# CIS520 – Operating Systems
# Handout 9
# Introduction to Paging

- Basic idea: allocate physical memory to processes in fixed size chunks called page frames. Present abstraction to application of a single linear address space. Inside machine, break address space of application up into fixed size chunks called pages. Pages and page frames are same size. Store pages in page frames. When process generates an address, dynamically translate to the physical page frame which holds data for that page.
- So, a virtual address now consists of two pieces: a page number and an offset within that page. Page sizes are typically powers of 2; this simplifies extraction of page numbers and offsets. To access a piece of data at a given address, system automatically does the following:
  - Extracts page number.
  - Extracts offset.
  - Translate page number to physical page frame id.
  - Accesses data at offset in physical page frame.
- How does system perform translation? Simplest solution: use a page table. Page table is a linear array indexed by virtual page number that gives the physical page frame that contains that page. What is lookup process?
  - Extract page number.
  - Extract offset.
  - Check that page number is within address space of process.
  - Look up page number in page table.
  - Add offset to resulting physical page number
  - Access memory location.

  Problem: for each memory access that processor generates, must now generate two physical memory accesses.
- Speed up the lookup problem with a cache. Store most recent page lookup values in TLB. TLB design options: fully associative, direct mapped, set associative, etc. Can make direct mapped larger for a given amount of circuit space.
- How does lookup work now?
  - Extract page number.
  - Extract offset.
  - Look up page number in TLB.
  - If there, add offset to physical page number and access memory location.
  - Otherwise, trap to OS. OS performs check, looks up physical page number, and loads translation into TLB. Restarts the instruction.

- Like any cache, TLB can work well, or it can work poorly. What is a good and bad case for a direct mapped TLB? What about fully associative TLBs, or set associative TLB?
- Fixed size allocation of physical memory in page frames dramatically simplifies allocation algorithm. OS can just keep track of free and used pages and allocate free pages when a process needs memory. There is no fragmentation of physical memory into smaller and smaller allocatable chunks.
- But, are still pieces of memory that are unused. What happens if a program's address space does not end on a page boundary? Rest of page goes unused. Book calls this internal fragmentation.
- How do processes share memory? The OS makes their page tables point to the same physical page frames. Useful for fast interprocess communication mechanisms. This is very nice because it allows transparent sharing at speed.
- What about protection? There are a variety of protections:
  - Preventing one process from reading or writing another process' memory.
  - Preventing one process from reading another process' memory.
  - Preventing a process from reading or writing some of its own memory.
  - Preventing a process from reading some of its own memory.

  How is this protection integrated into the above scheme?
- Preventing a process from reading or writing memory: OS refuses to establish a mapping from virtual address space to physical page frame containing the protected memory. When program attempts to access this memory, OS will typically generate a fault. If user process catches the fault, can take action to fix things up.
- Preventing a process from writing memory, but allowing a process to read memory. OS sets a write protect bit in the TLB entry. If process attempts to write the memory, OS generates a fault. But, reads go through just fine.
- Virtual Memory Introduction.
- When a segmented system needed more memory, it swapped segments out to disk and then swapped them back in again when necessary. Page based systems can do something similar on a page basis.
- Basic idea: when OS needs to a physical page frame to store a page, and there are none free, it can select one page and store it out to disk. It can then use the newly free page frame for the new page. Some pragmatic considerations:
  - In practice, it makes sense to keep a few free page frames. When number of free pages drops below this threshold, choose a page and store it out. This way, can overlap I/O required to store out a page with computation that uses the newly allocated page frame.
  - In practice the page frame size usually equals the disk block size. Why?
  - Do you need to allocate disk space for a virtual page before you swap it out? (Not if always keep one page frame free) Why did BSD do this? At some point OS must refuse to allocate a process more memory because has no swap space. When can this happen? (malloc, stack extension, new process creation).
- When process tries to access paged out memory, OS must run off to the disk, find a free page frame, then read page back off of disk into the page frame and restart process.
- What is advantage of virtual memory/paging?
  - Can run programs whose virtual address space is larger than physical memory. In effect, one process shares physical memory with itself.
  - Can also flexibly share machine between processes whose total address space sizes exceed the physical memory size.

  - Supports a wide range of user-level stuff - See Li and Appel paper.

- Disadvantages of VM/paging: extra resource consumption.
  - Memory overhead for storing page tables. In extreme cases, page table may take up a significant portion of virtual memory. One Solution: page the page table. Others: go to a more complicated data structure for storing virtual to physical translations.
  - Translation overhead.