Recap …

- Virtual memory
  - Keep only a few pages in memory, rest on disk
  - On-demand paging: retrieve a page when needed
  - Page fault: A referenced page is not loaded in memory
  - Page replacement algorithm
    - Principle of locality of reference
  - FIFO, OPT, LRU
  - Stack algorithms
  - LRU approximations
    - Based on dirty bit and reference bit
    - Additional Reference-Bits Algorithm
    - Second-Chance (Clock) Algorithm
    - Enhanced Clock Algorithm with Dirty/Modify Bit

CSCI 3753
Operating Systems

Memory Management

Working Set and Memory-Mapped Files

Chapters 8 and 9

Lecture Notes By

Shivakant Mishra

Computer Science, CU-Boulder

Last Update: 11/02/17

Allocation of Memory Frames

- How many frames does each process get allocated? How many frames does the OS get allocated versus user processes?
- Variety of policies:
  - based on number of frames
  - based on whether frames are allocated locally or globally

Allocation Policies

1. Equal allocation
   - split m frames equally among n process
   - m/n frames per process
   - problem: doesn’t account for size of processes, e.g. a large database process versus a small client process whose size is << m/n
   - needs to be adaptive as new processes enter and the value of n fluctuates

2. Proportional allocation

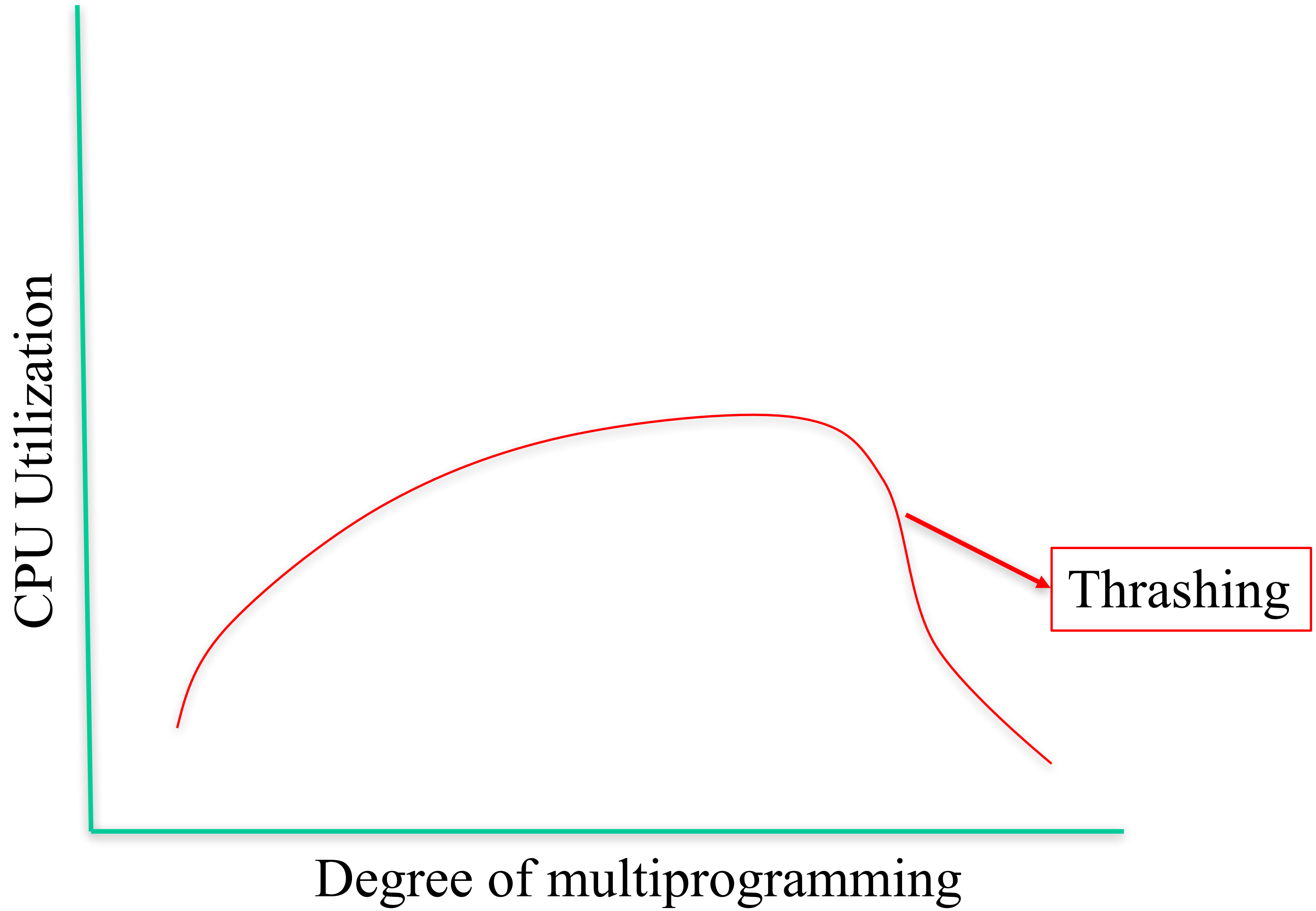
- allocate the number of frames relative to the size of each process
- Let S_i = size of process P_i
- $S = \Sigma S_i$
- Allocate $a_i = (S_i /S) * m$ frames to process P_i
- proportion a_i can vary as new processes start and existing processes finish
- Also, if size is based on the code size, or address space size, then that is not necessarily the number of pages that will be used by a process

3. Minimum # frames:
   - determine the minimum # of frames to allocate to each process to run. Ideally, this is just one page, i.e. the page in which the program counter is currently executing.
   - In practice, some CPUs support *complex* instructions.
     - multi-address instructions. Each address could belong to a different page.
     - Also, there can be multiple levels of indirection in the addressing, i.e. pointers. Each such level of indirection could result in a different page being accessed in order to execute the current instruction. Up to N levels of indirection may be supported, which means may need up to N pages as the minimum.

Working Set Model

- Multiprogramming Environment
  - All processes share the limited number of page frames available
  - Goal: To improve CPU utilization as much as possible
  - Multiprogramming vs CPU utilization (two factors)
    - Increase in the degree of multiprogramming → Increase in CPU utilization
    - Increase in the degree of multiprogramming → Less number of page frames per process → Increase in the number of page faults → Decrease in CPU utilization

Multiprogramming vs CPU Utilization



Thrashing

- Most of the CPU time is spent in swapping pages between disk and main memory.
- How can we maximize CPU utilization, but avoid thrashing?
- Problems:
  - Thrashing depends on the types of processes currently running
  - Even a single process' behavior can vary over time according to the phase of process
- Contemporary models are based on *working set*

Working Set

- How much memory (number of page frames) should be allocated to a process?
- Working set of a process is the set of pages that the process is currently using
  - Determined by principle of locality of reference.
- To reduce the # of page faults, preload the working set of a process before a process runs.
  - Prepaging as opposed to demand paging.

Working Set

- *w*(t, t-T): set of pages accessed in past T time units.
- T is adjusted dynamically to keep the # of page faults between a low and a high thresholds
  - # of page faults exceed the high threshold: increase T → decrease the degree of multiprogramming.
  - # of page faults fall below the low threshold: decrease T → increase the degree of multiprogramming.

Working Set Principle

A process runs only if its working set is in memory.

Thrashing: Alternative

- Instead of using a working set model, just directly measure the page fault frequency (PFF)
  - When PFF > upper threshold, then increase the # frames allocated to this process
  - When PFF < lower threshold, then decrease the # frames allocated to this process (it doesn't need so many frames)
  - Windows NT used a version of this approach

Memory-Mapped Files

- Normally, each read/write from/to a file requires a system call plus file manager involvement plus reading/writing from/to disk
- Programmers can improve performance by copying part of or entire file into a local buffer, manipulating it, then writing it back to disk
  - This requires manual action on the part of the programmer
- Instead, it would be faster and simpler if the file could be loaded into memory (almost) transparently so that reads/writes take place from/to RAM
- Use the virtual memory mechanism to map (parts of) files on disk to pages in the logical address space

Steps for memory-mapping a file

1. Obtain a handle to a file by creating or opening it
2. Reserve virtual addresses for the file in your logical address space
3. Declare a (portion of a) file to be memory mapped by establishing a mapping between the file and your virtual address space using an OS function like mmap()
   – void * mmap(void **start*, size_t *length*, int *prot*, int *flags*, int *fd*, off_t *offset*)
     • map *length* bytes beginning at *offset* into file *fd*, preferably at address *start* (hint only), *prot* = R/W/X/no access, *flags* = map_fixed, map_shared, map_private
     • returns pointer to mmap'ed area
4. When file is first accessed, it's demand paged into physical memory
5. Subsequent read/write accesses to (that portion of) the file are served by physical memory

Advantages of Memory-Mapped Files

- After the first access, all subsequent reads/writes from/to a file (in memory) are fast
- Multiple processes can map the same file concurrently and share efficiently
  - In Windows, this mapping mechanism is also used to create shared memory between processes and is the preferred memory for sharing information among address spaces
  - on Linux, have separate mmap() and shared memory calls, e.g. shmget() and shmat()

Memory-Mapped I/O (vs. Files)

- Similar behavior to memory-mapped files
- Memory-mapped I/O maps device registers (instead of file pages) to memory locations
  - reads/writes from/to these memory addresses are easy and are automatically caught by the MMU (just as for mem-mapped files), causing the data to be automatically sent from/to the I/O devices
  - e.g. writing to a display's memory-mapped frame buffer, or reading from a memory-mapped serial port