

Semaphores

Semaphore Types
Simple Synchronization
Synch./Coordination
Problems...

Types of Semaphores....

Recall the “implementation” of a semaphore.

We can modify our semaphore implementation to prevent `s.count` from becoming negative. `s.count` can then be of type unsigned integer!!

Also, the modified version will ease the definition of two different types of semaphores:

- binary semaphores
- general counting semaphores

```
type semaphore...
```

```
typedef struct{  
    unsigned int count;  
    list_of_processes queue;  
  
} semaphore;
```

```
P(s):    if s.count ≥ 1 then s.count --  
         else block(s)
```

where `block(s)` places the process on `s.queue` and invokes the process scheduler.

```
V(s):    if s.queue is non-empty then  
         wakeup(s);  
         else s.count ++;
```

where `wakeup(s)` removes a process from `s.queue` and places it into the ready list.

binary vs. counting

The type of semaphore can generally be determined by inspecting the definition.

For a **binary semaphore**, the values are restricted to 0 and 1 (i.e., `s.count` is binary).

A **counting semaphore** can take on any integer value $v \geq 0$; i.e., `s.count` can be used to reflect the number of available resources

- Semaphores used to provide mutual exclusions are generally defined as binary semaphores:
 - typical definitions:
 - `semaphore mutex = 1;`
 - `semaphore mutex = 0;`
- Note: do not confuse counting semaphores that are initialized to 1 with a binary semaphore. There is a difference \rightarrow `s.count` is bounded to an max. value of 1.
- The exact type is often implicit in the type of use (eg. `mutex`)

simple synchronization

Counting semaphores are used to manage limited resources and corresponding access to them.

Binary semaphores are usually used to facilitate mutual exclusion; i.e., only a single process is allowed to access the CS.

Notation found in the literature may vary:

- $P(s) == \text{wait}(s) == \text{down}(s)$
- $V(s) == \text{signal}(s) == \text{up}(s)$

Note: Implementation of queuing determines if starvation of waiting processes can occur!!

Example: Mutex with Semaphores:

```
semaphore mutex = 1;
```

```
process  $p_i$ :
```

```
while(1){  
    P(mutex);  
    access CS;  
    V(mutex);  
    do the rest of the prog.  
}
```

What would happen if mutex was initialized to 0?

Classical Problems

The study of Process synchronization and coordination has led to a number of problem types:

- Mutual Exclusion problem
- Producer/Consumer problem
- Readers/Writers problem

We have already seen solutions to the MUTEX problem.

To demonstrate the Producer/Consumer problem, we consider the famous **bounded buffer problem**. First, a simple version:

- Let buffer B consist of n empty buffer slots;
- Let e be a counting semaphore that keeps track of how many empty slots are left in B.
- Let f be a counting semaphore that keeps track of how many full slots there are in B.
- Last but not least we use a binary semaphore b to provide mutex.
- The semaphores are initialized as:
 - $e = n$ (why ??)
 - $f = 0$ (why ??)
 - $b = 1$ (why ??)

..bounded buffer problem

Bounded Buffer Solution:

```
semaphore e = n, f = 0, b = 1;
cobegin
  producer: while(1){
    produce next element;
1:      P(e);
2:      P(b);
    add element to B;
3:      V(b);
4:      V(f); }
//
  consumer: while(1){
5:      P(f);
6:      P(b);
    remove element from B;
7:      V(b);
8:      V(e);
    consume the element; }
coend;
```

■ A few things to consider:

- Will this solution work for multiple consumers and producers?
- Does it matter in which order we call $P(e)$ - $P(b)$ and $P(f)$ - $P(b)$?
- Can we swap just lines 1&2 ??
- Can we swap just lines 7&8 ??
- Does it matter how buffer B is being implemented?

The readers/writers problem

One of the most studied problems in process synch. is the readers/writers problem.

Let $R = \{r_1..r_m\}$ a set of processes that read from a database. Let $W = \{w_1..w_k\}$ be the set of processes that write values to the same database.

The system is subject to the following constraints:

1. Readers and writers can never access the database simultaneously;
1. Only a single writer at a time is allowed in the database;
1. Multiple readers may access the database simultaneously;

We can distinguish 3 versions of the R/W problem:

1. Weak Reader Priority
2. Strong Reader Priority
3. Writer Priority

For the **weak reader priority**, an arriving writer waits until there are **no more active readers**.

Strong reader priority meets the constraints of weak reader priority, with the addition that a **waiting reader has priority over a waiting writer**.

For **writer priority**, an arriving reader waits until there are **no more active or waiting writers**.

Weak Reader Priority

Semaphore Solution for Weak Reader Priority:

shared var

```
int nreaders = 0;
```

```
semaphore: rmutex, wmutex = 1;
```

writer_i:

```
while(1){  
    ... // progr. outside the DB;
```

```
    P(wmutex);
```

```
    write data to the DB;
```

```
    V(wmutex);
```

```
}
```

reader_i:

```
while(1){  
    ... // prog. outside the DB;
```

```
    P(rmutex);
```

```
    if (nreaders == 0){
```

```
        nreaders++;
```

```
        P(wmutex);
```

```
    }
```

```
    else nreaders++;
```

```
    V(rmutex);
```

```
    read the DB;
```

```
    P(rmutex);
```

```
    nreaders--;
```

```
    if (nreaders == 0)
```

```
        V((wmutex);
```

```
        V(rmutex)
```

```
}
```


...weak reader discussion

- What are the things to be noted in the weak reader solution:
 - 1. multiple readers can enter the database.
 - 1. only the **first reader** blocks arriving writers ($P(wmutex)$).
 - 1. The **last reader** unblocks waiting writers and enables arriving writers to enter the DB ($V(wmutex)$)
- 1. we need to maintain a counter that keeps track of the number of readers that are in the DB.
- 1. does **nreaders** also count waiting readers?
- 1. why is this a **weak reader** priority solution??
- 1. What do you suggest to change to make it a **strong reader priority**?

Strong Reader Priority...

Semaphore Solution for Strong Reader Priority:

```
shared var
  int nreaders = 0;
  semaphore: rmutex, wmutex, srmutex = 1;
```

```
writeri:
```

```
while(1){
  ... // progr. outside the DB;
```

```
  P(srmutex);
  P(wmutex);
```

```
    write data to the DB;
```

```
  V(wmutex);
  V(srmutex);
```

```
}
```

```
readeri:
```

```
while(1){
  ... // prog. outside the DB;
```

```
  P(rmutex);
  if (nreaders == 0){
    nreaders++;
    P(wmutex);
  }
  else nreaders++;
```

```
  V(rmutex);
  read the DB;
  P(rmutex);
  nreaders--;
  if (nreaders == 0)
    V((wmutex);
  V(rmutex)
```

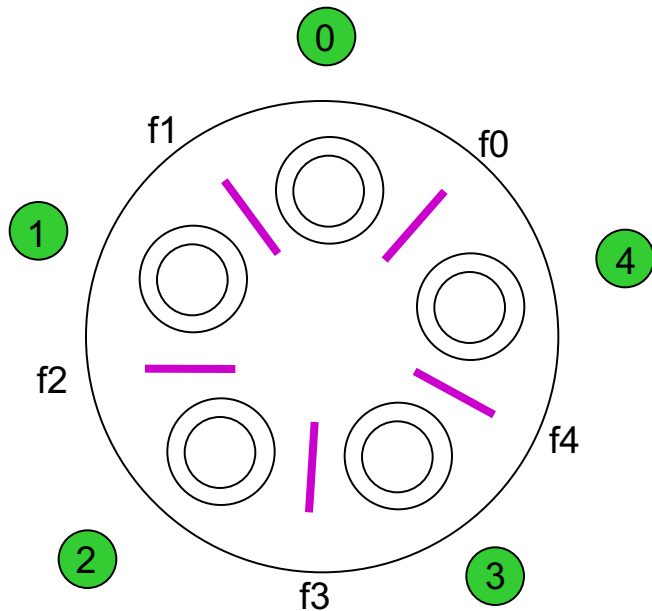
```
}
```

...strong reader discussion

- When the first reader enters the DB, $n_{\text{reader}} > 0$.
- All subsequent readers, i.e., those waiting on **rmutex** and those arriving can enter while $n_{\text{readers}} > 0$;
- The first arriving writer is blocked on **wmutex** while $n_{\text{readers}} > 0$.
- subsequent writers are blocked on **srmutex**.
- Arriving readers are blocked on **wmutex** while a writer is active.
- subsequent readers will wait on **rmutex**.
- A departing writer will signal **wmutex**, thereby unblocking the first reader.
- It will also signal **srmutex**, thereby unblocking a writer, which will then wait on **wmutex**.

..dinner-time: Dining Philosophers

5 philosophers, p_i ($1 \leq i < 5$), sit at a round table with a bowl of spaghetti in the center. In front of each philosopher is a plate. To the left and the right of the plate is one fork. In order to eat, a philosopher need to use 2 forks. (see picture)



Philosophers mostly think but at unspecified times they get hungry and wish to eat. To do that, philosopher p_i must pick up forks f_i and $f_{(i+1)\%5}$.

The problem: develop a fork acquisition protocol that is:

1. deadlock free
2. fair
3. maximally concurrent

..dining philosophers...

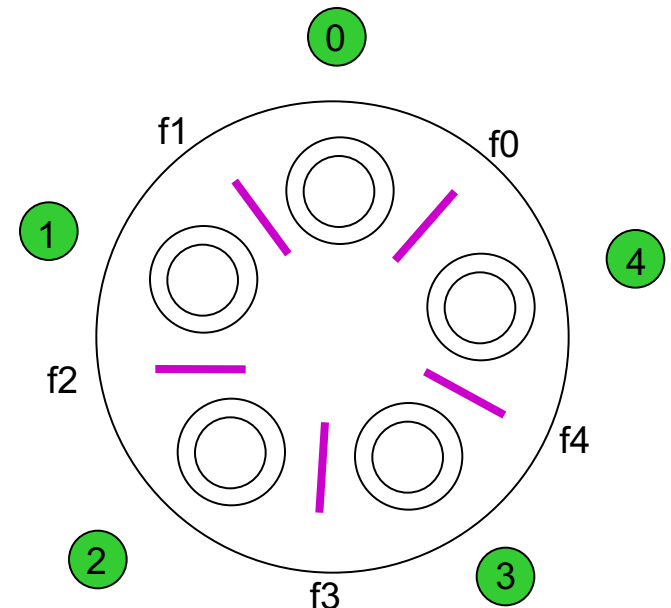
We can abstract the problem by the following code:

```
p(i):{  
    while(1){  
        think(i);  
        grab_forks(i);  
        eat(i);  
        return_forks(i);  
    }  
}
```

The key is in the implementation of *grab_forks(i)*, as it determines if p_i is allowed to eat.

a naïve semaphore solution:

```
grab_forks(i):  
    P(f[i]); P(f[(i + 1)%5]);  
  
return_forks(i):  
    V(f[i]); V(f[(i + 1)%5]);
```



...food for thought...

- What's wrong with the naïve solution?? → **DEADLOCK !!**
- How can this happen? Analyze!
- One way to break a circular wait is to implement a **global counter** (counting semaphore) that will **limit** the number philosophers to grab a fork **to $n-1$** .
- How does this help?
- Another way to break a circular wait is to implement one philosopher (i.e., p_1) to ask for the right fork before asking for the left. All others are asking for the left fork first.
- However, this solution violates the concurrency requirement as with one eating others might be blocked.
- How?? Let's analyze

another thought....

- We decide to divide the philosophers in an even and an odd group (aren't they all).
 - the even group picks up left before right fork
 - the odd group picks up right before left fork
- We still need to worry about starvation of individual philosophers.
- With regular semaphores, there is not much we can do, and the solutions are never satisfactory.
- We will revisit the dining philosophers later, with new coordination mechanisms.
- Examples:
 - AND synchronization
 - Monitors
- Think about why the dining philosopher problem is so important and what it contributes to research in process synchronization!