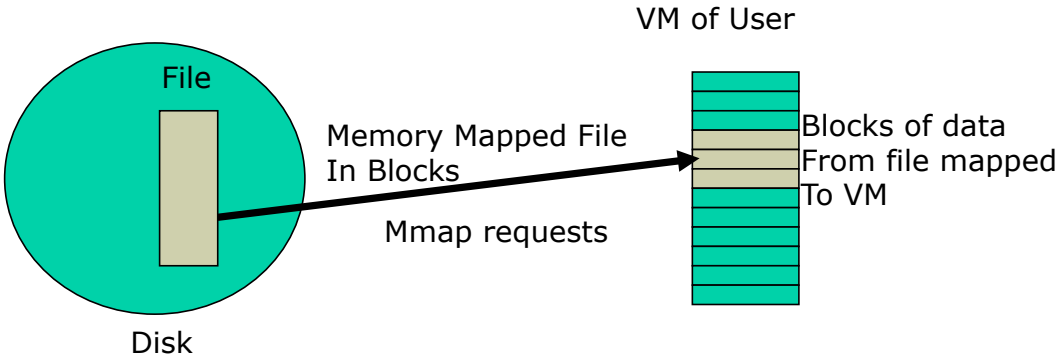
Virtual Memory

Lawrence Angrave and Vikram Adve

Contents

- Memory mapped files
- Page sharing
- Page protection
- Virtual Memory and Multiprogramming
  - Page eviction policies
  - Page frame allocation and thrashing
  - Working set model and implementation

Memory Mapped Files



Uses of Memory Mapped Files

Dynamic loading. By mapping executable files and shared libraries into its address space, a program can load and unload executable code sections dynamically.

Fast File I/O. When you call file I/O functions, such as read() and write(), the data is copied to a kernel's intermediary buffer before it is transferred to the physical file or the process. This intermediary buffering is slow and expensive. Memory mapping eliminates this intermediary buffering, thereby improving performance significantly.

Uses of Memory Mapped Files

Streamlining file access. Once you map a file to a memory region, you access it via pointers, just as you would access ordinary variables and objects.

Memory sharing. Memory mapping enables different processes to share *physical* memory pages

Memory persistence. Memory mapping enables processes to share memory sections that persist independently of the lifetime of a certain process.

POSIX <sys/mman.h>

```
caddr_t mmap(
   caddress_t map_addr,
                        /* VM address hint;
                            0 for no preference */
   size_t length,       /* Length of file map*/
   int protection,      /* types of access*/
   int flags,           /*attributes*/
   int fd,              /*file descriptor*/
   off_t offset);       /*Offset file map start*/
```

Protection Attributes

PROT_READ /* the mapped region may be read */

PROT_WRITE /* the mapped region may be written */

PROT_EXEC /* the mapped region may be executed */

SIGSEGV signal if you reference memory with wrong protection mode.

Map first 4kb of file and read int

```
#include <errno.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/types.h>
int main(int argc, char *argv[])
{
        int fd;
        void * pregion;
        if (fd= open(argv[1], O_RDONLY) <0)
        {
                perror("failed on open");
                return –1;
        }
```

Map first 4kb of file and read int

```
        /*map first 4 kilobytes of fd*/
        pregion =mmap(NULL, 4096,  PROT_READ,
                              MAP_SHARED, fd, 0);
        if (pregion==(caddr_t)-1)
        {
                perror("mmap failed")
                return –1;
        }
        close(fd); /*close physical file: we don't need it */
        /* access mapped memory;
           read the first int in the mapped file */
        int val= *((int*) pregion);
}
```


munmap, msync

int **munmap**(*caddr_t addr, int length*);

int **msync** (*caddr_t addr, size_t length, int flags*);

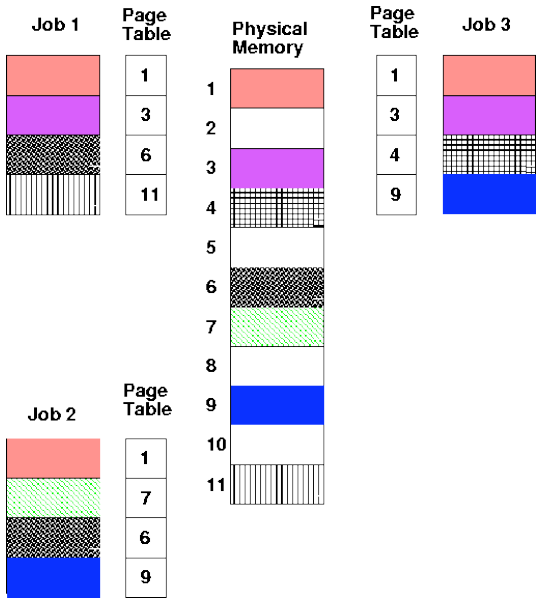
addr must be multiple of page size:

```
    size_t page_size = (size_t)
                sysconf (_SC_PAGESIZE);
```

Sharing Pages

- Code and data can be shared
  - Map common page frame in two processes
- Code, data must be *position-independent*
  - VM mappings for *same* code, data in different processes are different

Shared Pages



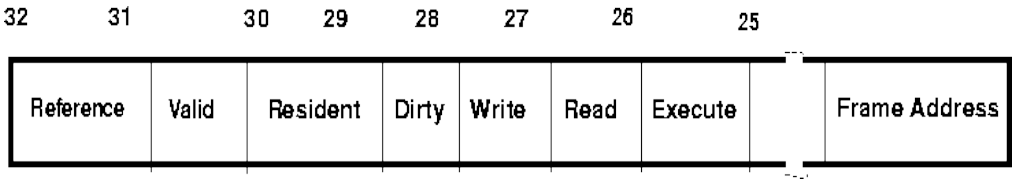
Protection

Why Page Protection?

Implementing Page Protection

- **R**ead, **W**rite, e**X**ecute bits in page table entry
- Check is done by hardware during access
  - Illegal access generates SIGSEGV
- Each process can have different protection bits

Page Protection via PTE



Legend:

Reference - page has been accessed

Valid - page exists

Resident - page is cached in primary memory

Dirty - page has been changed since page in

Virtual Memory Under Multiprogramming

Eviction of Virtual Pages

- On page fault: Choose VM page to page out
- How to choose which data to page out?

Allocation of Physical Page Frames

- How to assign frames to processes?

Terminology

- **Reference string**: memory reference sequence generated by a program:
  - Reference = (R/W, address)
- **Paging** – moving pages to or from disk
- **Demand Paging** – moving pages only when needed
- **Optimal** – the best (theoretical) strategy
- **Eviction** – throwing something out
- **Pollution** – bringing in useless pages/lines
-

Issue: Eviction

- Hopefully, kick out a less-useful page
  - Dirty pages require writing, clean pages don't
- Goal: kick out the page that's least useful
- Problem: how do you determine utility?
  - Heuristic: *temporal locality* exists
  - Kick out pages that aren't likely to be used again

Page Replacement Strategies

- **The Principle of Optimality**
  - Replace page that will be used the farthest in the future.
- **Random page replacement**
  - Choose a page randomly
- **FIFO - First in First Out**
  - Replace the page that has been in primary memory the longest
- **LRU - Least Recently Used**
  - Replace the page that has not been used for the longest time
- **LFU - Least Frequently Used**
  - Replace the page that has been used least often
- **NRU - Not Recently Used**
  - An approximation to LRU.
- **Working Set**
  - Keep in memory those pages that the process is actively using.

Principal of Optimality

- Description:
  - Assume each page can be labeled with number of references that will be executed before that page is first referenced.
  - Then the optimal page algorithm would choose the page with the highest label to be removed from the memory.
- Impractical! Why?

- Provides a basis for comparison with other schemes.
- If future references are known
  - should not use demand paging
  - should use “pre-paging” to overlap paging with computation.

Frame Allocation for Multiple Processes

- How are the page frames allocated to individual virtual memories of the various jobs running in a multi-programmed environment?
- Simple solution
  - Allocate a minimum number (??) of frames per process.
    - One page from the current executed instruction
    - Most instructions require two operands
    - include an extra page for paging out and one for paging in

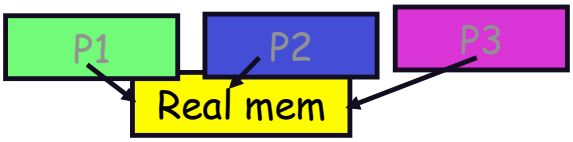
Multi-Programming Frame Allocation

- Solution 2
  - allocate an equal number of frames per job
    - but jobs use memory unequally
    - high priority jobs have same number of page frames as low priority jobs
    - degree of multiprogramming might vary
- Solution 3:
  - allocate a number of frames per job proportional to job size
    - how do you determine job size: by run command parameters or dynamically?

Multi-Programming Frame Allocation

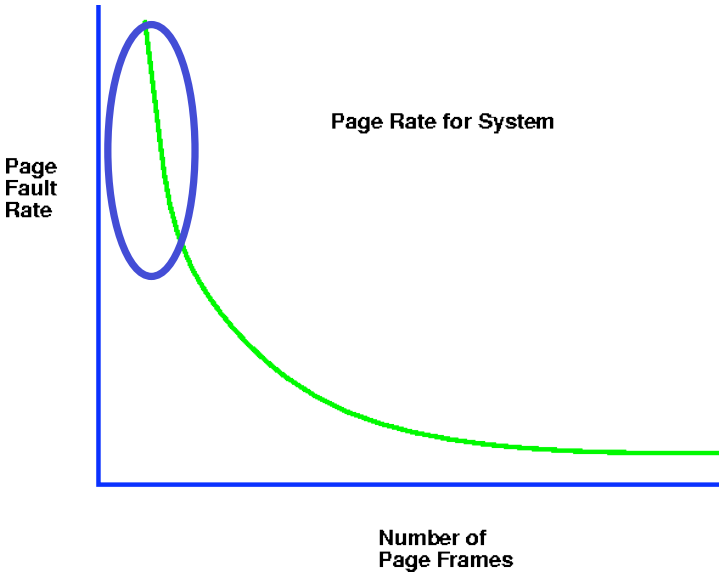
- Why is multi-programming frame allocation is important?
  - If not solved appropriately, it will result in a severe problem--- Thrashing

Thrashing

- Thrashing: As page frames per VM space decrease, the page fault rate increases.
- 

  - Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out.
  - Processes will spend all of their time blocked, waiting for pages to be fetched from disk
  - I/O devs at 100% utilization but system not getting much useful work done
  - Memory and CPU mostly idle

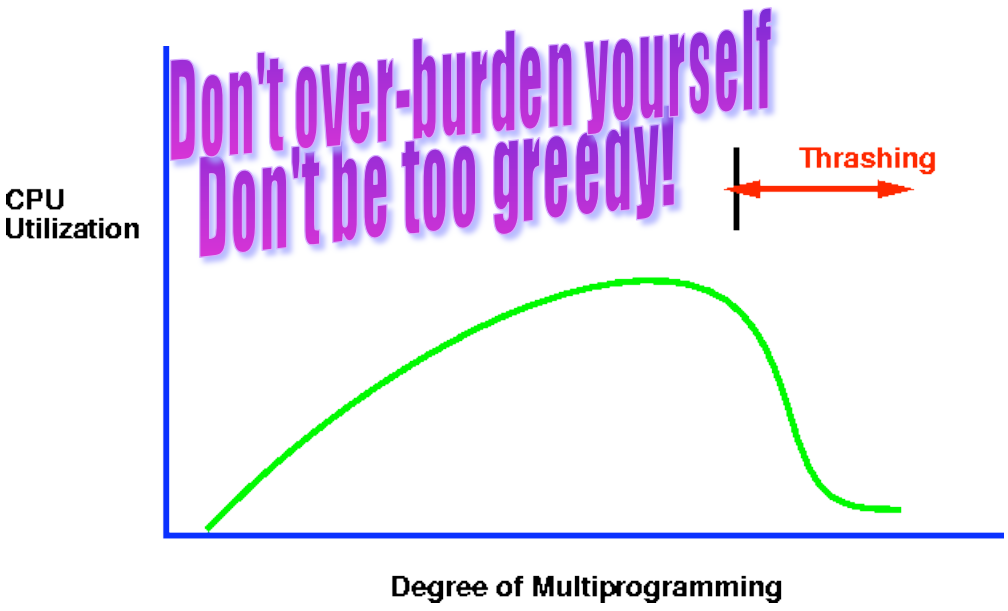
Page Fault Rate vs. Size Curve



Why Thrashing?

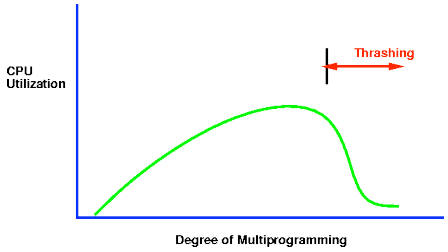
- Computations have locality
- As page frames decrease, the page frames available are not large enough to contain the locality of the process.
- The processes start faulting heavily
  - Pages that are read in, are used and immediately paged out.

Results of Thrashing



Why?

- As the page fault rate goes up, processes get suspended on page out queues for the disk.
- The system may start new jobs.
- Starting new jobs will reduce the number of page frames available to each process, increasing the page fault requests.
- System throughput plunges.



Solution: Working Set

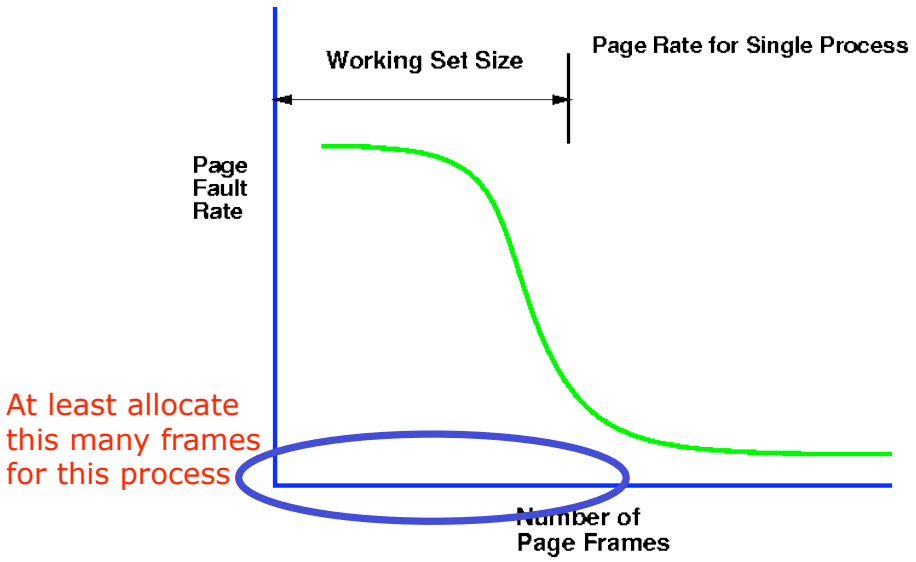
- Main idea
  - figure out how much memory a process needs to keep most of its recent computation in memory with very few page faults
- How?
  - The working set model assumes locality
  - the principle of locality states that a program clusters its access to data and text in time
    - Recently accessed page is more likely to be accessed again than less recently accessed page
  - Thus, as the number of page frames increases above some threshold, the page fault rate will drop dramatically

Working set (1968, Denning)

- What we want to know: collection of pages process must have in order to avoid thrashing
  - This requires knowing the future. And our trick is?
- Working set:
  - Pages referenced by process in last T seconds of execution considered to comprise its working set
  - T : the working set parameter

- Usages of working set sizes?
  - Cache partitioning: give each app enough space for WS
  - Page replacement: preferentially discard non-WS pages
  - Scheduling: process not executed unless WS in memory

Working Set



Calculating Working Set

Window size is Δ

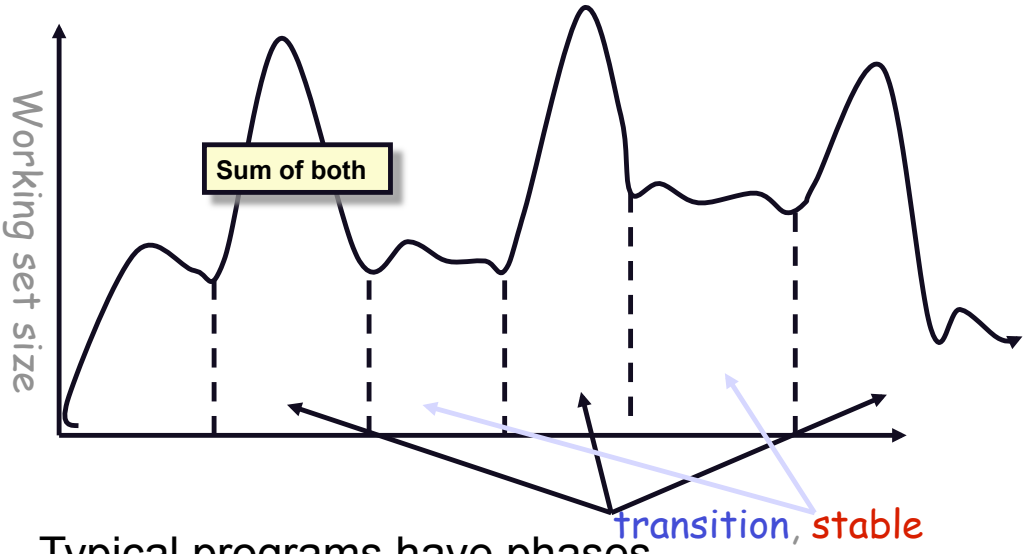
12 references, 8 faults

Page Refs	$\Delta = 4$ References					
	Fault?	Page Contents				
A	yes	A				
B	yes	A	B			
C	yes	A	B	C		
D	yes	A	B	C	D	
A	no	A	B	C	D	
B	no	A	B	C	D	
E	yes	A	B	D	E	
A	no	A	B	E		
B	no	A	B	E		
C	yes	A	B	C	E	
D	yes	A	B	C	D	
E	yes	B	C	D	E	

Working Set in Action to Prevent Thrashing

- Algorithm
- if #free page frames > working set of some suspended $process_i$, then activate $process_i$ and map in all its working set
- if working set size of some $process_k$ increases and no page frame is free, suspend $process_k$ and release all its pages

Working sets of real programs



- Typical programs have phases

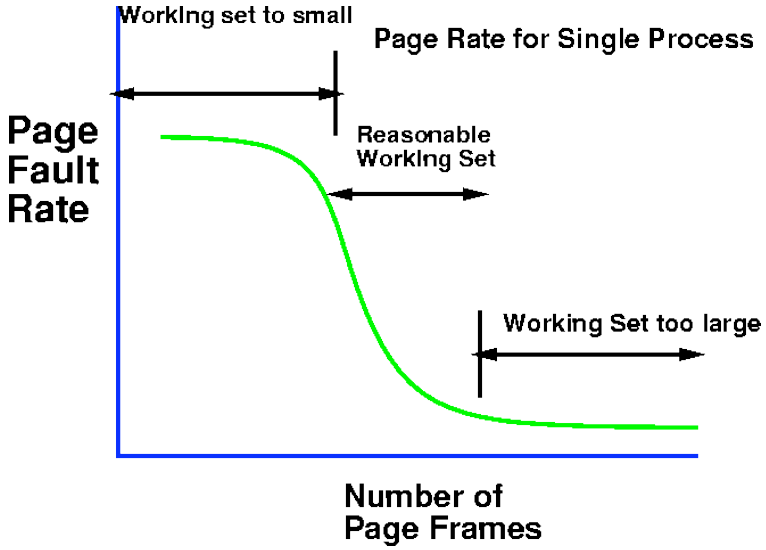
Working Set Implementation Issues

- Moving window over reference string used for determination
- Keeping track of working set

Page Fault Frequency Working Set

- Approximation of pure working set
  - Assume that if the working set is correct there will not be many page faults.
  - If page fault rate increases beyond assumed knee of curve, then increase number of page frames available to process.
  - If page fault rate decreases below foot of knee of curve, then decrease number of page frames available to process.

Page Fault Frequency Working Set



Summary

- Memory mapped files
- Page sharing
- Page protection
- Virtual Memory and Multiprogramming
  - Page eviction policies
  - Page frame allocation and thrashing
  - Working set model and implementation