Page Replacement
Chap 21, 22

Virtual Memory Concept

- **Virtual memory**
  - Concept
    - A technique that allows the execution of processes that are not completely in memory
      - Partition each user's program into multiple blocks
      - Load into memory the blocks that is necessary at each time during execution
        - Only part of the program needs to be in memory for execution
        - Noncontiguous allocation
      - Logical memory size is not constrained by the amount of physical memory that is available
  - Separation of logical memory as perceived by users from physical memory

Virtual Memory Concept

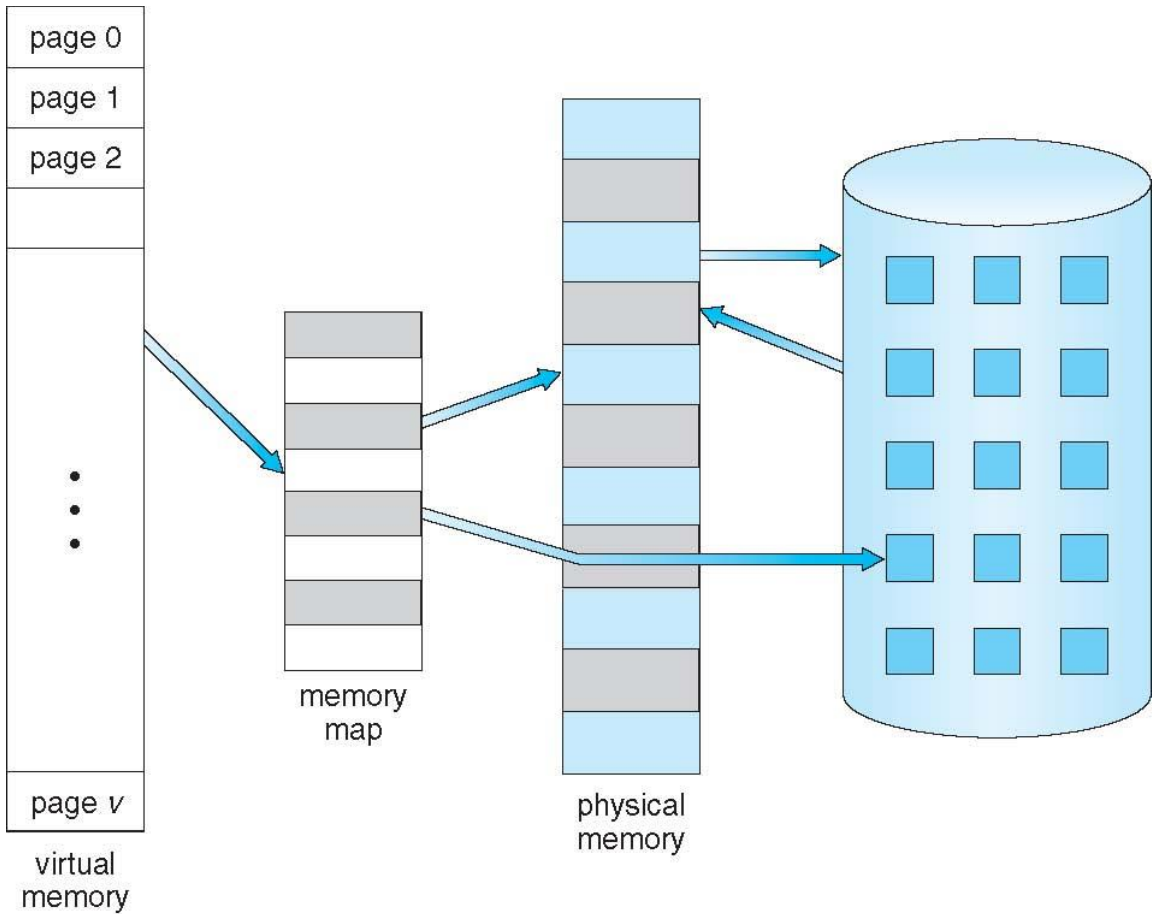
- **Virtual memory**
  - Benefits
    - Easier programming
      - Programmer no longer needs to worry about the amount of physical memory available
      - Allows address spaces to be shared by several processes
    - Higher multiprogramming degree
      - Increase in CPU utilization and throughput (not in response time and turnaround time)
    - Less I/O for loading and swapping processes into memory
      - Faster execution of processes

Virtual Memory Concept

- **Virtual memory**
  - Drawbacks
    - Address mapping overhead
    - Page fault handling overhead
    - Not adequate for real-time (embedded) systems

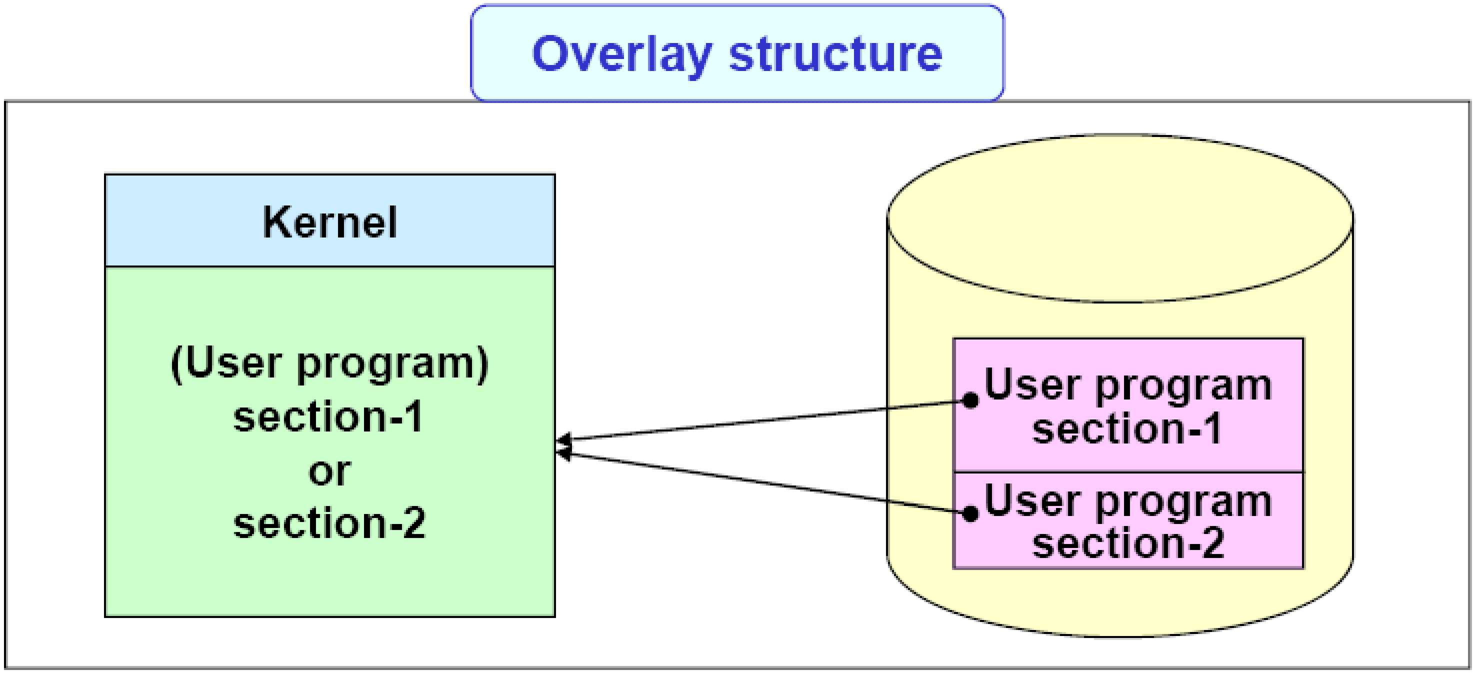
Virtual Memory That is Larger Than Physical Memory

OS make use of a larger, slower device to transparently provide the illusion of a large virtual address space



Memory Overlay (old system)

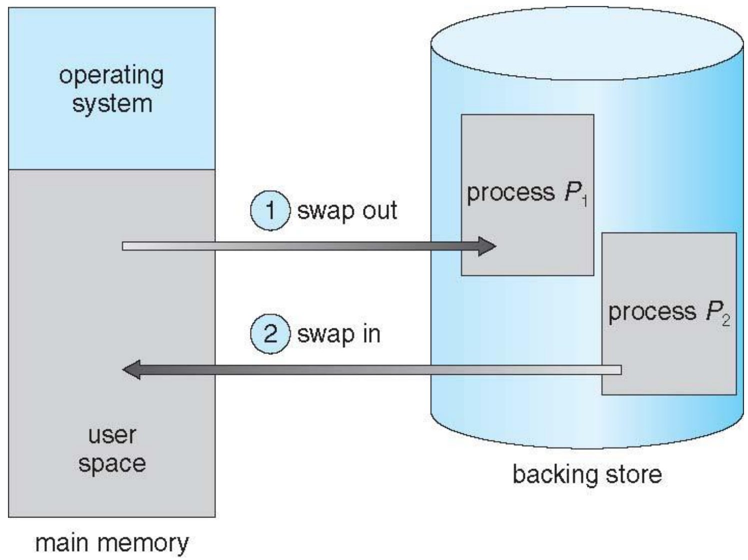
- **Program-size > memory-size**
  - w/o OS support
  - Requires support from compiler/linker/loader



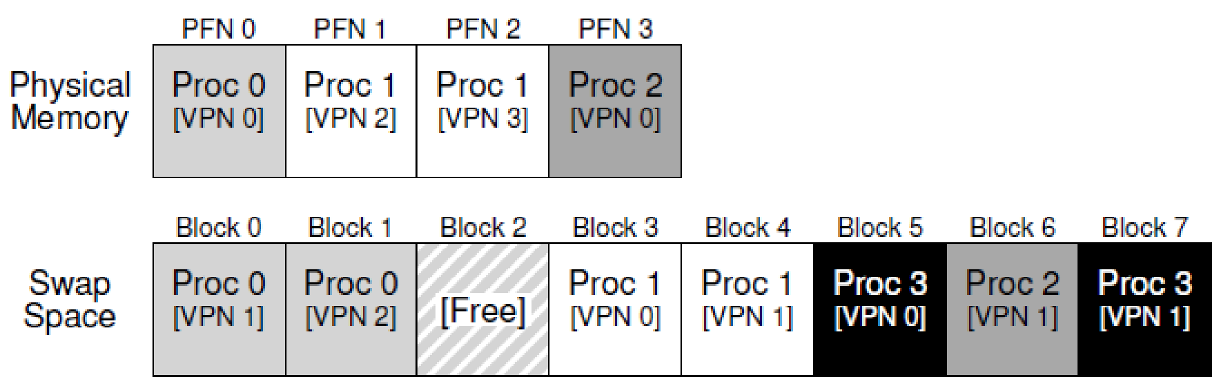
Swapping

- A process can be swapped temporarily out of memory to a **backing store** (**swap device**)

Process-level swapping



Page-level swapping



Swapping

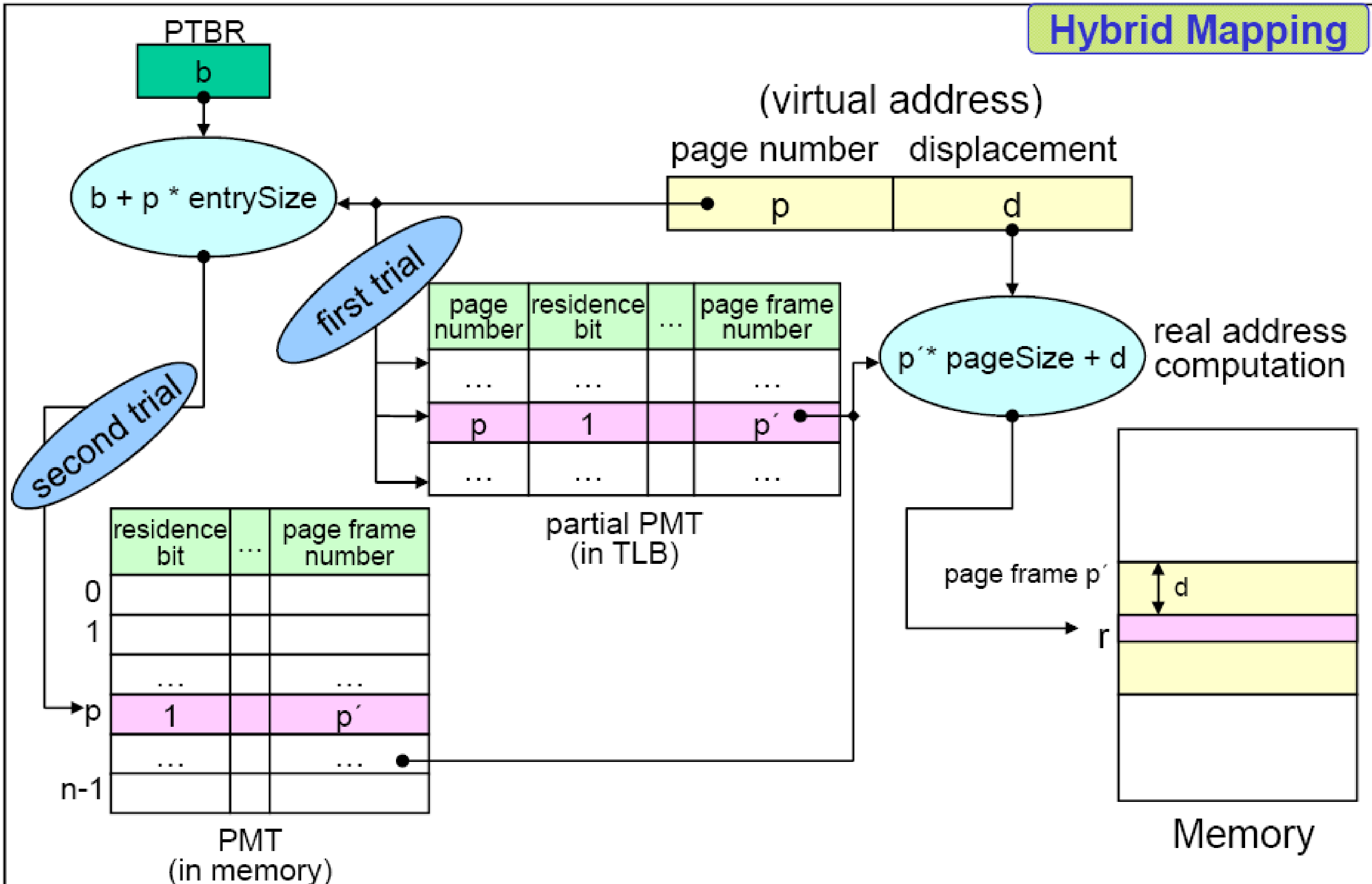
- **Notes on swapping**
  - Time quantum vs swap time
    - Time quantum should be substantially larger than swap time (context switch time) for efficient CPU utilization
  - Memory areas to be swapped out
    - Swap only what is actually used
  - Pending I/O
    - If the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped
    - Solutions
      - Never swap a process with pending I/O
      - Execute I/O operations only into kernel buffers (and deliver it to the process memory when the process is swapped in)

Demand Paging

- **Paging (Demand paging) system**
  - Partition the program into the same size blocks (pages)
  - Loading of executable program
    - Initially, load pages only as they are needed
    - During execution, load the pages when they are demanded (referenced)
    - Pages that are never accessed are never loaded into physical memory
  - With each page table entry a present (residence) bit is associated
    - Present = true: in-memory, memory resident
    - Present = false: not-in-memory
  - Initially present bit is set to false on all entries
  - During MMU address translation, if present bit in page table entry is false $\Rightarrow$ page fault

Demand Paging

- Address Mapping



Page Fault

- **If there is a reference to a page, first reference to that page will trap to operating system:**

  page fault

1. **Operating system looks at another table to decide:**
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
2. **Find free frame**
3. **Swap page into frame via scheduled disk operation**
4. **Reset tables to indicate page now in memory**
   Set present bit = T
5. **Restart the instruction that caused the page fault**

Steps in Handling a Page Fault

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                    // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else
        if (CanAccess(PTE.ProtectBits) == False)
            RaiseException(PROTECTION_FAULT)
        else if (PTE.Present == True)
            // assuming hardware-managed TLB
            TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
            RetryInstruction()
        else if (PTE.Present == False)
            RaiseException(PAGE_FAULT)
```

```
PFN = FindFreePhysicalPage()
if (PFN == -1)                  // no free page found
    PFN = EvictPage()           // run replacement algorithm
DiskRead(PTE.DiskAddr, pfn)     // sleep (waiting for I/O)
PTE.present = True              // update page table with present
PTE.PFN     = PFN               // bit and translation (PFN)
RetryInstruction()              // retry instruction
```

Stages in Demand Paging

1. **Trap to the operating system**
2. **Save the user registers and process state**
3. **Determine that the interrupt was a page fault**
4. **Check that the page reference was legal and determine the location of the page on the disk**
5. **Issue a read from the disk to a free frame:**
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. **While waiting, allocate the CPU to some other user**
7. **Receive an interrupt from the disk I/O subsystem (I/O completed)**
8. **Save the registers and process state for the other user**
9. **Determine that the interrupt was from the disk**
10. **Correct the page table and other tables to show page is now in memory**
11. **Wait for the CPU to be allocated to this process again**
12. **Restore the user registers, process state, and new page table, and then resume the interrupted instruction**

Performance of Demand Paging

- **Effective access time**
  - Memory access time
    - 10 ~ 200 nanoseconds (Assume 200ns)
  - Average paging service time: about 8 ms
  - Page fault rate: p ($0 \le p \le 1$)
  - EAT(Effective Access Time)
    - EAT = (1-p)*ma + p*PagingTime
      = (1-p)*200 + p*8,000,000
      = 200 + 7,999,800*p
    - When p = 1/1000, EAT = 8.2 us (40 x ma)
  - If we want less than 10% degradation,
    - EAT = 200 + 7,999,800*p < 220
    - P < 0.0000025 (= 1/400,000)

Demand Paging Optimizations

- **Swap space I/O faster than file system I/O even if on the same device**
  - Swap allocated in larger chunks, less management needed than file system
- **Demand page in from program binary on disk, but discard rather than paging out when freeing frame**
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- **Prefetching**
  - OS could guess that a page is about to be used, and thus bring it in ahead of time
- **Mobile systems**
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)
  - cf. zswap

Page Replacement

- Prevent over-allocation of memory
- Use modify (update, dirty) bit to reduce overhead of page transfers
  - only modified pages are written to disk
  - If modify == 1, the contents of the page in memory and in disk are not same
    - Write-back (to disk) is necessary for the page
- Large virtual memory can be provided on a smaller physical memory

Page Replacement

1. Find the location of the desired page on disk

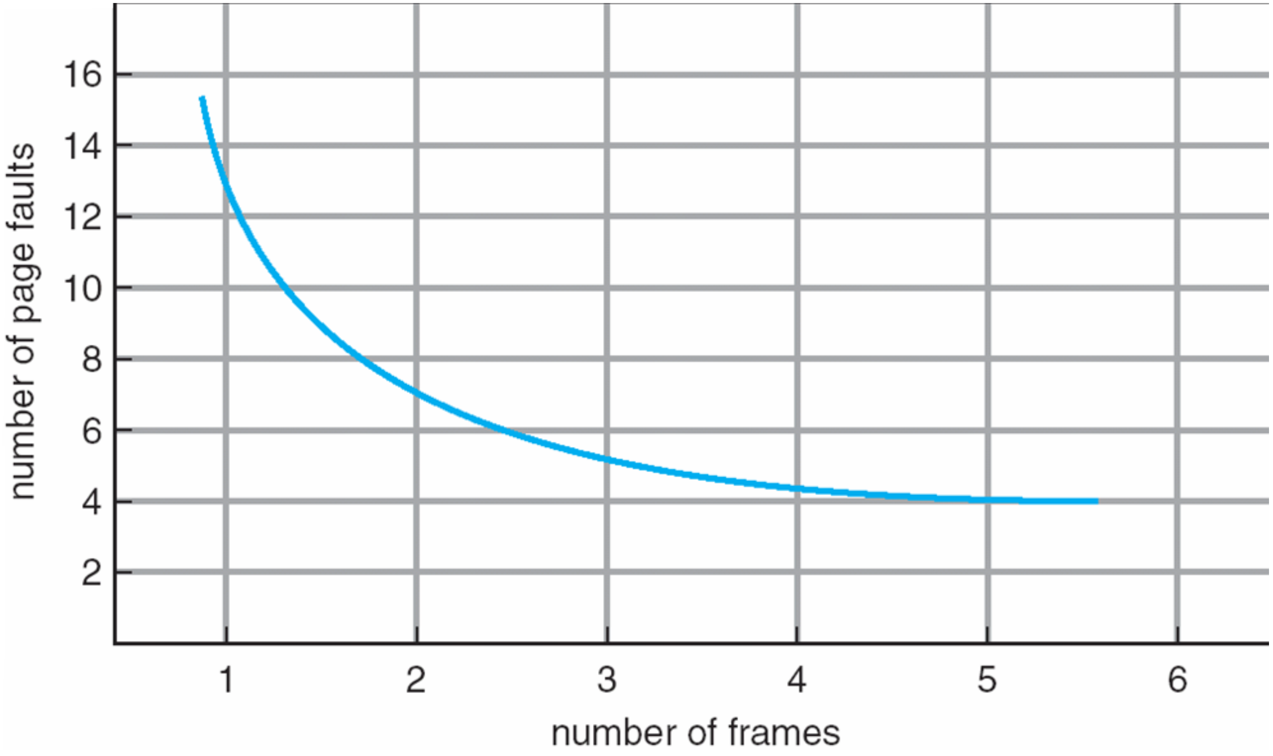
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame
     - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

Graph of Page Faults Versus The Number of Frames



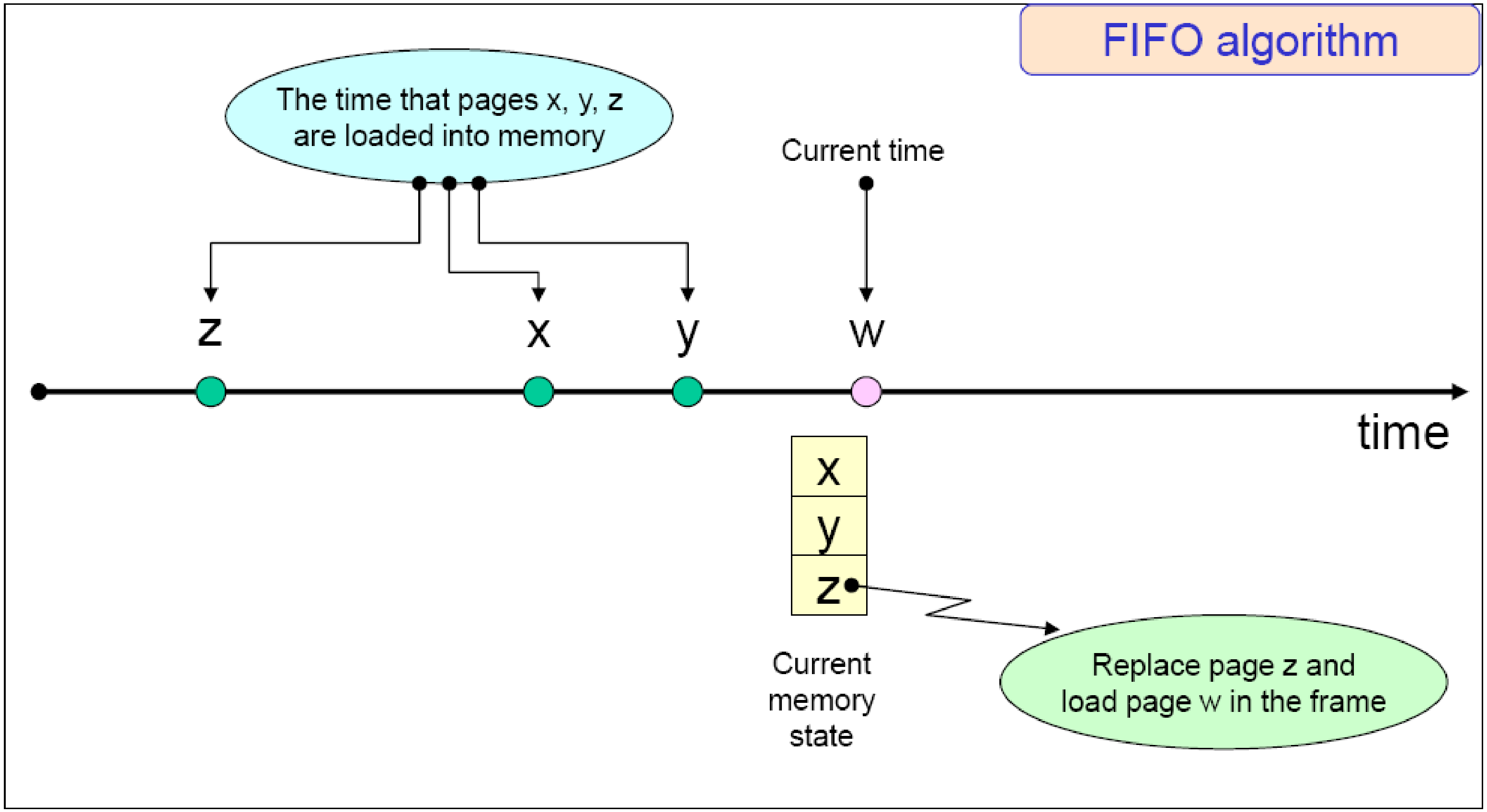
When Replacements Really Occur?

- OS keeps a small portion of memory free more proactively
- Watermark scheme
  - high watermark (HW) and low watermark (LW)
  - When OS notices that there are fewer than LW pages available, a background thread (swap daemon or page daemon) that is responsible for freeing memory runs.
  - The thread evicts pages until there are HW pages available.
  - The background thread then goes to sleep.
  - many systems will cluster or group a number of pages and write them out at once to the swap partition, thus increasing the efficiency of the disk

First-In-First-Out (FIFO) Algorithm

- Choose the page to be replaced based on when the page is previously loaded into memory
- Scheme
  - Replace the oldest page
- Requirements
  - Timestamping (memory load time for each page) is necessary
- Characteristics
  - May replace frequently used pages
- **FIFO anomaly** (**Belady's anomaly**)
  - In FIFO algorithm, page fault frequency may increase even if more memory frames are allocated

First-In-First-Out (FIFO) Algorithm



First-In-First-Out (FIFO) Algorithm

- **FIFO algorithm: Example**
  - 4 page frames allocated, initially empty

ω = 1 2 6 1 4 5 1 2 1 4 5 6 4 5

Memory state change (FIFO)

Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Ref. string	1	2	6	1	4	5	1	2	1	4	5	6	4	5
Memory state	1	1	1	1	1	5	5	5	5	5	5	5	4	4
		2	2	2	2	2	1	1	1	1	1	1	1	5
			6	6	6	6	6	2	2	2	2	2	2	2
					4	4	4	4	4	4	4	6	6	6
Page fault	F	F	F		F	F	F	F				F	F	F

- Number of page faults: 10

First-In-First-Out (FIFO) Algorithm

- **FIFO algorithm: Anomaly example**

ω = 1 2 3 4 1 2 5 1 2 3 4 5

Number of page frames: 3

Time	1	2	3	4	5	6	7	8	9	10	11	12
Ref. string	1	2	3	4	1	2	5	1	2	3	4	5
Memory state	1	1	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4
Page fault	F	F	F	F	F	F	F			F	F	

Number of page frames: 4

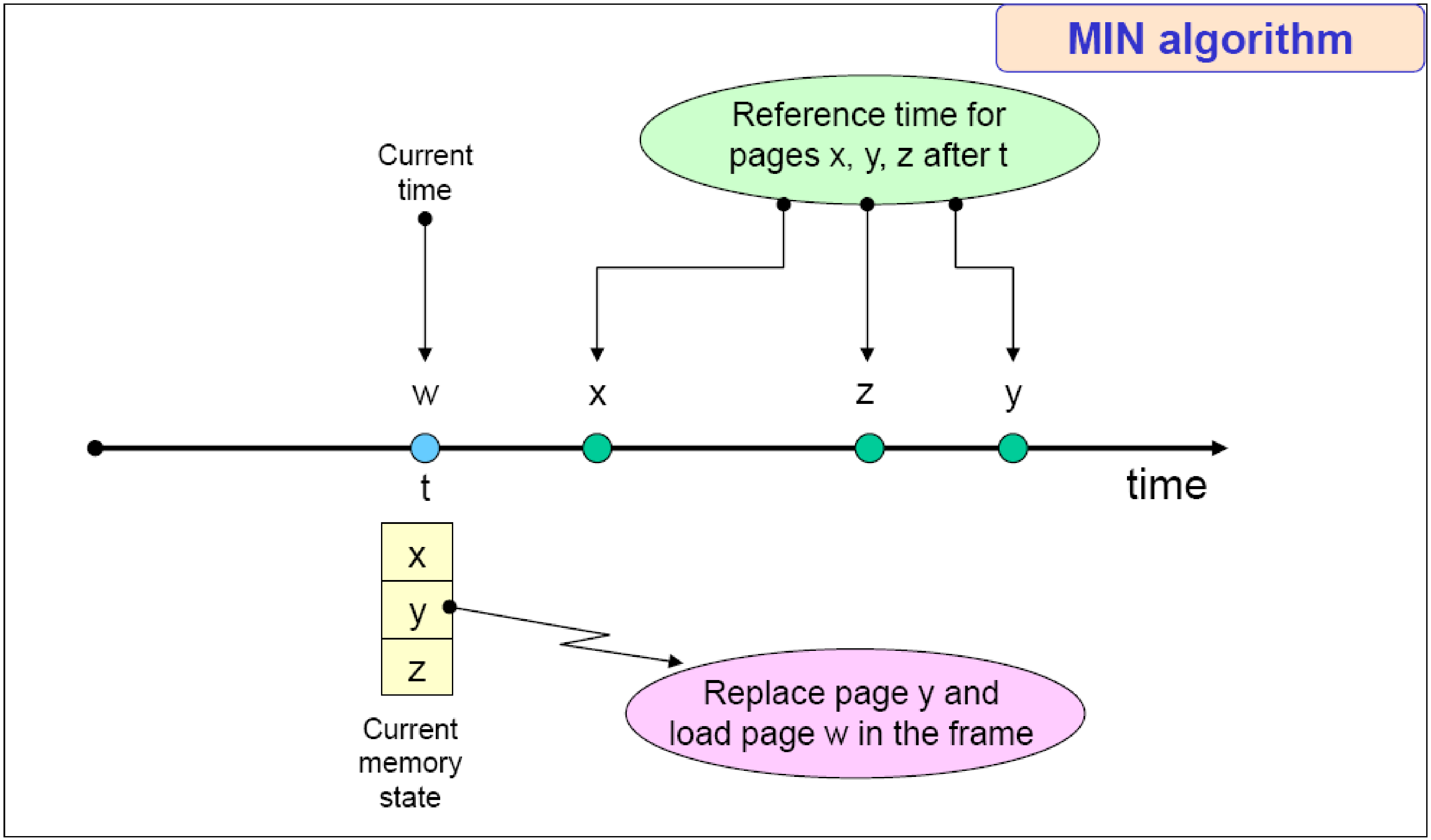
Time	1	2	3	4	5	6	7	8	9	10	11	12
Ref. string	1	2	3	4	1	2	5	1	2	3	4	5
Memory state	1	1	1	1	1	1	5	5	5	5	4	4
		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	4	3	3	3
Page fault	F	F	F	F			F	F	F	F	F	F

- Number of page faults: 9
- Number of page faults: 10

MIN algorithm (OPT algorithm)

- Proposed by Belady in 1966
- Minimizes page fault frequency (proved)
- Scheme
  - Replace the page that will not be used for the longest period of time
  - Tie-breaking rule
    - Page with greatest (or smallest) page number
- Unrealizable
  - Can be used only when the process's reference string is known a priori
- Usage
  - Performance measurement tool for replacement schemes

MIN algorithm (OPT algorithm)



MIN algorithm (OPT algorithm)

- **MIN algorithm: Example**
  - 4 page frames allocated, initially empty

ω = 1 2 6 1 4 5 1 2 1 4 5 6 4 5

Memory state change (MIN)

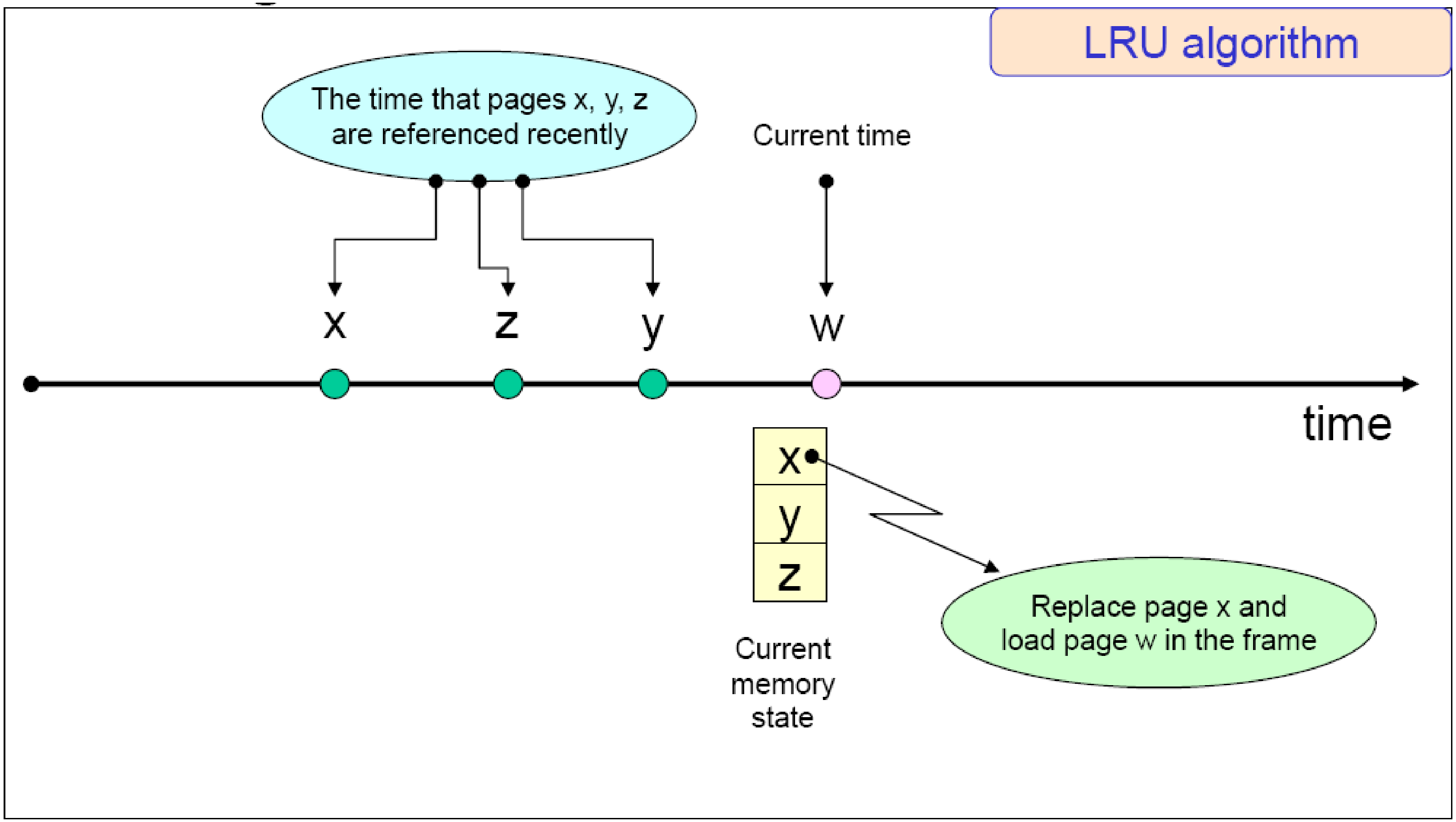
Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Ref. string	1	2	6	1	4	5	1	2	1	4	5	6	4	5
Memory state	1	1	1	1	1	1	1	1	1	1	1	6	6	6
		2	2	2	2	2	2	2	2	2	2	2	2	2
			6	6	6	5	5	5	5	5	5	5	5	5
					4	4	4	4	4	4	4	4	4	4
Page fault	F	F	F		F	F						F		

- Number of page faults: 6

Least Recently Used (LRU) Algorithm

- Choose the page to be replaced based on the reference time
- Scheme
  - Replace the page that has not been used for the longest period of time
- Requirements
  - Timestamping (page reference time) is necessary
- Characteristics
  - Based on program locality
  - Approximates to the performance of MIN algorithm
- **Used in most practical systems**
- Drawbacks
  - Timestamping overhead at every page reference
  - Number of page faults increases steeply when the process executes large loop with insufficiently allocated memory

Least Recently Used (LRU) Algorithm



Least Recently Used (LRU) Algorithm

- **LRU algorithm: Example**
  - 4 page frames allocated, initially empty

ω = 1 2 6 1 4 5 1 2 1 4 5 6 4 5

Memory state change (LRU)

Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Ref. string	1	2	6	1	4	5	1	2	1	4	5	6	4	5
Memory state	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		2	2	2	2	5	5	5	5	5	5	5	5	5
			6	6	6	6	6	2	2	2	2	6	6	6
					4	4	4	4	4	4	4	4	4	4
Page fault	F	F	F		F	F		F				F		

- Number of page faults: 7

Implementation of LRU algorithm

- By counter
  - Use PMT with count field
  - Increment processor clock or counter for each memory access
  - Record the value of processor clock or counter in the corresponding PMT entry for each page reference
  - Can get the relative order of recent access to each page
  - PMT search for selecting a page to be replaced

Implementation of LRU algorithm

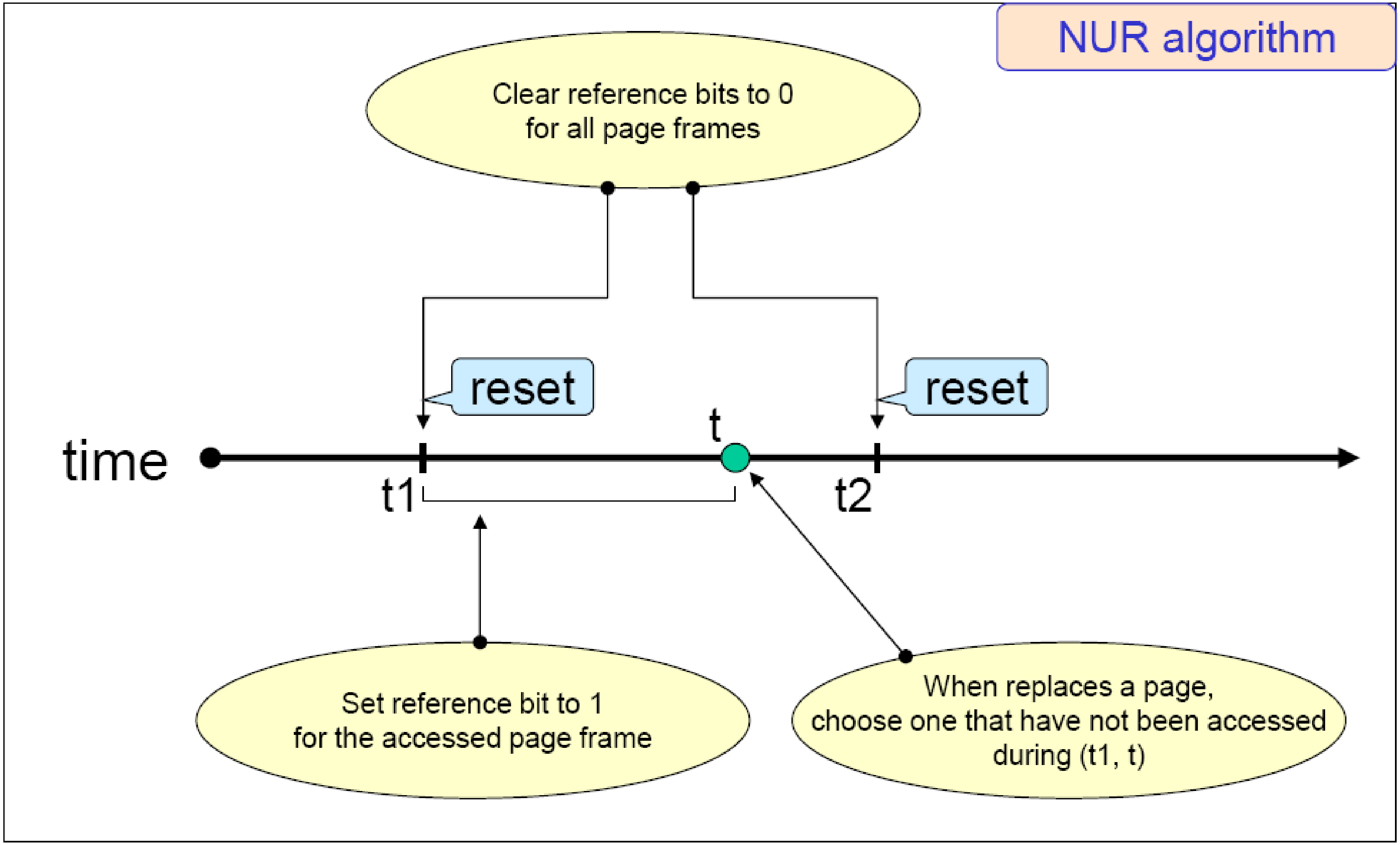
- By stack
  - Stack
    - Stack for each process, whose entry is page number
    - Maintains the stack elements (page numbers) in the order of recent access
    - Can delete an element in the middle of the stack
  - When no page fault
    - Deletes the referenced page number from the stack, and inserts it on top of the stack
  - When page fault
    - Displaces the page whose number is at the bottom of the stack, deletes it from the stack, and inserts incoming page number on top of the stack

LRU Approximation Algorithms

- **LRU needs special hardware and still slow**
- **NUR(Not Used Recently) algorithm**
  - LRU approximation scheme
  - Lower overhead than LRU algorithm
  - Uses bit vectors
    - Reference bit vector
    - Update bit vector
  - Scheme
    - Check reference/update bit and choose a victim page
    - Order of replacement (reference bit: r, update bit: m)
      - ① Replace the page with (r, m) = (0, 0)
      - ② Replace the page with (r, m) = (0, 1)
      - ③ Replace the page with (r, m) = (1, 0)
      - ④ Replace the page with (r, m) = (1, 1)

LRU Approximation Algorithms

- **NUR algorithm**



LRU Approximation Algorithms

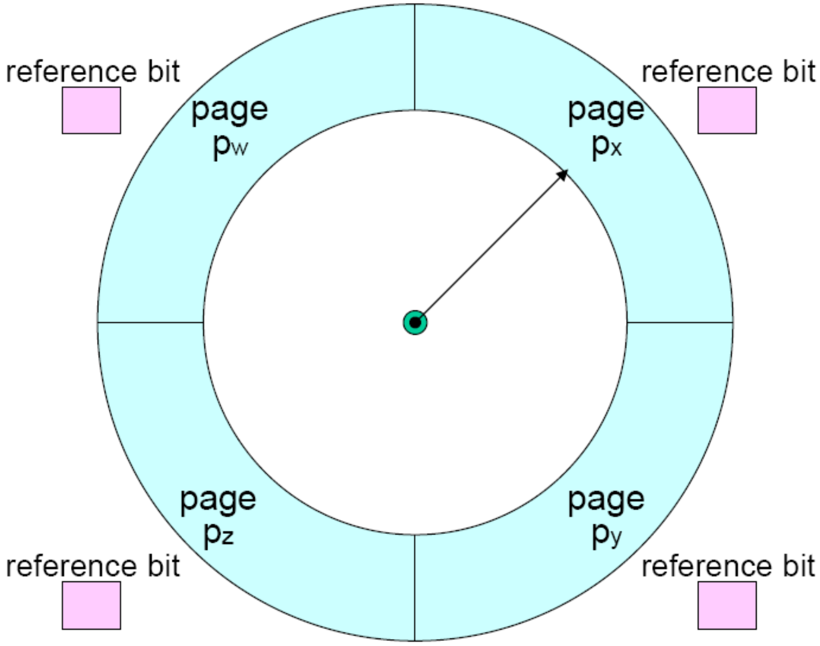
- **Additional reference-bits algorithm**
  - LRU approximation
  - History register for each page
  - Recording the reference bits at regular intervals
    - Timer interrupts at regular intervals
    - OS shifts the reference bit for each page into the high-order bit of its history register, shifting the other bits right by one bit and discarding the low-order bit
  - Replacement based on the value of history register
    - Interpret the value of the history register as unsigned integers
    - Choose the page with smallest value as a victim

LRU Approximation Algorithms

- **Additional reference-bits algorithm**
- Example
  - Value of the 8-bit history registers
    - 00000000
      - No reference during 8 time periods
    - 11111111
      - Referenced at least once in each period
    - Page **a** with 01100100 and page **b** with 00110111
      - Page **a** has been used more recently that page **b**

LRU Approximation Algorithms

- **Clock algorithm (Second-chance algorithm)**
  - Used for IBM VM/370 OS
  - Uses reference bit
    - No periodical reset for reference bits
  - Choose the page to be replaced using pointer that circulates the list of pages (page frames) in memory



LRU Approximation Algorithms

- **Clock algorithm**
  - Scheme
    - Choose the page to be replaced with the pointer moving clockwise
    - Mechanism

      ① Check the reference bit *r* of the page that the pointer points

      ② If *r* == 0, select the page as a victim

      ③ If *r* == 1, reset the reference bit to 0 and advances the pointer goto step-①
  - Characteristics
    - Earlier memory load time → higher probability of displacement
      - Similar to FIFO algorithm
    - Page replacement based on the reference bit
      - Similar to LRU (or NUR) algorithm

LRU Approximation Algorithms

- **Clock algorithm: Example**

- Assumptions
  - 4 page frames are allocated to the process
  - Initially, it has pages a, b, c, d in memory
  - Reference bits of the 4 page frames are all 1

ω = c a d b e b a b c d

Memory state change (Clock)

Time		0	1	2	3	4	5	6	7	8	9	10
Ref. string			c	a	d	b	e	b	a	b	c	d
Memory state	frame 0		→a/1	→a/1	→a/1	→a/1	e/1	e/1	e/1	e/1	→e/1	d/1
	frame 1		b/1	b/1	b/1	b/1	→b/0	→b/1	b/0	b/1	b/1	→b/0
	frame 2		c/1	c/1	c/1	c/1	c/0	c/0	a/1	a/1	a/1	a/0
	frame 3		d/1	d/1	d/1	d/1	d/0	d/0	→d/0	→d/0	c/1	c/0
Page fault							F		F		F	F
Pclock							e		a		c	d
Qclock							a		c		d	e

- Pclock : Pages loaded into memory
- Qclock : Pages displaced from memory

LRU Approximation Algorithms

- **Enhanced clock algorithm**
  - Similar to clock algorithm
  - Considers update (dirty, modified) bit as well as reference bit
- Scheme
  - Choose the page to be replaced with the pointer moving clockwise
  - Mechanism

    ① Check (r, m) of the page that the pointer points

    ② (0, 0): select the page as a victim, advances the pointer

    ③ (0, 1): set (r, m) to (0, 0), put the page on the cleaning list, goto step-⑥

    ④ (1, 0): set (r, m) to (0, 0), goto step-⑥

    ⑤ (1, 1): set (r, m) to (0, 1), goto step-⑥

    ⑥ Advances the pointer, goto step-①

LRU Approximation Algorithms

- **Enhanced clock algorithm: Example**

- Assumptions
  - 4 page frames are allocated to the process
  - Initially, it has pages a, b, c, d in memory
  - All reference bits are 1, all update bits are 0

$\omega = c\ a^W\ d\ b^W\ e\ b\ a^W\ b\ c\ d$

Memory state change (Enhanced clock)

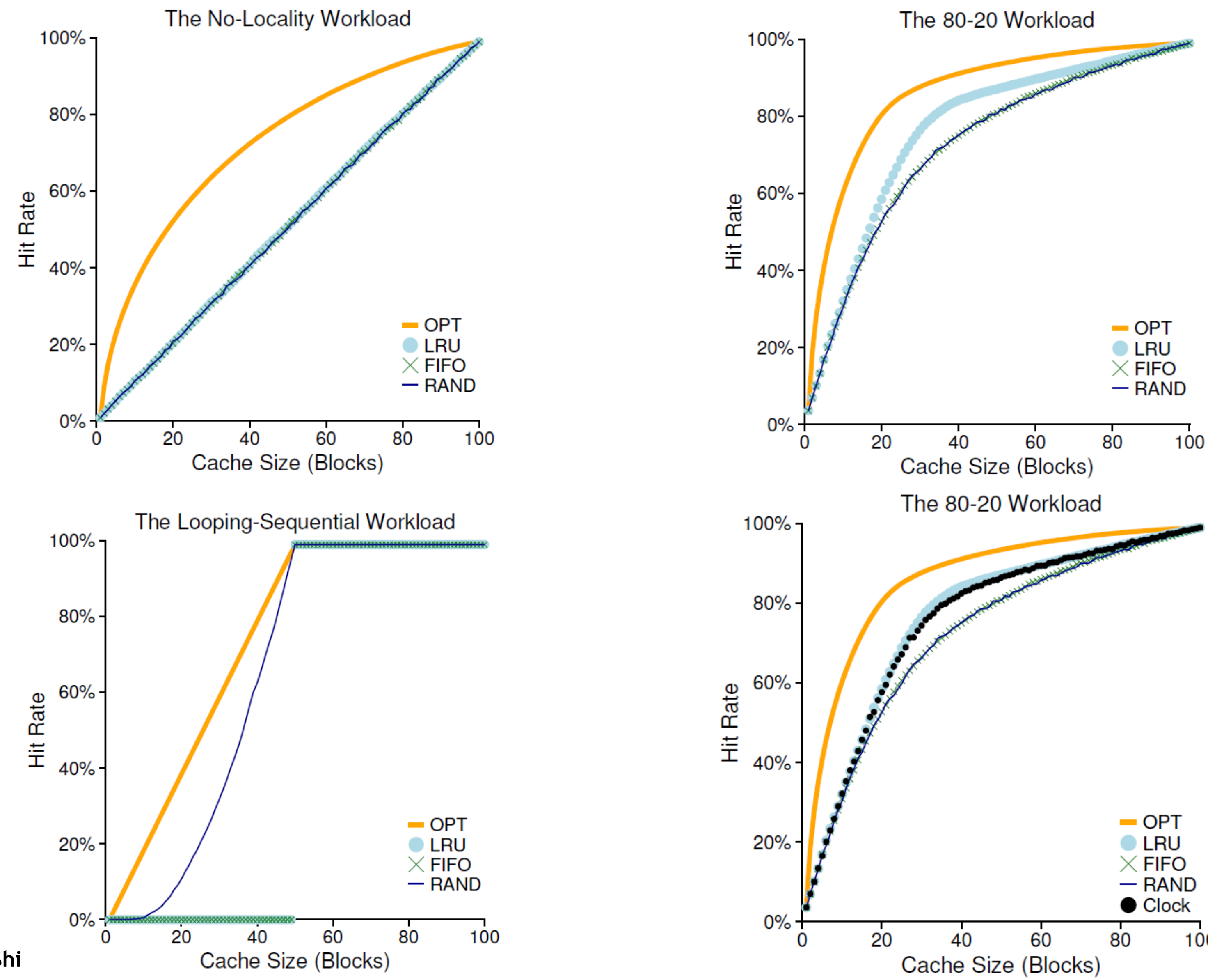
Time		0	1	2	3	4	5	6	7	8	9	10
Ref. string			c	a^W	d	b^W	e	b	a^W	b	c	d
Memory state	frame 0	→a/10	→a/10	→a/11	→a/11	→a/11	a/00	a/00	a/11	a/11	→a/11	a/00
	frame 1	b/10	b/10	b/10	b/10	b/11	b/00	b/10	b/10	b/10	b/10	d/10
	frame 2	c/10	c/10	c/10	c/10	c/10	e/10	e/10	e/10	e/10	e/10	→e/00
	frame 3	d/10	d/10	d/10	d/10	d/10	→d/00	→d/00	→ d/00	→ d/00	c/10	c/00
Page fault P2nd-chance Q2nd-chance							F e c				F c d	F d b

- Superscript W : page update
- Underline : the page is on the cleaning list
- P2nd-chance : pages loaded into memory
- Q2nd-chance : pages displaced from memory

Counting Algorithms

- **Keep a counter of the number of references that have been made to each page**
  - Not common
- **Lease Frequently Used (LFU) Algorithm**
  - replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm**
  - based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Workload Examples



Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Bypasses buffering, locking, etc
- O_DRIECT mode

Load control strategies

- Control the multiprogramming degree of a system
- Related to allocation strategy
- Should maintain multiprogramming degree at the appropriate range

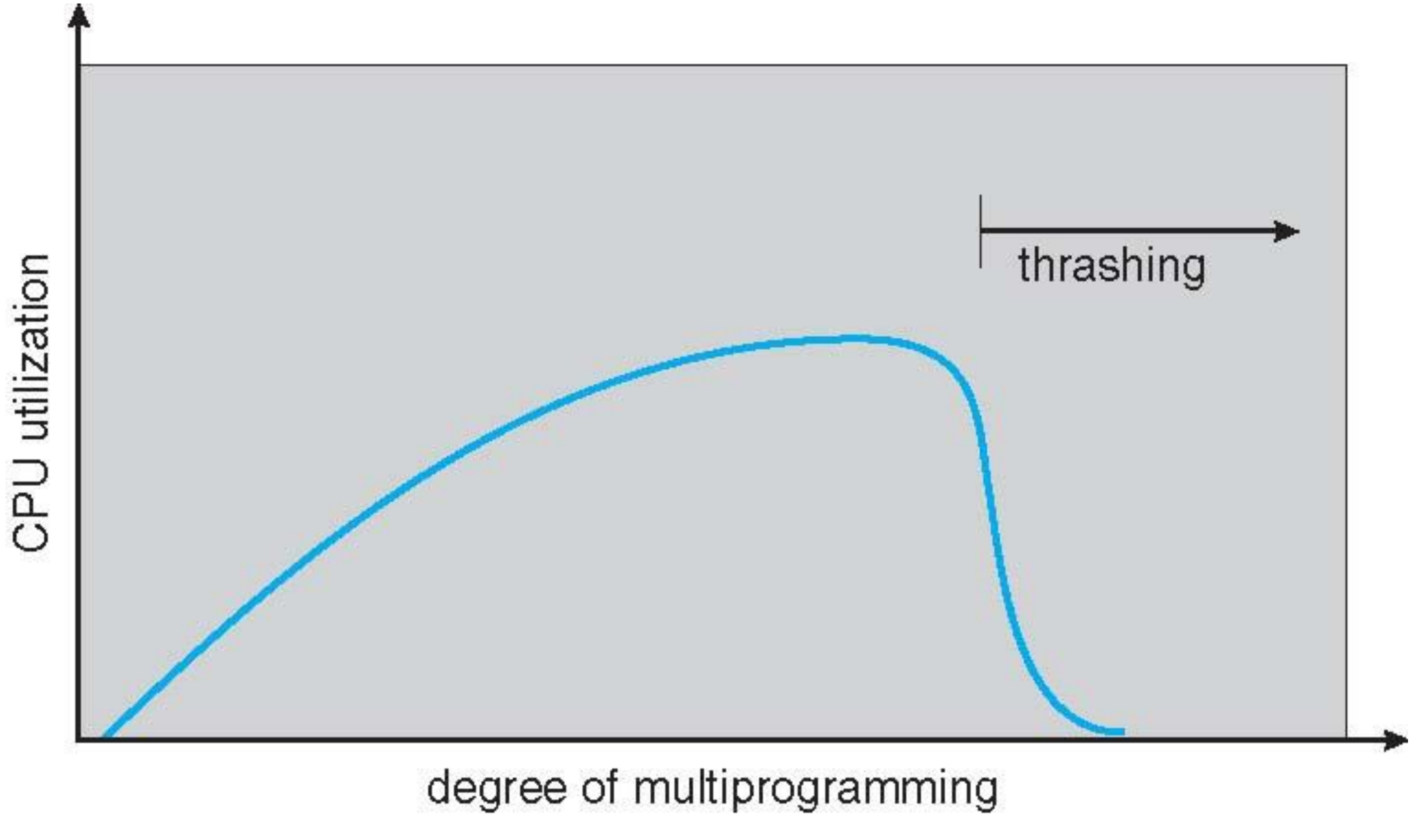
- **Underloaded**
  - System resource waste, performance degradation
- **Overloaded**
  - System resource contention, performance degradation
  - Thrashing (excessive paging activity)
- **Plateau** ranges are located at different positions for different systems

Thrashing

- **If a process does not have “enough” pages, the page-fault rate is very high**
  - Page fault to get page ➔ Replace existing frame ➔ But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- **Thrashing**
  - a process is busy swapping pages in and out

Thrashing



Demand Paging and Thrashing

- **Why does demand paging work?**
  - **Locality model**
    - **Spatial locality and Temporal locality**
  - Process migrates from one locality to another
  - Localities may overlap

- **Why does thrashing occur?**
  - Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

Be Lazy: Demand Zeroing

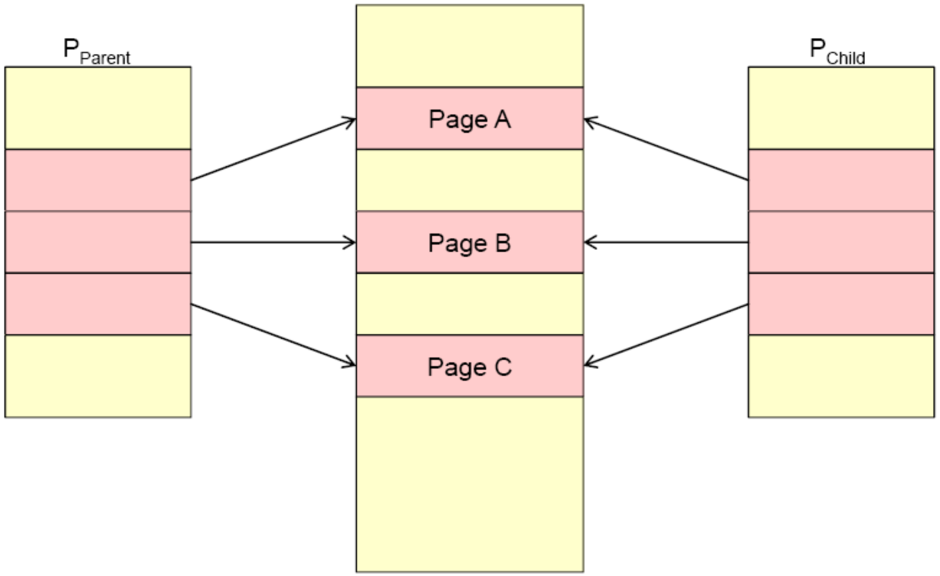
- To add a page to your address space (heap)
  - OS adds a page to your heap by finding a page in physical memory,
  - zeroing it (required for security)
  - and then mapping it into your address space (i.e., setting up the page table).
  - High cost particularly if the page does not get used
- Demand zeroing
  - OS puts an entry in the page table that marks the page inaccessible
  - If the process then reads or writes the page, a trap takes place.
  - When handling the trap, the OS notices that this is actually a demand-zero page.
  - at this point, the OS then does the needed work of finding a physical page, zeroing it, and mapping it into the process's address space.
  - If the process never accesses the page, all of this work is avoided

Be Lazy: Copy-on-Write

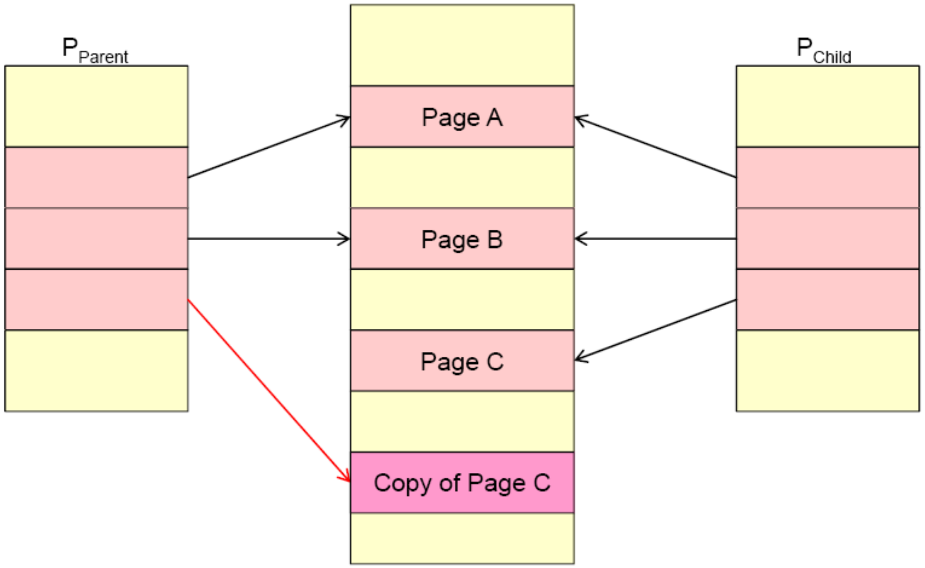
- **Copy-on-Write**
  - When OS needs to copy a page from one address space to another, instead of copying it, it can map it into the target address space and mark it read-only in both address spaces. ➔ fast copy
  - For read operation, no further action is taken
  - For write operation, a trap occurs
    - OS (lazily) allocate a new page, fill it with the data, and map this new page into the address space of the faulting process.
- **fork**() system call
  - Creates a child process as a duplicate of its parent
  - Should create a copy of the parent's address space for the child, duplicating the pages of the parent
- With **copy-on-write** scheme
  - Allows the parent and the child processes initially share the pages (marked as copy-on-write)
  - When any process writes to a shared page, a copy of the shared page is created

Be Lazy: Copy-on-Write

After **fork**() system call



After Parent modifies Page-C



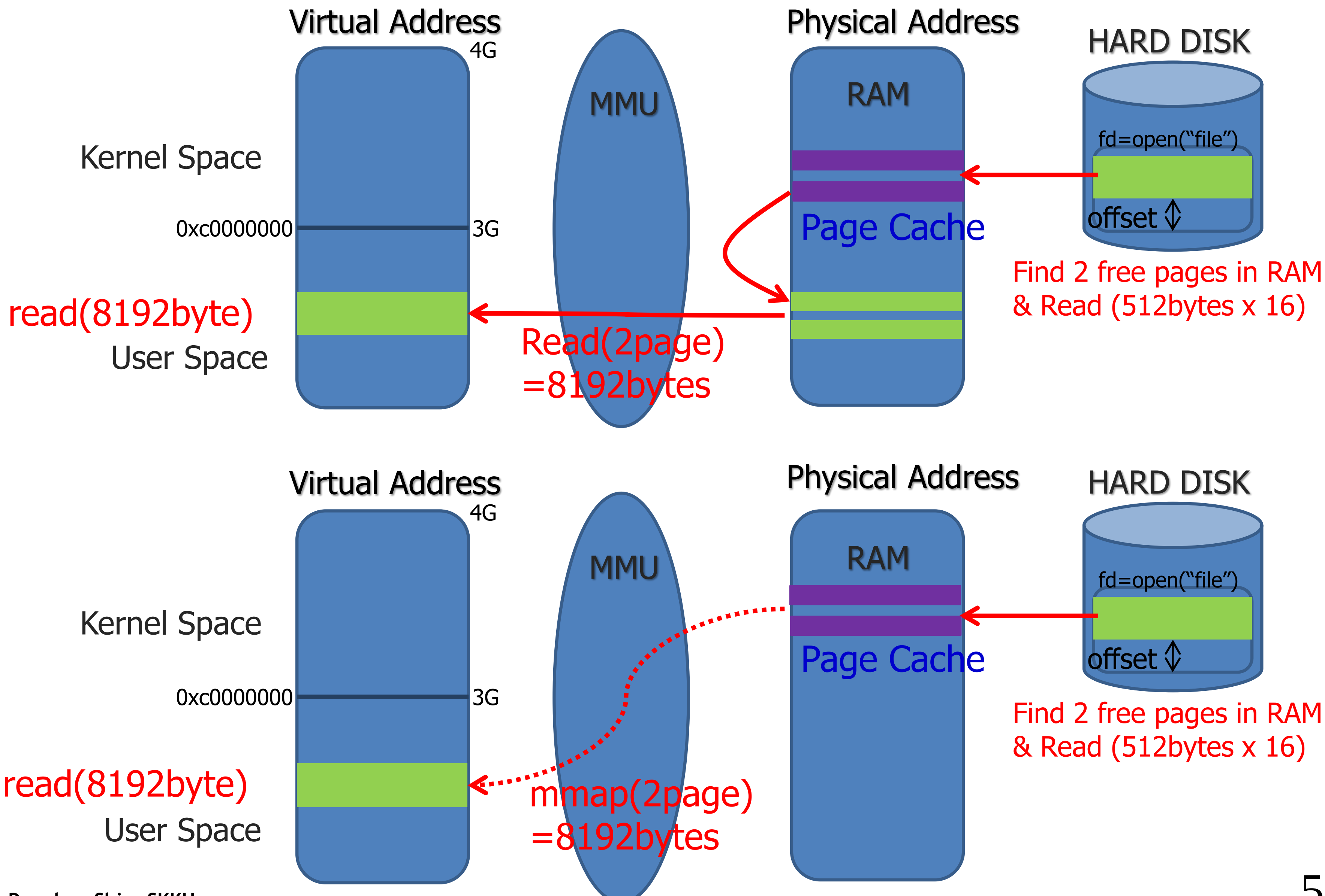
Be Lazy: Copy-on-Write

- **Notes on copy-on-write**
  - Only the copy-on-write pages that are modified by either process are copied
    - All unmodified pages are shared by the parent and child processes
  - Only the pages that can be modified can be marked as copy-on-write
  - Operating systems that use copy-on-write scheme
    - Linux, Solaris, Windows XP

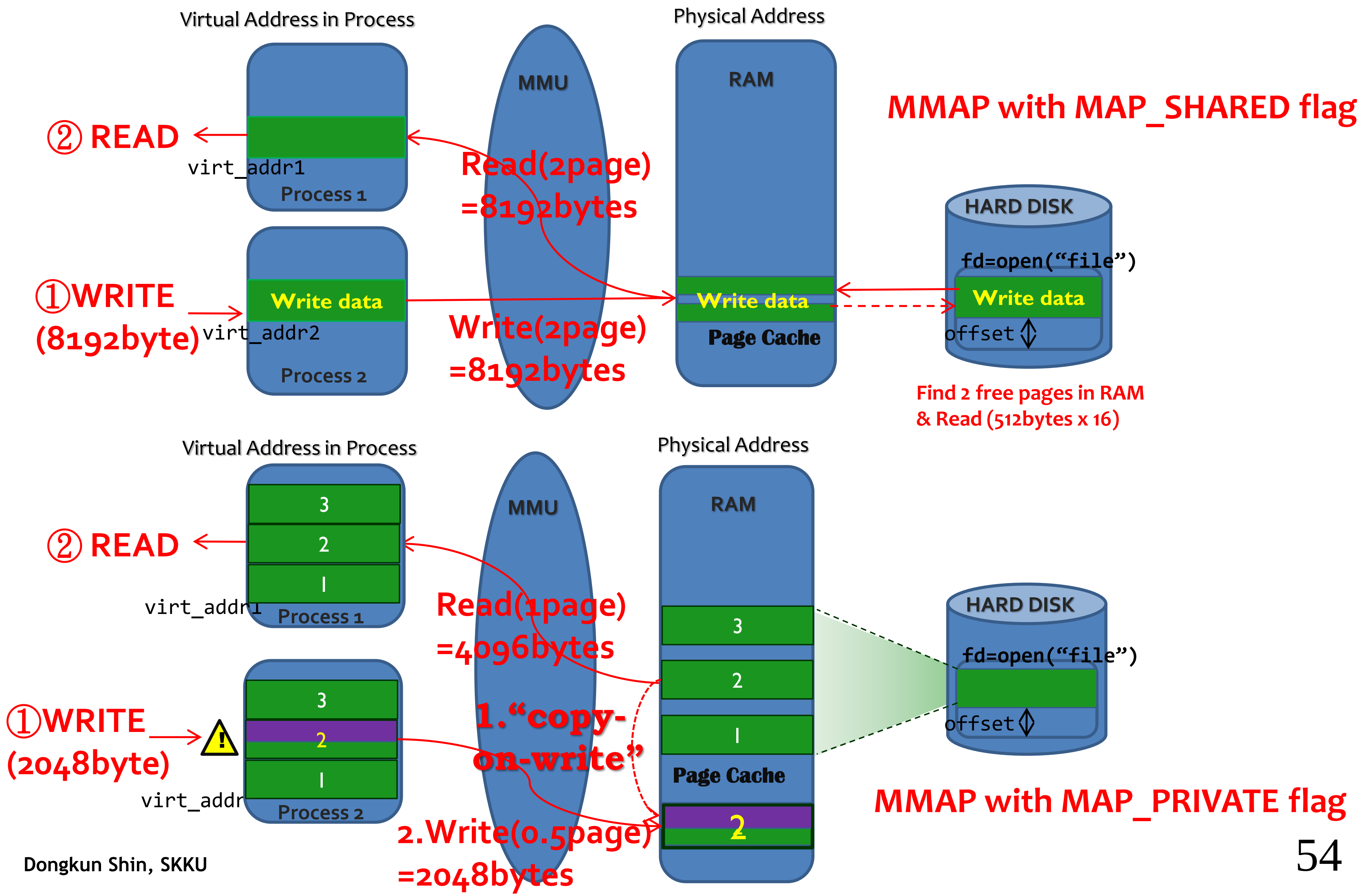
Memory-Mapped Files

- **Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory**
- **A file is initially read using demand paging**
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- **Simplifies and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls**
- **Also allows several processes to map the same file allowing the pages in memory to be shared**
- **But when does written data make it to disk?**
  - Periodically and / or at file `close()` time
  - For example, when the pager scans for dirty pages

MMAP


Virtual Address
4G
Kernel Space
0xc0000000
3G
read(8192byte)
User Space
MMU
Read(2page)
=8192bytes
Physical Address
RAM
Page Cache
HARD DISK
fd=open("file")
offset
Find 2 free pages in RAM
& Read (512bytes x 16)
Virtual Address
4G
Kernel Space
0xc0000000
3G
read(8192byte)
User Space
MMU
mmap(2page)
=8192bytes
Physical Address
RAM
Page Cache
HARD DISK
fd=open("file")
offset
Find 2 free pages in RAM
& Read (512bytes x 16)

MMAP on Shared File



Other Considerations -- Prepaging

- **Prepaging (Prefetch)**
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume *s* pages are prepaged and *α* of the pages is used
    - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging
      $s * (1 - \alpha)$ unnecessary pages?
    - α near zero $\Rightarrow$ prepaging loses

Other Issues – Page Size

- **Sometimes OS designers have a choice**
  - Especially if running on custom-built CPU
- **Page size selection must take into consideration:**
  - Fragmentation
  - Page table size
  - Resolution
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness

Smaller page size
+ Smaller internal fragmentation
+ Match program locality more accurately
  o Reduction in total I/O
  o Better utilization of memory
    (Less total allocation of memory)
- Larger page table size
- Increase in I/O time
- Increase in the number of page faults

- **Always power of 2, usually in the range 2^7 (128 bytes) to 2^{22} (4MB)**
- **On average, growing over time**

Other Issues – TLB Reach (Coverage)

- **TLB Reach**
  - The amount of memory accessible from the TLB
  - TLB Reach = (TLB Size) X (Page Size)
- **Ideally, the working set of each process is stored in the TLB**
  - Otherwise there is a high degree of page faults
- **Increase the number of entries in TLB**
  - Expensive
- **Increase the Page Size**
  - lead to an increase in fragmentation as not all applications require a large page size
- **Provide Multiple Page Sizes**
  - Allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation
  - Requires OS support
    - One of the fields in TLB entry must indicate the size of the page (frame) corresponding to the TLB entry

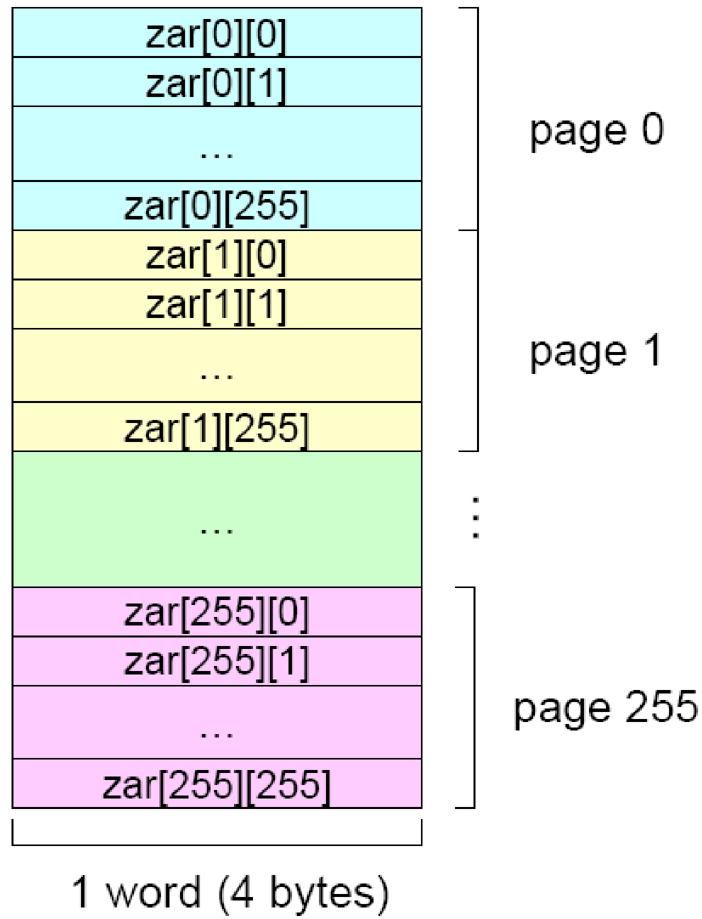
Other Issues – Program Structure

- **Program restructuring**
  - System performance can be improved if the user (or compiler/linker/loader) has an awareness of the paged nature of memory or underlying demand paging system
  - Program restructuring by user or compiler/linker/loader

Other Issues – Program Structure

- **Example**
  - Assumptions
    - Paging system of 1KB page size
    - sizeof(int): 4 Bytes

```
// Program-1
int main()
{
  int zar[256][256];
  int i, j;

  for(j = 0; j < 256; j++)
    for(i = 0; i < 256; i++)
      zar[i][j] = 0;
  return 0;
}
```



For each execution of loop instance, pages 0~255 are referenced sequentially and this process is repeated 255 times
For execution with no page faults, The process should have 256 page frames
- Not practical

Other Issues – Program Structure

```
// Program-1
int main()
{
  int zar[256][256];
  int i, j;

  for(j = 0; j < 256; j++)
    for(i = 0; i < 256; i++)
      zar[i][j] = 0;
  return 0;
}
```

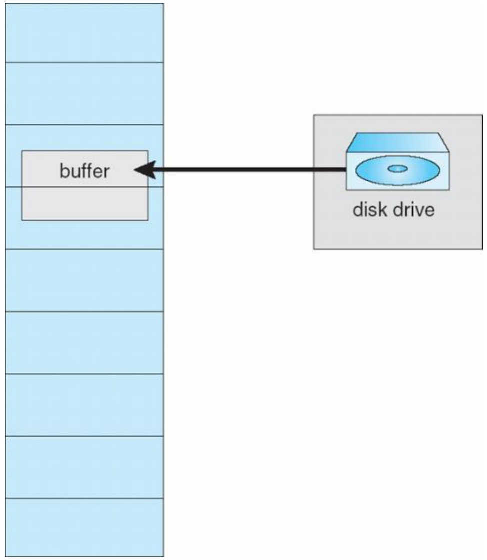
```
// Program-2
int main()
{
  int zar[256][256];
  int i, j;

  for(i = 0; i < 256; i++)
    for(j = 0; j < 256; j++)
      zar[i][j] = 0;
  return 0;
}
```

Other Issues – Program Structure

- Notes
  - User
    - Careful selection of data structures and programming structures can increase locality (hence lower page-fault rate)
      - Stack: has good locality
      - Hash table: produces bad locality
      - Pointer: diminishes locality
    - OOPs tend to have a poor locality of reference
  - Linker/Loader
    - Avoid placing routines across page boundaries
    - Let routines that call each other many times be packed into the same page

Other Issues – I/O interlock

- **I/O Interlock – Pages must sometimes be locked into memory**
  - Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm (lock bit)
  - Another solution: never to execute I/O to user memory
- **Pinning of pages to lock into memory**



Homework

- Homework in Chap 21 (**vmstat**)
- Homework in Chap 22 (paging-policy.py)