Memory Management

OPS Lecture 11, G53OPS/G52OSC

Geert De Maere
(Jason Atkin – OSC)
Geert.DeMaere@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2015

Recall
Last Lecture

- Virtual memory relies on **localities** which constitute **groups of pages** that are **used together**, e.g., related to a function (code, data, etc.)
  - Processes move **from locality to locality**
  - If all required pages are **in memory**, **no page faults** will be generated
- **Page tables** become **more complex** (present/absent bits, referenced/modified bits, multi-level, inverted, etc)

Goals for Today

Overview

- Several **key decisions** have to be made when **using virtual memory**
  - When are pages **fetched** $\Rightarrow$ demand or pre-paging
  - **What pages** are **removed** from memory $\Rightarrow$ page replacement algorithms
  - **How many pages** are allocated to a processes and are they **local or global**
  - **When** are pages **removed** from memory $\Rightarrow$ paging daemons
- What **problems** may occur in virtual memory $\Rightarrow$ trashing

- Avoiding **unnecessary pages** and **page replacement** is important!
- Let *ma*, *p*, and *pft* denote the **memory access time** (10-200ns), **page fault rate**, and **page fault time**, the **access time** is then given by:

$$(1-p)\times ma + pft \times p \tag{1}$$

- With an **access time** of 200ns (10^{-9}) and a **page fault time** of 8ms (10^{-3})

$$(1 - p) \times 200 + p \times 8000000 \tag{2}$$

- Access time is **proportional to page fault rate** (keep in mind: all pages would have to be loaded without demand paging)

Demand Paging
On Demand

- **Demand paging** starts the process with **no pages in memory**
  - The first instruction will immediately cause **a page fault**
  - **More page faults** will follow, but they will **stabilise over time** until moving to the **next locality**
- Pages are only **loaded when needed**, i.e. following **page faults**
- The set of pages that is currently being used is called its **working set** ($\Leftrightarrow$ resident set)

Pre-Paging

Predictive

- When the process is started, all pages expected to be used (i.e. the working set) could be **brought into memory at once**
  - This can drastically **reduce the page fault rate**
  - Retrieving multiple (**contiguously stored**) pages **reduces transfer times** (seek time, rotational latency, etc.)
- **Pre-paging** loads pages (as much as possible) **before page faults are generated** ($\Rightarrow$ also used when processes are **swapped out/in**)

Page Replacement

Concepts

- The OS must choose a **page to remove** when a new **one is loaded** (and all are occupied)
- This choice is made by **page replacement algorithms** and **takes into account**
  - When the page is **last used/expected to be used** again
  - Whether the **page has been modifed** (only modified pages need to be written)
- Replacement choices have to be **made intelligently** ($\Leftrightarrow$ random) to **save time**/avoid **trashing**

Page Replacement
Algorithms

1. **Optimal** page replacement
2. **FIFO** page replacement
   - Second chance replacement
   - Clock replacement
3. **Not recently used** (NRU)
4. **Least recently used** (LRU)

Page Replacement
Optimal Page Replacement

- In an **ideal/optimal** world
  - Each page is labeled with the **number of instructions** that will be executed/length of time before it is **used again**
  - The page which is **not referenced** for the **longest time** is the optimal one to remove
- The **optimal approach** is **not possible to implement**
  - It can be used for **post-execution analysis** $\Rightarrow$ what would have been the minimum number of page faults
  - It provides a **lowerbound** on the **number of page faults** (used for comparison with other algorithms)

Page Replacement

First-In, First-Out (FIFO)

- FIFO maintains a **linked list** and **new pages** are added at the end of the list
- The **oldest page** at the **head of the list** is evicted when a page fault occurs
- The **(dis-)advantages** of FIFO include:
  - It is **easy** to understand/implement
  - It **performs poorly** $\Rightarrow$ heavily used pages are just as likely to be evicted as a lightly used pages

Page Replacement
FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames**
- Consider the following page **references in order**:
  `0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4`
- The number of **page faults** that are generated is **13**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	5	5	5	3	3	3	3	3	3	3	3	3	3	3	3	4
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	6	6	6	1	1	1	1	1	1	1	1
PF4	-	-	-	3	3	3	3	3	7	7	7	7	7	7	7	7	7	7	7	7	2	2	2	2

Figure: FIFO Page Replacement

Page Replacement

Second Chance FIFO

- Second chance is a **modification of FIFO**:
  - If a page at the front of the list has **not been referenced** it is **evicted**
  - If the reference bit is set, the page is **placed at the end** of list and its **reference bit reset**
- The **(dis-)advantages** of second chance FIFO include:
  - The algorithm is **relatively simple**, but it is **costly to implement** because the list is constantly changing
  - The algorithm **can degrade to FIFO** if all pages were initially referenced

Page Replacement

The Clock Replacement Algorithm

- The **second-chance implementation** can be improved by **maintaining the page list as a circle** (this is the only difference)
  - A **pointer** points to the oldest page
  - In this form the algorithm is called (one-handed) clock
  - It is faster, but can still be **slow if the list is long**
- The **time spent** on maintaining the list **is reduced**

Page Replacement
The Clock Replacement Algorithm

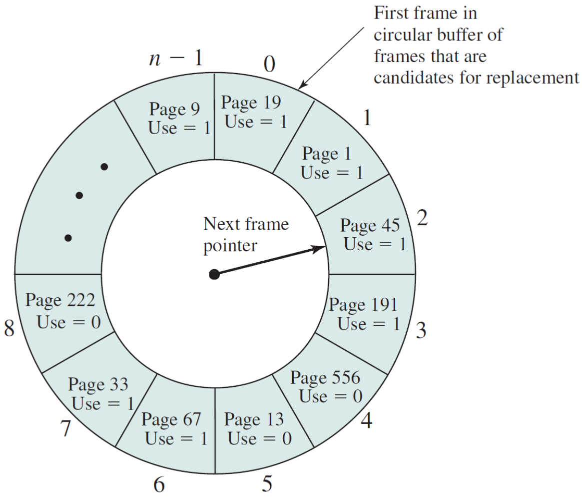

Figure: Clock Replacement Algorithm (Stallings)

Page Replacement
Not Recently Used (NRU)

- **Referenced/modified** bits are kept in the page table
  - Referenced bits are clear at start the start, and **nulled at regular intervals** (e.g. system clock interrupt)
- Four different **page "types"** exist
  - class 0: not referenced, not modified
  - class 1: not referenced, modified
  - class 2: referenced, not modified
  - class 3: referenced, modified

Page Replacement

Not Recently Used (NRU, Cont'ed)

- **Page table entries** are inspected upon every **page fault** $\Rightarrow$ a page from the **lowest numbered non-empty class** is removed (can be implemented as a clock)
- The NRU algorithm provides a **reasonable performance** and is easy to understand and implement
- The performance of NRU can be improved by working with two buffers, containing a **modified** and **free list**

Page Replacement
Least-Recently-Used

- Least recently used **evicts the page** that has **not been used the longest**
  - The OS must **keep track** of when a page was **last used**
  - Every **page table entry** contains a **field for the counter**
  - This is **not cheap** to implement as we need to maintain a **list of pages** which are **sorted** in the order in which they have been used (or search for the page)
- The algorithm can be **implemented in hardware** using a **counter** that is incremented after each instruction

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
  `0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4`
- The number of **page faults** that are generated is **12**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	4
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	4	4	1	1	1	1	1	1	1	1
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	5	5	5	5	5	5	5	2	2	2	2
PF4	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Figure: Least Recently Used

Resident Set

Size of the Resident Set

- How many pages should be allocated to individual processes:
    - **Small resident sets** enable to store **more processes** in memory $\Rightarrow$ improved CPU utilisation
    - **Small resident sets** may result in **more page faults**
    - **Large resident sets** may **no longer reduce** the **page fault rate** (**diminishing returns**)
- A trade-off exists between the **sizes of the resident sets** and **system utilisation**

Resident Set

Size of the Resident Set

- Resident set sizes may be **fixed** or **variable** (i.e. adjusted at runtime)
- For **variable sized** resident sets, **replacement policies** can be:
  - **Local**: a page of the same process is replaced
  - **Global**: a page can be taken away from a different process
- Variable sized sets require **careful evaluation of their size** when a **local scope** is used (often based on the **working set** or the **page fault frequency**)

Working Sets

Defining and Monitoring Working Sets

- The **resident set** comprises the set of pages of the process that are in memory
- The **working set** $W(t, \Delta)$ comprises the set referenced pages in the last Δ (= working set window) virtual time units for the process
- The **working set size** can be used as a guide for the number frames that should be allocated to a process
- Δ can be defined as "**memory references**" or as "**actual process time**"
    - The the set of most recently used pages
    - The set of pages used within a pre-specified time interval

Working Sets

Defining and Monitoring Working Sets

- The working set is a **function of time** t:
  - Processes **move between localities**, hence, the pages that are included in the working set **change over time**
  - **Stable** intervals alternate with intervals of **rapid change**
- The working set size is a **non-decreasing function of** Δ, asymptotically increases, and stabilises at N, with N denoting the number of pages for the process

Working Sets

Monitoring Working Sets

- Choosing the right value for Δ is paramount:
  - Too **small**: inaccurate, pages are missing
  - Too **large**: too many unused pages present
  - **Infinity**: all pages of the process are in the working set
- Working sets can be used to guide the **size of the resident sets**
  - Monitor the working set
  - Remove pages from the resident set that are not in the working set

Resident Sets

Local vs. Global Replacement

- **Global replacement policies** can select frames from the entire set, i.e., they can be “taken” from other processes
  - Frames are **allocated dynamically** to processes
  - Processes cannot control their own page fault frequency, i.e., the **PFF** of one process is **influenced by other processes**
- **Local replacement policies** can only select frames that are allocated to the current process
  - Every process has a **fixed fraction of memory**
  - The locally **“oldest page”** is not necessarily the globally “oldest page”

Paging Daemon

Pre-cleaning ($\Leftrightarrow$ demand-cleaning)

- It is more efficient to **proactively** keep a number of **free pages** for **future page faults**
  - If not, we may have to **find a page** to evict and we **write it to the drive** (if modified) first when a page fault occurs
- Many systems have a background process called a **paging daemon**
  - This process but **runs at periodic intervals**
  - It inspect the state of the frames and, if too few pages are free, it **selects pages to evict** (using page replacement algorithms)

Paging Daemon

Pre-cleaning ($\Leftrightarrow$ demand-cleaning)

- It is more efficient to **proactively** keep a number of **free pages** for **future page faults**
  - If not, we may have to **find a page** to evict and we **write it to the drive** (if modified) first when a page fault occurs
- Many systems have a background process called a **paging daemon**
  - This process but **runs at periodic intervals**
  - It inspect the state of the frames and, if too few pages are free, it **selects pages to evict** (using page replacement algorithms)
- Paging daemons can be combined with **buffering** (free and modified lists) $\Rightarrow$ write the modified pages but keep them in main memory when possible

Trashing

Defining Trashing

- Assume **all available pages are in active use** and a new page needs to be loaded:
  - The page that will be **evicted** will have to be **reloaded soon afterwards**, i.e., it is still active
- **Trashing** occurs when pieces are swapped out and loaded again immediately

Trashing
A Vicious Circle?

- CPU utilisation is too low $\Rightarrow$ scheduler **increases degree of multi-programming**
  - $\Rightarrow$ Pages are allocated to new processes and **taken away from existing processes**
    - $\Rightarrow$ I/O **requests are queued** up as a consequence of page faults
- CPU **utilisation drops further** $\Rightarrow$ scheduler increases degree of multi-programming

Trashing
Causes/Solutions

- **Causes** of trashing include:
  - The degree of multi-programming is too high, i.e., the total **demand** (i.e., the sum of all **working set** sizes) **exceeds supply** (i.e. the available frames)
  - An individual process is allocated **too few pages**
- This can be **prevented** by, e.g., using good **page replacement policies** and reducing the **degree of multi-programming** (medium term scheduler)

Summary

Take-Home Message

- Pre-paging/demand paging
- Optimal, FIFO, Second Chance FIFO, NRU, LRU page replacement
- Page allocations to processes (variable, fixed, local, global)
- Page Daemons
- Trashing