MONASH University
Information Technology

FIT3142 Distributed Computing

Topic 7: Concurrency, Parallelism, Synchronisation, Deadlocks, Safety

Dr Carlo Kopp
Faculty of Information Technology
Monash University

© 2008 – 2016 Monash University

Why Study Synchronisation, MUTEX, Deadlock?

- Synchronisation, mutual exclusion and deadlocks are very common problems which arise when multiple entities are competing for access to shared resources; All three are usually covered in units or textbooks dealing with operating systems, where the shared resources are CPU/core time and shared main memory, and sometimes I/O devices;
- In a distributed / parallel system the problem is complicated by the need to allow for transmission delays in passing messages through the IPC between tasks (processes), which makes some solutions used in operating systems problematic;
- Practical skills: Failure to properly address these problems will result in often catastrophic application failures or bugs, often very difficult to isolate and correct.

Reading / Overview

- **Coulouris et al, Distributed Systems, 5E:**
  - Ch.14 Time and Global States
  - Ch.15 Coordination and Agreement
  - Ch.16 Transactions and Concurrency Control
  - Ch.17 Distributed Transactions
- **Synchronization in Distributed systems**
  - Clock synchronization
  - Event Ordering
  - Distributed Mutual Exclusion
  - Deadlock
  - Election Algorithm
  - Distributed Transactions

What is Synchronisation?

- ***World English Dictionary:*** **synchronize *or* synchronise (ˈsɪŋkrəˌnaɪz):**
- ✓ **to occur or recur or cause to occur or recur at the same time or in unison**
- ✓ **to indicate or cause to indicate the same time: *synchronize your watches***
- ✓ **(*tr*) *films* to establish (the picture and soundtrack records) in their correct relative position**
- ✓ **(*tr*) to designate (events) as simultaneous**
- **Synchronisation is about making two or more entities achieve a known state or condition at a known or identical time – for instance a receiver (e.g. task) must be ready before it can accept a message from a transmitter (e.g. task).**

Synchronisation in Distributed Systems

- A Distributed System consists of a collection of distinct processes that are spatially separated and run concurrently;
- In systems with multiple concurrent processes, it is economical to share the system resources;
- Sharing may be cooperative or competitive;
- Both competitive and cooperative sharing require adherence to certain rules of behavior that guarantee that correct interaction occurs – otherwise chaotic behaviour may arise;
- The rules of enforcing correct interaction are implemented in the form of synchronization mechanisms;
- Synchronisation mechanisms may be part of an operating system, or communications mechanism like a library;

Issues in implementing synchronization

- **In single CPU systems, synchronization problems such as mutual exclusion can be solved using semaphores and monitors. These methods rely on the existence of shared memory, which can be accessed very quickly by all tasks;**
- **We cannot use semaphores and monitors in distributed systems since two processes running on different machines cannot expect to have access to any shared memory;**
- ***In a distributed system there are always finite time delays from messages to travel from one process to another, so any mechanism we use must account for these delays;***
- **Even simple matters such as determining if one event happened before another event require careful thought.**

Issues implementing synchronization in DS/PS

- **In distributed systems, it is usually not possible and often not desirable to collect all the information about the system in one place and synchronization among processes is difficult due to the following features of distributed systems:**

✓ **The relevant information is scattered among multiple machines.**

✓ **Processes make decisions based only on *local* information.**

✓ **Any single point of failure in the system should be avoided.**

✓ **No common clock or other precise global time source exists.**

- ***Synchronisation in distributed/parallel systems requires unique algorithms which account for the unique behaviours in such systems, especially time delays.***

Time in Distributed Systems: Why?

- **External reasons: We often want to measure time accurately**
  - ■ For billings: How long was computer X used?
  - ■ For legal reasons: When was credit card W charged?
  - ■ For traceability: When did this attack occurred? Who did it?
  - – System must be in sync with an external time reference
    - > Usually the world time reference: UTC (Universal Coordinated Time) or derived GPS master clock;
- **Internal reasons: many distributed algorithms use time**
  - ■ Kerberos (authentication server) uses time-stamps
  - ■ This can be used to serialise transactions in databases
  - ■ This can be used to minimise updates when replicating data
  - – System must be in sync internally - No need to be synchronised on an external time reference

Clock Synchronization

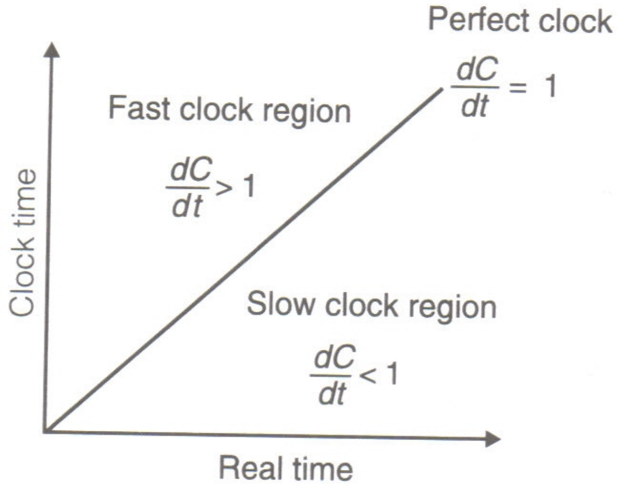
- Time is unambiguous in a centralized system – every process "sees" the same master clock in the machine.
- A process can simply make a system call to learn the time – the operating system then looks at the clock hardware.
- If process *A* asks for the current time, and a little later process *B* asks for the time, the value of *B_time* > *A_time*.
- In a distributed system, if process *A* and *B* are on different machines, *B_time* may not be greater than A_time.
- This is because the hardware clocks on these machines may not be precisely synchronised.
- Even if the clocks are synchronised, there may be a synchronisation error which is large enough to matter.
- *Example: Recent CERN quantum physics experiment failure.*

Imperfect Clocks

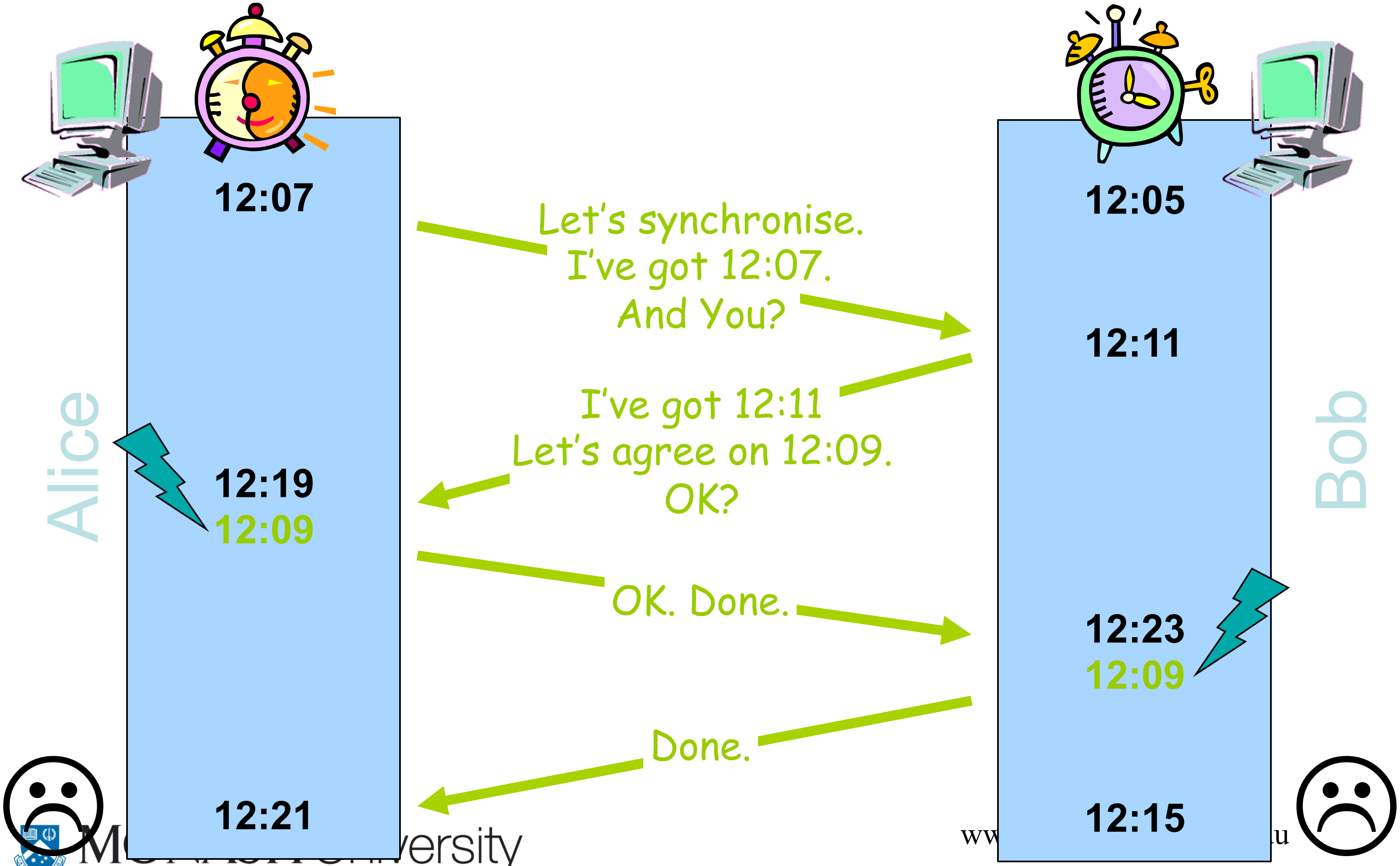
- **Human-made clocks are imperfect**
  - They run slower or faster than “real” physical time
  - How much faster or slower is termed “clock drift”
  - A drift of 1% (i.e. $1/100=10^{-2}$) means the clock adds or loses a second every 100 seconds

- **Suppose, when the real time is *t*, the time value of a clock *p* is $C_p(t)$. If the maximum drift rate allowable is *ρ*, a clock is said to be non-faulty if the following condition holds -**

$$1-\rho \leq \frac{dC}{dt} \leq 1+\rho$$



Clock Synchronisation



Alice

12:07

Let's synchronise.
I've got 12:07.
And You?

Bob

12:05

12:11

I've got 12:11
Let's agree on 12:09.
OK?

12:19
12:09

OK. Done.

12:23
12:09

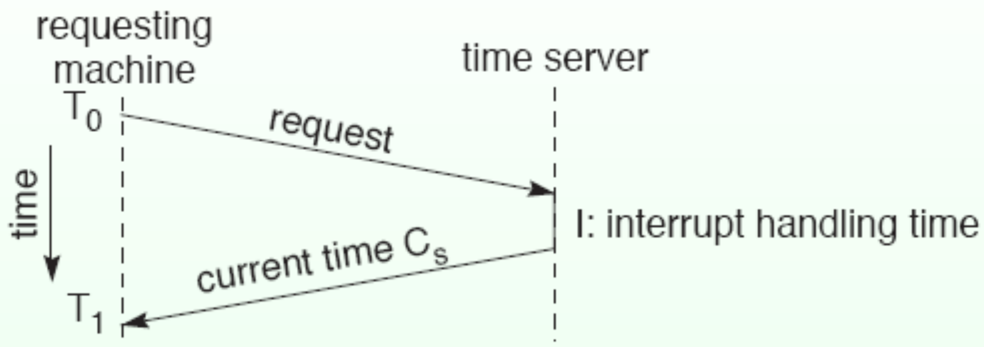
Done.

12:21

12:15

Cristian’s Algorithm

- This algorithm synchronizes clocks of all other machines to the clock of one machine, time server.
- If the clock of the time server is adjusted to the real time, all the other machines are synchronized to the real time.
- Every machine requests the current time to the time server.
- The time server responds to the request as soon as possible.
- The requesting machine sets its clock to $C_s + (T1 - T0 - I)/2$.
- In order to avoid clocks moving backward, clock adjustment must be introduced gradually.



The Berkeley Algorithm

- Developed by Gusella and Zatti.
- Unlike Cristian's Algorithm the server process in Berkeley algorithm, called the *master* periodically polls other *slave* process.
- Generally speaking the algorithm is as follows:
  - A *master* is chosen with a ring based election algorithm (Chang and Roberts algorithm).
  - The *master* polls the *slaves* who reply with their time in a similar way to Cristian's algorithm
  - The *master* observes the round-trip time (RTT) of the messages and estimates the time of each *slave* and its own.
  - The *master* then averages the clock times, ignoring any values it receives far outside the values of the others.
  - Instead of sending the updated current time back to the other process, the *master* then sends out the amount (positive or negative) that each *slave* must adjust its clock. This avoids further uncertainty due to RTT at the *slave* processes.
  - Everybody adjusts their time.

Averaging Algorithm

- Both Cristian ’s algorithm and the Berkeley algorithm are centralized algorithms with the disadvantages such as the existence of the single point of failure and high traffic volume concentrated in the master clock server.
- The “Averaging algorithm” is a decentralized algorithm.
- This algorithm divides time into resynchronization intervals with a fixed length R.
- Every machine broadcasts the current time at the beginning of each interval according to its clock.
- A machine collects all other broadcasts for a certain interval and sets the local clock by using the average of the arrival times.

Logical Clocks versus Physical Clocks

- **Lamport showed that:**
  - Clock synchronization need not be absolute
  - If two processes do not interact their clocks need not be synchronized.
  - What matters is *they agree on the order in which events occur*.
- **For many purposes, it is sufficient that all interacting machines agree on the same time – they share a "frame of reference". It is not essential that this agreed time is the same as "real time".**
- ***Clocks which agree across a group of computers but not necessarily with "real time" are termed "logical clocks".***
- ***Clocks that agree in time values, within a certain time limit (i.e. error), are "physical clocks".***

Lamport's Synchronization Algorithm

- **This algorithm only determines event order, but does not synchronize clocks.**
- **"Happens-before" relation:**
  - "*A* →*B*" is read "*A* happens before *B*": This means that *all processes agree* that event *A* occurs before event *B*.
- **The happens-before relation can be observed directly in two situations:**
  - If *A* and *B* are events in the same process, and *A* occurs before *B*, then *A*→*B*.
  - If *A* is the event of a message being sent by one process, and *B* is the event of the message being received by another process, then *A*→*B*

    .
- **"Happens-before" is a "transitive" relation – if element *a* is related to an element *b*, and *b* is in turn related to an element *c*, then *a* is also related to *c***

MONASH University
Information Technology

Lamport's Synchronization Algorithm

- If two events, *X* and *Y* happen in different processes that do not exchange messages (not even indirectly via third parties), then neither *X* →*Y* nor *Y* →*X* is true. These events are then termed "concurrent"
- What we need is a way to assign a time value *C*(*A*) on which all processes agree for every event *A*. The time value must have the following properties:
  - If *A* →*B*, then *C*(*A*) < *C*(*B*).
  - Clock time must always go forward, never backward.
- Suppose there are three processes which run on different machines as in the following figure.
- Each processor has its own local clock. The rates of the local clocks are different.

MUTEX / Mutual Exclusion in Distributed Systems

- **When multiple processes access shared resources, using the concept of *critical sections* is a relatively easy way to manage access to shared resources:**

✓ ***A critical section is a section in a program that accesses shared resources.***

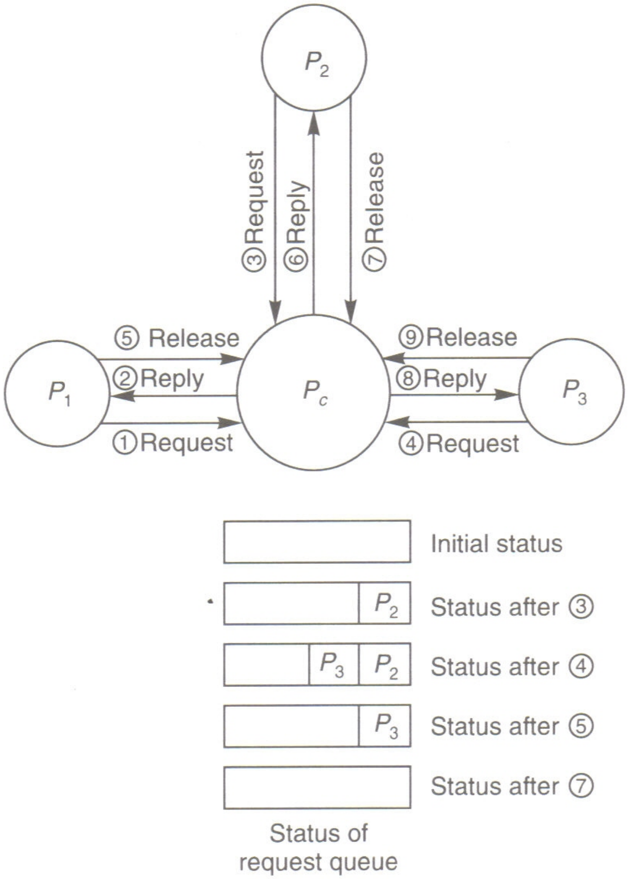
✓ ***A process enters a critical section before accessing the shared resource to ensure that no other process will use the shared resource at the same time.***

- **Critical sections are protected using semaphores and monitors in single-processor systems. We cannot use either of these mechanisms in distributed systems due to the time delays in propagating messages between machines.**

A Centralized Algorithm – Coordinator Process

- This algorithm simulates mutual exclusion in single processor systems.
- One process is elected as the "coordinator".
- When a process wants to enter a critical section of the code, it sends a request to the coordinator stating which critical section it wants to enter.
- If no other process is currently in that critical section, the coordinator returns a reply granting permission.
- If a different process is already in the critical section, the coordinator queues the request.
- When the process exits the critical section, the process sends a message to the coordinator releasing its exclusive access.
- The coordinator takes the first item off the queue of deferred request and sends that process a grant message.

A Centralized Algorithm



A Centralized Algorithm

Advantages

- ✓ Since the service policy is first-come first-serve, it is fair and no process waits forever.
- ✓ It is simple and thus easy to implement.
- ✓ It requires only three messages, request, grant, and release, per use of a critical section.

Disadvantages

- ✓ If the coordinator crashes, the entire system may go down.
- ✓ Processes cannot always distinguish a dead coordinator process from a "permission denied" message;
- ✓ A single coordinator may become a *performance bottleneck* if requests arrive at a high frequency, or the propagation delay of the messages is large.

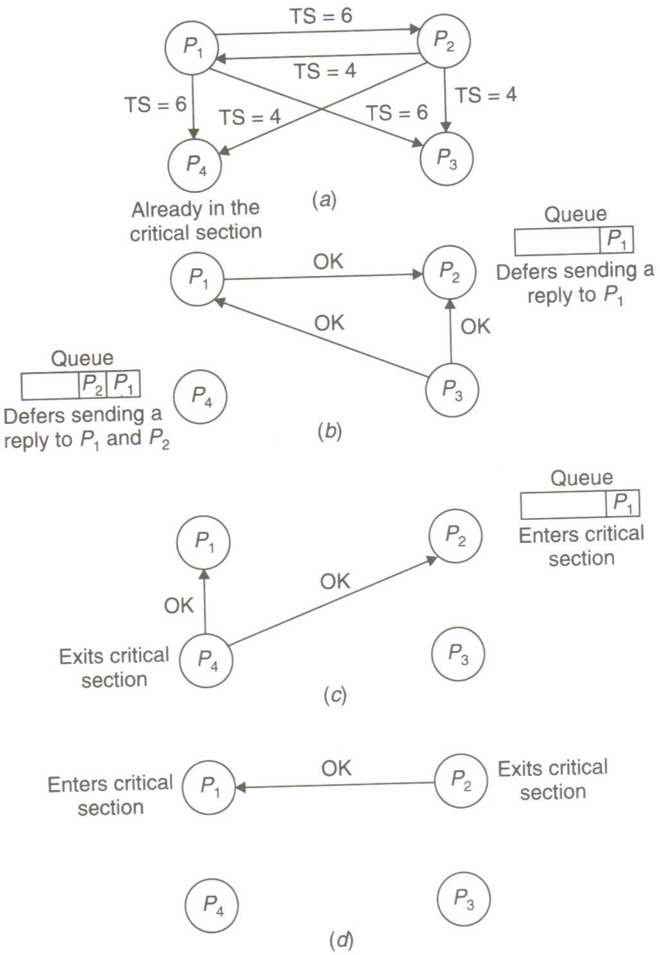
A Distributed Algorithm

- **The distributed algorithm proposed by Ricart and Agrawala requires ordering of all events in the system. We can use the Lamport's algorithm for the ordering.**
- **When a process wants to enter a critical section, the process sends a request message to all other processes. The request message includes**
  - Name of the critical section
  - Process number
  - Current time
- **The other processes receive the request message.**
  - If the process is not in the requested critical section and also has not sent a request message for the same critical section, it returns an OK message to the requesting process.
  - If the process is in the critical section, it does not return any response and puts the request to the end of a queue.
  - If the process has sent out a request message for the same critical section, it compares the time stamps of the sent request message and the received message.

A Distributed Algorithm

- **If the time stamp of the received message is smaller than the one of the sent message, the process returns an OK message.**
- **If the time stamp of the received message is larger than the one of the sent message, the request message is put into the queue.**
- **The requesting process waits until all processes return OK messages.**
- **When the requesting process receives all OK messages, the process enters the critical section.**
- **When a process exits from a critical section, it returns OK messages to all requests in the queue corresponding to the critical section and removes the requests from the queue.**
- **Processes enter a critical section in time stamp order using this algorithm.**
- ***This is a "consensus" mechanism as all processes must agree it is "OK to enter critical section".***

A Distributed Algorithm



Mutual Exclusion Algorithms: A Comparison

- The number of messages exchanged (i.e. messages per entry/exit of a single critical section):

A. Centralized: 3
B. Distributed: 2*(n − 1)
C. Ring: 2

- Reliability problems which can disable the mutual exclusion mechanism:

A. Centralized: coordinator crashes
B. Distributed: any process crashes
C. Ring: lost token, process crashes

Deadlocks in Distributed Systems

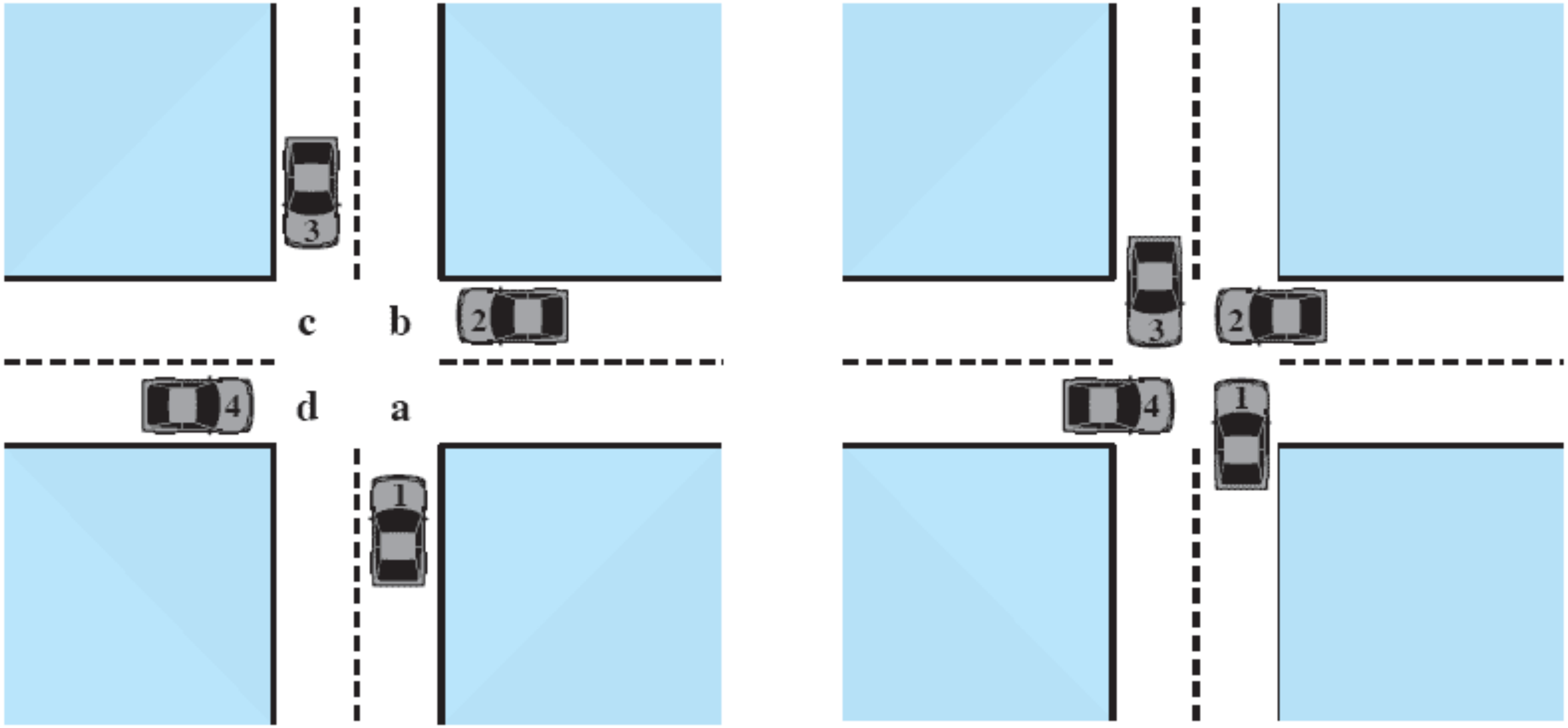
- **A *deadlock* is a condition where a process cannot proceed because it needs to obtain a resource held by another process and it itself is holding a resource that the other process needs.**
- **We can consider two types of deadlock:**
  - **communication deadlock** occurs when process A is trying to send a message to process B, which is trying to send a message to process C which is trying to send a message to A.
  - A **resource deadlock** occurs when processes are trying to get exclusive access to devices, files, locks, servers, or other resources.

- **We will not differentiate between these types since we can consider communication channels to be resources without loss of generality.**

What is a Deadlock? (Roy 2008)

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution
- Involve conflicting needs for resources by two or more processes

Deadlock Example (Roy 2008)



(a) Deadlock possible

(b) Deadlock

Figure 6.1 Illustration of Deadlock

Necessary Conditions for a Deadlock

- **Four conditions have to be met for deadlock to be present:**
  - **Mutual exclusion.** A resource can be held by at most one process
  - **Hold and wait.** Processes that already hold resources can wait for another resource.
  - **Non-preemption.** A resource, once granted, cannot be taken away from a process.
  - **Circular wait.** Two or more processes are waiting for resources held by one of the other processes.

Handling Deadlocks in DS/PS

- Strategies:

A. The “Ostrich algorithm”;

B. Deadlock detection and recovery;

C. Deadlock prevention by careful resource allocations;

D. Deadlock avoidance by designing the system in such a way that deadlocks simply cannot occur;

Summary

- **Three Real Time Clock Synchronisation Methods**
  - Cristian’s Method
  - Berkeley Algorithm
  - Averaging Algorithm
- **Logical (Clock) Synchronisation Techniques**
  - Lamport
- **Mutual Exclusion Approaches**
  - Centralised
  - Distributed
- **Main Deadlock Modeling Areas**
  - Necessary condition for occurrence
  - Deadlock Detection
  - Deadlock Handling