Memory Management

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, this 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

Working Set Model

- At time t, let

  $$W(t, \tau) = \{ \text{ set of pages referenced in last } \tau \text{ time units } \}$$

- Working set principle ties process management to memory management:
- Principle: *A process may execute only if its working set is resident in main memory. A page may not be removed from main memory if it is in the working set of an executing process.*

Properties of Working Set

- Size of a working set can var:

  $$1 < |W(t, \tau)| < \min(\tau, \text{number of pages in process})$$

- $W(t, \tau) \subseteq W(t+1, \tau)$ so this is a stack algorithm
- Working set of a process undergoes periods of fairly consistent size alternating with periods of larger size
  - Larger size (stable range) is when process is in locality
  - Smaller size (transition range) is when process is transitioning from one locality to the next

Properties of Working Set

- Larger periods typically account for 98% of process time
- Remaining 2% has at least half of all page faults
  - During transition range, page fault rates are 100–1000 times more than in stable range
- Ideally, τ large enough so working set contains all pages being frequently accessed, and small enough so it contains *only* those pages
  - Typical value of τ is 0.5 seconds

Example

- Here, let τ = 4, so the working set is the set of pages referenced within the last 4 time units
- Assume pages 5, 4, and 2 are in memory at time 1, as below

time	–2	–1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ref	5	4	2	1	2	3	4w	5	4	2	3w	4	5	1	2w	3	4	5
page 1	—	—	—	1	1	1	1	1	—	—	—	—	—	1	1	1	1	1
page 2	—	—	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
page 3	—	—	—	—	—	3	3	3	3	3	3	3	3	3	3	3	3	3
page 4	—	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
page 5	5	5	5	5	5	—	—	5	5	5	5	5	5	5	5	5	5	5
pf	•	•	•	•		•		•		•								
pages in						3		5										
pages out						5			1									

Implementation Issues

- Requires a virtual clock
- Whenever a page is accessed, the current time according to the virtual clock is recorded in page table
- The working set contains all pages accessed within τ of the present time
- *Problem*: too expensive

Approximations

- All approximate membership of working set by examining which pages have been referenced since last page fault or last few page faults
- How they do this examination differs

WSCLOCK

- Use a clock-type scan through the frame table whenever there is a page fault
- If use bit set:
  - Clear it
  - Store virtual time of process owning the page in that frame in referenced time field; that’s an approximation of when page last referenced
- If use bit clear:
  - Compare current virtual time of process owning the page in that frame to the time in the referenced time field
  - If difference is greater than t, page is not in process’ current working set and can be removed
- If no page can be removed, swap out a process

Working Set Size (WSS)

- Memory manager maintains estimates of sizes of working sets
- When a process is swapped in, working set size is estimated by counting the number of pages recently accessed
  - For example, by looking at the use bits in process’ page table
- When that many page frames become available, process is put onto ready list

Page Fault Frequency (PFF)

- Bases membership in working set on page fault frequency
- Ide is to compute working set at each page fault
- Define parameter *p*
- At each page fault, compare time since previous page fault to *p*
  - If this time is less than *p*, add page to the working set
  - If this time is greater than *p*, remove from the working set all pages not referenced since previous page fault
- *Implementation*: on each page fault, clear all use bits

Other Paging Considerations

- Prepaging
- I/O interlock
- Page size
- Program structure

Prepaging

- When (re)started a process, try to bring into memory at one time all pages needed
  - Idea is to reduce initial faulting
- Example: for working set, keep a list of pages in current working set with each swapped-out process
- Cost-benefit tradeoff: some prepaged pages won’t be needed
  - Is the cost of bringing them in more than servicing the interrupts caused by page faulting were they not brought in?

I/O Interlock

- When doing DMA from or to a buffer in user space, the page may need to be locked into memory to enable the transfer to complete
  - This page *cannot* be paged out!
- Solution 1: Do all I/O to system memory and then copy to the user buffer when it is in memory
- Solution 2: Associate a lock bit with each page; when set, page cannot be removed from memory

Lock Bit

- Can also be used to prevent replacement of pages belonging to a process that was just swapped in but not yet executing
- Example: a process is brought in and put on the ready queue
  - Higher priority process is running and page faults
  - Higher priority process might take frame from newly-arrived, lower priority process as those pages have not yet been used
  - If those pages have their lock bits set, they will not be selected

Page Size

- When designing new machine, considerations for choosing page size:
- Size of page table is inversely proportional to size of page
  - Example: virtual memory uses 2^{32} words; system can have 2^{22} pages of 2^{10} words or 2^{20} pages of 2^{12} words
  - Means large page sizes are better as each process needs a copy of its page table
- Memory utilization better with smaller page sizes as less internal fragmentation
- Time to read/write a larger page less than that needed to write 2 smaller pages which when combined have the same size as the larger page

Page Size

- Large page size reduces rate of page faults, so there is less time servicing interrupts, doing I/O related to paging, etc.
- Some systems allow more than one page size
  - GE 645 allowed pages of either 64 words or 1024 words
  - IBM 370 allowed pages of either 2048 or 4096 words

Program Structure

- Change it to (not order of array indices):

```
for (j = 0; j < 1024; j++)
    for (i = 0; i < 1024; i++)
        array[j][i] = 0;
```

- Accesses are array[0][0], array[0][0], array[0][2] …
- Now in the worst case, only 1024 page faults

Program Structure

- Say page size is 1024 words:

```
for (j = 0; j < 1024; j++)
  for (i = 0; i < 1024; i++)
    array[i][j] = 0;
```

- Accesses are array[0][0], array[1][0], array[2][0] …
- C stores arrays in row major order, so each row (array[0][0], array[0][1], array[0][2], …) is all on one page
- In the worst case, the above causes 1024^2 = 1,048,576 page faults

Program Structure

- Change it to (note order of array indices):

```
for (j = 0; j < 1024; j++)
    for (i = 0; i < 1024; i++)
        array[j][i] = 0;
```

- This time, accesses are array[0][0], array[0][1], array[0][2] …
- C stores arrays in row major order, so each row (array[0][0], array[0][1], array[0][2], …) is all on one page
- In the worst case, the above causes 1024 page faults
    - *Much* fewer than before!

Program Structure

- Data structures: some have good locality, others do not
  - Stacks have good locality
  - Hash tables do not
- Arrangement of routines: put routines that call each other on the same page to reduce page faulting

Devices, Input, and Output

Kernel Level I/O Routines

- Device drivers: move data to, from secondary storage
- Each type of device has its own device driver
  - All processes access drivers via system calls
- How do processes view devices
  - Transparency: manufacturer, model, and in some cases type of device do not affect how processes access it
  - Example: virtual devices, which are devices simulated by kernel, with data kept either in memory (but with interface of disk) or on secondary storage
  - Example: printer spooler, which to the program is simply a printer but it is really writing data to disk, which will later be sent to printer

Issues with Device I/O

- Goals: what a good process/device interface should do
- Device hardware: what device looks like
- Device interface: how device are connected to computer
- Device drivers: what kernel modules that interact with devices look like
- Process interface: how processes access devices

Goals of Kernel I/O routines

- Character code independence
- Device independence
- Efficiency
- Uniform treatment of devices

Character Code Independence

- Kernel I/O subsystem must translate character codes from various devices to uniform internal representation
  - Example: end-of-line can be <NL> (\n), <CR> (\r), <CR><NL> (\r\n), . . .
- Kernel does this right after characters arrive in memory but before they are given to process, or before they are written to the device
  - So programmer need not worry about this
- Internal codes vary; examples:
  - ASCII
  - UNICODE-16, UNICODE-32 (supersets of ASCII)
  - EBCDIC