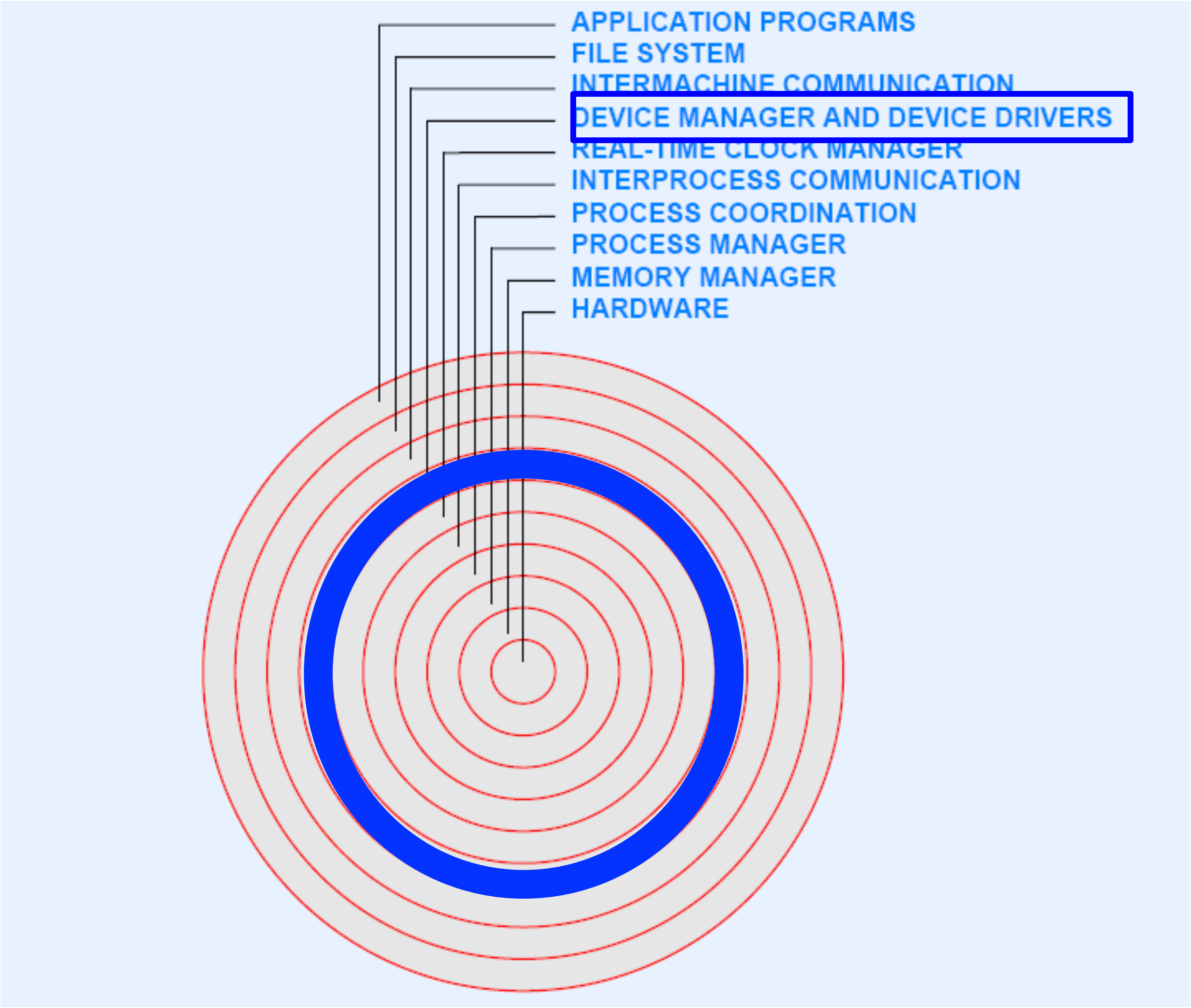
CSCI 8530
Advanced Operating Systems

Part 8

High-level Memory Management

Location of High-level Memory Management in the Hierarchy



Our Approach to Memory Management (Review)

- Divide memory manager into two pieces
- Low-level piece
  - A basic facility
  - Provides functions for stack and heap allocation
  - Treats memory as exhaustible resource
- High-level piece
  - Accommodates other memory uses
  - Assumes both operating system modules and sets of applications need dynamic memory allocation
  - Prevents exhaustion

Motivation for Memory Partitioning

- Competition exists for kernel memory
- Many of the subsystems in the operating system
  - allocate blocks of memory, and
  - have needs that change dynamically
- Examples:
  - A disk subsystem allocates buffers for disk blocks.
  - A network subsystem allocates packet buffers.
- Interaction among subsystems can be subtle and complex, e.g., deadlock can occur
  - The network process can block waiting for a disk buffer, but all memory is used for network buffers

Managing Memory

- Conflicting philosophies and tradeoffs
  - Protecting information
  - Sharing information
- Extreme examples
  - Xinu has much sharing; little protection
  - Original Unix™ had much protection; little sharing

The Concept of Firewalling

- Desires of an OS designer
  - Predictable behavior
  - Provable assertions (e.g., “network traffic will never deprive the disk driver of buffers”)
- Realities
  - Subsystems are designed independently
  - Memory used by one subsystem can interfere with others
- Conclusions
  - We must *not* treat memory as a single, global resource
  - We need a way to isolate subsystems

Providing Abstract Memory Resources

Assertion: To be able to make guarantees about subsystem behavior, one must partition memory into abstract resources with each resource dedicated to one subsystem.

A Few Examples of Memory Resources

- Disk buffers
- Network buffers
- Message storage
- Inter-process communication buffers (e.g., Unix pipes)

Note: each subsystem should operate safely and independently.

Xinu High-level Memory Manager

- Partitions memory into a set of *buffer pools*
  - Ex: Disk buffers or buffers for network packets
- Each pool is created once and persists until the system shuts down
- At pool creation time we fix the
  - Size of buffers in the pool
  - Number of buffers in the pool
- Once a pool has been created, buffer allocation and release
  - Is dynamic
  - Uses a synchronous interface

Xinu Buffer Pool Functions

poolinit – Initialize the entire mechanism
mkpool – Create a pool
getbuf – Allocate buffer from a pool
freebuf – Return buffer to a pool

- *mkpool: allocating m*emory for a pool when the pool is formed.
- *getbuf*: waiting on the semaphore until a buffer available, and then unlinking the first buffer from the list.

Note that although the buffer pool system allows callers to allocate a buffer from a pool and later release the buffer back to the pool, the pool itself cannot be deallocated, which means that the memory occupied by the pool can never be released.

Traditional Approach to Identifying a Buffer

- Use address of lowest byte in the buffer as the buffer address
- Guarantees each buffer has unique ID
- Allows buffer to be identified by a single pointer
- Works well in C
- Is convenient for programmers

Consequences of Using a Single Pointer

- *Freebuf*
  - Must return buffer to the correct pool
  - Takes buffer identifier as argument
- Information about buffer pools must be kept in a table
- Therefore, *freebuf* needs to find the pool from which buffer was allocated

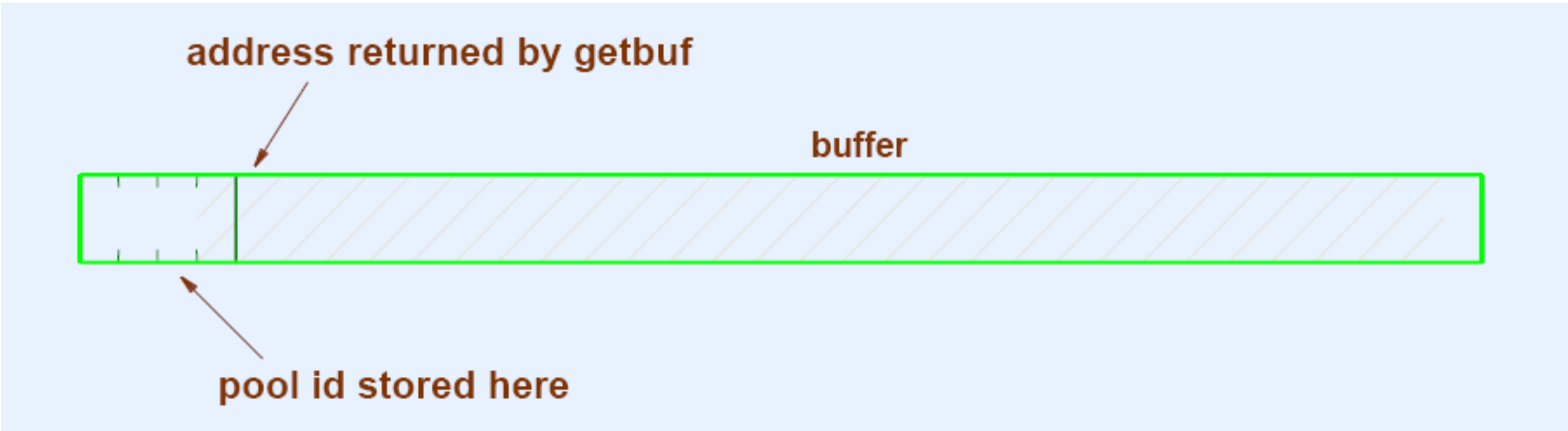
Finding the Pool for a Buffer

- Possibilities
  - Search the table of buffer pools to find the correct pool
  - Use an external data structure to map buffer address to pool (e.g., keep a list of allocated buffers and the pool to which each belongs)
- An alternative
  - Have *getbuf* return two values: a pool ID and a buffer address
  - Have *freebuf* take pool ID and buffer address as arguments
  - Inconvenient for programmers

Solving the Single Pointer Problem

- The Xinu solution
  - Pass single buffer address to user
  - Store pool ID with each buffer
- Implementation trick
  - Allocate enough extra bytes to hold an ID
  - Store the pool ID in the extra bytes
  - Place the extra bytes *before* the buffer
  - Return a pointer to the buffer, not the extra bytes
- Caller can use buffer without knowing about the extra bytes

Illustration of Pool ID Stored with Buffer



- Additional four bytes preceding buffer store the pool ID
- *Getbuf* returns single pointer to data area
- *Freebuf* expects same pointer as *getbuf* returned
- Transparent to applications using the buffer pool

Question

- What is the potential disadvantage of storing a pool ID in front of each buffer?

Answer

- User may desire buffers to be *aligned*
- Example: some device hardware requires buffers to start on a page boundary
- Additional bytes cause alignment problems

Buffer Pool Operations

- Create a pool
  - Use *getmem* to allocate memory for buffers in the pool
  - Form a singly-linked list of buffers (storing links in the buffers themselves)
  - Allocate a semaphore to count buffers
- Allocate a buffer from a pool
  - Block on the semaphore until a buffer is available
  - Take the buffer at the head of the list
- Deallocate a buffer
  - Insert the buffer at the head of the list
  - Signal the semaphore

Xinu Mkbufpool (part 1)

```
/* mkbufpool.c - mkbufpool */

#include <xinu.h>

/*------------------------------------------------------------------------
 * mkbufpool - Allocate memory for a buffer pool and link the buffers
 *------------------------------------------------------------------------
 */
bpid32 mkbufpool(
        int32 bufsiz,                /* Size of a buffer in the pool */
        int32 numbufs                /* Number of buffers in the pool*/
       )
{
        intmask mask; /* Saved interrupt mask */
        bpid32 poolid; /* ID of pool that is created */
        struct bpentry *bpptr; /* Pointer to entry in buftab */
        char *buf; /* Pointer to memory for buffer */

        mask = disable();
        if (bufsiz<BP_MINB || bufsiz>BP_MAXB
            || numbufs<1 || numbufs>BP_MAXN
            || nbpools >= NBPOOLS) {
                restore(mask);
                return (bpid32)SYSERR;
        }
```

1. The buffer size is out of range
2. The requested number of buffers is negative
3. The buffer pool table is full

Xinu Mkbufpool (part 2)

```
/* Round request to a multiple of 4 bytes */
bufsiz = ( (bufsiz + 3) & (~3) );
buf = (char *)getmem( numbufs * (bufsiz+sizeof(bpid32)) );
if ((int32)buf == SYSERR) {
        restore(mask);
        return (bpid32)SYSERR;
}
poolid = nbpools++;
bpptr = &buftab[poolid];
bpptr->bpnext = (struct bpentry *)buf;
bpptr->bpsize = bufsiz;
if ( (bpptr->bpsem = semcreate(numbufs)) == SYSERR) {
        nbpools--;
        restore(mask);
        return (bpid32)SYSERR;
}
bufsiz+=sizeof(bpid32);
for (numbufs-- ; numbufs>0 ; numbufs-- ) {
        bpptr = (struct bpentry *)buf;
        buf += bufsiz;
        bpptr->bpnext = (struct bpentry *)buf;
}
bpptr = (struct bpentry *)buf;
bpptr->bpnext = (struct bpentry *)NULL;
restore(mask);
return poolid;
}
```

Allocate the needed memory

Allocates an entry in the buffer pool table and fills in entries.

1. Create a semaphore,
2. Save the buffer size
3. Store the address of the allocated memory in bpnex.

Iterate through the allocated memory, dividing the block into a set of buffers

Xinu Getbuf (part 1)

Allocate a buffer calls wait on a pool's semaphore

```
/* getbuf.c - getbuf */

#include <xinu.h>

/*------------------------------------------------------------------------
 * getbuf - Get a buffer from a preestablished buffer pool
 *------------------------------------------------------------------------
 */
char *getbuf(
        bpid32 poolid                   /* Index of pool in buftab */
      )
{
        intmask mask; /* Saved interrupt mask */
        struct bpentry *bpptr;          /* Pointer to entry in buftab */
        struct bpentry *bufptr;         /* Pointer to a buffer */

        mask = disable();

        /* Check arguments */
        if ( (poolid < 0 || poolid >= nbpools) ) {
                restore(mask);
                return (char *)SYSERR;
        }
        bpptr = &buftab[poolid];
```

Xinu Getbuf (part 2)

```
	/* Wait for pool to have > 0 buffers and allocate a buffer */

	wait(bpptr->bpsem);
	bufptr = bpptr->bpnext;

	/* Unlink the first buffer from the pool list */
	bpptr->bpnext = bufptr->bpnext;

	/* Record pool ID in first four bytes of buffer and skip */
	*(bpid32 *)bufptr = poolid;
	bufptr = (struct bpentry *)(sizeof(bpid32) + (char *)bufptr);
	restore(mask);
	return (char *)bufptr;
}
```

Waiting on the semaphore until a buffer is available

Xinu Freebuf (part 1)

```
/* freebuf.c - freebuf */

#include <xinu.h>

/*------------------------------------------------------------------------
 * freebuf - Free a buffer that was allocated from a pool by getbuf
 *------------------------------------------------------------------------
 */
syscall freebuf(
            char *bufaddr /* Address of buffer to return */
        )
{
        intmask mask; /* Saved interrupt mask */
        struct bpentry *bpptr; /* Pointer to entry in buftab */
        bpid32 poolid; /* ID of buffer's pool */

        mask = disable();

        /* Extract pool ID from integer prior to buffer address */
        bufaddr -= sizeof(bpid32);
        poolid = *(bpid32 *)bufaddr;
        if (poolid < 0 || poolid >= nbpools) {
                restore(mask);
                return SYSERR;
        }
```

Xinu Freebuf (part 2)

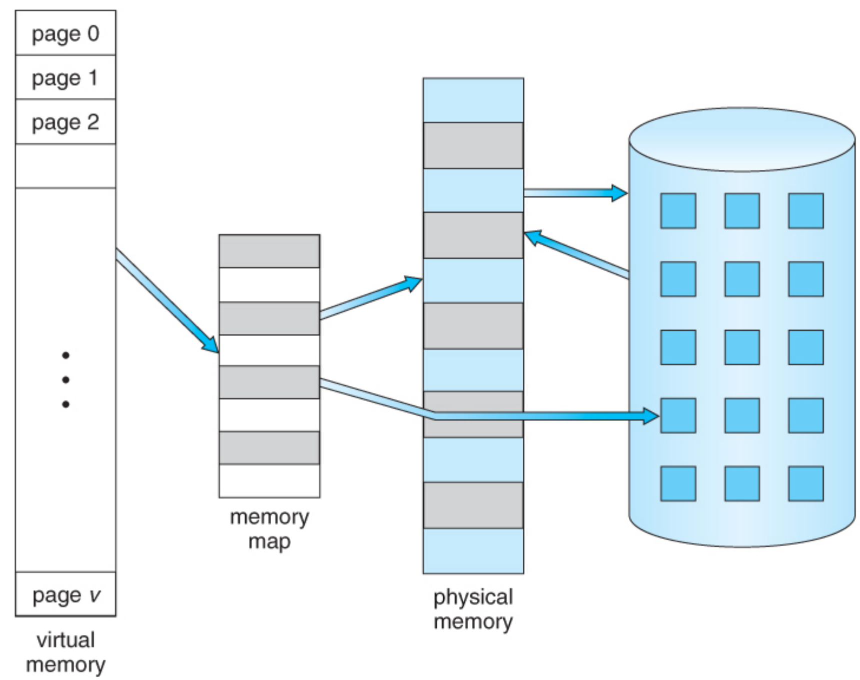
```
    /* Get address of correct pool entry in table */

    bpptr = &buftab[poolid];

    /* Insert buffer into list and signal semaphore */

    ((struct bpentry *)bufaddr)->bpnext = bpptr->bpnext;
    bpptr->bpnext = (struct bpentry *)bufaddr;
    signal(bpptr->bpsem);
    restore(mask);
    return OK;
}
```

Allow other processes to use the buffer

VIRTUAL MEMORY

Definition of Virtual Memory

- Abstraction of physical memory
- Provides separation from underlying hardware details
- Primarily applied to applications (user processes)
- Allows applications to run independent of
  - Physical memory size
  - Position in physical memory



General Approach

- Heavyweight process
  - Lives in an isolated address space
  - All addresses are *virtual*
- Operating system
  - Establishes policies
  - Provides support for virtual address space creation
  - Configures the hardware
- Underlying hardware
  - Dynamically translates from virtual address to physical address
  - Provides support to help OS make policy decisions

Virtual Address Space

- Can be smaller than physical memory
  - A 32-bit computer with more than 2^{32} bytes (four GB) of memory
- Can be larger than the physical memory
  - A 64-bit computer with less than 2^{64} bytes (16 million terabytes) of memory
- Historical note: On early computers, physical memory was larger. Then virtual memory was larger until physical memory caught up. Now with 64-bit architectures, we find virtual memory is once again larger than physical memory.

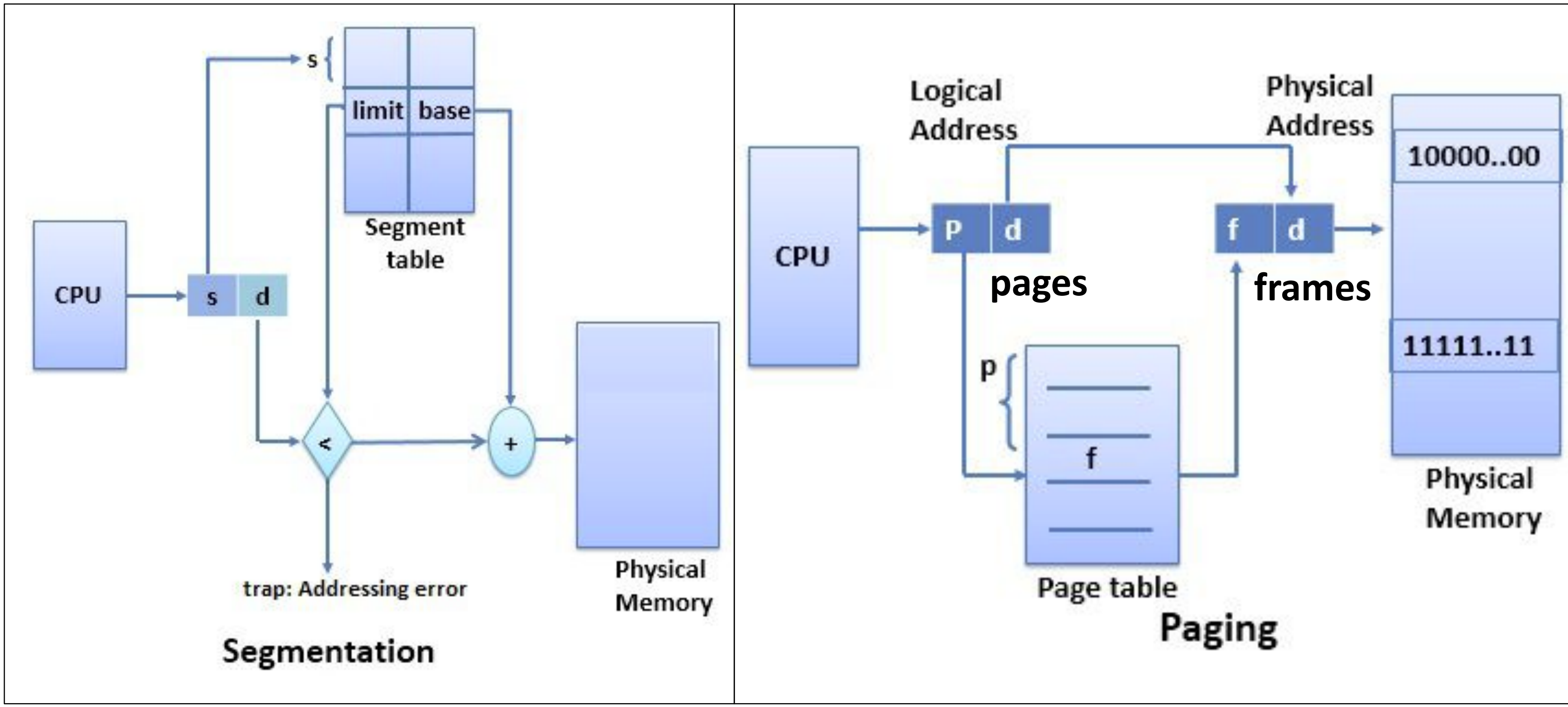
Multiplexing Virtual Address Spaces Onto Physical Memory

- General idea
  - Store a complete copy of each process's address space on secondary storage
  - Move items to main memory as needed
  - Write items back to disk to create space in memory for other items
- Questions
  - How much of a process's address space should reside in memory?
  - When should items be loaded into memory?
  - When should items be written back to disk?

General Approaches for Virtual Memory Management Systems

- *Swapping*
  - Transfer an *entire* address space (complete process image)
- *Segmentation*
  - Divide the image into *large* segments
  - Transfer segments as needed
- *Paging*
  - Divide image into *small* and *fixed-size* pieces
  - Transfer individual pieces as needed

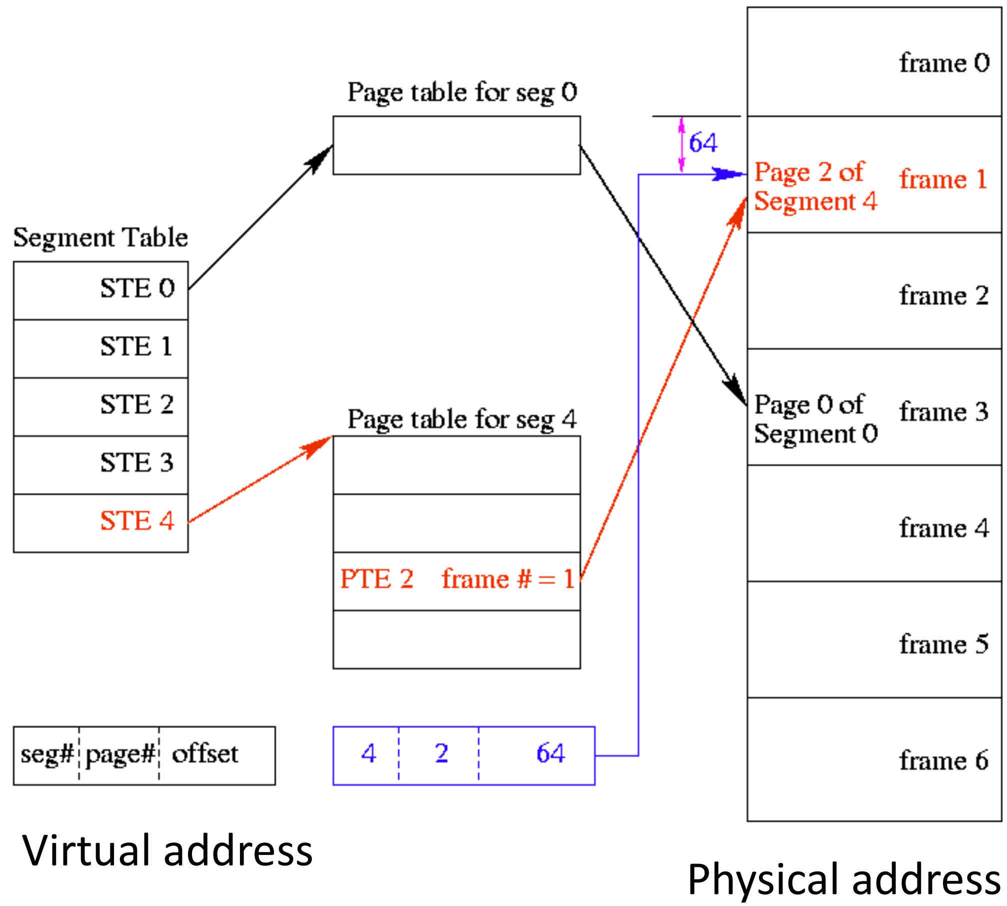
Segmentation V.S. Paging



https://techdifferences.com/difference-between-paging-and-segmentation-in-os.html

General Approaches (continued)

- *Segmentation with paging*
  - Divide an image into large segments
  - Further subdivide segments into fixed-size pages
- Note: Simple paging has emerged as the most popular



https://cs.nyu.edu/courses/spring04/G22.2250-001/lectures/lecture-12.html

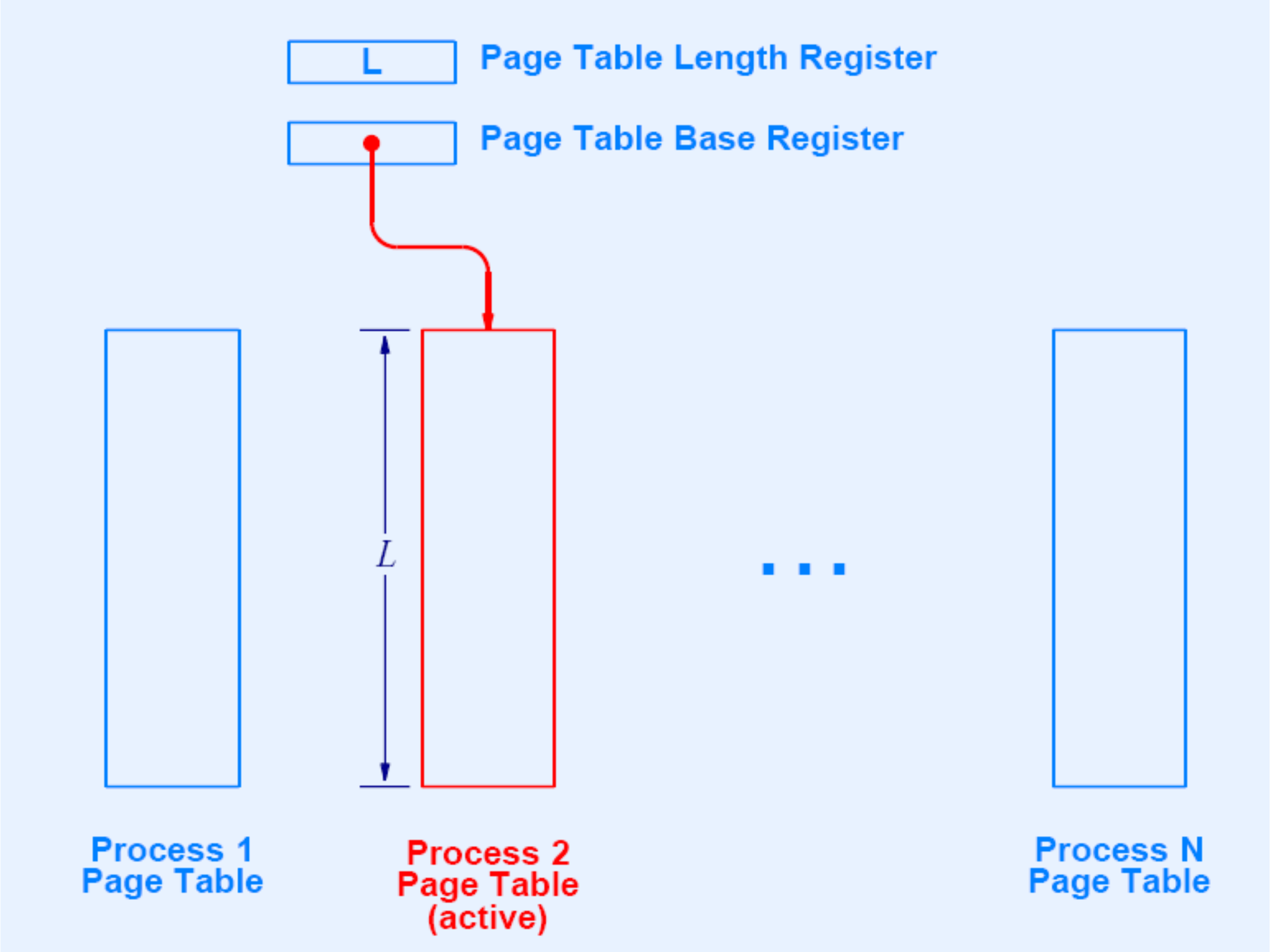
Hardware Support for Paging

- Page tables
  - One page table per process
  - Storage location depends on hardware
    - A page table sides in kernel memory
    - MMU hardware (on some systems)
- Page table base register
  - Internal to the processor
  - Contains the address of current process's page table
  - Must be changed during a context switch

Hardware Support for Paging (continued)

- Page table length register
  - Internal to the processor
  - Specifies the number of entries in a page table
  - Determines the size of the virtual address space
  - Can be changed during context switch if the size of the virtual address space differs among processes

Illustration of VM Hardware Registers



- Only one page table is active at given time

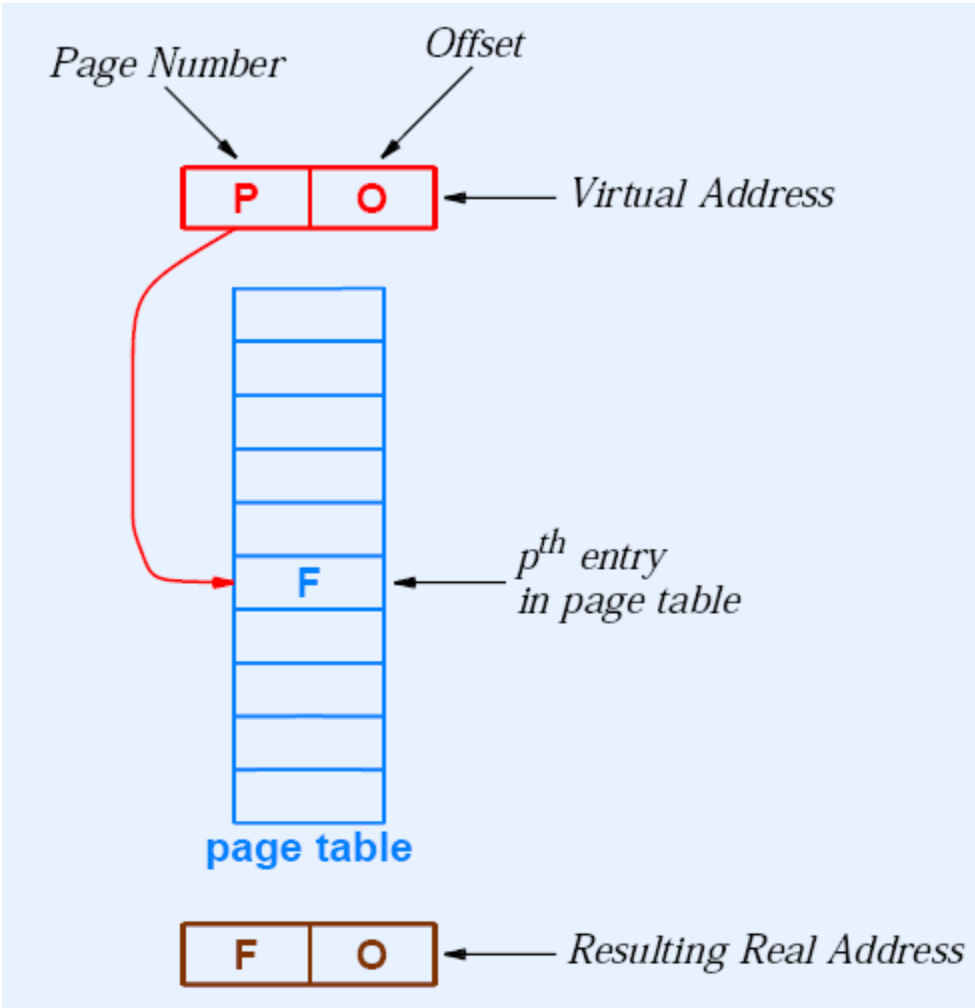
Address Translation

- Is performed by memory management hardware
- Must be applied to *every* memory reference
- Hardware translates from process's virtual address to corresponding physical memory address
- Translation consists of array lookup
  - The hardware treats the high-order bits of an address as a page number
- If the translation fails, a *page fault* is generated

Address Translation with Paging

- For now, we will assume
  - The operating system is not paged
  - The physical memory area beyond the operating system kernel is used for paging
  - The page size is 4 Kbytes
- Think of the physical memory area use for paging as a giant array of *frames*, where each frame can hold one page (i.e. a frame is 4K bytes)

Illustration of Address Translation



- Pages start in memory locations that have zeroes in the low-order bits
- Page table entry contains physical frame address
- Choosing the page size to be a power of 2 eliminates division and modulus

In Practice

- Some hardware offers separate page tables for text, data, and stack
  - Disadvantage: complexity
  - Advantage: independent policies
- The size of virtual space may be limited to physical memory size
- The kernel address space can also be virtual (but it hasn’t worked well in practice)

Demand Paging

- Keep the process image on secondary storage
- Treat main memory as cache of *recently-referenced pages*
- Allocate space in the cache dynamically as needed
- Move pages between the secondary store and main memory on demand (when referenced)

The Importance Of Hardware Support For Virtual Memory

- Every memory reference must be translated from a virtual address to a physical address
  - Instructions must be translated as well as data (to accommodate branching)
  - Indirect addresses that are generated at runtime must be translated
- Hardware support is essential
  - For efficiency
  - For recovery if a fault occurs
  - To record which pages are being used

In Practice

- A single instruction may reference many pages
  - Instruction fetch
  - Operand fetch
  - Indirect reference resolution
  - Memory copy instructions on CISC hardware
- Special-purpose hardware speeds page lookup
  - *Translation Look-aside Buffer* (*TLB*)
  - Implemented with T-CAM (parallel)
  - TLB caches most recent address translations

In Practice (continued)

- VM mappings change during context switch
  - Some hardware requires OS to flush TLB
  - Other hardware uses tags to distinguish among address spaces
    - Tag assigned by OS
    - Typically, a process ID

Bits That Record Page Status

- Part of page table entry for each page
- Understood by hardware
- *Used/Referenced Bit* (“accessed” bit on Intel)
  - Set whenever page referenced
  - Applies to all fetch and store operations
- *Modified/Dirty Bit* (“dirty” bit on Intel)
  - Set on store operation
- *Presence Bit* (“present” bit on Intel)
  - Set by OS if page currently *resident* in memory

Questions for OS Designers

- Which VM policies are most effective?
- Which pages from which address spaces should be in memory at any time?
- Should some pages be locked in memory? (If so, which ones?)
- How does a VM policy interact with other policies (e.g., scheduling?)
- Should high-priority processes /threads have guarantees about the number of resident pages?
- If a system supports libraries that are shared among many processes, which paging policy applies?

A Critical Tradeoff for Demand Paging

- Paging overhead and latency for given process can be reduced by giving the process more physical memory (more frames)
- Processor utilization and overall throughput are increased by increasing level of multitasking
- Throughput is maximized when all ready processes are resident

Terminology

- *Reference string*: a list of page references emitted by a process
- *Resident set*: the subset of process's pages currently present in main memory
- *Page*: a fixed size piece of process's address space
- *Frame*: a "slot" in main memory exactly the size of a page
- *Dirty*: a page that has been modified since it was last written to secondary store
- *Page fault*: an error that occurs when a referenced page is not present

Page Replacement

- Hardware
  - Detects a page fault
  - Raises an exception
- Operating system
  - Receives the exception
  - Allocates a frame
  - Retrieves the needed page
  - Allows other processes to execute while page is being fetched
  - Allows blocked process to continue once the page arrives

Frame Allocation

- A frame must be allocated when a page fault occurs
  - Allocation is trivial if free (unfilled) frames exist
  - Allocation is difficult if all frames are currently used
- A policy is needed because operating system must
  - Select one of the resident pages and save a copy on disk
  - Mark the page table to indicate the page is no longer resident
  - Load the needed page into the frame
  - Set the appropriate page table entry
  - Return from the page fault to restart the operation
- Question: which frame should be selected when all are in use?

Choosing a Frame

- Competition can be
  - Global: consider frames from all processes when choosing
  - Local: choose a frame within the same process that caused the page fault
- Some possible selection policies
  - *First In First Out* (*FIFO*)
  - *Least Recently Used* (*LRU*)
  - *Least Frequently Used* (*LFU*)

Example Page Replacement Algorithms

- We will consider three examples
  - Belady’s optimal page replacement algorithme
  - Global clock (second chance algorithm)
  - Working set algorithm

Belady’s Optimal Algorithm

- Chooses a page that will be referenced farthest in the future
- Provably optimal
- Totally impractical (of theoretical interest only)
- Useful for comparison of other algorithms
- Corollary: increasing the physical memory size does not always decrease the rate of page faults (known as *Belady’s Anomaly*).

FIFO: 9 Page Faults with 3 Frames

Trace:	1	2	3	4	1	2	5	1	2	3	4	5
Fault?	✓	✓	✓	✓	✓	✓	✓			✓	✓	

Frame 1	1	1	1	4	4	4	5	5	5	5	5	5
Frame 2		2	2	2	1	1	1	1	1	3	3	3
Frame 3			3	3	3	2	2	2	2	2	4	4

FIFO: 10 Page Faults with 4 Frames

Trace:	1	2	3	4	1	2	5	1	2	3	4	5
Fault?	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓

Frame 1	1	1	1	1	1	1	5	5	5	5	4	4
Frame 2		2	2	2	2	2	2	1	1	1	1	5
Frame 3			3	3	3	3	3	3	2	2	2	2
Frame 4				4	4	4	4	4	4	3	3	3

Global Clock Algorithm

- Originated in the MULTICS operating system
- Allows processes to compete with one another (hence the term *global*)
- Relatively low overhead
- Widely implemented (the most popular practical method)

Global Clock Paradigm

- Clock algorithm is activated when a page fault occurs
- Searches frames in memory, and selects a frame to use
- Gives a frame containing a referenced page a “second chance” before reclaiming
- Gives a frame containing a modified page a “third chance” before reclaiming
- In the worst case: the clock sweeps through all frames twice before reclaiming one
- Does *not* require external data structure other than standard page table bits

Operation of the Global Clock

- Uses a global pointer that picks up where it left off previously
  - Sweeps slowly through all frames in memory
  - Starts moving when a frame is needed
  - Stops moving once a frame has been selected
- Check both the *reference bit* and *modify bit* to determine which page to replace
  - (reference, modify) pairs form classes:
    - (0,0): not used or modified, *replace*!
    - (0,1): not recently used but modified: OS needs to write, but may not be needed anymore
    - (1,0): recently used and unmodified: may be needed again soon, but doesn’t need to be written
    - (1,1): recently used and modified
- The algorithm keeps a copy of the actual modified bit to know whether page is dirty

In Practice

- The global clock always reclaims a small set of frames when one is needed
  - If the OS finds (0,0)
  - If the OS finds (0,1)
  - For pages with the reference bit set, the reference bit is cleared
- The reclaimed frames are cached for subsequent references
- Advantage: collecting multiple frames avoids running the clock frequently

Level of Multitasking

- If too many processes are running
  - Each process will not have many frames
  - Paging activity will be high
  - Throughput will be low
- If too few processes are running, paging activity is low, but
  - The processor will not have a process to run while other processes wait for I/O
  - Throughput will be low

Load Balancing

- Refers to adjusting the number of processes that compete for frames
- Goal is to maximize throughput
- Works best in systems that have
  - A steady, predictable workload
  - A large set of available processes
- Is performed in cooperation with, and response to, the global clock
- Multitasking level (number of active processes) is
  - Decreased when page faults are frequent
  - Increased when paging level is low

Difficulties in Load Balancing

- Choosing thresholds
- Collecting measurements
- Anticipating paging activity
- It is easy to overreact

Working Set Algorithm

- More theory
- Finer level of assessment
- Definition

  Let δ be the size of a window on a reference string measured in page references. A process's working set **at time** i, w(i)**, consists of the set of pages in the** δ **references immediately preceding time** i**.**

- Note: a working set does not contain duplicates because it is a set in a mathematical sense.

Working Set Example for δ = 12

time →

3 2 5 $\boxed{3}$ 8 3 27 3 6 27 2 11 5 1 $\boxed{2}$

w(1) = 3
w(2) = 3, 2
w(3) = 3, 2, 5
w(4) = 3, 2, 5
w(5) = 3, 2, 5, 8
w(6) = 3, 2, 5, 8
w(7) = 3, 2, 5, 8, 27
w(8) = 3, 2, 5, 8, 27
w(9) = 3, 2, 5, 8, 27, 6
w(10) = 3, 2, 5, 8, 27, 6
w(11) = 3, 2, 5, 8, 27, 6
w(12) = 3, 2, 5, 8, 27, 6, 11
w(13) = 3, 2, 5, 8, 27, 6, 11
w(14) = 3, 2, 5, 8, 27, 6, 11, 1
w(15) = 3, 2, 5, 8, 27, 6, 11, 1

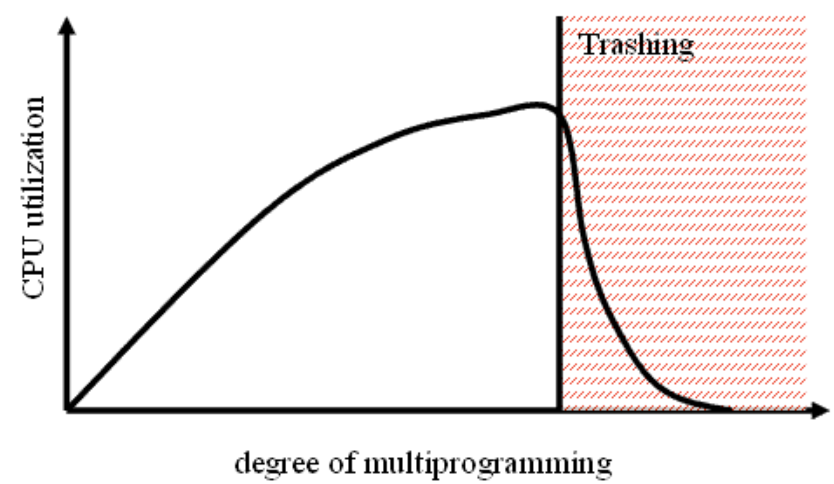
Working Set Terminology

- *Working set size*: number of pages in the process's working set
- *Resident set size*: number of pages of process currently resident
- Note: Both resident and working sets vary over time

Working Set Replacement Policy

Allocate to each process a resident set size equal to the process' working set size.

too big-> memory is wasted
too small->thrashing



Level of Multitasking Under Working Set

- Level of multitasking
  - Is adjusted dynamically
  - Is a function of current working set sizes and memory size
- Idea: compare the sum of the working set sizes, *W*, and frames, *F*
  - Reduce multitasking when

    $W > F$
  - Increase multitasking when

    $W << F$

Assessment of Working Set Model

- Pros
  - Simulation shows it performs well
  - It uses paging performance as a way to balance load
- Cons
  - Need both a minimum and maximum bound on resident set size
  - Difficult (impossible) to capture the needed information
    - Working set must be recomputed on each per memory reference
    - Even a hardware solution is inadequate

Questions for Thought

- Virtual memory was once the most important research topic in operating systems.
  - What changed?
  - Why did the topic fade?
  - Has the problem been solved completely?

Summary (1/2)

- We considered two forms of high-level memory management
- Inside the kernel
  - Define a set of abstract resources
  - Firewalling memory used by each prevents interference
  - Mechanism uses buffer pools
  - Buffer referenced by single address
- Outside the kernel
  - Swapping, segmentation, or paging

Summary (2/2)

- Demand paging
  - Fixed size pages
  - Brought into memory when referenced
- Algorithms
  - Belady’s algorithm (theoretical)
  - Global clock (widely used and practical)
- Working set (theoretical) relates paging to load balancing