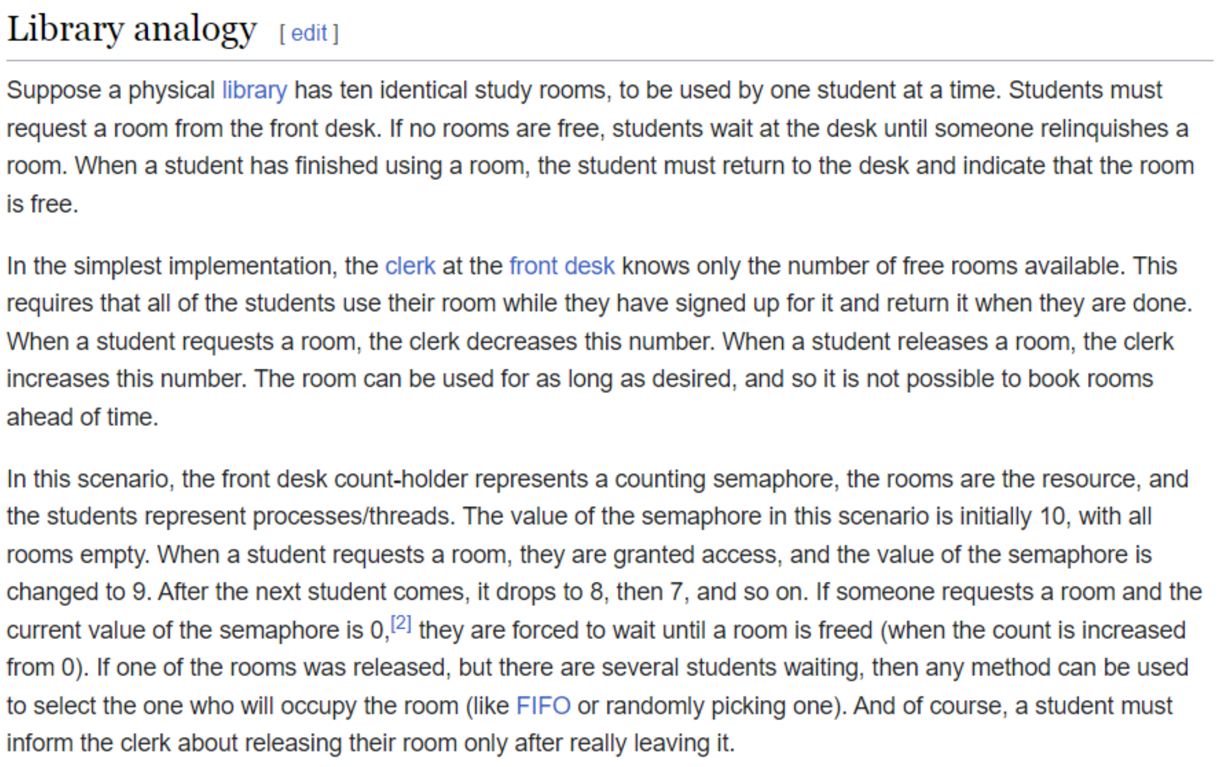
Synchronisation mechanisms

Operating system level synchronisation mechanisms

Semaphore

- A semaphore is a mechanism to control concurrency.
- Semaphores are used in scenarios when you need to limit the number of threads that can access a particular resource. For example, if you have a pool of connections to a database and you only want up to 10 threads to access the pool simultaneously, you can use a counting semaphore initialized to 10.



Library analogy [edit]

Suppose a physical library has ten identical study rooms, to be used by one student at a time. Students must request a room from the front desk. If no rooms are free, students wait at the desk until someone relinquishes a room. When a student has finished using a room, the student must return to the desk and indicate that the room is free.

In the simplest implementation, the clerk at the front desk knows only the number of free rooms available. This requires that all of the students use their room while they have signed up for it and return it when they are done. When a student requests a room, the clerk decreases this number. When a student releases a room, the clerk increases this number. The room can be used for as long as desired, and so it is not possible to book rooms ahead of time.

In this scenario, the front desk count-holder represents a counting semaphore, the rooms are the resource, and the students represent processes/threads. The value of the semaphore in this scenario is initially 10, with all rooms empty. When a student requests a room, they are granted access, and the value of the semaphore is changed to 9. After the next student comes, it drops to 8, then 7, and so on. If someone requests a room and the current value of the semaphore is 0,[2] they are forced to wait until a room is freed (when the count is increased from 0). If one of the rooms was released, but there are several students waiting, then any method can be used to select the one who will occupy the room (like FIFO or randomly picking one). And of course, a student must inform the clerk about releasing their room only after really leaving it.

- Semaphores are either considered *binary* or *counting*: the former may have value of 0 or 1, and the latter may have any non-negative value. Both binary and counting semaphores can only be increased or decreased by 1.

Generalized semaphore is a counting semaphore that can be increased/decreased by any number.

- Decrementing a semaphore which value is zero waits until the value becomes nonzero, that is until another process / thread increments the semaphore.
- A semaphore may be increased by a process which hasn't decreased it's value before, therefore a semaphore can have its value greater than the initial value.
- Two atomic operations are allowed on a semaphore:
    - P — decrease
    - V — increase
- Mutual exclusion using semaphore

```
shared sem: Semaphore := 1
P(sem);
critical_section();
V(sem);
remainder_section();
```

Mutexes and conditional variables

- Operations on mutexes:
    - `lock` — acquires mutex
    - `unlock` — releases mutex
    - `trylock` — acquires mutex only if it's free
- Operations on conditional variables:
    - `wait` — waiting
    - `signal` — wakes up one of the waiting threads
    - `broadcast` — wakes up all waiting threads
- POSIX mutex:

```
pthread_mutex_lock(pthread_mutext_t *m);
pthread_mutex_unlock(pthread_mutext_t *m);
pthread_mutex_trylock(pthread_mutext_t *m);
```

- POSIX conditional variable:

```
pthread_cond_wait(pthread_cont_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
pthread_cond_broadcast(pthread_cond_t *c);
```

Classical problems of synchronisation

Producer-consumer problem

- The producer generates data items and puts them into the buffer with a limited capacity.
- The consumer takes the items, thereby removes them from the buffer, leaving free space.
- At the producer side, the synchronisation problem appears when the buffer is full, so there is no free space for a new (already produced) item.
- At the consumer side, the synchronisation problem appears when the buffer is empty, so there is no item to be taken.
- A solution for one producer and one consumer using counting semaphores:

```
// Shared data:
const n: Integer := buffer_size;
shared buffer: array [0..n-1] of ItemType;
shared free: Semaphore := n;
shared occupied: Semaphore := 0;

// Producer
local i: Integer := 0;
local item: ItemType;
while ... do begin
    produce(item);
    P(free);
    buffer[i] := item;
    i := (i+1) mod n;
    V(occupied);
end;

// Consumer
local j: Integer := 0;
local item: ItemType;
while ... do begin
    P(occupied);
    item := buffer[j];
    j := (j+1) mod n;
    V(free);
    consume(item);
end;
```

- A solution for multiple producers and consumers using counting semaphores:

```
// Shared data
const n: Integer := buffer_size;
shared buffer: array [0..n-1] of ItemType;
shared free: Semaphore := n;
shared occupied: Semaphore := 0;
shared pmutex: Semaphore := 1;
shared cmutex: Semaphore := 1;

// Producer
shared i: Integer := 0;
local item: ItemType;
while ... do begin
    produce(item);
    P(free);
    P(pmutex);
    buffer[i] := item;
    i := (i+1) mod n;
    V(pmutex);
    V(occupied);
end;

// Consumer
shared j: Integer := 0;
local item: ItemType;
while ... do begin
    P(occupied);
    P(cmutex);
    item := buffer[j];
    j := (j+1) mod n;
    V(cmutex);
    V(free);
    consume(item);
end;
```

Readers-writers problem

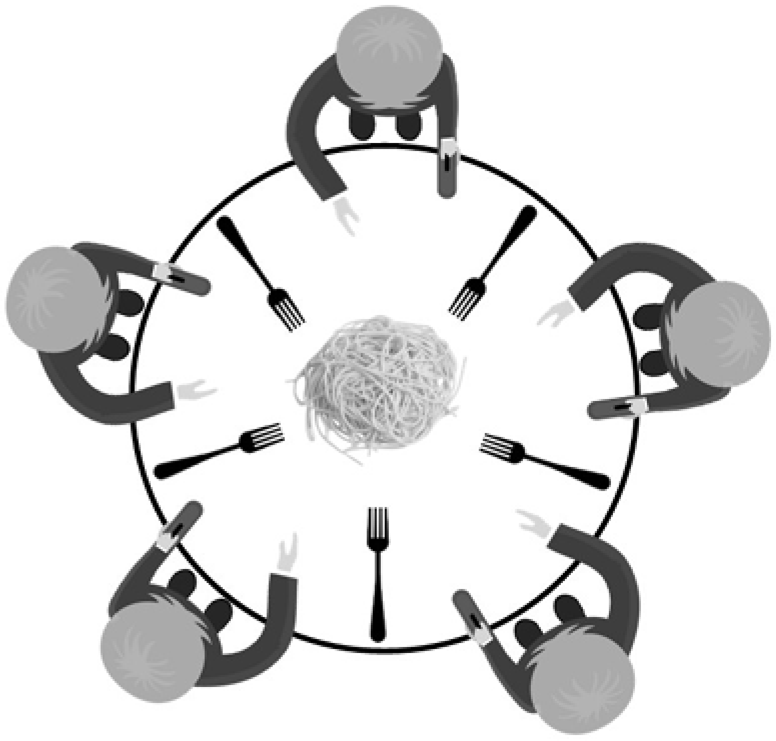
- There are two kinds of users — readers and writers that use a shared resource (for example a reading room or library).
- Readers can use the resource in the shared mode, i.e. multiple readers can occupy the reading room at the same time.
- Writers can use the resource in the exclusive mode, i.e. if there is a writer in the reading room then no one can occupy that room except him.
- The problem of synchronization is to prevent the user from entering the reading room in an incompatible mode.
- Solution to this problem using semaphores:

```
// Shared data
shared n_readers: Integer := 0;
shared mutex_r: Binary_Semaphore := true;
shared mutex_w: Binary_Semaphore := true;

// Reader
while ... do begin
    P(mutex_r);
    n_readers := n_readers + 1;
    if n_readers = 1 then P(mutex_w);
    V(mutex_r);
    reading();
    P(mutex_r);
    n_readers := n_readers - 1;
    if n_readers = 0 then V(mutex_w);
    V(mutex_r);
end;

// Writer
while ... do
    P(mutex_w);
    writing();
    V(mutex_w);
end;
```

Dining philosophers problem

- Five philosophers sit at a round table with a bowl of spaghetti.
- Each philosopher alternatively thinks and eats.
- To eat spaghetti the philosopher needs two forks.
- Forks are placed between each pair of adjacent philosophers, so each of the five forks is shared by two philosophers.

- Each fork is used by at most one philosopher at a time, therefore at most one of two neighbours can eat.
- The problem is to synchronise the philosophers so that each of them can eventually eat something, avoiding unnecessary waiting and following the rules of access to forks.



- Solution to this problem using semaphores:

```
// Shared data
shared allow: Semaphore := 4;
shared fork: array[0..4] of Binary_Semaphore := true;

// Philosopher number i (i = 0,1,2,3,4)
while ... do begin
    thinking();
    P(allow);
    P(fork[i]);
    P(fork[(i+1) mod 5]);
    eating();
    V(fork[i]);
    V(fork[(i+1) mod 5]);
    V(allow);
end;
```

Sleeping barbers problem

- There is a barbershop with p chairs in the waiting room and n barber chairs used by barbers to serve their customers.
- When there is a customer in the shop, they wake up a barber, then the barber finds a free chair to serve the customer.
- If no free barber chair is available, the customer takes a seat in the waiting room.
- If there is no free chairs in the waiting room, the customer leaves the barbershop.
- The issue here is to synchronize the barbers and the customers, i.e. to lead to the simultaneous state of a barber serving a customer and the customer being served by the barber.
- Solution to the problem using semaphores

```
// Shared data
const p: Integer = number_of_waiting_room_chairs;
const n: Integer = number_of_barber_chairs;
shared n_wait: Integer := 0;
shared mutex: Binary_Semaphore := true;
shared customer: Semaphore := 0;
shared barber: Semaphore := 0;
shared chair: Semaphore := n;

// Customer
while ... do begin
    P(mutex);
    if n_wait < p then
    begin
        n_wait := n_wait + 1;
        V(customer);
        V(mutex);
        P(barber);
        served();
    end
    else V(mutex);
end;

// Barber
while ... do begin
    P(customer);
    P(chair);
    P(mutex);
    n_wait := n_wait - 1;
    V(barber);
    V(mutex);
    serving();
    V(chair);
end;
```

High-level language constructs for synchronisation

Monitor

- The monitor is a programming language construct that provides equivalent functionality to that of semaphores and that is easier to control.
- A typical monitor consists of:
    - local data accessible only by the monitor's procedures
    - definition of public procedures that are the only means to access the data

- The monitor ensures that only one process can be active within the monitor at a time.
- A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor.
- A process executing a monitor procedure may be suspended by executing the wait command for the conditional variable. A suspended process can be woken up by another process which invoked the signal command on the same conditional variables.
- A monitor can be used to implement so called bounded cyclic buffer — a data structure that uses a fixed size buffer as if it were connected end-to-end like in a circle.

  ```
  type Buffer = monitor
      pool: array [0..n-1] of ItemType;
      in, out, count: Integer;
      empty, full: Condition;

      procedure entry put(item: ItemType);
      begin
          if count = n then full.wait;
          pool[in] := item;
          in := (in+1) mod n;
          count := count + 1;
          empty.signal;
      end;

      procedure entry get(var item: ItemType);
      begin
          if count = 0 then empty.wait;
          item := pool[out];
          out := (out+1) mod n;
          count := count - 1;
          full.signal;
      end;
  ```

```
    begin
        in := 0;
        out := 0;
        count := 0;
    end.
```

- The monitor can also be used to solve the producer-consumer problem

```
// Shared data
shared buf: Buffer;

// Producer
local item: ItemType;
while ... do begin
    produce(item);
    buf.put(item);
end;

// Consumer
local item: ItemType;
while ... do begin
    buf.get(item);
    consume(item);
end;
```

- The monitor can also be used to solve the readers-writers problem:

```
// This is only a sketch of solution !!!
// Shared data
shared reading_room: Monitor;

// Reader
reading_room.reader_entry();
reading();
```

```
reading_room.reader_exit();

// Writer
reading_room.writer_entry();
writing();
reading_room_writer_exit();
```