Concurrent Systems

Nebenläufige Systeme

VIII. Monitor

Wolfgang Schröder-Preikschat

November 27, 2014

Agenda

Preface

Fundamentals
- Mutual Exclusion
- Condition Variable
- Signalling Semantics

Implementation
- Data Structures
- Use Case
- Operations

Summary

Outline

Preface

Fundamentals
- Mutual Exclusion
- Condition Variable
- Signalling Semantics

Implementation
- Data Structures
- Use Case
- Operations

Summary

Subject Matter

- discussion on **abstract concepts** as to “a shared variable and the set of meaningful operations on it” [7, p. 121]:
  - monitor
    - a *language notation*, initially denoted by **critical region** [6, 7]
    - associates a set of procedures with a shared variable
    - enables a compiler to:
      - i check that only these procedures are carried out on that variable
      - ii ensure that the respective operations exclude each other in time
  - condition
    - one or more special variables that do “not have any stored value accessible to the program” [12, p. 550]
    - used to indicate and control a particular wait mode
    - for the respective process inside the monitor
- in functional terms, get to know “monitor” as fundamental means of synchronisation independent of linguistic features
  - explanation of various styles: Hansen, Hoare, Concurrent Pascal, Mesa
  - according to this, schematic representation of implementation variants
- demonstrate basic functions of a fictitious (language) run-time system

Motivation

- for all advantages, semaphores are to be approached with caution:
  - too low level, programmers must keep track of all calls to *P* and *V*
  - although different, used for both uni- and multilateral synchronisation
- out of it, various design and languages concepts originated:
  - secretary
    - idea for structuring control of sharing [5, p. 135–136]
  - critical region
    - **mutual exclusive** use of a shared variable [6]
  - event variable
    - a shared variable associated with an **event queue** [6]
  - path expressions
    - synchronisation rules within type definitions [2]
  - monitor
    - **class-like** synchronised data type [7, 12, 14]
    - inspired by SIMULA 67 [4, 3]
    - first implemented in Concurrent Pascal [9]
    - comes in a characteristic of many kinds [1, 10]
- however, the concept is beyond a programming-language construct
  - it is fundamental for system programming and system-level operation

Hint (Monitor [7, p. 121])

The purpose of a monitor is to control the scheduling of resources among individual processes according to a certain policy.

Outline

Preface

Fundamentals
- Mutual Exclusion
- Condition Variable
- Signalling Semantics

Implementation
- Data Structures
- Use Case
- Operations

Summary

Class Concept Expanded by Coordination

- key aspect is to facilitate solely <u>indirect access</u> to shared variables by means of **monitor procedures**
  - by definition, these procedures have to execute by **mutual exclusion**
    - on behalf of the calling process, the procedure prologue applies for exclusive occupation of the monitor ⇝ *lockout* simultaneous processes
    - on behalf of the occupying process, at return the procedure epilogue releases the monitor again ⇝ *proceed* locked processes, if any
  - usually, a compiler is in charge of ejecting the procedure pro- and epilogue
    - only infinite loops or hardware failures may prevent epilogue execution
    - only constructs beyond the **frame of reference** may force abnormality[1]
  - in logical respect, deadlocks due to programmed absence of unblocking of critical sections are impossible
- accordingly, instructions for synchronisation are cross-cutting concern of the monitor and no longer of the whole non-sequential program
  - particularly, instructions to protect critical sections are not made explicit
  - given that foreign-language synchronisation primitives cannot be used[1]

[1] Thinking of a multi-language system.

Intentional Process Delay

- multilateral (blocking) synchronisation is implicit basis of a monitor, but **unilateral synchronisation** needs to be made explicit
  - Hansen
    - proposed to attach a shared variable to an *event* [6, p. 577]
    - with *cause* and *await* as intrinsic functions for event signalling
  - Hoare
    - proposed a non-attached *condition variable* [12, p. 550]
    - with *wait* and *signal* as intrinsic functions for condition handling
- in operating-system terms, per variable an **event queue** of processes waiting by reason of a certain condition
  - sticking point is how the event queue is being acted upon:
    - Hansen
      - <u>all</u> processes <u>can</u> be transferred to the monitor waitlist (*cause*)
      - suggests that the former take priority over the latter [7, p. 118]
      - remodels his idea to a *single-process waitlist* [8, 9]: <u>all</u> ≡ <u>one</u>
    - Hoare
      - exactly <u>one</u> out of the waiting processes <u>is</u> selected (*signal*)
      - decrees that the chosen one is immediately resumed [12, p. 550]
  - but signalling is non-effective (void) if no process would be waiting on it
- in this spirit, the **signalling convention** makes the wide difference and affects structuring of monitor-based non-sequential programs [13]

Signalling Conventions

cf. [13]

- explicit signal operation assumed, **signal-and-ϕ**, with ϕ indicating the behaviour of the signalling process as follows:
  - wait
    - join monitor **entrance queue** and leave the monitor
    - resume all signalled processes (one at a time)
    - re-enter the monitor, compete against all processes
  - urgent wait
    - join **preferential queue** and leave the monitor
    - resume one signalled process (first come, first served)
    - re-enter the monitor, enjoy priority over entrant processes
  - return
    - leave the monitor and resume the single signalled process
  - continue
    - carry on holding the monitor, keep inside the procedure
    - resume all signalled processes (one at a time) at return
- in case of absence of a signal primitive, signalling may still happen:
  - automatic
    - leave the monitor and re-evaluate waiting conditions
    - if so, resume no longer waiting processes (one at a time)
- a main issue is the **control transfer** between signaller and signallee

Waiting inside a monitor

Without leaving the monitor, another process is unable to signal.

Atomicity of Control Transfer

- consequence for the **ownership structure** of monitor and signaller:
  - change
    - signal and wait, urgent wait, or return
  - keep
    - signal and continue or automatic signalling
- with an **indivisible change** in ownership a signallee has guarantee on the still effective invalidation of its waiting condition:
  - wait
    - only for one out of possibly many signalled processes
    - if applicable, the order of process resumption is undefined
    - a resumed signallee may change the condition for the others
    - makes re-evaluation of the waiting condition necessary
    - ↪ `while (!condition)`, wait: tolerant to false signalisation
  - urgent wait
    - exactly for the single signalled process
    - by definition, the process to be resumed is predetermined
    - no other process can re-establish the waiting condition
    - makes re-evaluation of the waiting condition unnecessary
    - ↪ `if (!condition)`, wait: intolerant to false signalisation
  - return
    - *ditto*
- keeping ownership by the signaller means fewer context switches and, thus, less background noise but higher (signal) allocation latency

Outline

Preface

Fundamentals
- Mutual Exclusion
- Condition Variable
- Signalling Semantics

Implementation
- Data Structures
- Use Case
- Operations

Summary

Fundamental Data Types

```
typedef struct monitor {
    semaphore_t mutex;  /* initial {1} */
#ifdef __FAME_MONITOR_SIGNAL_URGENT_WAIT__
    lineup_t urgent;    /* urgent waiting signallers */
#endif
} monitor_t;

typedef struct condition {
    monitor_t *guard;   /* enclosing monitor */
    lineup_t event;     /* signal-awaiting processes */
} condition_t;
```

- data type used for keeping track of **waiting processes** (cf. p. 18):

```
typedef struct lineup {
    int count;          /* number of waiting processes */
    event_t crowd;      /* wait-for event */
} lineup_t;
```

Monitor Procedures


```
extern void lockout(monitor_t*);   /* enter monitor */
extern void proceed(monitor_t*);   /* leave monitor */

extern void watch(condition_t*);   /* wait on signal */
extern void spark(condition_t*);   /* signal condition */
```

- consider these operations an additional **run-time system** element for a compiler of a "concurrent C-like" programming language
  - calls to *lockout* and *proceed* will be automatically generated as part of the pro- and epilogue of the respective monitor procedure
  - similarly, calls to *watch* and *spark* will be generated for the corresponding applications of condition variables
  - in addition, instances of type *monitor* and *condition* will be automatically ejected, too, by the code generation process of such a compiler
- further improvements [12, p. 551] are imaginable to also better reflect the different signalling semantics

Consolidating Example I

- a bounded buffer is controlled by a **pair** of condition variables:

```
#include "monitor.h"

#define BUF_SIZE 80

typedef struct buffer {
    condition_t space;        /* control of reusables */
    condition_t data;         /* control of consumables */
    char store[BUF_SIZE];     /* reusable resource */
    unsigned in, out;         /* store housekeeping */
    unsigned count;           /* wait/signal condition */
} buffer_t;
```

- instantiation of the necessary monitor and condition variables:

```
static monitor_t storehouse = {1};   /* monitor is free */
static buffer_t buffer = {           /* actual buffer */
    {&storehouse}, {&storehouse}     /* link to monitor */
};
```


Consolidating Example II

- handmade monitor procedure to put one item into the buffer:

```
1  void put(char item) {
2      lockout(&storehouse);    /* procedure prologue */
3      {
4          while (buffer.count == BUF_SIZE)
5              watch(&buffer.space);
6
7          buffer.store[buffer.in] = item;
8          buffer.in = (buffer.in + 1) % BUF_SIZE;
9          buffer.count += 1;
10
11         spark(&buffer.data);
12     }
13     proceed(&storehouse);    /* procedure epilogue */
14 }
```

- 2–3 monitor **entrance**, usually to be generated by a compiler
- 4–11 **body** of monitor procedure, to be programmed by a human
- 12–13 monitor **exit**, usually to be generated by a compiler

Consolidating Example III

- handmade monitor procedure to get one item out of the buffer:

```
char get() {
    char item;

    lockout(&storehouse);    /* procedure prologue */
    {
        while (buffer.count == 0) watch(&buffer.data);

        item = buffer.store[buffer.out];
        buffer.out = (buffer.out + 1) % BUF_SIZE;
        buffer.count -= 1;

        spark(&buffer.space);
    }
    proceed(&storehouse);    /* procedure epilogue */

    return item;
}
```

- monitor entrance and exit and body of monitor procedure as before

Event Queue — Classical Technique for Monitor Implementation

- a classic monitor implementation on **event queue** basis is considered:

```
typedef struct event { } event_t;;

extern void catch(event_t*);     /* expect event */
extern int  coast();             /* wait for event */
extern int  await(event_t*);     /* catch & coast */
extern int  cause(event_t*);     /* signal event */
```

catch
- makes the process unsusceptible against **lost wakeup**:
  - i non-effective in case of cooperative scheduling, otherwise
  - ii inhibits preemption or dispatching (SMP), resp., or
  - iii notifies event sensibility to potential signalers (*cause*)
- ensures that a process in running state is detectable by *cause*

coast
- if the process was not yet detected by *cause*, blocks on the event
- otherwise, clears the catch state and keeps the process running

await
- blocks the process on the specified event (i.e., signalled by *cause*)

cause
- unblocks processes (tentatively) waiting on the specified event

- based on this abstraction, **waitlist operations** can be composed next

Waitlist Operations I — Control of Blocking

```
inline void brace(lineup_t *this) {
    this->count++;              /* one more delaying */
    catch(&this->crowd);        /* ready to block/continue */
}

inline void shift(lineup_t *this) {
    coast();                    /* conditionally block */
    this->count--;              /* one less delaying */
}

inline void defer(lineup_t *this) {
    this->count++;              /* one more delaying */
    await(&this->crowd);        /* unconditionally block */
    this->count--;              /* one less delaying */
}

inline int level(lineup_t *this) {
    return this->count;         /* number delayed procs. */
}
```


Waitlist Operations II — Control of Unblocking

```
inline int avail(lineup_t *this) {
    if (this->count > 0)               /* any delayed? */
        cause(&this->crowd);           /* yes, unblock */
    return this->count;
}

inline int evoke(lineup_t *this) {
    int count = this->count;           /* save state */
    if (count > 0)                     /* any delayed? */
        admit(elect(&this->crowd)); /* yes, seize CPU */
    return count;
}
```

- note that *evoke* forces a process switch within a still locked monitor
  - as the case may be, the resuming process then unlocks the monitor
  - consequently, the monitor should not be protected by a **mutex** object
- thereto, a cut-through to basic **process management** is appropriate:

elect
- selects the next process, if any, from the specified waitlist

admit
- books the current process (signaller) "ready to run" <u>and</u>
- makes the elected process (signallee) available to the processor

Signalling Semantics

- as has been foreshadowed by a **configuration option** (cf. p. 12):
  - signal and continue — Mesa-style [14]
  - signal and return — Hansen-style as to Concurrent Pascal [8, 9]
  - signal and wait — Hansen-style as originally proposed [7]
  - signal and urgent wait — Hoare-style [12]
- some reflect **improvements** as proposed by Hoare [12, p. 551, 1.–4.]
  - starting point was the strict approach of *signal and urgent wait* monitor
  - here, the discussion is in the order as to increasing complexity/overhead
- as indicated by the data type (cf. p. 12), the designs presented next are typical for an approach using **event queues**
  - note that signalling is non-effective if no process is waiting on it (cf. p. 8)
  - this excludes the use of semaphores, as *V* leaves a signal trace
    - *V* always has an effect: at least it increases the semaphore value
- lightweight and efficient monitor operation benefits from **cross-layer optimisation** in constructive means
  - from language- to system-level run-time system to operating system

Signal and Continue

```
void lockout(monitor_t *this) { P(&this->mutex); }

void proceed(monitor_t *this) { V(&this->mutex); }

void watch(condition_t *this) {
    brace(&this-event);          /* prepare to release */
    proceed(&this->guard);       /* release monitor */
    shift(&this->event);         /* release processor */
}

void spark(condition_t *this) {
    avail(&this->event);         /* try signal process */
}
```

- as *watch* needs to release the monitor before releasing the processor, a potential **race condition** must be prevented
  - *brace* notifies upcoming blocking of the current process to the system
  - this is to assure the current process of progress guarantee as soon as the monitor was released and another process is enabled to *spark* a signal

Signal and Return

```
void lockout(monitor_t *this) { P(&this->mutex); }

void proceed(monitor_t *this) { V(&this->mutex); }

void watch(condition_t *this) {
    brace(&this-event);          /* prepare to release */
    proceed(this->guard);        /* release monitor */
    shift(&this->event);         /* release processor */
}

void spark(condition_t *this) {
    if (!avail(&this->event))    /* no watcher waiting? */
        proceed(this->guard);    /* release monitor */
}
```

- calling *spark* must be the **final action** within a monitor procedure
  - similar to the *continue* statement of Concurrent Pascal [9, p. 205]
- otherwise, the signaller could proceed inside an unlocked monitor if no signallee was detected

Signal and Wait

Combined Monitor Waitlist

```
1  void lockout(monitor_t *this) { P(&this->mutex); }
2
3  void proceed(monitor_t *this) { V(&this->mutex); }
4
5  void watch(condition_t *this) {
6      brace(&this->event);         /* prepare to release */
7      proceed(this->guard);        /* release monitor */
8      shift(&this->event);         /* release processor */
9  }
10
11 void spark(condition_t *this) {
12     if (evoke(&this->event))     /* signallee done! */
13         lockout(this->guard);    /* re-enter monitor */
14 }
```

- as the case may be, the signaller blocks on a condition variable:
  - 12 in case of a pending signallee, the signaller interrupts execution
    - a process switch inside the looked monitor takes place (cf. p. 19)
    - in the further course, another process unlocks/releases the monitor
  - 13 accordingly, the signaller must make sure to **relock** the monitor

Signal and Urgent Wait I

Monitor Entrance/Exit

```
void lockout(monitor_t *this) { P(&this->mutex); }

void proceed(monitor_t *this) {
    if (!avail(&this->urgent))   /* no urgent waiting */
        V(&this->mutex);         /* release monitor */
}
```

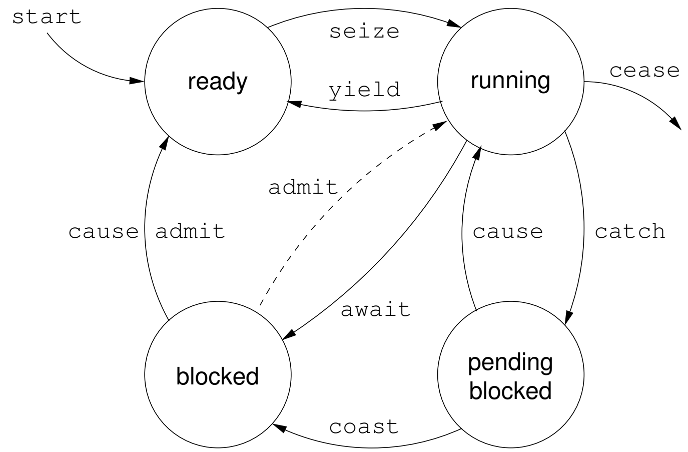
- in contrast to the solutions discussed before, **exit** from the monitor needs to check two waitlists for pending processes:
  - i the re-entrance waitlist (*urgent*), but only in case of urgent processes
  - ii the entrance waitlist (*mutex*), else
- by definition, urgent processes interrupted own operation in favour of processes pending for *event* handling
  - urgent processes caused events, recently, and want be resumed, expressly
- indicator of urgent waiting processes is a counter by means of which the number of process blockings is registered

Signal and Urgent Wait II


```
1  void watch(condition_t *this) {
2      brace(&this->event);          /* prepare to release */
3      proceed(this->guard);         /* release monitor */
4      shift(&this->event);          /* release processor */
5  }
6
7  void spark(condition_t *this) {
8      if (avail(&this->event))      /* watcher waiting? */
9          defer(&this->guard->urgent);    /* urgent wait */
10 }
```

- as the case may be, *spark* makes the current process urgent waiting
  - a **preferential queue** (Ger. *Vorzugswarteschlange*) is used to this end
  - *defer* results in a process switch from line 9 to line 4, back and forth
  - from *spark* to *shift*, out of *watch*, and back to *spark* at monitor exit
- urgent waiting processes keep *proceed* off from unlocking the monitor
  - when the monitor owner returns or blocks, an urgent process resumes
  - as a consequence, the monitor should not be protected by a **mutex**

Process States and State Transitions



- ready ↔ running: wait (←), scheduler (↔)
- running ↔ blocked: urgent wait (→), wait (←, iff *full preemptive*)
- blocked → ready: all, iff *effective signalling* (i.e., waiting signallee)
- running ↔ pending: all (→), signallee released monitor (←)
- pending → blocked: all, no overlap of signaller and signallee

Outline

Preface

Fundamentals
- Mutual Exclusion
- Condition Variable
- Signalling Semantics

Implementation
- Data Structures
- Use Case
- Operations

Summary

Résumé

- in linguistic terms, a monitor is a **language notation** for a critical region and one or more associated shared variables
  - a shared class [7, p. 226–232], inspired by SIMULA 67 [3]
  - linked with event queues [6] or condition variables [12], resp.
  - differentiated by several signalling semantics and conventions [13]
- in operating-system terms, a monitor is a means of **control** of the **scheduling** of resources among interacting processes
  - mutual-exclusive use of non-preemptable reusable resources
  - coordinated use of consumable resources according to a causal chain
- in system-pogramming terms, a monitor can be readily implemented by a **binary semaphore** and **event queues**
  - note that a **mutex** is to be rejected for the *signal and wait* variants

Hansen

> *In practice, monitors would, of course, be implemented by uninterruptible operations in assembly language. [11, p. 31]*

Reference List I

[1] Buhr, P. A. ; Fortier, M. :
Monitor Classification.
In: *ACM Computing Surveys* 27 (1995), März, Nr. 1, S. 63–107

[2] Campbell, R. H. ; Habermann, A. N.:
The Specification of Process Synchronization by Path Expressions.
In: Gelenbe, E. (Hrsg.) ; Kaiser, C. (Hrsg.): *Operating Systems: Proceedings of an International Symposium held at Rocquencourt, April 23–25, 1974*,
Springer-Verlag London, 1974 (Lecture Notes in Computer Science 16), S. 89–102

[3] Dahl, O.-J. ; Myhrhaug, B. ; Nygaard, K. :
SIMULA Information: Common Base Language / Norwegian Computing Center.
1970 (S-22). –
Forschungsbericht

[4] Dahl, O.-J. ; Nygaard, K. :
SIMULA—An ALGOL-Based Simulation Language.
In: *Communications of the ACM* 9 (1966), Sept., Nr. 9, S. 671–678

[5] Dijkstra, E. W.:
Hierarchical Ordering of Sequential Processes.
In: *Acta Informatica* 1 (1971), S. 115–138

Reference List II

[6] Hansen, P. B.:
Structured Multiprogramming.
In: *Communications of the ACM* 15 (1972), Jul., Nr. 7, S. 574–578

[7] Hansen, P. B.:
Operating System Principles.
Englewood Cliffs, N.J., USA : Prentice-Hall, Inc., 1973. –
ISBN 0–13–637843–9

[8] Hansen, P. B.:
A Programming Methodology for Operating System Design.
In: Rosenfeld, J. L. (Hrsg.) ; International Federation for Information Processing (IFIP) (Veranst.): *Information Processing 74: Proceedings of the IFIP Congress 74*.
Amsterdam : North-Holland, 1974, S. 394–397

[9] Hansen, P. B.:
The Programming Language Concurrent Pascal.
In: *IEEE Transactions on Software Engineering* SE-I (1975), Jun., Nr. 2, S. 199–207

Reference List III

[10] Hansen, P. B.:
Monitors and Concurrent Pascal: A Personal History.
In: Bergin, Jr., T. (Hrsg.) ; Gibson, Jr., R. G. (Hrsg.): *History of Programming Languages—II*.
New York, NY, USA : ACM, 1996. –
ISBN 0–201–89502–1, S. 121–172

[11] Hansen, P. B.:
The Invention of Concurrent Programming.
In: Hansen, P. B. (Hrsg.): *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*.
New York, NY, USA : Springer-Verlag New York, 2002. –
ISBN 0–387–95401–5, S. 3–61

[12] Hoare, C. A. R.:
Monitors: An Operating System Structuring Concept.
In: *Communications of the ACM* 17 (1974), Okt., Nr. 10, S. 549–557

[13] Howard, J. H.:
Signaling in Monitors.
In: Yeh, R. T. (Hrsg.) ; Ramamoorthy, C. V. (Hrsg.): *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*.
Los Alamitos, CA, USA : IEEE Computer Society Press, 1976, S. 47–52

Reference List IV

[14] Lampson, B. W. ; Redell, D. D.:
Experiences with Processes and Monitors in Mesa.
In: *Communications of the ACM* 23 (1980), Febr., Nr. 2, S. 105–117

[15] Schmidt, D. C.:
Strategized Locking, Thread-safe Interface, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components.
In: *C++ Report* 11 (1999), Sept., Nr. 9, S. 1–9

[16] Schröder-Preikschat, W. :
The Logical Design of Parallel Operating Systems.
Upper Saddle River, NJ, USA : Prentice Hall International, 1994. –
ISBN 0–13–183369–3

[17] Schröder-Preikschat, W. :
Processes.
In: Lehrstuhl Informatik 4 (Hrsg.): *Concurrent Systems*.
FAU Erlangen-Nürnberg, 2014 (Lecture Slides), Kapitel 3

Monitor Entry/Exit Revisited

- handmade monitor procedures are prone to absence of unblocking the monitor before return: *proceed* is missing or will never be executed
  - object constructors/destructors find a remedy [16, p. 220, Sec. 6.1.4]

```
class atomic {
    static monitor_t sluice;
public:
     atomic() { lockout(&sluice); };
    ~atomic() { proceed(&sluice); };
};
```

  - exit from the scope of an *atomic* instance implicitly performs *proceed*:

```
int64_t inc64(int64_t *i) {
    atomic inc; return *i + 1;
}
```

- a technique that is also known as the **scoped locking** pattern [15]