# Operating Systems Principles
# Virtual Memory and Paging

Mark Kampe
(markk@cs.ucla.edu)

## Virtual Memory and Paging





## Memory Management

1. allocate/assign physical memory to processes
   – explicit requests: malloc (sbrk)
   – implicit: program loading, stack extension
2. manage the virtual address space
   – instantiate virtual address space on context switch
   – extend or reduce it on demand
3. manage migration to/from secondary storage
   – optimize use of main storage
   – minimize overhead (waste, migrations)



## Memory Management Goals

1. transparency
   – process sees only its own virtual address space
   – process is unaware memory is being shared
2. efficiency
   – high effective memory utilization
   – low run-time cost for allocation/relocation
3. protection and isolation
   – private data will not be corrupted
   – private data cannot be seen by other processes



## Primary and Secondary Storage

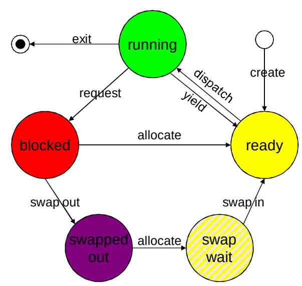
- primary = main (executable) memory
  – primary storage is expensive and very limited
  – only processes in primary storage can be run
- secondary = non-executable (e.g. Disk/SSD)
  – blocked processes can be moved to secondary storage
  – swap out code, data, stack and non-resident context
  – make room in primary for other "ready" processes
- returning to primary memory
  – process is copied back when it becomes unblocked



## Why we swap

- make best use of a limited amount of memory
  – process can only execute if it is in memory
  – can’t keep all processes in memory all the time
  – if it isn't READY, it doesn't need to be in memory
  – swap it out and make room for other processes
- improve CPU utilization
  – when there are no READY processes, CPU is idle
  – CPU idle time means reduced system throughput
  – more READY processes means better utilization

## scheduling states with swapping



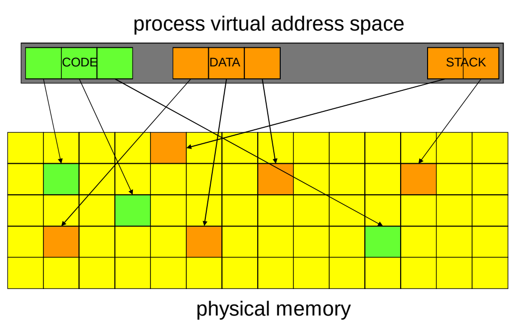



## Pure Swapping

- each segment is contiguous
  – in memory, and on secondary storage
  – all in memory, or all on swap device
- swapping takes a great deal of time
  – transferring entire data (and text) segments
- swapping wastes a great deal of memory
  – processes seldom need the entire segment
- variable length memory/disk allocation
  – complex, expensive, external fragmentation



## paged address translation





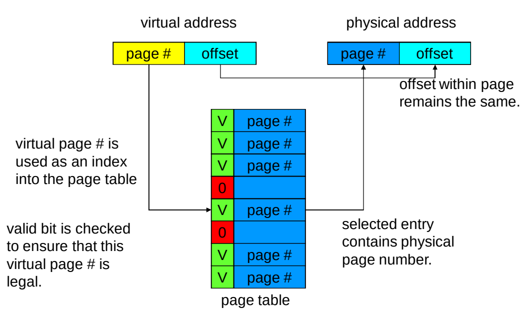

## Paging and Fragmentation

a segment is implemented as a set of virtual pages

- internal fragmentation
  – averages only ½ page (half of the last one)
- external fragmentation
  – completely non-existent (we never carve up pages)

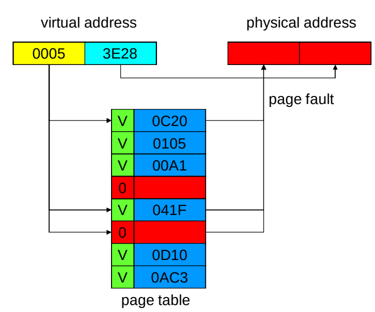


## Paging Memory Management Unit






## Paging Relocation Examples



## Demand Paging

- paging MMU supports *not present* pages
  - generates a fault/trap when they are referenced
  - OS can bring in page, retry the faulted reference
- entire process needn't be in memory to run
  - start each process with a subset of its pages
  - load additional pages as program *demands* them
- they don't need all the pages all the time
  - code execution exhibits reference locality
  - data references exhibit reference locality



## Demand Paging – advantages

- improved system performance
  - fewer in-memory pages per process
  - more processes in primary memory
    - more parallelism, better throughput
    - better response time for processes already in memory
  - less time required to page processes in and out
- fewer limitations on process size
  - process can be larger than physical memory
  - process can have huge (sparse) virtual space



## Page Fault Handling

- initialize page table entries to *not present*
- CPU faults when invalid page is referenced
  1. trap forwarded to page fault handler
  2. determine which page, where it resides
  3. find and allocate a free page frame
  4. block process, schedule I/O to read page in
  5. update page table point at newly read-in page
  6. back up user-mode PC to retry failed instruction
  7. unblock process, return to user-mode



## Demand Paging and Performance

- page faults hurt performance
  - increased overhead
    - additional context switches and I/O operations
  - reduced throughput
    - processes are delayed waiting for needed pages
- key is having the "right" pages in memory
  - right pages -> few faults, little overhead/delay
  - wrong pages -> many faults, much overhead/delay
- we have little control over what we bring in
  - we read the pages the process *demands*
- key to performance is which pages we evict



## Belady's Optimal Algorithm

- Q: which page should we replace?
  A: the one we won't need for the longest time
- Why is this the right page?
  - it delays the next page fault as long as possible
  - minimum number of page faults per unit time
- How can we predict future references?
  - Belady cannot be implemented in a real system
  - but we can run implement it for test data streams
  - we can compare other algorithms against it



## Approximating Optimal Replacement

- note which pages have recently been used
  - use this data to predict future behavior
- Possible replacement algorithms
  - random, FIFO: straw-men ... forget them
- Least Recently Used
  - assert near future will be like recent past
    - programs do exhibit temporal and spatial locality
    - if we haven't used it recently, we probably won't soon
  - we don't have to be right 100% of the time
    - the more right we are, the more page faults we save

## Why Programs Exhibit Locality

- Code locality
  - code in same routine is in same/adjacent page
  - loops iterate over the same code
  - a few routines are called repeatedly
  - intra-module calls are common
- Stack locality
  - activity focuses on this and adjacent call frames
- Data reference locality
  - this is common, but not assured



## True LRU is hard to implement

- maintain this information in the MMU?
  - MMU notes the time, every time a page is referenced
  - maybe we can get a per-page read/written bit
- maintain this information in software?
  - mark all pages invalid, even if they are in memory
  - take a fault the first time each page is referenced
  - then mark this page valid for the rest of the time slice
- finding oldest page is prohibitively expensive
  - 16GB memory / 4K page = 4M pages to scan



## Practical LRU surrogates

- must be cheap
  - can't cause additional page faults
  - avoid scanning the whole page table (it is big)
- clock algorithms … a surrogate for LRU
  - organize all pages in a circular list
  - position around the list is a surrogate for age
  - progressive scan whenever we need another page
    - for each page, ask MMU if page has been referenced
    - if so, reset the reference bit in the MMU; skip page
    - if not, consider this page to be the least recently used



## True Global LRU Replacement

| reference | a | b | c | d | a | b | d | e | f | a | b | c | d | a | e | d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

True LRU — loads 4, replacements 7

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 0 | a | | | | ! | | | | f | | | | d | | | ! |
| frame 1 | | b | | | | ! | | | | a | | | | ! | | |
| frame 2 | | | c | | | | | e | | | | c | | | | |
| frame 3 | | | | d | | | ! | | | | b | | | | e | |



## LRU Clock Algorithm

| reference | a | b | c | d | a | b | d | e | f | a | b | c | d | a | e | d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

LRU clock — loads 4, replacements 7

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 0 | a | | | | ! | ! | ! | ! | f | | | | d | | | ! |
| frame 1 | | b | | | | ! | ! | ! | | a | | | | ! | ! | |
| frame 2 | | | c | | | | | e | | | b | | | | e | |
| frame 3 | | | | d | | | ! | ! | ! | | | c | | | | |
| clock pos | 0 | 1 | 2 | 3 | 0 | 0 | 0 | 0/2 | 0/3 | 1 | 2 | 3 | 0 | 1 | 1/2 | 3 |

True LRU — loads 4, replacements 7

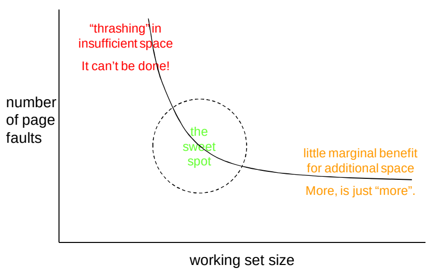
| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 0 | a | | | | a | | | | f | | | | d | | | d |
| frame 1 | | b | | | | b | | | | a | | | | a | | |
| frame 2 | | | c | | | | | e | | | | c | | | | |
| frame 3 | | | | d | | | d | | | | b | | | | e | |



## Working Sets – per process LRU

- Global LRU is probably a blunder
  - bad interaction with round-robin scheduling
  - better to give each process it's own page pool
  - do LRU replacement within that pool
- fixed # of pages per process is also bad
  - different processes exhibit different locality
    - which pages are needed changes over time
    - number of pages needed changes over time
  - much like different natural scheduling intervals
- we clearly want dynamic working sets

## "natural" working set size






## (Optimal Working Sets)

- What is optimal working set for a process?
  - number of pages needed during next time slice
- what if try to run process in fewer pages?
  - needed pages replace one another continuously
  - this is called "thrashing"
- how can we know what working set size is?
  - by observing the process behavior
- which pages should be in the working-set?
  - no need to guess, the process will fault for them



## Implementing Working Sets

- managed working set size
  - assign page frames to each in-memory process
  - processes page against themselves in working set
  - observe paging behavior (faults per unit time)
  - adjust number of assigned page frames accordingly
- page stealing (WS-Clock) algorithms
  - track last use time for each page, for owning process
  - find page least recently used (by its owner)
  - processes that need more pages tend to get more
  - processes that don't use their pages tend to lose them



## Working Set Clock Algorithm

| page frame | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| referenced | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| process | $P_0$ | $P_0$ | $P_1$ | $P_2$ | $P_2$ | $P_1$ | $P_1$ | $P_0$ | $P_2$ | $P_0$ | $P_1$ | $P_2$ | $P_0$ | $P_1$ | $P_2$ |
| last ref | 15 | 51 | 69 | 65 | 80 | 15 | 75 | 33 | 72 | 54 | 23 | 25 | 45 | 25 | 47 |

clock ptr

current execution times $P_0 = 55$ $P_1 = 75$ $P_2 = 80$ t = 15

$P_0$ gets a fault
page 6 was just referenced
clear ref bit, update time
page 7 is (55-33=22) ms old
$P_0$ replaces his own page



## Working Set Clock Algorithm

| page frame | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| referenced | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| process | $P_0$ | $P_0$ | $P_1$ | $P_2$ | $P_2$ | $P_1$ | $P_1$ | $P_0$ | $P_2$ | $P_0$ | $P_0$ | $P_2$ | $P_0$ | $P_1$ | $P_2$ |
| last ref | 15 | 51 | 69 | 65 | 80 | 15 | 75 | 33 | 72 | 54 | | 25 | 45 | 25 | 47 |

clock ptr

current execution times $P_0 = 55$ $P_1 = 75$ $P_2 = 80$ t = 25

$P_0$ gets a fault
page 6 was just referenced
page 7 is (55-33=22) ms old
$P_0$ replaces his own page

$P_0$ gets a fault
page 6 was just referenced
page 7 is (55-33=22) ms old
page 8 is (80-72=8) ms old
page 9 is (55-54=1) ms old
page 10 is (75-23=52) ms old
$P_0$ steals this page from $P_1$



## Thrashing Prevention

- working set size characterizes each process
  - how many pages it needs to run for τ milliseconds
- What if we don't have enough memory?
  - sum of our working sets exceeds available memory
- we cannot squeeze working set sizes
  - this will result in thrashing
- reduce number of competing processes
  - swap some of the ready processes out
  - to ensure enough memory for the rest to run
- we can round-robbin who is in and out

## Pre-loading – a page/swap hybrid

- what happens when process swaps in
- pure swapping
  - all pages present before process is run, no page faults
- pure demand paging
  - pages are only brought in as needed
  - fewer pages per process, more processes in memory
- what if we pre-load the last working set?
  - far fewer pages to be read in than swapping
  - *probably* the same disk reads as pure demand paging
  - far fewer initial page faults than pure demand paging



## Clean and Dirty Pages

- consider a page, recently paged in from disk
  - there are two copies, on on disk, one in memory
- if the in-memory copy has not been modified
  - there is still a valid copy on disk
  - the in-memory copy is said to be "clean"
  - we can replace page without writing it back to disk
- if the in-memory copy has been modified
  - the copy on disk is no longer up-to-date
  - the in-memory copy is said to be "dirty"
  - if we write it out to disk, it becomes "clean" again



## preemptive page laundering

- clean pages can be replaced at any time
  - copy on disk is already up to date
  - clean pages give flexibility to memory scheduler
  - many pages that can, if necessary, be replaced
- ongoing background write-out of dirty pages
  - find and write-out all dirty, non-running pages
    - no point in writing out a page that is actively in use
  - on assumption we will eventually have to page out
  - make them clean again, available for replacement
- this is the outgoing equivalent of pre-loading



## Copy on Write

- *fork(2)* is a very expensive operation
  - we must copy all private data/stack pages
  - sadly most will be discarded by next *exec(2)*
- assume child will not update most pages
  - share all private pages, mark them *copy on write*
  - change them to be read-only for parent and child
  - on write-page fault, make a copy of that page
  - on exec, remaining pages become private again
- *copy on write* is a common optimization



## assignments

- reading for the next lecture
  - Inter-Process Communication
  - named pipes … simple stream communication
  - send(2), recv(2) … network communication
  - mmap(2) … shared memory segments
  - Arpaci ch 25 … Introduction
  - Arpaci ch 26 … Concurrency and Threads
  - Arpaci ch 27 … Thread API
  - User-Mode Threads



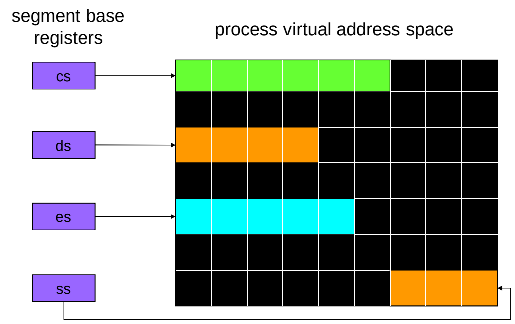
# Supplementary Slides

## paging and segmentation

- pages are a very nice memory allocation unit
  - they eliminate internal and external fragmentation
  - they admit of a very simple and powerful MMU
- they are not a particularly natural unit of data
  - programs are comprised of, and operate on, segments
  - segments are the natural "chunks" of virtual address space
    - e.g. we map a new segment into the virtual address space
  - each code, data, stack segment contains many pages
- two levels of memory management abstraction
  - a virtual address space is comprised of segments
  - relocation & swapping is done on a page basis
  - segment base addressing, with page based relocation
- user processes see segments, paging is invisible



## segmentation on top of paging






## Segments – collections of pages

- a segment is a named collection of pages
  - each page has a home on secondary storage
- operations on segments:
  - create/open/destroy
  - map/unmap segment to/from process
  - find physical page number of virtual page n
- connection between paging & segmentation
  - segment mapping implemented w/page mapping
  - page faulting uses segments to find requested page



## Managing Secondary Storage

- where do pages live when not in memory?
  - we swap them out to secondary storage (disk)
  - how do we manage our swap space?
- as a pool of variable length partitions?
  - allocate a contiguous region for each process
- as a random collection of pages?
  - just use a bit-map to keep track of which are free
- as a file system?
  - create a file per process (or segment)
  - file offsets correspond to virtual address offsets



## Paging and Shared Segments

- shared memory, executables and DLLs
- created/managed as mappable segments
  - one copy mapped into multiple processes
  - demand paging same as with any other pages
  - 2ndary home may be in a file system
- shared pages don't fit working set model
  - may not be associated with just one process
  - global LRU may be more appropriate
  - shared pages often need/get special handling



## Virtual Memory and I/O

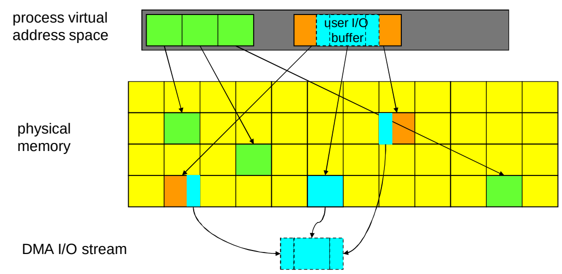
- user I/O requests use virtual buffer address
  - how can a device controller find that data
- kernel can copy data into physical buffers
  - accessing user data through standard mechanisms
- kernel may translate virtual to physical
  - give device the corresponding physical address
- CPU may include an I/O MMU
  - use page tables to translate virt addrs to phys
  - all DMA I/O references go through the I/O MMU

## Scatter/Gather I/O

- many controllers support DMA transfers
  - entire transfer must be contiguous in physical memory
- user buffers are in paged virtual memory
  - user buffer may be spread all over physical memory
  - *scatter*: read from device to multiple pages
  - *gather*: writing from multiple pages to device
- same three basic approaches apply
  - copy all user data into contiguous physical buffer
  - split logical req into chain-scheduled page requests
  - I/O MMU may automatically handle scatter/gather



## *"gather"* writes from paged memory






## *"scatter"* reads into paged memory