CS 134:
Operating Systems
Memory—Fun with Paging

CS 134:
Operating Systems
Memory—Fun with Paging

Overview

Page Faults
- Cost of Faults

Page Replacement
- Algorithms
- Easy Approaches
- Realistic Approaches

Optimizing Page Replacement
- Tweaking Clock
- “Pre-poning” Work

Working Sets
- Allocation Policies
- Thrashing

Overview
Page Faults
Cost of Faults
Page Replacement
Algorithms
Easy Approaches
Realistic Approaches
Optimizing Page Replacement
Tweaking Clock
“Pre-poning” Work
Working Sets
Allocation Policies
Thrashing

Page Faults

What needs to happen when a page fault occurs?

Page Faults

What needs to happen when a page fault occurs?

Page Faults

What happens. . .

- User process accesses invalid memory—traps to OS

Page Faults

What happens. . .

- User process accesses invalid memory—traps to OS

Page Faults

What happens. . .

- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state

Page Faults

What happens. . .
- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state

Page Faults

What happens. . .

- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state
  - Checks access was actually legal

Page Faults
What happens. . .
User process accesses invalid memory—traps to OS
OS:
Saves process state
Checks access was actually legal

Page Faults

What happens...

- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state
  - Checks access was actually legal
  - Finds a free frame

Page Faults
What happens...
User process accesses invalid memory—traps to OS
OS:
Saves process state
Checks access was actually legal
Finds a free frame

Page Faults

What happens. . .

- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state
  - Checks access was actually legal
  - Finds a free frame
  - Reads from disk to free frame—I/O wait, process blocked

Page Faults
What happens. . .
- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state
  - Checks access was actually legal
  - Finds a free frame
  - Reads from disk to free frame—I/O wait, process blocked

Page Faults

What happens. . .

- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state
  - Checks access was actually legal
  - Finds a free frame
  - Reads from disk to free frame—I/O wait, process blocked
  - Gets interrupt from disk (I/O complete)—process ready

Page Faults
What happens. . .
User process accesses invalid memory—traps to OS
OS:
Saves process state
Checks access was actually legal
Finds a free frame
Reads from disk to free frame—I/O wait, process blocked
Gets interrupt from disk (I/O complete)—process ready

Page Faults

What happens. . .

- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state
  - Checks access was actually legal
  - Finds a free frame
  - Reads from disk to free frame—I/O wait, process blocked
  - Gets interrupt from disk (I/O complete)—process ready
  - Scheduler restarts process—process running

Page Faults

What happens. . .
- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state
  - Checks access was actually legal
  - Finds a free frame
  - Reads from disk to free frame—I/O wait, process blocked
  - Gets interrupt from disk (I/O complete)—process ready
  - Scheduler restarts process—process running

Page Faults

What happens. . .

- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state
  - Checks access was actually legal
  - Finds a free frame
  - Reads from disk to free frame—I/O wait, process blocked
  - Gets interrupt from disk (I/O complete)—process ready
  - Scheduler restarts process—process running
  - Adjusts page table

Page Faults
What happens. . .
- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state
  - Checks access was actually legal
  - Finds a free frame
  - Reads from disk to free frame—I/O wait, process blocked
  - Gets interrupt from disk (I/O complete)—process ready
  - Scheduler restarts process—process running
  - Adjusts page table

Page Faults

What happens. . .

- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state
  - Checks access was actually legal
  - Finds a free frame
  - Reads from disk to free frame—I/O wait, process blocked
  - Gets interrupt from disk (I/O complete)—process ready
  - Scheduler restarts process—process running
  - Adjusts page table
  - Restores process state

Page Faults

Page Faults
What happens. . .
User process accesses invalid memory—traps to OS
OS:
Saves process state
Checks access was actually legal
Finds a free frame
Reads from disk to free frame—I/O wait, process blocked
Gets interrupt from disk (I/O complete)—process ready
Scheduler restarts process—process running
Adjusts page table
Restores process state

Page Faults

What happens. . .

- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state
  - Checks access was actually legal
  - Finds a free frame
  - Reads from disk to free frame—I/O wait, process blocked
  - Gets interrupt from disk (I/O complete)—process ready
  - Scheduler restarts process—process running
  - Adjusts page table
  - Restores process state
  - Returns to user code

Page Faults

What happens. . .
- User process accesses invalid memory—traps to OS
- OS:
  - Saves process state
  - Checks access was actually legal
  - Finds a free frame
  - Reads from disk to free frame—I/O wait, process blocked
  - Gets interrupt from disk (I/O complete)—process ready
  - Scheduler restarts process—process running
  - Adjusts page table
  - Restores process state
  - Returns to user code

Page Faults (cont.)

How long?

- Disk is slow
- 5–15 ms is a conservative guess
- Main memory takes 5–15 ns
- Page fault is about **1 million times slower** than a regular memory access
- Page faults must be rare! (Need locality!)

Page Faults (cont.)

How long?

- Disk is slow
- 5–15 ms is a conservative guess
- Main memory takes 5–15 ns
- Page fault is about **1 million times slower** than a regular memory access
- Page faults must be rare! (Need locality!)

A “Back of an Envelope Calculation”

How often are there page faults?

An example from a desktop machine:

- In 14 days
  - 378,110 page-ins
  - Average load < 4% $\rightarrow$ 12 hours actual compute time
  - 8.75 page faults per second *average*
- 1,000,000,000 memory accesses per second (a guess)
- 43,200,000,000,000 memory accesses in 12 hours
- 1 page-in every 114,252,466 memory accesses
- Using 5 ns for memory, 5 ms for disk:
  - $t_{\text{avg}} = (5,000,000 * 1 + 5 * 114,252,465)/114,252,466$
  - $t_{\text{avg}} = 5.04\text{ns}$

A “Back of an Envelope Calculation”
How often are there page faults?
An example from a desktop machine:
In 14 days
378,110 page-ins
Average load < 4% → 12 hours actual compute time
8.75 page faults per second average
1,000,000,000 memory accesses per second (a guess)
43,200,000,000,000 memory accesses in 12 hours
1 page-in every 114,252,466 memory accesses
Using 5 ns for memory, 5 ms for disk:
$t_{\text{avg}} = (5,000,000 * 1 + 5 * 114,252,465)/114,252,466$
$t_{\text{avg}} = 5.04\text{ns}$

Here’s the problem with t_{avg}: it’s spread over 14 days, including time when the desktop’s owner was asleep. It’s an *average*! So what if just 1% of those 378K page-ins happened last Monday morning when the owner started work? All of a sudden we’re spending $3781 \times 5\text{ms} = 18.905\text{sec}$ waiting for the machine to respond. And the reality is that many more than 1% of the page-ins happen when the owner is most actively using the machine. . .

Part of the problem is “cold start,” when a program is faulting itself in. We can improve that in several ways; for example we can pre-load the first page of instructions, perhaps the first page of each dynamic library. We can also detect sequential page accesses and prefetch future pages. Or going further, we can remember what happened last time the program ran and bring in those pages.

Page Faults (cont.)

Other kinds of page faults:

- Demand-page executables from their files, not swap device
- Copy-on-write memory—great for fork
- Lazy memory allocation
- Other tricks…

Page Faults (cont.)
Other kinds of page faults:
- Demand-page executables from their files, not swap device
- Copy-on-write memory—great for fork
- Lazy memory allocation
- Other tricks…

What kind of other tricks? Well, for example, debugging and tracing; VM translation; buffer-overflow prevention.

Page Replacement

What happens when we run out of free frames?

Page Replacement
What happens when we run out of free frames?

Page Replacement

What happens when we run out of free frames?

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Add modified (dirty) bit to page table.
  - Only modified pages are written to disk.

Page Replacement

What happens when we run out of free frames?

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Add modified (dirty) bit to page table.
  - Only modified pages are written to disk.

Page Replacement

What happens when we run out of free frames?

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Add modified (dirty) bit to page table.
  - Only modified pages are written to disk.

This brings us to ***Virtual Memory***—we can provide a larger logical address space than we have physical memory

Page Replacement

What happens when we run out of free frames?
- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Add modified (dirty) bit to page table.
  - Only modified pages are written to disk.

This brings us to ***Virtual Memory***—we can provide a larger logical address space than we have physical memory

Page-Replacement Algorithms

Deciding which page to kick out is tricky
How to compare algorithms?

- Run them on a stream of page numbers corresponding to execution of a (hypothetical?) program

(We want to achieve the lowest page-fault rate, i.e., minimum t_{avg})

Page-Replacement Algorithms

Deciding which page to kick out is tricky
How to compare algorithms?
- Run them on a stream of page numbers corresponding to execution of a (hypothetical?) program
(We want to achieve the lowest page-fault rate, i.e., minimum t_{avg})

Page Replacement Algorithms

For example, suppose memory accesses by the system are

```
00002e00 00002e04 00002e08 00002e0c
00002f00 00002f04 00003216 00003800
00002f08 00001eb0 00001eb4 00001eb8
00005380 00002f0c 00002f10 00002f14
00002f18 00002f1c 00002f20 00002f24
00004d84 00004d88 00004d8c 00005380
00003800 00003216 00002f28 00005380
00002f2c 00002f30
```


Page Replacement Algorithms
For example, suppose memory accesses by the system are
00002e00 00002e04 00002e08 00002e0c
00002f00 00002f04 00003216 00003800
00002f08 00001eb0 00001eb4 00001eb8
00005380 00002f0c 00002f10 00002f14
00002f18 00002f1c 00002f20 00002f24
00004d84 00004d88 00004d8c 00005380
00003800 00003216 00002f28 00005380
00002f2c 00002f30

Page Replacement Algorithms

For example, suppose memory accesses by the system are

00002e00 00002e04 00002e08 00002e0c
00002f00 00002f04 **00003**216 00003800
00002f08 **00001**eb0 00001eb4 00001eb8
00005380 **00002**f0c 00002f10 00002f14
00002f18 00002f1c 00002f20 00002f24
00004d84 00004d88 00004d8c **00005**380
00003800 00003216 **00002**f28 **00005**380
00002f2c 00002f30

The stream of page numbers for the above execution is

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

Page Replacement Algorithms
For example, suppose memory accesses by the system are
00002e00 00002e04 00002e08 00002e0c
00002f00 00002f04 **00003**216 00003800
00002f08 **00001**eb0 00001eb4 00001eb8
00005380 **00002**f0c 00002f10 00002f14
00002f18 00002f1c 00002f20 00002f24
00004d84 00004d88 00004d8c **00005**380
00003800 00003216 **00002**f28 **00005**380
00002f2c 00002f30
The stream of page numbers for the above execution is
2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

Page-Replacement Policies

When you need to free up a frame, how do you choose?

Class Exercise

What are some *easy* strategies?

Page-Replacement Policies

When you need to free up a frame, how do you choose?

Class Exercise

What are some *easy* strategies?

Random (RAND)

Throw out a random page.

Random (RAND)
Throw out a random page.

NRU (Not Recently Used) is the VAX VMS algorithm: periodically clear referenced bits, and evict a random not-referenced page; see book for details.

Random (RAND)

Throw out a random page.

RAND is

- Easy to implement
- Prone to throwing out a page that's being used
  - The page will get paged back in
  - Hope it is lucky and won't get zapped again next time

(NRU is a variant on RAND)

Random (RAND)
Throw out a random page.
RAND is
Easy to implement
Prone to throwing out a page that's being used
The page will get paged back in
Hope it is lucky and won't get zapped again next time
(NRU is a variant on RAND)

NRU (Not Recently Used) is the VAX VMS algorithm: periodically clear referenced bits, and evict a random not-referenced page; see book for details.

First-in First-out Policy (FIFO)

Throw out the oldest page.

Try the following stream of page numbers with 3 frames and with 4 frames:

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

First-in First-out Policy (FIFO)
Throw out the oldest page.
Try the following stream of page numbers with 3 frames and with 4 frames:
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Do this on the board with the class.

First-in First-out Policy (FIFO)

Throw out the oldest page.

FIFO is

- Easy to implement
- Prone to throwing out a page that's being used
  - The page will get paged back in
  - It will then be young again, and will not be thrown out again for a long time
- Prone to *Belady's Anomaly*—increasing the number of frames can sometimes increase the number of page faults

First-in First-out Policy (FIFO)

Throw out the oldest page.

FIFO is
- Easy to implement
- Prone to throwing out a page that's being used
  - The page will get paged back in
  - It will then be young again, and will not be thrown out again for a long time
- Prone to *Belady's Anomaly*—increasing the number of frames can sometimes increase the number of page faults

Optimal Page-Replacement Policy (OPT)

Replace the page that won't be accessed for the longest time

Optimal Page-Replacement Policy (OPT)

Replace the page that won't be accessed for the longest time

Optimal Page-Replacement Policy (OPT)

Replace the page that won't be accessed for the longest time

OPT is

- Provably optimal
- Impossible to implement
- Useful as a benchmark

Optimal Page-Replacement Policy (OPT)

Replace the page that won't be accessed for the longest time

OPT is

- Provably optimal
- Impossible to implement
- Useful as a benchmark

Least Recently Used (LRU)

Choose to replace the page that hasn't been accessed for the longest time.

Class Exercise

Why is LRU hard to implement?

Least Recently Used (LRU)
Choose to replace the page that hasn't been accessed for the longest time.
Class Exercise
Why is LRU hard to implement?

Least Recently Used (LRU)

Choose to replace the page that hasn't been accessed for the longest time.

LRU is

- Hard to implement
- Fairly close to OPT in performance

Class Exercise

What's the worst case for LRU?

Can it happen in real programs?

Least Recently Used (LRU)

Choose to replace the page that hasn't been accessed for the longest time.

LRU is
- Hard to implement
- Fairly close to OPT in performance

Class Exercise
What's the worst case for LRU?

Can it happen in real programs?

Clock (aka Second Chance)

Hardware maintains a “referenced” bit in the page table

- Set by hardware when page is accessed
- Only cleared by the OS

Use FIFO page replacement, but:

- If a page has its referenced bit set, clear it and move on to the next page

Clock is

- Easy to implement
- An approximation of LRU

2013-05-17

CS34
└Page Replacement
  └Realistic Approaches
    └Clock (aka Second Chance)

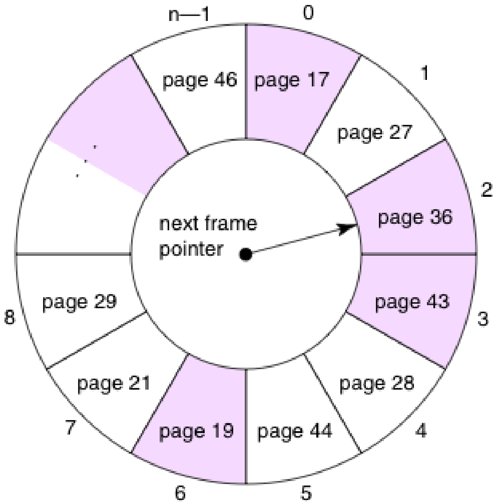
Clock (aka Second Chance)

Hardware maintains a “referenced” bit in the page table
- Set by hardware when page is accessed
- Only cleared by the OS

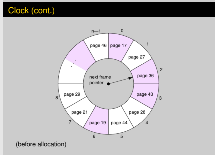
Use FIFO page replacement, but:
- If a page has its referenced bit set, clear it and move on to the next page

Clock is
- Easy to implement
- An approximation of LRU

Clock (cont.)

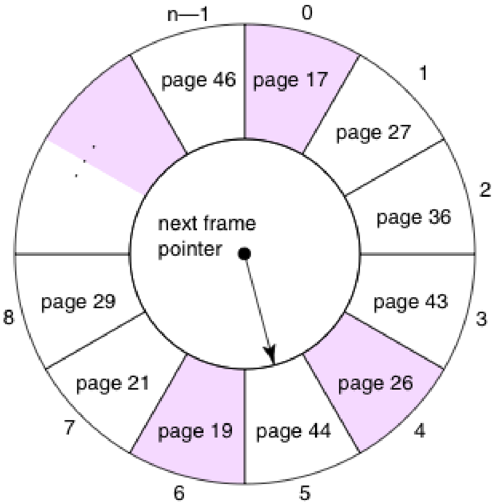


(before allocation)

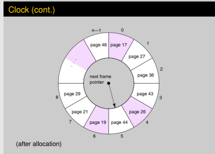


Here, the magenta frames are ones that have been referenced. When there is a fault on virtual page 26 (poor choice of example, since it's hard to see 28/26 difference on the diagram), we clear the referenced bits on physical pages 2 and 3, then place virtual 26 into physical 4, setting its referenced bit, and advance the pointer to physical 5.

Clock (cont.)

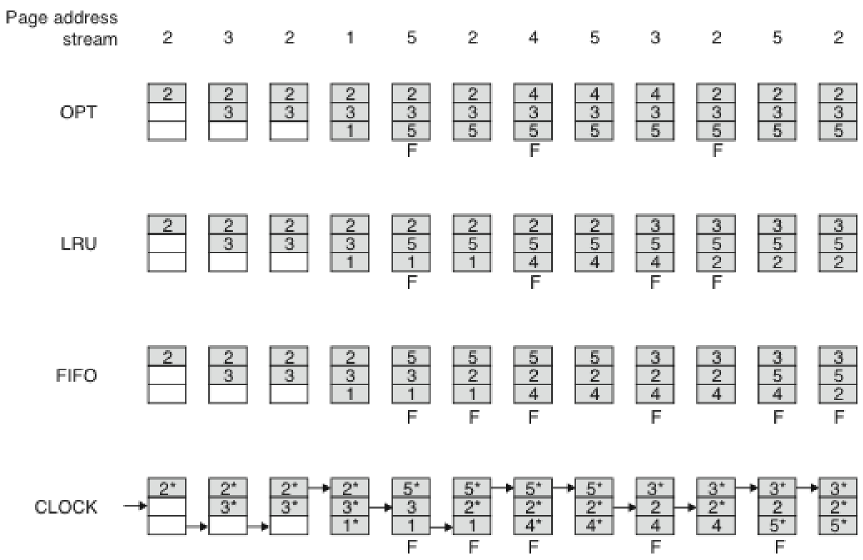


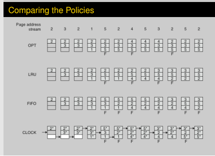
(after allocation)



If there is another fault immediately, we'll put the new page in physical page 5, replacing virtual page 44.

Comparing the Policies





Spend some time on this slide.
An "F" under a column indicates that there was a fault. All algorithms are the same for the first four accesses, and all fault on the fifth access. Note that the first three faults aren't shown.
The last "F" under OPT could replace either virtual 4 or virtual 3.
On CLOCK, the fifth access finds all referenced bits set, so it chooses the page that was originally under the hand—after scanning every other page in the system! Fortunately, this is rare in real systems with thousands or even millions of pages.

Optimizations

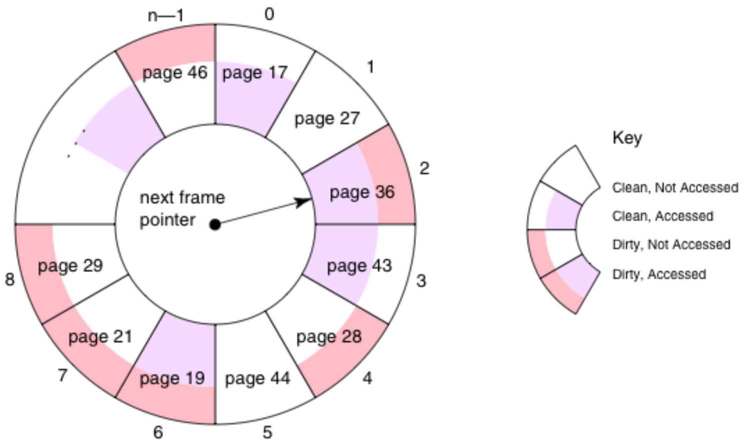
Why is page replacement slow?

Optimizations

Why is page replacement slow?

One of the big costs is writing dirty pages. That's especially bad because it may mean seeking to the swap space to write, then somewhere else to read.

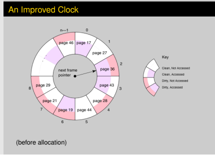
An Improved Clock



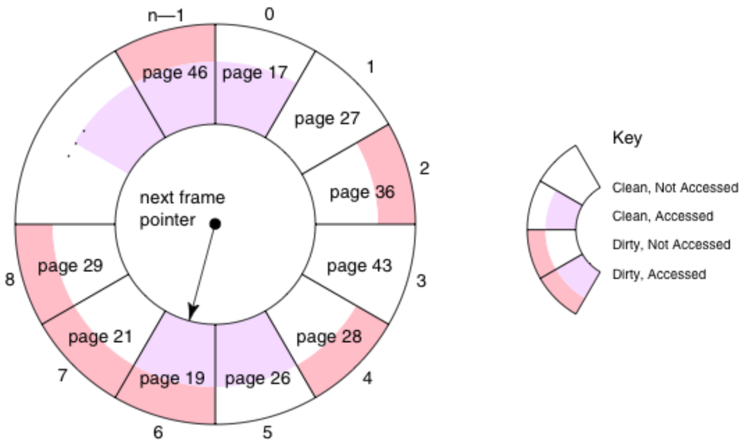
(before allocation)

2013-05-17

CS34
└─Optimizing Page Replacement
  └─An Improved Clock

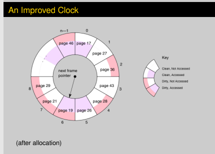


An Improved Clock



(after allocation)

2013-05-17



Here, we've skipped over virtual page 28 (physical 4) because it's dirty, and instead we've replaced physical 6. That saves us the disk write—but if we keep going in this mode, eventually all pages will be dirty, not accessed. We'll come back to that point in a moment.

More Clock

How quickly does the hand go around?

Why is that an issue?

More Clock
How quickly does the hand go around?
Why is that an issue?

It's an issue because the notion of "recently used" depends on how fast you go around. Adding memory has an effect; so does adding other programs.

Two-Handed Clock

Have two clock hands, separated by fixed amount:

- Leading hand clears referenced bit
- Lagging hand frees unreferenced pages
- “Recently used” now depends only on distance between hands

Two-Handed Clock

Have two clock hands, separated by fixed amount:

- Leading hand clears referenced bit
- Lagging hand frees unreferenced pages
- “Recently used” now depends only on distance between hands

Page Buffering

Try to do some work *ahead of time*—keep a list of “free” pages

- Find a page that doesn’t appear to be being used
- Write it to disk if dirty
- Free it if clean
- Can be implemented with queue of “ready to free” pages
  - Can reprieve page from queue if it gets referenced

Even FIFO page replacement is workable with page buffering.

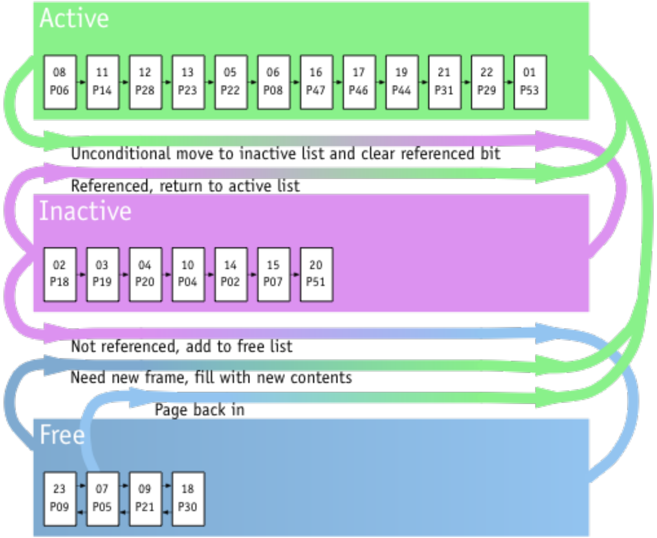
Page Buffering

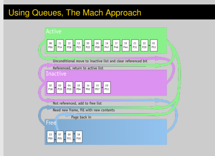
Try to do some work *ahead of time*—keep a list of “free” pages

- Find a page that doesn’t appear to be being used
- Write it to disk if dirty
- Free it if clean
- Can be implemented with queue of “ready to free” pages
  - Can reprieve page from queue if it gets referenced

Even FIFO page replacement is workable with page buffering.

Using Queues, The Mach Approach





This is a practical implementation of a buffering algorithm.
Only dirty pages can be reactivated, because otherwise the OS has lost track of "what they really are."

Attempts to keep 2/3 of pages active, 1/3 inactive, 5% in free list.

Enough Frames?

How do you know if you have enough frames to work with. . . ?

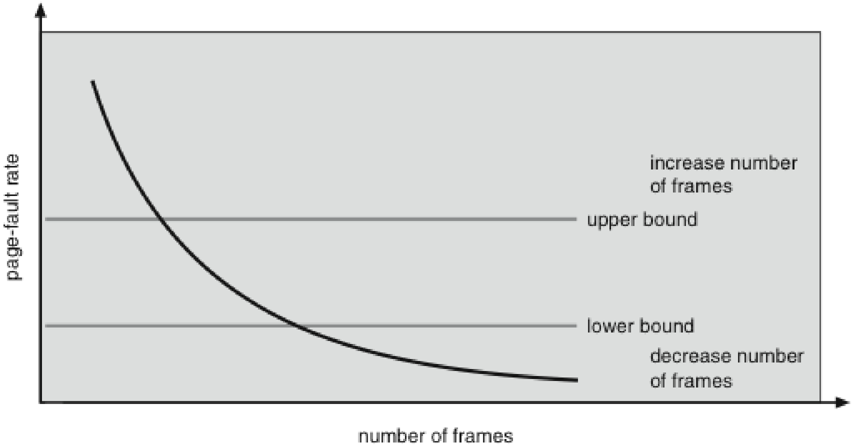
Enough Frames?

How do you know if you have enough frames to work with. . . ?

Working Sets

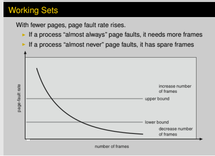
With fewer pages, page fault rate rises.

- If a process “almost always” page faults, it needs more frames
- If a process “almost never” page faults, it has spare frames



2013-05-17

CS34
└Working Sets
└Working Sets



Working Sets

How can we keep track of the working set of a process?

Working Sets
How can we keep track of the working set of a process?

Formal definition is "pages referenced in last *k* accesses." Close approximation is "pages referenced in last *n* ms." Note that this is pretty close to what the CLOCK algorithm does, except that other processes can interfere. Which leads to. . .

Local vs. Global

Whose pages do we take?

Local vs. Global

Whose pages do we take?

Frame Allocation Policies

So far, we've examined paging without thinking about processes—but what about processes?

- Each process needs a bare minimum number of pages (set by hardware characteristics of machine)
- Frames need to be shared out *fairly* between processes

Frame Allocation Policies

So far, we've examined paging without thinking about processes—but what about processes?

- Each process needs a bare minimum number of pages (set by hardware characteristics of machine)
- Frames need to be shared out *fairly* between processes

Local, Fixed Frame Allocation

Give each of the n processes $1/n$ of the available frames

- Each process can only take frames from itself

Class Exercise

What do you think?

Local, Fixed Frame Allocation
Give each of the n processes $1/n$ of the available frames
Each process can only take frames from itself
Class Exercise
What do you think?

Local, Proportional Frame Allocation

Give each process frames in proportion to the amount of *virtual* memory they use

Class Exercise

What do you think?

Local, Proportional Frame Allocation
Give each process frames in proportion to the amount of *virtual* memory they use
Class Exercise
What do you think?

- Some processes use a lot of VM, but don't access it often
- Some processes use a little VM, but access it often
- Not fair

Global, Variable Allocation

Just take the “best” (e.g., LRU) page, no matter which process it belongs to. . .

Class Exercise

Is this policy fair?

If not, why not?

Global, Variable Allocation

Just take the “best” (e.g., LRU) page, no matter which process it belongs to. . .

Class Exercise

Is this policy fair?

If not, why not?

Local, Variable Allocation

Each program has a frame allocation

- Use *working set* measurements to adjust frame allocation from time to time.
- Each process can only take frames from itself.

Class Exercise

What's wrong with this policy?

- I.e., what assumptions are we making that could be wrong?

Local, Variable Allocation

Each program has a frame allocation
- Use *working set* measurements to adjust frame allocation from time to time.
- Each process can only take frames from itself.

Class Exercise

What's wrong with this policy?
- I.e., what assumptions are we making that could be wrong?

Wrong assumptions: that we can measure working sets properly. That we can fit all working sets in memory.

Local, Variable Allocation

Each program has a frame allocation

- Use *working set* measurements to adjust frame allocation from time to time.
- Each process can only take frames from itself.

Class Exercise

What's wrong with this policy?

- I.e., what assumptions are we making that could be wrong?

What should we do if the working sets of all processes are more than the total number of frames available?

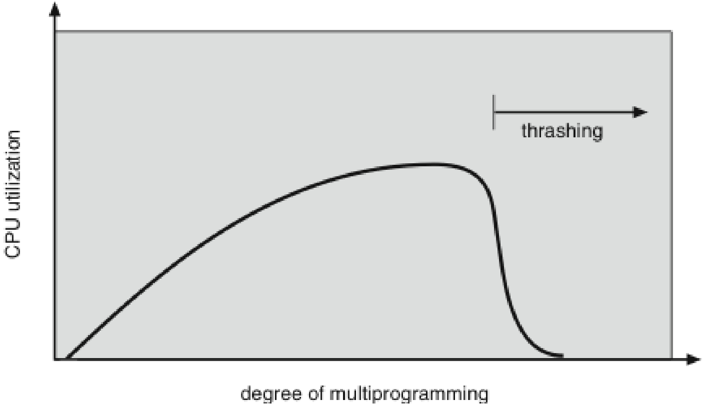
Local, Variable Allocation
Each program has a frame allocation
- Use *working set* measurements to adjust frame allocation from time to time.
- Each process can only take frames from itself.
Class Exercise
What's wrong with this policy?
- I.e., what assumptions are we making that could be wrong?
What should we do if the working sets of all processes are more than the total number of frames available?

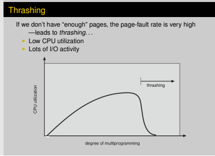
Wrong assumptions: that we can measure working sets properly. That we can fit all working sets in memory.

Thrashing

If we don't have "enough" pages, the page-fault rate is very high —leads to *thrashing*. . .

- Low CPU utilization
- Lots of I/O activity





Thrashing

Under local replacement policy, only problem process is affected (usually)

- Can detect and swap out until can give bigger working set
- If can't give big enough, might want to kill. . .

Under global replacement policy, whole machine can be brought to its knees!

. . . But even under local policy, disk can become so busy that no other work gets done!

Thrashing

Under local replacement policy, only problem process is affected (usually)

- Can detect and swap out until can give bigger working set
- If can't give big enough, might want to kill. . .

Under global replacement policy, whole machine can be brought to its knees!

. . . But even under local policy, disk can become so busy that no other work gets done!