Operating Systems

14. Memory: Virtual Memory 2

Hyunchan, Park

http://oslab.chonbuk.ac.kr

Division of Computer Science and Engineering

Chonbuk National University

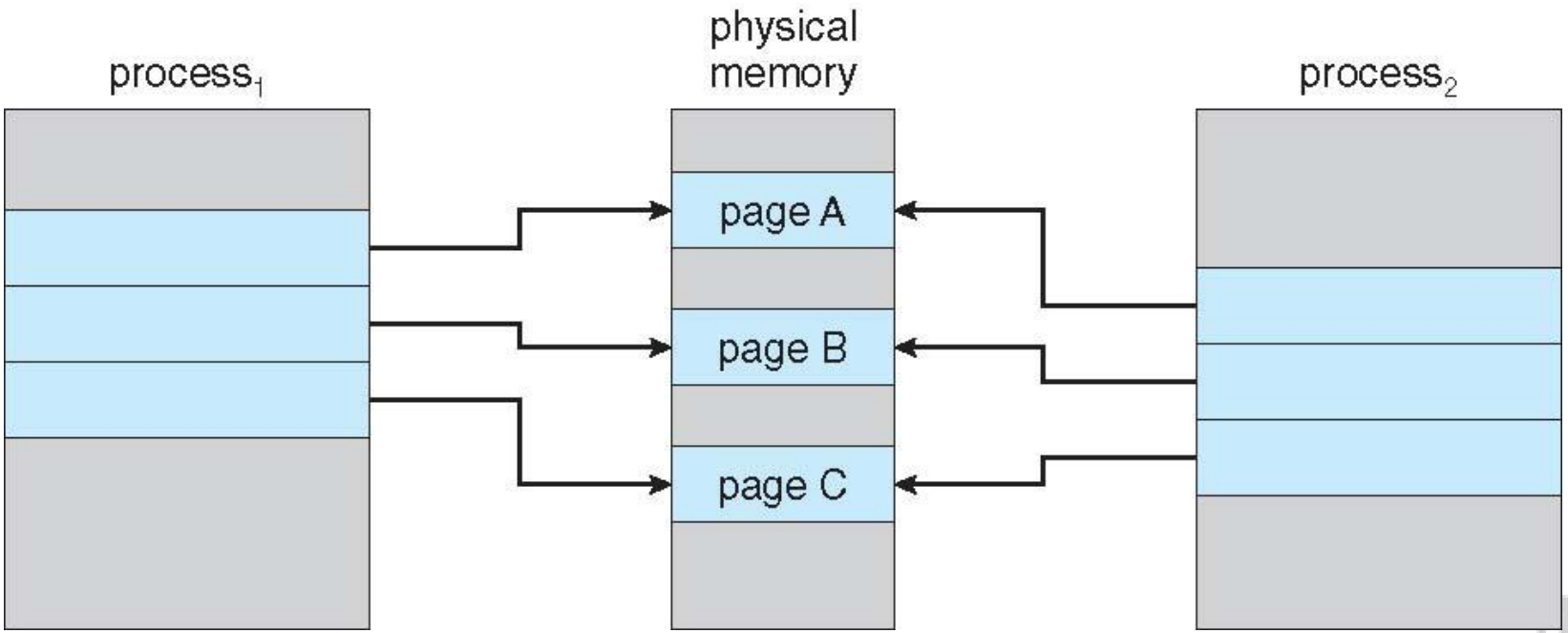
Contents

- Copy-on-Write
- Memory-Mapped Files
- Allocating Kernel Memory
  - Buddy system
  - Slab allocator
- Other Considerations
- Operating-System Examples

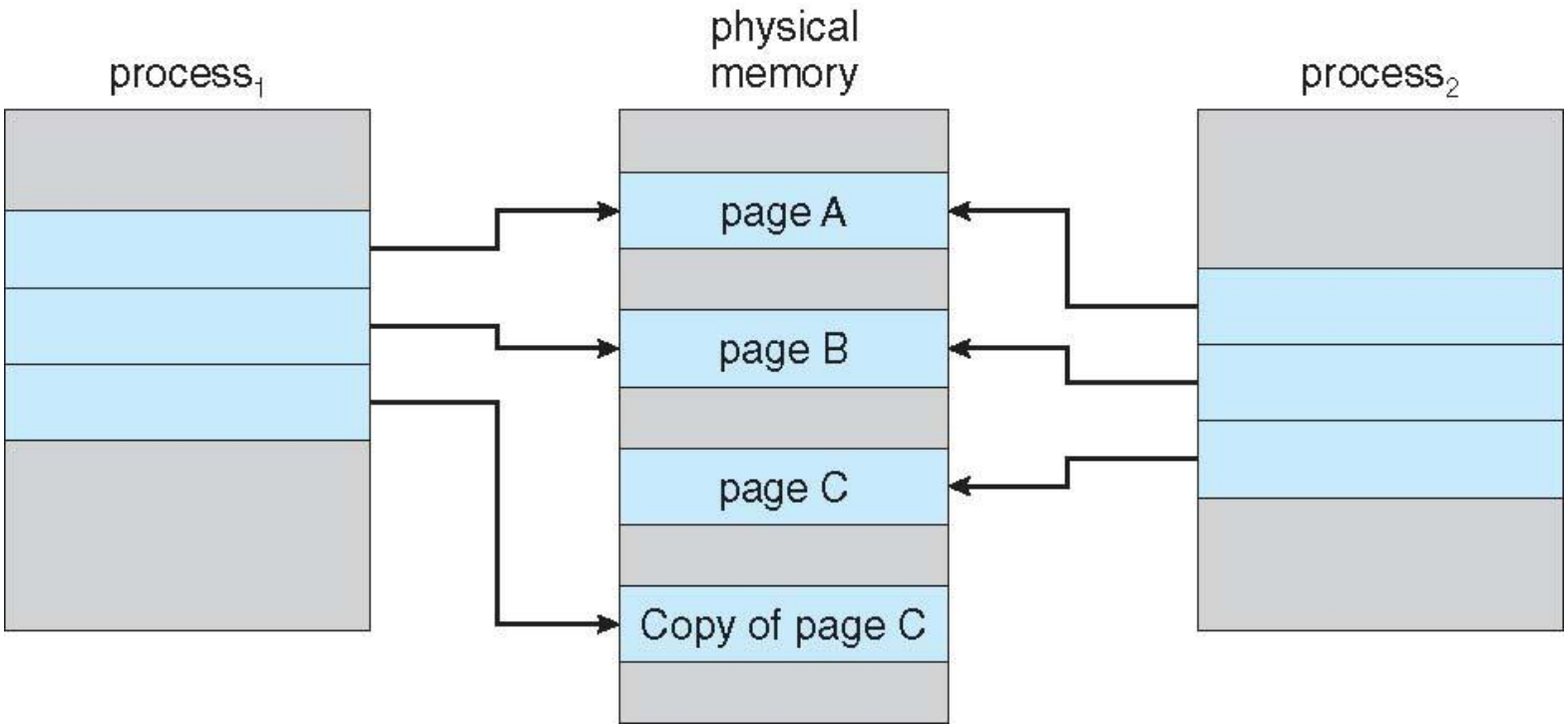
Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a pool of zero-fill-on-demand pages
  - Pool should always have free frames for fast demand page execution
    - Don’t want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



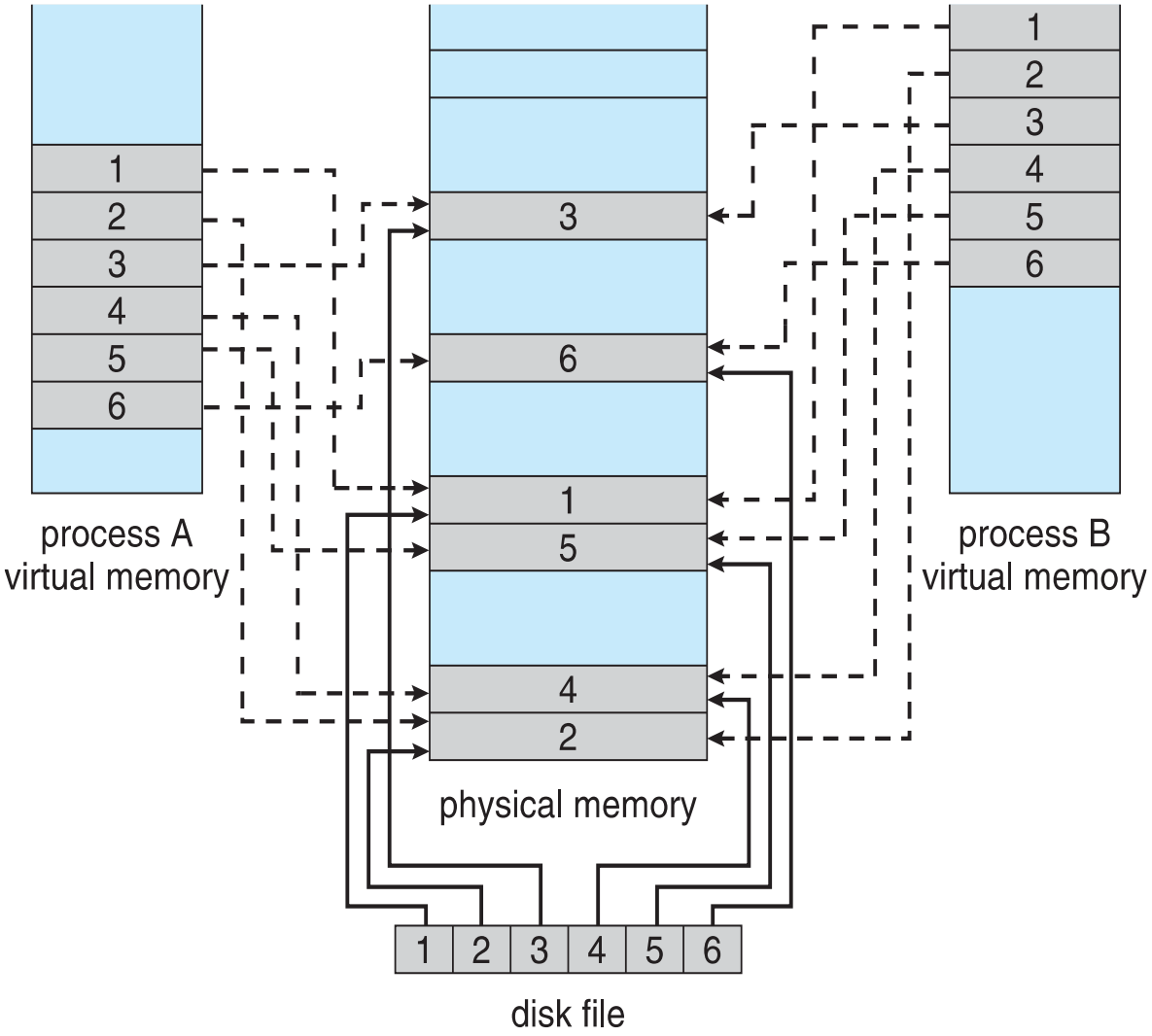
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file close() time
  - For example, when the pager scans for dirty pages

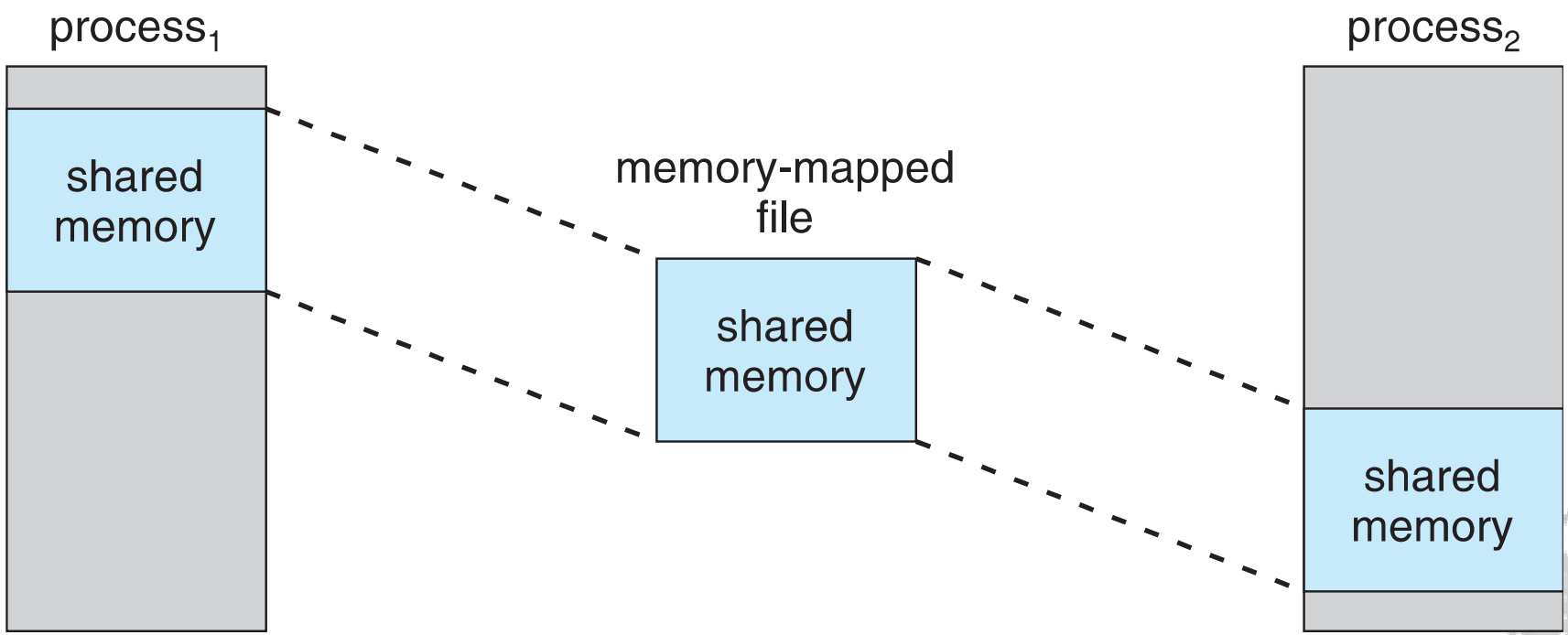
Memory-Mapped File Technique for all I/O

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via mmap() system call
  - Now file mapped into process address space
- For standard I/O (open(), read(), write(), close()), mmap anyway
  - But map file into kernel address space
  - Process still does read() and write()
    - Copies data to and from kernel space and user space
  - Uses efficient memory management subsystem
    - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)

Memory Mapped Files



Shared Memory via Memory-Mapped I/O



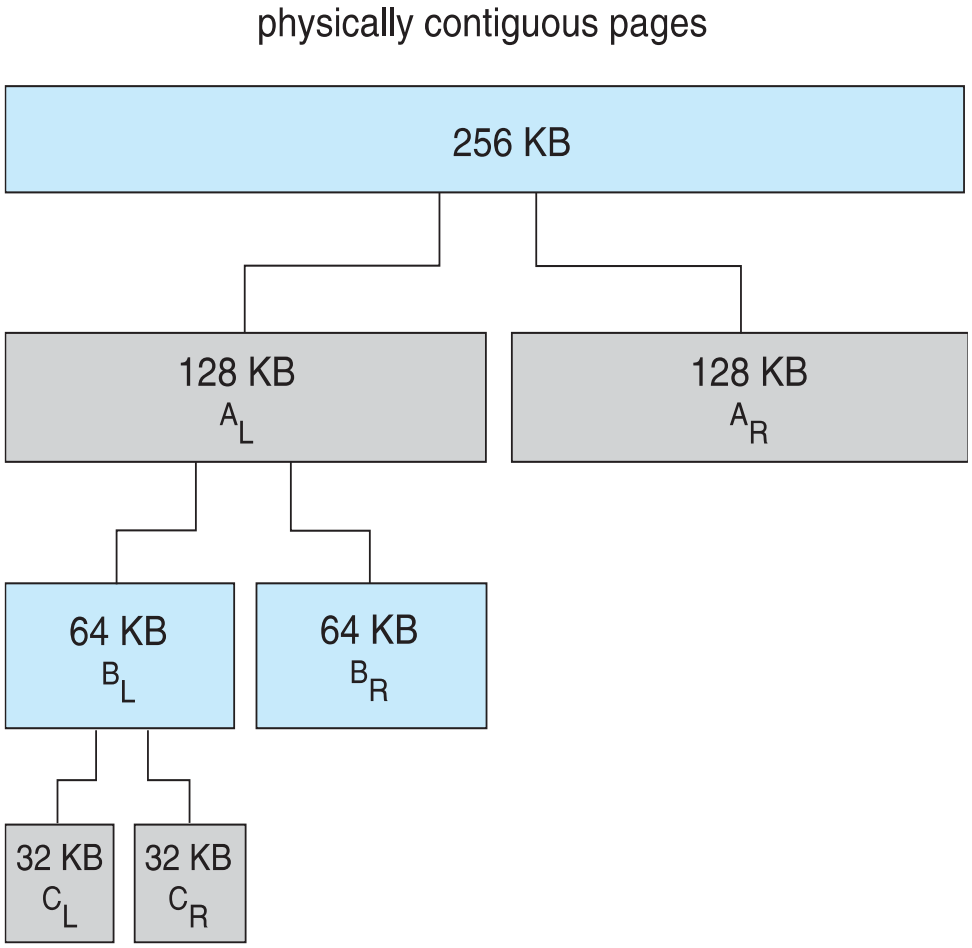
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O
- To efficiently use the kernel memory
  - In aspect of space: buddy system
  - In aspect of performance: slab allocator

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using power-of-2 allocator
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

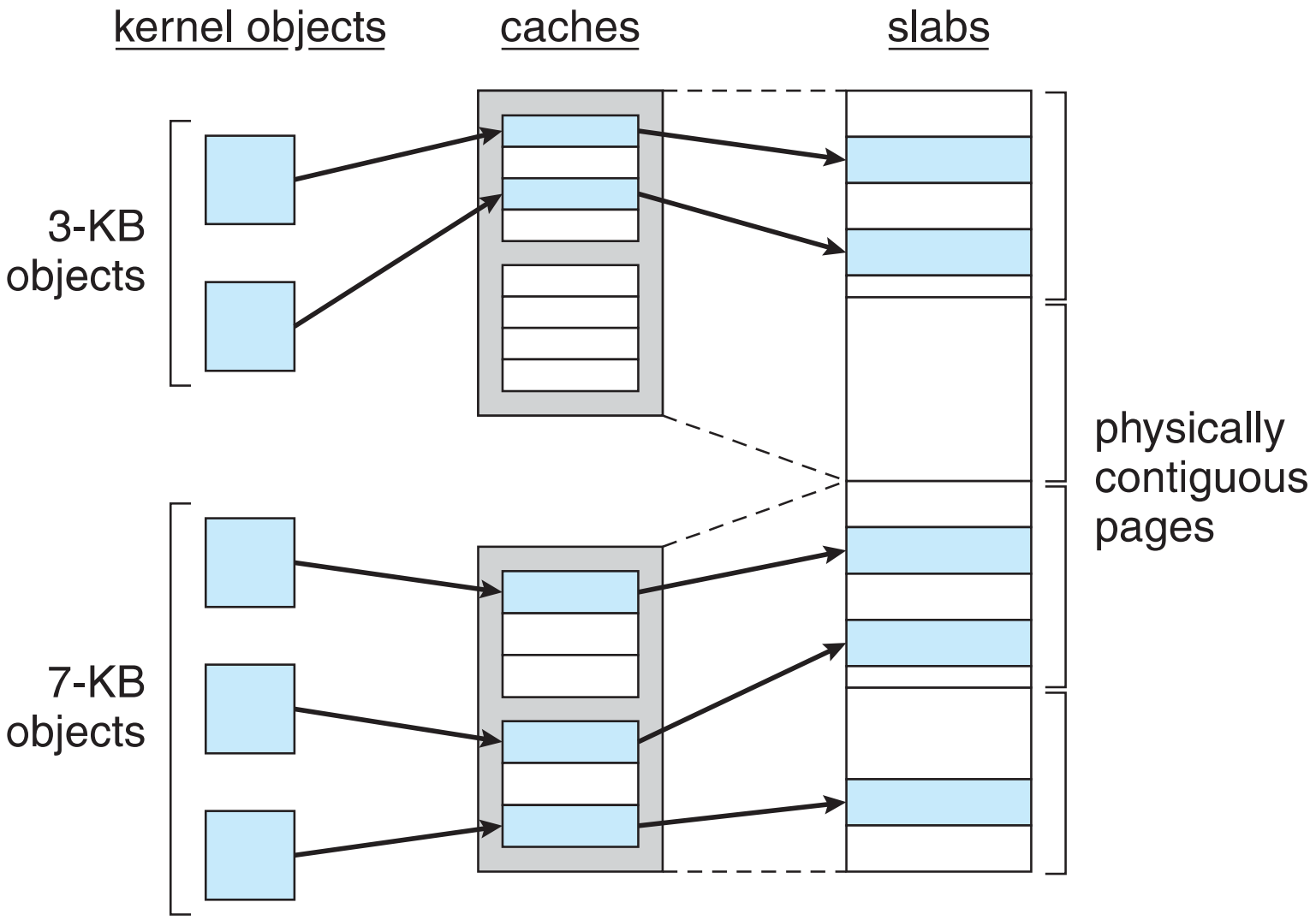
Buddy System Allocator



Slab Allocator

- Slab is one or more physically contiguous pages
- Cache consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with objects – instantiations of the data structure
- When cache created, filled with objects marked as free
- When structures stored, objects marked as used
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Slab Allocator in Linux

- For example process descriptor is of type struct task_struct
  - Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free struct task_struct
- Slab can be in three possible states
  - Full – all used
  - Empty – all free
  - Partial – mix of free and used
- Upon request, slab allocator
  - Uses free struct in partial slab
  - If none, takes one from empty slab
  - If no empty slab, create new empty

Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

Other Considerations -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume s pages are prepaged and α of the pages is used
    - Is cost of s * α save pages faults > or < than the cost of prepaging s * (1- α) unnecessary pages?
    - α near zero ⇒ prepaging loses

Other Issues – Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - Resolution
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

Other Issues – Program Structure

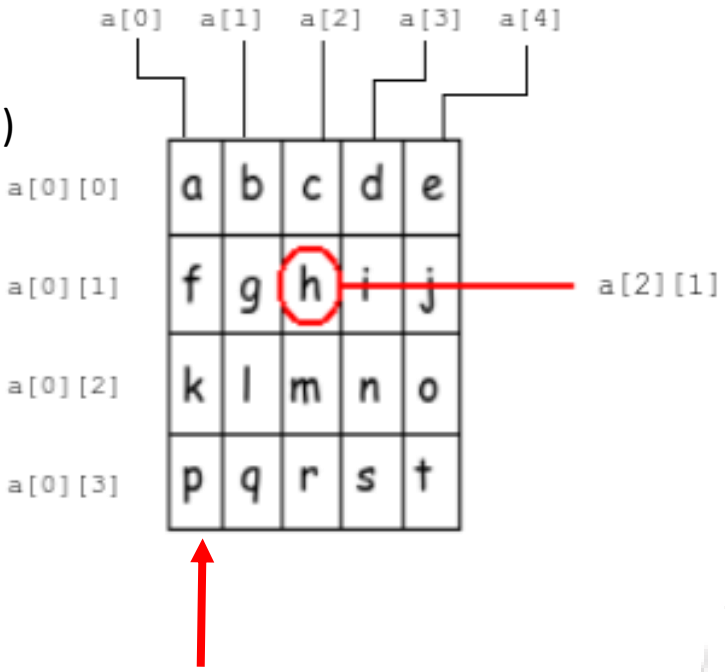
- Program structure w/ 128 free frames
  - Int [128,128] data;
  - Each row is stored in one page (page size = 512B)
  - Program 1

```
for (j = 0; j <128; j++)
  for (i = 0; i < 128; i++)
      data[i,j] = 0;
```

  - 128 x 128 = 16,384 page faults
  - Program 2

```
for (i = 0; i < 128; i++)
  for (j = 0; j < 128; j++)
      data[i,j] = 0;
```

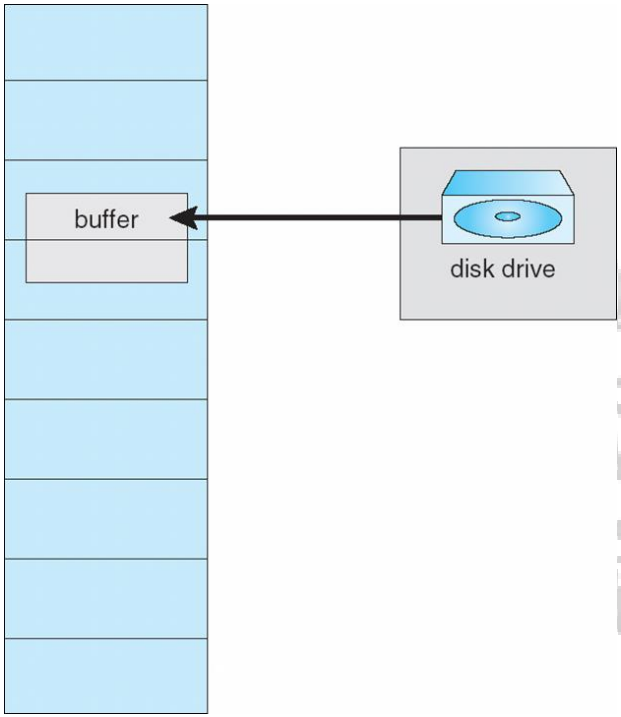
  - 128 page faults



a page

Other Issues – I/O interlock

- I/O Interlock – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- Pinning of pages to lock into memory



Other Issues - Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are NUMA – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating lgroups
    - Structure to track CPU / Memory low latency groups
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup

Operating System Examples

- Windows

- Solaris

Windows

- Uses demand paging with clustering. Clustering brings in pages surrounding the faulting page
- Processes are assigned working set minimum and working set maximum
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, automatic working set trimming is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Solaris

- Maintains a list of free pages to assign faulting processes
- Lotsfree – threshold parameter (amount of free memory) to begin paging
- Desfree – threshold parameter to increasing paging
- Minfree – threshold parameter to being swapping
- Paging is performed by pageout process
- Pageout scans pages using modified clock algorithm
- Scanrate is the rate at which pages are scanned. This ranges from slowscan to fastscan
- Pageout is called more frequently depending upon the amount of free memory available
- Priority paging gives priority to process code pages

Solaris 2 Page Scanner