Virtual Memory — Paging II

ICS332 — Operating Systems

Henri Casanova (henric@hawaii.edu)

Spring 2018

Paging is great but...

- The previous set of lecture notes ends with all the benefits of paging

Paging is great but...

- The previous set of lecture notes ends with all the benefits of paging
- But there are some challenges / problems
- Two big problems:
- **Problem #1:** Paging has extra overhead
- **Problem #2:** Page tables can be very large
- Let's understand these problems and come up with solutions

Paging is great, but it's expensive :(

- Each address coming out of the CPU is virtual
- Address translation (from virtual to physical) has to be performed for **EVERY** address issued by the CPU

Paging is great, but it's expensive :(

- Each address coming out of the CPU is virtual
- Address translation (from virtual to physical) has to be performed for **EVERY** address issued by the CPU

- Assume that the page size is 4 KiB

Paging is great, but it's expensive :(

- Each address coming out of the CPU is virtual
- Address translation (from virtual to physical) has to be performed for **EVERY** address issued by the CPU

- Assume that the page size is 4 KiB
- Assume that the address space is 8 MiB

Paging is great, but it's expensive :(

- Each address coming out of the CPU is virtual
- Address translation (from virtual to physical) has to be performed for **EVERY** address issued by the CPU

- Assume that the page size is 4 KiB
- Assume that the address space is 8 MiB
- Then the page table needs to hold $2^{23}/2^{12} = 2^{11} = 2048$ entries

Paging is great, but it's expensive :(

- Each address coming out of the CPU is virtual
- Address translation (from virtual to physical) has to be performed for **EVERY** address issued by the CPU

- Assume that the page size is 4 KiB
- Assume that the address space is 8 MiB
- Then the page table needs to hold $2^{23}/2^{12} = 2^{11} = 2048$ entries

- The page table is in RAM, and it will be accessed very frequently
- When a new process is dispatched to the CPU, the dispatcher loads a special register with the address of the beginning of the process's page table: the Page Table Base Register (PTBR)

Paging is great, but it's expensive :(

- Each address coming out of the CPU is virtual
- Address translation (from virtual to physical) has to be performed for **EVERY** address issued by the CPU

- Assume that the page size is 4 KiB
- Assume that the address space is 8 MiB
- Then the page table needs to hold $2^{23}/2^{12} = 2^{11} = 2048$ entries

- The page table is in RAM, and it will be accessed very frequently
- When a new process is dispatched to the CPU, the dispatcher loads a special register with the address of the beginning of the process's page table: the Page Table Base Register (PTBR)
- This makes it fast to switch between page tables at each context switch

Paging is great, but it's expensive :(

- Each address coming out of the CPU is virtual
- Address translation (from virtual to physical) has to be performed for **EVERY** address issued by the CPU

- Assume that the page size is 4 KiB
- Assume that the address space is 8 MiB
- Then the page table needs to hold $2^{23}/2^{12} = 2^{11} = 2048$ entries

- The page table is in RAM, and it will be accessed very frequently
- When a new process is dispatched to the CPU, the dispatcher loads a special register with the address of the beginning of the process's page table: the Page Table Base Register (PTBR)
- This makes it fast to switch between page tables at each context switch
- This does not speed up translation though
- In fact, the memory access time is doubled: 1) Access an entry in the page table; 2) Based on that entry access the physical address

Paging is great, but it's expensive :(

- Each address coming out of the CPU is virtual
- Address translation (from virtual to physical) has to be performed for **EVERY** address issued by the CPU

- Assume that the page size is 4 KiB
- Assume that the address space is 8 MiB
- Then the page table needs to hold $2^{23}/2^{12} = 2^{11} = 2048$ entries

- The page table is in RAM, and it will be accessed very frequently
- When a new process is dispatched to the CPU, the dispatcher loads a special register with the address of the beginning of the process's page table: the Page Table Base Register (PTBR)
- This makes it fast to switch between page tables at each context switch
- This does not speed up translation though
- In fact, the memory access time is doubled: 1) Access an entry in the page table; 2) Based on that entry access the physical address
- We just made our RAM twice as slow :(

Locality and Caching

However!

- Temporal locality: repeated access to the same memory location

Locality and Caching

However!

- Temporal locality: repeated access to the same memory location
  e.g., `counter += f(1, alpha, beta)`
  $\Rightarrow$ `counter` is accessed over and over
  $\Rightarrow$ the same frame is accessed over and over

Locality and Caching

However!

- Temporal locality: repeated access to the same memory location
  e.g., `counter += f(1, alpha, beta)`
  $\Rightarrow$ `counter` is accessed over and over
  $\Rightarrow$ the same frame is accessed over and over
- Spatial locality: repeated access to nearby memory locations

Locality and Caching

However!

- Temporal locality: repeated access to the same memory location
  e.g., `counter += f(1, alpha, beta)`
  $\Rightarrow$ `counter` is accessed over and over
  $\Rightarrow$ the same frame is accessed over and over
- Spatial locality: repeated access to nearby memory locations
  e.g., `a[i] = a[i-1] + a[i-2]`
  $\Rightarrow$ all three array elements are very likely in the same frame
  $\Rightarrow$ the same frame is accessed over and over

Locality and Caching

However!

- Temporal locality: repeated access to the same memory location
  e.g., `counter += f(1, alpha, beta)`
  $\Rightarrow$ `counter` is accessed over and over
  $\Rightarrow$ the same frame is accessed over and over
- Spatial locality: repeated access to nearby memory locations
  e.g., `a[i] = a[i-1] + a[i-2]`
  $\Rightarrow$ all three array elements are very likely in the same frame
  $\Rightarrow$ the same frame is accessed over and over
- Therefore, as a process executes, the address translation request look like:
  - Give me Frame Number of Page 12

Locality and Caching

However!

- Temporal locality: repeated access to the same memory location
  e.g., `counter += f(1, alpha, beta)`
  $\Rightarrow$ `counter` is accessed over and over
  $\Rightarrow$ the same frame is accessed over and over
- Spatial locality: repeated access to nearby memory locations
  e.g., `a[i] = a[i-1] + a[i-2]`
  $\Rightarrow$ all three array elements are very likely in the same frame
  $\Rightarrow$ the same frame is accessed over and over
- Therefore, as a process executes, the address translation request look like:
  - Give me Frame Number of Page 12
  - Give me Frame Number of Page 12 again

Locality and Caching

However!

- Temporal locality: repeated access to the same memory location
  e.g., `counter += f(1, alpha, beta)`
  $\Rightarrow$ `counter` is accessed over and over
  $\Rightarrow$ the same frame is accessed over and over
- Spatial locality: repeated access to nearby memory locations
  e.g., `a[i] = a[i-1] + a[i-2]`
  $\Rightarrow$ all three array elements are very likely in the same frame
  $\Rightarrow$ the same frame is accessed over and over
- Therefore, as a process executes, the address translation request look like:
  - Give me Frame Number of Page 12
  - Give me Frame Number of Page 12 again
  - Give me Frame Number of Page 12 again

Locality and Caching

However!

- Temporal locality: repeated access to the same memory location
  e.g., `counter += f(1, alpha, beta)`
  $\Rightarrow$ `counter` is accessed over and over
  $\Rightarrow$ the same frame is accessed over and over
- Spatial locality: repeated access to nearby memory locations
  e.g., `a[i] = a[i-1] + a[i-2]`
  $\Rightarrow$ all three array elements are very likely in the same frame
  $\Rightarrow$ the same frame is accessed over and over
- Therefore, as a process executes, the address translation request look like:
  - Give me Frame Number of Page 12
  - Give me Frame Number of Page 12 again
  - Give me Frame Number of Page 12 again
  - and again, and again...

Locality and Caching

However!

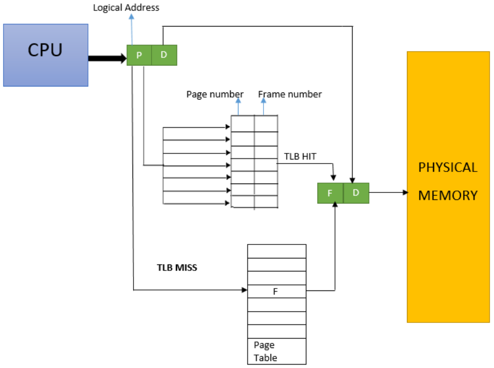
- Temporal locality: repeated access to the same memory location
  e.g., `counter += f(1, alpha, beta)`
  $\Rightarrow$ `counter` is accessed over and over
  $\Rightarrow$ the same frame is accessed over and over
- Spatial locality: repeated access to nearby memory locations
  e.g., `a[i] = a[i-1] + a[i-2]`
  $\Rightarrow$ all three array elements are very likely in the same frame
  $\Rightarrow$ the same frame is accessed over and over
- Therefore, as a process executes, the address translation request look like:
  - Give me Frame Number of Page 12
  - Give me Frame Number of Page 12 again
  - Give me Frame Number of Page 12 again
  - and again, and again...
- We should REMEMBER (i.e., cache) previous translation results!!

The TLB

- Caching of previous translations is done by a hardware component called...
- The Translation Lookaside Buffer (TLB)

The TLB

- Caching of previous translations is done by a hardware component called...
- The Translation Lookaside Buffer (TLB)
  - Each entry in the TLB is a <key, value> pair
  - You give it a key
  - The key is compared *in parallel* with all stored keys
  - If the key is found, then the associated value is returned



(Image Source: Wikipedia —Translation lookaside buffer. 2016-11-19)

TLB Performance

- Typical TLB characteristics:
  - Contains 12 to 4,096 entries
  - Performance:
    - On hit: less than 1 clock cycle
    - On miss: 10-100 clock cycles
  - Miss rate: 0.01 - 1%

TLB Performance

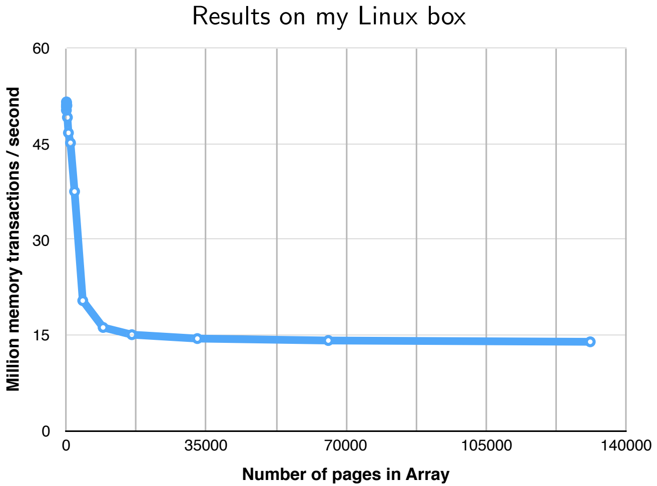
- Typical TLB characteristics:
  - Contains 12 to 4,096 entries
  - Performance:
    - On hit: less than 1 clock cycle
    - On miss: 10-100 clock cycles
  - Miss rate: 0.01 - 1%
- A Replacement Policy must be defined when the TLB is full:
  - Least Recently Used (LRU)? Random?
- Some TLBs allow for some entries to be un-evictable
  - e.g., kernel pages

Experiment: How useful is the TLB

tlb_stress.c: a piece of code that allocates an array spanning multiple pages and then writes values at random locations (runs for some 20 seconds each time)

Experiment: How useful is the TLB

tlb_stress.c: a piece of code that allocates an array spanning multiple pages and then writes values at random locations (runs for some 20 seconds each time)



The TLB and Context-Switches

- What happens with the TLB on a context-switch?
- Wipe the TLB?
  - VPN 7 of process A is not the same in the same frame as VPN of process B
  - Called a “TLB flush”
  - But perhaps unnecessary aggressive (the two processes could happily share the TLB)
  - So your machine doesn't do a flush
- ASIDs: Address-Space IDentifiers
  - Each TLB entry is annotated with a process identifier
  - The TLB can contain entries associated to multiple processes (kernel code, shared libraries, multi-threaded program, ...)
  - Each lookup attempts to match entry ASIDs with the ASID of the current process (and if mismatch then it's a TLB miss)

One down, one to go...

- **Problem #1:** Paging has extra overhead
- Solution: Use a TLB
  - Only works because our programs have locality "naturally"
  - which is why caches work, and the TLB is a kind of cache

One down, one to go...

- **Problem #1:** Paging has extra overhead
- Solution: Use a TLB
    - Only works because our programs have locality "naturally"
    - which is why caches work, and the TLB is a kind of cache
- Problem #2: Page tables can be very large
- Let's look at this one now...

Page Table Structure

- I've shown page tables like this:

Page Table

P0	14	✓
P1	13	✓
P2	18	✓
P3	20	✓
P4	xx	-
P5	xx	-
P6	xx	-
P7	xx	-

- But, once again, this is not quite right!

Page Table Entries

- One thing we haven't talked about yet: how many bits are needed for a page table entry?
  - I've shown the page table as just a table with numbers in it (and the valid bit)
  - But the page table consumes space in RAM

Page Table Entries

- One thing we haven't talked about yet: how many bits are needed for a page table entry?
  - I've shown the page table as just a table with numbers in it (and the valid bit)
  - But the page table consumes space in RAM
- Let us consider a system with 32-bit physical addresses, i.e., a 4GiB RAM

Page Table Entries

- One thing we haven't talked about yet: how many bits are needed for a page table entry?
  - I've shown the page table as just a table with numbers in it (and the valid bit)
  - But the page table consumes space in RAM
- Let us consider a system with 32-bit physical addresses, i.e., a 4GiB RAM
- The *n*-th entry in the page table is:
  - The physical frame number
  - A few bits (for now we've seen the valid bit, ASID bits, but there are other things)

Page Table Entries

- One thing we haven't talked about yet: how many bits are needed for a page table entry?
    - I've shown the page table as just a table with numbers in it (and the valid bit)
    - But the page table consumes space in RAM
- Let us consider a system with 32-bit physical addresses, i.e., a 4GiB RAM
- The *n*-th entry in the page table is:
    - The physical frame number
    - A few bits (for now we've seen the valid bit, ASID bits, but there are other things)
- Let us assume a page/frame size of 4 KiB $= 2^{12}$ bytes

Page Table Entries

- One thing we haven't talked about yet: how many bits are needed for a page table entry?
    - I've shown the page table as just a table with numbers in it (and the valid bit)
    - But the page table consumes space in RAM
- Let us consider a system with 32-bit physical addresses, i.e., a 4GiB RAM
- The *n*-th entry in the page table is:
    - The physical frame number
    - A few bits (for now we've seen the valid bit, ASID bits, but there are other things)
- Let us assume a page/frame size of 4 KiB $= 2^{12}$ bytes
- We have $2^{32}/2^{12} = 2^{20}$ frames in RAM

Page Table Entries

- One thing we haven't talked about yet: how many bits are needed for a page table entry?
    - I've shown the page table as just a table with numbers in it (and the valid bit)
    - But the page table consumes space in RAM
- Let us consider a system with 32-bit physical addresses, i.e., a 4GiB RAM
- The *n*-th entry in the page table is:
    - The physical frame number
    - A few bits (for now we've seen the valid bit, ASID bits, but there are other things)
- Let us assume a page/frame size of 4 KiB $= 2^{12}$ bytes
- We have $2^{32}/2^{12} = 2^{20}$ frames in RAM
- So the frame number can be encoded on 20 bits

Page Table Entries

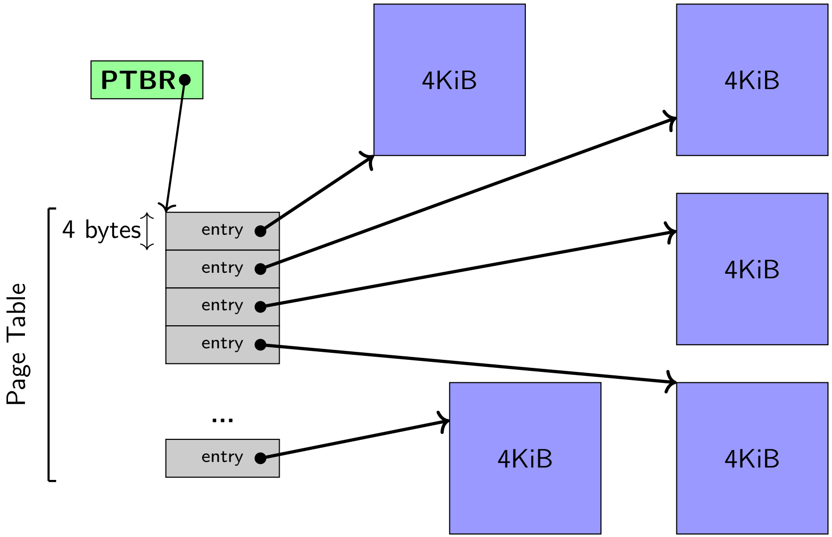
- One thing we haven't talked about yet: how many bits are needed for a page table entry?
  - I've shown the page table as just a table with numbers in it (and the valid bit)
  - But the page table consumes space in RAM
- Let us consider a system with 32-bit physical addresses, i.e., a 4GiB RAM
- The *n*-th entry in the page table is:
  - The physical frame number
  - A few bits (for now we've seen the valid bit, ASID bits, but there are other things)
- Let us assume a page/frame size of 4 KiB $= 2^{12}$ bytes
- We have $2^{32}/2^{12} = 2^{20}$ frames in RAM
- So the frame number can be encoded on 20 bits
- So a page table entry is 20 bits for the frame number, and then extra bits for "other stuff"

Page Table Entries

- One thing we haven't talked about yet: how many bits are needed for a page table entry?
  - I've shown the page table as just a table with numbers in it (and the valid bit)
  - But the page table consumes space in RAM
- Let us consider a system with 32-bit physical addresses, i.e., a 4GiB RAM
- The *n*-th entry in the page table is:
  - The physical frame number
  - A few bits (for now we've seen the valid bit, ASID bits, but there are other things)
- Let us assume a page/frame size of 4 KiB $= 2^{12}$ bytes
- We have $2^{32}/2^{12} = 2^{20}$ frames in RAM
- So the frame number can be encoded on 20 bits
- So a page table entry is 20 bits for the frame number, and then extra bits for "other stuff"
- Let's say that 32 bits $=$ 4 bytes are used (which is typical for a 32-bit architecture)

Page Table Entries

- On a picture:



A Note on Page Table Structure

- The page table is **just an array of entries**
  - The entry for page 0 is the first element of the array
  - The entry for page 1 is the second element of the array
  - The entry for page i is the i-th element of the array

- So when we say "lookup an entry" we don't mean a *search*
- Looking up the entry for page i means: PTBR $+$ i $\times$ entry size

A Note on Page Table Structure

- The page table is **just an array of entries**
  - The entry for page 0 is the first element of the array
  - The entry for page 1 is the second element of the array
  - The entry for page i is the i-th element of the array

- So when we say “lookup an entry” we don’t mean a *search*
- Looking up the entry for page i means: PTBR $+ \ i \times$ entry size

- For instance:
  - The PTBR contains address 0xAAAA0000
  - The page table entry size is 4-bytes
  - I want to “lookup” the entry for page 10
  - The entry for that page is at address 0xAAAA0028

    (i.e., PTBR $+ \ 4 \times 10$)
  - We get the 4 bytes at that address
  - These bytes are: the frame number, the valid bit, other useful bits

Page Table Size

- So we have page table entries that are each 4 bytes
- Let's consider a process with a 4GiB address space (i.e., uses all available physical RAM)

Page Table Size

- So we have page table entries that are each 4 bytes
- Let's consider a process with a 4GiB address space (i.e., uses all available physical RAM)
- This process has $2^{32}/2^{12} = 2^{20}$ pages
  - Because the page size is 2^{12} bytes

Page Table Size

- So we have page table entries that are each 4 bytes
- Let's consider a process with a 4GiB address space (i.e., uses all available physical RAM)
- This process has $2^{32}/2^{12} = 2^{20}$ pages
  - Because the page size is 2^{12} bytes
- The process' page table thus has 2^{20} entries

Page Table Size

- So we have page table entries that are each 4 bytes
- Let's consider a process with a 4GiB address space (i.e., uses all available physical RAM)
- This process has $2^{32}/2^{12} = 2^{20}$ pages
  - Because the page size is 2^{12} bytes
- The process' page table thus has 2^{20} entries
- Therefore, the page table takes up $2^{20} \times 2^{2} = 2^{22}$ bytes
  - which is 4 MiB

Page Table Size

- So we have page table entries that are each 4 bytes
- Let's consider a process with a 4GiB address space (i.e., uses all available physical RAM)
- This process has $2^{32}/2^{12} = 2^{20}$ pages
    - Because the page size is 2^{12} bytes
- The process' page table thus has 2^{20} entries
- Therefore, the page table takes up $2^{20} \times 2^{2} = 2^{22}$ bytes
    - which is 4 MiB

- So we need 4 MiB of contiguous RAM space to store the page table
- Let me repeat...

Page Table Size

- So we have page table entries that are each 4 bytes
- Let's consider a process with a 4GiB address space (i.e., uses all available physical RAM)
- This process has $2^{32}/2^{12} = 2^{20}$ pages
    - Because the page size is 2^{12} bytes
- The process' page table thus has 2^{20} entries
- Therefore, the page table takes up $2^{20} \times 2^{2} = 2^{22}$ bytes
    - which is 4 MiB

- So we need 4 MiB of contiguous RAM space to store the page table
- Let me repeat...

We need 4 MiB of **contiguous** RAM space!!!!

Page Table Size

We need 4 MiB of **contiguous** RAM space!!!!

- We use paging to avoid large contiguous slabs of RAM
- To implement paging we use page tables
- But page tables are large contiguous slabs of RAM

Page Table Size

We need 4 MiB of **contiguous** RAM space!!!!

- We use paging to avoid large contiguous slabs of RAM
- To implement paging we use page tables
- But page tables are large contiguous slabs of RAM

To avoid big slabs of RAM we need big slabs of RAM

Page Table Size

We need 4 MiB of **contiguous** RAM space!!!!

- We use paging to avoid large contiguous slabs of RAM
- To implement paging we use page tables
- But page tables are large contiguous slabs of RAM

To avoid big slabs of RAM we need big slabs of RAM

Splitting the Page Table into Pages!!

- What do we do when we have big slabs or RAM?
- We split them into pages!

Splitting the Page Table into Pages!!

- What do we do when we have big slabs or RAM?
- We split them into pages!
- So the (large) page table is stored in multiple, possible non-contiguous pages
- The main questions: how many page table entries can fit in a page?

Splitting the Page Table into Pages!!

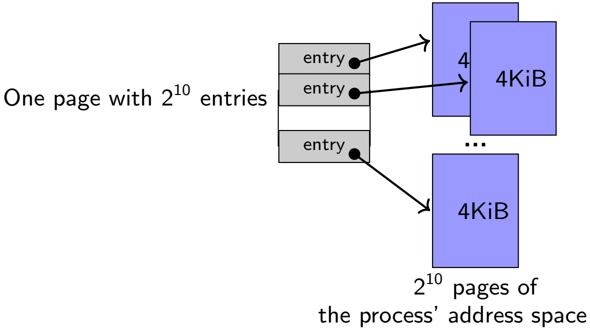
- What do we do when we have big slabs or RAM?
- We split them into pages!
- So the (large) page table is stored in multiple, possible non-contiguous pages
- The main questions: how many page table entries can fit in a page?
- In out example, a page is 4KiB and an entry is 4 bytes
- So a page can contain 2^{10} (1,024) entries

Splitting the Page Table into Pages!!

- What do we do when we have big slabs or RAM?
- We split them into pages!
- So the (large) page table is stored in multiple, possible non-contiguous pages
- The main questions: how many page table entries can fit in a page?
- In out example, a page is 4KiB and an entry is 4 bytes
- So a page can contain 2^{10} (1,024) entries
- In the previous slide we said that our page table needs to have 2^{20} entries
- Therefore, we need $2^{20}/2^{10} = 2^{10}$ pages of page table entries
- That's right: "page table pages"

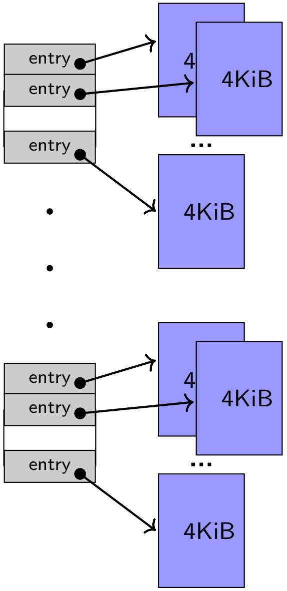
- Let's see this on a picture...

Page Table Pages

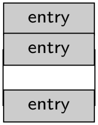


Page Table Pages

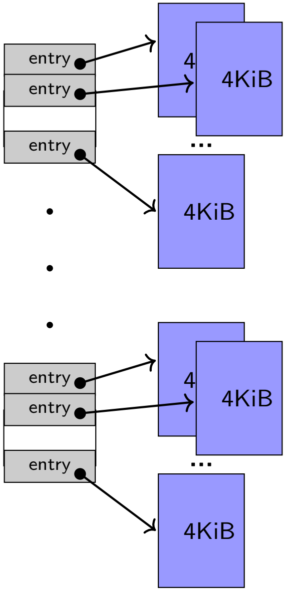
2^{10} pages of entries, for a total of $2^{10} \times 2^{10} = 2^{20}$ pages of pages of the process' address space



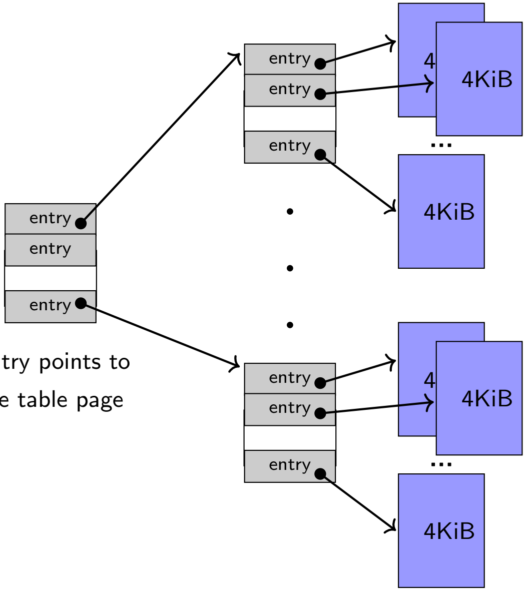
Page Table Pages



One page with 2^{10} entries

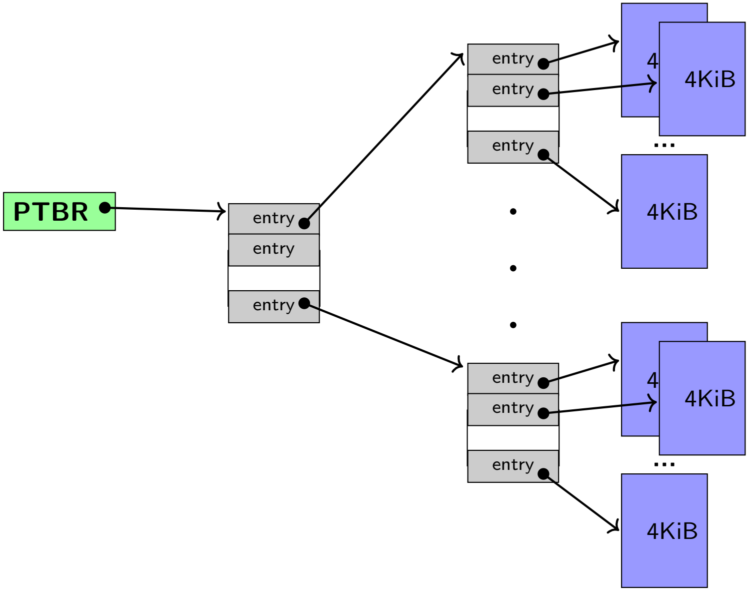


Page Table Pages

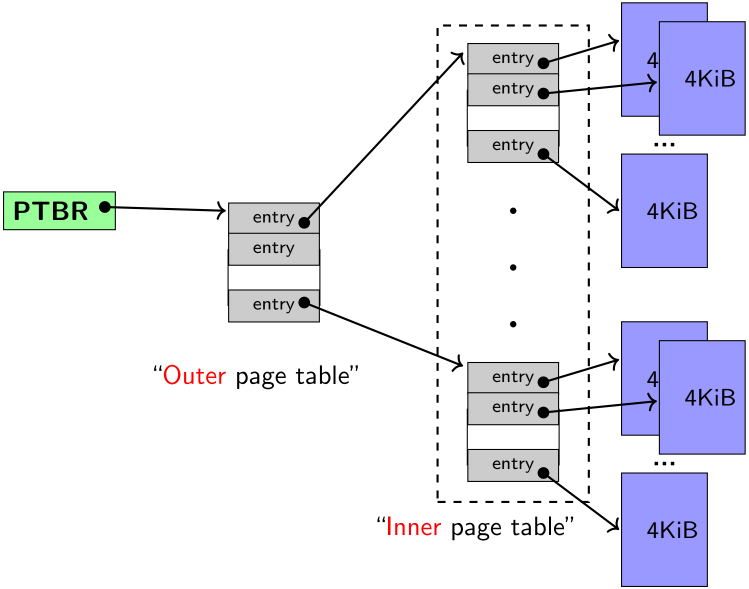


Each entry points to
a page table page

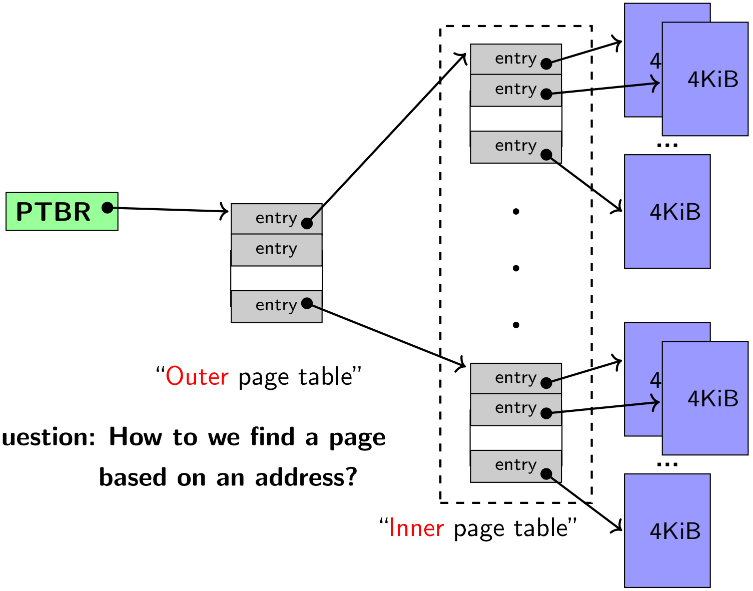
Page Table Pages



Page Table Pages



Page Table Pages



Question: How to we find a page based on an address?

Hierarchical Page Tables

- The picture on the previous slide is a hierarchical page table
- Given a 32-bit virtual address we split it as follows:

10-bit index into outer page table	10-bit index into inner page table page	12-bit offset

- The first 10 address bits: to pick one of the 2^{10} entries in the outer page table should we use to find an inner page table page
- The next 10 address bits: to pick one the the 2^{10} entries in the inner page table page should we use to find an address space page
- The next 12 address the offset in that page

Hierarchical Page Tables

- The picture on the previous slide is a hierarchical page table
- Given a 32-bit virtual address we split it as follows:

10-bit index into outer page table	10-bit index into inner page table page	12-bit offset

- The first 10 address bits: to pick one of the 2^{10} entries in the outer page table should we use to find an inner page table page
- The next 10 address bits: to pick one the the 2^{10} entries in the inner page table page should we use to find an address space page
- The next 12 address the offset in that page

- This working perfectly, luckily, because a page contained 2^{10} entries and 2^{12} bytes

Hierarchical Page Tables: Address Translation

$p1$	$p2$	offset

- (Note: [@] means "Contents at address @")
- Address of the the outer page table: PTBR

Hierarchical Page Tables: Address Translation

$p1$	$p2$	offset

- (Note: [@] means "Contents at address @")
- Address of the the outer page table: PTBR
- Address of the relevant outer page table entry: PTBR $+ 4 \times$ p1

Hierarchical Page Tables: Address Translation

$p1$	$p2$	offset

- (Note: [@] means "Contents at address @")
- Address of the the outer page table: PTBR
- Address of the relevant outer page table entry: PTBR $+ 4 \times$ p1
- Address of the relevant page table page: [PTBR $+ 4 \times$ p1]

Hierarchical Page Tables: Address Translation

*p*1	*p*2	offset

- (Note: [@] means "Contents at address @")
- Address of the the outer page table: PTBR
- Address of the relevant outer page table entry: PTBR $+ 4 \times$ p1
- Address of the relevant page table page: [PTBR $+ 4 \times$ p1]
- Address of the relevant entry therein: [PTBR $+ 4 \times$ p1] $+ 4 \times$ p2

Hierarchical Page Tables: Address Translation

*p*1	*p*2	offset

- (Note: [@] means "Contents at address @")
- Address of the the outer page table: PTBR
- Address of the relevant outer page table entry: PTBR + 4 × p1
- Address of the relevant page table page: [PTBR + 4 × p1]
- Address of the relevant entry therein: [PTBR + 4 × p1] + 4 × p2
- Address of the page: [[PTBR + 4 × p1] + 4 × p2]

Hierarchical Page Tables: Address Translation

$p1$	$p2$	offset

- (Note: [@] means "Contents at address @")
- Address of the the outer page table: PTBR
- Address of the relevant outer page table entry: PTBR $+ 4 \times$ p1
- Address of the relevant page table page: [PTBR $+ 4 \times$ p1]
- Address of the relevant entry therein: [PTBR $+ 4 \times$ p1] $+ 4 \times$ p2
- Address of the page: [[PTBR $+ 4 \times$ p1] $+ 4 \times$ p2]
- Physical address: [[PTBR $+ 4 \times$ p1] $+ 4 \times$ p2] $+$ offset

(See OSC figure 8.17)

In-class Exercise

- Page size: 32 KiB
- Logical addresses: 39 bits
- Page table entry size: 8 bytes

- Using 2-level paging, how is a logical address split into 3 outer page, inner page, and offset (denoted p1, p2, offset)?

- Questions to ask oneself:
  - How many bits for the offset?
  - How many page table entries can fit in a page? (gives us p2)
  - Then compute p1 as 39 - p1 - offset

In-class Exercise (Solution)

- Page size: 32 KiB
- Logical addresses: 39 bits
- Page table entry size: 8 bytes (= 64 bits)
- Using 2-level paging, how is a logical address split into 3 outer page, inner page, and offset (denoted p1, p2, offset)?

- There are $2^5 \times 2^{10} = 2^{15}$ bytes in a page, offset = 15
- We can have up to $2^{39-15} = 2^{24}$ pages in the address space
- We have $2^{15}/2^3 = 2^{12}$ page table entries in a page
- Therefore an inner page table page points to 2^{12} pages: p2 = 12
- Therefore, p1 = 39 - p2 - offset = 39 - 12 - 15 = 12
- This is yet another "lucky" case in which everything fits perfectly (because the inner page table has exactly 2^{12} entries)

Another In-class Exercise

- Page size: 64 KiB
- Logical addresses: 41 bits
- Page table entry size: 4 bytes

- Using 2-level paging, how is a logical address split into 3 outer page, inner page, and offset (denoted p1, p2, offset)?
- What fraction of the outer page table is utilized?

Another In-class Exercise (Solution)

- Page size: 64 KiB
- Logical addresses: 41 bits
- Page table entry size: 4 bytes (= 64 bits)

- offset = 16 bits (because 2^{16} bytes in a page)
- An inner page table page points to $2^{16}/2^{2} = 2^{14}$ pages
- Therefore, p2 = 14
- And p1 = 41 - 14 - 16 = 11
- The outer page table page thus needs to hold 2^{11} entries
- But it could hold up to 2^{14} entries
- Therefore, only $2^{11}/2^{14} = 1/8 = 12.5\%$ of it are used!

Hierarchical Page Tables are it then?

For 64-bit addresses, with 2-level paging, we are still in trouble though...

- 4 KiB page size
- Assume 64-bit virtual addresses
- One outer page can address $2^{12}/8 = 2^{12}/2^3 = 2^9$ inner pages
- Therefore: 64 - 12 - 9 = 43 bits to address all outer pages

Hierarchical Page Tables are it then?

For 64-bit addresses, with 2-level paging, we are still in trouble though...

- 4 KiB page size
- Assume 64-bit virtual addresses
- One outer page can address $2^{12}/8 = 2^{12}/2^3 = 2^9$ inner pages
- Therefore: 64 - 12 - 9 $=$ 43 bits to address all outer pages
- The total of outer page size must be: $2^{43} * 8 = 64 * 2^{40}$ $=$ 64 TiB!

Hierarchical Page Tables are it then?

For 64-bit addresses, with 2-level paging, we are still in trouble though...

- 4 KiB page size
- Assume 64-bit virtual addresses
- One outer page can address $2^{12}/8 = 2^{12}/2^3 = 2^9$ inner pages
- Therefore: 64 - 12 - 9 $=$ 43 bits to address all outer pages
- The total of outer page size must be: $2^{43} * 8 = 64 * 2^{40} = 64$ TiB!
- So we need an extra level: 33 (second outer page) $+$ 10 $+$ 10 $+$ 12
- But the second outer page is still $2^{33} * 8 = 64$ GiB and we now have three indirections

Hierarchical Page Tables are it then?

For 64-bit addresses, with 2-level paging, we are still in trouble though...

- 4 KiB page size
- Assume 64-bit virtual addresses
- One outer page can address $2^{12}/8 = 2^{12}/2^3 = 2^9$ inner pages
- Therefore: 64 - 12 - 9 $=$ 43 bits to address all outer pages
- The total of outer page size must be: $2^{43} * 8 = 64 * 2^{40} = 64$ TiB!
- So we need an extra level: 33 (second outer page) $+ 10 + 10 + 12$
- But the second outer page is still $2^{33} * 8 = 64$ GiB and we now have three indirections
- Conclusion: Hierarchical page tables become memory hogs for large address spaces with small pages

Hierarchical Page Tables are it then?

For 64-bit addresses, with 2-level paging, we are still in trouble though...

- 4 KiB page size
- Assume 64-bit virtual addresses
- One outer page can address $2^{12}/8 = 2^{12}/2^{3} = 2^{9}$ inner pages
- Therefore: 64 - 12 - 9 $=$ 43 bits to address all outer pages
- The total of outer page size must be: $2^{43} * 8 = 64 * 2^{40} = 64$ TiB!
- So we need an extra level: 33 (second outer page) $+ 10 + 10 + 12$
- But the second outer page is still $2^{33} * 8 = 64$ GiB and we now have three indirections
- Conclusion: Hierarchical page tables become memory hogs for large address spaces with small pages
- In practice: Virtual addresses are not 64-bit (/proc/cpuinfo) but more like 48-bit
- In practice: 4 levels are used

Hashed Page Tables

- A completely different idea:
- Pick a maximum (desirable) size for the page table (say N)
- Create a hash function that associates any VPN to an integer of 0..N-1
- Structure the page table as a hash table using the hash function (each entry in 0..N-1 is a list of PFN)

- This is interesting but not really done in practice

Inverted Page Tables

- Yet another idea:
- One table for all processes
- One entry per physical memory frame
- Each entry is: ASID + logical page number

Inverted Page Tables

- Yet another idea:
- One table for all processes
- One entry per physical memory frame
- Each entry is: ASID + logical page number
- CPU issues addresses like: PID + VPN + offset
- And page table contains entries like (PID, p) to PFN
- Searching for (PID, p) is expensive
- And need for a mechanism to implement shared memory

- Was used in: PowerPC, UltraSPARC, IA-64 (Itanium) – Discontinued

Conclusion

- Paging is a good idea, but it has its problems

Conclusion

- Paging is a good idea, but it has its problems
- Problem #1: Address translation is slow
  - Solution: Use a TLB

Conclusion

- Paging is a good idea, but it has its problems
- Problem #1: Address translation is slow
  - Solution: Use a TLB
- Problem #2: The Page Table shouldn't be contiguous
  - Solution: Use a hierarchical structure
  - The hierarchical structure makes translation slower, but we don't case because we have a TLB anyway!

Conclusion

- Paging is a good idea, but it has its problems
- Problem #1: Address translation is slow
  - Solution: Use a TLB
- Problem #2: The Page Table shouldn't be contiguous
  - Solution: Use a hierarchical structure
  - The hierarchical structure makes translation slower, but we don't case because we have a TLB anyway!

- We still have one big question: What happens when a process needs a new page, and there is no free frame???
- We can now do all of Homework #7...