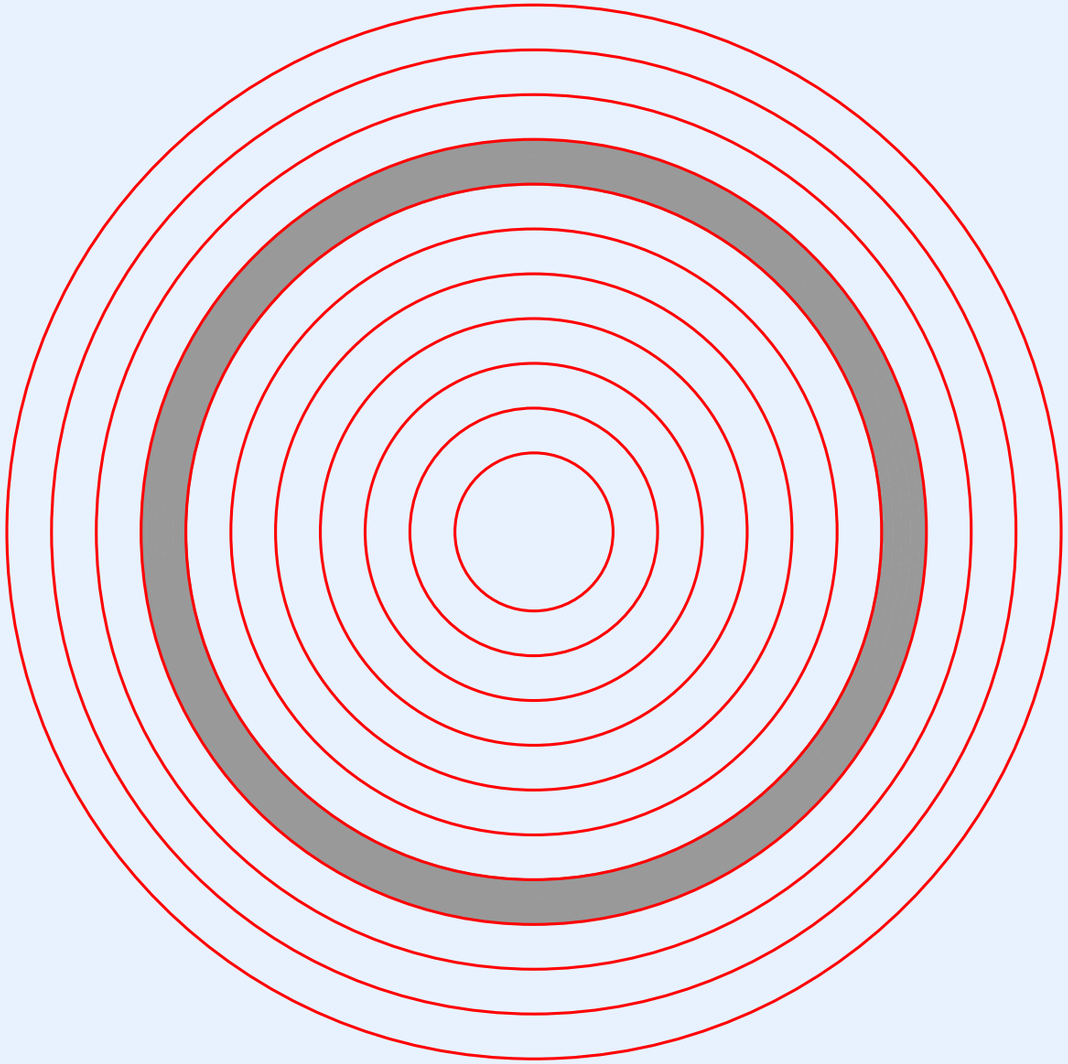
Module X

High-Level Memory Management

Location Of High-Level Memory Management In The Hierarchy



Our Approach To Memory Management (Review)

- Divide the 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 subsystems in the operating system
  - Allocate blocks of memory
  - Have needs that change dynamically
- Examples
  - The disk subsystem allocates buffers for disk blocks
  - The network subsystem allocates packet buffers
- Interaction among subsystems can be subtle and complex

Managing Memory Demands

- Overall goals can conflict
  - Protect information
  - Share information
- Extremes
  - Xinu has much sharing and almost no protection
  - The original Unix™ system had much protection and almost no sharing

The Concept Of Subsystem Isolation

- An OS designer desires
  - Predictable behavior
  - Provable assertions (e.g., “network traffic will never deprive the disk driver of buffers”)
- The reality
  - Subsystems are designed independently; there is no global policy or guarantee about their memory use
  - If one subsystem allocates memory excessively, others can be deprived
- Conclusions
  - We must not treat memory as a single, global resource
  - We need a way to isolate subsystems from one another

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 Abstract Resources

- Disk buffers
- Network buffers
- Message buffers
- A separate address space for each process as in Unix
- Inter-process communication buffers (e.g., Unix pipes)
- Note that
  - Each subsystem should operate safely and independently
  - An operating system designer may choose to define finer granularity separations
    * A separate set of buffers for each network interface (Wi-Fi and Ethernet)
    * A separate set of buffers for each disk

The Xinu High-Level Memory Manager

- Partitions memory into groups of *buffer pools*
- Each pool is created once and persists until the system shuts down
- All buffers in a given pool are the same size
- At pool creation, the caller specifies
  - The size of buffers in the pool
  - The number of buffers in the pool
- Once a pool has been created, buffer allocation and release is dynamic
- The system provides a completely synchronous interface

Xinu Buffer Pool Functions

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

- Memory for a pool is allocated by *mkbufpool* when the pool is formed

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.

The Traditional Approach To Identifying A Buffer

- Most systems use the address of lowest byte in the buffer as the buffer address
- Doing so means
  - Each buffer is guaranteed to have a unique ID
  - A buffer can be identified by a single pointer
- The scheme
  - Works well in C
  - Is convenient for programmers

Consequences Of Using A Single Pointer As An ID

- Consider function *freebuf*
  - It must return a buffer to the correct pool
  - It takes the buffer identifier as argument
- Information about buffer pools must be kept in a table
- Given a buffer, *freebuf* needs to find the pool from which the buffer was allocated

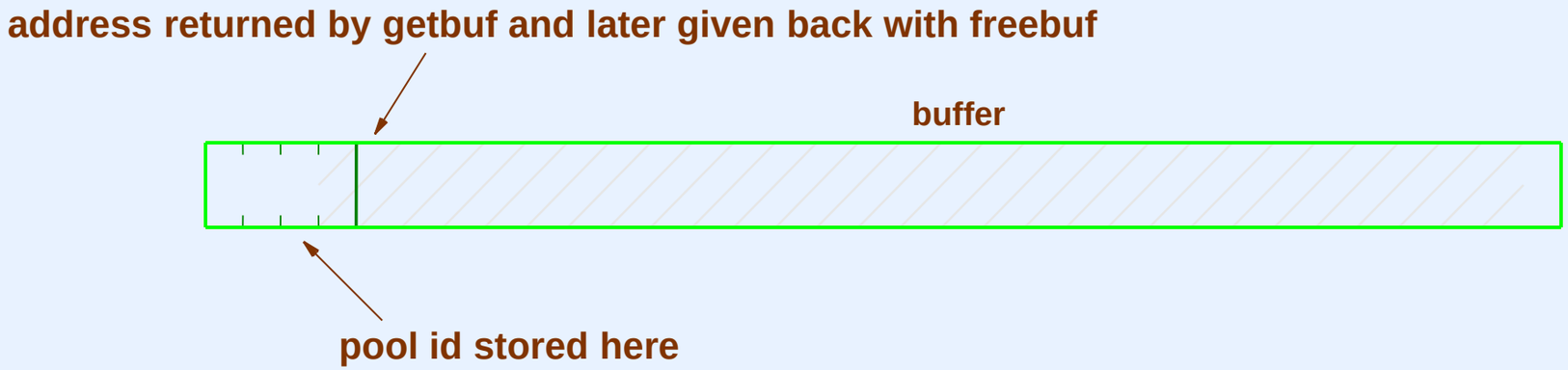
Finding The Pool To Which A Buffer Belongs

- Obvious possibilities
  - Search the table of buffer pools to find the correct pool
  - Use an external data structure to map a buffer address to the correct pool (e.g., keep a list of allocated buffers and the pool to which each belongs)
- An alternative
  - Have *getbuf* pass the caller two values: a pool ID and a buffer address
  - Have *freebuf* take two arguments: a pool ID and a buffer address
- Unfortunately, using two arguments
  - Is inconvenient for programmers
  - Does not work well in C

Solving The Single Pointer Problem

- Xinu uses a clever trick to avoid passing two values
  - Use the address of the lowest usable byte as a buffer identifier
  - Store a pool ID along with each buffer, but hide it from the user
- The implementation
  - When allocating a buffer, allocate enough extra bytes to hold the pool 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
- A process can use a buffer without knowing that the extra bytes exist

Illustration Of A Pool ID Stored With A Buffer



- Xinu allocates four bytes more than the user specifies
- Conceptually, the additional bytes precede the buffer, and are used to store the ID of the buffer pool
- *Getbuf* returns a single pointer to the data area of the buffer (beyond the extra bytes)
- *Freebuf* expects the same pointer that *getbuf* returns to a caller
- The pool ID is transparent to applications using the buffer pool

Potential Downsides Of The Xinu Scheme

- Some device hardware requires a buffer to start on a page boundary, but adding four bytes to the size may ruin alignment
- If the pool id is accidentally overwritten, the buffer will either be returned to the wrong pool or an error will occur because the pool ID is invalid

Buffer Pool Operations

- Create a pool (*mkpool*)
  - Increase the requested buffer size by 4 to hold a pool ID
  - Use *getmem* to allocate memory for all the buffers that will be in the pool
  - Form a singly-linked list of the buffers (storing links in the buffers themselves)
  - Allocate a semaphore to count buffers
  - Return an ID of the allocated buffer pool
- Allocate a buffer from a pool (*getbuf*)
  - Take the pool ID as an argument, and use it locate the correct buffer pool
  - *Wait* on the semaphore associated with a pool (i.e., block until a buffer is available)
  - Extract a buffer from the free list, insert the ID, and return the buffer to the caller

Buffer Pool Operations
(continued)

- Free (deallocate) a previously-allocated buffer (*freebuf*)
  - Extract the pool ID from the extra bytes that precede the buffer
  - Use the pool ID to locate the buffer pool
  - Insert the buffer at the head of the list for the pool
  - Signal the semaphore associated with the pool

Buffer Pool Definitions

```
/* bufpool.h */

#ifndef NBPOOLS
#define NBPOOLS 20              /* Maximum number of buffer pools       */
#endif

#ifndef BP_MAXB
#define BP_MAXB 8192            /* Maximum buffer size in bytes         */
#endif

#define BP_MINB 8               /* Minimum buffer size in bytes         */
#ifndef BP_MAXN
#define BP_MAXN 2048            /* Maximum number of buffers in a pool  */
#endif

struct  bpentry {               /* Description of a single buffer pool  */
        struct  bpentry *bpnext;/* Pointer to next free buffer          */
        sid32   bpsem;          /* Semaphore that counts buffers        */
                                /*    currently available in the pool   */
        };

extern  struct  bpentry buftab[];/* Buffer pool table                   */
extern  bpid32  nbpools;        /* Current number of allocated pools    */
```

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;
        }
```

Xinu Mkbufpool (Part 2)

```
        /* Round request to a multiple of 4 bytes */
        bufsiz = ( (bufsiz + 3) & (~3) );
        /* Increase buffer size to include a pool ID */

        bufsiz += sizeof(bpid32);
        buf = (char *)getmem( numbufs * bufsiz );
        if ((int32)buf == SYSERR) {
                restore(mask);
                return (bpid32)SYSERR;
        }
        poolid = nbpools++;
        bpptr = &buftab[poolid];
        bpptr->bpnext = (struct bpentry *)buf;
        if ( (bpptr->bpsem = semcreate(numbufs)) == SYSERR) {
                freemem(buf, numbufs * bufsiz );
                nbpools--;
                restore(mask);
                return (bpid32)SYSERR;
        }
        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;
}
```

Xinu Getbuf (Part 1)

```
/* 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 buffer from pool */

        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;
}
```

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;
}
```

Virtual Memory

Definition Of Virtual Memory

- An abstraction of physical memory
- It separates a process's view of memory from underlying hardware
- Primarily used with applications (user processes)
- Provides each application process with an address space that is independent of
  - Physical memory size
  - A position in physical memory
  - Isolated from other process's address spaces
- Many mechanisms have been proposed and used

General Approach

- Typically used with a heavyweight process
  - The process appears to run in an isolated address space
  - All addresses are *virtual*, meaning that each process has an address space that starts at address zero
- The operating system
  - Establishes policies for memory use
  - Creates a separate virtual address space for each process
  - Configures the hardware as needed
- The underlying hardware
  - Dynamically translates from virtual addresses to physical addresses
  - Provides support to help the operating system make policy decisions

A Virtual Address Space

- Can be smaller than the physical memory
  * Example: a 32-bit computer with more than 2^{32} bytes (four GB) of physical memory
- Can be larger than the physical memory
  * Example: a 64-bit computer with less than 2^{64} bytes (16 million terabytes) of memory
- Historic note: on early computers, physical memory was larger. Then, virtual memory was larger until physical memory caught up. Now, the move to 64-bit architectures means 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 pieces of the address space to main memory as needed
  - Write pieces back to disk to create space in memory for other pieces
- Questions
  - How much of a process's address space should reside in memory?
  - When should a particular piece be loaded into memory?
  - When should a piece be written back to disk?

Approaches That Have Been Used

- *Swapping*
  - Transfer an entire process's address space (all code, data, and stack) to memory when selecting a process to run
  - Write the entire address space back to disk when switching to another process
- *Segmentation*
  - Divide the image into large "segments" (e.g., make the code and data for each function a segment)
  - Transfer a segment to memory as needed (e.g., when the function is called)
- *Paging*
  - Divide image into small, fixed-size pieces called *pages*
  - Transfer a page to memory when referenced

Approaches That Have Been Used
(continued)

- *Segmentation with paging*
  - Divide an image into very large segments (e.g., a module with multiple functions)
  - Further subdivide each segment into fixed-size pages
- Notes
  - The programming language community favors some form of segmentation
  - Hardware engineers favor paging

A Widely-Used Approach

- Paging has emerged as the most widely used approach for virtual memory because
  - Choosing a reasonable page size (e.g., 4K bytes) makes the paging overhead reasonable for most applications
  - Using a page size that is a power of two enables the hardware to be extremely efficient

Choosing a page size that is a power of two makes it possible to build extremely efficient address mapping hardware.

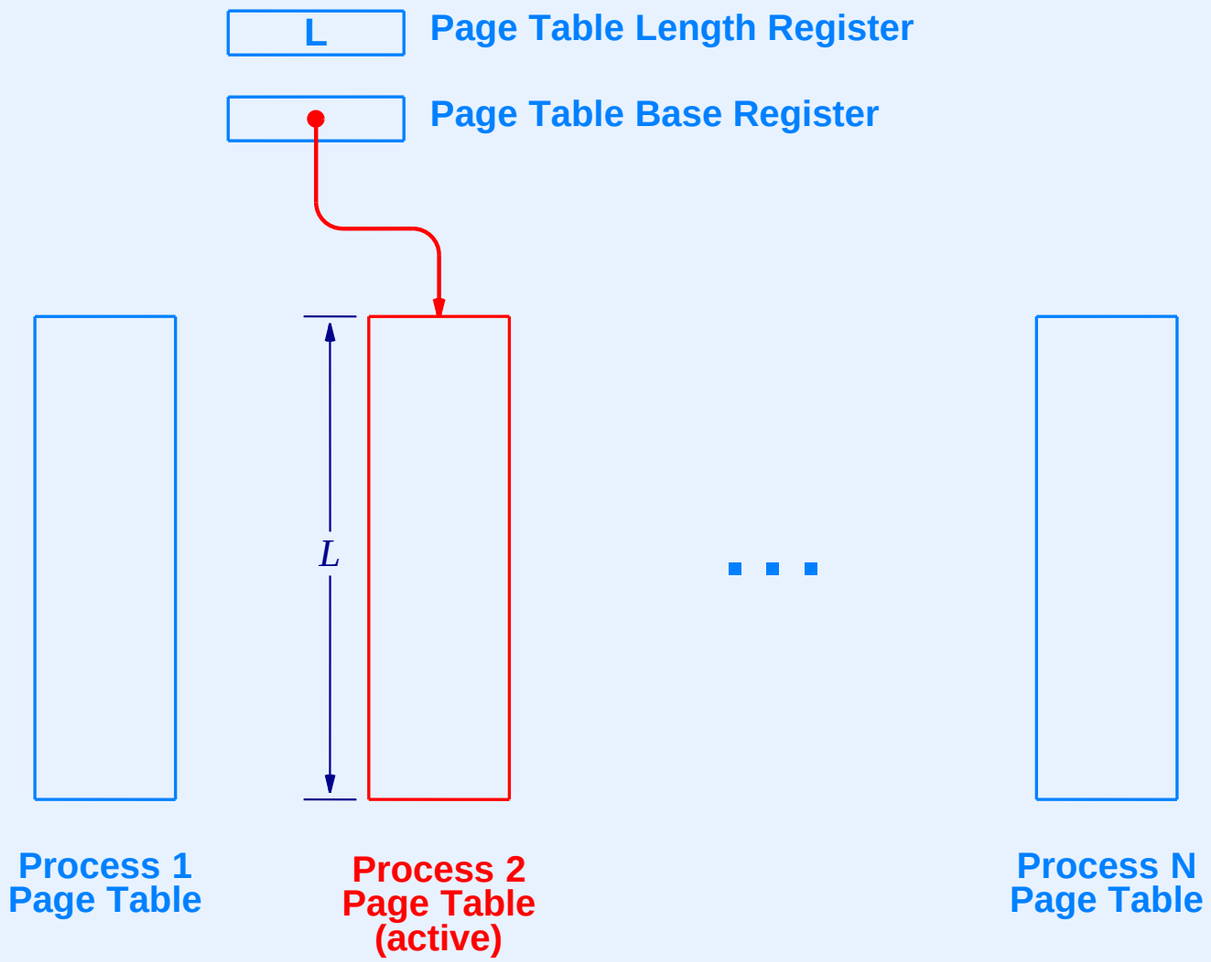
Hardware Support For Paging

- Page tables
  - The operating system allocates one page table per process
  - The location at which a page table is stored depends on the hardware
    * Typical: store page tables in kernel memory
    * Specialized: store page tables in *Memory Management Unit (MMU)* hardware
- A page table base register
  - Internal to the processor
  - Specifies the location of the page table currently being used (i.e., the page table for the current process)
  - Must be changed during a context switch

Hardware Support For Paging
(continued)

- A page table length register
  - Internal to the processor
  - Specifies the number of entries in the current page table
  - Can be changed during context switch if the size of the virtual address space differs among processes
  - Can be used to limit the size of a process's virtual address space

Illustration Of VM Hardware Registers



- Only one page table is active at a given time (the page table for the current process)

Address Translation

- A key part of virtual memory
- Refers to the translation from the virtual address a process uses to the corresponding physical memory address
- Is performed by memory management hardware
- Must occur on *every* memory reference
- A hardware unit performs the translation

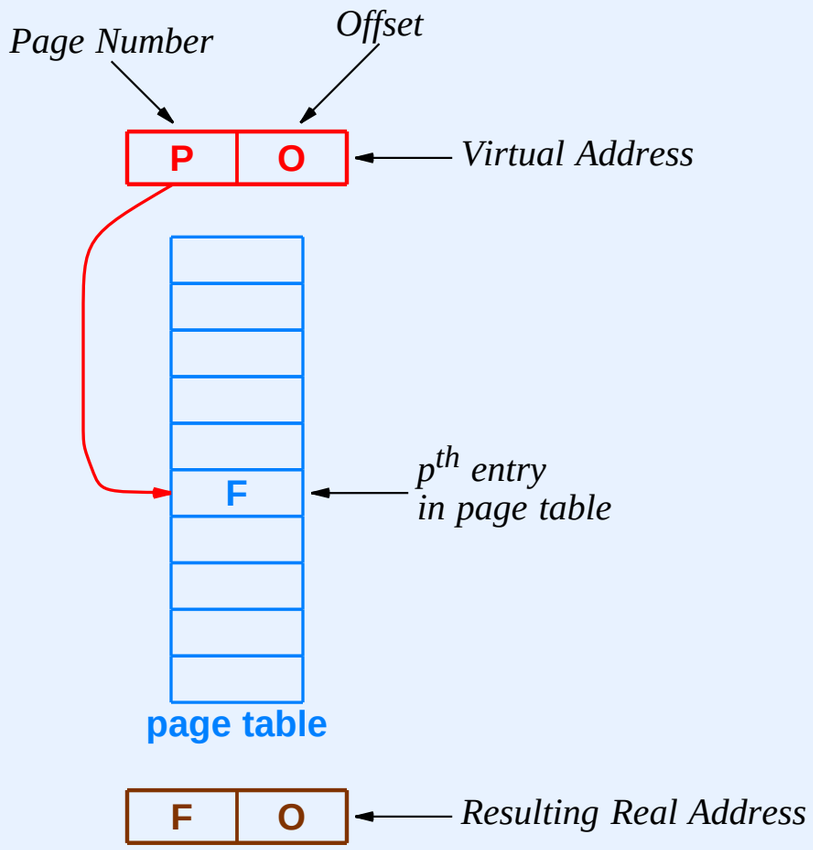
Address Translation With Paging

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

Virtual And Physical Addresses

- *Address translation* maps a virtual address to a physical address
- To make hardware efficient
  - Choose a page size that is a power of 2
  - Use the upper bits in a virtual address as a page number, *P*
  - Use the lower bits in a virtual address as an offset into the page, *O*
- To map an address
  - Extract the page number, *P*
  - Use *P* as an index into the page table array and find the frame where the page currently resides in memory
  - Add the offset, *O*, to get the physical address of the byte being referenced

Illustration Of Address Translation



- Each page table entry contains a physical frame address
- Choosing a page size to be a power of 2 means hardware can perform translation without using multiplication, division, or modulus operations

In Practice

- The size of virtual space may be limited to physical memory size
- Some hardware offers separate page tables for text, data, and stack segments
  - The chief disadvantage: extra complexity
  - The advantage: the three can operate independently
- The kernel address space can also be virtual (but it hasn't worked well in practice)

Page Table Sizes For 32 and 64 Bit Computers

- For a 32-bit address space where each page is 4 Kbytes
  - There are 2^{20} page table entries of 4 bytes per entry
  - The total page table size for one process: 4 Mbytes
- For a 64-bit address space where each page is 4 Kbytes
  - There are 2^{52} page table entries of 4 bytes per entry
  - The total page table size for one process: 16,777,216 Gbytes!
- Conclusion: we cannot have complete page tables for a 64-bit address space

Paging In A 64-Bit System

- To reduce page table size, use multiple levels of page tables
  - The high-order bits of an address form an index into the top-level page table
  - The next bits form an index into the second-level page table (but only a few second-level page tables are defined)
- Key idea: only the lowest and highest pieces of the address space need to be mapped (text, data, bss, and heap at the bottom, and stack at the top)
- The same technique *can* be applied to 32-bit address spaces to reduce page table size

The Concept Of Demand Paging

- Keep the entire memory image of each process on secondary storage
- Treat main memory as cache of recently-referenced pages
- Copy a page into memory dynamically when the page is referenced
- Copy a page from the secondary store to a frame in main memory on demand (when the page is referenced)
- When a frame is needed for a newly-referenced page, move one of the pages currently in memory back to its place on secondary storage

The Importance Of Hardware Support For Virtual Memory

- Every memory reference must be translated from a virtual address to a physical address, including
  - The address of an instruction as well as data
  - Branch addresses computed as a *jump* instruction executes
  - Indirect addresses that are generated at runtime
- 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!
  - To fetch the instruction
  - To fetch each operand
  - To follow indirect references
  - To store results
- On hardware that supports a memory copy instruction, one instruction can reference *multiple* pages
- The point: hardware support is needed to perform high-speed address translation for each of the above

Hardware Support For Address Mapping

- In addition to normal address translation, a special-purpose hardware unit further speeds page lookup and makes paging practical
- The special hardware unit
  - Is called a *Translation Look-aside Buffer* (*TLB*)
  - Is implemented with T-CAM
- A TLB caches most recent address translations and returns translations quickly
- Good news: many applications tend to make repeated references to the same page (i.e., a high locality of reference), so a TLB works well

Mappings In a TLB And Context Switch

- Facts
  - Each process has an address space that starts at zero
  - Each process has its own page zero
  - The location of page 0 in memory may differ among processes, and page 0 from some processes may not even be in memory
- Consequence: address translation must change when switching context from one process to another
- The point: the mappings cached in a TLB will not remain valid when switching context from one process to another

How An Operating System Manages A TLB

- When it switches context from one process to another, an operating system must ensure that the old mappings in the TLB are not used
- On some hardware, the operating system flushes the TLB to remove all current entries
- On other hardware, tags are used to distinguish among address spaces
- Tags used in a TLB
  - A unique tag is assigned to each process by the OS (typically, the process ID)
  - The operating system tells the VM hardware which tag to use
  - When placing a mapping in the TLB, the hardware appends the current tag to the address
  - When searching the TLB, the hardware appends the current tag to the address
  - Advantage: the OS only needs to change the tag when switching context

Can Page Tables Be Paged?

- On some hardware, yes
- Store all page tables in memory
- Lock the current page table to avoid paging it
- The current thinking about paging page tables
  - It introduces extra overhead
  - Lookup becomes less efficient
  - Large memory sizes make it impractical

Bits That Record Page Status

- Each page table entry contains status bits that are understood by the hardware
- At least three status bits are needed

Name	Meaning
Use Bit	Set by hardware whenever the page is referenced (fetch or store)
Modify Bit	Set by hardware whenever a store operation changes data on the page
Presence Bit	Set by OS to indicate whether the page is resident in memory

Page Replacement

- The hardware
  - Generates a *page fault* exception when a referenced page is not resident
  - The operating system handles the exception
- The operating system
  - Allocates a frame in physical memory
  - Retrieves the needed page from secondary storage (allowing other processes to execute while page is being fetched)
  - Once the page arrives, marks the page table entry to indicate the page is now resident
  - Restarts the process that caused the page fault

Researchers Have Studied Many Aspects Of Paging

- Which replacement policies are most effective?
- Which pages from a given address space 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 should apply to a shared library?

A Critical Trade-off For Demand Paging

- For a given process, paging represents delay; from a system perspective, paging is merely overhead
- Paging overhead and latency for a given process can be reduced by giving the process more physical memory (more frames)
- However, processor utilization and overall throughput can be increased by increasing the level of multiprogramming (i.e., by having more concurrent processes ready to run when one of them blocks to wait for I/O or some other reason)
- Extremes
  - Paging is minimized when the current process has maximal memory
  - Throughput is maximized when all ready processes are resident
- Researchers considered the question, “What is the best tradeoff?”

Frame Allocation

- When a page fault occurs, the operating system must obtain a frame to hold the page
- If a frame is currently unused, the selection is trivial — select the unused frame
- If all frames are currently occupied by pages from various processes, the operating system must
  - Select one of the resident pages and save a copy on disk
  - Mark the page table entry to indicate that the page is no longer resident
  - Select the frame that has been vacated
  - Obtain the page that caused the page fault, and fill in the appropriate page table entry to point to the frame
- Question: which frame should be selected when all are in use?

Choosing A Frame

- Researchers have studied
  - Global competition: when choosing a frame, include resident pages from *all* processes in the selection
  - Local competition: when choosing a frame for process P, select from among the other pages that process P has resident
- Researchers have also studied various policies
  - *Least Recently Used* (*LRU*)
  - *Least Frequently Used* (*LFU*)
  - *First In First Out* (*FIFO*)
- In the end, a basic approach has been adopted: *global clock*

The Global Clock Algorithm

- Originated in the MULTICS operating system
- Allows all processes to compete with one another (hence the term *global*)
- Has relatively low overhead
- Has become the most popular practical method

Global Clock Paradigm

- The clock algorithm is activated when a page fault occurs
- It searches through all frames in memory, and selects a frame to use
- The term *clock* is used because the algorithm starts searching where it left off the last time
- A frame containing a referenced page is given a “second chance” before being reclaimed
- A frame containing a modified page is given a “third chance” before being reclaimed
- In the worst case: the clock sweeps through all frames twice before reclaiming one
- Advantage: the algorithm does *not* require any external data structure other than the standard page table bits

Operation Of The Global Clock

- The clock uses a global pointer that picks up where it left off previously
  - It sweeps through all frames in memory
  - It only starts moving when a frame is needed
  - It stops moving once a frame has been selected
- During the sweep, the algorithm checks *Use* and *Modify* bits of each frame
- It reclaims the frame if the *Use*/*Modify* bits are *(0,0)*
- It changes *(1,0)* into *(0,0)* and bypasses the frame
- It changes *(1,1)* into *(1,0)* and bypasses the frame
- The algorithm keeps a copy of the actual modified bit to know whether a page has actually changed since it was read from secondary storage (i.e., is *dirty*)

In Practice

- A global clock is usually configured to reclaim a small set of frames when one is needed
- The reclaimed frames are cached for subsequent references
- Advantage: collecting multiple frames means the clock will run less frequently

A Problem With Paging: Thrashing

- Imagine a large set of processes each referencing their pages at random
- At first, free frames in memory can be used to hold pages
- Eventually, the frames in memory fill up, and *each* new reference causes a page fault, which results in
  - Choosing a frame (the clock algorithm runs)
  - Writing the existing page to secondary storage (disk I/O)
  - Fetching a new page from secondary storage (more disk I/O)
- The processor spends most of the time paging and waiting for disk I/O, so little computation can be performed
- We use the term *thrashing* to describe fetching a new page often
- Having a large memory on a computer helps avoid thrashing

The Importance/Unimportance Of Paging Algorithms

- Facts
  - At one time, page replacement algorithms were the primary research topic in operating systems
  - Sophisticated mathematical analysis was done to understand their behavior
  - By the 1990s, interest in page replacement algorithms faded
  - Now, almost no one uses complex replacement algorithms
- Why did the topic fade?
- Was the problem completely solved?
- Answer: physical memories became so large that very few systems need to replace pages
- A computer scientist once quipped that paging only works if systems don’t page

Summary

- We considered two forms of high-level memory management
- Inside the kernel
  - Define a set of abstract resources
  - Isolate the memory used by each subsystem to prevent interference
  - Use a buffer pool mechanism (although a buffer has a length and a location, our mechanism allows both values to be referenced by a single address)
- Outside the kernel
  - Choose swapping, segmentation, or demand paging

Summary
(continued)

- Demand paging is the most popular VM technology
  - It uses fixed size pages (typically 4K bytes)
  - A page is brought into memory when referenced
- The global clock algorithm is widely used for page replacement

Questions?