Concurrent Systems

Nebenläufige Systeme

VIII. Monitor

Wolfgang Schröder-Preikschat

December 11, 2018
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:
      1. check that only these procedures are carried out on that variable
      2. 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
key aspect is to facilitate solely indirect access 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 \(\rightsquigarrow\) **lockout** simultaneous processes
  - on behalf of the occupying process, at return the **procedure epilogue** releases
    the monitor again \(\rightsquigarrow\) **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

- 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

---

1Thinking 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** all processes can 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]: all \(\equiv\) one
    - **Hoare** exactly one out of the waiting processes is 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]
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 signallees

**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
### Fundamental Data Types

**C Code:**

```c
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-waiting processes */
} condition_t;

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

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

```c
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-waiting processes */
} condition_t;

typedef struct lineup {
    int count; /* number of waiting processes */
    event_t crowd; /* wait-for event */
} lineup_t;
```
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` are 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
a bounded buffer is controlled by a pair of condition variables:

```c
#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 = {
    &storehouse, &storehouse /* link to monitor */
};
```
handmade monitor procedure to put one item into the buffer:

```c
void put(char item) {
    lockout(&storehouse); /* procedure prologue */
    {
        while (buffer.count == BUF_SIZE)
            watch(&buffer.space);

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

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

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

```c
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

a classic monitor implementation on event queue basis is considered:

typedef struct event { } event_t;

catch (event_t*); /* expect event */
coast(); /* wait for event */
await(event_t*); /* catch & coast */
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 signallers (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 (signalled by cause)
cause ■ unblocks processes (tentatively) waiting on the specified event

based on this abstraction, waitlist operations can be composed next
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. */
}
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” and
  - makes the elected process (signalee) 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

```c
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
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

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 (evoke(&this->event)) /* signallees done! */
        lockout(this->guard); /* re-enter monitor */
}

as the case may be, the signaller blocks on a condition variable:

in case of a pending signallees, 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

accordingly, the signaller must make sure to relock the 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
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)) /* watcher waiting? */
        defer(&this->guard->urgent); /* urgent wait */
}

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 proceed with monitor locked (cf. p. 24)
- 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

- **start** → **ready**
- **ready** → **block**
- **block** → **ready**
- **ready ↔ running**
- **running ↔ block**
- **block** → **ready**
- **running ↔ pending**
- **pending** → **block**

- **ready ↔ running**: wait (←), scheduler (↔)
- **running ↔ block**: urgent wait (→), wait (←, iff full preemptive)
- **block ↔ ready**: all, iff effective signalling (i.e., waiting signalle)
- **running ↔ pending**: all (→), signalle released monitor (←)
- **pending → block**: all, no overlap of signaller and signalle
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-programming 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 un-interruptible operations in assembly language._ [11, p. 31]
Reference List I

[1] **Buhr, P. A.; Fortier, M.**
Monitor Classification.

[2] **Campbell, R. H.; Habermann, A. N.**
The Specification of Process Synchronization by Path Expressions.


SIMULA—An ALGOL-Based Simulation Language.
In: *Communications of the ACM* 9 (1966), Sept., Nr. 9, S. 671–678

Hierarchical Ordering of Sequential Processes.
[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.  

[8] Hansen, P. B.: 
A Programming Methodology for Operating System Design. 

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

The Invention of Concurrent Programming.  
New York, NY, USA : Springer-Verlag New York, 2002. –  

Monitors: An Operating System Structuring Concept.  

[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).  


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]

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

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

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

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