Operating Systems Synchronization

An Introduction

Mindset

Think: Listen, read, explore, try, think

Motivation for Synchronization

- Cooperating Computers/Systems/Processes/Threads/Tasks
  - Communication
  - Information/Resourse Sharing
  - Modularity/Convenience
  - Economy/Performance/Responsiveness/Scalability

Synchronization Overview

- Characterizing Cooperating Processes/Synchronization
- Sections (e.g. Critical Section)
- Some Mechanisms for dealing with Critical Section
  - Mutex
  - Semaphore
  - Monitor

- Other topics
  - Peterson's Solution
  - Classical Problems in Synchronization: Dining Philosopher, Bounded Buffer Problem, Producer-Consumer Problems

Issues of Cooperating Process (Part 1)

- Race Condition
  - When several processes manipulate same data concurrently and outcome depends on order of execution.

- Can you give examples of race conditions?

Issues of Cooperating Process Example

- Race Condition Example
  - When several processes manipulate same data concurrently and outcome depends on order of execution.

Banker Makes a Deposit

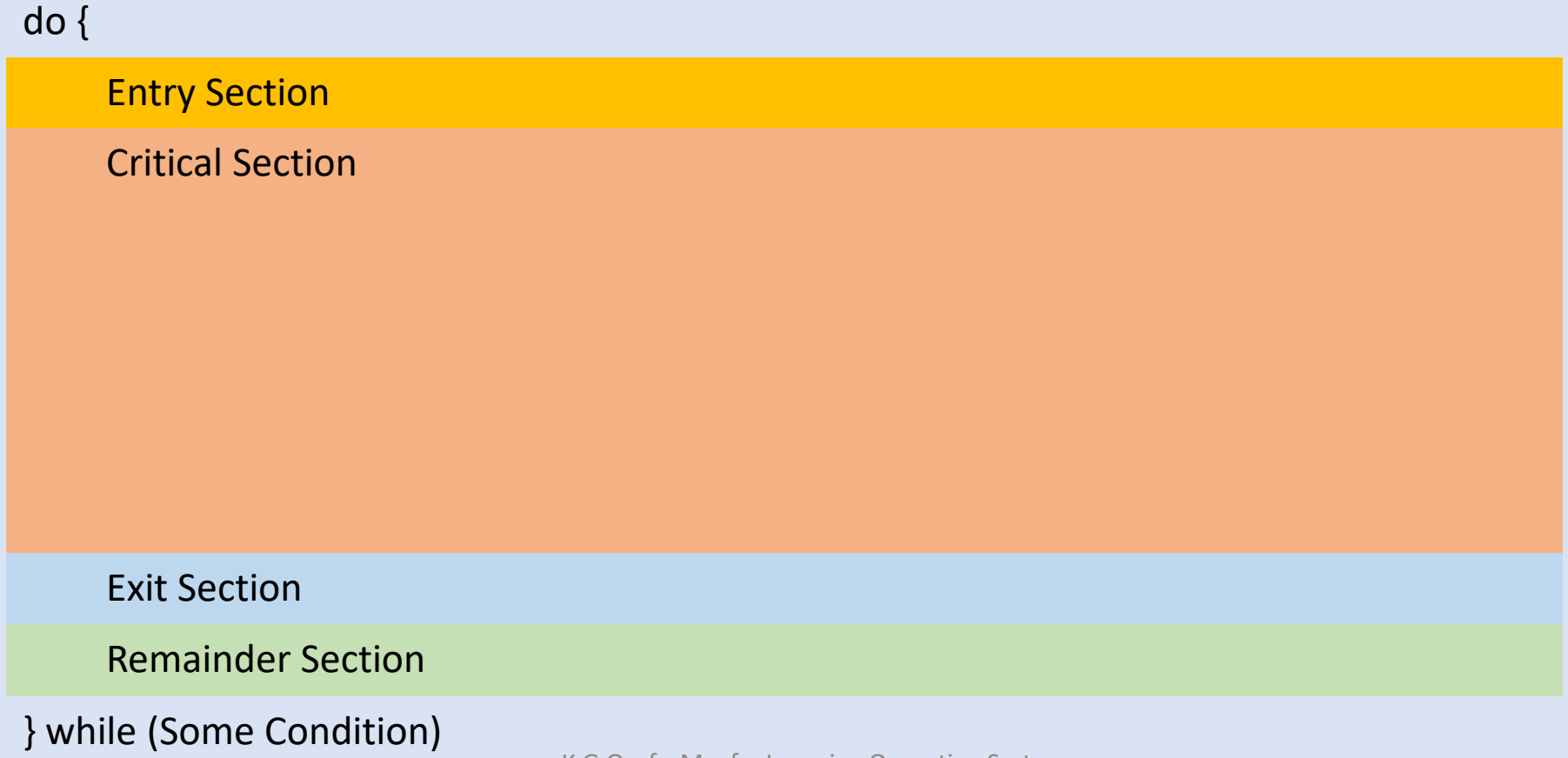
Step 1: Banker reads balance from an account

Step 2: Banker calculates the new balance
(e.g. adding deposit/subtracting withdrawal)

Step 3: Banker stores or writes amount

Characterizing Cooperating Process

- Entry Section, Critical Sections, Exit Section and Remainder Section



Characterizing Cooperating Process Example

- Entry Section, Critical Sections, Exit Section and Remainder Section

do {	//Banker Makes a Deposit
Entry Section	…. some work before starting the transaction
Critical Section	Step 1: Banker reads balance from an account Step 2: Banker calculates the new balance (e.g. adding deposit/subtracting withdrawal) Step 3: Banker stores or writes amount
Exit Section	… some work after starting the transaction
Remainder Section	
} while (SomeCondition)	

Issues of Cooperating Process (Part 2)

- Race Condition
  - When several processes manipulate same data concurrently and outcome depends on order of execution.
- Characterizing parts of code
  - Entry Section, Critical Section, Exit Section, Remainder Section
- Critical Section Requirements
  - Mutual Exclusion
  - Progress
  - Bounded Waiting

Approaches to handling Critical Sections by OS

- Preemptive Kernel vs Non-preemptive Kernels
  - Preemptive Kernel allows process to be preempted while in kernel mode
  - Non-preemptive Kernel allows process to run unless it blocks or yield control of CPU
- Cooperating Process/Tasks and Race Conditions in Kernels
- Behavior of Preemptive and Non-Preemptive Kernels
  - Preemptive Kernels can provide responsiveness
  - Non-preemptive Kernel avoids race conditions

Concepts in Synchronization

- Locking
- Atomicity
- Busy Waiting
- Spin Lock

Characterizing Cooperating Process

Shared Data: shared_item; Boolean item_available

do {

Entry Section	while (!**item_available**) {
	}
	item_available = false;

Critical Section	**// access and or modify shared item**
Exit Section	**Item_available** = true;
Remainder Section	

} while (SomeCondition)

Characterizing Cooperating Process

Shared Data: currency account_balance; Boolean account_available = true

do {	//Banker Makes a Deposit	

Entry Section	while (!**account_available**) {
	}
	account_available = false;
		…. some work before starting the transaction

Critical Section	var balance = **account_balance** balance = balance + deposit_amount **account_balance** = balance	Step 1: Banker reads balance from an account Step 2: Banker calculates the new balance Step 3: Banker stores or writes amount
Exit Section	**account_available** = true;	… some work after starting the transaction
Remainder Section		…. some work before starting the transaction
} while (SomeCondition)		

Characterizing Cooperating Process

Shared Data: currency account_balance; Boolean account_available

do {	//Banker Makes a Deposit
Entry Section	**lock_account**();
Critical Section	var balance = **account_balance** balance = balance + deposit_amount **account_balance** = balance
Exit Section	**release_account();**
Remainder Section	
} while (SomeCondition)	

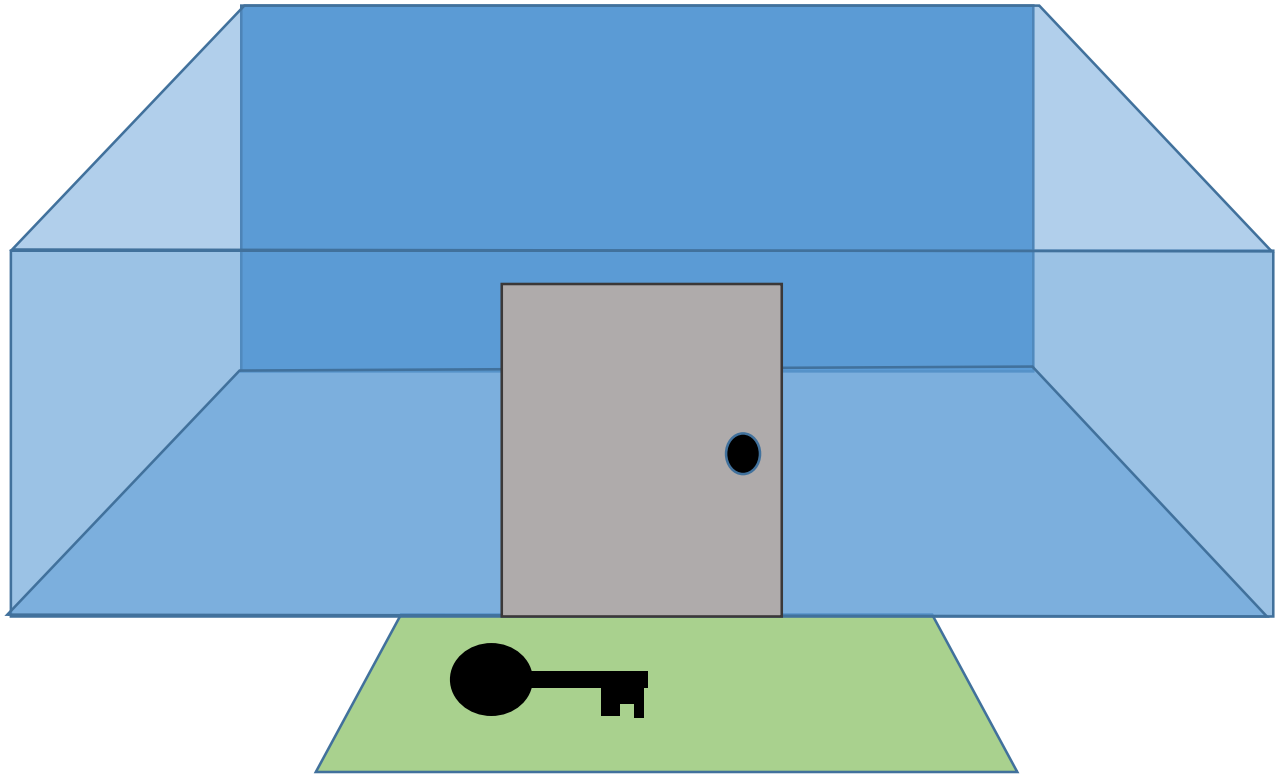
```
function lock_account()
{
        while (!account_available) {
        }
        account_available = false;
}

function release_account()
{
        account_available = true;
}
```

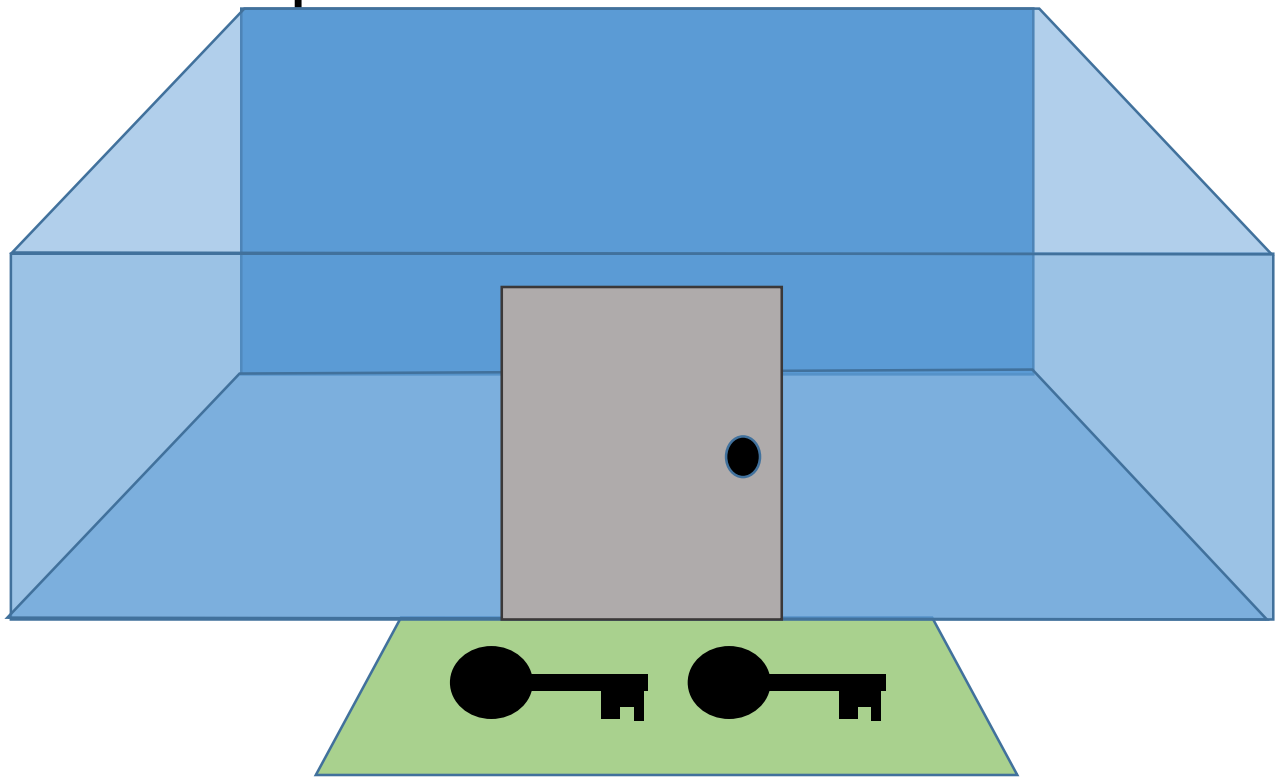
Synchronization Techniques

- Mutex Locks
  - A mutually exclusive lock with methods to
    - lock()/acquire()/hold()
    - unlock()/release()
- Semaphores
  - Integer variable accessible through
    - wait()/decrement()/acquire()
    - signal()/increment()/release()
- Monitors

Scenario: Mutex



Scenario: Semaphore



Scenario: Monitor