CMPT 300 — Operating Systems I

Summer 1999

Segment 8:

Memory Management

Melissa O'Neill

Background — How Processes Get into Memory

Class Exercise: What transformations does the C source below need go through to become a running process?

```
int main ( ) {
    write(1, "Hello World\n", 12);
    return 0;
}
```


Background — How Processes Get into Memory

Compiler compiles code to obtain the following:

```
.cstring                              select area for storing strings
Lstr0:    .ascii "Hello World\12\0"   string starts at Lstr0
.text                                 select area for storing program code
.globl _main                          mark _main as an external symbol
_main:    pea 12                      push 12 onto the stack (last arg to write)
          pea Lstr0                   push the address of the string onto stack
          pea 1                       push 1 onto the stack (first arg to write)
          jbsr _write                 call write
          addw #12,sp                 pop three arguments off the stack
          clrl d0                     clear d0 for return of zero
          rts                         return
```


Background — How Processes Get into Memory

Assembler compiles code to obtain the following:

helloworld.o
symbols:

```
00000000 (TEXT,text) external _main
         (undefined) external _write
```

(TEXT,text) section:

```
00000000 4878 000c 4879 0000 001e 4878 0001 61ff
00000010 ffff fff0 4280 defc 000c 4e75 4e71
```

(TEXT,cstring) section:

```
0000001e 4865 6c6c 6f20 576f 726c 640a 0000
```

relocation information for (__TEXT,__text):

```
address  pcrel length extern  symbolnum/value
00000010 True  long   True    _write
00000006 False long   False   2 (TEXT,cstring)
```

Background — How Processes Get into Memory

Linker adds startup code and resolves relocation entries:

```
                        .
                        .
                        .
000033bc f342 2040 4879 0000 4010 2f28 0020 61ff
000033cc 0000 073a 4e5e 4e75 4878 000c 4879 0000
000033dc 3f26 4878 0001 61ff 04ff ff2a 4280 defc
000033ec 000c 4e75 4e71 4856 2c4f 48e7 3800 226e
000033fc 0008 4282 4283 4281 1219 2001 787f c084
                        .
                        .
                        .
                        .
                        .
                        .
00003f14 5f5f 6568 5f66 7261 6d65 005f 5f54 4558
00003f24 5400 4865 6c6c 6f20 576f 726c 640a 0000
00003f34 6568 0000
```

(And this version isn't the end of the story!)

Background — How Processes Get into Memory

So far, we have:

- A *Load Image*

The operating system still needs to:

- Decide if it has resources to run the program right now (long-term scheduler)
- Decide where to put the program in memory
- Perform any additional setup
- Start executing the program

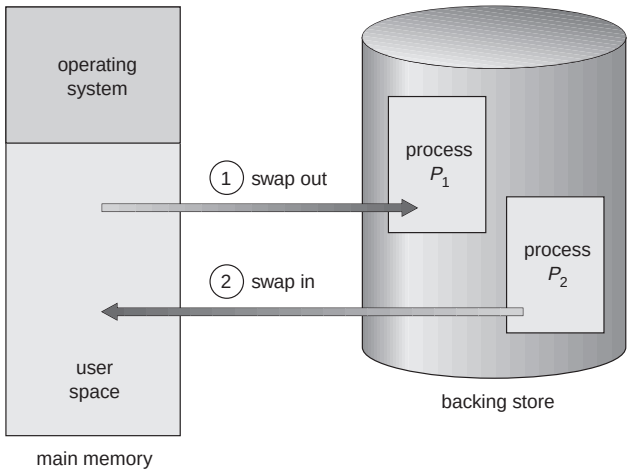
Uniprogramming OS

Only one process — can always locate running process in same place

- static linking
- loading is easy

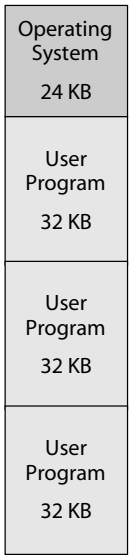
Multiprogramming OS — Simple, using Swapping

Add swapping to Uniprogramming OS



Fixed Partitioning

Add more memory, to allow multiple processes



Fixed Partitioning (continued)

But

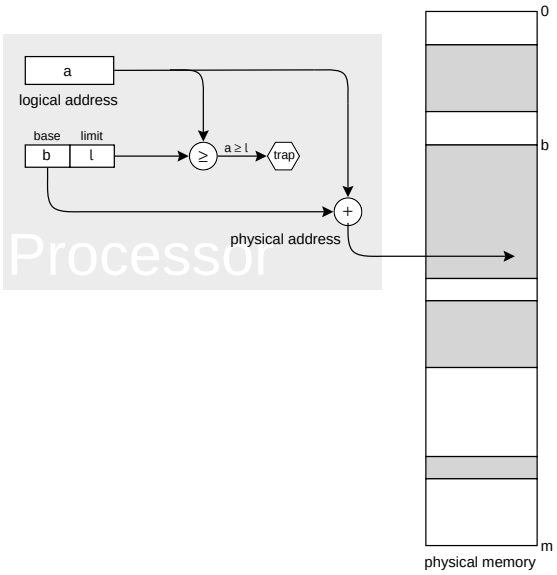
- Now we don't know where the process is going to be located in memory
- Loading must deal with relocation?

Runtime Relocation

Two methods — first method:

- more sophisticated hardware, add base base register to user addresses
  - *logical address* — used in programs
  - *physical address* — actual address used in programs

Base and Limit Registers



Runtime Relocation (continued)

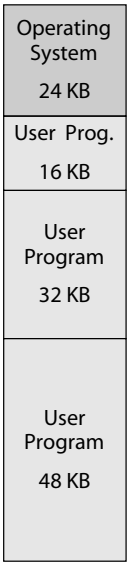
Second method:

- position independent code
  - *use relative addressing modes*

All this extra memory, but we can't run big jobs, or run lots of small jobs!

- Wasted space inside partitions is *internal fragmentation*

Fixed Partitioning with Unequal-Sized Partitions

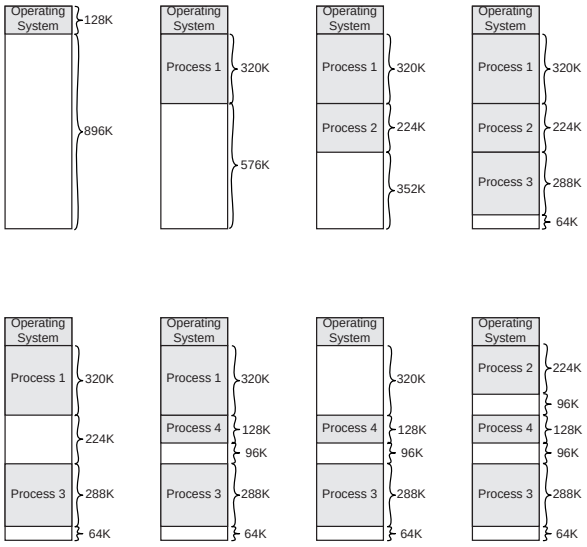


Dynamic partitioning

Thus,

- Don't make the partitions fixed size!
- Put the process in a *hole* large enough to accommodate it
- Keep a *free list* of holes

Dynamic Partitioning (continued)



Which hole?

Best fit?

- Choose the smallest hole that is large enough

Worst fit?

- Choose the largest hole that is large enough

First fit?

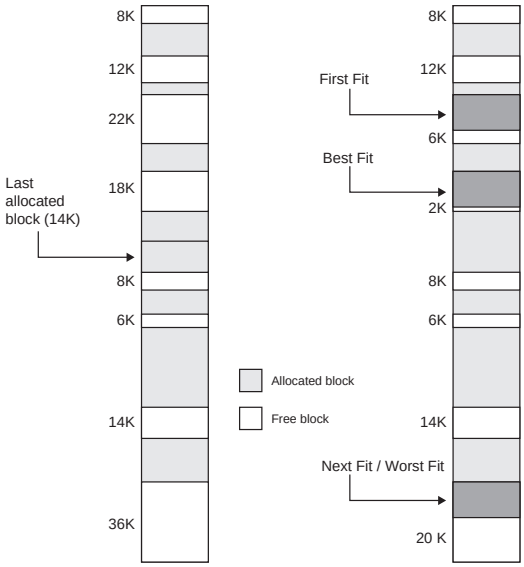
- Choose the first hole that is large enough

Next fit?

- Choose the first hole that is large enough, starting the search after the last hole we allocated from

Class Exercise: Which method is best?

Which hole? (continued)



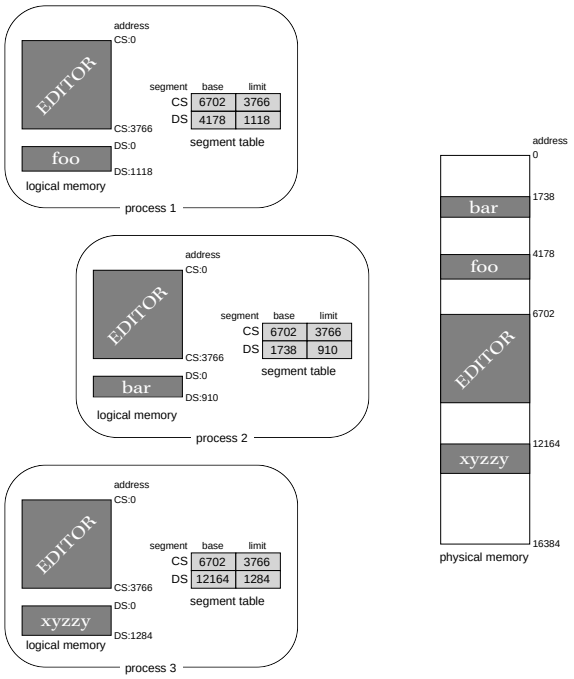
External Fragmentation

- All methods are prone to fragmentation
- Best fit and first fit are the best methods, with least fragmentation on average
- Can eliminate fragmentation by *compaction*

Code Sharing?

- What if two people are running the same editor?
- Introduce segments — code segment and data segment:
  - Program code is put in a *program segment* (read only), shared between processes
  - Program data is put in a *data segment*, unique to each process

- If two segments are a good idea, would more segments be even better?



Segmentation Architecture

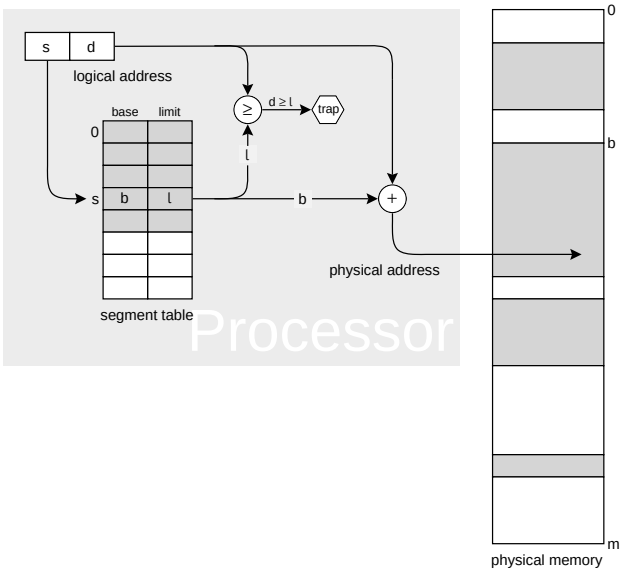
- Logical address consists of the pair

  <segment-number, offset>

  (e.g., use 32-bit logical address, first 8 bits are segment number, remaining 24 bits are offset within the segment — 256 segments, of max size 16,777,216 bytes (16MB))

- *Segment table* — maps two-dimensional user-defined addresses into one-dimensional physical addresses; each table entry has:
  - Base — contains the starting address of the segment in physical memory
  - Limit — specifies the length of the segment
- Segment table should be small enough to fit inside processor.

Segmentation Architecture (contd.)



Segmentation Architecture (contd.)

- Relocation.
  - Dynamic
  - By segment table
- Sharing.
  - Shared segments
  - Same segment number
- Allocation.
  - First fit/best fit
  - External fragmentation

Class Exercise: Do shared segments *need* to have the same segment number.

If so, why?

If not, why? (Why might we give them the same segment number anyway?)

Segmentation Architecture

Class Exercise: What if a program wants more contiguous data space than a segment can hold? Is this a problem?

Segmentation Architecture — Protection

- With each entry in segment table, associate:
  - Validation bit — 0 => illegal segment
  - Read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level

Segmentation Architecture — Cache Issues

Class Exercise: Should the processor cache data based on its logical address, or its physical address?

What are the tradeoffs?

Segmentation Architecture — Fragmentation

- Internal fragmentation — not a problem
- External fragmentation — a problem: compaction takes too long

What's the cause of the external fragmentation?

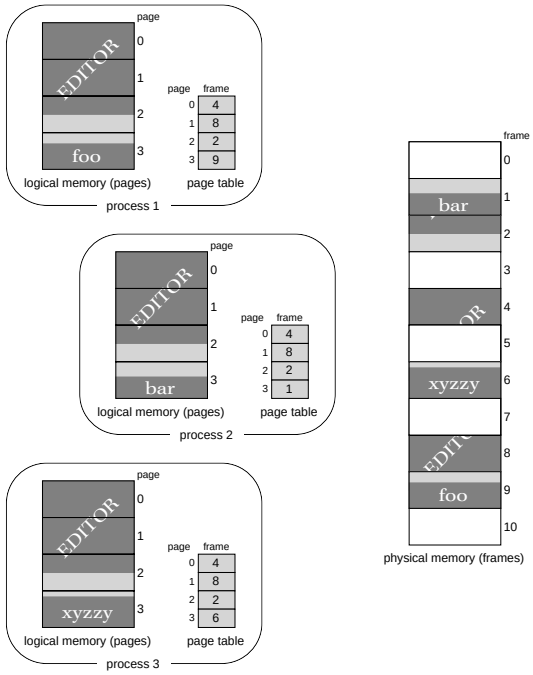
- Differing segment sizes

Solution?

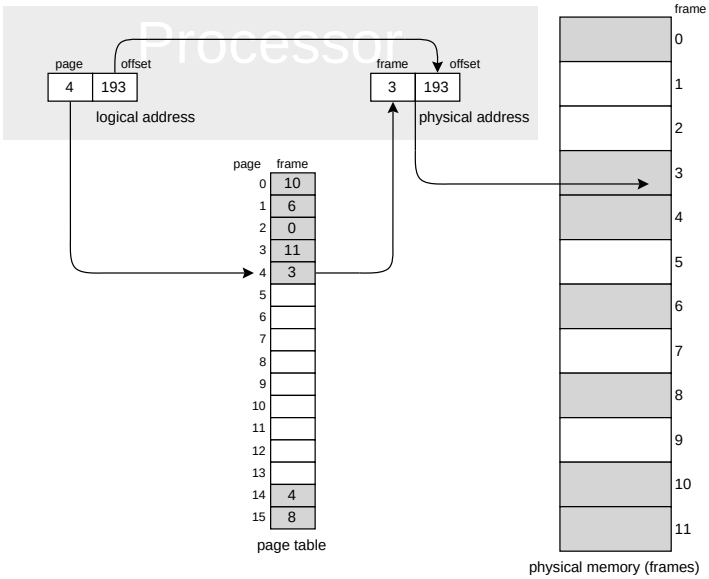
- Make all segments the same size!
  - But now we have internal fragmentation!
  - Better make the segments small, to minimize wastage — remember, we can cope with small segments

Tiny Segments

- All segments are the same size
- No need for limit registers
- No longer reflect program structure
- Call them ***pages***
- Physical locations for pages are called *page* ***frames***
- Now have a *lot* of pages.
  - Suppose page size is 4K, a 32-bit logical address now consists of a 20-bit page number and a 12-bit offset
  - 20-bit page number => 1,048,576 possible pages!
  - Too many pages to remember inside the processor
  - Use memory — give processor a page-table base register (PTBR), and a page-table length register (PTLR)

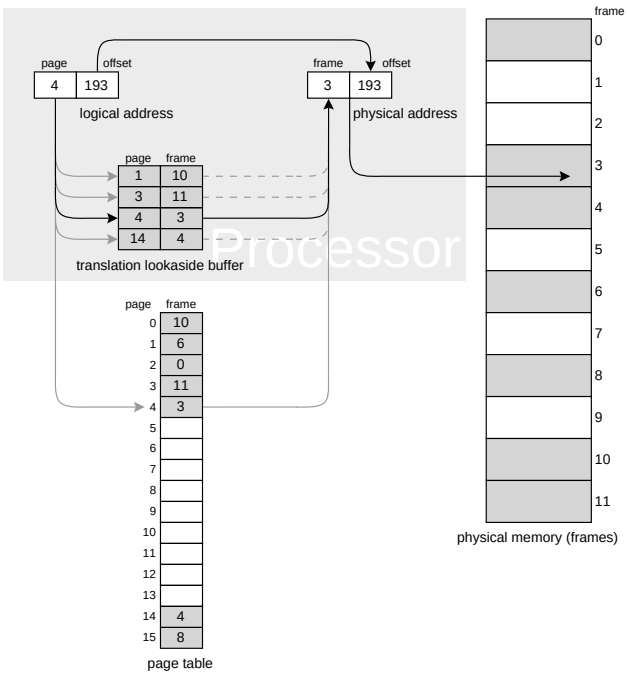


Paging Hardware (basic)



Improving Performance

- Logical-to-physical address translation now requires a memory access
  - Two memory accesses for every word read!
- Locality
  - We usually access more than one memory location on the same page.
- Add cache for page-table entries — call it "the translation look-aside buffer" (TLB)



What If Logical Address Space Is Sparsely Filled?

For example,

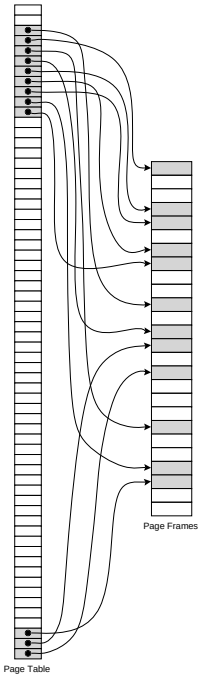
- Locate kernel code highest in memory, in same place for each process
- Locate shared libraries high in memory, in same place for each process
- Locate stack somewhere high, growing downwards
- Locate memory pool for malloc and free low, growing upwards
- Locate program code low

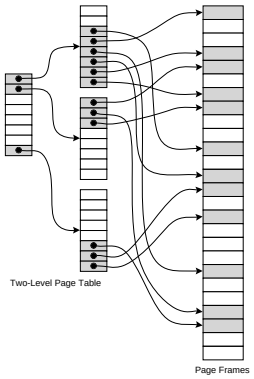
Solution:

- Two-level (or three-level) page tables

Or, alternatively:

- Segmented page tables





Two-level page table:

- 10-bit upper page number (0-1023)
- 10-bit lower page number (0-1023)
- 12-bit offset (0-4095)

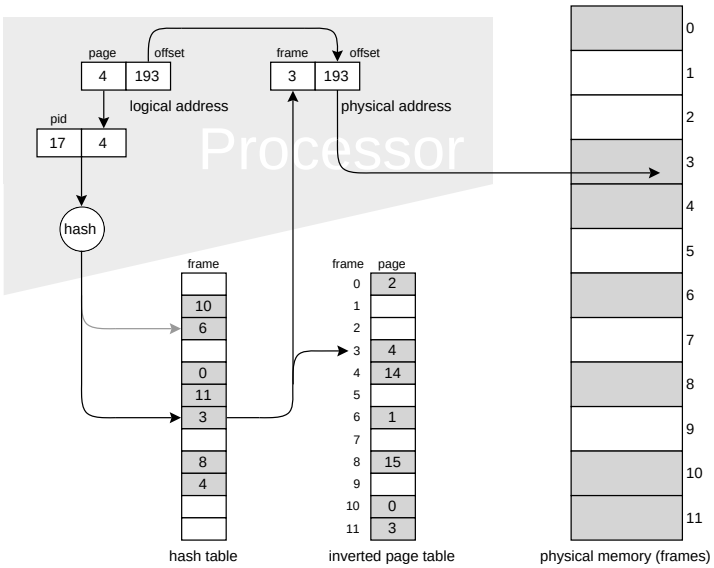
Inverted Page Table

Another solution:

- One entry for each frame of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.

Class Exercise: Are the contents of the hash table defined on a per-process or system-wide basis?

Inverted Page Table (contd.)



Recap

We now have a memory scheme where

- Programs use *logical addresses*
- Memory sharing is easy
- Processes are either in memory or swapped out
- Hardware can detect *invalid* accesses to memory and trap to the operating system

But programs do not need all their code all the time

- Use overlays — work for the programmer
- Or, make paging smarter — eliminate all-or-nothing aspect of swapping
  - Swap out the pages that aren't being used
  - Swap in pages as they are needed

Demand Paging

We now have a memory scheme where we

- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users/processes
- Mark pages not in memory as *invalid* in page table

When program accesses an invalid page, two possibilities

- Need to bring page into memory

Demand Paging — Hardware Support

Thus,

- Invalid accesses generate a trap
- Need to restart program after the trap
- Must seem like "nothing happened"

Example:

- The C-code for:

  ```
  (--mystack) = new_item;
  ```

  may be implemented as a single instruction:

  ```
  move    d2, -(a3)
  ```

  which means "decrement register a3 by one, then store d2 in the address given in a3"

Class Exercise: Why is this instruction problematic to restart if the memory access to store d2 causes a page fault?

Page Faults

Call these traps to OS to load pages ***page faults***

What happens?

- User process accesses invalid memory — traps to OS
- OS saves process state
- OS checks access was actually legal
- Find a free frame
- Read from swap to free frame — I/O wait, process blocked
- Interrupt from disk (I/O complete) — process ready
- Scheduler restarts process — process running
- Adjust page table
- Restore process state
- Return to user code

Page Faults (contd.)

How long?

- Disk is slow
- 25 ms is a conservative guess
- Main memory takes 10–50 ns
- A memory access that causes a page fault page fault is about **1 million times slower** than a regular memory access
- Page faults must be rare! (Need locality!)

Page Faults (contd.)

How often — an example, my workstation

- In the last 25 days
  - 332,273 page-ins
  - 513 hours idle, 87 hours busy
- 10,000,000 memory accesses per second (a guess)
- 313,200 seconds in 87 hours (87 * 60 * 60)
- 3,132,000,000,000 memory accesses in 25 days
- 1 page-in every 9,425,984 memory accesses

Page Faults (contd.)

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 — see your assignment

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.

Thus, we have:

- ***Virtual memory*** — we can provide a larger logical address space than we have physical address space

Page Replacement Algorithms

To evaluate algorithms,

- We want to achieve the lowest page-fault rate
- Run them on a stream of page numbers corresponding to the execution of a (hypothetical?) program

For example, suppose memory accesses by the system are

- **0002**fe00 0002fe04 0002fe08 0002fe0c 0002ff00
  0002ff04 **0003**0216 00030800 **0002**ff08 **0001**6eb0
  00016eb4 00016eb8 **0005**0380 **0002**ff0c 0002ff10
  0002ff14 0002ff18 0002ff1c 0002ff20 0002ff24
  00040d84 00040d88 00040d8c **0005**0380 **0003**0800
  00030216 **0002**ff28 **0005**0380 **0002**ff2c 0002ff30
- The stream of page numbers for the above execution is
  2 3 2 1 5 2 4 5 3 2 5 2

Random (RAND)

Throw out a random page.

- Easy to implement
- May throw 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

First-in First-out Policy (FIFO)

Throw out the oldest page.

- Easy to implement
- May throw 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

  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

Optimal Page-Replacement Policy (OPT)

Choose to replace the page that won't be accessed for the longest time.

- 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.

- Hard to implement
- Fairly close to OPT in performance

Class Exercise: Why is LRU hard to implement?

How would you implement it?

Class Exercise: What's the worst case for LRU?
Can it happen in real programs?

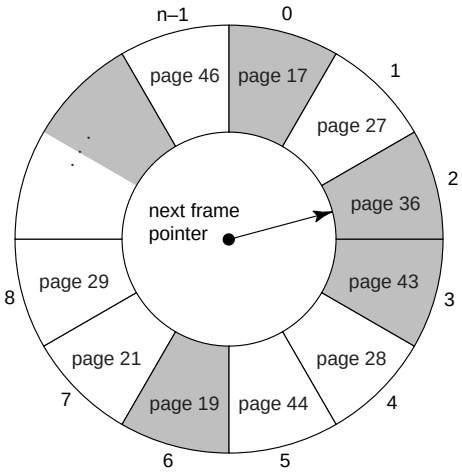
Clock (aka Second Chance)

Hardware maintains a "referenced" bit in the page table, set by hardware when page is accessed (only cleared by software, i.e., the operating system).

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

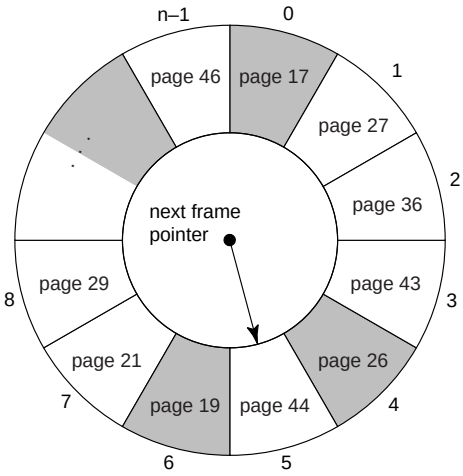
- Easy to implement
- Approximates LRU

Clock (contd.)



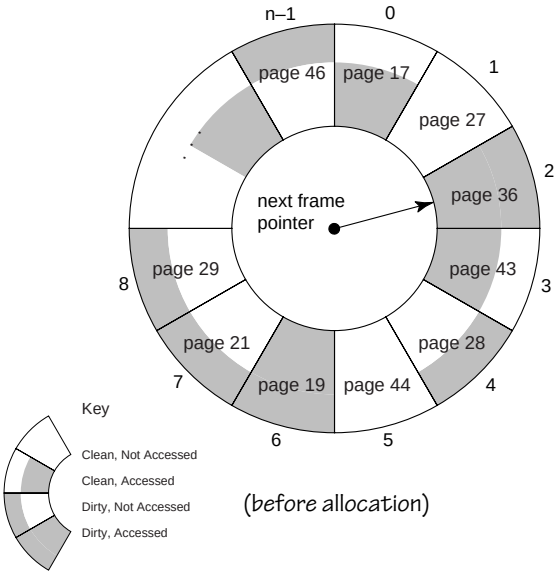
(before allocation)

Clock (contd.)



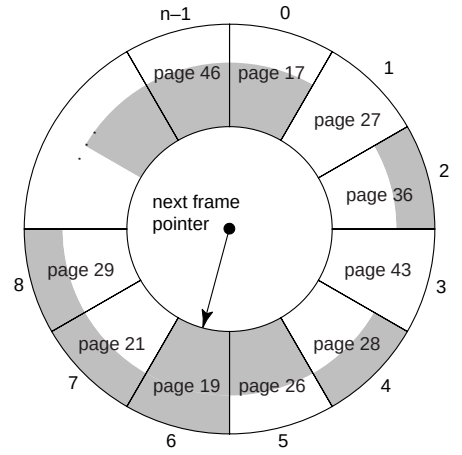
(after allocation)

An Improved Clock



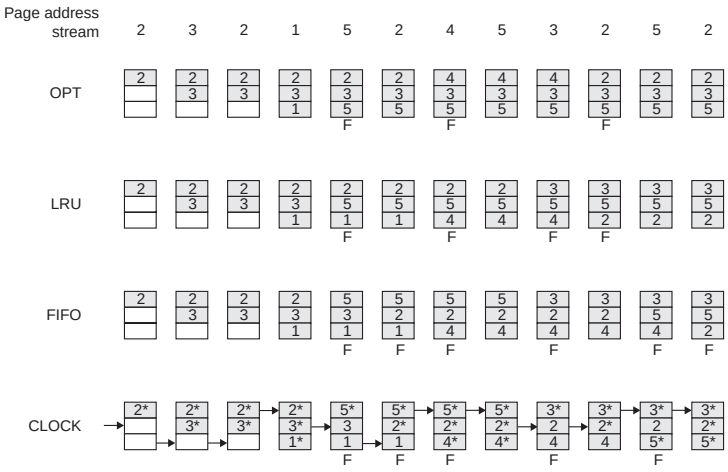
(before allocation)

An Improved Clock



(after allocation)

Comparing the Policies



Page Buffering

Freeing up a page is slow

- Find a page that doesn't appear to be being used much
- Perhaps write it to disk

Maintain a queue of free pages

- Kernel thread finds discardable pages and adds them to the end of the
  - Writeout queue, if the page is dirty
  - Free queue, if the page is clean
- Another kernel thread takes pages from the writeout queue, writes them to disk, and puts them into the free queue

Allow pages to be reprieved

- Even FIFO page replacement is workable with page buffering.

Page Sizes

One of the most popular page sizes is 4 KB

Class Exercise: Why is 4 KB so popular? (Hint, assume a two level page table, where each page table fits on a page).

What is the advantage of large pages?

What is the advantage of smaller pages?

Page Sizes (contd.)

Smaller pages

- Better capture program locality
- Reduce internal fragmentation

Larger pages

- Give better I/O performance
- Reduce page-table size
- Reduce TLB misses

Variable page sizes?

- Try to get best of both worlds
- More complex for hardware and operating system

Page Locking

Sometimes we want to prevent a page from getting paged out, even if it seems like it isn't being used.

- Make the page *locked* (aka *tied down*, *wired down*)

Class Exercise: When would page locking be useful?

Paging the Kernel

Class Exercise: Should (parts of) the operating system be in virtual memory and paged in and out just like user programs?

What are the tradeoffs?

Frame Allocation Policies

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

- Each process needs 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
- Some processes don't need that many pages
- Some processes need more

Make each process declare how many pages it will need beforehand?

— Yuck! (?)

Local, Proportional Frame Allocation

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

- 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

(We could also allocate memory in proportion to process priority, with similar problems.)

Global, Variable Allocation

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

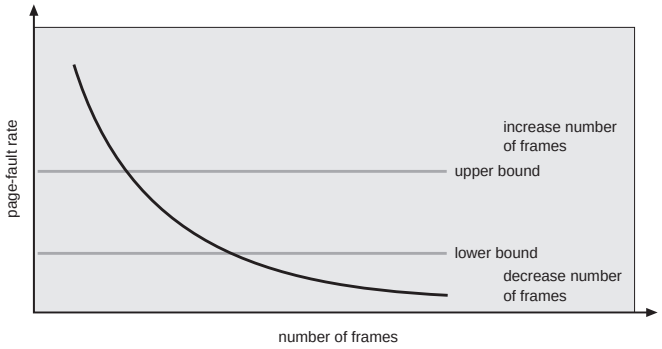
Class Exercise: Is this policy fair?

If not, why not?

Working Sets

As you take pages away from a process, its 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



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.

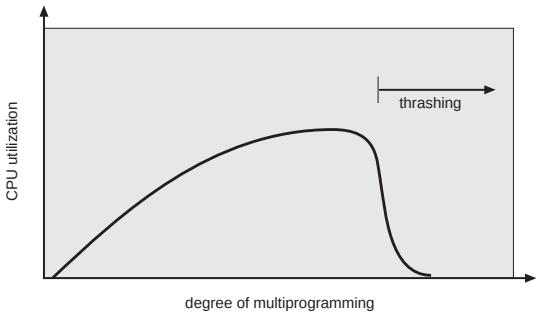
Class Exercise: What's wrong with this policy (i.e., what assumptions are we making that could be wrong) ?

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

Thrashing

If a process does not have "enough" pages, the page-fault rate is very high. This leads to:

- Low CPU utilization
- Lots of I/O activity



Thrashing (contd.)

Under local replacement policies

- Mostly, just problem process affected

Under global replacement policies

- Whole machine brought to its knees

Segment Review

You should be able to:

- Describe fixed and dynamic partitioning
- Describe internal and external fragmentation
- Contrast paging and segmentation
- Implement shared memory in all forms of paging and segmentation systems
- List assumptions we typically make about programs
- Determine the assumptions made in any memory architecture, and how performance varies between situations where these assumptions are true and situations where these assumptions are false
- Explain the trade-offs of logically addressed and physically addressed processor caches
- Describe and contrast possible layouts for page tables
- Describe and give the tradeoffs of RAND, FIFO, LRU, and Clock page replacement policies
- List conditions for thrashing

Segment Review (continued)

- Explain the trade-offs involved in deciding a page size
- Explain the trade-offs involved in paging the kernel
- Give reasons for locking a page in memory
- Describe and contrast frame allocation schemes