Virtual Memory (Part II)

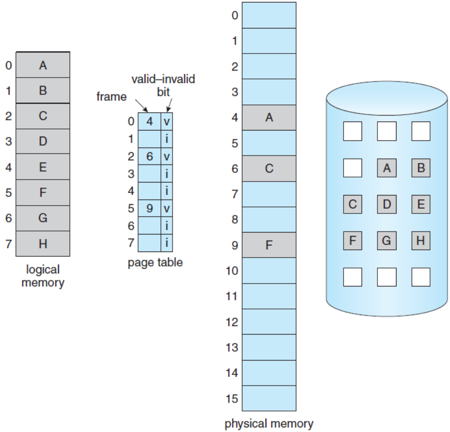
Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Reminder

Reminder

- Partially-loaded programs
- Virtual memory: much larger than physical memory



Reminder

- Demand paging similar to paging + swapping
- Locality
- Page fault
- Page replacement algorithms:
  - FIFO, optimal, LRU, LRU-approximate, counting-based

Allocation of Frames

Allocation of Frames

- How do we allocate the fixed amount of free memory among the various processes?

Allocation of Frames

- How do we allocate the fixed amount of free memory among the various processes?

- If we have 93 free frames and two processes, how many frames does each process get?

Frame Allocation Constraints (1/2)

- The maximum number of frames is the total frames in the system.

Frame Allocation Constraints (1/2)

- The maximum number of frames is the total frames in the system.
- Each process needs minimum number of frames.

Frame Allocation Constraints (1/2)

- The maximum number of frames is the total frames in the system.

- Each process needs minimum number of frames.
  - Example: IBM 370: 6 pages to handle MOVE instruction:
  - Instruction is 6 bytes, might span 2 pages
  - 2 pages to handle from
  - 2 pages to handle to

Frame Allocation Constraints (2/2)

- Why the minimum number of frames for each process?

Frame Allocation Constraints (2/2)

- Why the minimum number of frames for each process? performance

Frame Allocation Constraints (2/2)

- Why the minimum number of frames for each process? performance
- Decreases the number of frames:
  - Increases the page-fault rate

Frame Allocation Constraints (2/2)

- Why the minimum number of frames for each process? performance

- Decreases the number of frames:
  - Increases the page-fault rate

- When a page fault occurs before an executing instruction is complete, the instruction must be restarted.

Frame Allocation Constraints (2/2)

- Why the minimum number of frames for each process? performance

- Decreases the number of frames:
  - Increases the page-fault rate

- When a page fault occurs before an executing instruction is complete, the instruction must be restarted.
  - We must have enough frames to hold all the different pages that any single instruction can reference.

Allocation Schemes

- Fixed allocation

- Priority allocation

Allocation Schemes

- Fixed allocation
  - Equal allocation
  - Proportional allocation

- Priority allocation

Fixed Allocation (1/4)

- Equal allocation

- Split m frames among n processes: $\frac{m}{n}$ frames to each process.

Fixed Allocation (1/4)

- Equal allocation
- Split m frames among n processes: $\frac{m}{n}$ frames to each process.
- Example, if there are 93 frames and 5 processes
  - Each process will get 18 frames.
  - The 3 leftover frames can be used as a free-frame buffer pool.

Fixed Allocation (2/4)

- Assume 62 free frames, and the frame size is 1KB

Fixed Allocation (2/4)

- Assume 62 free frames, and the frame size is 1KB
- Two processes:
  - A student process: 10KB
  - An interactive database: 127KB

Fixed Allocation (2/4)

- Assume 62 free frames, and the frame size is 1KB

- Two processes:
  - A student process: 10KB
  - An interactive database: 127KB

- Equal allocation: gives each process 31 frames

Fixed Allocation (2/4)

- Assume 62 free frames, and the frame size is 1KB
- Two processes:
  - A student process: 10KB
  - An interactive database: 127KB
- Equal allocation: gives each process 31 frames
- Wasting 21 frames

Fixed Allocation (3/4)

- Proportional allocation
- Allocate according to the size of process.
- Dynamic as degree of multiprogramming, process sizes change.

Fixed Allocation (3/4)

- Proportional allocation
- Allocate according to the size of process.
- Dynamic as degree of multiprogramming, process sizes change.
- $s_i =$ size of process p_i

Fixed Allocation (3/4)

- Proportional allocation
- Allocate according to the size of process.
- Dynamic as degree of multiprogramming, process sizes change.
- s_i = size of process p_i
- $S = \sum s_i$

Fixed Allocation (3/4)

- Proportional allocation
- Allocate according to the size of process.
- Dynamic as degree of multiprogramming, process sizes change.
- s_i = size of process p_i
- $S = \sum s_i$
- m = total number of frames

Fixed Allocation (3/4)

- Proportional allocation
- Allocate according to the size of process.
- Dynamic as degree of multiprogramming, process sizes change.
- s_i = size of process p_i
- $S = \sum s_i$
- m = total number of frames
- $a_i = \frac{s_i}{S} \times m$: allocation for p_i

Fixed Allocation (4/4)

- Assume 62 free frames, and the frame size is 1KB

Fixed Allocation (4/4)

- Assume 62 free frames, and the frame size is 1KB
- Two processes:
  - A student process: 10KB
  - An interactive database: 127KB

Fixed Allocation (4/4)

- Assume 62 free frames, and the frame size is 1KB
- Two processes:
  - A student process: 10KB
  - An interactive database: 127KB
- Equal allocation: $s_1 = 10, s_2 = 127, S = 137, m = 62$

Fixed Allocation (4/4)

- Assume 62 free frames, and the frame size is 1KB
- Two processes:
  - A student process: 10KB
  - An interactive database: 127KB
- Equal allocation: $s_1 = 10, s_2 = 127, S = 137, m = 62$
- $a_1 = \frac{10}{137} \times 62 \approx 4$ $\qquad$ $a_2 = \frac{127}{137} \times 62 \approx 57$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size.

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 vs. Local Allocation

- Global replacement: process selects a replacement frame from the set of all frames; one process can take a frame from another

Global vs. Local Allocation

- Global replacement: process selects a replacement frame from the set of all frames; one process can take a frame from another
  - The process execution time can vary greatly.
  - The greater throughput so more common.

Global vs. Local Allocation

- Global replacement: process selects a replacement frame from the set of all frames; one process can take a frame from another
  - The process execution time can vary greatly.
  - The greater throughput so more common.

- Local replacement: each process selects from only its own set of allocated frames

Global vs. Local Allocation

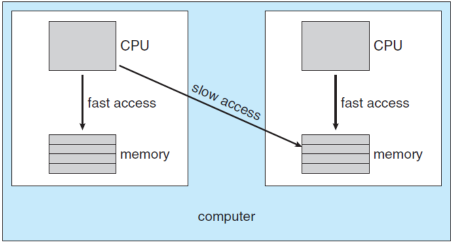
- Global replacement: process selects a replacement frame from the set of all frames; one process can take a frame from another
  - The process execution time can vary greatly.
  - The greater throughput so more common.

?

- Local replacement: each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - Possibly underutilized memory

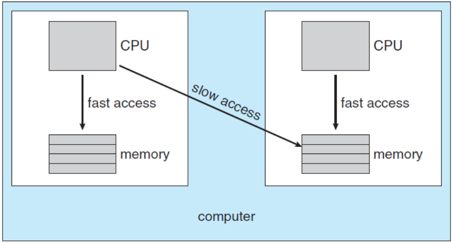
Non-Uniform Memory Access

- So far all memory accessed equally.
- Many systems are NUMA: speed of access to memory varies.



Non-Uniform Memory Access

- So far all memory accessed equally.
- Many systems are NUMA: speed of access to memory varies.



- Optimal performance: allocate memory close to the CPU on which the thread is scheduled.

Thrashing

Thrashing (1/2)

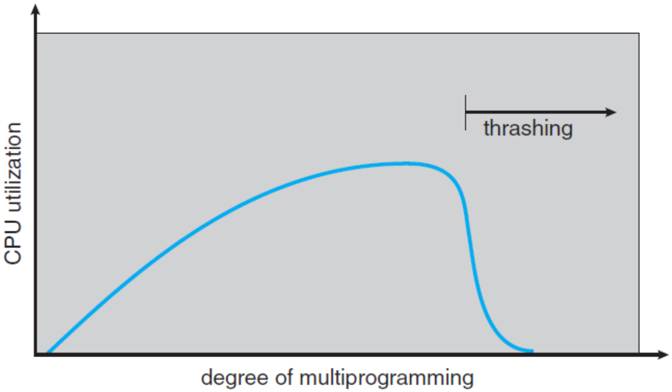
- If a process does not have enough pages, the page-fault rate is very high.
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back

Thrashing (1/2)

- If a process does not have enough pages, the page-fault rate is very high.
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back

- This leads to:
  - Low CPU utilization
  - OS thinks it needs to increase the degree of multiprogramming
  - Another process added to the system

Thrashing (2/2)



- Thrashing: a process is busy swapping pages in and out.

Prevent Thrashing

- Providing a process with as many frames as it needs.

Prevent Thrashing

- Providing a process with as many frames as it needs.
- How do we know how many frames it needs?

Locality Model (1/2)

- A locality is a set of pages that are actively used together.

Locality Model (1/2)

- A locality is a set of pages that are actively used together.

- A process moves from locality to locality.

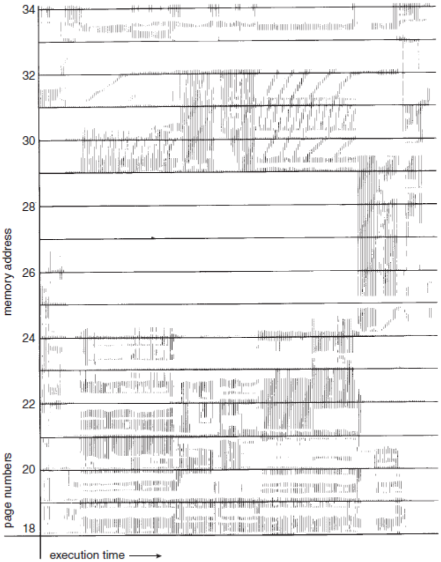
Locality Model (1/2)

- A locality is a set of pages that are actively used together.
- A process moves from locality to locality.
- A program is generally composed of several different localities, which may overlap.

Locality Model (1/2)

- A locality is a set of pages that are actively used together.
- A process moves from locality to locality.
- A program is generally composed of several different localities, which may overlap.
- For example, when a function is called, it defines a new locality: consists of memory references to the instructions of the function call, its local variables, and a subset of the global variables.

Locality Model (2/2)



Locality and Thrashing

- A process will fault for the pages in its locality, until all these pages are in memory.

Locality and Thrashing

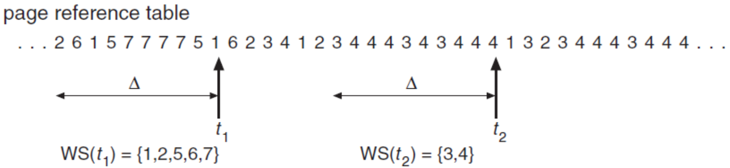
- A process will fault for the pages in its locality, until all these pages are in memory.
- After allocating all the pages of the locality, it will not fault again until it changes localities.

Locality and Thrashing

- A process will fault for the pages in its locality, until all these pages are in memory.
- After allocating all the pages of the locality, it will not fault again until it changes localities.
- If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash.

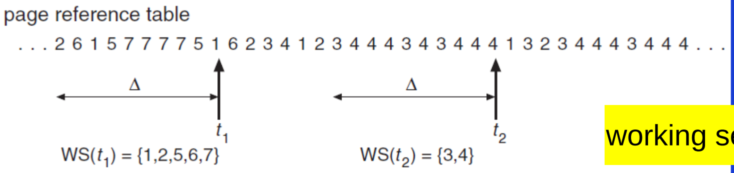
Working-Set Model (1/2)

- $\triangle$: working-set window: a fixed number of page references
- WSS_i: working set of process p_i: total number of pages referenced in the most recent $\triangle$ (varies in time).



Working-Set Model (1/2)

- △: working-set window: a fixed number of page references
- WSS_i: working set of process p_i: total number of pages referenced in the most recent △ (varies in time).
  - If △ too small will not encompass entire locality
  - If △ too large will encompass several localities
  - If $\triangle = \infty$ will encompass entire program



working set = 10

Working-Set Model (2/2)

- m: total number of frames

- D: total demand frames: $D = \sum WSS_i$:
  - Approximation of locality

Working-Set Model (2/2)

- m: total number of frames

- D: total demand frames: $D = \sum WSS_i$:
  - Approximation of locality

- If $D > m$, then thrashing

Working-Set Model (2/2)

- m: total number of frames

- D: total demand frames: $D = \sum WSS_i$:
  - Approximation of locality

- If $D > m$, then thrashing

- Policy: if $D > m$, then suspend or swap out one of the processes.

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\triangle = 10000$
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts we copy and clear the reference-bit values for each page
  - If a page fault occurs: examine the 2 bits to determine whether a page was used within the last 10,000 to 15,000 references.

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit

- Example: $\triangle = 10000$
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts we copy and clear the reference-bit values for each page
  - If a page fault occurs: examine the 2 bits to determine whether a page was used within the last 10,000 to 15,000 references.

- Why is this not completely accurate?

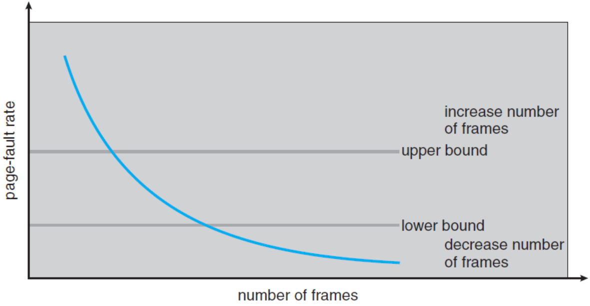
Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\triangle = 10000$
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts we copy and clear the reference-bit values for each page
  - If a page fault occurs: examine the 2 bits to determine whether a page was used within the last 10,000 to 15,000 references.
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency

- More direct approach than WSS
- Establish acceptable page-fault frequency (PFF) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

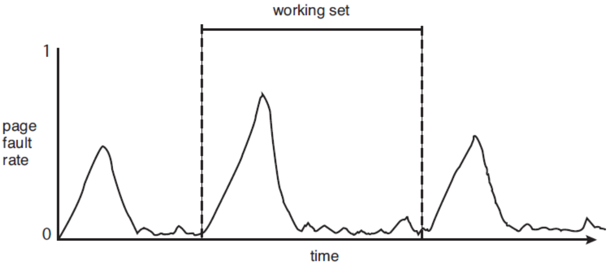
?



Working Sets and Page-Fault Rates

?

- Direct relationship between working set of a process and its page-fault rate.
- Working set changes over time.
- Peaks and valleys over time.



Allocating Kernel Memory

Allocating Kernel Memory

- Treated differently from user memory

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 I/O devices

Managing Free Memory Strategies

- Buddy system
- Slab allocation

Buddy System (1/2)

- Allocates memory from fixed-size segment consisting of physically-contiguous pages

Buddy System (1/2)

- Allocates memory from fixed-size segment consisting of physically-contiguous pages

- Memory allocated using power-of-2 allocator

Buddy System (1/2)

- 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.

Buddy System (1/2)

- 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.

Buddy System (1/2)

- Allocates memory from fixed-size segment consisting of physically-contiguous pages

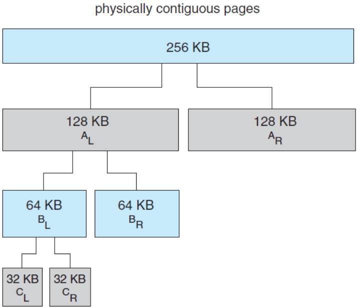
- Memory allocated using power-of-2 allocator
  - Satisfies requests in units sized as power of 2.
  - Request rounded up to next highest power of 2.
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2.

Buddy System (1/2)

- 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 (2/2)

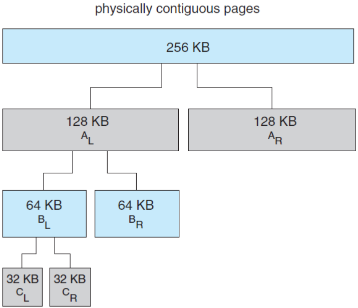
- Assume 256KB chunk available, kernel requests 21KB.
  - Split into AL and AR of 128KB each.
  - One further divided into BL and BR of 64KB.
  - One further into CL and CR of 32KB each.



Buddy System (2/2)

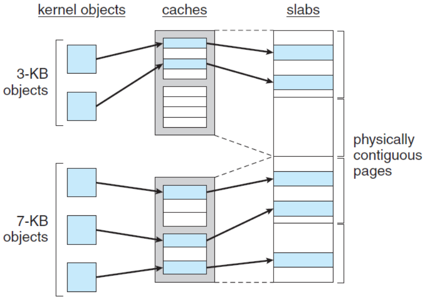
- Assume 256KB chunk available, kernel requests 21KB.
  - Split into AL and AR of 128KB each.
  - One further divided into BL and BR of 64KB.
  - One further into CL and CR of 32KB each.

?
- Advantage: quickly coalesce unused chunks into larger chunk
- Disadvantage: fragmentation



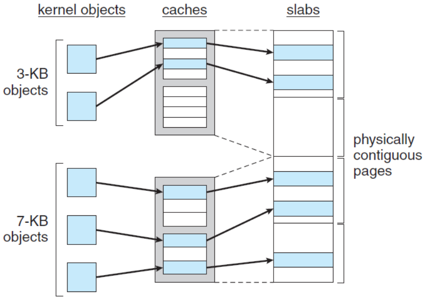
Slab Allocation (1/2)

- Alternate strategy
- Slab is one or more physically contiguous pages.



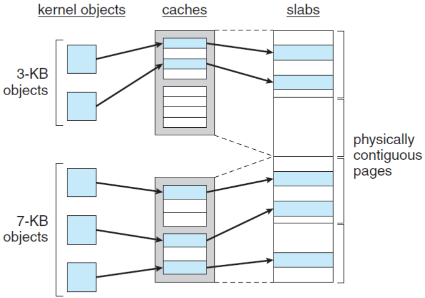
Slab Allocation (1/2)

- Alternate strategy
- Slab is one or more physically contiguous pages.
- Cache consists of one or more slabs.



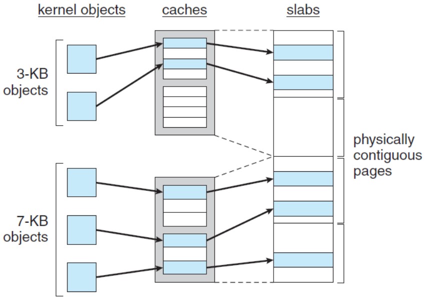
Slab Allocation (1/2)

- Alternate strategy
- Slab is one or more physically contiguous pages.
- Cache consists of one or more slabs.
- Single cache for each unique kernel data structure, e.g., a separate cache for file objects, a separate cache for semaphores, and so forth.



Slab Allocation (1/2)

- Alternate strategy
- Slab is one or more physically contiguous pages.
- Cache consists of one or more slabs.
- Single cache for each unique kernel data structure, e.g., a separate cache for file objects, a separate cache for semaphores, and so forth.
- Objects: instantiations of the data structure



Slab Allocation (2/2)

- When cache created, filled with objects marked as free.

Slab Allocation (2/2)

- When cache created, filled with objects marked as free.
- When structures stored, objects marked as used.

Slab Allocation (2/2)

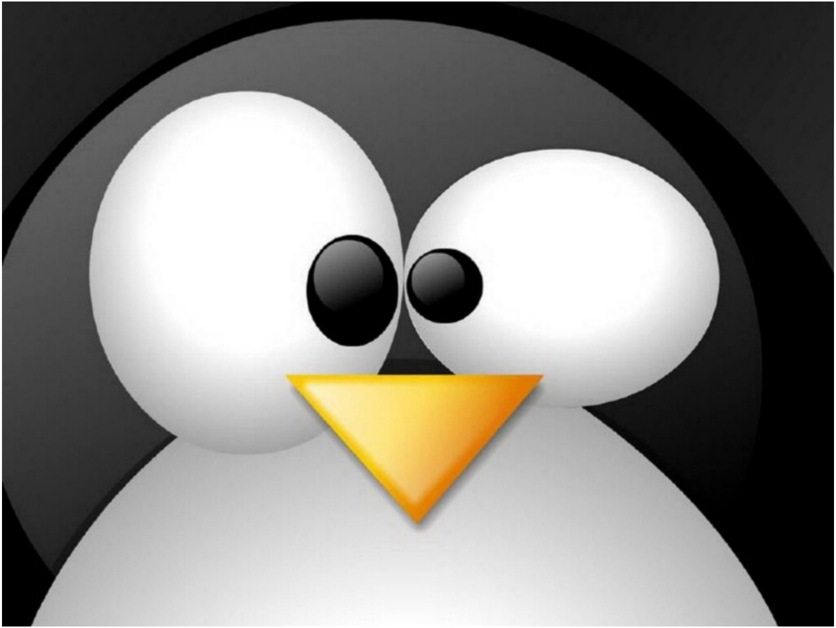
- When cache created, filled with objects marked as free.
- When structures stored, objects marked as used.
- If slab is full of used objects, next object allocated from empty slab.

Slab Allocation (2/2)

- 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.

Slab Allocation (2/2)

- 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 and fast memory request satisfaction.



Slab Allocator in Linux (1/3)

- Process descriptor: `type struct task_struct`
- Approx 1.7KB of memory
- New task $\rightarrow$ 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

Slab Allocator in Linux (2/3)

- Upon request, slab allocator
  1. Uses free struct in partial slab.
  2. If none, takes one from empty slab.
  3. If no empty slab, create new empty.

Slab Allocator in Linux (3/3)

- Linux originally used the buddy system.

Slab Allocator in Linux (3/3)

- Linux originally used the buddy system.
- Kernel 2.2 had SLAB.

Slab Allocator in Linux (3/3)

- Linux originally used the buddy system.
- Kernel 2.2 had SLAB.
- Recent distribution added two more allocators:
  - SLOB (Simple List of Blocks): for systems with limited memory, maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB, removes per-CPU queues, metadata stored in page structure, from kernel 2.6.24

Memory-Mapped Files

Memory-Mapped Files (1/3)

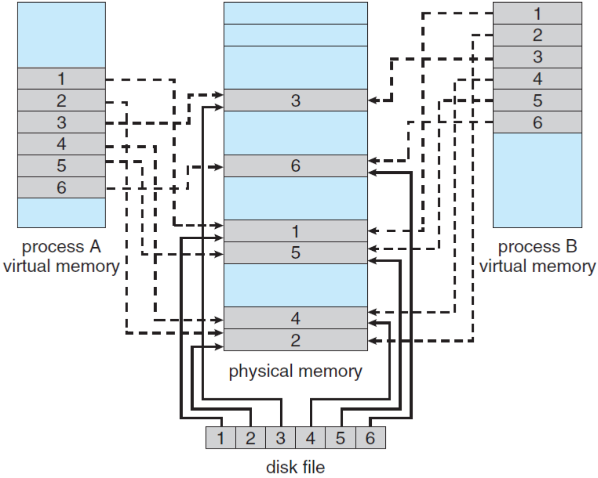
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory.

Memory-Mapped Files (1/3)

- 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.

Memory-Mapped Files (2/3)



Memory-Mapped Files (3/3)

- Process can explicitly request memory mapping a file via `mmap()` system call: map the file into kernel address space

Memory-Mapped Files (3/3)

- Process can explicitly request memory mapping a file via `mmap()` system call: map the file into kernel address space

- Simplifies and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls.

Memory-Mapped Files (3/3)

- Process can explicitly request memory mapping a file via `mmap()` system call: map the file into kernel address space

- 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.

Memory-Mapped Files (3/3)

- Process can explicitly request memory mapping a file via `mmap()` system call: map the file into kernel address space
- 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.
- When does written data make it to disk?

Memory-Mapped Files (3/3)

- Process can explicitly request memory mapping a file via `mmap()` system call: map the file into kernel address space
- 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.
- When does written data make it to disk?
  - Periodically and/or at file `close()` time.

Memory-Mapped I/O

- Many computer architectures provide memory-mapped I/O.

Memory-Mapped I/O

- Many computer architectures provide memory-mapped I/O.
- Ranges of memory addresses are mapped to the device registers.

Memory-Mapped I/O

- Many computer architectures provide memory-mapped I/O.
- Ranges of memory addresses are mapped to the device registers.
- Reads and writes to these memory addresses cause the data to be transferred to and from the device registers.
  - Called, I/O port

Other Considerations

Prepaging

- To reduce the large number of page faults that occurs at process startup.

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.

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.
- If prepaged pages are unused, I/O and memory was wasted.

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.
- If prepaged pages are unused, I/O and memory was wasted.
- Assume s pages are prepaged and a fraction $0 \leq \alpha \leq 1$ of the pages are used.
  - Cost of $s \times \alpha$ $>$ or $<$ than the cost of prepaging $s \times (1 - \alpha)$ unnecessary pages?
  - If α close to 0: prepaging loses; if α close to 1, prepaging wins

Page Size

- 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

TLB Search

- TLB reach: the amount of memory accessible from the TLB

TLB Search

- TLB reach: the amount of memory accessible from the TLB
- TLB reach $=$ (TLB Size) $\times$ (Page Size)

TLB Search

- TLB reach: the amount of memory accessible from the TLB
- TLB reach = (TLB Size) $\times$ (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults.

TLB Search

- TLB reach: the amount of memory accessible from the TLB
- TLB reach = (TLB Size) × (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

TLB Search

- TLB reach: the amount of memory accessible from the TLB
- TLB reach = (TLB Size) × (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
- Provide multiple page sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Program Structure

- `int[128,128] data`: each row is stored in one page
- Program 1

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

Program Structure

- `int[128,128] data`: each row is stored in one page
- Program 1

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

$128 \times 128 = 16,384$ page faults

Program Structure

- `int[128,128] data`: each row is stored in one page
- Program 1

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

$128 \times 128 = 16,384$ page faults

- Program 2

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

Program Structure

- `int[128,128] data`: each row is stored in one page
- Program 1

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

$128 \times 128 = 16,384$ page faults

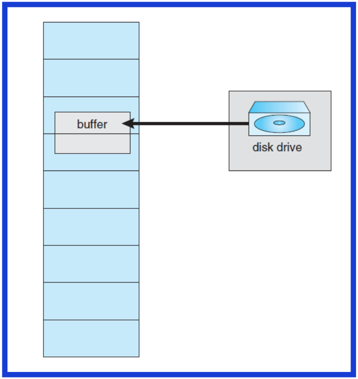
- Program 2

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

128 page faults

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.



Summary

Summary

- Frame allocation: fixed and priority allocations

Summary

- Frame allocation: fixed and priority allocations
- Global and local allocation

Summary

- Frame allocation: fixed and priority allocations
- Global and local allocation
- Thrashing: total demand frames $>$ total num. of frames

Summary

- Frame allocation: fixed and priority allocations
- Global and local allocation
- Thrashing: total demand frames $>$ total num. of frames
- Prevent trashing: working set model and page fault frequency

Summary

- Frame allocation: fixed and priority allocations
- Global and local allocation
- Thrashing: total demand frames $>$ total num. of frames
- Prevent trashing: working set model and page fault frequency
- Allocating kernel memory: buddy system and slab allocation

Summary

- Frame allocation: fixed and priority allocations
- Global and local allocation
- Thrashing: total demand frames > total num. of frames
- Prevent trashing: working set model and page fault frequency
- Allocating kernel memory: buddy system and slab allocation
- Memory-mapped files and I/O

Questions?

Acknowledgements

Some slides were derived from Avi Silberschatz slides.