Peterson-Algorithm

more elegant, simple than Dekker Alg.

process $P_i$:

\begin{verbatim}
loop
  flag[i] := true;
  turn := j;
  while flag[j] and turn = j do nothing;
  critical section
    flag[i] := false;
    remainder section
end loop
\end{verbatim}

Initialization:

flag[i] := false;
flag[j] := false;

---

- mutual exclusion
  - $P_0$ sets flag[0] to true $\Rightarrow$ $P_1$ blocked
  - $P_1$ in k.A. $\Rightarrow$ flag[1] = true $\Rightarrow$ $P_0$ blocked

- no infinite mutual blocking:
  assumption: $P_0$ is blocked in while-loop
  (i.e. flag[1] = true and turn = 1)
  - $P_1$ does not want into c.s. $\Rightarrow$ flag[1] = true
  - $P_1$ waiting for c.s. $\Rightarrow$ turn = 1
  - $P_1$ monopolizes c.s. $\Rightarrow$ $P_1$ has to set turn to 0 before entering c.s. again
Bakery Algorithm

Solution to mutual exclusion for \( n \) processes

- assign numbers for c.s.; process with lowest number enters first
- \( P_i \) and \( P_j \) with same numbers:
  \( P_i \) enters before \( P_j \) if \( i < j \)
- ensure that: current numer \( \geq \) previously assigned numbers

\[
\text{var} \quad \text{choosing: array [0..n-1] of boolean;}
\text{number: array [0..n-1] of integer;}
\]

- defined and proven by Leslie Lamport
- prove on his homepage

\[
\begin{align*}
\text{loop} & \quad \text{choosing}[i] := \text{true;} \\
& \quad \text{number}[i] := 1 + \max(\text{number}[0], \ldots, \text{number}[n-1]); \\
& \quad \text{choosing}[i] := \text{false;}
\end{align*}
\]

\[
\begin{align*}
\text{for } j := 0 \text{ to } n-1 \text{ do} \\
\begin{align*}
& \quad \text{begin} \\
& \quad \text{while } \text{choosing}[j] \text{ do nothing; /* warte auf Nummer */} \\
& \quad \text{while } \text{number}[j] <> 0 \text{ and} \\
& \quad \quad \text{(number}[j],j) < (\text{number}[i],i) \text{ do nothing;} \\
& \quad \text{end;} \\
& \quad \text{critical section} \\
& \quad \text{number}[i] := 0; \\
& \quad \text{remainder section}
\end{align*}
\end{align*}
\]

Bakery Algorithm, ctd.
Bakery Algorithm, ctd.

- lack of lower level mutual exclusion
  - each memory location written only by one process
  - reads shared
- algorithm works even if read is overlapped by write!
  - independent of read value
- was discovered during proof

Hardware supported Solutions
Atomic Actions in Hardware

- interrupt disabling
  - suitable for uniprocessor system
  - limits OS when dispatching other tasks
- special machine instructions
  - Test and Set
  - Exchange

Test and Set

- hardware instruction executing two actions atomically (cannot be interrupted)
  ⇒ mutual exclusion

```pascal
function testset (var i: integer): boolean;
begin
  if i == 0 then
    begin
      i := 1;
      testset := true;
    end;
  else testset := false;
end.
```
Test and Set for Mutual Exclusion

global var.:  
\[ \text{var } b: \text{ integer; } \]
\[ b := 0; \]

process \( P_i \):  
\[ \text{while not testset}(b) \text{ do nothing; } \]
\[ \text{critical section } \]
\[ b := 0; \]
\[ \text{remainder section } \]

- simplifies prologue of c.s.
- requires busy waiting
- *Starvation* of processes possible

Semaphores
Semaphore

- mechanism for synchronization, which does not need busy waiting (provided by OS)
- Semaphore S: Integer-Variable, which can be accessed only via two atomic functions: `wait` and `signal`
- instead of busy waiting: blocking of waiting process in `blocked queues`
- E. Dijkstra

Semaphore

usage of semaphore to guard critical section

```plaintext
wait (S);
critical section
signal (S);
remainder section
```
Semaphores – Data Structure

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

- processes waiting for a semaphore S put in Blocked Queue of S
- Signal removes a process from blocked queue and puts it into Ready Queue

Semaphores - Operations

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

initialization: semaphores must be initialized to non negative value
Semaphores - Operations

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

Semaphores - Operations

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 - Implementation

- *wait* and *signal* have to be atomic operations
- ensuring semaphore operations with test and set
  - additional component in record *flag* in semaphore data structure
  - before c.s.: busy waiting until *flag* = 0
  - *wait* and *signal* are very short, so busy waiting overhead acceptable

Semaphores - Remarks

- *S.count* \(\geq 0\): number of processes, which can execute *wait* in sequence without having to wait
- initialization of S with value \(n\) \(\Rightarrow n\) processes can enter c.s. at same time
- *S.count* < 0: \(|S.count|\) processes blocked in Queue of S
Semaphores - Remarks

- **binary semaphore**: only values 0 and 1
- **counting semaphore**: can have arbitrary non-negative integer values
- operations *wait* and *signal* also called *P* and *V*

Mutual Exclusion

**initialization**: `init (S, 1)`

**process** \(P_i\):

- `wait (S)`;
- `critical section`;
- `signal (S)`;
- `remainder section`

- maximum one process in c.s.
- order of processes entering c.s. is not defined a-priori
Condition Synchronization

- processes: \( P1 \) and \( P2 \)
- part of code \( C1 \) in \( P1 \) has to be executed before \( C2 \) in \( P2 \)
- semaphore for Condition Synchronization

Initialization:

\[
\begin{align*}
\text{init (S, 0)} \\
\text{P1:} & \quad \text{C1; signal (S);} \\
\text{P2:} & \quad \text{wait (S); C2;}
\end{align*}
\]

Alternating Access to Critical Section

- example: data transfer from \( P1 \) to \( P2 \)
- \( P1 \) writes, \( P2 \) reads shared memory
- no duplication or loss of data

Init.:

\[
\begin{align*}
\text{init (S1, 1); init (S2, 0);} \\
\text{P1:} & \quad \text{loop} \\
& \quad \text{gen. data; wait (S1); write ShM; signal (S2); end loop} \\
\text{P2:} & \quad \text{loop} \\
& \quad \text{wait (S2); read ShM; signal (S1); use data; end loop}
\end{align*}
\]
Semaphores - Examples

Producer-Consumer Problem

- produce generates data, which is read by consumer
- single data entries are given to consumer via buffer
P-C with unlimited Buffer

- producer can write data anytime
- consumer has to wait for data
- "in" points to next free element
- "out" points to next element to read

```
```

out       in

P-C: semaphor

- mutual exclusion: at each point in time, only one process may access buffer (semaphor "S")
- condition synchronization:
  - consumer may only read, when buffer contains at least one unread data entry
  - (Counting semaphor "N" reflects number of element in buffer; in-out)
P-C: Implementation

Initialization:

\[
\text{init (S, 1); init(N, 0); in := out := 0;}
\]

Producer:

\[
\text{loop}
\]

\[
\text{produce (v); P (S); append (v); V (S); V (N);
end loop}
\]

Consumer:

\[
\text{loop}
\]

\[
\text{P (N); P (S); w := take (); V (S); consume (w); end loop}
\]

P-C with Ringbuffer

- limited buffer with \( K \) elements
- read: at least one “new” data entry required
- write: at least one element writable
Ringbuffer: Semaphor

- semaphor as in case with unlimited buffer
  - mutual exclusion: only one process may access buffer at each time (S)
  - condition synchronization: consumer may only read when at least one unread data entry in buffer (N)
- condition synchronization: producer may only write, when at least one empty space in buffer (E)

P-C Ringbuffer-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 (v):
    w := b[out];
    out := (out + 1) mod K;
    return w;

**Producer:**

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

**Consumer:**

loop
    P (N);
    P (S);
    w := take (v);
    V (S);
    V (E);
    consume (w);
end loop
Order of P and V

order of V-Operations “arbitrary”

**note:** order of P-Operations important!

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

*Consumer:* ...

\[
\begin{align*}
\text{P (S)}; \\
\text{P (N)}; \\
w := \text{take ();}
\end{align*}
\]

swapped! wrong!!!

deadlock, if consumer enters c.s. when buffer is empty