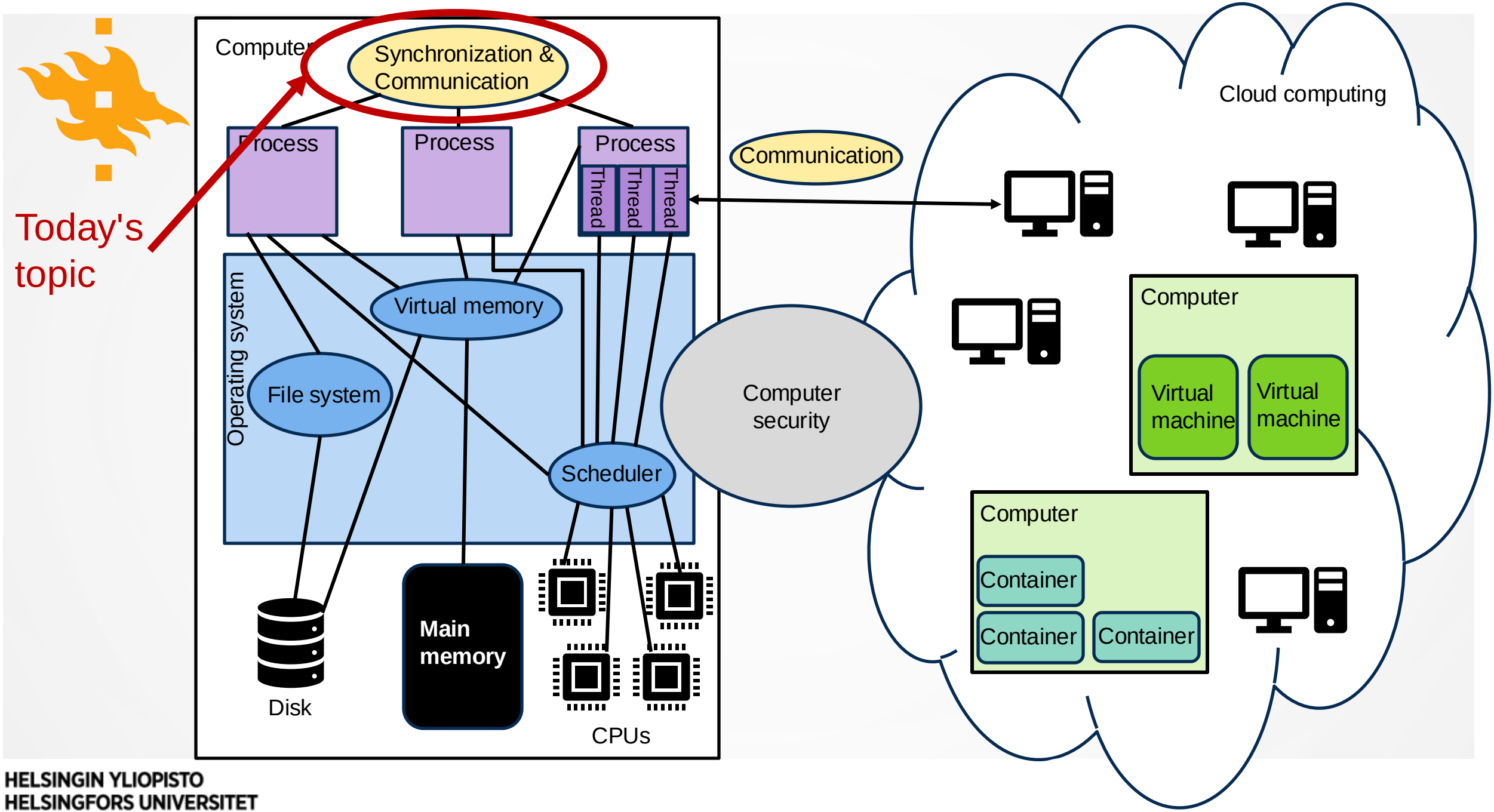
COMPUTING PLATFORMS

Concurrency: More prototype problems, common concurrency bugs, and deadlock


Today's topic
Computer
Synchronization & Communication
Process
Process
Process
Thread
Thread
Thread
Operating system
Virtual memory
File system
Scheduler
Disk
Main memory
CPUs
Communication
Computer security
Cloud computing
Computer
Virtual machine
Virtual machine
Computer
Container
Container
Container

LEARNING OUTCOMES

- After today's lecture you
  - Can describe and explain **readers/writers problem**
  - Can describe and fix common concurrency bugs
  - Understand the **conditions of deadlock**
  - Are able to explain the differences between **deadlock prevention, deadlock avoidance and deadlock detection**
  - Are able to describe and analyze the **dining philosophers** problem

READERS/WRITERS PROBLEM

- Data area (file, block of memory, …) shared by a group of processes
- Some of the processes only read (**readers**) the data area,
  some of the processes only write (**writers**) to the data area
- Conditions that must be satisfied:
  - Any number of **readers** may simultaneously read (**shared access**)
  - Only one **writer** at a time may write (**exclusive access**)
  - If a **writer** is writing, no **reader** may be reading

READERS/WRITERS PROBLEM

```
import threading

readers = 0
writers = 0
lock = threading.Lock()
oktoread = threading.Condition(lock)
oktowrite = threading.Condition(lock)
```

```
def reader():
    global readers
    global writers
    while(True):
        lock.acquire()
        while writers > 0:
            oktoread.wait()
        readers += 1
        lock.release()
        print("reading")
        lock.acquire()
        readers -= 1
        if readers == 0:
            oktowrite.notify()
        lock.release()
```

```
def writer():
    global readers
    global writers
    while(True):
        lock.acquire()
        while readers + writers > 0:
            oktowrite.wait()
        writers += 1
        lock.release()
        print("writing")
        lock.acquire()
        writers -= 1
        oktoread.notify_all()
        oktowrite.notify()
        lock.release()
```

READERS/WRITERS PROBLEM

Number of readers and writers currently in critical section

Preprotocol executed before reading / writing

```
import threading

readers = 0
writers = 0
lock = threading.Lock()
oktoread = threading.Condition(lock)
oktowrite = threading.Condition(lock)
```

```
def reader():
    global readers
    global writers
    while(True):
        lock.acquire()
        while writers > 0:
            oktoread.wait()
        readers += 1
        lock.release()
        print("reading")
        lock.acquire()
        readers -= 1
        if readers == 0:
            oktowrite.notify()
        lock.release()
```

```
def writer():
    global readers
    global writers
    while(True):
        lock.acquire()
        while readers + writers > 0:
            oktowrite.wait()
        writers += 1
        lock.release()
        print("writing")
        lock.acquire()
        writers -= 1
        oktoread.notify_all()
        oktowrite.notify()
        lock.release()
```

Lock for protecting critical sections accessing shared variables and condition variables

Postprotocol executed after reading / writing

READERS/WRITERS PROBLEM

```
import threading

readers = 0
writers = 0
lock = threading.Lock()
oktoread = threading.Condition(lock)
oktowrite = threading.Condition(lock)
```

```
def reader():
    global readers
    global writers
    while(True):
        lock.acquire()
        while writers > 0:
            oktoread.wait()
        readers += 1
        lock.release()
        print("reading")
        lock.acquire()
        readers -= 1
        if readers == 0:
            oktowrite.notify()
        lock.release()
```

```
def writer():
    global readers
    global writers
    while(True):
        lock.acquire()
        while readers + writers > 0:
            oktowrite.wait()
        writers += 1
        lock.release()
        print("writing")
        lock.acquire()
        writers -= 1
        oktoread.notify_all()
        oktowrite.notify()
        lock.release()
```

- Can a reader starve?
- Can a writer starve?
- Why do we use `oktoread.notify_all()` when writing is finished?
- What happens if both readers and writers are waiting when a writer finishes?

ATOMICITY-VIOLATION BUGS

- What can go wrong when executing the two threads shown on the right?
- How can this be fixed?

```
d = dict()

def thread1():
    global d
    if not (d is None):
        d['apple'] = 2

def thread2():
    global d
    d = None
```

ATOMICITY-VIOLATION BUGS

- What can go wrong when executing the two threads shown on the right?
  - Consider the execution order:
    Thread1 checks that d is not None
    Thread2 sets d to None
    Thread1 tries to add 'apple' to the dictionary: Error
- How can this be fixed? - Add a mutex

```
d = dict()

def thread1():
    global d
    if not (d is None):
        d['apple'] = 2

def thread2():
    global d
    d = None
```

```
d = dict()
lock = threading.Lock()

def thread1():
    global d
    lock.acquire()
    if not (d is None):
        d['apple'] = 2
    lock.release()
```

```
def thread2():
    global d
    lock.acquire()
    d = None
    Lock.release()
```

ORDER-VIOLATION BUGS

- What can go wrong when executing the two threads shown on the right?
- How can this be fixed?

```
d = None

def thread1():
    global d
    d = dict()

def thread2():
    global d
    d['apple'] = 2
```

ORDER-VIOLATION BUGS

- What can go wrong when executing the two threads shown on the right?
  - If Thread2 goes first before Thread1, `d` has not been initialized and thus Thread2 results in error
- How can this be fixed?

```
d = None

def thread1():
    global d
    d = dict()

def thread2():
    global d
    d['apple'] = 2
```

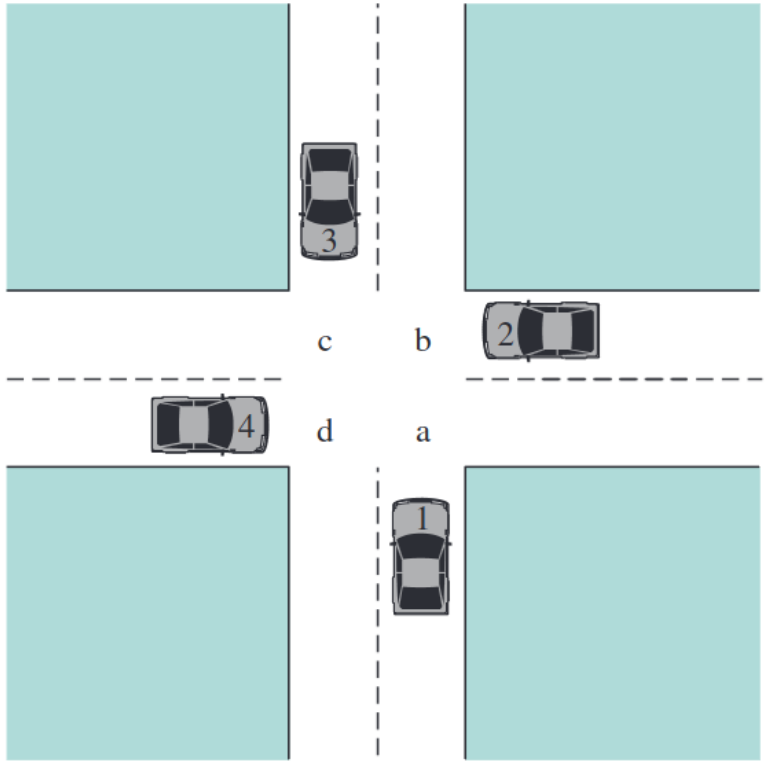
```
d = None
lock = threading.Lock()
initdone = threading.Condition(lock)

def thread1():
    global d
    lock.acquire()
    d = dict()
    initdone.notify()
    lock.release()
```

```
def thread2():
    global d
    lock.acquire()
    if d is None:
        initdone.wait()
    d['apple'] = 2
    lock.release()
```

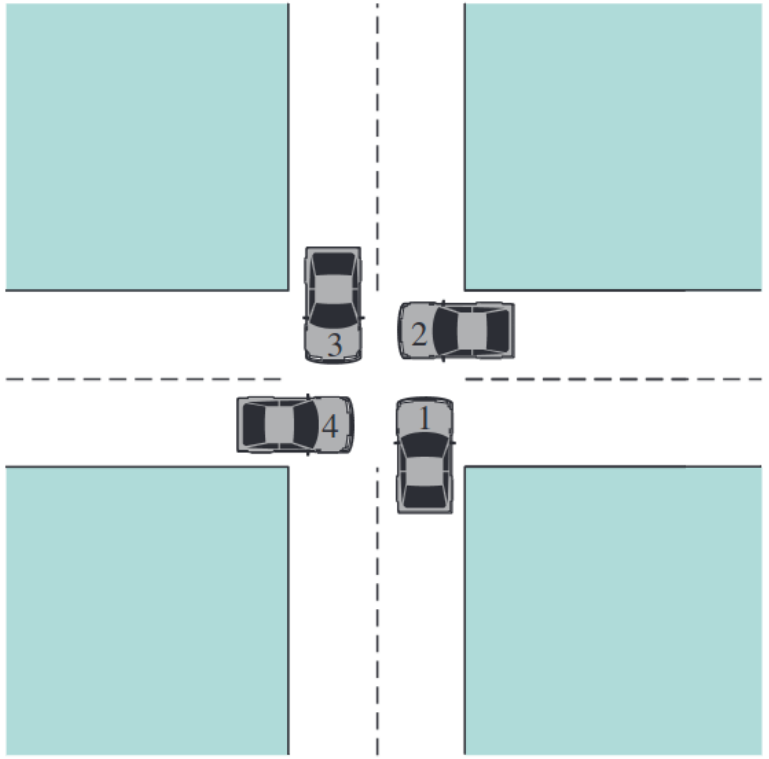
WHAT IS DEADLOCK?

- Permanent blocking of processes that either compete for resources or communicate with each other
- A set of processes is deadlocked when **each process** in the set **is blocked waiting for an event** that can **only be triggered by another blocked process** in the set
- Permanent
- No efficient solutions

EXAMPLE: TRAFFIC DEADLOCK



(a) Deadlock possible



(b) Deadlock

Figure 6.1 **Illustration of Deadlock**

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

DEADLOCK OR NOT?

```
Process P:
…
Get A
…
Get B
…
Release A
…
Release B
…
```

```
Process Q:
…
Get B
…
Get A
…
Release B
…
Release A
…
```

A SMALL CHANGE: DEADLOCK OR NOT?

```
Process P:
…
Get A
…
Release A
…
Get B
…
Release B
…
```

```
Process Q:
…
Get B
…
Get A
…
Release B
…
Release A
…
```

DIFFERENT KINDS OF RESOURCES

- Some resources are **reusable**
  - Can be used by one process at a time but is not depleted by use
  - Examples: processors, I/O channels, memory, devices, data structures including files and databases, locks, condition variables
- Other resources are **consumable**
  - Resource can be produced and consumed
  - Examples: interrupts, signals, messages, information, data in buffers
- Both kind of resources can create deadlocks

EXAMPLE: MEMORY REQUESTS

- 200 Kbytes of memory is available, and the following sequence of requests by two processes occurs

```
Process P1:
…
Request 80 Kbytes
…
Request 60 Kbytes
…
```

```
Process P2:
…
Request 70 Kbytes
…
Request 80 Kbytes
…
```

- **Deadlock** occurs if both processes progress to their second request (without releasing their first request)

EXAMPLE: MESSAGES

- Consider a pair of processes, in which each process attempts to first receive a message from another process and then send a message to the other process:

```
Process P1:
…
Receive from P2
…
Send M1 to P2
…
```

```
Process P2:
…
Receive from P1
…
Send M2 to P1
…
```

- **Deadlock** occurs if **Receive** is blocking

RESOURCE ALLOCATION GRAPHS

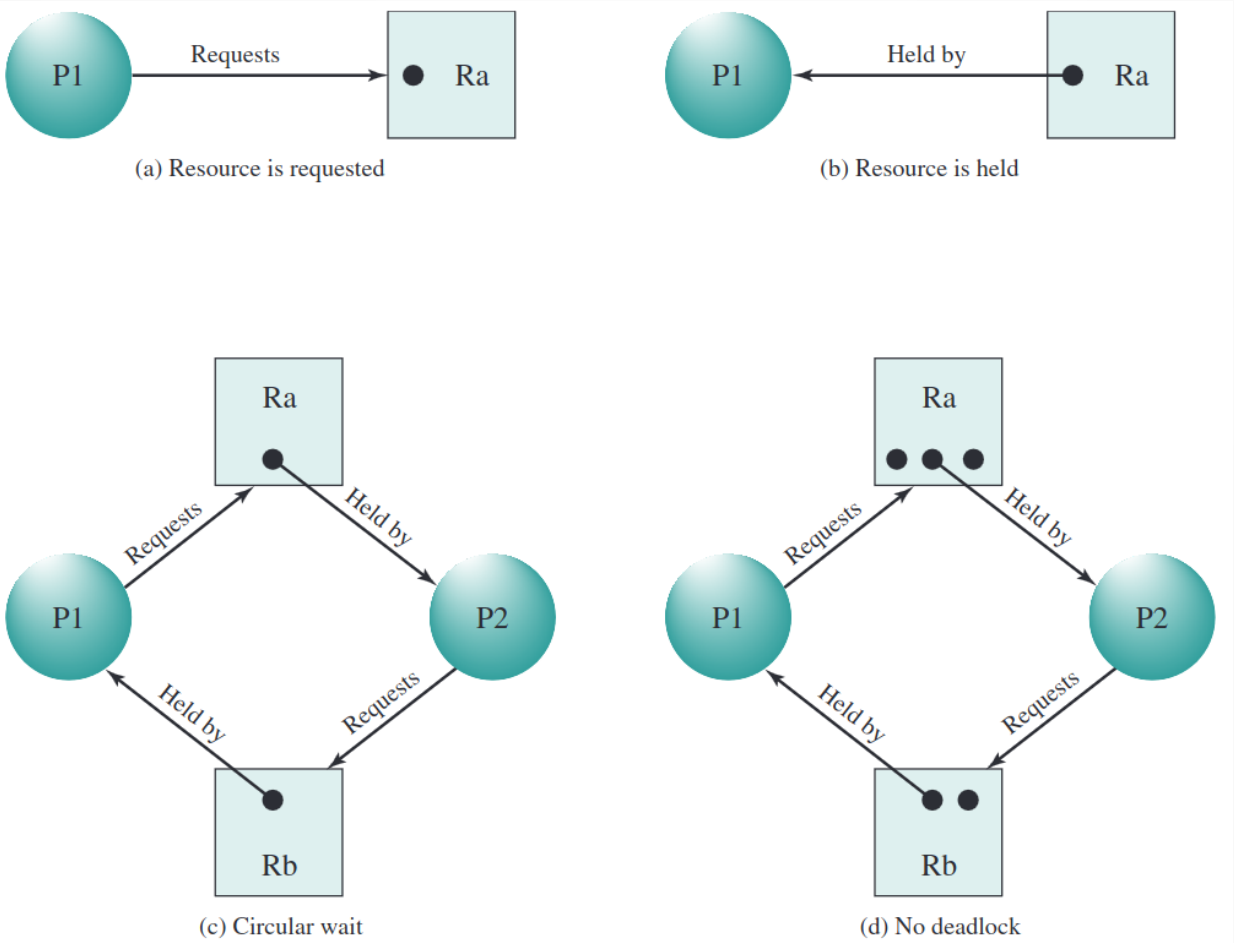


Figure 6.5 **Examples of Resource Allocation Graphs**

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

EXAMPLE: TRAFFIC DEADLOCK

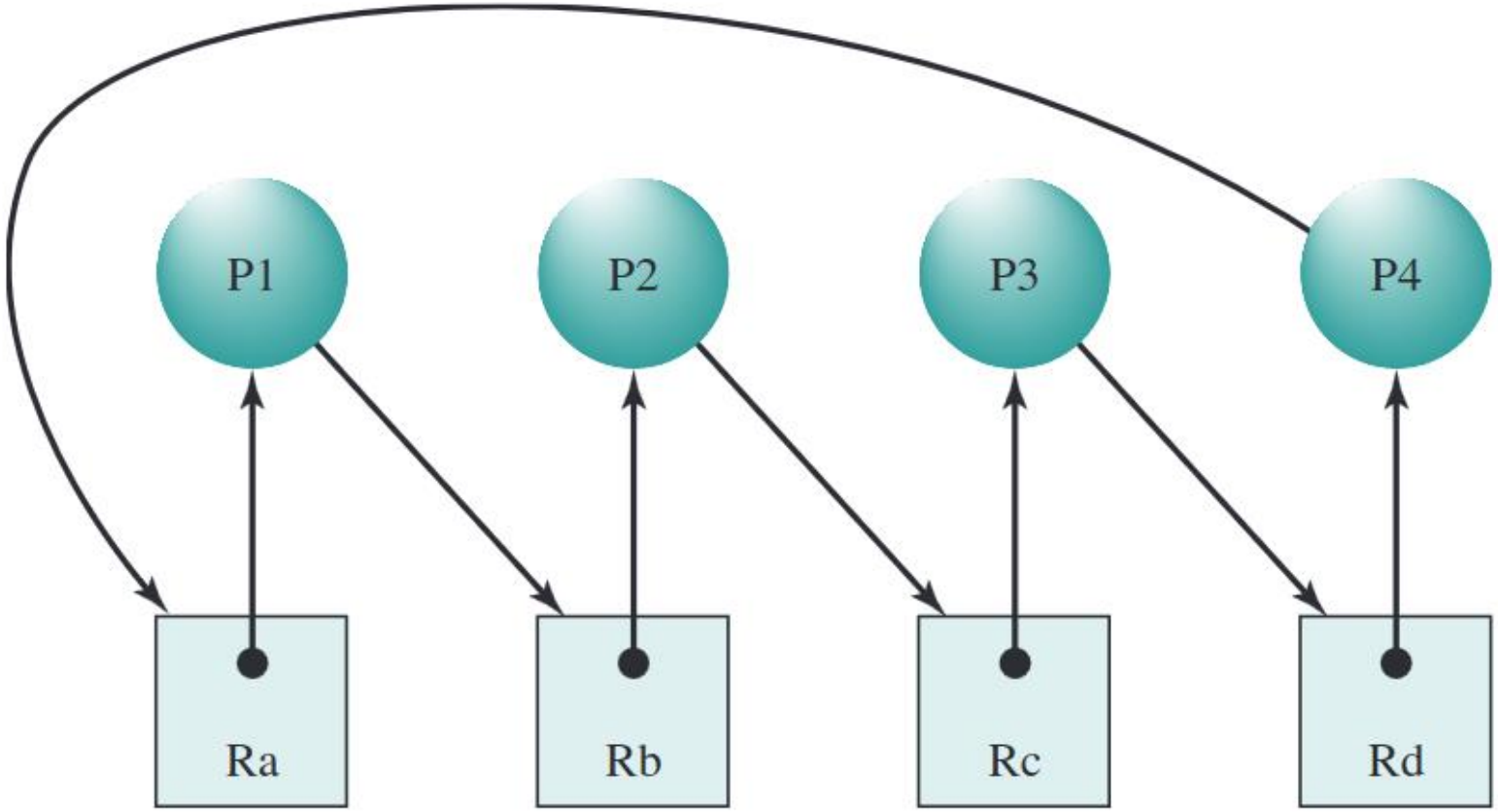


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

DEADLOCK PROBLEMS

- How to (statically) prevent deadlocks?
  - How to make it so that deadlocks cannot occur at all?
- How to (dynamically) avoid deadlocks?
  - Disallow deadlocks to happen at runtime?
  - What to do exactly if you disallow some needed operations?
- How to know if deadlock might occur?
- How to know if deadlock exists?
  - How to locate deadlocked processes?
- How to break existing deadlocks?
  - Without too much damage...
  - Automatically?
- How to prove that a solution is free of deadlock?
  - Analyze possible system state and transitions?

CONDITIONS OF DEADLOCK

1. **Mutual exclusion**
   - Only one process may use a resource at a time (e.g. critical section, CPU, printer,…)
2. **Hold and wait**
   - Process may hold allocated resources while waiting assignment of others
3. **No pre-emption**
   - No resource can be removed from a process holding it "by force"

These conditions are **necessary, but not sufficient** conditions for deadlock!

CONDITIONS OF DEADLOCK

1. **Mutual exclusion**
   - Only one process may use a resource at a time (e.g. critical section, CPU, printer,…)
2. **Hold and wait**
   - Process may hold allocated resources while waiting assignment of others
3. **No pre-emption**
   - No resource can be removed from a process holding it "by force"
4. **Circular wait** (dynamic condition!)
   - Closed chain of processes exists such that each process holds at least one resource needed by the next process in the chain.

HOW TO DEAL WITH DEADLOCK?

- Deadlock **prevention**
  - Disallow one of the three **necessary conditions for deadlock** occurrence or prevent **circular wait** condition from happening
- Deadlock **avoidance**
  - Do not grant a resource request if this allocation might lead to deadlock
- Deadlock **detection**
  - Grant resource request if possible but periodically check for the presence of deadlock and take action to recover

DEADLOCK PREVENTION STRATEGY

- Design a system so that deadlock cannot occur
- Two main approaches
  - Indirect: prevent the occurrence of one of the three necessary (static) conditions
  - Direct: prevent the occurrence of the necessary (dynamic) condition (circular wait)

DEADLOCK PREVENTION

- No mutual exclusion?
  - Usually difficult to avoid (mutex is there for a reason)
- No hold-and-wait?
  - Process must request all its required resources at one time and it will be blocked until all the requests can be granted
  - Difficult or not very practical

DEADLOCK PREVENTION CONT'D

- Pre-emption?
  - If a process holding some resources is denied a further request, that process must release its resources and requests them again
  - OS may preempt the second process and require it to release its resources
  - May be difficult in practice
- No circular wait?
  - Define a **linear ordering** of the resources
  - Always reserve resources in this order

DEADLOCK DETECTION

- Check for deadlock can be made for each resource request
  - Happens frequently (every request?), can also be done less frequently depending on how likely it is for a deadlock to occur
- Advantages
  - Leads to early detection
  - Relatively simple algorithm
- Disadvantages
  - Frequent checks consume considerable processor time
  - Must be prepared to handle the deadlocks that are found!

RECOVERY STRATEGIES WHEN DEADLOCK IS FOUND

1. Abort all deadlocked processor threads
   - Might be OK for multithreaded application, most common solution
2. Backup each (or some) deadlocked process to a previously defined and saved checkpoint and restart those processes
   - Hope that deadlock does not occur again
   - Requires **roll-back** and **restart**
3. Successively abort deadlocked processes until deadlock no longer exists
4. Successively preempt resources until deadlock no longer exists

DEADLOCK AVOIDANCE

- **Deadlock prevention**: constrain resource requests to prevent at least one of the four conditions of deadlock
- **Deadlock avoidance**:
  - Decision is made **dynamically** at runtime whether the **current or future resource allocation requests** will, if granted, potentially lead to a deadlock
  - Requires knowledge of (possible) future process requests

DEADLOCK AVOIDANCE APPROACHES

- **Resource allocation denial**
  - Do not grant an incremental resource request to a process if this allocation might lead to deadlock
  - Very conservative
- **Process initiation denial**
  - Do not start a process if its demands might lead to a deadlock
  - Ultra conservative
- No good solution really exists

DEADLOCK AVOIDANCE: PROS AND CONS

- Advantages:
  - Not necessary to preempt and rollback processes as in deadlock detection
  - Less restrictive than deadlock prevention
- Disadvantages:
  - Maximum resource requirement for each process must be stated in advance
  - Must be able to handle resource request denial
  - Processes considered must be **independent** and with **no synchronization requirements**
  - Must be a **fixed number of resources to allocate**
  - No process may exit while holding resources

DINING PHILOSOPHERS PROBLEM

- **Five philosophers** live in a house where a table is set for them. The life of each philosopher consists principally of **thinking** and **eating**, and through years of thought, all of the philosophers had agreed that the only food that contributed to their thinking efforts was spaghetti. Due to a lack of manual skill, **each philosopher requires two forks** to eat spaghetti.
- Round table, five plates, and five forks
- Philosopher wishing to eat goes to their assigned place and using the two forks on either side of the plate, takes and eats some spaghetti

Djikstra, E.W. "Hierarchical ordering of sequential processes", *Acta Informatica* **1**(2), 1971.

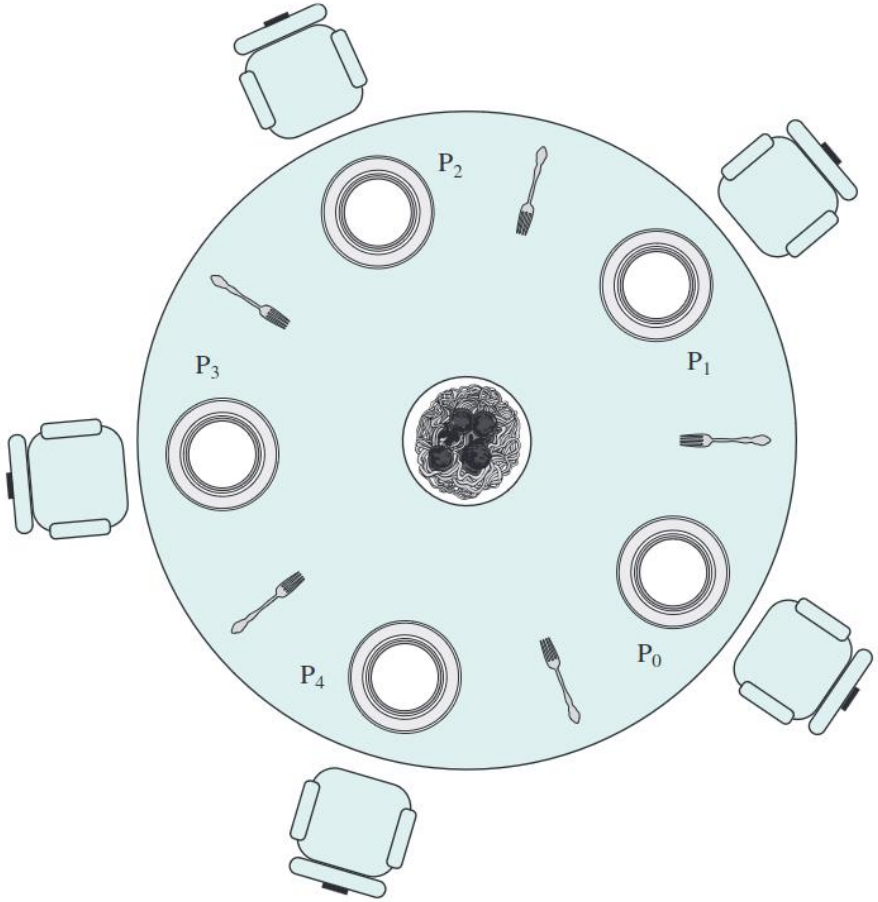


Figure 6.11 Dining Arrangement for Philosophers

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

DINING PHILOSOPHERS PROBLEM

- Need two forks to eat, reserve one at a time
- No two philosophers can use the same fork at the same time
- No philosopher must starve to death (avoid deadlock and starvation)
- How to reserve forks?
  - No chatting, no talking (no communication)
  - Can only see forks left/right
  - Which protocol to follow when reserving the forks?

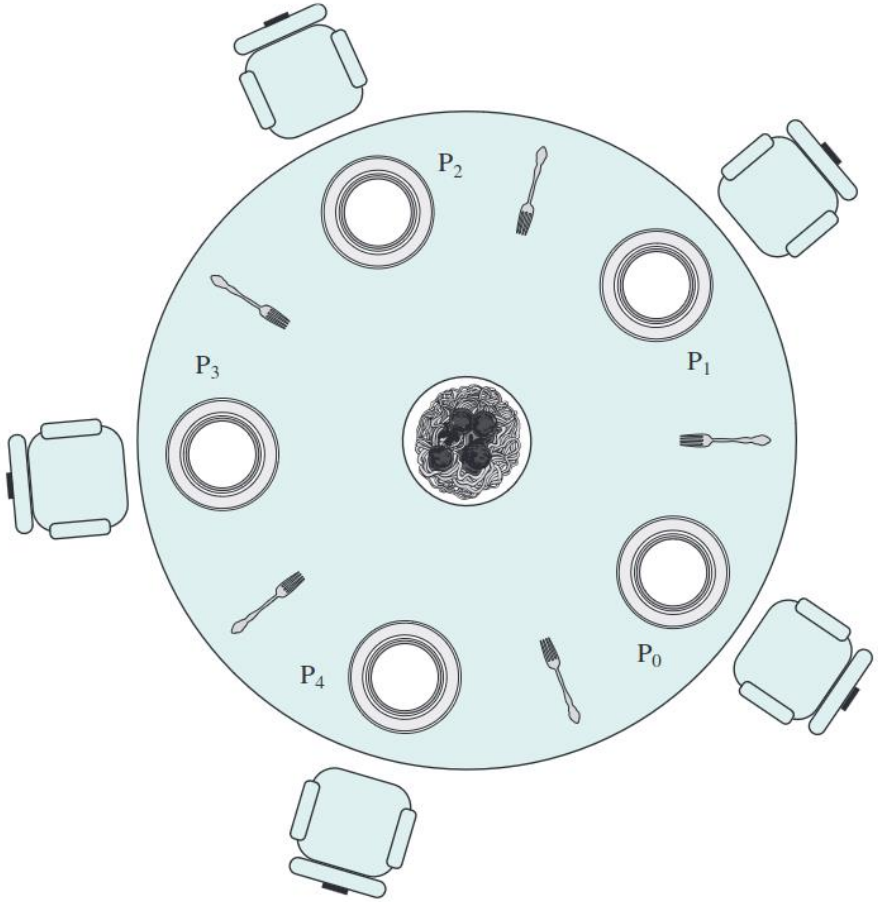


Figure 6.11 Dining Arrangement for Philosophers

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

(NON)SOLUTION USING LOCKS AND CONDITION VARIABLES

```
import threading

lock = threading.Lock()
forkready  = [threading.Condition(lock)]*5
fork = [True]*5

def get_fork(fid):
  lock.acquire()
  while not forks[fid]:
    forkready[fid].wait()
  fork[fid] = False
  lock.release()

def release_fork(fid):
  lock.acquire()
  fork[fid] = True
  forkready[fid].notify()
  lock.release()
```

Mutex to protect shared variables and condition variables

Condition variables to wait for fork i to be free

True if fork i is free, False if it is taken

Why does this not work?

```
def philosopher(k):
  while True:
    <think>
    get_fork(k)
    get_fork((k+1)%5)
    <eat spaghetti>
    release_fork(k)
    release_fork((k+1)%5)
```

(NON)SOLUTION USING LOCKS AND CONDITION VARIABLES

```
import threading

lock = threading.Lock()
forkready  = [threading.Condition(lock)]*5
fork = [True]*5

def get_fork(fid):
  lock.acquire()
  while not forks[fid]:
    forkready[fid].wait()
  fork[fid] = False
  lock.release()

def release_fork(fid):
  lock.acquire()
  fork[fid] = True
  forkready[fid].notify()
  lock.release()
```

Why does this not work?

- Philosopher 1 gets fork 1
- Philosopher 2 gets fork 2
- Philosopher 3 gets fork 3
- Philosopher 4 gets fork 4
- Philosopher 5 gets fork 5
- Each philosopher now holds one fork and waits for another – **deadlock!**

```
def philosopher(k):
  while True:
    <think>
    get_fork(k)
    get_fork((k+1)%5)
    <eat spaghetti>
    release_fork(k)
    release_fork((k+1)%5)
```

SOLUTION: ALLOW AT MOST 4 PHILOSOPHERS IN THE DINING ROOM

```
import threading

lock = threading.Lock()
room = threading.Condition(lock)
forkready  = [threading.Condition(lock)]*5
fork = [True]*5
room_counter = 0

def get_fork(fid):
  lock.acquire()
  while not fork[fid]:
    forkready[fid].wait()
  fork[fid] = False
  lock.release()

def release_fork(fid):
  lock.acquire()
  fork[fid] = True
  forkready[fid].notify()
  lock.release()
```

Condition variable for waiting to enter the room

Number of philosophers in the room

Why does this work?

```
def enter_room():
  lock.acquire()
  while room_counter >= 4:
    room.wait()
  room_counter += 1
  Lock.release()

def exit_room():
  lock.acquire()
  room_counter -= 1
  room.notify()
  lock.release()
```

```
def philosopher(k):
  while True:
    <think>
    enter_room()
    get_fork(k)
    get_fork((k+1)%5)
    <eat spaghetti>
    release_fork(k)
    release_fork((k+1)%5)
    exit_room()
```

SOLUTION: ALLOW AT MOST 4 PHILOSOPHERS IN THE DINING ROOM

```
import threading

lock = threading.Lock()
room = threading.Condition(lock)
forkready  = [threading.Condition(lock)]*5
fork = [True]*5
room_counter = 0

def get_fork(fid):
  lock.acquire()
  while not fork[fid]:
    forkready[fid].wait()
  fork[fid] = False
  lock.release()

def release_fork(fid):
  lock.acquire()
  fork[fid] = True
  forkready[fid].notify()
  lock.release()
```

Why does this work?

- At most 4 philosophers in the room. There are five forks so at least one philosophers gets both forks
- Breaks the circular wait condition of deadlock

```
def enter_room():
  lock.acquire()
  while room_counter >= 4:
    room.wait()
  room_counter += 1
  Lock.release()

def exit_room():
  lock.acquire()
  room_counter -= 1
  room.notify()
  lock.release()
```

```
def philosopher(k):
  while True:
    <think>
    enter_room()
    get_fork(k)
    get_fork((k+1)%5)
    <eat spaghetti>
    release_fork(k)
    release_fork((k+1)%5)
    exit_room()
```

ORIGINAL SOLUTION BY DIJKSTRA (RESOURCE HIERARCHY)

- Assign a **partial order to forks**; resources are requested in this order
- No two resources unrelated by order will ever be used by a single unit of work at the same time
- Hence, forks will be numbered 1 through 5 and each philosopher always picks up the lower-numbered fork first, and then the higher-numbered fork
- No deadlock!
  - If four of the philosophers simultaneously pick up their lower-numbered fork, only the highest-numbered fork will remain on the table, so the 1st or 5th philosopher will not be able to pick up any fork
  - Only one philosopher has access to the highest-numbered fork and is able to eat using two forks
  - Thus breaks the circular wait condition.

SUMMARY

- Readers/writers problem
- Deadlock: permanent blocking of a set of processes/threads that either compete for system resources or communicate with each other
- Four necessary and sufficient conditions of deadlock
- Dealing with deadlock
  - Prevention: guarantees that deadlock will not occur
  - Detection: OS checks for deadlock and takes action
  - Avoidance: analyze each new resource request
- Dining philosophers – classical example