Beware of some German slides!

Betriebssysteme

Vorlesung im Herbstsemester 2008
Universität Mannheim

Kapitel 4a: Virtual Memory in ULIX

Felix C. Freiling
Lehrstuhl für Praktische Informatik 1
Universität Mannheim

(Diese Folien sind nicht Teil des Basiskurses)

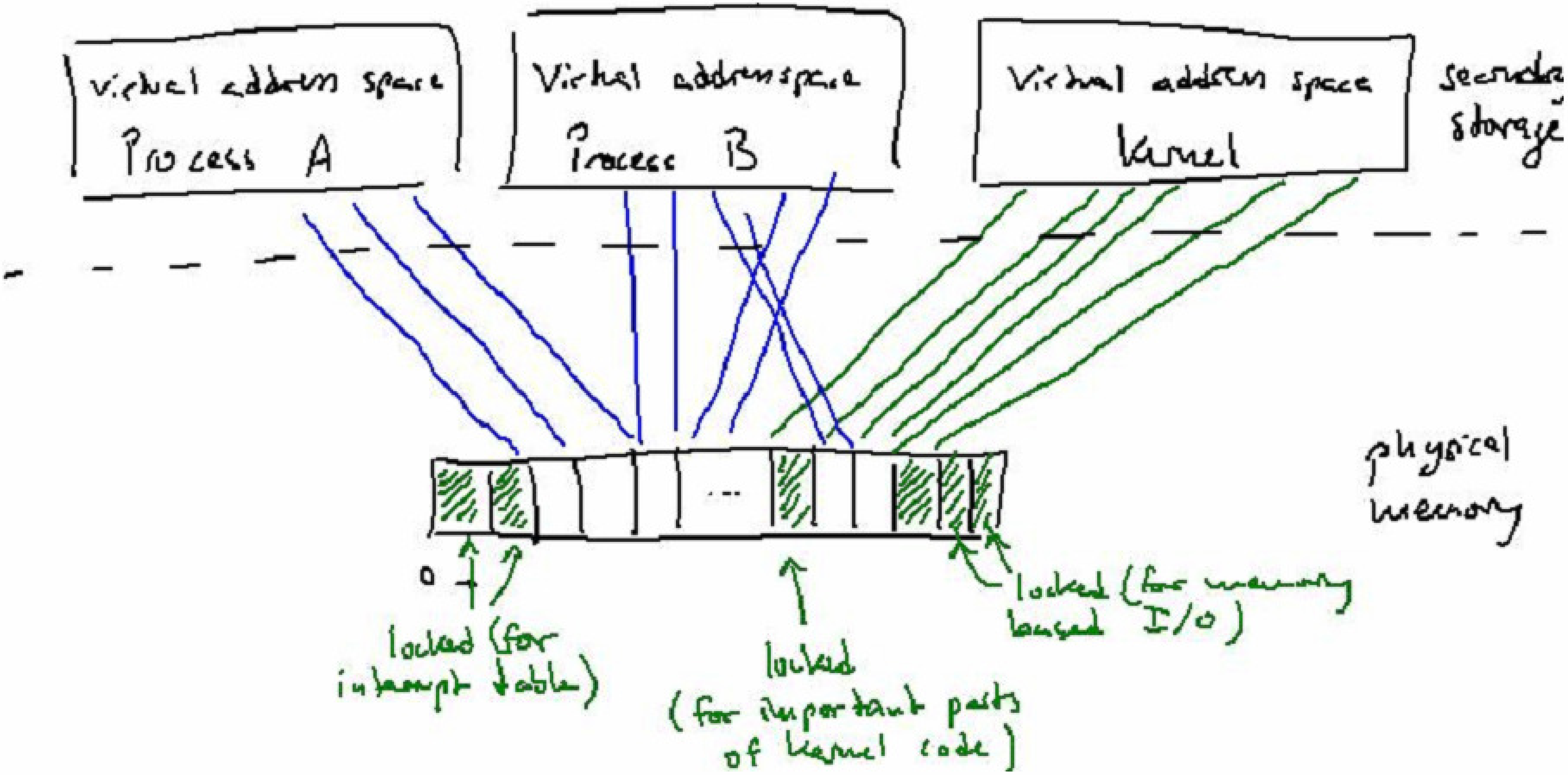
Overview

- Main memory as cache
- Page descriptors and frame descriptors
- Page allocation at system startup
- Page replacement
  - FIFO
  - Second chance
  - Clock
  - Third chance

ULIX Virtual Memory: Design Principles

- Every process will have its own virtual memory
  - Own page table tree
- Pages are stored in page frames
  - Pages can be locked down
  - Locked down pages cannot be paged out
- Page replacement is global
  - Treats all frames equally, no matter to which virtual memory they belong
- ULIX implements demand paging
  - No pre-paging (yet)
- Kernel has its own virtual memory
  - Accessible from virtual memory of every process (in system mode)

Main Memory as Cache



Notes

- Physical memory is a cache for virtual address spaces (pages) of processes
- Physical memory is completely divided up into page frames
  - Page frames hold pages of virtual address spaces
  - Some pages are locked down into the frame
- Cache management data is also kept in some page frame
  - Must be locked down

Reasons for being Locked Down

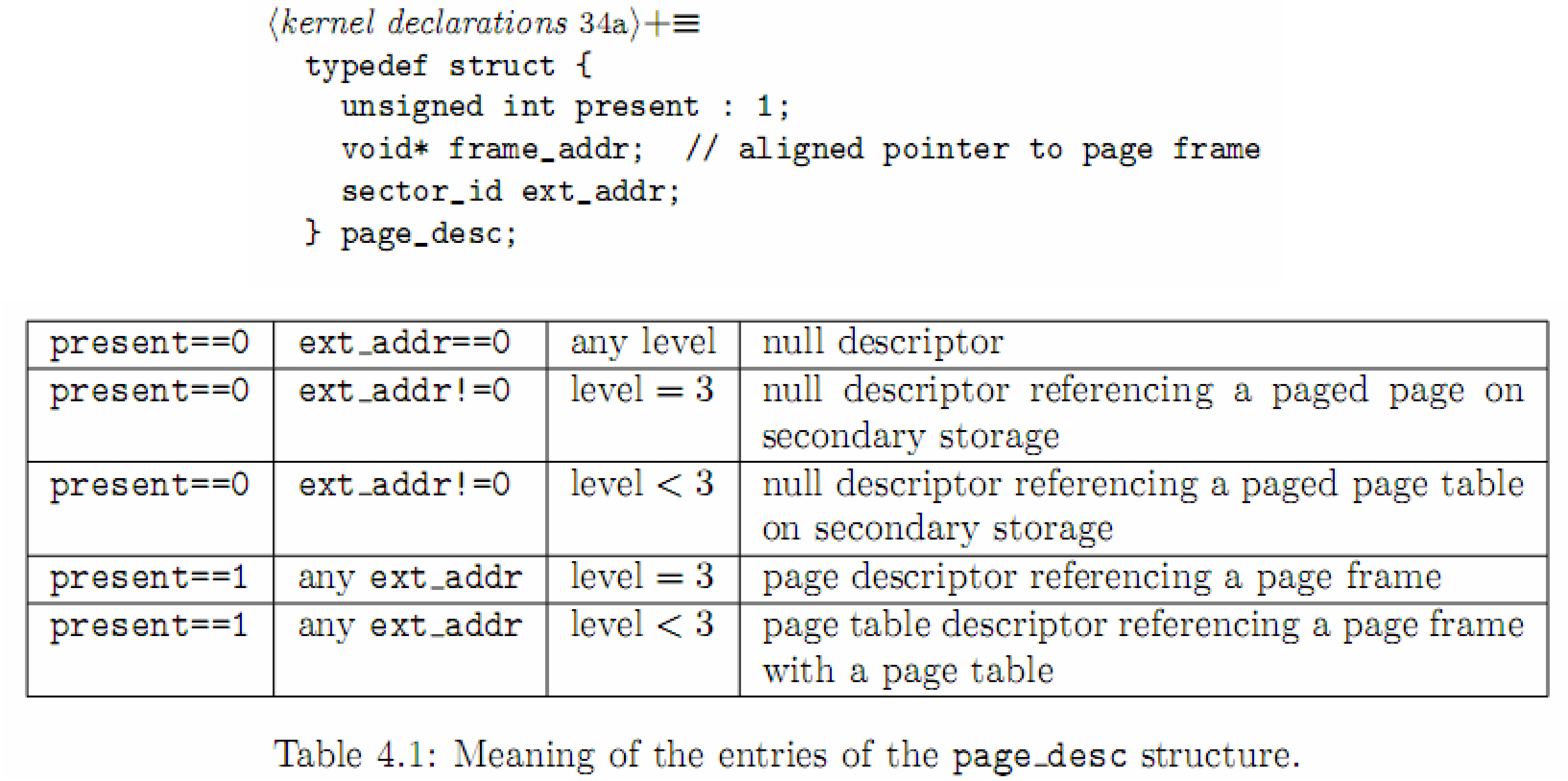
- Vital parts of kernel code and data cannot be paged out
  - Cache management data (frame table)
  - Cache management code (interrupt handlers)
- Frames containing memory mapped I/O registers or interrupt table
- Frames that are “in transfer” (see later)

Overview

- Main memory as cache
- **Page descriptors and frame descriptors**
- Page allocation at system startup
- Page replacement
  - FIFO
  - Second chance
  - Clock
  - Third chance

Page Descriptors and Page Tables

- Page tables are arrays of page descriptors
- Page descriptor structure:



⟨*kernel declarations* 34a⟩+≡

```
typedef struct {
  unsigned int present : 1;
  void* frame_addr;  // aligned pointer to page frame
  sector_id ext_addr;
} page_desc;
```

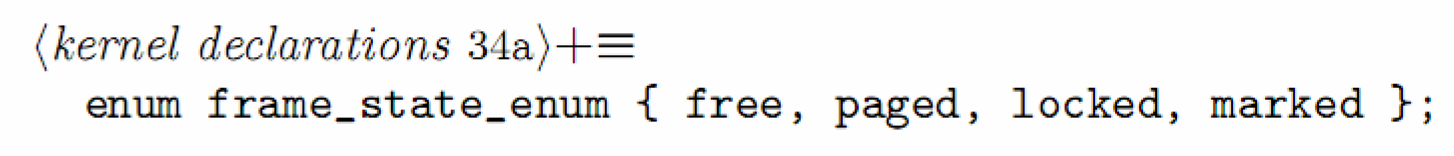
`present==0`	`ext_addr==0`	any level	null descriptor
`present==0`	`ext_addr!=0`	level $= 3$	null descriptor referencing a paged page on secondary storage
`present==0`	`ext_addr!=0`	level < 3	null descriptor referencing a paged page table on secondary storage
`present==1`	any `ext_addr`	level $= 3$	page descriptor referencing a page frame
`present==1`	any `ext_addr`	level < 3	page table descriptor referencing a page frame with a page table

Table 4.1: Meaning of the entries of the `page_desc` structure.

Frame Descriptors and Frame Table

- There is only one physical memory, so there is only one set of page frames
  - Entire physical memory is divided into page frames of equal size
- Frames act as cache entries, so they need management information
  - Information is kept in frame descriptors, one per frame
- Management information is kept in frame table
  - Array of frame descriptors

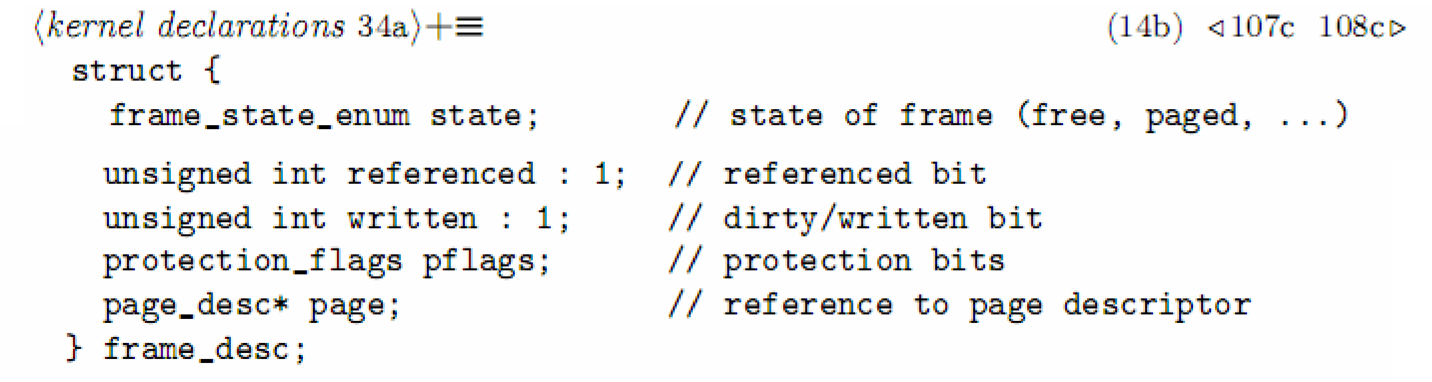
State of a Frame



```
⟨kernel declarations 34a⟩+≡
  enum frame_state_enum { free, paged, locked, marked };
```

- free = empty and ready for use
- paged = full, some memory paged is currently mapped into this frame
- locked = paged, but not allowed to page out
- marked = special state (see later)

Frame Descriptor



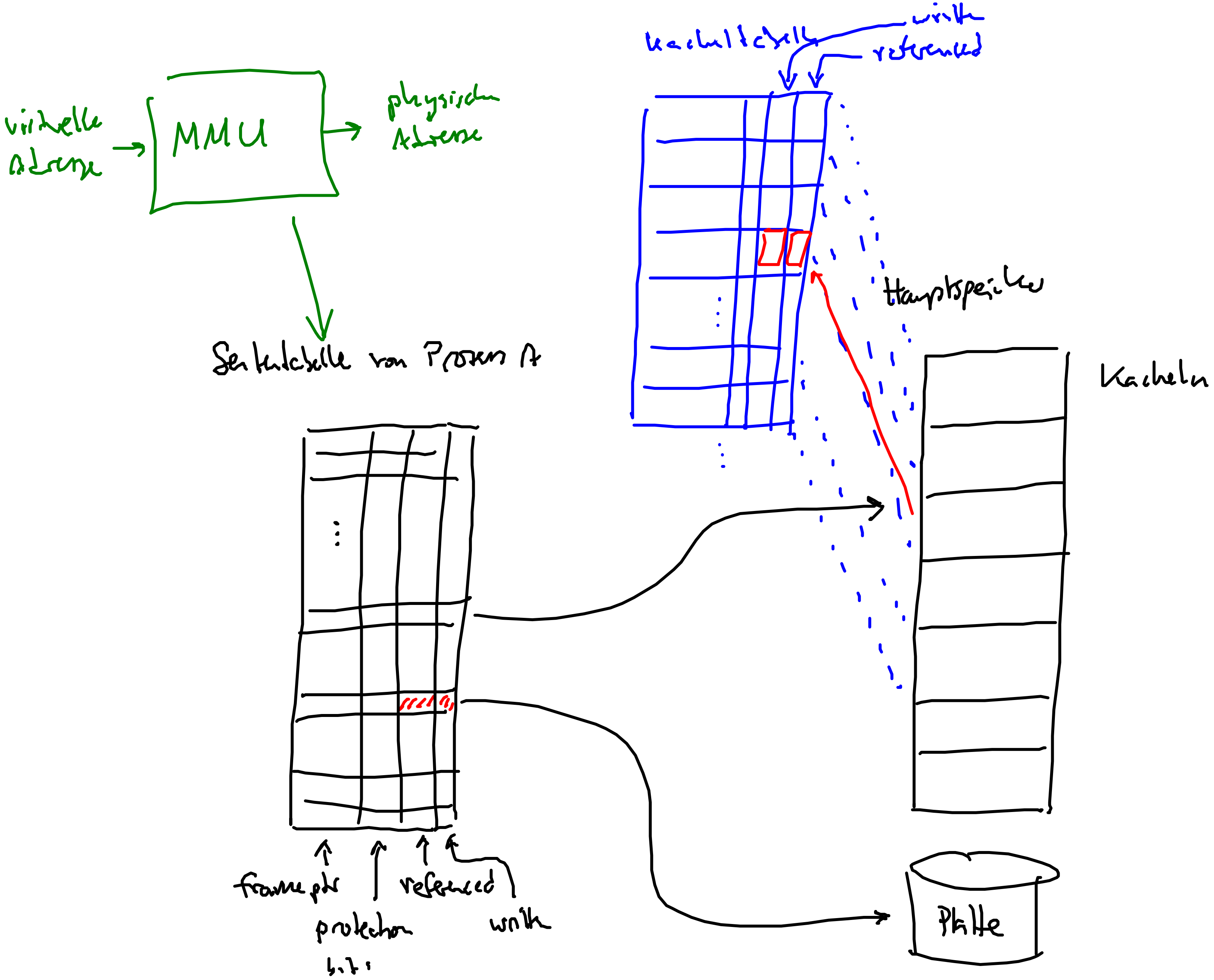
```
⟨kernel declarations 34a⟩+≡                                   (14b) ◁107c 108c▷
  struct {
    frame_state_enum state;        // state of frame (free, paged, ...)
    unsigned int referenced : 1;  // referenced bit
    unsigned int written : 1;     // dirty/written bit
    protection_flags pflags;      // protection bits
    page_desc* page;              // reference to page descriptor
  } frame_desc;
```

- Contains
  - frame state
  - cache management bits
  - protection bits
  - “backwards” reference to corresponding page descriptor (if page is not free)

Sorry, where are these bits kept?

Cache Management Bits

- It the lecture, the frame table was ignored
- Cache management bits (dirty/written, referenced) were kept in the page table
- Since these bits are only relevant when a page is paged, they can also be stored in the frame table
  - Only requirement: hardware must find these bits are correctly manipulate them
  - Example: When a page is written, the written bit should be set


virtuelle Adresse
MMU
physische Adresse
Seitentabelle von Prozess A
Kacheltabelle
write
referenced
Hauptspeicher
Kacheln
Platte
frame ptr
protection bits
referenced
write

Frame Table

⟨*common declarations* 14a⟩+≡

```
#define NUMBER_OF_FRAMES MAX_ADDRESS \ PAGE_SIZE
```

⟨*kernel global variables* 108b⟩≡

```
frame_desc frame_table[NUMBER_OF_FRAMES];
```

⟨*initialize kernel global variables* 108d⟩≡

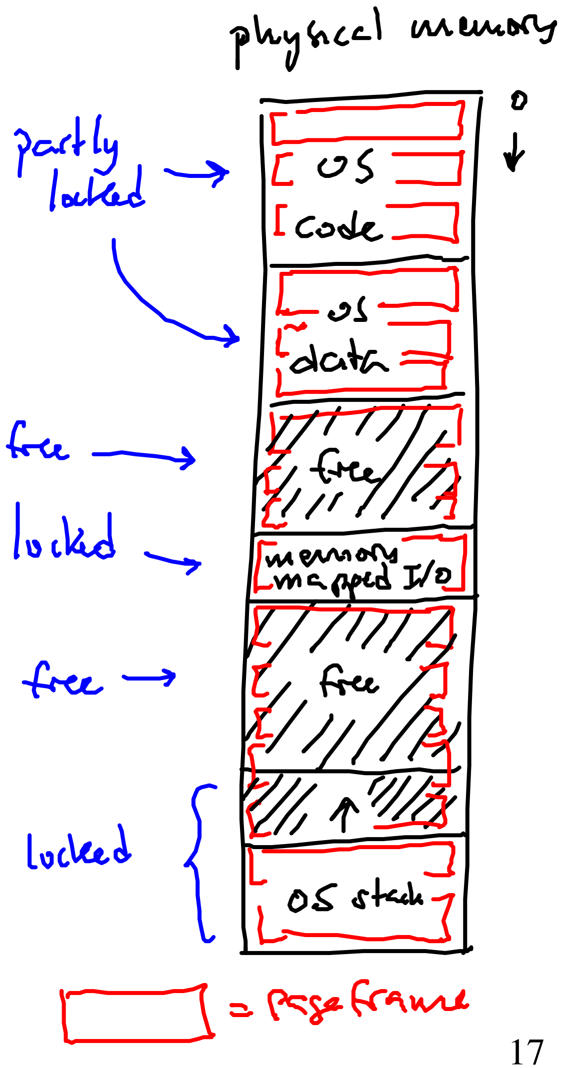
```
for (frame_id i = 0; i < NUMBER_OF_FRAMES; i++) {
  frame_table[i].state = free;
  frame_table[i].referenced = FALSE;
  frame_table[i].written = FALSE;
  frame_table[i].pflags.allow_read = FALSE;
  frame_table[i].pflags.allow_write = FALSE;
  frame_table[i].pflags.allow_exec = FALSE;
  frame_table[i].page = null;
}
```

Overview

- Main memory as cache
- Page descriptors and frame descriptors
- **Page allocation at system startup**
- Page replacement
  - FIFO
  - Second chance
  - Clock
  - Third chance

Layout of Physical Memory

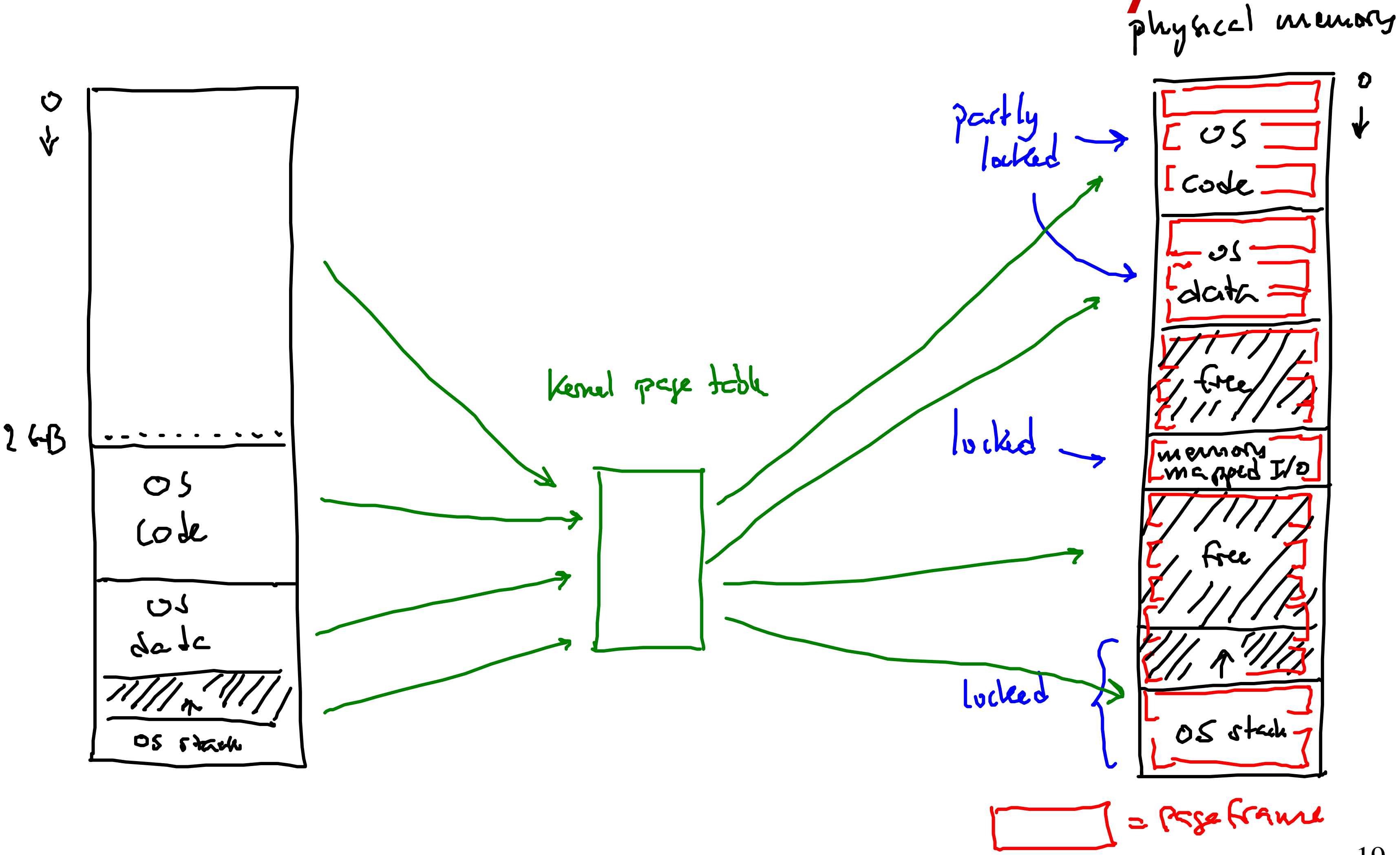
- Physical memory is filled by boot loader
- Frames are “spread over” physical memory
    - Those frames that cover critical code/data are locked
- This must be encoded in the frame table at system startup
    - The frame table must be shaped so that it matches reality
- Many frames remain free



Frame Table in Memory

- The frame table is kept somewhere in memory
- This part of memory must also be locked down
- Self-reference: Frame table contains frame descriptor of the memory in which it resides

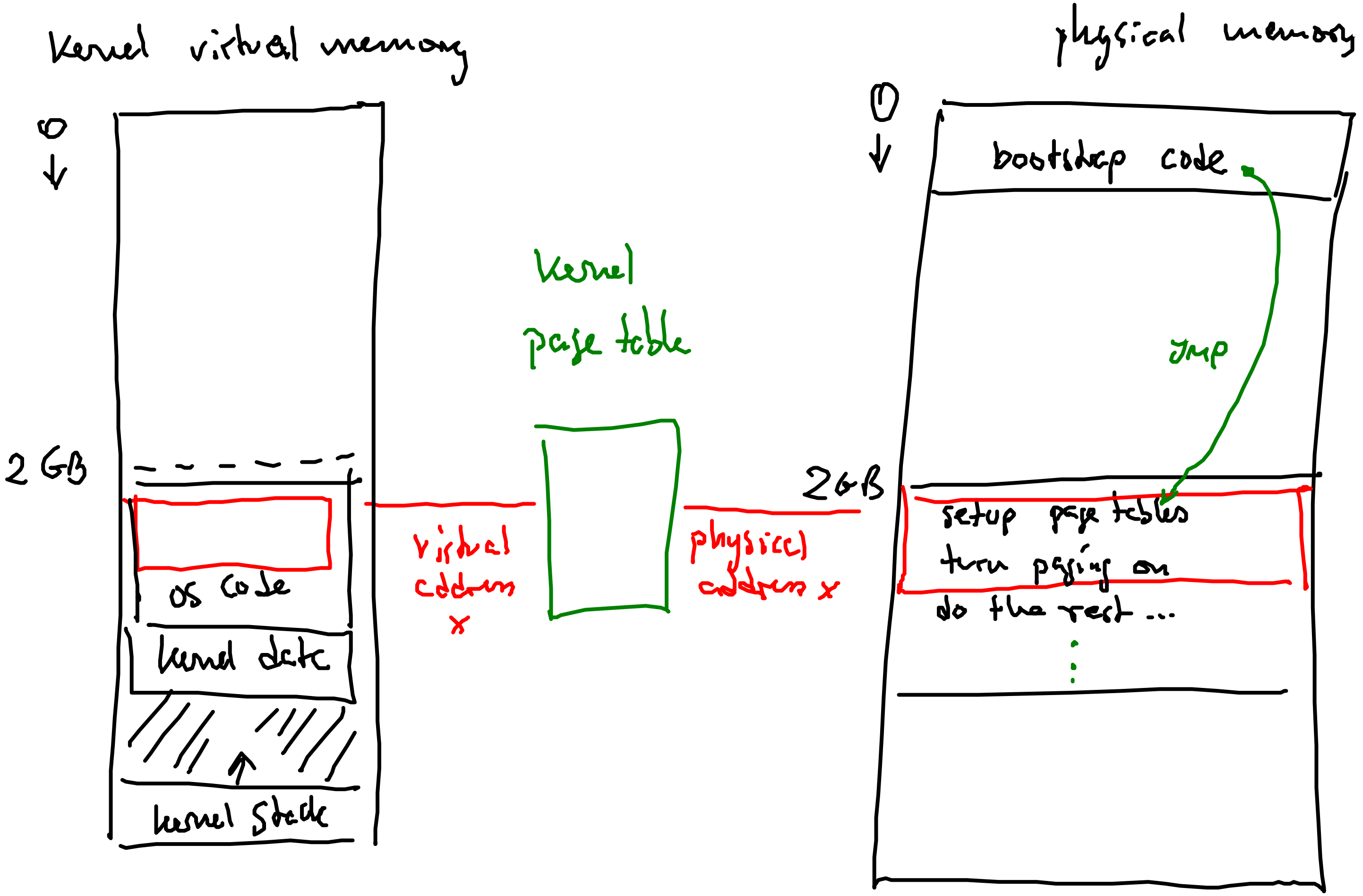
Kernel Virtual Memory



Discussion

- Kernel runs in its own virtual memory
- Page table of kernel is constructed at system startup
- When paging is turned on for the first time, this page table becomes active
  - Might result in a jump (see exercise)
- Bootstrap code must run in an area of transparent paging
  - Virtual addresses are equal to physical addresses

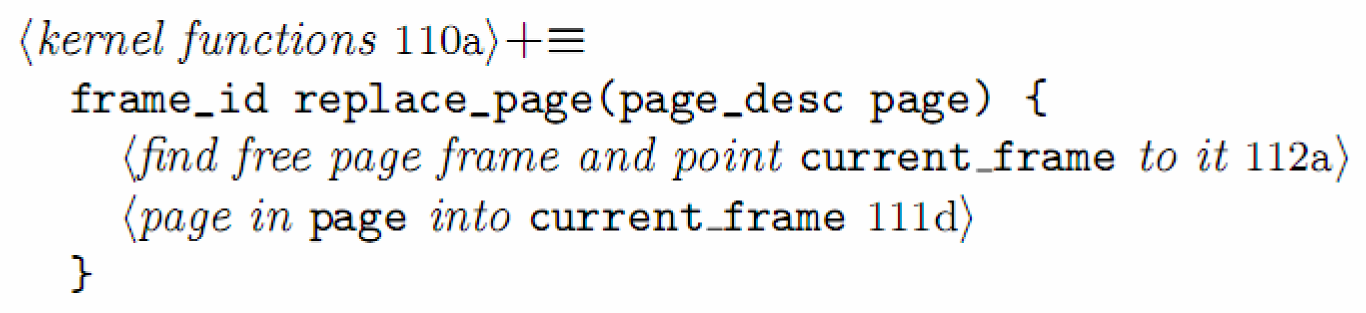
Bootstrapping Kernel Virtual Memory



Overview

- Main memory as cache
- Page descriptors and frame descriptors
- Page allocation at system startup
- **Page replacement**
  - FIFO
  - Second chance
  - Clock
  - Third chance

Paging-in a Page

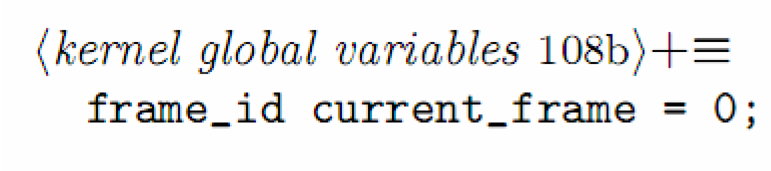


```
⟨kernel functions 110a⟩+≡
  frame_id replace_page(page_desc page) {
    ⟨find free page frame and point current_frame to it 112a⟩
    ⟨page in page into current_frame 111d⟩
  }
```

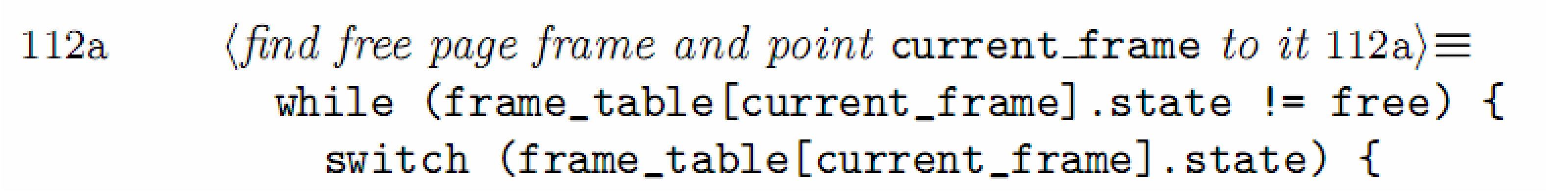
- If kernel needs to page-in a page **`page`**, it calls **`replace_page`**
- Function returns the **`frame_id`** of the page frame into which **`page`** was paged
- Two steps:
  - Find a free frame (page something out if necessary)
  - Page-in the requested page

Preparing FIFO

- Pointer to next replacement candidate



```
⟨kernel global variables 108b⟩+≡
  frame_id current_frame = 0;
```

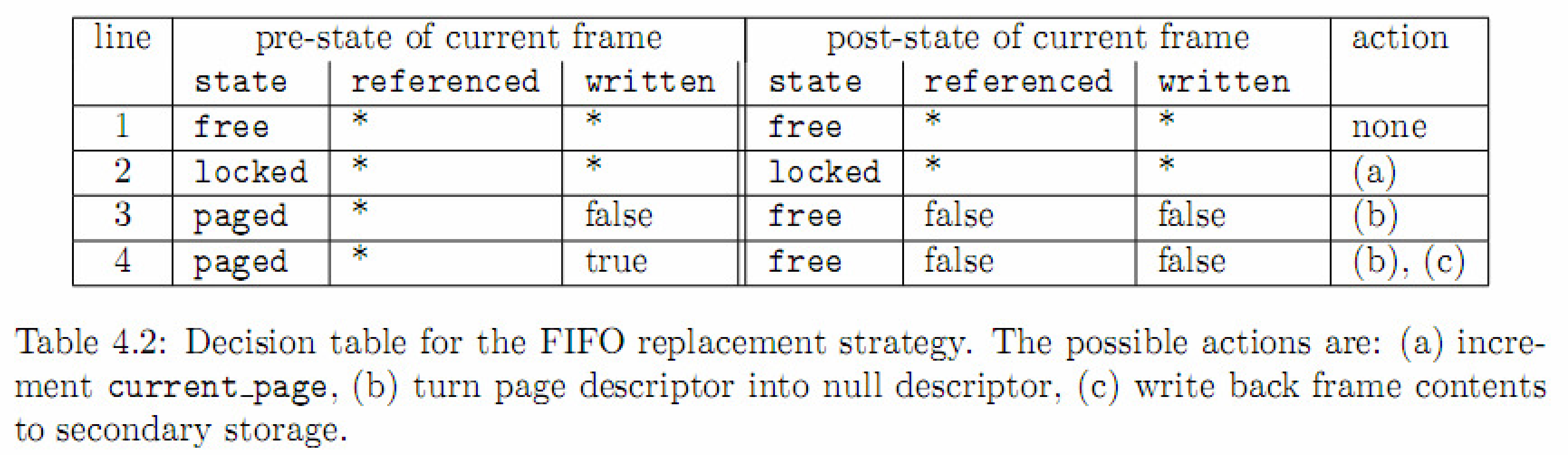


```
112a      ⟨find free page frame and point current_frame to it 112a⟩≡
            while (frame_table[current_frame].state != free) {
              switch (frame_table[current_frame].state) {
```

Check/modify current frame and **`current_frame`**

```
              }
            }
```

Implementing FIFO

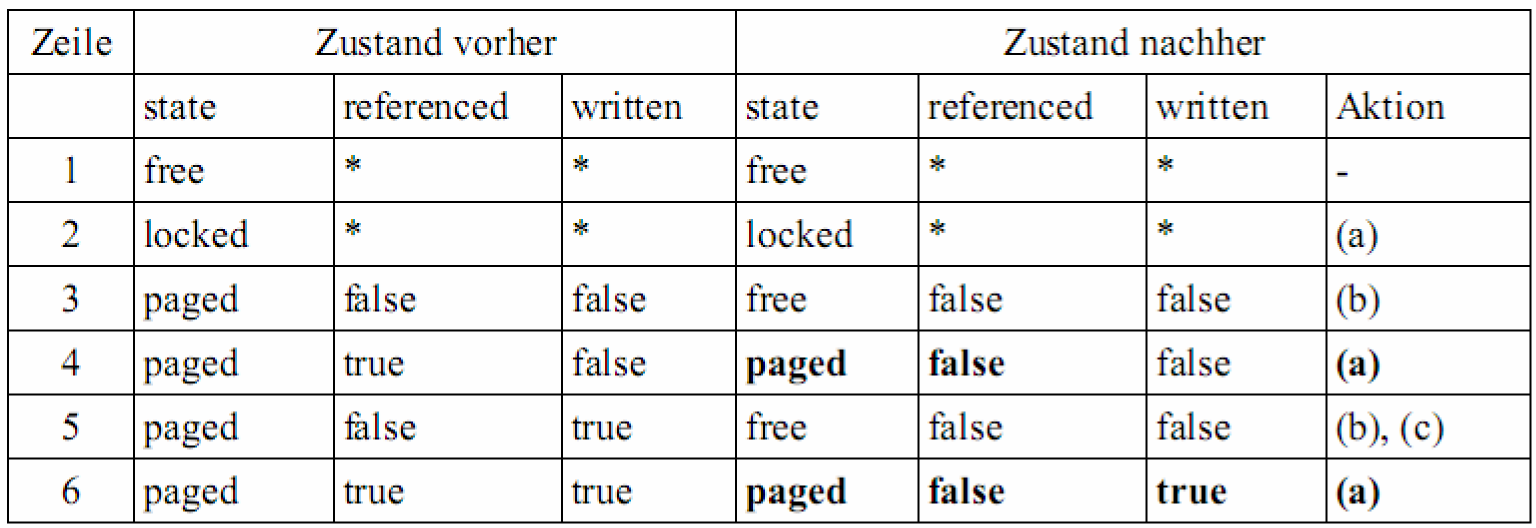


line	pre-state of current frame			post-state of current frame			action
	`state`	`referenced`	`written`	`state`	`referenced`	`written`	
1	`free`	*	*	`free`	*	*	none
2	`locked`	*	*	`locked`	*	*	(a)
3	`paged`	*	false	`free`	false	false	(b)
4	`paged`	*	true	`free`	false	false	(b), (c)

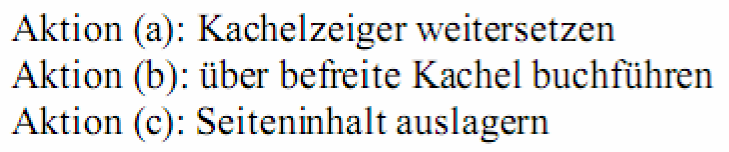
Table 4.2: Decision table for the FIFO replacement strategy. The possible actions are: (a) increment `current_page`, (b) turn page descriptor into null descriptor, (c) write back frame contents to secondary storage.

- Implemented as a decision table or nested switch statement

Implementing Second Chance



Zeile	Zustand vorher			Zustand nachher			
	state	referenced	written	state	referenced	written	Aktion
1	free	*	*	free	*	*	-
2	locked	*	*	locked	*	*	(a)
3	paged	false	false	free	false	false	(b)
4	paged	true	false	**paged**	**false**	false	**(a)**
5	paged	false	true	free	false	false	(b), (c)
6	paged	true	true	**paged**	**false**	**true**	**(a)**



Aktion (a): Kachelzeiger weitersetzen
Aktion (b): über befreite Kachel buchführen
Aktion (c): Seiteninhalt auslagern

- Changes to FIFO are highlighted

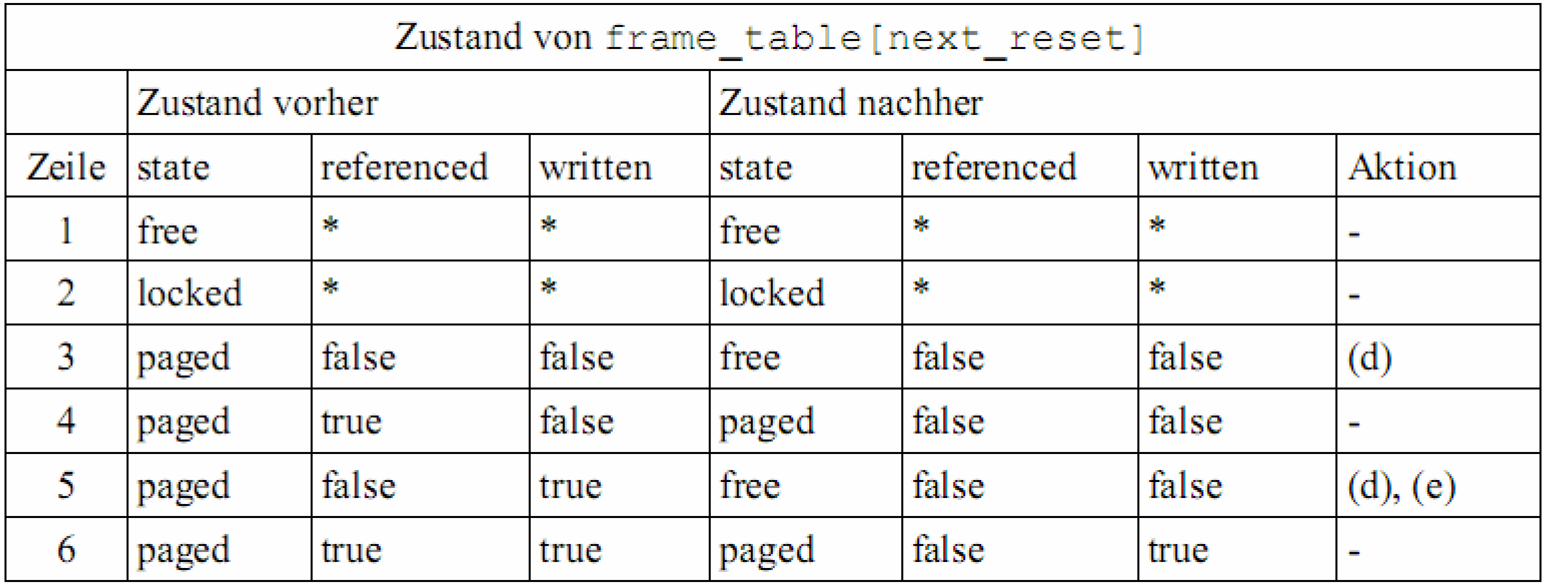
Implementing Clock

- Define additional counter: current_reset
  - Points to next frame that should be reset
- Invariant:

```
(current_reset + d) mod MAX_FRAMES = current_frame
```

- First step: test **`frame_table[current_frame]`**
  - If free, finished
  - Else manipulate **`frame_table[current_reset]`** and increment both counters

Manipulating
`frame_table[current_reset]`



Zustand von `frame_table[next_reset]`							
	Zustand vorher			Zustand nachher			
Zeile	state	referenced	written	state	referenced	written	Aktion
1	free	*	*	free	*	*	-
2	locked	*	*	locked	*	*	-
3	paged	false	false	free	false	false	(d)
4	paged	true	false	paged	false	false	-
5	paged	false	true	free	false	false	(d), (e)
6	paged	true	true	paged	false	true	-

Aktion (d): über veränderten Kachelzustand buchführen:

Aktion (e): Seiteninhalt auslagern

Third Chance

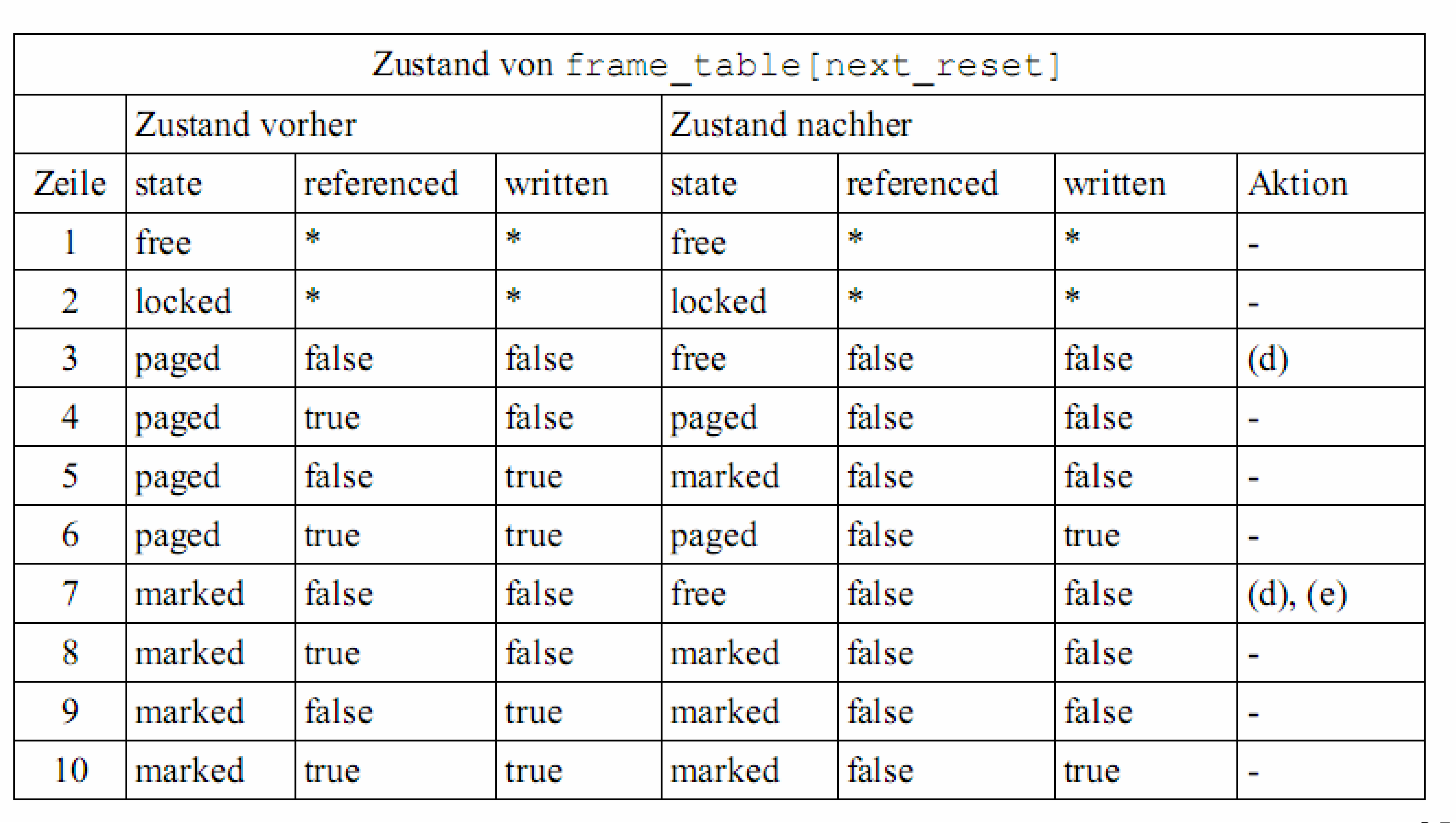
- Variant of Second Chance/Clock
  - Pages that are referenced get second chance
  - Pages that are dirty get third chance
- Try to avoid writing them back to disk

- Can implement it like Second Chance:
  - Test written flag
  - If true, reset flag and go to next candidate (third chance)

Implementation Third Chance

- Problem: Resetting the written flag may result in a dirty page not being written back to disk
- Need to remember, that written flag was previously set
  - New state of frame: marked

Decision Table for Third Chance



Zustand von `frame_table[next_reset]`							
	Zustand vorher			Zustand nachher			
Zeile	state	referenced	written	state	referenced	written	Aktion
1	free	*	*	free	*	*	-
2	locked	*	*	locked	*	*	-
3	paged	false	false	free	false	false	(d)
4	paged	true	false	paged	false	false	-
5	paged	false	true	marked	false	false	-
6	paged	true	true	paged	false	true	-
7	marked	false	false	free	false	false	(d), (e)
8	marked	true	false	marked	false	false	-
9	marked	false	true	marked	false	false	-
10	marked	true	true	marked	false	true	-

Summary

- Main memory as cache
- Page descriptors and frame descriptors
- Page allocation at system startup
- Page replacement
    - FIFO
    - Second chance
    - Clock
    - Third chance

Outlook

- Only 60% implemented
- Not tested, but feels good

- Challenging and not discussed here: system setup/bootstrapping