BILKENT UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING
CS342 OPERATING SYSTEMS

LECTURE
VIRTUAL MEMORY
(CHAPTER 9)

Dr. İbrahim Körpeoğlu
http://www.cs.bilkent.edu.tr/~korpe

References

- *The slides here are adapted/modified from the textbook and its slides: Operating System Concepts, Silberschatz et al., 7th & 8th editions, Wiley.*

REFERENCES

- Operating System Concepts, 7th and 8th editions, Silberschatz et al. Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum, 3rd edition, 2009.

Outline

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

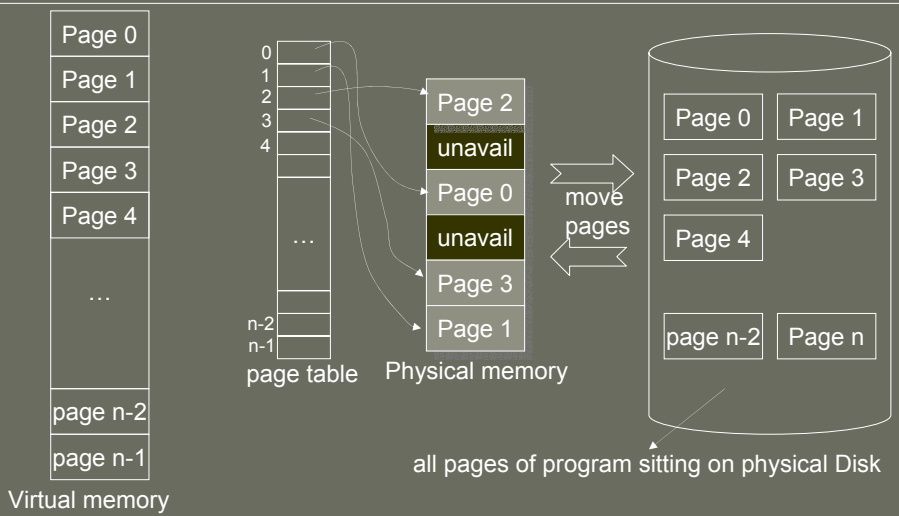
Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

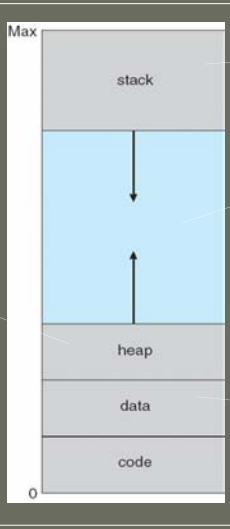
Background

- **Virtual memory** – separation of user logical memory from physical memory.
- Benefits:
  - Only part of the program needs to be in memory for execution
    - You can execute more programs concurrently
  - Logical address space can therefore be much larger than physical address space
    - You can execute programs larger than physical memory
  - Allows address spaces to be shared by several processes
    - Library or memory segment can be shared
  - Allows for more efficient process creation

Virtual Memory That is Larger Than Physical Memory



A typical virtual-address space layout of a process



function parameters; local variables; return addresses

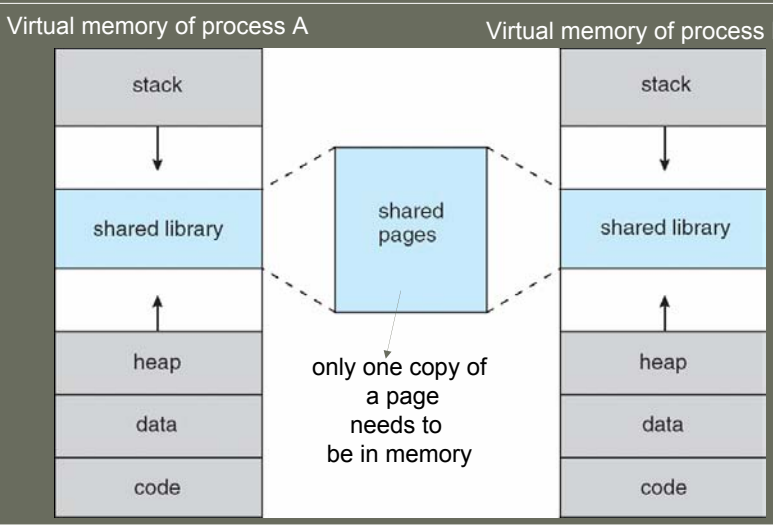
unused address space

will be used whenever needed

malloc() allocates space from here (dynamic memory allocation)

global data (variables)

Shared Library Using Virtual Memory



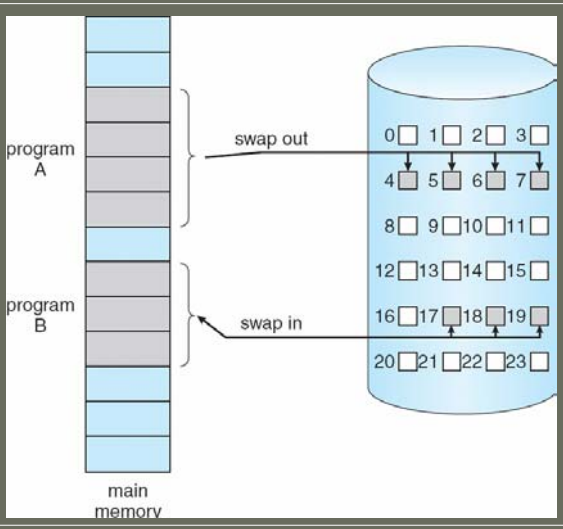
Implementing Virtual Memory

- Virtual memory can be implemented via:
  - Demand paging
    - Bring pages into memory when they are useda, i.e. allocate memory for pages when they are used
  - Demand segmentation
    - Bring segments into memory when they are used, i.e. allocate memory for segments when they are used.

Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

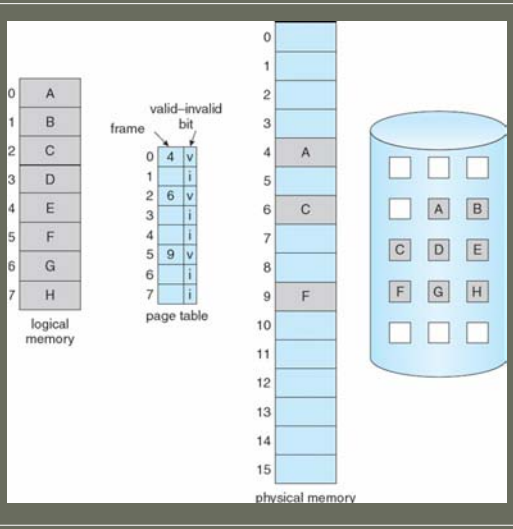
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

page table

- During address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



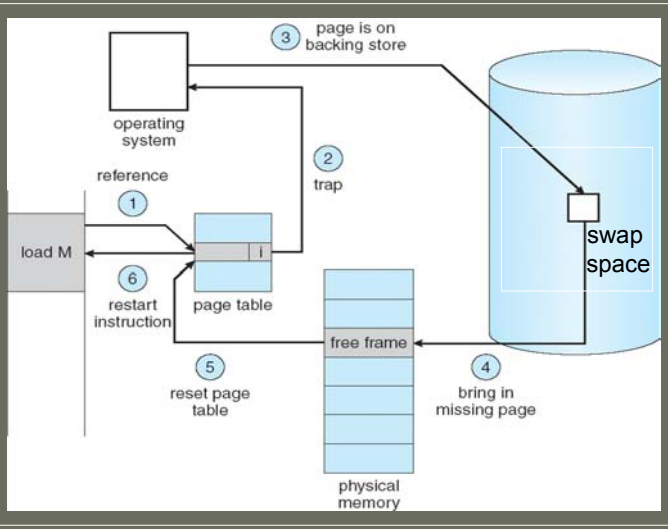
Page Fault

- While CPU is executing an instruction or trying to fetch an instruction: if there is a reference to a page, *first* reference to that page will trap to operating system (since page will not be in memory): this is called **page fault**

Page fault handling steps by OS:

1. Operating system looks at another table to decide:
   - Invalid reference (page is in *unused* portion of address space) ⇒ abort
   - Just not in memory (page is in used portion, but not in RAM)
2. Get empty frame
   (we may need to remove a page; if removed page is modified, we need disk I/O to swap it out)
3. Swap page into frame
   (we need disk I/O)
4. Reset tables (install mapping into page table)
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

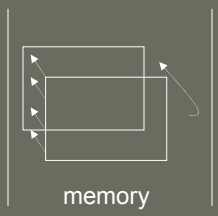
Steps in Handling a Page Fault



Page Fault (Cont.)

- If page fault occur when trying to fetch an instruction, fetch the instruction again after bringing the page in.
- If page fault occurs while we are executing an instruction: Restart the instruction after bringing the page in.

- For most instructions, restarting the instruction is no problem.
- But for some, we need to be careful. Example:
  - block move instruction



Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time to Memory (EAT)

EAT = (1 – *p*) x memory access
+ *p* (page fault overhead
+ swap page out
+ swap page in
+ restart overhead
)

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault (p = 1/1000), then
  EAT = 8.2 microseconds.

This is a slowdown by a factor of 40!!
(200 ns / 8.2 microsec ~= 1/40)

Process Creation

- Virtual memory allows other benefits during process creation:

  - Copy-on-Write

  - Memory-Mapped Files (later)

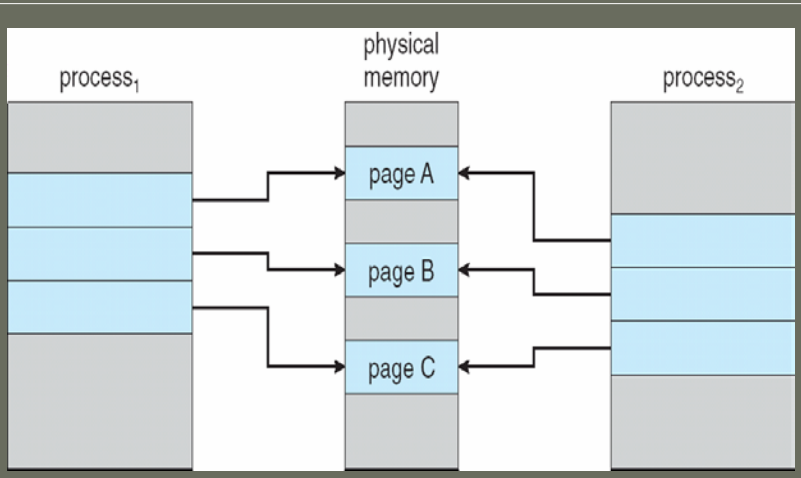
Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

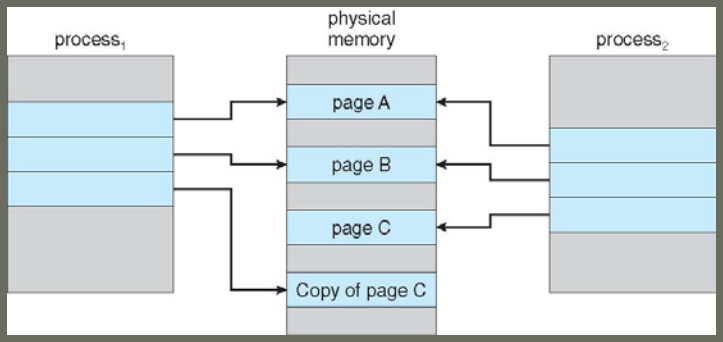
  If either process modifies a shared page, only then is the page copied

- COW allows more efficient process creation as only modified pages are copied

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



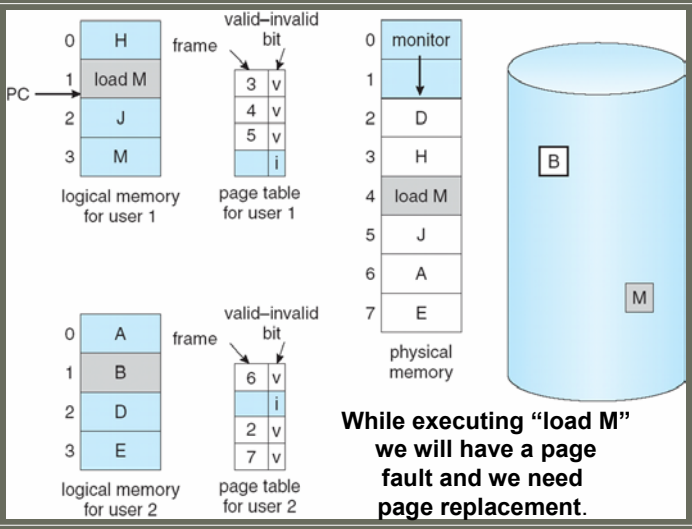
What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - Algorithm ? Which page should be remove?
  - performance – want an algorithm which will result in minimum number of page faults
- With page replacement, same page may be brought into memory several times

Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk while removing/replacing a page.
- Page replacement completes separation between logical memory and physical memory
  - large virtual memory can be provided on a smaller physical memory

Need For Page Replacement



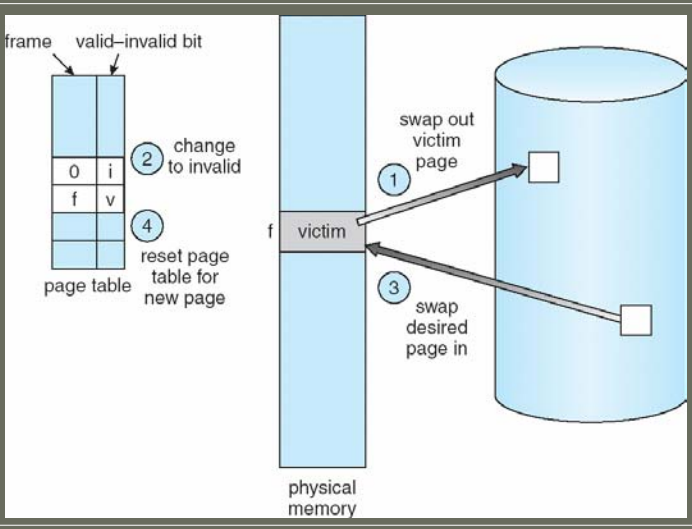
While executing "load M" we will have a page fault and we need page replacement.

Basic Page Replacement

Steps performed by OS while replacing a page upon a page fault:

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame; if the victim page is modified, write it back to disk.
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Restart the process

Page Replacement



Page Replacement Algorithms

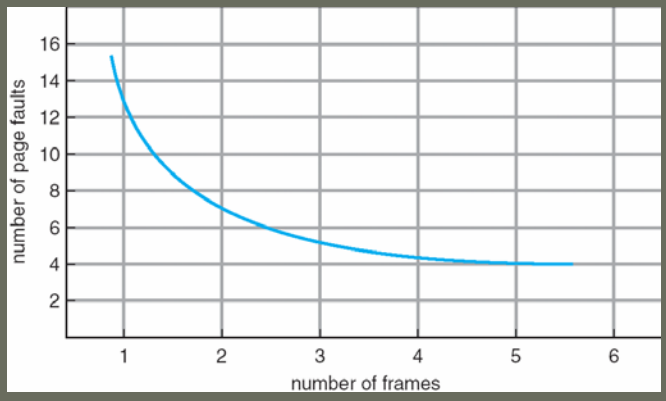
- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Driving reference string

- Assume process makes the following memory references in a system with 100 bytes per page:
- 0100 0432 0101 0612 0102 0103 0104 0101 0611 0102 0103 0104 0101 0610 0102 0103 0104 0609 0102 0105
- Pages referenced with each memory reference
  - 0, 4, 1, 6, 1, 1, 1, 1, 6, 1, 1, 1, 1, 6, 1, 1, 6, 1, 1
- Corresponding page reference string
  - 0, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

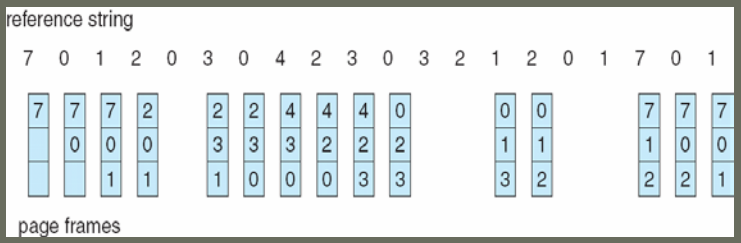
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)
- 4 frames

1	1	4	5		
2	2	1	3	9 page faults	
3	3	2	4		

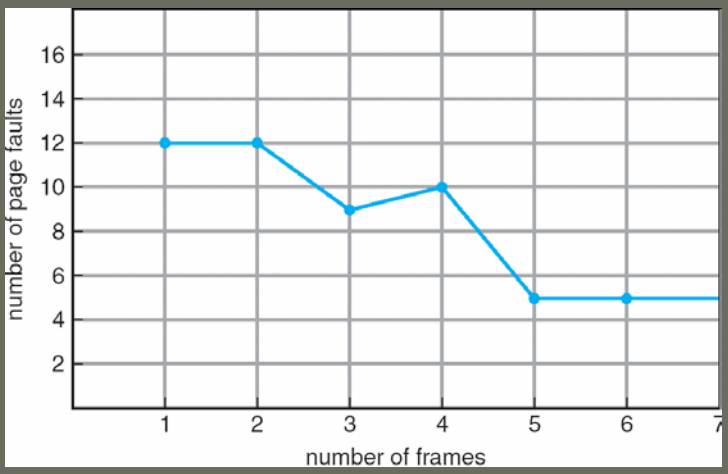
1	1	5	4		
2	2	1	5	10 page faults	
3	3	2			
4	4	3			

- Belady's Anomaly: more frames ⇒ more page faults

FIFO Page Replacement



FIFO Illustrating Belady's Anomaly



Optimal Algorithm

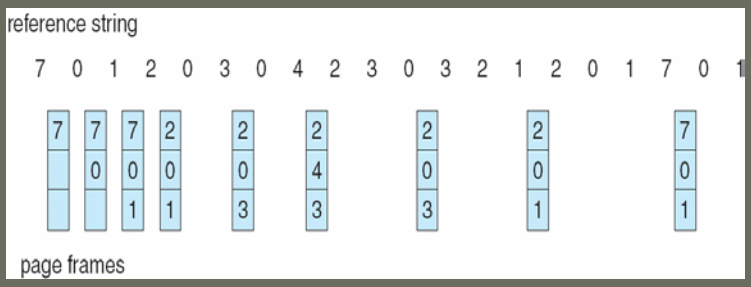
- Replace page that will not be used for longest period of time
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	4
2	6 page faults
3	
4	5

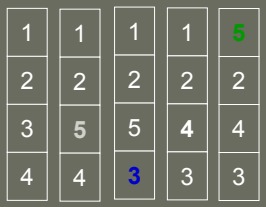
- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement



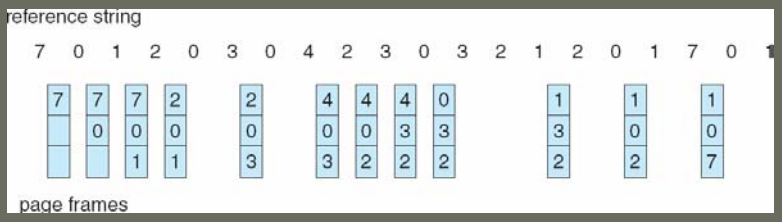
Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



8 page faults

LRU Page Replacement



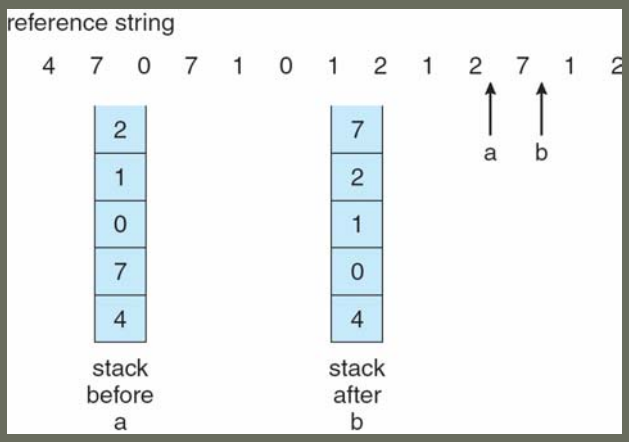
LRU Algorithm Implementation

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be replaced, look at the counters to determine which one to replace
    - The one with the smallest counter value will be replaced

LRU Algorithm Implementation

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed (with every memory reference; costly)
  - No search for replacement (replacement fast)

Use of a Stack to Record The Most Recent Page References

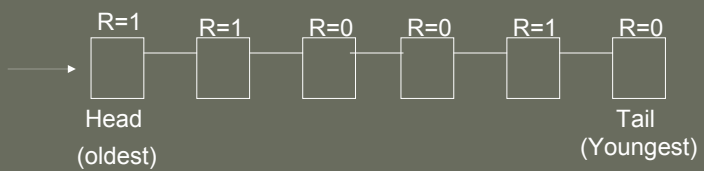


LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0 (not referenced/used)
  - When page is referenced, bit set to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however (several pages may have 0 value)
  - Reference bits are cleared periodically (with every clock interrupt);

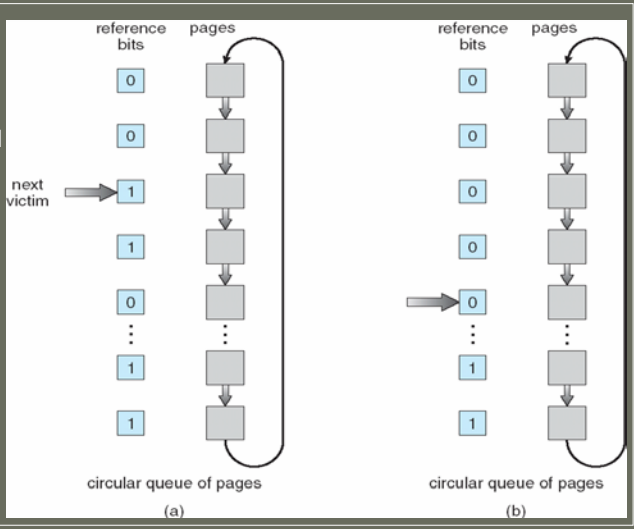
Second change algorithm

- Second chance
  - FIFO that is checking if page is referenced or not
  - Need reference bit
  - If page to be replaced, look to the FIFO list; remove the page close to head of the list and that has reference bit 0.
  - If there is a page you encounter that has reference bit 1, move it to the back after clearing the reference bit. Try to find another page that has 0 as reference bit.
  - May require to change all 1's to 0's and then come back to the beginning of the queue.



Second-Chance (clock) Page-Replacement Algorithm

Second chance can be implemented using a circular list of pages; Then it is also called Clock algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm**: replaces page with smallest count
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process
  - $s_i =$ size of process p_i
  - $S = \sum s_i$
  - $m =$ total number of frames
  - $a_i =$ allocation for $p_i = \frac{s_i}{S} \times m$

Example:

$$m = 64$$
$$s_i = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

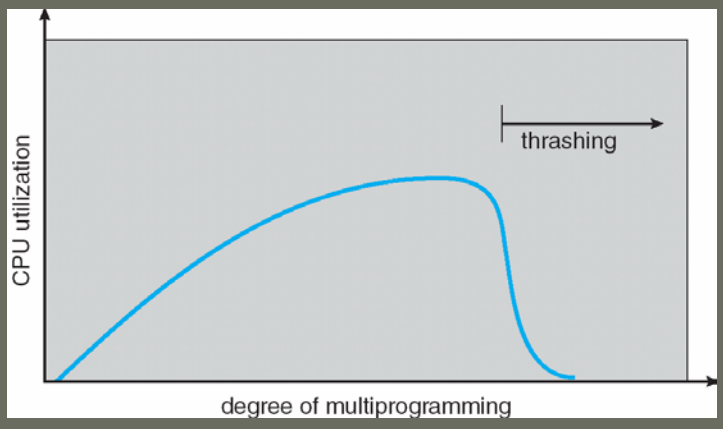
Global versus Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
- **Local replacement** – each process selects from only its own set of allocated frames

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out

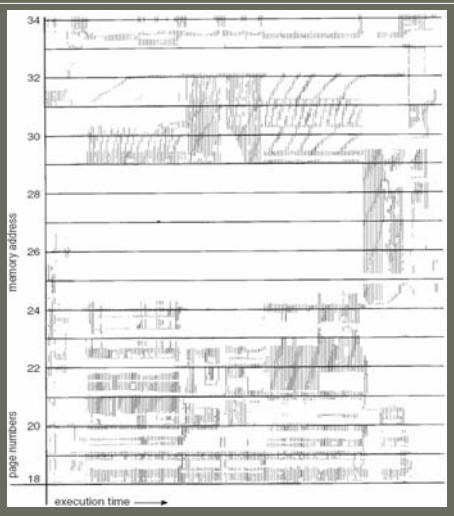
Thrashing (Cont.)



Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  Σ size of locality > total memory size

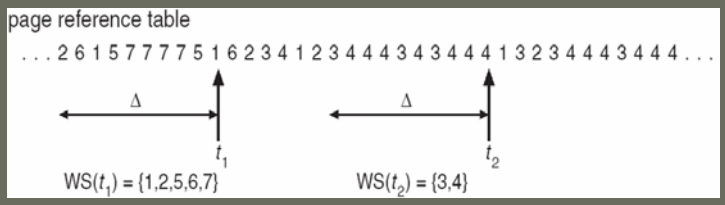
Locality In A Memory-Reference Pattern



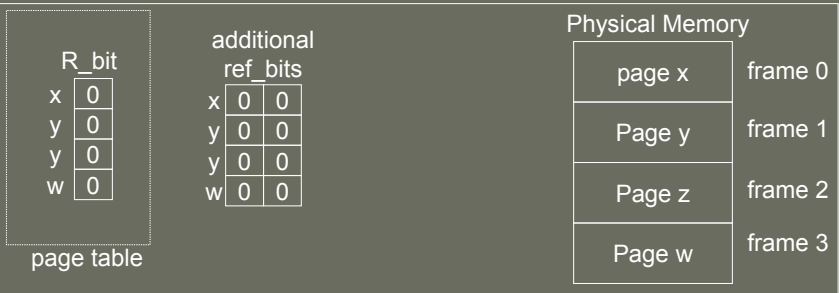
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand for frames
- if $D > m \Rightarrow$ Thrashing (m: #frames in memory)
- Policy if $D >$ m, then suspend one of the processes

Working-Set Model



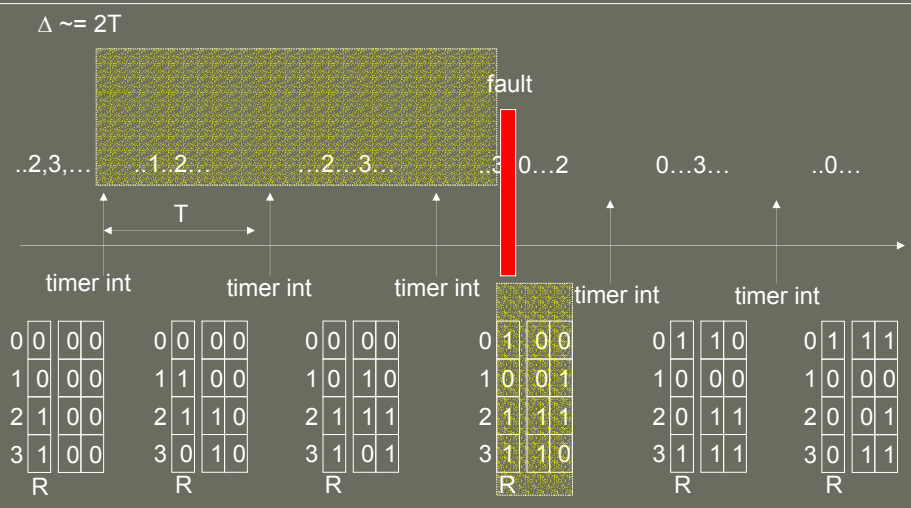
Keeping Track of Working-Set



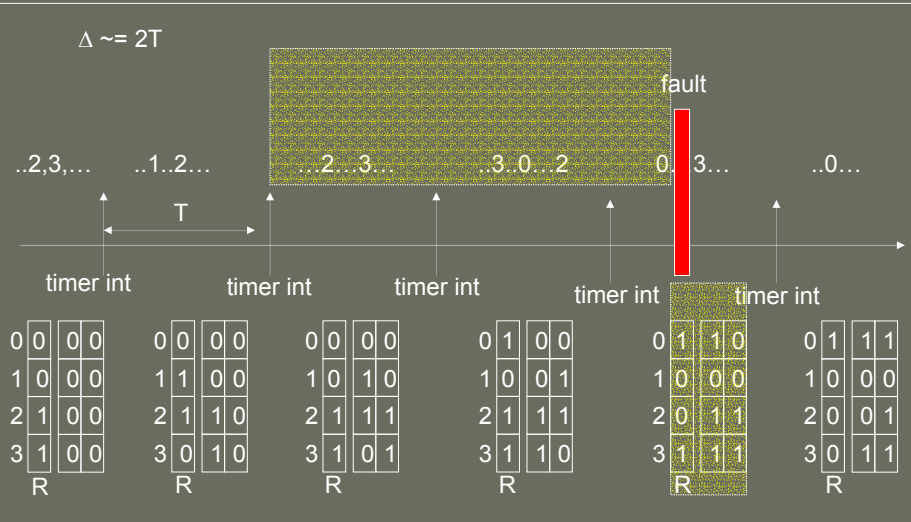
Keeping Track of Working-Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000 (time units)
- Timer interrupts after every 5000 time units
- Keep in memory 2 bits for each page
- Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
  - Granularity is 5000 time_units (we don't know when exactly in it reference has occurred)
- Improvement = 10 bits and interrupt every 1000 time units.

Keeping Track of Working-Set (example)

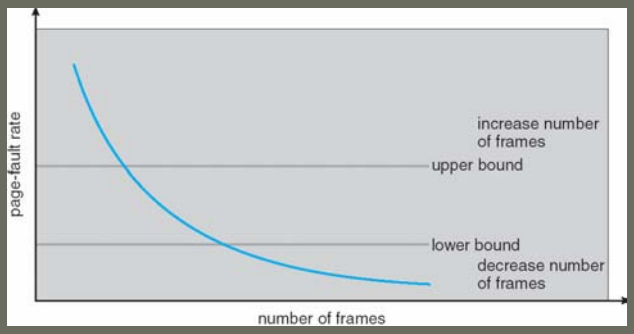


Keeping Track of Working-Set (example continued)

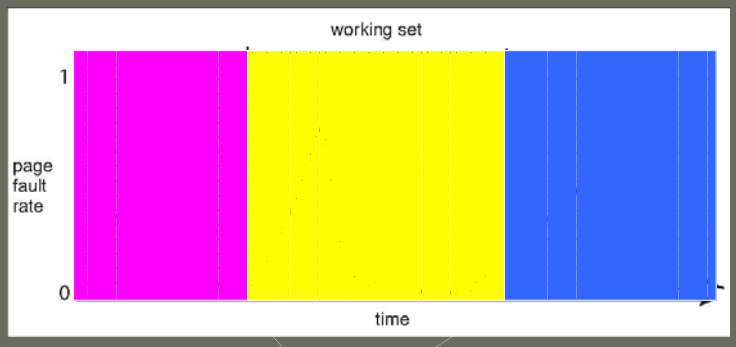


Page-Fault Frequency (PFF) Scheme

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

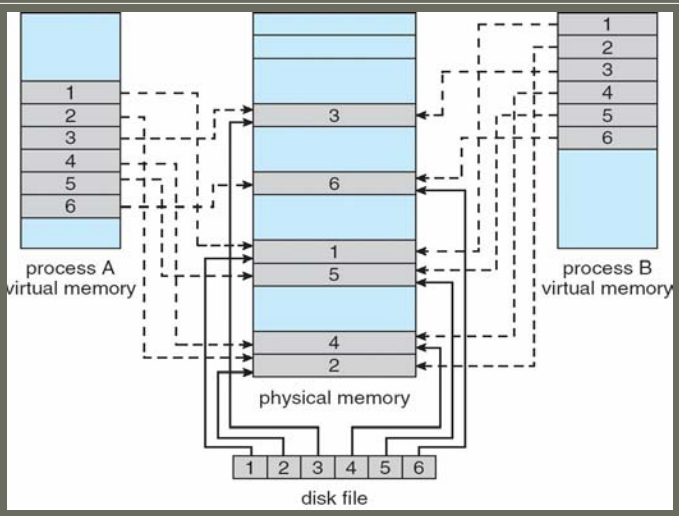


transition from one working set to another

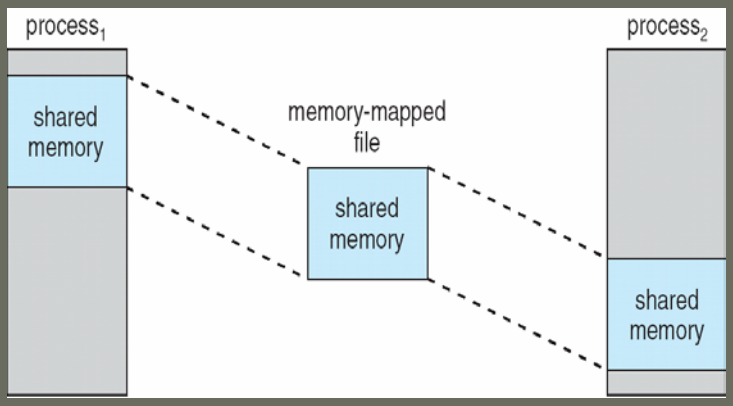
Memory-Mapped Files

- 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 file access by treating file I/O through memory rather than `read()` `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory Mapped Files



Memory-Mapped Shared Memory in Windows



Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures (objects) of varying sizes
    - Process descriptors, semaphores, file objects, ….
  - Those structures have sizes much less than the page size
  - Some kernel memory needs to be contiguous

- This is dynamic memory allocation problem.
- But using first-fit like strategies (heap management strategies) cause external fragmentation

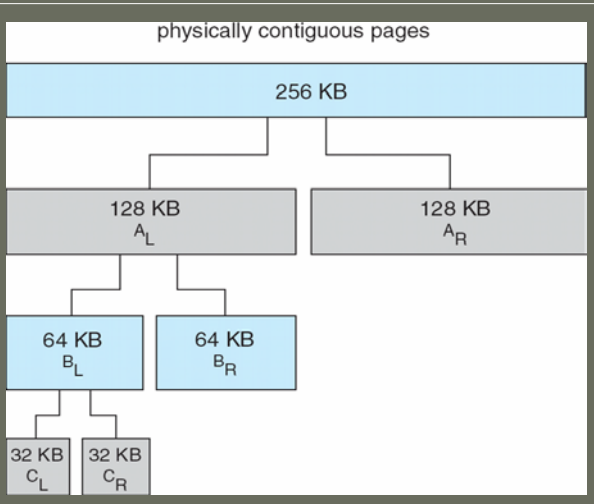
Allocating Kernel Memory

- We will see two methods
  - Buddy System Allocator
  - Slab Allocator

Buddy System Allocator

- 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

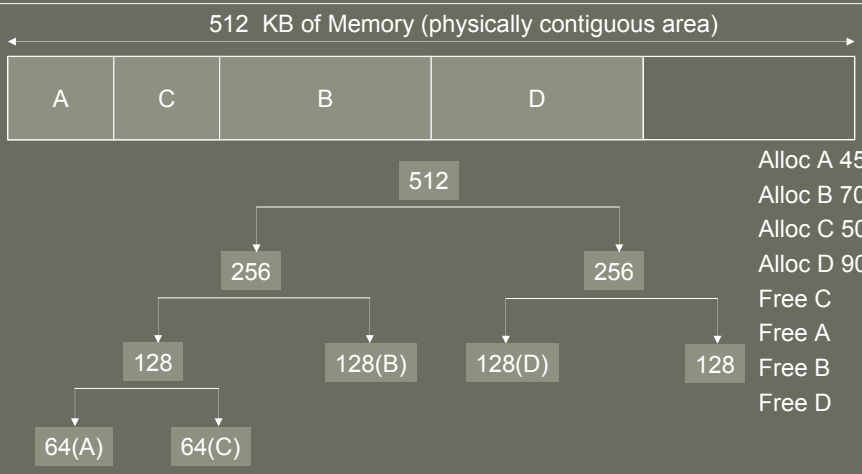
Buddy System Allocator



Example

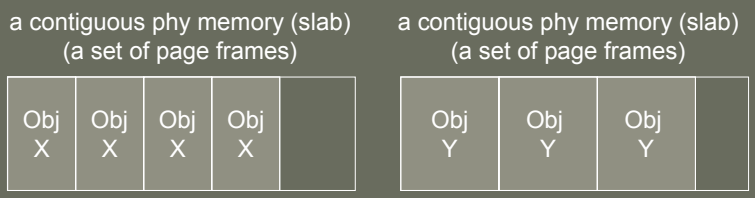
- Object A needs memory 45 KB in size
- Object B needs memory 70 KB in size
- Object C needs memory 50 KB in size
- Object D needs memory 90 KB in size
- Object C removed
- Object A removed
- Object B removed
- Object D removed

Example



Slab Allocator

- Alternate strategy
- Within kernel, a considerable amount of memory is allocated for a finite set of objects such as process descriptors, file descriptors and other common structures
- Idea:

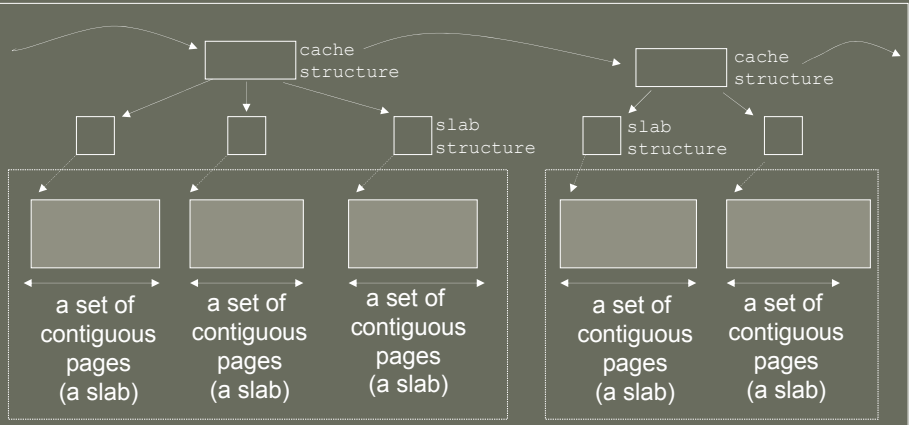


Obj X: object of type X
Obj Y: object of type Y

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 slots (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

Slabs and Caches



Cache structure

- A set of slabs that contain one type of object is considered as a cache.
- Cache structure is a structure that keeps information about the cache and includes pointers to the slabs.

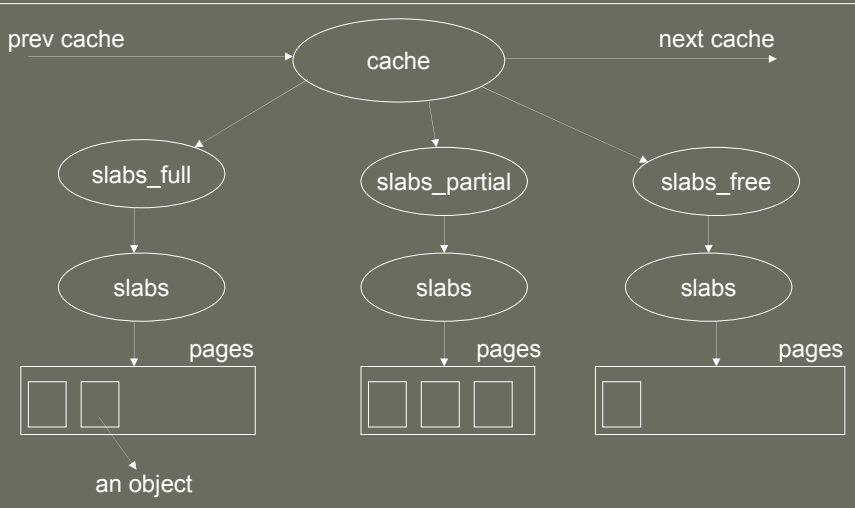
```
struct kmem_cache_s {
        struct list_head slabs_full;      /* points to the full slabs */
        struct list_head slabs_partial;   /* points to the partial slabs */
        struct list_head slabs_free;      /* points to the free slabs */
        unsigned int objsize;             /* size of objects stored in this cache */
        unsigned int flags;
        unsigned int num;
        spinlock_t spinlock;
        …
        …
}
```


Slab structure

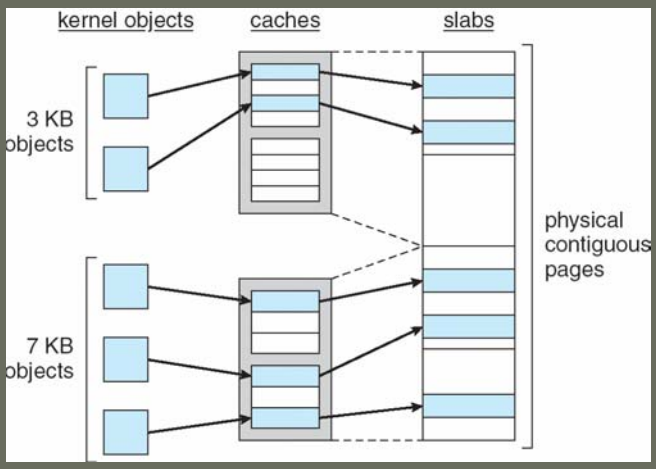
- A slab stucture is a data structure that points to a contiguous set of page frames (a slab) that can store some number of objects of same size.
- A slab can be considered as a set of slots (slot size = object size). Each slot in a slab can hold one object.
- Which slots are free are maintained in the slab structure

```
typedef struct slab_s {
        struct list_head list;
        unsigned long colouroff;
        void *s_mem;            /* start address of first object */
        unsigned int inuse;     /* number of active objects */
        kmem_bufctl_t free;     /* info about free objects */
} slab_t;
```

Layout of Slab Allocator



Slab Allocation



Slab Allocator in Linux

- cat **/proc/slabinfo** will give info about the current slabs and objects

cache names: one cache for each different object type

```
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount> <
sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
ip_fib_alias          15    113     32  113    1 : tunables  120   60    8 : slabdata      1      1      0
ip_fib_hash           15    113     32  113    1 : tunables  120   60    8 : slabdata      1      1      0
dm_tio                 0      0     16  203    1 : tunables  120   60    8 : slabdata      0      0      0
dm_io                  0      0     20  169    1 : tunables  120   60    8 : slabdata      0      0      0
uhci_urb_priv          4    127     28  127    1 : tunables  120   60    8 : slabdata      1      1      0
jbd_4k                 0      0   4096    1    1 : tunables   24   12    8 : slabdata      0      0      0
ext3_inode_cache  128604 128696    504    8    1 : tunables   54   27    8 : slabdata  16087  16087      0
ext3_xattr         24084  29562     48   78    1 : tunables  120   60    8 : slabdata    379    379      0
journal_handle        16    169     20  169    1 : tunables  120   60    8 : slabdata      1      1      0
journal_head          75    144     52   72    1 : tunables  120   60    8 : slabdata      2      2      0
revoke_table           2    254     12  254    1 : tunables  120   60    8 : slabdata      1      1      0
revoke_record          0      0     16  203    1 : tunables  120   60    8 : slabdata      0      0      0
scsi_cmd_cache        35     60    320   12    1 : tunables   54   27    8 : slabdata      5      5      0
….
files_cache           104    170    384   10    1 : tunables   54   27    8 : slabdata     17     17      0
signal_cache          134    144    448    9    1 : tunables   54   27    8 : slabdata     16     16      0
sighand_cache         126    126   1344    3    1 : tunables   24   12    8 : slabdata     42     42      0
task_struct           179    195   1392    5    2 : tunables   24   12    8 : slabdata     39     39      0
anon_vma             2428   2540     12  254    1 : tunables  120   60    8 : slabdata     10     10      0
pgd                    89     89   4096    1    1 : tunables   24   12    8 : slabdata     89     89      0
pid                   170    303     36  101    1 : tunables  120   60    8 : slabdata      3      3      0
```

active objects

size

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

- Page size selection must take into consideration:
  - Fragmentation
    - Small page size reduces fragmentation
  - table size
    - Large page size reduces page table size
  - I/O overhead
    - Large page size reduce I/O overhead (seek time, rotation time)
  - Locality
    - Locality is improved with smaller page size.

Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) x (Page Size)

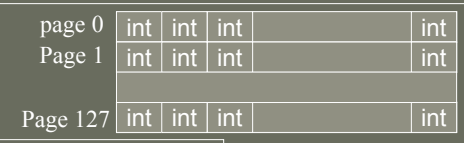
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults

- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size

- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Other Issues – Program Structure

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page assuming pagesize=512 bytes



- 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

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