Semaphores

Silvia Lizeth Tapia Tarifa

September 10, 2025

University of Oslo

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

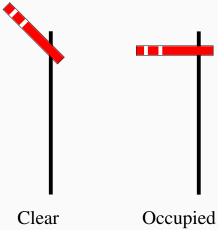
- Semaphores: Syntax and semantics
- Synchronization examples:
  - Mutual exclusion (critical sections)
  - Barriers (signaling events)
  - Producers and consumers (split binary semaphores)
  - Bounded buffer: resource counting
  - Dining philosophers: mutual exclusion – deadlock
  - Readers and writers: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



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:

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);

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*!

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

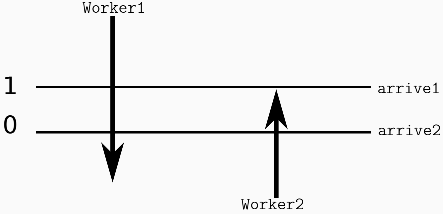
Semaphores may be used for *signaling events*

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



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 buf;  # 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

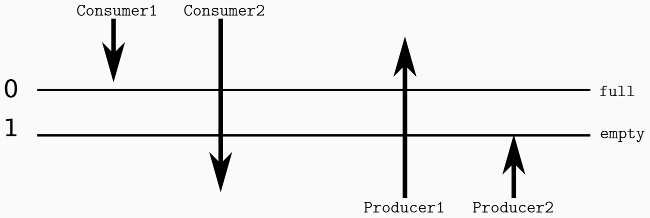
Example: Producer/Consumer with Split Binary Semaphores

Await

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

Await

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



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 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
```

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!
- How to enable several producers and consumers?

full	0	$\xrightarrow{\text{Producer}}$	1	$\xrightarrow{\text{Consumer}}$	0	$\xrightarrow{\text{Producer}}$	1	$\xrightarrow{\text{Producer}}$	2	$\xrightarrow{\text{Producer}}$	3	$\xrightarrow{\text{Consumer}}$
empty	3		2		3		2		1		0	

Increasing the Number of Processes

How to enable several producers and consumers?

New synchronization problems

- *Avoid* that two producers *deposit* to `buf[rear]` before `rear` is updated
- *Avoid* that two consumers *fetch* from `buf[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 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
sem mutexDeposit; mutexFetch := 1; # protect the data stuct.
```

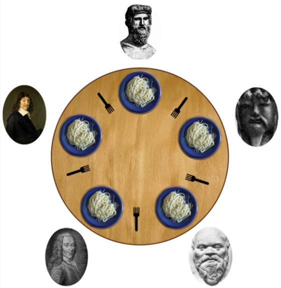
Await

```
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: Dining Philosophers



source:wikipedia.org

- 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
- A philosopher needs two forks to eat (and none for thinking)

Dining Philosophers: Sketch

Await

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

Task:

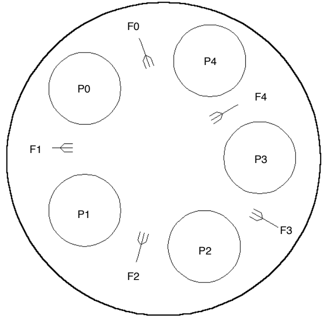
Program the actions `acquire forks` and `release forks`

Dining philosophers: 1st attempt

- Forks as *semaphores*
- Philosophers: pick up left fork first

Await

```
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]);
  }
}
```



Dining philosophers: 2nd attempt

Breaking the symmetry

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

Await

```
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]);
  }
}
```

Await

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

Dining philosophers

- 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
- 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

- Dining philosophers: Pair of processes compete for access to “forks”
- 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);
  }
}
```

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);
  }
}
```

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

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

```
< nr := nr + 1; >
# read
< nr := nr - 1; >
```

Await

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

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* → 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

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

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
```

Await

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

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

Dining Philosophers: Naïve Solution in Java (I)

Philosophers in Java

- Philosopher has references to two binary Semaphores (leftFork and rightFork),
- and the functions eat(), sleep() and run()

Java

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

philosophers = new Philosopher[numberOfPhilosophers];
for (int i=0; i < philosophers.length; i++)
  philosophers[i] =
    new Philosopher(i,forks[i], forks[(i+1) % forks.length]);
```

Dining Philosophers: Naïve Solution in Java (II)

Java

```java
while(true) {
  think();                              // think
  if(i == 0) {
      rightFork.acquire();                      // acquire forks
      leftFork.acquire();

  } else {
      leftFork.acquire();                       // acquire forks
      rightFork.acquire();
  }
  eat();                                        // eat
  leftFork.release();                   // release forks
  rightFork.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 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

```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.