Real-time systems
“Foundations in synchronization and resource management”

Mathieu Delalandre
Foundations in synchronization and resource management

1. Synchronization for mutual exclusion
   1.1. Introduction to synchronization
   1.2. Principles of concurrency
   1.3. Synchronization methods for mutual exclusion

2. Resource management
   2.1. Resource acquisition and management
   2.2. Resources-allocation graph and sequences
   2.3. Resources-allocation and synchronization primitives
   2.4. Holding and preemption of resources
   2.5. Deadlocks and necessary conditions
   2.6. Safe and unsafe states
   2.7. Resource management protocols
Introduction to synchronization (1)

Cooperating/independent process: A process is cooperating if it can affect (or be affected) by the other processes. Clearly, any process than shares data and uses Inter-Process Communication is a cooperating process. Any process that does not share data with any other process is independent.

Inter-process communication (IPC) refers to the set of techniques for the exchange of data among different processes. There are several reasons for providing an environment allowing IPC:

- **Information sharing**, several processes could be interested in the same piece of information, we must provide a framework to allow concurrent access to this information.

- **Modularity**, we may to construct the system in a modular fashion, dividing the system functions into separate block.

- **Convenience**, even an individual user may work on many related tasks at the same time e.g. editing, printing and compiling a program.

- **Speedup**, in the multi-cores case, if we are interested to run faster a particular task, we must break it into sub-tasks.
**Introduction to synchronization (2)**

**Process synchronization:** It refers to the idea that multiple processes are to join up or handshake at a certain point, so as to reach an agreement or to commit to a certain sequence of action. Clearly, any cooperating process is concerned with synchronization. We can classify the ways in which processes synchronize on the basis of the degree to which they are aware of each other’s existence:

- **Processes unaware of each other:** These are independent processes that are not intended to work together. Although the processes are not working together, the OS needs to be concerned about concurrency and mutual exclusion problems with resources.

- **Processes indirectly aware of each other:** These are processes that are not necessarily aware of each other by their respective process ids, but that share access to some objects such as an I/O buffer. Such process exhibit coordination in sharing common objects.

- **Processes directly aware of each other:** These are cooperating processes that are able to communicate with each other by process ids and that are designed to work jointly in some activity. Again, such processes exhibit coordination.

<table>
<thead>
<tr>
<th>Degree of awareness</th>
<th>Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes unaware of each other</td>
<td>Mutual exclusion</td>
</tr>
<tr>
<td>Processes indirectly aware of each</td>
<td>Coordination by sharing</td>
</tr>
<tr>
<td>other</td>
<td></td>
</tr>
<tr>
<td>Processes directly aware of each</td>
<td>Coordination by communication</td>
</tr>
<tr>
<td>other</td>
<td></td>
</tr>
</tbody>
</table>

- [Process synchronization diagram]

---

4
Foundations in synchronization and resource management

1. Synchronization for mutual exclusion
   1.1. Introduction to synchronization
   1.2. Principles of concurrency
   1.3. Synchronization methods for mutual exclusion

2. Resource management
   2.1. Resource acquisition and management
   2.2. Resources-allocation graph and sequences
   2.3. Resources-allocation and synchronization primitives
   2.4. Holding and preemption of resources
   2.5. Deadlocks and necessary conditions
   2.6. Safe and unsafe states
   2.7. Resource management protocols
Principles of concurrency (1)

**Inter-process communication (IPC):** It is a set of techniques for the exchange of data among multiple processes or threads.

**Race conditions** arise when separate processes of execution depend on some shared states. Operations upon shared states could result in harmful collisions between these processes.

**Critical section** is a piece of code (of a process) that accesses a shared resource (data structure or device) that must not be concurrently accessed by other concurrent/cooperating processes.

**Mutual exclusion:** Two events are mutually exclusive if they cannot occur at the same time. Mutual exclusion algorithms are used to avoid the simultaneous use of a resource by the “critical section” pieces of code.

**Process synchronization:** It refers to the idea that multiple processes are to join up or handshake at a certain point, so as to reach an agreement or commit to a certain sequence of action.

**Resource acquisition** is related to the operation sequence to request, access and release a non-sharable resource by a process. This is the synchronization problem for mutual exclusion, between processes (2 or n).
Principles of concurrency (2)

**Race conditions** arise when separate processes of execution depend on some shared states. Operations upon shared states could result in harmful collisions between these processes. e.g. spooling with 2 processes A, B and a Daemon D

<table>
<thead>
<tr>
<th>Slot</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>∅</td>
</tr>
<tr>
<td>3</td>
<td>lesson.pptx</td>
</tr>
<tr>
<td>4</td>
<td>paperid256.rtf</td>
</tr>
<tr>
<td>5</td>
<td>∅</td>
</tr>
<tr>
<td>6</td>
<td>∅</td>
</tr>
<tr>
<td>7</td>
<td>∅</td>
</tr>
</tbody>
</table>

- **S** the spooling directory
- **in** current writing index of S
- **out** current reading index of S
- **P** a process
- **D** the printer daemon process
- **X.a** A data a part of a process X

Notations:
- **(1)** P.in = in
- **(2)** S[P.in] = P.name
- **(3)** in = P.in + 1
- **(4)** D.out = out
- **(5)** D.name = S[D.out]
- **(6)** out = D.out + 1
- **(7)** print

(1) to (7) are atomic instructions
Principles of concurrency (3)

**Race conditions** arise when separate processes of execution depend on some shared states. Operations upon shared states could result in harmful collisions between these processes. E.g. spooling with 2 processes A,B and a Daemon D

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A→1</td>
<td>7</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>7</td>
<td>6</td>
<td>X.name</td>
</tr>
<tr>
<td>B→1,2,3</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>B.name</td>
<td>7</td>
<td>6</td>
<td>X.name</td>
</tr>
<tr>
<td>A→2,3</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>A.name</td>
<td>7</td>
<td>6</td>
<td>X.name</td>
</tr>
<tr>
<td>D→4,5,6,7</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>A.name</td>
<td>8</td>
<td>7</td>
<td>A.name</td>
</tr>
</tbody>
</table>

P→x process P executes instruction x

initial states

- A reads “in”
- B reads “in”, writes in “S” and increments “in”
- A writes in “S”, and increments “in”, the harmful collision is here
- D prints the file, the B one will be never processed

**Notations**

- P: a process
- D: the printer daemon process
- X.a: A data a part of a process X
- S: the spooling directory
- in: current writing index of S
- out: current reading index of S
- S[P.in]: P name
- S[D.out]: D name
- print

**Diagram:**

- **IPC** raises **Race conditions**
- **Critical section** solved by **Mutual exclusion**
- **Synchronization** considered as
- **Resource acquisition**
Principles of concurrency (4)

**Critical section** is a piece of code (of a process) that accesses a shared resource (data structure or device) that must not be concurrently accessed by other concurrent/cooperating processes. A critical section will usually terminate in fixed time, a process will have to wait a fixed time to enter it.
Principles of concurrency (5)

**Mutual exclusion:** Two events are mutually exclusive if they cannot occur at the same time. Mutual exclusion algorithms are used to avoid the simultaneous use of a resource by the “critical section” pieces of code. Mutual exclusion could be achieved using synchronization.

**Process synchronization:** It refers to the idea that multiple processes are to join up or handshake at a certain point, so as to reach an agreement or commit to a certain sequence of action.
**Principles of concurrency (6)**

A **resource** is any physical or virtual component of limited availability within a computer system e.g. CPU time, hard disk, device (USB, CD/DVD, etc.), network, etc.

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Can be used in parallel by several processes</th>
<th>e.g. read only memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>shareable</strong></td>
<td>Can be accessed by a single process at a time</td>
<td>e.g. write only memory, device, CPU time, network access, etc.</td>
</tr>
<tr>
<td><strong>no shareable</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Resource acquisition** is related to the operation sequence to request, access and release a no shareable resource by a process. This is the synchronization problem for mutual exclusion, between processes (2 or n).

<table>
<thead>
<tr>
<th>Request</th>
<th>If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Access</strong></td>
<td>The process can operate on the resource.</td>
</tr>
<tr>
<td><strong>Release</strong></td>
<td>The process releases the resource.</td>
</tr>
</tbody>
</table>

**IPC** raises **Race conditions** which are defined by the **Critical section**. Resource acquisition is solved by **Mutual exclusion**.
Foundations in synchronization and resource management

1. Synchronization for mutual exclusion
   1.1. Introduction to synchronization
   1.2. Principles of concurrency
   1.3. Synchronization methods for mutual exclusion

2. Resource management
   2.1. Resource acquisition and management
   2.2. Resources-allocation graph and sequences
   2.3. Resources-allocation and synchronization primitives
   2.4. Holding and preemption of resources
   2.5. Deadlocks and necessary conditions
   2.6. Safe and unsafe states
   2.7. Resource management protocols
Synchronization for mutual exclusion

<table>
<thead>
<tr>
<th>Methods</th>
<th>Approach</th>
<th>Type</th>
<th>Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabling interrupts</td>
<td>disabling interrupts</td>
<td>hardware</td>
<td>No</td>
</tr>
<tr>
<td>Swap, TSL, CAS</td>
<td>busy wait</td>
<td>software</td>
<td>Possible</td>
</tr>
<tr>
<td><em>Perterson's algorithm</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary semaphore / mutex</td>
<td>sleep wakeup</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>
Synchronization methods for mutual exclusion

“Interrupt disabling”

**Interrupt disabling:** within an uniprocessor system, processes cannot have overlapped execution, they can be only interleaved. Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted. This capability can be provided in the form of primitives defined in the OS kernel, for disabling and enabling interrupts when entered in a critical section.

E.g.

Scheduling of two processes A, B accessing a critical section without interrupt disabling

Scheduling of two processes A, B accessing a critical section with interrupt disabling

**Access a critical section**
disable interrupt

**Release a critical section**
enable interrupt

The price of this approach is high
✓ The efficiency of execution, at scheduling level, could be noticeably degraded (a C process, not interested with the section, while A accesses the section)
✓ This approach cannot work in a multi-processor architecture
Synchronization for mutual exclusion

<table>
<thead>
<tr>
<th>Methods</th>
<th>Approach</th>
<th>Type</th>
<th>Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabling interrupts</td>
<td>disabling interrupts</td>
<td>hardware</td>
<td>No</td>
</tr>
<tr>
<td>Swap, TSL, CAS</td>
<td>busy wait</td>
<td>software</td>
<td>Possible</td>
</tr>
<tr>
<td><em>Perterson's algorithm</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary semaphore / mutex</td>
<td>sleep wakeup</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>
Synchronization methods for mutual exclusion

“Swap, TSL and CAS”

TSL is an alternative instruction to Swap, achieving in one-shot a if and a set instruction, atomically.

TSL RX, LOCK

(1) copy

(2) set to 1 if lock at 0

atomic instruction

Request

(1) Request the critical section with p
(2) do TSL RX, LOCK
(3) while RX equals 1

Run in the critical section with p

do something ....

Release

(4) Release the critical section with p
(5) set LOCK at 0

e.g. with three processes A, B and C considering the scheduling

<table>
<thead>
<tr>
<th>RX</th>
<th>LOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Na</td>
<td>$0</td>
</tr>
<tr>
<td>TSL RX, LOCK</td>
<td>$0</td>
</tr>
</tbody>
</table>

“access case - Lock at 0”
RX set to 0
LOCK moves to 1

B→1,2
<table>
<thead>
<tr>
<th>RXA</th>
<th>RXB</th>
<th>RXC</th>
<th>LOCK</th>
<th>by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>0</td>
<td>Ø</td>
</tr>
<tr>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Ø</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Ø</td>
<td>0</td>
<td>Ø</td>
</tr>
<tr>
<td>A→3,2</td>
<td>Ø</td>
<td>Ø</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>B→3,4,5</td>
<td>Ø</td>
<td>Ø</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>C→1,2,3,2,3</td>
<td>Ø</td>
<td>Ø</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>A→3,4,5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Ø</td>
</tr>
<tr>
<td>C→2,3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C→4,5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Ø</td>
</tr>
</tbody>
</table>

B accesses the section
A is blocked
B releases the section
A can access
C is blocked
A releases the section
C can access
C releases the section
## Synchronization for mutual exclusion

<table>
<thead>
<tr>
<th>Methods</th>
<th>Approach</th>
<th>Type</th>
<th>Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabling interrupts</td>
<td>disabling interrupts</td>
<td>hardware</td>
<td>No</td>
</tr>
<tr>
<td>Swap, TSL, CAS</td>
<td>busy wait</td>
<td>hardware</td>
<td>No</td>
</tr>
<tr>
<td><em>Perterson's algorithm</em></td>
<td></td>
<td>software</td>
<td>Possible</td>
</tr>
<tr>
<td>binary semaphore / mutex</td>
<td>sleep wakeup</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>
Synchronization methods for mutual exclusion
“binary semaphores / mutex” (1)

Semaphore is a synchronization primitive, composed of queue and a variable, controlled with two operations down and up.

A binary semaphore takes only the values 0 and 1. A mutex is a binary semaphore for which a process that locks it must be the one that unlocks it.

The down operation decreases the semaphore’s value or sleeps the current process.

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>false</td>
</tr>
<tr>
<td>stack</td>
<td>∅</td>
</tr>
</tbody>
</table>

e.g. “normal” down

down

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>true</td>
</tr>
<tr>
<td>stack</td>
<td>∅</td>
</tr>
</tbody>
</table>

e.g. blocked down

If true

sleep p_k, and push p_k in the stack
Synchronization methods for mutual exclusion
“binary semaphores / mutex” (2)

Semaphore is a synchronization primitive, composed of queue and a variable, controlled with two operations down and up.

A binary semaphore takes only the values 0 and 1. A mutex is a binary semaphore for which a process that locks it must be the one that unlocks it.

The up operation increases the semaphore’s value and wakes up the processes in the stack.

<table>
<thead>
<tr>
<th>BEFORE</th>
<th>AFTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>true</td>
</tr>
<tr>
<td>stack</td>
<td>∅</td>
</tr>
</tbody>
</table>

e.g. “normal” up

<table>
<thead>
<tr>
<th>BEFORE</th>
<th>AFTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>true</td>
</tr>
<tr>
<td>stack</td>
<td>P</td>
</tr>
</tbody>
</table>

if true, wakeup pj and pop pj from the stack

dispatcher

short-term scheduler

running process

Main memory

up

P

pj

pk

 semaphore

value

...
Synchronization methods for mutual exclusion
“binary semaphores / mutex” (3)

Semaphore is a synchronization primitive, composed of queue and a variable, controlled with two operations down and up.

A binary semaphore takes only the values 0 and 1. A mutex is a binary semaphore for which a process that locks it must be the one that unlocks it.

The algorithm for mutual exclusion using a binary semaphore is

sem is a semaphore

<table>
<thead>
<tr>
<th>Request</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Request the critical section with ( p ) \n</td>
<td>(2) down sem</td>
</tr>
<tr>
<td>(3) Run in the critical section with ( p ) \n</td>
<td>do something ….</td>
</tr>
<tr>
<td>(4) Release the critical region with ( p )</td>
<td></td>
</tr>
<tr>
<td>(5) up sem</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sem value</th>
<th>by</th>
<th>A state</th>
<th>B state</th>
<th>C state</th>
</tr>
</thead>
<tbody>
<tr>
<td>false S</td>
<td>∅</td>
<td>ready</td>
<td>ready</td>
<td>ready</td>
</tr>
<tr>
<td>true A</td>
<td>∅</td>
<td>ready</td>
<td>blocked</td>
<td></td>
</tr>
<tr>
<td>true B</td>
<td>∅</td>
<td>ready</td>
<td>blocked</td>
<td></td>
</tr>
<tr>
<td>true C.B</td>
<td>∅</td>
<td>ready</td>
<td>blocked</td>
<td></td>
</tr>
<tr>
<td>true C-B</td>
<td>∅</td>
<td>ready</td>
<td>blocked</td>
<td></td>
</tr>
<tr>
<td>true C-∅</td>
<td>∅</td>
<td>ready</td>
<td>ready</td>
<td></td>
</tr>
</tbody>
</table>

A accesses the section, sem becomes true
While accessing the semaphore, B blocks
A exits and pops up B, B holds the section
B exits and pops up C, C holds the section
C exits and puts the semaphore to false
Foundations in synchronization and resource management

1. Synchronization for mutual exclusion
   1.1. Introduction to synchronization
   1.2. Principles of concurrency
   1.3. Synchronization methods for mutual exclusion

2. Resource management
   2.1. Resource acquisition and management
   2.2. Resources-allocation graph and sequences
   2.3. Resources-allocation and synchronization primitives
   2.4. Holding and preemption of resources
   2.5. Deadlocks and necessary conditions
   2.6. Safe and unsafe states
   2.7. Resource management protocols
Resource acquisition and management

A resource is any physical or virtual component of limited availability within a computer system e.g. CPU time, hard disk, device (USB, CD/DVD, etc.), network, etc.

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Can be used in parallel by several processes</th>
<th>e.g. read only memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>shareable</strong></td>
<td>Can be accessed by a single process at a time</td>
<td>e.g. write only memory, device, CPU time, network access, etc.</td>
</tr>
<tr>
<td><strong>no shareable</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Resource acquisition is related to the operation sequence to request, access and release a no sharable resource by a process. This is the synchronization problem for mutual exclusion, between processes (2 or n).

<table>
<thead>
<tr>
<th>Request</th>
<th>If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>The process can operate on the resource.</td>
</tr>
<tr>
<td>Release</td>
<td>The process releases the resource.</td>
</tr>
</tbody>
</table>

Resource management deals with the allocation of the no shareable resources of a computer to tasks/processes being performed on that computer.
Foundations in synchronization and resource management

1. Synchronization for mutual exclusion
   1.1. Introduction to synchronization
   1.2. Principles of concurrency
   1.3. Synchronization methods for mutual exclusion

2. Resource management
   2.1. Resource acquisition and management
   2.2. Resources-allocation graph and sequences
   2.3. Resources-allocation and synchronization primitives
   2.4. Holding and preemption of resources
   2.5. Deadlocks and necessary conditions
   2.6. Safe and unsafe states
   2.7. Resource management protocols
A resource allocation graph is a tool that helps in characterizing allocation of resources. A resource allocation graph is a directed graph that describes a state of the system of resources as well as processes. Every resource and process type is represented by a node, and their relations (e.g. request, resource holding) by edges.

Notation

About nodes

- Resource of type $R_i$ with 4 instances (resource node)
- Process $P_i$ (process node)

About edges

- $P_i$ is waiting for one instance of $R_i$ ("request" edge)
- $P_i$ holds one instance of $R_i$ ("hold" edge)

Resource acquisition

Single access

request

use

release

P3 holds R1, P1 and P2 cannot access

When P3 releases R1, P1 or P2 (not the both due to mutual exclusion) can access
A resource allocation graph is a tool that helps in characterizing allocation of resources. A resource allocation graph is a directed graph that describes a state of the system of resources as well as processes. Every resource and process type is represented by a node, and their relations (e.g. request, resource holding) by edges.

### Notation

**About nodes**
- Resource of type $R_i$ with 4 instances (resource node)
- Process $P_i$ (process node)

**About edges**
- $P_i$ is waiting for one instance of $R_i$ (“request” edge)
- $P_i$ holds one instance of $R_i$ (“hold” edge)

### Resource acquisition

**Multiple access - disjointed use**

1. $P_1$ requests, uses and releases $R_1$
2. $P_1$ requests, uses and releases $R_2$
A **resource allocation graph** is a tool that helps in characterizing allocation of resources. A resource allocation graph is a directed graph that describes a state of the system of resources as well as processes. Every resource and process type is represented by a node, and their relations (e.g. request, resource holding) by edges.

### Notation

<table>
<thead>
<tr>
<th>About nodes</th>
<th>About edges</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="resource.png" alt="Resource" /> Resource of type ( R_i ) with 4 instances (resource node)</td>
<td><img src="process.png" alt="Process" /> Process ( P_i ) (process node)</td>
</tr>
<tr>
<td><img src="process.png" alt="Process" /> ( P_i ) is waiting for one instance of ( R_i ) (&quot;request&quot; edge)</td>
<td><img src="process.png" alt="Process" /> ( P_i ) holds one instance of ( R_i ) (&quot;hold&quot; edge)</td>
</tr>
</tbody>
</table>

### Resource acquisition

**Multiple access - jointed use**

1. \( P_1 \) requests \( R_1 \) and \( R_2 \) in any order
2. \( P_1 \) uses \( R_1 \) and \( R_2 \) and releases them in any order
A resource allocation sequence is the order by which resources are utilized (request, use and release) by processes e.g. a resource acquisition sequence involving 4 processes (P1, P2, P3 and P4), 3 resources of two types (R1, R2); we have R1, R2 accessed in a disjoint (P1) and joint (P2, P3) ways, R1 accessed in a single way (P4)

The resource-allocation graph at \( t_0 \)

(1)-(2) P3 accesses R1
P2 accesses R1
P4 accesses R2

(2)-(3) P4 releases R2
P3 accesses R2

(3)-(4) P3 releases R1,R2
P1 accesses R1
P2 accesses R2

(4)-(5) P2 releases R1,R2
P1 releases R1 and accesses R2

(5)-(6) P1 releases R2
Foundations in synchronization and resource management

1. Synchronization for mutual exclusion
   1.1. Introduction to synchronization
   1.2. Principles of concurrency
   1.3. Synchronization methods for mutual exclusion

2. Resource management
   2.1. Resource acquisition and management
   2.2. Resources-allocation graph and sequences
   2.3. Resources-allocation and synchronization primitives
   2.4. Holding and preemption of resources
   2.5. Deadlocks and necessary conditions
   2.6. Safe and unsafe states
   2.7. Resource management protocols
The allocation of resources depends, in main part, of the used synchronization primitives.
e.g. a resource allocation involving 3 processes (P0, P1 and P2), 2 resources (R0 and R1) using Mutex.

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>R0</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Q₀(t)</td>
<td>U₀</td>
</tr>
<tr>
<td>P0</td>
<td>15</td>
<td>s+9</td>
<td>6</td>
</tr>
<tr>
<td>P1</td>
<td>12</td>
<td>s+5</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>Na</td>
<td>Na</td>
</tr>
</tbody>
</table>

- C is the capacity of a process
- s is the start date of a process
- Q(t) is the request time (i.e. down on Mutex)
- U is the needed time to use the resource, with
  \[ Q(t) + U \leq s + C \]
- R(t) is the release time (i.e. up on Mutex) with
  \[ R(t) = Q(t) + U \]
  \[ U = R(t) - Q(t) \]
Resources-allocation and synchronization primitives (2)

The allocation of resources depends, in main part, on the used synchronization primitives. e.g. a resource allocation involving 3 processes (P0, P1 and P2), 2 resources (R0 and R1) using Mutex.

<table>
<thead>
<tr>
<th>CPU Burst</th>
<th>5</th>
<th>6</th>
<th>3</th>
<th>4</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>P1</td>
<td>P0</td>
<td>P2</td>
<td>P1</td>
<td>P0</td>
<td>P1</td>
<td>P0</td>
<td>P2</td>
</tr>
<tr>
<td>Event</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
</tbody>
</table>

The diagram illustrates the allocation of resources and synchronization with different events for processes P0, P1, and P2, using resources R0 and R1.
Foundations in synchronization and resource management

1. Synchronization for mutual exclusion
   1.1. Introduction to synchronization
   1.2. Principles of concurrency
   1.3. Synchronization methods for mutual exclusion

2. Resource management
   2.1. Resource acquisition and management
   2.2. Resources-allocation graph and sequences
   2.3. Resources-allocation and synchronization primitives
   2.4. Holding and preemption of resources
   2.5. Deadlocks and necessary conditions
   2.6. Safe and unsafe states
   2.7. Resource management protocols
Holding and preemption of resources (1)

**Hold and wait of resources:** The resource allocation is done with an “hold and wait” condition of resources, without hold and wait, resource utilization could be low, starvation probability higher and programming task harder.

Without hold and wait, whenever a process requests resources, it does not hold any other resources. 

e.g. consider a process that
1. copy data from DVD to disk files
2. sort the files
3. print the files on a printer

We can consider two protocols to manage this, with and without holding

---

**Protocol 1: “with holding”**

1. The process $P$ has no resource, it can make a request.
2. The process $P$ gets all the resources in one shot.
3. The process $P$ copies, sorts and prints.
4. The process $P$ releases its resources.
Holding and preemption of resources (2)

**Hold and wait of resources:** The resource allocation is done with an “hold and wait” condition of resources, without hold and wait, resource utilization could be low, starvation probability higher and programming task harder.

Without hold and wait, whenever a process requests resources, it does not hold any other resources

- e.g. consider a process that
  1. copy data from DVD to disk files
  2. sort the files
  3. print the files on a printer

We can consider two protocols to manage this, with and without holding

**Protocol 2 “without holding”**

1. The process P has no resource, it can make a request.
2. The process P gets part of the resources (DVD, disk)
3. The process P copies an sorts
4. The process P releases its resources
5. P has no resource, it can make a request. It gets part of the resources (disk, printer).
6. The process P prints.
7. The process P releases its resources.
Holding and preemption of resources (3)

**Preemption of resources**: the resource allocation is done with a “no preemption” condition on resources.
- some resources can be preempted in a system, when their states can be easily saved and restored later (CPU registers, memory, etc.).
- some resources are intrinsically no preemptible (e.g. printer, tape drives, etc.).

**without preemption**, the request sequence is
1. we check whether resources are available
2. if yes, we allocate them
3. if no, we wait

**with preemption**, the request sequence is
1. we check whether resources are available
2. if yes, we allocate them
3. if no, we check whether resources are allocated to other processes waiting for additional resources
4. if so, we preempt the desired resources
5. if no, we wait
Foundations in synchronization and resource management

1. Synchronization for mutual exclusion
   1.1. Introduction to synchronization
   1.2. Principles of concurrency
   1.3. Synchronization methods for mutual exclusion

2. Resource management
   2.1. Resource acquisition and management
   2.2. Resources-allocation graph and sequences
   2.3. Resources-allocation and synchronization primitives
   2.4. Holding and preemption of resources
   2.5. Deadlocks and necessary conditions
   2.6. Safe and unsafe states
   2.7. Resource management protocols
Deadlocks and necessary conditions

**Deadlock** refers to a specific condition when two or more processes are each waiting for each other to release a non-shareable resource, or more than two processes are waiting for resources in a circular chain.

**The necessary conditions** are such that if they hold simultaneously in a system, deadlocks could arise.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Mutual exclusion</td>
<td>At least one resource must be held in a non sharable mode, that is only one process a time can use this resource.</td>
</tr>
<tr>
<td>2. Hold and wait</td>
<td>A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.</td>
</tr>
<tr>
<td>3. No preemption</td>
<td>Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding.</td>
</tr>
<tr>
<td>4. Circular wait</td>
<td>A set {P0, P1, … Pn} of waiting process must exit such that -P0 is waiting for a resource held by P1 -P1 is waiting by a resource held by P2 -…. -Pn-1 is waiting by a resource held by Pn -Pn is waiting by a resource held by P0</td>
</tr>
</tbody>
</table>

P1 is waiting for one instance of R2, held by P2.
P2 is waiting for one instance of R1, held by P1.
Foundations in synchronization and resource management

1. Synchronization for mutual exclusion
   1.1. Introduction to synchronization
   1.2. Principles of concurrency
   1.3. Synchronization methods for mutual exclusion

2. Resource management
   2.1. Resource acquisition and management
   2.2. Resources-allocation graph and sequences
   2.3. Resources-allocation and synchronization primitives
   2.4. Holding and preemption of resources
   2.5. Deadlocks and necessary conditions
   2.6. Safe and unsafe states
   2.7. Resource management protocols
Safe and unsafe states (1)

Goal of the safety based protocols is to maintain the system in a safe state

- A **safe state** is as follow, considering
  1. a given set of processes \( S = \{P_0, \ldots, P_n\} \).
  2. we have a resource allocation state \( R_s \) corresponding to the available resources and the resources held by \( \{P_0, \ldots, P_n\} \).
  3. we have a safe state, if it exists a sequence of requests \( <P_0, \ldots, P_n> \) that could satisfy all the processes, considering the available resources and the ones than can be released by the processes.

- An **unsafe state** is not a safe state.

- A **deadlock state** is unsafe, but not all the unsafe states are deadlocks.
Safe and unsafe states (2)

e.g. we consider the allocation problem with three processes \{P0, P1, P2\} to access a resource R of 12 instances, the needs of process are P0 = 10, P1 = 4, P2 = 9.

At \( t_i \), we consider the following allocation state

<table>
<thead>
<tr>
<th>Processes</th>
<th>Hold</th>
<th>Rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

Free resources 3

The state is safe because it exists a request sequence that satisfies all the processes.

<table>
<thead>
<tr>
<th>Processes</th>
<th>Hold</th>
<th>Rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Free resources 12

Only P1 can access additional resources

P1 accesses 2 R and releases all

P0 accesses 5 R and releases all

P2 can accesses 7 R and releases all
Safe and unsafe states (3)

e.g. we consider the allocation problem with three processes \{P0, P1, P2\} to access a resource R of 12 instances, the needs of process are P0 = 10, P1 = 4, P2 = 9

At \( t_i \), we consider another allocation state in which P2 held one more resource

<table>
<thead>
<tr>
<th>Processes</th>
<th>Hold</th>
<th>Rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Free resources 2

The state is unsafe because it exists none request sequence that satisfies all the processes.

Only P1 can access additional resources

<table>
<thead>
<tr>
<th>Processes</th>
<th>Hold</th>
<th>Rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Free resources 2

P1 accesses 2 R and releases all

<table>
<thead>
<tr>
<th>Processes</th>
<th>Hold</th>
<th>Rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Free resources 4

The free resources cannot satisfy P0 or P2
Joint progress diagram, illustrates the concept of safety in a graphic and easy-to-understand way, by showing the progress of two processes competing for resources, with each of the process needing exclusive use of resources for a certain period of time.

e.g. “deadlock” with two processes P,Q and resources A,B

- Every point of a path line in the diagram represents a joint state of the two processes.
- All the paths must be vertical or horizontal, neither diagonal. Motion is always to the north or east, neither to the south or west (because processes cannot backward in time, off course).
- When a path is next to an instruction line, its request is granted, otherwise it is blocked.
- Gray zones are forbidden regions due to mutual exclusion.
- The light-gray area (bottom-left to mutual exclusion zones) is referred as the unsafe region.
- The top-right corners bounded in the unsafe region are deadlocks.
Safe and unsafe states (5)

**Joint progress diagram**, illustrates the concept of safety in a graphic and easy-to-understand way, by showing the progress of two processes competing for resources, with each of the process needing exclusive use of resources for a certain period of time.

E.g. “deadlock” with two processes P, Q and resources A, B

1. P acquires A and then B, Q executes and blocks on a request for B. P releases A and B. When Q resumes execution, it will be able to acquire the both resources.

2. P acquires A and then B then releases A and B. When Q resumes execution, it will be able to acquire the both resources.

3, 4 are inverted paths of (1, 2).

5. Q acquires B and then P acquires A. Deadlock is inevitable, Q will block on A and P will block on B.

6. P acquires A and Q acquires B. P blocked when accessing B, same for Q with A. Deadlock is here.
Safe and unsafe states (6)

**Joint progress diagram**, illustrates the concept of safety in a graphic and easy-to-understand way, by showing the progress of two processes competing for resources, with each of the process needing exclusive use of resources for a certain period of time.

e.g. “no deadlock” with two processes P,Q and resources A,B

(1) P acquires A then releases A. P acquires B, Q executes and blocks on a request for B. P releases B. When Q resumes execution, it will be able to acquire the both resources.

(2) P acquires then releases A and B. When Q resumes execution, it will be able to acquire the both resources.

(3,4) are inverted paths of (1,2).

(5) Q acquires B and then P acquires and releases A. Q acquires A then releases B and A. When P resumes execution, it will be able to acquire B.

(6) Q acquires B and then P acquires and releases A. Q acquires A then releases B. P acquires then releases B. When Q resumes execution, it will be able to release A.

When deadlocks cannot appear, unsafe states cannot exist.
Foundations in synchronization and resource management

1. Synchronization for mutual exclusion
   1.1. Introduction to synchronization
   1.2. Principles of concurrency
   1.3. Synchronization methods for mutual exclusion

2. Resource management
   2.1. Resource acquisition and management
   2.2. Resources-allocation graph and sequences
   2.3. Resources-allocation and synchronization primitives
   2.4. Holding and preemption of resources
   2.5. Deadlocks and necessary conditions
   2.6. Safe and unsafe states
   2.7. Resource management protocols
Resource management protocols
“Introduction” (1)

A resource management protocol is the mechanism (code convention, algorithms, system, etc.) in charge of the resource management. Main goals of such protocols are to avoid/prevent deadlocks, to deal with resource starvation and to optimize resources allocation. Three main approaches exist, based on prevention, avoidance and detection.

-Ostrich-like, do nothing

-Prevention ensures that at least one of the necessary conditions cannot hold, to prevent the occurrence of deadlocks.

-Avoidance authorizes deadlocks, but makes judicious choices to assure that the deadlock point is never reached. With avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead into a deadlock.

-Detection and recovery do not employ prevention and avoidance, then deadlocks could occur in the system. They aim to detect deadlocks that occur, and to recover safe states.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Deadlocks could exist</th>
<th>Deadlocks could appear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ostrich-like</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Prevention</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Avoidance</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Detection &amp; recovery</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
A resource management protocol is the mechanism (code convention, algorithms, system, etc.) in charge of the resource management. Main goals of such protocols are to avoid/prevent deadlocks, to deal with resource starvation and to optimize resources allocation. Three main approaches exist, based on prevention, avoidance and detection.

-Ostrich-like, do nothing

-Prevention ensures that at least one of the necessary conditions cannot hold, to prevent the occurrence of deadlocks.

-Avoidance authorizes deadlocks, but makes judicious choices to assure that the deadlock point is never reached. With avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead into a deadlock.

-Detection and recovery do not employ prevention and avoidance, then deadlocks could occur in the system. They aim to detect deadlocks that occur, and to recover safe states.
Resource management protocols

“The ostrich-like protocol”

Ostrich-like protocol: i.e. to ignore the problem

<table>
<thead>
<tr>
<th>Cons</th>
<th>Pros</th>
</tr>
</thead>
<tbody>
<tr>
<td>When deadlocks could appear, systems could be blocked.</td>
<td>-Regarding the systems, frequency of deadlocks could be low.</td>
</tr>
<tr>
<td></td>
<td>-Finite capacity of systems could raise in deadlocks (e.g. job queue size, file table), deadlocks are part of OS.</td>
</tr>
<tr>
<td></td>
<td>-OS design is a complex task, resource management protocols could result in bugs and hard implementation.</td>
</tr>
<tr>
<td></td>
<td>-Without resource management protocols, systems will gain a lot in performance.</td>
</tr>
<tr>
<td></td>
<td>-Resource management protocols involve constraints for users and impact the ergonomics of systems.</td>
</tr>
<tr>
<td></td>
<td>-etc.</td>
</tr>
</tbody>
</table>
# Resource management protocols

**“Prevention protocol”**

Prevention protocols ensure that at least one of the necessary conditions cannot hold, to prevent the occurrence of deadlocks.

<table>
<thead>
<tr>
<th>Necessary conditions</th>
<th>Statute about prevention</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Mutual exclusion</td>
<td>Resources in a computer are intrinsically no shareable (printer, write-only memory, etc), prevention protocols can’t be defined from this condition.</td>
<td>Not applicable.</td>
</tr>
<tr>
<td>2. Hold and wait</td>
<td>Without hold and wait, resource utilization could be low, starvation probability higher and programming task harder.</td>
<td>Applicable with severe performance lost.</td>
</tr>
<tr>
<td>3. No preemption</td>
<td>Some resources can be preempted in a system, when their states can be easily saved and restored later (CPU registers, memory, etc.). Some other resources are intrinsically no preemptible (e.g. printer, tape drives, etc.), prevention protocols cannot be then defined from this condition.</td>
<td>Not applicable.</td>
</tr>
<tr>
<td>4. Circular wait</td>
<td>One way to ensure that deadlocks never hold is to impose total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration. This involves to coerce programming of processes to this order access.</td>
<td>Applicable with programming constraints.</td>
</tr>
</tbody>
</table>
Resource management protocols
“The avoidance protocol” (1)

Resource allocation denial protocols are based on avoidance, they require additional information about how resources will be requested. Based on the on-line requests, the system considers the resource currently available and allocated to evaluate the future requests.

Total, available, allocated and claim resources state about the resource allocation in the system.

A resource-allocation component maintains the on-line the resource-allocation state of the system and the available resource instances.
Resource management protocols

“The avoidance protocol” (2)

**Resource allocation denial protocols** are based on avoidance, they require additional information about how resources will be requested. Based on the on-line requests, the system considers the resource currently available and allocated to evaluate the future requests.

**without avoidance**, access to resource is decided at the synchronization level

**with avoidance**, access to resource is decided at the avoidance algorithm level and the synchronization one
Resource management protocols
“The detection & recovery protocol”

Detection and recovery protocols do not employ prevention and avoidance, then deadlocks could occur. They aim to detect deadlocks that occur, and to recover safe states. If a deadlock is detected two approaches can be employed, based on rollback and process killing.

Detection and recovery with rollback

Resource allocation: The algorithm collects the allocation states processes / resources and maintains the current allocation state.

Deadlock detection: based on different detection methods, the algorithm searches for deadlock(s). If negative, the algorithm saves the current state, otherwise it goes to recovery.

Recovery: If a deadlock is detected, the algorithm uses the safe-states to restore the system.