# CSC 236 - Computer Organization and Assembly Language for Computer Scientists

## 4. Process(dynamic) = running program (static)

*ex. class – static, object – dynamic*

*ex. multi process –chrome, multi-thread – fox*

process includes machine state *ex. memory and registers and I/O*

lifeless.

sits on disk (flash-based SSDs in some modern systems) in executable format with instruction, waiting to spring into action

Time sharing

creates illusion by virtualizing CPU è multiple processes but slower performance

How to implement virtualization?

mechanisms (low-level machinery) – methods/protocols

How to virtualize? running one process, stop and run again

Counterpart : space sharing

context switch – stop running one program and start sunning another on a CUP

policies – high-level intelligence

*ex. scheduling policy make decision about which program to run through historical information (run more), workload knowledge (type of program), and performance metrics (optimizing for interactive performance or throughput?)*

*TIP: separating policy ( which question) vs mechanism (how questions)*

*Ex "which proves should the OS run" vs "how does an OS perform a context switch"*

## 4.1 The Abstraction: A Process

Process

taking an inventory of the different pieces of the system is accesses/affects during execution

Machine state

what a program can read or update when it is running. At any given time, what parts of the machine are important to the execution of this program?

memory : address space (instruction, data)

registers : instruction explicitly read/update registers

*ex. PC (program counter or IP instruction pointer) tells which instruction of the program is executed.*

*Stack pointer and associated frame pointer are used to mange the stack for function parameters, local variables and return address*

## 4.2 Process API

Create new processes according to the input

Destroy process forcefully (may exit themselves when complete)

Wait for a process to stop running

Miscellaneous control suspends a process and then resume it

status – information about how long it run and its state

## 4.3 Process Creation : A Little More Detail

Load code and static data into memory (address space of the process)

*Ex modern in a lazy way – only loading needed pieces from disk to memory*

*Ex paging and swapping*

Allocate run-time stack and give it to process

Stack :

in c, used for local variables, function parameter and return addresses

fill in arguments parameters to the main function *ex. argc argv*

Allocate heap – malloc needing more memory – not malloc

Initialization task related to I/O

*ex. Unix has three open file descriptors for standard input, output, and error*

## 4.4 Process States

Running on a processor and executing instructions

Blocked

performed some operations è not ready to run until

Ready to run but OS chose not to run it at this given moment

Other event takes place

*ex. initiates I/O request to a disk è other process can use the processor*

*decisions are made by OS scheduler*

*ex. context switch vs. CPU idle time*

| *P1 run* | *blocked* | *ready* | *run* | *Complete* |
| --- | --- | --- | --- | --- |
| *P2 ready* | *run* | *complete* |  |  |

(new – being created à(admitted) to ready

running à (exit) terminated

terminated/final/zombie(UNIX) state – exited but not cleaned up

*other processes (*usually the parent that created the process) *can examine the return code to see if it executed successfully (0) or not (non-zero)*

wait is called by parent to notify OS to clean up the relevant data structure – useful for parent to get the output and clean up

)

## 4.5 Data Structures

process list or PCB (process control block)– necessary for running multiple program

read processes and additional information to track running and blocked process

register context holds the contents of its register for a stopped process.

stop – registers saved to memory à restore – resume running

Restore - placing their values back into the actual physical registers

common – process state, program counter, CPU registers, CPU scheduling infor., memory management infor., accounting infor. (for payment *ex. Cloud*), I/O status infor.

*ex. XINU*

*struct pentry {…}*

*extern struct pentry porctab[];*

*extern int numproc, nextproc, currpid;*

## 5. System call – Process API

## 5.1 The fork() System Call : create a new process

4 processor if rc = fork() after the int rc = fork();

getpid() –

gets PID (process identifier used to name the process)

the process created is almost an exact copy of the calling process (there are two copies of the program p1.c running and are to return from the fork()).

Why almost? own the address space, register, PC… fork returns different value

Child – the newly-created process start at the fork() call – receive 0 from fork

Parent – receive the PID of the newly-created child from fork

Non-determinism

the output is not deterministic < = OS has freedom to choose

either one might run at that point ç scheduler determines which runs first *ex. multi-threaded program*

## 5.2 The wait() System Call :

parents waits for child to finish task = > child is sure to print first

wait() returns to parent when the child finishes executing è deterministic

sleep(1) in child branch, and sleep(2) in parent branch = > child is likely to print first

## 5.3 The exec() : runs program different from the calling program

*ex. execl, execlp(), execle(), execv(), execvp() and execvpe()*

…

runs the wc in p3.c – infor. about lines, words and bytes in the file

how?

load code from the given executable file and overwrites its current code segment(static data). heap, stack and other parts of the memory space of the program are re-initialized.

it transforms (not creates) the running program into a different running program.

Child has a copy of the parent, but in the different memory, load the image from the disk

Try code replace "wc" with the current location path

## 5.4 Why? Motivating The API

separation of fork and exec è lets shell run after fork and before exec

shell is a user program showing you a prompt and waiting for input

fork() creates child process to run command, exec() to run command, wait() waits for completion

*ex. wc p3.c > new.txt*

*child created è shell close standard output and opens new.txt è exec()*

STDOUT\_FILENO will be the first available and get assigned after open()

all the output is redirected to p4.output

close(STDOUT<FILENO); will close the default output file

open() will be output file

## 5.5 Other Parts of The API

pipe()

output is connected to an internal pipe, input of another is connected to the same pipe. è output is used as input to the next

*ex. count word occurs with pipes and grep and wc*

*grep –o foo file | wc –l and marvel at the result*

Other system calls

kill() sends signal to a process/directives to die

ps – running program information

top – processes of the system and CPU and other resources they are eating up

MemuMeters keeps track of CPU utilized

## 7.1 Workload assumption

* + Each job runs for the same amount of time
  + All jobs arrive at the same time
  + Once started, each job run to completion
  + All jobs only use the CPU (no I/O)
  + Run-time of each is known

## 7.2 Scheduling Metrics (used to measure sth) enables us to compare different scheduling policies

* Performance metric
* turnaround time = complete time - arrival time to system
* Fairness and performance are odds
* Performance up => cost of preventing jobs from running => fairness down

## 7.3 FIFO or FCFO (First In/Come, First Out) - non-preemptive

* Simple and easy to implement
* *Ex A = B = C = 10s, A < B < C arrived turnaround time T average*
* *Ta = 10s, Tb = 20s, Tc = 30s, T average = 20s*
* *If relax 1)… and A = 100s, B = 10s, C = 10s => T avg = 110s*
* Convoy effect : relatively-short potential consumers of a resource get queued behind heavyweight resource consumer

## 7.4 SJF (Shortest Job First) - non-preemptive

* optimal scheduling algorithm
* applied to system where average turnaround time matters
* *Ex Ta = 120s, Tb = 10s, Tc = 20s, T avg = 50s*
* *If relax 2)… convoy effect happens if A > B and C, Tavg = 103s*

## 7.5 STCF (Shortest Time-to-Completion First)

* PSJF (Preemptive Shortest Job First)
* relax 3), context switch => preemptive
* *Ex Ta = 120-0, Tb = 20-10, Tc=30-10, T avg = 50s*

## 7.6 A New Metric: Response Time <= time sharing

* T response = T run - T arrival -- interactive performance
* *Ex A = B = 10s, A = 0, B = C = 10s arrives,*
* *R avg = 3.33s*
* 7.7 RR (Round Robin) - preemptive
* Time-slicing : runs a job for a time slice (scheduling quantum)
  + Time slice = a multiple of the timer-interrupt period
* *Ex timer = 10ms, TS = 10, 20,…ms*
* Shorter slice => better R performance, but more context switch cost
* RR used when R is only metric
* *Ex A = B = C = 5s, A = B = C arrives, 1s slice*
* *SJF : R avg = 5s T avg = 10s*
* *RR : R avg = 1s, T avg = 14s*
* Amortization : used when cost to some operation is fixed to reduce costs
* Context switching from saving/restoring registers and scheduling
* program run -> build up state (CUP caches, TLBs, on-chip hardware -> switch -> flush current state and new state brought in => T performance down
* 7.8 Incorporating I/O
* relax 4) : all programs perform I/O
* Decision 1 : schedule another job when blocked
* Blocked 1) waiting for I/O completion 2) sending to hard disk drive
* Time < = current I/O load of the drive
* Decision 2 : I/O completes = > ready
* Overlap operations to maximize utilization of system.
* *Ex A (10ms CPU + 10ms I/O…) B(50ms CPU)*
* *Ex performing I/O or sending message to remote machine*

| * *STCF (runs A then B)* | * *Overlap* |
| --- | --- |
|  |  |

* 7.9 No More Oracle
* SJF/STCF cannot run if relax 5)
* how long
* 4-2 + 8-4 + 10-6 + 14-9 + 15-4 + 19-8 + 20-14 = 43 /5=8.6

8

* MLFQ first in CTSS(Compatible Time-Sharing System) -> Turing Award
* Optimize turnaround time without priori knowledge (SJF, STCF)
* Learn from history < = job has phases of behavior and predictable
* Optimize response time for interactive users (RR)

## 8.1 MLFQ : Basic Rules

* 1. P(a) > P(b), A runs;
  2. P(a) = P(b), A & B run in RR
* Queues with different priority level. How to determine priority?
* Assigned according to job observed behavior
* *Ex relinquishes CPU to wait for input = > up, interactive behavior requires. Intensively long time = > down*
* 8.2 Attempt #1 : How to Change Priority
  1. A job enters system -> highest priority(topmost) : assume it is the shortest job
  2. a. job uses entire time slice while running -> reduce priority
* b. job gives up CPU before time slice is up -> stay
* *Ex long running job with short job*
* *A : long-running CPU-intensive job, run first*
* *B : short-running interactive job, arrive latter*

| * *Time* | * *-100* | * *-1 10* | * *-120* | * *-200* |
| --- | --- | --- | --- | --- |
| * *A* | * *--* | * *Q2* | * *Q1* | * *--* |
| * *B* | * *Q2-Q0* | * *-* | * *-* | * *Q0* |

* Problems
  1. Starvation : too many short-run job = > long-running job never get CPU time (#2)
  2. Gaming the scheduler (#3)
* *Ex relinquishing CPU by issuing I/O when time slice is over*
  1. Change behavior over time (#2)
* *Ex CPU-bound -> interactivity*
* 8.3 Attempt #2 : The Priority Boost
  1. After some time S, move all jobs to the topmost queue
* Solved problem :
  1. Starvation - long-run job gets CPU time in RR fashion
  2. Change behavior -> proper priority after boost time
* What S be set to? Voo-doo constants
* High = > long-run starves, low = > short-run not getting proper share of CPU
* 8.4 Attempt #3: Better Accounting
* To prevent gaming of scheduler, we rewrite rule4 to anti-gaming rule…
  1. a job uses up its time allotment at given level -> priority is reduced

| * *Time* | * *10* | * *30* | * *80* |
| --- | --- | --- | --- |
| * *A* | * *Q2* | * *Q1* | * *Q0* |

* 8.5 Tuning MLFQ and Other Issues
* How to parameterize as a scheduler?
* # of queues, time slice per queue, boost time? - workload lead to balance
* *Ex higher level queue - shorter time slice*
* Ousterhout’s Law - avoiding voo-doo constants
* Try to make the system learn a good value = > not straightforward
* Configuration file with default parameter values to tweak
* 60 queues, 20ms - #00ms time slice, 1s boost time
* Mathematical formulae to calculate priority level
* *Ex FreeBSD and decay-usage algorithms*
* Advice from the user to give OS hints
* *Ex hints in scheduler(nice), memory manager(madvise)…*

13

* MLFQ first in CTSS(Compatible Time-Sharing System) -> Turing Award
* Optimize turnaround time without priori knowledge (SJF, STCF)
* Learn from history < = job has phases of behavior and predictable
* Optimize response time for interactive users (RR)
* 8.1 MLFQ : Basic Rules
  1. P(a) > P(b), A runs;
  2. P(a) = P(b), A & B run in RR
* Queues with different priority level. How to determine priority?
* Assigned according to job observed behavior
* *Ex relinquishes CPU to wait for input = > up, interactive behavior requires. Intensively long time = > down*
* 8.2 Attempt #1 : How to Change Priority
  1. A job enters system -> highest priority(topmost) : assume it is the shortest job
  2. a. job uses entire time slice while running -> reduce priority
* b. job gives up CPU before time slice is up -> stay
* *Ex long running job with short job*
* *A : long-running CPU-intensive job, run first*
* *B : short-running interactive job, arrive latter*

| * *Time* | * *-100* | * *-1 10* | * *-120* | * *-200* |
| --- | --- | --- | --- | --- |
| * *A* | * *--* | * *Q2* | * *Q1* | * *--* |
| * *B* | * *Q2-Q0* | * *-* | * *-* | * *Q0* |

* Problems
  1. Starvation : too many short-run job = > long-running job never get CPU time (#2)
  2. Gaming the scheduler (#3)
* *Ex relinquishing CPU by issuing I/O when time slice is over*
  1. Change behavior over time (#2)
* *Ex CPU-bound -> interactivity*
* 8.3 Attempt #2 : The Priority Boost
  1. After some time S, move all jobs to the topmost queue
* Solved problem :
  1. Starvation - long-run job gets CPU time in RR fashion
  2. Change behavior -> proper priority after boost time
* What S be set to? Voo-doo constants
* High = > long-run starves, low = > short-run not getting proper share of CPU
* 8.4 Attempt #3: Better Accounting
* To prevent gaming of scheduler, we rewrite rule4 to anti-gaming rule…
  1. a job uses up its time allotment at given level -> priority is reduced

| * *Time* | * *10* | * *30* | * *80* |
| --- | --- | --- | --- |
| * *A* | * *Q2* | * *Q1* | * *Q0* |

* 8.5 Tuning MLFQ and Other Issues
* How to parameterize as a scheduler?
* # of queues, time slice per queue, boost time? - workload lead to balance
* *Ex higher level queue - shorter time slice*
* Ousterhout’s Law - avoiding voo-doo constants
* Try to make the system learn a good value = > not straightforward
* Configuration file with default parameter values to tweak
* 60 queues, 20ms - #00ms time slice, 1s boost time
* Mathematical formulae to calculate priority level
* *Ex FreeBSD and decay-usage algorithms*
* Advice from the user to give OS hints
* *Ex hints in scheduler(nice), memory manager(madvise)…*

14

## 14. Memory 14.1 Types of Memory

Stack(automatic memory) : de/allocation are managed implicitly by the compiler for you  
 Compiler make space on the stack when you initialize parameter  
 Compiler deallocate the memory when you return from the function  
 Heap(long-lived memory) : de/allocation are explicitly handled by you  
 Ex int \*x (stack)= (int \*)(cast)malloc( sizeof( int ) ) (heap);

## 14.2 The malloc() Call

Including stdlib.h to check if you are calling malloc correctly  
 sizeof() compile-time operator to make sure the size. It is not a function call  
 Ex sizeof(array) returns sizeof(pointer to the array)  
 int x[10]; //sizeof(x) = 4 (32-bit machine) or 8(64-bit machine)  
 Use malloc(strlen(s) + 1) to malloc a string  
 free() is needed without automatic memory management (garbage collector)  
  
14.4 Common Errors  
 Forgetting to allocate memory before some calls (strcpy) = > segmentation fault  
 Not allocating enough memory = > buffer overflow  
 Not initialize allocated memory = > uninitialized read  
 Reads from the heap some data of unknown value  
 Not free memory = > memory leak  
 Garbage collector won't free referred memory. OS clean up when process dies  
 Free memory repeatedly = > double free OR before you are done = > dangling pointer  
 Free other value = > invalid frees  
 Two types of memory management  
 1st level is by OS : hands out memory to processes when run, and takes back when processes exit or die = > no memory leak once process exits  
 2nd is within each process ex free()  
 Leaking memory not cause operational problem, but takes application memory  
 Tools - purify, and valgrind  
   
14.5 Underlying OS Support  
 malloc and free library(X system) calls = > library manages space in virtual address space. It is built on top of system calls -> calls OS for more memory/release  
 brk(void \* add) (sbrk is passed an increment but similar purpose)  
 change the location of the program's break : location of the end of heap  
 New break >/< current = > in/decrease size of heap  
 • Never directly call them. They are used by memory-allocation library  
 mmap() creates an anonymous memory region(related to swap space) in program  
   
14.6 Other Calls  
 calloc() initializes allocates memory to zeros. realooc() to resize the allocated memory

## 15 Virtualization of CPU - IDE(limited direct execution) :

* PS interposing at critical point to maintain control over hardware
* Virtualization of memory :
* Efficiency : use hardware support (rudimentary -> complex (TLBs, page-table…))
* Control : only access own memory with help from hardware, VM system
* Flexibility : programs can use their address spaces in their way
* The Crux - how to efficiently and flexibly virtualize memory?
* (Hardware-based) address translation : memory reference
* Hardware transforms memory access, virtual address(provided by instruction) -> physical
* Hardware redirect application memory references to actual location
* OS should manage memory to keep track of free list(memory location not in use)
* 15.1 Assumptions
  1. User's address space must be placed contiguously in physical memory
  2. Size of the address space < physical memory
  3. Each address space is the same size
* 15.2 An Example
* Interposition : hardware interpose on each memory access and translate ex transparency
* Ex process and its address space
* It's not the same, all three are for x=x+3
* Move data from mem to register, add to register, move from register to mem

| * int x = 20; | * 128 movl 0x0(%ebx), %eax | * Address x is in ebx -> load 0+ebx into eax (general-purpose register) |
| --- | --- | --- |
| * x = x + 3; | * 132 addl $0x03, %eax | * Add 3 to eax register |
|  | * 135 movl %eax, 0x0(%ebx) | * Store value in eax back to memory at the same location |

| * 0-1  * 1-2 | * Program code | * 128 * 132 * 135 | * Address is at 128, but value is at stack 15KB * Code and data are in process's address space |
| --- | --- | --- | --- |
| * 2-4 | * Heap |  |  |
| * 4-14 | * free |  |  |
| * 14-16 | * Stack | * (15)20 | * Value of x |

* But OS want to place process in physical memory (not address 0)
* How relocate this process and make it transparent to process?
* Ex physical memory with single relocated process

| * 0-16 | * ~32 | * ~48 | * ~64 |
| --- | --- | --- | --- |
| * OS | * Not in use | * code | heap-> | allocated but not in use | <-stack | * Not in use |

* Process's address space is placed in memory. OS relocated the process into physical address 32KB, the other two are free(free list : 16~32, 48~64)
* 15.3 Dynamic (Hardware-based) Relocation
* Physical add = virtual add + base
* Two registers in CPU : base-and-bounds pair (one per CPU)
  1. allows placing address space to physical memory
  2. process only access own address space
* Base register : transformation to physical mem
* OS decides where to load from physical memory(32) and sets register to that value
* Generating memory reference(virtual), it is translated by the processor…
* Ex Hardware in turn add contents of the base register to this add. -> physical add.

| * int x = 20; | * 128 movl 0x0(%ebx), %eax |
| --- | --- |

* PC (program counter) is set to 128
* Hardware needs to fetch this instruction :
* Add value to base register value of 32(32768) -> physical add. (32896)
* Fetch instruction from physical add.
* Processor executes instruction (loads from virtual add. 15
* Processor add to the register (32) -> final physical add. 47
* Address Translation : virtual add. -> physical add.
* Hardware takes a virtual add., transforms it into physical add.
* Relocation happens at runtime and possible to move address space after the process has started running (dynamic)
* OS will issue the hardware to change the base register
* Bounds register : protection
* Check memory reference is legal (in bounds). Holds size of address space -> check virtual address against it first before adding the base
* Ex bounds register would always be 16KB. When virtual add. is > bounds < 0, CPU raise an exception -> likely to terminate the process
* Holds physical address of the end of the add. space -> add the base and ensure in bound
* 15.4 Hardware Support : A Summary
* Separated modes : processor status word keeps track which mode is running and switch
* Privileged/kernel mode - access to entire machine
* User mode - run applications, limit access
* Base and bounds registers : part of the MMU (memory management unit)
* User mode runs, hardware translates add. by adding base value to the virtual add.
* Hardware provides instruction to modify the registers = > OS can change them in different processes
* privileged instruction - register only modify in kernel mode
* Exception generated by CPU when access illegal add. -> CPU stop user program -> arrange OS "out-of-bounds" exception handler.
* Dynamic relocation : hardware requirement

| * Privileged mode | * *prevent user-mode processes from executing privileged operations* |
| --- | --- |
| * Base/bounds register | * *support address translation and bounds checks* |
| * Ability to translate virtual add. and check if in bounds | * *Circuitry to do translations and check limits; in this case, quite simple* |
| * Privileged instruction(s) to update base/bounds (context switch) | * *OS must be able to set these values before letting a user program run* |
| * Privileged instruction(s) to register exception handlers | * *OS must be able to tell hardware what code to run if exception occurs* |
| * Ability to raise exceptions | * *When processes try to access privileged instructions or out-of-bounds memory* |

* 15.5 OS Issues
* Memory management
* Base/bounds management
* Exception handling
* When process is created, find space for add. space
* 2)3) : free list in physical add. as array to search when process is created
* When a process is terminated, puts memory back on free list, clean up data structure
* When a context switch occurs, save & restore base-and-bounds pair
* Reason : value differs for each program
* Stop run -> save base bounds registers to memory, process structure or PCB(process control block)
* Resume process -> resume value of two registers on the CPU
* Steps to move address space when process is stopped…
* De-schedule process -> copy add. space from current to new location -> update and save register to point to new location
* Exception handler installed at boot time via privileged instructions
* OS at boot

| * Kernel mode | * Hardware |
| --- | --- |
| * Initialize trap table * Start interrupt timer * Initialize process table * Initialize free list | * Remember address of (system call, time, illegal mem-access/instruction) handler * Start timer; interrupt after X ms |

* Limited Direct Execution Protocol and Dynamic Relocation
* Kernel mode - prepare to start P A : 1) allocate entry in process table 2) allocate mem for process 3) set base/bounds registers 4) return-from-trap(into A) -> Hardware - 1) restore register of A 2) move to user mode 3) jump to A's PC -> Program(user mode) - 1) A run 2) fetch instruction -> Hardware - translate virtual add. and perform fetch -> Program - execute instruction -> Hardware - if explicit load/save: 1) ensure add. is legal 2) translate virtual add. and perform load/save ->… -> Hardware - 1) timer interrupt 2) move to kernel mode 3) jump to interrupt handler ->
* Kernel - 1) handle the trap 2) call switch() routine i) save reg(A) to proc-struct(A) including base/bounds ii) restore reg(B) from proc-strct(B) including base/bounds 3) return from trap into B -> Hardware - 1) restore reg(B) 2) move to user mode 3) jump to B's PC -> Program - execute bad load -> Hardware - 1) load out-of-bounds 2) move to kernel 3) jump to trap handler -> Kernel - handle the trap ii) decide to terminate B ii) de-allocate B's memory iii) free B's entry in process table

16

Intro :

Free space between heap and stack = > bb registers relocate entire mem. space = > hard to run a program when the entire add. space not fit into mem.

Internal fragmentation : allocated memory not used,

virtual space size constraint

## 16 Segmentation : Generalized Base/Bounds

Idea : having a bb pair per logical segment(a contiguous portion of the add. space of a particular length) of the address space

Three logically different segments : code, stack and heap

Segmentation allows OS to place each segment in diff. parts of physical mem. = > not fill physical mem. with unused virtual add. Space

Ex segments in physical mem. unused address space = sparse add. space.

| 0~16 | ~26 | ~28 | ~32 | ~34 | ~36 | ~64 |
| --- | --- | --- | --- | --- | --- | --- |
| OS | Not use | stack | Not use | code | heap | Not use |

Code mem : virtual add. = 100 in code segment, add the base value to the offset (100+32KB, or 32868), check in bounds(2KB) = > physical add. 32868

1 KB = 1024 B

1 KB ~ (2KB-1)

Heap mem : virtual add. = 4200, physical add. = 4200+34KB = 39016, not in bound = > Extract offset into heap ex. Heap virtual add. 4KB(4096), offset = 4200-4096=104, physical add = 104+32KB = 34920

Segmentation fault/violation : access illegal add.

Illegal add. : virtual add.=7KB heap. Hardware detect out of bound -> trap into OS -> termination of offending process

16.2 Which Segment are we Referring to?

Explicit approach : 2 bits for segment reference, 12 bits for virtual address. 01 heap -> use heap bb pair to relocate add. to physical location. 00 code

| 0 ~ 4KB-1 | ~8KB-1 | 12KB -1 | ~ 14KB -1 |
| --- | --- | --- | --- |
| Code  0^2+1^14 | heap | stack | Not used |

2^14 for all the segments, 4KB ~ 8KB - 1 heap and stack 8KB ~ 12KB - 1 to stack

Ex 4200 virtual add. -> 34920 physical add.

| 13~12 segment | 1 1~0 offset = 104 |
| --- | --- |
| 01 | 000001 101000 |

Protection illegal address access (all are in hardware : no source code but hardware circuit)

Segment = (VirtualAddress & SEG\_MASK0x3000) >> SEG\_SHIFT12

Offset = (VirtualAddress & OFFSET\_MASK0xFFF

If (Offset >= Bounds[Segment])

RaiseException(PROTECTION\_FAULT)

Else

PhyAddr = Base[Segment] + Offset

Register = AccessMemory(PhyAddr)

Ex 0x1400 virtual -> physical 1KB + 1KB = > OS interrupts

Some put code and heap the same segment -> 1 bit for segment reference

Implicit approach : hardware determines the segment by noticing how add. was formed

Ex address generated from program counter -> in code segment

Address is based off the stack/base pointer -> in stack. Other in heap

16.3 What about the Stack?

Stack grows (up)backward (28~26 why not just start 26) -> virtual add. (16~14)

Hardware support : bb register + way the segment grows (1 positive, 0 negative)

Code and heap grow positive 1, stack grows negative 0

Ex virtual 15, physical 27. binary 11 1100 0000 0000, offset =3KB

Segment can be 4KB, negative offset = -1KB, physical add = 28 - 1

Bound check use absolute value of the negative offset

16.4 Support for Sharing ex code sharing

Protection bits : basic support adds a few bits per segment, indicating r/w segment /execute code in segment

Read-only = > share among processes without harming isolation

Hardware checks if the virtual add is in bounds + access is permissible

| Segment | Base | Size | Grow positive | Protection |
| --- | --- | --- | --- | --- |
| Code 00 | 32 | 2 | 1 | Read-execute |
| Heap 01 | 34 | 2 | 1 | Read-write |
| Stack 10 | 28 | 2 | 0(28~26) | Read-write |

Ex 0x2C00 is in stack(2) = 3 KB, offset is distance from 8KB = 0xC00

Stack virtual address 8KB ~ 12KB - 1 <-> physical address 4KB ~ 5KB - 1

Mapping 12KB - 1 to 5KB - 1

12KB - 1 - 8KB - offset = 5KB - 1 - X

X = 1KB + offset = 1KB + 0xC00 = 4KB

| 8KB | Offset | Distance to 12 KB | 12KB - 1 |
| --- | --- | --- | --- |
| … 4KB | … | Distance to 5KB | 5KB - 1 |

16.5 Fine-grained vs. Coarse-grained Segmentation

Coarse-grained : chops up the add space into relatively large, coarse chunks

Fined-grained : flexible, allows add space to consist a large number of smaller segments

Hardware support segment table (creation of a large number of segments = > enable system to use segments in more flexible ways)

16.6 OS Support

Segment registers must be saved and restored when OS do context switch

How to manage physical memory if segment with different size?

External fragmentation : free memory space between allocated mem = > hard to allocate new segments/grow existing ones

Solution

Compact physical memory by rearranging the existing segments = > new allocation request --- expensive

Ex OS stop running processes, copy the data to be contiguous mem, change their segment register values to new location

Use free-list management algorithm

Best fit (returns the mem with closest size of desired allocation), worst fit, first fit, buddy algorithm

Ex

0x0010 movl 0x1100, %r8d

0x0014 addl $0x3, %r8d

0x0017 movl %r8d, 0x1100

| code 0 | 16KB(0x4000) | 4KB(0x1000) | pos |
| --- | --- | --- | --- |
| heap 1 | 22KB(0x5800) | 4KB | pos |
| stack2 | 28KB(0x7000) | 2KB | neg |

it will access mem 5 times = 3 to load instruction + 2 to access data

fetch instruction at 0x4010 -> load from 0x6900

fetch instruction at 0x4014 (CPU do the addition)

fetch instruction at 0x4017 -> store to 0x6900

0x0010 -> 0x4010

0x1100 -> 0x5800+0x0100 = 0x5900

0x2100 in 8KB ~ 12KB is out of the range of 26KB ~ 28KB = > exception = > stack may grow to 4KB and its base changes to 30KB

| stack2 | 30KB(0x7800) | 4KB | neg |
| --- | --- | --- | --- |

physical address = 26KB + 0x0100 = 0x6900

stack11, bits in address = 16 = > starting virtual add = 0x11 + (0)14, end virtual add = 0x11 + (1)14 + 0x1

## 17 Paging - space divided into fixed-sized units

More diffi. Case - variable-sized units ex malloc and free = > external fragmentation

17.1 Assumptions

Basic interface (malloc, free) - library must know the size when free the pointer

free list in heap : generic data structure used to manage free space. Not be a list per se, but data structure to track free space

Internal fragmentation - hands out more memory space to allocate

External fragmentation - free space between allocated memory

No reallocation after allocation

No compaction of free space is possible, useful to combat fragmentation

Compaction used in OS to deal with fragmentation when implementing segmentation

Allocator manages a contiguous region of bytes = > ask for the region to grow

Ex user-level memory-allocation lib might call kernel to grow the heap(sbrk).

17.2 Low-Level Mechanism

Splitting and Coalescing : common tech in most any allocator

Splitting : when request < free memory = > split it into two to match the request. First chuck return to the caller, second remains on the list

Coalescing : order address when free memory and merge to larger free memory

*Ex free list : request > 10 = > fail or NULL; request = 10 either;*

| *0~10* | *~20* | *~30* |  | *Head->* | *Add. 0->* | *Add. 20->* | *NULL* |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Free* | *Used* | *free* |  |  | *Len 10* | *Len 10* |  |

*request < 10 splitting ex 1 byte and use the second element on the list*

*Head -> add:0, len:10 -> add:21, len:9 -> NULL* (len 9^3??)

How the memory looks like in reality? Chunk? Or any length?

*free(10) : head -> add:0, len:20 -> add:21, len:9 -> NULL*

Tracking the Size of Allocated Regions

free() needs to know the size -> extra infor. Stored in a header block in mem(before the handed-out chunk of mem)

Size of the free region = size of the header + size of the allocated space

Ex allocated block of size 20 bytes pointed to by ptr.

| Header used by malloc | 20 bytes returned to caller |
| --- | --- |
| htr points to the start  Size: 20, magic: 1234 | ptr points to the start |

Header contains the size :

typedef struct \_\_header\_t {

int size;

int magic; #additional integrity checking and other infor.

} header\_t;

When call free (assert(hptr->magic == 1234) )

void free(void \*ptr) {

header\_t \*hptr = (void \*)ptr - sizeof(header\_t);

...

Embedding a Free List

Node of the list

typedef struct \_\_node\_t {

int size;

struct \_\_node\_t \*next;

} node\_t;

Initialize heap and first element of the free list (mmap returns pointer to a free chunk)

node\_t \*head = mmap(NULL, 4096, PROT\_READ|PROT\_WRITE,

MAP\_ANON|MAP\_PRIVATE, -1, 0);

head->size = 4096 - sizeof(node\_t);

head->next = NULL;

Heap with one free chunk

The free list is usually after the allocated memory

Coalesce the list, merge the free space, add to the free list

Growing the Heap

Runs out of space = > fail and return NULL OR sbrk to grow the heap

OS finds free physical pages, map them into add. space, return value of new heap end

17.3 Basic Strategies for managing free space (fast and less fragmentation)

Best Fit

Larger smallest size in the free list -> less fragmentation

But not fast < = exhaustive search for correct free block

Worst Fit

Largest chunk and return = > keep the remaining chunk on the free list

Leave big chunks free instead of small chunks (by best-fit)

But costly to full search and excess fragmentation and high overheads

First Fit

First block that is enough = > fast, but pollutes big mem with small objects

Address-based ordering(keeps the list ordered by add. of free space) can solve = > easier coalescing, reduced fragmentation

Next Fit

Keeps an extra pointer to the location in the list where one was looking last

Spread the searches for free space throughout the list and avoid splintering of the beginning of the list = > fast

Examples

Head -> 10 -> 30 -> 20 -> NULL

| Best fit | Head -> 10 -> 30 -> 5(20-15) -> NULL |
| --- | --- |
| Worst/first fit | Head -> 10 -> 15 -> 20 -> NULL |

17.4 Other Approaches

Segregated list to manage objects if particular application has one/few popular-sized request

Having a chunk of mem dedicated = > less fragmentation and quick search

How much mem to dedicate?

Slab allocator keeps free objects on the lists in a pre-initialized state

kernel boots up, it allocates a number of object caches for kernel objects that are likely to be requested frequently -> object caches are each segregated free lists

When a cache is running low on free space, it requests some slabs of mem from a more general mem allocator

Buddy Allocation to make coalescing simple ex binary buddy allocator

Free memory first has size 2N. Divide the mem until it the smallest to return.

Merge level by level when free mem = > restore entire space / stop when pair is in use

Address only differs by a single bit

Internal fragmentation

## 18 Segmentation in VM to chop things up into variable-sized pieces

Dividing space into diff. chunks can be fragmented

Paging chop up space into fixed-sized pieces.

Physical mem as an array is page frames. Each frame contain a single VM page

physical frame and virtual page. page table is a mapping between PF and VP

page table entry

high bits = > number of pages, low bit for offset = > page size

How many low bits to have page size 16 byte? 4

How many pages to have with 10 wide bits and its size is 16 byte? 64=26

translation steps

extract VPN(virtual page number) from VA

calculate add of PTE(page table entry)

PTE \* VPN

fetch PTE (mem, others are in registers or CPU)

extract PFN(physical frame number)

build PA

fetch PA to register (mem)

Ex. 4KB -> 12 bytes (2^12),

0x0010. first 0 -> load form 0x5000, second 0 -> 0x2000 + offset

0x1100. first 1 load from 0x5000+4, load from 0x0100

0x2..., 0x5000+offset

Ex TLB has eq or more mem space.

18.1 Example and Overview

The pages of the virtual add space are placed at diff locations in physical mem, OS also need some physical mem for itself

Pros :

Flexibility : support abstraction of add. Space effectively, regardless how a process uses the add. Space ex no assumption about the direction heap/stack(why)

Simplicity : ex OS wants 64bytes, it finds first four pages in a free list

| 0~16 | ~32 | ~48 | ~64 | ~80 | ~96 | ~112 | ~128 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| OS | Unused | Page 3 of AS | P.0 of AS | Unused | P.2 of AS | Unused | P.1 of AS |
| PF 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

PF stands for page frame of physical memory

Page table : pre-process(Inverted page table is an exception) data structure to store address translations = > know where in physical memory each page resides

Ex page table of the last example VP = virtual page, PF = page grame

(VP 0 -> PF 3), (VP 1 -> PF 7), (VP 2 -> PF 5), (VP 3 -> PF2)

VPN(virtual page number) select the page

Offset tells which byte of the page we are interested in (< = pages length)

Ex 10 bit = 1 KB, 12 bit = 4 KB

Ex address translation (instruction fetch already happened)

movl <virtual address>, %eax to load data from VA to eax register

6 bits VA = 26 = 64, 4 page = 2 bit VPN + 4 bit offset

If VA = 21 = 010101, 0101 = 101 5th byte of VP 01, offset remains

Physical address 1 1 10101

PFN or PPN(page frame / physical page number) in page table

18.2 Where are Page Tables Stored?

PTE(page table entry) holds physical translation + any other(?)

Since page tables are big, no special on-chip hardware in MMU(mem management unit) to store the page table of currently running process. Instead, storing page table for each process in mem.

Page table can be stored in OS virtual mem, but just assume it's in physical mem.

| PFN |  | G | PAT | D | A | … |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | PCD | PWT | U/S | R/W | P |

18.3 What's Actually in Page Table (stored in the physical add.)?

Page table maps virtual add(VPN) to physical add(PFN)

Linear page table : array with index(VPN) to search PTE

In PTE, valid bit indicates if the translation is valid

Mark all unused page in the address pace invalid = > remove the need to allocate physical frames for pages = > save mem

*ex code/heap at one end, stack at the other. Unused space in between are invalid = > generates trap to OS*

Protection bit = if the page could be r/w/executed

Reference bit = track if a page has been accessed = > popular? Keep in mem

Others : present bit = if the page is in physical mem or disk(it has been swapped out); dirty bit = if the page has been modified after brought into mem.

In the last table, Present; Read/Write; User/Supervisor(if user-mode process can access) PWT,PDC, PAT, G(how hardware caching works for pages), Access and Dirty.

18.4 Paging : Also Too Slow

Example: movl 21, %eax

Fetch proper page table entry from process's page table -> translate virtual add to physical add. -> load/fetch data from physical mem.

Page-table base register contains physical add of starting location of the PT

VPN = (VirtualAddress & VPN\_MASK(0x30=110000)) >> SHIFT(=4)

PTEAddr = PageTableBaseRegister + (VPN \* sizeof(PTE))

Ex VA=21 (010101), VPN=01 as index in the array of PTE (pointed by page table base register)

Hardware fetch PTE from mem after knowing physical add.

PTE = AccessMemory(PTEAddr)

Check if process can access the page

if (PTE.Valid == False)

RaiseException(SEGMENTATION\_FAULT)

else if (CanAccess(PTE.ProtectBits) == False)

RaiseException(PROTECTION\_FAULT)

else…

Extract PFN and concatenate it with offset from the virtual add.

offset = VirtualAddress & OFFSET\_MASK

PhysAddr = (PTE.PFN << PFN\_SHIFT) | offset

Register = AccessMemory(PhysAddr)

Hardware fetch the data from mem and put it into register eax

Extra mem reference to first fetch the translation from the page table = > costly

18.5 A Memory Trace

One page table/process in the system, determined by hardware or OS(modern)

int array[1000]; for(i = 0; i < 1000; i++) {array[i] = 0;}

| 1024 movl $0x0,(%edi,%eax,4) | Moves the value 0 into VMA of the location of the array  The add = %edi \* %eax(holds array index(i) \* 4(int size) |
| --- | --- |
| 1028 incl %eax | Increments array index in eax |
| 1032 cmpl $0x03e8,%eax | Compares content in 0x03e8(1000). != four line |
| 1036 jne 0x1024 | Jumps back to the top of the loop |

4 mem access to fetch, eax has 3, edi has 1, others has 2

Ex assume virtual add space size = 64KB, page size = 1KB

Virtual memory

There is the virtual page the code lives on

Page size = 1, virtual ad = 1024 on the 2nd page of VA space (VPN = 1) VPN 1 -> PFN 4

There is the array. Size=4000(1000 ints). Assume 40000~44000 VA

VPN = 39…VPN=42 and needs to map for these pages

Trace mem reference of the program. 1 instruction fetch generates 2 mem ref:

1. To page table to find physical frame
2. To instruction to fetch it to CPU

One explicit mem ref. in mov instruction to add another page table access(to translate virtual add to physical) and then array access

Diagram

PageTable [1] and pagetable[39]??

the index increase by 4 for every index increment

the instruction to fetch is always this 4

## 19 TLB : translation-lookaside buffer, add-translation cache, hardware cache of popular

store whole key/value in cache (VPN, PFN)

valid bit (if the entry has a valid translation) difference between it and the Page Table Entry

set the valid bit to 0 initially

context switch

protection bit ()

V2P address translation

Hardware first check TLB if translation is held therein = > no consult page table

Hardware handles TLB miss

19.1 TLB Basic Algorithm - how hardware handle translation?

Linear page table as an array, hardware-managed TLB handles page table access

VPN = (VirtualAddress & VPN\_MASK) >> SHIFT //extract VPN from VA

(Success, TlbEntry) = TLB\_Lookup(VPN) //TLB holds translation? Y has TLB hit and success

if (Success == True) // TLB Hit

if (CanAccess(TlbEntry.ProtectBits) == True) //extract PFN from TLB

Offset = VirtualAddress & OFFSET\_MASK//concatenate onto offset

PhysAddr = (TlbEntry.PFN << SHIFT) | Offset //form PA

Register = AccessMemory(PhysAddr) // access mem

else

RaiseException(PROTECTION\_FAULT)

else //TLB miss - CPU not find translation in TLB

PTEAddr = PTBR + (VPN \* sizeof(PTE))

PTE = AccessMemory(PTEAddr) // access page table to find translation

if (PTE.Valid == False)

RaiseException(SEGMENTATION\_FAULT)

else if (CanAccess(PTE.ProtectBits) == False)

RaiseException(PROTECTION\_FAULT)

else //valid and accessible VM ref

TLB\_Insert(VPN, PTE.PFN, PTE.ProtectBits) //updates TLB translation COSTLY

RetryInstruction() //retry instruction

TLB misses = > more mem accesses and cost more

19.2 Ex. Accessing An Array

Spatial locality

int a[10]; //4 bytes \* 10, VA=100, 8-bit VA space, 16-byte=2-bit pages

VA breaks down into a 4-bit VPN(16 V pages) and 4-bit offset(16 bytes)

for(i = 0 ~ 10) { sum += a[i]; }

a[0], a[3], a[7] TLB miss : CPU see a load to VA 100 -> hardware extract VPN(6) and use it to check TLB.

others TLB hit : translation already loaded into TLB

hit% = 70% higher is better

page size bigger may also improve the performance

Temporal locality : quick re-refer of mem item in time

TIP

caching, one of mot fundamental performance tech, used to make common cast fast.

take advan. of locality in instruction and data ref.

temporal locality : access and re-access soon *ex loop*

spatial locality : access mem near accessed mem

19.3 Who handles TLB Miss?

Hardware : Knows page table location through page-table base register(PTEAddr = PTBR + (VPN \* sizeof(PTE))) and exact format

walk the page table to find the entry -> extract the translation -> update TLB and retry

Ex multi-level table

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

RaiseException(TLB\_MISS)

Software-managed TLB : (OS)

hardware raise exception and pause current instruction stream -> kernel mode privilege level and jumps to trap handler(code in OS to handle TLB miss)

special "privileged" instruction to update TLB

return from trap

diff from system call(resume execution at the instruction after trap into OS), resume execution at the instruction that caused the trap(so that it will can retry) -> TLB hit

retry instruction

OS careful about not causing infinite chain of TLB misses

(if OS doesn't know the location of Miss Handler code, OS use TLB to search it, fail -> infinite loop)

keep TLB miss handler in physical mem(unmapped to add. translation)

reserve entries in TLB for permanently-valid translation and use it for handler

Pros

flexible : any data structure to implement page table without necessitating hardware change

simplicity : hardware not do much on a miss

TIP

CISC(complex instruction set computing) camp : many instructions

Ex string copy needs 2 pointers + length + copy bytes from source to destination. CISC high-level primitives to make assembly language easy to use, code more compact

RICS(reduced instruction) : faster

instruction sets are compiler targets. Compiler want few simple primitives to generate high-performance code. simple, uniform and fast

19.4 TLB Contents

Has fully associative: 32/64/128 entries (VPN | PFN | other bits)

Search entire TLB in parallel to find translation

TIP : TLB valid bit != Page table valid bit

PTE invalid : page not allocated by process OR accessed by abnormal process. Usual response is to trap to OS and kill process

TLB valid = TLB entry has a valid translation

Ex initial state for each ELB entry = invalid since no add. translations are cached

also used in context switch to ensure about-to-run process not use V2P translation from a previous process

19.5 TLB Issue : Context Switches

V2P translation only valid for current process -> to-run process not run translation from previously run process

Ex TLB content when..

P1 : valid TLB and 10th virtual page of P1 is mapped to physical frame 100, P2 : 10th virtual page of P2 is mapped to physical frame 170

| VPN | PFN | Valid | Prot |
| --- | --- | --- | --- |
| 10 | 100 | 1 | rwx |
| \_ | \_ | 0 | \_ |
| 10 | 170 | 1 | rwx |

how hardware distinguish VPN entry? Solution in the following

Flush TLB on context switch :

Software-based by an explicit/privileged hardware instruction

Hardware-managed TLB : flush enacted when page-table base register is changed (OS must change PTBR on context switch)

Problem : costly <= miss when restart

To reduce the cost - Hardware support

ASID(address space identifier) as a PID(process identifier) but fewer bits

| VPN | PFN | Valid | Prot | ASID |
| --- | --- | --- | --- | --- |
| 10 | 100 | 1 | rwx | 1 |
| \_ | \_ | 0 | \_ | \_ |
| 50 | 100 | 1 | rwx | 2 |

TLB holds translations from diff processes without confusion

OS set some privileged register to ASID of current process on context switch

when diff VPN and same PFN(same physical page)...

P1 maps the page into 10th page, P2 into 50th page of the add. space

sharing code page is useful as it reduces physical page # = > mem down

19.6 Issue : Replacement Policy

Cache replacement : which old entry to replace when installing a new entry in the TLB? goal is to minimize miss rate and increase hit rate

Solution

evict LRU(least-recently-used) entry

evict TLB mapping at random (random policy) : simple and avoid corner-case

reasonable policy like LRU behaves unreasonably when a program loops over n+1 pages with TLB size = n

19.7 A Real TLB Entry - software-managed

Ex MIPS

Why we need VPN and PFN?

any PTE can go to any entry in TLB

PFN

32-bit address space with 4KB pages => 20-bit VPN and 12-bit offset

TLB only 19 bits for VPN

user addresses only come from half the address space(rest reserved for kernel)

VPN translates up to 24-bit PFN => support system up to 64 physical mem (224 4KB pages)

MIPS TLB components

global G used for pages globally shared among processes

ASID is ignored if G is set (it will be cycled)

Ex

| valid | VPN | PFN | ASID |
| --- | --- | --- | --- |
| 0 |  |  |  |
| 1 | 1 | 1 | 11 |
| 1 | 1 | 4 | 12 |
| 1 | 0 | 3 | 11 |

Q : more than 256 processes running?

Coherence bit C determines how page is cached by hardware

Dirty D marked when the page has been written to

Valid V tells hardware if there is a valid translation present in the entry

Page mask field not shown, supports multiple page sizes

TLB has slots reserved for OS.

Wired register set by OS to tell hardware how many slots to reserve for OS (use for code and data)

Instruction to update and manage TLB (privilefed)

TLBP probes TLB to see if particular translation is there

TLBR reads content of TLB entry into register

TLBWI replaces a specific TLB entry

TLBWR replaces random TLB entry

TIP : RAM isn't always RAM

RAM(random-access mem) implies you can access any part of RAM

Randomly accessing address space, particular if pages # accessed exceeds TLB coverage, leads to severe performance penalty

## 20 Paging Strategies

table is too big = > much mem

32-bit add. space(232), 4KB(212) pages and a 4-byte page-table entry -> 220 virtual pages

PTE valid - data stored in it, PDE valid - there is at least one data stored in this page

20.1 Simple Solution : Bigger Pages

reduce the size of the page table => some problems:

big pages waste within each page = > internal fragmentation

20.2 Hybrid Approach: Paging and Segments

Base/Bound pairs

base - PA of the PT (not each segment), bound - end of the PT

three base bound pairs for code, heap and stack.

BB contains PA of linear page table of the segment = > each process has 3 page tables

Segments over PTs

split VPN to top four bit (1/2 for h) segment + 1 bit index + VPN

Seg code + PT Index + offset

TLB miss ->

HW uses SN(segment bit = (VA & segm\_mask) >> SN\_shift) to determines which BB register to use

VPN = (VA & VPN\_mask) >> VPN\_shift

A of PTE = Base[SN] + VPN \* sizeof(PTE)

Problem:

assume certain usage pattern of the A space ex. large but sparsely-used heap

external fragmentation

20.3 Multi-level Page Tables (PTE indexed by VPN)

Page directory (1st level) tells the page of a page table or no valid pages

PDE has a valid bit & PFN(PTE)

valid bit = at least 1 page in this PT is valid

Pros

only allocates using A space and PT

OS can use next free mem and grow PT

Cost

Time-space tradeoffs: one more load from mem to get the right translation

Complexity

Ex.

extract PDIndex (Page directory index) from the top of VPN to find PDEaddr = PageDirBase + PDIndex \* sizeof(PDE). -> invalid => exception

Valid: left VPN is PTIndex(page table index) -> get PTEaddr = (PDE.PFN << Shift) + (PTIndex \* sizeof(PTE) ) PFN is shift PDE to right place

Multi-level Control Flow

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

// first, get page directory entry

PDIndex = (VPN & PD\_MASK) >> PD\_SHIFT

PDEAddr = PDBR + (PDIndex \* sizeof(PDE))

PDE = **AccessMemory(PDEAddr)**

if (PDE.Valid == False)

RaiseException(SEGMENTATION\_FAULT)

else

// PDE is valid: now fetch PTE from page table

PTIndex = (VPN & PT\_MASK) >> PT\_SHIFT

PTEAddr = (PDE.PFN << SHIFT) + (PTIndex \* sizeof(PTE))

PTE = **AccessMemory(PTEAddr)**

if (PTE.Valid == False)

RaiseException(SEGMENTATION\_FAULT)

else if (CanAccess(PTE.ProtectBits) == False)

RaiseException(PROTECTION\_FAULT)

else

TLB\_Insert(VPN, PTE.PFN, PTE.ProtectBits)

RetryInstruction()

20.4 Inverted Page Table

only one table for each physical page of the system (entry tells us which entry is using this page)

20.5 Swapping to disk

Kernel virtual mem, allowing system to swap page table to disk

## 21 Swapping

* Intro
* swap space = 2 \* size of mem usually
* Hard disk drive - additional level in mem hierarchy to run more processes not fitting in mem = SLOW
* Q why larger add space?
* convenience and ease of use: no worry about program size
* Memory Overlays requires programmer to manually move pieces of code/data in/out of mem as they were needed (arrange for code/data to be in mem before a function call)
* Why important?
* Swap space allows OS to support illusion of a large VM
* Multi-programming demanded swapping out some pages
* 21.1 Swap Space
* Def. reserve space on disk for moving pages(OS needs to remember its address)
* Size of swap space = > max # of mem pages to be used by a system
* Ex. program binary
* Code page are initially found on disk, and loaded into mem when runs
* swap it to disk after use if needed
* 21.2 The Present Bit
* Assumption : hardware-managed TLB (translate to PA before fetching desired data)
* Extract VPN from VA, check TLB...
* (1)hit: produces PA and fetch it from mem;
* 2)miss: HW locates page table in mem with PT base register, looks up PTE with VPN as index (present bit = 1), valid -> extracts PFN from PTE to TLB -> retries instruction
* Swap adds more machinery...
* HW looks in PTE, not in P mem (present bit = 0), search in disk.
* Page fault (actually page miss): accessing a page that is not in P mem
* OS invokes to service page fault using page-fault handler
* 21.3 The Page Fault
* OS received a page fault -> looks in PTE to find address -> swap the page into mem
* After disk I/O(process blocked), OS updates PT to mark the page present, update PFN field of PTE(record in-mem location) -> retry instruction
* TLB miss -> retry instruction
  + I/O is expensive, overlap of I/O of one process and execution of another is another is a way a multi-programmed system can make the most effective use of HW
* 21.4 What If Memory is Full?
* Page-replacement policy : replace/page out one/more pages to make room for new page
* 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
* 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)
* 21.5 Page Fault Control Flow
* Present & valid -> miss handler grab PFN from PTE, retry
* Page fault hander swap out some page if need, and swap page into mem
* Invalid page -> trap this invalid access -> OS trap handler
* PFN = FindFreePhysicalPage()
* if (PFN == -1) //no free page
* PFN = EvictPage() //replacement
* **DiskRead(PTE.DiskAddr, pfn)** //wait - blocked
* PTE.present = True //update the PT with present bit
* PTE.PFN = PFN //and translate PFN
* RetryInstruction()
* 21.6 When Replacements Occur
* HW(high watermark) and LW(low watermark) helps decide when to start evicting page
* OS notices fewer than LW pages available -> background thread to free mem
* Thread evicts pages until there are HW pages available
* Swap/page daemon = background thread goes to sleep
* Performance optimization:
* systems clusters/groups pages and write them out at once -> disk efficiency up
* Control Flow changes
* Not replacement directly. Instead, no mem available -> inform background thread to free pages