Memory Implementation Issues

Lecture 33

Klara Nahrstedt

CS241 Administrative

- Read Stallings Chapter 8.1 and 8.2 about VM
- LMP3 starts today
- Start Early!!!

Contents

Brief Discussion of Second Chance Replacement Algorithm

Paging basic process implementation

Frame allocation for multiple processes

Thrashing

Working Set

Memory-Mapped Files

Second Chance Example

12 references,
9 faults

Page Refs	3 Page Frames			
	Fault?	Page Contents		
A	yes	A•		
B	yes	B•	A•	
C	yes	C•	B•	A•
D	yes	D•	C	B
A	yes	A•	D•	C
B	yes	B•	A•	D•
E	yes	E•	B	A
A	no	E•	B	A•
B	no	E•	B•	A•
C	yes	C•	E	B
D	yes	D•	C•	E
E	no	D•	C•	E•

Basic Paging Process Implementation(1)

Separate page out from page in

- Keep a pool of free frames
- when a page is to be replaced, use a free frame
- read the faulting page and restart the faulting process while page out is occurring

Why?

- Alternative: Before a frame is needed to read in the faulted page from disk, just evict a page
- Disadvantage with alternative:
  - A page fault may require 2 disk accesses: 1 for writing-out the evicted page, 1 for reading in the faulted page

Basic Paging Process Implementation(2)

Paging out

- Write dirty pages to disk whenever the paging device is free and reset the dirty bit
- Benefit?
  - Remove the paging out (disk writes) process from the critical path
  - allows page replacement algorithms to replace clean pages

What should we do with paged out pages?

- Cache paged out pages in primary memory (giving it a second chance)
  - Return paged-out pages to a free pool but remember which page frame they are.
  - If system needs to map page in again, reuse page.

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 and low priority jobs
  - degree of multiprogramming might vary

Multi-Programming Frame Allocation

Solution 3:

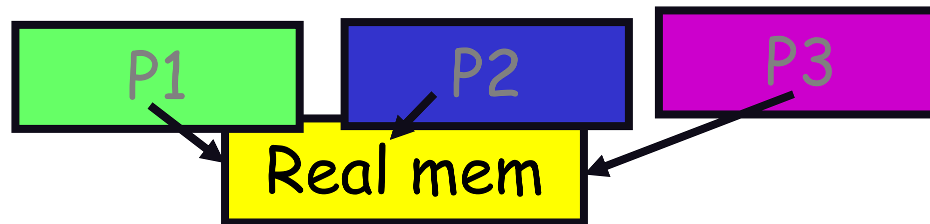
- allocate a number of frames per job proportional to job size
  - how do you determine job size: by run command parameters or dynamically?

Why multi-programming frame allocation is important?

- If not solved appropriately, it will result in a severe problem--- Thrashing

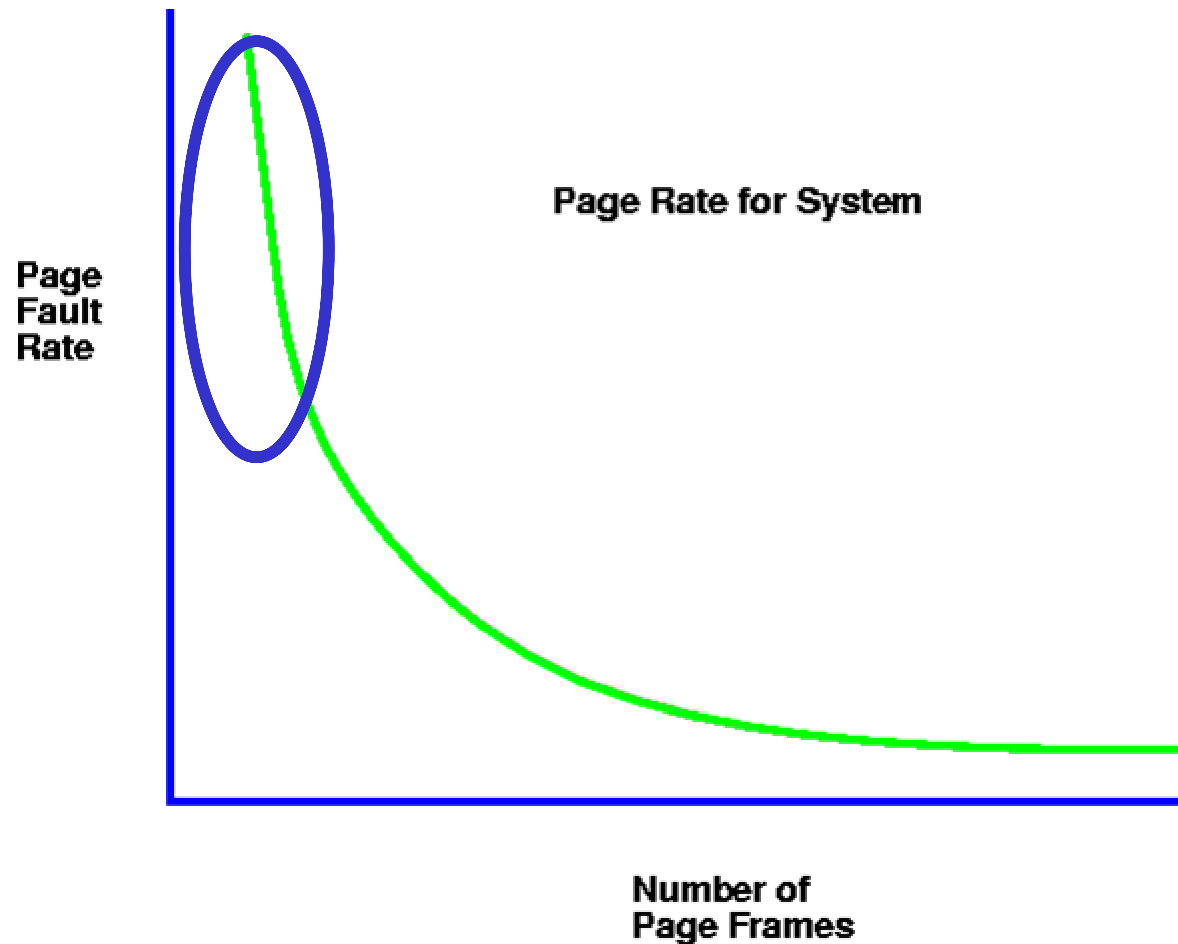
Thrashing: exposing the lie of VM

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

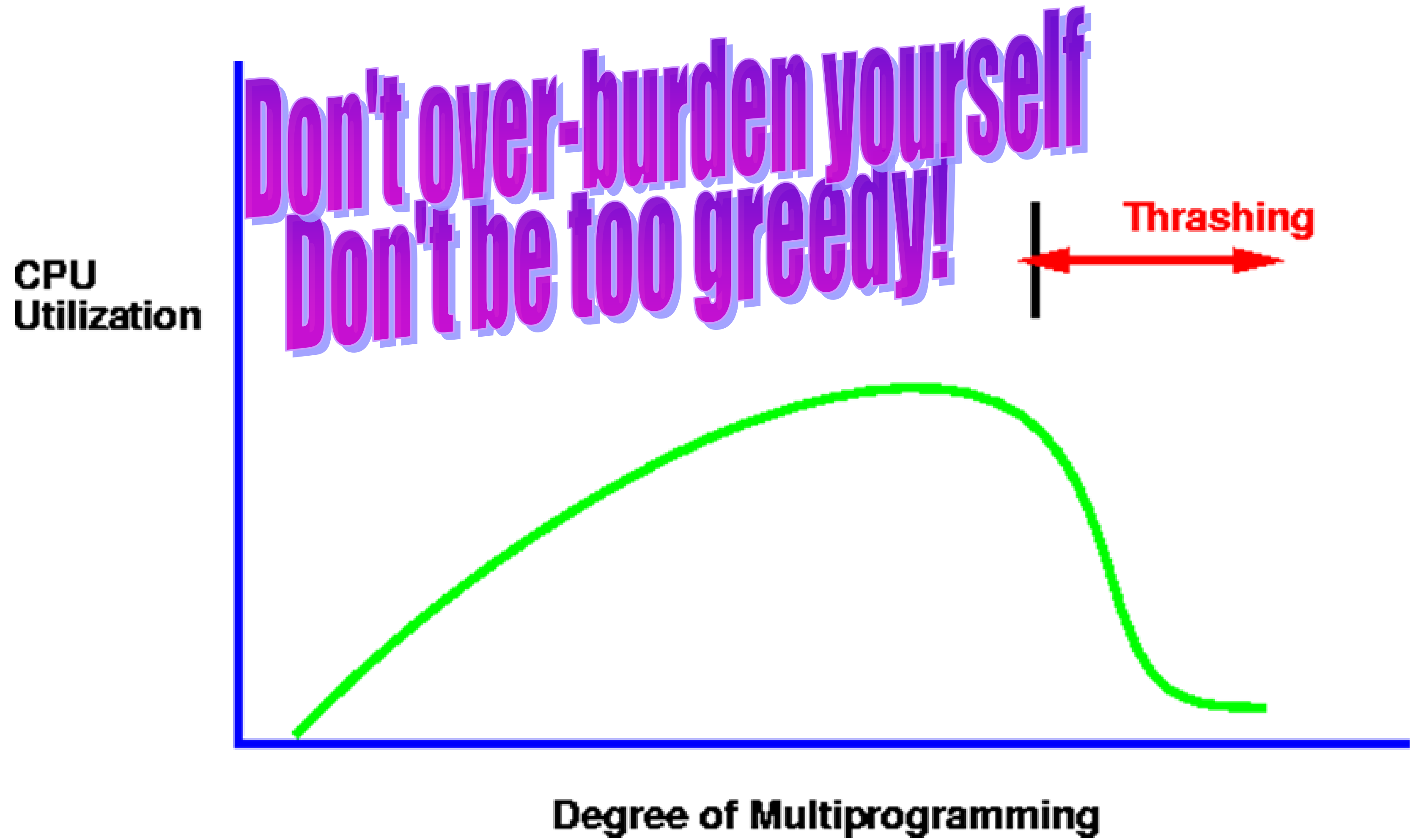
Page Fault Rate vs. Size Curve



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



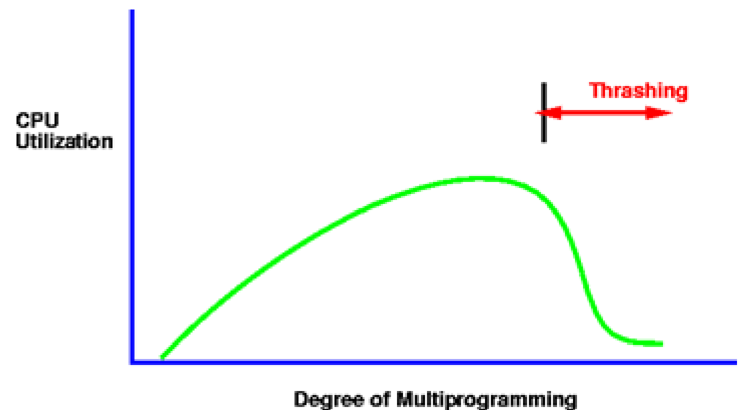
Why?

As the page fault rate goes up, processes get suspended on page out queues for the disk.

The system may try to optimize performance by starting 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 does a process need to keep most the 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 temporally
  - A recently accessed page is more likely to be accessed again
- 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?

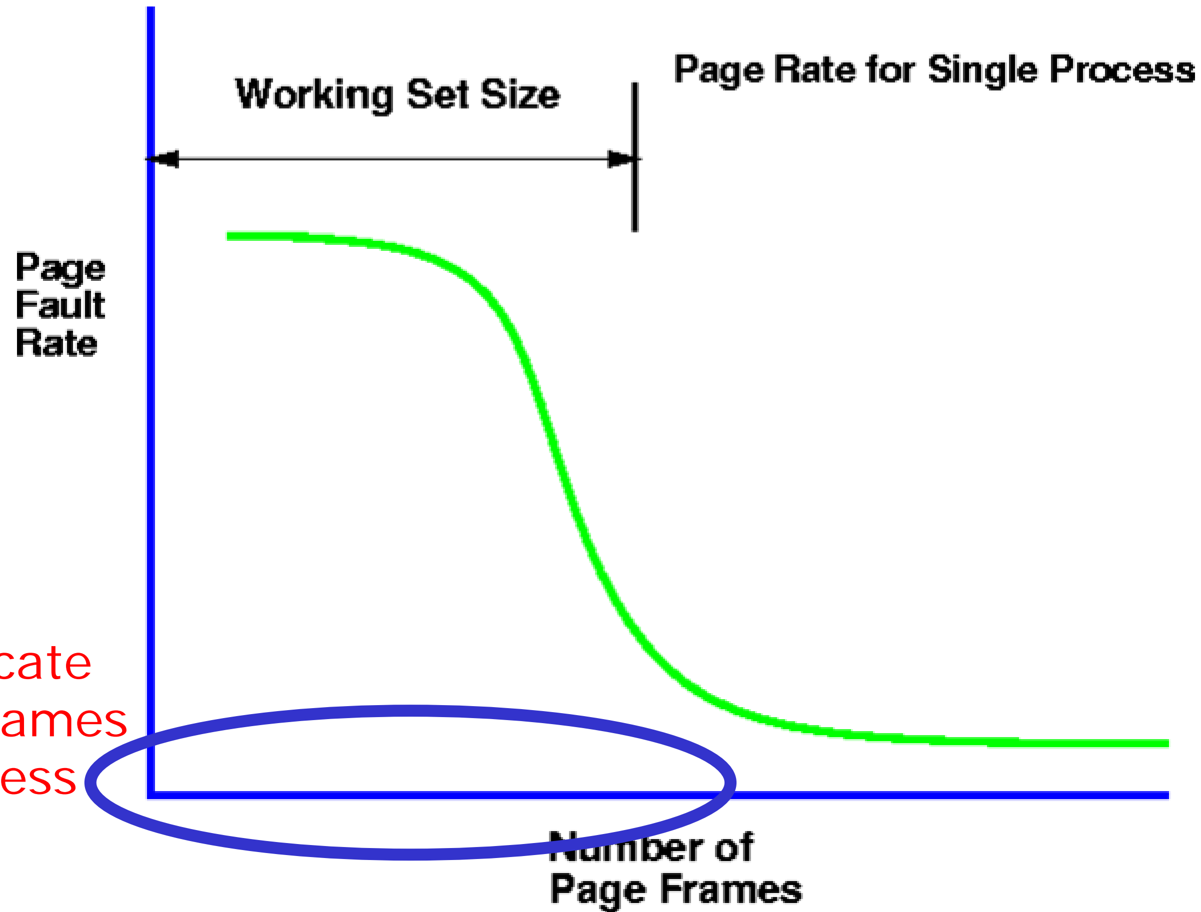
Working set:

- 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



At least allocate this many frames for this process

Calculating Working Set

Window size is $\triangle$

12 references,
8 faults

Page Refs	$\triangle = 4$ 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

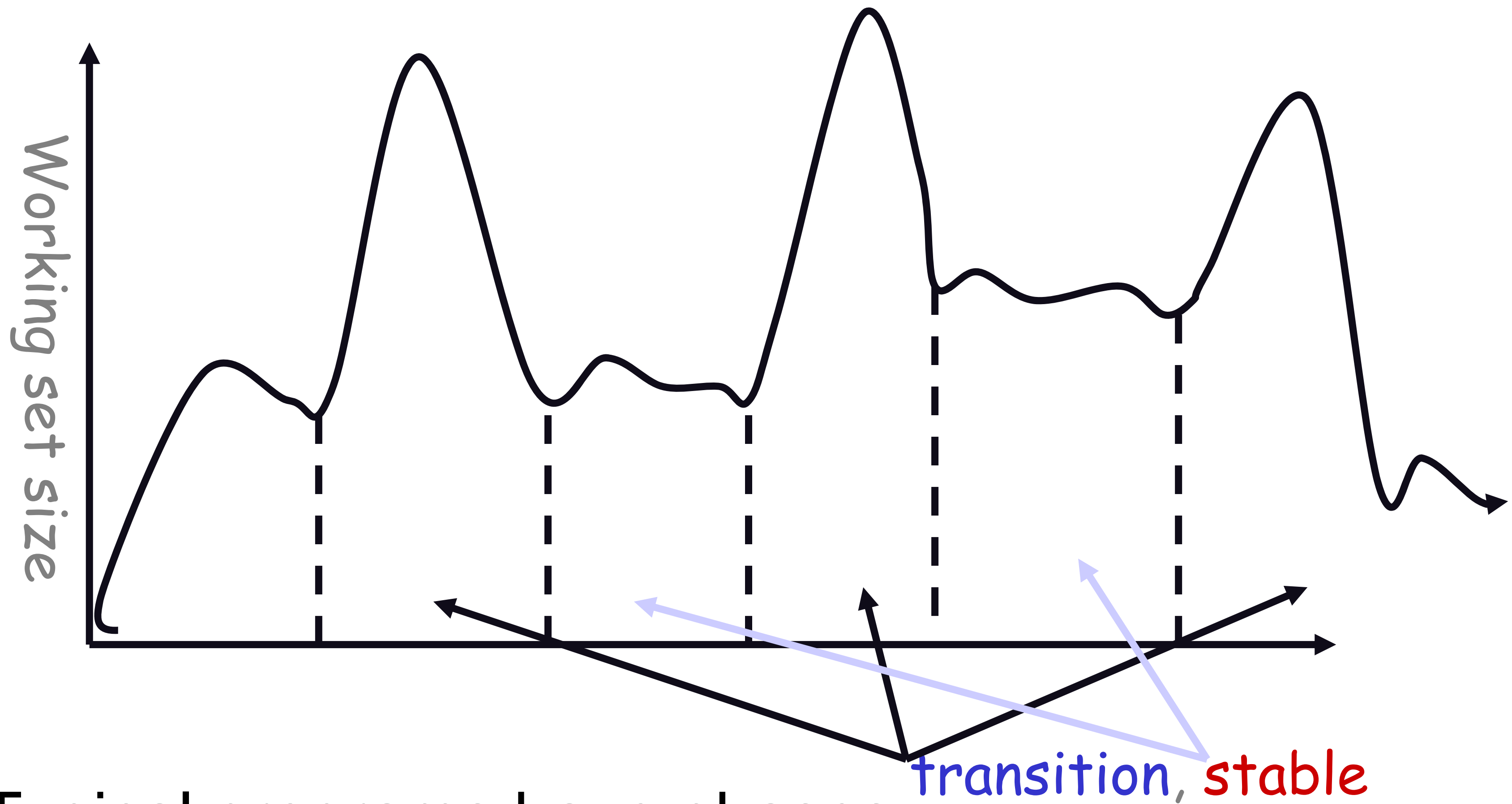
Working Set in Action to Prevent Thrashing

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

Working Set Implementation

Approximate working set model using timer and reference bit

Set timer to interrupt after approximately x references, τ.

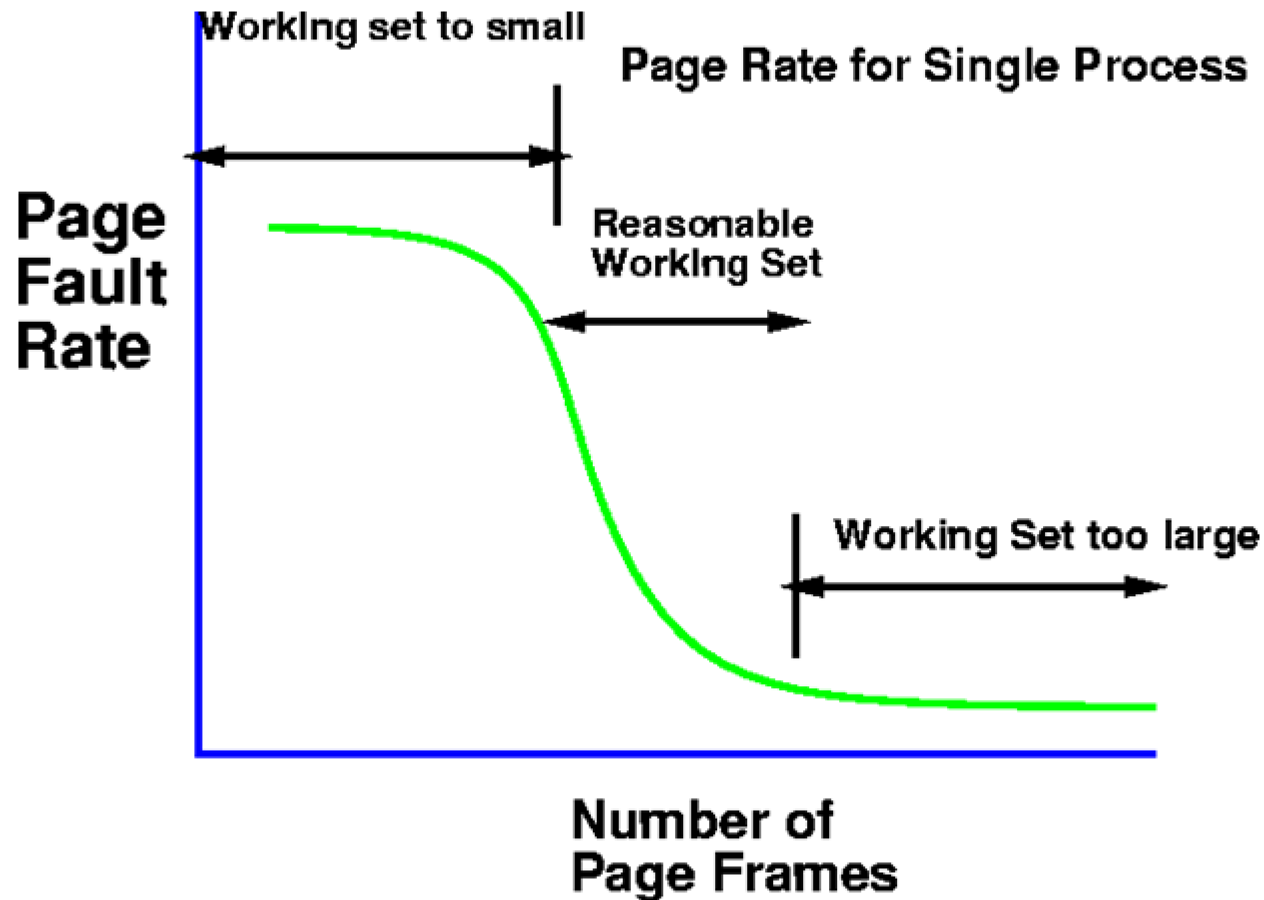
Remove pages that have not been referenced and reset reference bit.

Page Fault Frequency Working Set

Another 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



Page Size Considerations

small pages require large page tables

large pages imply significant amounts of page may not be referenced

locality of reference tends to be small (256), implying small pages

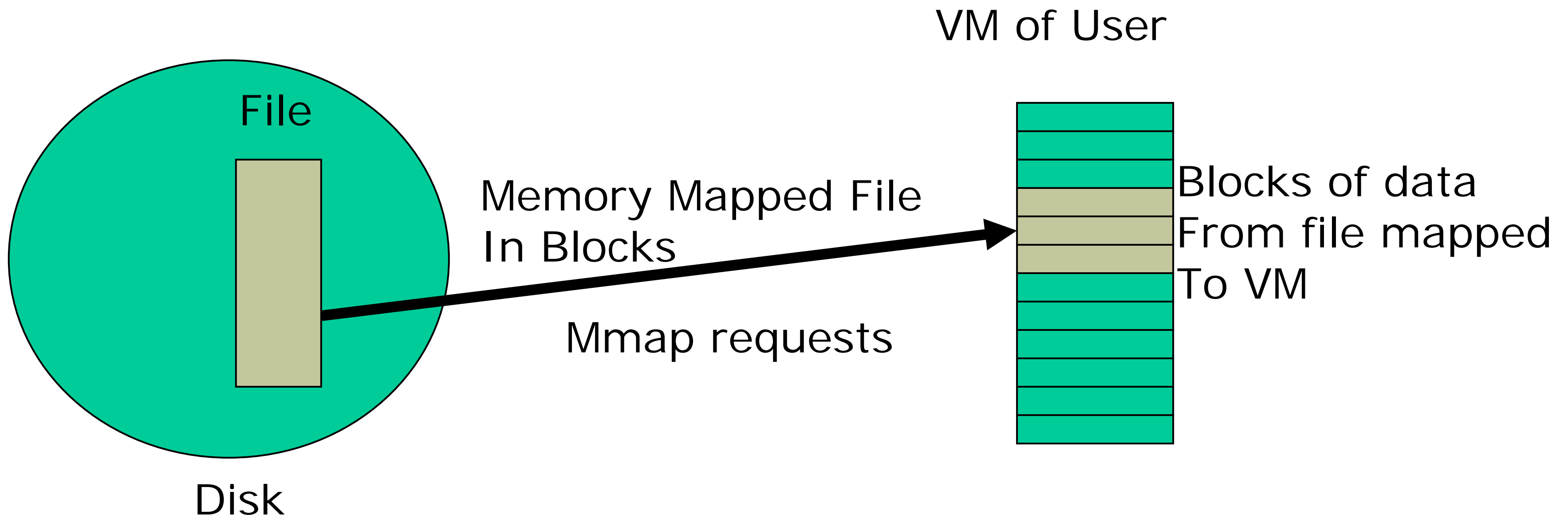
i/o transfers have high seek time, implying larger pages. (more data per seek.)

internal fragmentation minimized with small page size

Real systems (can be reconfigured)

- Windows: default 8KB
- Linux: default 4 KB

Memory Mapped Files



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.

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 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,

/* map_addr is VM address to map file, use 0 to allow system to choose*/

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 */

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 the physical file because we don't need it */
/*access mapped memory; read the first int in the mapped file */
int val= *((int*) pregion);
}
```

munmap

int munmap(caddr_t addr, int length);

int **msync** (*void *address, size_t length, int flags*)

size_t page_size = (size_t) sysconf (_SC_PAGESIZE);

SIGSEGV signal allows you to catch references to memory that have the wrong protection mode.

Summary

Second Chance Replacement Policy

Paging basic implementation

Multiprogramming frame allocation

- Thrashing
- Working set model
- Working set implementation

Page size consideration

Memory-Mapped Files