Operating System Process and Resource Management

Inter-Process Communication

SUPINFO

Course Objectives

- ✓ Understand the problem of concurrent access
- ✓ Understand the resolution algorithms
- ✓ Find optimized solutions

Course Plan

1. Locking and Deadlocking
2. Remove Active Waiting

1. Locking and Deadlocking

1. Locking and Deadlocking

Introduction

- Modeling real systems in a sequential program must:
  - Define in its code the management of the components of its environment
  - Be able to modify the program when new activities are included
  - Be able to modify the program when changing the architecture (number of CPU, *etc*.)

- The real world is made of competing programs that work together, for example:
  - A switch uses information from multiple sensors simultaneously and provides orders to multiple actors
  - A reservation system considers that there are several agencies in parallel

- Synchronizing processes can lead to problems such as **locking**, **deadlocking** or **starvation**

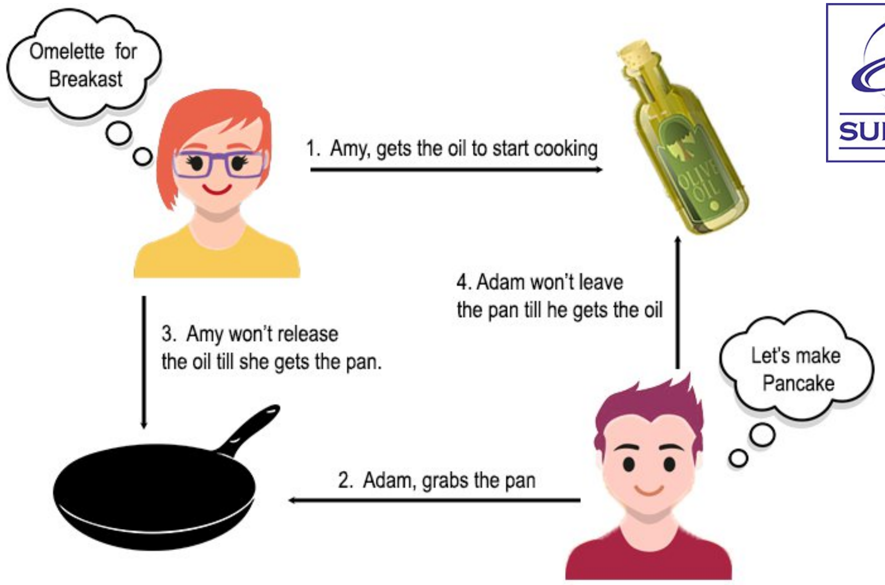
1. Locking and Deadlocking

Introduction

- Our current programs are multi-tasking and inter-tasking, they can:
  - Share and exchange information
  - Synchronize to respect the use and protection of data
- We have 2 models of systems:
  - Centralized, that communicate and synchronize using a common memory
  - Distributed, that communicate and synchronize by sending messages
- **Multi-tasking** = on a single processor
- **Parallel** = several processors with a common memory
- **Distributed** = remote machines (networked) and without a common memory

1. Locking and Deadlocking

Examples



User	Action	Output
Bob	getState(room1)	free
Alice	getState(room1)	free
Bob	book(room1)	OK
Alice	**book(room1)**	**OK**

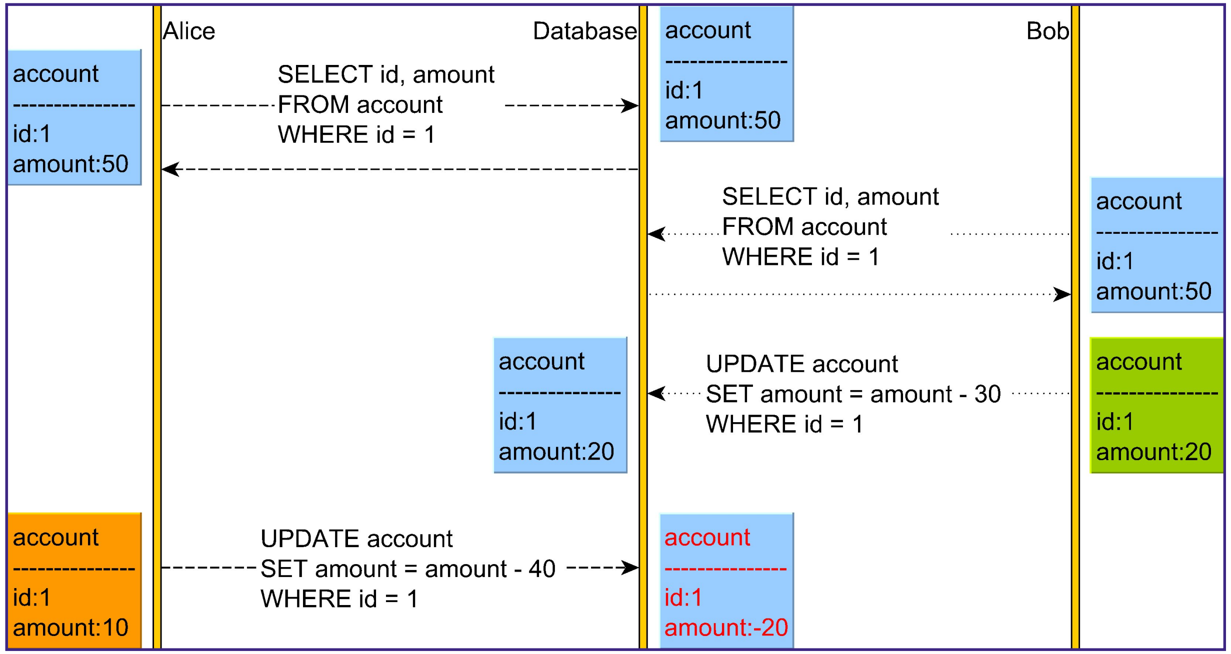
2 reservations for the same room!

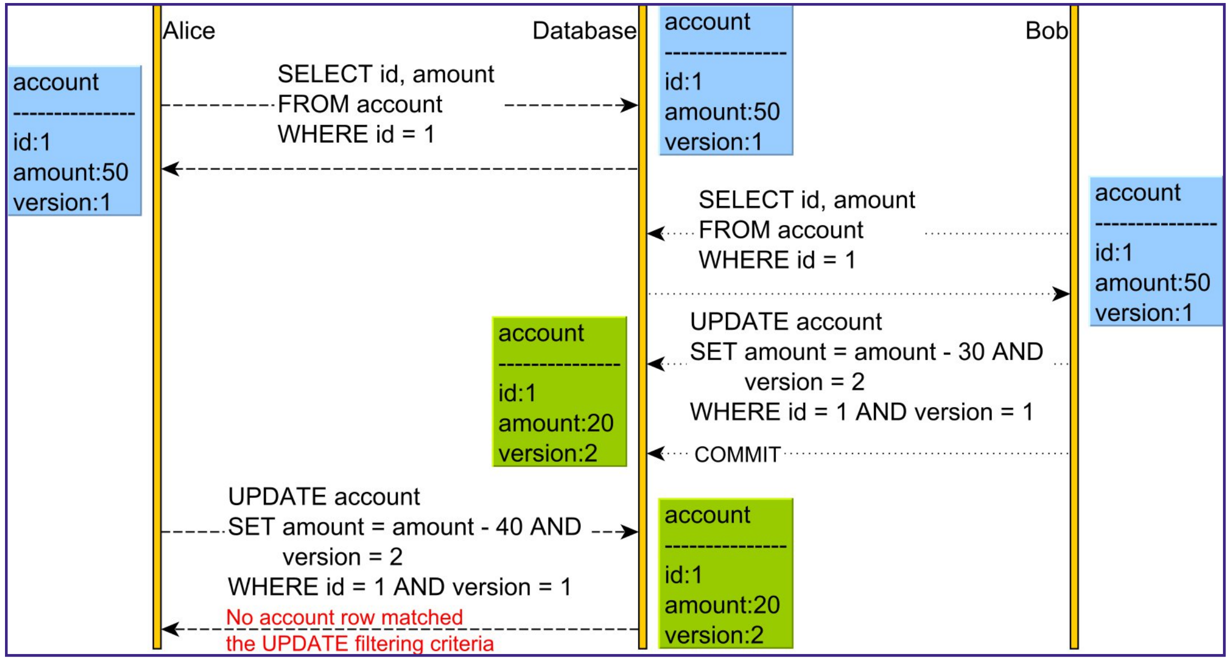
User	Action	Output
Bob	getState(room1)	free
Alice	getState(room1)	free
Bob	book(room1)	OK
Alice	**book(room1)**	**KO**

Updated state or back check

1. Locking and Deadlocking

Examples





1. Locking and Deadlocking

Critical section

- The previous examples show that data protection systems are needed, we denote several (like for centralized systems):
  - **Test and Set**, low-level (processor)
  - **Interrupt masking**, dangerous (kernel)
  - **Semaphore**, common and simple
  - **Monitor**, high-level (language level)

- All these systems act on data that must be protected (program level)

- The **critical section** (CS) is the part of a program whose execution should not **intermingle** with other programs
  - Once a task enters a critical section, it must be allowed to complete that section without allowing other tasks to execute the same data

1. Locking and Deadlocking

Problem generalization

- When a process manipulates a sensitive data, we say that it is in a **critical section**

- When a process does not manipulate sensitive data and it goes out of the critical section, we say that it is in the **remainder section** (RS)

Repeat
- Entry section
- **Critical section**
- Exit section
- **Remainder section**

Forever

1. Bob requests a room reservation

   Entry section

Critical section

2. Database states that room1 is available
3. Room1 is allocated to Bob and tagged as reserved

   Exit section

1. Locking and Deadlocking

Relevance

- We find this problem in:
  - Databases
  - Resource sharing (files, connections, networks, *etc.*)
  - Automats (like ATM)
  - Hardware (hard disk, *etc.*)
  - Software (MMORPG, client/server, *etc.*)
  - *etc.*

1. Locking and Deadlocking

Necessary criteria for a valid solution

- **Mutual exclusion**: at anytime, at most one process can be in a critical section for a given variable
- **Non-interference**: if a process stops in its remainder section, this should not affect the other processes
- The problem in the critical section is to find a mutual exclusion algorithm
  - Software-provided solutions
  - Hardware-provided solutions
  - OS-provided solutions

1. Locking and Deadlocking

First algorithmic solution

- We start with 2 processes: P0 and P1

- When we consider the task P***i***, P***j*** will always refer to another task (***i*** ≠ ***j***)

- This software solution will be algorithmically composed of processes giving each other the turn:
  1. The shared variable token is initialized to 0 or 1
  2. The CS of P***i*** is executed if and only if token = ***i***
  3. P***i*** is waiting if P***j*** is in CS

```
Repeat #Pi process
    While(token ≠ i) { #wait }
    CS
    token = j
    RS
Forever
```

1. Locking and Deadlocking

First algorithmic solution

```
Repeat #P0 process
    While(token ≠ 0) { #wait }
    CS
    token = 1
    RS
Forever
```

```
Repeat #P1 process
    While(token ≠ 1) { #wait }
    CS
    token = 0
    RS
Forever
```

1. Locking and Deadlocking

Courtesy excess

- Generalization to *n* processes: each time, before a process can enter its critical section, it must wait until all the others have had this opportunity
  - A Boolean variable per process: ***flag[0]*** and ***flag[1]***
  - P***i*** signals that it wants to execute its CS: ***flag[i] = true***

```
Repeat #Pi process
    flag[i] = true
    While(flag[j] = true) { #wait }
    CS
    flag[i] = false
    RS
Forever
```

1. Locking and Deadlocking

Courtesy excess

```
Repeat #P0 process
    flag[0] = true
    While(flag[1] = true) { #wait }
    CS
    flag[0] = false
    RS
Forever
```

```
Repeat #P1 process
    flag[1] = true
    While(flag[0] = true) { #wait }
    CS
    flag[1] = false
    RS
Forever
```

1. Locking and Deadlocking

Dekker algorithm

- Initialization: c**0**, c**1**, token = **0** (previous locking broken thanks to the token)

```
Repeat #P0 process
    NCS0: #non-critical section
        c0 = false
        While(c1 = false) {
            c0 = true
            While(token = 1) { #wait }
            c0 = false
        }
    CS0: #critical section
        token = 1
        c0 = true
Forever
```

```
Repeat #P1 process
    NCS1: #non-critical section
        c1 = false
        While(c0 = false) {
            c1 = true
            While(token = 0) { #wait }
            c1 = false
        }
    CS1: #critical section
        token = 0
        c1 = true
Forever
```

1. Locking and Deadlocking

Dekker algorithm

- The correct operation of Dekker's algorithm depends on the fact that each process always ends up progressing (fairness)

- A simple fairness assumption, sufficient to reason about the Dekker algorithm, is as follows: any process that has the ability to execute an instruction will always end up doing so

- Among the properties we specify for a concurrent program, we distinguish the following 2 categories:
  - **Safety properties**: these are properties that express that certain undesirable states are never reached, they do not depend on the use of a fairness assumption
  - **Liveliness properties**: these express that desirable states will inevitably be reached, they are usually only true in the presence of a fairness assumption

1. Locking and Deadlocking

Test-and-set instruction

Non-divisible and unbreakable atomic instruction

```
bool TestAndSet(int &i)
{
    If(i = 0) {
        i = 1
        return true
    } else {
        return false
    }
}
```

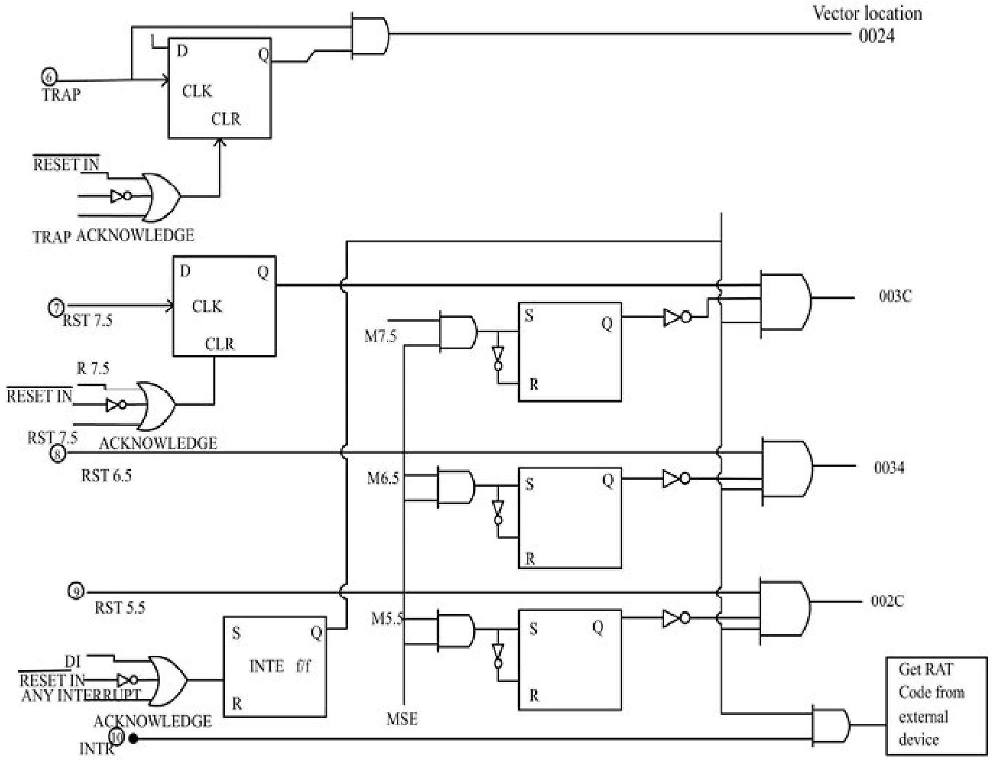
- An algorithm using test-and-set for mutual exclusion:
  - Shared variable **i** is initialized to 0
  - It is the first P**i** (which sets **i** to 1) that enters CS

```
Repeat #Pi process
    While(TestAndSet(i) = false) { #wait }
    CS
    i = 0
    RS
Forever
```

1. Locking and Deadlocking

Need to use other methods

- The major drawback of software solutions is **active waiting**
- Hardware solutions are for mutual exclusion
- **Interrupt masking**: a process that wants to enter a critical section inhibits interrupts to conserve the processor until it restores them



1. Locking and Deadlocking

Exercise

- Define uniprogramming and multiprogramming
- Why is a process table necessary in a time-sharing system?
- Is a process table also required in a personal system where only one process exists, with access to the entire machine during its execution?

1. Locking and Deadlocking

Questions

2. Remove Active Waiting

2. Remove Active Waiting

Semaphore

- Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes, it is an elementary structure of synchronization (limiting active waiting)

- A semaphore S is a set containing:
  - An internal (integer) counter: **c(S)**
  - A process queue : **q(S)**
  - 2 atomic operations/primitives (non-breakable execution): **P(S)** and **V(S)**
  - An atomic operation of initialization (and creation) of the counter

- A semaphore S is said to be binary if its counter remains lower or equal to 1

2. Remove Active Waiting

Semaphore

- **P(S)** operation is also called wait, sleep, or down operation, and **V(S)** operation is also called signal, wake-up, or up operation

- Let p be the process that executes **P(S)** or **V(S)** with **c(S)** arbitrarily initialized:

P(S)

```
c(S) = c(S)-1
If(c(S)<0) {
    state(p) = locked
    input(p, q(S))
}
```

V(S)

```
c(S) = c(S)+1
If(c(S)≤0) {
    output(p, q(S))
    state(p) = eligible
    input(p, q(eligible))
}
```

2. Remove Active Waiting

Semaphore

- Binary (**mutex**) semaphore:
  - The purpose of exclusion for each process is to protect access to a single resource (memory, disk, *etc.*)
  - The counter **c(S)** is initialized to 1

```
P(S) #Takes the critical section
CS
V(S) #Releases the critical section
```

- Counting (**synchronization**) semaphore:
  - The purpose of synchronization is that one process must wait for another to continue (or start) its execution
  - The counter **c(S)** is initialized to 0

P**0**

```
job0()
V(S) #Wakes up P1
```

P**1**

```
P(S) #Waits P0
job1()
```

2. Remove Active Waiting

Rendezvous

- The goal of the rendezvous (generalization of the previous problem) is that a process waits for *n* other processes to be at a specific execution to continue its execution:
  - **Sync**, a semaphore initialized to 0: *Sync = S(0)*
  - **Mutex**, a semaphore initialized to 1: *Mutex = S(1)*
  - **Wait**, a semaphore initialized to 0: *Wait = S(0)*
  - A counter (integer) **nb** initialized to 0: *nb = 0*
  - An indicator (integer) the **n** processes waiting: *n*

P**i**

```
P(Mutex)
nb = nb+1
If(nb = n) {
    V(Sync)
}
nb = 0
V(Mutex)
P(Wait)
```

P*rdv*

```
…
P(Sync)
V(Wait, n)
…
```

2. Remove Active Waiting

Deadlocking risk

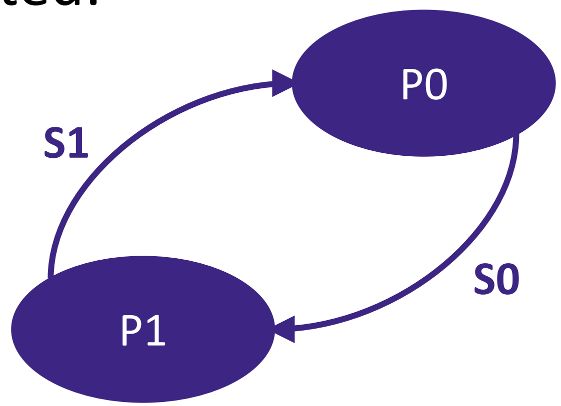
- For the deadlocks we have:
  - S0 and S1, 2 mutex semaphores
  - **P()** always executed in the same order

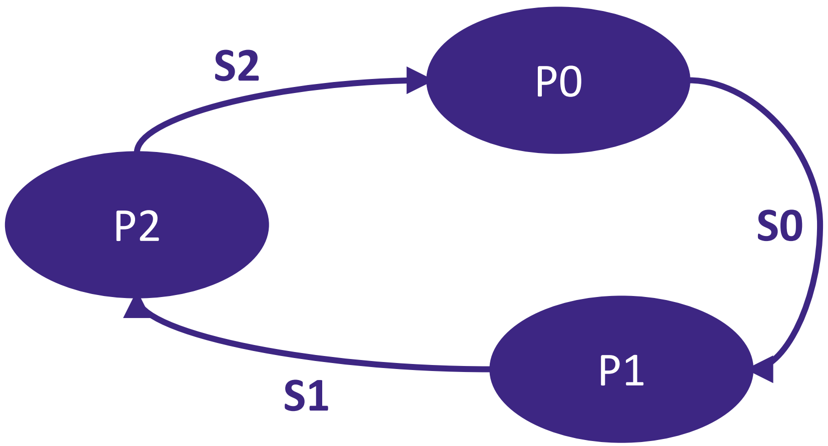
P**i**	P**j**
P(S0)	P(S1)
P(S1)	P(S0)
CS	**CS**
V(S1)	V(S0)
V(S0)	V(S1)

2. Remove Active Waiting

Deadlocking risk

- To prevent and respond to deadlocking, the calls between the processes must be represented:



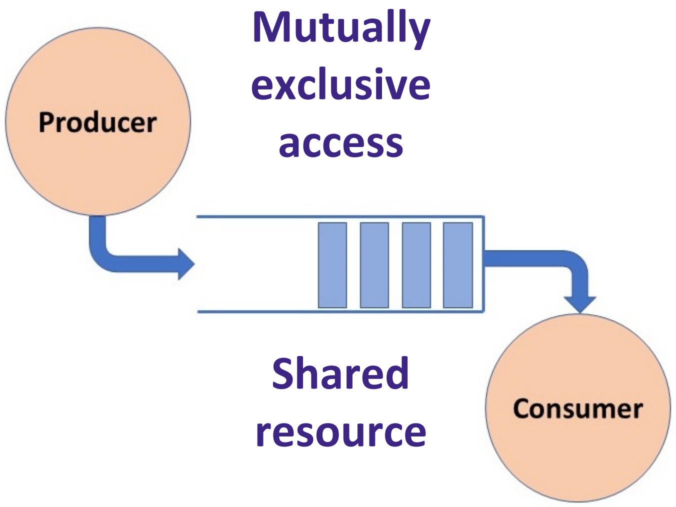


- If there is a call from one process to another and vice versa, then there is a deadlock

- To limit this locking, we must add the operations **V()** to release missing resources

2. Remove Active Waiting

Producer-consumer problem



- 1-slot model:
  - The producer and the consumer are 2 cyclic processes

Producer	Consumer
produce(messageP) deliver(slot, messageP)	remove(slot, messageC) consume(messageC)

2. Remove Active Waiting

Producer-consumer problem

- Synchronized producer-consumer model with a shared array (deliver/remove) with 1 slot; we use 2 semaphores, **full** and **empty**, initialized to 0 and 1 (**empty** indicates if the slot of the shared array is empty, inversely for **full**)

Producer	Consumer
P(**empty**) produce(messageP) deliver(slot, messageP) V(**full**)	P(**full**) remove(slot, messageC) consume(messageC) V(**empty**)
or	**or**
produce(messageP) P(**empty**) deliver(slot, messageP) V(**full**)	P(**full**) remove(slot, messageC) V(**empty**) consume(messageC)

2. Remove Active Waiting

Monitor

- The monitor is an advanced synchronization tool, introduced by Brinch Hansen in 1972-73 and Hoare in 1974; it consists of :
  - State variables
  - Internal procedures
  - External procedures (entry points)
  - Conditions
  - Synchronization primitives

- An example of a monitor for incrementing and decrementing a variable *i*

Monitor incr_decr

```
incr_decr: monitor
    var i: integer

    procedure increment
    begin
        i = i+1
    end

    procedure decrement
    begin
        i = i-1
    end

    begin
        i = 0
    end
end incr_decr
```

2. Remove Active Waiting

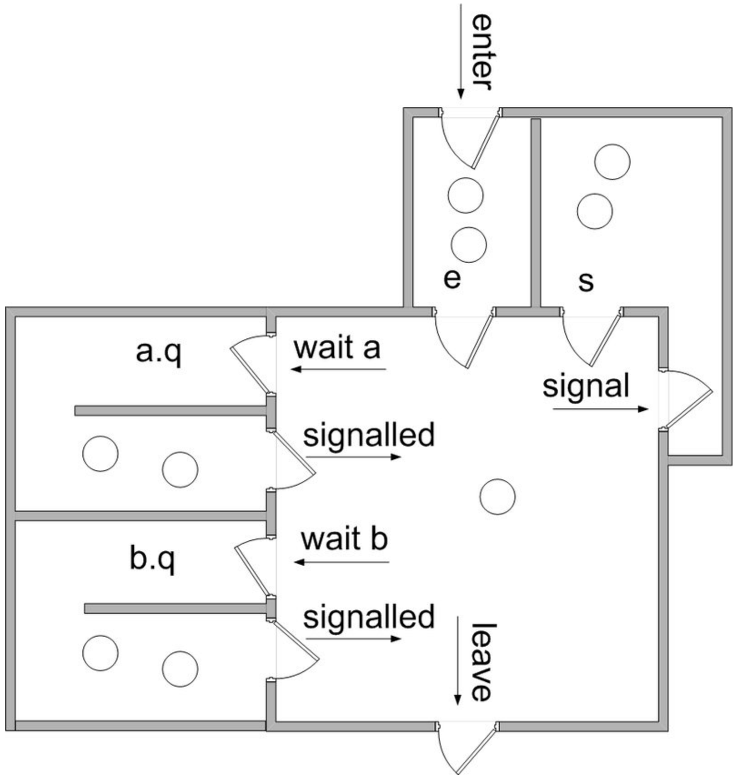
Monitor

- Each monitor procedure is executed in mutual exclusion
- The access to the monitor is done in mutual exclusion, except if a process executes the **wait** primitive
- Special monitor instructions: the **empty**, **wait** and **signal** primitives on condition type variables
- Consider **c** as a condition:
  - **c.wait**: lock the process and put it in waiting for the condition **c**
  - **c.empty**: *true* if no process is waiting for condition **c**, *false* otherwise
  - **c.signal**: if not **c.empty** then wake up a process waiting for condition **c**

2. Remove Active Waiting

Monitor

- Inside the monitors each monitor has a global waiting queue and each variable condition **c** acts on a waiting queue:
  - **c.wait**: put the process in the waiting queue associated with condition **c**
  - **c.empty**: check if the waiting queue associated with condition **c** is empty
  - **c.signal**: take the next process out of the queue associated with the condition **c**



2. Remove Active Waiting

Monitor

- Only one process is active at a time within a monitor, so it is not autonomous to implement the **signal** primitive

- A monitor is usually implemented with semaphores

- Such a system is thus built by successive levels of abstraction, and a level *i* is implemented by the level *i-1* primitives

2. Remove Active Waiting

Exercise

- Write a monitor to perform the rendezvous of **n** processes: the purpose of this monitor is to allow the waiting of **n** processes according to the condition of the number of processes waiting and the processes available

- Solve the producer-consumer problem with monitors using a single slot

2. Remove Active Waiting

Questions

Operating System Process and Resource Management

Inter-Process Communication

Thank you for your attention