Concurrent Systems

Nebenläufige Systeme

XII. Guarded Sections

Wolfgang Schröder-Preikschat

February 4, 2021

Subject Matter

discussion on abstract concepts as to structural measures suited in paving the way for non-blocking synchronisation

- guarded sections
- pre-/postlude sections

both approaches common is the fact that processes of whichever kind will never be blocked at entrance to a critical section

- however their requests to enter and pass through may be delayed
- an alternating sequencer takes care of retroactive request processing
- this constrains overlapping and, thus, eases non-blocking request queues
  - per sample of interrupt-transparent synchronisation [14], for instance

- similar to an explicit (“eventual values” [9, 10]) or implicit future [2], it is shown how to deal with “direct-result critical sections”
- using concepts such as the promise [7] or promise pipelining [12]
- functional programming meets distributed computing for synchronisation

one learns that guarded sections largely resemble conventional critical sections, but with a much more relaxed execution model

1Not to be confused with “guarded commands” [4].
Interrupt Handling

**Definition (Interrupt)**
Mechanism of a (soft- or hardware) processor to prompt software to draw attention to an external process asynchronously, unpredictably, and unreproducibly.

- a **sudden upcall** (acc. [3]) performed by a processor in the middle of or between actions, depending on the processor model
- a start of a simultaneous process on this very processor in **stacking** mode
- most notably, this process is characteristic of a **run-to-completion** flow
- as to operating systems, usually a **trinity** of problem-specific routines is to be considered—and assumed in the following:
  - **guardian**  
  - **prelude**  
  - **postlude**
- what all have in common is the **asynchronism** to the current process that was interrupted and will be delayed by their particular actions

Responsibility Assignment

**Hint (Interrupt Latency)**
*In order to make loss of interrupts improbable, CPU priority*\[^a\] must be cancelled and OS priority*\[^b\] must be taken in minimum time.*

\[^a\]*Interrupt requests of the same and lower priority are disabled.
\[^b\]*All interrupt requests are enabled.

- conceptually, prelude and postlude together constitute the interrupt handler to be dispatched due to an **interrupt request** (IRQ):
  - **guardian**  
    - identifies and activates the prelude for the given IRQ
    - in case of an **edge-triggered** IRQ, takes OS priority before it
    - in case of a **level-triggered** IRQ, takes OS priority afterwards
  - **prelude**  
    - operates and “unloads” the device to satisfy the IRQ source
    - starts immediately if enabled by the CPU priority
    - as the case may be, releases its postlude for post-processing
  - **postlude**  
    - operates the device, if still required, and particularly the system
    - starts when no more preludes are stacked and, thus, pending
    - as the case may be, interacts with a process instance

Relevance of Postlude

**Hint (Asynchronous System Trap, AST [11, p.414])**
*On the VAX, a software-initiated interrupt to a service routine. ASTs enable a process to be notified of the occurrence of a specific event asynchronously with respect to its execution. In 4.3 BSD, ASTs are used to initiate process rescheduling.*

- essentially, the interrupt handler postlude equates to such an AST
  - a mechanism that forces an interrupted process back into system mode:
    - 1. when no interrupt handler prelude is pending (i.e., stacked)
    - 2. in the moment when the interrupt handler guardian terminates (i.e., returns)
  - as if this very process performs a system call to the interrupt postlude
- caution is advised when an **interrupt-handler control flow** expands
  - **guardian**  
  - **prelude**  
  - **postlude**  
  - no risk of race conditions or system-stack overflow
- purpose of the postlude is to safely allow such control-flow expansions
  - its activation is controlled similar to the control of guarded sections
Execution Sequencing of Postludes

- heading for postlude execution depends on the particular prelude
  - a prelude is a function, its return value indicates the postlude to be run
  - a return value of NULL indicates that this prelude asks for no postlude
- according to the model, an interrupt indeed causes a new process but not a new process instance
  - the guardian is such a process, it operates in the name of the interrupted process instance and commands no own context
  - same applies for the sequencer, it is an optional guardian continuation and takes care for safe postlude processing

Guardian and Sequencer
From FLIH to SLIH (cf. p. 36ff.)

Guardian

```
void guardian(long irq) {
    static usher_t tube = { 0, (0, &tube.load.head) };
    extern remit_t *(flih[])(usher_t *);
    remit_t *task;
    #ifdef __FAME_INTERRUPT_EDGE_TRIGGERED__
    pivot(&tube.busy, +1); admit(IRQ);
    else
        /* activate prelude & satisfy IRQ source */
        task = (+flih[irq](&tube));
        /* prevent lost unload */
        /* forward pending postludes */
        remit(task);
        /* take OS priority */
        /* settle insertion point */
        /* find end of orig. list */
        if (task != 0) deter(&tube, task);
        /* enqueue postlude & */
        /* leave with CPU priority */
        /* take OS priority , again */
        /* terminate chain : FIFO */
        remit_lfs_ms(queue_t *this, chain_t *item) {
            chain_t *last;
            item->link = 0; /* terminate chain: FIFO */
            last = this->tail; /* settle insertion point */
            this->tail = item; /* create new partial list */
            while (last->link != 0) /* overlapping enqueue! */
                last = last->link; /* find end of orig. list */
            last->link = item; /* insert & combine lists */
        }
        /* create new partial list */
        /* forward pending postludes */
        /* take OS priority , again */
        /* leave with CPU priority */
        /* take OS priority */
        /* activate prelude & satisfy IRQ source */
```
Lock-Free Synchronised Dequeue

cf. [14]

```c
chain_t * dequeue_lfs_ms (queue_t *this) {
    chain_t *item;

    if (((item = this->head.link) /* next item fetched */
        && (!this->head.link = item->link)) {
        this->tail = &this->head; /* is last one, reset */
        if (item->link != 0) { /* overlapping enq.! */
            chain_t *help, *lost = item->link;
            do {
                /* recover latecomers */
                help = lost->link; /* remember next & */
                enqueue_lfs_ms(this, lost); /* rearrange */
            } while ((lost = help) != 0);
        }
    }

    return item;
}
```

**Hint (Lock Freedom)**

Some process will complete an operation in a finite number of steps, regardless of the relative execution speeds of the processes. [8, p. 142]

Critical is dequeuing as to the last element and overlapped by one or more enqueues, thus, filling up the queue again

One moment the fetched item was last, now latecomers must be recovered

---

Recapitulation

- In the pre-/postlude model, sequencer becomes that process in the context of which interrupt handling is carried out
  - More precisely, the process at the bottom of an interrupt-handler stack
  - Put differently, the interrupted process that “activated” the guard (p. 9)

**Hint (Pro-/Epilogue [15, 14])**

At first glance, interrupt handler pre-/postludes seemingly resemble the pro-/epilogue model. While this is quite true for preludes, it does not hold for postludes. Epilogue execution is a synchronous event as to the interrupted kernel-level process, in contrast to postludes — which are more comparable to AST handlers.

Postlude guide through is not unlike procedure chaining [13, p. 10], a technique to serialise execution of conflicting threads

- Differences are due to the constrained pre-/postlude overlapping pattern
- Unless stack-based scheduling [1], any process overlapping is assumed
- This similarity gives reason to think about a generalisation of the pre-/postlude model to synchronise process-instance events

---

Wait-Free Solution

Special Instructions

```c
void enqueue_wfs_ms(queue_t *this, chain_t *item) {
    chain_t *last;
    item->link = 0; /* terminate chain: FIFO */
    last = FAS(&this->tail, item);
    last->link = item; /* eventually append item */
}

chain_t * dequeue_wfs_ms(queue_t *this) {
    chain_t *item = this->head.link;
    if (item) {
        /* check for last item */
        if (item->link) /* is not, non-critical */
            this->head.link = item->link;
        else if (CAS(&this->tail, item, &this->head))
            CAS(&this->head.link, item, 0);
    }

    return item;
}
```

**Hint (Lock Freedom)**

With the following mapping to GCC atomic intrinsic functions:

```
#define FAS(ref, val) __sync_lock_test_and_set(ref, val)
#define CAS __sync_bool_compare_and_swap
```

---

Outline

- Preface
- Hardware Events
  - Fundamentals
  - Sequencing
  - Implementation
- Process Events
  - Fundamentals
  - Sequencing
  - Implementation
- Summary
Critical Sections Revisited

- assuming a stack represented as LIFO (last in, first out) single-linked list, whose push- and pop-operations need to be critical sections

```c
void push(lifo_t *list, chain_t *item) {
    acquire(&list->lock); /* enter critical section */
    item->link = list->link;
    list->link = item;
    release(&list->lock); /* leave critical section */
}
```

Processes proceed successively, **neither** depends on the computation result

```c
chain_t *pull(lifo_t *list) {
    chain_t *item;
    acquire(&list->lock); /* enter critical section */
    if ((item = list->link) != 0)
        list->link = item->link;
    release(&list->lock); /* leave critical section */
    return item;
}
```

Processes proceed successively, **each** depends on the computation result

---

Handling of a Critical-Section Function

- fall back on known **linguistic concepts** in order to pattern a solution for the above-mentioned problem:
  - **future**: the promise to deliver a value at some later point in time
  - **promise**: read-only placeholder object created for a not yet existing result
    - the result is computed concurrently and can be later collected
  - **kept**: traced back to [7], a writeable, single-assignment container
    - can be used to successfully complete a future with a value

Each future instance has a dedicated **resolver** taking care of (a) value assignment and (b) **promise states**:

- **kept**: value computed, assignment took place
- **broken**: computation aborted, assignment ceases to take place
- **pending**: process in progress, assignment did not just yet take place

- Based on these states, a process is able to synchronise on the **event** that the promise to deliver a value was either kept or broken
  - The resolver (process inside the critical section) acts as producer
  - The future using process acts as consumer

---

Conditional Fire-and-Forget Pattern

- Processes heading for passing through a critical section will proceed unstopped, though simultaneous **passage requests** are serialised
  - At the end of a critical section, these requests will be processed one at a time
  - Accordingly, the **exit protocol** does not have to take care of blocked processes but rather immediately incurred passage requests
  - The particular leaving process attends to handle accumulated entry calls
  - Thus, critical-section execution is **asynchronous** to its requesting process
  - In case of data dependencies as to the computation within a critical section, synchronisation on **result delivery** becomes necessary
  - Thereto, computation results need to be returned and accepted by proxy
  - To this end, the following measures have to be provided:
    - As additional element of the corresponding passage request, a placeholder for the computation result (**consumable resource**) and
    - A signalling mechanism to indicate result delivery (**logical synchronisation**)

- In the final analysis, critical sections are **twofold**, namely one that is procedure- and another one that is function-like
  - With the former delivering no direct result, in contrast to the latter

---

Execution Sequencing of Critical Sections

- Heading for a critical section depending on the **state of occupancy**:
  - **unoccupied**: guard grants requester access to the critical section
  - The critical section becomes occupied by the requester
  - **occupied**: guard denies requester access to the critical section
  - The request gets queued and the requester bypasses

- Leaving a critical section depending on the **request queue state**:
  - **empty**: critical section becomes unoccupied, the process continues
  - **full**: the actual leaving process becomes sequencer and re-enters the critical section for each queued request

---

Refined for **promise pipelining** [12] to overcome latency in distributed systems.
*Synchronisation of Direct-Result Critical Sections*

**Run-to-Stopover for Peer Processes**

- A passage request may refer to a multi-elementary future object:
  1. A promise indicator (kept, broken, pending)
  2. A placeholder of problem-specific type as to the critical section
  3. A binary semaphore that is used in producer/consumer mode
    - I.e., a signalling semaphore applicable by different processes
- In case of a direct-result critical section, the sequencer takes the part of a resolver that also have to signal the “kept” or “broken” state
  - V does the signalling and by means of P the signal can be consumed

**Execution Characteristics of the Critical Section**

- Critical sections controlled by processes in a run-to-completion style can be handled straightforwardly

**Definition (Run to Completion (Process))**

A potentially preemptive process free from self-induced wait states as to the possible non-availability of reusable or consumable resources.

- Processes will not await external events from inside the critical section
- Control of a run-to-stopover style of execution of a critical section depends on the locality of peer processes

**Definition (Run to Stopover (Process))**

A potentially preemptive process possibly being subject to wait states.

- Processes waiting on events caused by an external process (e.g., I/O)
- Processes interacting with an internal process due to resource sharing
- Both styles of execution concern the period of a critical section, only
  - But at large, a process may be classified run to completion and stopover

**Overlapping Pattern**

- Notably is the implication in terms of the constructive restriction of overlappings as to simultaneous requester and sequencer processes
  1. Requesters of any guarded section may overlap each other
  2. Self-overlapping of a sequencer is impossible
  3. Only sequencers of different guarded sections may overlap each other
- Regarding the whole request processing chain and the involvement of requester and sequencer process one may realise:
  - Multiple requester may enqueue passage requests possibly simultaneously, but they will never dequeue these
  - A single sequencer only dequeues passage requests, but this may happen simultaneously to enqueues of one or more requesters
- This multiple-enqueue/single-dequeue mode of operation eases the design of a non-blocking synchronised passage-request queue
- Furthermore, synchronisation then happens to be even wait-free [6, 5]

**Hint (Wait Freedom)**

*Any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes.* [8, p. 124]
Data Type I

```c
typedef struct guard {
    int book; /* # of concurrent requests */
    queue_t load; /* pending passage requests */
} guard_t;
```

- invariably, a chain-like queue of registered "passage requests"
- mandatory, sufficient for elementary guarded sections
- with a twofold meaning of the `book` attribute depending on its value
  - the actual number of passage requests pending for processing
  - the state of occupancy (cf. p. 20): occupied if `book > 0`, unoccupied else
- variably, additional stuff for advanced control of guarded sections:
  - some timeout that ensures progress for the actual major sequencer
  - a minor sequencer to replace the major sequencer at timeout
  - any management data to prevent priority inversion, if applicable

Data Type II

```c
typedef struct order {
    chain_t next; /* passage-request chaining */
    item_t post; /* argument placeholder */
} order_t;
```

- layout of an argument vector for passage-request parameters:

```c
typedef union item {
    long (*lump)[N > 1]; /* argument vector */
    long sole; /* single argument */
} item_t;
```

- depending on the number of parameters, the structure describes a multi-
or uni-element argument vector
- in the multi-element case, the argument vector is placed adjacent to its
  item or order, resp., instance (cf. p. 41)
- in addition, this vector also serves as placeholder for a future value

Pieces the Puzzle Together

```c
#define __FAME_GUARD_ADVANCED__
...
#endif
guard_t;
```

- vouch for sequential execution of a guarded critical section:
  ```c
  inline order_t * vouch(guard_t *this, order_t *work) {
      enqueue(&this->load, work);
      if (FAA(&this->book, 1) == 0)
          return dequeue(&this->load);
      return 0;
  }
  ```
  - remember this passage request
  - check state of occupancy and book passage request
  - was unoccupied, became sequencer, accept first passage request
  - could be a request different from the one that was just remembered

- clear the next passage request, if any, pending for processing:
  ```c
  inline order_t * clear(guard_t *this) {
      if (FAA(&this->book, -1) > 1)
          return dequeue(&this->load);
      return 0;
  }
  ```
  - count completion and check for further pending requests
  - remove next passage request, if any available

Claiming and Clearing

- for editing of passage-request parameters, optional:
  ```c
  order_t * task = order(2); /* two parameters */
  (*task->post.lump)[0] = (long)index;
  (*task->post.lump)[1] = value;
  ```
  - entry protocol, agreement on the sequencer process:
    ```c
    extern guard_t gate;
    if (vouch(&gate, task)) do /* enter section */
    ```
    - midsection (i.e., actual critical section), solo attempt:
      ```c
      /* Several Species of Small Furry Animals
      * Gathered Together in a Cave and
      * Grooving with a Pict */
      ```
    - exit protocol, processing of pending passage requests:
      ```c
      while (! (task = clear(&gate))); /* leave section */
      ```
      - besides logical synchronisation in the midsection, any other programming
        statements are doable as well—like in conventional critical sections
Résumé

- guarding of critical sections at operating-system as well as instruction set architecture level and in a non-blocking manner
- processes are never delayed at entrance of an already occupied critical section, however their requests to pass through
- not unlike procedure chaining, but also supporting in-line functions
- at both levels, overlappings as to simultaneous processes result in a multiple-enqueue/single-dequeue model of request handling
- the sequencer will be the only process being in charge of dequeuing
- that is, the continuation of a requester (lev. 3) or the guardian (lev. 2)\(^4\)
- whereby this continuation is commander-in-chief of a critical section
- when a requester process requires a direct result from the sequencer process, interaction in a consumer/producer-style takes place
- in such a case, the respective request is associated with a future object
- it carries the promise of the sequencer to deliver a result to the requester
- a future-specific signalling semaphore then indicates result availability
- besides supporting conventional critical sections, this approach eases design of non-blocking synchronised non-sequential programs

\(^4\)Operating-system machine or instruction set architecture level, respectively.
[9] Hibbard, P.:
Parallel Processing Facilities.

[10] Hibbard, P.; Hisgen, A.; Rodeheffer, T.:
A Language Implementation Design for a Multiprocessor Computer System.

The Design and Implementation of the 4.3 BSD UNIX Operating System.
Addison-Wesley, 1989. – ISBN 0–201–06196–1

[12] Liskov, B. J. H.; Shrira, L.:
Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems.

[13] Pu, C.; Massalin, H.:

[14] Schöen, F.; Schröder-Preikschat, W.; Spinczyk, O.; Spinczyk, U.:
On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System.
In: Lee, I. (Hrsg.) ; Kaiser, J. (Hrsg.) ; Kikuno, T. (Hrsg.) ; Selic, B. (Hrsg.):

---

Guardian Insulating and Invoking

1. _joint:
2. pushl %ecx # save volatile register
3. movl $0 , %ecx # pass IRQ number
4. _jointN: # come here for IRQ number N > 0
5. pushl %edx # save another volatile register
6. pushl %eax # ditto
7. call _guardian # fastcall to guardian
8. popl %eax # restore volatile register
9. popl %edx # ditto
10. popl %ecx # ditto
11. iret # resume interrupted process

- each IRQ entry in the CPU exception vector is associated with a joint

1. _joint42:
2. pushl %ecx # save volatile register
3. movl $42 , %ecx # pass IRQ number
4. jmp _jointN # switch to common joint section...
**Simple Interrupt Handler**

- first-level interrupt handler (FLIH), at CPU/OS priority (p. 11, l. 7)

```c
remit_t * prelude( /* optional */ usher_t * tube ) {
    static remit_t task = { {}, postlude }; /* Come here for device pre-processing & *
    * device-related IRQ acknowledgement. */
    deter(tube, &task); /* force postlude to queue */
    return 0; /* don’t request shortcut */
}
```

- second-level interrupt handler (SLIH), at OS priority (p. 11, l. 7/13)

```c
void postlude( /* optional */ order_t * todo ) {
    /* Come here for device post-processing & *
    * any asynchronous system interaction. */
    V((semaphore_t *) todo -> post.sole);
}
```

**Interrupt-Handler Guard**

- a doorman (Ger. Pförtner) for guarded sections at the low level of handling asynchronous program interrupts, a specialised guard:

```c
typedef guard_t usher_t;
inline void deter(usher_t * tube, remit_t *task) {
    enqueue(&tube->load, &task->data.next);
}
inline remit_t * untie(usher_t * tube) {
    return (remit_t *) dequeue(&tube->load);
}
inline void flush(usher_t * tube) {
    remit_t * next;
    do if ((next = untie(tube))) remit(next);
    while (next != 0);
}
```

**Job Definition and Start**

- a SLIH or an interrupt-handler postlude, resp., is a passage request (cf. p. 27) attended by a procedure address

```c
typedef struct remit {
    order_t data; /* parameter set */
    void (*code)(order_t *); /* procedure address */
} remit_t;
inline void remit(remit_t *this) {
    (*this->code)(&this->data); /* run that job */
}
```

- an SLIH or an interrupt-handler postlude, resp., is a passage request (cf. p. 27) attended by a procedure address

**Art of Waiting Inside a Guarded Section**

- straightforward is the use of a signalling semaphore:

```c
typedef semaphore_t indicator_t;
inline void enroll(indicator_t *hint) { }
inline void repose(indicator_t *hint) { P(hint); }
inline void arouse(indicator_t *hint) { V(hint); }
```

- a V saves the signalling event in the semaphore, causing P to continue

**Art of Waiting**

- straightforward is the use of a signalling semaphore:

```c
typedef semaphore_t indicator_t;
inline void enroll(indicator_t *hint) { }
inline void repose(indicator_t *hint) { P(hint); }
inline void arouse(indicator_t *hint) { V(hint); }
```

- that is to say, a request object with implicit processing method

```c
typedef struct remit {
    order_t data; /* parameter set */
    void (*code)(order_t *); /* procedure address */
} remit_t;
inline void remit(remit_t *this) {
    (*this->code)(&this->data); /* run that job */
}
```

- at process-event level, this structure specifies different parameterised critical sections associated with the same guarded section

- it allows for procedure chaining similar to that of Synthesis [13, p. 10]

**Job Definition and Start**

- a SLIH or an interrupt-handler postlude, resp., is a passage request (cf. p. 27) attended by a procedure address

**Art of Waiting Inside a Guarded Section**

- straightforward is the use of a signalling semaphore:

```c
typedef semaphore_t indicator_t;
inline void enroll(indicator_t *hint) { }
inline void repose(indicator_t *hint) { P(hint); }
inline void arouse(indicator_t *hint) { V(hint); }
```

- that is to say, a request object with implicit processing method

```c
typedef struct remit {
    order_t data; /* parameter set */
    void (*code)(order_t *); /* procedure address */
} remit_t;
inline void remit(remit_t *this) {
    (*this->code)(&this->data); /* run that job */
}
```

- at process-event level, this structure specifies different parameterised critical sections associated with the same guarded section

- it allows for procedure chaining similar to that of Synthesis [13, p. 10]
inline order_t * order(unsigned long n) {
    order_t * item;
    if (n < 2)
        item = (order_t *) malloc(sizeof(order_t));
    else {
        item = (order_t *) malloc(sizeof(order_t) + n * sizeof(long));
        item->post.lump = (void *)
            ((long)item + sizeof(*item));
    }
    return item;
}
inline void ditch(order_t * item) {
    free(item);
}

in order to decrease latency and lower overhead, specialisation towards
the use of an order pool is recommended

typedef struct future {
    promise_t data; /* prospective value */
    indicator_t gate; /* signalling element */
} future_t;

a future object is the promise—of a guarded section, here—to deliver
a result at some later point in time:

typedef enum status {
    PENDING, KEPT, BROKEN
} status_t;

typedef struct promise {
    status_t bond; /* processing state */
    item_t item; /* future-value placeholder */
} promise_t;

whereby the promise is a result placeholder, on the one hand, and keeps
track of the status of result delivery, on the other hand

inline status_t probe(future_t * this) {
    return this->data.bond;
}
inline void trust(future_t * this) {
    enroll(&this->gate);
}
inline item_t * exact(future_t * this) {
    repose(&this->gate);
    return probe(this) == KEPT ? &this->data.item : 0;
}
inline void bring(future_t * this, status_t bond) {
    this->data.bond = bond;
    arouse(&this->gate);
}
inline void prove(future_t * this, item_t * item) {
    this->data.item = *item;
    bring(this, KEPT);
}
inline void abort(future_t * this) { bring(this, BROKEN); }