Operating Systems
2010/2011

Action Synchronization

Johan Lukkien
Agenda

• Action synchronization
  – formalization
  – Semaphores
  – producer/consumer
• POSIX examples
• Action synchronization
  – mutual exclusion
  – bounded buffer
Communication & synchronization

• Synchronization: .... limitation of possible traces
  – coordination of execution such as to let this execution satisfy a certain invariant
    • i.e., avoid the traces that violate that invariant
  – or just steering the execution to have some property
    • e.g. such that a certain assertion holds during execution
  – typically, by sometimes blocking thread execution until an assertion has become true

• We use the statement *await (B)* to denote blocking until a condition *B* holds. We study then some ways to implement this statement
Example: Vendor and Machine

\[
\begin{align*}
\text{Proc } & \text{Vendor } = \\
& \left[ \text{ while } \text{true do} \right. \\
& \quad \text{DriveToFactory}; \\
& \quad \text{await } (\text{Stock}+\text{Load} \leq \text{MAX}); \\
& \quad \{ \text{Stock}+\text{Load} \leq \text{MAX} \} \\
& \quad \text{Stock} := \text{Stock}+\text{Load}; \\
& \quad \text{DriveBack}; \\
& \quad \text{ReLoad} \\
& \left. \text{od} \right] \\
\end{align*}
\]

\[
\begin{align*}
\text{Proc } & \text{Machine } = \\
& \left[ \text{ while } \text{true do} \right. \\
& \quad \text{await } (\text{Stock}>0); \\
& \quad \{ \text{Stock}>0 \} \\
& \quad \text{Stock} := \text{Stock}-1; \\
& \quad \text{Manufacture} \\
& \left. \text{od} \right] \\
\end{align*}
\]
Issues around the example (1/2)

• Implementing the *await* using repeated testing works if
  – the assignments (and tests) are atomic *and* ...

  • however, usually, the update is a sequence of actions – i.e., a critical section, which is not atomic ... hence needs mutual exclusion
    – Even a single actions like \( x := x + 1 \) becomes \( r := x; r := r + 1; x := r \), where \( r \) is an internal register with atomic assignments

  – ... at most one Vendor and one Machine exists
  • otherwise, ‘race conditions’ occur (why and how?)

• Repeated testing is called: *busy waiting*, acceptable only if
  – waiting is guaranteed short or
  – there is nothing else to do anyway (e.g. in dedicated hardware)

• Busy waiting, when done at the level of an application above an OS, costs performance (why?)
  – hence, rely on OS primitives to solve waiting
  – we are studying this
Issues around the example (2/2)

• May introduce extra variables to steer behavior more precisely
  – e.g. no Machine is allowed when Vendor is waiting
  – exercise

• The shared variables give problems
  – these lead to an essential non-compositionality: when a (correct) program is modified, everything must be verified again to check for new interference
    • e.g. going from one to two machines
  – a ‘distributed’ realization, with one ‘maintainer’ (writer) per shared variable is often better/easier
Agenda

• Action synchronization
  – formalization
  – Semaphores
  – producer/consumer

• POSIX examples

• Action synchronization
  – mutual exclusion
  – bounded buffer
Specifying synchronization

**Invariant**: assertion that holds at *all* control points

**Examples:**
- \( I : \) “mutual exclusion is maintained”
- \( I : y \leq x \) in the program below (assuming the assignments are atomic)

```
Initially: x=0 \land y=0

while true do <x := x+1>; <y := y+1> od
\|  
while true do <y := y-1>; <x := x-1> od
```
Terminology: naming and counting

Naming of actions

Initially: $x=0 \land y=0$

\[
\text{while true do A: } <x := x+1>; \quad \text{B: } <y := y+1> \quad \text{od}
\]

\[
\parallel
\]

\[
\text{while true do C: } <y := y-1>; \quad \text{D: } <x := x-1> \quad \text{od}
\]

If $A$ is an action in the program, $c_A$ denotes the number of completed executions of $A$. $c_A$ can be regarded as an auxiliary variable that is initially 0 and is incremented atomically each time $A$ is executed.

\[
A \quad \rightarrow \quad <A; c_A := c_A+1>
\]
Topology properties

Topology invariants: derived directly from the program text

Example: two actions always occurring one after the other

Initially: \( x=0 \land y=0 \)

\[
\text{while true do A: } <x := x+1>; \quad \text{B: } <y := y+1> \text{ od}
\]

\[
\text{while true do C: } <y := y-1>; \quad \text{D: } <x := x-1> \text{ od}
\]

Invariants:

\[
i_0: x = c_A - c_D \quad \quad i_2: 0 \leq c_A - c_B \leq 1
\]

\[
i_1: y = c_B - c_C \quad \quad i_3: 0 \leq c_C - c_D \leq 1
\]
Example

Showing invariance of \( I: y \leq x \)

\[
y \leq x
= \{ I0, I1 \}
  \quad cB - cC \leq cA - cD
= \{ I2: cB \leq cA, I3: cD \leq cC \}
  \quad \text{true}
\]

Note: such a proof \textit{must} refer somehow to topology because the property relies on it.
Synchronization conditions

- Action synchronization is specified by an inequality on action counts, or on program variables *directly related to this counting*.
- We refer to such an inequality as a *synchronization condition*, or a *synchronization invariant*.

\[
P_X = \\
x := 0; \\
\text{while } \text{true do} \\
\text{A: } <x := x+1> \\
\text{od} \\
\]

\[
P_Y = \\
y := 0; \\
\text{while } \text{true do} \\
\text{B: } <y := y+1> \\
\text{od} \\
\]

- Example: synchronize \( P_X \) and \( P_Y \) such that invariant

\[
I_0: x \leq y \quad (= cA \leq cB)
\]

is maintained.
The vendor-machine problem

• Invariant:
  – \( \text{Stock} = \text{Load} \times c (\text{Stock} := \text{Stock} + \text{Load}) - c (\text{Stock} := \text{Stock} - 1) \)

• Synchronization condition:
  – \( 0 \leq \text{Load} \times c (\text{Stock} := \text{Stock} + \text{Load}) - c (\text{Stock} := \text{Stock} - 1) \leq \text{MAX} \)
Semaphores (Dijkstra)

• Semaphore $s$ is an integer with initial value $s_0 \geq 0$ and **atomic** operations $P(s)$ and $V(s)$. The effect of these operations is defined as follows:

  \[
  P(s): \langle \text{await}(s>0); s := s-1 \rangle \\
  V(s): \langle s := s+1 \rangle
  \]

• “$>$” denotes again atomicity: the implementation of $P$ and $V$ must guarantee this

• ‘$\text{await}(s>0)$’ represents blocking until ‘$s>0$’ holds. This is indivisibly combined with a decrement of $s$

• a semaphore is therefore always non-negative

• Other names for $P$ and $V$: *wait/signal, wait/post, lock/unlock*

• Semaphores can be used to implement mutual exclusion
Semaphore invariants

From the definition we derive two semaphore properties (invariants):

\[ \begin{align*}
S0: & \quad s \geq 0 \\
S1: & \quad s = s_0 + cV(s) - cP(s)
\end{align*} \]

\( S0, S1 \): functional properties ("safety"). Combining:

\[ S2: \quad cP(s) \leq s_0 + cV(s) \]

hence, semaphores realize a synchronization invariant by definition

The implementation must pay attention on two more semaphore properties
- Progress: blocking is allowed only if the safety properties would be violated
- Semaphores may be fair (called strong, e.g. FIFO) or unfair (called weak)
Solve the producer/consumer problem

\[ P_X = \]
\[ x := 0; \]
\[ \textbf{while} \ true \ \textbf{do} \]
\[ A: <x := x+1> \]
\[ \textbf{od} \]
\[ P_Y = \]
\[ y := 0; \]
\[ \textbf{while} \ true \ \textbf{do} \]
\[ B: <y := y+1> \]
\[ \textbf{od} \]

Synchronize \( P_X \) and \( P_Y \) such that invariant
\[ I_0: x \leq y \]

is maintained.
Program topology

Use the program topology:
\[ x = cA \text{ and } y = cB \]
hence, \( I_0 \) can be rewritten

\[ I_0: cA \leq cB \]

Introduce semaphore \( s \); let \( A \) be \textit{preceded by} \( P(s) \) and \( B \) be \textit{followed by} \( V(s) \).

Topology:

\[ I_1: cA \leq cP(s) \]
\[ I_2: cV(s) \leq cB \]

Combine with semaphore invariant \( S_4 \):

\[ cA \leq cP(s) \leq s_0 + cV(s) \leq s_0 + cB \]

Hence, choosing \( s_0 = 0 \) does the job.
More restrictions

Suppose that we also want:

\[ I3: y \leq x + 10, \ i.e., \ \mathbf{cB} \leq \mathbf{cA} + 10 \]

Introduce a new semaphore \( t \). Let \( A \) be followed by \( V(t) \) and \( B \) be preceded by \( P(t) \). Then,

\[
\mathbf{cB} \leq \mathbf{cP(t)} \leq t_0 + \mathbf{cV(t)} \leq t_0 + \mathbf{cA}
\]

Choose \( t_0 = 10 \).

\[
\begin{align*}
P_X &= \quad x := 0; \\
& \quad \textbf{while } \textbf{true } \textbf{do} \\
& \quad \quad \textbf{P}(s); \ A: <x := x + 1>; \ V(t) \\
& \quad \textbf{od} \\
\end{align*}
\]

\[
\begin{align*}
P_Y &= \quad y := 0; \\
& \quad \textbf{while } \textbf{true } \textbf{do} \\
& \quad \quad \textbf{P}(t); \ B: <y := y + 1>; \ V(s) \\
& \quad \textbf{od} \\
\end{align*}
\]
And more...

Suppose that instead of $I_0$ we want

$I_4$: $2x \leq y$, i.e., $2cA \leq cB$

Let $A$ be preceded by two times $P(s)$ (denoted as $P(s)^2$). Then,

$2cA \leq cP(s)$

hence,

$2cA \leq cP(s) \leq s_0 + cV(s) \leq s_0 + cB$

etc....
Action Synchronization

**Given**: - collection of tasks/threads executing actions $A$, $B$, $C$, $D$; 
- a required *synchronization condition (invariant)*
  
  \[
  \text{SYNC: } a \cdot c_A + c \cdot c_C \leq b \cdot c_B + d \cdot c_D + e
  \]

  for non-negative constants $a, b, c, d, e$.

**Solution**: introduce semaphore $s$, $s_0 = e$ and replace

- $A \rightarrow P(s)^a; A$
- $B \rightarrow B; V(s)^b$
- $C \rightarrow P(s)^c; C$
- $D \rightarrow D; V(s)^d$

**Note**: during execution of $A$ and $C$ we have strict inequality in $\text{SYNC}$.
The vendor-machine problem

- Invariant:
  - $Stock = Load * c(Stock := Stock + Load) - c(Stock := Stock - 1)$

- Synchronization condition:
  - $0 \leq Load * c(Stock := Stock + Load) - c(Stock := Stock - 1) \leq MAX$

- Solution
  - Introduce two semaphores, $s$ and $t$
    - $s0 = 0$, $t0 = MAX$
    - adapt “$Stock := Stock + Load$”
      - precede with $Load$ times $P(t)$, follow with $Load$ times $V(s)$
    - adapt “$Stock := Stock - 1$”
      - precede with $P(s)$, follow with $V(t)$

- Note: mutual exclusion problem not solved with this. Needs separate attention
Synchronizing Vendor and Machine

**Proc Vendor** =
\[\begin{array}{l}
| [ \text{while true do} \\
\text{DriveToFactory;} \\
P(t)^{\text{Load}}; \\
\{ \text{Stock} + \text{Load} \leq \text{MAX} \} \\
\text{Stock} := \text{Stock} + \text{Load}; \\
V(s)^{\text{Load}}; \\
\text{DriveBack;} \\
\text{ReLoad} \\
\text{od} \\
\text{]} \]

**Proc Machine** =
\[\begin{array}{l}
| [ \text{while true do} \\
P(s); \\
\{ \text{Stock} > 0 \} \\
\text{Stock} := \text{Stock} - 1; \\
V(t); \\
\text{Manufacture} \\
\text{od} \\
\text{]} \]

\text{Load times a } P(t) \text{ operation}
\text{ i.e., } P(t); P(t); ... P(t);
Remarks

- One semaphore for each synchronization condition.
- Synchronization conditions may be conflicting. A deadlock may result.

**Example:** consider $P_X$ and $P_Y$ as before with

$$l4: \ 2c_A \leq c_B$$
$$l3: \ c_B \leq c_A+10$$

After a few steps, this system deadlocks

- Sometimes a deadlock can be avoided by imposing *extra* restrictions.
- Finding synchronization conditions can be painful.
Agenda

• Action synchronization
  – formalization
  – Semaphores
  – producer/consumer

• POSIX examples

• Action synchronization
  – mutual exclusion
  – bounded buffer
Counting semaphores (POSIX 1003.1b)

• Naming and creation
  – “name” within kernel, persistent until re-boot, like a filename
    • Posix names: for portability
      – start names with ‘/’
      – do not use any subsequent ‘/’
    • for use between processes or between threads
  – also “unnamed” semaphores, for use in shared memory
    • shared memory between processes
  – hence, two interfaces for creation and destruction
    • initialize existing memory structure & OS-level allocation

```c
sem_t *sem;
sem = sem_open (name, flags, mode, init_val); /* name is system-wide */
status = sem_close (sem); /* semaphore still reachable */
status = sem_unlink (name); /* now it is removed */
status = sem_init (sem, pshared, init_val); /* memory space for sem must be defined, e.g. through shm or malloc */
status = sem_destroy (sem); /* pshared: whether multiple processes * access sem; should be true */
```
Semaphore operations

- Basic interface, designed for speed

- Obtaining the value is tricky
  - value is unstable
  - negative value: interpret as number of waiters (length of queue)

```c
status = sem_wait (sem);
status = sem_trywait (sem); /* returns error (EBUSY?) if sem == 0 */
status = sem_post (sem);
status = sem_getvalue (sem, &val); /* current value */
    /* when negative: absolute value = # waiters */
```
#include <stdio.h>  
#include <fcntl.h>  
#include <pthread.h>  
#include <semaphore.h>  

sem_t *s, *t;

void Producer ()
{
    int i;
    for (i=0; i<10; i++) {
        sem_wait (t); printf("Produce "); fflush (stdout);
        sem_post (s); sleep (1);
    }
}

void Consumer ()
{
    int i;
    for (i=0; i<10; i++) {
        sem_wait (s); printf("Consume "); fflush (stdout);
        sem_post (t); sleep (2);
    }
}
void main ()
{
    pthread_t thread_id;

    s = sem_open ("Mysem-s", O_CREAT | O_RDWR, 0, 0);
    if (s == SEM_FAILED) { perror ("sem_open"); exit (0); }

    t = sem_open ("Mysem-t", O_CREAT | O_RDWR, 0, 4);
    if (t == SEM_FAILED) { perror ("sem_open"); exit (0); }

    pthread_create (&thread_id, NULL, Producer, NULL);
    Consumer ();
    pthread_join (thread_id, NULL);
    sem_close (s); sem_close (t);
    sem_unlink ("Mysem-s"); sem_unlink ("Mysem-t");
}
Output

- Produce Consume Produce Consume Produce Consume Produce Consume Produce Consume Produce Consume
- Produce Consume Produce Consume Produce Consume Produce Consume Produce Consume

- Question: is there a shared resource visible (and a race condition?)
Agenda

• Concurrency concepts
• Action synchronization
  – formalization
  – Semaphores
  – producer/consumer
• POSIX examples
• Action synchronization
  – mutual exclusion
  – bounded buffer
Mutual exclusion

Given are $N$ different processes, repeatedly executing a critical section.

\[
Pr_{(n, 0 \leq n < N)} = \begin{align*}
\text{while true do} & \\
\text{NonCriticalSection}(n) & \\
\text{CsEntry}(n) & \\
\text{CriticalSection}(n) & \\
\text{CsExit}(n) & \\
\text{od}
\end{align*}
\]

Maintain as synchronization requirement

\[
M: \left(\sum n: 0 \leq n < N: \_\text{CsEntry}(n) - \_\text{CsExit}(n)\right) \leq 1
\]
Mutual exclusion (cnt’d)

Rewriting

\[ M: \Sigma c_{\text{CsEntry}(n)} \leq 1 + \Sigma c_{\text{CsExit}(n)} \]

With action synchronization: introduce \( m, m_0 = 1 \).

\[
\begin{align*}
\text{CsEntry}(n) & \rightarrow P(m); \text{CsEntry}(n) \\
\text{CsExit}(n) & \rightarrow \text{CsExit}(n); V(m)
\end{align*}
\]

\((\text{CsEntry}(n)/\text{CsExit}(n)\) themselves can be “skip”.)

Semaphore \( m \) is called a \textit{binary semaphore} or a \textit{mutex} as opposed to a \textit{general semaphore} that can assume arbitrary non-negative values.
Making assignments critical sections

\[
\text{Proc } \text{Vendor } = \\
[\text{ while true do} \\
\text{  DriveToFactory;} \\
\text{  } P(t)^{\text{Load}}; \\
\text{  } \{ \text{Stock+Load} \leq \text{MAX} \} \\
\text{  } P(m); \\
\text{  } \text{Stock} := \text{Stock}+\text{Load}; \\
\text{  } V(m); \\
\text{  } V(s)^{\text{Load}}; \\
\text{  } \text{DriveBack;} \\
\text{  } \text{ReLoad} \\
\text{  od} \\
]\]

\[
\text{Proc } \text{Machine } = \\
[\text{ while true do} \\
\text{  } P(s); \\
\text{  } \{ \text{Stock} \geq 0 \} \\
\text{  } P(m); \\
\text{  } \text{Stock} := \text{Stock}-1; \\
\text{  } V(m); \\
\text{  } V(t); \\
\text{  } \text{Manufacture} \\
\text{  od} \\
]\]
Checking the correctness criteria

- Since we have solved a synchronization problem and introduced blocking we must verify the correctness criteria.

- **Functional correctness** (i.e., mutual exclusion) and **minimal waiting** are by construction.

- **Deadlock**: see next slide

- **Fairness**: the solution is just as fair as the semaphore(s).
Reasoning about deadlock

• A deadlocked state is a system state in which a set of threads or processes are blocked indefinitely
  – typically, each thread is blocked on another thread in the same set

• Prove absence of deadlock, typically by contraposition
  – assume, a deadlock occurs
  – investigate which blocked sets are possible (often: just 1)
  – show a contradiction
    • in principle: examine all possible combinations of blocking actions in all tasks

• Example: (exclusion semaphore from page 31)
  – Suppose a process is blocked on $P(m)$ - indefinitely
  – Since $m=0$ there is a process that is in its CS, hence also blocked indefinitely
  – This process apparently never leaves its CS

  – Hence, if all critical sections terminate, there is no deadlock caused by a semaphore used just for exclusion

• What about the vendor/machine example?
POSIX: mutex (1003.1c)

- Special, two-state (i.e., 1 / 0) semaphore: *mutex*
  - between threads
  - specifically for mutual exclusion
- Restrictions
  - don’t use copies of a mutex in the calls below
  - *lock()* and *unlock()* always by same thread (“ownership”)

```c
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
/* static initialization, not always possible */
status = pthread_mutex_init (&m, attr); /* attr: NULL; should return 0 */
status = pthread_mutex_destroy (&m); /* should return 0 */
status = pthread_mutex_lock (&m); /* should return 0 */
status = pthread_mutex_trylock (&m); /* returns EBUSY if m is locked */
status = pthread_mutex_unlock (&m); /* should return 0 */
```

\[ P(m)\]

\[ V(m)\]
Agenda

• Action synchronization
  – formalization
  – Semaphores
  – producer/consumer

• POSIX examples

• Action synchronization
  – mutual exclusion
  – bounded buffer
(Un)bounded buffer

Specification:
1. Sequence of values received equals sequence of values sent.
2. No receive before send.
3. For the bounded buffer: number of sends cannot exceed number of receives by more than a given positive constant $N$. 
Design

• Data structure supporting FIFO: queue \( q \), with operations \( PUT(q, x) \) and \( GET(q, y) \)
  – Introduce variable \( q \) of type queue.

• Exclusive access is required since \( PUT \) and \( GET \) are not atomic.
  – Introduce semaphore \( m \), \( m_0 = 1 \).

• The second requirement translates into \( c_{GET}(q,...) \leq c_{PUT}(q,...) \)
  – Introduce semaphores \( t \), \( t_0 = 0 \).

• The third requirement translates into \( c_{PUT}(q,...) \leq c_{GET}(q,...) + N \)
  – Introduce semaphore \( s \), \( s_0 = N \).
First solution

```haskell
type buffer =
record  q: queue of elem;
       s, t, m: Semaphore
end;
```

Notice the order of the \texttt{P}-operations: critical sections should always terminate

```haskell
proc Send (var b: buffer; x: elem) =
\[ \text{\begin{array}{l}
    \text{\begin{array}{l}
        with b do
        P(s); P(m); PUT(q,x); V(m); V(t)
    \end{array}}
    \text{od}
    \end{array} }\]
```

```haskell
proc Receive (var b: buffer; var y: elem) =
\[ \text{\begin{array}{l}
    \text{\begin{array}{l}
        with b do
        P(t); P(m); GET(q,y); V(m); V(s)
    \end{array}}
    \text{od}
    \end{array} }\]
```
Discussion

- **Functional correctness** and **minimal waiting** are again by construction.

- **Absence of deadlock** is due to the fact that the critical sections (i.e., the statements between $P(m)$ and $V(m)$) terminate; any permanent blocking must therefore be on the synchronization semaphores. The implementation *does not introduce* deadlock.

- The only competition is on accessing the queue. Only if semaphore $m$ is weak and the buffer is unbounded, an unlimited number of sends may occur.
Implementation: using arrays

Consider an infinite array as an implementation of a queue. Variables \( r \) and \( w \) denote read- and write positions respectively (initially 0).

\[
\text{type} \quad \text{queue} = \begin{array}{l}
\text{record} \quad b: \text{array of elem; } \\
\quad r, w: \text{int}
\end{array}
\]

\[
\begin{align*}
\text{proc} \quad \text{PUT} \quad (\text{var} \ q: \text{queue}; \ x: \text{elem}) &= \begin{array}{l}
\quad \| \begin{array}{l}
\quad \text{with} \ q \ 	ext{do} \\
\quad \quad \{ \ w = \text{cPUT}(q, \ldots) \} \\
\quad \quad b[w] := x; \ w := w+1
\end{array} \\
\quad \od
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
\text{proc} \quad \text{GET} \quad (\text{var} \ q: \text{queue}; \ \text{var} \ y: \text{elem}) &= \begin{array}{l}
\quad \| \begin{array}{l}
\quad \text{with} \ q \ 	ext{do} \\
\quad \quad \{ \ r = \text{cGET}(q, \ldots) \} \\
\quad \quad y := b[r]; \ r := r+1
\end{array} \\
\quad \od
\end{array} \\
\end{align*}
\]
Optimization

• We want to use a finite array of length $N$, used with indices modulo $N$
• Question: is it possible to leave out semaphore $m$ for synchronization?
  – then, the array may never be accessed at the same place
    • neither by $r = w$ or by $w - r = N$
  – to analyse, consider a concurrent access of consumer and producer

writer at “$b[w] := x$” and reader at “$y := b[r]$”
\[ \Rightarrow \quad \{ \text{use the program text + action synchronization: strict inequality} \} \]
\[ w = \text{PUT}(q) < \text{GET}(q)+N \land \]
\[ r = \text{GET}(q) < \text{PUT}(q) \]
\[ \Rightarrow \quad \{ \text{arithmetic} \} \]
\[ 0 < w - r < N \]

• Semaphore $m$ for exclusion is not needed!
• An array of size $N$, used in a circular manner suffices.
Putting it together

\[
\text{type buffer = record } \quad q \colon \text{queue of elem; } \\
\hspace{1em} s, t \colon \text{Semaphore} \\
\text{end;}
\]

\[
\text{type queue (elem) = record } \quad b \colon \text{array [0..N] of elem; } \\
\hspace{1em} r, w \colon \text{int} \\
\text{end;}
\]

\[
\text{proc Send (var b: buffer; x: elem) = } \\
\text{[ with b, q do P(s); b[w] := x; w := (w + 1) mod N; V(t) od ]}
\]

\[
\text{proc Receive (var b: buffer; var y: elem) = } \\
\text{[ with b, q do P(t); y := b[r]; r := (r + 1) mod N; V(s) od ]}
\]
Exercise

A.1 Consider the parallel execution of the three program fragments below.

\[
\begin{align*}
&\text{while true do A: } x := x+2 \text{ od} \\
&\text{while true do B: } y := y-1 \text{ od} \\
&\text{while true do C: } x := x-1; D: y := y+2 \text{ od}
\end{align*}
\]

Initially, \( x = y = 0 \)

Synchronize the system in order to maintain

\( I_0: 0 \leq y \)
\( I_1: x \leq 10 \)

Can you give an argument for absence of deadlock? Which additional restrictions might cause deadlock?
Exercise

A.2 Solve the *Vendor/Machine* problem.

- What to do if the assignments to *Stock* are not atomic?
- What if there are several *Vendors* and several *Machines* (both in case the assignments are and are not atomic)?
Exercises

A.3 Given are $N$ processes of the form

$$Pr_{(n, 0 \leq n < N)} = \textbf{while true do } X(n) \textbf{ od}$$

Here, $X(n)$ is a non-atomic program section that must be executed under exclusion. In addition, synchronize this system such that:

a. the sections are executed one after the other, in order:
   $$X(0); X(1); X(2); \ldots; X(N-1); X(0)\ldots$$

b. $X(i)$ is executed at least as often as $X(i+1)$, for $0 \leq i < N-1$.

In the solutions, first state appropriate synchronization conditions.
Exercises

A.4 Given is a collection of processes using system procedures $A_0$ and $A_1$. Synchronize the execution of these procedures such that exclusion is provided and that one execution of $A_0$ and two executions of $A_1$ alternate:

$$A_0;A_1;A_1;A_0;A_1;A_1 \ldots$$

• Is there any danger of deadlock?
• What about the fairness?
A.5 A collection of processes uses a collection of $K$ resources. For each resource there is an associated data structure, recorded in an array.

The processes repeatedly reserve and release resources using procedures $Reserve(i)$ and $Release(i)$. Through a call of $Reserve(i)$, variable $i$ is assigned the index of a free resource which is then claimed. This resource is subsequently released through $Release(i)$.

Write these two functions. Take care of exclusion on the array.

```plaintext
var Res: array [0..K-1] of
    record avail: bool;
    { other variables }
end

Proc Reserve (var i: int)
Proc Release (i: int)
```
A.7 Given are $N$ processes of the following form

\[
\text{Proc } \text{Philosopher} \ (n, 0 \leq n < N) = \\
\text{[} \text{while true do NonCriticalSection}(n); \\
\text{CriticalSection}(n) \\
\text{od} \\
\text{]} 
\]

The critical sections pertain to the use of two resources out of a total of $N$ resources; \textit{Philosopher}(n) uses resources number $n$ and $n+1$, with addition modulo $N$. Solve this problem. Discuss deadlock and fairness in particular.
Exercises

A.8 Consider the parallel execution of the three program fragments below.

\[
\text{while true do A0: } x := x + 2; \ A1: y := y - 1; \ A2: z := z - 1 \ \text{od}
\]

\[
\text{while true do B: } y := y + 2 \ \text{od}
\]

\[
\text{while true do C0: } z := z + 1; \ C1: x := x - 2 \ \text{od}
\]

Initially, \( x = y = z = 0 \)

Synchronize the system in order to maintain

\( I0: x + y + z \leq 10 \)
\( I1: y \leq 5 \)

The direct solution has danger of deadlock. Give a scenario. Can you repair it by additional restrictions?
Exercises

• **B.1** Suppose that a bounded buffer is to be shared by two producers. What must be changed?

• **B.2** Two consumers use the same bounded buffer. The first consumer needs 3 portions each turn and the second needs 4. Solve this problem (assuming first-come-first-serve) and answer the following questions:
  – Is waiting minimal? If not, can you imagine a situation that leads to a deadlock?
  – Does your solution work for a circular buffer of size 2?
  – Now make a general routine to retrieve \( n \) messages.
  – Specialize this solution for the case of a 1-place buffer.

**Note:** the behavior of the two consumers is their *given* behavior, you do not need to enforce that.
Exercises

• **B.3** N producers produce messages for one consumer. The messages must be handled exclusively, one by one. Producer *i* waits until the consumer has handled its message.
  1. Write programs for producers and consumer.
  2. Specialize your solution for the case of a buffer with just one single place.

• **B.4** Consider two processes. One process produces a whole video frame per cycle, the other consumes the frame sample by sample. There are *m* samples per frame. We have a two place buffer for the frames. The producer can only produce a frame when a place is available. Formalize this problem (write programs) and give a properly synchronized implementation of the two processes.
Summary: preventing deadlock

- The exercises A4, A7, A8, give the following insights for deadlock prevention

- Let critical sections terminate
  - in principle, no \( P \) operations between \( P(m)...V(m) \)

- Use a fixed order in \( P \)-operations on semaphores
  - \( P(m);P(n);...\) in one process may deadlock with \( P(n);P(m);...\) in another process
  - in fact: satisfy the synchronization conditions in a fixed order

- Beware of greedy consumers
  - Let \( P(a)^k \) be an indivisible operation when there is a danger of deadlock

In general: avoid cyclic waiting!

We come back to deadlock later.
Competing Vendors: semaphore $x$, $x_0 = 1$

**Proc Vendor =**

`\[\text{while true do}\]

  DriveToFactory;
  P(x); P(t)_{Load}; V(x);
  \{ Stock + Load \leq MAX \}
  P(m);
  Stock := Stock + Load;
  V(m);
  V(s)_{Load};
  DriveBack;
  ReLoad

\textbf{od}

\textbf{]}