# Operating System Concepts

## Lecture 25: Segmented Paging

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

# Today's class

- Memory-management strategies
  - contiguous memory allocation
  - non-contiguous memory allocation
    - segmentation
    - paging
    - segmented paging

# Putting it all together

- base & bounds registers
  - OS loads the registers
  - simple and fast, but does not support sharing, incremental increase of heap/stack, etc.
  - complex memory allocation

# Putting it all together

- base & bounds registers
  - OS loads the registers
  - simple and fast, but does not support sharing, incremental increase of heap/stack, etc.
  - complex memory allocation
- segment table
  - top few bits encode the segment number. Each segment has base and bounds
  - supports sharing
  - complex memory allocation

# Putting it all together

- base & bounds registers
  - OS loads the registers
  - simple and fast, but does not support sharing, incremental increase of heap/stack, etc.
  - complex memory allocation
- segment table
  - top few bits encode the segment number. Each segment has base and bounds
  - supports sharing
  - complex memory allocation
- page table
  - **each process has its own page table,** but TLB contains entries for several different processes simultaneously (**address-space identifiers** are used to determine which entry belongs to which process)
  - memory allocation is done in small page sizes (4K – 16K)
  - pages are contiguous in virtual address space, but they are arbitrarily located in physical memory
    - avoids external fragmentation and the need for compaction
  - paging supports sharing

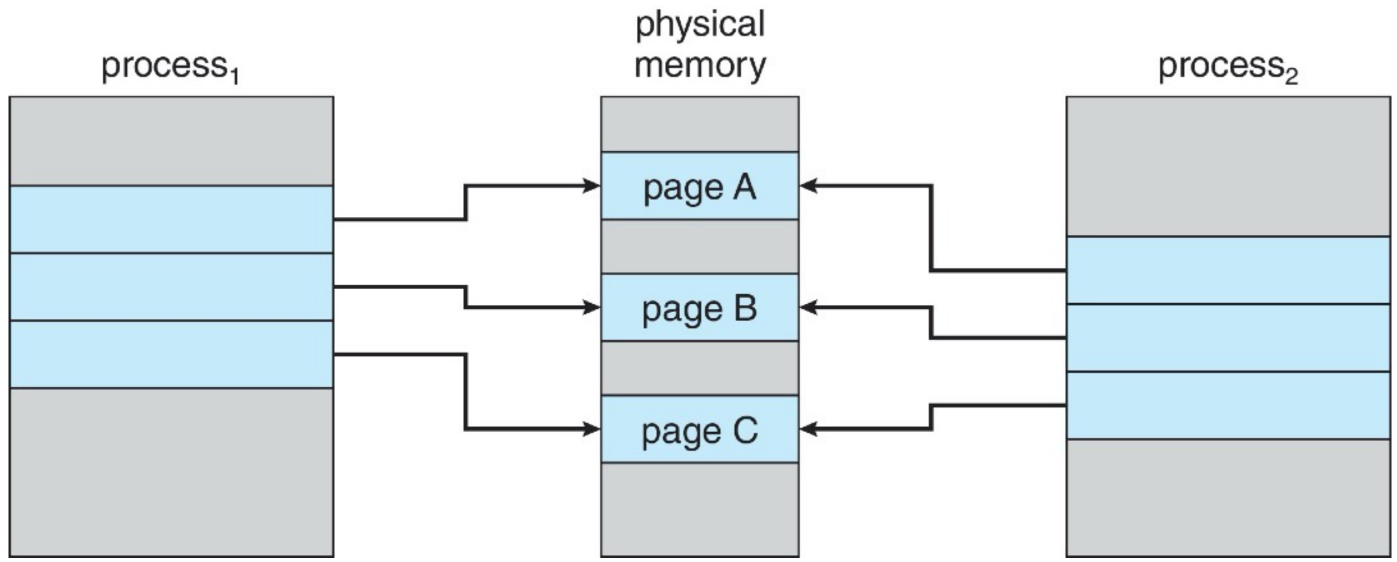
# Efficient forking in UNIX

- UNIX fork makes a complete copy of a process
- segmentation and paging allow for a more efficient implementation of the `fork()` system call
  - copy segment/page table into the child
  - mark parent and child segments/pages read-only (the share the same pages)
  - start child process and return to the parent
- most of the time child calls exec right after fork
  - without writing to read-only segments/pages
  - so duplicating the pages belonging to the parent is unnecessary most of the time

# Efficient forking in UNIX

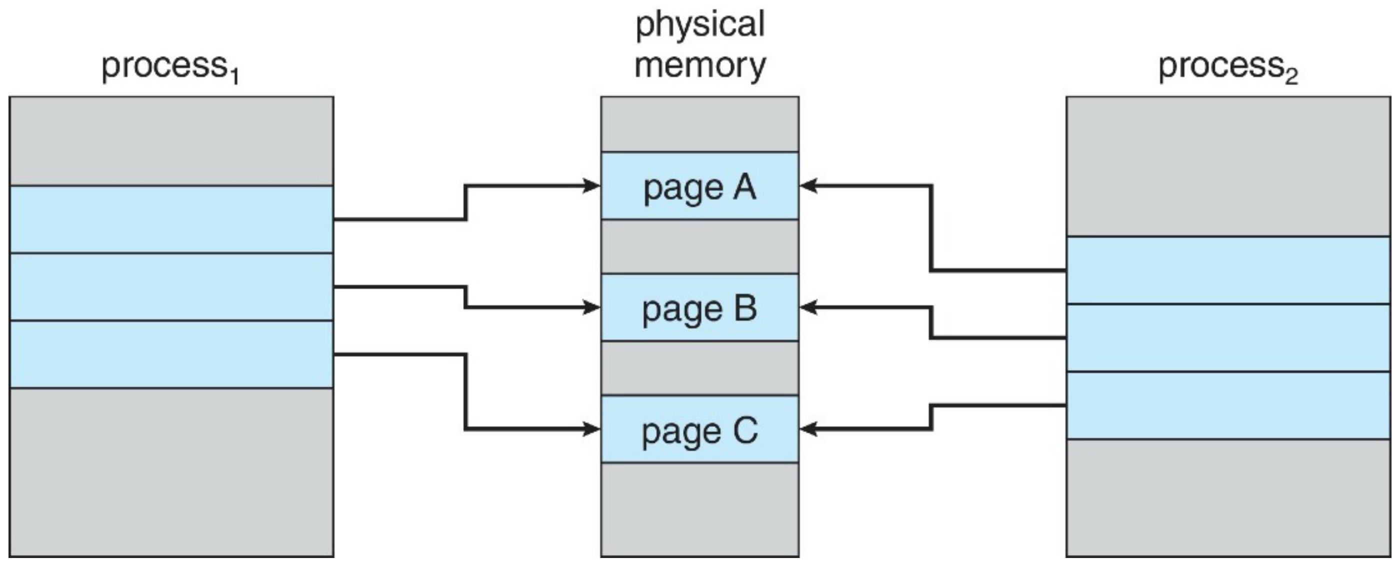
- UNIX fork makes a complete copy of a process
- segmentation and paging allow for a more efficient implementation of the `fork()` system call
  - copy segment/page table into the child
  - mark parent and child segments/pages read-only (the share the same pages)
  - start child process and return to the parent
- most of the time child calls exec right after fork
  - without writing to read-only segments/pages
  - so duplicating the pages belonging to the parent is unnecessary most of the time
- but if the child or the parent writes to a segment/page (e.g., stack, heap)
  - trap into the kernel
  - make a copy of the segment/page (this technique is known as **copy-on-write**)
  - mark both segments/pages as writable
  - resume execution

# Efficient forking in UNIX

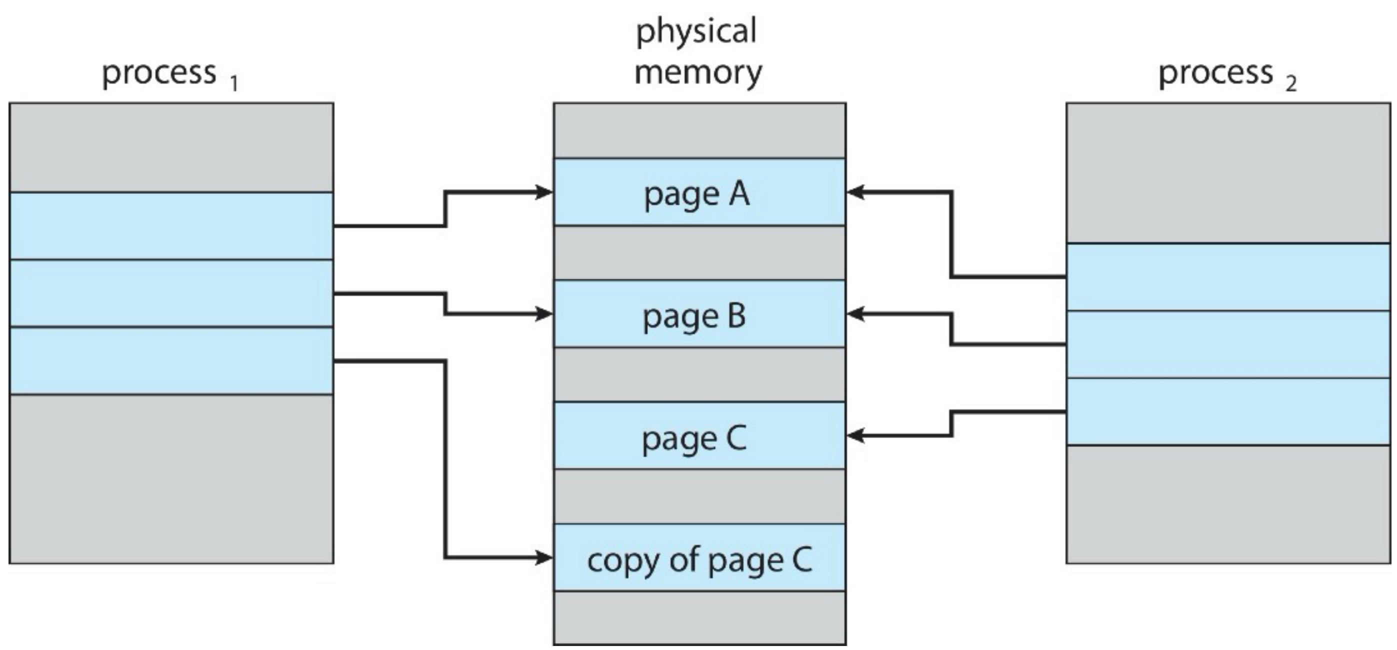



before process 1 modifies Page C

# Efficient forking in UNIX




before process 1 modifies Page C



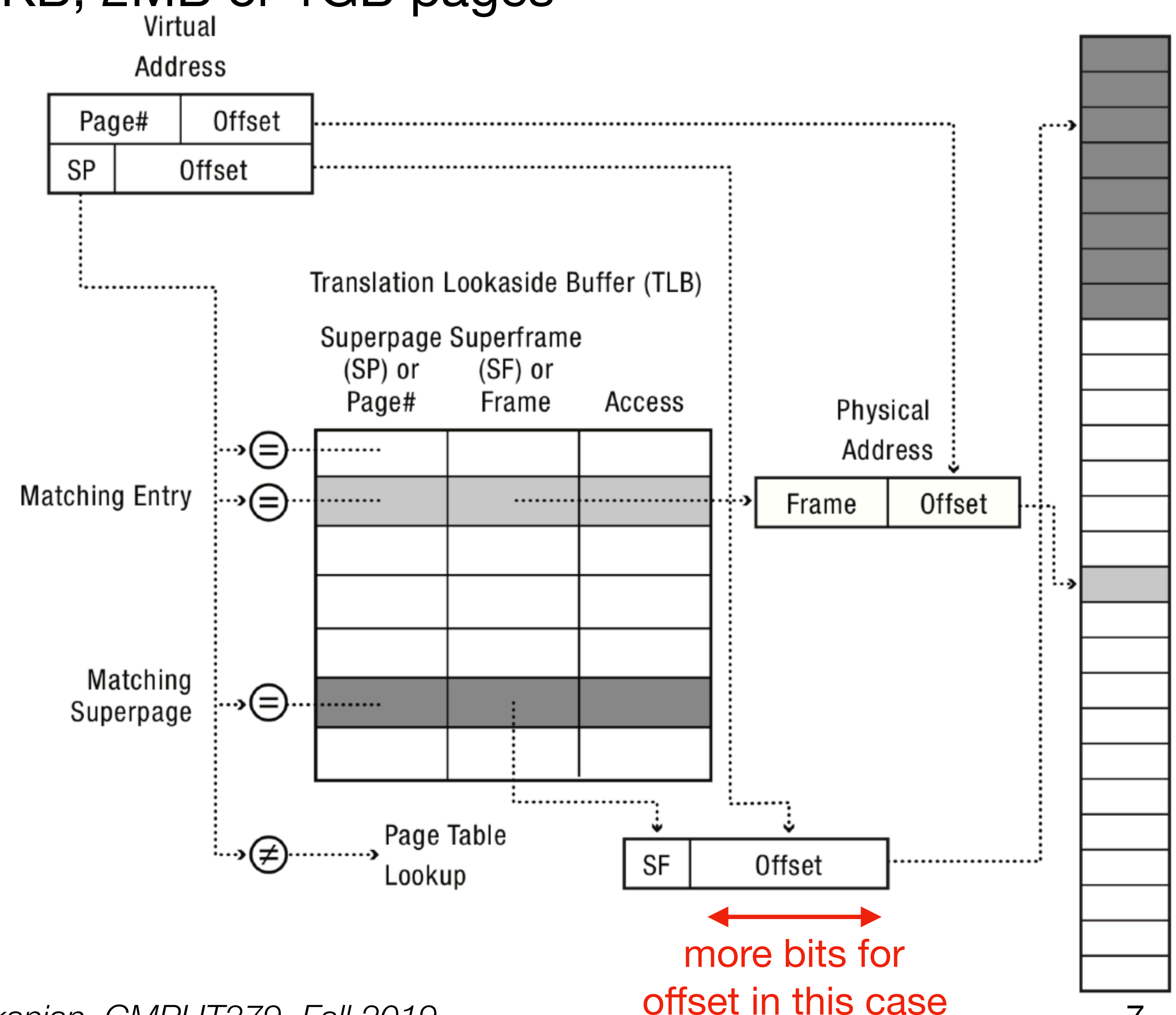

after process 1 modifies Page C
(a copy of the shared page is created)

# Growing heap or stack

- when a process uses memory beyond the end of stack or dynamically allocates memory beyond the end of heap
  - segmentation fault occurs
  - OS allocates some memory
  - OS zeros the newly allocated memory (this is known as **zero-on-reference**) to avoid accidentally leaking information!
  - OS modifies the segment table
  - the process is resumed

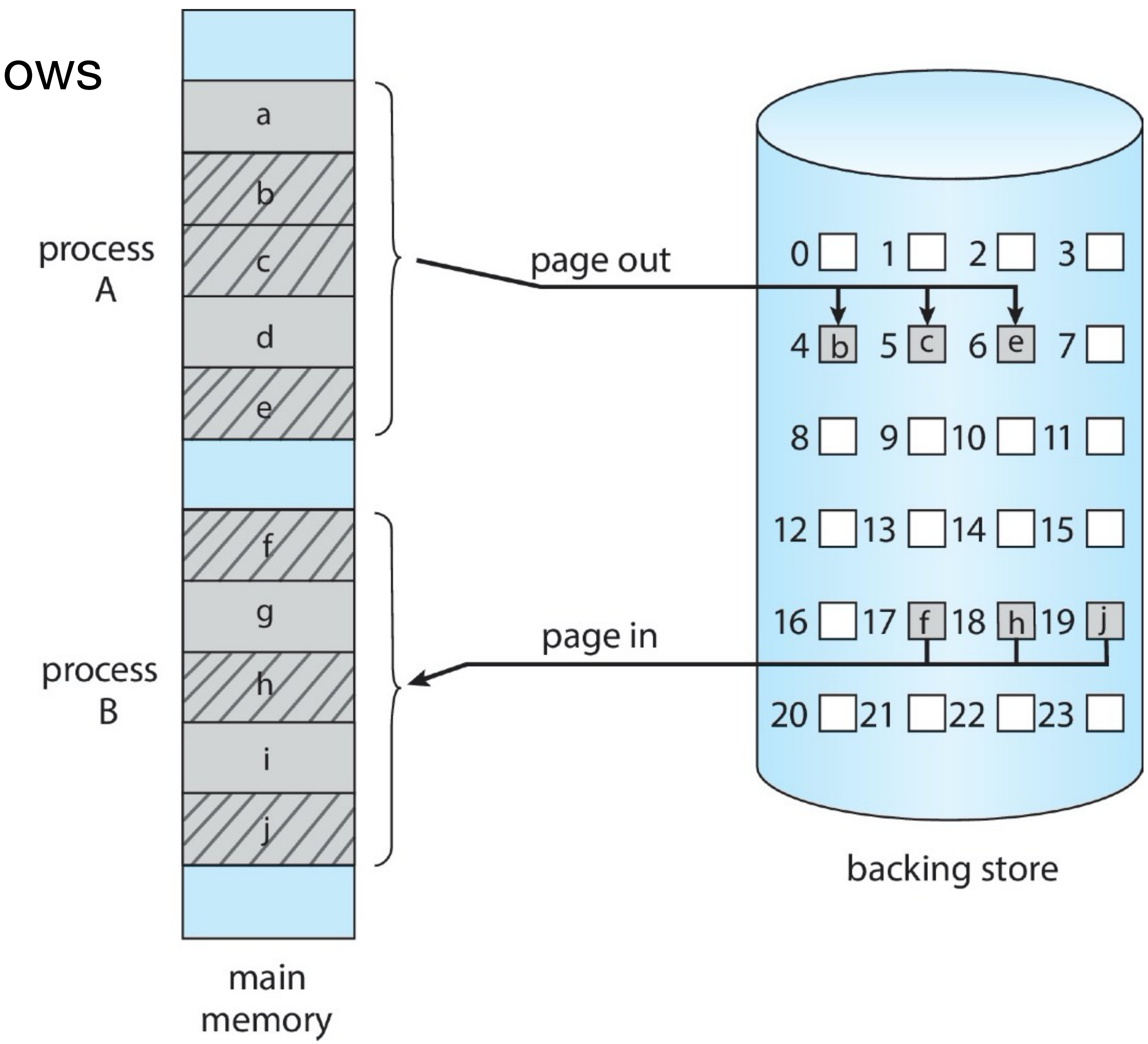
# Superpages

- on many systems, the first part of a TLB entry can be
  - a page number or a superpage (a set of contiguous pages in one page table) number
- x86 TLB entries can point to 4KB, 2MB or 1GB pages



# Swapping with paging

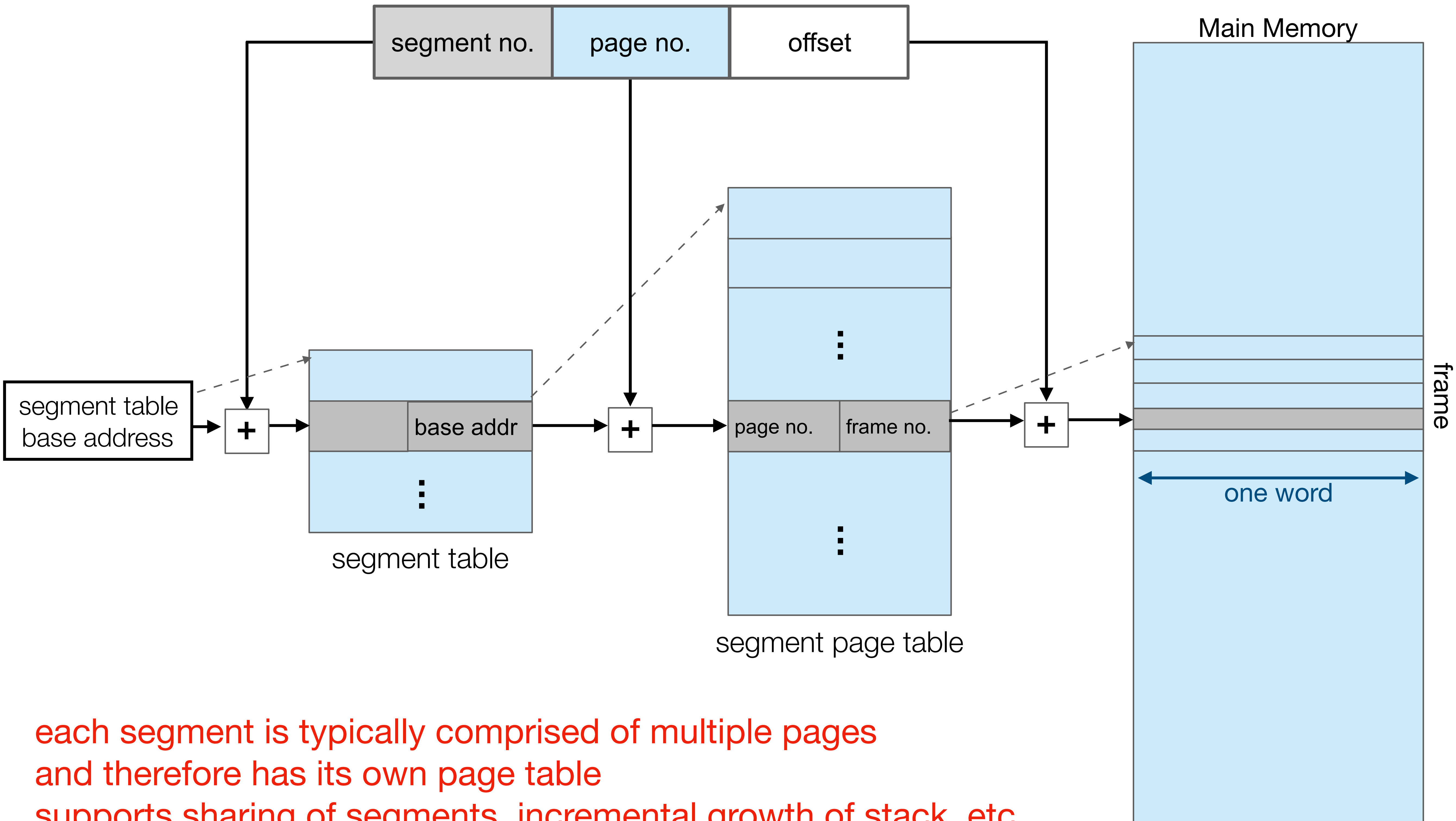
- specific pages of a process (rather than an entire process) are swapped
  - common in Linux and Windows



# Combining segmentation and paging

- think of virtual address space as a collection of segments (logical units) of arbitrary sizes
- think of physical memory as a sequence of fixed size page frames
- segments are typically larger than page frames
- map a logical segment onto multiple page frames by paging the segments

# Segmented paging




each segment is typically comprised of multiple pages
and therefore has its own page table
supports sharing of segments, incremental growth of stack, etc.

# Addresses in segmented paging

- a virtual address becomes a segment number, a page within that segment, and an offset within the page
- the segment number indexes into the segment table which yields the base address of the page table for that segment
- check the remainder of the address (page number and offset) against the limit of the segment
- use the page number to index the page table
  - the entry is the frame (just like paging)
- add the frame and the offset to get the physical address

# Example of segmented paging

given a memory size of 256 addressable words, a segment page table indexing 8 pages, a page size of 32 words, and 8 logical segments

- how many bits is a physical address?
- how many bits for the seg #, page #, offset?
- how many bits is a virtual address?

# Example of segmented paging

given a memory size of 256 addressable words, a segment page table indexing 8 pages, a page size of 32 words, and 8 logical segments

- how many bits is a physical address? **8 bits**
- how many bits for the seg #, page #, offset? **3, 3, and 5 bits**
- how many bits is a virtual address? **3 + 3 + 5 = 11 bits**

# What if the virtual address space is large?

- 32-bit logical address space, 4 KB page size ($2^{12}$) → over 1 million page table entries
  - each page table entry is typically 4 bytes => the size of the page table is over 4 MB!
  - hence the page table does not fit in one page

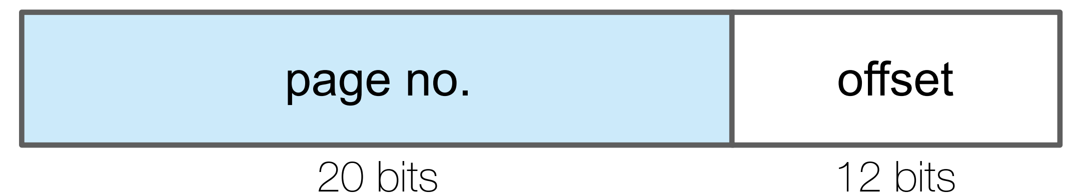
# What if the virtual address space is large?

- 32-bit logical address space, 4 KB page size ($2^{12}$) → over 1 million page table entries
  - each page table entry is typically 4 bytes => the size of the page table is over 4 MB!
  - hence the page table does not fit in one page
- 64-bit logical address space, 4 KB page size → over 4 quadrillion page table entries

# Paging in modern computer systems

memory structures for paging can get huge using straight-forward methods for modern systems that support a larger address space ($2^{32}$ to $2^{64}$)

- consider a 32-bit logical address space
  - page size of 4KB ($2^{12}$)
  - page table would have 1 million entries ($2^{32} / 2^{12}$)




- if each entry is 4 bytes then we need 4MB of physical address space for storing the page table alone
  - it is larger than the page size and we don't want to allocate that contiguously in main memory
  - in this case, we cannot use more than 10 bits for the page number if we want to fit the entire page table into one page