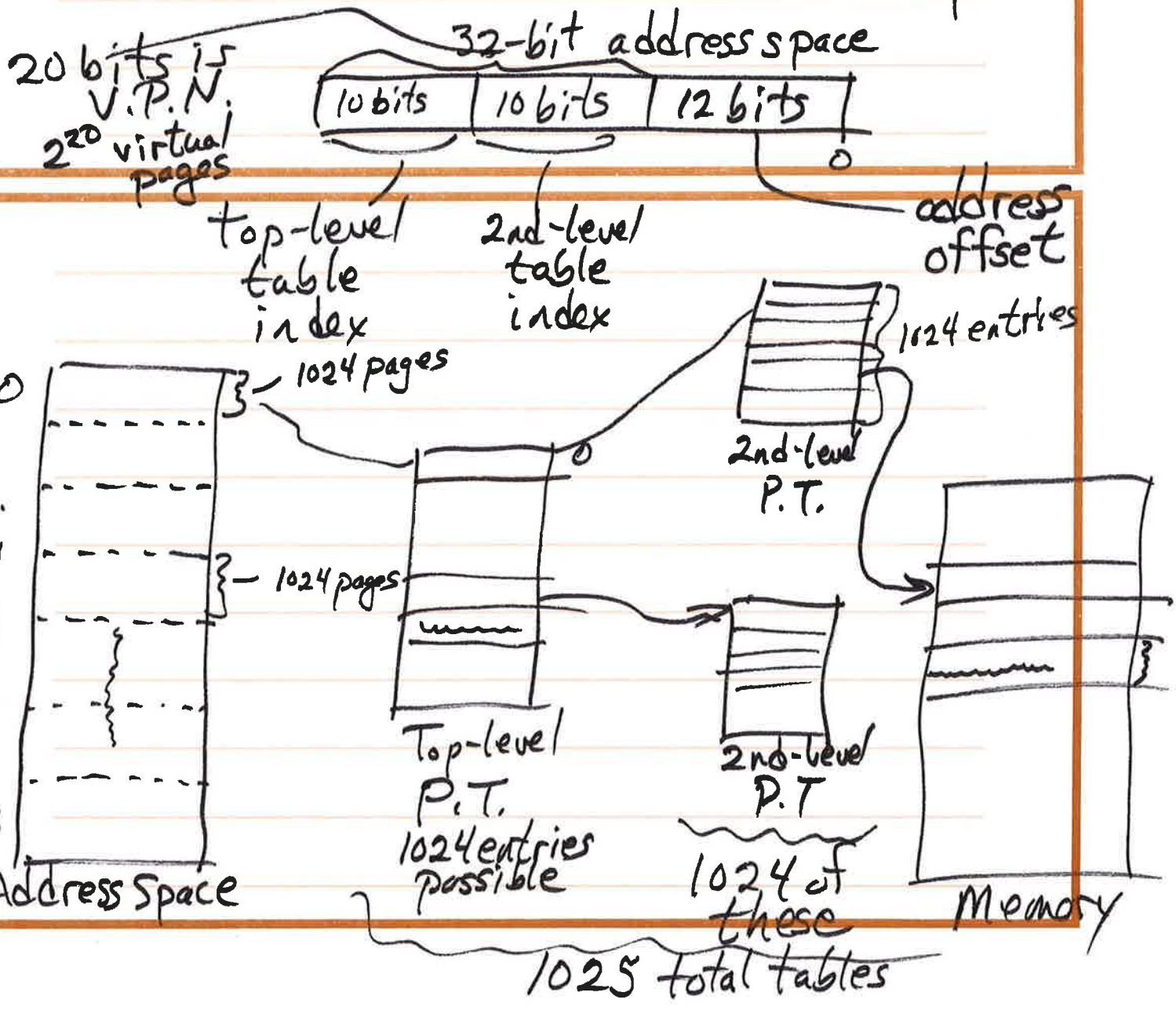
10/13/10

Multi-level Page Translation

Lowest-level table, which points to the physical page where the virtual page is located, is always a page table

Example: 2-level paging

Divide the V.A. into 3 parts

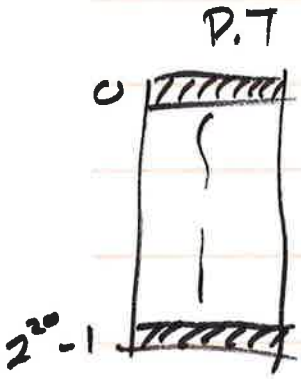


How much do we save in unused page table entries?

Worst Case: Single-Level Paging

20 bits for V.P.N.

12 bits for offset



2^{20} entries in P.T.

> 1 million

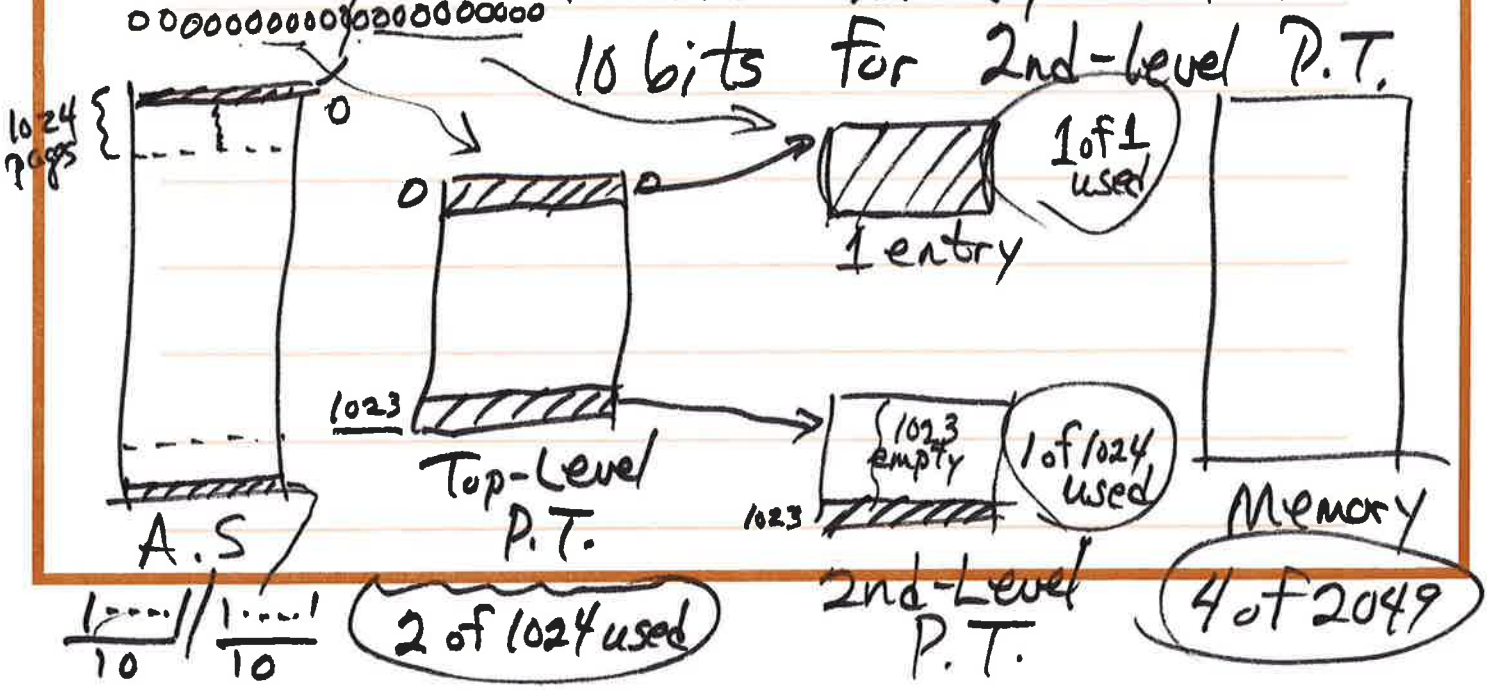
2 entries used

$2^{20}-2$ unused entries in P.T.

2-Level Paging

10 bits for top-level P.T.

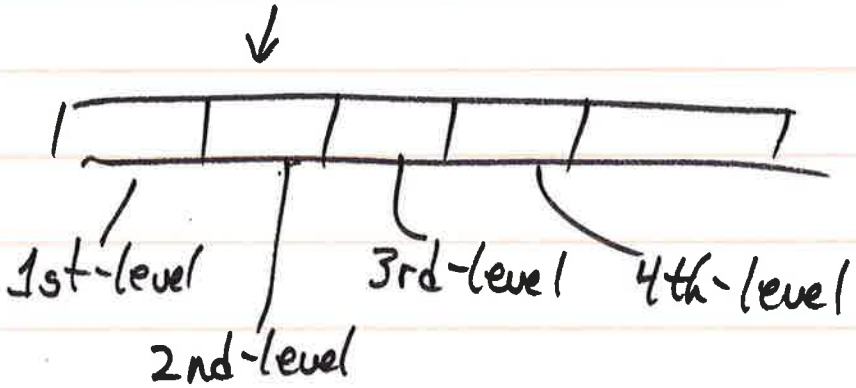
10 bits for 2nd-level P.T.



In real OSes:

32-bit: 3 or 4 translation levels

64-bit: 4 translation levels



But... We have a new problem

Example: 4-level translation

MMU

1. V.A. received & validated
2. MMU accesses memory for 1st-level P.T.
3. " " " " 2nd-level P.T.
4. " " " " 3rd-level P.T.
5. " " " " 4th-level P.T.
6. " " " for user V.A.

5 memory accesses for 1 user program address

Translation Lookaside Buffer (TLB)

Cache of page table entries

Maps virtual pages to physical page
- Like a page table

The V.P.N. is the "Key"

Contains recently accessed V.P.s & their physical page location

MMU Process

1. Receive & validate V.A.
2. Compute the V.P.N.
3. Use VPN to lookup in the TLB
   - If found, directly compute physical address- done
4. If not found in TLB, go to the translation tables
   - Add this entry to the TLB

Issue: What happens on a context switch?

2 Choices

① Invalidate the TLB

② Save the TLB

Issue: Real OSes, when they get busy, don't have enough memory

The OS must manage memory—it can be full

↓

Global Page replacement

↳ the OS can remove a page from any process

Virtual Memory Worst Case Scenario

A virtual address, needed by a user program - the VPN of this address is not in the TLB

We go to the translation tables to find the V.P. location

- Part of the needed T.T. are not in memory - "swapped" to disk

- We load in the T.T. we need
- We find that the virtual page, needed by user program, was in memory.

GOAL: We NEVER want to have to load a kernel data structure from disk, when the V.P., that is needed, is already in memory

Idea: Kernel needs some data structure that maps physical memory - independent of user programs

Maps physical pages to virtual pages

Inverted Page Table - IPT

- Only 1 for O.S.

- Always in memory
- 1 entry for each physical memory page
- Managed by kernel
- "Indexed" by physical page #

An issue:

We start (in mmu) with a Virtual Address. We compute the VPN. If not in TLB, we look in IPT. But, the IPT is "indexed" by physical page #

In the real OSes, the IPT is a hash table

- Key: V.P.N. & Process ID
- Lookups are

IPT contains @ least:

- physical page #
- virtual page #
- process id
- valid bit
- dirty bit - when true, physical page has been modified

MMU Process

1. Receive & validate V.A.
2. Check TLB
   If found, compute P.A. – done
3. If not in TLB, look in IPT
   If found, compute P.A, update the TLB – done
4. If not in IPT, go to the translation tables (needed V.P. is not in memory)
   a. Load needed page into an available page of memory
   b. Update IPT, TLB
   c. Updation translation table(s)
   d. Restart the user instruction

Page Fault

Assume plenty of memory

Demand Paged Virtual Memory

Don't preload any virtual pages into memory— wait for a page fault

What if physical memory is full on a page fault?

- I must evict a page of memory
  - What if the selected page is dirty?
    - We must save this dirty page to disk

MMU Process

1. Receive & validate V.A.
2. Check TLB
3. If not in TLB, check IPT
4. If not not in IPT, check translation tables
   a. To "look" in our T.T., we use the V.P.N. to find the appropriate entry
   b. Read that entry to find the "disk" location
   c. Copy this page (if on disk) to a physical memory page
      - if none available, evict one [Don't worry about how to pick one]
      - For the selected physical page:
        - Is it dirty?
          Yes- save it to disk
        - Update the translation table

d. Now we can load in the needed virtual page - into the newly available physical memory page.

e. Update the translation table for the newly loaded page of the current process

f. Update IPT & TLB for the newly loaded page

g. The evicted page, if it is in my process, might be present in the TLB. If so, invalidate that TLB entry.

h. Restart the user instruction

2 Remaining Issues

1. Which page of physical memory to replace?

2. When we evict a dirty page, where do we put it on disk?

Project 3- Parts 1 & 2

A Suggested Approach

You compile/run from vm directory
UNTIL Part 3 is ready
↳ network directory

vm directory is setup to use the Nachos TLB

- Nachos will no longer use machine->pageTable
- Only use machine->TLB

MUST DO: comment out the line in RestoreState

machine->pageTable=pageTable

On a TLB miss, Nachos generates a PageFaultException

Step 1: Leave preloading code in addrspace constructor
- Still lots of memory

Proj 2.

Nachos will only use the TLB
- TLB misses → PageFaultException

Use page table to populate the TLB

The register containing the faulting virtual address: BadVAddrReg 39

Divide that value by PageSize
- that's the VPN - page table index
- Read the physical page #

Copy all the values from that page table entry into a TLB entry

The TLB is an array of TranslationEntry

How to "find" a good slot in TLB?

- Nachos TLB is size 4 - TLBSize
- use FIFO
- a global counter - currentTLB
- When adding an entry to the TLB

currentTLB = (currentTLB++) % TLBSize;

index position in TLB

On a context switch

If you test with Exec, you must invalidate the TLB

- set all valid bits false

Context switches occur with interrupts off

Do not use a lock to control access to the TLB

- use interrupts to control TLB access - disable/restore

Result: All project 2 user program will run