CSC369 Lecture 7

Larry Zhang, November 2, 2015

Midterm Summary

- Average: 65%, Median: 64%
  - Q1: 76% Q2: 66% Q3: 42% Q4: 48%
- Highest Mark: **48 / 50**
- Verify MarkUs has your correct mark.
- Solutions and marking schemes
  - http://www.cs.toronto.edu/~ylzhang/csc369f15/files/midterm-solution.pdf
- Remarking requests
  - Fill in this form: http://www.cs.toronto.edu/~ylzhang/csc369f15/files/midterm-remarking.pdf
  - Attach it to the test and submit to Larry.
- Drop date: November 4th

A note on final exam

The average difficulty of final exam questions will be higher than the midterm.

➔ Midterm tests your quick reactions within a short amount of time.

➔ In the final exam you'll have enough time to think everything through.

A2 FAQ

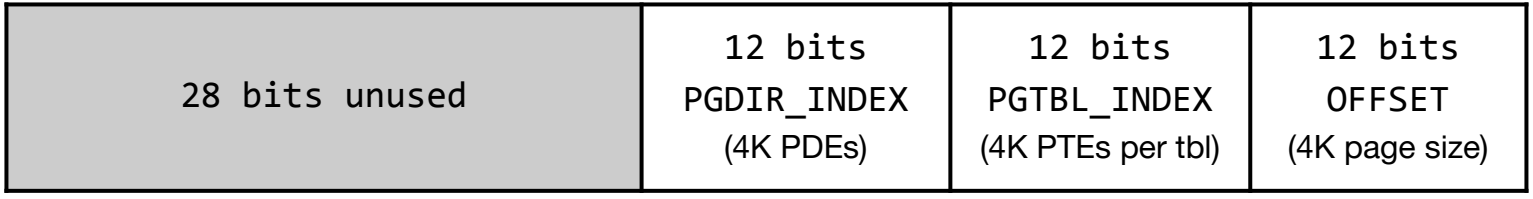
There is **NO** FAQ because you didn't ask questions frequently!

You'll be 100% ready for A2 after today's lecture, there will be no excuse not to start!

Ask questions (for everyone in this class)!

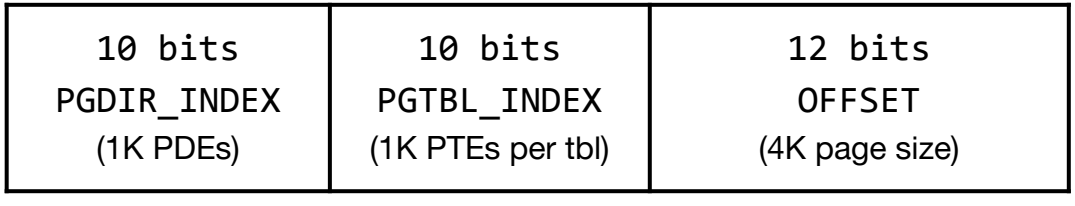
Tips and FAQs will be updated on Piazza.

A2: 64-bit / 32-bit virtual address traces



64-bit

```
PGDIR_SHIFT = 24 bit, i.e.,
PGDIR_INDEX = VADDR >> 24
```



32-bit

```
PGDIR_SHIFT = 22 bit, i.e.,
PGDIR_INDEX = VADDR >> 22
```

You code does NOT need to worry about the difference between 32 and 64 because you just use variable names instead of hard-coded numbers.

Development Environment for A2

- ➔ Any Linux should work well
- ➔ MacOS with gcc installed should also work
- ➔ Don’t use the disk image from A1.
- ➔ Make sure to test everything on cslinux before submission.

Paging continued...

Recap

- ➔ Paging is a good approach for memory virtualization, because of its simplicity and its fully virtualized view of address space.
- ➔ But paging also has its problems
  - ◆ Page tables can be too large
    - solution: multi-level page tables
  - ◆ Paging can be too slow, too…

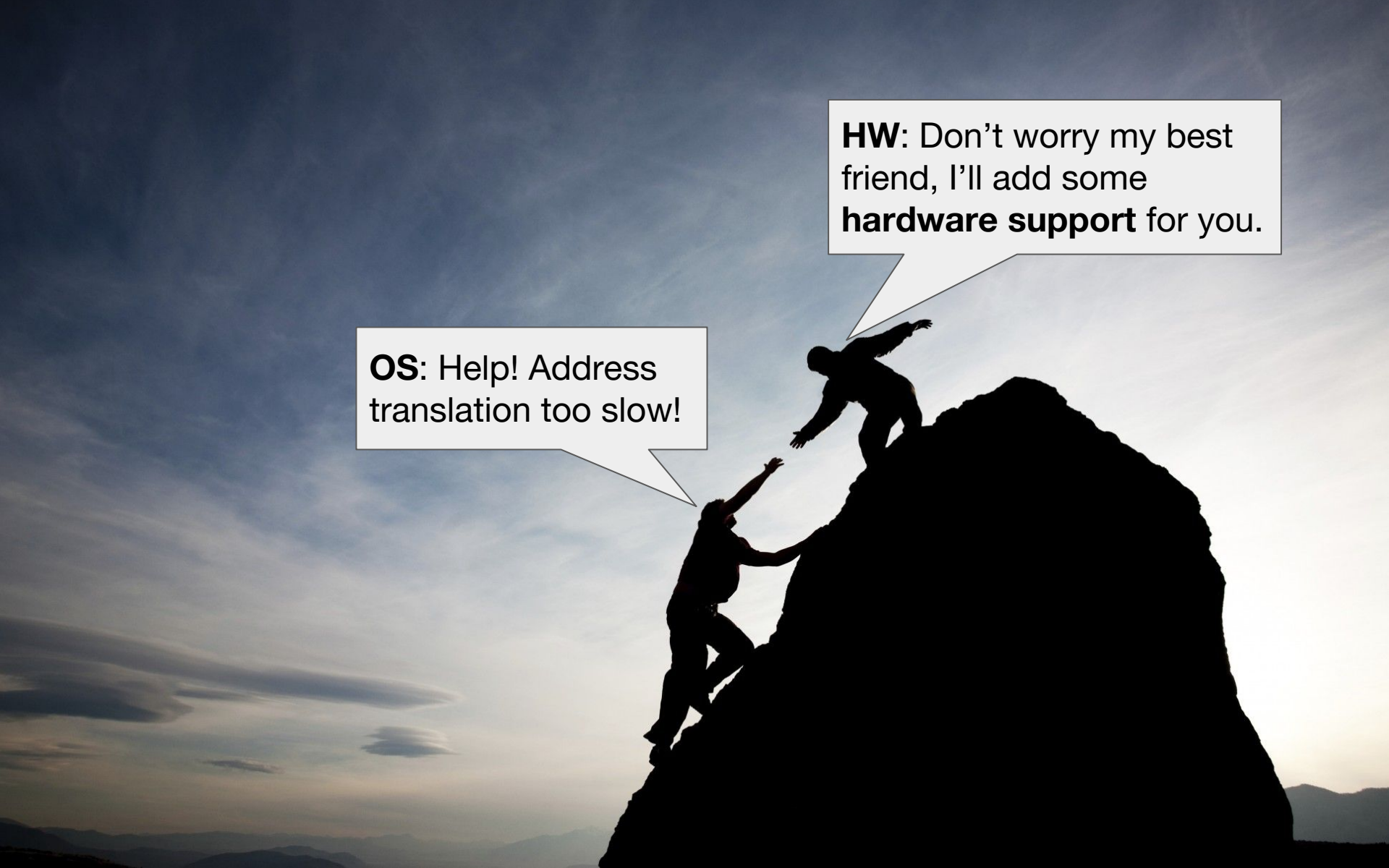
Paging can be too slow...

VPN_MASK: a binary with 1's at the bit locations for VPN, and 0's at all other bits. OFFSET_MASK: similar.

What happens when we access a virtual address (assume linear page table)

1. Extract VPN from virtual address:
   a. `VPN = (VAddr & VPN_MASK) >> SHIFT`
2. Compute the physical address of the PTE
   a. `PTEAddr = PTBR + (VPN * sizeof(PTE))`
3. Fetch the PTE
   a. `PTE = AccessMemory(PTEAddr)`
4. If `PTE.ValidBit` is False, raise `SEGMENTATION_FAULT`
5. else if cannot access because `PTE.ProtectBits`, raise `PROTECTION_FAULT`
6. else
   a. `Offset = VAddr & OFFSET_MASK`
   b. `PhysAddr = (PTE.PFN << PFN_SHIFT) | Offset`
   c. `AccessMemory(PhysAddr)`

For each memory access, we have to perform (costly) memory access **twice**! That's a slow-down by a factor of two! Even **worse** for multi-level page tables!


HW: Don’t worry my best friend, I’ll add some **hardware support** for you.
OS: Help! Address translation too slow!

to speed up address translation, hardware provides

Translation **L**ookaside **B**uffer

which really should have been called

address-translation cache

so TLB is basically cache

- ➔ it is part of the Memory Management Unit (MMU)
- ➔ small, fully associative hardware cache of recently used translations
  - ◆ **small**, so it's **fast** by laws of physics
  - ◆ **fully associative**, i.e., all entries looked up in parallel, so it's **fast**
  - ◆ **hardware**, so it's **fast**
  - ◆ It is **so fast** that the lookup can be done in a **single CPU cycle**.
- ➔ a successful lookup in TLB is called a **TLB hit**, otherwise it is a **TLB miss**

What is in TLB?

- ➔ lookups: **VPN -> PFN** (plus some other bits)
  - ◆ from current process' VPN to the physical address' PFN
- ➔ Each entry is like

VPN	PFN	other bits (valid, protect, etc)

- ➔ a TLB typical has 32, 64 or 128 entries.

TLB Issue #1: Context Switch

- ➔ Context switch **invalidates** all entries in TLB. Why?
  - Because the VPN stored in a TLB entry is for "**current**" process, which becomes meaningless when switched to another process.
  - Could lead to **wrong translation** if not careful.
- ➔ Possible solutions:
  - simply **flush** the the TLB on context switch, i.e., set all valid bits to 0.
    - safe, but inefficient.
    - Think of two Processes A and B that frequently context switch between each other.
      - every time switched out and in again, have to start all over filling TLB.
  - add Address Space Identifier (ASID) to TLB entry
    - it's basically PID, but shorter (e.g., 8 bits instead of 32 bits)
    - avoids wrong translation without having to flush all entries.

TLB Issue #2: Replacement Policy

- ➔ When TLB is full, and we want to add a new entry to it, we will have to **evict** an existing entry.
- ➔ Which one to evict?
- ➔ Will discuss more about it soon...

Control Flow with TLB

```
 1   VPN = (VAddr & VPN_MASK) >> SHIFT
 2   (Success, TLBEntry) = TLB_Lookup(VPN)
 3   If Success:  // TLB Hit
 4      if CanAccess(TLBEntry.Protect):
 5         Offset = VAddr & OFFSET_MASK
 6         PhysAddr = (TLBEntry.PFN << SHIFT) | Offset
 7         AccessMemory(PhysAddr)
 8      else:
 9         RaiseException(PROTECTION_FAULT)
10   else:  // TLB Miss
11      RaiseException(TLB_MISS)
       // OS then traps into kernel mode, then jumps to
       // a trap handler that handles TLB misses,
       // i.e., lookup in page table, insert to TLB, evict, …
       // then return-from-trap, and retry the translation
       // (which will be a TLB Hit)
```

This is a (modern) approach where **software** (OS) handles TLB misses; some older systems also also used **hardware** to handle TLB misses.
The software approach is more flexible and simple.
Hardware approach is faster but is inflexible.

TLBs exploit locality

- ➔ Processes only use a handful of pages at a time
  - ◆ a TLB with 64 entries can map 64 * 4K = 192KB of memory, which usually covers most of the frequently accessed memory by a process within certain timespan.
- ➔ In reality, TLB **hit rates** (hit / (hit + miss)) are typically very high (> 99%)
- ➔ Caching is an important idea, use it when possible
  - ◆ In fact, we will use it again very soon.

Which is faster? Why?


```
#include <stdlib.h>
#define N 500000000

int main(int argc, char **argv) {
    int i;
    int *x = malloc(N * sizeof(int));
    if(!x) return 1;
    for (i = 0; i < 100000; i++)
        x[i] = 0;
    return 0;
}
```

```
$ time ./a.out

real 0m0.004s
user 0m0.000s
sys  0m0.003s
```

```
#include <stdlib.h>
#define N 500000000

int main(int argc, char **argv) {
    int i;
    int *x = malloc(N * sizeof(int));
    if(!x) return 1;
    for (i = 0; i < 100000; i++)
        x[i*5000] = 0;
    return 0;
}
```

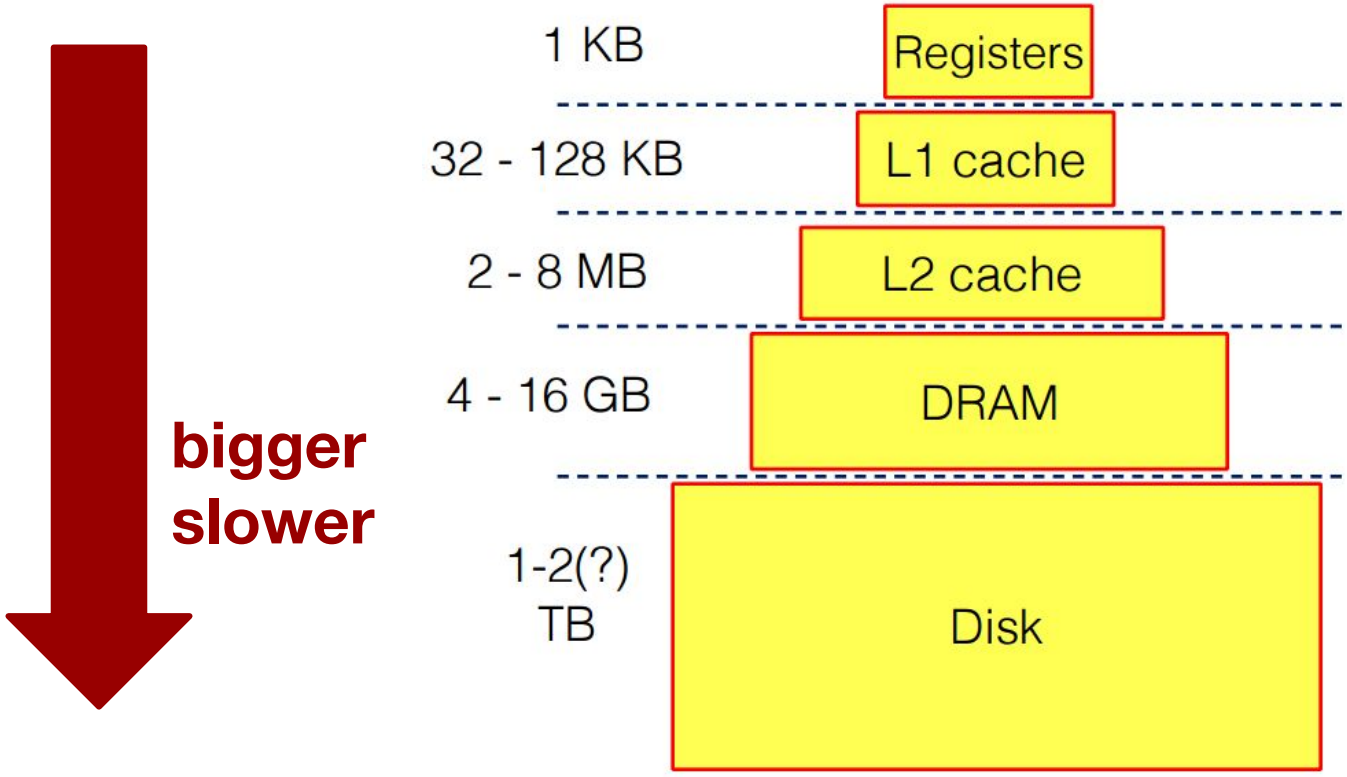
```
$ time ./a.out

real 0m0.874s
user 0m0.007s
sys  0m0.865s
```

RAM isn't always RAM (random-access memory)!

Knowing OS lets you engineer your code's performance like never before!

demand paging (swapping)

Quick Recap: The Memory Hierarchy



Network

demand paging

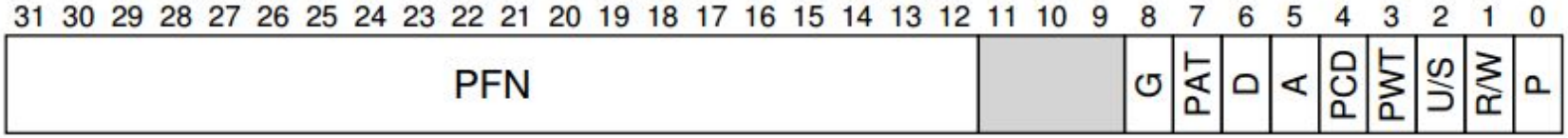
It would be nice if we can keep all memory pages in RAM.

But sometimes it is impossible.

- ➔ Have many processes using memory, in combination exceeds the size of physical memory.
- ➔ Even one process's memory usage can be larger than the size of physical memory, because virtual address space can be larger than the physical space.
- ➔ Then we have to use the **hard disk** to store some of the pages, e.g., the pages that are not being accessed.
- ➔ so we can still provide this beautiful illusion of a large virtual address space.
- ➔ The mechanism of moving pages between memory and disk is called **demand paging**.

demand paging

- **swap space (swap file)**: a **reserved** space on hard disk for moving pages back and forth.
  - UNIX/Linux: a separate disk partition (try `swapon -s`)
  - Windows: a gigantic binary file called “pagefile.sys”
- Initially pages are allocated from memory.
- When memory fills up, allocating more page requires some other pages to be evicted from memory
- Evicted pages go to disk (**page out**).
  - where? the swap space. Need to remember the disk address (swap offset)



- ➔ a **present bit** in the PTE is used to indicate whether a page in present in memory (1) or on disk (0)
- ➔ if the present bit is 0, then the other bits in the PTE actually stores the **disk address** of the page.
  - ◆ the address was updated when it paged out
- ➔ after **paging in** a page, the present bit is set to 1 and the content of PTE is updated correspondingly.
- ➔ In A2, use **VALID** to indicate if it is in memory, and use **ONSWAP** to indicate if it is on disk.

OS needs to keep track the usage of each physical frame, so that it can allocate physical frames when demanded.

➔ use the **coremap** data structure

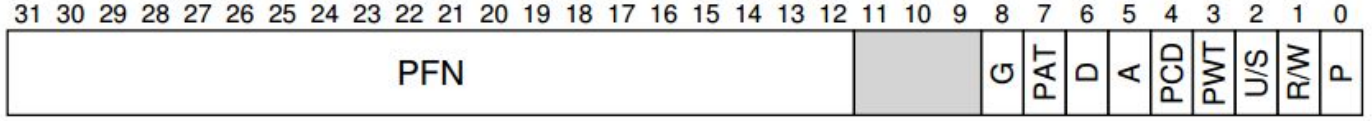
➔ for each physical frame (`frame` struct in A2), keep

- PTE of the virtual page that is allocated at this frame.
- Address Space ID or PID
  - In Assignment 2 we don't store this because we only simulate one process at a time.
- whether the frame is in use
- usage info for replacement algorithms
  - in A2 you can add necessary attributes to the `frame` struct.

Demand paging control flow

- ➔ If page is present in memory, then it's fine, do the same thing as before
- ➔ When it evicts a page, the OS sets zero the present bit of the PTE and stores the location of the page in the swap space in the PTE
- ➔ When a process accesses the page, the non-present PTE will cause trap -- this is called **page fault**
- ➔ The trap will run the OS **page fault handler**
- ➔ Handler uses the non-present PTE to locate page in swap file
- ➔ Reads page into a physical frame, updates PTE to point to it
- ➔ Restarts process (and the page will be present)

Nuance



➔ PTE usually has a **dirty bit** to indicate whether that page has been modified since it was last paged in.

- if **dirty == 0**, then no need to write to disk when paged out, since nothing has changed
- if **dirty == 1**, then must write to disk when paged out.
- Invented by Corbato (the MLFQ guy)

Quick summary of demand paging

- ➔ Essentially, the memory acts like the **cache** of the swap space.
- ➔ Like in all cache-like mechanisms, we need to worry about one common issue …
- ➔ **the replacement policy**
- ➔ when we have to evict a frame to disk, which one should we choose?

Replacement Policies

Replacement algorithm

➔ Something to worry about for any cache-like mechanism

- in demand paging, physical page frames are the cache of swap space
- our discussion will be in more general abstract terms, such as using “items” instead of “pages”.

➔ Goal of algorithms:

- reduce fault/miss rate by selecting the **best victim** to evict

Assumption (an unrealistic one)

We know the **whole** memory reference trace of the program

including the **future** ones at any point in time.

Which is the best possible victim to evict?

Current cache

3	1	6	9

The whole memory trace:

1, 3, 6, 9, 3, **4**, 9, 5, 1, 3, 4, 9, 7, 9, 3 (END)

Need to evict someone here. Which one is the best victim?

Evict the item (6) that will **never** be accessed again, so will never miss on it.

So Algorithm #1: evict the one that will never be used again…, good.

But does this algorithm always work?

Current cache

3	1	6	9

Algorithm #1 busted.
Now what?

The whole memory trace:

1, 3, 6, 9, 3, **4**, 9, 5, 1, 3, 4, 9, 7, 9, **6**, 3 (END)

best victime is still 6.

The best thing now is to evict the item whose next access is the **furthest in the future**, so that we will miss as **infrequently** as possible.

So Algorithm #2

- ➔ Evict the page whose next access is the **furthest in the future**.
- ➔ This algorithm is called **Belady's algorithm**, a.k.a., **MIN** or **OPT.**
- ➔ It is **proven** to achieve the **optimal** hit rate, i.e., no other algorithm can possibly do better that it.
- ➔ Developed by László Bélády in 1966.

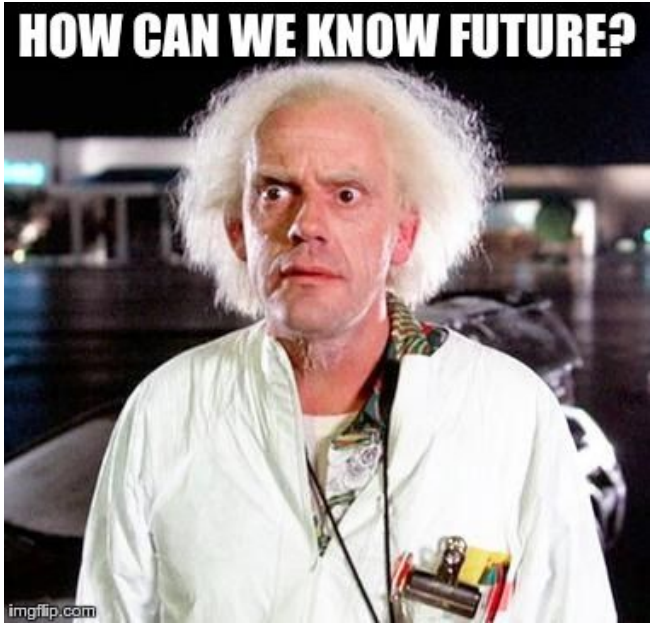
Attention CSC373 students!

A quiz from Dan:

Belady's algorithm is a **greedy** algorithm.

Prove its optimality, and discuss with Dan.

Belady’s algorithm is great, except that ...



In practice, Belady’s algorithm is useful as a yardstick for measuring **how close** other algorithms are to “perfect”.

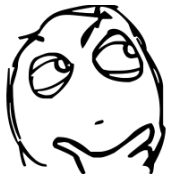
Aside: types of cache misses

➔ **cold-start miss**: initially the cache is empty, so the first reference to an item must be a miss.

➔ **capacity miss**: occurs because the cache ran out of space and had to evict an item to bring a new item into the cache.

so, other replacement algorithms

that we can actually use in practice



So, after all, we don’t know the future.

But somehow we want to predict the future behaviours of a computer program. How?

Imagine how your predict the future behaviour of a person...

We can often do that pretty well because people’s behaviours follow certain **patterns**.

Computer programs are people, too.

Locality: the patterns in computer program behaviours

Programs are not just random sequences of instructions, they are implemented in specific ways, so their behaviour follow certain patterns.

- ➔ **Spatial locality**: it an address A is accessed, then addresses A-1 and A+1 are also likely to be accessed.
  - ◆ Example?
- ➔ **Temporal locality**: if an address is accessed at time T, then it is also likely to be accessed again in the near future T + Δt
  - ◆ Example?
- ➔ It is not a rule that all programs must obey, but in general it is a good heuristic to keep in mind when designing computer systems.

FIFO: a simple policy

FIFO

- an item is appended to the tail of a **queue** when they enter the system; when a replacement occurs, the element at the front of the queue (the **first-in** element) gets evicted.

- When is FIFO good?
  - when the oldest item is **unlikely** to be used again
- When is FIFO bad?
  - when the oldest item is **likely** to be used again
  - we don't have any info to say whether it is likely or not
- FIFO suffers from "Belady's Anomaly"
  - the miss rate might **increase** when the cache becomes **bigger**.

Example: FIFO with queue size **3**

cold-start miss
capacity miss
HIT

reference trace: *0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4*

	0	***1***	***2***	***3***	***0***	***1***	***4***	***0***	***1***	***2***	***3***	***4***
newest	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
oldest			0	1	2	3	0	0	0	1	4	4

9 misses

Example: FIFO with **larger** queue size **4**

cold-start miss
capacity miss
HIT

reference trace: *0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4*

	0	***1***	***2***	***3***	***0***	***1***	***4***	***0***	***1***	***2***	***3***	***4***
newest	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
oldest				0	0	0	1	2	3	4	0	1

10 misses, i.e., bigger cache size but more misses, Belady's anomaly

Things start being funny here, where each reference requests an item that's just evicted.

FIFO: Verdict

➔ It’s simple to implement

➔ but it’s performance is not good because it doesn’t understand the programs’ behaviours well enough, even it is somehow trying to...

Random: an even simpler policy

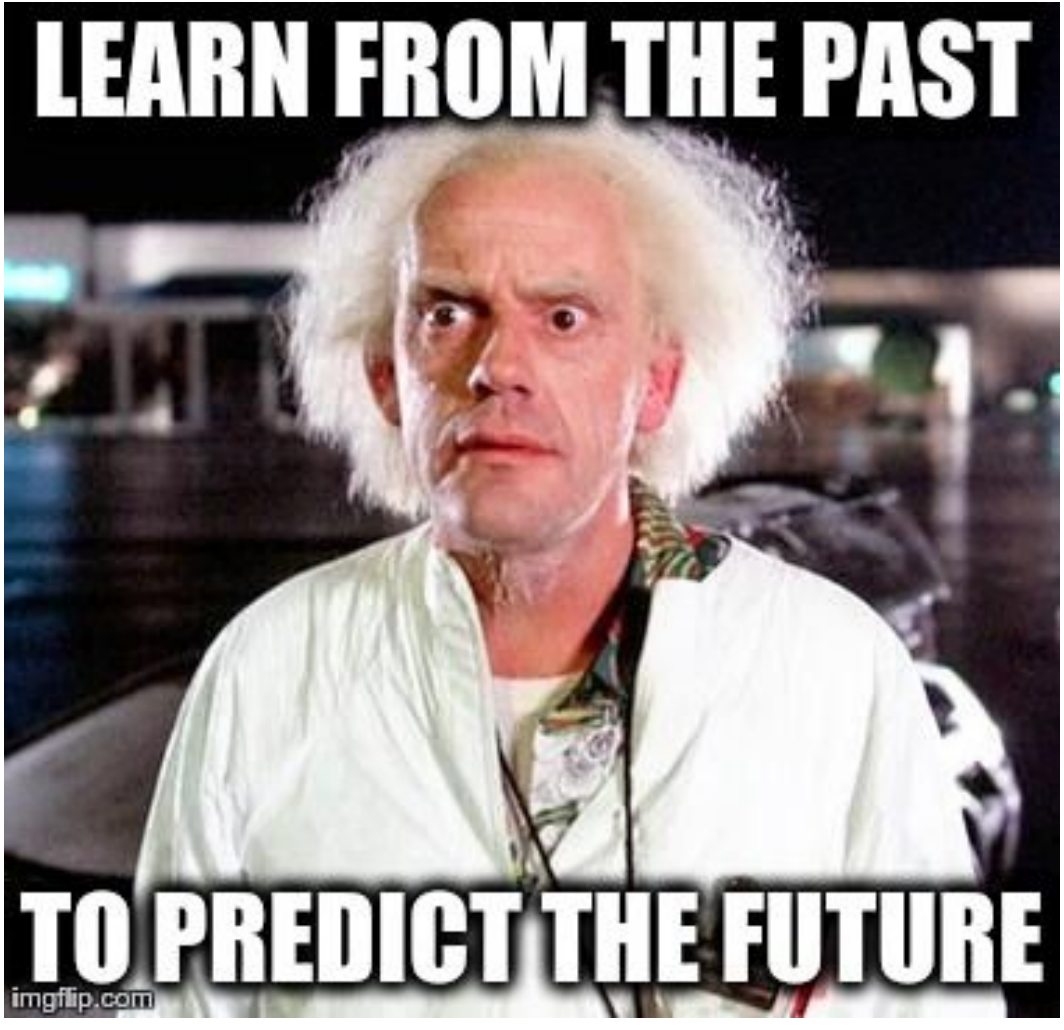
Random

- ➔ Simply pick a random victim to evict, not even trying to be intelligent.
- ➔ Performance depends on luck, but in practice it is usually a little better than FIFO.
- ➔ It doesn’t suffer from the corner cases like Belady’s anomaly.
- ➔ Because of its super-simplicity, it is actually used in practice, e.g., TLB replacement is usually Random.

LRU the Almighty

Least Recently Used (LRU)

- ➔ Sometimes you have so many apps in your phone that you have to delete some.
- ➔ How to do you decide which to delete?
- ➔ Delete the one that you **haven't used for the longest time**. Why?
- ➔ Because if I haven't used it for a long time in the **past**, then probably I will not use it in a long time in the **future**.
- ➔ feels like we've seen this approach before ...


LEARN FROM THE PAST
TO PREDICT THE FUTURE
imgflip.com

Example: LRU with queue size **4**

cold-start miss
capacity miss
HIT

reference trace: *0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4*

	0	***1***	***2***	***3***	***0***	***1***	***4***	***0***	***1***	***2***	***3***	***4***
MRU	0	1	2	3	0	1	4	0	1	2	3	4
		0	1	2	3	0	1	4	0	1	2	3
			0	1	2	3	0	1	4	0	1	2
LRU				0	1	2	3	3	3	4	0	1

8 misses

LRU is awesome

- ➔ Theoretically, you can construct a worst-case input on which LRU performs very poorly.
- ➔ But in practice, its performance is **near-optimal**, i.e., very close to the hit rate reached by Belady's OPT algorithm.
- ➔ This is an example where "worst-case analysis" does not tell the whole story.
- ➔ A family of similar policies exist such as MRU, MFU (most frequently used) and LFU, but nobody beats LRU.

Implementing Exact LRU

Option 1:

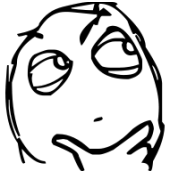
- ➔ Remember a timestamp for every reference
- ➔ Evict page with oldest timestamp
- ➔ Issues:
  - ◆ Need to make PTE large enough to hold the timestamp
  - ◆ Need to examine every page on eviction to find the one with the oldest timestamp

Option 2:

- ➔ Keep pages in a stack. On reference, move the page to the top of the stack.
- ➔ On eviction, replace page at bottom.
- ➔ Issues:
  - ◆ Need costly software operation to manipulate stack on EVERY memory reference.

Exact LRU is not very easy to implement.

So it’s worth considering policies that **approximate** LRU but easier to implement.



Second Chance Algorithm

- ➔ Each PTE has a **reference bit** (or use bit) to indicate whether it has been used.
- ➔ try from page with oldest arriving timestamp, also check the reference bit
  - ◆ if ref bit is 0, then replace the page (the page is old and not use recently)
  - ◆ if ref bit is 1, don't replace, set ref bit to 0, and set arrival time to current time (the page is old but was recently used, so give it a second chance)
  - ◆ pages that are used often enough to keep ref bit being 1 will not be replaced.
- ➔ maintaining and checking timestamps is cumbersome, need to do something clever.

Cleverly Implementing Second Chance (**Clock**)

- ➔ Arrange all pages frames in a big circle (clock).
- ➔ The clock hand is used to choose good LRU candidate.
- ➔ When replacement must occur, check the item currently pointed by clock hand
  - ◆ if ref bit is 1, then the page is recently used, so NOT a good candidate to evict; set ref bit to 0 and move forward.
  - ◆ if ref bit is 0, then the page must have been unused for **at least one clock circle**. Replace it.
    - not exactly the least-recently used, but it's "**old enough**".

Brief mentions: other policies

- ➔ Real OS typical don't run replacement upon **every single** page fault.
  - ◆ maintain a pool of free pages
  - ◆ run replacement when pool becomes too small ("**low water mark**"), free enough pages to refill pool up to "**high water mark**".
  - ◆ batch processing is more efficient
- ➔ **Prefetching**: OS can guess a page is about to be used and thus bring it in ahead of time.
- ➔ **Thrashing**: when memory is simply oversubscribed by too many running processes, the system will be paging constantly. To avoid it
  - ◆ **admission control**: prevent too many processes running beforehand
  - ◆ **out-of-memory** killer: when thrashing happens, choose a memory-intensive process and kills it; used by Linux

Summary

Let’s review the whole control flow of a memory access with TLB, two-level page table and demand paging.

```
VPN = (VAddr & VPN_MASK) >> SHIFT
(Success, TLBEntry) = TBL_Lookup(VPN)
if Success: // TLB Hit
   if CannAccess(TLBEntry.Protect):
      Offset = VAddr & OFFSET_MASK
      PhysAddr = (TLBEntry.PFN << SHIFT) | Offset
      data = AccessMemory(PhysAddr)
   else:
      RaiseException(PROTECTION_FAULT)
else: // TLB Miss
   PDIndex = (VPN & PD_MASK) >> PD_SHIFT
   PDEAddr = PDBR + PDIndex * sizeof(PDE)
   PDE = AccessMemory(PEDAddr)
   if not PDE.Valid:
      RaiseException(SEGMENTATION_FAULT)
   else:
      PTIndex = (VPN & PT_MASK) >> PT_SHIFT
      PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
      PTE = AccessMemory(PTEAddr)
      if not PTE.Valid:
         RaiseException(SEGMENTATION_FAULT)
      else:
         if not CanAccess(PTE.Protect):
            RaiseException(PROTECTION_FAULT)
         else if PTE.Present:
            TLB_Insert(VPN, PTE)
            RetryInstruction()
         else if not PTE.Present: // Page Fault
            PFN = FindFreePhysicalPage()
            if PFN == -1:
               PFN = EvictPage()
            DiskRead(PTE.DiskAddr, PFN)
            PTE.Present = True
            PTE.PFN = PFN
            RetryInstruction()
```

```
INPUT: VAddr
OUTPUT: data
```

How many time does an instruction have to run if it is not in TLB and is on swap?

➔ 3 times

This whole flow is quite complicated, but for the user…

```
VPN = (VAddr & VPN_MASK) >> SH
(Success, TLBEntry) = TBL_Look
if Success:
   if CannAccess(TLBEntry.Protect):
      Offset = VAddr & OFFSET_MASK
      PhysAddr = (TLBEntry.PFN << SHIFT) | Offset
      data = AccessMemory(PhysAddr)
   else:
      RaiseException(PROTECTION_FAULT)
else:
   PDIndex = (VPN & PD_MASK) >> PD_SHIFT
   PDEAddr = PDBR + PDIndex * sizeof
   PDE = AccessMemory(PEDAddr)
   if not PDE.Valid:
      RaiseException(ST
   else:
      PTIndex
      PTE                                    (PTE))
                                          LT)
                                  Protect):
                                  PROTECTION_FAULT)
                                  esent:
                                  sert(VPN, PTE)
                                  tryInstruction()
                  else if not PTE.Present:
                        PFN = FindFreePhysicalPage()
                        if PFN == -1:
                           PFN = EvictPage()
                        DiskRead(PTE.DiskAddr, PFN)
                        PTE.Present = True
                        PTE.PFN = PFN
                        RetryInstruction()
```

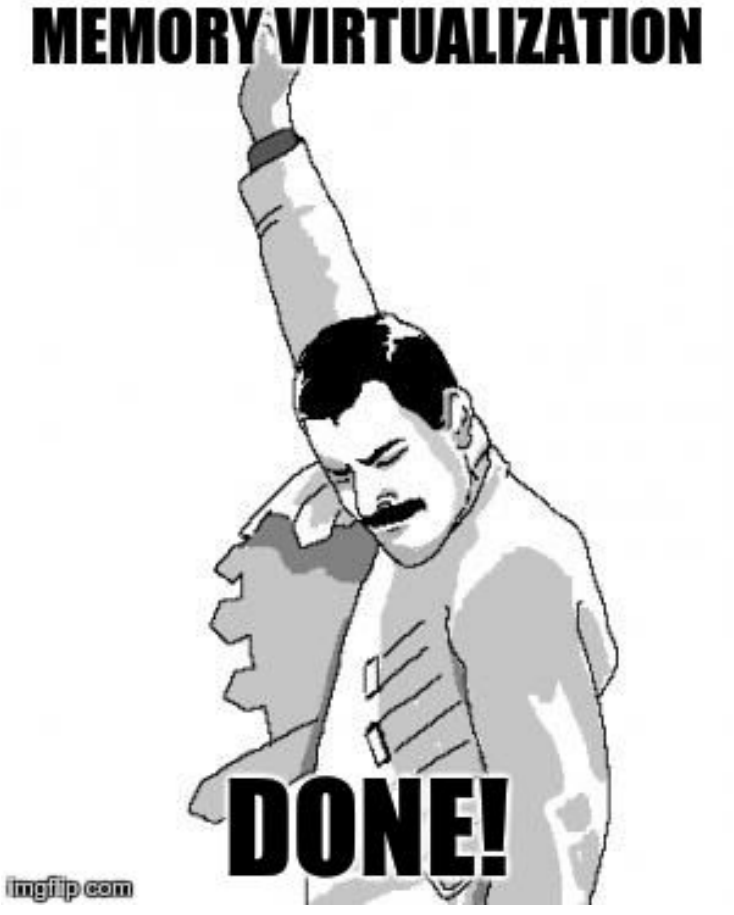
Users only see this!

```
INPUT: VAddr
OUTPUT: data
```

TRANSPARENT

This whole flow is quite complicated, but for the user…

it's all transparent!


MEMORY VIRTUALIZATION
DONE!
imgflip.com

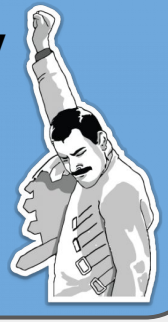
Virtualization

Virtualizing CPU
- Processes
- Threads
- Scheduling ...

Virtualizing Memory
- Address space
- Segmentation
- Paging ...

Concurrency

- Threads
- Locks
- Conditional variables
- Semaphores ...



Persistence

- File systems
- Journaling, logging
- I/O devices
- File integrity ...

next week

Tutorial this week: Exercise with Replacement Algorithms