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

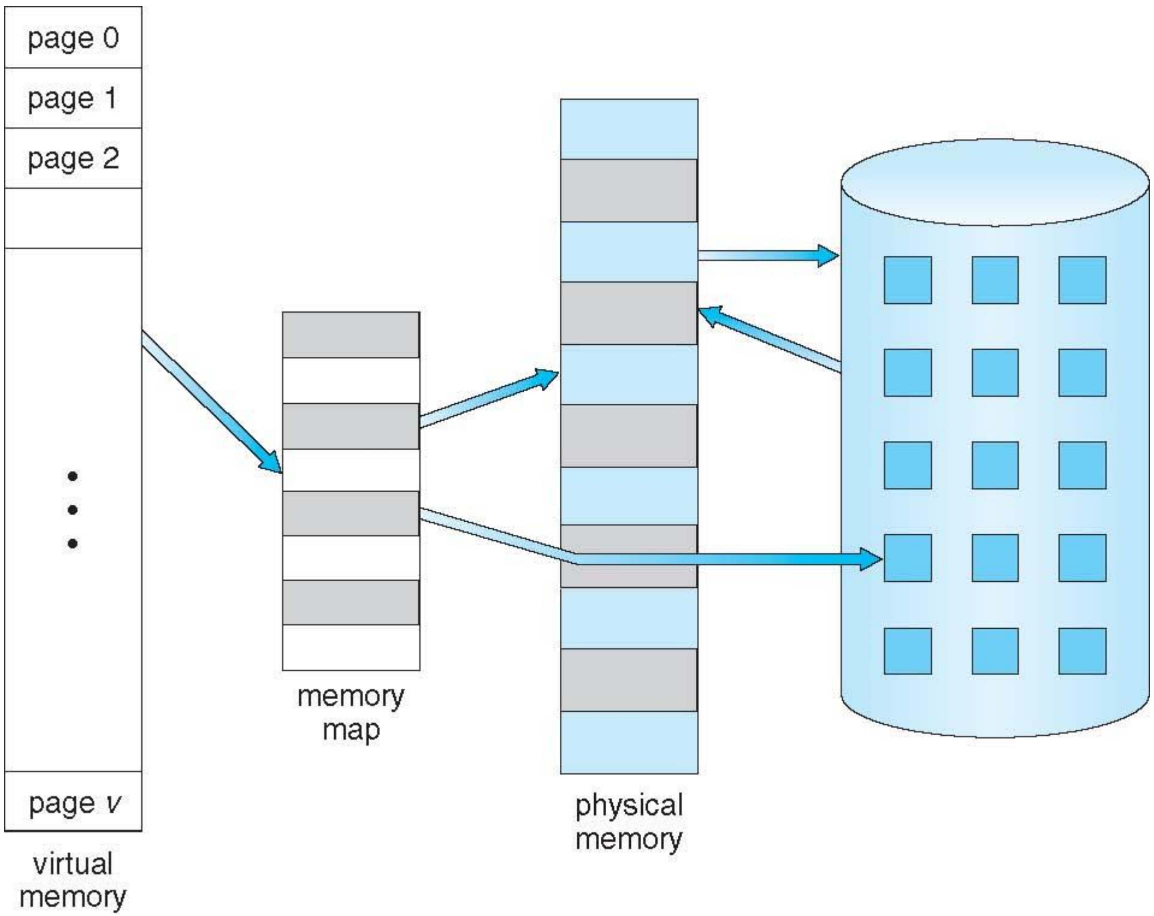
- **Virtual memory**
  - Benefits
    - Easier programming
      - Programmer no longer needs to worry about the amount of physical memory available
      - Allows address spaces to be shared by several processes
    - Higher multiprogramming degree
      - Increase in CPU utilization and throughput (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

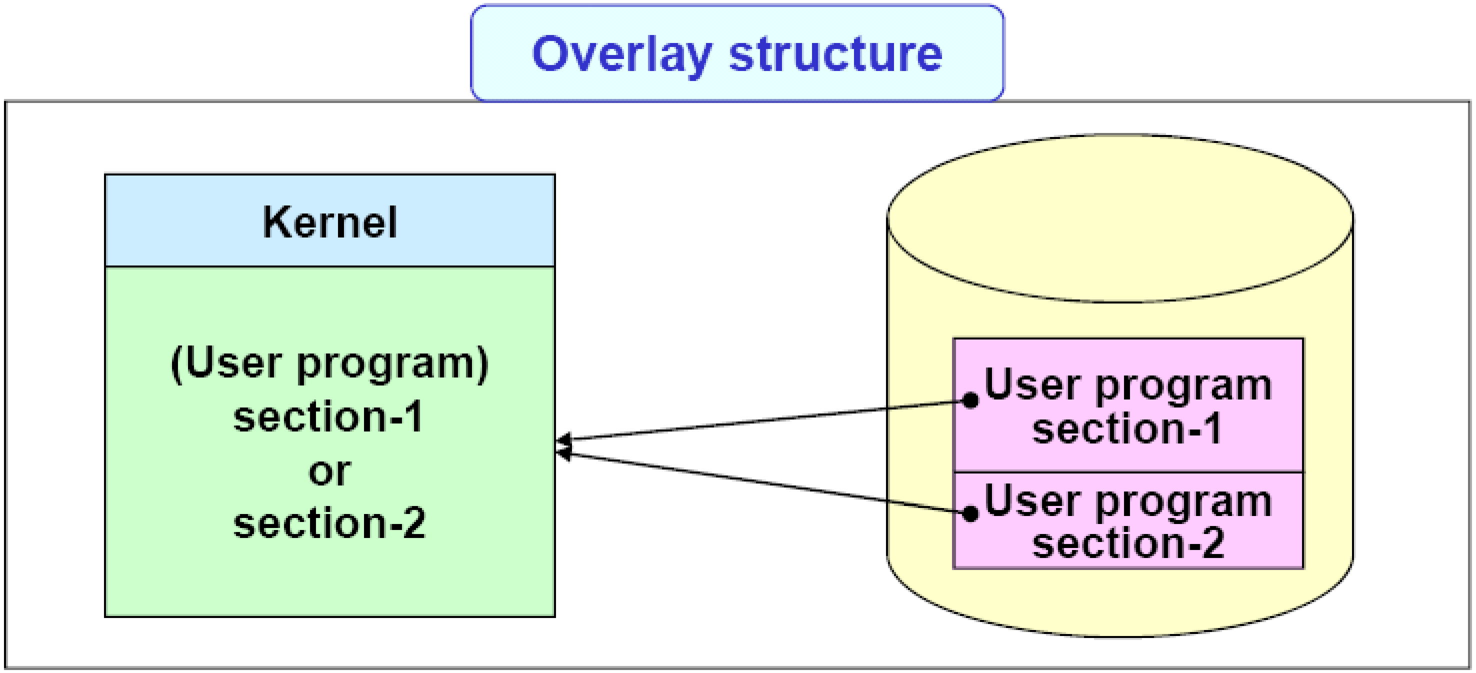
Virtual Memory That is Larger Than Physical Memory

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



Memory Overlay (old system)

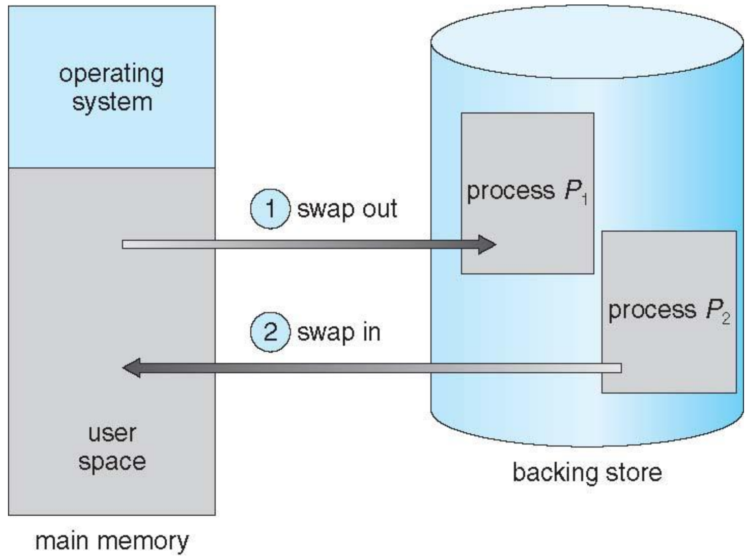
- **Program-size > memory-size**
  - w/o OS support
  - Requires support from compiler/linker/loader



Swapping

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

Process-level swapping



Page-level swapping

Physical Memory	PFN 0	PFN 1	PFN 2	PFN 3
	Proc 0 [VPN 0]	Proc 1 [VPN 2]	Proc 1 [VPN 3]	Proc 2 [VPN 0]

Swap Space	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7
	Proc 0 [VPN 1]	Proc 0 [VPN 2]	[Free]	Proc 1 [VPN 0]	Proc 1 [VPN 1]	Proc 3 [VPN 0]	Proc 2 [VPN 1]	Proc 3 [VPN 1]

Swapping

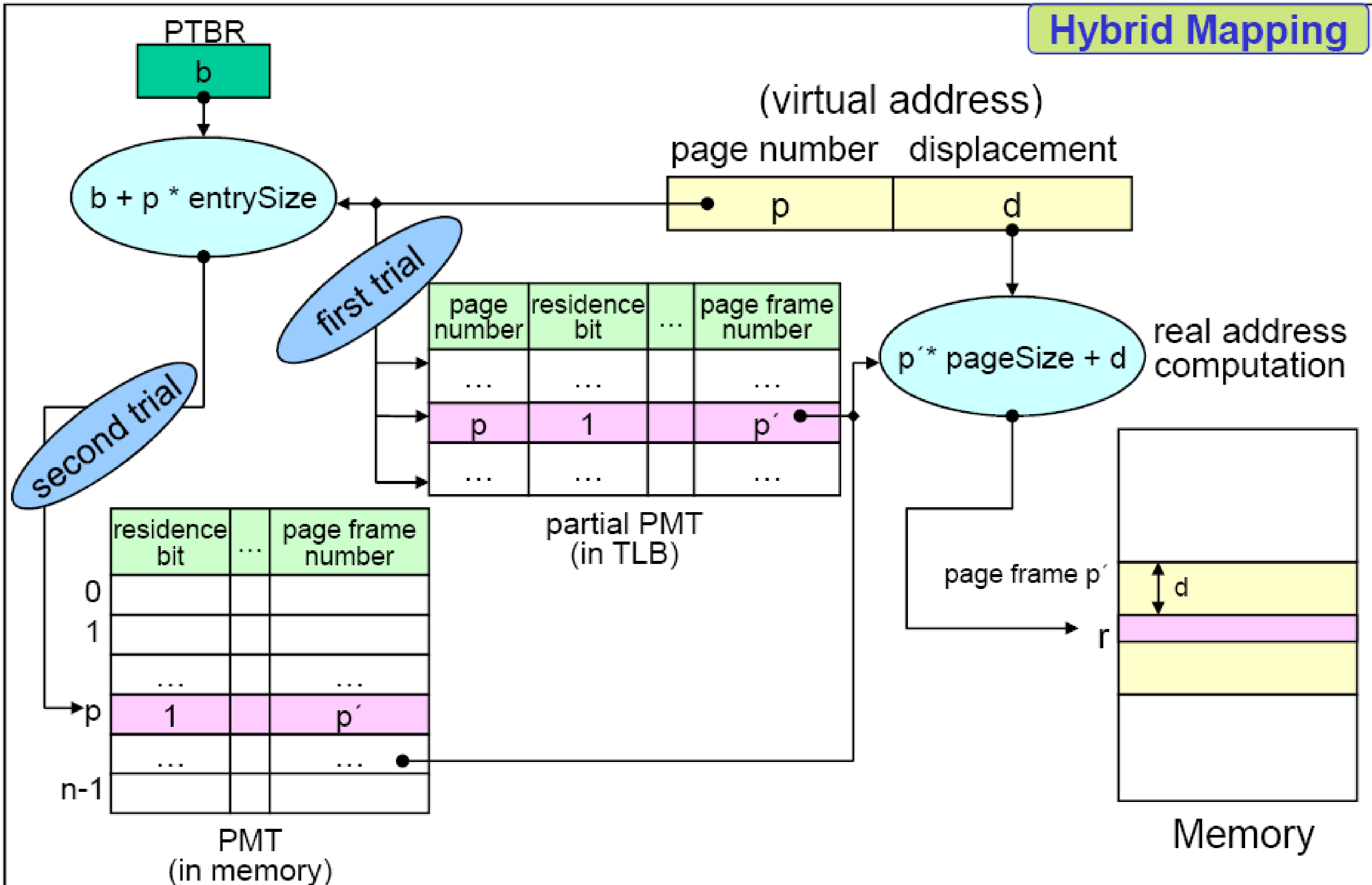
- **Notes on swapping**
  - Time quantum vs swap time
    - Time quantum should be substantially larger than swap time (context switch time) for efficient CPU utilization
  - Memory areas to be swapped out
    - Swap only what is actually used
  - Pending I/O
    - If the I/O is asynchronously accessing 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
- **Pages from program binary on disk can be simply discarded 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, Low Memory Killer
  - 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

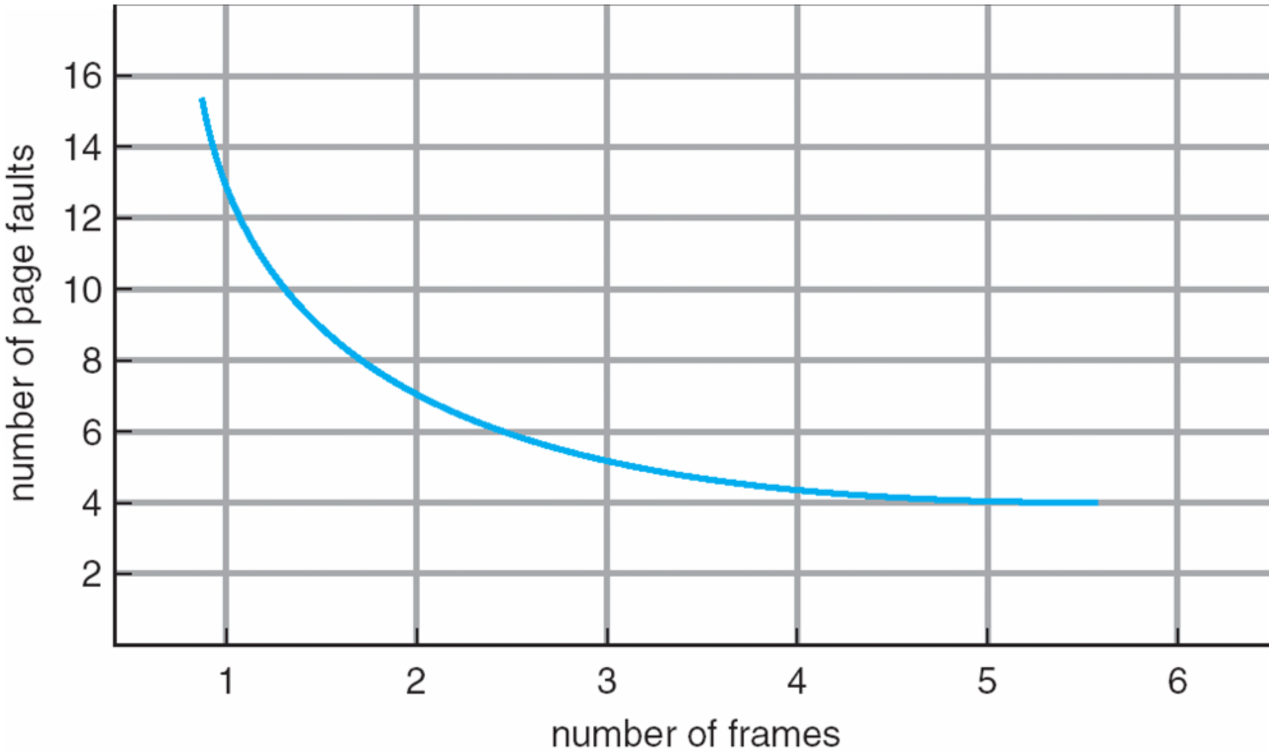
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame
     - Write victim frame to disk if dirty

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

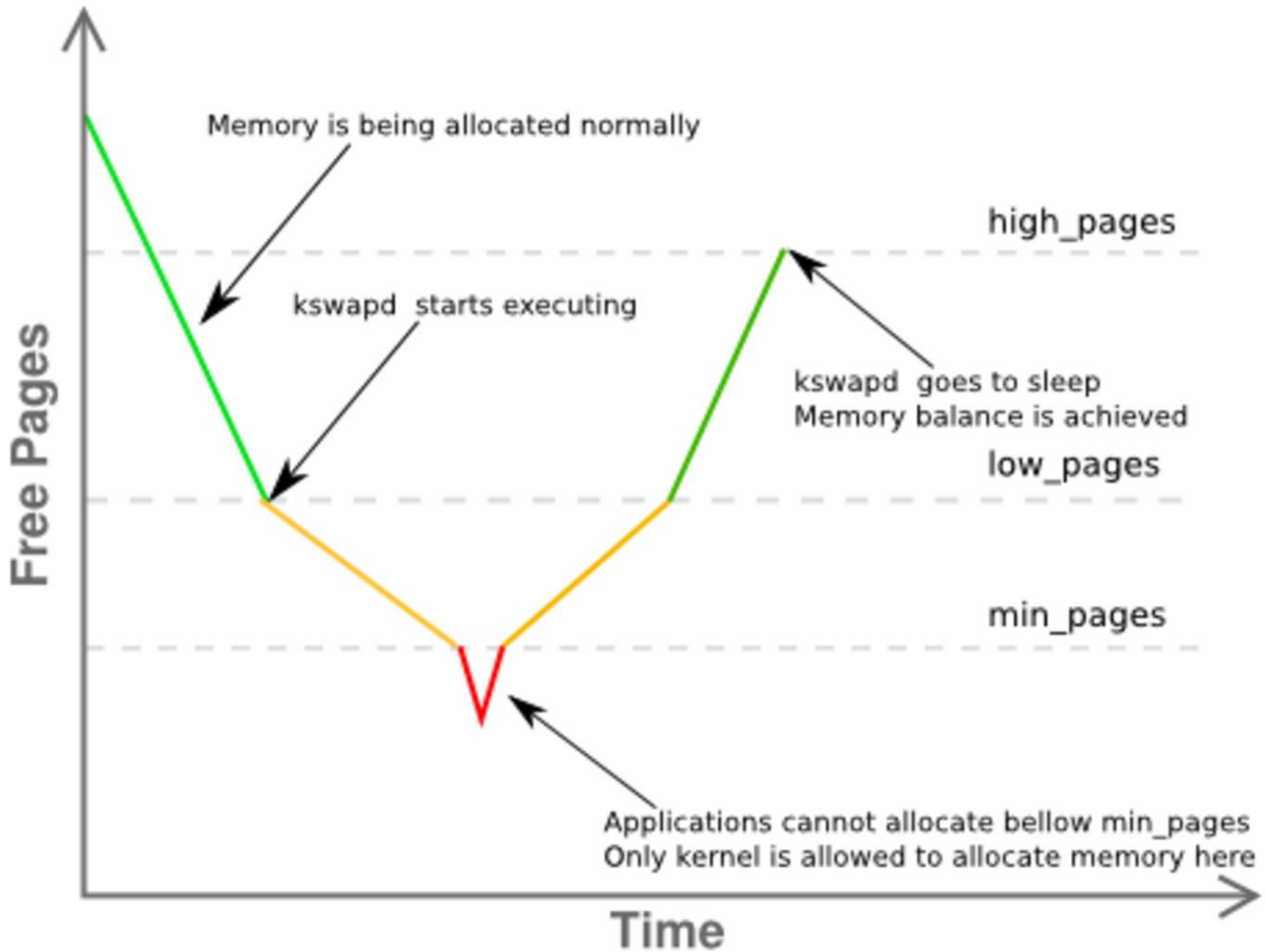
Graph of Page Faults Versus The Number of Frames



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

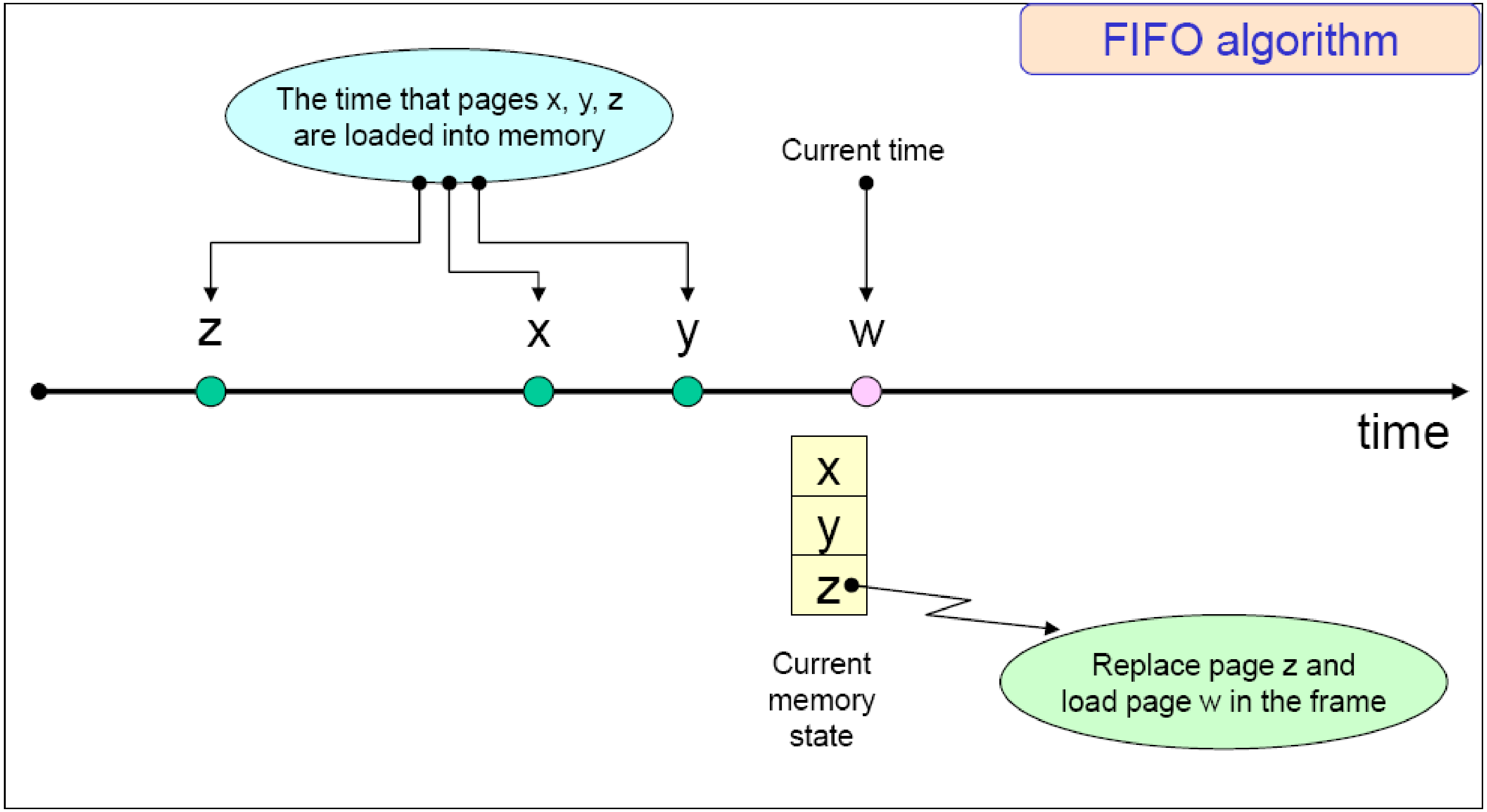
Memory Reclamation at Linux



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

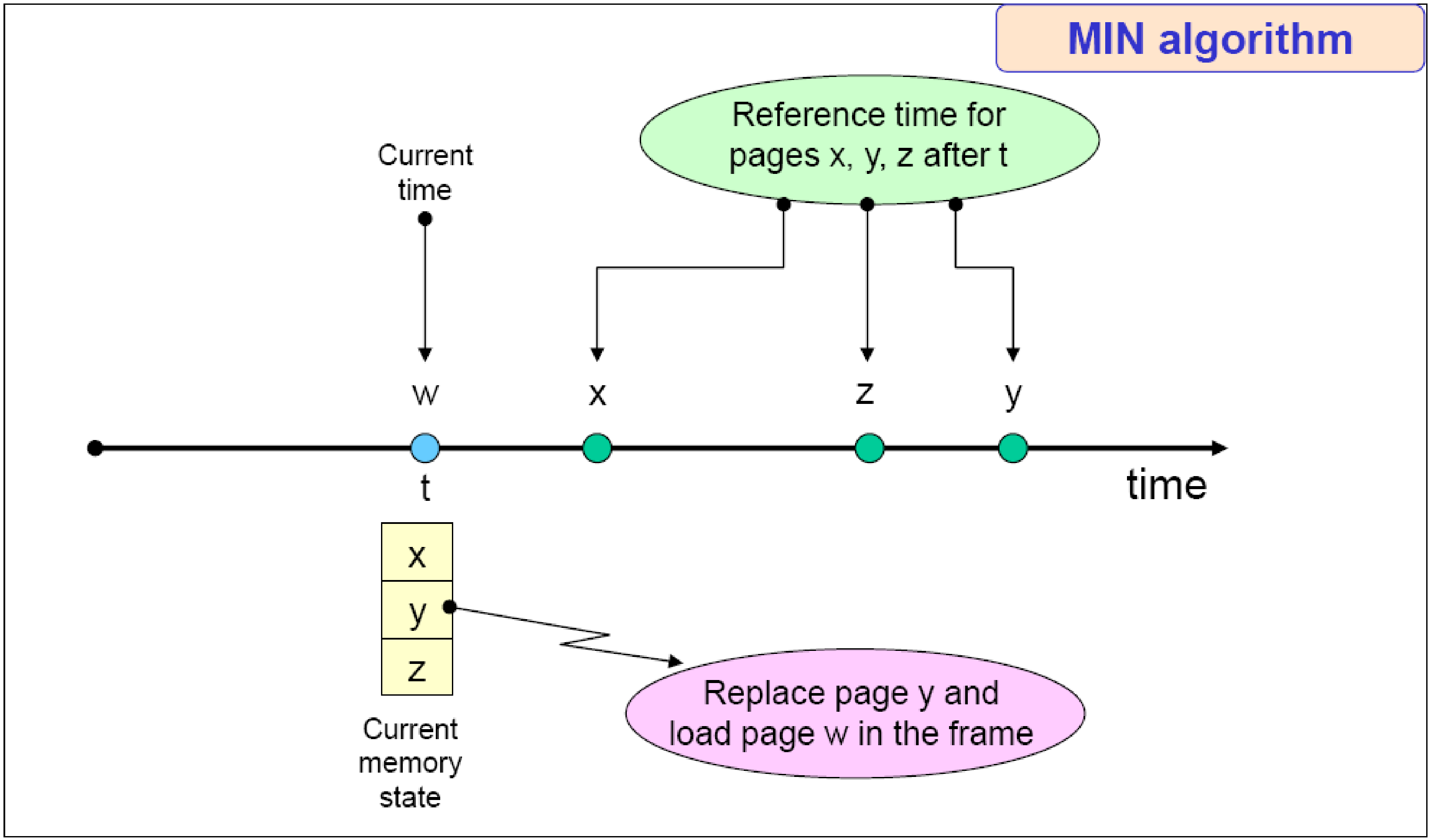
- Number of page faults: 9

- 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)

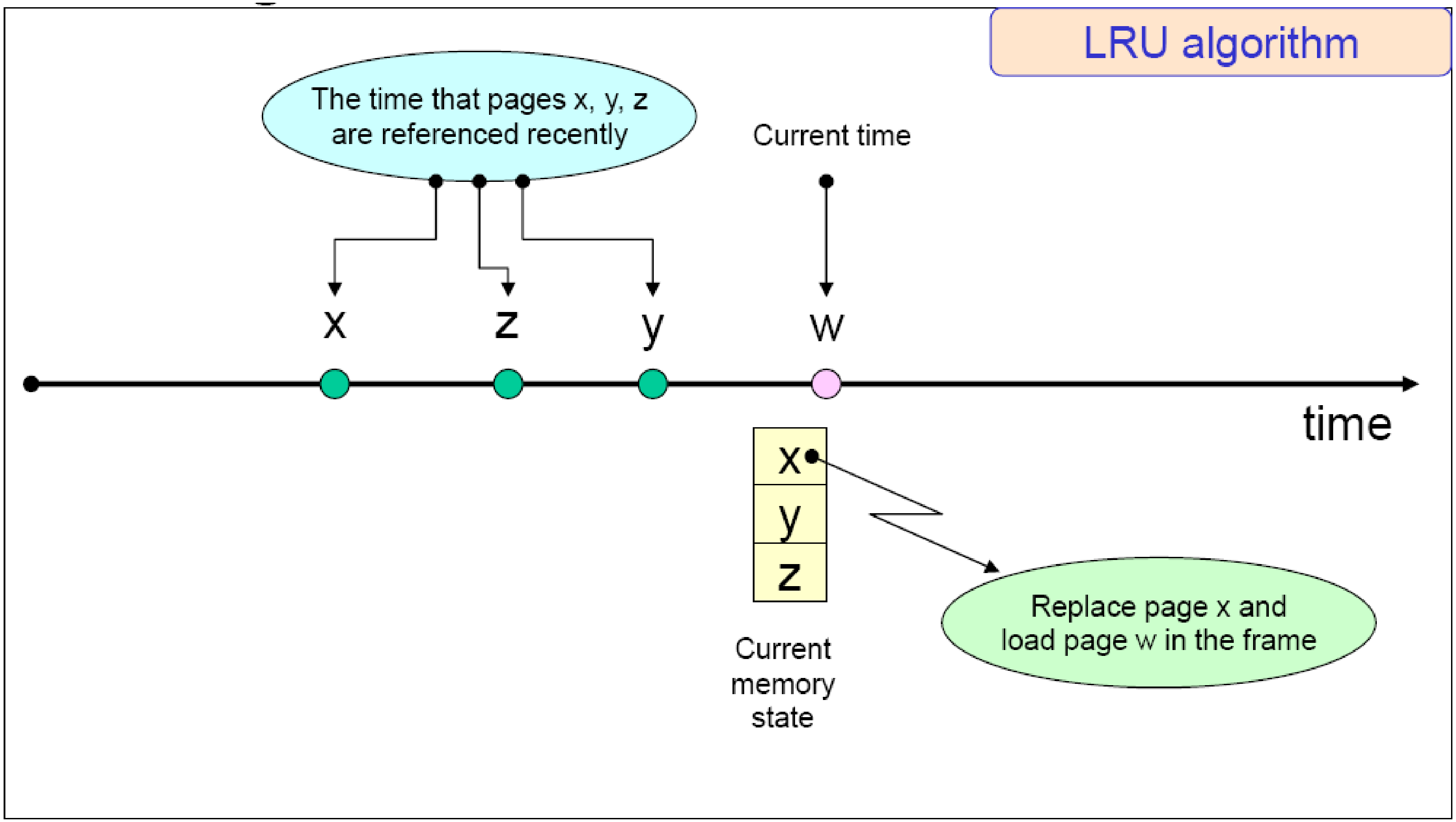
Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Ref. string	1	2	6	1	4	5	1	2	1	4	5	6	4	5
Memory state	1	1	1	1	1	1	1	1	1	1	1	6	6	6
		2	2	2	2	2	2	2	2	2	2	2	2	2
			6	6	6	5	5	5	5	5	5	5	5	5
					4	4	4	4	4	4	4	4	4	4
Page fault	F	F	F		F	F						F		

- 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