Mutual Exclusion and Synchronization

Ezio Bartocci
Institute for Computer Engineering
ezio.bartocci@tuwien.ac.at
OS and Processes, parallel running processes
  – Use/access same data / resources (shared memory, files, devices)
  – Exchange data and cooperate

Therefore we need:
  – Access in a „disciplined“ way to shared data/resources
  – Controlled execution for distributed series of actions

Otherwise erroneous behaviour: Inconsistency or rather dependency of results of a series of executed tasks of the single processes
Example 1

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = a + 100 )</td>
<td>( a = a - 100 )</td>
<td>( a_{\text{alt}} - 100 )</td>
</tr>
<tr>
<td><code>mov A, a</code></td>
<td><code>mov B, a</code></td>
<td><code>mov B, a</code></td>
</tr>
<tr>
<td><code>add A, 100</code></td>
<td><code>sub B, 100</code></td>
<td><code>sub B, ...</code></td>
</tr>
<tr>
<td><code>mov a, A</code></td>
<td><code>mov a, B</code></td>
<td><code>mov a, A</code></td>
</tr>
<tr>
<td><code>mov A, a</code></td>
<td><code>mov B, a</code></td>
<td><code>mov A, a</code></td>
</tr>
<tr>
<td><code>add A, ...</code></td>
<td><code>sub B, ...</code></td>
<td><code>add A, ...</code></td>
</tr>
<tr>
<td><code>mov a, A</code></td>
<td><code>mov a, B</code></td>
<td><code>mov a, A</code></td>
</tr>
</tbody>
</table>

\( a_{\text{neu}} \): \( a_{\text{alt}} \)
Example 2

Process A

\[
\begin{align*}
t &:= 0; \\
\text{loop} & \text{ } \\
\text{ShM} &:= t; \\
t &:= (t+1) \mod 10; \\
\text{end loop}
\end{align*}
\]

Process B

\[
\begin{align*}
\text{loop} & \text{ } \\
t &:= \text{ShM}; \\
\text{print}(t); \\
\text{end loop}
\end{align*}
\]

„parallel“ execution of A a. B

<table>
<thead>
<tr>
<th>ShM</th>
<th>print</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Output:

\[
\begin{align*}
1 & 1 & 2 \\
0 & 2 & 2 \\
0 & 1 & 2
\end{align*}
\]
Overview

• Terms and Targets
  – Mutual Exclusion
  – Condition Synchronization

• Programming of mutual exclusion and condition synchronization
  – Mechanisms (semaphore, messages, etc.)
  – Examples
Interaction between Processes

- **Mutual Exclusion**
  - Prevent „concurrent“ access to resources
  - Uninterruptability of a series of actions (atomicity)
  - Goal: keep data consistent

- **Condition Synchronization**
  - Wait for an event to happen
  - Goal: specific series of operations
Mutual Exclusion and Cond. Synch

- Orthogonal concepts
- Four possible combinations

<table>
<thead>
<tr>
<th>MutEx.</th>
<th>CondSync.</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td>−</td>
<td>Independent action</td>
</tr>
<tr>
<td>−</td>
<td>+</td>
<td>given sequence</td>
</tr>
<tr>
<td>+</td>
<td>−</td>
<td>consistency</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>consistency and sequence</td>
</tr>
</tbody>
</table>
Terms: Mutual Exclusion

Critical section, shared memory region

Data

Critical section

$P_1$  $P_2$
Critical Section

- When a process accesses shared data or resources, it is in the *critical section*
- **Mutual Exclusion**: at any time there is not more than one process in its critical section
- Entering critical sections need to take place in a controlled way
Process Structure for Critical Sections

- No process is allowed to enter its critical section, when there is already another process in its critical section.

- Process structure

  ...  
  Prologue c.s.  
  critical Section  
  Epilogue c.s.  
  ...  

  Entering Protocol  

  Exit-Protocol
Requirements to a c.s. Solution

• Mutual Exclusion

• Progress
  – When there is no process in its c.s. and processes want to access their c.s., the decision which process is next must not postponed indefinitely
  – (no lifelock, „after you“)

• Bounded Waiting
  – The number of entrances in the c.s. is bounded after a process requested to enter its c.s.
  – (no starvation)
Possibilities for Solutions

• Software-based solution
  – Assumption: atomicity of a read/write operation from/to the memory

• Hardware-supported solution
  – More powerful atomic machine instructions

• Higher synchronization structures
  – Provide the programmer with data structures and functions
  – semaphore, monitor, message passing, …
Mutual Exclusion in Software
Software-based Solutions

- At first 2 processes, $P_0$ and $P_1$
- Then, generalization to n processes, $P_i$, with $P_i \not< P_j$, for $i \not< j$
- HW: atomic reading/writing a single value
- Synchronisation based on global variable
- *Busy Waiting*: waiting for a condition to occur by constantly checking
- (busy ... needs CPU time)

```
while condition != true do nothing
```
Dekker-Algorithm, 1. Try

global:

\[ \text{var turn: } 0 .. 1; \]

Process \( P_i \):

\[ \text{while turn} <> i \text{ do nothing; critical section} \]
\[ \text{turn} = j; \]
\[ \text{remainder section} \]

- Mutual exclusion given
- Processes can enter c.s. only alternately
- When one process terminates, the other one is blocked endlessly
Dekker-Algorithm, 2. Try

global:

```
var flag: array [0..1] of boolean;
```

Process $Pi$:

```
while flag[j] do nothing;
flag[i] := true;
```

**Critical Section**
```
flag[i] := false;
```

**Remainder Section**
```
flag[j] := false;
```

**Initialization**:
```
flag[i] := false;
flag[j] := false;
```

- When one process terminates, the other one can continue
- Mutual exclusion not guaranteed
Dekker-Algorithm, 2. Try

global:

flag
False False

Process $P_0$:

\[ \text{while flag[1] do nothing; } \]
\[ \text{flag[0] := true; } \]
\[ \text{critical section} \]
\[ \text{flag[0] := false; } \]
\[ \text{remainder section} \]

Process $P_1$:

\[ \text{while flag[0] do nothing; } \]
\[ \text{flag[1] := true; } \]
\[ \text{critical section} \]
\[ \text{flag[1] := false; } \]
\[ \text{remainder section} \]

• Mutual exclusion not guaranteed
global: \[ \text{var flag: array } [0..1] \text{ of boolean;} \]

Process \( P_i \):
\[
\begin{align*}
\text{flag}[i] & := \text{true;} \\
\textbf{while} \text{ flag}[j] \textbf{ do nothing;} \\
\text{critical section} \\
\text{flag}[i] & := \text{false;} \\
\text{remainder section}
\end{align*}
\]

- Mutual exclusion given
- But: mutual blocking (\textit{Deadlock}) possible
Dekker-Algorithm, 3. Try

Process $P_0$:

flag[0] := true;

\textbf{while} flag[1] \textbf{do} nothing;

critical section
flag[0] := false;
remainder section

Process $P_1$:

flag[1] := true;

\textbf{while} flag[0] \textbf{do} nothing;

critical section
flag[1] := false;
remainder section

Deadlock
Dekker-Algorithm, 4. Try

Process $P_i$:  
flag[i] := true; 
while flag[j] do 
begin 
  flag[i] := false; 
  wait for a short time; 
  flag[i] := true; 
end;  
critical section 
flag[i] := false; 
remainder section

- Mutual exclusion given, no deadlock 
- But: *Lifelock*, active processor without productivity
Dekker-Algorithm, 4. Try

global:

flag := false

Process \( P_0 \):

flag[0] := true;
while flag[1] do
begin
flag[0] := false;
wait for a short time;
flag[0] := true;
end;
critical section
flag[0] := false;
remainder section

Process \( P_1 \):

flag[1] := true;
while flag[0] do
begin
flag[1] := false;
wait for a short time;
flag[1] := true;
end;
critical section
flag[1] := false;
remainder section
flag[i] := true;
while flag[j] do
  if turn = j then begin
    flag[i] := false;
    while turn = j do nothing;
    flag[i] := true;
  end;
critical section

flag[i] := false;

remainder section

Turn … decides the order of entering the c.s. in case of a total concurrency (see 4. try)

flag … State of both processes
More elegant, easier to understand than Dekker alg.

Process $P_i$:

```plaintext
loop
    flag[i] := true;
    turn := j;
    while flag[j] and turn = j do nothing;
    critical section
        flag[i] := false;
        remainder section
end loop
```

Initialization:

```plaintext
flag[i] := false;
flag[j] := false;
```
• **Mutual Exclusion**
  - $P_0$ sets $flag[0]$ to $true$ $\rightarrow$ $P_1$ blocks
  - $P_1$ in c.s. $\rightarrow$ $flag[1] = true$ $\rightarrow$ $P_0$ blocks

• **No endless mutual blocking**
  
  Assumption: $P_0$ blocks in while-loop (i.e., $flag[1] = true$ and $turn = 1$)
  - $P_1$ does not want to enter c.s. $\rightarrow$ $flag[1] = true$
  - $P_1$ waits to enter c.s. $\rightarrow$ $turn = 1$
  - $P_1$ monopolizes c.s. $\rightarrow$ $P_1$ needs to set $turn = 0$ before entering the c.s. again
Bakery Algorithm

Solution to the c.s.-problem for n processes

• Assigns numbers to c.s.; process with lowest number (>0) enters first
• Number 0 ... no demands for c.s.
• $P_i$ and $P_j$ with same number: $P_i$ enters c.s. before $P_j$, if $i < j$
• Always valid: newly assigned number $\geq$ previously assigned number

```
var choosing: array [0..n-1] of boolean;
number: array [0..n-1] of integer;
```
Bakery Algorithm

```
loop
    choosing [i] := true;
    number [i] := 1 + max(number [0], … , number [n-1]);
    choosing [i] := false;
for j := 0 to n-1 do
    begin
        while choosing [j] do nothing;
        while number [j] <> 0 and (number [j],j) < (number [i],i) do nothing;
    end;
end loop
```
Hardware-based Solution

• Interrupt Disabling
  – Suitable for uniprocessors
  – But: restrictions for the OS on dispatching other tasks
  – Control return to OS?
  – Protection of sequences of instructions in OS

• More powerful atomic machine instructions
  – Test and Set
  – Exchange (Swap)
Test and Set

• Hardware instruction which executes two actions in an atomic way (i.e., uninterrupted)
• \(\rightarrow\) Mutual Exclusion

```plaintext
function testset (var i: integer): boolean;
begin
    if i = 0 then
        begin
            i := 1;
            return testset := true;
        end;
    else return testset := false;
end.
```
Test and Set for C.S.

global var.:

```plaintext
var b: integer;
b := 0;
```

Process $P_i$:

```plaintext
while not testset(b) do nothing;

critical section
b := 0;

remainder section
```
Back-up the C.S. witch „exchange“

- Simple prologue for the c.s. for any number of processes
- Needs busy waiting
- Starvation of processes possible

```pascal
var b: integer;
b := 0;

critical section
exchange (key, b); remainder section

var key: integer;
key := 0;

key := 1;
do exchange (key, b) while key = 1;
```
Busy Waiting

- Iterates a loop until a synchronization criteria is fulfilled
- Process wastes CPU time with waiting
- No suitable mechanism for synchronizing user-code
- Alternative to busy waiting: block waiting processes in *Blocked Queues* (see section „Processes“)
Semaphore
Semaphore

• Synchronization mechanism without the need for busy waiting (offered by the O.S.)

• Semaphore S: Integer-variable, that is initialized and later accessed with only two atomic functions - *wait* and *signal*
Goal: use semaphores protect a critical section in a „simple way“

wait (S);
critical section
signal (S);
remainder section
Semaphore - Datastructure

• Semaphore is a record:

```
type semaphore = record
    value: integer;
    queue: list of process;
end;
```

• Processes that wait for a semaphore S are added to the *Blocked Queue* of S (S.queue)
Semaphore - Operations

init (S, val): S.value := val; S.queue := empty list;

wait (S): S.value := S.value - 1;
  if S.value < 0
  then add this process to S.queue and block it;

signal (S): S.value := S.value + 1;
  if S.value <= 0
  then remove a process P from S.queue and place P on ready queue;

Semaphores must be initialized with a non-negative value ("1" for mutual exclusion)
Mutual Exclusion

Initialization: \text{init (S, 1)}

Prozess $P_i$: \text{wait (S)}; \text{critical section} \text{signal (S)}; \text{remainder section}

- Max. one process in C.S.
- Order in which processes enter the C.S.?
- $\rightarrow$ FIFO Queue $\rightarrow$ Bounded Waiting, Fairness
Semaphore - Implementation

- *wait* and *signal* must be executed as atomic operations
- Protect the semaphore operation (=C.S.) with Test and Set
  - Additional record-component *flag* in semaphore data structure
  - For C.S.: busy waiting with test of *flag*
  - Because *wait* and *signal* are really short, there is hardly any busy waiting; the overhead is justifiable
Notes on Semaphores

- \( S\text{.count} \geq 0 \): Number of processes that can execute \textit{wait} one after the other without blocking
- Initialize \( S \) with value \( n \)
- \( n \) processes can simultaneously access resources
- \( n \) processes can simultaneously access resources
- \( n \) processes can simultaneously access resources
- \( n \) processes can simultaneously access resources
- \( S\text{.count} < 0 \): \(|S\text{.count}|\) processes blocked in queue of \( S \) (see shown implementation)
Notes on Semaphores

- **Binary Semaphore**: accepts only the values 0 and 1
- **Counting Semaphore**: accepts any number of (if necessary non-negative) integer values – depends on implementation
- Operations *wait* and *signal* are also called *P* (*proberen / passeren*) and *V* (*vrijgave / verhogen*) in literature
Condition Synchronization

- Different processes: $P_1$ and $P_2$
- Code section $C_1$ in $P_1$ must be executed before section $C_2$ in $P_2$
- Semaphore for condition synchronization

Initialization:

- $P_1$: $C_1$; signal ($S$);
- $P_2$: wait ($S$); $C_2$;
Example: data transfer from $P1$ to $P2$
$P1$ writes to, $P2$ reads from shared memory
No duplication or loss of data

Init.: \( \text{init (...)} \)

$P1$: \( \text{loop} \)
\hspace{1em} \text{gen. data;}
\hspace{1em} \text{write ShM;}
\hspace{1em} \text{end loop} \)

$P2$: \( \text{loop} \)
\hspace{1em} \text{read ShM;}
\hspace{1em} \text{use data;}
\hspace{1em} \text{end loop} \)

Alternating Access
Alternating Access

- Example: data transfer from $P1$ to $P2$
- $P1$ writes to, $P2$ reads from shared memory
- No duplication or loss of data

Init.: \( \text{init (S1, 1); init (S2, 0);} \)

$P1$: 

```
loop
  gen. data;
  wait (S1);
  write ShM;
  signal (S2);
end loop
```

$P2$: 

```
loop
  wait (S2);
  read ShM;
  signal (S1);
  use data;
end loop
```
Exercise

- Given: 3 cyclic processes, A, B, C
- A generates data and writes it to ShM
- B and C read data from ShM
- (without limiting the parallelism)
- Each data set written by A shall be read exactly once by B and C.
Semaphore – Examples
Producer-Consumer Problem

- Producer generates information that is read by consumers
  - E.g., printing task to a printer
- Single data sets are given, via a buffer, to a consumer
P-C with Unlimited Buffer

- Producer can write a data element at any time
- Consumer waits for data
- “in” points to next free buffer element
- “out” points to next readable buffer element
• Mutual Exclusion: only one process has access to the buffer at any time (Semaphore “S”)

• Condition synchronization: a consumer is allowed to read when there is at least one unread data element available in the buffer

• (Counting Semaphore “N” represents the number of available elements in the buffer)
Initialization:

init (S, 1); init(N, 0); in := out := 0;

append (v):
  b[in] := v;
  in := in + 1;

take ():
  w := b[out];
  out := out + 1;
  return w;

Producer:

loop
  produce (v);
  P (S);
  append (v);
  V (S);
  V (N);
end loop

Consumer:

loop
  P (N);
  P (S);
  w := take ();
  V (S);
  V (N);
  consume (w)
end loop
- Limited buffer with $K$ elements
- Read: at least one „new“ value necessary
- Write: at least one element writable
Ring-Buffer: Semaphore

• Semaphore as with unlimited buffer
  – Mutual Exclusion: only one process is allowed to point to the buffer at any time (S)
  – Condition synchronization: the consumer is allowed the read if and only if there is at least one data element available in the buffer (N)

• Condition synchronization: a producer is allowed to write if and only if there is at least one empty memory set available (E)
P-C Ring-Buffer Implementation

Initialization:
init (S, 1); init (N, 0); init (E, \( K \));
in := out := 0;

append (v):
\( b[in] := v; \)
in := (in + 1) mod \( K \);

take ():
\( w := b[out]; \)
out := (out + 1) mod \( K \);
return w;

Producer:
loop
produce (v);
P (E);
P (S);
V (S);
end loop

Consumer:
loop
P (N);
P (S);
w := take ();
append (v);
V (S);
V (E);
consume (w)
end loop
Sequence of V-operations „irrelevant“

**Attention:** Sequence of P-operations is relevant!

**Producer:**
- produce (v);
- P (E);
- P (S);
- append (v);

**Consumer:**
- ...  
  - P (S);
  - P (N);
  - w := take ();

**Deadlock,** if consumer enters C.S. with empty buffer (N=0)

exchanged! wrong!!!
There is a data area shared among a number of processes. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers).

- Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file
- If a writer is writing to the file, no reader may read it
init (x, 1); init (y, 1); init (z, 1); init (wsem, 1); init (rsem, 1); rc := 0; wc := 0;

**Reader:**

```plaintext
loop
   P (x);
   rc := rc + 1;
   if rc = 1 then P (wsem);
   V (x);
   read;
   P (x);
   rc := rc - 1;
   if rc = 0 then V (wsem);
   V (x);
end loop
```

**Writer:**

```plaintext
loop
   P (wsem);
   write;
   V (wsem);
end loop
```

**Readers have priority**

**Writers are subjected to starvation**
Reader-Writer Problem

Reader:

\[
\text{loop}
\]
\[
\begin{align*}
    & P(z); \\
    & P(rsem); \\
    & P(x); \\
    & rc := rc + 1; \\
    & \text{if } rc = 1 \text{ then } P(wsem); \\
    & V(x); \\
    & V(rsem); \\
    & V(z); \\
    \end{align*}
\]

\[
\begin{align*}
    & \text{read}; \\
    & P(x); \\
    & rc := rc - 1; \\
    & \text{if } rc = 0 \text{ then } V(wsem); \\
    & V(x); \\
\end{align*}
\]

end loop

Writer:

\[
\text{loop}
\]
\[
\begin{align*}
    & P(y); \\
    & wc := wc + 1; \\
    & \text{if } wc = 1 \text{ then } P(rsem); \\
    & V(y); \\
    & P(wsem); \\
    & \text{write}; \\
    & V(wsem); \\
    & P(y); \\
    & wc := wc - 1; \\
    & \text{if } wc = 0 \text{ then } V(rsem); \\
    & V(y); \\
\end{align*}
\]

end loop

\text{Writer have priority}

Readers are allowed to access to the data area once at least one Writer has declared a desire to write.
Reader-Writer Problem

**Reader:**

\[
\text{loop} \\
\quad P(z); \\
\quad P(rsem); \\
\quad P(x); \\
\quad rc := rc + 1; \\
\quad \text{if } rc = 1 \text{ then } P(wsem); \\
\quad V(x); \\
\quad V(rsem); \\
\quad V(z); \\
\quad \text{read;} \\
\quad P(x); \\
\quad rc := rc - 1; \\
\quad \text{if } rc = 0 \text{ then } V(wsem); \\
\quad V(x); \\
\quad \text{end loop}
\]

**Writer:**

\[
\text{loop} \\
\quad P(rsem); \\
\quad P(wsem); \\
\quad \text{write;} \\
\quad V(wsem); \\
\quad V(rsem); \\
\quad \text{end loop}
\]

*Writer* have priority,
no starvation of readers

Compared to previous solution?
Classical synchronization problem (in teaching)
5 philosophers think and eat
Each philosopher needs two forks to eat
Seek: solution without deadlock and starvation
Dining Philosophers Problem

- One process represents one philosopher
- One semaphore per fork

```plaintext
fork: array[0..4] of semaphore;
foreach i in [0..4]
  init (fork[i], 1);
```

First attempt: deadlock, if all philosophers take the first fork simultaneously

First attempt:

```
Process P_i:

loop
  think;
  P (fork[i]);
  P (fork[(i+1) mod 5]);
  eat;
  V (fork[(i+1) mod 5]);
  V (fork[i]);
end loop
```
Dining Philosophers: Solution

- Additional semaphore that allows max. 4 philosophers to take their fork
- At least one process takes the fork in reversed order
- Atomic P-operation for multiple semaphores

\[ \text{mP} \left( \text{fork}[i], \text{fork}[\text{(i+1) mod 5}] \right); \]
P and V for various Semaphores

- $mP, mV$: atomic operations $P$ and $V$ for a set of semaphores
- $mP (S_1, S_2, \ldots, S_n)$: block until all semaphores $S_1$ to $S_n$ are bigger than 0
  - Solution to the ordering-problem of $P$-operations
- $mV (S_1, S_2, \ldots, S_n)$: increments all semaphores $S_1$ to $S_n$ by 1
Given: two types of processes: $A$, $B$
Processes of type $A$ and $B$ access the ShM in an exclusive way
Processes of type $A$ have priority over processes of type $B$
Spinlocks

- Mehrwertige (Multi-value?) semaphores realized with busy waiting
- Protection of short critical section on multiprocessor systems
- Sicherung kurzer kritischer Abschnitte auf multiprocessor systems
- CPU-overhead, but no process switch

\[
P(S):
S := S - 1;
\textbf{while} S < 0 \textbf{do} \text{nothing};
\]

\[
V(S):
S := S + 1;
\]
Sequencer and Eventcounts

Mechanisms to directly control the sequences of actions (Cond. Sync.)
Eventcount $E$: integer variable to count the number of occurrences of events. The initial value of $E$ is 0.

$\text{advance (} E \text{)}$: increments $E$ by 1

$\text{await (} E, \text{ val) }: $ blocks until $E \geq \text{ val}$
Every 10th occurrence of an event E1 shall print the number of events.

At each occurrence of E1:

\[
\text{advance (E)};
\]

\[
P1: \quad t := 0; \\
\text{loop} \\
\quad t := t + 10; \\
\quad \text{await (E, t);} \\
\quad \text{print (t);} \\
\text{end loop}
\]
Sequencer $S$: integer variable with initial value 0, on which the operation $ticket$ is defined. 

$ticket (S)$: atomic action, provides the sequencer’s value and subsequently increments the value of $S$ by 1.

Eventcounts and Sequencer: compare distribution of tickets (numbers) and waiting to be called at a counter (council office, doctor, etc.)
• Protecting a C.S.

sequencer: S;
eventcount: E;
await (E, ticket (S));
critical section;
advance (E);

semaphore: S;
init (S, 1);
P (S);
critical section;
V (S);

cmp.:
E and S: Producer-Consumer

sequencer: Pticket, Cticket;  eventcount: In, Out;

Producer:

```
loop
  produce (v);
  t := ticket (Pticket);
  -- only a single producer
  await (In, t);
  -- writable element
  await (Out, t-K+1);
  b[t mod K] := v;
  -- element written
  -- next producer
  advance (In);
end loop
```

Consumer:

```
loop
  u := ticket (Cticket);
  -- only a single consumer
  await (Out, u);
  -- readable data
  await (In, u+1);
  w := b[u mod K];
  -- data read
  -- next consumer
  advance (Out);
  consume (w)
end loop
```
Eventcounts with/without Sequencer

- Eventcounts *with sequencers*, if new starting processes need to synchronize with running processes
- (e.g., server with multiple clients)
- Usage of eventcounts *without sequencers*
  - Condition synchronization
  - Fixed size of parallel running processes
  - Predefined (possibly cyclic) synchronization pattern
  - Change states in *State Machine*
Eventcounts without Sequencer

```
t := 0;
loop
    await (E, t);
    S0_act;
    advance (E);
    t := t+3;
end loop
```

```
t := 1;
loop
    await (E, t);
    S1_act;
    advance (E);
    t := t+3;
end loop
```

```
t := 2;
loop
    await (E, t);
    S2_act;
    advance (E);
    t := t+3;
end loop
```
Access to shared resource:
- Local copies of $E$ at each node
- ticket ($S$): request a ticket from master
- advance ($E$): message to all nodes to increment the local copy of the event count
- await ($E, val$): unchanged
Problems with Sync-Constructs

• Operations on semaphores or eventcounts and sequencers are distributed over processes, thus unclear/confusing
• Operations must be called correctly in all processes
• A single erroneous process causes erroneous behavior of all processes that work together

▷ Monitors
Monitor, Messages
Monitor

• Software module, consists of:
  – Procedures
  – Local data
  – Code for initialization

• Properties:
  – Access local variable: monitor procedure
  – Processes enter the monitor via monitor procedures
  – Max. 1 process in monitor at any time
Monitor

- Monitor provides mutual exclusion, no explicit programming
- Shared memory is created in monitor region
- Condition synchronization with monitor variable
  - Programming waiting-conditions for *Condition Variables*
  - Monitor entrance if waiting-condition of condition variable is fulfilled
Condition Variables

• Local in monitor (accessed only in monitor)
• Accessed only via access functions
  – \textit{cwait (c)}: blocks calling process until condition (variable) \( c \) gets value \textit{true}
  – \textit{csignal (c)}: continues a process that is waiting for condition \( c \)
    • if multiple processes wait, select one
    • if no process waits, no action
    • no saving effect (< > semaphore-wait)
Monitor: Producer-Consumer

monitor ProducerConsumer

\( b: \text{array}[0..K-1] \text{ of items; } \)
\( \text{In} := 0, \text{ Out} := 0, \text{ cnt} := 0 : \text{integer; } \)
\( \text{notfull, notempty: condition; } \)

append (v):
\[
\text{if (cnt = K) cwait (notfull);} \\
b[\text{In}] := v; \\
\text{In} := (\text{In} + 1) \mod K; \\
\text{cnt} := \text{cnt} + 1; \\
c\text{signal (notempty);}
\]

take (v):
\[
\text{if (cnt = 0) cwait (notempty);} \\
v := b[\text{Out}]; \\
\text{Out} := (\text{Out} + 1) \mod K; \\
\text{cnt} := \text{cnt} - 1; \\
c\text{signal (notfull);}
\]
Message Passing

- Method for interprocess communication (IPC) – for single computers or distributed systems
- Mechanisms for synchronisation and mutual exclusion
- A message is an atomic data structure
- Functions to send and receive
  - send (destination, message)
  - receive (source, message)
Characteristics of Messages

- **Synchronization**
  - send: blocking, non-blocking
  - receive: blocking, non-blocking, test for arrival

- **Data management**
  - Queue
  - Received data override old values

- **Addressing**
  - 1:1 vs. 1:N
  - physical vs. logical
  - direct vs. indirect (mailbox, post addressing)
Mailbox and Post Addressing

Mailbox

Port

P_1 \quad Q_1

\ldots \quad \ldots

P_n \quad Q_n

P_1 \quad Q

\ldots \quad \ldots

P_n
Event Messages for MutEx

- Processes use a shared Mailbox `mutex` and a message (token)
- Blocking `receive`, (non-blocking) `send`

**init:**
```
send (mutex, msg);
```

**Process $P_i$:**
```
loop
  receive (mutex, msg);
  critical section;
  send (mutex, msg);
  remainder section;
end loop
```
Summary

• Requirements to parallel running processes
  – Consistent access to shared data
  – Predefined sequence of actions
• Mutual exclusion $\rightarrow$ consistency
• Condition synchronization $\rightarrow$ sequence
• Critical section
  – Actions, that manipulate shared data
  – Protected by constructs for synchronization
Summary

• Protecting the critical section
  – Solutions depend on instruction set
  – Supported by operating system
    ➲ no busy waiting!

• Semaphore
  – init, wait and signal, or P and V
  – Attention: Sequence of wait operations

• Eventcounts and Sequencer
  – await, advance, ticket
  – Open vs. clear predefined synchronization sequence
Summary

• Monitor
  – Encapsulation of shared objects, access functions and condition variables

• Messages
  – Atomic data structures
  – Different semantic for synchronization
  – Protection of C.S. with mailbox and token-messages