Concurrent Systems

*Nebenläufige Systeme*

VIII. Monitor

Wolfgang Schröder-Preikschat

January 7, 2021
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
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
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
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

- **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
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

**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

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
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

- **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

in functional terms, get to know “monitor” as fundamental means of synchronisation independent of linguistic features

demonstrate basic functions of a fictitious (language) run-time system
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
Motivation

for all advantages, semaphores are to be approached with caution

out of it, various design and languages concepts originated:

- secretary
- critical region
- event variable
- path expressions

- 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]
for all advantages, semaphores are to be approached with caution

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]
Motivation

for all advantages, semaphores are to be approached with caution

out of it, various design and languages concepts originated

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*
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 $\leadsto$ *lockout* simultaneous processes
  - on behalf of the occupying process, at return the procedure epilogue releases the monitor again $\leadsto$ *proceed* locked processes, if any
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

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

\(^1\)Thinking of a multi-language system.
Class Concept Expanded by Coordination

- 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**

- usually, a compiler is in charge of ejecting the procedure pro- and epilogue

- in logical respect, deadlocks due to programmed absence of unblocking of critical sections are impossible

© wosch  CS (WS 2020/21, LEC 8)  Fundamentals – Mutual Exclusion
key aspect is to facilitate solely indirect access to shared variables by means of monitor procedures.

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.
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 \[6, \text{p.577}\] with cause and await as intrinsic functions for event signalling. Hoare proposed a non-attached condition variable \[12, \text{p.550}\] 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.

A sticking point is how the event queue is being acted upon. Hansen suggests that all processes can be transferred to the monitor waitlist (cause) \[7, \text{p.118}\] but actually all processes are considered as candidates. Hansen remodels his idea to a single-process waitlist \[8, 9\]: all processes are candidates but exactly one out of the waiting processes is selected (signal) \[12, \text{p.550}\], whereas 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 \[13\].
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
Intentional Process Delay

- multilateral (blocking) synchronisation is implicit basis of a monitor, but **unilateral synchronisation** needs to be made explicit

**Hoare**
- proposed a non-attached *condition variable* [12, p. 550]
- with *wait* and *signal* as intrinsic functions for condition handling
Intentional Process Delay

- Multilateral (blocking) synchronisation is implicit basis of a monitor, but **unilateral synchronisation** needs to be made explicit

- 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
Intentional Process Delay

- multilateral (blocking) synchronisation is implicit basis of a monitor, but **unilateral synchronisation** needs to be made explicit

- in operating-system terms, per variable an **event queue** of processes waiting by reason of a certain condition

**Hansen**  
- all processes can be transferred to the monitor waitlist (**cause**)  
- suggests that the former take priority over the latter [7, p. 118]
Intentional Process Delay

- multilateral (blocking) synchronisation is implicit basis of a monitor, but **unilateral synchronisation** needs to be made explicit

in operating-system terms, per variable an **event queue** of processes waiting by reason of a certain condition

**Hansen**
- all processes can be transferred to the monitor waitlist (**cause**)  
  - suggests that the former take priority over the latter [7, p. 118]  
  - remolds his idea to a **single-process waitlist** [8, 9]: all $\equiv$ one
Intentional Process Delay

- multilateral (blocking) synchronisation is implicit basis of a monitor, but **unilateral synchronisation** needs to be made explicit

- in operating-system terms, per variable an **event queue** of processes waiting by reason of a certain condition

**Hoare**

- exactly **one** out of the waiting processes **is selected** (*signal*)
- decrees that the chosen one is immediately resumed [12, p. 550]
Intentional Process Delay

- Multilateral (blocking) synchronisation is implicit basis of a monitor, but **unilateral synchronisation** needs to be made explicit.

- In operating-system terms, per variable an **event queue** of processes waiting by reason of a certain condition.

  **Hansen**  
  - All processes can be transferred to the monitor waitlist (**cause**).

  **Hoare**  
  - Exactly one out of the waiting processes is selected (**signal**).
  - But signalling is non-effective (void) if no process would be waiting on it.
Intentional Process Delay

- Multilateral (blocking) synchronisation is implicit basis of a monitor, but **unilateral synchronisation** needs to be made explicit.

- In operating-system terms, per variable an **event queue** of processes waiting by reason of a certain condition.

  - **Hansen** - all processes can be transferred to the monitor waitlist (**cause**).

  - **Hoare** - exactly one out of the waiting processes is selected (**signal**).

- 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
explicit signal operation assumed, **signal-and-**\(\phi\), with \(\phi\) 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

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)
Signalling Conventions

- explicit signal operation assumed, **signal-and-**$\phi$, with $\phi$ indicating the behaviour of the signalling process as follows:

  - **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

© wosch CS (WS 2020/21, LEC 8) Fundamentals – Signalling Semantics
Signalling Conventions

- explicit signal operation assumed, **signal-and-**\(\phi\), with \(\phi\) indicating the behaviour of the signalling process as follows:

  - return: leave the monitor and resume the single signalled process
Signalling Conventions

- explicit signal operation assumed, \textbf{signal-and-}\(\phi\), with \(\phi\) indicating the behaviour of the signalling process as follows:

  - \textbf{continue}  
    - carry on holding the monitor, keep inside the procedure  
    - resume all signalled processes (one at a time) at return
Signalling Conventions

- 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)
Signalling Conventions

- explicit signal operation assumed, **signal-and-ϕ**

- a main issue is the **control transfer** between signaller and signallee
Signalling Conventions

- explicit signal operation assumed, \textit{signal-and-}\phi

- a main issue is the \textbf{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

A resumed signaller may change the condition for the others, making re-evaluation of the waiting condition necessary.

- if \(!\text{condition}\), wait: intolerant to false signalisation

- if \(!\text{condition}\), wait: tolerant to false signalisation

Keeping ownership by the signaller means fewer context switches and, thus, less background noise but higher (signal) allocation latency.
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 signalle has guarantee on the still effective invalidation of its waiting condition
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 signalleee 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 signalleee may change the condition for the others
    - makes re-evaluation of the waiting condition necessary
    - **while (!condition), wait:** tolerant to false signalisation
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:

  **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**
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

- urgent wait
  - exactly for the single signalled process

- return
  - *ditto*
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

- keeping ownership by the signaller means fewer context switches and, thus, less background noise but higher (signal) allocation latency
abstraction for **mutual exclusion** of monitor-procedure executions:

```c
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*
Semaphore-based abstraction for mutual exclusion of monitor-procedure executions:

```c
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;
```

A mandatory feature is a binary semaphore further attributes as optional feature, depending on signalling semantics.

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

```c
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**
Fundamental Data Types II

Event signalling

1. `typedef struct condition {
   2.   monitor_t * guard ;
   3.   lineup_t queue ;
   4.   # ifdef __FAME_MONITOR_SIGNAL_WAIT__
   5.   lineup_t prime ;
   6.   # endif
   7. } 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 features, 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
Monitor Procedures

Functional Interface

1. `extern void occupy(monitor_t*); /* enter monitor */`
2. `extern void vacate(monitor_t*); /* leave monitor */`
3. `extern void comply(condition_t*); /* wait on signal */`
4. `extern void cancel(condition_t*); /* signal condition */`

Consider these operations an additional run-time system element for a compiler of a “concurrent C-like” programming language.
Monitor Procedures

Functional Interface

1. extern void occupy(monitor_t*);  /* enter monitor */
2. extern void vacate(monitor_t*);  /* leave monitor */
3.  
4. extern void comply(condition_t*);  /* wait on signal */
5. extern void cancel(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 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.
Monitor Procedures

Functional Interface

1. `extern void occupy(monitor_t*);` /* enter monitor */
2. `extern void vacate(monitor_t*);` /* leave monitor */
3. `extern void comply(condition_t*);` /* wait on signal */
4. `extern void cancel(condition_t*);` /* signal condition */

Consider these operations an additional **run-time system** element for a compiler of a “concurrent C-like” programming language.

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;
```
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 */
};
```
Consolidating Example II

Bounded-Buffer Fill

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
Consolidating Example III

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
  - signal and return
  - signal and wait
  - signal and urgent wait

  - Mesa-style [14]
  - Hansen-style as to Concurrent Pascal [8, 9]
  - Hansen-style as originally proposed [7]
  - Hoare-style [12]
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
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]

- 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
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]

- as indicated by the data type (cf. p. 12), the designs presented next are typical for an approach using **semaphores**

- lightweight and efficient monitor operation benefits from **cross-layer optimisation** in constructive means
  ■ from language- to system-level run-time system to operating system
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 */
}
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
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 */
    }
}

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 */
}
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
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.

- 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
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:

- By definition, urgent processes interrupted own operation in favour of processes pending for event handling.

- Indicator of urgent waiting processes is a counter by means of which the number of process blockings is registered.
```c
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->guard->prime.count++; /* sign-in urgent */
        V(&this->queue.event);   /* continue queued */
        P(&this->guard->prime.event); /* urgent wait */
        this->guard->prime.count--; /* sign-out urgent */
    }
}
```
Signal and Urgent Wait II

Condition Handling

```c
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->guard->prime.count ++; /* sign-in urgent */
        V(&this->queue.event);       /* continue queued */
        P(&this->guard->prime.event); /* urgent wait */
        this->guard->prime.count --; /* sign-out urgent */
    }
}
```

as the case may be, comply makes the current process urgent waiting

- a preferential queue (Ger. *Vorzugswarteschlange*) is used to this end
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->guard->prime.count ++; /* sign-in urgent */
        V(&this->queue.event); /* continue queued */
        P(&this->guard->prime.event); /* urgent wait */
        this->guard->prime.count --; /* sign-out urgent */
    }
}

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)
Résumé

[Text content not visible]
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]
Résumé

- in linguistic terms, a monitor is a **language notation** for a critical region and one or more associated shared variables

- 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
Résumé

- In linguistic terms, a monitor is a **language notation** for a critical region and one or more associated shared variables.

- In operating-system terms, a monitor is a means of **control** of the scheduling of resources among interacting processes.

- 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.
Résumé

- In linguistic terms, a monitor is a **language notation** for a critical region and one or more associated shared variables.

- In operating-system terms, a monitor is a means of **control** of the **scheduling** of resources among interacting processes.

- In system-programming terms, a monitor can be readily implemented by a **binary semaphore** and **event queues**.

**Hansen**

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

[5] Dijkstra, E. W.:
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.
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

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.
[14] Lampson, B. W.; Redell, D. D.:
Experiences with Processes and Monitors in Mesa.

[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.

[17] Schröder-Preikschat, W. :
Processes.
In: Lehrstuhl Informatik 4 (Hrsg.): Concurrent Systems.
FAU Erlangen-Nürnberg, 2014 (Lecture Slides), Kapitel 3
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
1 class atomic {
2     static monitor_t sluice;
3 public:
4     atomic() { occupy(&sluice); };
5     ~atomic() { vacate(&sluice); };
6 }
```

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

```cpp
1 int64_t inc64(int64_t *i) {
2     atomic inc; return *i + 1;
3 }
```

A technique that is also known as the `scoped locking` pattern [15]
Fundamental Data Types

- 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:

```
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 */
```
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**)

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

- non-effective in case of cooperative scheduling, otherwise
- inhibits preemption or dispatching (SMP), resp., or
- 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
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 ■ 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)
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 signallers (cause)

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

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

cause ■ unblocks processes (tentatively) waiting on the specified event
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 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
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. */
}
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;
}
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
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**

**therefore, 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 */
}
```
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 */
}
```
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]
Signal and Return

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)) /* signalleee done! */
        occupy(this->guard); /* re-enter monitor */
}
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)) /* signalleee done! */
        occupy (this->guard); /* re-enter monitor */
}

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

in case of a pending signalleee, the signaller interrupts execution
a process switch inside the locked monitor takes place (cf. p. 34)
in the further course, another process unlocks/releases the monitor
accordingly, the signaller must make sure to relock the monitor
void occupy(monitor_t *this) {
    P(&this->mutex);
}

void vacate(monitor_t *this) {
    if (!avail(&this->prime)) /* 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 (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.
void occupy(monitor_t *this) { P(&this->mutex); }

void vacate(monitor_t *this) {
    if (!avail(&this->prime)) /* 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 (prime), but only in case of urgent processes
  ii the entrance waitlist (mutex), else
void occupy(monitor_t *this) { P(&this->mutex); }

void vacate(monitor_t *this) {
    if (!avail(&this->prime))  /* 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 (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
void occupy(monitor_t *this) { P(&this->mutex); }

void vacate(monitor_t *this) {
    if (!avail(&this->prime)) /* 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 (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
```
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))    /* watcher waiting? */
        defer(&this->guard->prime);    /* urgent wait */
}
```
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))    /* watcher waiting? */
        defer(&this->guard->prime);    /* urgent wait */
}

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
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))  /* watcher waiting? */
    defer(&this->guard->prime); /* urgent wait */
}

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

- wait (←), scheduler (↔)
- urgent wait
- all, iff effective signalling (i.e., waiting signallee)
Process States and State Transitions

- **Ready** ↔ **Running**
  - wait (←), scheduler (↔)
  - urgent wait

- **Running** → **Blocked**
  - all, iff *effective signalling* (i.e., waiting signallees)

- **Blocked** → **Ready**
  - all (→), signallees released monitor (←)

- **Running** ↔ **Pending**
  - all, no overlap of signaller and signallees

- **Pending** → **Blocked**

---

cf. [17, p. 27]
Process States and State Transitions

cf. [17, p. 27]

- **ready ↔ running**
  - wait (←), scheduler (↔)
  - urgent wait (→), wait (←, iff full preemptive)
  - all, iff effective signalling (i.e., waiting signallee)

- **running ↔ blocked**
  - all (→), signallee released monitor (←)

- **blocked → ready**
  - all, no overlap of signaller and signallee

- **pending → blocked**

![Diagram of process states and state transitions]

© wosch  CS (WS 2020/21, LEC 8)  Addendum – Operations