CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu

Lecture 11: Page Replacement and Frame Allocation
March 13, 2007

© 2007 Matt Welsh – Harvard University

Today: Page Replacement

How do we decide which pages to kick out of physical memory when memory is tight?

How do we decide how much physical memory to allocate to a process?

© 2007 Matt Welsh – Harvard University

Benefits of sharing pages

How much memory savings do we get from sharing pages across identical processes?

- A lot! Use the “top” command...

```
Processes:  69 total, 2 running, 67 sleeping... 233 threads            10:08:27
Load Avg:  0.24, 0.33, 0.27     CPU usage:  20.0% user, 40.0% sys, 40.0% idle
SharedLibs: num =  206, resident = 43.2M code, 5.37M data, 8.23M LinkEdit
MemRegions: num = 10896, resident =  501M + 18.4M private,  216M shared
PhysMem:   164M wired,  567M active,  271M inactive, 1004M used, 19.9M free
VM: 13.9G +  129M   284113(0) pageins, 59113(0) pageouts
```

PID	COMMAND	%CPU	TIME	#TH	#PRTS	#MREGS	RPRVT	RSHRD	RSIZE	VSIZE
2481	soffice.bi	0.0%	1:30.76	14	419	1187	189M	86.3M	255M	1.18G
151	WindowServ	0.0%	16:29.84	3	431	1077	12.4M	84.0M	88.4M	466M
293	VirtueDesk	0.0%	0:54.26	1	76	321	8.98M	37.7M	17.0M	407M
2456	firefox-bi	0.0%	3:40.82	7	105	426	105M	34.8M	129M	495M
2399	Mail	0.0%	0:59.08	8	172	338	22.3M	19.6M	33.0M	389M
153	ATSServer	0.0%	0:11.14	2	142	170	1.86M	19.0M	8.45M	84.7M
234	Finder	0.0%	0:18.20	3	125	261	6.49M	18.9M	20.9M	364M
295	Quicksilve	0.0%	2:16.48	4	125	415	31.3M	15.5M	48.3M	390M
1165	iTunes	0.0%	4:45.66	7	285	337	36.3M	12.9M	38.0M	411M
308	DashboardC	0.0%	0:29.26	3	98	175	4.22M	12.8M	15.0M	349M
1508	Preview	0.0%	0:08.84	1	70	226	6.32M	11.6M	13.2M	347M

On my machine, 555 MB out of 982 MB in use are shared across processes (almost 57%!)

© 2007 Matt Welsh – Harvard University

Paging and swapping

However, on heavily-loaded systems, memory can fill up

- On ice.fas, 1022 MB out of 1024 MB are currently in use...

To achieve good performance, must move "inactive" pages out to disk

  - *If we didn't do this, what options would the system have if memory is full???*
- What constitutes an "inactive" page?
- How do we choose the right set of pages to copy out to disk?
- How do we decide when to move a page back into memory?

Swapping

- Usually refers to moving the memory for an entire process out to disk
- This effectively puts the process to sleep until OS decides to swap it back in

Paging

- Refers to moving individual pages out to disk (and back)
- We often use the terms "paging" and "swapping" interchangeably

© 2007 Matt Welsh – Harvard University

Page eviction and locality

When do we decide to evict a page from memory?

-
-
-

Page eviction and locality

When do we decide to evict a page from memory?

- Usually, at the same time that we are trying to allocate a new physical page
- However, the OS keeps a pool of "free pages" around, even when memory is tight, so that allocating a new page can be done quickly
- The process of evicting pages to disk is then performed *in the background*

Exploiting locality

- **Temporal locality**: Memory accessed recently tends to be accessed again soon
- **Spatial locality**: Memory locations *near* recently-accessed memory is likely to be referenced soon
  - *Why would you expect to see these patterns in real programs???*

Locality helps to reduce the frequency of paging

- Once something is in memory, it should be used many times

This depends on many things:

- The amount of locality and reference patterns in a program
- The *page replacement policy*
- The amount of physical memory and the *application footprint*

© 2007 Matt Welsh – Harvard University

Evicting the best page

Goal of the page replacement algorithm:

- Reduce **page fault rate** by selecting the “best” page to evict

The “best” pages are those that will never be used again

- However, it's impossible to know in general whether a page will be touched
- *If you happened to have information on future access patterns*, you can *prove* that evicting those pages that will be used the *furthest in the future* will *minimize* the page fault rate

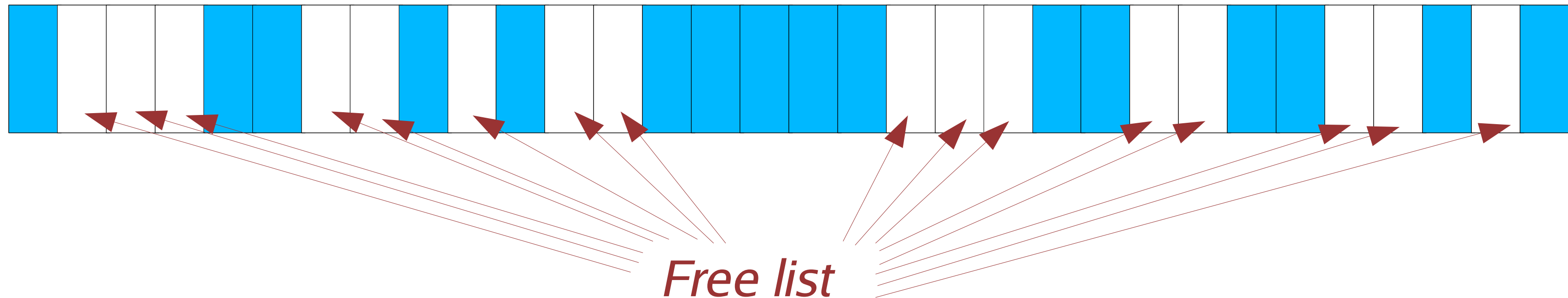
What is the best algorithm for deciding the order to evict pages?

- Much attention has been paid to this problem.
- Used to be a very hot research topic.
- These days, widely considered solved (at least, solved well enough)

© 2007 Matt Welsh – Harvard University

Page Replacement Basics

Most page replacement algorithms operate on some data structure that represents physical memory:



- Might consist of a bitmap, one bit per physical page
- Might be more involved, e.g., a reference count for each page (more soon!!)
- Free list consists of pages that are unallocated

Several ways of implementing this data structure

- Scan all process PTEs that correspond to mapped pages (valid bit == 1)
- Keep separate list or bitmap of physical pages
- **Inverted page table**: One entry per physical page, each entry points to PTE

© 2007 Matt Welsh – Harvard University

Algorithm #1: OPT (a.k.a MIN)

Evict page that won't be used for the longest time in the future

- Of course, this requires that we can see into the future...
- So OPT cannot be implemented!

This algorithm has the provably optimal performance

- Hence the name “OPT”
- Also called “MIN” (for “minimal”)

OPT is useful as a “yardstick” to compare the performance of other (implementable) algorithms against

© 2007 Matt Welsh – Harvard University

Algorithms #2 and 3: Random and FIFO

Random: Throw out a random page

- Obviously not the best scheme
- Although very easy to implement!

FIFO: Throw out pages in the order that they were allocated

- Maintain a list of allocated pages
- When the length of the list grows to cover all of physical memory, pop first page off list and allocate it

Why might FIFO be good?

-

Why might FIFO not be so good?

-

© 2007 Matt Welsh – Harvard University

Algorithms #2 and 3: Random and FIFO

Random: Throw out a random page

- Obviously not the best scheme
- Although very easy to implement!

FIFO: Throw out pages in the order that they were allocated

- Maintain a list of allocated pages
- When the length of the list grows to cover all of physical memory, pop first page off list and allocate it

Why might FIFO be good?

- Maybe the page allocated very long ago isn't being used anymore

Why might FIFO not be so good?

- Doesn't consider locality of reference!
- Suffers from *Belady's Anomaly*: Performance of an application might get *worse* as the size of physical memory *increases!!!*

© 2007 Matt Welsh – Harvard University

Belady's Anomaly

time →

Access pattern	0	1	2	3	0	1	4	0	1	2	3	4
Physical memory (3 page frames)	0	0	0	1	2	3	0	0	0	1	4	4
		1	1	2	3	0	1	1	1	4	2	2
			2	3	0	1	4	4	4	2	3	3

9 page faults!

time →

Access pattern	0	1	2	3	0	1	4	0	1	2	3	4
Physical memory (4 page frames)	0	0	0	0	0	0	1	2	3	4	0	1
		1	1	1	1	1	2	3	4	0	1	2
			2	2	2	2	3	4	0	1	2	3
				3	3	3	4	0	1	2	3	4

10 page faults!

© 2007 Matt Welsh – Harvard University

Algorithm #4: Least Recently Used (LRU)

Evict the page that was used the **longest time ago**

- Keep track of when pages are referenced to make a better decision
- Use past behavior to predict future behavior
  - *LRU uses past information, while MIN uses future information*
- When does LRU work well, and when does it not?

Implementation

- Every time a page is accessed, record a *timestamp* of the access time
- When choosing a page to evict, scan over all pages and throw out page with oldest timestamp

Problems with this implementation?

© 2007 Matt Welsh – Harvard University

Algorithm #4: Least Recently Used (LRU)

Evict the page that was used the **longest time ago**

- Keep track of when pages are referenced to make a better decision
- Use past behavior to predict future behavior
  - *LRU uses past information, while MIN uses future information*
- When does LRU work well, and when does it not?

Implementation

- Every time a page is accessed, record a *timestamp* of the access time
- When choosing a page to evict, scan over all pages and throw out page with oldest timestamp

Problems with this implementation?

- 32-bit timestamp for each page would double the size of every PTE
- Scanning all of the PTEs for the lowest timestamp would be slow
- So, we need an approximation!

Approximating LRU

Use the PTE reference bit and a small **counter** per page

- (Use a counter of, say, 2 or 3 bits in size, and store it in the PTE)

Periodically, scan all physical pages in the system

- If the page has not been accessed (PTE reference bit == 0), **increment** the counter
- If the page has been accessed (reference bit == 1), **set counter to zero**
- **Clear** the PTE reference bit in either case!

Counter will contain the number of scans since the last reference to this page.

- PTE that contains the highest counter value is the least recently used
- So, **evict the page with the highest counter**

© 2007 Matt Welsh – Harvard University

LRU example

time

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Accessed pages in blue

0 1 1 1 0 0 1 1 0 1 0 1 1 0 0

Increment counter for untouched pages

0 1 1 1 0 0 1 1 0 1 0 1 1 0 0

0 2 0 0 0 1 2 2 0 0 1 0 2 1 0

These pages have the highest counter value and can be evicted.

© 2007 Matt Welsh – Harvard University

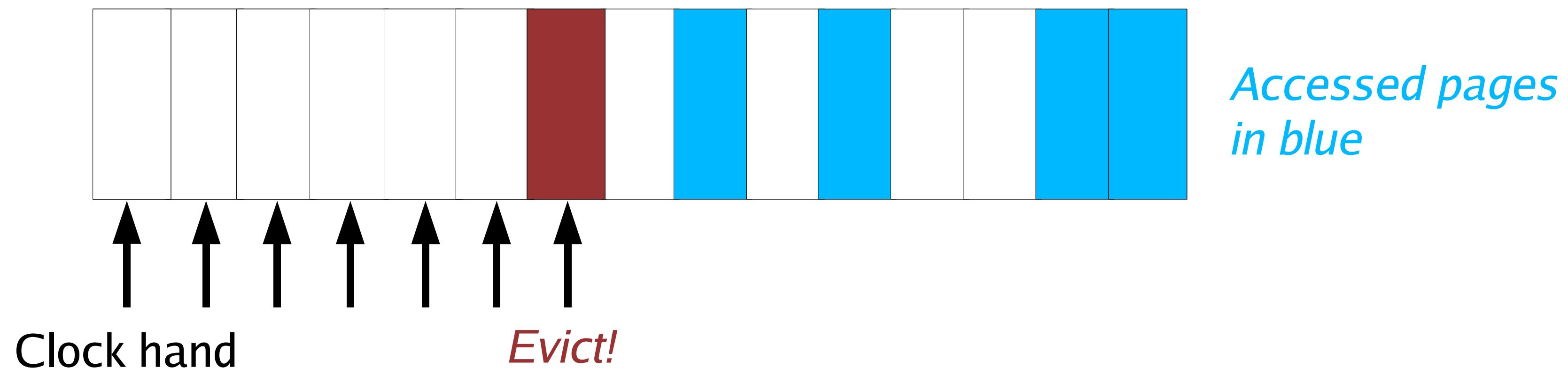
Algorithm #5: LRU Clock

LRU requires searching for the page with the highest last-ref count

- Can do this with a sorted list or a second pass to look for the highest value

Simpler technique: Clock algorithm

- "Clock hand" scans over all physical pages in the system
    - *Clock hand loops around to beginning of memory when it gets to end*
- If PTE reference bit == 1, **clear bit** and **advance hand**
- If PTE reference bit == 0, **evict** this page
    - *No need for a counter in the PTE!*



© 2007 Matt Welsh – Harvard University

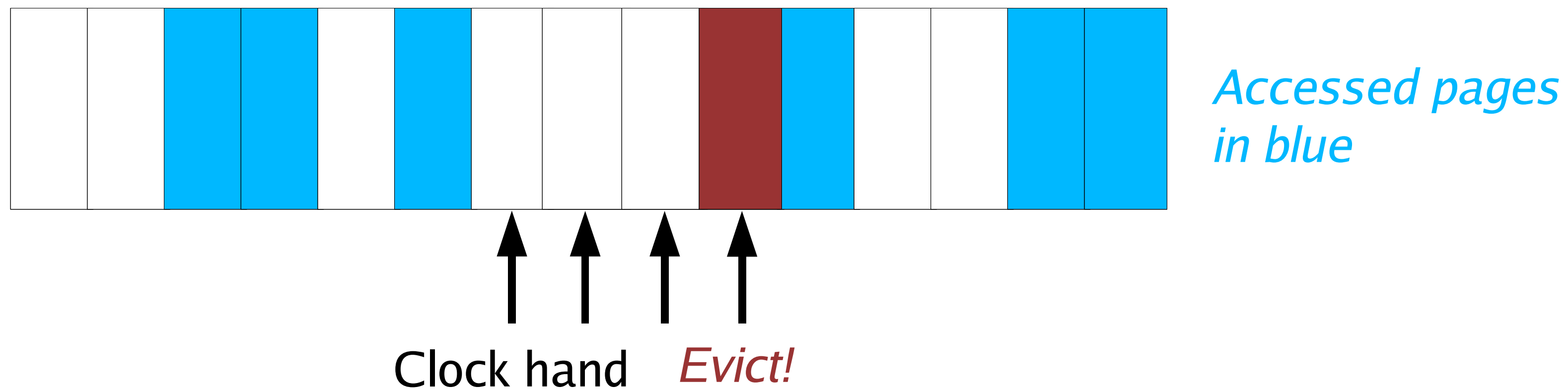
Algorithm #5: LRU Clock

LRU requires searching for the page with the highest last-ref count

- Can do this with a sorted list or a second pass to look for the highest value

Simpler technique: Clock algorithm

- "Clock hand" scans over all physical pages in the system
  - *Clock hand loops around to beginning of memory when it gets to end*
- If PTE reference bit == 1, **clear bit** and **advance hand**
- If PTE reference bit == 0, **evict** this page
  - *No need for a counter in the PTE!*



© 2007 Matt Welsh – Harvard University

Algorithm #5: LRU Clock

This is a lot like LRU, but operates in an iterative fashion

- To find a page to evict, just start scanning from current clock hand position
- What happens if all pages have ref bits set to 1?
- What is the *minimum* “age” of a page that has the ref bit set to 0?

Slight variant -- “nth chance clock”

- Only evict page if hand has swept by N times
- Increment per-page counter each time hand passes and ref bit is 0
- Evict a page if counter >= N
- Counter cleared to 0 each time page is used

© 2007 Matt Welsh – Harvard University

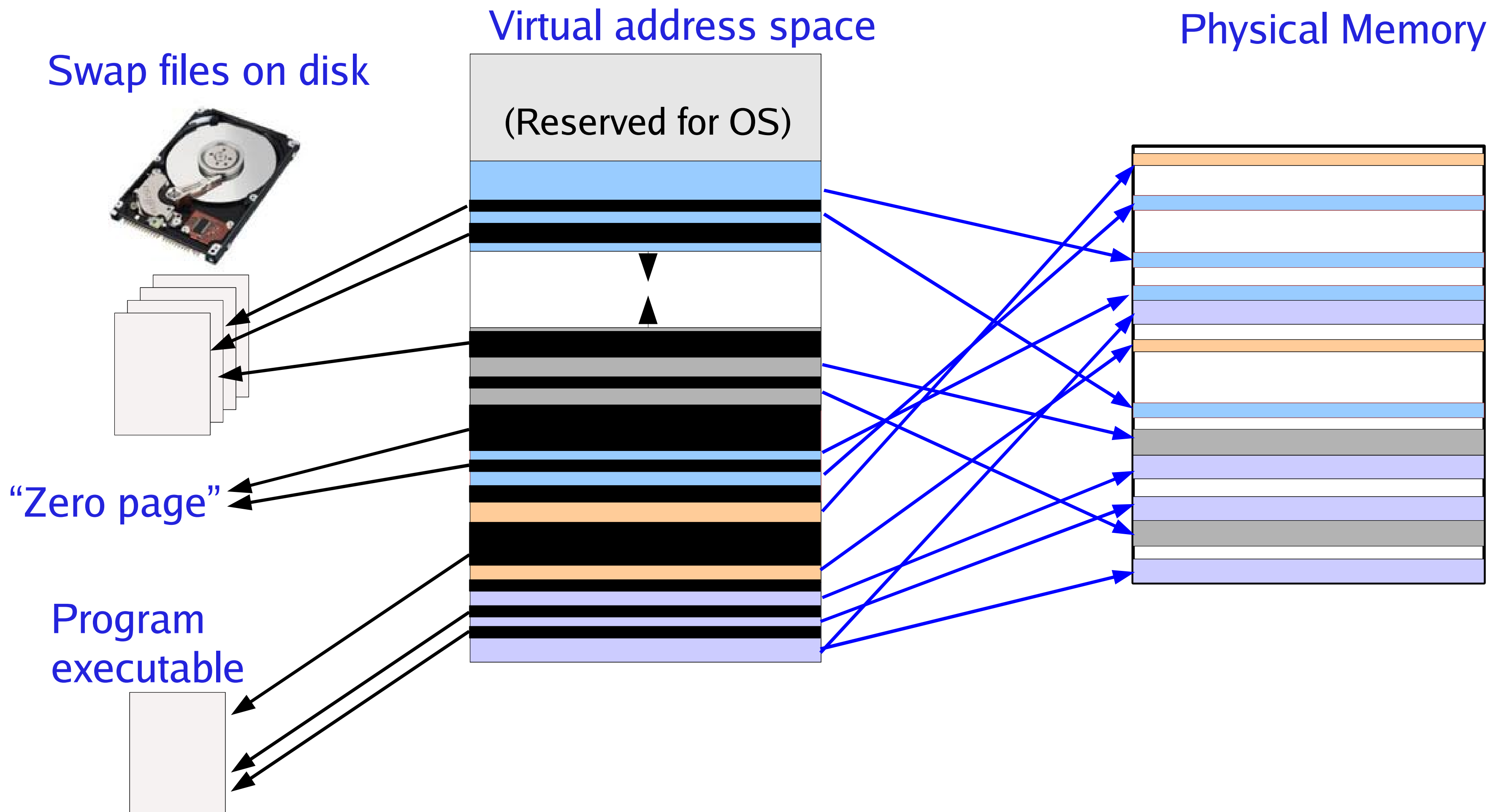
Swap Files

What happens to the page that we choose to evict?

- Depends on what kind of page it is and what state it's in!

OS maintains one or more **swap files** or partitions on disk

- Special data format for storing pages that have been swapped out

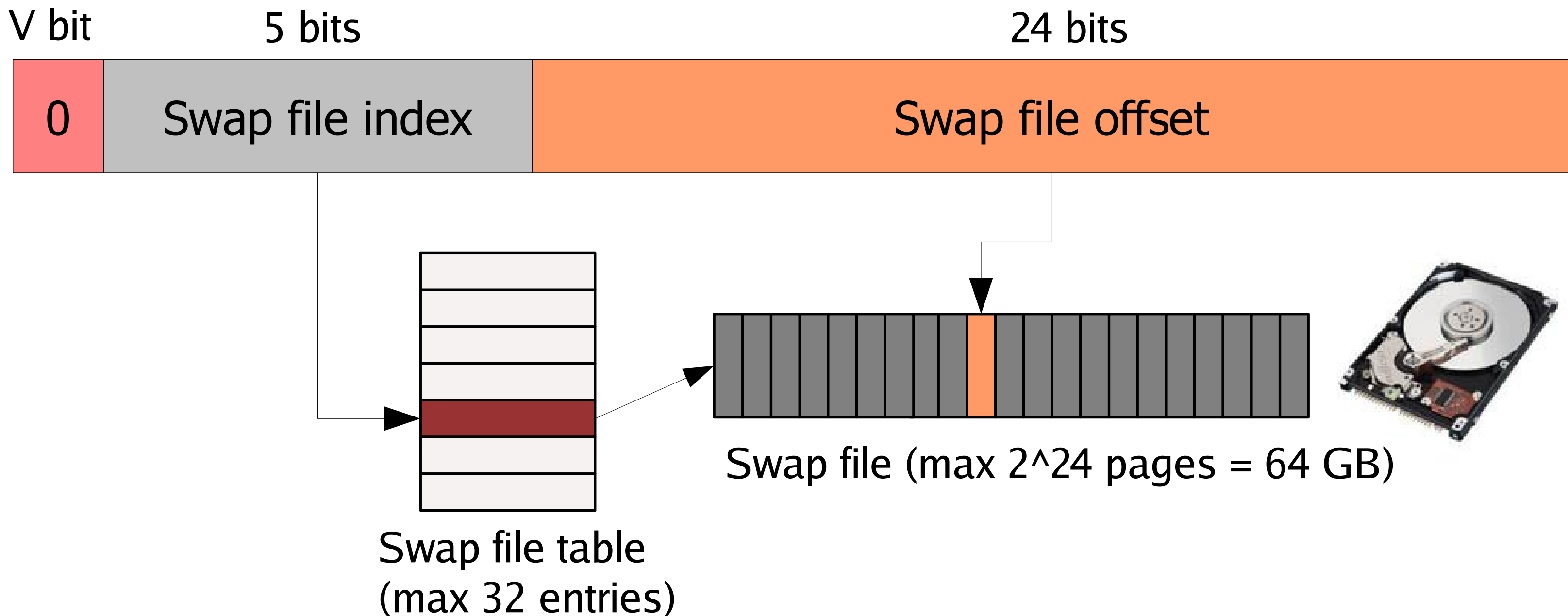


© 2007 Matt Welsh – Harvard University

Swap Files

How do we keep track of where pages are located on disk?

- Recall PTE format
- When V bit is 0, can recycle the PFN field to remember something about the page.



But ... not all pages are swapped in from swap files!

- What about executables?
- Or “zero pages”?
- How do we deal with these file types?

© 2007 Matt Welsh – Harvard University

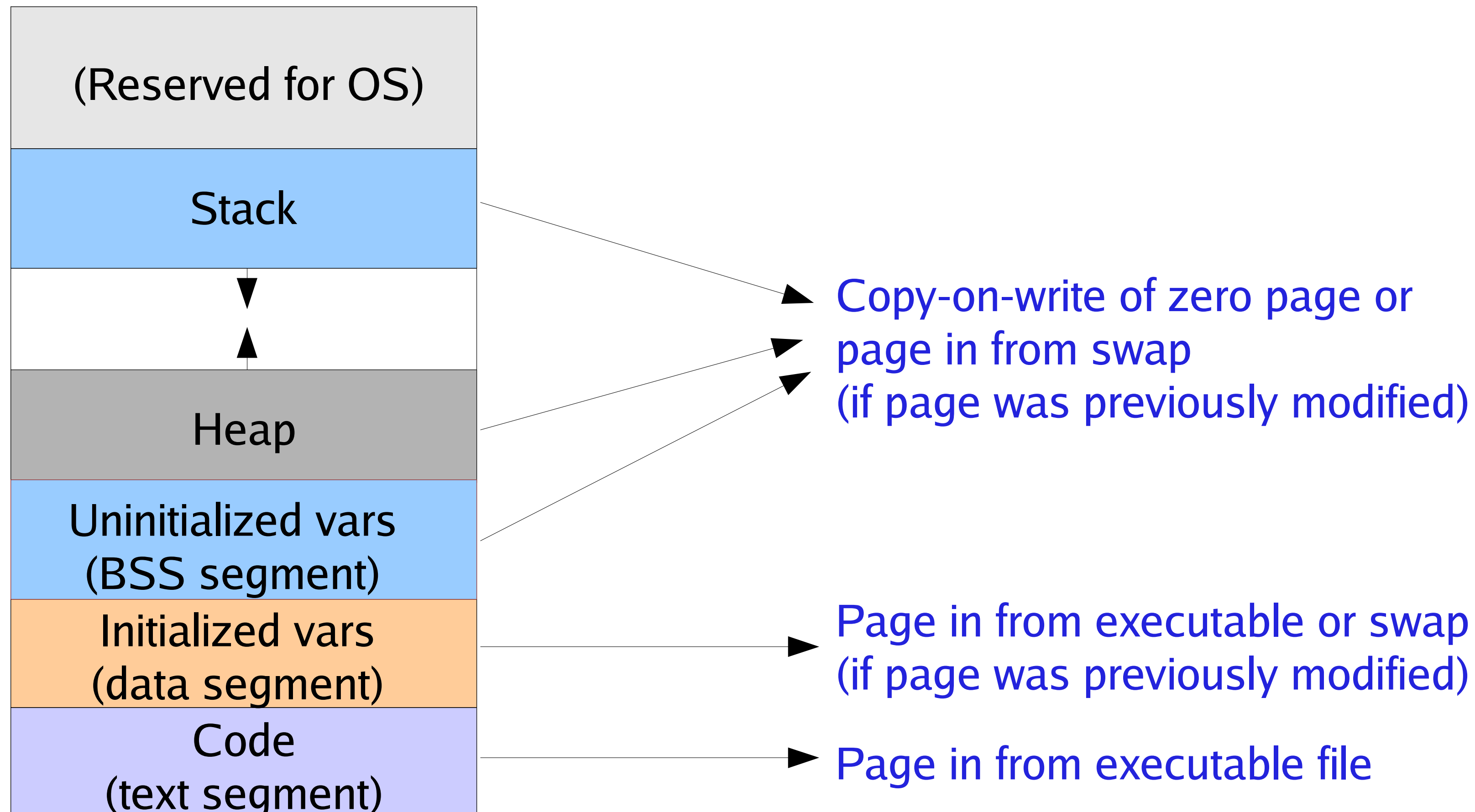
VM map structure

OS keeps a "map" of the layout of the process address space.

- This is separate from the page tables.
- In fact, the VM map is used by the OS to lay out the page tables.

This map can indicate where to find pages that are not in memory

- e.g., the disk file ID and the offset into the file.



© 2007 Matt Welsh – Harvard University

Page Eviction

How we evict a page depends on its type.

Code page:

- Just chuck it from memory – can recover it from the executable file on disk!

Unmodified (*clean*) data page:

- If the page has previously been swapped to disk, just chuck it from memory
  - *Assuming that page's backing store on disk has not been overwritten*
- If the page has never been swapped to disk, allocate new swap space and write the page to it
- Exception: unmodified zero page – no need to write out to swap at all!

Modified (*dirty*) data page:

- If the page has previously been swapped to disk, write page out to the swap space
- If the page has never been swapped to disk, allocate new swap space and write the page to it

© 2007 Matt Welsh – Harvard University

Physical Frame Allocation

How do we allocate physical memory across multiple processes?

- When we evict a page, which process should we evict it from?
- How do we ensure fairness?
- How do we avoid having one process hogging the entire memory of the system?

Fixed-space algorithms

- Per-process limit on the physical memory usage of each process
- When a process reaches its limit, it evicts pages *from itself*

Variable-space algorithms

- Physical size of processes can grow and shrink over time
- Allow processes to evict pages from other processes

One process paging can impact performance of entire system!

- One process that does a lot of paging will induce more disk I/O

© 2007 Matt Welsh – Harvard University

Better idea: Working Set

A process's *working set* is the set of pages that it currently "needs"

Definition:

- WS(P, t, w) = the set of pages that process P accessed in the time interval [t-w, t]
- "w" is usually counted in terms of number of page references
  - *A page is in WS if it was referenced in the last w page references*

Working set changes over the lifetime of the process

- Periods of high locality exhibit **smaller** working set
- Periods of low locality exhibit **larger** working set

Basic idea: Give process enough memory for its working set

- If WS is larger than physical memory allocated to process, it will tend to swap
- If WS is smaller than memory allocated to process, it's wasteful
- This amount of memory grows and shrinks over time

© 2007 Matt Welsh – Harvard University

Estimating the working set

How do we determine the working set of a process?

-
-
-

© 2007 Matt Welsh – Harvard University

Estimating the working set

How do we determine the working set of a process?

Simple approach: modified clock algorithm

- Sweep the clock hand at fixed time intervals
- Record how many seconds since last page reference
- All pages referenced in last T seconds are in the working set

Now that we know the working set, how do we allocate memory?

- If working sets for all processes fit in physical memory, done!
- Otherwise, reduce memory allocation of larger processes
  - *Idea: Big processes will swap anyway, so let the small jobs run unencumbered*
- Very similar to shortest-job-first scheduling: give smaller processes better chance of fitting in memory

How do we decide the working set time limit T?

- If T is too large, very few processes will fit in memory
- If T is too small, system will spend more time swapping
  - *Which is better?*

© 2007 Matt Welsh – Harvard University

Page Fault Frequency

Dynamically tune memory size of process based on # page faults

Monitor page fault rate for each process (faults per sec)

If page fault rate for process P is above threshold, give that process more memory

- Should cause process to fault less
- Doesn't always work!
    - *Why not???*

If page fault rate below threshold, reduce memory allocaton

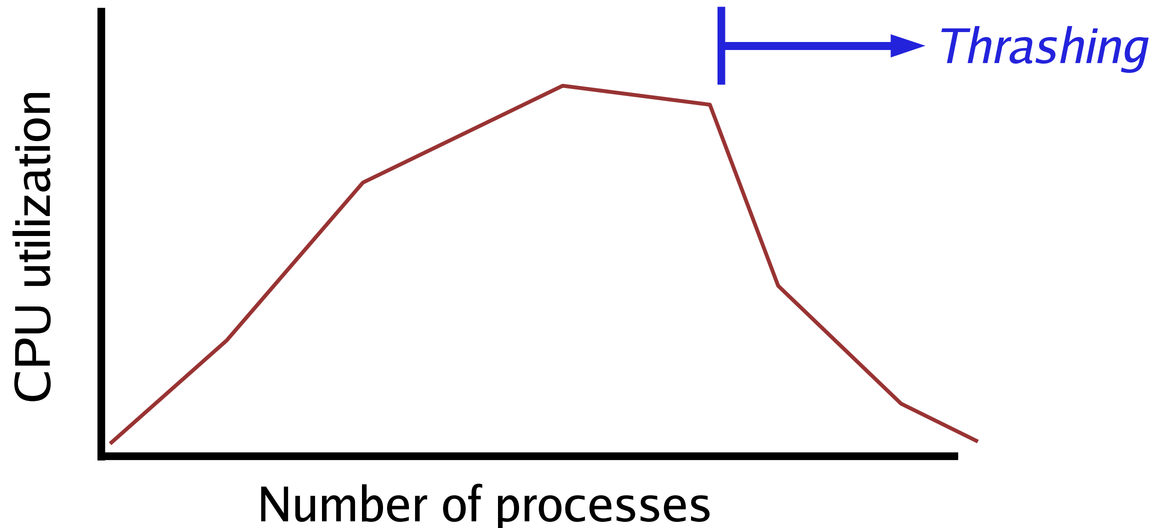
- Allow other processes to take advantage of increased space

© 2007 Matt Welsh – Harvard University

Thrashing

As system becomes more loaded, spends more of its time paging

- Eventually, no useful work gets done!



System is overcommitted!

- If the system has too little memory, the page replacement algorithm doesn't matter

Solutions?

- Change scheduling priorities to “slow down” processes that are thrashing
- Identify process that are hogging the system and kill them?
    - *Is thrashing a problem on systems with only one user?*

© 2007 Matt Welsh – Harvard University

Next Lecture

Detailed look at the Linux 2.4 Virtual Memory system and the x86 VM architecture