Chapter 6: Memory Systems

Smruti R Sarangi
IIT Delhi

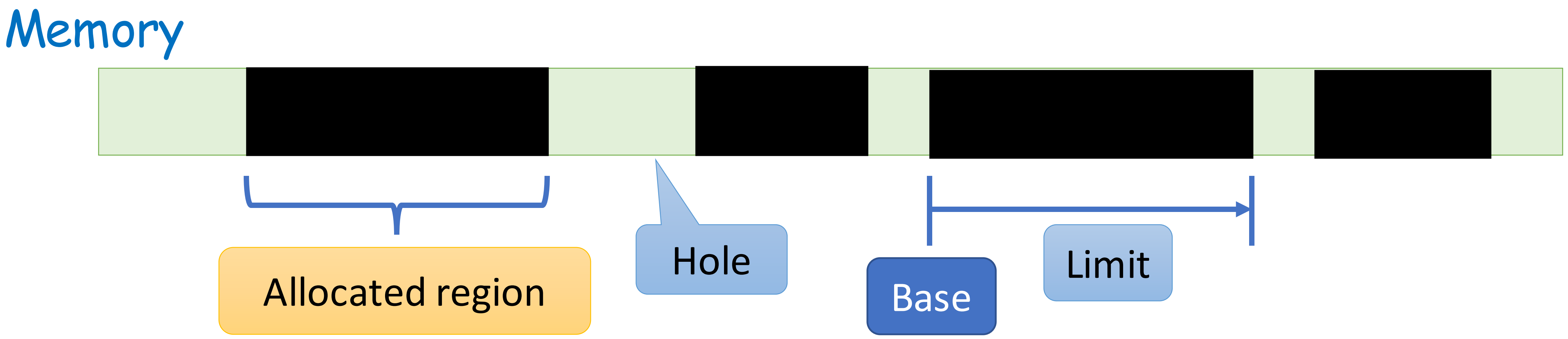
Outline of this Chapter

- Traditional Heuristics
- Virtual and Physical Address Spaces
- Page Management
- Kernel Memory Allocation

Heuristics for memory allocation: no virtual memory

Simple Memory Allocation

- Consider the era that did not have virtual memory
- OR systems that don't have virtual memory
- OR the parts of the kernel that need to use physical memory



Fragmentation

Space wastage

Internal Fragmentation

- Space wasted within an allocated region. Let us say that we waste the last 4 KB in a region.

External Fragmentation

- Holes between regions.

Algorithms to Allocate Space

- Let’s say that there is a request for a memory region R

Memory

Which hole do we fill?

Several Solutions

Algorithms to Allocate Space – II

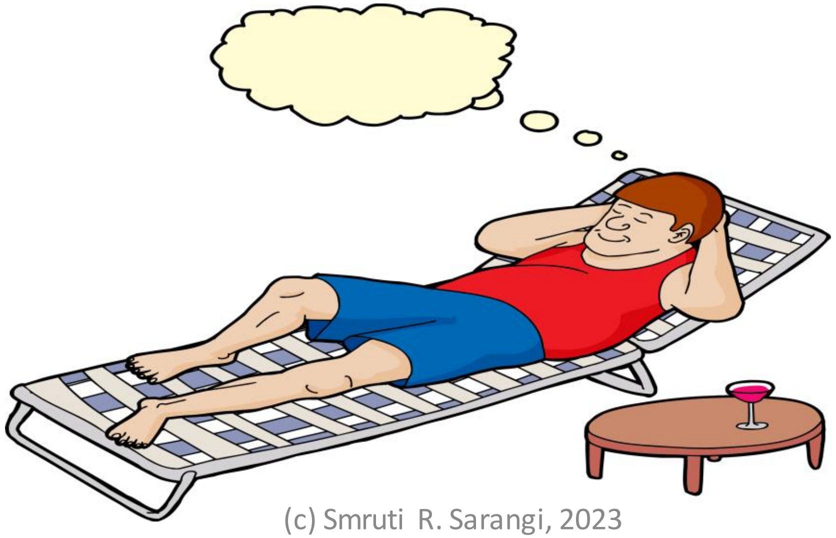
Best Fit	• Choose the smallest hole that is just about larger than *R*.
Worst Fit	• Choose the largest hole.
Next Fit	• Start searching from the last allocation that was made towards higher addresses (with wraparounds).
First Fit	• Choose the first available hole

Heuristics for memory allocation: with virtual memory

What about allocating free frames?

- We can only have internal fragmentation with virtual memory
- Use a bitmap or an Emde Boas tree to manage the list of free frames
- The main issue here is page replacement

> If the main memory is full, which frame in memory should be sent to the swap space?

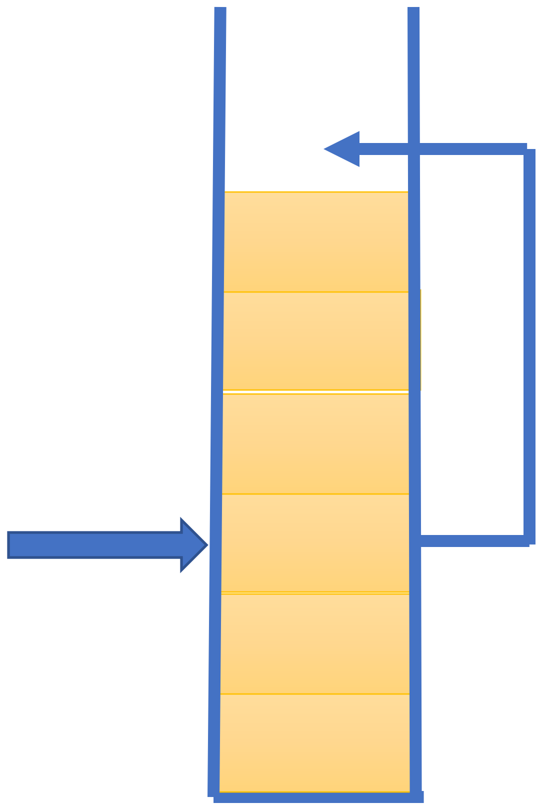


General Stack-based Algorithms

The notion of the stack distance

Hypothetically

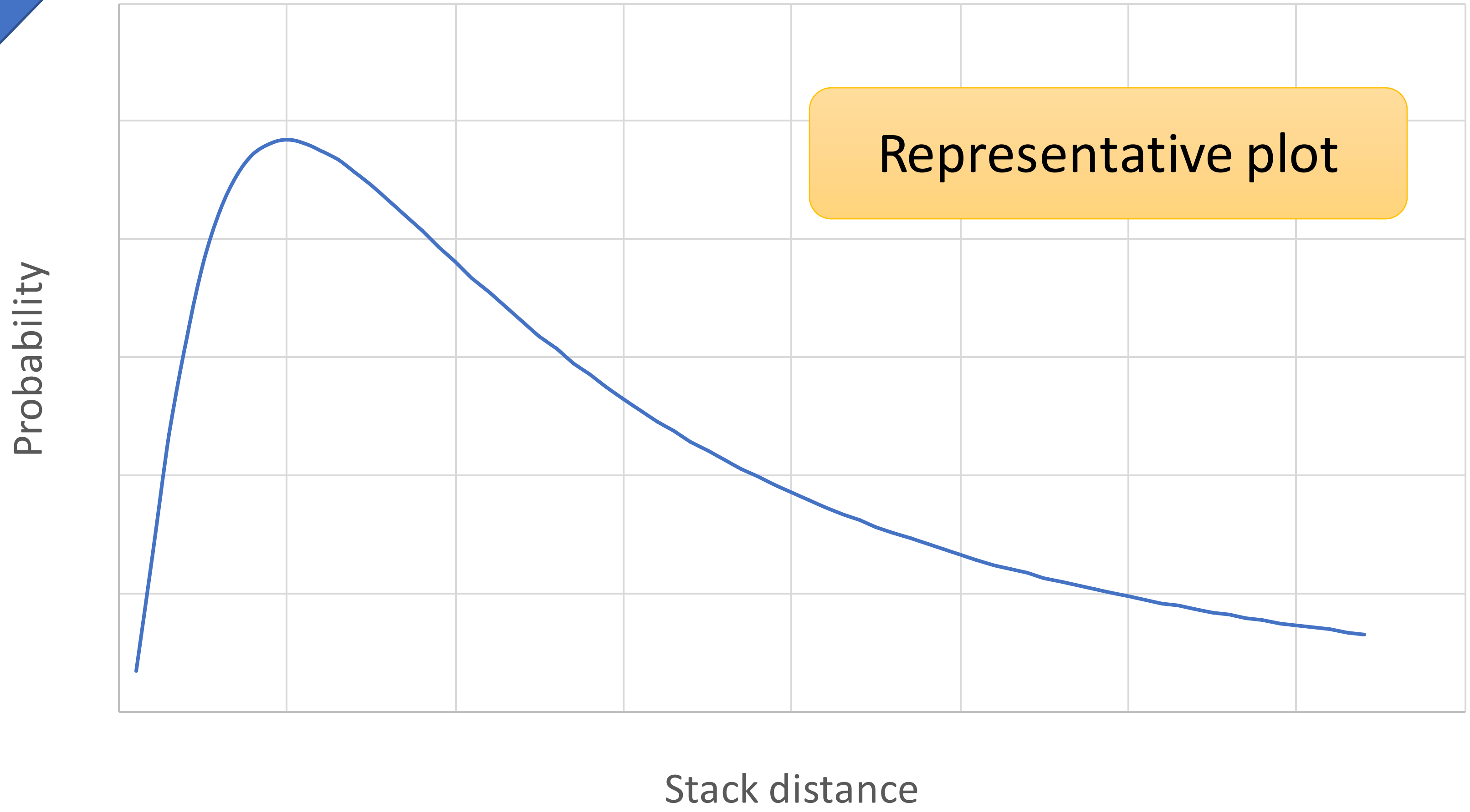
1. Maintain a stack of all page accesses
2. Whenever there is a page access, locate the entry in the stack.
3. Record the distance from the top of the stack → the stack distance.
4. Bring it to the top.



Typical Stack Distance Plot

A heavy tailed curve. Matches a log-normal distribution (most of the time)

Example



Significance of the Stack Distance

- It is a measure of temporal locality
- Higher is the stack distance, lower is the temporal locality and vice versa
- The log-normal curve implies the following:
  - There are very few accesses with ultra-small stack distances
  - There is a distinct peak and a long tail

+ Easy to interpret theoretical tool

− Practical algorithms are hard to realize

Optimal Algorithm

- Cost of a page replacement algorithm = Number of page faults

Hypothetical Optimal Algorithm

1. Order all the pages in increasing order of "next use" time. Assume you can predict the future.
2. Replace the page that will be accessed the last.

Use the same contradiction-based technique to prove optimality.

This is a stack-based algorithm. The replacement is done based on the stack distance.

The LRU (Least Recently Used) Algorithm

Impractical !!!

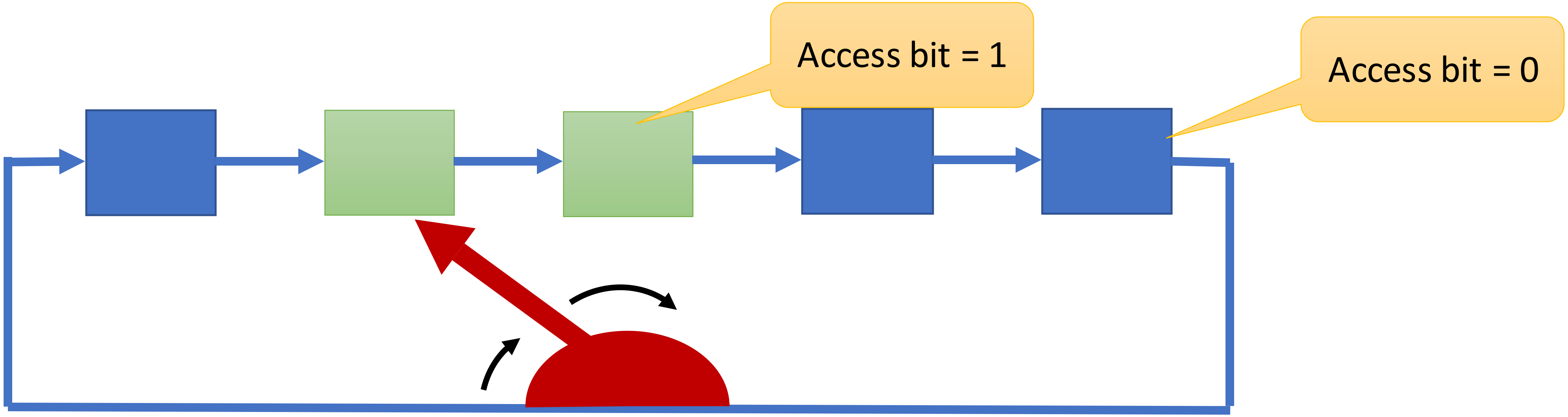
- Tag each page (in the memory) with the last time that it was accessed.
- Augment each entry in the page table with a timestamp.
- Choose the page whose last access time is the earliest
- Let us assume that the past is a good predictor of the future.
- These algorithms come with many theoretical guarantees

Every memory access cannot increment a counter. The overheads are prohibitive.

Such Stack-based Algorithms can be made Practical

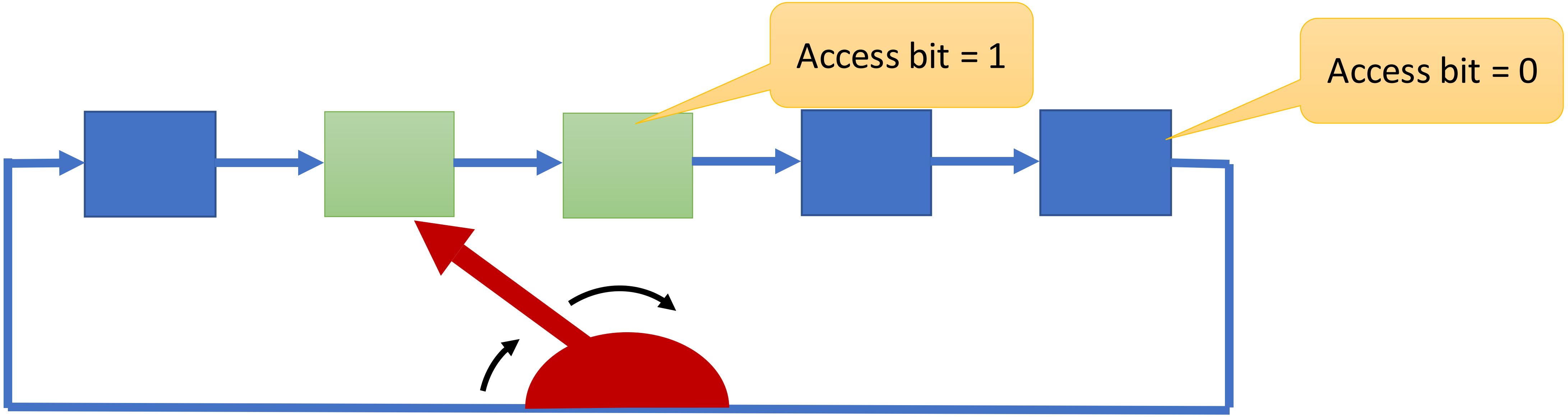
- Order all the physical pages (frames) in memory in some order (maybe FIFO order)
- Leverage their protection bits. Mark them "no access". Access bit = 0
- When we access a page with its access bit set to 0, there is a fault
  - Set it to 1
- Periodically, set all access bits to 0
- OR, alternatively record the time that it was set from 0 → 1. Reset it to 0 only if a certain **duration** has been exceeded.

WS_Clock Page Replacement Algorithm



- A pointer (like the minute hand of a clock) points to a physical page
- If its (access bit = 1) set it to 0
- Otherwise, replace it
- Next time, start the pointer from the same point and wraparound at the end

WS_Clock based Second Chance Algorithm



<Access bit, Modified bit>	New state	Action
<0, 0>	<0, 0>	Go ahead and replace
<0, 1>	<0, 0>	Schedule a write-back
<1, 0>	<0, 0>	Move forward
<1, 1>	<1, 0>	Frequently used frame; move forward. Schedule a write-back.

FIFO and the Belady's Anomaly [1]

10 faults

Consider the FIFO page replacement algorithm

Access sequence	1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	4	4	5	1	2	3	4	5
		1	2	3	3	3	4	5	1	2	3	4
			1	2	2	2	3	4	5	1	2	3
				1	1	1	2	3	4	5	1	2

4 frames

FIFO and the Belady's Anomaly

9 faults

Consider the FIFO page replacement algorithm

Access sequence	1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	5	5	3	4	4
		1	2	3	4	1	2	2	2	5	3	3
			1	2	3	4	1	1	1	2	5	5

3 frames

Final Word on Page Replacement

- Stack-based algorithms are by and large impractical
- We need to create approximations.
- FIFO and Random replacement algorithms may exhibit the Belady's anomaly
  - The ratio between the faults in a large memory and small memory is unbounded → can be as large as we want it to be [2].
- The clock-based algorithms approximate LRU and are known to work well.

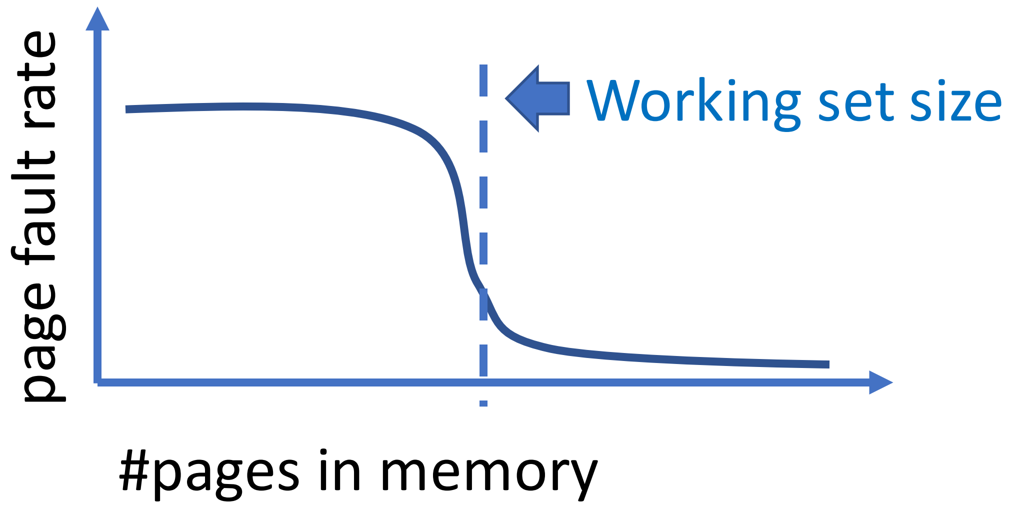
Working Sets

Assume you have a recency-based replacement algorithm

- A working set is a set of pages that a program access over a short duration.

How short is short?

Keep track of the working set and ensure that it is in memory

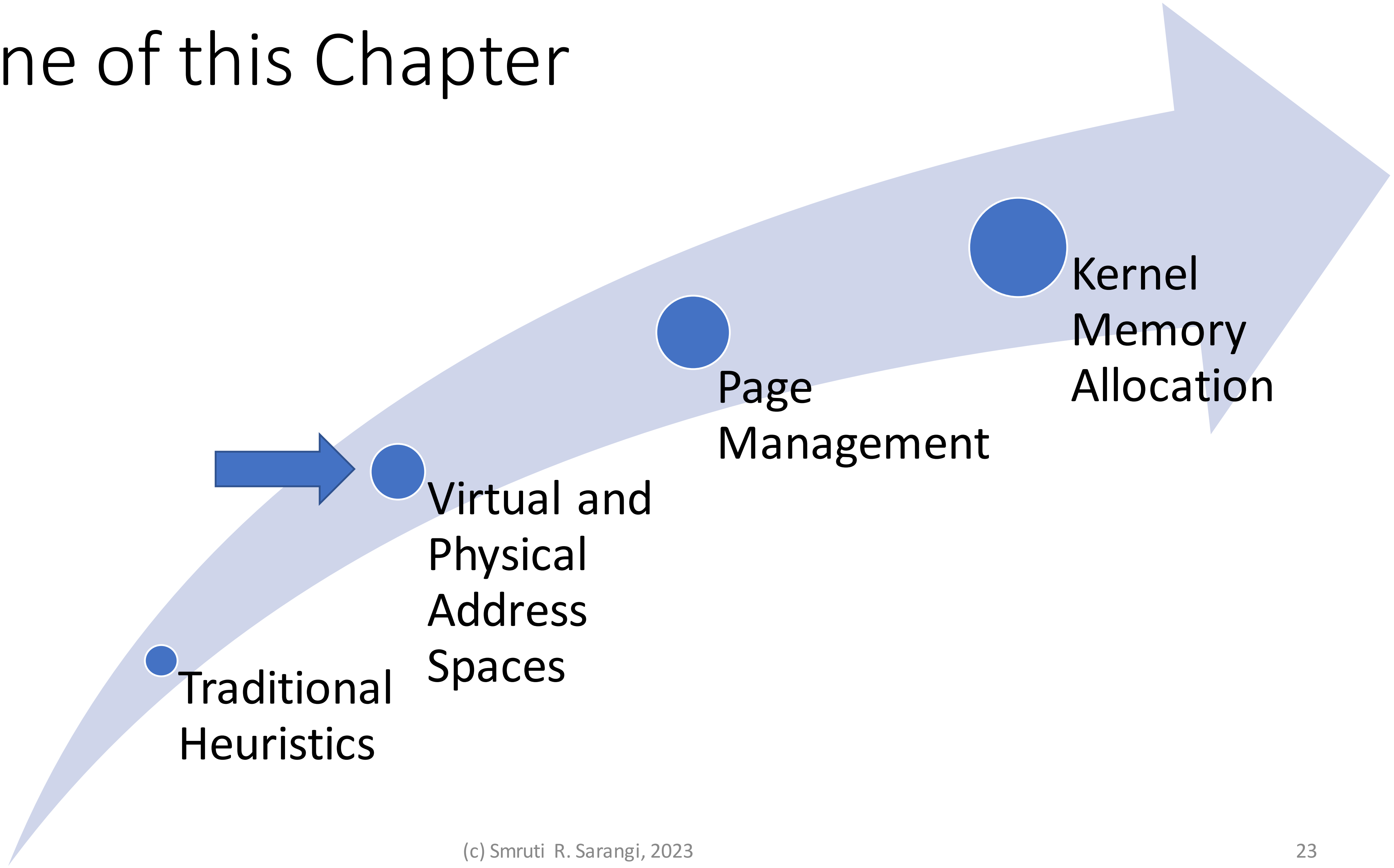


Thrashing

- Consider a system with a lot of processes and a paucity of memory
- If the space allocated to a process is less than its working set
  - The process will spend most of its time fetching its working set
  - The performance counters will indicate low CPU utilization
  - The kernel will see the low load average and add more processes to the CPU's run queue
  - This will make the problem even worse
- Ultimately the system will crash

This is called thrashing.

Outline of this Chapter



Let us start from *mm_struct*

```
struct mm_struct {
    ....
    pgd_t *pgd;
    ...
};
```

Pointer to the page table. The CR3 register is set to this value. Type: u64

PGD 57-49 → P4D 40-48 → PUD 31-39 → PMD 22-30 → PTE 13-21 → Page 1-12

CR3 register points to the PGD of the current process

Explanation

/arch/x86/include/asm/pgtable_types.h

5-Level Page Tables

Acronym	Full Form
PGD	Page Global Directory
P4D	Fourth level page table
PUD	Page Upper Directory
PMD	Page Middle Directory
PTE	Page Table Entry

57-bit virtual address

128 PB VA space

The variable *pgprot_t* contains the protection bits

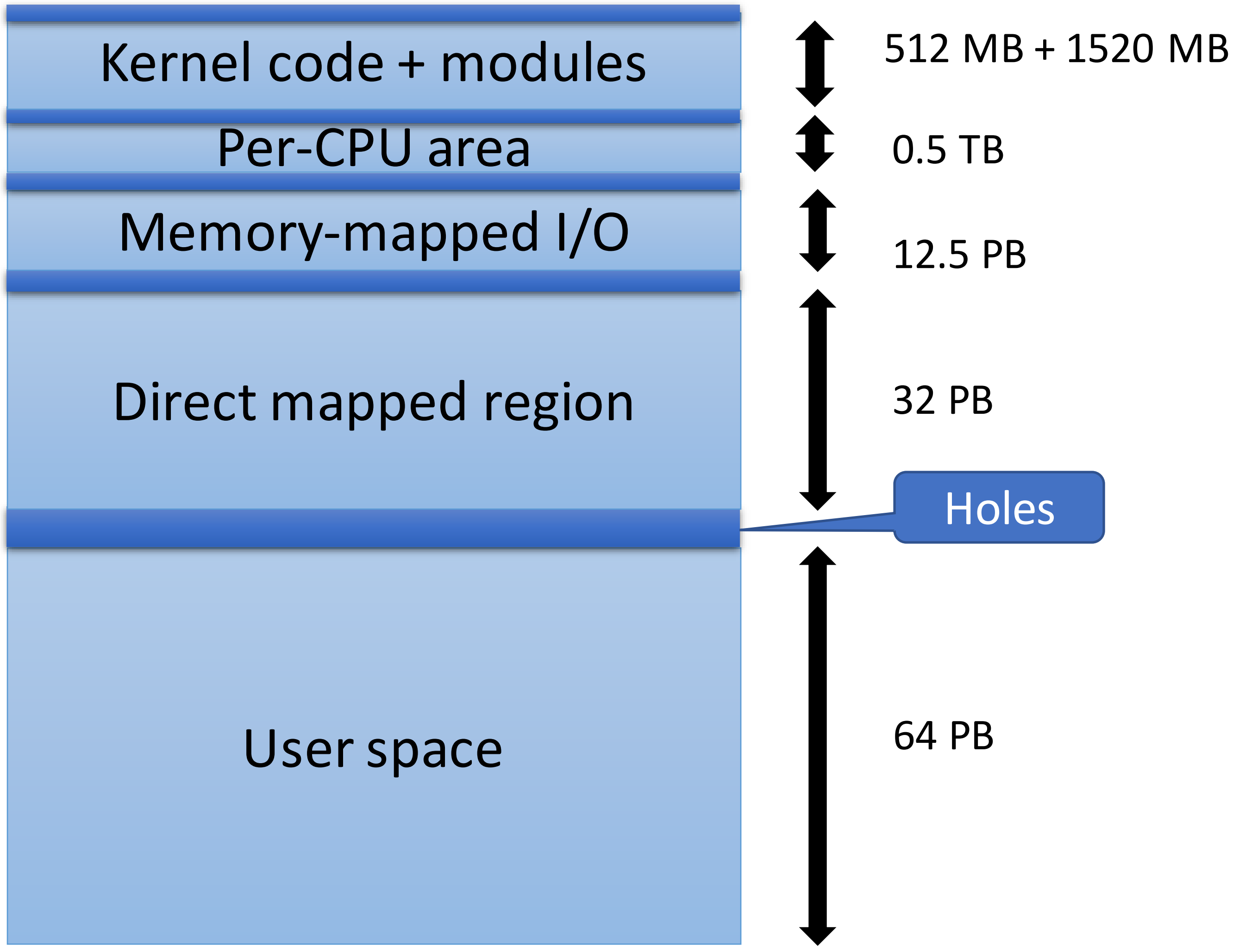
Acronym	Full Form
PROT_READ	Read permission
PROT_WRITE	Write permission
PROT_EXEC	Execute permission
PROT_SEM	Can be used for atomic ops
PROT_NONE	Page cannot be accessed

/include/uapi/asm-generic/mman-common.h

Virtual Memory Map

Documentation/x86/x86_64/mm.rst

- The memory map is not drawn to scale and many regions have not been shown.
- There are holes between regions. If any of the holes are written to, it is a fault.
- The direct-mapped zone can be used to access the physical memory directly (albeit by subtracting an offset)
- The kernel text (code) starts at physical address 0



The *follow_pte* function

/mm/memory.c

Key function used to **traverse** the page table

```
int follow_pte(struct mm_struct *mm, unsigned long address,
               pte_t **ptepp, spinlock_t **ptlp) {
    pgd_t *pgd;
    p4d_t *p4d;
    pud_t *pud;
    pmd_t *pmd;
    pte_t *ptep;

    pgd = pgd_offset(mm, address);
    p4d = p4d_offset(pgd, address);
    pud = pud_offset(p4d, address);
    pmd = pmd_offset(pud, address);
    ptep = pte_offset_map_lock(mm, pmd, address, ptlp);

    *ptepp = ptep;
    return 0;
}
```

This code does not show the cases where an entry does not exist.

Walk the page table

Set the return value. Keep the page locked.

Accessing any given level (let's say PMD)

```
// /include/linux/pgtable.h
pmd_t *pmd_offset(pud_t *pud, unsigned long address) {
    return pud_pgtable(*pud) + pmd_index(address);
}

unsigned long pmd_index(unsigned long address) {
    return (address >> PMD_SHIFT) & (PTRS_PER_PMD - 1);
}

// /arch/um/include/asm/pgtable-3level.h
#define pud_pgtable(val) ((pmd_t *) __va(pud_val(val) & PAGE_MASK))

// /arch/x86/include/asm/page.h
#define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

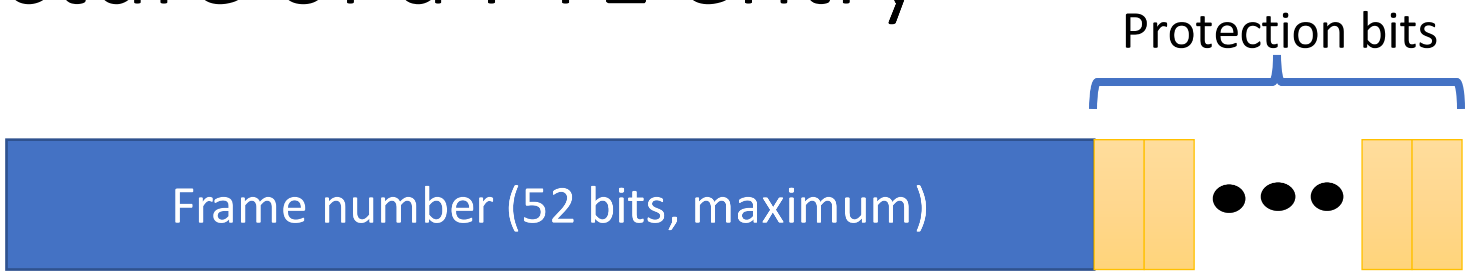
Extract the index from the address.

$\sim(2^{12} - 1)$

Starting virtual address of the kernel

/include/linux/mm_types.h

Structure of a PTE entry



- Some other important data structures that are a part of the memory subsystem
  - page → Data structure for every physical page (frame) in main memory. The aim is to record (to some extent) who is using the page and what for.
  - folio → Represents a contiguous set of bytes (physically and virtually). Its size is a power of two and is ≥ the page size.

/include/linux/mm_types.h

struct page

- flags
- One large union (20/40 bytes): it can store a bunch of things. Choices:
  - Pointer to the address space (I/O device whose pages are mapped)
  - Pointer to a pool of pages
  - Page map (mapped from a dedicated device or DMA pages)
- Reference count

This is a classic example of a data structure that has a very flexible structure: it can store anything (depending upon the end user).

struct folio

https://lwn.net/Articles/893512/

https://lwn.net/Articles/849538/

- Definition: A compound page is an aggregate of two or more contiguous pages
- A folio primarily points to a compound page.
  - It is primarily needed to manage millions of pages in large memories
- It points to the first page in a compound page
- It is very useful for memory mapped I/O (I/O devices and files)
- It is naturally aligned towards read prefetching and sequential writes
- Writes and modification bits can be maintained at the folio level
  - Easier to maintain LRU-based replacement lists

Folio of pages

- A folio acts like a single page
- It has its permission bits and copy-on-write state
- Whenever a page needs to be migrated, swapped out, or replicated (because COW)
  - If a page is a part of a folio, then the entire operation happens on the folio
- There used to be a concept called huge pages (still is)
  - We can have pages with size 2 MB and 1 GB
  - Some server processors support huge pages. This requires changes to addressing or multiple entries are created in the TLB (one for each page). The latter solution is very expensive.

pte → pfn → page

```
#define pte_pfn(x) phys_to_pfn(x.pte)
#define phys_to_pfn(p) ((p) >> PAGE_SHIFT)
```

```
#define __pfn_to_page(pfn)                    \
({  unsigned long __pfn = (pfn);              \
    struct mem_section *__sec = __pfn_to_section(__pfn);    \
    __section_mem_map_addr(__sec) + __pfn;    \
})
```

- The contents of the *pte* (page table entry) contain the physical frame number and other protection information
- *pfn* is just the physical page number that is obtained by shifting the *pte* by PAGE_SHIFT (=12)
- *struct page* corresponds to a physical frame. Physical pages are organized into different sections. Each section points to a memory map (an array of page structures).

TLB

TLB

/arch/x86/mm/tlb.c

- It is important to manage the TLB well.
- More than 99% of the requests are satisfied by the TLB.
- A TLB miss is quite expensive. It involves a costly page table walk.
- In x86 machines
  - By default, it is a 4 to 16-way set associative cache
  - Some processors allow the user to configure the associativity
  - Some processors can also have a 2-level TLB or a separate data TLB and i-TLB
  - Each entry of the TLB (corresponding to a virtual page number) contains a pointer to a physical page number, and has other protection bits (+ other data)
- Flushing the TLB
  - Just modifying the CR3 register flushes the TLB
  - Some entries can still be retained (not flushed) if the G (global) bit is set
  - The *invlpg* instruction can flush the entire TLB, or a specific page, or the pages belong to a process. How ???

Notion of the ASID (PCID)

- A context switch is heavy on the TLB
- The new process suffers from plenty of TLB misses
- The same is true when the old process runs on the core again
- If we can store the *pid* along with every TLB entry, our job is done
- Then two processes can run on the same core (one after the other) without flushing the TLB.
- Let us call this additional annotate the ASID (address space ID), Intel calls it PCID → Processor-Context ID

Problem:

A *pid* is an OS-specific concept, whereas the ASIC/PCID is a hardware concept. How to reconcile both?

Let the Hardware Win

- Maintain a small per-CPU array of PCIDs
- Cache the last few mm (memory maps) for tasks that executed on the CPU
- The PCID bits are set to [1, TLB_NR_DYN_ASIDS]
- TLB_NR_DYN_ASIDS = 6 [default value] (/include/asm/tlbflush.h)
- The PCID is the first 12 bits of the CR3 register
- Each TLB entry also contains the associated PCID
  - It is matched with the current PCID
  - The INVPCID instruction can be used to invalidate all the TLB entries that corresponds to a single PCID

Lazy TLB Mode

https://notes.shichao.io/utlk/ch2/

- Assume several CPUs share a page table
  - One of them runs a call that invalidates an entry
  - This is invalidated for the full process
  - All the TLBs need to be flushed
- Let's say that one of the CPUs is running a kernel thread
  - It need not invalidate the entry immediately
  - It can set the CPU to "lazy TLB mode"
  - The kernel threads don't have separate page tables. It is a common one that is appended with user mode page tables (one large user + kernel page table)

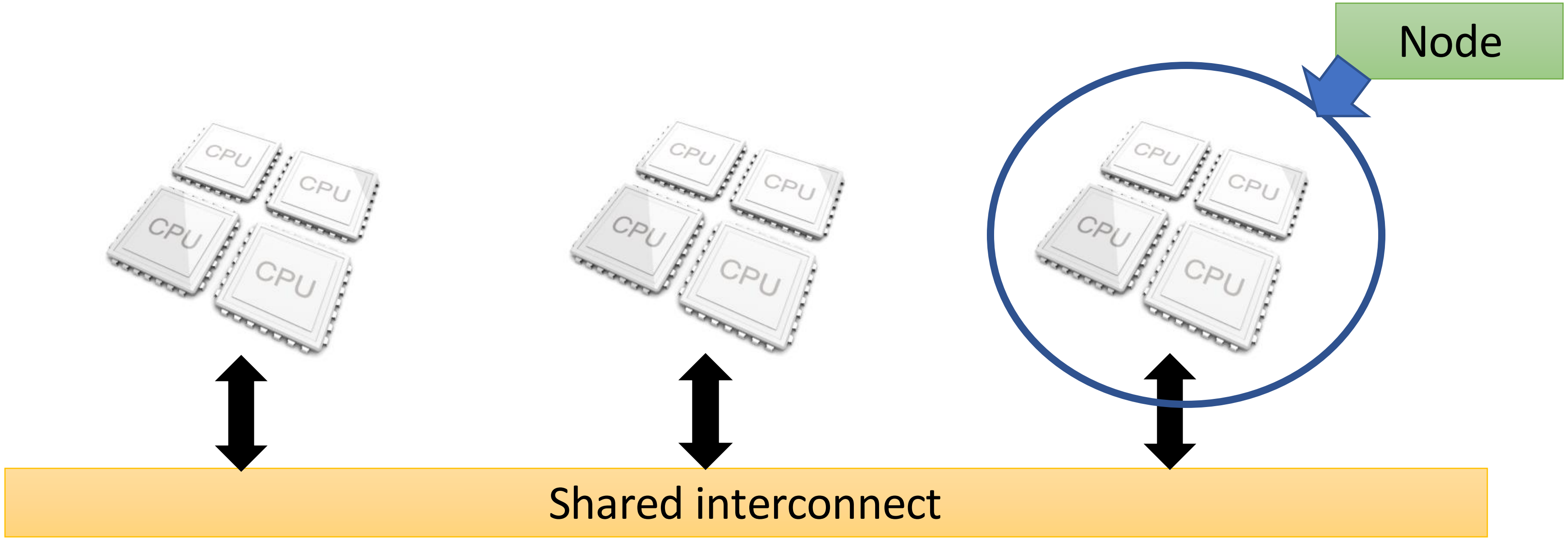
Lazy TLB Mode

Three cases

1. The kernel tries to access the invalidated page. This will happen only via fixed entry points. This cannot happen in an uncoordinated fashion. Appropriate checks can be carried out and exceptions can be thrown.
2. The kernel switches to another user process. In this case, all the TLB entries of the original process are flushed out.
3. The kernel switches back to the same user process. Just finish the work of invalidating all the entries that were deferred.

Partitioning Physical Memory

NUMA Machine

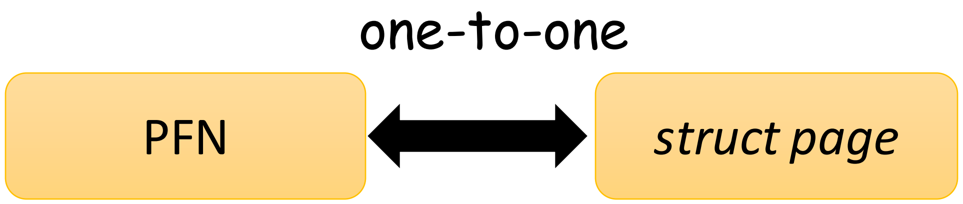


- CPUs have some amount of local memory, which is much faster
- They are also organized into clusters.
- A CPU can access all memory: intra-cluster and inter-cluster
- Accesses to intra-cluster memory is much faster.
- This is a non-uniform memory access machine (NUMA machine)
- Each cluster is known as a node

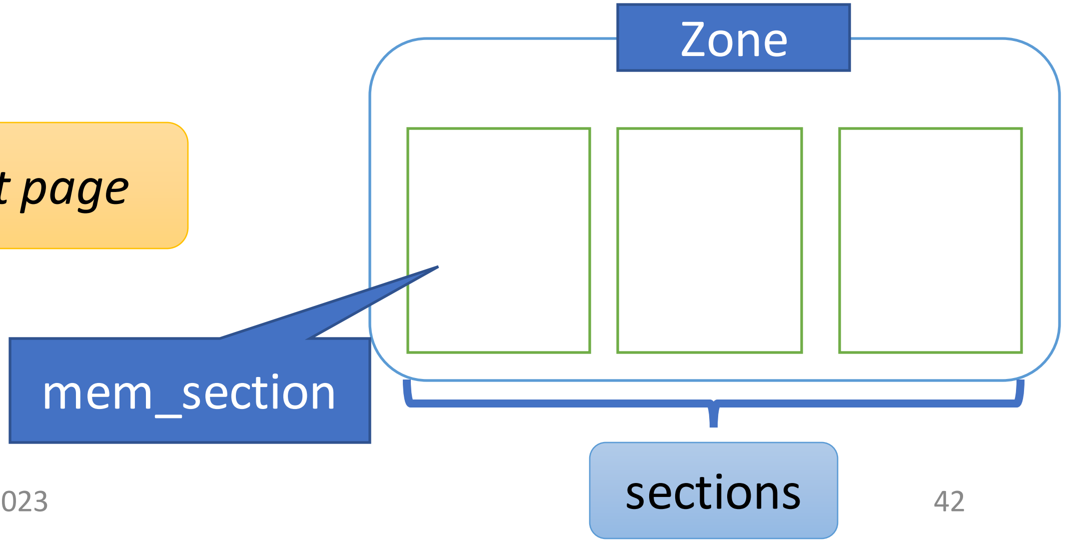
Is all physical memory in a node the same?

https://lwn.net/Articles/789304/

1. Partition the physical memory space (in a node) into zones
2. Treat the frames (physical pages) in each zone differently.
3. The zones may themselves be stored on different devices.
4. I/O Devices may need a dedicated region (assign a zone)



There is a *mem_map* for each **section** that stores this mapping



Physical Memory Zones in Linux

/include/linux/mmzone.h

```
enum zone_type {
    ZONE_DMA,
    ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM
    ZONE_HIGHMEM,
#endif
    ZONE_MOVABLE,
#ifdef CONFIG_ZONE_DEVICE
    ZONE_DEVICE,
#endif
    __MAX_NR_ZONES
};
```

- ZONE_DMA: Physical pages that are accessible only by the DMA controller
- ZONE_NORMAL: Normal frames
- ZONE_HIGHMEM: Useful in systems where the physical memory exceeds the size of max virtual memory.
- ZONE_MOVABLE: It is assumed that the corresponding memory device may be removed at any point of time and possibly re-inserted later.
- ZONE_DEVICE: These frames are stored in novel memory devices like NVM devices.

struct zone

```
struct zone {
    int node;

    struct pglist_data  *zone_pgdat;
    unsigned long       zone_start_pfn;

    atomic_long_t       managed_pages;
    unsigned long       spanned_pages;
    unsigned long       present_pages;

    const char          *name;

    struct free_area    free_area[MAX_ORDER];
}
```

Pointer to the *pglist_data structure*

zone_end_pfn = zone_start_pfn + spanned_pages- 1

Name of the zone

Free areas in a zone (physical memory)

Data Structure to Manage the Memory in a Zone

```
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES];
    struct zonelist node_zonelists[MAX_ZONELISTS];
    int nr_zones;

    unsigned long node_present_pages;
    unsigned long node_spanned_pages;
    int node_id;

    struct task_struct    *kswapd;
    struct lruvec         __lruvec;
} pg_data_t;
```

Ordering of zones

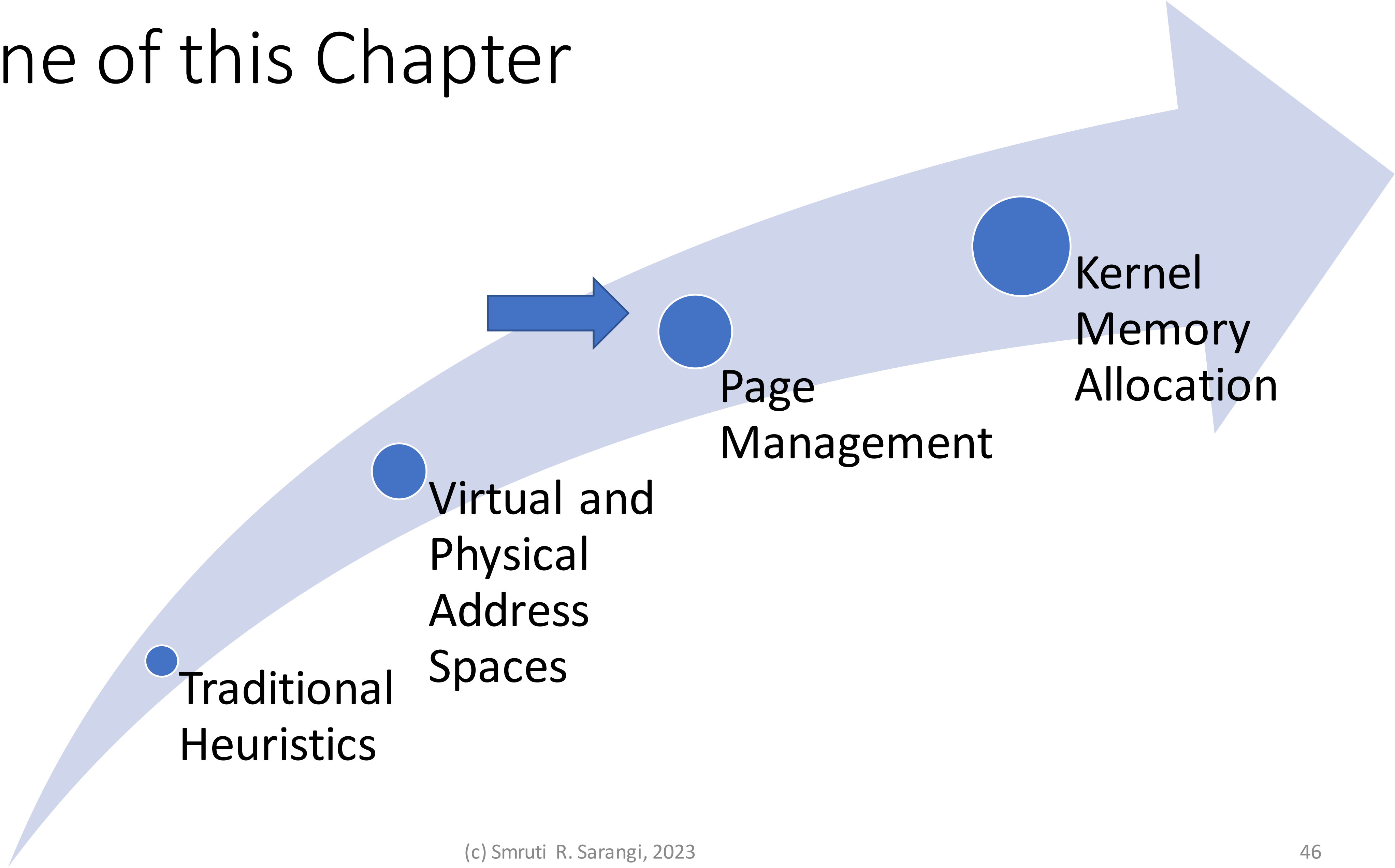
Zones organized hierarchically

Number of pages owned by NUMA node_id

Page swapping daemon

Maintains LRU state information

Outline of this Chapter



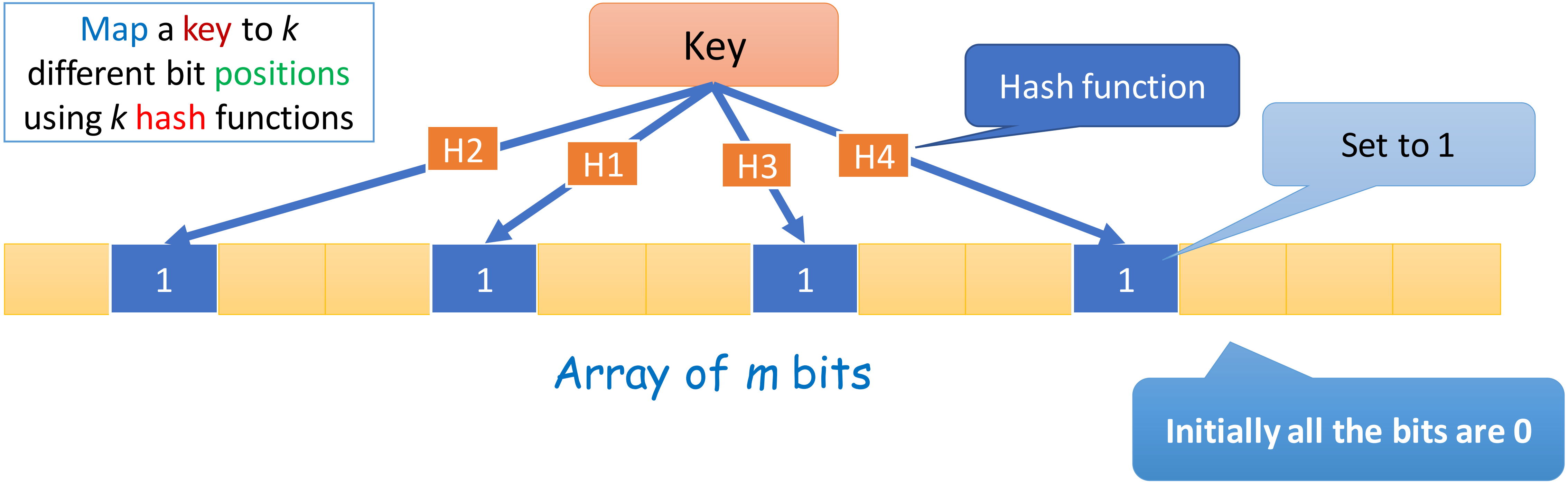
Required Background

- Bloom Filters
- PI Controller
- Reverse Maps

Inserting a Key in a Bloom Filter

A data structure that answers if an element is a member of a set (probabilistically)

Map a key to k different bit positions using k hash functions



Checking for Set Membership in a Bloom Filter

1. Given a key compute the *k* different hash values (bit positions)
2. Check that each bit position stores a 1. If at least one bit position stores a 0, then the key is not present.
3. Elements cannot be deleted unless we store a count at each bit position
4. We need to periodically reset the Bloom Filter

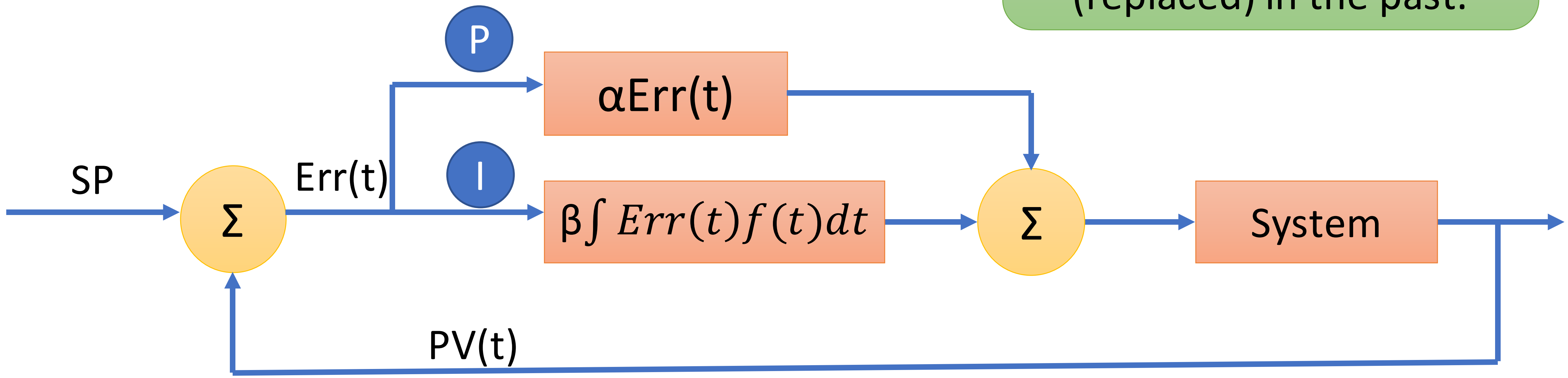
False positives are allowed

No false negatives

Background

The PI Controller

A refault is a page fault for a page that was evicted (replaced) in the past.



Term	Meaning	Explanation in this context
P term	Proportional part	refaulted/ (evicted + pinned)
I term	Integral part	Exponential moving average of P (factor = ½)
SP	Set point	Refault rate of the first tier of ANON
PV	Process variable	Refault rate of the first tier of FILE

Both should be the same (target)

Background

Error Term

```
err =       (pv->refaulted < 64 )||
            pv->refaulted /pv->total * swappiness <=
            (sp->refaulted + 1) / (sp->total + 64)* (200 – swappiness)
```

err = 1 *if*

1. Too few pages have refaulted (< 64)
2. The refault rate (refaulted/total) is lower than the set point (subject to the swappiness)
3. The swappiness is close to 0 (low)

Background

Reverse Mapping (called from *shrink_folio_list)*

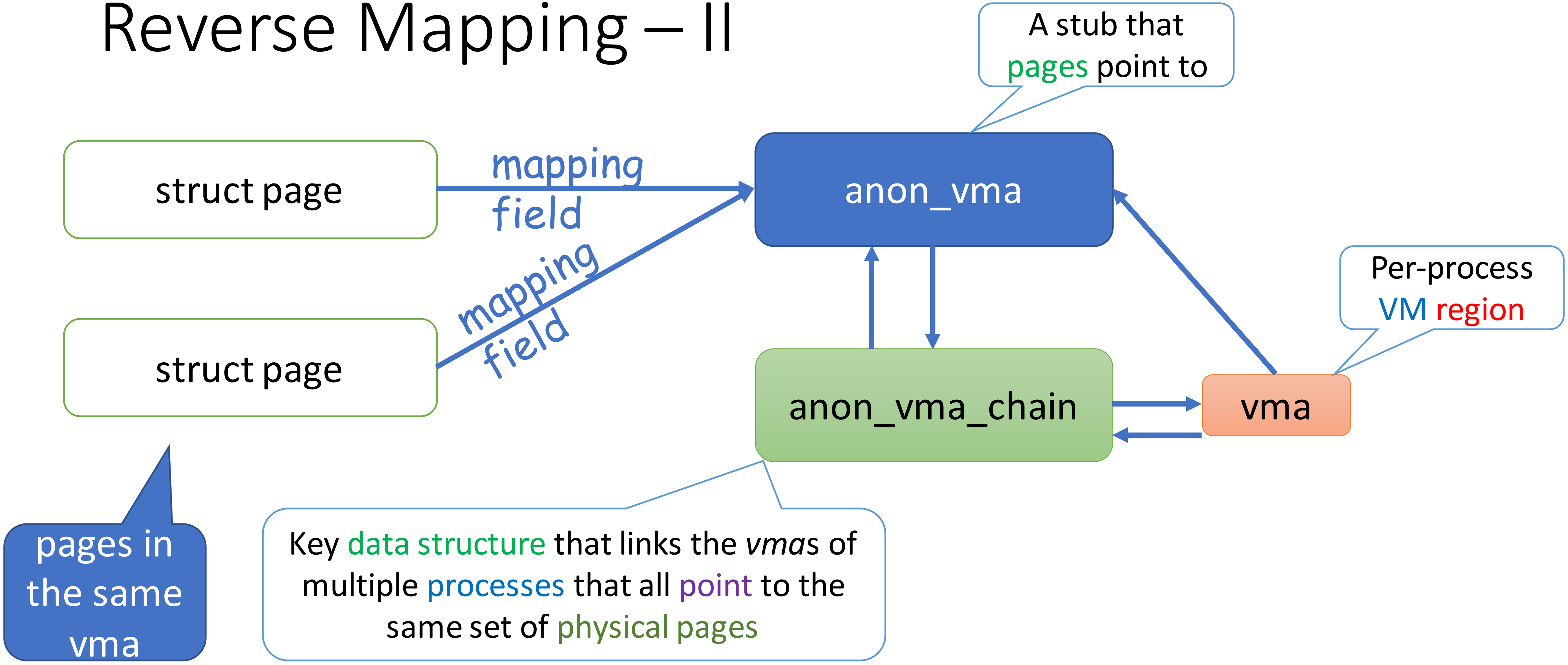
Reverse map (rmap)

struct page → List of PTEs

- While freeing a page (or folio), we need to consider the possibility of it being mapped to multiple address spaces (across processes)
- We would thus need to create a reverse map (rmap)
  - Map a struct page (or folio) to PTEs across processes
  - Swapping out a folio requires a rmap walk
  - Each PTE entry needs to be processed
- The idea is to efficiently associate a list of *vma regions* with each PTE

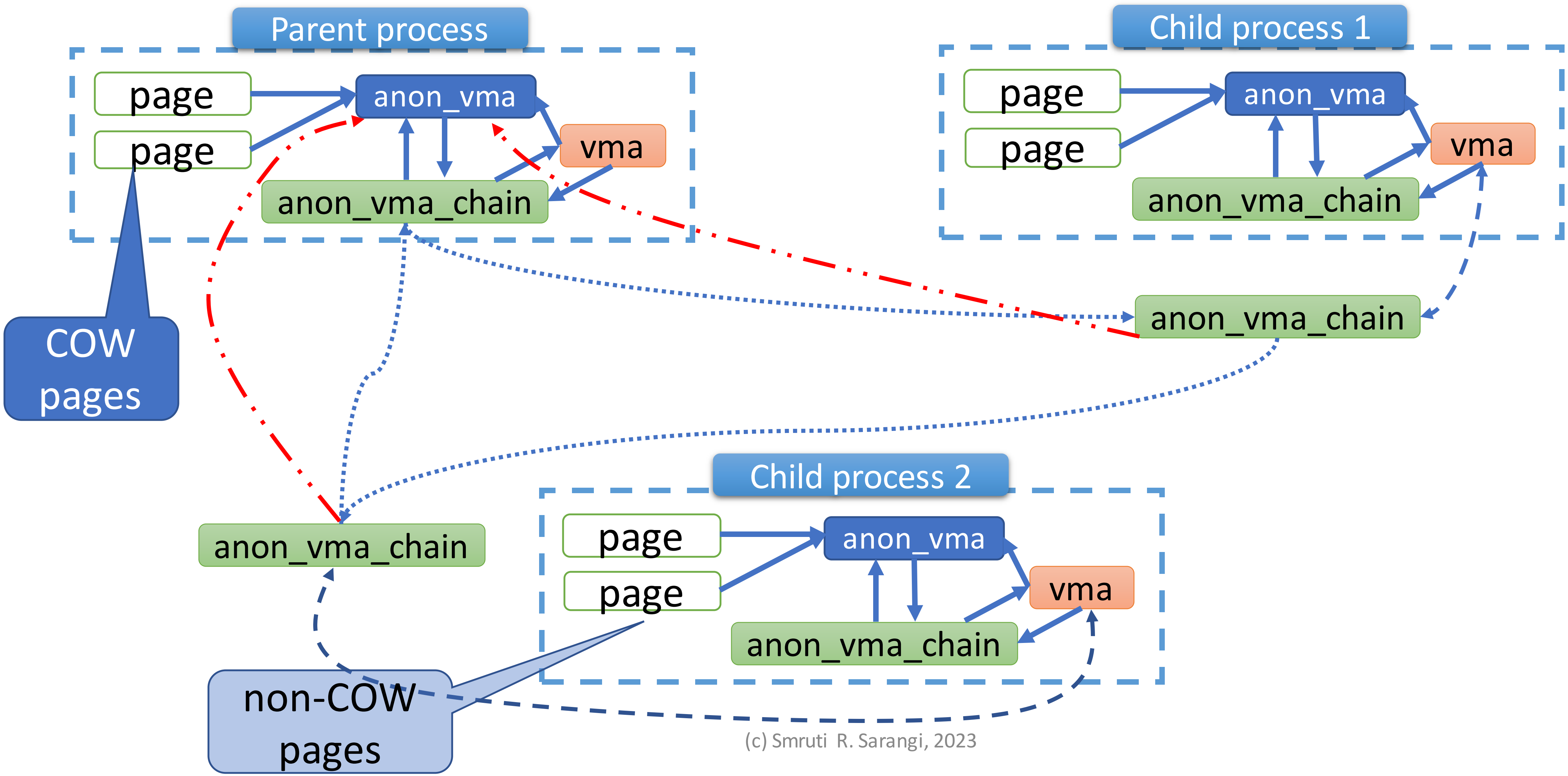
Background

Reverse Mapping – II



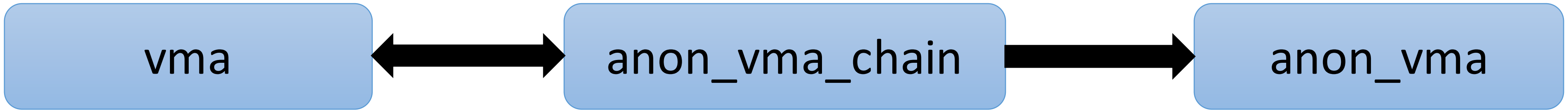
Background

Multi-process View



anon_vma_fork in *mm/rmap.c*

- Justification of the complex structure →
- The rest of the code relies on this structure



What is the need for the extra *anon_vma_chain* associated with a child process?

- An *anon_vma_chain* is associated with a single *anon_vma*
- We need one to point to the parent process's pages (via the corresponding *anon_vma*) that are also mapped to the child (COW pages). This is the job of the extra *anon_vma_chain*.
- The other one is needed for child pages in a *vma* to point to. These pages exclusively belong to the child.

Go forward

Page Replacement

https://elixir.bootlin.com/linux/latest/source/Documentation/mm/multigen_lru.rst

New MG-LRU page replacement algorithm

- Objectives of a page replacement algorithm.

- Leverage temporal locality
- Take advantage of spatial locality
- Use the slow path-fast path approach
- Simple heuristics that are well-suited to all kinds of workloads

/include/linux/mmzone.h

```
struct lruvec {
    struct pglist_data *pgdat;
    unsigned long refaults [ANON_AND_FILE];
    struct lru_gen_struct       lrugen;
    struct lru_gen_mm_state     mm_state;
};
```

Number of refaults

```
struct lru_gen_struct {
    unsigned long max_seq;
    unsigned long min_seq[ANON_AND_FILE];
    unsigned long timestamps[MAX_NR_GENS];
    struct list_head
            lists[MAX_NR_GENS][ANON_AND_FILE][MAX_NR_ZONES];
};
```

age at least *min_ttl ms* after *min_seq* is created

Latest

Oldest: separate for anon and file types

lru_gen_mm_state

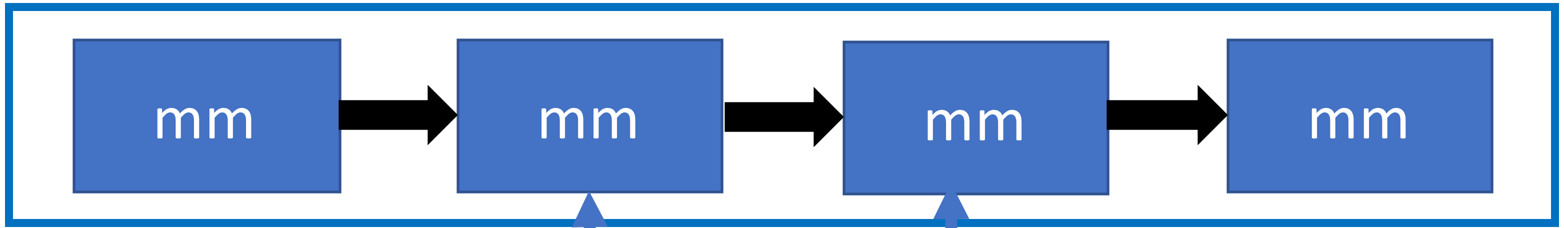
```
struct lru_gen_mm_state {
    unsigned long seq;
    struct list_head *head;
    struct list_head *tail;
    unsigned long *filters[NR_BLOOM_FILTERS];
    int nr_walkers;
};
```

Current sequence number

tail has finished. continue at *head*

list of Bloom filters

#page walkers



Array of *mm_struct* objects

/mm/vmscan.c

The basic idea of compressing the memory footprint

https://lwn.net/Articles/419713/

- Iterate through the list of zones
- Shrink each zone

Multi-Gen LRU Algorithm

- A group of pages is associated with a generation.
- The generation indicates recency of access.
- Take the priority of pages into account. Prefer unmapped and clean pages.
- Evict pages in the oldest generation (min gen) and also take the priority into account

/arch/x86/include/asm/pgtable.h

Tracking Accesses

Needed to maintain generations

```
pte_t pte_mkold(pte_t pte)
{
    return pte_clear_flags(pte, _PAGE_ACCESSED);
}
```

- This function is called by a bunch of functions that walk the page table
- The key idea is to clear the _PAGE_ACCESSED bit in the PTE entry
- The next time that the HW accesses this page
  - It can either raise a page fault
  - OR set the bit itself (directly)
- A second scan of this memory region will indicate, which pages have been accessed.

Indicator of recency

When do we age and subsequently evict?

kswapd

- This is a daemon that runs periodically: ages and reclaims pages

Call to page allocation routines

Should we run the Aging Algorithm?

should_run_aging(...)

- Youngest generation number: lrugen->maxseq
- Oldest generation number: lrugen->minseq[ANON] and lrugen->minseq[FILE]
- When *max_seq – min_seq + 1* <= MIN_NR_GENS, increment *max_seq*
- Given that *max_seq* and *min_seq* increase monotonically, not incrementing the sequence of a folio automatically ages it
- A sliding window of generations is maintained. It tracks [MIN_NR_GENS, MAX_NR_GENS] generations

Triggered by a call to age the *lruvec* and reclaim pages

The Aging Process of Pages

/mm/vmscan.c

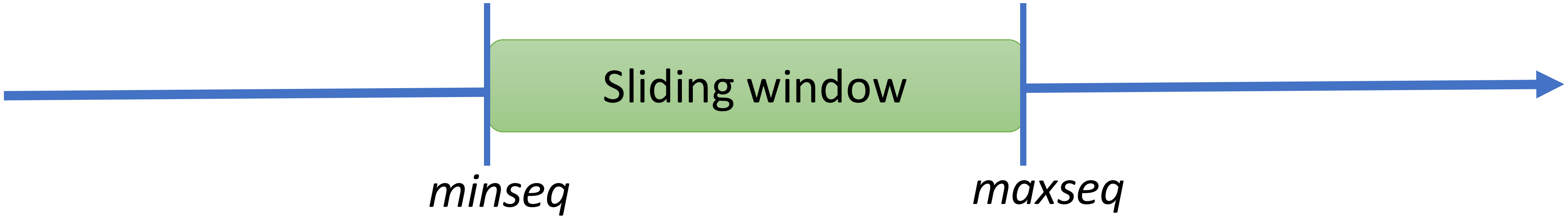
Corner case

max_seq – min_seq == MIN_NR_GENS

```
if (young * MIN_NR_GENS > total)
        return true;
if (old * (MIN_NR_GENS + 2) < total)
        return true;
return false;
```

Too many young pages.

Too few old pages.



Note the definition

In this function, young page’s gen == max_seq
old page’s gen == min_seq

The Aging Process Triggers *walk_mm*

Walk through all the mm_structs

- Walk through the individual page tables

Skip a PMD's pages if

Eviction Algorithm

Overview

- When the lrugen->lists[min_seq][type][] is empty, *min_seq* can be incremented for the corresponding type (anon or file). Otherwise, incrementing ⇒ eviction
- Maintain tiers for each generation – maintained in folio->flags
  - tier = $\log_2$ (#recorded_page_accesses)
  - Tracked via system calls for file page accesses
  - Tracked via access bit changes for ANON page accesses
  - The first tier has single-use clean pages that are unmapped
- Broad idea: Compare the file and anon *min_seq*[] values
  - Choose the one with the lower value
  - If both are the same, then choose that type whose first tier has a lower normalized refault rate

Typically in the page cache: SW cache for pages read from the disk

Evict the oldest generation

Main function called from *shrink_zones*

evict_folios (lruvec, int swappiness, ...)

Hyperparameter, typical value: 60

1. scanned = isolate_folios (lruvec, swappiness, ...)

Identify the folios that can possibly be evicted.

2. Reclaim pages by calling *shrink_folio_list*

Try to remove the folios that have been identified. It is possible that a folio is mapped to multiple processes.

Do additional bookkeeping and account for all corner cases.

evict_folios → isolate_folios

/mm/vmscan.c

Find the type that should be evicted

type

ANON = 0, FILE = 1, ANON_AND_FILE = 2

```
if (!swappiness)
    type = LRU_GEN_FILE;
else if (min_seq[LRU_GEN_ANON] < min_seq[LRU_GEN_FILE])
    type = LRU_GEN_ANON;
else if (swappiness == 1)
    type = LRU_GEN_FILE;
else if (swappiness == 200)
    type = LRU_GEN_ANON;
else
    type = get_type_to_scan(lruvec, swappiness, &tier);
```

Default if **swappiness** = 0

ANON is older

Choose FILE if **swappiness** = 1

Choose ANON if **swappiness** = 200

*get_type_to_scan (lruvec, swappiness, int *tier_idx)*

Collect the number of total and refault accesses

Choose the type that has the lower normalized refault rate

Set point is ANON, tier = 0

PI cntrl

```
int gain[ANON_AND_FILE] = {swappiness, 200 - swappiness};

read_ctrl_pos(lruvec, LRU_GEN_ANON, 0, gain[LRU_GEN_ANON], &sp);
read_ctrl_pos(lruvec, LRU_GEN_FILE, 0, gain[LRU_GEN_FILE], &pv);
type = positive_ctrl_err(&sp, &pv);

return type;   /* If (Error = 1) return FILE else ANON */
```

The process variable: refault rate of tier=0 for FILE

logic for comparing the normalized refault rate

The rest of the *isolate_folios* function

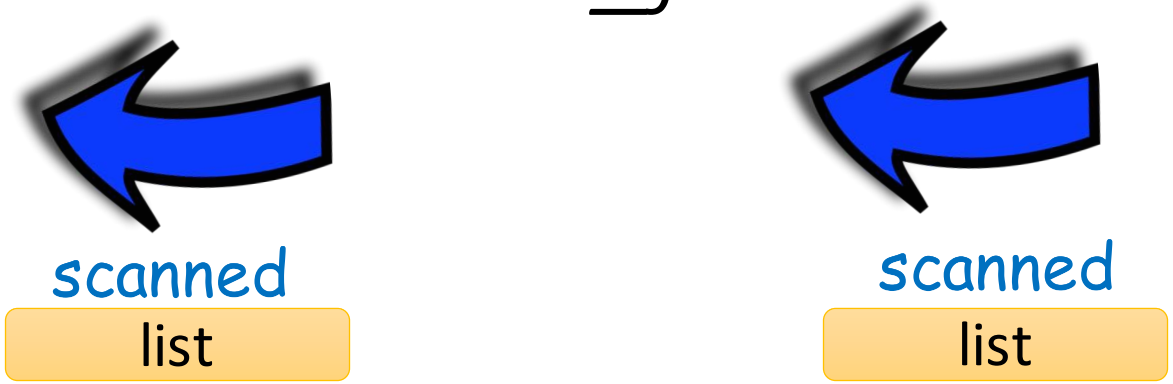
```
for (i = !swappiness; i < ANON_AND_FILE; i++) {
        if(scanned = scan_folios(lruvec, type, list, …) break;

      /* reset the type*/
     return scanned;
}
```

Check if there are evictable folios and return the number of such folios

more about scan_folios

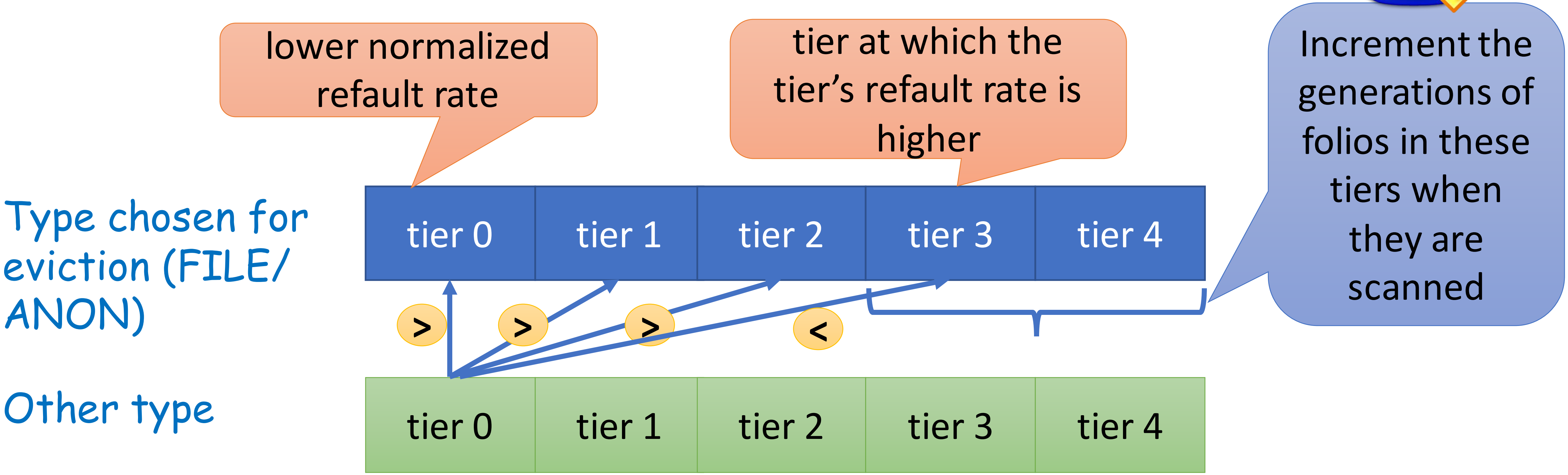
evict_folios → isolate_folios → scan_folios



1. Iterate through the zones that need to be shrunk (order: descending)
   I. Iterate through all the folios pointed to by lrugen->lists[min_seq][type][zone]
      a) Check that the folio can be evicted
      b) If the folio is pinned, is being written back, was recently accessed, or there is a race condition, skip it
2. Return the number of pages that can possibly be evicted, and a list of folios
   I. This is the variable scanned
   II. *list* variable → list of possibly evictable folios

More about *scan_folios*

Does some additional computation



Even if the generation is *min_seq*, if the folio is in a higher tier, give it another chance by incrementing its generation

evict_folios → shrink_folio_list

We have a list of folios that can possibly be evicted

1. Perform basic bookkeeping. Take the number of reclaimed pages into account.
2. Folios that are in the process of being written back may stall this process. Wait.
3. Given a folio that needs to be evicted
   i. Look around nearby addresses: *lru_gen_look_around*
   ii. Mark a few old (clear the access bit)
   iii. Note down PMD entries that point to primarily young (recently accessed) pages (in the Bloom filter)
4. Split large folios into smaller folios (for performance reasons)
5. Free buffers, flush the corresponding entries from the TLB, write back

lru_gen_look_around: main input vm_area

1. Consider the memory space that is mapped by one entry in the PMD
2. Further limit the size of the considered region to 64 pages
3. Iterate through all the pages
   I. If it is an old page (not recently accessed) continue.
   II. Get the folio associated with the page
   III. Mark the page old (clear its access bit)
   IV. If the folio's generation ≠ max_seq, record the page number in a bitmap

4. If the number of young pages is more than a certain threshold record the PMD number in the Bloom filter (makes page walking more efficient)
5. Set the folios of the pages (recorded in the bitmap) to the latest generation (max_seq) and do the rest of the bookkeeping.

❓ How do we use the Bloom filter (subsequently)?

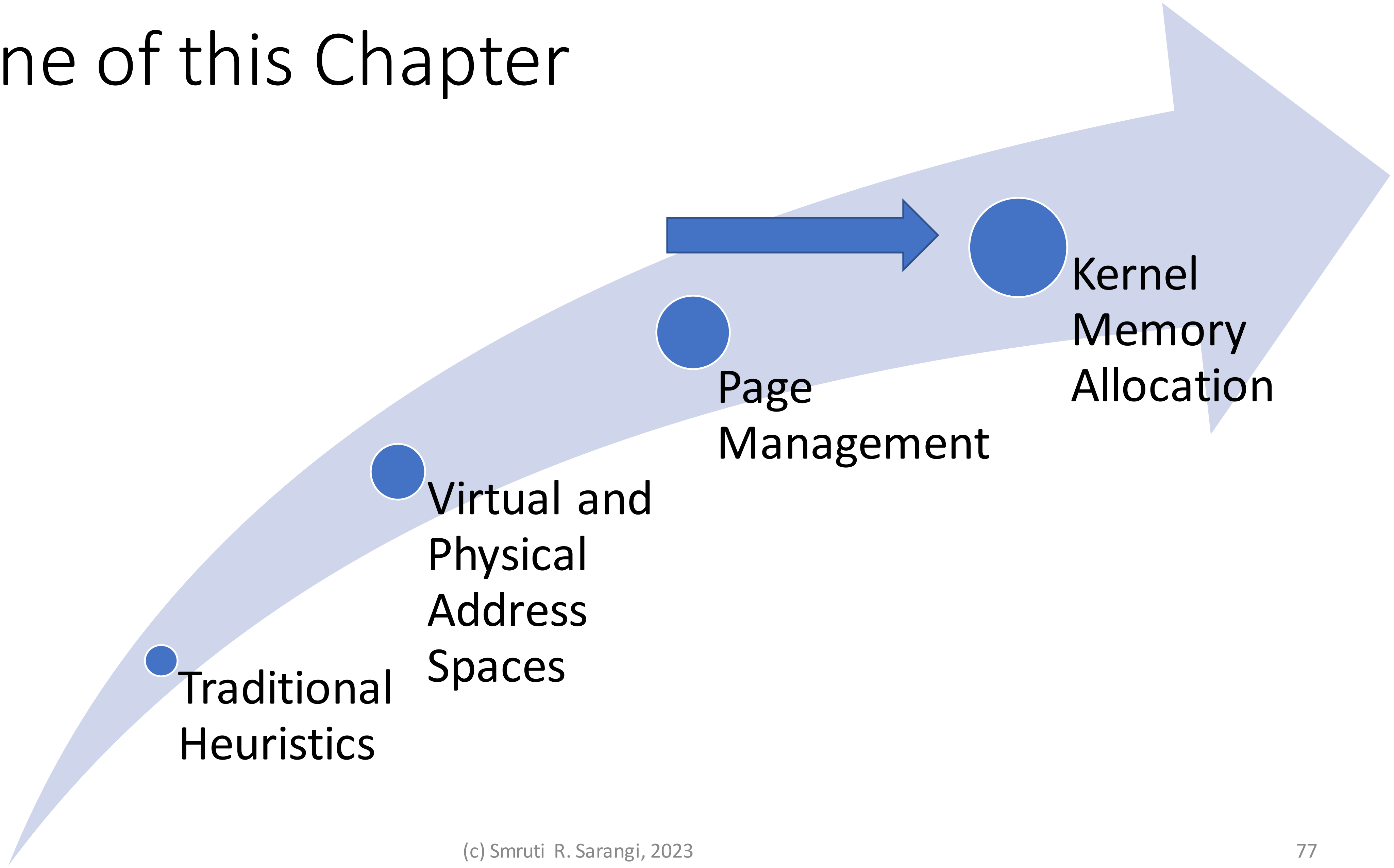
1. *kswapd*, sequential write commands, and page allocation modules try to age and evict pages.
2. They call the function *walk_mm()* that walks the page table and marks young (accessed) entries as old (unaccessed), as well as updates the generation of young entries.
3. Entries that are already old are skipped.
4. While walking the PMD tables, test if the PMD address is in the Bloom filter. If it is not there, then the PMD range is most likely old and should be skipped. Improves performance.

Miscellaneous Topics: Thrashing

- There is a need to keep the working set in memory
- Don't evict a page that is less than *N* ms old in memory
- N = 1000 is practically a good value for GUI-based applications
- If the working set cannot be maintained in memory
  - The OOM killer (out-of-memory) comes into action and terminates applications
- The behavior is configurable (hyperparameters)

MIN_NR_GENS   MAX_NR_GENS   *swappiness*

Outline of this Chapter



Buddy Allocation

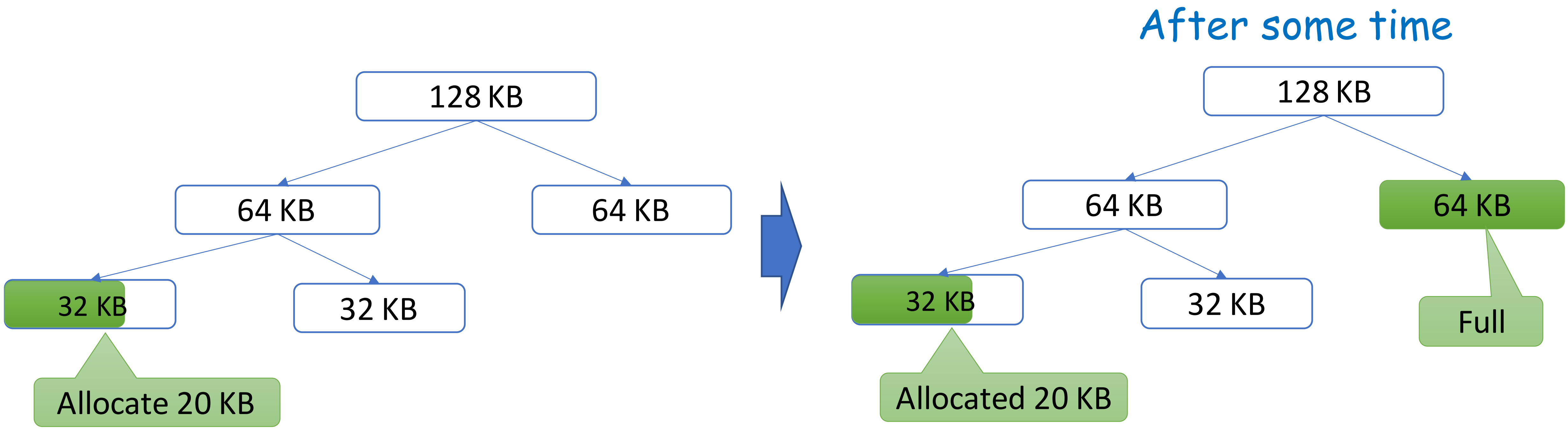
- To manage the physical memory, the buddy allocation technique is used in the kernel
- It can provide physical memory chunks of arbitrary sizes to processes
- It now has more relevance given the prevalence of folios

128 KB

Total memory size

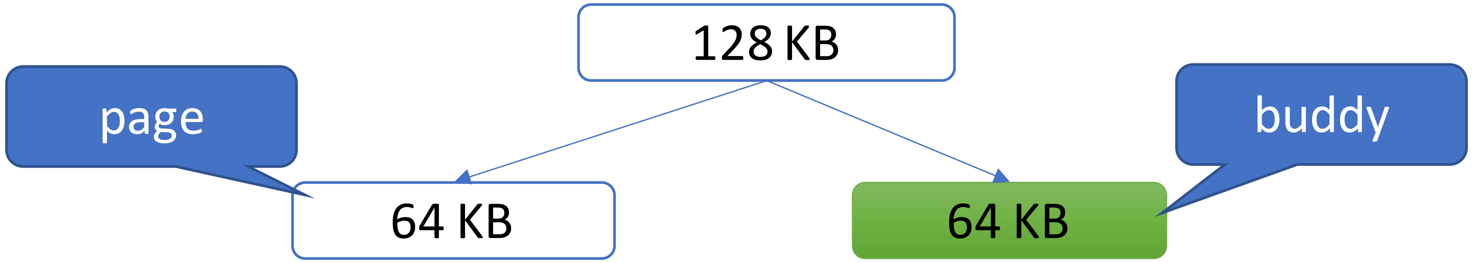
How do we allocate a 20 KB chunk?

Allocation using the Buddy Allocator



Now delete the 20 KB chunk

Contd ….



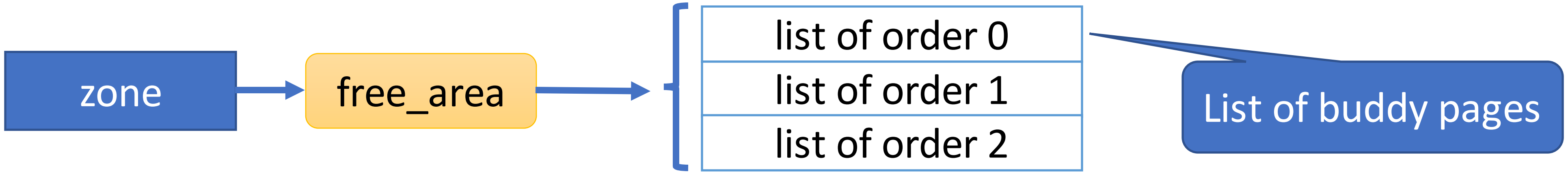
- The size of each buddy chunk is a power of two
- Adjacent free chunks (holes) can be merged
- Tries to minimize internal and external fragmentation
- Can handle requests for both small and large regions

Create a data structure to effectively manage the buddy list

Implementation of the Buddy System

```
struct zone {
    ...
    struct free_area    free_area[MAX_ORDER];
    ...
}

struct free_area {
    struct list_head    free_list[MIGRATE_TYPES]; /* unmovable, movable,
reclaimable, ...*/
    unsigned long       nr_free;
};
```



Traverse the *free_area* with an increasing order

/mm/page_alloc.c

__rmqueue_smallest (…)

```
for (current_order = order; current_order < MAX_ORDER; ++current_order) {
        area = &(zone->free_area[current_order]);
        page = get_page_from_free_area(area, migratetype);
        if (!page)
            continue;
        del_page_from_free_list(page, zone, current_order);
        return page;
    }
```

- Traverse the list of free pages: order=0 to order = MAX_ORDER – 1
- Pick the first free compound page (from the first list that is non-empty)
- Delete the selected page from the corresponding free list

Merging *free* Pages

/mm/page_alloc.c

```
void __free_one_page(struct page *page, unsigned long pfn,
        struct zone *zone, unsigned int order, ...)
{
    while (order < MAX_ORDER - 1) {
        buddy = find_buddy_page_pfn(page, pfn, order, &buddy_pfn);
        del_page_from_free_list(buddy, zone, order);

        ....
        combined_pfn = buddy_pfn & pfn;
        page = page + (combined_pfn - pfn);
        pfn = combined_pfn;
        order++;
    }

    set_buddy_order(page, order);
    add_to_free_list(page, zone, order, migratetype);
}
```

buddy_pfn = *pfn* ^ (1 << *order*)

Not free any more

pfn of the parent

Details of *combined_pfn*

Set the order of the large combined_page and free it

Slab Allocator

Dedicated object cache

- The buddy system is for generic memory allocation
- Most of the time objects of specific types are used. They have a fixed structure.
  - No reason to allocate and deallocate them.
  - Take up a large memory region and use it for dedicated object storage (of only one type)
  - For fast access, keep a cache of pre-initialized objects
  - Use and return → Similar to a pool of objects

Terms: slab cache chunk

The Kernel has Three Object Allocators

SLOB Allocator

- Old Solaris-based allocator that uses the first-fit method.

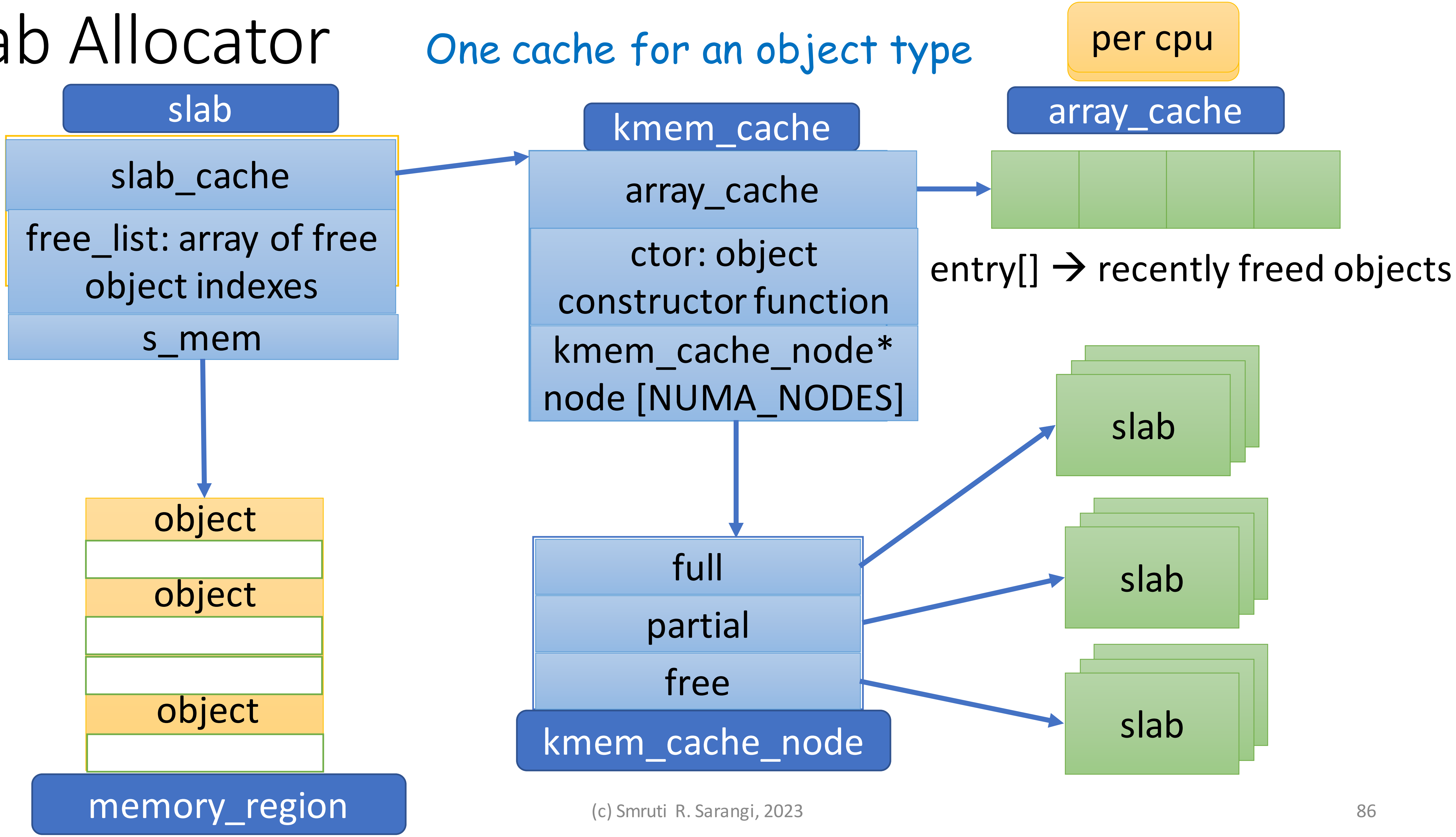
SLAB Allocator

- Most popular allocator.

SLUB Allocator

- Better performance, scalability and simpler

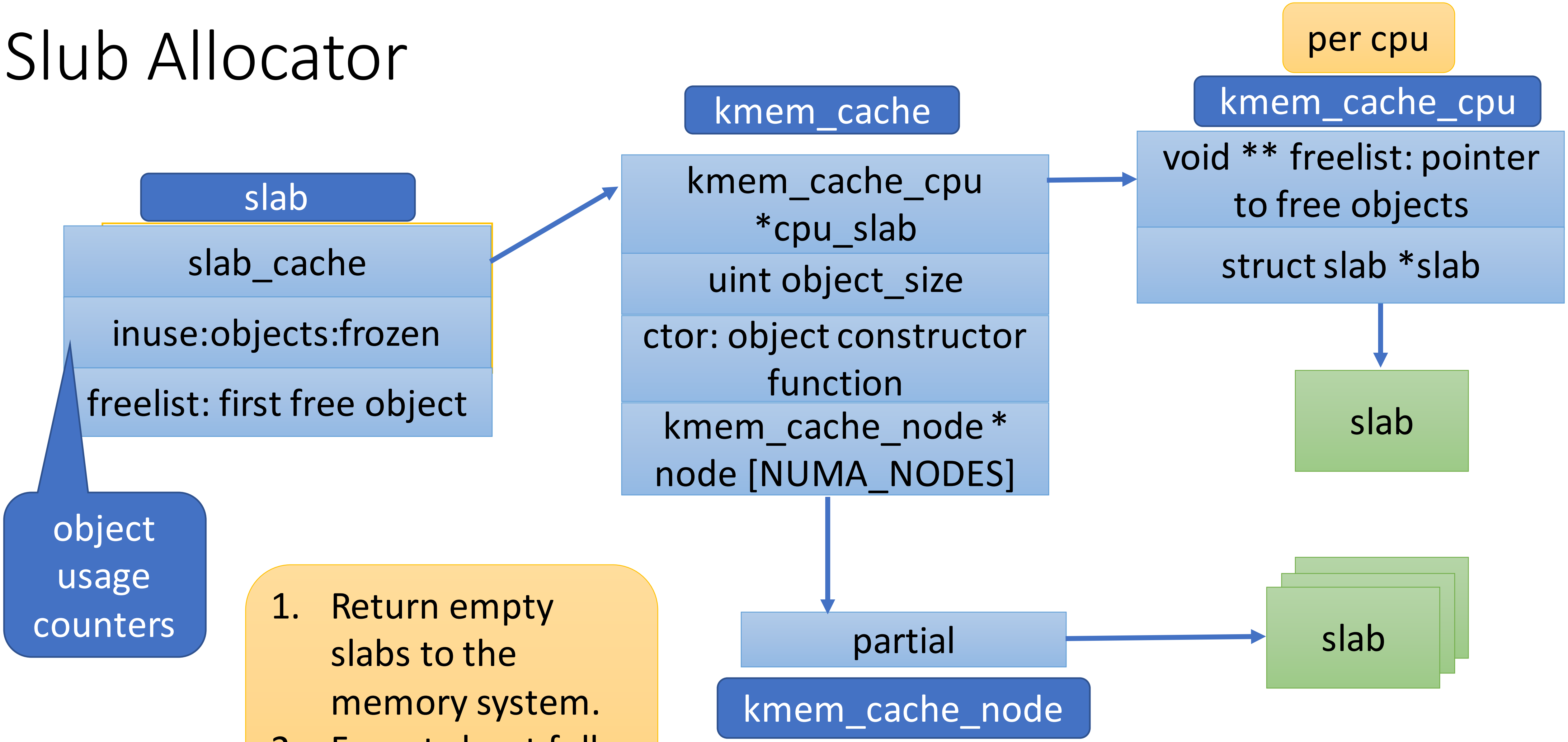
Slab Allocator



Quick Explanation of the SLAB scheme

- A dedicated region of the physical memory and the kernel's virtual memory stores slabs
- Every slab corresponds to a region of physical memory (set of contiguous physical pages)
- The *kmem_cache* is the key data structure that maintains all the slabs for a given object type. The external world only interacts with the *kmem_cache*
  - It maintains a list of slabs in three different states: full, free, and partial
  - This list is specific to each NUMA node
  - It also holds a per-cpu cache of free objects that can quickly be allocated
  - There are few *kmem_cache* structures in the system for storing very frequently used objects that are not allocated and deallocated every time they are used

Slub Allocator



1. Return empty slabs to the memory system.
2. Forget about full slabs.

Quick Explanation of the SLUB Allocator

- Much simpler than the SLAB allocator (still uses the notion of slabs)
- Here also, the *kmem_cache* is the key data structure that is visible to the external world
- In a slab → Maintain a freelist (object pointers)
- In the *kmem_cache* → Maintain an array of per-cpu slabs
  - One slab per CPU
  - Just maintain the list of partial slabs in a *kmem_cache* (a single list)
  - Basically means: forget about full and totally *free* slabs
  - *free* slabs are returned to the regular memory allocation system
  - When an object is freed in a full slab, add the full slab to the partial slab list

Bibliography

1. Belady, Laszlo A., Robert A. Nelson, and Gerald S. Shedler. "An anomaly in space-time characteristics of certain programs running in a paging machine." *Communications of the ACM* 12.6 (1969): 349-353.
2. FIFO anomaly is unbounded, arXiv:1003.1336

srsarangi@cse.iitd.ac.in