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

Mathieu Delalandre
Operating Systems
“IPC and synchronization”

1. Introduction
2. Synchronization for mutual exclusion
   2.1. Principles of concurrency
   2.2. Synchronization for mutual exclusion
3. Synchronization for coordination
   3.1. Some problems of coordination
   3.2. Solving to the Producer/Consumer problem
   3.3. Solving to the multiple Producer/Consumer problem
Cooperating/independent process: A process is cooperating if it can affect (or be affected) by the other processes. Clearly, any process that 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**: With parallelism, if we are interested to run faster a particular task, we must break it into sub-tasks.
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>
1. Introduction
2. Synchronization for mutual exclusion
   2.1. Principles of concurrency
   2.2. Synchronization for mutual exclusion
3. Synchronization for coordination
   3.1. Some problems of coordination
   3.2. Solving to the Producer/Consumer problem
   3.3. Solving to 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 (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 no sharable resource by a process. This is the synchronization problem for mutual exclusion, between processes (2 or n).
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

Race conditions define Critical section solves Mutual exclusion

Synchronisation

IPC raises

Critical section defines

Mutual exclusion solved by

Spooling directory

Notations

<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>

(1) in = 4

(2) out = 3

(3) print

(4) D.out=out

(5) D.name=S[D.out]

(6) out = D.out+1

(7) P.in=in

S[P.in] = P.name

in = P.in+1

Resource acquisition

Considered as

Critical section

IPC raises

Mutual exclusion solves

Synchronisation
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>Ø</td>
<td>Ø</td>
<td>Ø</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>Ø</td>
<td>7</td>
<td>Ø</td>
<td>B.name</td>
</tr>
<tr>
<td>A→2,3</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>Ø</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>Ø</td>
<td>8</td>
<td>7</td>
<td>A.name</td>
</tr>
</tbody>
</table>

P→x  process P executes instruction x

### Notations
- **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

### 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

### IPC
- race conditions
- critical section
- mutual exclusion
- synchronization
- resource acquisition
- mutual exclusion

### Resource acquisition
- mutual exclusion
- critical section
- synchronization
**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 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 “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>shareable</td>
<td>Can be used in parallel by several processes</td>
<td>e.g. read only memory</td>
</tr>
<tr>
<td>no shareable</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>
</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>

Mutual exclusion solved by synchronization mechanism
1. Introduction
2. Synchronization for mutual exclusion
   2.1. Principles of concurrency
   2.2. Synchronization for mutual exclusion
3. Synchronization for coordination
   3.1. Some problems of coordination
   3.2. Solving to the Producer/Consumer problem
   3.3. Solving to the multiple Producer/Consumer problem
Synchronization for mutual exclusion
“Introduction” (1)

1. The disabling preemptive scheduling (i.e. interrupts) approach

Process

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
</table>

t

Process

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
</table>

t

disable interrupts “can’t be B”
disable interrupts “can’t be A”

Correspond to the areas of critical sections

2. The “busy wait” or “spin waiting” approach

Process A

1. check

Shared memory

1. check

Process B

2. allow

2. allow

Critical section

3. access

3. access

4. exit

3. The “sleep - wakeup” approach

Process A

1. check

Shared memory

1. check

Process B

2. allow

2. allow

Critical section

3. sleep

3. sleep

4. wakeup

6. exit

5. access

5. access
Synchronization for mutual exclusion
“Introduction” (2)

<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</td>
<td>hardware</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>interrupts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swap, TSL, CAS</td>
<td>busy wait</td>
<td>software</td>
<td>possible</td>
</tr>
<tr>
<td>Perterson’s algorithm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary semaphore / mutex</td>
<td>sleep</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>wakeup</td>
<td></td>
<td></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 scheduling performance could be noticeably degraded (e.g. a C process, not interested with the section, can be blocked while A accesses the section).
✓ This approach cannot work in a multi-processor architecture.
Synchronization methods 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>possible</td>
</tr>
<tr>
<td>Perterson’s algorithm</td>
<td>sleep wakeup</td>
<td>software</td>
<td>no</td>
</tr>
<tr>
<td>binary semaphore / mutex</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Synchronization methods for mutual exclusion
“Swap, TSL and CAS” (1)

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

\[ \text{SWAP KEY,LOCK} \]

<table>
<thead>
<tr>
<th>KEY</th>
<th>LOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

“access case”
LOCK at 0, KEY at 1 both shift their values

```
A→1,2,3
B→1,2,3,4,3,4,3
C→1,2,3,4,3,4
A→4,3
B→4,5,6
C→3,4
```

“busy case”
LOCK and KEY at 1 both keep their values

```
B accesses the section
A is blocked
B releases the section
A can access
A releases the section
C can access
C releases the section
```

\[ \text{KEYA} \quad \text{KEYB} \quad \text{KEYC} \quad \text{LOCK} \quad \text{by} \]

<table>
<thead>
<tr>
<th>KEY</th>
<th>KEY</th>
<th>KEY</th>
<th>LOCK</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$0$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

(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 ….

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

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

**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>
<tr>
<td>RX</td>
<td>LOCK</td>
</tr>
<tr>
<td>Na</td>
<td>$1</td>
</tr>
<tr>
<td>TSL RX, LOCK</td>
<td>$1</td>
</tr>
</tbody>
</table>

#### "access case - Lock at 0"
- RX set to 0
- LOCK moves to 1

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

CAS LOCK, TEST, KEY

(1) copy
R → LOCK  TEST  KEY

atomic instruction

(2) set LOCK with KEY if LOCK and TEST are equal

R ← CAS LOCK, 0, 1

“access case” KEY at 1 and TEST, LOCK at 0
LOCK is updated

R ← CAS LOCK, 0, 1

“busy case” LOCK and TEST different nothing happens

(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 ….

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

Request

Release

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

<table>
<thead>
<tr>
<th></th>
<th>RA</th>
<th>RB</th>
<th>RC</th>
<th>LOCK</th>
<th>by</th>
</tr>
</thead>
<tbody>
<tr>
<td>B→1,2</td>
<td>∅</td>
<td>0</td>
<td>∅</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>A→1,2,3,2,3,2</td>
<td>1</td>
<td>0</td>
<td>∅</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>B→3,4,5</td>
<td>1</td>
<td>0</td>
<td>∅</td>
<td>0</td>
<td>∅</td>
</tr>
<tr>
<td>A→3,2</td>
<td>0</td>
<td>0</td>
<td>∅</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>C→1,2,3,2,3</td>
<td>0</td>
<td>0</td>
<td>1</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>0</td>
<td>∅</td>
</tr>
<tr>
<td>C→2,3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>C→4,5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>∅</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 methods 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>Peterson’s algorithm</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

“Peterson’s algorithm” (1)

Peterson’s algorithm deals with coordination between processes. Entrance in the critical section is granted for a process \( P \) if the others do not want to enter their critical sections, or if they have given previously the priority to \( P \).

- \( \text{ps} = \) processes, \( \text{turn} = \emptyset \) are sets of processes
- \( \text{flag} \) is a table of \( \text{ps} \) size

**Request**
- Request the critical section with \( p \)
  - \( \text{flag} \) at \( p \) is true
  - \( \text{turn} \) equals \( \text{ps} \) without \( p \)
  - while a \( \text{flag} \) at \( \text{turn} \) is true and \( p \) out of \( \text{turn} \)

  \( (1) \)

  busy wait

  \( (2) \)

- Run in the critical section with \( p \)
  - do something ….

**Release**
- Release the critical section with \( p \)
  - \( \text{flag} \) at \( p \) is false

**Example**
- with three processes \( P_i, P_j \) and \( P_k \)

<table>
<thead>
<tr>
<th>( P_i ) waits if</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
</tr>
<tr>
<td>(1) ( P_{j,k} ) set their flags at true</td>
</tr>
<tr>
<td>(2) ( P_{j,k} ) don’t set turn at ( P_i )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(1)</th>
<th>(2)</th>
<th>(1) &amp; (2)</th>
<th>while</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>stop</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>stop</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>stop</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>continue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( P_i ) accesses the critical section if</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
</tr>
<tr>
<td>(1) ( P_{j,k} ) set their flags at false</td>
</tr>
<tr>
<td>(2) ( P_{j,k} ) set turn at ( P_i )</td>
</tr>
</tbody>
</table>
Synchronization methods for mutual exclusion

“Peterson’s algorithm” (2)

Request

(1) Request the critical region with p
(2) flag at p is true
(3) turn equals ps without p
(4) while a flag at turn is true and p out of turn
(5) busy wait

Release

(6) Release the critical section with p
(7) flag at p is false

P_i accesses the critical section if

| or               | (1) P_j,k set their flags at false
|                  | (2) P_j,k set turn at P_i

P_i accesses the critical section if

<table>
<thead>
<tr>
<th>turn</th>
<th>flag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>∅</td>
<td>false</td>
</tr>
<tr>
<td>B→1,2</td>
<td>∅</td>
</tr>
<tr>
<td>A→1,2,3,4,5,4,5</td>
<td>B</td>
</tr>
<tr>
<td>B→3</td>
<td>A</td>
</tr>
<tr>
<td>A→4,6,7</td>
<td>A</td>
</tr>
<tr>
<td>B→4,6,7</td>
<td>A</td>
</tr>
</tbody>
</table>

B sets its flag at true
A is blocked because the flag of B is true and A is out of turn
B sets the turn variable to A
A is unblocked because the turn variable is set to A
B is unblocked because the flag of A is false
Synchronization methods for mutual exclusion
“Peterson’s algorithm” (3)

Request

(1) Request the critical region with \( p \)
(2) \textit{flag} at \( p \) is true
(3) \textit{turn} equals \( p_{s} \) without \( p \)
(4) while a \textit{flag} at \textit{turn} is true and \( p \) out of \textit{turn}
    busy wait
(5) \( P_{i} \) accesses the critical section if

\| \text{turn} \| \text{flag} \| \text{A} \| \text{B} \| \text{C} \\
\hline
\emptyset & \text{false} & \text{false} & \text{false} \\
B \rightarrow 1,2 & \emptyset & \text{false} & \text{true} & \text{false} \\
A \rightarrow 1,2,3 & B,C & \text{true} & \text{true} & \text{false} \\
C \rightarrow 1,2,3 & A,B & \text{true} & \text{true} & \text{true} \\
B \rightarrow 3 & A,C & \text{true} & \text{true} & \text{true} \\
A \rightarrow 4,6 & A,C & \text{true} & \text{true} & \text{true} \\
C \rightarrow 4,6 & A,C & \text{true} & \text{true} & \text{true} \\

or

(1) \( P_{j,k} \) set their flags at false
(2) \( P_{j,k} \) set turn at \( P_{i} \)

(6) Release the critical section with \( p \)
(7) \textit{flag} at \( p \) is false

B sets its flag to true
A sets its flag to true and turn with the other processes
C sets its flag to true and turn with the other processes
B sets turn with the other processes
A, C access the critical section at the same time, the “root” version of the Peterson algorithm does not respect mutual exclusion with \( n > 2 \) processes
Synchronization methods 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>Perterson’s algorithm</td>
<td>sleep wakeup</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>binary semaphore / mutex</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Synchronization methods for mutual exclusion
“binary semaphores / mutex” (1)

Semaphore is a synchronization primitive composed of a blocking queue/stack and a variable controlled with 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 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>

“normal” 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>

“blocked” down

Main memory

CPU

dispatcher

short-term scheduler

running process

if true sleep p_k, and push p_k in the stack

Semaphore

value

semaphore

P_k

ready queue

Main memory

P_k

value

P_k

stack

P

P

before after

before after
Synchronization methods for mutual exclusion  
“binary semaphores / mutex” (2)

**Semaphore** is a synchronization primitive composed of a blocking queue/stack and a variable controlled with 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 it must be the one that unlocks it.

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

<table>
<thead>
<tr>
<th>“normal” up</th>
<th>“unblocked” up</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>before</strong></td>
<td><strong>after</strong></td>
</tr>
<tr>
<td>value</td>
<td>true</td>
</tr>
<tr>
<td>stack</td>
<td>∅</td>
</tr>
</tbody>
</table>

![Diagram of semaphore operations and state transitions](image)

- **up** operation:
  - If the semaphore is not empty (value is false), it pops the process from the stack and wakes it up.
  - If the stack is empty, it checks the semaphore’s value.

- **semaphore** state transitions:
  - If true, wakes up a process and pops it from the stack.
  - If false, checks if the semaphore is empty.

- **dispatcher** and **short-term scheduler** manage the process in the stack.

- **running process** is handled by the **dispatcher** and **short-term scheduler**.

- **Main memory** holds the processes in the stack.

- **CPU** executes the running process.
Synchronization methods for mutual exclusion
“binary semaphores / mutex” (3)

The algorithm for mutual exclusion using a binary semaphore is

sem is a semaphore

Request
(1) Request the critical section with p
(2) down sem
(3) Run in the critical section with p
do something ….

Release
(4) Release the critical section with p
(5) up sem

<table>
<thead>
<tr>
<th>Request</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem</td>
<td>sem</td>
</tr>
<tr>
<td>value</td>
<td>by</td>
</tr>
<tr>
<td>false</td>
<td>∅</td>
</tr>
<tr>
<td>true</td>
<td>A</td>
</tr>
<tr>
<td>C→1,2</td>
<td>true</td>
</tr>
<tr>
<td>A→4,5</td>
<td>true</td>
</tr>
<tr>
<td>B→3,4,5</td>
<td>true</td>
</tr>
<tr>
<td>C→3,4,5</td>
<td>false</td>
</tr>
</tbody>
</table>

A accesses the section, sem becomes true
while accessing the semaphore, B blocks
while accessing the semaphore, C 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

“normal” down

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

“blocked” down

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

“normal” up

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

“unblocked” up

<table>
<thead>
<tr>
<th>value</th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>∣</td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td>P</td>
<td>∅</td>
</tr>
</tbody>
</table>
Operating Systems
“IPC and synchronization”

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

The *dinning-philosophers problem* is summarized as,

1. five philosophers sitting at a table doing one of 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 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 ones can read,
Some problems of coordination (2)

The producer-consumer (i.e. **bounded buffer**) describes how processes share a common, fixed-size 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 from an empty buffer. 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 synchronization with mutual exclusion when accessing the buffer,
Operating Systems
“IPC and synchronization”

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

<table>
<thead>
<tr>
<th>Methods</th>
<th>Approach</th>
<th>Type</th>
<th>Application problem</th>
<th>Coordination type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sleep-wakeup</td>
<td>sleep wakeup</td>
<td>Software</td>
<td>Producer / Consumer</td>
<td>coordination by communication</td>
</tr>
<tr>
<td>semaphore</td>
<td>sleep wakeup</td>
<td>Software</td>
<td>Multiple Producers / Consumers</td>
<td>coordination by sharing</td>
</tr>
<tr>
<td>semaphore / mutex</td>
<td>sleep wakeup</td>
<td>Software</td>
<td></td>
<td></td>
</tr>
<tr>
<td>monitor</td>
<td>sleep wakeup</td>
<td>Software</td>
<td></td>
<td></td>
</tr>
</tbody>
</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

**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 successful synchronization**

<table>
<thead>
<tr>
<th>Buffer</th>
<th>P state</th>
<th>C state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ready</td>
<td>blocked</td>
</tr>
<tr>
<td>P→1,3,4,5</td>
<td>1</td>
<td>ready</td>
</tr>
<tr>
<td>C→3,4</td>
<td>0</td>
<td>ready</td>
</tr>
<tr>
<td>P→1,3,4,5</td>
<td>1</td>
<td>ready</td>
</tr>
<tr>
<td>C→1,3,4,1,2</td>
<td>0</td>
<td>ready</td>
</tr>
</tbody>
</table>

P wakes up C
C restarts at PC*
here is a lost wakeup
when empty C will sleep

* is Program Counter
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 failure synchronization

<table>
<thead>
<tr>
<th>Buffer</th>
<th>P state</th>
<th>C state</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>P→1,3,4,5</td>
<td>1 ready blocked</td>
</tr>
<tr>
<td></td>
<td>P→1,3,4</td>
<td>2 ready blocked</td>
</tr>
<tr>
<td></td>
<td>P→1,3,4</td>
<td>3 ready blocked</td>
</tr>
<tr>
<td></td>
<td>… … …</td>
<td>… … …</td>
</tr>
<tr>
<td></td>
<td>P→1,2</td>
<td>n blocked blocked</td>
</tr>
</tbody>
</table>

here is a lost wakeup
C blocked on sleep

fill in buffer
P, C will sleep for always
Solving the Producer/Consumer problem
“sleep wakeup” (3)

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

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 successful synchronization

<table>
<thead>
<tr>
<th>command</th>
<th>wakeup waiting bit</th>
<th>process state</th>
</tr>
</thead>
<tbody>
<tr>
<td>sleep</td>
<td>0</td>
<td>sleep put to 0</td>
</tr>
<tr>
<td>wakeup</td>
<td>1</td>
<td>put to 1 wakeup</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>buffer</th>
<th>ww bits</th>
<th>P state</th>
<th>C state</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>C</td>
<td>P</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>ready</td>
<td>ready</td>
</tr>
<tr>
<td>C→1</td>
<td>0</td>
<td>0</td>
<td>ready</td>
</tr>
<tr>
<td>P→1,3,4,5</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C→2</td>
<td>1</td>
<td>0</td>
<td>ready</td>
</tr>
<tr>
<td>P→1,3,4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C→3,4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.......</td>
<td>.......</td>
<td>.......</td>
<td>.......</td>
</tr>
</tbody>
</table>

C gets a bit
C uses the bit
the synchronization will go on
Solving the Producer/Consumer problem

<table>
<thead>
<tr>
<th>Methods</th>
<th>Approach</th>
<th>Type</th>
<th>Application problem</th>
<th>Coordination type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sleep-wakeup</td>
<td>sleep wakes</td>
<td>Software</td>
<td>Producer / Consumer</td>
<td>coordination by communication</td>
</tr>
<tr>
<td>semaphore</td>
<td>sleep wakes</td>
<td>Software</td>
<td>Multiple Producers / Consumers</td>
<td>coordination by sharing</td>
</tr>
<tr>
<td>semaphore / mutex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>monitor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Solving the Producer/Consumer problem
“semaphores” (1)

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

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

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
</tr>
<tr>
<td>stack</td>
<td>∅</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
</tr>
<tr>
<td>stack</td>
<td>∅</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
</tr>
<tr>
<td>stack</td>
<td>∅</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
</tr>
<tr>
<td>stack</td>
<td>P</td>
</tr>
</tbody>
</table>
Solving the Producer/Consumer problem
“semaphores” (2)

The algorithm for solving the producer/consumer problem with semaphore 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 semaphore 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

<table>
<thead>
<tr>
<th>buffer value</th>
<th>fill value</th>
<th>empty value</th>
<th>P state</th>
<th>C state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>∅</td>
<td>2</td>
<td>∅</td>
</tr>
<tr>
<td>C→1</td>
<td>0</td>
<td>0</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>P→1,2,3</td>
<td>1</td>
<td>0</td>
<td>∅</td>
<td>1</td>
</tr>
<tr>
<td>C→2,3</td>
<td>0</td>
<td>0</td>
<td>∅</td>
<td>2</td>
</tr>
<tr>
<td>P→1,2,3</td>
<td>1</td>
<td>1</td>
<td>∅</td>
<td>1</td>
</tr>
<tr>
<td>P→1,2,3</td>
<td>2</td>
<td>2</td>
<td>∅</td>
<td>0</td>
</tr>
<tr>
<td>P→1</td>
<td>2</td>
<td>2</td>
<td>∅</td>
<td>0</td>
</tr>
</tbody>
</table>

- C sleeps at down on fill
- C wakes up at up on fill
- next scheduling, C restarts on pop
- fill in buffer
- P stopped at down on empty

### “normal” down

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
</tr>
<tr>
<td>stack</td>
<td>∅</td>
</tr>
</tbody>
</table>

### “blocked” down

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
</tr>
<tr>
<td>stack</td>
<td>P</td>
</tr>
</tbody>
</table>

### “normal” up

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
</tr>
<tr>
<td>stack</td>
<td>∅</td>
</tr>
</tbody>
</table>

### “unblocked” up

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
</tr>
<tr>
<td>stack</td>
<td>P</td>
</tr>
</tbody>
</table>
Operating Systems
“IPC and synchronization”

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

<table>
<thead>
<tr>
<th>Methods</th>
<th>Approach</th>
<th>Type</th>
<th>Application problem</th>
<th>Coordination type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sleep-wakeup</td>
<td>sleep wakeup</td>
<td>Software</td>
<td>Producer / Consumer</td>
<td>coordination by communication</td>
</tr>
<tr>
<td>semaphore</td>
<td>sleep wakeup</td>
<td>Software</td>
<td>Multiple Producers / Consumers</td>
<td>coordination by sharing</td>
</tr>
<tr>
<td>semaphore / mutex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>monitor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Solving to the multiple Producer/Consumer problem
“semaphores - mutex” (1)

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

fill

empty

buffer

0

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 are then not atomic operations.

pop
(1) \( w = b[\text{out}] \)
(2) \( \text{out} = (\text{out}+1) \mod n \)

push
(1) \( b[\text{in}] = v \)
(2) \( \text{in} = (\text{in}+1) \mod n \)

In addition, the buffer slots are data-dependent (e.g. byte, double, data structure, etc.), the (1) instruction could be a loop.

The “one-to-one solution” to the bounded buffer problem with multiple producers and/or consumers, becomes

\[
\begin{align*}
\text{fill} &= 0, \quad \text{empty} = n \text{ are semaphores} \\
\text{buffer} \text{ is the data structure}
\end{align*}
\]

consumer

loop
(1) down fill
(2) \( w = b[\text{out}] \)
(3) \( \text{out} = (\text{out}+1) \mod n \)
(4) up empty

producer

loop
(1) down empty
(2) \( b[\text{in}] = v \)
(3) \( \text{in} = (\text{in}+1) \mod n \)
(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, considering the no atomic access to the buffer is

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

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

P1, P2 update the In value, a null slot b[1] remains
C accesses a null slot b[1] and crashes the system (i.e. exception)
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**

We protect access to buffer with a mutex.
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

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

<table>
<thead>
<tr>
<th>buffer</th>
<th>fill</th>
<th>empty</th>
<th>mutex</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>value</td>
<td>S</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Ø</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>P1→1,2</td>
<td>0</td>
<td>Ø</td>
<td>3</td>
</tr>
<tr>
<td>P2→1,2</td>
<td>0</td>
<td>Ø</td>
<td>2</td>
</tr>
<tr>
<td>P1→3,4,5</td>
<td>1</td>
<td>1</td>
<td>Ø</td>
</tr>
<tr>
<td>P2→3</td>
<td>2</td>
<td>1</td>
<td>Ø</td>
</tr>
<tr>
<td>C→1,2</td>
<td>2</td>
<td>0</td>
<td>Ø</td>
</tr>
<tr>
<td>P2→4,5</td>
<td>2</td>
<td>1</td>
<td>Ø</td>
</tr>
<tr>
<td>C→3,4,5</td>
<td>1</td>
<td>1</td>
<td>Ø</td>
</tr>
</tbody>
</table>

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” (6)

The deadlocking (wrong) algorithm with the inverted code for solving the multiple producer/consumer problem with semaphore is

\[
\text{fill} = 0, \quad \text{empty} = n \quad \text{are semaphores} \\
\text{mutex is a mutex} \\
\text{buffer is the data structure}
\]

- **Consumer loop**
  1. Down **mutex**
  2. Down **fill**
  3. Pop **item** from **buffer**
  4. Up **empty**
  5. Up **mutex**

- **Producer loop**
  1. Down **mutex**
  2. Down **empty**
  3. Push a new **item** in **buffer**
  4. Up **fill**
  5. Up **mutex**

**Example:** one producers P, one consumer C

<table>
<thead>
<tr>
<th></th>
<th>empty</th>
<th>mutex</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>Ø</td>
<td>false</td>
</tr>
<tr>
<td>P→1,2</td>
<td>0</td>
<td>P</td>
</tr>
<tr>
<td>C→1</td>
<td>0</td>
<td>P</td>
</tr>
</tbody>
</table>

P, C will sleep for always
Solving the multiple Producer/Consumer problem

<table>
<thead>
<tr>
<th>Methods</th>
<th>Approach</th>
<th>Type</th>
<th>Application problem</th>
<th>Coordination type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sleep-wakeup</td>
<td>sleep</td>
<td>Software</td>
<td>Producer / Consumer</td>
<td>coordination by communication</td>
</tr>
<tr>
<td>semaphore</td>
<td>sleep</td>
<td>Software</td>
<td>Multiple Producers / Consumers</td>
<td>coordination by sharing</td>
</tr>
<tr>
<td>semaphore / mutex</td>
<td>sleep</td>
<td>Software</td>
<td></td>
<td></td>
</tr>
<tr>
<td>monitor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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.
Solving to 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.

1. Only one process at a time can access the monitor (based on a scheduler and a mutex).
2. Irregular in/out of monitor by processes are controlled with two operations, wait and signal, to be applied on conditions variables close to semaphore mechanisms.
A **monitor** is a special piece of code associated to “condition variables” that are providing mutual exclusion within the monitor.

1. Only one process at a time can access the monitor (based on a scheduler and a mutex).
2. Irregular in/out of monitor by processes are controlled with two operations, **wait** and **signal**, to be applied on conditions variables close to semaphore mechanisms.
3. Monitors are given in two implementations, Mesa and Hoare.

<table>
<thead>
<tr>
<th></th>
<th>Mesa</th>
<th>Hoare</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>wait</strong></td>
<td>common implementation</td>
<td></td>
</tr>
<tr>
<td><strong>signal</strong></td>
<td>specific to Mesa, also called notify</td>
<td>specific to Hoare</td>
</tr>
</tbody>
</table>

program

standard code

monitor

standard code

standard code

critical section
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.

1. Only one process at a time can access the monitor (based on a scheduler and a mutex).
2. Irregular in/out of monitor by processes are controlled with two operations, wait and signal, to be applied on conditions variables close to semaphore mechanisms.
3. Monitors are given in two implementations, Mesa and Hoare.

The wait operation

- Monitor: scheduler + access queue(s)
- Mutex: value
- Process: wait

<table>
<thead>
<tr>
<th>In all the case</th>
<th>before the wait</th>
<th>after the wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) p&lt;sub&gt;k&lt;/sub&gt; in the monitor</td>
<td>(2) p&lt;sub&gt;k&lt;/sub&gt; in the condition queue</td>
<td></td>
</tr>
</tbody>
</table>
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.
1. Only one process at a time can access the monitor (based on a scheduler and a mutex).
2. Irregular in/out of monitor by processes are controlled with two operations, \textbf{wait} and \textbf{signal}, to be applied on conditions variables close to semaphore mechanisms.
3. Monitors are given in two implementations, Mesa and Hoare.

The signal operation with a Mesa implementation, also called notify

<table>
<thead>
<tr>
<th>if queue empty</th>
<th>otherwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>before the signal</td>
<td>after the signal</td>
</tr>
<tr>
<td>$p_k$ in the monitor,</td>
<td>(1) $p_k$ in the monitor, p_j in</td>
</tr>
<tr>
<td>normal exit</td>
<td>the condition queue</td>
</tr>
<tr>
<td>(2) $p_k$ in the monitor,</td>
<td></td>
</tr>
<tr>
<td>p_j in the entry queue</td>
<td>(2) $p_k$ in the monitor, p_j in</td>
</tr>
</tbody>
</table>

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.
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.
1. Only one process at a time can access the monitor (based on a scheduler and a mutex).
2. Irregular in/out of monitor by processes are controlled with two operations, wait and signal, to be applied on conditions variables close to semaphore mechanisms.
3. Monitors are given in two implementations, Mesa and Hoare.

The Buhr’s representation of the Mesa monitor

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.q, b.q</td>
<td>are the queues of the condition variables a,b</td>
</tr>
<tr>
<td>e.q</td>
<td>queue of processes that want to enter</td>
</tr>
<tr>
<td>m</td>
<td>the monitor with one process at a time</td>
</tr>
<tr>
<td>enter</td>
<td>when a process requests the monitor</td>
</tr>
<tr>
<td>access</td>
<td>when a process gets the monitor (i.e. mutex)</td>
</tr>
<tr>
<td>exit</td>
<td>when a process exits the monitor</td>
</tr>
<tr>
<td>wait</td>
<td>when a process moves after a wait operation</td>
</tr>
<tr>
<td>notified</td>
<td>when a process leaves the condition variables’ queues following a notify operation</td>
</tr>
</tbody>
</table>
Solving the multiple Producer/Consumer problem
“monitor” (7)

The bounded buffer algorithm with several consumer(s) and producer(s), using a Mesa monitor, is

```
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
```
Solving the multiple Producer/Consumer problem - “monitor” (8)

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 C1 is in the empty queue
- scheduling of the entry queue is FCFS
- schedule considering the following sequence with a Mesa monitor

<table>
<thead>
<tr>
<th>buffer</th>
<th>count</th>
<th>Conditions</th>
<th>by</th>
<th>entry queue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>full</td>
<td>empty</td>
<td></td>
</tr>
</tbody>
</table>

P1→0,1,3,4
0 0 Ø       C1  Ø     Ø

C2→0
1 1 Ø       C1  P1     C2

P1→5,0
1 1 Ø Ø     P1-Ø     P1,C1,C2

P3→0
1 1 Ø Ø     Ø         Ø

P2→0
1 1 Ø Ø     Ø         P2,P3,P1,C1,C2

C2→1,3,4,5,0
0 0 Ø Ø     C2-Ø     C2,P2,P3,P1,C1

C1→1,2
0 0 Ø Ø     C1-Ø     C2,P2,P3,P1

P1→1,3,4,5,0
1 1 Ø Ø     P1-Ø     P1,C1,C2,P2,P3

P3→1,3,4,5,0
2 2 Ø Ø     P3-Ø     P3,P1,C1,C2,P2

P2→1,2
2 2 P2 Ø     P2-Ø     P3,P1,C1,C2

C2→1,3,4,5,0
1 1 Ø Ø     C2-Ø     C2,P2,P3,P1,C1

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

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

Producer loop
(0) call add new item

Consumer loop
(0) call remove item

Note: The table and diagram provide a visual representation of the state transitions and conditions for the producer/consumer system with the given specifications.
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.
1. Only one process at a time can access the monitor (based on a scheduler and a mutex).
2. Irregular in/out of monitor by processes are controlled with two operations, **wait** and **signal**, to be applied on conditions variables 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.
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.

1. Only one process at a time can access the monitor (based on a scheduler and a mutex).
2. Irregular in/out of monitor by processes are controlled with two operations, **wait** and **signal**, to be applied on conditions variables close to semaphore mechanisms.
3. Monitors are given in two implementations, Mesa and Hoare.

The Buhr’s representation of a Hoare monitor

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.q, b.q</td>
<td>are the queues of the condition variables a,b</td>
</tr>
<tr>
<td>e.q</td>
<td>queue of processes that want to enter</td>
</tr>
<tr>
<td>s.q</td>
<td>queue of processes that have been pushed out after a signal operation</td>
</tr>
<tr>
<td>m</td>
<td>the monitor with one process at a time</td>
</tr>
<tr>
<td>enter</td>
<td>when a process requests the monitor</td>
</tr>
<tr>
<td>access</td>
<td>when a process gets the monitor (i.e. mutex)</td>
</tr>
<tr>
<td>exit</td>
<td>when a process exits the monitor</td>
</tr>
<tr>
<td>wait</td>
<td>when a process moves after a wait operation</td>
</tr>
<tr>
<td>signalled</td>
<td>when a process leaves the condition variables’ queues following a signal operation</td>
</tr>
<tr>
<td>signal</td>
<td>when a process moved out after a successful signal operation</td>
</tr>
</tbody>
</table>
Solving the multiple Producer/Consumer problem - “monitor” (11)

The bounded buffer algorithm with several consumer(s) and producer(s), using a Hoare monitor, is

```
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

Main methods

- 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
```
Solving the multiple Producer/Consumer problem - “monitor” (12)

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

<table>
<thead>
<tr>
<th>buffer</th>
<th>count</th>
<th>Conditions</th>
<th>by</th>
<th>entry queue</th>
<th>Turn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>full</td>
<td>empty</td>
<td>signal</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Ø</td>
<td>C1</td>
<td>Ø</td>
<td>P1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
</tbody>
</table>

P1 enters/accesses
C2 enters
P5 blocked on (5)
C1 signalled on (3)
P3 enters
P2 enters
C2 blocked on (2)
C1 blocked on (2)
P1 loops on s.q
C2 signalled on (3)
P3 blocked on (5)