Concurrent Systems

Nebenläufige Systeme

XI. Non-Blocking Dynamic Data Structures

Wolfgang Schröder-Preikschat

January 28, 2021
Preface

Singly Linked List
   Working Principle
   Concurrent Operation

Collating Sequences
   Stack
   Queue

Summary
Preface

Singly Linked List
   Working Principle
   Concurrent Operation

Collating Sequences
   Stack
   Queue

Summary
discussion of the abstract concept of **non-blocking synchronisation** in the context of **dynamic data structures**

- namely the **singly linked list** with any access pattern of the operations
- this is compared to structures with a defined access pattern: LIFO, FIFO
  - the **stack** and the **queue** as a linked dynamic data structure

starting from conventional sequential solutions, semantically identical non-sequential alternatives are gradually developed

- race conditions are revealed and tricks for avoidance are presented
- the limits of the developed solutions are discussed

an important aspect in the considerations and driving idea is to find solutions for the **conservative handling** of dynamic data structures

- on the one hand, this means being able to reuse the data structures of the corresponding sequential solutions
  - coordination as a *minimal extension of system functions* [5] in its purest form
  - that is, the extension only refers to the *instructions* of a virtual machine

- on the other hand, the solutions rely on comparatively simple elementary operations such as CAS and FAS
Outline

Preface

Singly Linked List
   Working Principle
   Concurrent Operation

Collating Sequences
   Stack
   Queue

Summary
reunion with a **dynamic data structure** ([8, p. 9] and [10, p. 18]):
- basic abstraction from any entity as a **list element**

```c
typedef struct chain {
    struct chain *link; /* next list element */
} chain_t;
```

based on this, the following operations are defined:

- `chain_t *infix(chain_t *this, chain_t *item)`
- `chain_t *unfix(chain_t *this)`
- `chain_t *split(chain_t *this)`
- `chain_t *visit(chain_t *this, chain_t *item)`
- `chain_t *annex(chain_t *this, chain_t *item)`
- `chain_t *erase(chain_t *this, chain_t *item)`

- with `this` the chain link on which the operation should work,
- `item` the list entry to be inserted, searched for or deleted and
- a chain link or list entry as a result, depending on the operation
Chain Operations I

simple list manipulation: insert and unhook entry, split list

```c
inline chain_t *infix_dos(chain_t *this, chain_t *item) {
    item->link = this->link;
    return this->link = item;
}

inline chain_t *unfix_dos(chain_t *this) {
    chain_t *item;
    item = this->link;
    if (item != 0)
        this->link = item->link;
    return item;
}

inline chain_t *split_dos(chain_t *this) {
    chain_t *tail;
    tail = this->link;
    this->link = 0;
    return tail;
}
```

Caller assured: `infix`

An entry (item) to be inserted is not yet on the list.
Chain Operations II

more complex list manipulation: seek out, append, delete

```c
inline chain_t *visit_dos(chain_t *this, chain_t* item) {
    chain_t *next;
    while (this && ((next = this->link) != item))
        this = next;
    return this;
}

inline chain_t *annex_dos(chain_t *this, chain_t* item) {
    return infix_dos(visit_dos(this, 0), item);
}

inline chain_t *erase_dos(chain_t *this, chain_t *item) {
    chain_t *hand;
    hand = visit_dos(this, item);
    if (hand != 0)
        hand = unfix_dos(hand);
    return hand;
}
```
Application Example

...functionally quite useful

```c
#include "chain.h"

typedef struct list {
    chain_t head;
} list_t;

inline void push(list_t *this, chain_t *item) {
    infix(&this->head, item);
}

inline chain_t *pull(list_t *this) {
    return unfix(&this->head);
}

inline void enlist(list_t *this, chain_t *item) {
    annex(&this->head, item);
}

inline chain_t *delist(list_t *this) {
    return unfix(&this->head);
}
```

- good for classic collating strategies
  - LIFO list ～ stack (cf. [10, p. 19])
  - FIFO list ～ queue
Non-Sequential Execution

- All chain operations are prone to race conditions:
  - **infix** new entries can disappear
  - **unfix** the same entry can be extracted several times
  - **split** list entries at the front end can disappear
  - **visit** incorrect/invalid list section can be run through
  - **annex** an entry on a wrong list can be made (inherited error)
  - **erase** an entry was wrongly not deleted (inherited error)

- They are **not thread safe** and need to be synchronised
  - Blocking synchronisation can be child’s play here: use a **chain monitor**
    - Very simple is a medium-grain approach, with a lock on the entire list
    - More difficult is a fine-grained solution, with one lock per list item
  - In both cases, the locking effort is high compared to the actual operation
  - Non-blocking synchronisation is basically a fine-grained technique
    - Thus also of a similar complexity as the corresponding blocking variant
    - Because protecting a single list element is usually not enough
  - Simultaneous processes to the left and right of it must also be considered

- Short blocking times are asked what motivates an optimistic approach
  - Non-blocking is no more difficult than fine-grain blocking
Insert List Item: 1st Try

- attach an entry to the hook: almost too good to be true...

```c
inline chain_t *infix_lfs(chain_t *this, chain_t *item) {
    do item->link = this->link;
    while (!CAS(&this->link, item->link, item));
    return item;
}
```

- a certain collating strategy was wrongly assumed here: LIFO
  - this only manipulates the list on one side: at the top (this) of the stack
  - but now every other position on the list can be changed at the same time
  - note that the insertion point here (*this) can be any list element
    - this list item could be deleted while a new item is being appended to it
    - thus, the new list item disappears immediately with the deleted element
  - that this entry (*this) is currently being deleted must be indicated
    - the link pointer of the element concerned is to be regarded as frail
    - it may not be used for chaining...
unhook the attached element and attach its successor

```c
inline chain_t *unfix_lfs(chain_t *this) {
    chain_t *item;
    do if (((item = pure(this->link)) == 0) break;
    while (!purge(this, item, pure(item->link)));
    return item;
}
```

the deletion indicator is based on the tagging of chain attributes (`link`)
- when deleting (`purge`), the `link` pointer attribute becomes frail (l.4)
- when using such a pointer attribute, attention must be paid to purity (l.3)

the actual deletion is done using the following auxiliary function

```c
inline bool purge(chain_t *this, chain_t *item, chain_t *next) {
    bool done;
    CAS(&item->link, next, mark(next, FRAIL));
    if (!(done = CAS(&this->link, item, next)))
        CAS(&item->link, mark(next, FRAIL), next);
    return done;
}
```

- after successful deletion (CAS succeeds, l.10), the pointer remains frail
- regular reuse makes the pointer stable again $\leadsto$ potential race condition
Tagging of Pointer Attributes

misappropriation of bits of a pointer that are free due to alignment

typedef enum flag {
    FRAIL=(1<<0), /* pointer should not be used */
    DODGY=(1<<1)  /* reusing the pointer is tricky */
} flag_t;

simple bit operations based on this for setting, querying and cleaning

inline long mark(long item, flag_t flag) {
    return item | flag;
}

inline bool just(long item, flag_t flag) {
    return item & flag;
}

inline long pure(long item) {
    return item & ~(FRAIL|DODGY);
}
attach an entry to the hook: feasible only if not frail...

inline chain_t *infix_lfs(chain_t *this, chain_t *item) {
  do {
    item->link = this->link;
    if (just(item->link, FRAIL))
      return 0;
  } while (!CAS(&this->link, item->link, item));
  return item;
}

- if the hook entry is just being deleted, it is unclear where else to attach
  - in this case, the caller must determine the new insertion point
    - fail

overlapping deletion (purge) of the link (*this) leaves a frail pointer
- then CAS fails because this link pointer has been changed
  - no longer equals its original value (cf. l.3)
- the operation is retried but then detects the conflict (l.3) and aborts (l.4)
- however, reusing the deleted list item (*this) presents a problem
  - the link pointer can soon be purified again
    - potential race condition
Safe Reuse of a Purged List Entry

which side (of `unfix`) has to take care of this is quite controversial

caller

- only here is it known whether the list item will be reused at all
- whether an entry is made in a reused list item is undecidable here

-callee

- assume the following sequence of instructions:

1. `chain_t etc = { 0 }`;
2. `hook = infix(&etc, a); /* etc->a */`
3. `took = infix(hook, b); /* etc->a->b */`
4. `node = unfix(hook); /* etc->a, node == b */`
5. `if (node != 0) /* true: typo "took" */`
6. `infix(&etc, node); /* etc->b->a */`

- as a programming error that has the same effect (from l.2) as:

  `infix(&etc, unfix(&etc)) || infix(&etc, b)`

  - here unfixed list item “a” is reused, put back on the list

- executed in parallel with a potential race condition (cf. p. 14)

→ a solution on the callee side does not seem to help very much...

each side should regulate their race conditions for themselves

- this also applies to blocking synchronisation: *separation of concerns* [1]
lock-free variant

inline chain_t *split_lfs(chain_t *this) {
    chain_t *tail;
    do tail = this->link;
    while (!CAS(&this->link, tail, 0));
    return pure(tail);
}

wait-free variant

inline chain_t *split_wfs(chain_t *this) {
    return pure(FAS(&this->link, 0));
}

both variants are compatible with infix (p.14) and unfix (p.12)
Search for a Link Pointer Entry

- a list entry with a frail link is problematic, following it is questionable
  - this entry was or will be deleted soon, its link is therefore undefined

- if a frail link is encountered while searching:
  - either start over again and retry (l. 5) or abort with error code (l. 7–8)

```c
inline chain_t *visit_lfs(chain_t *this, chain_t *item) {
    chain_t *next, *root = this;
    while (this && ((next = this->link) != item))
        #ifdef __FAME_CHAIN_VISIT_RETRY__
            this = just(next, FRAIL) ? root : next;
        #else
            if (just(next, FRAIL)) return -1;
            else this = next;
        #endif
    return this;
}
```
# define __FAME_CHAIN_VISIT_RETRY__

append an element to the apparent end of the linked list:

```c
inline chain_t *annex_lfs(chain_t *this, chain_t *item) {
    return infix_lfs(visit_lfs(this, 0), item);
}
```

delete an entry from the linked list:

```c
inline chain_t *erase_lfs(chain_t *this, chain_t *item) {
    chain_t *hand = visit_lfs(this, item);
    if (hand != 0)
        hand = purge(hand, item, pure(item->link));
    return hand;
}
```

but that would be too good to be true, it is not that easy

- the composite operations consist of two complex individual steps:
  1. localisation of the list element that contains the link pointer sought
  2. application of the respective operation to the localised list element
- both steps happen one after the other $\sim$ prone to race condition
append an element to the apparent end of the linked list:

```c
inline chain_t *annex_lfs(chain_t *this, chain_t* item) {
    item->link = 0;
    do this = visit(this, 0);
    while (!CAS(&this->link, 0, item));
    return item;
}
```

after finding the end of the list, it may have been deleted or moved

delete an entry from the linked list:

```c
inline chain_t *erase_lfs(chain_t *this, chain_t *item) {
    chain_t *hand;
    do if ((hand = visit(this, item)) == 0) break;
    while (!purge(hand, item, pure(item->link)));
    return hand;
}
```

after finding the entry on the list, it may have been deleted
Outline

Preface

Singly Linked List
  Working Principle
  Concurrent Operation

Collating Sequences
  Stack
  Queue

Summary
the access method has a great influence on possible race conditions

- without a specific access pattern; a chain-like approach as before
  - shared variables are all link pointers in the list
  - when handling the link of any list element, attention must be paid to concurrent processes to the left and right of it

**LIFO**  
*last in, first out*; a stack-like approach

- shared variable is only the pointer to the list head
  - attention must be paid only to concurrent processes when handling the link to the list head, all other links in the list are not critical

**FIFO**  
*first in, first out*; a queue-like approach

- shared variables are the pointers to the list head and tail, only
  - quite similar to LIFO, except that handling the links to both the list head and the list tail has to be considered together

following topic is singly linked lists with LIFO and FIFO semantics
Data Type II

- a singly-linked list with collating sequence last in, first out (LIFO):

```c
#include "chain.h"

/* A stack data type, here represented
 * in terms of a singly-linked list. */
typedef struct stack {
    chain_t head; /* top of stack: list head */
} stack_t;
```

- based on this, the following operations are defined:
  ```c
  void push(stack_t *this, chain_t *item)
  chain_t *pull(stack_t *this)
  chain_t *peek(stack_t *this)
  ```
  - with `this` as the list head, that is, the stack pointer,
  - `item` as the list entry to be stacked and
  - a chain link or list entry as a result, depending on the operation
simply map onto the lock-free synchronised chaining operations:

```c
inline void push(stack_t *this, chain_t *item) {
    infix_lfs(&this->head, item);
}

inline chain_t *pull(stack_t *this) {
    return unfix_lfs(&this->head);
}
```

however, this mapping does not benefit from the LIFO access method overhead due to functionality that is not required is carried along:
- tagging of pointer attributes
- pointer tagging and untagging in `unfix`
- checking for a frail pointer in `infix`

additional work, as no specific access pattern could be assumed

this kind of **black-box reuse** without being able to derive a benefit from it in non-functional terms is a matter of dispute
Stack Operations: 2nd Try

- wait-free (synchronised) function

```c
inline chain_t *peek(stack_t *this) {
    return this->head.link;
}
```

- lock-free synchronised functions $\sim$ conceptual reuse

```c
inline void push_lfs(stack_t *this, chain_t *item) {
    do item->link = peek(this);
    while (!CAS(&this->head.link, item->link, item));
}
```

```c
inline chain_t *pull_lfs(stack_t *this) {
    chain_t *item;
    do if ((item = peek(this)) == 0) break;
    while (!CAS(&this->head.link, item, item->link));
    return item;
}
```
a much more complex object of investigation, at a second glance:

```
typedef struct queue {
    chain_t  head;     /* first item */
    chain_t *tail;     /* insertion point */
} queue_t;
```

- the tail pointer addresses the linkage element of a next item to be queued
- it does not directly address the last element in the queue, but indirectly

consequence is that even an empty queue shows a valid tail pointer:

```
inline chain_t *deplete(queue_t *this) {
    chain_t *list = this->head.link;

    this->head.link = 0;       /* null item */
    this->tail = &this->head;  /* linkage item */

    return list;
}
```

- used to reset a queue and at the same time return all its list members
same **precondition** as before: an item to be queued is not yet queued

- a simple **first-in, first-out method** (FIFO) is implemented

```c
inline void enqueue_dos(queue_t *this, chain_t *item) {
    item->link = 0;  /* finalise chain */
    this->tail->link = item;  /* append item */
    this->tail = item;  /* set insertion point */
}
```

3. the queue head pointer gets set to the first item implicitly

```c
inline chain_t *dequeue_dos(queue_t *this) {
    chain_t *node;
    if ((node = this->head.link)  /* filled? */
        && !(this->head.link = node->link))  /* last item? */
        this->tail = &this->head;  /* reset */
    return node;
}
```

10. the tail pointer must always be valid, even in case of an empty queue
Append Item: $enq \parallel enq$

inspired by the lock-free solution using atomic load/store [9, p. 28]:

```c
void enqueue_lfs(queue_t *this, chain_t *item) {
    chain_t *last;
    item->link = 0;
    do last = this->tail;
    while (!CAS(&this->tail, last, item));
    last->link = item;
}
```

A **plausibility check** shows correctness as to this overlap pattern:

6. critical shared data is the tail pointer, a local copy is **read**
   - each overlapping enqueue holds its own copy of the tail pointer
7. validate and, if applicable, **write** to update the tail pointer
   - the item becomes new fastener for subsequent enqueue operations
9. eventually, the item gets inserted and becomes queue member
   - the assignment operator works on local operands, only

**Hint (Onefold Update)**

*Only a single shared variable needs to be updated in this scenario.*
Remove Item: \textit{deq} \textvert \textit{deq}

inspired by the lock-free solution for a stack pull operation (p. 24):

```c
chain_t * dequeue_lfs(queue_t *this) {
    chain_t *node;

    do if ((node = this->head.link) == 0) return 0;
    while (!CAS(&this->head.link, node, node->link));

    if (node->link == 0)
        this->tail = &this->head;

    return node;
}
```

Hint (Onefold Update)

\textit{Only a single shared variable needs to be updated in this scenario.}

a \textbf{plausibility check} shows correctness as to this overlap pattern:

4 ■ critical shared data is the head pointer, a local copy is \textit{read}
   ■ each overlapping dequeue holds its own copy of the head element
5 ■ \textit{validate} and, if applicable, \textit{write} to update the head pointer
7 ■ each dequeued item is unique, only of them was last in the queue
8 ■ the tail pointer must always be valid, even in case of an empty queue
Neuralgic Points

crosswise simultaneous execution

- critical is when head \textit{and} tail pointer refer to the same “hot spot” and enqueue and dequeue happen simultaneously

- assuming that the \textit{shared queue} consists of only a single element:
  - \textit{enq} || \textit{deq}
    - enqueue memorised the chain link of that element
    - dequeue removed that element including the chain link
    - enqueue links the new element using an invalid chain link
    \rightarrow \textit{lost enqueue}: linking depends on dequeue progression

  - \textit{deq} || \textit{enq}
    - dequeue removed that element and notices “vacancy”
    - enqueue appends an element to the one just removed
    - dequeue assumes “vacancy” and resets the tail pointer
    \rightarrow \textit{lost enqueue}: resetting depends on enqueue progression

- enqueue and dequeue must assist each other to solve the problem:
  - identify the conditions under which lost-enqueue may happen
  - identify a way of interaction between enqueue and dequeue

- assist without special auxiliary nodes but preferably with simultaneous consideration of \textit{conservative data-structure handling}
Conservative Approach

- idea is to use the chain-link of a queue element as **auxiliary means**
  - for the interaction between enqueue and dequeue [6]
  - let last be the pointer to the chain link of the queue end tail and
  - let link\(_{last}\) be the chain link pointed to by last, then:

\[
\text{link}_{last} = \begin{cases} 
\text{last}, & \text{chain link is valid, was not deleted} \\
0, & \text{chain link is invalid, was deleted} \\
\text{else}, & \text{chain link points to successor element}
\end{cases}
\]

- link\(_{last}\) set to 0 models the per-element “deleted bit” as proposed in [2]
  - for a FIFO queue, only the end-tail element needs to carry that “bit”
- in contrast to [2], advanced idea is to do without a garbage-collection mechanism to dispose of the “deleted” queue end-tail element
  - purpose is to signal unavailability of the end-tail chain link to enqueue
  - thus, when dequeue is going to remove last it attempts to zero link\(_{last}\)
  - contrariwise, enqueue appends to last only if link\(_{last}\) still equals last
- signalling as well as validation can be easily achieved using CAS
  - algorithmic construction versus CDS [3, p.124] or DCAS [4, p.4-66]...
Append Item: 2nd Try

```c
void enqueue_lfs(queue_t *this, chain_t *item) {
    chain_t *last, *hook;

    item->link = item; /* self-reference: hook */

    do hook = (last = this->tail)->link; /* tail end */
        while (!CAS(&this->tail, last, item));

    if (!CAS(&last->link, hook, item)) /* endpiece? */
        this->head.link = item; /* no longer! */
}
```

validate availability of the ending and potential volatile chain link:

9. CAS succeeds only if the last chain link is still a self-reference
   in that case, the embracing last element was not dequeued

10. CAS fails if the last chain link is no more a self-reference
    in that case, the embracing last element was dequeued

    the item to be queued must be head element of the queue, because
    further enqueues use this very item as leading chain link (l. 7)
Remove Item: 2nd Try

lock-free synchronised

```c
chain_t *dequeue_lfs(queue_t *this) {
    chain_t *node, *next;

    do if ((node = this->head.link) == 0) return 0;
    while (!CAS(&this->head.link, node,
                ((next = node->link) == node ? 0 : next)));

    if (next == node) { /* self-reference, is last */
        if (!CAS(&node->link, next, 0)) /* try to help */
            this->head.link = node->link; /* filled */
        else CAS(&this->tail, node, &this->head);
    }

    return node;
}
```

validate tail-end invariance of a one-element queue (\(head = tail\)):
9  ■ CAS fails if the node dequeued no more contains a self-reference
10 ■ thus, enqueue happened and left at least one more element queued
11 ■ enqueue was assisted and the dequeued node could be last, really
Outline

Preface

Singly Linked List
   Working Principle
   Concurrent Operation

Collating Sequences
   Stack
   Queue

Summary
non-blocking synchronisation of **dynamic data structures**, without being able to benefit from well-defined access patterns, is tricky

- even the singly-linked list presents some difficulties
  - it is not enough to look at individual list elements in isolation
  - rather, entries in the immediate vicinity must also be taken into account

- for a doubly-linked list, the solutions look quite different

however, knowledge of **access patterns** is by no means a guarantee for simple and easy-to-use solutions

- LIFO is simple as there is only a single shared variable in the list
  - the link to the beginning of the list, the “stack pointer”

- in contrast to FIFO, where two such variables are to be kept consistent
  - the links to the beginning and the ending of the list, the queue pointers
  - but both links only form a critical case if the queue contains a single item

- driving force in the development of the solutions was a **conservative handling** of dynamic data structures

- reuse of the data structures given with the original sequential variants

- use of one-word atomic processor instructions such as CAS and FAS
**Reference List I**

[1] **Dijkstra, E. W.**:
*On the Role of Scientific Thought.*

[2] **Harris, T. L.**:
A Pragmatic Implementation of Non-blocking Linked-Lists.

[3] **IBM Corporation (Hrsg.)**:
*IBM System/370 Principles of Operation.*
Fourth.

[4] **Motorola Inc. (Hrsg.)**:
Rev. 1.
Schaumburg, IL, USA: Motorola Inc., 1992. (M68000PM/AD)


[8] **Schröder-Preikschat, W.**: Critical Sections. In: [7], Kapitel 4

[9] **Schröder-Preikschat, W.**: "Guarded Sections". In: [7], Kapitel 10
Empty and Reset a Queue

```c
inline chain_t *deplete_wfs(queue_t *this) {
    chain_t *list = &this->head.link;
    if (list != 0) { /* empty queue */
        if (!CAS(&this->head.link, list, 0))
            return 0;
        this->tail = &this->head; /* reset queue */
    }
    return list;
}
```

depletion (dt. Entleerung) of a queue delivers all entries at once, as a singly linked list, and defines the initial state for adding new entries

- the queue is full, try to take the listed entries as a whole
- make sure that a filled queue is only emptied once
- a simultaneous deplete or dequeue causes the operation to end here!
- define the initial state of an empty queue

Line 6 defines a “turning point” for simultaneous enqueue operations

- entries arriving beforehand are still added to the list to be delivered
- then incoming entries are collected in the newly created queue

wait-free synchronised
the problem with the queue operations shown is to ensure that two pointers are changed consistently

- both tail and head pointer change element by element per en-/dequeue
  - each enqueue creates an entry for the next “epoch” of queue processing
  - each dequeue removes an entry for processing in the current “epoch”

- complex list manipulations per operation that have to be synchronised

assuming only the tail pointer is read and written as usual, while the head pointer is only read and what it points to is only zeroed

- both epochs use their own list and are isolated from each other
- the work epoch starts with all the entries listed in the collective epoch

these two dedicated epochs motivate the following data structure:

```
typedef struct epoch {
    chain_t *work; /* currently processed entries */
    queue_t next; /* entries for the next period */
} epoch_t;
```

- if the work list runs empty, it takes over the next queued entries
- the queue is then completely emptied at once, reset to its initial state
Epoch Operations

In the **collection epoch**, entries are placed in the queue as usual.

```c
void collate_dos(epoch_t *this, chain_t *item) {
    enqueue_dos(&this->next, item);
}
```

The previously collected entries are processed in the **work epoch**.

```c
chain_t *discard_dos(epoch_t *this) {
    chain_t *item;
    if ((item = this->work) != 0) {
        if ((this->work = item->link) == 0)
            this->work = deplete_dos(&this->next);
        return item;
    }
}
```

- If there is no more work to be done, the queue of entries that have been collected in the meantime is emptied (cf. deplete, p. 25).
- That is, all queued entries are placed on the work list and the queue is reset for the next collection epoch.
collation of entries on a list that will be processed in the next epoch
- it notes the next entries that are received while processing earlier entries
- the queue (next) provided for this is a separate list from the work list

```c
void collate_lfs(epoch_t *this, chain_t *item) {
    chain_t *last;

    do last = this->next.tail;
    while (!CAS(&last->link, 0, item));

    CAS(&this->next.tail, last, item);
}
```

4–5 make sure that the next entry (item) is queued at the end: FIFO
- try again in case of a simultaneous collate() or discard()
- do this until the new last entry has been committed

7 commit the new last entry of the queue if there is no conflict
- this action releases any polling process from the loop (l. 4–5)
- which also applies when a new work epoch starts (cf. l. 6, p.38)

\(^1\)A spin on read mode of operation makes sense if contention is too high.
processing of entries on a work list collated in the previous epoch

- it takes the queue entries to be processed from its own, separated list
- this list will only be shortened, but not extended—*with one exception*

```c
chain_t *discard_wfs(epoch_t *this) {
    chain_t *item;
    if ((item = this->work) != 0)
        if (CAS(&this->work, item, item->link))
            if (CAS(&item->link, 0, item)) /* ran empty */
                this->work = deplete_wfs(&this->next);
            else CAS(&this->work, 0, item->link);
    return item;
}
```

3. there is at least one entry on the work list
4. delete the entry from the work list if it is still there
5–6. try to get new entries when the work list has been emptied
   - make sure that the last entry cannot be followed by another entry
   - end the current collection epoch, accept all entries at once
7. the last entry suddenly got a successor (l. 5, p. 41), note it!
8. deliver the next entry from the work list, if there is still one