Operating Systems 2010/2011

Action Synchronization

Johan Lukkien

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

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


```
Proc Vendor =
|[ while true do
        DriveToFactory;
        await (Stock+Load ≤ MAX);
        { Stock+Load ≤ MAX }
        Stock := Stock+Load;
        DriveBack;
        ReLoad
  od
]|
```

```
Proc Machine =
|[ while true do
        await (Stock>0);
        { Stock>0 }
        Stock := Stock-1;
        Manufacture
  od
]|
```

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

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Agenda

- Action synchronization
  - formalization
  - Semaphores
  - producer/consumer
- POSIX examples
- Action synchronization
  - mutual exclusion
  - bounded buffer

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

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 ∧ 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: $x=0 \wedge y=0$

while *true* **do A:** <$x := x+1$>; **B:** <$y := y+1$> **od**
||
while *true* **do C:** <$y := y-1$>; **D:** <$x := x-1$> **od**

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

A → <*A; cA := cA+1*>

Topology properties

Topology invariants: derived directly from the program text

Example: two actions always occurring one after the other

```
Initially: x=0 ∧ y=0

while true do A: <x := x+1>; B: <y := y+1> od
                    ||
while true do C: <y := y-1>; D: <x := x-1> od
```

Invariants:

$I0: x = \underline{c}A - \underline{c}D$ $\qquad$ $I2: 0 \le \underline{c}A - \underline{c}B \le 1$

$I1: y = \underline{c}B - \underline{c}C$ $\qquad$ $I3: 0 \le \underline{c}C - \underline{CD} \le 1$

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Example

Showing invariance of $I: y \leq x$

$y \leq x$

$= \quad \{ I0, I1 \}$

$\underline{c}B - \underline{c}C \leq \underline{c}A - \underline{c}D$

$= \quad \{ I2: \underline{c}B \leq \underline{c}A,\ I3: \underline{c}D \leq \underline{c}C \}$

true

Note: such a proof *must* refer somehow to topology because the property relies on it.

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

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;
    while true do
        A: <x := x+1>
    od
```

||

```
P_Y =
    y := 0;
    while true do
        B: <y := y+1>
    od
```

- Example: synchronize P_X and P_Y such that invariant

  $I0: x \leq y \qquad (= \underline{c}A \leq \underline{c}B)$

  is maintained.

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

The vendor-machine problem

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

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Semaphores (Dijkstra)

- Semaphore s is an integer *s* 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): < await(s>0); s := s-1 >

V(s): < s := s+1 >

- “< >” denotes again atomicity: the implementation of *P* and *V* must guarantee this
- ‘*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):

$$S0: s \geq 0$$

$$S1: s = s_0 + \underline{c}V(s) - \underline{c}P(s)$$

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

$$S2: \underline{c}P(s) \leq s_0 + \underline{c}V(s)$$

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

$P_X =$
  $x := 0;$
  while *true* **do**
    A: $\langle x := x+1 \rangle$
  od

||

$P_Y =$
  $y := 0;$
  while *true* **do**
    B: $\langle y := y+1 \rangle$
  od

Synchronize P_X and P_Y such that invariant

$I0: x \leq y$

is maintained.

Program topology

Use the program topology:

$x = \underline{c}A$ and $y = \underline{c}B$

hence, I0 can be rewritten

$I0: \underline{c}A \leq \underline{c}B$

Introduce semaphore *s*; let *A* be *preceded by P(s)* and *B* be *followed by V(s).*

Topology:

$I1: \underline{c}A \leq \underline{c}P(s)$

$I2: \underline{c}V(s) \leq \underline{c}B$

Combine with semaphore invariant *S4*:

$\underline{c}A \leq \underline{c}P(s) \leq s_0 + \underline{c}V(s) \leq s_0 + \underline{c}B$

Hence, choosing $s_0 = 0$ does the job.

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

More restrictions

Suppose that we also want:

$I3: y \le x+10$, i.e., $\underline{c}B \le \underline{c}A+10$

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

$$\underline{c}B \le \underline{c}P(t) \le t_0 + \underline{c}V(t) \le t_0 + \underline{c}A$$

Choose $t_0 = 10$.

$P_X =$		$P_Y =$
x := 0;	‖	*y := 0;*
while *true* **do**		**while** *true* **do**
P(s); **A**: <*x := x+1*>; *V(t)*		*P(t);* **B**: <*y := y+1*>; *V(s)*
od		**od**

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

And more...

Suppose that instead of *I0* we want

$I4: 2x \leq y$, i.e., $2\underline{c}A \leq \underline{c}B$

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

$2\underline{c}A \leq \underline{c}P(s)$

hence,

$2\underline{c}A \leq \underline{c}P(s) \leq s_0 + \underline{c}V(s) \leq s_0 + \underline{c}B$

etc....

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Action Synchronization

Given: - collection of tasks/threads executing actions *A, B, C, D*;
- a required *synchronization condition (invariant)*

$SYNC: \; a \cdot \underline{c}A + c \cdot \underline{c}C \leq b \cdot \underline{c}B + d \cdot \underline{c}D + e$

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

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

$A \rightarrow P(s)^a; A$ $\qquad$ $B \rightarrow B; V(s)^b$

$C \rightarrow P(s)^c; C$ $\qquad$ $D \rightarrow D; V(s)^d$

Note: during execution of *A* and *C* we have strict inequality in *SYNC*.

The vendor-machine problem

- Invariant:
  - *Stock = Load*c(Stock := Stock+Load) - c(Stock := Stock-1)*
- Synchronization condition:
  - 0 ≤ *Load*c(Stock := Stock+Load) - c(Stock := Stock-1) ≤ 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

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Synchronizing Vendor and Machine

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

not atomic
(needs another
semaphore)

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

Load times a *P(t)* operation
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

$I4: 2\underline{c}A \leq \underline{c}B$

$I3: \underline{c}B \leq \underline{c}A+10$

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

```
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)

```
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 */
```

Example

```
#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);
} }
```

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

(cnt'd)

```
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 Produce Consume Produce Produce Consume Produce Produce Consume Produce Consume Produce Consume Consume Consume 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

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Mutual exclusion

Given are *N* different process, repeatedly executing a *critical section*.

$Pr_{(n,\ 0 \leq n < N)}$ =
while *true* **do**
NonCriticalSection(n);
CsEntry(n);
CriticalSection(n);
CsExit(n)
od

Maintain as synchronization requirement

$M: (\Sigma\ n: 0 \leq n < N: \underline{c}CsEntry(n) - \underline{c}CsExit(n)) \leq 1$

Mutual exclusion (cnt'd)

Rewriting

$M: \Sigma\, \underline{c}CsEntry(n) \leq 1 + \Sigma\, \underline{c}CsExit(n)$

With action synchronization: introduce m, $m_0 = 1$.

$CsEntry(n) \quad \rightarrow P(m); CsEntry(n)$

$CsExit(n) \quad \rightarrow CsExit(n); V(m)$

(*CsEntry(n)/CsExit(n)* themselves can be "*skip*".)

Semaphore *m* is called a *binary semaphore* or a *mutex* as opposed to a *general semaphore* that can assume arbitrary non-negative values.

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Making assignments critical sections


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

```
Proc Machine =
|[ while true do
        P(s);
        { Stock>0 }
        P(m);
        Stock := Stock-1;
        V(m);
        V(t);
        Manufacture
   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).

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

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 *ex*clusion
- Restrictions
  - don't use copies of a mutex in the calls below
  - *lock()* and *unlock()* always by same thread ("ownership")

```
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) → pthread_mutex_lock

V(m) → pthread_mutex_unlock

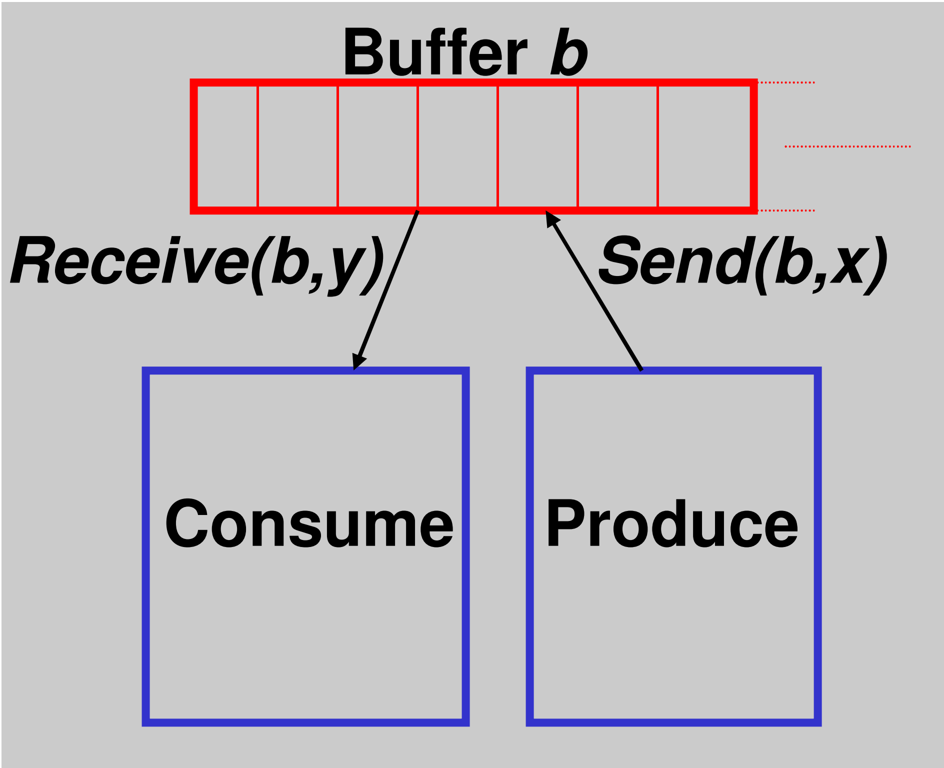
Agenda

- 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*.



Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

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 $\underline{c}GET(q,...) \leq \underline{c}PUT(q,...)$
  - Introduce semaphores t, $t_0 = 0$.
- The third requirement translates into $\underline{c}PUT(q,...) \leq \underline{c}GET(q,...) + N$
  - Introduce semaphore s, $s_0 = N$.

First solution

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

```
proc Send (var b: buffer; x: elem) =
|[ with b do
       P(s); P(m); PUT(q,x); V(m); V(t)
   od
]|
```

Notice the order of the *P*-operations: critical sections should always terminate

```
proc Receive (var b: buffer; var y: elem) =
|[ with b do
       P(t); P(m); GET(q,x); V(m); V(s)
   od
]|
```

Discussion

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

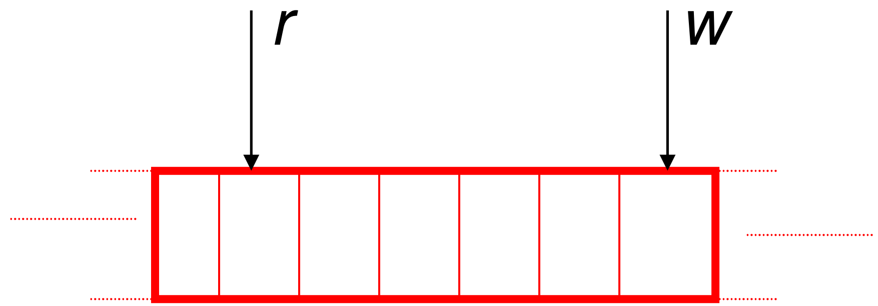
- **Absence of deadlock** is due to the fact that the critical sections (i.e., the statements between *P(m)* and *V(m)*) terminate; any permanent blocking must therefore be on the synchronization semaphores. The implementation *does not introduce* deadlock.

- 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).

```
type queue =
record   b: array of elem;
         r, w: int
end;
```




```
proc PUT (var q: queue; x: elem) =
|[  with q do
        { w = cPUT(q,...) }
        b[w] := x; w := w+1
    od ]|
```

```
proc GET (var q: queue; var y: elem) =
|[  with q do
        { r = cGET(q,...) }
        y := b[r]; r := r+1
    od ]|
```

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

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 "$y := b[r]$"

$\Rightarrow$ { use the program text + action synchronization: strict unequality }

$w = \underline{c}PUT(q,..) < \underline{c}GET(q,..)+N \ \wedge$

$r = \underline{c}GET(q,..) < \underline{c}PUT(q,..)$

$\Rightarrow$ { 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

type *buffer* =
record *q: queue* **of** *elem*;
s, t: Semaphore
end;

type *queue* (*elem*) =
record *b*: **array** [*0..N*) **of** *elem*;
r, w: int
end;

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

proc *Receive* (**var** *b: buffer*; **var** *y: elem*) =
|[**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.

while *true* **do A**: $x := x+2$ **od**
while *true* **do B**: $y := y-1$ **od**
while *true* **do C**: $x := x-1$; **D**: $y := y+2$ **od**

Initially, $x = y = 0$

Synchronize the system in order to maintain

$I0: 0 \leq y$

$I1: x \leq 10$

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

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

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); ....; X(N-1); X(0)...

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

In the solutions, first state appropriate synchronization conditions.

Exercises

A.4 Given is a collection of processes using system procedures *A0* and *A1*. Synchronize the execution of these procedures such that exclusion is provided and that one execution of *A0* and two executions of *A1* alternate:

A0; A1; A1; A0; A1; A1 …

- Is there any danger of deadlock?
- What about the fairness?

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Exercises

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.

Proc *Reserve* (**var** *i*: *int*)
Proc *Release* (*i*: *int*)

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

Exercises

A.7 Given are *N* processes of the following form

```
Proc Philosopher (n, 0 ≤ n < N) =
|[ while true do NonCriticalSection(n);
                 CriticalSection(n)
   od
]|
```

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

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Exercises

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

while *true* **do A0**: $x := x+2$; **A1**: $y := y-1$; **A2**: $z := z-1$ **od**

while *true* **do B**: $y := y+2$ **od**

while *true* **do C0**: $z := z+1$; **C1**: $x := x-2$ **od**

Initially, $x = y = z = 0$

Synchronize the system in order to maintain

I0: $x+y+z \le 10$
I1: $y \le 5$

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 indivisable 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* =
|[**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
od
]|

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