Concurrent Systems

Nebenläufige Systeme

VIII. Monitor

Wolfgang Schröder-Preikschat

January 8, 2020
Agenda

Preface

Fundamentals
  Mutual Exclusion
  Condition Variable
  Signalling Semantics

Implementation
  Data Structures
  Use Case
  Operations

Summary
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
- critical region
- event variable
- path expressions
- monitor
- idea for structuring control of sharing [5, p. 135–136]
- **mutual exclusive** use of a shared variable [6]
- a shared variable associated with an **event queue** [6]
- synchronisation rules within type definitions [2]
- **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]

yet, the subject matter is beyond programming-language constructs
- 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 $\sim lockout$ simultaneous processes
  - on behalf of the occupying process, at return the **procedure epilogue** releases the monitor again $\sim proceed$ 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\(^\dagger\)

- 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

\(^\dagger\text{Thinking of a multi-language system.}\)
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* 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].
Signalling Conventions

explicit signal operation assumed, signal-and-ψ, with ψ 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 signalled process 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 signalled process 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
Semaphore-based abstraction for **mutual exclusion** of monitor-procedure executions:

```
typedef struct monitor {
    semaphore_t mutex;  /* initial {1} */
#ifdef __FAME_MONITOR_SIGNAL_RETURN__
    bool elide;        /* leave monitor locked */
#endif
#ifdef __FAME_MONITOR_SIGNAL_URGENT_WAIT__
    lineup_t prime;    /* urgent waiting signallers */
#endif
} monitor_t;
```

- mandatory feature is a *binary semaphore*
- further attributes as optional feature, depending on *signalling semantics*

Data type used for keeping track of *waiting processes*:

```
typedef struct lineup {
    int count;      /* number of waiting processes */
    semaphore_t event;  /* wait-for event: initial {0} */
} lineup_t;
```
abstraction for **condition synchronisation** of interacting processes:

```c
typedef struct condition {
    monitor_t *guard;   /* surrounding monitor */
    lineup_t queue;     /* event awaiting list */
    #ifdef __FAME_MONITOR_SIGNAL_WAIT__
    lineup_t prime;     /* urgent waiting signallers */
    #endif
} condition_t;
```

- mandatory features are:
  - a suitable ink to the surrounding monitor object
  - a queue for processes expecting cancellation of their waiting condition

- further attributes as optional feature, depending on **signalling semantics**

- a condition variable is usually required for each waiting condition
  - their definition is part of the non-sequential program
  - as well as the typically problem-specific formulation of this condition
consider these operations an additional **run-time system** element for a compiler of a “concurrent C-like” programming language

- calls to `occupy` and `vacate` will be automatically generated as part of the pro- and epilogue of the respective monitor procedure
- similarly, calls to `comply` and `cancel` 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
Consolidating Example I

Bounded Buffer

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

1. `#include "monitor.h"
2. 
3. `#define BUF_SIZE 80`
4. 
5. `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;`
6. 

Instantiation of the necessary monitor and condition variables:

1. `static monitor_t storehouse = {1}; /* monitor is free */`
2. 
3. `static buffer_t buffer = {
   {&storehouse}, {&storehouse} /* link to monitor */
};`
handmade monitor procedure to put one item into the buffer:

```c
void put(char item) {
    occupy(&storehouse);  /* procedure prologue */
    {
        while (buffer.count == BUF_SIZE)
            comply(&buffer.space);  /* await event */

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

        cancel(&buffer.data);  /* cause event */
    }
    vacate(&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;

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

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

        cancel(&buffer.space);
    }
    vacate(&storehouse);    /* procedure epilogue */

    return item;
}

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

- note that signalling is non-effective if no process is waiting on it (cf. p. 8)
- this requires caution when using 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 occupy(monitor_t *this) { P(&this->mutex); }

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

void comply(condition_t *this) {
    this->queue.count ++;
    /* sign-in process */
    vacate(this->guard);
    /* release monitor */
    P(&this->queue.event);
    /* delay process */
    occupy(this->guard);
    /* re-acquire monitor */
    this->queue.count --;
    /* sign-out process */
}

void cancel(condition_t *this) {
    if (this->queue.count > 0) /* any registered? */
        V(&this->queue.event); /* continue one */
}

as comply needs to release the monitor before delaying the process, a potential race condition must be prevented
- still within the monitor, accounting for registered processes
Signal and Return

 Combined Final Action

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

void vacate(monitor_t *this) {
    if (this->elide) this->elide = false;
    else V(&this->mutex);
}

void comply(condition_t *this) {
    this->queue.count ++; /* sign-in process */
    vacate(this->guard); /* release monitor */
    P(&this->queue.event); /* delay process */
    this->queue.count --; /* sign-out process */
}

void cancel(condition_t *this) {
    if (this->queue.count > 0) { /* any registered? */
        this->elide = true; /* leave locked */
        V(&this->queue.event); /* continue complier */
    }
}
```
void occupy(monitor_t *this) { P(&this->mutex); }

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

void comply(condition_t *this) {
    this->queue.count++;
    /* sign-in process */
    vacate(this->guard);
    /* release monitor */
    P(&this->queue.event);
    /* delay process */
    this->queue.count--;
    /* sign-out process */
    V(&this->prime.event);
    /* urgent continue */
}

void cancel(condition_t *this) {
    if (this->queue.count > 0) {
        /* any registered? */
        V(&this->queue.event);
        /* continue one */
        P(&this->prime.event);
        /* urgent delay */
        occupy(this->guard);
        /* re-acquire monitor */
    }
}

© wosch, thoenig CS (WS 2019/20, LEC 8) Implementation – Operations
void occupy(monitor_t *this) { P(&this->mutex); }

void vacate(monitor_t *this) {
    if (this->prime.count > 0) /* urgent waiting? */
        V(&this->prime.event); /* yes, continue that */
    else
        V(&this->mutex); /* no, 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 (prime), 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
as the case may be, comply makes the current process urgent waiting.

- a **preferential queue** (Ger. *Vorzugswarteschlange*) is used to this end.

- urgent waiting processes proceed with monitor locked (cf. p. 22)
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


[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, Jr., T. (Hrsg.) ; Gibson, Jr., R. G. (Hrsg.): History of Programming Languages—II.

The Invention of Concurrent Programming.

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() { occupy(&sluice); };
    ~atomic() { vacate(&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]
abstractions for **mutual exclusion** of monitor-procedure executions and for **condition synchronisation** of interacting processes

- both remain syntactically identical, but not semantically
- in the given example they are reused (cf. p. 12)
  - here, however, without forced long jumps to “signal and return”
  - a certain programming convention is adopted instead (cf. p. 36)
- the main change is the list of waiting processes...

**data type used for keeping track of waiting processes:**

```c
typedef struct lineup {
  int count; /* number of waiting processes */
  event_t event; /* wait-for event */
} lineup_t;
```
a classic monitor implementation on **event queue** basis is considered:

```c
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 signallers (**cause**)

**coast**  ■ ensures that a process in running state is detectable by **cause**

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
waitlist Operations I

Control of Blocking

```c
inline void brace(lineup_t *this) {
    this->count ++;                /* one more delaying */
    catch(&this->event);           /* 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->event);          /* unconditionally block */
    this->count --;               /* one less delaying */
}

inline int level(lineup_t *this) {
    return this->count;           /* number delayed procs. */
}
```
```c
inline int avail(lineup_t *this) {
    if (this->count > 0) /* any delayed? */
        cause(&this->event); /* 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->event)); /* 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

thereeto, 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 (signallee) available to the processor
Signal and Continue

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

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

void comply(condition_t *this) {
    brace(&this->queue);    /* prepare to release */
    vacate(this->guard);    /* release monitor */
    shift(&this->queue);    /* release processor */
}

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

as comply 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 signal
Signal and Return

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

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

void comply(condition_t *this) {
    brace(&this->queue);  /* prepare to release */
    vacate(this->guard);  /* release monitor */
    shift(&this->queue);  /* release processor */
}

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

calling `cancel` 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
void occupy(monitor_t *this) { P(&this->mutex); }

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

void comply(condition_t *this) {
    brace(&this->queue); /* prepare to release */
    vacate(this->guard); /* release monitor */
    shift(&this->queue); /* release processor */
}

void cancel(condition_t *this) {
    if (evoke(&this->queue)) /* signallee done! */
        occupy(this->guard); /* re-enter monitor */
}

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 locked monitor takes place (cf. p. 34)
13 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 (**prime**), 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
as the case may be, cancel 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 cancel to shift, out of comply, and back to cancel at monitor exit

urgent waiting processes proceed with monitor locked (cf. p. 38)

- 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

cf. [17, p. 27]

ready ↔ running
running ↔ blocked
blocked → ready
running ↔ pending
pending → blocked

- 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 signallees