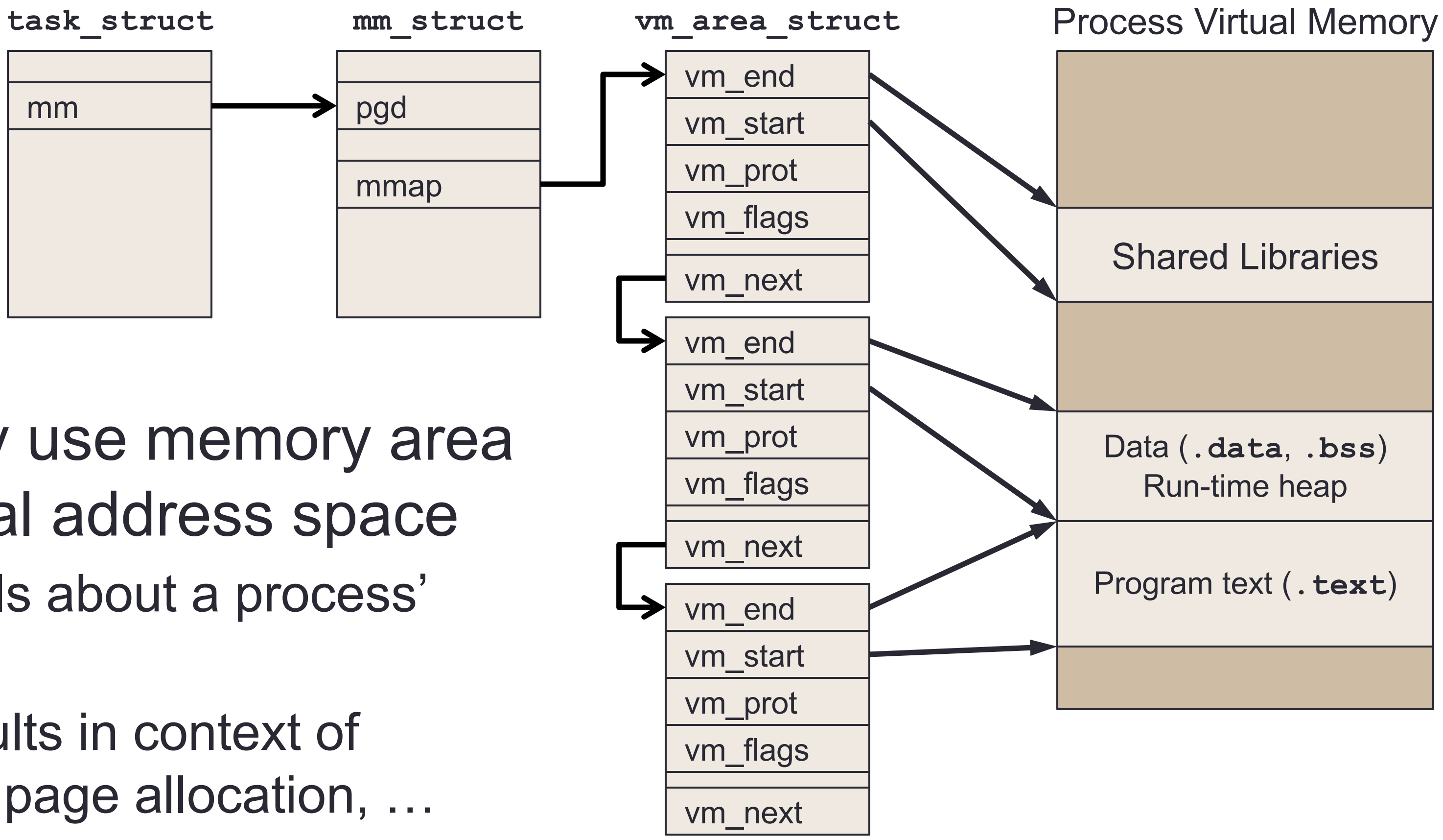
VIRTUAL MEMORY MANAGEMENT

CS124 – Operating Systems
Spring 2024, Lecture 17

Last Time: Memory Descriptors

- Last time, began discussing how the kernel manages virtual memory



- Example: kernels frequently use memory area descriptors to describe virtual address space
  - Keep track of higher-level details about a process' address space
  - Essential for resolving MMU faults in context of copy-on-write, shared memory, page allocation, …

Memory Descriptors (2)

- Besides virtual memory descriptors, kernels must also keep track of several other key details:
- Information about physical page frames
  - What frames can be used by processes, used for I/O buffers, etc.
  - What frames are currently in use, and by whom
- Information about swap space used by the system
  - Where is the swap space on disk
  - What locations are currently available to store a virtual page
  - What locations are occupied by a page, and whose page is it
  - (Mobile operating systems won’t have this)
- Managing this information is made more complex by the fact that multiple processes can share pages
  - Requires careful design to be efficient, avoid race conditions, etc.

Managing Page Frames

- The kernel maintains details about page frames in a **frame table**
- Different physical memory regions may be used for different purposes in the system
  - Kernel will require certain page frames for its own code and data
  - Also, many peripherals may require a physically contiguous memory area, in a specific address range, for DMA transfers
  - Remaining frames can be assigned to user processes for a variety of purposes

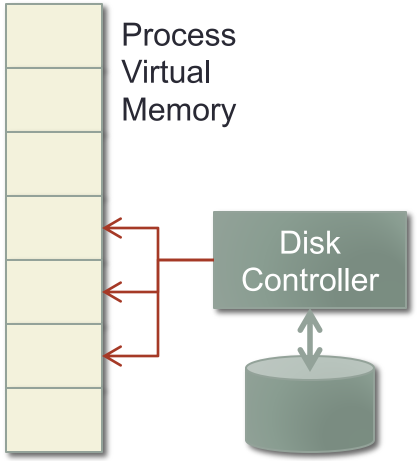
- Each entry of the frame table holds flags describing what the frame is being used for (or can be used for)

Managing Page Frames (2)

- Also need to record which frames are currently in use
- What process (or processes) is using each page frame?
  - When a page is evicted from a frame, must update the page tables of all processes that reference the page
- Where is the data in the page frame from?
  - When the page is evicted from the frame, the page’s origin affects what must be done
- Is the page in the frame currently **pinned**?
  - Pinned pages are not allowed to be evicted from physical memory

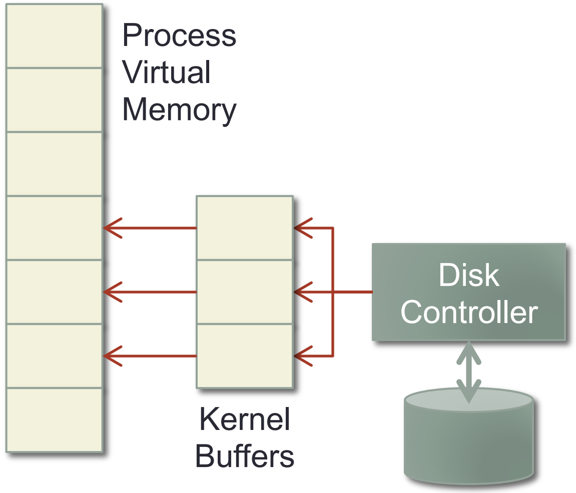
Managing Page Frames: Pinning

- A page can be pinned if it is currently being used by some long-running task
- A common scenario: a process requests an I/O operation
  - e.g. read or write multiple blocks of a disk file
- The kernel sets up a DMA transfer into specific virtual pages in the process' address space
  - But, this transfer will take some time to finish…
- If kernel chooses to evict some pages from memory, it cannot evict pages being used by the external peripheral
- The kernel can pin these pages so that the virtual memory pager won't evict them



Managing Page Frames: Pinning (2)

- Alternatively, the kernel can maintain its own I/O buffers
  - The DMA transfer is set up into the kernel's pages, not the process
  - When the I/O is complete, the kernel copies the data into the process' pages
- Allows the process to be entirely paged out of memory…
- But, this approach has several issues:
- Data is copied twice instead of once
  - Causes a significant performance impact
- Uses up more virtual memory than is strictly required for the transfer

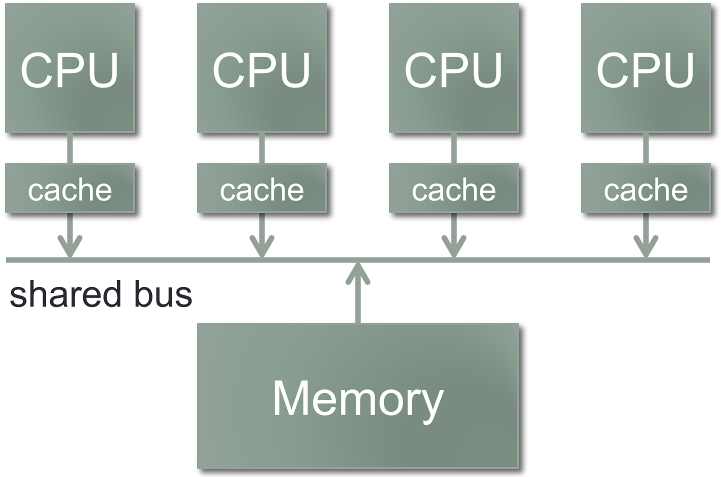


Managing Page Frames: Pinning (3)

- Several other reasons to support pinning pages into frames
- Frequently, some or all of the kernel pages are pinned
  - Don’t allow some or all of the kernel to be swapped out of memory
- Also can be used to manage newly swapped-in processes
- Example: a low-priority process *L* page-faults…
  - Kernel starts loading the required virtual memory page; *L* is blocked
  - When *L*’s page is loaded, it reenters the ready queue
  - But, might be a while before it receives the CPU
- After *L*’s page is loaded, but before *L* runs, a high-priority process *H* also page-faults
  - In a low-memory situation, the kernel must find some page to evict…
  - “Hey look, *L*’s page is unaccessed and unmodified… evict it!”
  - As part of the paging policy, the kernel can pin newly loaded pages until the corresponding process has had a chance to run

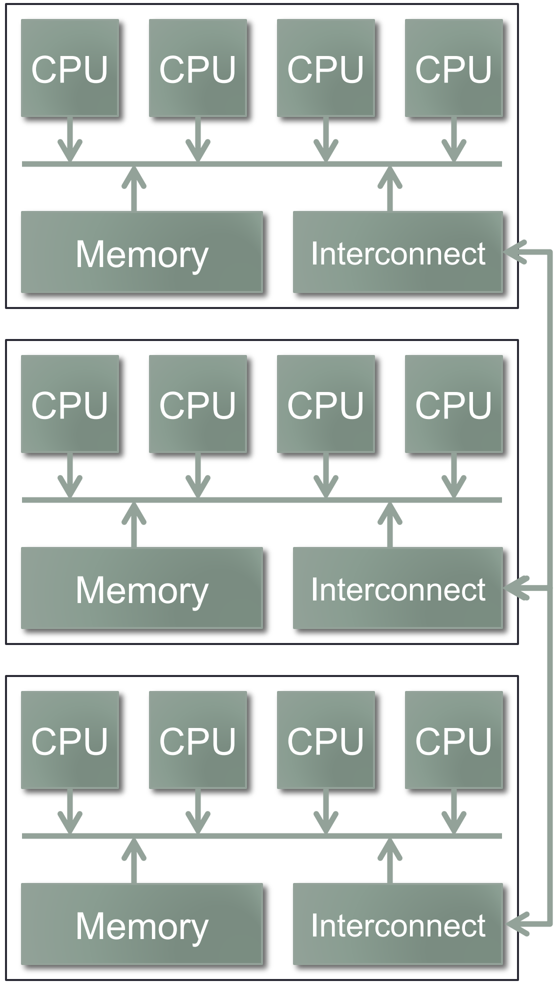
Multiprocessor Systems and Memory

- Multiprocessor systems can dramatically increase the complexity of memory management
- Smaller multiprocessor systems usually implement symmetric multiprocessing (a.k.a. SMP)
  - All processors have equal access to a centralized shared memory
  - Also called "uniform memory access"
- As multiprocessor systems scale, this approach becomes infeasible
  - Bus contention for accessing central memory becomes prohibitive
- Doesn't produce much benefit anyway: a given memory area usually won't be accessed by that many processors over a short period of time



Multiprocessor Systems and Memory (2)

- Larger multiprocessor systems often implement Non-Uniform Memory Access (NUMA)
  - A processor (or group of processors) has its own dedicated memory
  - Processors can access nonlocal memory transparently, but it's significantly slower to access
- Clearly the OS must be aware of what memory regions are fastest for each processor to access
- Frames can have a processor affinity
  - When OS assigns a page to a frame, it must ensure that the frame is on same CPU as the process



Frame Table Entries

- Kernels need to use small structures to track frame info
  - Don’t want to lose too much memory space due to recording and managing this information
- Example: Linux **`page`** descriptors are 32 bytes
  - Each “page descriptor” describes a page frame, including flags, a reference count, how many PTEs reference the frame, etc.
  - Less than 1% of memory is lost to these page descriptors
- **`page`** descriptors *indirectly* record all processes using a given page frame
  - Would be prohibitive to maintain e.g. a list of processes at this level
  - Instead, **`page`** descriptors contain a pointer to a high-level structure that references all virtual memory areas containing the page frame

Page Frame Contents

- The page in a frame can originate from several places
- **Anonymous memory** is memory whose contents do not come from a specific filesystem file
  - Used for general purposes, e.g. the memory heap, process stack, uninitialized program data, some kinds of shared memory, etc.
- When an anonymous memory page is initially allocated, the frame’s contents are simply initialized to all zeros
  - Prevent one process from seeing another process’ data
- When a page of anonymous memory is evicted, it must be stored in the system’s swap memory
  - It doesn’t have a specific file associated with it, so there’s no predetermined place to store it
  - (Similarly, an anonymous page doesn’t have an associated swap location until it has been evicted at least once…)

Page Frame Contents (2)

- A page may also come from a **memory-mapped file**
- The page's contents are initially loaded from a specific part of a file on the computer's filesystem
  - The virtual memory system effectively maps a file's contents into one or more page frames
- When the page is evicted from physical memory, it can be stored either in a swap area or in the originating file
  - Depends on what the file's contents are being used for
  - Kernel has several options in this circumstance
- Example: a binary program mapped into virtual memory
  - Probably want to disallow writing to the virtual pages anyway…
  - If page is evicted, don't need to write anything back to original file
  - When page is reloaded into memory, simply retrieve contents of original file again

Page Frame Contents (3)

- Example: page containing non-constant initialized data from a binary program
  - Definitely need to allow changes to this data in memory…
    - (OS can use copy-on-write if multiple processes run the same program)
  - If the page is evicted, don't want to write it back to the original file! Otherwise, future invocations of the file would see the changes.
  - Instead, save it to a separate swap area
- Example: page of a data file mapped into memory
  - The program *intends* to make changes to the data file in memory, and the program *intends* those changes to be written back to disk
  - In this case, if the page is evicted, write it back to the original file
  - (In fact, may want to synchronize the page back to disk more frequently so that other processes also see the file's changes)

Virtual Pages and Swap Space

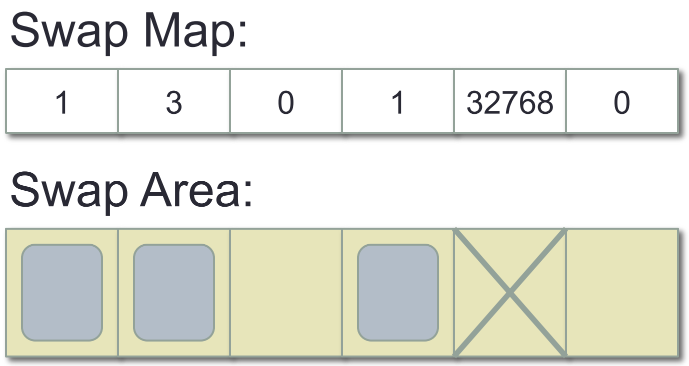
- Some pages must be saved into some kind of swap space
  - (When a page's changes will be discarded at process termination)
- Two choices:
  - A dedicated **swap partition**
  - A **swap file** managed on the computer's filesystem
- Dedicated swap partitions are generally much faster
  - No complex filesystem structures to navigate or manage
  - Storage layout is optimized for speed
  - Even if internal fragmentation occurs, swap partition is reinitialized every time the OS boots
- Problem: much harder to resize a dedicated swap partition
  - If swap memory isn't large enough for OS needs, cannot be resized automatically; requires administrator intervention

Virtual Pages and Swap Space (2)

- Dedicated swap partitions must also handle bad blocks
  - Filesystems typically handle this issue for us, but swap partitions don’t have that benefit
- Swap files tend to be slower to access
  - Must navigate and manage the filesystem structure
  - Swap file may become fragmented across the disk
- But, swap files can be resized much more easily when space needs to be increased
- Windows and macOS both use swap files
  - e.g. macOS swap files reside in **`/private/var/vm`** directory
- Linux can use either swap partitions or a swap files
  - Swap partition is preferred, for performance reasons

Swap Slots

- Storage used for page swapping is divided into **slots**
  - Each slot can hold one virtual page
- Required operations:
  - Find a free slot to store a page in, and save the page to the slot
  - Load a page from a slot, and possibly release the slot for reuse
- Linux uses a **swap map** to describe slots in a **swap area**
  - An array of counters specifying how many processes are using each corresponding slot
  - 0 means the slot is available for use
  - >1 means slot is shared by multiple processes (e.g. a shared library)
  - 32768 means the slot contains bad sectors and cannot be used



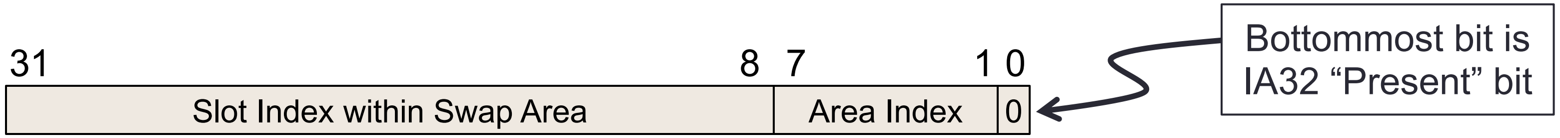
Swap Map:

1	3	0	1	32768	0

Swap Area:

Swap Slots (2)

- Linux supports having many swap areas (128 on 32-bit)
  - Swap area descriptors are maintained in an array
- A specific swap slot is identified by two values: the index of the swap area, and the index of the slot within the area
- These values are packed into a 32-bit value:



  - Given 4KiB pages, each swap area can hold up to 2^{24} pages, or 64GiB of swap space
  - With up to 128 swap areas, can have up to 8TiB of swap space
- The slot ID is stored into the page table entry of a swapped-out virtual page
  - Page fault handler can easily use this to reload a page into memory

Swap Slots (3)

- When a page fault occurs, the Linux page-fault handler can easily identify the specific swap slot that was accessed

31 8	7 1	0
Slot Index within Swap Area	Area Index	0

- Swap slot information specifies if the page is stored in swap space, or from a memory-mapped file
  - Kernel can go to the appropriate storage location and reload the page into memory
- Kernel page-fault handler:
  - Allocate an unused frame from the frame table
  - Update the process’ page table to refer to the frame
  - Load the page from disk into the frame (either from swap area, or from a specific named file and offset within the file)

Virtual Memory Policies

- Two major questions the kernel virtual memory system must answer:

- When a page frame must be reclaimed, how to choose which page to evict from memory?
  - This is determined by the **page replacement policy**

- How many page frames should each process be allowed to occupy?
  - e.g. should higher priority processes receive more page frames?
  - This is determined by the **page allocation policy**

Virtual Memory Measurements

- Obvious goal of page replacement policy is to minimize the number of page faults that occur over time
- There are many different page replacement policies…
- Must evaluate them against example sequences of memory accesses
    - Most useful if collected from actual program execution traces
    - Can also generate randomly, but this really won't reflect the typical program behavior
- Given a sequence of memory accesses, simulate a page replacement policy and determine its page-fault rate
- Better page replacement policies should, on average, generate lower page-fault rates

Virtual Memory Measurements (2)

- A program's memory access trace can be very verbose
- Given a sequence of memory accesses, e.g.
  - 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105, …
- Can shrink the size of this sequence in two ways
- First, we only care about which virtual pages were accessed, not the offsets within the pages
  - e.g. if the above memory had 100B pages, sequence becomes:
    1, 4, 1, 6, 1, 1, 1, 1, 6, 1, 1, 1, 1, 6, 1, 1, 1, 1, 6, 1, 1, …
- Second, adjacent accesses to the same page are highly unlikely to cause a page fault, in the average case
  - Eliminate repeated accesses to adjacent pages to produce:
    1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1, …
- Resulting sequence is called a **reference string**

Virtual Memory Measurements (3)

- Besides the replacement policy and a reference string, we must also know how many page frames are available
- Assumption: as the number of frames increases, the number of page faults should decrease
- Surprisingly, this isn't always the case (!!!)

- Some replacement policies exhibit **Belady's anomaly**
  - As the total number of frames increases, the page fault rate may also sometimes increase
  - Named after László Bélády, who discovered this anomaly in 1969
- An "ideal" replacement policy will never suffer from Belady's anomaly
  - Adding frames to the system will never increase the page-fault rate

FIFO Page Replacement Policy

- Simplest page replacement policy is **FIFO policy**
- Kernel pager maintains a FIFO queue for virtual pages
- When a page is brought into memory, it is added to the end of the FIFO
- When a page must be evicted from memory, it is taken from front of the FIFO
    - Pages will eventually make their way from back of FIFO to the front

- Note that whether a page has been accessed (or whether it is dirty) has nothing to do with when it is evicted
    - An extremely simplistic policy…

FIFO Page Replacement Policy (2)

- Example: memory with 3 page frames
  - Our FIFO will hold a maximum of 3 pages
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Sequence of accesses:

Page 1 (fault)	FIFO:			1	Page 5 (fault)	FIFO:	1	2	5
Page 2 (fault)	FIFO:		1	2	Page 1	FIFO:	1	2	5
Page 3 (fault)	FIFO:	1	2	3	Page 2	FIFO:	1	2	5
Page 4 (fault)	FIFO:	2	3	4	Page 3 (fault)	FIFO:	2	5	3
Page 1 (fault)	FIFO:	3	4	1	Page 4 (fault)	FIFO:	5	3	4
Page 2 (fault)	FIFO:	4	1	2	Page 5	FIFO:	5	3	4

- Out of 12 accesses, 9 produce page faults. Yuck.

FIFO Page Replacement Policy (3)

- What about increasing our physical memory to 4 frames?
  - Now the FIFO will hold 4 pages
- Same reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Sequence of accesses:

Page 1 (fault)	FIFO:				1	Page 5 (fault)	FIFO:	2	3	4	5
Page 2 (fault)	FIFO:			1	2	Page 1 (fault)	FIFO:	3	4	5	1
Page 3 (fault)	FIFO:		1	2	3	Page 2 (fault)	FIFO:	4	5	1	2
Page 4 (fault)	FIFO:	1	2	3	4	Page 3 (fault)	FIFO:	5	1	2	3
Page 1	FIFO:	1	2	3	4	Page 4 (fault)	FIFO:	1	2	3	4
Page 2	FIFO:	1	2	3	4	Page 5 (fault)	FIFO:	2	3	4	5

- Now, out of 12 accesses, 10 produce page faults! Worse!

Optimal Page Replacement Policy

- A better policy: the **optimal page-replacement policy**
  - Always evict the page that will not be used for the longest time
- This policy doesn't suffer from Belady's anomaly
  - Guaranteed to minimize the number of page faults, given a specific number of page frames
- One small problem: the OS must be able to predict the future…
  - Kernel pager has no idea what memory processes might access
  - (This policy is also called the **clairvoyant replacement policy**)
- But, we can always try to approximate the optimal policy
  - Use a process' previous behavior to predict its future behavior
- Also, if we have the full memory trace, we can simulate the optimal policy
  - Very helpful to compare different policies to the optimal policy
  - e.g. "a given policy comes within 5% of optimal, on average"

Optimal Page Replacement Policy (2)

- Previous example: memory with 3 page frames
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - Since we know the reference string, we know the optimal choices
- Sequence of accesses:

Page 1 (fault) OPT: 1
Page 2 (fault) OPT: 1 2
Page 3 (fault) OPT: 1 2 3
Page 4 (fault) OPT: 1 2 4

Page 1 accessed in 1 step
Page 2 accessed in 2 steps
Page 3 accessed in 6 steps
→ Evict page 3!

Page 1 OPT: 1 2 4
Page 2 OPT: 1 2 4
Page 5 (fault) OPT: 1 2 5

Page 1 accessed in 1 step
Page 2 accessed in 2 steps
Page 4 accessed in 4 steps
→ Evict page 4!

Page 1 OPT: 1 2 5
Page 2 OPT: 1 2 5
Page 3 (fault) OPT: 3 2 5

Only page 5 will be accessed again; evict page 1 or 2

Page 4 (fault) OPT: 3 4 5

Only page 5 will be accessed again; evict page 2 or 3

Page 5 OPT: 3 4 5

- Optimal policy: only 7 faults

Optimal Page Replacement Policy (3)

- Now, try a memory with 4 page frames
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Sequence of accesses:

Page 1 (fault)	OPT:	1			
Page 2 (fault)	OPT:	1	2		
Page 3 (fault)	OPT:	1	2	3	
Page 4 (fault)	OPT:	1	2	3	4
Page 1	OPT:	1	2	3	4
Page 2	OPT:	1	2	3	4

Page 5 (fault)	OPT:	1	2	3	5

Page 4 is accessed furthest into future → evict page 4

Page 1	OPT:	1	2	3	5
Page 2	OPT:	1	2	3	5
Page 3	OPT:	1	2	3	5
Page 4 (fault)	OPT:	4	2	3	5

Only page 5 will be accessed again; evict pages 1-3

Page 5	OPT:	4	2	3	5

- Now the optimal policy only generates 6 faults. Nice.

Next Time

- Continue discussion of page replacement policies
    - How can we approximate the optimal replacement policy?