IN5170/IN9170 – Models of concurrency

Semaphores, lecture 5

Fall 2022

05. 09. 2022



Last lecture: Locks and Barriers

- Complex techniques
- No clear separation between variables for synchronization and variables to compute results
- Busy waiting

Last lecture: Locks and Barriers

- Complex techniques
- No clear separation between variables for synchronization and variables to compute results
- 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

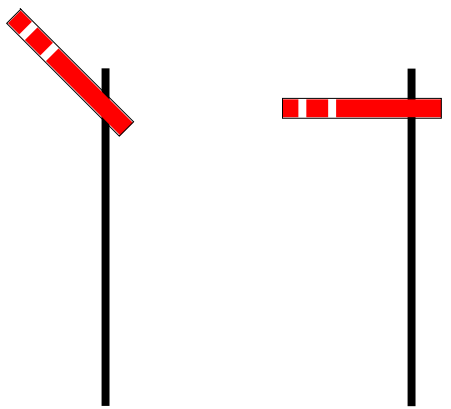
- Semaphores: Syntax and semantics
- Synchronization examples:
  - Mutual exclusion (critical sections)
  - Barriers (signaling events)
  - Producers and consumers (split binary semaphores)
  - Bounded buffer: resource counting
  - Dining philosophers: mutual exclusion – deadlock
  - Readers and writers: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

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

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



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, similar in Windows etc.

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 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:

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 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:

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

- Nowadays, 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

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 waken up by the op. system.

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 awaken in the order they where delayed

Example: Mutual Exclusion (critical section)

Mutex[1] implemented by a binary semaphore

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

Note:

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

[1] As mentioned: "mutex" is also used to refer to a data-structure, basically the same as binary semaphore itself.

Example: Barrier Synchronization

Semaphores may be used for signaling events

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

Example: Barrier Synchronization

Semaphores may be used for signaling events

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

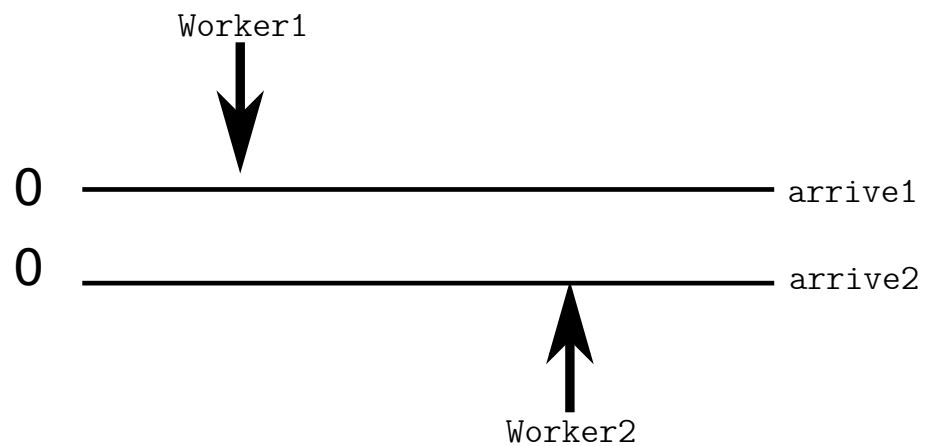
Note:

- signalling semaphores: usually initialized to 0 and
- signal with a V and then wait with a P

Example: Barrier Synchronization

Semaphores may be used for signaling events

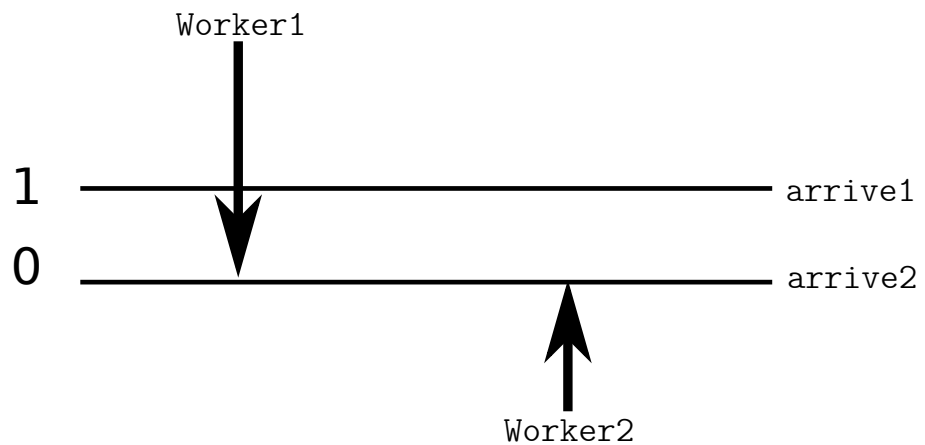
```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
                    ...
        V(arrive1);         reach the barrier
        P(arrive2);         wait for other processes
                    ...
}
process Worker2 {
                    ...
        V(arrive2);         reach the barrier
        P(arrive1);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



Example: Barrier Synchronization

Semaphores may be used for signaling events

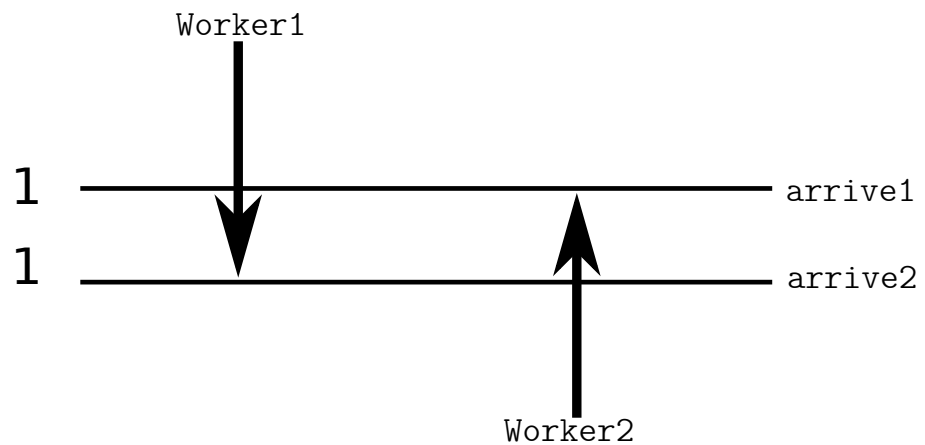
```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
                    ...
        V(arrive1);         reach the barrier
        P(arrive2);         wait for other processes
                    ...
}
process Worker2 {
                    ...
        V(arrive2);         reach the barrier
        P(arrive1);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



Example: Barrier Synchronization

Semaphores may be used for signaling events

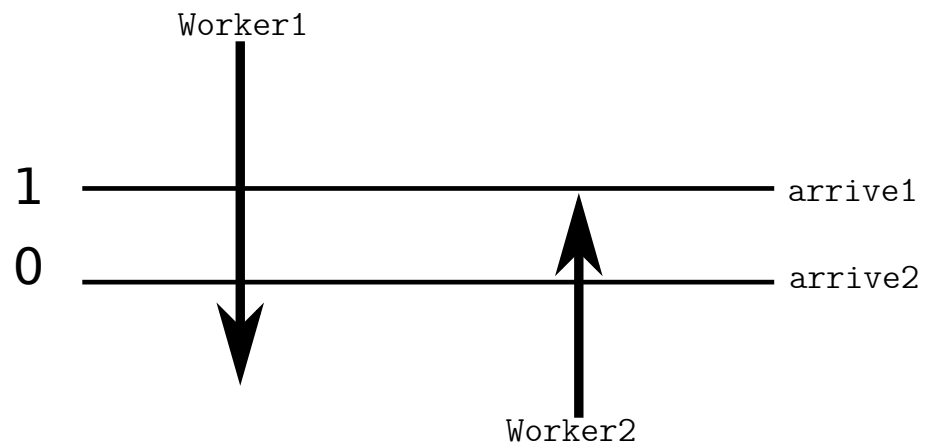
```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
                    ...
        V(arrive1);         reach the barrier
        P(arrive2);         wait for other processes
                    ...
}
process Worker2 {
                    ...
        V(arrive2);         reach the barrier
        P(arrive1);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



Example: Barrier Synchronization

Semaphores may be used for signaling events

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



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

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

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

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

Example: Producer/Consumer with Split Binary Semaphores

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

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

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

Note:

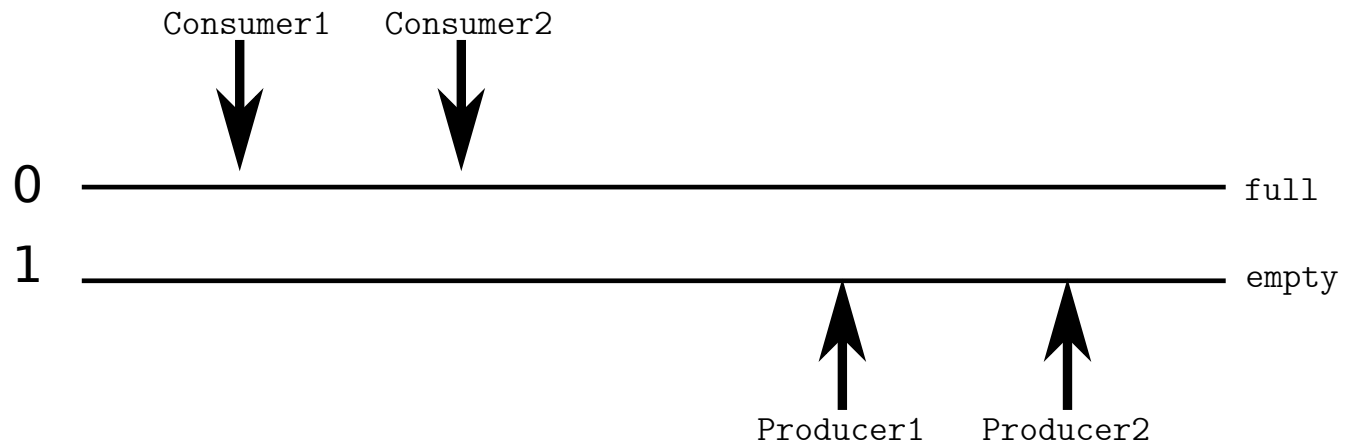
- remember also P/C with await
- `empty` and `full` are both binary semaphores, together they form a split binary semaphore.
- solution works with several producers/consumers

Example: Producer/Consumer with Split Binary Semaphores

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

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

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

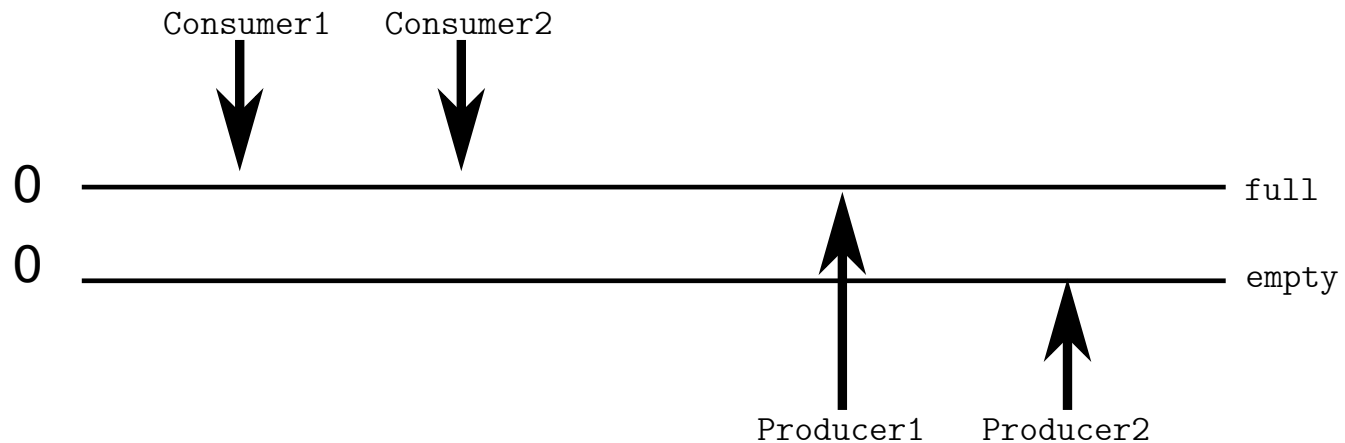


Example: Producer/Consumer with Split Binary Semaphores

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

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

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

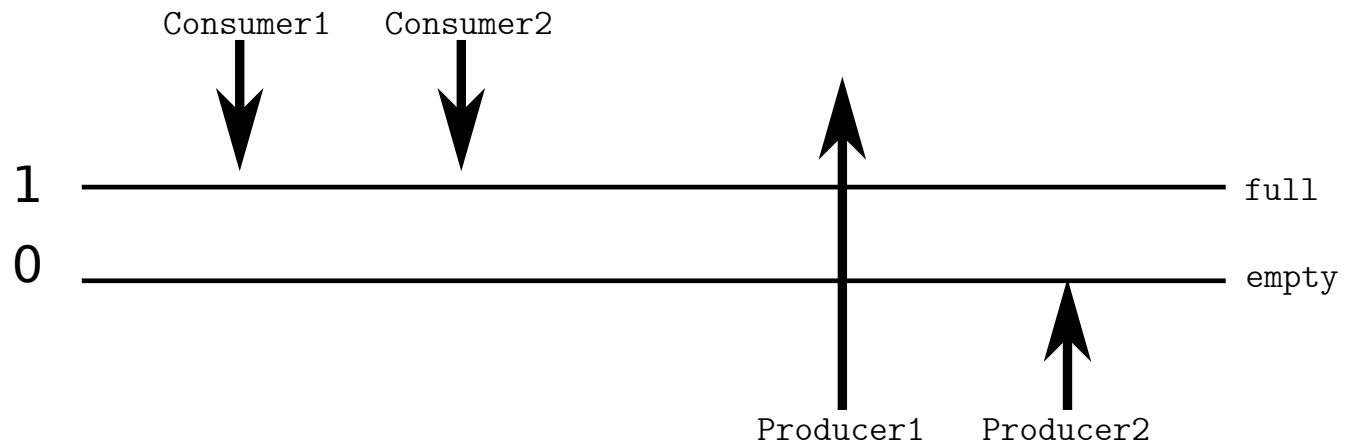


Example: Producer/Consumer with Split Binary Semaphores

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

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

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

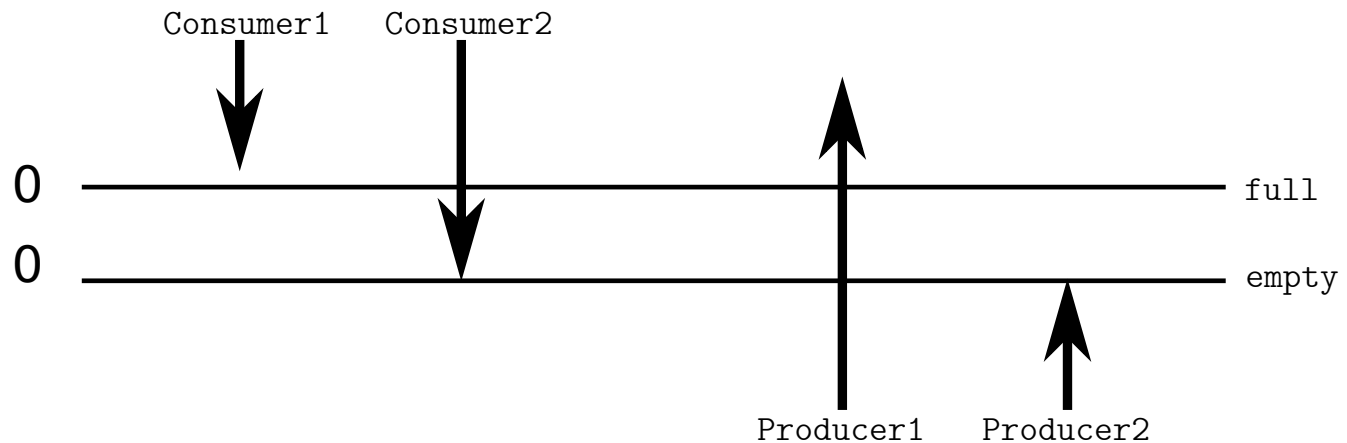


Example: Producer/Consumer with Split Binary Semaphores

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

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

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

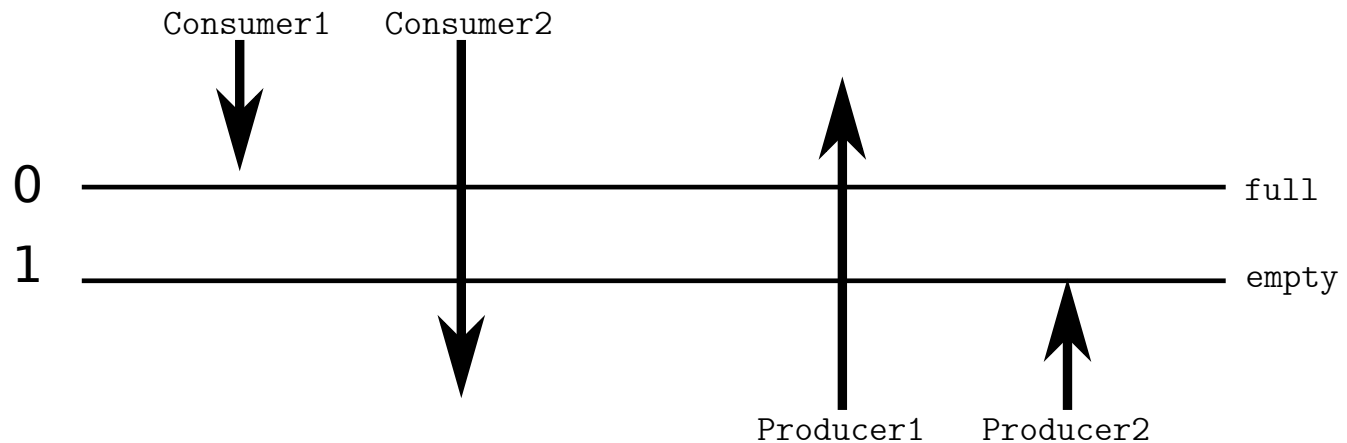


Example: Producer/Consumer with Split Binary Semaphores

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

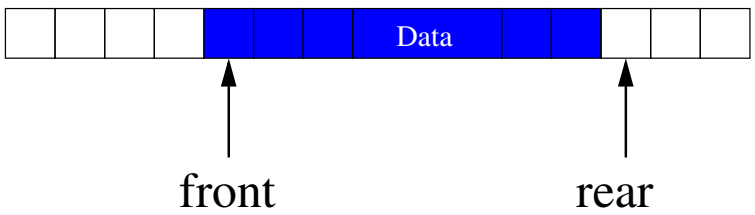
```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

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



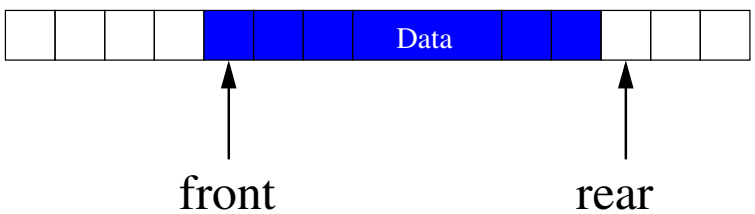
Producer/Consumer: Increasing Buffer Capacity

- previously: tight coupling
- 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
- 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

```
T buf[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
```

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

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

Producer/Consumer: Increased Buffer Capacity

```
T buf[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
```

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

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

```
full    0

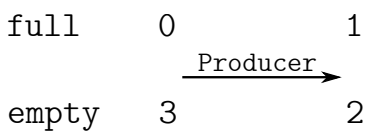
empty   3
```

Producer/Consumer: Increased Buffer Capacity

```
T buf[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
```

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

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

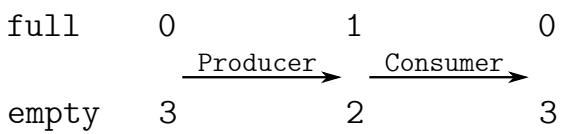


Producer/Consumer: Increased Buffer Capacity

```
T buf[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
```

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

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

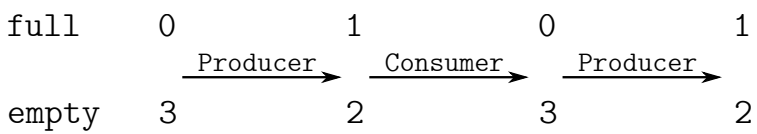


Producer/Consumer: Increased Buffer Capacity

```
T buf[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
```

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

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

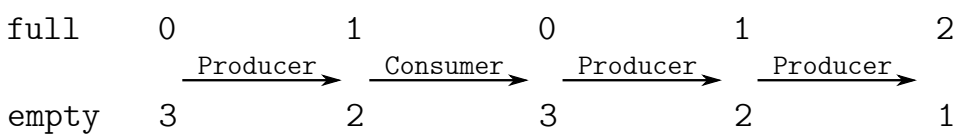


Producer/Consumer: Increased Buffer Capacity

```
T buf[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
```

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

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

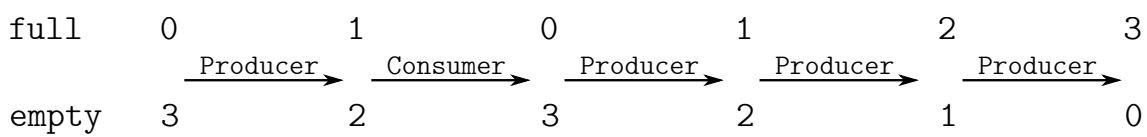


Producer/Consumer: Increased Buffer Capacity

```
T buf[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
```

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

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

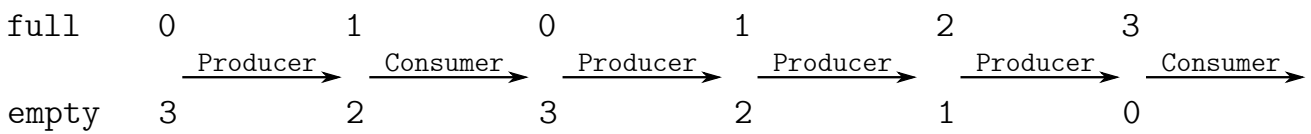


Producer/Consumer: Increased Buffer Capacity

```
T buf[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
```

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

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



Producer/Consumer: Increased Buffer Capacity

```
T buf[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
```

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

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

critical regions?

Producer/Consumer: Increased Buffer Capacity

```
T buf[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
```

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

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

critical regions? several producers or consumers?

How to enable several producers and consumers?

New synchronization problems

- Avoid that two producers deposits to `buf[rear]` before `rear` is updated
- Avoid that two consumers fetches from `buf[front]` before `front` is updated.

Solution

Additionally 2 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.

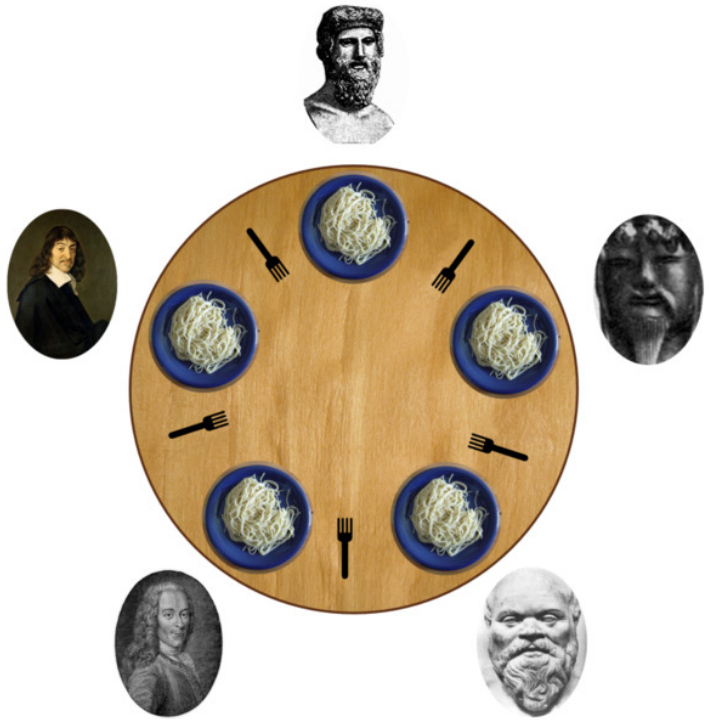
Example: Producer/Consumer with Several Processes

```
T buf[n]                                    # array, elem's of type T
int front := 0; rear := 0;                  # ''pointers''
sem empty := n;
sem full  := 0;
sem mutexDeposit; mutexFetch := 1; # protect the data stuct.
```

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

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

Problem: Dining Philosophers



[2] image from wikipedia.org

From top: Plato, Konfuzius, Socrates, Voltaire and Descartes.

Problem: Dining Philosophers

- famous sync. problem (Dijkstra)
- Five philosophers around a circular table.
- one fork placed between each pair of philosophers
- each philosopher alternates between thinking and eating
- philosopher needs two forks to eat (and none for thinking)



[2] image from wikipedia.org

From top: Plato, Konfuzius, Socrates, Voltaire and Descartes.

Dining Philosophers: Sketch

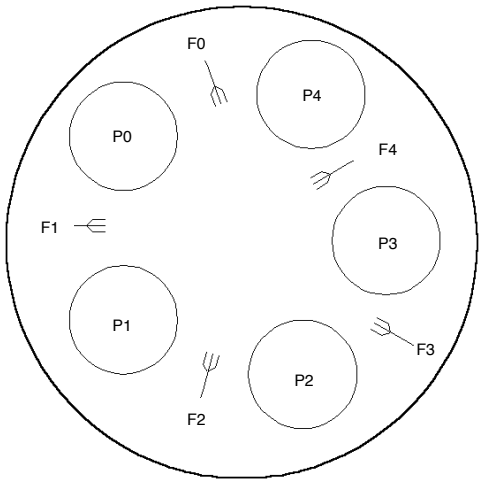
```
process Philosopher [i = 0 to 4] {
  while true {
    think;
    acquire forks;
    eat;
    release forks;
  }
}
```

now: program the actions `acquire forks` and `release forks`

Dining philosophers: 1st attempt

- forks as semaphores
- philosophers: pick up left fork first

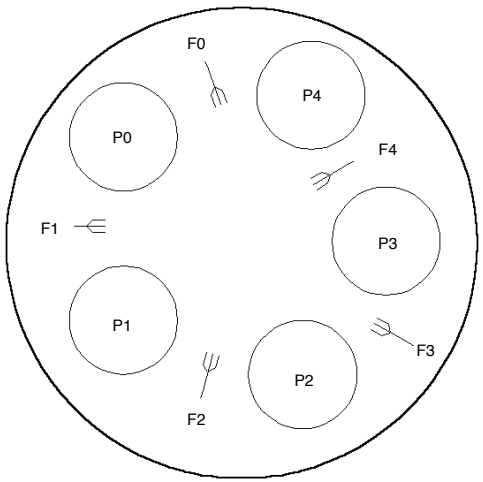
```
process Philosopher [i = 0 to 4] {
  while true {
    think;
    acquire forks;
    eat;
    release forks;
  }
}
```



Dining philosophers: 1st attempt

- forks as semaphores
- philosophers: pick up left fork first

```
sem fork[5] := ([5] 1);
process Philosopher [i = 0 to 4] {
  while true {
    think;
    P(fork[i]);
    P(fork[(i+1)%5]);
    eat;
    V(fork[i]);
    V(fork[(i+1)%5]);
  }
}
```



ok solution?

Example: Dining philosophers 2nd attempt

breaking the symmetry

To avoid deadlock, let 1 philosopher (say 4) grab the right fork first

```
process Philosopher [i = 0 to 3] {
  while true {
    think;
    P(fork[i]);
    P(fork[(i+1)%5]);
    eat;
    V(fork[i]);
    V(fork[(i+1)%5]);
  }
}
```

```
process Philosopher4 {
  while true {
    think;
    P(fork[4]);
    P(fork[0]);
    eat;
    V(fork[4]);
    V(fork[0]);
  }
}
```

Example: Dining philosophers 2nd attempt

breaking the symmetry

To avoid deadlock, let 1 philosopher (say 4) grab the right fork first

```
process Philosopher [i = 0 to 3] {
  while true {
    think;
    P(fork[i]);
    P(fork[(i+1)%5]);
    eat;
    V(fork[i]);
    V(fork[(i+1)%5]);
  }
}
```

```
process Philosopher4 {
  while true {
    think;
    P(fork[0]);
    P(fork[4]);
    eat;
    V(fork[4]);
    V(fork[0]);
  }
}
```

dining philosophers

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

Example: Readers/Writers Overview

- Classical synchronization problem
- Reader and writer processes, sharing access to a "database"
  - readers: read-only from the database
  - writers: update (and read from) the database

Example: Readers/Writers Overview

- Classical synchronization problem
- Reader and writer processes, sharing access to a "database"
  - readers: read-only from the database
  - writers: update (and read from) the database
- R/R access unproblematic, W/W or W/R: interference
  - writers need mutually exclusive access
  - When no writers have access, many readers may access the database

Readers/Writers Approaches

- Dining philosophers: Pair of processes compete for access to "forks"
- Readers/writers: Different classes of processes competes 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)

sem rw := 1

```
process Reader [i=1 to M] {
  while (true) {
    ...
    P(rw);

    read from DB

    V(rw);
  }
}
```

```
process Writer [i=1 to N] {
  while (true) {
    ...
    P(rw);

    write to DB

    V(rw);
  }
}
```

Readers/Writers with Mutex (1)

sem rw := 1

```
process Reader [i=1 to M] {
  while (true) {
    ...
    P(rw);

    read from DB

    V(rw);
  }
}
```

```
process Writer [i=1 to N] {
  while (true) {
    ...
    P(rw);

    write to DB

    V(rw);
  }
}
```

- safety ok
- but: unnecessarily cautious
- We want more than one reader simultaneously.

Readers/Writers with Mutex (2)

Initially:

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

```
process Reader [i=1 to M] {
  while (true) {
    ...
    < nr := nr + 1;
      if (nr=1)  P(rw) > ;

    read from DB

    < nr := nr - 1;
      if (nr=0)  V(rw) > ;
  }

}
```

```
process Writer [i=1 to N] {
  while (true) {
    ...

    P(rw);

    write to DB


    V(rw);
  }

}
```

Readers/Writers with Mutex (2)

Initially:

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

```
process Reader [i=1 to M] {
  while (true) {
    ...
    < nr := nr + 1;
      if (nr=1)  P(rw) > ;

    read from DB

    < nr := nr - 1;
      if (nr=0)  V(rw) > ;
  }

}
```

```
process Writer [i=1 to N] {
  while (true) {
    ...

    P(rw);

    write to DB


    V(rw);
  }

}
```

Semaphore inside await statement? Strange semantics. Does it work?

Readers/Writers with Mutex (3)

```
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 from DB
      P(mutexR)
      nr := nr - 1;
      if (nr=0)  V(rw);
      V(mutexR)
  }
}
```

Readers/Writers with Mutex (3)

```
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 from DB
      P(mutexR)
      nr := nr - 1;
      if (nr=0)  V(rw);
      V(mutexR)
  }
}
```

"Fairness"

What happens if we have a constant stream of readers?

Readers/Writers with Mutex (3)

```
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 from DB
      P(mutexR)
      nr := nr - 1;
      if (nr=0)  V(rw);
      V(mutexR)
  }
}
```

"Fairness"

"Reader's preference"

Readers/Writers with Condition Synchronization: Overview

- 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**

Reasonable invariant[a]

[a] Invariant outside critical regions

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

- introduce two counters:
  - `nr`: number of active readers
  - `nw`: number of active writers

The invariant may be:

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

Reasonable invariant[a]

[a] Invariant outside critical regions

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

- introduce two counters:
  - `nr`: number of active readers
  - `nw`: number of active writers

The invariant may be:

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

Reader:

```
< nr := nr + 1; >
```

read from DB

```
< nr := nr - 1; >
```

Writer:

```
< nw := nw + 1; >
```

write to DB

```
< nw := nw - 1; >
```

- maintain invariant $\Rightarrow$ add sync-code
- decrease counters: not dangerous
- before increasing, check/synchronize:
  - before increasing `nr`: `nw = 0`
  - before increasing `nw`: `nr = 0 and nw = 0`

Condition Synchronization: Without Semaphores

Initially:

```
int nr := 0;    # nunber of active readers
int nw := 0;    # number of active writers

## Invariant RW: (nr = 0 or nw = 0) and nw <= 1
```

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

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

implementation of `await`'s: possible via split binary semaphores

- May be used to implement different synchronization problems with different guards B_1, $B_2 \ldots$

General pattern

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

  both initialized to 0
  - semaphore: delay the processes waiting for B_i
  - counter: count the number of processes waiting for B_i

$\Rightarrow$ for readers/writers problem: 3 semaphores and 2 counters:

```
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 Synchr.: 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 CR because either:
  - the process checks itself,
  - or
-

Condition Synchr.: 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 CR because either:
  - the process checks itself,
  - or the process is only signaled if B_i holds
- and another pitfall:

Condition Synchr.: 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 CR because either:
  - the process checks itself,
  - or the process is only signaled if B_i holds
- and another pitfall: Avoid deadlock by checking the counters before the delay semaphores are signaled.
  - `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 Synchr.: Reader

```
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 ∨ nw = 0 ) ∧ 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 from DB;

      P(e); nr:=nr-1; SIGNAL;      # < nr:=nr-1 >
  }
}
```

Note: First reader does nr+1 without having e, the next ones have e.

With Condition Synchronization: Writer

```
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 to DB;

      P(e); nw:=nw-1; SIGNAL                 # < nw:=nw-1>
  }
}
```

With Condition Synchronization: Signalling

- SIGNAL

```
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 process!
- SIGNAL has no P operation, each path has exactly one V oper.
- using the conditions to see who goes next.
- Called "passing the baton" technique (as in relay competition).

Example: 1 Reader, 1 Writer, Reader starts

nr	0
nw	0
e	1
dw	0
w	0

```
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 from DB;

      P(e); nr:=nr-1; SIGNAL;      # < nr:=nr-1 >
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0
nw	0	0
e	1	0
dw	0	0
w	0	0

```
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 from DB;

      P(e); nr:=nr-1; SIGNAL;       # < nr:=nr-1 >
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1
nw	0	0	0
e	1	0	0
dw	0	0	0
w	0	0	0

```
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

nr	0	0	1	1
nw	0	0	0	0
e	1	0	0	1
dw	0	0	0	0
w	0	0	0	0

```
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 to DB;

      P(e); nw:=nw-1; SIGNAL                 # < nw:=nw-1>
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1
nw	0	0	0	0	0
e	1	0	0	1	0
dw	0	0	0	0	0
w	0	0	0	0	0

```
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 to DB;

      P(e); nw:=nw-1; SIGNAL                 # < nw:=nw-1>
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1
nw	0	0	0	0	0	0
e	1	0	0	1	0	0
dw	0	0	0	0	0	1
w	0	0	0	0	0	0

```
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 to DB;

      P(e); nw:=nw-1; SIGNAL                 # < nw:=nw-1>
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1
nw	0	0	0	0	0	0	0
e	1	0	0	1	0	0	1
dw	0	0	0	0	0	1	1
w	0	0	0	0	0	0	0

```
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 from DB;

      P(e); nr:=nr-1; SIGNAL;      # < nr:=nr-1 >
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	1
nw	0	0	0	0	0	0	0	0
e	1	0	0	1	0	0	1	0
dw	0	0	0	0	0	1	1	1
w	0	0	0	0	0	0	0	0

```
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 from DB;

      P(e); nr:=nr-1; SIGNAL;      # < nr:=nr-1 >
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	1	0
nw	0	0	0	0	0	0	0	0	0
e	1	0	0	1	0	0	1	0	0
dw	0	0	0	0	0	1	1	1	1
w	0	0	0	0	0	0	0	0	0

```
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

nr	0	0	1	1	1	1	1	1	0	0
nw	0	0	0	0	0	0	0	0	0	0
e	1	0	0	1	0	0	1	0	0	0
dw	0	0	0	0	0	1	1	1	1	0
w	0	0	0	0	0	0	0	0	0	1

```
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

nr	0	0	1	1	1	1	1	1	0	0	0
nw	0	0	0	0	0	0	0	0	0	0	0
e	1	0	0	1	0	0	1	0	0	0	0
dw	0	0	0	0	0	1	1	1	1	0	0
w	0	0	0	0	0	0	0	0	0	1	1

```
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 to DB;

      P(e); nw:=nw-1; SIGNAL                 # < nw:=nw-1>
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	1	0	0	0	0
nw	0	0	0	0	0	0	0	0	0	0	0	1
e	1	0	0	1	0	0	1	0	0	0	0	0
dw	0	0	0	0	0	1	1	1	1	0	0	0
w	0	0	0	0	0	0	0	0	0	1	1	0

```
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 to DB;

      P(e); nw:=nw-1; SIGNAL                 # < nw:=nw-1>
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	1	0	0	0	0	0
nw	0	0	0	0	0	0	0	0	0	0	0	1	1
e	1	0	0	1	0	0	1	0	0	0	0	0	1
dw	0	0	0	0	0	1	1	1	1	0	0	0	0
w	0	0	0	0	0	0	0	0	0	1	1	0	0

```
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 to DB;

      P(e); nw:=nw-1; SIGNAL            # < nw:=nw-1>
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	1	0	0	0	0	0	0
nw	0	0	0	0	0	0	0	0	0	0	0	1	1	1
e	1	0	0	1	0	0	1	0	0	0	0	0	1	0
dw	0	0	0	0	0	1	1	1	1	0	0	0	0	0
w	0	0	0	0	0	0	0	0	0	1	1	0	0	0

```
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 to DB;

      P(e); nw:=nw-1; SIGNAL                 # < nw:=nw-1>
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0
nw	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0
e	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0
dw	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
w	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0

```
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 to DB;

      P(e); nw:=nw-1; SIGNAL                 # < nw:=nw-1>
  }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
nw	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

```
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
```

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 one 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.

Next Lecture

Wednesday, practical lecture on semaphores.

References I

[Andrews, 2000] Andrews, G. R. (2000).
Foundations of Multithreaded, Parallel, and Distributed Programming.
Addison-Wesley.

[Dahl, 1992] Dahl, O.-J. (1992).
Verifiable Programming.
International Series in Computer Science. Prentice Hall, New York, N.Y.

[Gordon, 2016] Gordon, M. (2016).
Background reading on Hoare Logic.
https://www.cl.cam.ac.uk/archive/mjcg/HL/Notes/Notes.pdf.

[Hoare, 1969] Hoare, C. A. R. (1969).
An Axiomatic Basis of Computer Programming.
Communications of the ACM, 12:576–580.