Memory Management

Stack Algorithms

- One for which the set of pages in memory for n frames $M(n)$ is a subset of the set of pages in memory for n+1 frames $M(n+1)$
  - That is, $M(n) \subseteq M(n+1)$
  - Examples: OPT, LRU

Others Like LRU

- Least Frequently Used (LFU)
  - Keep a count of the number of times each page is referenced, and replace the page with the smallest number
- Most Frequently Used (MFU)
  - Keep a count of the number of times each page is referenced, and replace the page with the largest number
  - Theory is that the page with the smallest count has been brought in most recently, so is waiting to be used

Clock

- Set use bit on each page reference; to choose victim, begin with the page that the pointer points to; if use bit is set, clear it and advance pointer, otherwise the page is replaced
- Example: reference string is 1 2 3 4 5 4 2 3 4 5 1 2 3 4 5
- 3 frames available: total of 14 page faults

	1	2	3	4w	5	4	2	3w	4	5	1	2w	3	4	5
frame 1	1/1	1/1	1/1	4/1	4/1	4/1	4/1	3/1	3/1	3/1	1/1	1/1	1/1	4/1	4/1
frame 2		2/2	2/1	2/0	5/1	5/1	5/1	5/0	4/1	4/1	4/0	2/1	2/1	2/0	5/1
frame 3			3/1	3/0	3/0	3/0	2/1	2/0	2/0	5/1	5/0	5/0	3/1	3/0	3/0
pf	•	•	•	•	•		•	•	•	•	•	•	•	•	•

How It Works

- Bring in 4w: current contents of memory changes as follows:
  - Initially, →1/1, 2/1, 3/1
  - Advance pointer: 1/0, → 2/1, 3/1
  - Advance pointer: 1/0, 2/0, → 3/1
  - Advance pointer: → 1/0, 2/0, 3/0; replace 1/0 with 4/1

Clock

- Example: reference string is 1 2 3 4 5 4 2 3 4 5 1 2 3 4 5
- 4 frames available: total of 10 page faults
- Notation: *n*/*u* means page number *n* has use bit set to *u*

	1	2	3	4w	5	4	2	3w	4	5	1	2w	3	4	5
frame 1	1/1	1/1	1/1	1/1	5/1	5/1	5/1	5/1	5/1	5/1	5/0	5/0	5/0	4/1	4/0
frame 2		2/2	2/1	2/1	2/0	2/0	2/1	2/1	2/1	2/1	1/1	1/1	1/1	1/1	5/1
frame 3			3/1	3/1	3/0	3/0	3/0	3/1	3/1	3/1	3/0	2/1	2/1	2/1	2/0
frame 4				4/1	4/0	4/1	4/1	4/1	4/1	4/1	4/0	4/0	3/1	3/1	3/0
pf	•	•	•	•	•						•	•	•	•	•

How It Works

- Bring in 1 after second 5: current contents of memory changes as follows:
  - Initially, 5/1, → 2/1, 3/1, 4/1
  - Advance pointer: 5/1, 2/0, → 3/1, 4/1
  - Advance pointer: 5/1, 2/0, 3/0, → 4/1
  - Advance pointer: → 5/1, 2/0, 3/0, 4/0
  - Advance pointer: 5/0, → 2/0, 3/0, 4/0; replace 2/0 with 1/1

Clock Variant

- Second Chance: do not circle back to the beginning; instead pick the last one that has the use bit cleared, and start at the beginning each time

Not Recently Used (NRU, NUR)

- Like LRU, but based on use bits rather than when page is brought into memory
- Consider use, dirty bits as a pair (use, dirty)
- This gives four classes
  - Class 0: (0, 0)
  - Class 1: (0, 1)
  - Class 2: (1, 0)
  - Class 3: (1, 1)
- Pick victim randomly from the lowest numbered class
- When a page is brought in, *all* use bits are cleared

Not Recently Used (NRU, NUR)

- Example: reference string is 1 2 3 4* 5 4 2 3* 4 5 1 2* 3 4 5
- 3 frames available: total of 11 page faults
- Notation: *n*w means page number *n* is written to; *n*/*ab* means page number *n*, use bit is *a*, dirty bit is *b*

	1	2	3	4w	5	4	2	3w	4	5	1	2w	3	4	5
frame 1	1/10	1/10	1/10	1/00	5/10	5/10	2/10	2/10	2/10	5/10	1/10	1/10	1/10	4/10	4/00
frame 2		2/10	2/10	4/11	4/01	4/11	4/01	4/01	4/11	4/11	4/01	2/11	2/11	2/01	5/10
frame 3			3/10	3/00	3/00	3/00	3/00	3/11	3/11	3/11	3/01	3/01	3/11	3/01	3/01
pf	•	•	•	•	•		•			•	•	•		•	•

How It Works

- Bring in 4w: current contents of memory changes as follows:
  - Initially, 1/10, 2/10, 3/10
  - Classes: 1/10 (2), 2/10 (2), 3/10 (2)
  - Pick randomly from class 2, as all are class 2; we choose 2/10
  - Clear use bits of pages in memory: 1/00, ***, 3/10
  - Insert new page: 1/00, 4/11, 3/10

How It Works

- Bring in 1 after second 5: current contents of memory changes as follows:
  - Initially, 5/10, 4/11, 3/11
  - Classes: 5/10 (2), 4/11 (3), 3/11 (3)
  - Pick randomly from class 2; there is only one choice
  - Clear use bits of pages in memory: ***, 4/01, 3/01
  - Insert new page: 1/10, 4/01, 3/01

Second-Chance Cyclic

- Like NUR, but not random; advance a pointer as in clock algorithm
- As before, four classes
  - Class 0: (0, 0); after: select this page
  - Class 1: (0, 1); after: (0, 0)* [indicating this page needs to be copied out]
  - Class 2: (1, 0); after: (0, 0)
  - Class 3: (1, 1); after: (0, 1)
- Loop through memory until a page can be removed

Second-Chance Cyclic

- Example: reference string is 1 2 3 4* 5 4 2 3* 4 5 1 2* 3 4 5
- 3 frames available: total of 11 page faults
- Notation: *n*w means page number *n* is written to; *n*/*ab* means page number *n*, use bit is *a*, dirty bit is *b; n*/00* means page number *n*, both bits cleared but if page replaced, write out the page first

	1	2	3	4w	5	4	2	3w	4	5	1	2w	3	4	5
frame 1	1/10	1/10	1/10	4/11	4/11	4/11	4/11	4/00*	4/11	4/11	4/00*	2/11	2/11	2/00*	5/10
frame 2		2/10	2/10	2/00	5/10	5/10	5/10	3/11	3/11	3/11	3/00*	3/00*	3/11	3/00*	3/00*
frame 3			3/10	3/00	3/00	3/00	2/10	2/00	2/00	5/10	1/10	1/10	1/10	4/10	4/10
pf	•	•	•	•	•		•			•	•	•		•	•

How It Works

- Bring in 4w: current contents of memory changes as follows:
  - Initially, →1/10, 2/10, 3/10
  - Advance pointer: 1/00, → 2/10, 3/10
  - Advance pointer: 1/00, 2/00, → 3/10
  - Advance pointer: → 1/00, 2/00, 3/00; replace 1/00 with 4/11

How It Works

- Bring in 1 after second 5: current contents of memory changes as follows:
  - Initially, →4/11, 3/11, 5/10
  - Advance pointer: 4/01, → 3/11, 5/10
  - Advance pointer: 4/01, 3/01, → 5/10
  - Advance pointer: → 4/01, 3/01, 5/00
  - Advance pointer: 4/00*, → 3/01, 5/00
  - Advance pointer: 4/00*, → 3/01, 5/00
  - Advance pointer: 4/00*, 3/00*, → 5/00; replace 5/00 with 1/10

Ad Hoc Techniques

- Goal is to improve performance
- System keeps a pool of free frames
- When a process needs a page:
  - Read page into free frame
  - Write out the victim if necessary
  - Add its frame to the free frame pool

Ad Hoc Techniques

- Bringing in pages need not wait for a dirty page to be written out
- Do I/O periodically rather than on each page replacement
    - Example: number of free frames falls below some threshold
    - Advantage: if a page is needed but has not yet been written out, just remove the frame holding the page from the free frame pool and use it; no I/O required
    - If paging device is idle, find pages with dirty bit set, write them out, clear dirty bit

Frame Allocation Algorithms

- Many strategies
  - Use all frames before replacing pages
  - Keep some frames free so that when a page fault occurs, you can bring in a page while choosing a victim
- But: how does the system allocate frames to a process?
  - Problem arises when using demand paging with multiprogramming
- The most page frames a process can get is all of them!
- The least depends on the architecture of the system
  - Page fault causes instruction to restart, so this bounds maximum number of pages a single instruction can reference

Examples:

- PDP-8: 1 memory address per instruction, so minimum number of frames required is 3 frames:
  - 1 frame for the instruction
  - 1 frame for the address, which may be a pointer, so . . .
  - 1 frame for the address that the pointer points to
- PDP-11: an instruction may be more than 1 word long, so minimum number of frames required is 6 frames:
  - 2 frames for instruction, which may reference 2 addresses, for each of which:
  - 1 frame per address, which may be a pointer, so . . .
  - 1 frame for the address the pointer points to

Examples:

- Data General Nova 3: allows multiple levels of indirection
- Each 16 bit word had 15 bits for the address and 1 indirect bit
    - In theory, indirection could go on forever!
- Engineers modified architecture to allow at most 16 levels of indirection
- Minimum number of frames required was 18 frames per instruction:
    - 1 for the instruction
    - Up to 17 for the address

Global Allocation

- Frames for replacement pages are pooled, and when frame needed it is taken from this pool
- But there are problems . . .
  - Program no longer controls its own paging behavior
  - External factors may affect program performance

Local Allocation

- Number of frames allocated to a process is fixed
- Frames for replacement pages come from there
- With equal allocation, if there are m frames and n processes, each process gets m/n frames
- With proportional allocation, each process is assigned a virtual memory size s_i; let S be their sum; process p_i gets $s_i m/S$ frames

Examples

- Example: system has 2 processes
  - One with virtual memory size of 10K
  - The other with virtual memory size of 127K
  - 62 free frames
- Equal allocation: each process gets 62/2 = 31 frames
- Proportional allocation:
  - Process p1 gets $10 \times 62/(10+127) \approx 4.52$ or 5 frames; process p2 gets $127 \times 62/(10+127) \approx 57.47$ or 57 frames

Consequences

- If number of processes goes up, each process loses frames; if number of processes goes down, each process gets more frames
- Problem: all processes are treated equally, *regardless* of their priority
- Solutions:
  - Use a proportional allocation scheme that factors in priorities
  - Allow high priority process to take frames from low priority process

Thrashing

- Process spends more time paging than executing
- Most commonly occurs when set of pages needed to avoid page faulting for every reference will not fit into set of frames allocated to process
- Throughput plunges
- Processes paging increases, but processes do no work
- Effective memory access time increases
- If frame allocation is local, tis limits the effect to one process, but the increased contention for paging device increases effective memory access time for all processes

Example

- Operating system monitors CPU utilization
- When too few processes executing, operating system brings in new process
- Assume global page replacement algorithm:
    1. Process needs more frames, acquires them from other processes
    2. Those processes begin page faulting, and queueing for paging device
    3. Ready queue empties
    4. CVPU utilization drops
    5. Operating system brings more processes in
    6. Those processes acquire frames from executing processes

    Go back to 2

Principle of locality

- Principle: *As a program runs, it moves from locality to locality*
- A *locality* is a set of instructions, data that is grouped close to one another
- Principle says that references tend to be to addresses grouped closely together