Mechanism – Paging: **Finding Free Page Frame**

2023/24 COMP3230B

Not Enough Physical Memory

- Assumption 1
  - ~~Assume the address space of a process is relatively small, i.e., size of physical memory is much larger than a process’s address space.~~

- In real life, process’s address space is quite large (e.g., 48-bit → 256 TiB), but physical memory is limited (e.g., 16 GiB)

- The Crux
  - If the system does not have enough physical memory, how to run many processes at the same time?

Contents

- To support **multiple** running processes, each with large address space, OS uses storage disks to **temporarily** store **portions** of processes' address spaces
  - Swap Space and Page Fault
- We need to move out some virtual pages and make way for newly request or access pages
  - Replacement Policies
  - Evaluation of the policies
- Thrashing

Related Learning Outcomes

- ILO 2b - describe the principles and techniques used by OS in effectively virtualizing memory resources.

- ILO 3 [Performance] - analyze and evaluate the algorithms of . . . and explain the major performance issues . . .

Readings & References

- Required Readings
  - Chapter 21 – **Beyond Physical Memory: Mechanisms**
    - http://pages.cs.wisc.edu/~remzi/OSTEP/vm-beyondphys.pdf
  - Chapter 22 – **Beyond Physical Memory: Policies**
    - http://pages.cs.wisc.edu/~remzi/OSTEP/vm-beyondphys-policy.pdf

The Crux

- How can OS make use of a larger, slower device to ***transparently*** provide the illusion of a **large virtual address space**?

Swap Space

- Most OSs create a special area (partition) of the disk as **swap space**
  - OS swaps virtual pages out of physical memory to it and swaps virtual pages back into physical memory from it

- To mitigate the performance overhead due to swapping
  - This partition is not associated with/managed by file management
  - Consists of consecutive tracks/blocks to increase disk read/write performance

Page Fault

Page Frame Number			R		D	P	R	W	E	V

- Present bit
  - Indicates whether this virtual page is **in physical memory or on disk**
  - During address translation,
    - if the processor finds that this page is a valid page, but the page's **present bit is zero**, means not in physical memory, the processor generates a **page fault** (a type of exception)
    - That triggers the OS to invoke the **page-fault handler**, to load the missing page from secondary storage into physical memory
  - When the page is not in memory, OS needs to know the disk address of the page in the swap space
    - Usually, the disk address is stored in the PTE; probably share with the bits used for storing the PFN

Page Fault

- OS (page-fault handler) finds the disk address in the PTE and issues an I/O request to fetch the page into memory
- While the I/O is in flight, OS places the current process in **blocked state**, and selects another ready process to run
- When I/O completes, OS
  - **Updates** the PFN field and present bit of the corresponding PTE
  - Then **unblock the process** and **triggers** the process to **retry the instruction** (which triggers this page fault)
    - This causes a TLB miss, and TLB hardware fetches the new translation information from the PTE and updates the TLB cache
    - Then retry the instruction again

Finding Free Page Frames

- OS page-fault handler needs to find a free page frame for placing the incoming page. Where to find it?
- Get it from the Free-list
  - As page is a fixed-size block and same size of a frame, any free frame in anywhere should be okay
- What if memory is full?
  - OS has to first swap out one or more virtual pages to make room for the requested page(s)

Page Replacement Policy

- Strategy used by OS to decides **which virtual page to move out** from main memory to make space for incoming page

- The replacement policy is **critical** to the performance of the application
  - If select a wrong page to kick out, this will result in experiencing more page faults
  - Thus, using not appropriate policy can cause a program to run at disk-like speeds, which is **much much much** slower than the CPU speed

Replacement Policy

- Replacement policy is being used in various system components, and the goal is to **minimize** the number of **misses** or to **maximize** the **hits**
  - In cache management → cache miss
  - In TLB management → TLB miss
  - In virtual memory management → page fault

- One common way to assess the effectiveness of policy is to measure the **hit rate** and calculate the average memory access time (**AMAT**), which is defined as $(Hit_{\%} \cdot T_M) + (Miss_{\%} \cdot T_D)$
  - Where T_M is the cost of accessing memory when hit, and T_D is the cost of accessing memory when missed (including get back the page from disk and access the memory in that page)

Find a “victim” page

- A replacement strategy is characterized by
  - The heuristic it uses to select a page for replacement
  - Execution overhead it incurs

Evaluation

- Evaluate individual policy by running the algorithm on a particular sequence of memory references (called **reference string**) and computing the hit rate of that reference string

- In all our examples, the reference string is
  - 2 3 2 1 5 2 4 5 3 2 5 2

- When assessing individual policy, we assume a process has 3 page frames, and all are initially empty

Optimal Replacement

- The victim page is the page that **will be** accessed furthest **in the future** (as compared to other in memory pages)
  - i.e., will not be referenced for longest period of time in the future

- It always leads to the **fewest misses** overall
  - ***Impossible to have perfect knowledge of future events***
  - Acts as a baseline for comparing how well a policy performs

2	3	2	1	5	2	4	5	3	2	5	2
2	2	2	2	2	2	4	4	4	4	4	4
	3	3	3	3	3	3	3	3	2	2	2
			1	5	5	5	5	5	5	5	5
F	F		F	F		F			F		

Cold-start misses

- There are 6 faults with this reference string; thus, the hit rate is 50%

First-In-First-Out Replacement

- The victim page is the page that has been in the system the longest
  - pages are placed in a queue and the oldest page is selected

- Easy to implement and relatively low overhead
- Unfortunately, FIFO can replace heavily used pages which is the oldest

Oldest page is labeled in red

2	3	2	1	5	2	4	5	3	2	5	2
2	2	2	2	5	5	5	5	3	3	3	3
	3	3	3	3	2	2	2	2	2	5	5
			1	1	1	4	4	4	4	4	2
F	F		F	F	F	F		F		F	F

- There are 9 faults with this reference string – the hit rate is 25%
- Not that practical for real-life systems

Least-Recently-Used Replacement

- Exploits temporal locality by selecting the victim page that has not been referenced for the longest time

- Can provide better performance than FIFO
- Increased overhead
  - Hardware needs to maintain the timings of last reference of all pages
  - When replacing a page, OS scans all time fields to find the least-recently-used page

- LRU may perform poorly if the least-recently used page is the next page to be referenced by a program
  - e.g., a while loop may consist of many virtual pages, when jump back to the top, the **LRU page maybe the one that going to be referenced next**

Least-Frequently-Used Replacement

- The victim page is the page that is the least intensively referenced
  - Based on the heuristic that a page not referenced often is not likely to be referenced in the future
- Each page has a counter, and is updated each time the page is referenced
- Have the possibility of selecting wrong page for replacement
  - A page that **was referenced heavily in the past** may never be referenced again, but will stay in memory while newer, **active pages are replaced**

LRU & LFU

- LRU
  - There are 7 faults with this reference string – the hit rate is 41.7%

Least-recently used pages are labeled in red

2	3	2	1	5	2	4	5	3	2	5	2
2	2	2	2	2	2	2	2	3	3	3	3
	3	3	3	5	5	5	5	5	5	5	5
			1	1	1	4	4	4	2	2	2
F	F		F	F		F		F	F		

- LFU
  - There are 6 faults with this reference string – the hit rate is 50%

Least-frequently used pages are labeled in red. Use FIFO to break the tie.

2	3	2	1	5	2	4	5	3	2	5	2
2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	5	5	5	5	5	5	5	5
			1	1	1	4	4	3	3	3	3
F	F		F	F		F		F			

Approximating LRU

- A way to **approximate LRU** with little (space) overhead is to use a **use (reference) bit** to indicated that a page has recently been referenced

- The use bit of a virtual page is set to 1 by hardware
  - When the page is first loaded in memory upon page fault; or
  - When the page is referenced again

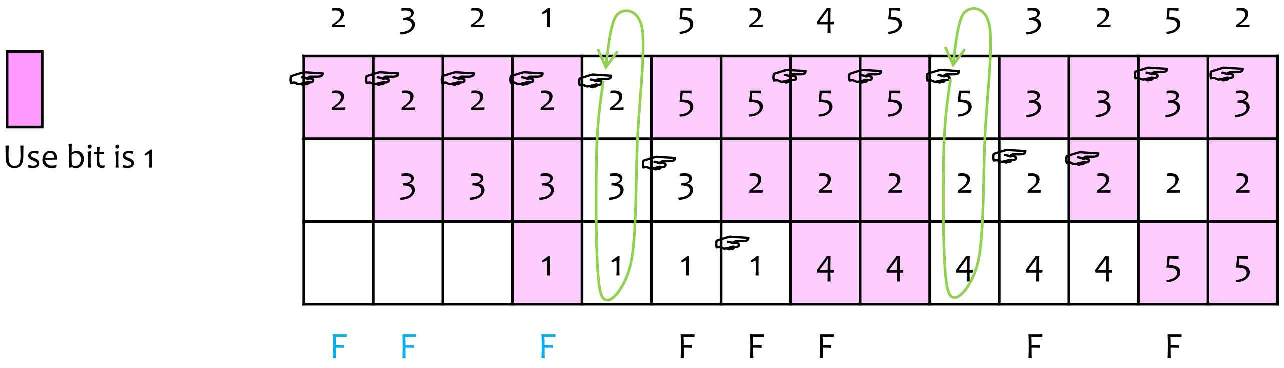
- OS needs some way to clear the use bit as eventually all pages have the use bits set to 1
  - One possible method
    - To screen out pages that are not actively used anymore, the system **periodically resets all the use bits to zero**
    - On the assumption that active pages will be referenced again in the near future

Clock Replacement

- Another scheme which is similar to the approximating LRU
- System has **a pointer** (like a clock hand) points to the virtual page which is the "**oldest**" **at** this moment
- The system treats all the page frames as in a circular list
- When it is time to find a victim page, the system **checks the pointer**
  - if the pointer points to a page **with use bit equals 0**, replace this one
  - Otherwise, **reset the use bit to 0** and advance the pointer to next virtual page
    - Reason: although the page is the "oldest", it has been recently accessed; thus, give it a second chance and treat it as a "new" page
  - The process continues until a page with use bit equals to 0 is found

Clock Replacement

- There are 8 faults with this reference string – the hit rate is 33.3%



Fetch Strategy

- Fetch Strategy
  - When a process's address space is divided into pages and not all pages need to be in main memory. OS needs to consider when to load in a page?

- Demand Paging
  - System **loads** a virtual page only when the running process **explicitly references** the page
  - Pro – Only loads pages that process actually needs; space is not wasted
  - Con – Every time a new page is referenced, a page fault is generated, the process must wait

Fetch Strategy

- Prefetching (Anticipatory Paging)
  - OS attempts to **predict** the virtual pages a process will need and **preloads** these pages **when has free page frames**
  - Must be carefully designed so that overhead incurred by the strategy does not reduce system performance
    - This strategy requires significant resources – page frames and disk I/O. If inaccurately determines which pages a process will need, might result in worse performance than in a demand paging system
- In Linux and other OSs, Demand + Prefetching to exploit spatial locality

Thrashing

- A **serious issue** appears in a multiprogramming or time-sharing system using virtual memory technique
- With **many processes** running, each competes for physical memory to place its **working set**
  - The set of virtual pages that the process is using actively
    - Note: the working set is changing during runtime
- If there are too many processes, **the memory demand exceeds the available physical memory**, what will be the consequence?
  - Processes will be busy swapping pages in and out, and we see that the **page-fault rate** will be **very high**

Thrashing

- Demands for page frames is too great, while one process is fetching a page and is waiting, the pages it already has can be stolen by other processes; when it resumes, it immediately faults again
- This is not just happening to one process; is **experienced by all**
- This leads to **low CPU utilization**
  - the processor is spending a significant amount of time doing nothing – all processes are waiting on page-in requests

- Solution: Suspend or kill some of the processes; don’t allow users to oversubscribe

Summary

- Virtual Memory system gives an illusion to the process that it has large amount of main memory to store the process's address space

- In real life, physical memory is scare resource; OS makes use of slower, larger disks to support the virtualization of memory

- When the CPU tries to access a virtual page that is not in physical memory, OS will be invoked to handle this; it is responsible to load the page from swap space to main memory

- If the system does not have enough free physical memory, OS needs to make the decision in selecting some pages to swap out

Summary

- Page replacement policy is critical to system performance; a wrong decision will induce more page faults
- Realistic replacement policies make use of past accessing history to guide the OS in selecting suitable pages for eviction
- LRU and LFU are performing better than others; however, they are more complicated and have higher implementation overhead
- Thrashing will be appeared if the system is oversubscribed with too many running processes