# CS202 (003): Operating Systems Virtual Memory III, Weensy OS

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

# Last Time

# Page faults

A reference is illegal, either because it's not mapped in the page tables or because there is a protection violation.

This is a quite powerful mechanism!
(It turns out you can build interesting functionalities by triggering page faults)

# How does OS get involved in page fault (in x86)?

Process constructs a trap frame and transfer execution to an interrupt/trap handler


%rsp

stack segment (ignore)

former value of stack pointer

former value of rflags

code segment (ignore)

Instruction that caused the trap

[ ..... U/S | W/R | P]

When page fault happens, the **kernel** sets up the process's page entries properly, or terminates the process

%rip now points to the code handle the trap (using Interrupt Descriptor Table)

%cr2 holds the faulting virtual address

U/S: user mode fault / supervisor mode fault
R/W: access was read / access was write
P: not-present page / protection violation


| | | |
|---|---|---|
| P | 0 | The fault was caused by a non-present page. |
| | 1 | The fault was caused by a page-level protection violation. |
| W/R | 0 | The access causing the fault was a read. |
| | 1 | The access causing the fault was a write. |
| U/S | 0 | A supervisor-mode access caused the fault. |
| | 1 | A user-mode access caused the fault. |
| RSVD | 0 | The fault was not caused by reserved bit violation. |
| | 1 | The fault was caused by a reserved bit set to 1 in some paging-structure entry. |
| I/D | 0 | The fault was not caused by an instruction fetch. |
| | 1 | The fault was caused by an instruction fetch. |
| PK | 0 | The fault was not caused by protection keys. |
| | 1 | There was a protection-key violation. |
| SGX | 0 | The fault is not related to SGX. |
| | 1 | The fault resulted from violation of SGX-specific access-control requirements. |

Figure 4-12. Page-Fault Error Code

# When does page fault occur?

Overcommitting physical memory

*“Your program thinks it has 64GB of memory, but your hardware has 16 GB of physical memory”*

How does this work?

**Disk was (is) used to store memory pages**

Advantages: address space looks huge

Disadvantages: access to "paged" memory (as disk pages that live on the disk are known) are **slow**

**Rough Implementation**

On a page fault, the kernel reads in the faulting page. It may need to send a page to disk (when satisfy the following TWO):

1. kernel is out of memory
2. the page that it selects to write out is dirty)

# What are some other use cases of page fault?

Store memory pages across the network
(Distributed Shared Memory)

*On a page fault, the page fault handler went and retrieved the needed page from some other machine*

Copy-on-write
(fork, mmap, …)

*When creating a copy of another process, don't copy its memory. Just copy its page tables, mark the pages as read-only*

*When a write happens, a page fault results. at that point, the kernel allocates a new page, copies the memory over, and restarts the user program to do a write*
*Then, only do copies of memory when there is a fault as a result of a write*

Accounting

*Good way to sample what percentage of the memory pages are written to in any time slice: mark a fraction of them not present, see how often you get faults*

# Paging in day-to-day use

| | |
|---|---|
| Demand paging | Program code is loaded into memory only when it's needed, not all at once |
| Growing the stack | The seemingly contiguous virtual memory can scatter across different locations in physical memory |
| BSS page allocation<br>(Block Started by Symbol) | The OS can save memory by not allocating physical pages for the BSS until the program actually tries to use variables in this segment. |
| Shared text | Sharing the read-only parts of a program between multiple processes running the same program |
| Shared libraries | Multiple programs can use the same library code in memory, saving space |
| Shared libraries | Allowing multiple processes to access the same memory region |

# Costs of page faults

What does paging from the disk cost?

Average memory access time
(AMAT)

$$(1-p) * \texttt{memory\_access\_time} + \texttt{p} * \texttt{page\_fault\_time}$$

**where $p$ is the prob of a page fault**

$\texttt{memeory\_access\_time}(\texttt{t}_\texttt{M}) \approx 100\texttt{ns}$ $\texttt{disk\_access\_time}(\texttt{t}_\texttt{D}) \approx 10\texttt{ms} = 10^7\texttt{ns}$

What does $p$ need to be to ensure that paging hurts performance by less than 10%?

$$1.1 * t_M > (1-p) * t_M + p * t_D$$

$$p = 0.1 * \frac{t_M}{t_D - t_M} \approx \frac{10^1 ns}{10^7 ns} = 10^{-6}$$

Page faults are **super-expensive**!

*"need to pay attention to the slow case if it's really slow and common enough to matter."*

# A Cache System

## A Cache System

*any system that temporarily stores frequency used data*

the cache itself is smaller than the the storage it is cached on

## Cache Miss

*the requested data isn't in the cache*

1. fetch the missing data from the slower main storage
2. If the cache is full, decide which existing entry to evict to make room

How to decide which entry to throw away if we get a cache miss?

# VM as a Cache System

A Cache System

*any system that temporarily stores frequency used data*

the cache itself is smaller than the the storage it is cached on

- Virtual memory is an **abstraction** that provides programs with the illusion of a large, contiguous memory space
- Physical RAM is typically much smaller than the virtual address space
- The operating system *keeps only a subset of all pages (fixed-size blocks of memory) in physical RAM at any given time*
- The rest of the pages are **stored on disk** (in the swap space or paging file)

How to decide which page to throw away if we get a ‘page-not-present in memory’ fault?

# Replacement policy

FIFO

throw out the oldest

MIN (optimal)

throw away the entry that won't be used for the longest time

LRU

throw out the least recently used

# Replacement policy

FIFO

throw out the oldest

MIN (optimal)

throw away the entry that won't be used for the longest time

LRU

throw out the least recently used

How do we evaluate these algorithms?

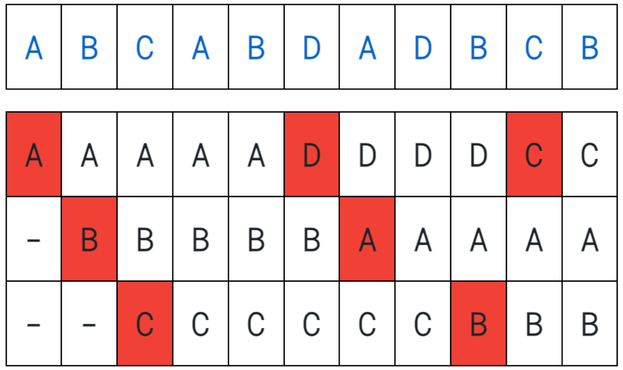
**Input:** Reference string (sequence of page accesses)
Cache size (i.e. physical memory)

**Output:** # of cache evictions (i.e. number of swaps)

# Replacement policy

## FIFO

throw out the oldest



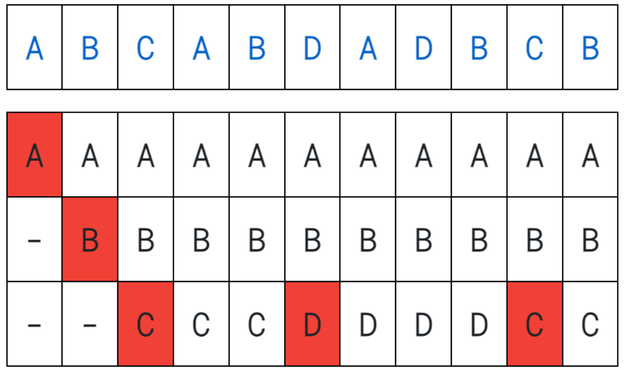

Number of Hits: 4

Page Faults: 7

Hit Rate: 36.36%

## MIN (optimal)

throw away the entry that won't be used for the longest time



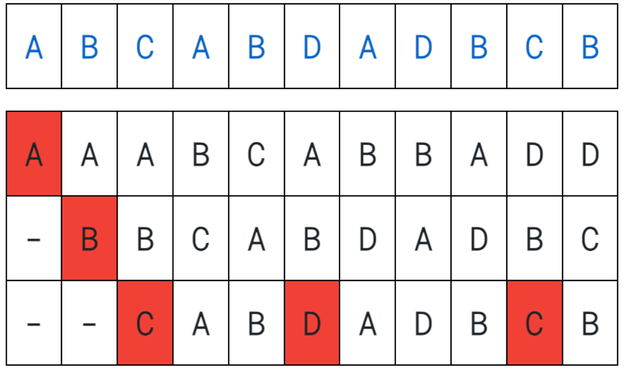

Number of Hits: 6

Page Faults: 5

Hit Rate: 54.55%

## LRU

throw out the least recently used




Number of Hits: 6

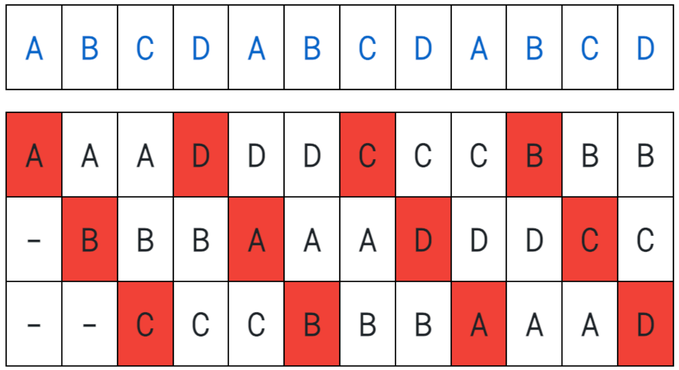
Page Faults: 5

Hit Rate: 54.55%

# Replacement policy

## FIFO

throw out the oldest



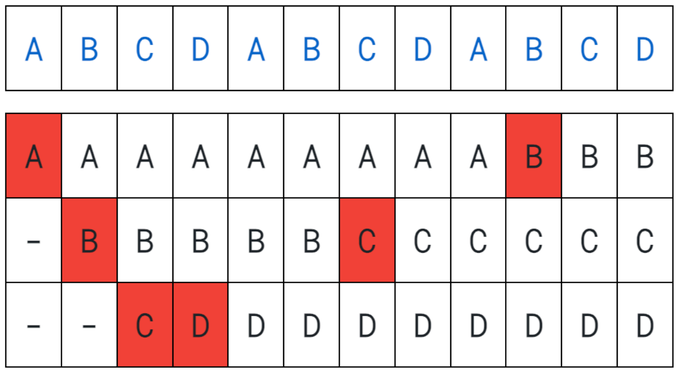

Number of Hits: 0

Page Faults: 12

Hit Rate: 0.0%

## MIN (optimal)

throw away the entry that won't be used for the longest time



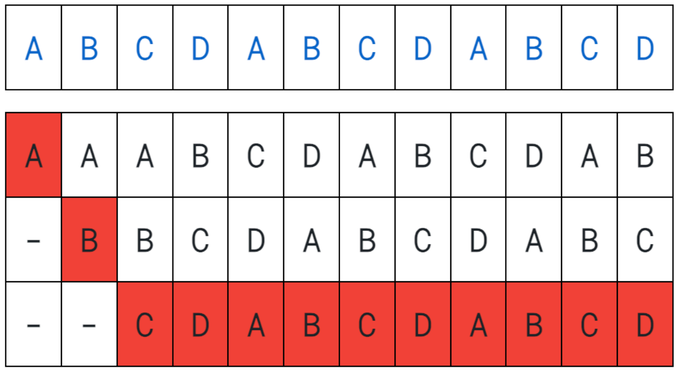

Number of Hits: 6

Page Faults: 6

Hit Rate: 50.0%

## LRU

throw out the least recently used




Number of Hits: 0

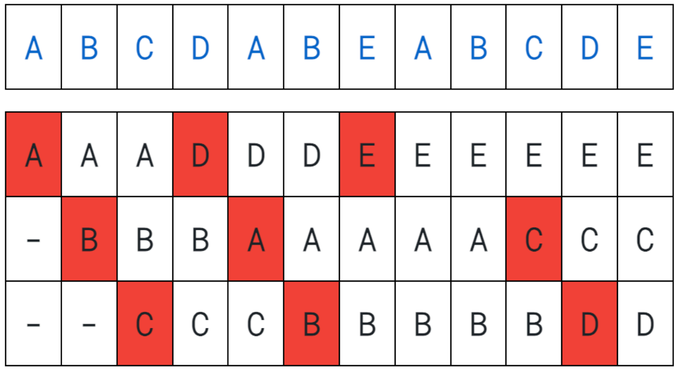
Page Faults: 12

Hit Rate: 0.0%

# Replacement policy (adding new memory)

## FIFO

throw out the oldest



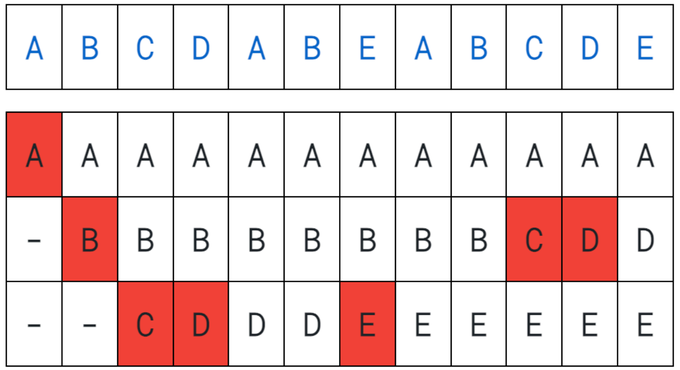

Number of Hits: 3

Page Faults: 9

Hit Rate: 25.0%

## MIN (optimal)

throw away the entry that won't be used for the longest time



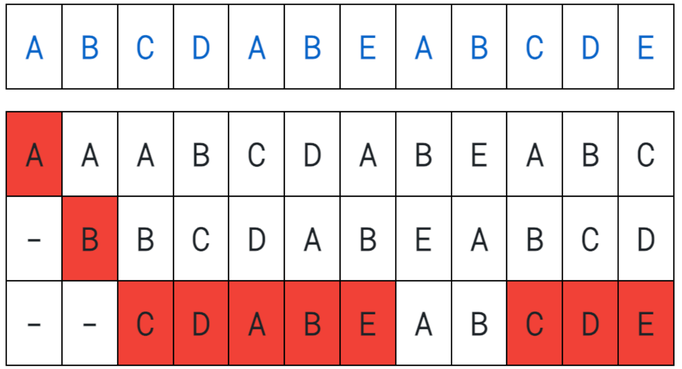

Number of Hits: 5

Page Faults: 7

Hit Rate: 41.67%

## LRU

throw out the least recently used




Number of Hits: 2

Page Faults: 10

Hit Rate: 16.67%

# Replacement policy

FIFO

throw out the oldest

MIN (optimal)

throw away the entry that won't be used for the longest time

LRU

throw out the least recently used

**Pretty decent!**

It approximates OPT when: principle of temporal locality holds strongly

# Implementing LRU

In OS, it doubles the memory traffic
(since after every reference, have to move some structure to the head of some list)

In hardware, it's **a lot of work** to timestamp each reference and keep the list ordered

Implementing LRU in OS/hardware is a lot of pain!

# Approximating LRU

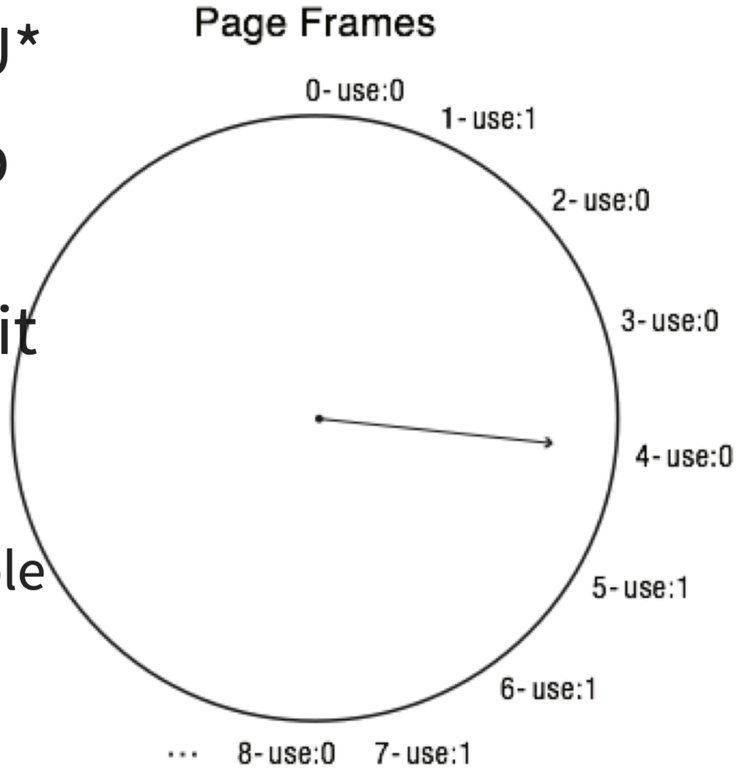
## Clock Algorithm: Not Recently Used

*"Second-chance" algorithm*

Approximating LRU*

Periodically, sweep through all pages

- Used? Clear use bit
- Unused? reclaim
  - update core map
  - invalidate page table
  - write back if dirty
  - TLB shootdown
  - add to free list




*(*yes, LRU was already an approximation...)*



https://www.cs.cornell.edu/courses/cs4410/2018sp/schedule/slides/08-vm.pdf

# Generalizing CLOCK: Nth Chance

- With each page, OS maintains a counter to indicate the number of sweeps that page has gone through.
- On page fault, OS checks accessed bit:
  - If 1, then clear it, and also clear the counter.
  - If 0, then increment the counter; if count == N, replace page.

Large N implies better approximation to LRU:
e.g., N = 1000 is a very good LRU approximation.
However, a large N implies more work by the OS before a page can be replaced.

Decent approximations to LRU, assuming that past is a good predictor of the future

*N = 1 implies the default clock algorithm.*

https://www.cse.iitd.ernet.in/~sbansal/os/lec/l30.html#:~:text=Nth%20chance%3A%20The%20clock%20algorithm,N%20chances%20before%20evicting%20it.

# Thrashing

Process requires more memory than the system has

Each time a page is brought in, another page, whose contents will soon be referenced, is thrown out

A program touches 50 pages (each equally likely) but only have 40 physical page frames

If we have enough physical pages, 100ns/ref
If we have too few physical pages, assuming every 5th reference leads to a page fault, then:
4 ref * 100 ns + 1 page fault * 10ms for disk I/O
This lead to 5 refs per (10ms + 400ns) ~ 2ms/ref = **20,000x slowdown!**

# Thrashing

Process requires more memory than the system has

Each time a page is brought in, another page, whose contents will soon be referenced, is thrown out

**What we want**: virtual memory the size of disk with access time the speed of physical memory

**What we have**: memory with access time roughly at the same magnitude as disk access

**Note:** this issue is not limited to page access, but we are discussing this issue in the context of page access

# Thrashing - what are the causes?

**What we want**: virtual memory the size of disk with access time the speed of physical memory

**What we have**: memory with access time roughly at the same magnitude as disk access

process don’t reuse memory (no temporal locality)
OR
process reuses memory but the memory that is absorbing most of the accesses doesn’t fit

**Each processes fit the memory individually, but too much to fit for all processes in the system!**

# Thrashing - What do we do?

**Each processes fit the memory individually, but too much to fit for all processes in the system!**

## Working Set

**The pages a process has touched over some trailing window of time**

Only run a set of processes s.t. the union of their working sets fit in memory

## Page fault frequency

**Track the metric**
**(# page faults/instructions executed)**

If that thing rises above a threshold, and there is not enough memory on the system, swap out the process

# Lab 4: Weensy OS
# (Yes, it is already released)

In Lab 4, you will write a mini OS, WeensyOS, that implements the virtual memory architecture and a few important system calls.

# Weensy OS structure

Processes

*Files with p-**

Kernel Code

*Files with k-**

**look at process.h for**

`sys_page_alloc()` for process allocating memory

(`sys_page_alloc` is analogous to `brk()` or `mmap()` in POSIX)

`exception_return()` for when returning back into user space

`%rax` is what the application return value is

**look at kernel.h for**

process control block (PCB): `struct proc`

* Process registers, process state
* Process page table - a pointer (kernel virtual address, which is the identical physical address)
* to an L1 page table L1 page table's first entry points to a page table, and so on...

`virtual_memory_lookup()`:

lookup a physical page using pagetable and virtual memory.

`virtual_memory_map()`:

map virtual address -> physical address

```
typedef struct physical_pageinfo {
    int8_t owner;  //kernel, reserved, free, pid
    int8_t refcount;
} physical_pageinfo;

static physical_pageinfo pageinfo
      [PAGENUMBER(MEMSIZE_PHYSICAL)];
// one physical_pageinfo struct per _physical_ page
```

pageinfo array

# Weensy OS Memory Related

WeensyOS begins with the kernel and all processes sharing a single address space.
This is defined by the kernel_pagetable.

Kernel's pagetable is identity-mapped: Virtual address X maps to physical address X.
As you work through the project, you will shift processes to use independent address space where each process can access only a subset of physical memory.

The OS supports 3MB of virtual memory on top of 2MB of physical memory.
(Recall the point of virtualization, from the perspective of the process, it thinks it has 3MB of memory. But in reality, it doesn't. )

Assume page size to be 4KB, each entry in the page table is 64 bit.
How to we support 3MB of virtual memory? How many L4 pagetable do we need?
(2 L4 page tables)

# Weensy OS Macros and Constants

| Macro | Meaning |
|---|---|
| PAGESIZE | Size of a memory page. Equals 4096 (or, equivalently, `1 << 12`). |
| PAGENUMBER(addr) | Page number for the page containing `addr`. Expands to an expression analogous to `addr / PAGESIZE`. |
| PAGEADDRESS(pn) | The initial address (zeroth byte) in page number `pn`. Expands to an expression analogous to `pn * PAGESIZE`. |
| PAGEINDEX(addr, level) | The index in the `level`th page table for `addr`. `level` must be between 0 and 3; 0 returns the level-1 page table index (address bits 39–47), 1 returns the level-2 index (bits 30–38), 2 returns the level-3 index (bits 21–29), and 3 returns the level-4 index (bits 12–20). |
| PTE_ADDR(pe) | The physical address contained in page table entry `pe`. Obtained by masking off the flag bits (setting the low-order 12 bits to zero). |

| Constant | Meaning |
|---|---|
| KERNEL_START_ADDR | Start of kernel code. |
| KERNEL_STACK_TOP | Top of kernel stack. The kernel stack is one page long. |
| console | Address of CGA console memory. |
| PROC_START_ADDR | Start of application code. Applications should not be able to access memory below this address, except for the single page at `console`. |
| MEMSIZE_PHYSICAL | Size of physical memory in bytes. WeensyOS does not support physical addresses ≥ this value. Defined as `0x200000` (2MB). |
| MEMSIZE_VIRTUAL | Size of virtual memory. WeensyOS does not support virtual addresses ≥ this value. Defined as `0x300000` (3MB). |

Bring your questions next Tuesday!