CS2106 Live Class: Disjoint memory schemes

$_/|/|U_Ch@NgRu!$

May 10, 2021

1 Summary of virtual memory

1. Completely separate logical memory address from physical memory
   Amount of physical memory no longer restrict the size of logical memory address
2. More efficient use of physical memory(page currently not in use can be on secondary storage)
3. Allow more processes to reside in memory(Important: not every memory has having their whole part in the memory), which improve the utilization of memory
4. Less I/O for load or swap entire program into memory, so that the program could be possibly faster

1.1 Implementation Issues

1. Page table structure: How to structure the page table for efficiency
2. Page Replacement Algorithm: Each process has limited number of resident memory pages
3. Frame Allocation Policies: Limited physical memory frames

2 Structure of Virtual Memory

1. Logical memory space(address space) is split into fixed size page
2. Some pages may be in physical memory
3. others in secondary storage

If CPU finds out a certain page is not memory resident, it will trigger a page fault, which will be handled by OS, to bring the non-memory resident page into the physical memory.
In case of huge memory space, page table via **direct paging** will also be huge, and remain

fragmented in different pages, and remain fragmented in different pages. A neater design of page table is needed

2.1 Logical address space space

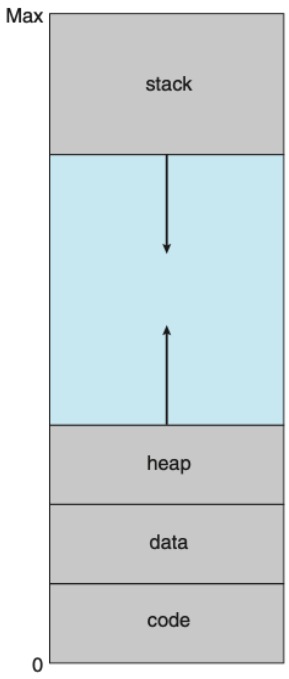


Figure 8.2 Virtual address space.

The virtual address space **of a process**refers to the logical(virtual) view of how a process is stored in memory
The virtual address begins at a certain logical address(address0), and exists in contiguous memory;
The continuous virtual address may be organized in page frame and the physical page frame assigned to the continuous virtual address space may not be continuous
It's memory management unit(MMU)'s job to map logical pages to physical page frames in memory
Note in the image above, we allocas the heap to grow upward in memory as it is used for dynamic memory allocation. We allow the stack to grow dowward in memory through successive functions calls. The large blank space between the heap and the stack is part of the virtual address space but will require actual physical pages only is the heap or stack grows.

2.1.1 Sparse address

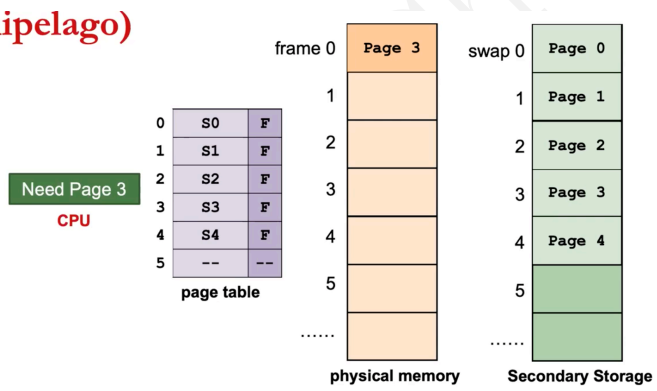
Virtual address spaces that include holes are known as sparse address space

2.2 shared memory

The virtual memory allows process to share memory by multiple process(especially system libraries)

1. System libraries can be shared by several processes through mapping of the shared object into a virtual address space. Although each process consider the shared libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all processes. A library is mapped **read-only** into space of each process that is linked with it
2. Virtual memory enables processes to share memory(the shared memory in IPC)
3. Virtual memory can allow pages to be shared during process creation with the fork() system call, thus speeding up process creation()(the lazy fork())

3 demand page



Page is loaded into memory only when it is needed

3.0.1 The benefit of demand page

When an executable program needs to be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. The problem with this approach is that we may not initially need the entire program in memory. Suppose a program starts with a

list of available options from which the user is to select. Loading the entire program into memory is a kind of waste because not all options are needed during that scenario.

3.0.2 System design principles

1. Abstraction
2. Modularity
3. Laziness
4. Speclation: eager way, will do even not needed
5. Layering
6. Locality principle
7. Indirection

The demand page is based on laziness(copy-and-write is also based on laziness)

3.1 Lazy swapper

A lazy swapper never swaps a page into memory unless that page will be needed. Note that there is a difference between swapper and pager, because a **process** is a **sequence of pages**, rather than as one large contiguous space

3.1.1 swapper and pager

1. swapper manipulates entire processes
2. pager is concerned with the indivitual page of a process

3.2 Valid bit

When a process is to be swapped in, the pager guess which pages will be used before the process is swapped out again. Also, pager will avoid reading into memory pages that will not be used anyway.
There is a valid-invalid bit(hardware support) to distinguish between the pages that are in memory and that are in disk.

- If the bit is set to "Invalid", the page either is not valid, or is valid but in disk, a **page fault** will be invoked
  - the page-table entry for a page that is not currently in memory is either simply marked invalid

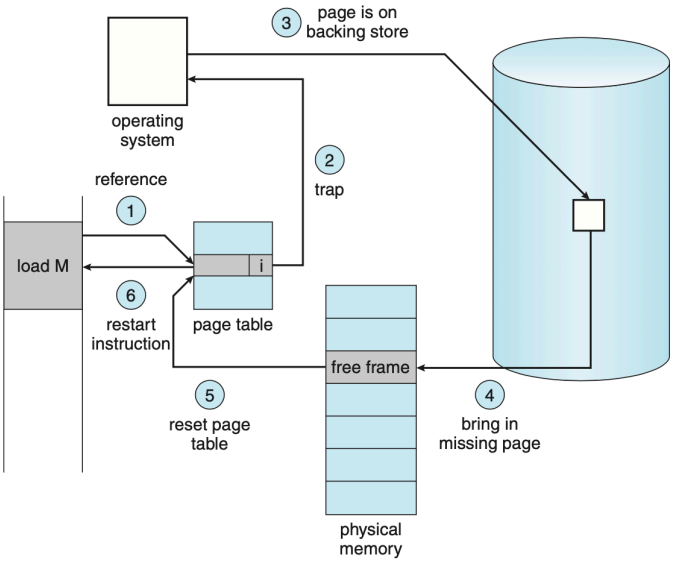
OR

  - the page-table entry for a page that is not currently in memory is either simply marked invalid contains the address of the page on disk

- If the bit is set to "Valid", it is valid and in memory

4 Page fault

Access to a page marked invalid causes a page fault. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set. This trap is the result of operating system's failure to bring the desired page into memory



The steps of page fault

1. We check an internal table(usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access
2. If the reference is invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in
3. find a free frame(by taking one from the free-frame list)
4. schedule a disk operation to read the desired page into the newly allocated frame

5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory
6. restart the instruction that was interrupted by the trap. The process can now access the page as through it have always been in memory

4.1 Difficulty of page fault

If one instruction can modify several different location. (E.g IBM system 360/370 can move up to 256 bytes from one location to another) There is possibly overlapping. During a block move, if a page fault for either source or destination block is invoked when the move is partially done(there is also case where the source and destination block is the same one, so a race condition happens)

4.1.1 Two solutions

1. Microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will make the page fault happen before any modification
2. Uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are writtten back into memory before the trap occurs. This action restores the memory to its state before the instruction was started, so that the instruction can be replaced

4.2 Statistical reasoning of demand paging

- The memory access time, denoted as ma, ranges from 10 to 200 nano seconds(if no page fault)
- Let p be the probability of a page fault ($0 \leq p \leq 1$).
- effective access time=(1-p)× ma +p× page fault time

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following events to occur

1. Trap to the operating system
2. Save the user register and process state
3. Determine that the interrupt(hard ware) was a page fault
4. Check that the page reference was legal and determine the location of the page on the dist
5. Issue a read from the disk to a free frame:
   (a) waiting in a queue for this device until the read request is served

(b) wait for the device seek and latency time

(c) Begin the transfer of the page to a free frame

6. While waiting, allocate the CPU to some other user(CPU scheduling)
7. receive an interrupt from the disk I/O subsystem(I/O completed)
8. Save the registers and process state for the other user(if step 6 is executed)
9. determine that the interrupt was from the disk
10. Determine that the interrupt was from the disk
11. correct the page table and other tables to show that the desired page is now in memory
12. wait for the CPU to be allocated to this process again
13. restore the user registers, process state and new page table and resume the interrupted instruction

In conclusion, there are three time consuming part:

1. Serve the page-fault interrupt(1 to 100 Ms)
2. Read in the page (usually around 8 Ms)
3. Restart the process(1 to 100 Ms)
4. queuing time(dependent on scheduling program)

4.3 Copy-on-write

The technique invoked during the fork, the child share part of the physical frames with parent even though the logical address shows differently, until child wanna write on the shared part
Reasons :

1. allows rapid process creation
2. minimize the number of new pages that must be allocated(if all child copy parent's text, data, heap etc.; if all child just call exec right after fork, the copied parent's memory is useless)

5 Page Table Structure

5.1 Direct Paging

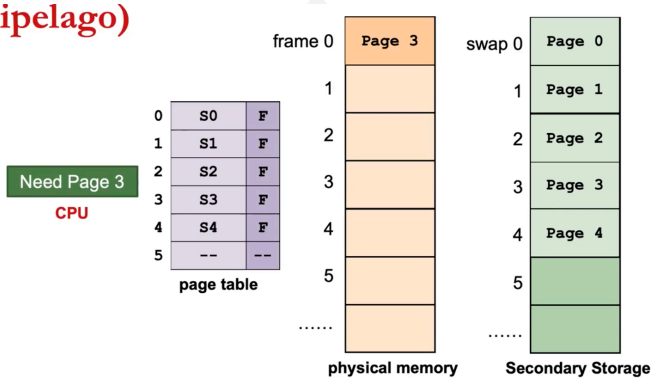
With 2^p pages in logical memory space

1. p bits to specify one unique page
2. 2^p page tbale entries(PTE) withL physical frame number+ additional information bits
   Example1:
   - Virtual Address 32 bits, page size=4KiB
   - P= 32- 12= 20 (12 bits of offsite and 20 bits of entries)
   - Size of PTE = 2 bytes
   - Page Table Size=2^20*2 bytes=2MiB

   Example 2: A computer with Virtual Address 64 bits,page size=4KiB(12bits for offset); Physical memory 16GB, with Physical address of 34 bits; memory size 2^{34}
   - Virtual Address 64 bits, page size=4KiB(12bits for offset)
   - VA 64-bit -> 16 ExaBytes of virtual address space
   - Physical memory 16GB, with Physical address of 34 bits
   - The number of virtual pages=$\frac{2^{64}}{2^{12}} = 2^{52}$ PTE enyries
   - Physical pages $\frac{2^{34}}{2^{12}} = 2^{22}$
   - Size of PTE = 8 bytes in reality
   - Page Table Size=$2^{52}*8$ bytes=$2^{55} Bperprocess$

5.2 2-Level paging

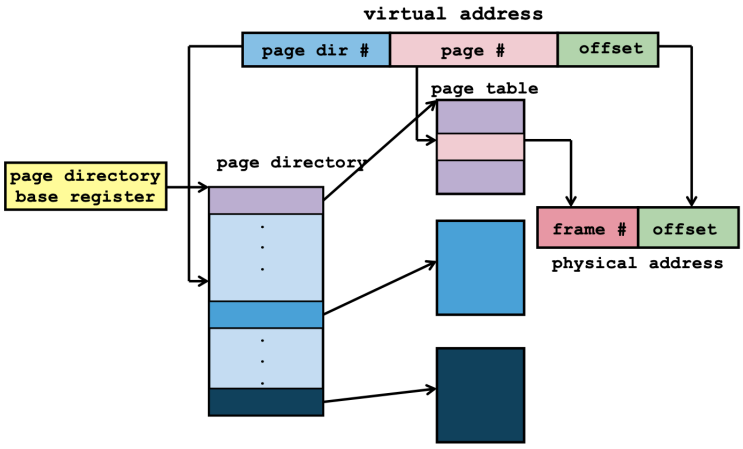
(archipelago)



We split the full page table into smaller page tables, each with page table number. If the original page table has 2^P entries, with 2^M smaller page tables, M bits is needed to unique identify one page table; each smaller page table contains 2^{P-M} entries.

1. Each smaller page table should ideally be of the same size as the page size, to avoud page fault(page table itself is a page).
2. To keep track of these smaller page tables, a single page directory is needed, which contains the 2^M
3. Page table information is kept with the process information and takes up physical memory space
4. Modern computer provides huge logical memory space(4Gb is normal)

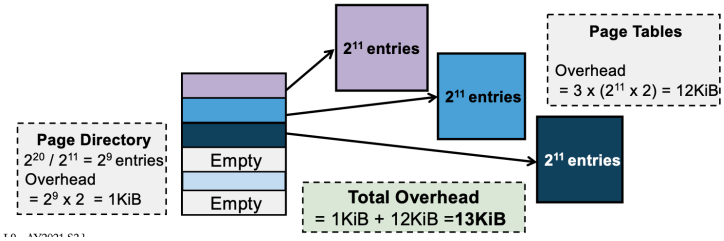
The problem of the 2-level paging is that every time needs three accesses(1 for directories, 1 for page table and 1 for the physical memory), which is slow
The way is to use TLB, which will keep track of the final entries not the directory; The directories will be only be called when the page entries needed is not in the TLB, which is called page table walks; So there is a seperate memory cache that is used for cache page table, which is in memory management unit



There is also a page directory base register, which points to the starting address of the page directory

5.2.1 Advantages

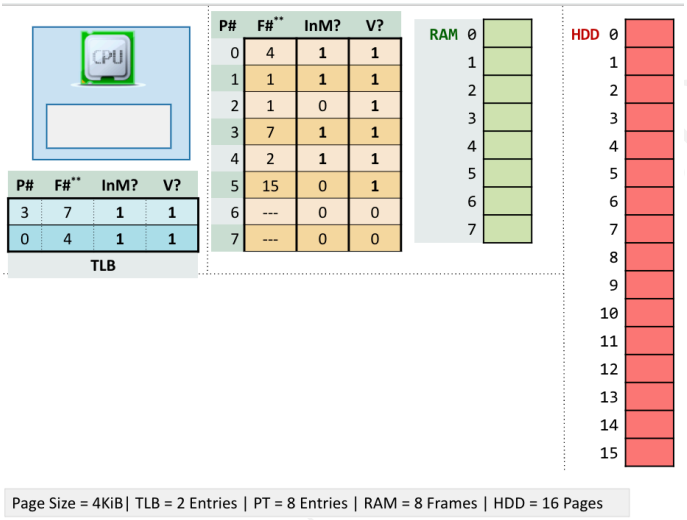
1. we can have empty entries in the page directory, the corresponding page tables need not be allocated(The ”empty” in the between directories)
2. Using the same setting as the previous example
   (a) Assume only 3 page tables are in use
   (b) Overhead= 1 page directory +smaller page table



The machine above has virtual address: 32bits; Page size 4KiB(2^{12}); PTE size 2 bytes
Hence each smaller page has $\frac{2^{12}}{2} = 2^{11}$

The overhead of the page directory is $2^9 \times 2 = 1KiB$ So the overhead of the 2-level paging needs $3 \times (2^{11} \times 2) + 2^9 \times 2$=13KiB
Without the 2-level paging, the overhead needs $2^{20} \times 2 = 2MiB$

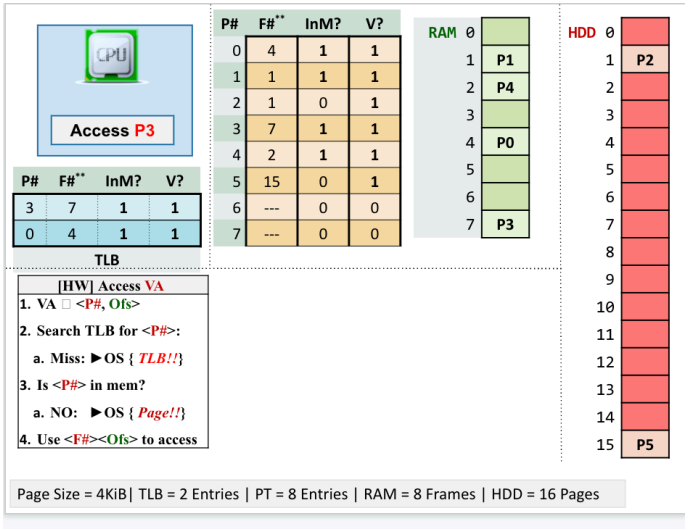
5.3 Demo: The memory allocation of page table



Page Size = 4KiB| TLB = 2 Entries | PT = 8 Entries | RAM = 8 Frames | HDD = 16 Pages

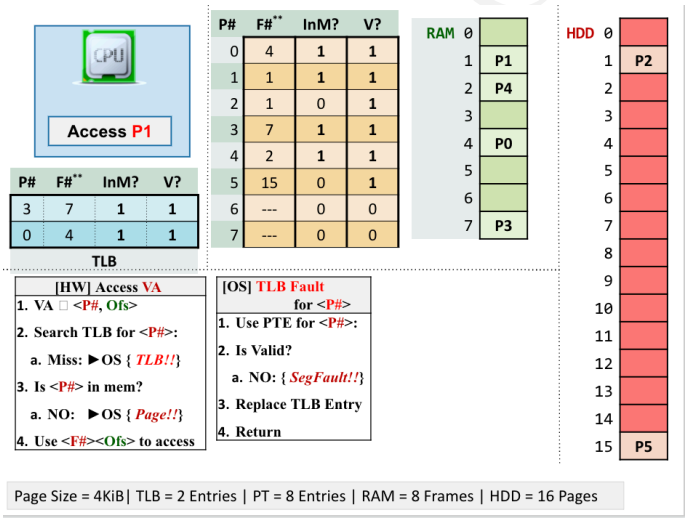
- TLB in cache
- the page table is also stored in RAM

5.3.1 Access P3



1. Firstly translate the virtual address, first search TLB and found the page 3 is in the TLB
2. check and found that the Page 3 is in memory with InM bit
3. check and found that the page 3 is valid with V bit
4. So use the <F><Off> to access

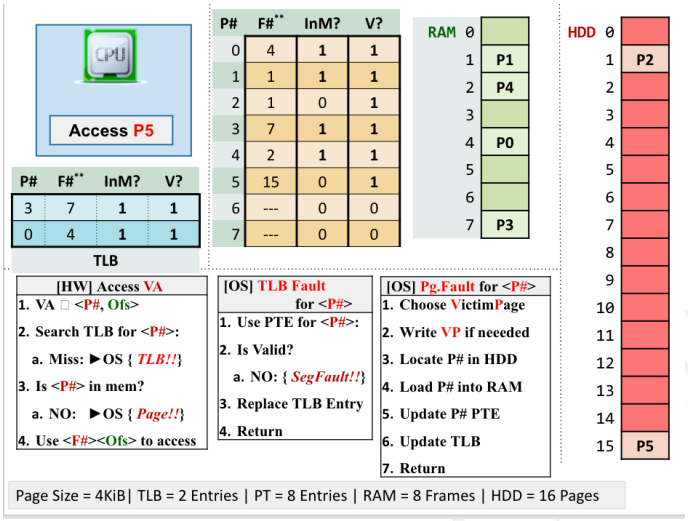
5.3.2 Access P1



1. first search TLB and found the page 1 is not in the TLB

2. so past control to the operating system for TLB fault and use the page table entry as the page number
3. page entry 1 is found in RAM page table,
4. **OS** check and found that the page 1 is valid with V bits and replace TLB entry
5. check and found that the Page 1 is in memory with InM bit
6. access with <F><Off>

5.3.3 Access P5



1. first search TLB and found the page 5 is not in TLB
2. so past control to operating system for TLB fault and use the page table entry as the page number
3. PTE 1 is found in RAM , and the OS checks the PTE1 and found it valid, hence replace TLB entry
4. in HW, check whether <P#> is in mem, NO, so pass to OS for page fault
5. OS chooses VictimPage(FIFO,CLOCK, LRU)
6. write VP if needed
7. Locate P# in HDD
8. load P# in RAM
9. Update P# PTE(it would be easier with a invert PTE)

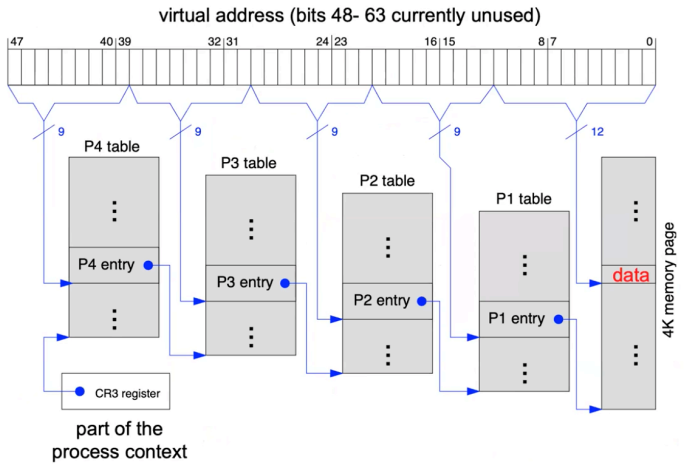
10. Update TLB

5.3.4 Page Thrushing

Page fault happens frequently

5.4 Hierachical page table

This picture occurs in examination quit often

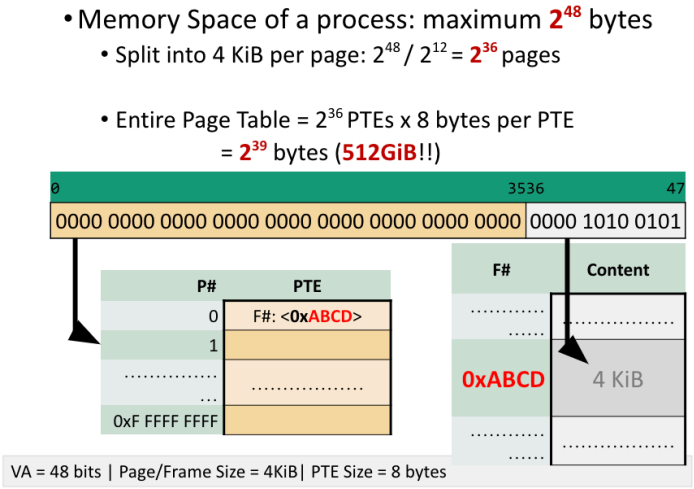


Most of today's system applies 64-bit, but the system ususally does not use up all of the virtual address, usually just use 48 bits(bits 48-63 currently unused

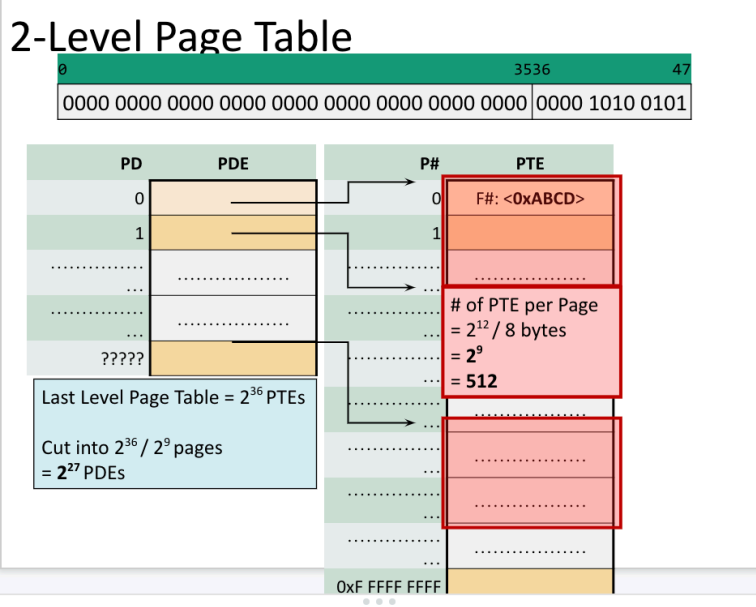
- There is a register that keeps the root directory(CR3 in the image above), which is stored as part of process context
- In practice, each entry size is 8 bytes, each sub-level table's size is 4KiB, so each sub-level table has $\frac{2^{12}}{8} = 2^9$, therefore, it takes 9 bits to index a sub-level table
- The frame number can only be got in the final page table(with the help of offset, 12 in the image above), where
- The TLB stores the lat level page table, if the TLB is misses, needs to start from the first level to get the target page table, which is done by MMU(memory management unit,hardware)
- The structure of the 5-level page table is fixed, because it is implemented in the hardware, it's a contrast between hardware and software

5.5 The statistical reasoning of multi-level page table

5.5.1 one level



5.5.2 two-level

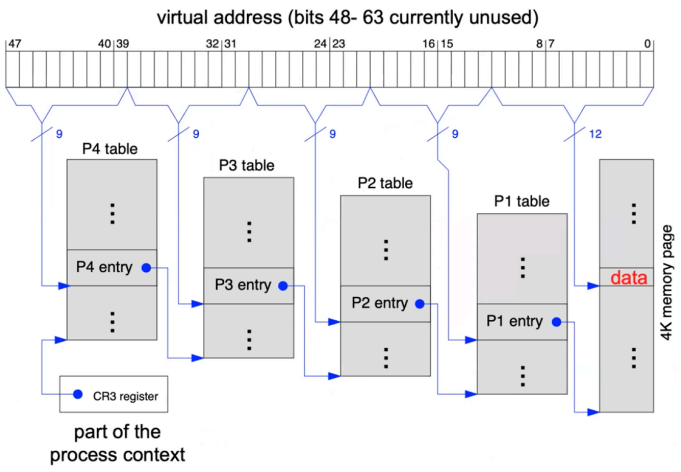


6 Inverted Page Table

Normal page table is per-processs information. Also, mapping from physical frame to logical memory is inaccessible

Inverted Page Table keeps a single mapping of physical of physical frame to <pid, page num-

ber>. Here the page number is specific to pid. The pid indicate the information of process



The offset of the inverted page table entry will be the frame number of physical frames. Hash is needed to make the query faster.
the number of entries is equal to the number of frames in the main memory. It can be used to overcome the drawbacks of page table.
We can save this wastage by just inverting the page table. We can save the details only for the pages which are present in the main memory. Frames are the indices and the information saved inside the block will be Process ID and page number.
Since two different processes can have similar set of virtual addresses, it becomes necessary in Inverted Page Table to store a process Id of each process to identify it' s address space uniquely. This is done by using the combination of PId and Page Number. So this Process Id acts as an address space identifier and ensures that a virtual page for a particular process is mapped correctly to the corresponding physical frame.

7 Page Replacement Algorithm

In the case of page fault, we need to evit(free) a memory page, by writing back the old page to the memory if the page id modified, therefore dirty

- clean page: not modified, no need to write back
- dirty page: modified, need to write back

To decide which page to evict, we need to like at a suitable replacement algorithm. A good replace algorithm should reduce the total number of page faults
In actual memory reference: Logical address = Page Number +Offset, for page replacement Algprithm, only page number is important

7.1 Goals

1. Minimize page fault
2. Fast(the run time of the algorithm)

memory access time:$T_{access} = (1-p) * T_{mem} + p * T_{pageFault}$

- p=probability of page fault
- T_{mem}=access time for memory resident page
- $T_{pageFault}$=access time if page fault occurs

Since $T_{pageFault}$ is much larger than T_{mem}, need to reduce p to keep T_{access} reasonable

7.2 Optimum(OPT)

Optimum page replacement algorithm replace the page that will not be used again for the longest period of time

1. If the page is in memory, only update next use time
2. Else
   (a) If memory is not full, load it and update next use time
   (b) Else, choose the existing page with the largest next use time to evict, and load the new page

This guarantees minimum number of page faults
However, it is not realizable, due to requirement of future knowledge of memory access

7.3 FIFP page replacement

Memory pages are evicted based on their loading time. One evicts the oldest memory page. Therefore,
If the page is in memory, use. Else

- If memory is not full, load it and update loaded-at time
- Else choose the existing page with smallest loaded-at time to evict, and load the new page

The problem with FIFO algorithm is Belady's Anomaly, where the number of page fault can increase if the number of physical frame increases. The reason is FIFO does not exploit temporal locality
E.g.
Use 3/4 frames to try 1 2 3 4 1 2 5 1 2 3 4 5
It can be found that 3 frames gives better result of 4 frames

8 Least Recently Used(LRU)

In LRU, we replace the page that has not been used in the longest time. It avoids Belady's Anomaly
If the page is in memory, only update **last use time** Else

- If memory is not full, load it and update last use time
- Else, choose the existing page with smallest last use time to evict, and load the new page

8.1 Implementation of LRU

8.1.1 Use a counter

1. A logical "time" counters, which is incremented for every memory reference
2. Page table entry has a "time-of-use" field
    - Store the time counter value whenever reference occurs
    - Replace the page with smallest "time-of-use", no need to search through all entries

8.1.2 Stack

Maintain a stack of page number
If page X is referenced:

- Remove from the stack(for existing entry)
- Push on top of stack

Replace the page at the bottom of stack: No need to search through all entries

8.2 Disadvantage of LRU

the last use time can overflow, and also the searching of smallest last use time page requires a complete search

9 CLOCK(Second-Change Page Replacement)

It used a modified FIFO to give a second change to pages that are accessed. We need the page table entry to maintain a reference bit:

- 1 = Accessed
- 0 = Not Accessed
- Initialize the next victim pointer to the first fram when it is loaded
- If the page is in memory, set reference bit to 1
- else
  - If memory is not full, load it and set reference bit to 0
  - Else, advance victim pointer and check again
  - If all pages have reference bit 1, change the original victim to the new page

10 Frame Allocation

To allocate N physical memory frames among M processes, some simple approaches can be

- Equal distribution, every one get N/M frames
- Proportional allocation to the size of process, $size_p/size_{total} * N$

10.1 Local replacement

During page replacement, if victim page is selected among pages of the same process that causes page fault, it is known as **local replacement**.

10.1.1 Advantages of local replacement

stable performance between multiple runs, since number of pages allocated remains constant

10.1.2 Problem of local replacement

If frame allocated are not enough, potentially hindering the progress of a process

10.2 Global replacement

it is known as **global replacement**
Local replacement has advantage of stable performance between multiple runs, since number of pages allocated remains constant

10.2.1 Advantages

Allowing self-adjustment between processes, so that process that needs more frame can get from others

10.2.2 Disadvantage

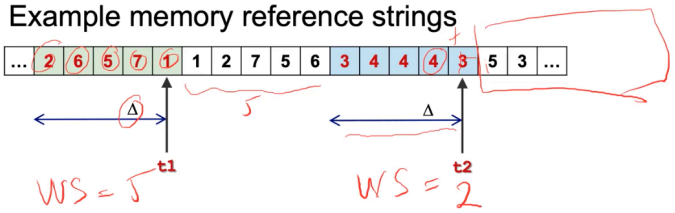
Frame allocated to a process can differ from run to run If global replacement is used, thrashing steals page from other process, which can cause**cascading thrashing** If local replacement is used, though thrashing is localized, single process doing thrashing can hog the I/O and degrades performance of other processes
With thrashing, ten process may behave poor than seven processes
IMPORTANT: thrashing happens on both local and global replacement(if there are too many process than the number of frame can support)
Problem of both equal allocation and proportional allocation is that the data needed by a process may variate as time goes by(the set of pages of a process can change)

11 Working Set Model

Every set window has a time interval
We define the working set Window δ as an interval of time. We denote W(t, δ) to be the active pages in interval at time t. We allocate enough frames for pages in W(t, δ) to reduce possibility of page fault.
The accuracy of working set model is directly affected by the δ chosen

- Too small: may miss pages in the current working set
- Too big: May contains pages from a different working set

- Example memory reference strings



in t1 (t1+δ),W(t1, δ)=5

in t2 (t2+δ),W(t2, δ)=2 (because in t2, only 2 pages existing, which needs two frame)