses may cause interference
- Readers and writers have **asymmetric requirements:**
  - Every *writer* needs *mutually exclusive* access
  - When no writers have access, *many readers* may access the database

Readers/Writers: Approaches

- Pizza makers: Pair of processes compete for access to "tools"
- Readers/writers: Different *classes* of processes compete for access to the database
  - Readers *compete* with writers
  - Writers *compete* both with readers and other writers
- **General synchronization problem:**
  - Readers: must wait until no writers are active in DB
  - Writers: must wait until no readers or writers are active in DB
- Here: two different approaches
  1. **Mutex:** easy to implement, but "*unfair*"
  2. **Condition synchronization:**
     - Using a *split binary semaphore*
     - Easy to adapt to different scheduling strategies

Readers/Writers with Mutex (1)

Await

```
sem rw := 1;
```

Await

```
process Reader [i=1 to M] {
  while (true) {
    P(rw);
    # read
    V(rw);
  }
}
```

Await

```
process Writer [i=1 to N] {
  while (true) {
    P(rw);
    # write
    V(rw);
  }
}
```

Readers/Writers with Mutex (1)

Await

```
sem rw := 1;
```

Await

```
process Reader [i=1 to M] {
  while (true) {
    P(rw);
    # read
    V(rw);
  }
}
```

Await

```
process Writer [i=1 to N] {
  while (true) {
    P(rw);
    # write
    V(rw);
  }
}
```

We want *more than one reader* simultaneously.

Readers/Writers with Mutex (2)

Await

```
int nr := 0; # number of active readers
sem rw := 1 # lock for reader/writer mutex
```

Await

```
process Reader [i=1 to M] {
  while (true) {
    < nr := nr + 1;
      if (nr=1) P(rw) >;
    # read
    < nr := nr - 1;
      if (nr=0) V(rw) > ;
  }
}
```

Await

```
process Writer [i=1 to N] {
  while (true) {
    P(rw);
    # write
    V(rw);
  }
}
```

Readers/Writers with Mutex (2)

Await

```
int nr := 0; # number of active readers
sem rw := 1 # lock for reader/writer mutex
```

Await

```
process Reader [i=1 to M] {
  while (true) {
    < nr := nr + 1;
      if (nr=1) P(rw) >;
    # read
    < nr := nr - 1;
      if (nr=0) V(rw) > ;
  }
}
```

Await

```
process Writer [i=1 to N] {
  while (true) {
    P(rw);
    # write
    V(rw);
  }
}
```

How do semaphores work *inside* atomic sections?

Readers/Writers with Mutex (3)

Await

```
int nr = 0; # number of active readers
sem rw = 1; # lock for reader/writer exclusion
sem mutexR = 1; # mutex for readers

process Reader [i=1 to M] {
  while (true) {
    P(mutexR)
      nr := nr + 1;
      if (nr=1) P(rw);
    V(mutexR)
    # read
    P(mutexR)
      nr := nr - 1;
      if (nr=0) V(rw);
    V(mutexR)
  }
}
```

Readers/Writers with Condition Synchronization: Overview

Reader's preference

- With a constant stream of readers, the writer will never run
- Even under strong fairness

- Previous *mutex* solution solved *two* separate synchronization problems
  - **rw** : *Readers and writers* for access to the *database*
  - **mutexR**: *Reader vs. reader* for access to the *counter*

- Now: a solution based on **condition synchronization**

Invariant

Reasonable invariant for the critical sections

1. When *a writer* accesses the DB, *no one else* can
2. When *no writers* access the DB, *one or more readers* may get access

Invariant

Reasonable invariant for the critical sections

1. When *a writer* accesses the DB, *no one else* can
2. When *no writers* access the DB, *one or more readers* may get access

Introducing state for the invariant

Introduce two counters:

- nr: number of active readers
- nw: number of active writers

Invariant

Reasonable invariant for the critical sections

1. When *a writer* accesses the DB, *no one else* can
2. When *no writers* access the DB, *one or more readers* may get access

Introducing state for the invariant

Introduce two counters:

- nr: number of active readers
- nw: number of active writers

Invariant

RW: $(nr = 0 \text{ or } nw = 0) \text{ and } nw \leq 1$

(same as:) *RW':* $nw{=}0 \text{ or } (nw = 1 \text{ and } nr = 0)$

Code for counting Readers and Writers

Await

$< \mathsf{nr} := \mathsf{nr} + 1; >$
$\#$ read
$< \mathsf{nr} := \mathsf{nr} - 1; >$

Await

$< \mathsf{nw} := \mathsf{nw} + 1; >$
$\#$ write
$< \mathsf{nw} := \mathsf{nw} - 1; >$

- Add synchronization code to maintain the invariant
- Decreasing counters is not dangerous
- Before increasing, we need to check some conditions for synchronization
  - before increasing `nr`: `nw = 0`
  - before increasing `nw`: `nr = 0 and nw = 0`

Condition Synchronization: Without Semaphores

Await

```
int nr := 0; # nunber of active readers
int nw := 0; # number of active writers
# Invariant RW: (nr = 0 or nw = 0) and nw <= 1
```

Await

```
process Reader [i=1 to M]{
  while (true) {
    < await (nw=0)
      nr := nr+1>;
    # read
    < nr := nr − 1>
  }
}
```

Await

```
process Writer [i=1 to N]{
  while (true) {
    < await (nr = 0 and nw = 0)
      nw := nw+1>;
    # write
    < nw := nw − 1>
  }
}
```

Condition Synchronization: Converting to Split Binary Semaphores

Convert awaits with different guards B_1, B_2 ... to Split Binary Semaphores

- *Entry:* semaphore e, initialized to *1*
- For each guard B_i:
  1. associate one *counter* and
  2. one *delay-semaphore*

  Both initialized to *0*
  - Semaphore *delays* the processes waiting for B_i
  - Counters counts the number of *processes waiting* for B_i

Condition Synchronization: Converting to Split Binary Semaphores

Convert awaits with different guards B_1, B_2... to Split Binary Semaphores

- *Entry:* semaphore e, initialized to *1*
- For each guard B_i:
  1. associate one *counter* and
  2. one *delay-semaphore*

  Both initialized to *0*
  - Semaphore *delays* the processes waiting for B_i
  - Counters counts the number of *processes waiting* for B_i

For *readers/writers* problem we need *3 semaphores* and *2 counters*:

Await

```
sem e = 1;
sem r = 0; int dr = 0; # condition reader: nw == 0
sem w = 0; int dw = 0; # condition writer: nr == 0 and nw == 0
```

Condition Synchronization: Converting to Split Binary Semaphores (2)

- `e`, `r` and `w` form a *split binary semaphore*.
- All execution paths *start* with a *P-operation* and *end* with a *V-operation* $\rightarrow$ Mutex

Condition Synchronization: Converting to Split Binary Semaphores (2)

- `e`, `r` and `w` form a *split binary semaphore*.
- All execution paths *start* with a *P-operation* and *end* with a *V-operation* $\rightarrow$ Mutex

Signaling

We need a signal mechanism *SIGNAL* to pick which semaphore to signal.

- SIGNAL: make sure the invariant holds
- B_i holds when a process enters *CS* because either:
  - the process checks itself,

Condition Synchronization: Converting to Split Binary Semaphores (2)

- `e`, `r` and `w` form a *split binary semaphore*.
- All execution paths *start* with a *P-operation* and *end* with a *V-operation* $\rightarrow$ Mutex

Signaling

We need a signal mechanism *SIGNAL* to pick which semaphore to signal.

- SIGNAL: make sure the invariant holds
- B_i holds when a process enters *CS* because either:
  - the process checks itself,
  - or the process is only *signalled* if B_i holds

Condition Synchronization: Converting to Split Binary Semaphores (2)

- `e`, `r` and `w` form a *split binary semaphore*.
- All execution paths *start* with a *P-operation* and *end* with a *V-operation* → Mutex

Signaling

We need a signal mechanism *SIGNAL* to pick which semaphore to signal.

- SIGNAL: make sure the invariant holds
- B_i holds when a process enters *CS* because either:
  - the process checks itself,
  - or the process is only *signalled* if B_i holds
- **Another pitfall:**
  Avoid *deadlock* by checking the counters before the delay semaphores are signalled.
  - r is not signalled (V(r)) *unless* there is a delayed reader
  - w is not signalled (V(w)) *unless* there is a delayed writer

Condition Synchronization: Reader

Await

```
int nr := 0, nw = 0; # counter variables (as before)
sem e := 1; # entry semaphore
int dr := 0; sem r := 0; # delay counter + sem for reader
int dw := 0; sem w := 0; # delay counter + sem for writer
# invariant RW: (nr = 0 or nw = 0 ) and nw <= 1
process Reader [i=1 to M]{ # entry condition: nw = 0
  while (true) {
      P(e);
      if (nw > 0) { dr := dr + 1; # < await (nw=0)
                    V(e); # nr:=nr+1 >
                    P(r)};
      nr:=nr+1; SIGNAL;
      # read
      P(e); nr:=nr−1; SIGNAL; # < nr:=nr−1 >
  }
}
```

With Condition Synchronization: Writer

Await

```
process Writer [i=1 to N]{ # entry condition: nw = 0 and nr = 0
  while (true) {
      P(e); # < await (nr=0 and nw=0)
      if (nr > 0 or nw > 0) { # nw:=nw+1 >
          dw := dw + 1;
          V(e);
          P(w) };
      nw:=nw+1; SIGNAL;
      # write
      P(e); nw:=nw−1; SIGNAL # < nw:=nw−1>
  }
}
```

With Condition Synchronization: Signalling

Await

```
if (nw = 0 and dr > 0) {
        dr := dr −1; V(r); # awake reader
}
elseif (nr = 0 and nw = 0 and dw > 0) {
        dw := dw −1; V(w); # awake writer
}
else V(e); # release entry lock
```

- This passes the control (the "baton") to an appropriate next process
- SIGNAL has no P operation, each path has exactly one V opereration.
- Using the conditions to see who goes next.
- Called "passing the baton" technique (as in relay competition).
- Conditions for awakening must be disjoint

Example: 1 Reader, 1 Writer, Reader starts

r	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
dr	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
e	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	1
dw	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
w	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

Await

```
process Reader [i=1 to M]{ # entry condition: nw = 0
  while (true) {
      P(e);
      if (nw > 0) { dr := dr + 1; # < await (nw=0
                    V(e); # nr:=nr+1 >
                    P(r)};
      nr:=nr+1; SIGNAL;
      # read
      P(e); nr:=nr-1; SIGNAL; # < nr:=nr-1 >
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

r	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
dr	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
e	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	1
dw	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
w	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

Await

```
process Reader [i=1 to M]{ # entry condition: nw = 0
  while (true) {
      P(e);
      if (nw > 0) { dr := dr + 1; # < await (nw=0
                    V(e); # nr:=nr+1 >
                    P(r)};
      nr:=nr+1; SIGNAL;
      # read
      P(e); nr:=nr-1; SIGNAL; # < nr:=nr-1 >
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

r	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
dr	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
e	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	1
dw	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
w	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

Await

```
if (nw = 0 and dr > 0) {
        dr := dr −1; V(r); # awake reader
}
elseif (nr = 0 and nw = 0 and dw > 0) {
        dw := dw −1; V(w); # awake writer
}
else V(e); # release entry lock
```

Example: 1 Reader, 1 Writer, Reader starts

r	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
dr	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
e	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	1
dw	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
w	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

Await

```
process Writer [i=1 to N]{ # entry condition: nw = 0 and nr = 0
  while (true) {
      P(e); # < await (nr=0 && nw=0
      if (nr > 0 or nw > 0) { # nw:=nw+1 >
          dw := dw + 1;
          V(e);
          P(w) };
      nw:=nw+1; SIGNAL;
      # write
      P(e); nw:=nw-1; SIGNAL # < nw:=nw-1>
  }
```

Semaphores in Java

Basic Methods of Semaphores in Java

- Semaphore(int n)
  - constructor for semaphores
  - initializes semaphore value with integer n *set of permits*
- acquire()
  - corresponds to the P operation
  - tries to decrease the number of permits by 1
  - blocks, if that is not possible and waits, until semaphore gives permit
- release()
  - corresponds to the V operation
  - increases the number of permits by 1

Pizza Makers: Naïve Solution in Java (I)

Pizza Makers in Java

- Pizza maker has references to two binary Semaphores (leftTool and rightTool),
- and the functions makePizza(), rest() and run()

Java

```
Semaphore[] tools = new Semaphore[numberOfPizzaMakers];
for (int i=0; i <tools.length; i++)
  tools[i] = new Semaphore(1);

pizzaMakers = new PizzaMaker[numberOfPizzaMakers];
for (int i=0; i < pizzaMakers.length; i++)
  pizzaMakers[i] =
    new PizzaMaker(i,tools[i], tools[(i+1) % tools.length]);
```

Pizza Makers: Naïve Solution in Java (II)

Java

```
while(true) {
  complain();   // complain
  if(i == 0) {
      rightTool.acquire();  // acquire tools
      leftTool.acquire();

  } else {
      leftTool.acquire();  // acquire tools
      rightTool.acquire();
  }
  makePizza();   // make pizza
  leftTool.release();  // release tools
  rightTool.release();
}
```

The Condition Interface

- A condition allows to transfer the ownership of the lock without lock/unlock
- Each condition is, thus, bound to a lock

The Condition Interface

- A condition allows to transfer the ownership of the lock without lock/unlock
- Each condition is, thus, bound to a lock

The Condition interface includes the following methods:

- cond.await()
  - The lock associated with the Condition is atomically released (unlock) and the thread becomes disabled
  - After cond is signalled, the thread continues with its instructions.
- cond.signal()
  - Wakes up one thread that is waiting on this Condition

The Condition Interface

- A condition allows to transfer the ownership of the lock without lock/unlock
- Each condition is, thus, bound to a lock

The Condition interface includes the following methods:

- cond.await()
  - The lock associated with the Condition is atomically released (unlock) and the thread becomes disabled
  - After cond is signalled, the thread continues with its instructions.
- cond.signal()
  - Wakes up one thread that is waiting on this Condition
- Note: threads interacting with cond still need to acquire and release its lock!

Java

```java
Lock mutex = new ReentrantLock();
Condition condition = mutex.newCondition();

public void waitingThread() throws InterruptedException {
  mutex.lock(); // thread acquires the lock
  try {
    while(/*not finished*/) {
      condition.await(); // Release the lock and wait for signal
      /* thread does something (1) */
    }
  } finally {
    mutex.unlock(); // thread releases the lock
  }
}
```

Java

```
Lock mutex = new ReentrantLock();
Condition condition = mutex.newCondition();

public void signallingThread() throws InterruptedException {
  mutex.lock(); // thread acquires the lock ;
  try {
    /* thread does something (2) */
    condition.signal(); // Signal (wake up) one waiting thread
  } finally {
    mutex.unlock(); // thread releases the lock
  }
}
```

Producer Consumer with Locks and Conditions

Example can be found at: Example link

Conclusion

Condition synchronization

- One semaphore to protect shared variables (the counters)
- For each condition: a semaphore + a "delay" counter
- On entry: increase delay counter if your condition is not true
- Wait on your condition semaphore
- Decide who is next (SIGNAL) using
  - the conditions, and
  - the delay counters to see who is waiting to enter
- SIGNAL whenever someone should get a chance to enter.