Operating Systems

Virtual Memory, x86, and Page Replacement

Daniel Gruss

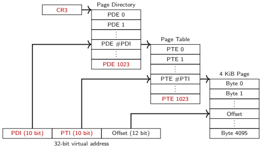
2023-10-08

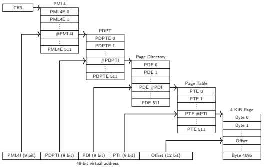
Table of contents

Efficient Address Translation

Efficient Address Translation

- What about speed?
- How many actual memory accesses per intended memory access?





x86-32

- 1 access into page directory
- 1 access into the page table page
- 1 access into memory

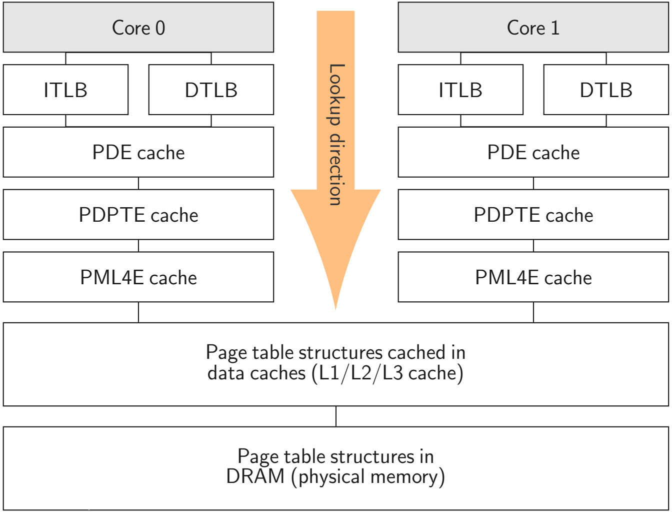
x86-64

- 1 access into page-map-level-4
- 1 access into the page-directory pointer
- 1 access into page directory
- 1 access into the page table page
- 1 access into memory

Speed up things again

- Translation Look-aside Buffer (TLB)
- Cache recent virtual → physical translations
- TLB: A page-table-entry cache
  - Cache hit: use translation
  - Cache miss: walk multi-level page table
- TLB entry
  - virtual page number
  - physical page frame number
  - access permissions

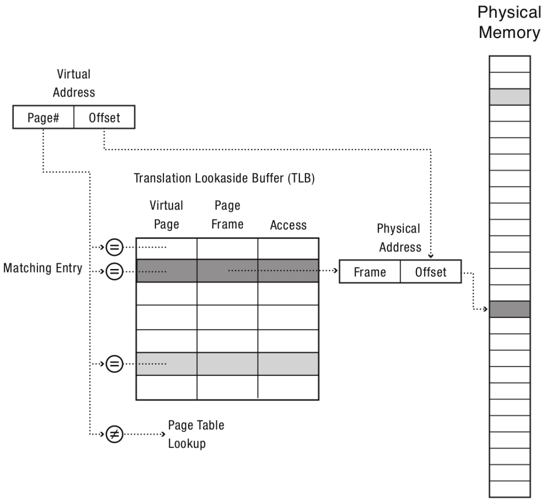
How many memory accesses per cache miss?



TLB / Caching

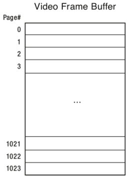
- Why does caching help?
- Principle of locality
  - If a memory address is accessed, likely nearby addresses are referenced in the future
  - Nearby: same page, uses identical address translation (without offset)
  - High degree of locality: almost all page translations from TLB

TLB


Physical
Memory
Virtual
Address
Page#
Offset
Translation Lookaside Buffer (TLB)
Virtual
Page
Page
Frame
Access
Physical
Address
Frame
Offset
Matching Entry
Page Table
Lookup

When Do TLBs Work/Not Work?

- Video Frame Buffer: 32 bits x 1K x 1K = 4MB
- redraw screen - processor may touch every pixel
- 1024 TLB entries required



Superpages / Page sizes

Set of contiguous pages in physical memory that map a contiguous region of virtual memory

- e.g. 2 MB superpage consists of 512 regular pages (4 KB)
- aligned to lie on a 2 MB boundary

→ fewer TLB-Entries needed

How long does the TLB stay valid?

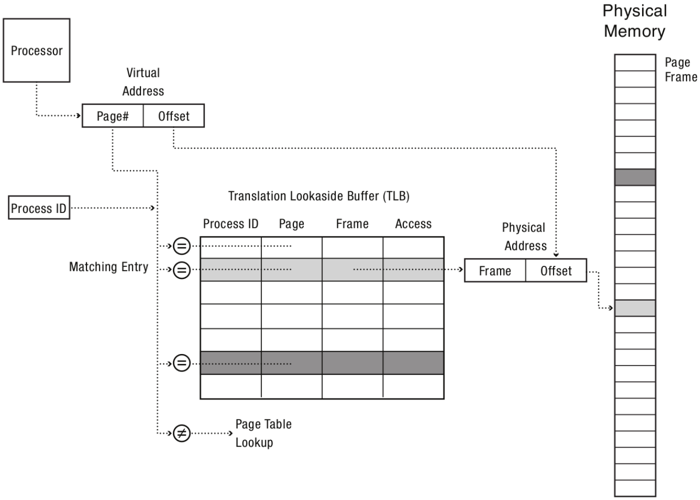
Context switches:

- Do we have to invalidate the entire TLB?

Solution: Tagged TLB

- Each TLB entry has a tag (PID or CR3 or ...)
- TLB hit only if tag matches current register state

Tagged TLB



Processor
Virtual Address
Page#
Offset
Process ID
Translation Lookaside Buffer (TLB)
Process ID
Page
Frame
Access
Matching Entry
Page Table Lookup
Physical Address
Frame
Offset
Physical Memory
Page Frame

How long does the TLB stay valid? (2)

What happens when OS changes permissions on a page?

- demand paging (zero on reference)
- copy on write

TLB may contain old information

- OS must ask hardware to purge TLB entry

On a multicore: TLB shootdown

- OS must ask each CPU to purge TLB entry

Booting

Starting in Real Mode

- 16 bit mode
- Address space: 1 MB
- How is that possible?
- CS register has a 20-bit base address
    - actually only 4 bit, but shifted by 16 bits to the left

  → 4 bits (base/prefix) + 16 bits (address/offset) = 20 bit address

9.1.4 First Instruction Executed

The first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFF0H. This address is 16 bytes below the processor’s uppermost physical address. The EPROM containing the software-
initialization code must be located at this address.

Booting in Real Mode

- Address: `0xFFFFFFF0`
- How is that possible?
  - CS register also has a 32-bit base address (initialized to `0xFFFF0000`)
- What if I have $< 4\,GB$ RAM?
  - physical address space $\neq$ RAM directly mapped

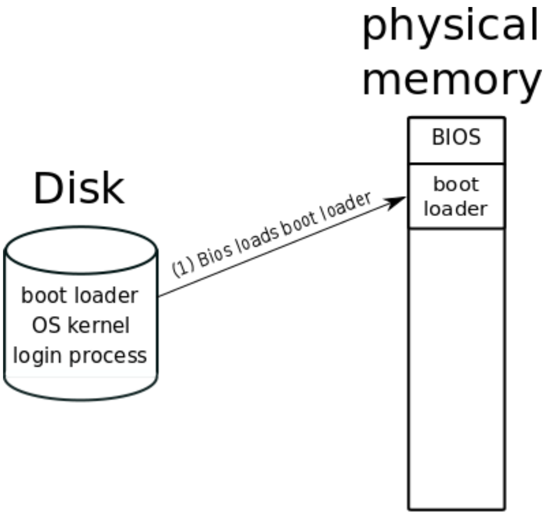
Physical Address Space

```
00000000-007fffff (prio 0, RW): alias ram-below-4g (this is our RAM)
000a0000-000bffff (prio 1, RW): vga-lowmem (remember for later)
000c0000-000dffff (prio 1, RW): pc.rom
000e0000-000fffff (prio 1, R-): alias isa-bios
fd000000-fdffffff (prio 1, RW): vga.vram
febc0000-febdffff (prio 1, RW): e1000-mmio
febf0400-febf041f (prio 0, RW): vga ioports remapped
febf0500-febf0515 (prio 0, RW): bochs dispi interface
febf0600-febf0607 (prio 0, RW): qemu extended regs
fffc0000-ffffffff (prio 0, R-): pc.bios (ahhh!)
...
```

BIOS

- BIOS initializes hardware platform
- Switch to protected mode (32 bit)
- Select a device to boot from
- Load MBR from device into memory
- Execute code from MBR

Booting x86 Intel (Illustration)



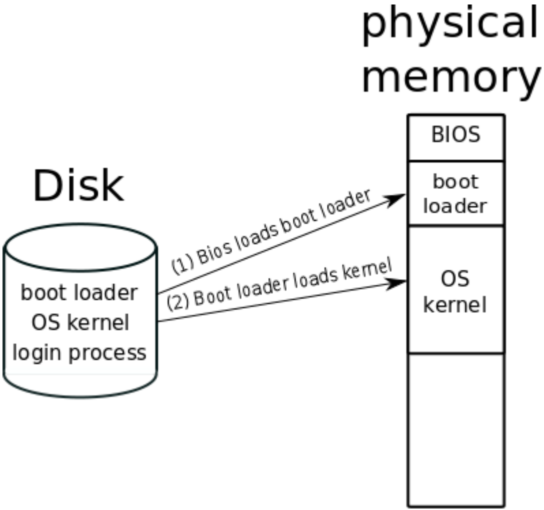
GRUB

- Boot loader for Linux, SWEB, . . .
- Loads the OS image from disk and starts OS

GRUB

One of the important features in GRUB is flexibility; GRUB understands file-systems and kernel executable formats, so you can load an arbitrary operating system the way you like, without recording the physical position of your kernel on the disk. Thus you can load the kernel just by specifying its file name and the drive and partition where the kernel resides.

Booting



physical memory
Disk
boot loader
OS kernel
login process
(1) Bios loads boot loader
(2) Boot loader loads kernel
BIOS
boot loader
OS kernel

Booting the OS

- Prepare hardware
- Start device drivers and initialize devices
- Start initial processes (e.g. *init*-process)

Booting the OS (SWEB)

Kernel is a compiled binary (e.g. an ELF binary)

```
% readelf -a kernel.x | grep Entry
  Entry point address:                0x801001ba

% objdump -S kernel.x | less
801001ba <entry>:
801001ba:       55                      push    %ebp
801001bb:       89 e5                   mov     %esp,%ebp
801001bd:       83 ec 10                sub     $0x10,%esp
801001c0:       89 1d 00 90 14 00       mov     %ebx,0x149000
```

Wait, that's C-Code!

Booting the OS (SWEB)

```
extern "C" void entry()
{
  asm("mov %ebx,multi_boot_structure_pointer - BASE");

  PRINT("Booting...\n");
```

Booting the OS (SWEB)

```
PRINT("Clearing Framebuffer...\n");
memset((char*) 0xB8000, 0, 80 * 25 * 2);

PRINT("Clearing BSS...\n");
char* bss_start = TRUNCATE(&bss_start_address);
memset(bss_start, 0, TRUNCATE(&bss_end_address) - bss_start);

PRINT("Initializing Kernel Paging Structures...\n");
//...
```

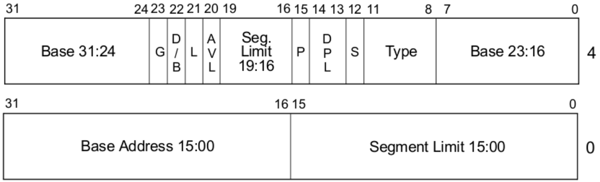
Booting the OS (SWEB)

```
PRINT("Enable PSE and PAE...\n");
asm("mov %cr4,%eax\n"
    "or $0x20, %eax\n"
    "mov %eax,%cr4\n");

PRINT("Setting CR3 Register...\n");
asm("mov %[pd],%%cr3" : : [pd]"r"(TRUNCATE(kernel_page_map_level_4)));

PRINT("Enable EFER.LME and EFER.NXE...\n");
asm("mov $0xC0000080,%ecx\n"
    "rdmsr\n"
    "or $0x900,%eax\n"
    "wrmsr\n");
//...
PRINT("Enable Paging...\n");
asm("mov %cr0,%eax\n"
    "or $0x80000001,%eax\n"
    "mov %eax,%cr0\n");
```

```
PRINT("Setup TSS...\n");
TSS* g_tss_p = (TSS*) TRUNCATE(&g_tss);
g_tss_p->ist0_h = -1U;
g_tss_p->ist0_l = (uint32) TRUNCATE(boot_stack) | 0x80004000;
g_tss_p->rsp0_h = -1U;
g_tss_p->rsp0_l = (uint32) TRUNCATE(boot_stack) | 0x80004000;
```



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 3-8. Segment Descriptor

Booting the OS (SWEB)

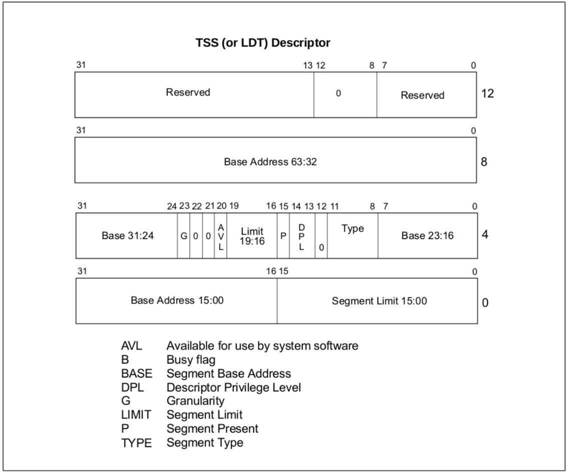


Figure 7-4. Format of TSS and LDT Descriptors in 64-bit Mode

Booting the OS (SWEB)

```
static void setSegmentDescriptor(uint32 index, uint32 baseH, uint32 baseL, uint32
    limit, uint8 dpl, uint8 code, uint8 tss);

  PRINT("Setup Segments...\n");
  setSegmentDescriptor(1, 0, 0, 0, 0, 1, 0);
  setSegmentDescriptor(2, 0, 0, 0, 0, 0, 0);
  setSegmentDescriptor(3, 0, 0, 0, 3, 1, 0);
  setSegmentDescriptor(4, 0, 0, 0, 3, 0, 0);
  setSegmentDescriptor(5, -1U, (uint32) TRUNCATE(&g_tss) | 0x80000000,
                    sizeof(TSS) - 1, 0, 0, 1);

  PRINT("Loading Long Mode GDT...\n");

  struct GDT32Ptr gdt32_ptr;
  gdt32_ptr.limit = sizeof(gdt) - 1;
  gdt32_ptr.addr = (uint32) TRUNCATE(gdt);
  asm("lgdt %[gdt_ptr]" : : [gdt_ptr]"m"(gdt32_ptr));
  // ...
```

Booting the OS (SWEB)

```
  PRINT("Setting Long Mode Segment Selectors...\n");
  asm("mov %%ax, %%ds\n"
      "mov %%ax, %%es\n"
      "mov %%ax, %%ss\n"
      "mov %%ax, %%fs\n"
      "mov %%ax, %%gs\n"
      : : "a"(KERNEL_DS));

  PRINT("Calling entry64()...\n");
  asm("ljmp %[cs],$entry64-BASE\n" : : [cs]"i"(KERNEL_CS));

  PRINT("Returned from entry64()? This should never happen.\n");
  asm("hlt");
}
```

Booting the OS (SWEB)

```
  PRINT("Setting Long Mode Segment Selectors...\n");
  asm("mov %%ax, %%ds\n"
      "mov %%ax, %%es\n"
      "mov %%ax, %%ss\n"
      "mov %%ax, %%fs\n"
      "mov %%ax, %%gs\n"
      : : "a"(KERNEL_DS));

  PRINT("Calling entry64()...\n");
  asm("ljmp %[cs],$entry64-BASE\n" : : [cs]"i"(KERNEL_CS));

  PRINT("Returned from entry64()? This should never happen.\n");
  asm("hlt");
}
```

Booting the OS (SWEB)

```
extern "C" void entry64()
{
  PRINT("Parsing Multiboot Header...\n");
  parseMultibootHeader();
  PRINT("Initializing Kernel Paging Structures...\n");
  initialisePaging();
  PRINT("Setting CR3 Register...\n");
  asm("mov %%rax, %%cr3" : : "a"(VIRTUAL_TO_PHYSICAL_BOOT(ArchMemory::
      getRootOfKernelPagingStructure())));
  PRINT("Switch to our own stack...\n");
  asm("mov %[stack], %%rsp\n"
      "mov %[stack], %%rbp\n" : : [stack]"i"(boot_stack + 0x4000));
```

Booting the OS (SWEB)

```
  PRINT("Loading Long Mode Segments...\n");

  gdt_ptr.limit = sizeof(gdt) - 1;
  gdt_ptr.addr = (uint64)gdt;
  asm("lgdt (%%rax)" : : "a"(&gdt_ptr));
  asm("mov %%ax, %%ds\n"
      "mov %%ax, %%es\n"
      "mov %%ax, %%ss\n"
      "mov %%ax, %%fs\n"
      "mov %%ax, %%gs\n"
      : : "a"(KERNEL_DS));
  asm("ltr %%ax" : : "a"(KERNEL_TSS));
  PRINT("Calling startup()...\n");
  asm("jmp *%[startup]" : : [startup]"r"(startup));
  while (1);
}
```

Booting the OS (SWEB)

```
extern "C" void startup()
{
  writeLine2Bochs("Removing Boot Time Ident Mapping...\n");
  removeBootTimeIdentMapping();
  system_state = BOOTING;

  PageManager::instance();
  writeLine2Bochs("PageManager and KernelMemoryManager created \n");

  main_console = ArchCommon::createConsole(1);
  writeLine2Bochs("Console created \n");
  // ...
```

```
Scheduler::instance();

//needs to be done after scheduler and terminal, but prior to enableInterrupts
kprintf_init();

debug(MAIN, "Threads init\n");
ArchThreads::initialise();
debug(MAIN, "Interupts init\n");
ArchInterrupts::initialise();

ArchInterrupts::setTimerFrequency(IRQ0_TIMER_FREQUENCY);
```

Booting the OS (SWEB)

```
ArchCommon::initDebug();

vfs.initialize();
debug(MAIN, "Mounting DeviceFS under /dev/\n");
DeviceFSType *devfs = new DeviceFSType();
vfs.registerFileSystem(devfs);
default_working_dir = vfs.root_mount("devicefs", 0);

debug(MAIN, "Block Device creation\n");
BDManager::getInstance()->doDeviceDetection();
debug(MAIN, "Block Device done\n");

for (BDVirtualDevice* bdvd : BDManager::getInstance()->device_list_)
{
  debug(MAIN, "Detected Device: %s :: %d\n", bdvd->getName(), bdvd->
      getDeviceNumber());
}
```

```
// initialise global and static objects
extern ustl::list<FileDescriptor*> global_fd;
new (&global_fd) ustl::list<FileDescriptor*>();
extern Mutex global_fd_lock;
new (&global_fd_lock) Mutex("global_fd_lock");
// ...
debug(MAIN, "Timer enable\n");
ArchInterrupts::enableTimer();
KeyboardManager::instance();
ArchInterrupts::enableKBD();
```

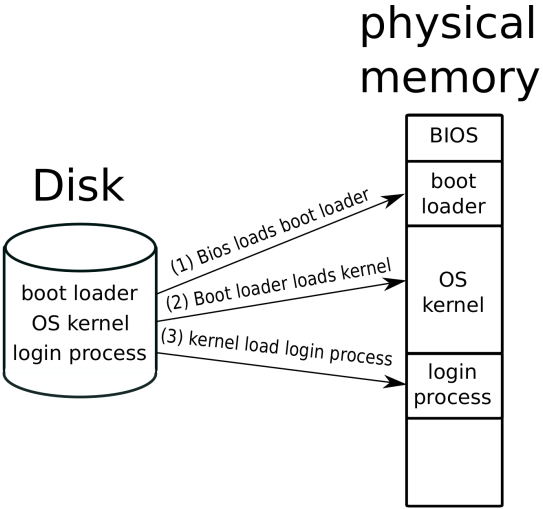
Booting the OS (SWEB)

```
  debug(MAIN, "Adding Kernel threads\n");
  Scheduler::instance()->addNewThread(main_console);
  Scheduler::instance()->addNewThread(new ProcessRegistry(new FileSystemInfo(*
      default_working_dir), user_progs /*see user_progs.h*/));
  Scheduler::instance()->printThreadList();

  kprintf("Now enabling Interrupts...\n");
  system_state = RUNNING;

  ArchInterrupts::enableInterrupts();

  Scheduler::instance()->yield();
  //not reached
  assert(false);
}
```

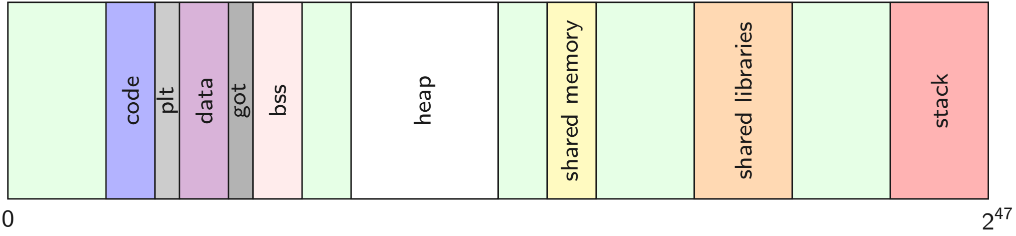

physical
memory
Disk
boot loader
OS kernel
login process
(1) Bios loads boot loader
(2) Boot loader loads kernel
(3) kernel load login process
BIOS
boot
loader
OS
kernel
login
process

Memory Layout

Loader

- Parse binary (headers)
- different binary formats
  - .COM - program always starts at byte 256 (also used in CP/M)
  - a.out
  - COFF
  - **Executable and Linking Format (ELF)**

Memory Layout User Space on

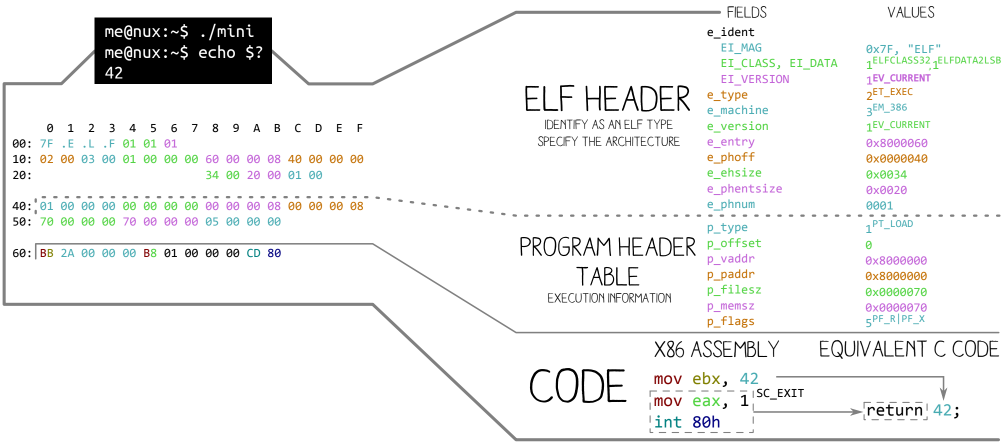


- PLT: Procedure Linkage Table
- GOT: Global Object Table

Code

- Executable
- Usually readable
- Usually not writable

ELF Binaries



- object files (compiled code)
- dynamic libraries
- static libraries

Loading

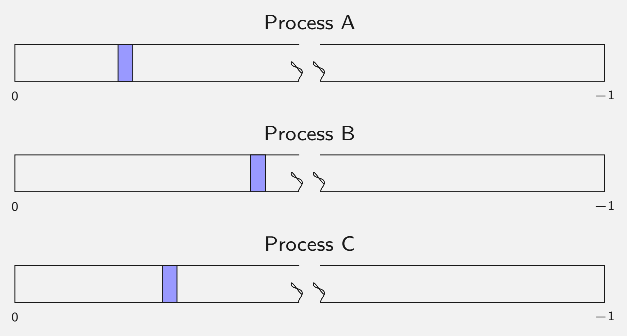
- What about the stack?
- Size? Address?
  - Locate a suitable address area for the stack
  - Define the initial size of the area
- Load on demand
  - Data from binary
  - Zeros (security!)

Address Space Layout Randomization (ASLR)

Every program start, use different random offsets for

- program sections
- libraries
- heap
- stacks

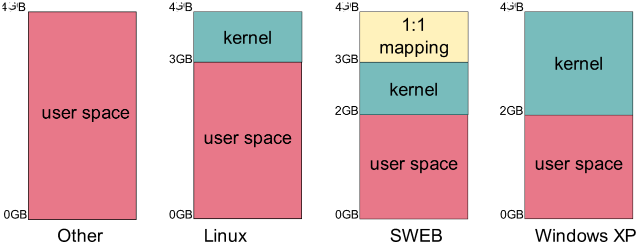
→ Addresses are unpredictable for an attacker



Memory Layout Revisited

- OS has to layout the linear memory for a process
- only addresses can be accessed that are mapped into the process address space via the page table mechanism
- decision: do we also map the kernel into the process address space?

Memory layout



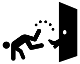
Memory Layout x86-32

- 32-bit addresses: memory locations between 0 GB and 4 GB
- x86 requires a minimal region of the kernel to be mapped (for context switches)
- typically a large part of the linear address space is reserved for the kernel
- inaccessible due to userspace permission bit (set to 0 for kernel pages)

```
0000000000000000-00007fffffffffff (=47 bits) user space
ffff800000000000-ffff87ffffffffff (=43 bits) hypervisor
ffff880000000000-ffffc7ffffffffff (=64 TB) identity mapping
ffffc90000000000-ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffea0000000000-ffffeaffffffffff (=40 bits) virtual memory map
ffffec0000000000-fffffbffffffffff (=44 bits) KASAN shadow memory
ffffff0000000000-ffffff7fffffffff (=39 bits) ESP fixup stacks
ffffffef00000000-fffffffeffffffff (=64 GB) EFI region mappings
ffffffff80000000-ffffffff9fffffff (=512 MB) kernel code/data
ffffffffa0000000-ffffffffff5fffff (=1526 MB) kernel modules
ffffffffff600000-ffffffffffdfffff (=8 MB) vsyscalls
```

Page Replacement

Page Replacement

- At some point in time, physical memory will become full
- We need to make space available → throw out (= evict ) a page
  - Unmodified code and data could be reloaded from binary
  - What about other memory contents (modified from disk or generated)?
- When do we perform page replacement? For now:
  - When not a single page is available, and
  - a thread T tries to allocate a page.

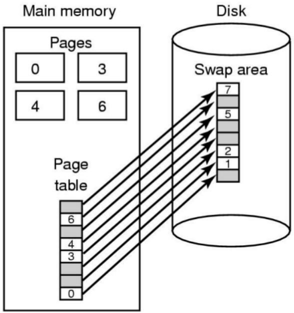
  → We evict a page, clear it, and return it (the now free page) directly to thread T.

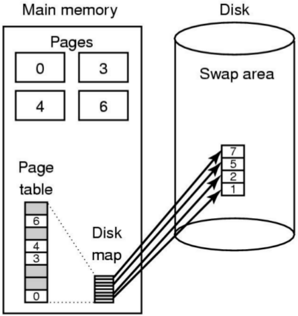
What to do with modified pages?

Swap Out

- Reserve a special area on the disk
  - swap file
  - swap partition
  - swap disk
- Write modified page there
- Evict it from RAM

Swapping





- static assignment
- low overhead
- not "on demand": waste of disk space

- dynamic assignment
- larger overhead
- on demand: no wasted disk space

The Most Simple Page Replacement Algorithm (PRA): Random

- Simply evict a random page, any page. 
- How good is Random PRA? It's a good start...
  - Often used as replacement algorithm in caches (ARM processors)
- Source of randomness? Not that important, e.g., `rdtsc`

Random PRA

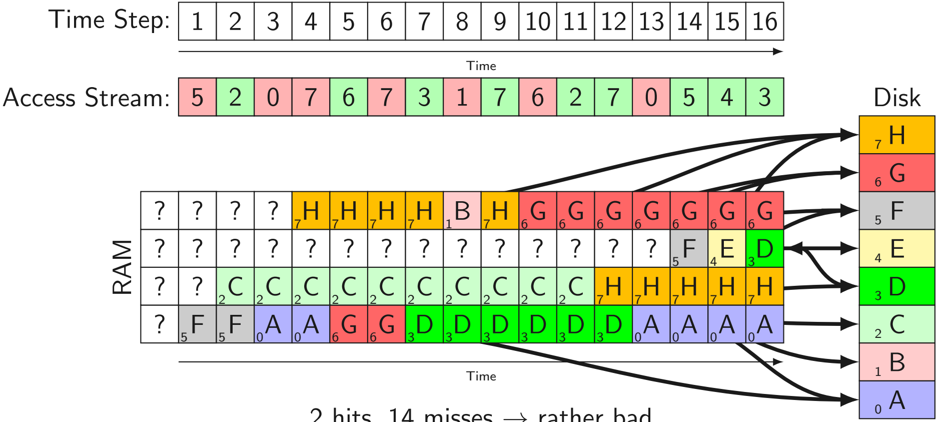
Pros:

- Very simple to implement in software or hardware
- No state, no precomputations, fast decision, tiny code base
- May perform better than several more complex PRAs

Cons:

- PRA could use more information to not evict pages which are frequently used / required in the near future

Random PRA



2 hits, 14 misses → rather bad

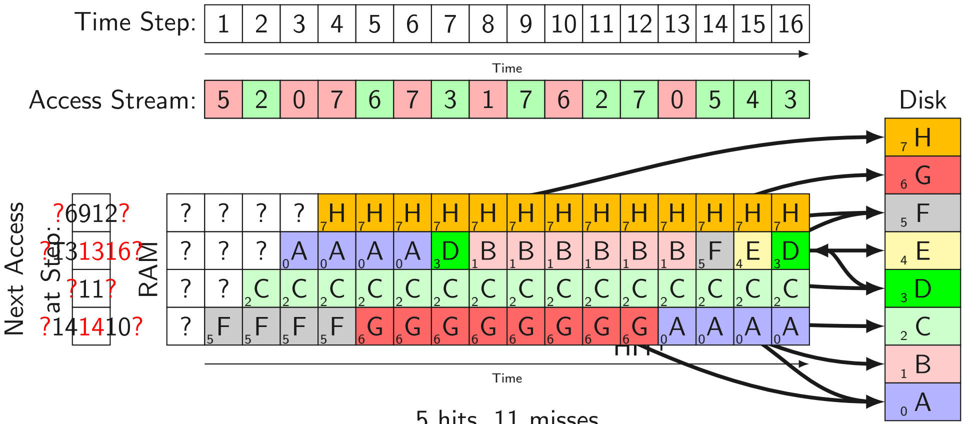
Daniel Gruss

The Best Replacement Algorithm: Optimal PRA

Let's assume, we can predict the future 

1. Store number of steps until next access (per page)
2. Remove page with largest number

Optimal PRA



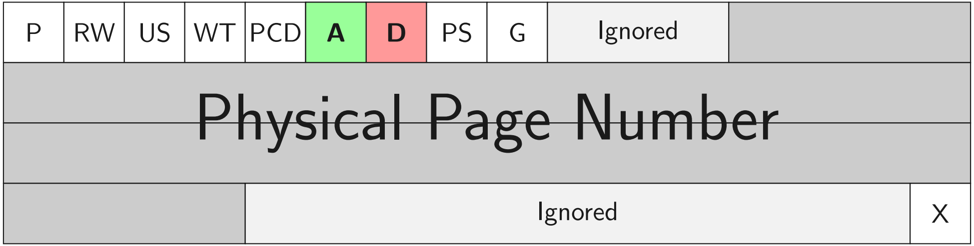
5 hits, 11 misses

Predict the Future

- We can't look into the future...
- Principle of locality
  - future memory access might be near past memory accesses
  - design idea of virtually all sophisticated PRAs

→ How do we learn past memory accesses?

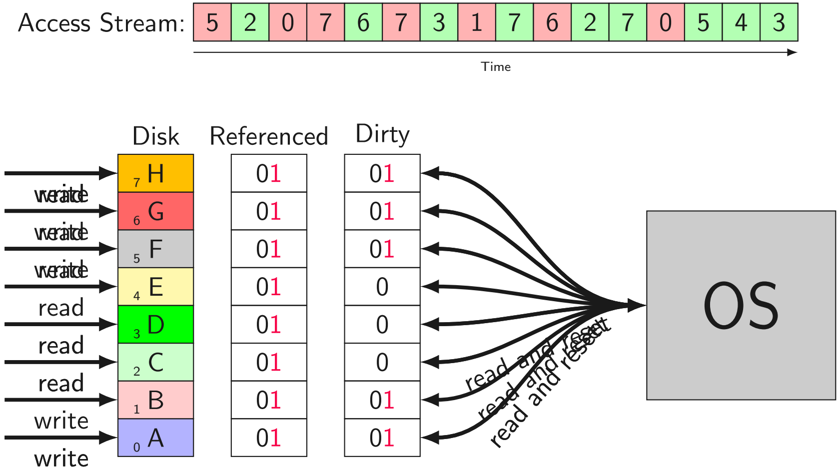
How to detect past memory read and write accesses?



How to detect accesses?

Problem: 1 bit of information is not a detailed trace of past memory accesses

How do we get the information we need?



Access Stream:
5 2 0 7 6 7 3 1 7 6 2 7 0 5 4 3
Time
Disk
Referenced
Dirty
7 H
6 G
5 F
4 E
3 D
2 C
1 B
0 A
01
0
read
write
read and reset
OS

General Ideas on Page Replacement: Four Classes of Pages

Which is the best class to choose pages from for replacement?

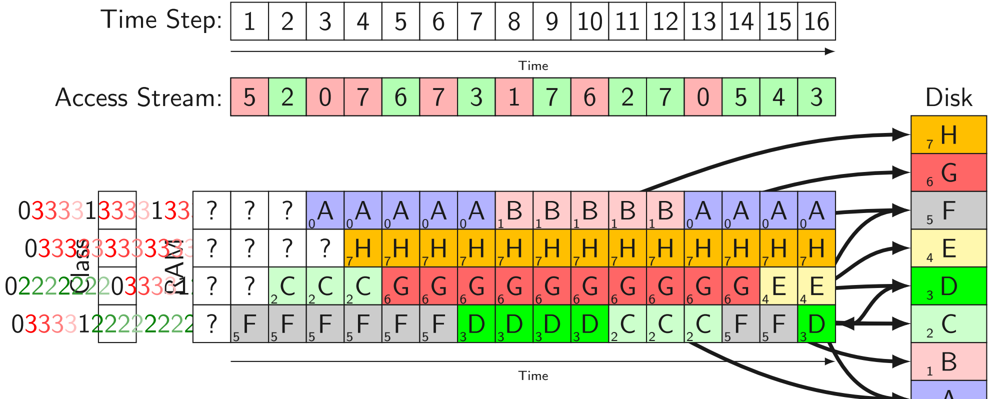
Class	Referenced	Dirty	Properties
0	0	0	Not used in a while and not modified $\rightarrow$ just evict
1	0	1	Not used in a while but modified $\rightarrow$ write back, then evict
2	1	0	Recently used but not modified $\rightarrow$ prefer eviction of other pages
3	1	1	Recently used and modified $\rightarrow$ only evict as a last resort

Dirty = it's not stored identically on the disk

Not Recently Used PRA

- Basically: Random PRA with classes (0-3)
- Performes better than Random PRA
- Design Decision: How far does "recently" go?

NRU PRA



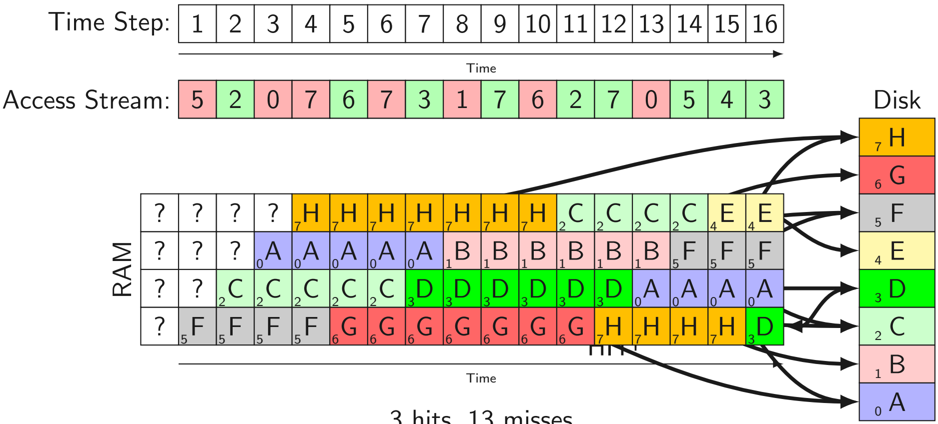
4 hits, 12 misses

First-In First-Out (FIFO) PRA

First-in First-Out

- Queue/List of all pages (e.g. `std::queue`)
- Load a page: `push_back`
- Page to replace: `pop_front`
- Very simple algorithm
- Rarely used in practice
- Performance can even be worse than Random PRA(!)
  + FIFO anomaly / Belady's anomaly: increasing memory size can reduce performance

FIFO PRA



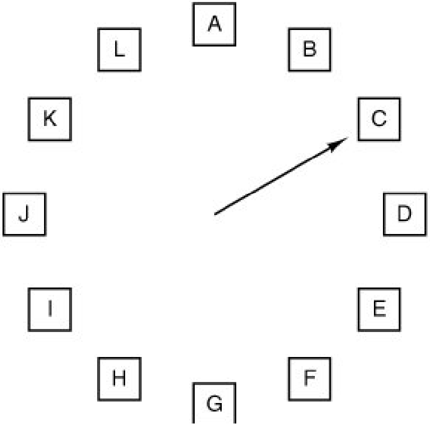
3 hits, 13 misses

Second Chance PRA

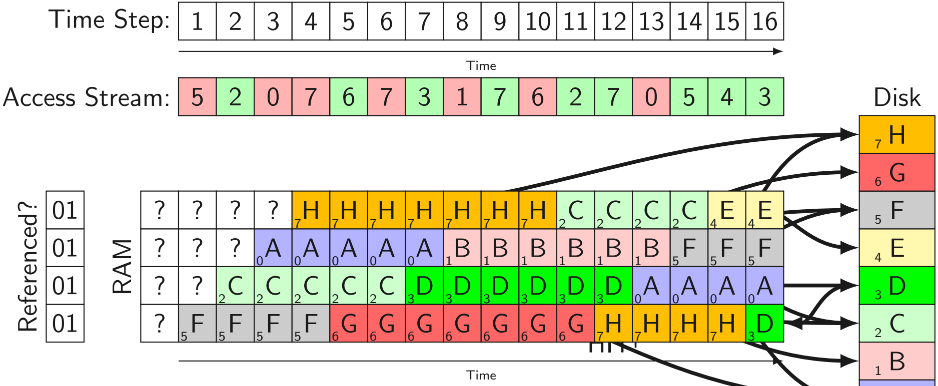
- Idea: Make FIFO great again!
  - We could call it FI(ANR)FO: "First-in-and-not-referenced first-out"
- Check "referenced"-bit:
  - $R = 0$? evict
  - $R = 1$? set $R = 0$ and go to next page
- Performance may degenerate to FIFO PRA ($\rightarrow$ which may be worse than Random PRA)

Clock PRA

- Virtually identical to Second Chance!
- Only difference is the data structure
  - Second Chance: List + List Operations (`push_back`, `pop_front`)
  - Clock: Linked List + Pointer
- Performance may degenerate to FIFO PRA (→ which may be worse than Random PRA)


A
B
C
D
E
F
G
H
I
J
K
L

Second Chance PRA / Clock PRA



3 hits, 13 misses

Least Recently Used (LRU)

- Principle of Locality: Pages that were recently accessed will more likely be accessed again
- Idea: Evict the page that was least recently accessed (used)
- How do we find this page?

Least Recently Used (LRU)

- LRU data structure:
  - (Linked) list of all pages
  - Upon access: Move page to end of list
  - Page to evict? `pop_front`
- Can this be done in software?
  - Only with extreme performance penalty (enforce every memory access to cause a page fault)
    
- Can this be done in hardware?
  - Reordering large data structures of variable size in hardware is difficult

Pseudo-LRU

- Global data structure for physical page "ages"
  - Related: Where do you store the reference count for CoW-pages?
- Upon access to a page: Store current value of `rdtsc` (cycle counter)
- Page replacement: Search data structure for lowest stored `rdtsc` value

Can we implement this?

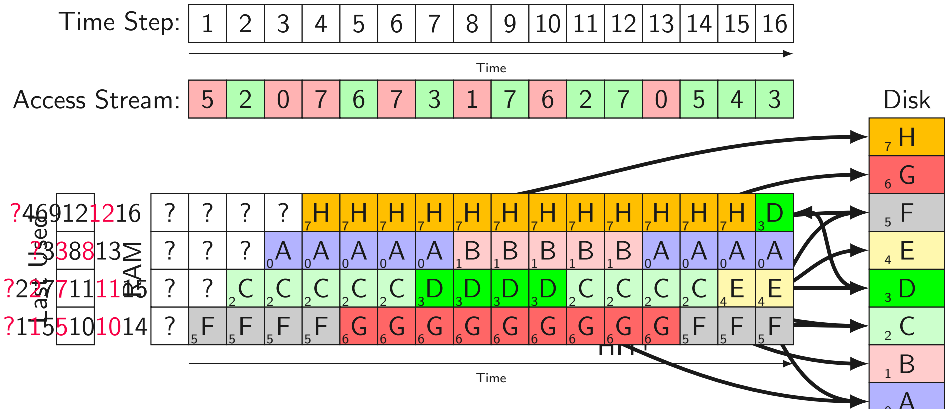
Pseudo-LRU

- Same trick as before:
  - Poll page tables: read and reset referenced bits
  - Store `rdtsc` value as age in the global data structure
- When do we do this?
  - A thread continuously running and checking
  - Upon de-scheduling

= LRU PRA (which is actually pseudo-LRU)

Performance? You have 8 MB RAM and loop over a 8.1 MB array → very bad performance

LRU PRA



4 hits, 12 misses

Not Frequently Used (NFU)

- Again: Principle of Locality 📍
- Idea: Record frequency of accesses and evict page with lowest access frequency
- Approximate frequency by access count
- How do we obtain the access count?

Not Frequently Used (NFU)

- Global data structure for physical page access frequency
- Upon access to a page: Increment access counter
- Page replacement: Search for lowest counter value

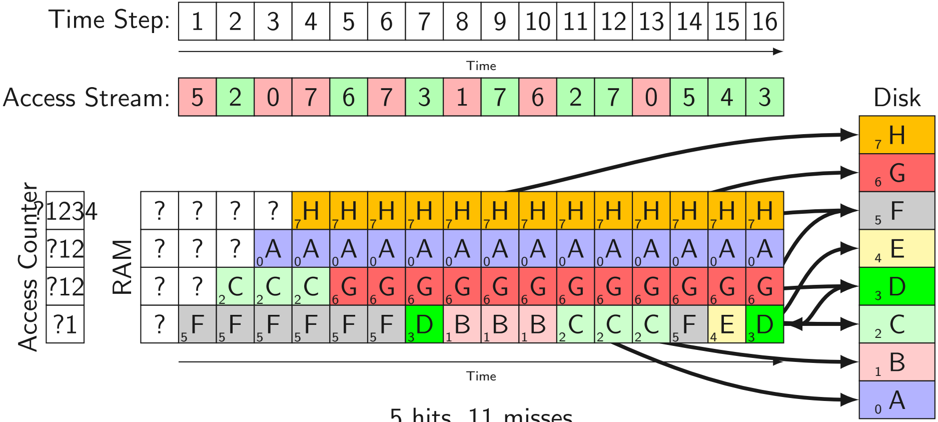
Can we implement this?

NFU PRA

- Same trick as before:
  - Poll page tables: read and reset referenced bits
  - Increment access counter in the global data structure
- When do we do this?
  - A thread continuously running and checking
  - Upon de-scheduling

= NFU PRA

- Performance? Boot code very unlikely to be swapped (because it was used a lot during boot up)



5 hits, 11 misses

- NFU has problems because it never forgets (cf. human brain)

- Idea: Make NFU's memories slowly fade away

→ Let access information age over time

- How do we observe an access to a page?

Aging

- Global data structure for age
- Upon access to a page: Set most-significant bit to 1 (e.g. `1000`)
- In a constant frequency: Age all pages by shifting value in global data structure to the right (e.g. `1000` $\rightarrow$ `0100`)
- Page replacement: Search for lowest numerical value (=highest age)

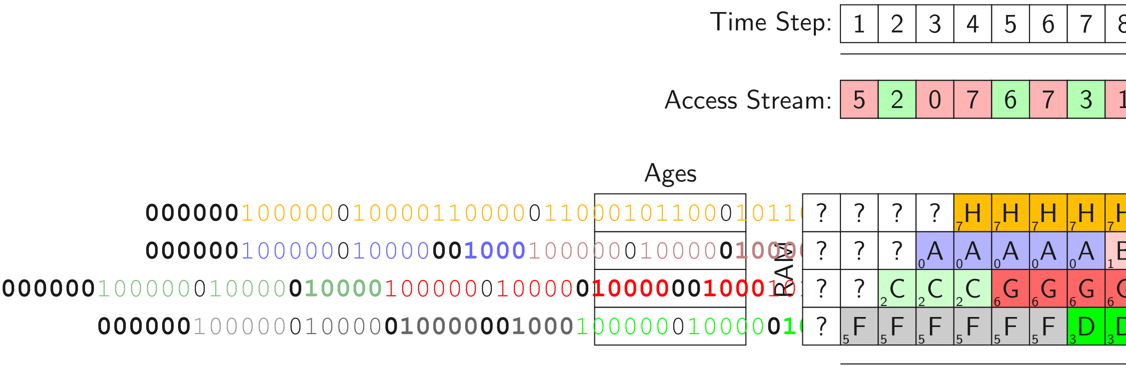
Can we implement this?

Aging

- Same tricks as before:
  - Poll page tables: read and reset referenced bits
  - Set most-significant bit in the global data structure
- When do we do this?
  - Before aging (shifting)
  - Upon de-scheduling
- When do we age (shift) the values?
  - Set up a dedicated periodic interrupt
  - Upon every *n*-th timer interrupt

= Aging PRA

- Performance? One of the most widely used PRAs in practice



4 hits,

Limitations of Aging PRA

- No age difference between pages in same aging cycle
- Limited number of bits:
    - if counter $= 0$, no difference if unused since 10 or 100 ticks

$\rightarrow$ more bits is better (but also uses more space)

Process-aware PRAs

- So far we completely ignored processes...
- Can we measure how **fair** PRAs are (wrt. processes)? 
  - Process performance? Difficult to compare...
  - Same amount of memory for every process? Tiny shell vs. 3D game

  → Same page faults per second (= page fault frequency)!
- How do we make every process have the same number of page faults per second?

Thrashing

- Thrashing: system deals more with page faults and swapping than with work 
- Processes need more RAM than exists : always too many page faults
- Page fault frequency too high? → not enough RAM

→ Swap out entire processes until page fault frequency decreases

→ Only schedule processes where all required pages are in RAM

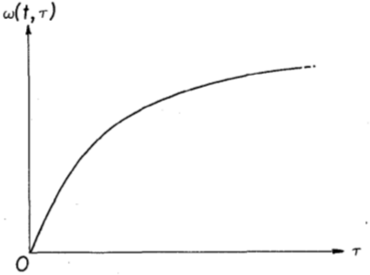
Peter Denning, 1968, abbreviated:
We define the working set $W(t, \tau)$ of a process at time t to be the collection of information referenced by the process during the process time interval $(t - \tau, t)$.

- τ = the *working set parameter*
- $\omega(t, \tau)$ = number of pages in $W(t, \tau)$

More ideas:

- Prepaging: preload all pages in the working set before scheduling
- PRA: only swap pages which are in no working set
- Adaptive τ!

Behavior of $\omega(t, \tau)$:



Adaptive Working Set Size

Define working set size by

- Time: All pages younger than τ are in the working set. (suggested by Denning)
- Huge Shift Register: shift in page number upon access. (difficult to implement)
- Page Count: The N youngest pages are in the working set.

Page fault frequency too high?

- Globally reduce τ, or the size of the shift register, or N respectively

Working Set Today

- Prepaging not common
- Working Set is no PRA ...
- ... but commonly used to form a process-aware PRA
- Same approximations as in other PRAs:
  - polling referenced bits
  - storing information in a global data structure

Working Set PRAs – The Essentials (Page Count Variant)

Working Set:

- Every process has a working set size N
- Every process has M mapped pages
- Each page has a timestamp
  - not real time, process time, `clock()`
- Working Set: The N youngest pages

Process-aware PRA:

- Any page in **no** working set (of any process) is **swappable**
- Use global PRA on **swappable** pages
  - e.g., Clock → WSClock

Adaptive process-aware PRA:

- Update N upon certain occasions
- Set $N = N - 1$ for **all** processes to reduce memory pressure
  - e.g. when trying to swap a page but none are swappable
- Set $N = N + 1$ for a process P to adjust for increasing memory usage
  - e.g. when P experiences a pagefault

→ Page fault frequency will settle to the same value for every process

Working Set PRAs – The Essentials (Time Variant)

Working Set:

- Every process has a working set parameter τ
- Every process has M mapped pages
- Each page has a timestamp
  - not real time, process time, `clock()`
- Working Set: All pages younger than τ

Process-aware PRA:

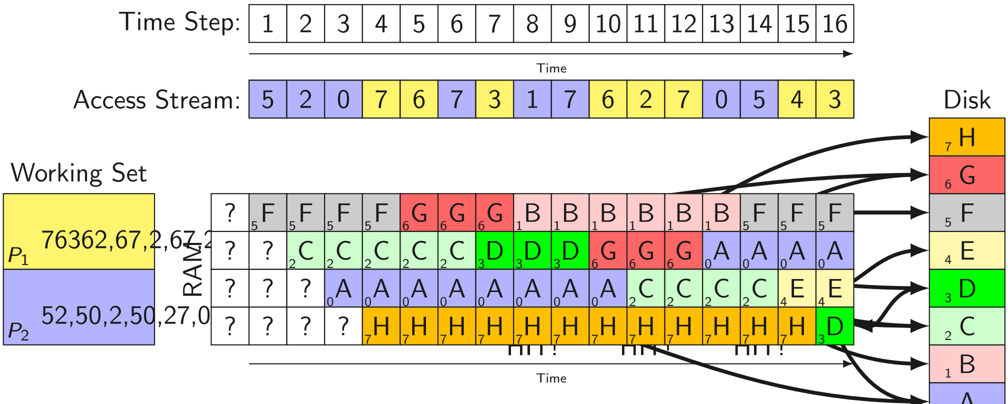
- Any page in **no** working set (of any process) is **swappable**
- Use global PRA on **swappable** pages
  - e.g., Clock $\rightarrow$ WSClock

Adaptive process-aware PRA:

- Update τ upon certain occasions
- Decrease τ slightly to reduce memory pressure
  - e.g. when trying to swap a page but none are swappable
- Increase τ slightly to adjust for increasing memory usage
  - e.g. when P experiences a pagefault

$\rightarrow$ Page fault frequency will settle to the same value for every process

WSRandom PRA



P_1: 1 hit, 7 misses P_2: 2 hits, 6 misses

PRA Design: Local vs. Global

PRA selects page for eviction ...

- **local:** ... from the same process
  - Is the working set size fixed or adaptive?
- **global:** ... from any process

Working set algorithms are inherently global

Global strategies usually perform better:

- Process needs more pages:
  - Thrashing although other processes might have spare pages
- Process needs fewer pages:
  - Memory waste despite possible thrashing in another processes

Pre-Swapping

- Page allocation latency crucial for performance
- Bad Latency when going through a lot of steps:
  1. No free physical page
  2. No clean pages
  3. Swap out page (wait for disk)
  4. Return released page to user
- Better: don't let it get this far
  - How realistic is that?

Pre-Swapping

Some classes are cheaper for swapping than others:

Class	Referenced	Dirty	Properties
0	0	0	Not used in a while and not modified $\rightarrow$ just evict

Pre-Swapping

A Paging Daemon doing Pre-Swapping

- Paging Daemon mostly inactive
- Checks regularly: Evictable/unused page frames below threshold?
  - Swap a dirty page
  - Keep it in RAM
  - Set dirty-bit to 0

→ Pre-swapped pages are evictable pages

→ Evictable pages are as good as unused pages (performance-wise)

Page Pinning

Maybe a page is required to stay in RAM?→ Pinning

Scenario:

1. A process requests I/O (e.g. `read(FD,bufferm,nrBytes)`) and blocks
2. Other processes raise page faults
   - This might replace the destination page

   → DMA transfer would go to wrong location

Avoiding this scenario:

→ Page must be locked in memory (= excluded from PRA)

- Alternatively: use (non-evictable) kernel buffers