Memory Management III

Operating Systems

Oleg Goldshmidt

ogoldshmidt@computer.org

Lecture 6

Virtual Memory

- must logical memory be mapped to physical?
- usually, parts of the program are not needed
  - error handling
  - overallocated arrays, lists, symbol tables, etc
  - unused options and features
- not everything is needed at the same time
- execute a program that is only partially in memory
  - the size of physical memory no longer a constraint
  - higher degree of multiprogramming, higher CPU utilization, higher throughput, with no response time or turnaround time penalty
  - potentially less I/O for swapping

Demand Paging

- lazy swapping: never swap a page into memory unless the page is needed
  - not swapping but paging
- when swapping a process in the pager guesses which pages will be used before the next swap-out
  - does not load pages unlikely to be used
- `pte` of a page not currently in memory has invalid bit on
  - no effect if the process never accesses the page
  - if an invalid page is accessed, a page fault occurs
    - hardware traps into the OS

Handling Page Faults

- check the PCB to find out if the reference is valid
  - if it is invalid — segfault
- find a free frame (the OS keeps track of free memory)
- schedule a disk I/O to read the swapped-out page into the frame
- when the I/O is completed, modify the page table to indicate the page is in memory
  - also modify the process internal tables (in PCB)
- continue the interrupted process from the instruction that caused the page fault — the needed page is now in memory

Page Faults: Analysis I

- pure demand paging: a process can start executing with no pages in memory
  - the OS will set the instruction pointer to the address of the first instruction, which is on a non-resident page
  - there will be a page fault, and the page will be brought into memory
- in principle, there may be several page faults per instruction
  - rare in practice
- same hardware support as for paging and swapping: page tables and backing store

Page Faults: Analysis II

- a fault may occur when we fetch an instruction, or the operands, or try to store the result
- need to restart the instruction again after bringing the page into memory
- a problem when, say moving a block (e.g., `MVC` on IBM 360/370, which moves up to 256 bytes from one location to another, possibly overlapping, location)
- possible solutions
  - check beginning and end before moving
  - use temporary registers to hold the overwritten values, write them back into memory before handling the fault

Demand Paging Performance I

- basic parameters
  - p — probability of a page fault
  - t_m — memory access time
  - t_f — page fault time
- $t_e = (1 - p)t_m + pt_f$ — effective access time
- usually $p << 1$
- slowdown: $t_e/t_m = 1 + p(t_f/t_m)$
- acceptable performance: $p(t_f/t_m) < 1$

Demand Paging Performance II

- handling a page fault
  - service the page fault interrupt: trap to OS — save registers and process state — determine that the trap was a page fault — determine disk location
  - read in the page: issue a read command to a free frame — wait in the disk queue — wait for the device seek and/or latency time — begin the transfer — context switch to another process — interrupt from the disk
  - restart the process: save the running process's registers and state — determing the interrupt was from disk — update the page tables — wait for CPU — context switch
- context switches are optional

Page Replacement

- what if there are no free frames?
  - we can swap out a process
  - or find a frame that is not used and swap it out
- doubles the page fault service time t_f: one page is written to disk, one read from disk
- optimization: use dirty bit to indicate that the page has been modified since swap-in
  - only write dirty pages back to disk
- for efficient demand paging we need a frame allocation algorithm and a page replacement algorithm

Page Replacement Algorithms

- goals
  - minimize the page fault rate
  - maximize the degree of multiprogramming
- evaluation: “reference string”
  - generated randomly
  - recorded trace from a real system and workload
- collapse subsequent references to the same page
- example: for 100 byte pages:

```
0100, 0432, 0101, 0612, 0102, 0103, 0104,
0101, 0611, 0102, 0103, 0104, 0101, 0610,
0102, 0103, 0104, 0101, 0609, 0102, 0105

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1
```

Page Replacement: FIFO

- record arrival time for each page
- swap out the oldest page in the system
- alternatively, maintain a FIFO of pages
- example (with 3 frames):

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

- oldest page may contain initialization code or a heavily used variable

Belady's Anomaly

- if we increase the number of available frames we expect the page fault rate to go down
- not always the case!

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

Optimal Page Replacement

- there is an optimal page replacement algorithm with the lowest page fault rate for a fixed number of frames
- replace the page that won't be used for the longest time

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		2			2			2				7		
	0	0	0		0		4			0			0				0		
		1	1		3		3			3			1				1		

- no Belady's anomaly
- problem: we don't know the future
- useful to compare practical algorithms with

Page Replacement: LRU

- an approximation to optimal replacement
- replace the page not used for the longest time
- may swap out a page that will be used soon

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

- good performance, but difficult to implement

LRU Implementation Issues

- need to maintain a data structure updated on every memory reference
  - accessible in constant time
  - update time much shorter than memory access
- counters — maintain time of last reference per page
  - a global counter updated on each memory reference
  - per page counter updated with the value of global counter when the page is referenced
  - need to search page tables, clock may overflow, etc.
- stack (or, rather, list)
  - on reference, a page is moved to head, tail is replaced (no search)
- both options require HW support and are not practical

LRU Approximations I

- reference bit per page, set to 1 when page is referenced
  - replace a page with 0 reference bit — no order information
- additional reference bits
  - keep, e.g., 8 bits per page to record the reference history
  - on timer interrupt copy the reference bit into the high order bit, shift the rest to the right, discard the lowest
- second chance algorithm
  - keep a FIFO (or a circular list), check the reference bit, if it is 1 clear it, reset arrival time, and move to the next page
  - heavily used pages will not be replaced

LRU Approximations II

- enhanced second chance algorithm
  - check both the reference bit and the dirty bit
  - possibilities:
    - (0,0) — neither recently used nor modified, the best candidate for replacement
    - (0,1) — not recently used but dirty, not quite as good because a write out is needed
    - (1,0) — recently used but clean, likely to be used again soon?
    - (1,1) — recently used and dirty, likely to be used again and needs a write-out
  - replace the first page in the lowest non-empty class

Counting Algorithms

- count the references to each page
- LFU (least frequently used)
  - replace the page with the lowest count
  - a page may have a high count but be no longer in use
  - shift the count right regularly to age the pages
- MFU (most frequently used)
  - replace the page with highest count
  - pages with smallest count have just been brought in
- workload dependent
- not good approximations to the optimal algorithm

Allocating Frames

- how many frames should we allocate to a process?
- considerations:
  - there is a minimum number of frames dependent on the architecture (how many frames will an instruction need?)
    - indirect addressing must be taken into account
  - fixed number of frames per process does not take into account the process size or needs
  - allocate frames in proportion to the process size
  - take priority into account

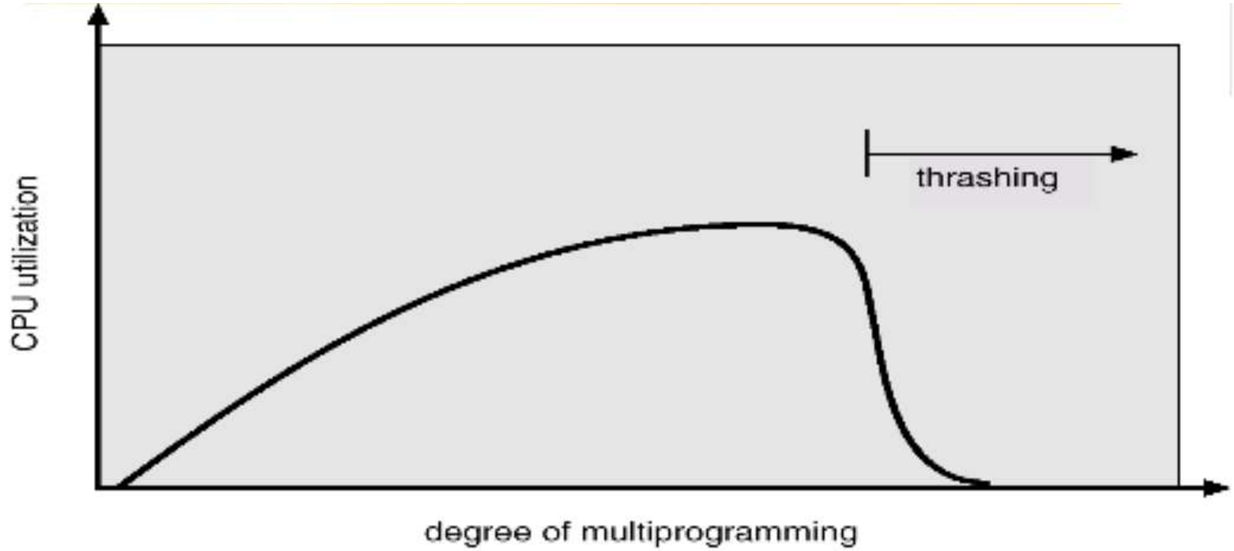
Local vs. Global Allocation

- local allocation
  - the number of frames allocated to a process is constant
  - when a process needs a new page it replaces one of its own
- global allocation
  - any frame can be chosen for replacement
  - less predictable performance

Thrashing I

- what if a process has too few pages?
  - it will need a new page, and will replace an existing one
  - if the replaced page is heavily used it will cause another fault
  - the process will spend more time paging than running — thrashing
- typical cause of thrashing
  - CPU utilization decreases, OS brings another process in
  - with global allocation, processes take frames from each other, fill the paging device queue, empty the ready queue
  - CPU utilization decreases further

Thrashing II



- local replacement helps to an extent
- a thrashing process queues for paging device, increasing the service time for page faults — still affects others
- how many pages does a process actually use?

Locality Model

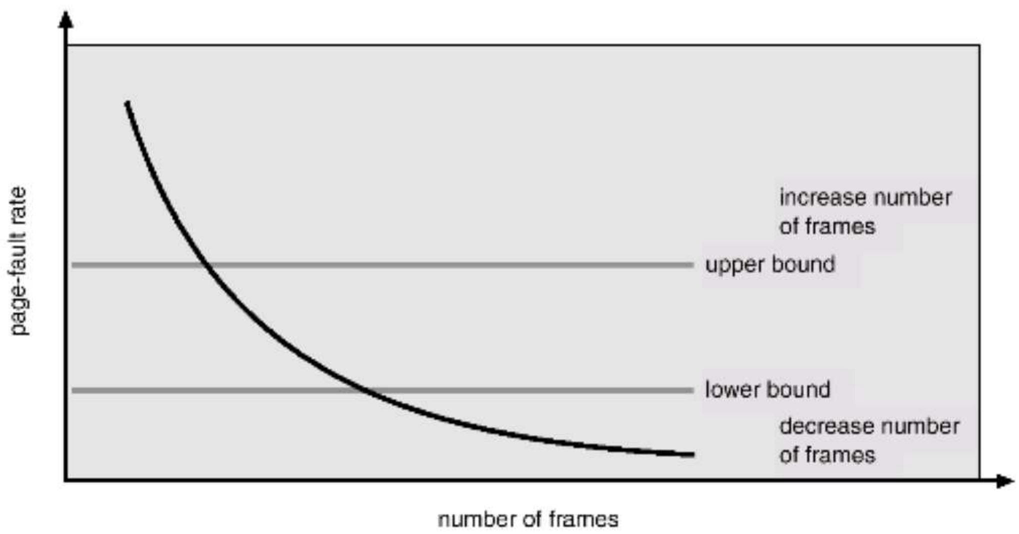
- locality — a set of pages actively used together
- a program consists of several localities that may overlap
- when a function is called it defines a new locality
  - the function's instructions
  - local variables
  - a subset of global variables
- on return the process leaves this locality
- locality model: all programs have this locality pattern, determined by the program structure and data
  - the basis for most caching decisions
- allocate enough frames for the current locality

Working Set Model

- Δ — working set window, a fixed # of page references
- WSS_i — working set size or process i — total number of pages referenced in the most recent Δ
  - if Δ is too small, it will not encompass the locality
  - if Δ is too large, it will encompass several localities
  - $\Delta \to \infty$ — entire program
- $D = \sum_i WSS_i$ — total demand for frames
- M — total memory
- if $D > M$ thrashing will occur — suspend processes
- prepaging: remember the working set for the swapped-out process, bring all the pages in at once

Page Fault Frequency

- one can also monitor the page fault frequency to control thrashing
- establish acceptable range of fault rate
  - if page fault rate is too high, process gains a frame
  - if page fault rate is too low, process loses a frame



Choosing Page Size

- small pages
  - less fragmentation
  - better locality tracking (less I/O)
  - less time to transfer pages to/from disk
- large pages
  - smaller page tables
  - less TLB flushing
  - more efficient swapping (disk latency and seek time dominate transfer time)
  - fewer page faults
- historical trend — toward larger pages
- sometimes beneficial to use huge pages for particular applications

Page Tables in Linux I

`include/asm/page.h`

```
#define PAGE_SHIFT 12
#define PAGE_SIZE  (1UL << PAGE_SHIFT)
```

- similarly for `PGDIR_SIZE`, `PUD_SIZE`, `PMD_SIZE` (in `include/asm/pgtable.h`)

`pte` bit (examples)	meaning
`_PAGE_PRESENT`	resident in memory, not swapped out
`_PAGE_RW`	writable
`_PAGE_USER`	accessible from userspace
`_PAGE_DIRTY`	dirty bit
`_PAGE_ACCESSED`	reference bit

Page Tables in Linux II

- pgd_offset(), pmd_offset(), pte_offset() point into different levels of page table
- pte_none(), pmd_none(), etc.— checks existence of entry
- pte_present(), etc.— check the _PAGE_PRESENT bits
- pmd_bad(), pgd_bad() — check entries when passed as input to functions that may change the entry value
  - architecture-dependent, but normally start with checking that the page is present and accessed

Walking Through Page Tables

mm/memory.c:__follow_page():

```
pgd = pgd_offset(mm, address);
if (pgd_none(*pgd) || unlikely(pgd_bad(*pgd)))
        goto out;
pud = pud_offset(pgd, address);
if (pud_none(*pud) || unlikely(pud_bad(*pud)))
        goto out;
pmd = pmd_offset(pud, address);
if (pmd_none(*pmd) || unlikely(pmd_bad(*pmd)))
        goto out;
ptep = pte_offset_map(pmd, address);
if (!ptep)
        goto out;
pte = *ptep;
```

Process Address Space I

include/linux/sched.h

```
struct task_struct {
    ...
    struct mm_struct *mm;
    ...
};
```

- only one mm_struct per process
- threads of a process — all task_struct's that point to the same mm_struct

Process Address Space II

`include/linux/sched.h`

```
struct mm_struct {
    struct vm_area_struct * mmap;
    ...
};
```

- VMA's share protection attributes and purpose
- examples: shared library loaded into the address space, heap, etc.
- VMA's can be viewed in `/proc/<pid>/maps`

Virtual Memory Areas

include/linux/mm.h

```
struct vm_area_struct {
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;
    /* linked list of VM areas per task,
       sorted by address */
    struct vm_area_struct *vm_next;
    ...
    /* protection, flags, backing store, etc */
    struct vm_operations_struct * vm_ops;
};
```

- vm_ops: open(), close(), nopage()
- mmap(2), munmap(2) system calls

Memory Usage Of A Process I

```
# egrep "^Vm" /proc/5909/status
VmSize:     4448 kB
VmLck:         0 kB
VmRSS:      1408 kB
VmData:      304 kB
VmStk:        84 kB
VmExe:       556 kB
VmLib:      1380 kB
VmPTE:        28 kB
```

- Size = code + data + stack
- RSS = resident set size (memory mapped in RAM)
- Size and RSS don’t count page tables, `task_struct`

Out Of Memory I

- "OOM Killer"
  - very controversial
  - many suggestions to remove it
- when a system needs more memory, e.g., expanding the heap via `brk(2)` or remapping an address space via `mremap(2)`, it will check if it has enough memory to satisfy the request
- `vm_enough_memory()` checks how many pages are potentially available
  - total free pages, total page cache, total free swap pages, filesystem caches, etc.
- if `false` is returned to the caller the caller returns `-ENOMEM` to userspace

Out Of Memory II

- if nothing helps `out_of_memory()` is called
- selects a process that uses a lot of memory but has not been running for a long time
  - long running processes are unlikely to cause memory shortage
- assumes that processes with root privileges are well-behaved
- try not to kill a process that can access HW directly
- walk through the tasks again and find those sharing `mm_struct` with the selected task (i.e., all the threads), and send `SIGTERM` (for `RAWIO` processes) or `SIGKILL`