Operating Systems
“Inter-Process Communication (IPC) and synchronization”

Mathieu Delalandre
University of Tours, Tours city, France
mathieu.delalandre@univ-tours.fr

Lecture available at http://mathieu.delalandre.free.fr/teachings/operating1.html

Operating Systems
“IPC and synchronization”

1. Introduction
2. Synchronization for mutual exclusion
   2.1. Principles of concurrency
   2.2. Synchronization methods for mutual exclusion
3. Synchronization for coordination
   3.1. Some problems of coordination
   3.2. Solving the Producer/Consumer problem
   3.3. Solving the multiple Producer/Consumer problem

Introduction (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 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 a concurrent access to this information.

- ✓**Modularity**: we may to construct the system in a modular fashion, dividing a function of the system into separate blocks.

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

- ✓**Speedup**: with parallelism, if we are interested to run faster a particular task, we must break it into sub-tasks.

Introduction (2)

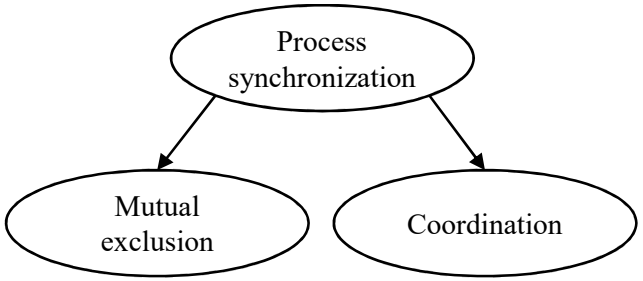
Process synchronization: refers to the idea that multiple processes join up to reach an agreement or to commit a sequence of action. Clearly, any cooperating process is concerned with synchronization. We can classify the synchronization on the basis of the degree to which the processes are aware of each other.

✓**Processes unaware of each other:** are independent and not intended to work together. Although the processes are not working together, the OS must deal with the **concurrency** and **mutual exclusion** problems.

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

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

Degree of awareness	**Synchronization**
Processes unaware of each other	Mutual exclusion
Processes indirectly aware of each other	Coordination by sharing
Processes directly aware of each other	Coordination by communication



Operating Systems
“IPC and synchronization”

1. Introduction
2. Synchronization for mutual exclusion
   2.1. Principles of concurrency
   2.2. Synchronization methods for mutual exclusion
3. Synchronization for coordination
   3.1. Some problems of coordination
   3.2. Solving the Producer/Consumer problem
   3.3. Solving the multiple Producer/Consumer problem

Principles of concurrency (1)

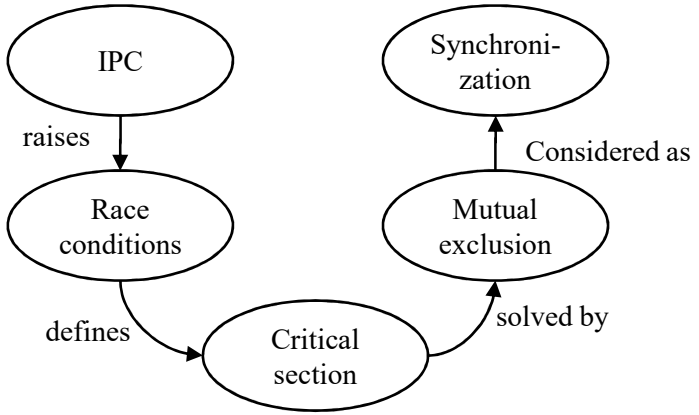
Inter-process communication (IPC) is a set of techniques for the exchange of data among multiple processes or threads.

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.

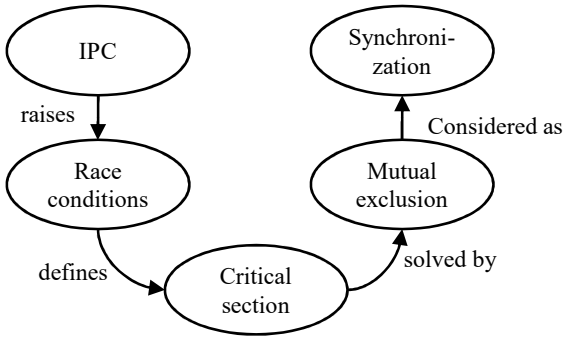
Critical section is a piece of code that accesses a shared resource (a data structure or a device) that must not be concurrently accessed by other concurrent/cooperating processes.

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 piece of code of the critical section.

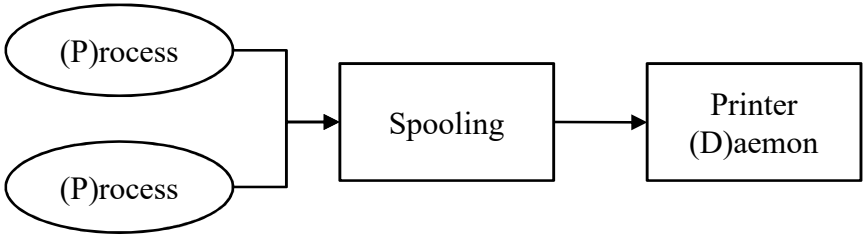
Process synchronization: refers to the idea that multiple processes join up to reach an agreement or to commit a sequence of action.



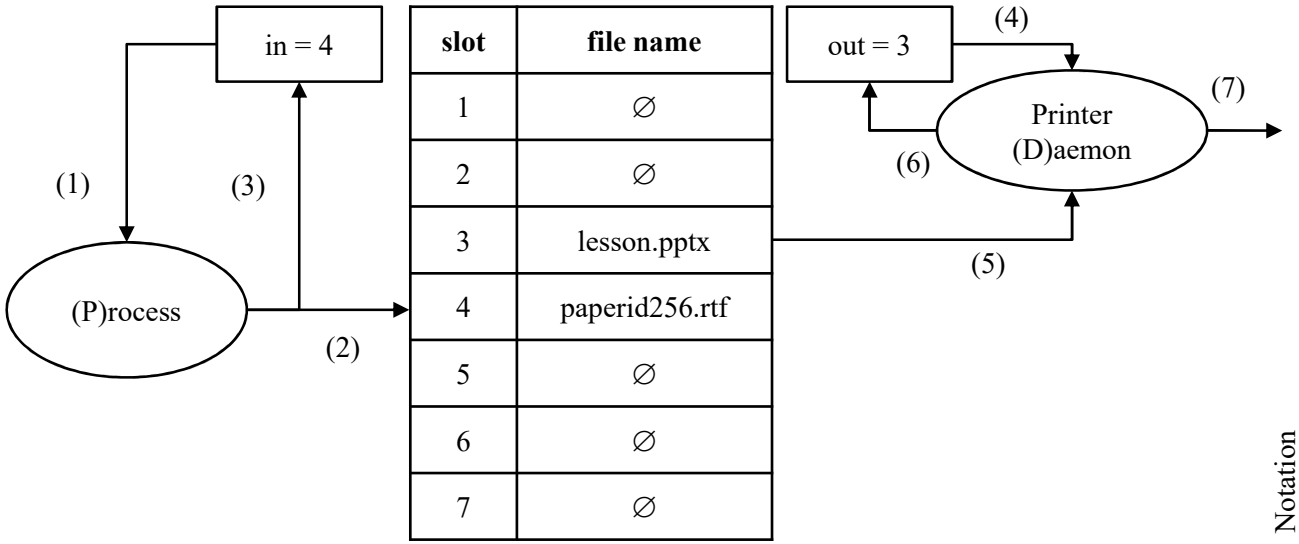
Principles of concurrency (2)



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.



(S)pooling directory



slot	file name
1	∅
2	∅
3	lesson.pptx
4	paperid256.rtf
5	∅
6	∅
7	∅

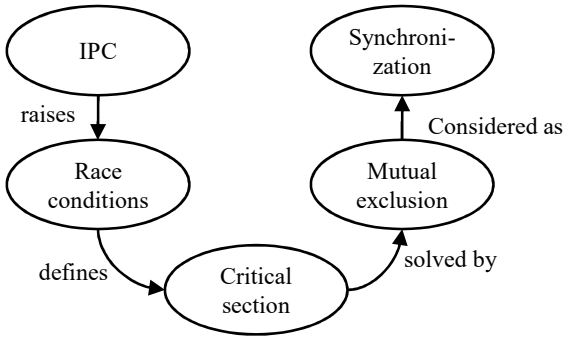
(1) to (7) are atomic instructions

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

Notation

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

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

	in	A.in	B.in	S[7]	out	D.out	D.name
	7	∅	∅	∅	7	6	X.name
A→1	7	7	∅	∅	7	6	X.name
B→1,2,3	8	7	7	B.name	7	6	X.name
A→2,3	8	7	7	A.name	7	6	X.name
D→4,5,6,7	8	7	7	A.name	8	7	A.name

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

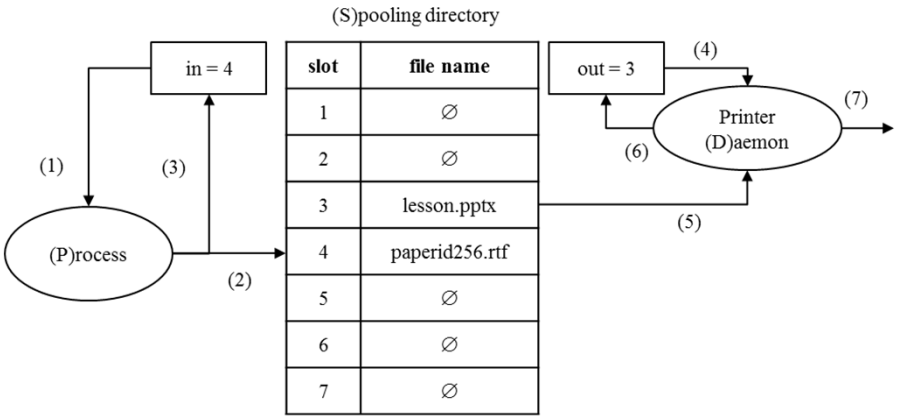
D prints the file of A, the B one will be never processed

P→x,y process P executes the instructions x,y

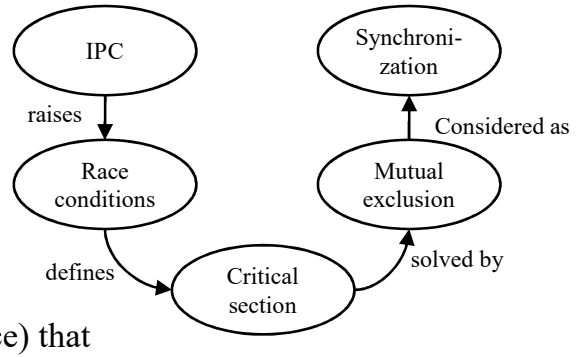
P	loop	
	(1)	P.in=in
	(2)	S[P.in] = P.name
	(3)	in = P.in+1
D	loop	
	(4)	D.out=out
	(5)	D.name=S[D.out]
	(6)	out = D.out+1
	(7)	print

Notation

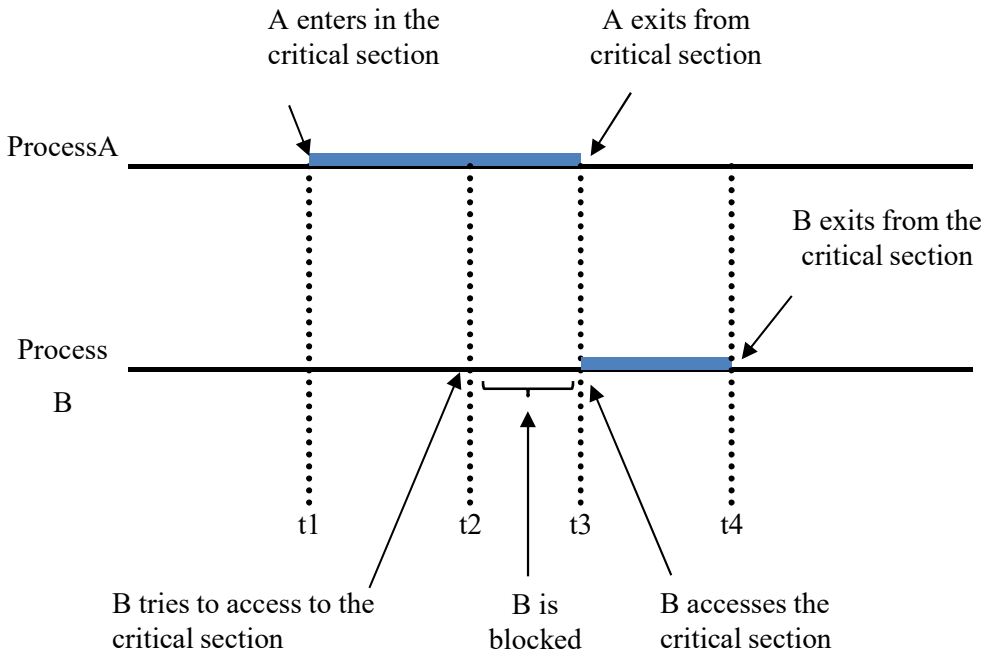
S	the spooling directory
in	current writing index of S
out	current reading index of S
P	a process
D	the printer daemon process
X.a	A data a part of a process X



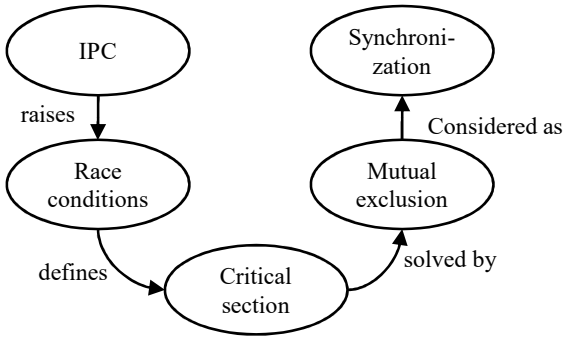
Principles of concurrency (4)



Critical section is a piece of code that accesses a shared resource (a data structure or a device) that must not be concurrently accessed by other concurrent/cooperating processes. A critical section will usually terminate within a 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 piece of code of the critical section.

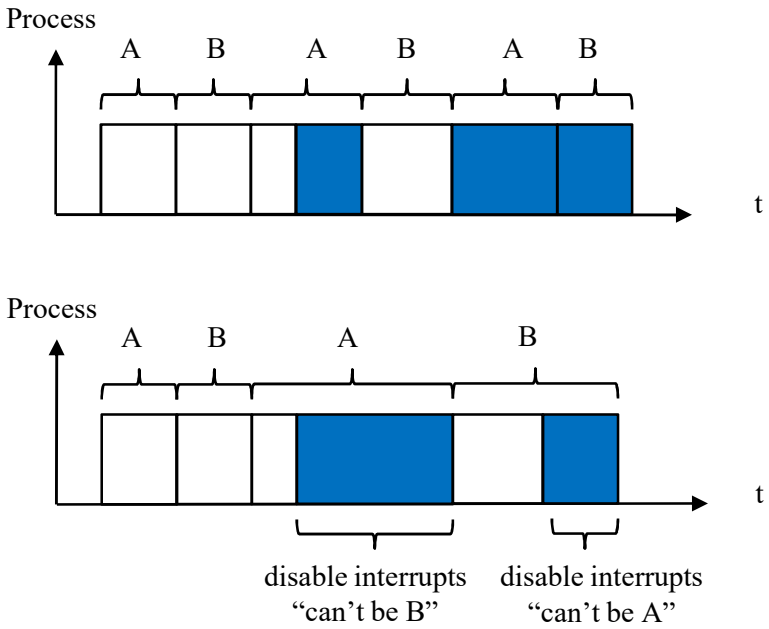
Process synchronization: refers to the idea that multiple processes join up to reach an agreement or to commit a sequence of action.

Operating Systems
“IPC and synchronization”

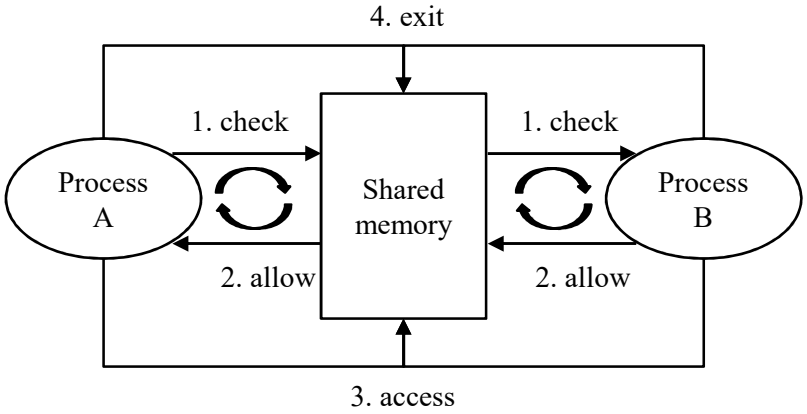
1. Introduction
2. Synchronization for mutual exclusion
   2.1. Principles of concurrency
   2.2. Synchronization methods for mutual exclusion
3. Synchronization for coordination
   3.1. Some problems of coordination
   3.2. Solving the Producer/Consumer problem
   3.3. Solving the multiple Producer/Consumer problem

Synchronization methods for mutual exclusion "Introduction" (1)

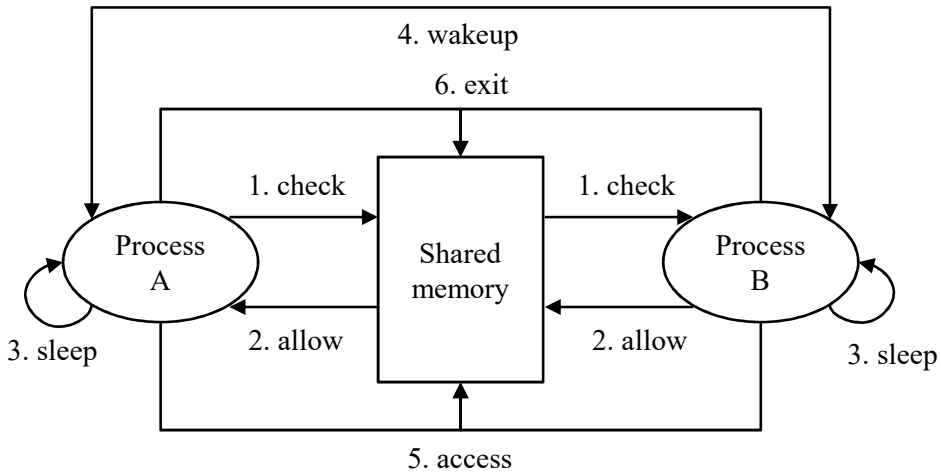
1. Scheduling while disabling the interrupts



2. Busy-waiting



3. Sleep and wakeup



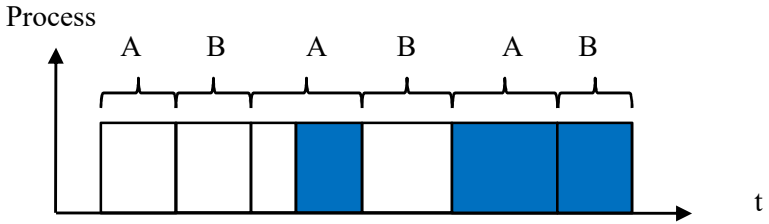
Synchronization methods for mutual exclusion “Introduction” (2)

<table>
<tr><th>Methods</th><th>Approach</th><th>Type</th><th>Process</th><th>Ordering</th><th>Starvation</th></tr>
<tr><td>disabling the interrupts</td><td>disabling the interrupts</td><td rowspan="2">hardware</td><td rowspan="2">≥2</td><td>yes</td><td>no</td></tr>
<tr><td>Swap, TSL, CAS</td><td rowspan="2">busy-waiting</td><td rowspan="2">no</td><td rowspan="2">possible</td></tr>
<tr><td>Perterson’s algorithm</td><td rowspan="2">software</td><td>2</td></tr>
<tr><td>binary semaphore / mutex</td><td>sleep and wakeup</td><td>≥2</td><td>yes</td><td>no</td></tr>
</table>

Synchronization methods for mutual exclusion
"Disabling the interrupts"

Disabling the interrupts: within an uniprocessor system, processes cannot have an overlapped execution. To guarantee a 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 the interrupts when entering in a critical section. e.g.

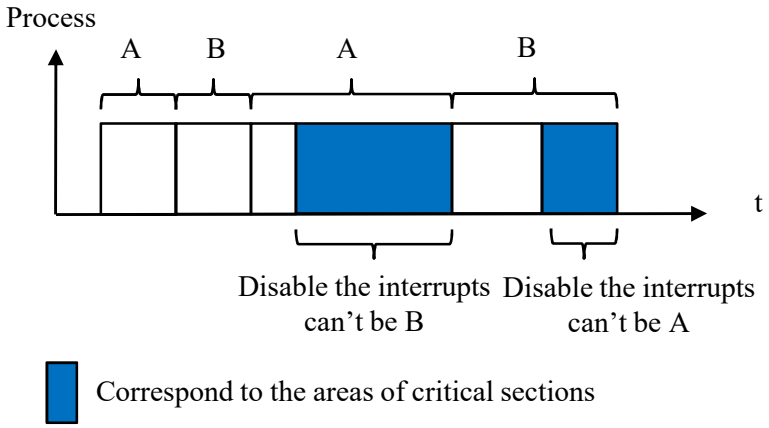
Scheduling two processes A, B accessing a critical section with interruption



Scheduling two processes A, B accessing a critical section while disabling the interrupts

Access a critical section
disable the interrupts

Release a critical section
enable the interrupts



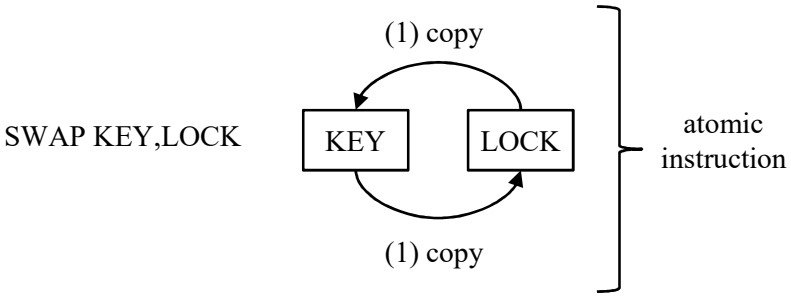
The price of this approach is high:
✓the scheduling performance could be noticeably degraded,
✓this approach cannot work in a multi-processor architecture.

Synchronization methods for mutual exclusion

<table>
<tr><th>Methods</th><th>Approach</th><th>Type</th><th>Process</th><th>Ordering</th><th>Starvation</th></tr>
<tr><td>disabling the interrupts</td><td>disabling the interrupts</td><td rowspan="2">hardware</td><td rowspan="2">≥2</td><td>yes</td><td>no</td></tr>
<tr><td>Swap, TSL, CAS</td><td rowspan="2">busy-waiting</td><td rowspan="2">no</td><td rowspan="2">possible</td></tr>
<tr><td>Perterson’s algorithm</td><td rowspan="2">software</td><td>2</td></tr>
<tr><td>binary semaphore / mutex</td><td>sleep and wakeup</td><td>≥2</td><td>yes</td><td>no</td></tr>
</table>

Synchronization methods for mutual exclusion "Swap, TSL and CAS" (1)

Swap (or exchange) is an hardware instruction, exchanging in one-shot the content of two locations, atomically.



	KEY	LOCK
	\$1	\$0
SWAP KEY,LOCK	\$0	\$1

Access to the section
LOCK at 0, KEY at 1
both shift their values

	KEY	LOCK
	\$1	\$1
SWAP KEY,LOCK	\$1	\$1

Busy-waiting
LOCK and KEY at 1
both keep their values

Request
- (1) Request the critical section with **P**
- (2) set **KEY** at 1
- (3) do Swap **KEY**, **LOCK**
- (4) while **KEY** equals 1

Run in the critical section with **P**
do something ….

Release
- (5) Release the critical section with **P**
- (6) set **LOCK** at 0

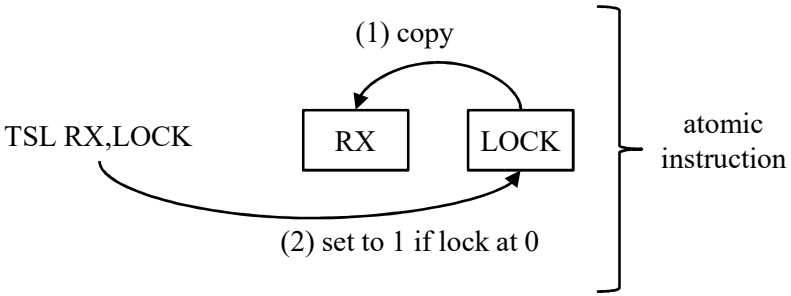
e.g. three processes A, B and C considering the scheduling

	KEYA	KEYB	KEYC	LOCK	Section	
	∅	∅	∅	0	∅	
B→1,2,3	∅	0	∅	1	B	B accesses the section
A→1,2,3,4,3,4,3	1	0	∅	1	B	A is blocked
B→4,5,6	1	0	∅	0	∅	B releases the section
A→4,3	0	0	∅	1	A	A can access
C→1,2,3,4,3,4	0	0	1	1	A	C is blocked
A→4,5,6	0	0	1	0	∅	A releases the section
C→3,4	0	0	0	1	C	C can access
C→5,6	0	0	0	0	∅	C releases the section

P→x,y process P executes the instructions x,y

Synchronization methods for mutual exclusion "Swap, TSL and CAS" (2)

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



	RX	LOCK
	Na	\$0
TSL RX, LOCK	\$0	\$1

Access to the section
RX set to 0
LOCK moves to 1

	RX	LOCK
	Na	\$1
TSL RX, LOCK	\$1	\$1

Busy-waiting
RX set to 1
Nothing happens on LOCK

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

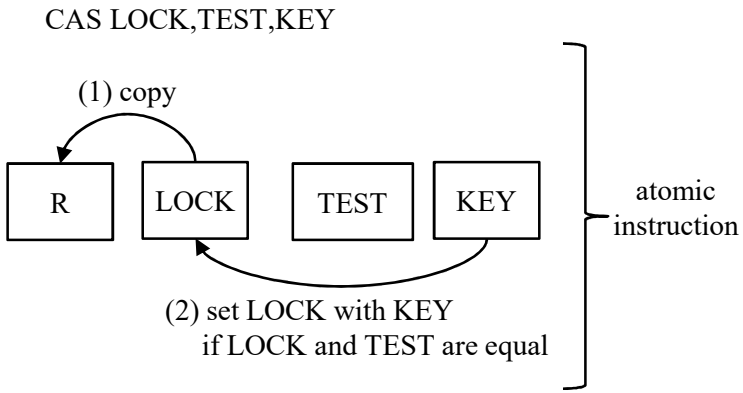
e.g. three processes A, B and C considering the scheduling

	RXA	RXB	RXC	LOCK	Section	
	∅	∅	∅	0	∅	
B→1,2	∅	0	∅	1	B	B accesses the section
A→1,2,3,2,3,2	1	0	∅	1	B	A is blocked
B→3,4,5	1	0	∅	0	∅	B releases the section
A→3,2	0	0	∅	1	A	A can access
C→1,2,3,2,3	0	0	1	1	A	C is blocked
A→3,4,5	0	0	1	0	∅	A releases the section
C→2,3	0	0	0	1	C	C can access
C→4,5	0	0	0	0	∅	C releases the section

P→x,y process P executes the instructions x,y

Synchronization methods for mutual exclusion "Swap, TSL and CAS" (3)

CAS is a tradeoff to the TSL instruction checking a memory location **LOCK** against a test value **TEST**. If they are same, a swap occurs between the **LOCK** and a **KEY** value. The old **LOCK** value (before the swapping) is still returned.



	R	LOCK
	Na	$0
R ← CAS LOCK,0,1	$0	$1

Access to the section
KEY at 1 and TEST, LOCK at 0
LOCK is updated

	R	LOCK
	Na	$1
R ← CAS LOCK,0,1	$1	$1

Busy-waiting
LOCK and TEST different
nothing happens

Request
(1) Request the critical section with **P**
(2) do **R** equals CAS **LOCK**, **0**, **1**
(3) while key **R** 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. three processes A, B and C considering the scheduling

	RA	RB	RC	LOCK	Section	
	∅	∅	∅	0	∅	
B→1,2	∅	0	∅	1	B	B accesses the section
A→1,2,3,2,3,2	1	0	∅	1	B	A is blocked
B→3,4,5	1	0	∅	0	∅	B releases the section
A→3,2	0	0	∅	1	A	A can access
C→1,2,3,2,3	0	0	1	1	A	C is blocked
A→3,4,5	0	0	1	0	∅	A releases the section
C→2,3	0	0	0	1	C	C can access
C→4,5	0	0	0	0	∅	C releases the section

P→x,y process P executes the instructions x,y

Synchronization methods for mutual exclusion

<table>
<tr><th>Methods</th><th>Approach</th><th>Type</th><th>Process</th><th>Ordering</th><th>Starvation</th></tr>
<tr><td>disabling the interrupts</td><td>disabling the interrupts</td><td rowspan="2">hardware</td><td rowspan="2">≥2</td><td>yes</td><td>no</td></tr>
<tr><td>Swap, TSL, CAS</td><td rowspan="2">busy-waiting</td><td rowspan="2">no</td><td rowspan="2">possible</td></tr>
<tr><td>Perterson’s algorithm</td><td rowspan="2">software</td><td>2</td></tr>
<tr><td>binary semaphore / mutex</td><td>sleep and wakeup</td><td>≥2</td><td>yes</td><td>no</td></tr>
</table>

Synchronization methods for mutual exclusion "Peterson's algorithm" (1)

The Peterson's algorithm solves the mutual exclusion problem between two processes. Entrance in the critical section is granted for a process P if the other process doesn't want to enter, or if it has given previously the priority to P.

global variables

P_i, **P_j** are two processes**, i**, **j** are two integers
turn is an integer, **flag** is a boolean table

Request

Request the critical section with **P_i**
flag[i] = true
turn = **j**
while ((**flag[j]** == **true**) && (**turn** == **j**))
(1) (2)
busy-waiting

P_i waits if	
and	(1) **P_j** sets its flag at true (2) **P_j** doesn't set the turn for **P_i**

(1)	(2)	(1) && (2)	while
1	1	1	wait
0	0	0	access
0	1	0	access
1	0	0	access

Run in the critical section with **P_i**
do something ….

Release

Release the critical section with **P_i**
flag[i] = **false**

P_i accesses the critical section if	
or	(1) **P_j** sets its flag at false (2) **P_j** sets the turn for **P_i**

Synchronization methods for mutual exclusion "Peterson's algorithm" (2)

Request

(1) Request the critical section with $\mathbf{P_i}$
(2) **flag[i]** = true
(3) **turn** = **j**
(4) while ((**flag[j]** == **true**) && (**turn** == **j**))
(5) busy-waiting

Release

(6) Release the critical section with $\mathbf{P_i}$
(7) **flag[i]** = **false**

$\mathbf{P_i}$ accesses the critical section if	
or	(1) $\mathbf{P_j}$ sets its flags at false (2) $\mathbf{P_j}$ sets the turn for $\mathbf{P_i}$

e.g. two processes A, B considering the scheduling

	turn	flag		Section	
		A	B		
	∅	false	false	∅	
B→1,2	∅	false	true	∅	B sets its flag at true
A→1,2,3,4,5,4,5	B	true	true	∅	A is blocked because the flag of B is true and turn is set with the B value
B→3	A	true	true	∅	B sets the turn variable to A
A→4,6,7	A	false	true	A-∅	A accesses the section as the turn variable is set to A
B→4,6,7	A	false	false	B-∅	B accesses the section as the flag of A is false

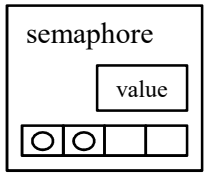
P→x,y process P executes the instructions x,y

Synchronization methods for mutual exclusion

<table>
<tr><th>Methods</th><th>Approach</th><th>Type</th><th>Process</th><th>Ordering</th><th>Starvation</th></tr>
<tr><td>disabling the interrupts</td><td>disabling the interrupts</td><td rowspan="2">hardware</td><td rowspan="2">≥2</td><td>yes</td><td>no</td></tr>
<tr><td>Swap, TSL, CAS</td><td rowspan="2">busy-waiting</td><td rowspan="2">no</td><td rowspan="2">possible</td></tr>
<tr><td>Perterson’s algorithm</td><td rowspan="2">software</td><td>2</td></tr>
<tr><td>binary semaphore / mutex</td><td>sleep and wakeup</td><td>≥2</td><td>yes</td><td>no</td></tr>
</table>

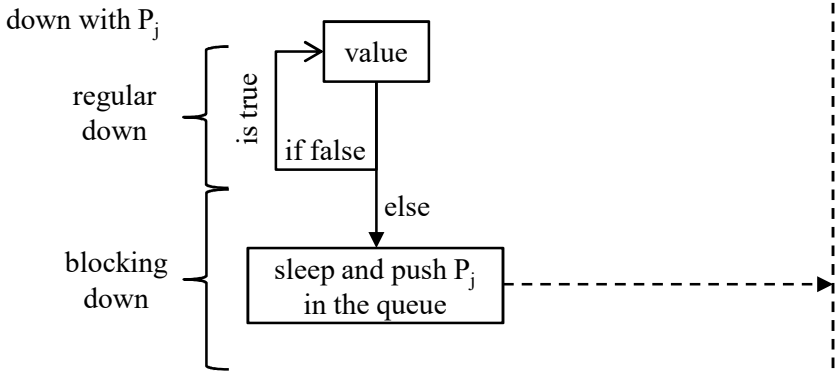
Synchronization methods for mutual exclusion "binary semaphores / mutex" (1)

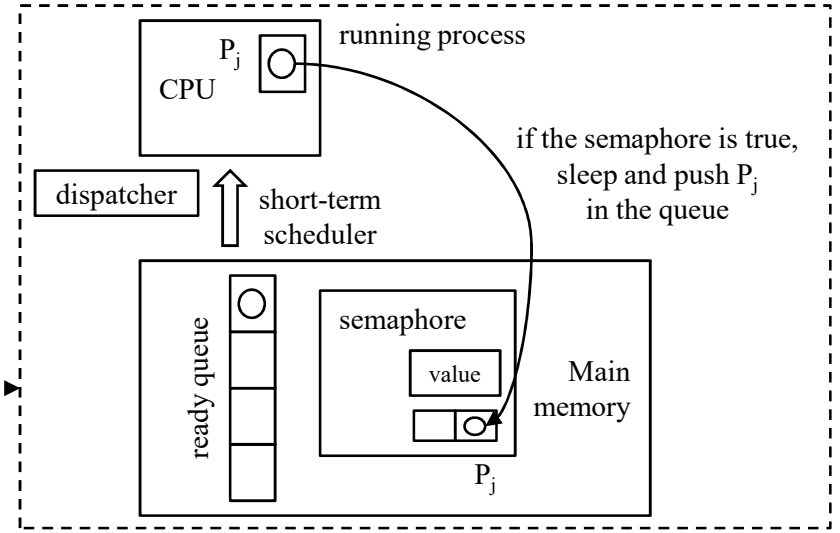
Semaphore is a synchronization primitive composed of a blocking queue and a variable controlled with two operations **down / up**.



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

The **down** operation decreases the value of the semaphore or sleeps the current process and pushes it into the queue.





regular down

	before	after
value	false	true
queue	∅	∅

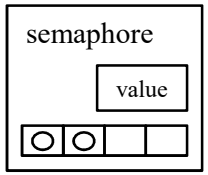
blocking down

	before	after
value	true	true
queue	∅	P

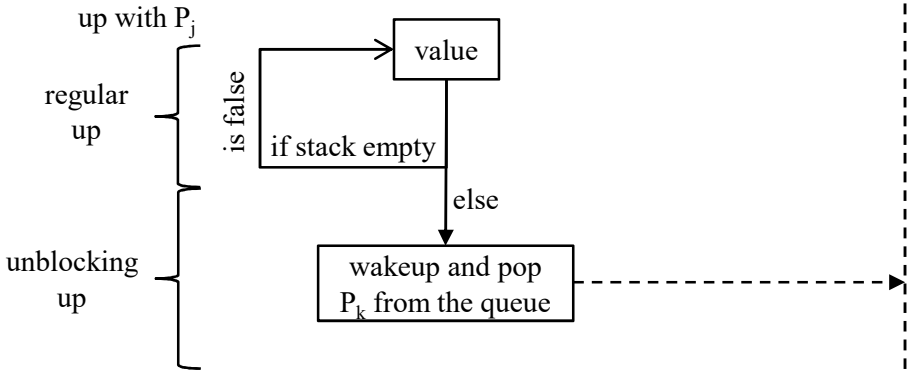
Synchronization methods for mutual exclusion "binary semaphores / mutex" (2)

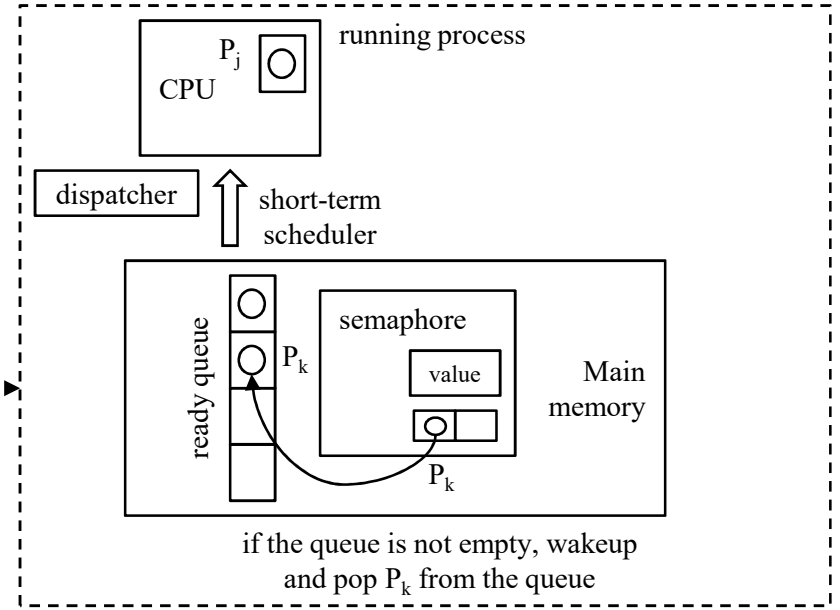
Semaphore is a synchronization primitive composed of a blocking queue and a variable controlled with two operations **down / up**.

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



The **up** operation increases the value of the semaphore or wakeups a process in the queue.





regular up

	before	after
value	true	false
queue	∅	∅

unblocking up

	before	after
value	true	true
queue	P	∅

Synchronization methods for mutual exclusion "binary semaphores / mutex" (3)

The algorithm for mutual exclusion using a binary semaphore is

sem is a semaphore, **P** is the process, (1) to (5) the instructions

(1) before the request
do something ….
(2) down **sem**
(3) run in the critical section with **P**
do something ….
(4) before the release
do something ….
(5) up **sem**

e.g. three processes A, B and C considering the scheduling, the solution is presented with a table

	sem		Section	A state	B state	C state	
	value	Q					
	false	∅	∅	ready	ready	ready	
A→1,2,3	true	∅	A	ready	ready	ready	**A** accesses the section, sem becomes true
B→1,2	true	B	A	ready	blocked	ready	while accessing the semaphore, **B** blocks
C→1,2	true	C,B	A	ready	blocked	blocked	while accessing the semaphore, **C** blocks
A→4,5	true	C	A-B	ready	ready	blocked	**A** exits and pops up **B**, **B** holds the section
B→3,4,5	true	∅	B-C	ready	ready	ready	**B** exits and pops up **C**, **C** holds the section
C→3,4,5	false	∅	C-∅	ready	ready	ready	**C** exits and puts the semaphore to false

P→x,y process P executes the instructions x,y

regular down

	before	**after**
value	false	true
queue	∅	∅

blocking down

	before	**after**
value	true	true
queue	∅	P

regular up

	before	**after**
value	true	false
queue	∅	∅

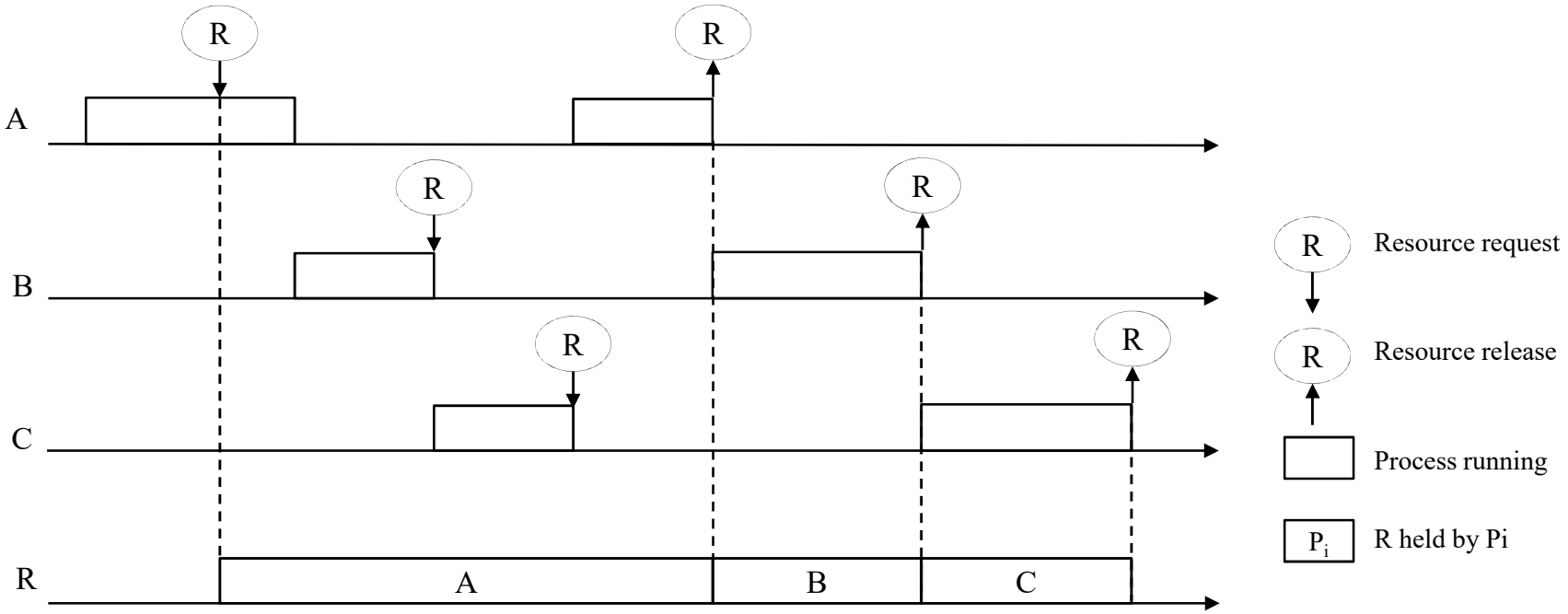
unblocking up

	before	**after**
value	true	true
queue	P	∅

Synchronization methods for mutual exclusion “binary semaphores / mutex” (4)

The algorithm for mutual exclusion using a binary semaphore is

e.g. three processes A, B and C considering the scheduling, the solution is diagram



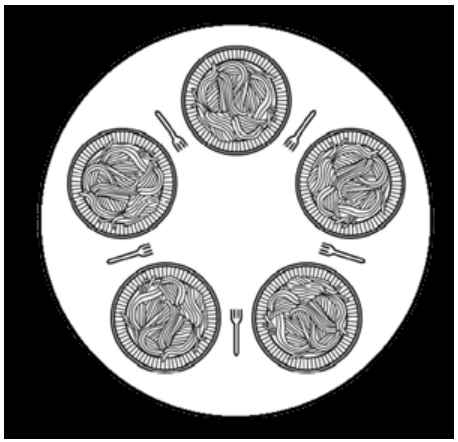
Operating Systems
“IPC and synchronization”

1. Introduction
2. Synchronization for mutual exclusion
   2.1. Principles of concurrency
   2.2. Synchronization methods for mutual exclusion
3. Synchronization for coordination
   3.1. Some problems of coordination
   3.2. Solving the Producer/Consumer problem
   3.3. Solving the multiple Producer/Consumer problem

Some problems of coordination (1)

The **dinning-philosophers problem** is summarized as:

(1) five philosophers sitting at a table are doing one of the two things: eating or thinking.
(2) a fork is placed in between each pair of adjacent philosophers.
(3) while eating, they are not thinking, and while thinking, they are not eating.
(4) a philosopher must eat with two forks (i.e. if thinking, none fork are used).
(5) each philosopher can only use the forks on his immediate left and immediate right.



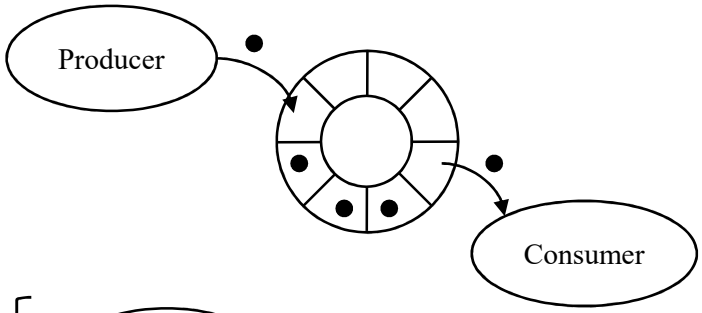
The **readers-writer problem** concerns synchronization of processes when accessing the same database in a R/W mode. It is summarized as:

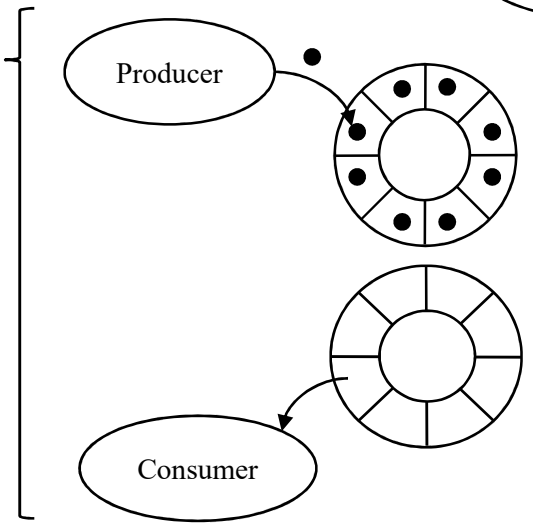
(1) several processes can read the database at the same time.
(2) when at least a process reads, no one can write.
(3) only a single process can write at the same time.
(4) when a process writes, no one can read.

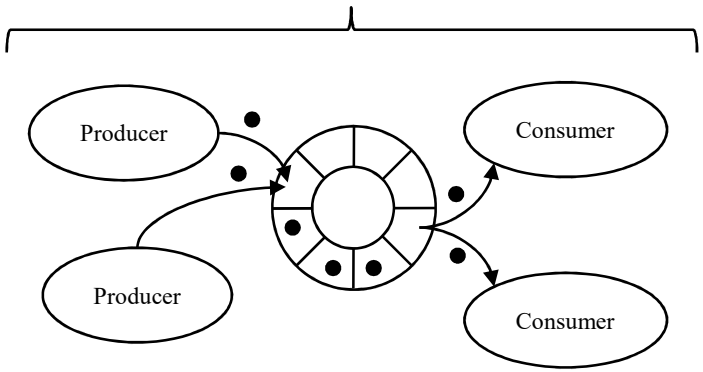
Some problems of coordination (2)

The producer-consumer (i.e. **bounded buffer)** describes how processes share a common buffer. The problem is to make sure that a process will not try to add data into the buffer if it's full, or to remove data if empty.
The problem is summarized as:

(1) the producer and the consumer share a common, fixed-size, buffer,
(2) the producer puts information into the buffer, and the consumer takes it out,
(3) processes are blocked when the size limit is reached, empty for the consumer or full for the producer,
(4) processes are unblocked when the buffer recovers a regular size,
(5) we can generalize the problem to m producers and n consumers, but this extends the synchronization with a mutual exclusion while accessing the buffer.







Operating Systems
“IPC and synchronization”

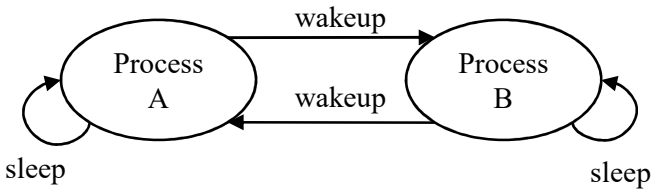
1. Introduction
2. Synchronization for mutual exclusion
   2.1. Principles of concurrency
   2.2. Synchronization methods for mutual exclusion
3. Synchronization for coordination
   3.1. Some problems of coordination
   3.2. Solving the Producer/Consumer problem
   3.3. Solving the multiple Producer/Consumer problem

Solving the Producer/Consumer problem "Introduction"

<table>
<tr><th>Methods</th><th>Approach</th><th>Type</th><th>Application problem</th><th>Coordination type</th></tr>
<tr><td>sleep and wakeup</td><td rowspan="4">sleep and wakeup</td><td rowspan="4">software</td><td rowspan="2">Producer / Consumer</td><td>coordination by communication</td></tr>
<tr><td>semaphore</td><td rowspan="3">coordination by sharing</td></tr>
<tr><td>semaphore / mutex</td><td rowspan="2">Multiple Producers / Consumers</td></tr>
<tr><td>monitor</td></tr>
</table>

Solving the Producer/Consumer problem "sleep wakeup" (1)

Sleep and wakeup are atomic actions to change the states of processes for synchronization.



The producer/consumer algorithm is

consumer, **producer** are processes

```
consumer
    loop
(1)     if buffer is empty
(2)         sleep
(3)     pop item from buffer
(4)     if buffer was full (i.e. actual size = n-1)
(5)         wakeup producer
```

e.g. two processes P, C with a successful synchronization

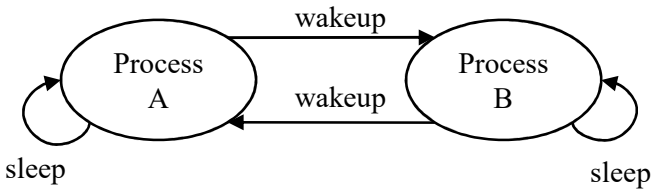
	buffer	P state	C state	
	0	awake	sleepy	
P→1,3,4,5	1	awake	awake	P wakeups C
C→3,4	0	awake	awake	C restarts at Program Counter
P→1,3,4,5	1	awake	awake	here is a lost wakeup
C→1,3,4,1,2	0	awake	sleepy	when empty, C will sleep

P→x,y process P executes the instructions x,y

```
producer
    loop
(1)     if buffer is full
(2)         sleep
(3)     push a new item in buffer
(4)     if buffer was empty (i.e. actual size =1)
(5)         wakeup consumer
```

Solving the Producer/Consumer problem "sleep wakeup" (2)

Sleep and wakeup are atomic actions to change the states of processes for synchronization.



The producer/consumer algorithm is

consumer, **producer** are processes

```
consumer
    loop
(1)     if buffer is empty
(2)         sleep
(3)     pop item from buffer
(4)     if buffer was full (i.e. actual size = n-1)
(5)         wakeup producer
```

```
producer
    loop
(1)     if buffer is full
(2)         sleep
(3)     push a new item in buffer
(4)     if buffer was empty (i.e. actual size =1)
(5)         wakeup consumer
```

e.g. two processes P, C with a synchronization and failure

	buffer	P state	C state	
	0	awake	awake	
C→1	0	awake	awake	
P→1,3,4,5	1	awake	awake	here is a lost wakeup
C→2	1	awake	sleepy	C blocked on sleep
P→1,3,4	2	awake	sleepy	fill in buffer
P→1,3,4	3	awake	sleepy	
….	…	…	…	
P→1,2	n	sleepy	sleepy	P, C will sleep for always

P→x,y process P executes the instructions x,y

Solving the Producer/Consumer problem "sleep wakeup" (3)

Sleep and wakeup with a wakeup waiting bit is an extension of the method to support the lost wakeups.

The producer/consumer algorithm is

consumer, **producer** are processes

```
consumer
     loop
(1)      if buffer is empty
(2)           sleep
(3)      pop item from buffer
(4)      if buffer was full (i.e. actual size = n-1)
(5)           wakeup producer
```

```
producer
     loop
(1)      if buffer is full
(2)           sleep
(3)      push a new item in buffer
(4)      if buffer was empty (i.e. actual size =1)
(5)           wakeup consumer
```

command	process state: ready	process state: waiting
wakeup	put to 1	wakeup

command	wakeup waiting bit: 0	wakeup waiting bit: 1
sleep	sleep	put to 0

e.g. two processes P, C with a successful synchronization

	buffer	ww bits P	ww bits C	P state	C state	
	0	0	0	awake	awake	
C→1	0	0	0	awake	awake	
P→1,3,4,5	1	0	1	awake	awake	C gets a bit
C→2	1	0	0	awake	awake	C uses the bit
P→1,3,4	2	0	0	awake	awake	
C→3,4	1	0	0	awake	awake	
….	…			…	…	the synchronization will go on

P→x,y process P executes the instructions x,y

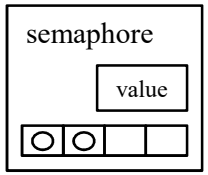
Solving the Producer/Consumer problem

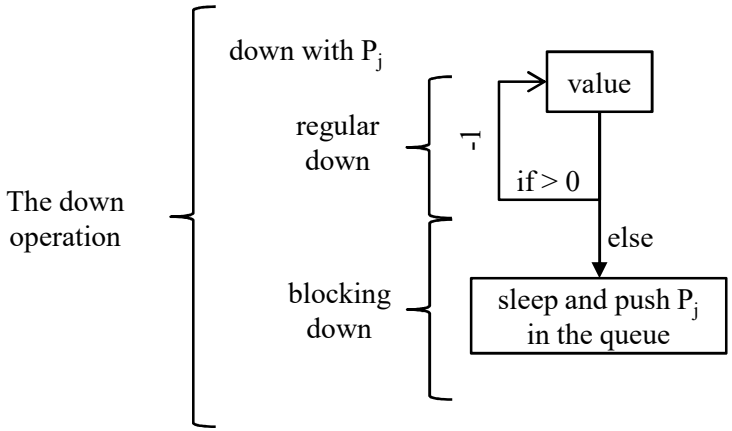
<table>
<tr><th>Methods</th><th>Approach</th><th>Type</th><th>Application problem</th><th>Coordination type</th></tr>
<tr><td>sleep and wakeup</td><td rowspan="4">sleep and wakeup</td><td rowspan="4">software</td><td rowspan="2">Producer / Consumer</td><td>coordination by communication</td></tr>
<tr><td>semaphore</td><td rowspan="3">coordination by sharing</td></tr>
<tr><td>semaphore / mutex</td><td rowspan="2">Multiple Producers / Consumers</td></tr>
<tr><td>monitor</td></tr>
</table>

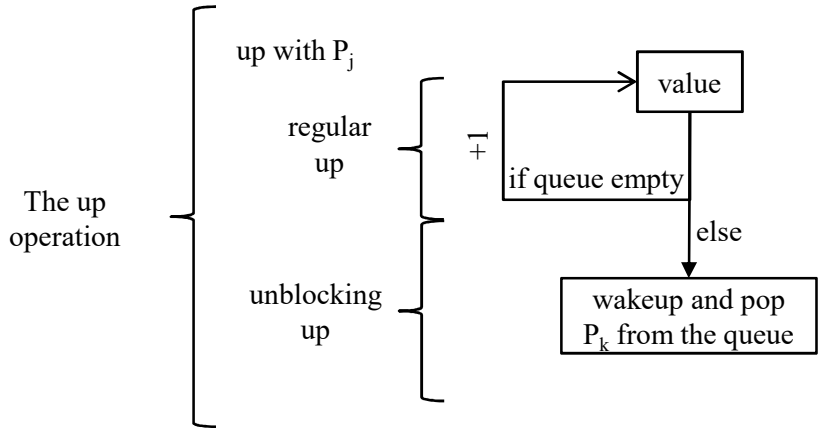
Solving the Producer/Consumer problem "semaphores" (1)

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

A **counting (or a general) semaphore** is not a binary semaphore, it embeds a variable covering the range [0,+∞ [.







regular down

	before	after
value	2	1
queue	∅	∅

blocking down

	before	after
value	0	0
queue	∅	P

regular up

	before	after
value	2	3
queue	∅	∅

unblocking up

	before	after
value	0	0
queue	P	∅

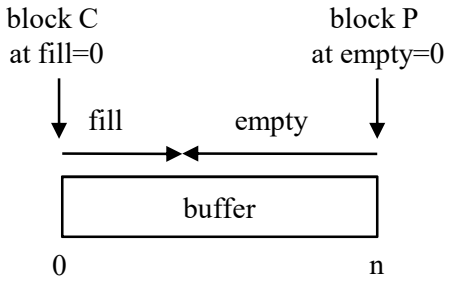
Solving the Producer/Consumer problem "semaphores" (2)

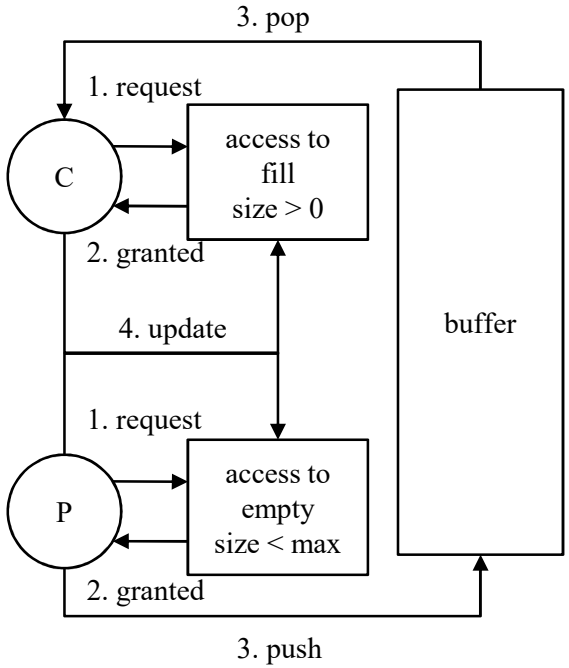
The algorithm for solving the producer/consumer problem with semaphores is

fill = 0, **empty** = n are semaphores
buffer is the data structure

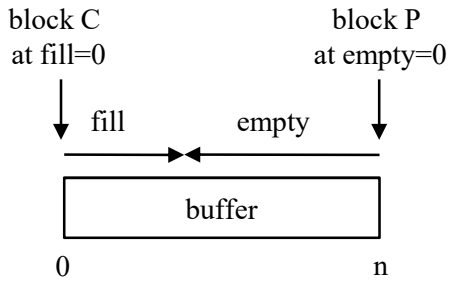
consumer
loop
(1) down **fill**
(2) pop **item** from **buffer**
(3) up **empty**

producer
loop
(1) down **empty**
(2) push a new **item** in **buffer**
(3) up **fill**





Solving the Producer/Consumer problem "semaphores" (3)



The algorithm for solving the producer/consumer problem with semaphores is

fill = 0, **empty** = n are semaphores
buffer is the data structure

consumer
loop
(1) down **fill**
(2) pop **item** from **buffer**
(3) up **empty**

producer
loop
(1) down **empty**
(2) push a new **item** in **buffer**
(3) up **fill**

e.g. two processes P, C with n=2

	buffer	fill		empty		P state	C state	
		value	Q	value	Q			
	0	0	∅	2	∅	ready	ready	
C→1	0	0	C	2	∅	ready	blocked	C sleeps at down on fill
P→1,2,3	1	0	∅	1	∅	ready	ready	P wakeups C at up on fill
C→2,3	0	0	∅	2	∅	ready	ready	next scheduling, C restarts on pop
P→1,2,3	1	1	∅	1	∅	ready	ready	fill in buffer
P→1,2,3	2	2	∅	0	∅	ready	ready	
P→1	2	2	∅	0	P	blocked	ready	P stopped at down on empty

P→x,y process P executes the instructions x,y

regular down

	before	**after**
value	2	1
queue	∅	∅

blocking down

	before	**after**
value	0	0
queue	∅	P

regular up

	before	**after**
value	2	3
queue	∅	∅

unblocking up

	before	**after**
value	0	0
queue	P	∅

Operating Systems
“IPC and synchronization”

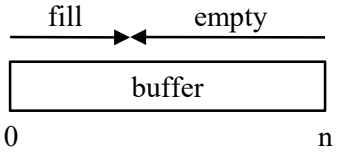
1. Introduction
2. Synchronization for mutual exclusion
   2.1. Principles of concurrency
   2.2. Synchronization methods for mutual exclusion
3. Synchronization for coordination
   3.1. Some problems of coordination
   3.2. Solving the Producer/Consumer problem
   3.3. Solving the multiple Producer/Consumer problem

Solving the multiple Producer/Consumer problem "Introduction"

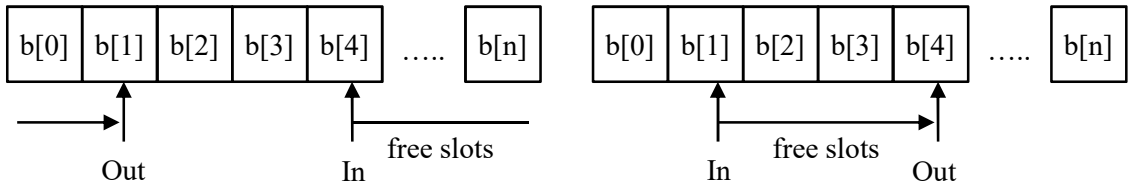
<table>
<tr><th>Methods</th><th>Approach</th><th>Type</th><th>Application problem</th><th>Coordination type</th></tr>
<tr><td>sleep and wakeup</td><td rowspan="4">sleep and wakeup</td><td rowspan="4">software</td><td rowspan="2">Producer / Consumer</td><td>coordination by communication</td></tr>
<tr><td>semaphore</td><td rowspan="3">coordination by sharing</td></tr>
<tr><td>semaphore / mutex</td><td rowspan="2">Multiple Producers / Consumers</td></tr>
<tr><td>monitor</td></tr>
</table>

Solving the multiple Producer/Consumer problem "semaphores - mutex" (1)

fill and empty work from a buffer having a size bounded between 0 to n.



The buffer is treated as a circular storage, and pointer values must be expressed modulo the size of the buffer. Therefore, we can have In > Out or In < Out depending the access case.



The pop and push operations are then not atomic.

pop
(1) w = b[out]
(2) out = (out+1)%(n+1)

push
(1) b[in] = v
(2) in = (in+1)%(n+1)

In addition, the buffer slots are data-dependent (e.g. byte, double, data structure, etc.), the read/write operations b[] could be instruction sets.

The one-to-one solution to the Producer/Consumer problem with a bounded-buffer.

fill = 0, **empty** = n are semaphores
buffer is the data structure

consumer
loop
(1) down **fill**
(2) w= b[out]
(3) out = (out+1)%(n+1)
(4) up **empty**

producer
loop
(1) down **empty**
(2) b[in] = v
(3) in = (in+1)%(n+1)
(4) up **fill**

Solving the multiple Producer/Consumer problem "semaphores - mutex" (2)

Applying the one-to-one solution to the bounded-buffer problem with multiple producers and/or consumers.

fill = 0, **empty** = n are semaphores
buffer is the data structure

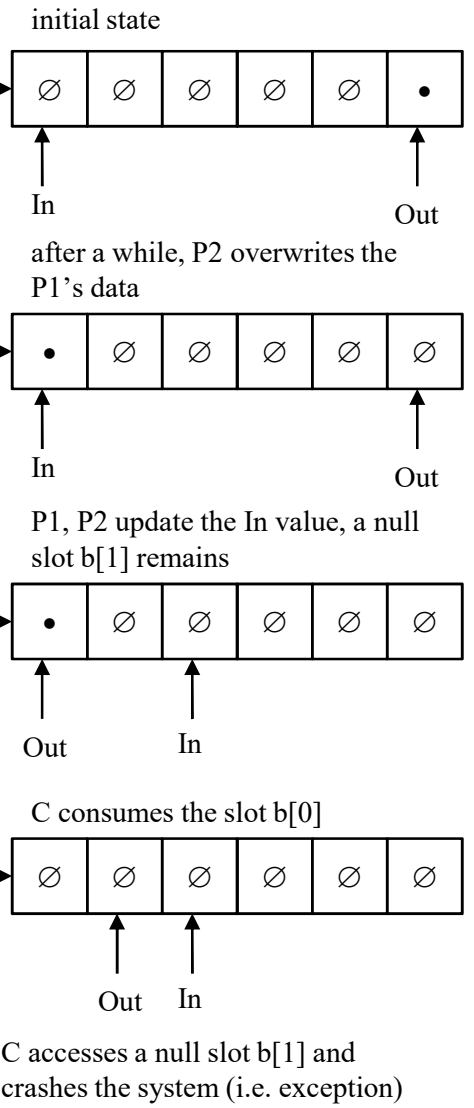
consumer
loop
(1) down **fill**
(2) w= b[out]
(3) out = (out+1)%(n+1)
(4) up **empty**

producer
loop
(1) down **empty**
(2) b[in] = v
(3) in = (in+1)%(n+1)
(4) up **fill**

e.g. two producers P1, P2, one consumer C with n=5

	buffer	In	Out	fill		empty	
				value	Q	value	Q
	1	0	5	1	∅	5	∅
C→1,2	0	0	5	0	∅	5	∅
P1→1,2	1	0	5	0	∅	4	∅
P2→1,2	1	0	5	0	∅	3	∅
C→3,4	1	0	0	0	∅	4	∅
P1→3,4	1	1	0	1	∅	4	∅
P2→3,4	1	2	0	2	∅	4	∅
C→1,2,3,4	0	2	1	1	∅	5	∅
C→1,2	0	2	2	0	∅	5	∅

P→x,y process P executes the instructions x,y



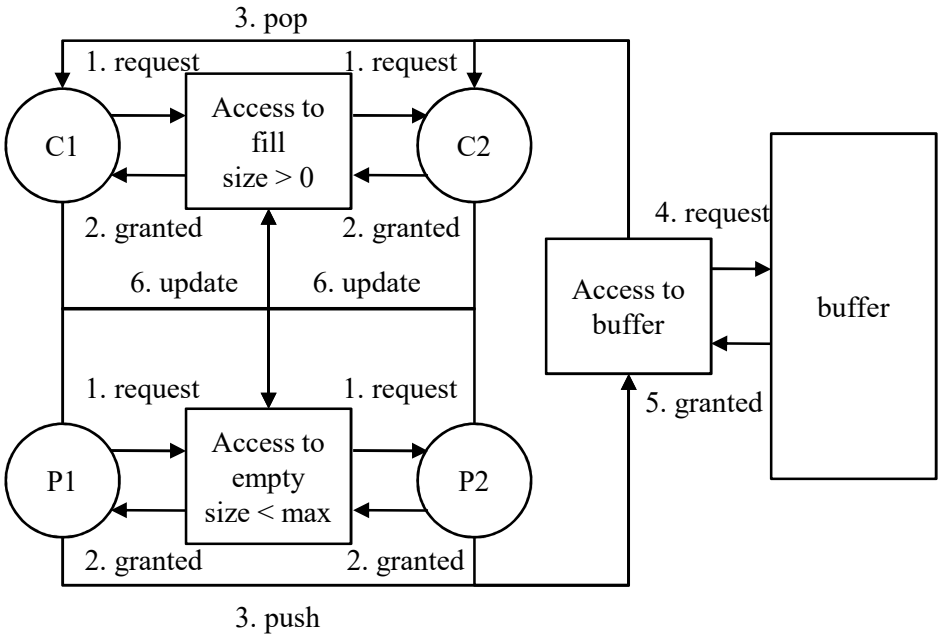
Solving the multiple Producer/Consumer problem "semaphores - mutex" (3)

The solution is then to protect access to buffer with a mutex. The general algorithm for solving the multiple producer/consumer problem with semaphore becomes

fill = 0, **empty** = n are semaphores
mutex is a mutex
buffer is the data structure

consumer
loop
(1) down **fill**
(2) down **mutex**
(3) pop **item** from **buffer**
(4) up **mutex**
(5) up **empty**

producer
loop
(1) down **empty**
(2) down **mutex**
(3) push a new **item** in **buffer**
(4) up **mutex**
(5) up **fill**



Solving the multiple Producer/Consumer problem "semaphores - mutex" (4)

The solution is then to protect access to buffer with a mutex. The general algorithm for solving the multiple producer/consumer problem with semaphore becomes

fill = 0, **empty** = n are semaphores
mutex is a mutex
buffer is the data structure

consumer
loop
(1) down **fill**
(2) down **mutex**
(3) pop **item** from **buffer**
(4) up **mutex**
(5) up **empty**

producer
loop
(1) down **empty**
(2) down **mutex**
(3) push a new **item** in **buffer**
(4) up **mutex**
(5) up **fill**

e.g. two producers P1, P2, one consumer C with n = 4

	buffer	fill		empty		mutex		
		value	Q	value	Q	value	Q	Section
	0	0	∅	4	∅	false	∅	∅
P1→1,2	0	0	∅	3	∅	true	∅	P1
P2→1,2	0	0	∅	2	∅	true	P2	P1
P1→3,4,5	1	1	∅	2	∅	true	∅	P2
P2→3	2	1	∅	2	∅	true	∅	P2
C→1,2	2	0	∅	2	∅	true	C	P2
P2→4,5	2	1	∅	2	∅	true	∅	C
C→3,4,5	1	1	∅	3	∅	false	∅	∅

P→x,y process P executes the instructions x,y

P2 is blocked on mutex while P1 accesses the buffer, here mutual exclusion applies

C is blocked on mutex while P2 accesses the buffer, here there is no mutual exclusion

Solving the multiple Producer/Consumer problem "semaphores - mutex" (5)

Inverting the code for solving the multiple producer/consumer problem will result in a deadlock.

fill = 0, **empty** = n are semaphores
mutex is a mutex
buffer is the data structure

consumer
loop
(1) down **mutex**
(2) down **fill**
(3) pop **item** from **buffer**
(4) up **empty**
(5) up **mutex**

we shift
we shift

producer
loop
(1) down **mutex**
(2) down **empty**
(3) push a new **item** in **buffer**
(4) up **fill**
(5) up **mutex**

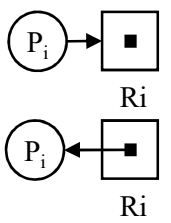
we shift
we shift

e.g. one producers P, one consumer C

	empty		mutex			P state	C state
	value	Q	value	Q	Section		
	0	∅	false	∅	∅	ready	ready
P→1,2	0	P	true	∅	P	blocked	ready
C→1	0	P	true	C	P	waiting	blocked

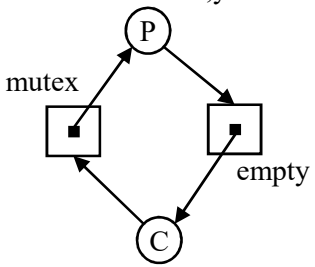
P,C will sleep for always

P→x,y process P executes the instructions x,y



Pi waits for Ri

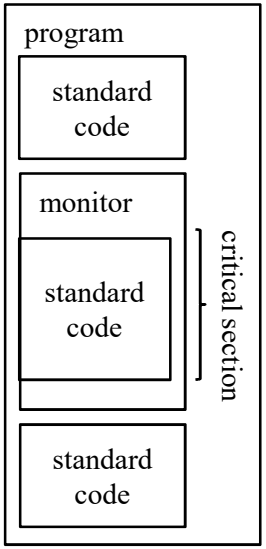
Pi holds Ri



Solving the multiple Producer/Consumer problem

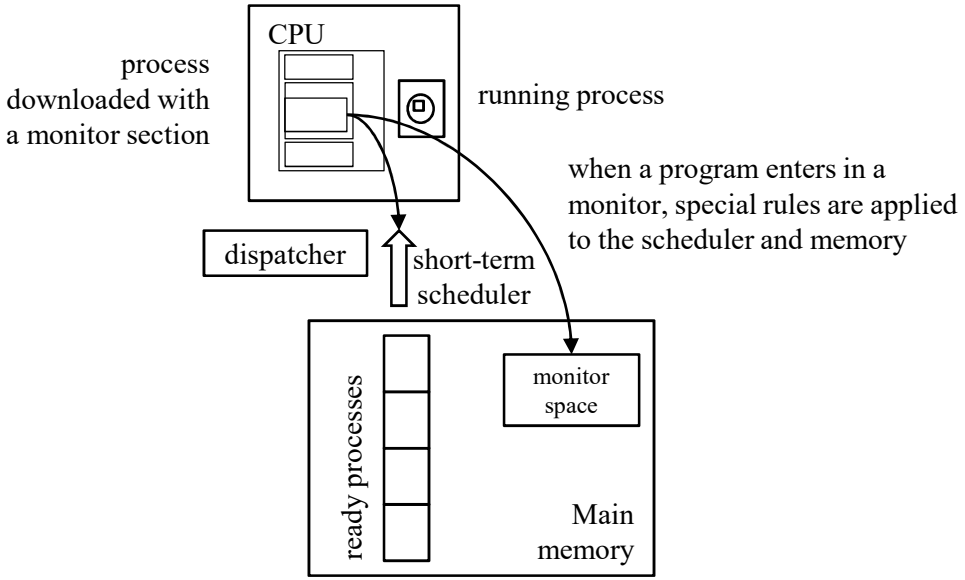
<table>
<tr><th>Methods</th><th>Approach</th><th>Type</th><th>Application problem</th><th>Coordination type</th></tr>
<tr><td>sleep and wakeup</td><td rowspan="4">sleep and wakeup</td><td rowspan="4">software</td><td rowspan="2">Producer / Consumer</td><td>coordination by communication</td></tr>
<tr><td>semaphore</td><td rowspan="3">coordination by sharing</td></tr>
<tr><td>semaphore / mutex</td><td rowspan="2">Multiple Producers / Consumers</td></tr>
<tr><td>monitor</td></tr>
</table>

Solving the multiple Producer/Consumer problem “monitor” (1)

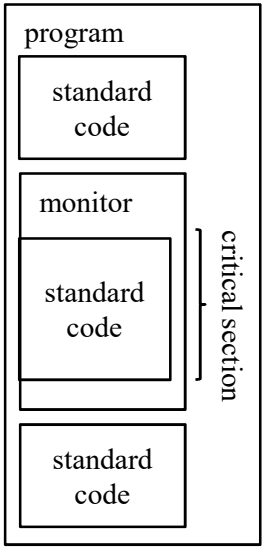


A **monitor** is a special piece of code, associated to condition variables, that are providing mutual exclusion within the monitor. Special rules are applied to the scheduler and memory:

1. only one process at a time can access the monitor,
2. irregular in/out of monitor by processes are controlled with two operations, **wait** and **signal**, to be applied on the condition variables that are close to semaphore mechanisms,
3. monitors are given in two implementations, Mesa and Hoare.

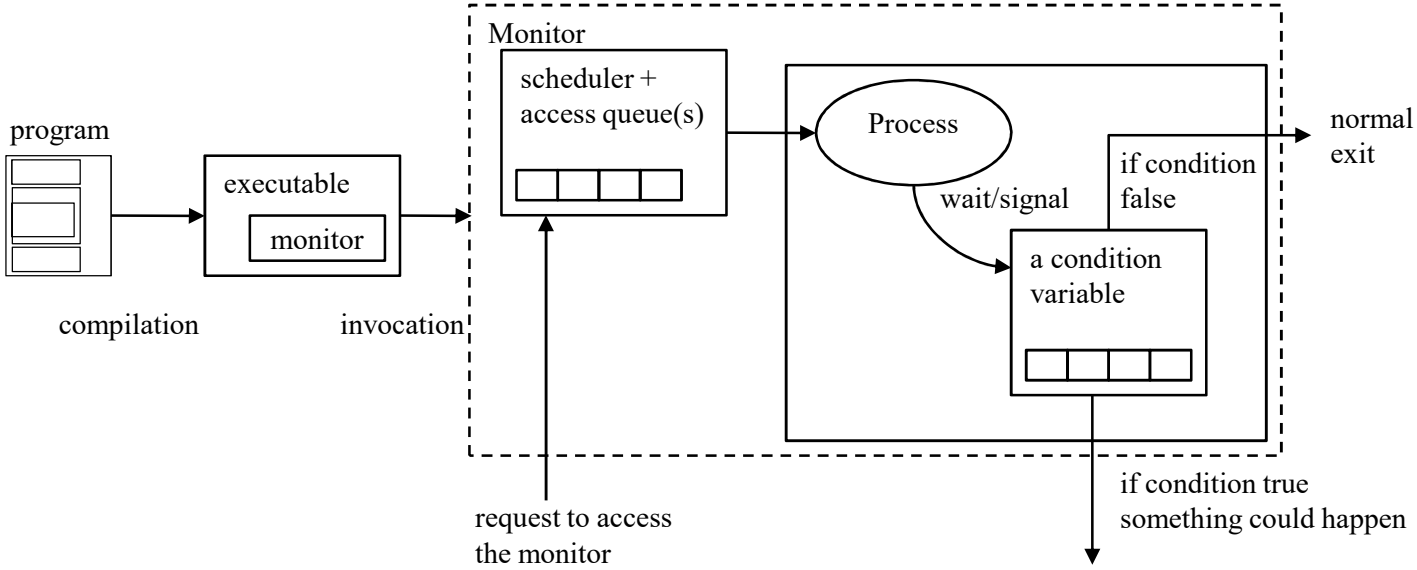


Solving the multiple Producer/Consumer problem “monitor” (2)

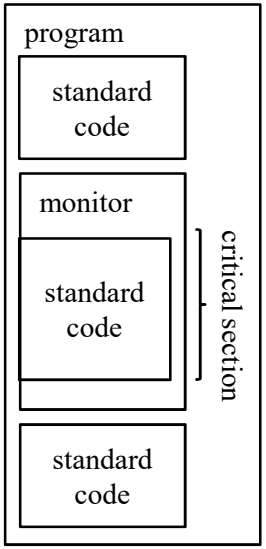


A **monitor** is a special piece of code, associated to condition variables, that are providing mutual exclusion within the monitor. Special rules are applied to the scheduler and memory:

1. only one process at a time can access the monitor,
2. irregular in/out of monitor by processes are controlled with two operations, **wait** and **signal**, to be applied on the condition variables that are close to semaphore mechanisms,
3. monitors are given in two implementations, Mesa and Hoare.



Solving the multiple Producer/Consumer problem “monitor” (3)

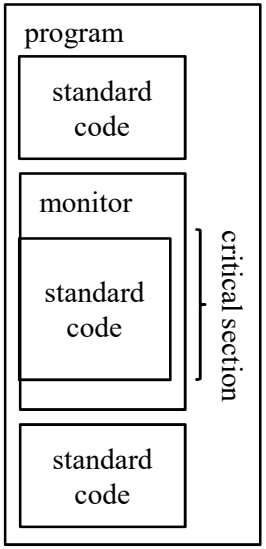


A **monitor** is a special piece of code, associated to condition variables, that are providing mutual exclusion within the monitor. Special rules are applied to the scheduler and memory:

1. only one process at a time can access the monitor,
2. irregular in/out of monitor by processes are controlled with two operations, **wait** and **signal**, to be applied on the condition variables that are close to semaphore mechanisms,
3. monitors are given in two implementations, Mesa and Hoare.

	Mesa	**Hoare**
wait	common implementation	
signal	specific to Mesa, also called notify	specific to Hoare

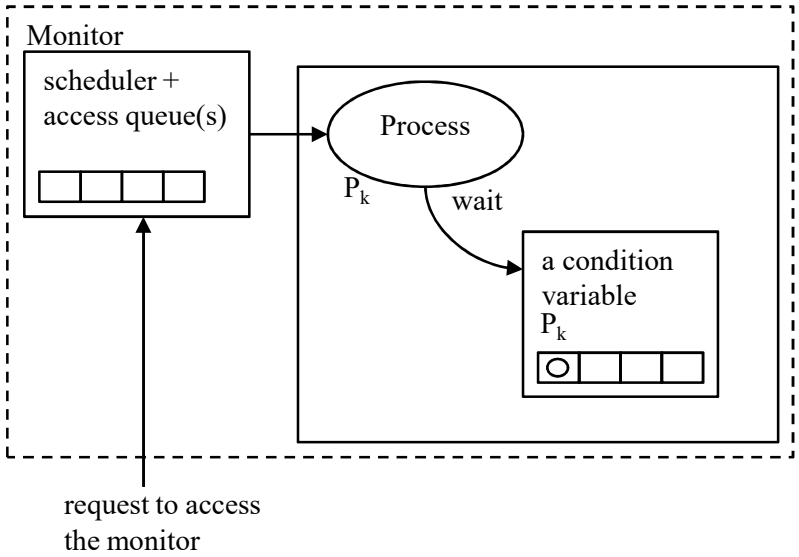
Solving the multiple Producer/Consumer problem “monitor” (4)



A **monitor** is a special piece of code, associated to condition variables, that are providing mutual exclusion within the monitor. Special rules are applied to the scheduler and memory:

1. only one process at a time can access the monitor,
2. irregular in/out of monitor by processes are controlled with two operations, **wait** and **signal**, to be applied on the condition variables that are close to semaphore mechanisms,
3. monitors are given in two implementations, Mesa and Hoare.

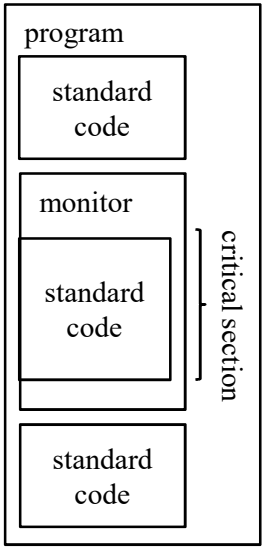
The wait operation



After a wait operation, a process moves to the queue of the condition variable.

	in all the cases
before the wait	P_k in the monitor
after the wait	P_k in the condition queue

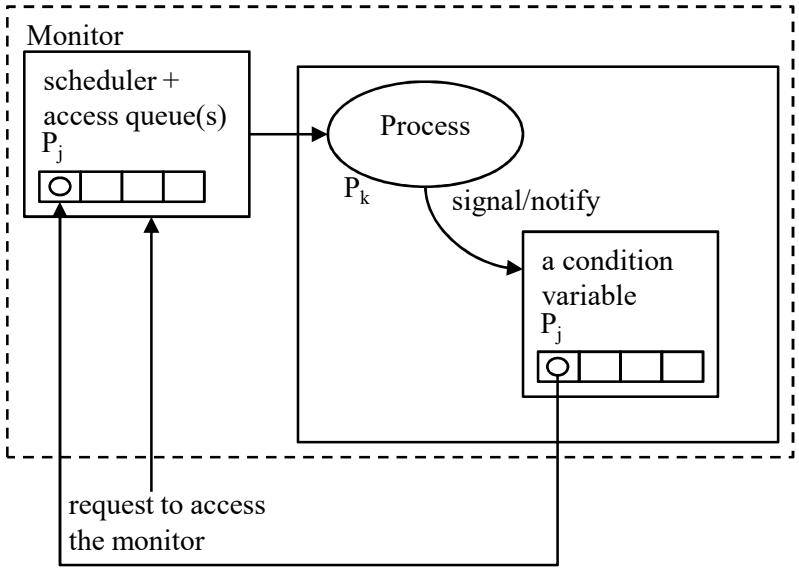
Solving the multiple Producer/Consumer problem “monitor” (5)



A **monitor** is a special piece of code, associated to condition variables, that are providing mutual exclusion within the monitor. Special rules are applied to the scheduler and memory:

1. only one process at a time can access the monitor,
2. irregular in/out of monitor by processes are controlled with two operations, **wait** and **signal**, to be applied on the condition variables that are close to semaphore mechanisms,
3. monitors are given in two implementations, Mesa and Hoare.

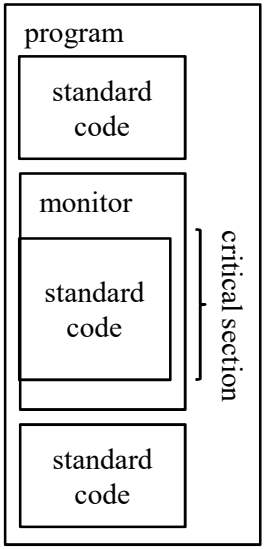
The signal operation with a Mesa implementation, also called notify



If at least one process is in the condition queue, it is notified but the signaling process continues. The signaled process will be resumed at some convenient future time, when the monitor will be available.

	if queue empty	**otherwise**
before the signal	P_k in the monitor, normal exit	P_k in the monitor, P_j in the condition queue
after the signal		P_k in the monitor, P_j in the entry queue

Solving the multiple Producer/Consumer problem “monitor” (6)



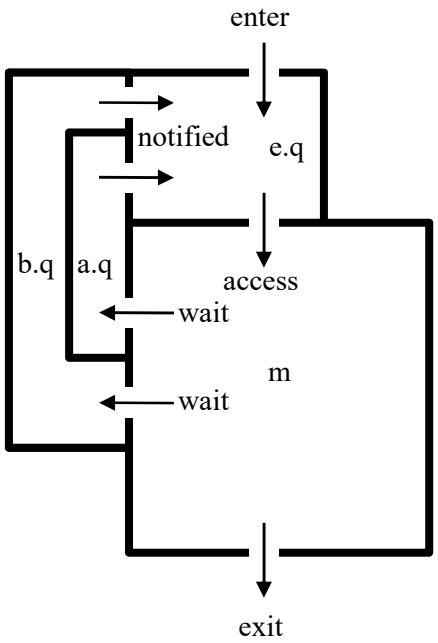
A **monitor** is a special piece of code, associated to condition variables, that are providing mutual exclusion within the monitor. Special rules are applied to the scheduler and memory:

1. only one process at a time can access the monitor,
2. irregular in/out of monitor by processes are controlled with two operations, **wait** and **signal**, to be applied on the condition variables that are close to semaphore mechanisms,
3. monitors are given in two implementations, Mesa and Hoare.

The Buhr’s representation of the Mesa monitor

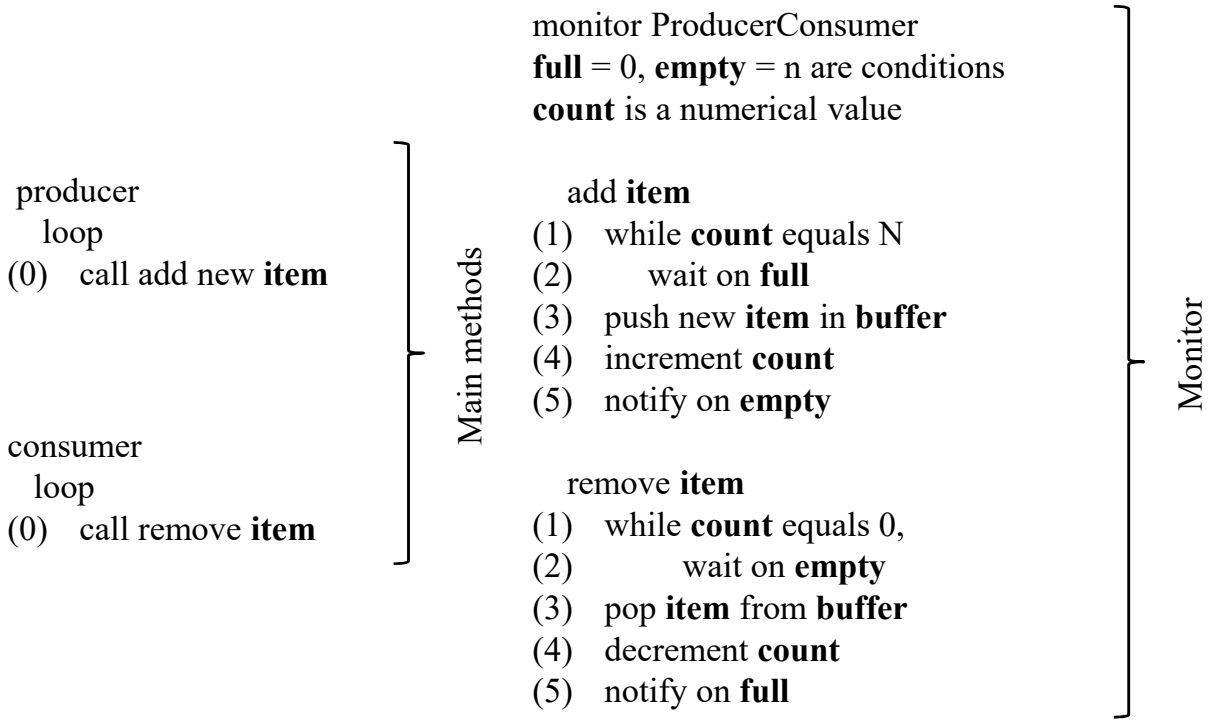
Notation

a.q, b.q	are the queues of the condition variables a, b
e.q	queue of processes that want to enter
m	the monitor with one process at a time
enter	when a process requests the monitor
access	when a process gets the monitor
exit	when a process exits the monitor
wait	when a process moves after a wait operation
notified	when a process leaves the queue of a condition variable following a notify operation



Solving the multiple Producer/Consumer problem "monitor" (7)

The bounded-buffer algorithm with multiple consumers and producers, using a Mesa monitor.



Solving the multiple Producer/Consumer problem "monitor" (8)

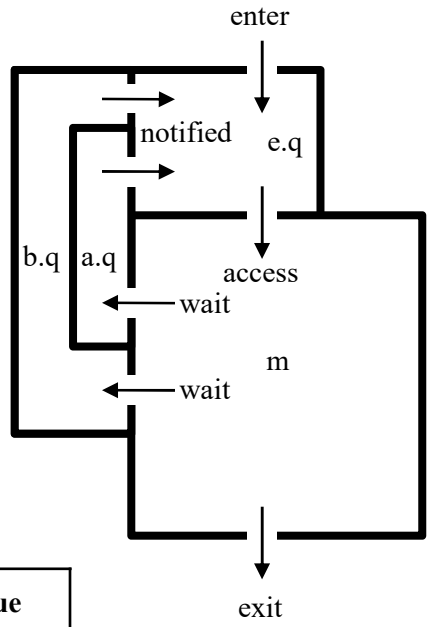
producer
loop
(0) call add new **item**
consumer
loop
(0) call remove **item**

monitor ProducerConsumer
full = 0, **empty** = n are conditions
count is a numerical value

add **item**
(1) while **count** equals N
(2) wait on **full**
(3) push new **item** in **buffer**
(4) increment **count**
(5) notify on **empty**

remove **item**
(1) while **count** equals 0,
(2) wait on **empty**
(3) pop **item** from **buffer**
(4) decrement **count**
(5) notify on **full**

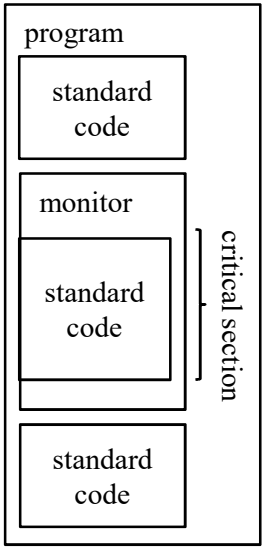
Solve the following problem:
- 3 producers (P1,P2,P3) and 2 consumers (C1,C2).
- max size N of the buffer is 2.
- at t=0, buffer is empty and C1 is in the empty queue.
- scheduling of the entry queue is FCFS.
- schedule considering the following sequence with a Mesa monitor.



	buffer	count	Conditions		Section	entry queue →	
			full	empty			
	0	0	∅	C1	∅	∅	
P1→0,1,3,4	1	1	∅	C1	P1	P1-∅	P1 enters and accesses
C2→0	1	1	∅	C1	P1	C2	C2 enters
P1→5,0	1	1	∅	∅	P1-∅	P1,C1,C2	C1 pushed in e.q, P1 enters
P3→0	1	1	∅	∅	∅	P3,P1,C1,C2	P3 enters
P2→0	1	1	∅	∅	∅	P2,P3,P1,C1,C2	P2 enters
C2→1,3,4,5,0	0	0	∅	∅	C2-∅	C2,P2,P3,P1,C1	C2 accesses and enters
C1→1,2	0	0	∅	C1	C1-∅	C2,P2,P3,P1	C1 restarts and blocks on (2)
P1→1,3,4,5,0	1	1	∅	∅	P1-∅	P1,C1,C2,P2,P3	C1 pushed in e.q, P1 enters
P3→1,3,4,5,0	2	2	∅	∅	P3-∅	P3,P1,C1,C2,P2	P3 accesses and enters
P2→1,2	2	2	P2	∅	P2-∅	P3,P1,C1,C2	P2 blocked on (2)
C2→1,3,4,5,0	1	1	∅	∅	C2-∅	C2,P2,P3,P1,C1	P2 pushed in e.q, C2 enters

P→x,y process P executes the instructions x,y

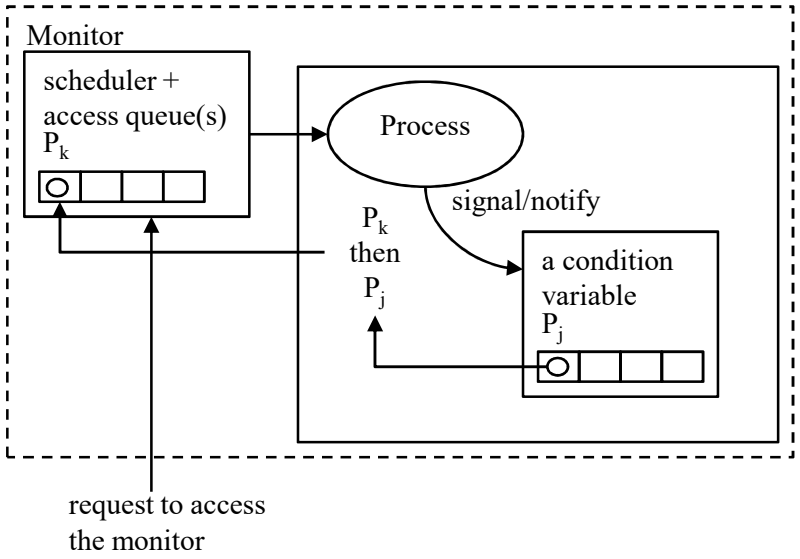
Solving the multiple Producer/Consumer problem “monitor” (9)



A **monitor** is a special piece of code, associated to condition variables, that are providing mutual exclusion within the monitor. Special rules are applied to the scheduler and memory:

1. only one process at a time can access the monitor,
2. irregular in/out of monitor by processes are controlled with two operations, **wait** and **signal**, to be applied on the condition variables that are close to semaphore mechanisms,
3. monitors are given in two implementations, Mesa and Hoare.

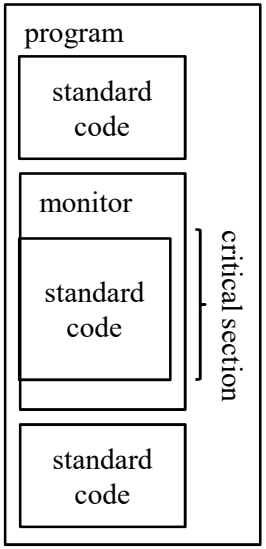
The signal operation with a Hoare implementation



If at least one process is in the condition queue, it runs immediately after the signal operation. The signaling process will be pushed in a specific access queue.

	if queue empty	**otherwise**
before the signal	P_k in the monitor, normal exit	P_k in the monitor, P_j in the condition queue
after the signal		P_j in the monitor, P_k moves to a specific access queue called signal

Solving the multiple Producer/Consumer problem “monitor” (10)



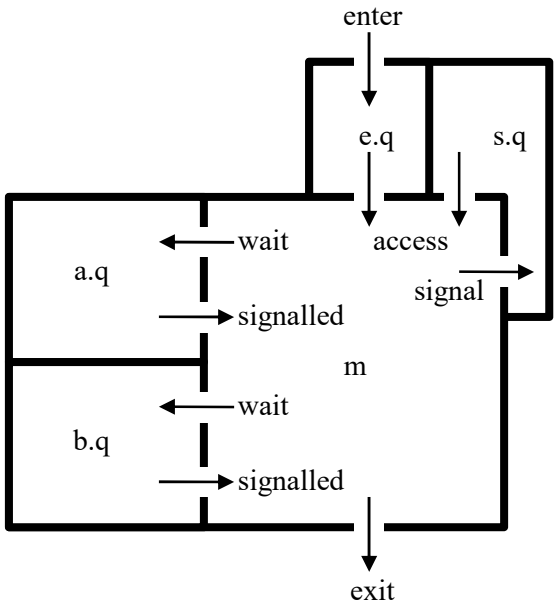
A **monitor** is a special piece of code, associated to condition variables, that are providing mutual exclusion within the monitor. Special rules are applied to the scheduler and memory:

1. only one process at a time can access the monitor,
2. irregular in/out of monitor by processes are controlled with two operations, **wait** and **signal**, to be applied on the condition variables that are close to semaphore mechanisms,
3. monitors are given in two implementations, Mesa and Hoare.

The Buhr’s representation of a Hoare monitor

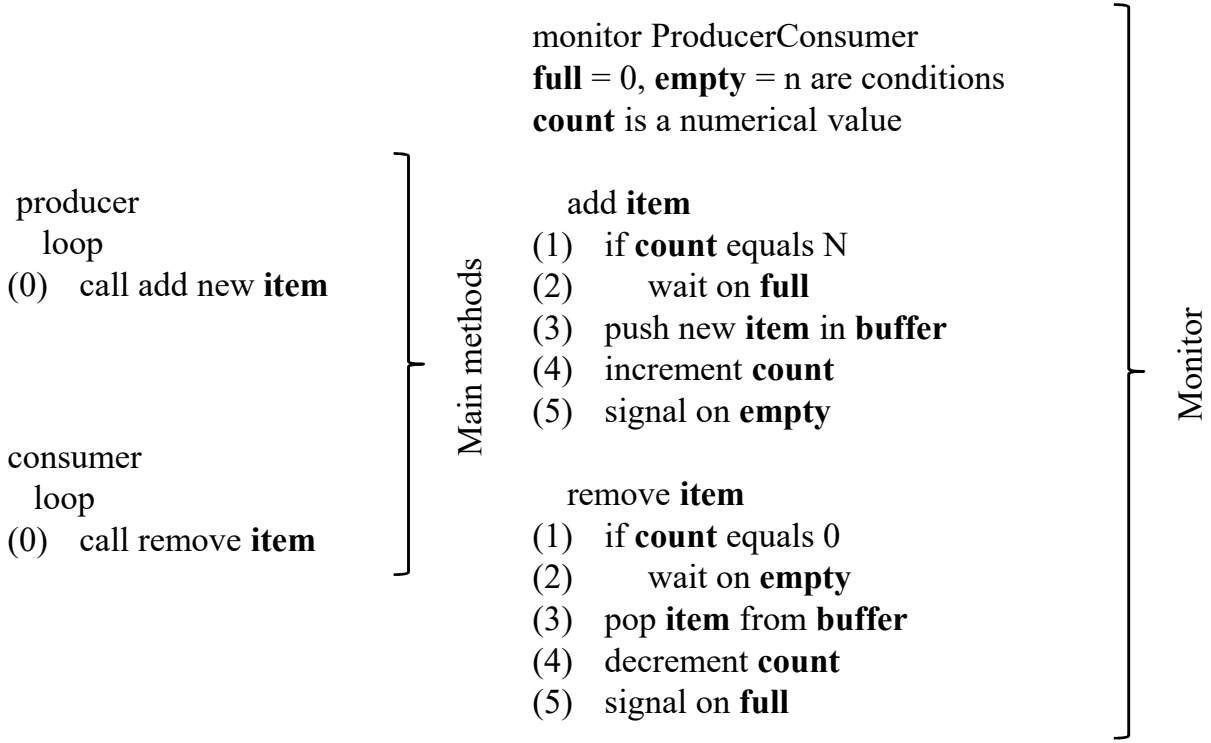
Notation

a.q, b.q	are the queues of the condition variables a, b
e.q	queue of processes that want to enter
s.q	queue of processes that have been pushed out after a signal operation
m	the monitor with one process at a time
enter	when a process requests the monitor
access	when a process gets the monitor
exit	when a process exits the monitor
wait	when a process moves after a wait operation
signalled	when a process leaves the queue of a condition variable following a signal operation
signal	when a process moves out after a successful signal operation



Solving the multiple Producer/Consumer problem “monitor” (11)

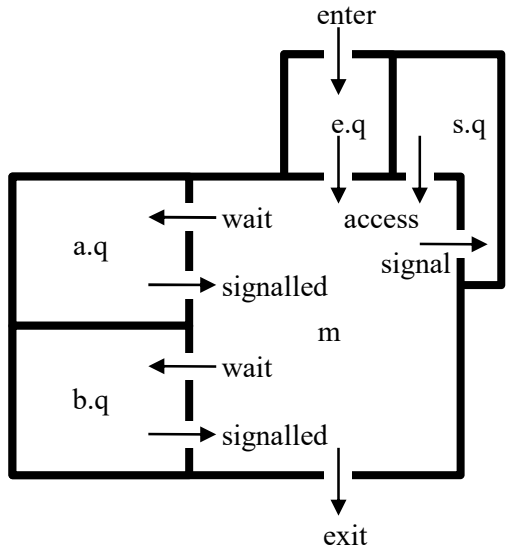
The bounded-buffer algorithm with multiple consumers and producers, using a Hoare monitor.



Solving the multiple Producer/Consumer problem "monitor" (12)

```
producer
loop
(0)   call add new item

consumer
  loop
(0)   call remove item

monitor ProducerConsumer
full = 0, empty = n are conditions
count is a numerical value

 add item
(1)   if count equals N
(2)       wait on full
(3)   push new item in buffer
(4)   increment count
(5)   signal on empty

   remove item
(1)   if count equals 0
(2)       wait on empty
(3)   pop item from buffer
(4)   decrement count
(5)   signal on full
```

Extend the previous problem with an Hoare monitor:
- Scheduling between the (E)ntry and the (S)ignal queues is a Round Robin with a time slice 3/4 (E) and 1/4 (S). At the turn 1, the time slice starts with (E).



	buffer	count	Conditions			Section	entry queue →	Turn	
			full	empty	signal				
	0	0	∅	C1	∅	∅	∅	∅	
P1→0,1,3,4	1	1	∅	C1	∅	P1	∅	1 (E)	P1 enters/accesses
C2→0	1	1	∅	C1	∅	P1	C2	1 (E)	C2 enters
P1→5	1	1	∅	∅	P1	P1-C1	C2	1 (E)	P1 blocked on (5)
C1→3,4,5,0	0	0	∅	∅	P1	C1-∅	C1,C2	Signalled	C1 signalled on (3)
P3→0	0	0	∅	∅	P1	∅	P3,C1,C2	∅	P3 enters
P2→0	0	0	∅	∅	P1	∅	P2,P3,C1,C2	∅	P2 enters
C2→1,2	0	0	∅	C2	P1	C2-∅	P2,P3,C1	2 (E)	C2 blocked on (2)
C1→1,2	0	0	∅	C1,C2	P1	C1-∅	P2,P3	3 (E)	C1 blocked on (2)
P1→1,3,4,5	1	1	∅	C1	P1-P1	P1-C2	P2,P3	4 (S)	P1 loops on s.q
C2→3,4,5,0	0	0	∅	C1	P1	C2-∅	C2,P2,P3	Signalled	C2 signalled on (3)
P3→1,3,4,5	1	1	∅	∅	P3,P1	P3-C1	C2,P2	1 (E)	P3 blocked on (5)

P→x,y process P executes the instructions x,y