Computer Systems B

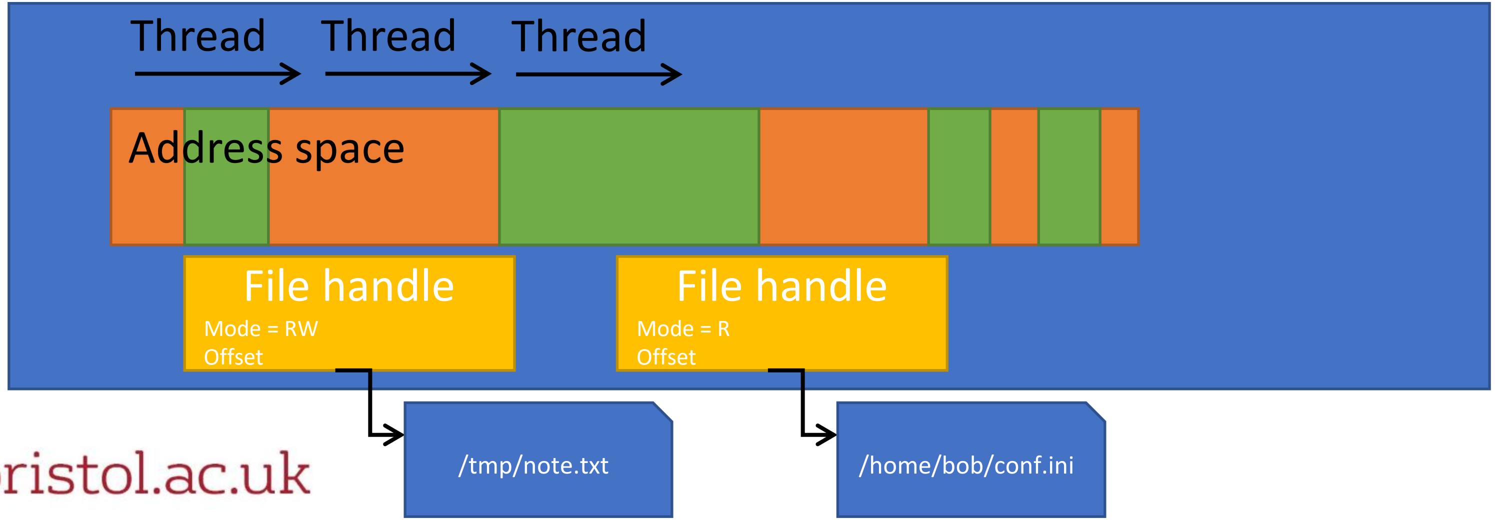
COMS20012

Introduction to Operating Systems and Security

Previously

OS is about abstractions

- There is four main abstractions that are core to an OS:
  - Threads – CPU
  - Virtual Memory – Memory
  - Files – Disk
  - Process – Encapsulate everything



Schedule

- Week 6 – Process
- Week 7 – Threads and in particular scheduling
- Week 8 – Virtual address space
- Week 9 – File Systems

Keep in mind this is an introduction course, there is a lot more complexity in practice than what we have time to see.

Why virtual memory?

Physical Memory

- Physical addresses are P bits long
  - Maximum amount of addressable physical memory is 2^P
- OS161's MIPS is 32 bits
  - 2^{32} physical addresses
  - Maximum of 4GB memory
- Modern CPU support large amount of addressable memory
  - X86_64
  - Physical 52 bits
  - Virtual 48 bits
- Far exceed current RAM technology
  - This won't be true forever ;)

Physical Memory

- Is finite
- Need to be shared between all processes
- Need to be carefully managed to avoid processes stepping on each other toes

Classic OS solution: **hide complexity through an abstraction**

Virtual Memory the basic

- The kernel provide a virtual memory for each process
- Virtual memory hold code, data and stack(s) for a process
- If virtual memory addresses are V bits
  - Amount of addressable virtual is 2^V
  - On OS161/MIPS V=32
- Running processes see **only** virtual memory
  - Program counter and stack pointer hold **virtual addresses**
  - Pointers to variable are **virtual addresses**
  - Jumps/branches refers to **virtual addresses**
- Each process is **isolated** in its virtual memory and **cannot address** other processes virtual memory

Why virtual memory?

- Isolate processes from each other
- Potentially to support virtual memory larger than physical memory
- Total size of virtual memories can be greater than the physical memory
  - Provide greater support for multiprocessing

University of BRISTOL

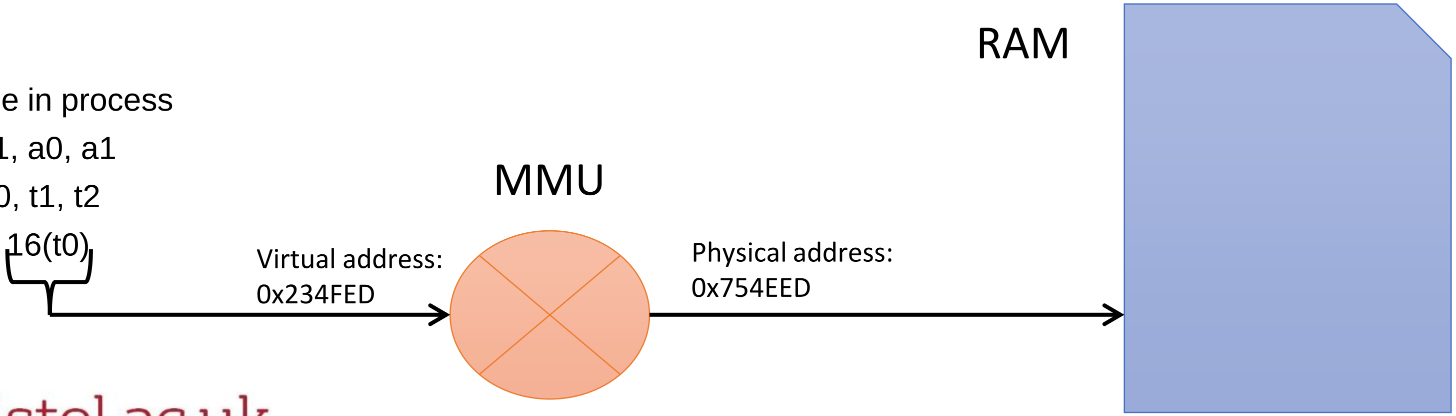
Segmented Virtual Memory

bristol.ac.uk

Memory-mapping Unit (MMU)

- MMU is a piece of hardware
  - Translate virtual addresses to physical addresses
  - Only configurable by a privileged process (i.e. the kernel)
- Virtual addresses are what a process uses
- Physical addresses is what the CPU present to the RAM

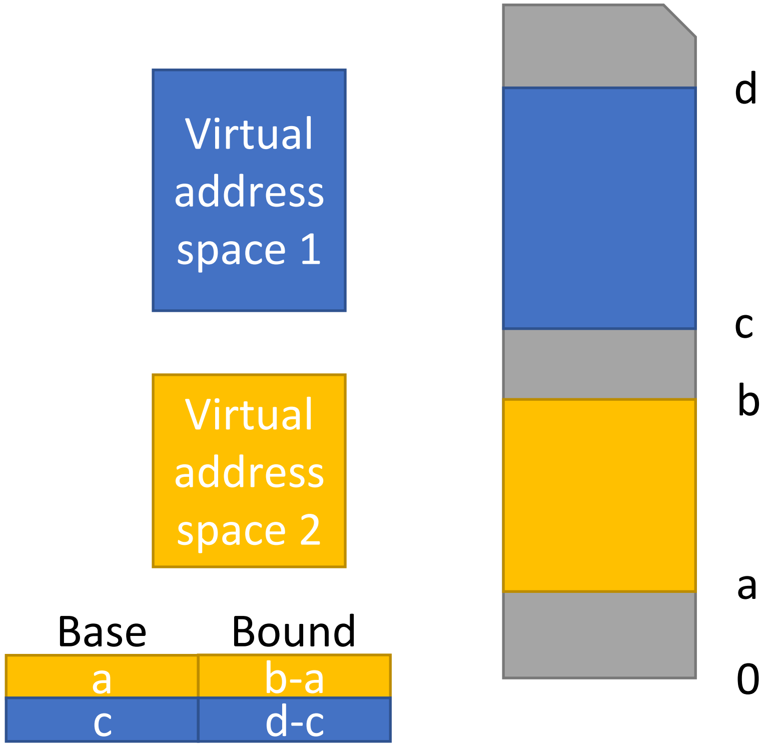
```
// code in process
sub t1, a0, a1
add t0, t1, t2
lw t4; 16(t0)
```



Early attempt: base + bound

- Associate virtual address with base and bound register
- Base: where the physical address space start
- Bound: the length of the address space (both virtual and physical)
- MMU formula:

```
if (virtual_add > bound)
        error()
else
        physical_add = virtual_add + base
```



Base + Bound pros and cons

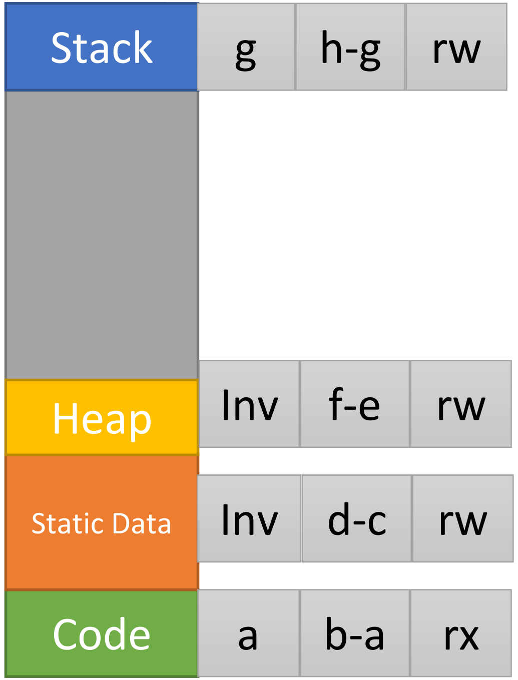
- Allow each virtual address space to be of different size
- Allow each virtual address space to be mapped into any physical RAM of sufficient size
- Straightforward isolation: just ensure no overlap!

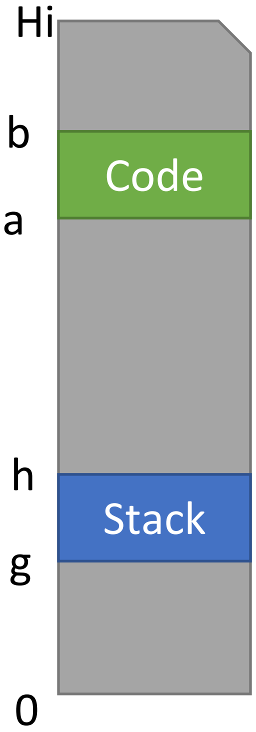
- Waste physical memory if the virtual address space is not fully used (i.e. hole between stack and heap)
- Same privilege everywhere read/write/execute
- Sharing memory can only happen by overlapping top and bottom of two spaces (if need to be shared by more than 2?)

Segmentation

- A single address space has multiple logical segement
  - Code: read/execute, fixed size
  - Static data: read/write, fixed size
  - Heap: read/write, dynamic size
  - Stack: read/write, dynamic size
- Each segment is associated with privilege + base + bound
  - At a given time some segment may not be mapped into the physical RAM
  - When not mapped they are **swapped** to disk (more on this later)

Segmentation





seg = find_seg(virtual_add)
if (offset(vritual_add) > seg.bound)
    error()
else
    physical_add = offset(virtual_add) + seg.base

Defining find_seg and offset:

– Partition approach

- ➢ High order bits for segment
- ➢ Low order bits for offset

– Explicit approach

- ➢ Virtual address as offset
- ➢ Instruction needs segment to be explicit

Segmentation Advantages

- Shared advantage with base + bound
  - Small address space metadata (few segments, few information about those segments)
  - Isolation is easy just ensure there is no overlap
  - Can map segment in any large enough region of physical RAM
- Advantage over base + bound
  - Can share memory at the segment granularity
  - Waste less memory (i.e. hole between heap and stack doesn't need to be mapped)
  - Enables segment granularity memory protection

Segmentation Disadvantages

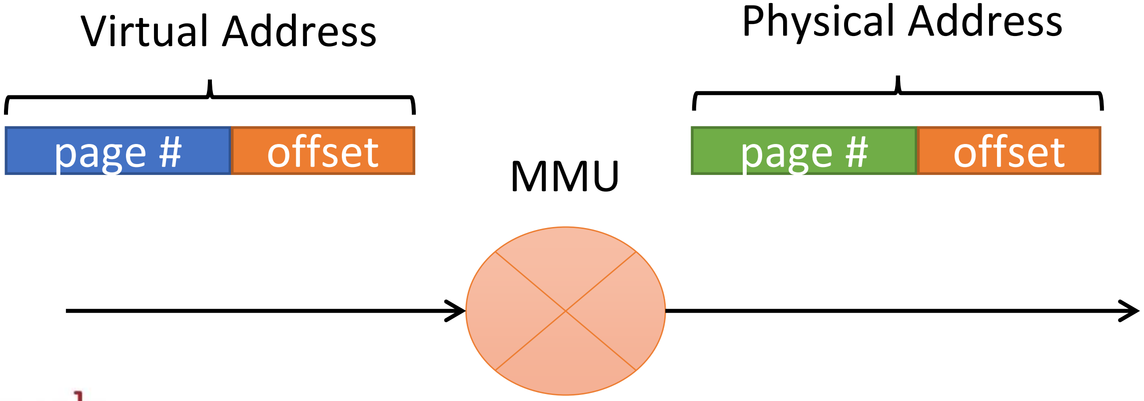
- Segment may be large
  - Need to map the whole segment into memory even to access a single byte
  - Cannot map only the part of the segment that is utilized
- Need to find free physical memory large enough to accommodate a segment
  - Several algorithm can be used **first fit, worst fit, best fit** (see exercises)
  - All have **trades-off**
- Explicit segment management is not very elegant (better with partitioned address)

Paging

bristol.ac.uk

Paging

- Let's solve two problems at once:
  - Makes allocation problem trivial
    - Fixed sized units called pages
    - No more bounds !
  - Use space efficiently
    - Small fixed size (no need to use large chunk of memory to access a single byte)
    - No more segment, address is divided in a collection of pages



Good and Bad

- Good
  - Can allocate virtual address space with fine granularity
  - Only need to bring small pages that the process needs into the RAM
- Bad
  - Bookeeping becomes more complex
  - Lots of small pages to keep track of

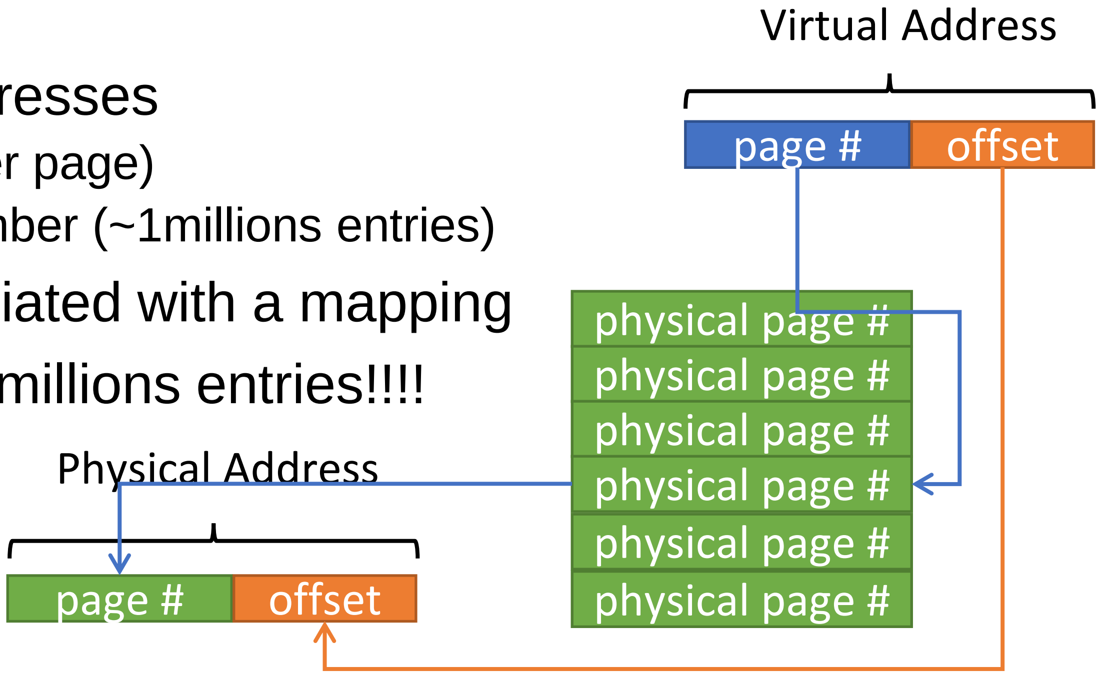
Good and Bad

- Good
  - Can allocate virtual address space with fine granularity
  - Only need to bring small pages that the process needs into the RAM
- Bad
  - Bookeeping becomes more complex
  - Lots of small pages to keep track of

Let’s see how to deal with this!

Single-level page table

- Need to keep around a maping between virtual page and physical page
- Suppose 32bits addresses
  - 12bits offset (4kb per page)
  - 20 bits for page number (~1millions entries)
- Each process associated with a mapping
- Need a table with 1 millions entries!!!!



Problems

- Most address space are sparse
  - Not all pages are used
  - In our example most process would use less than 1 million pages
- That means a huge map full of NULL entries

Problems

- Most address space are sparse
  - Not all pages are used
  - In our example most process would use less than 1 million pages
- That means a huge map full of NULL entries

What a computer scientist do?

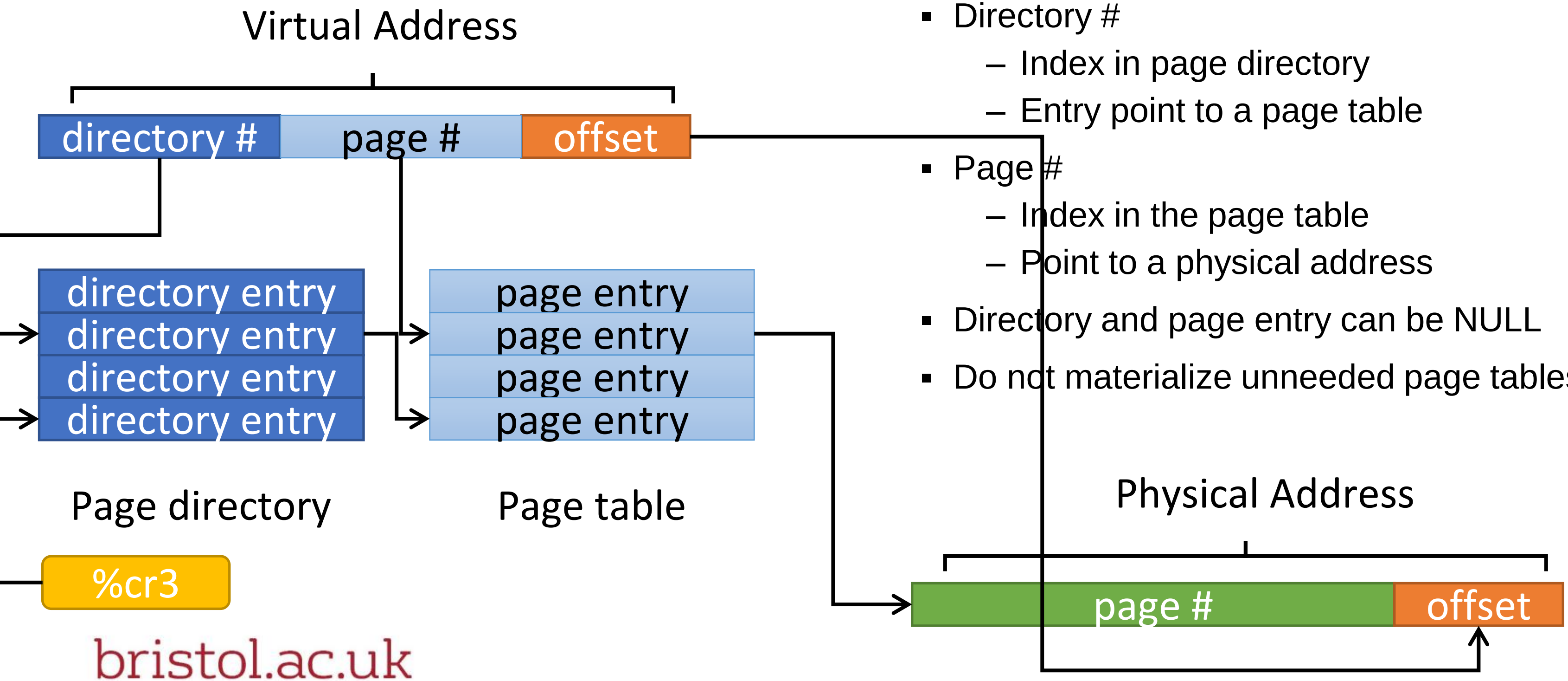
Problems

- Most address space are sparse
  - Not all pages are used
  - In our example most process would use less than 1 million pages
- That means a huge map full of NULL entries

What a computer scientist do?

We add a level of indirection!

Two-level page table



- Directory #
  - Index in page directory
  - Entry point to a page table
- Page #
  - Index in the page table
  - Point to a physical address
- Directory and page entry can be NULL
- Do not materialize unneeded page tables

bristol.ac.uk

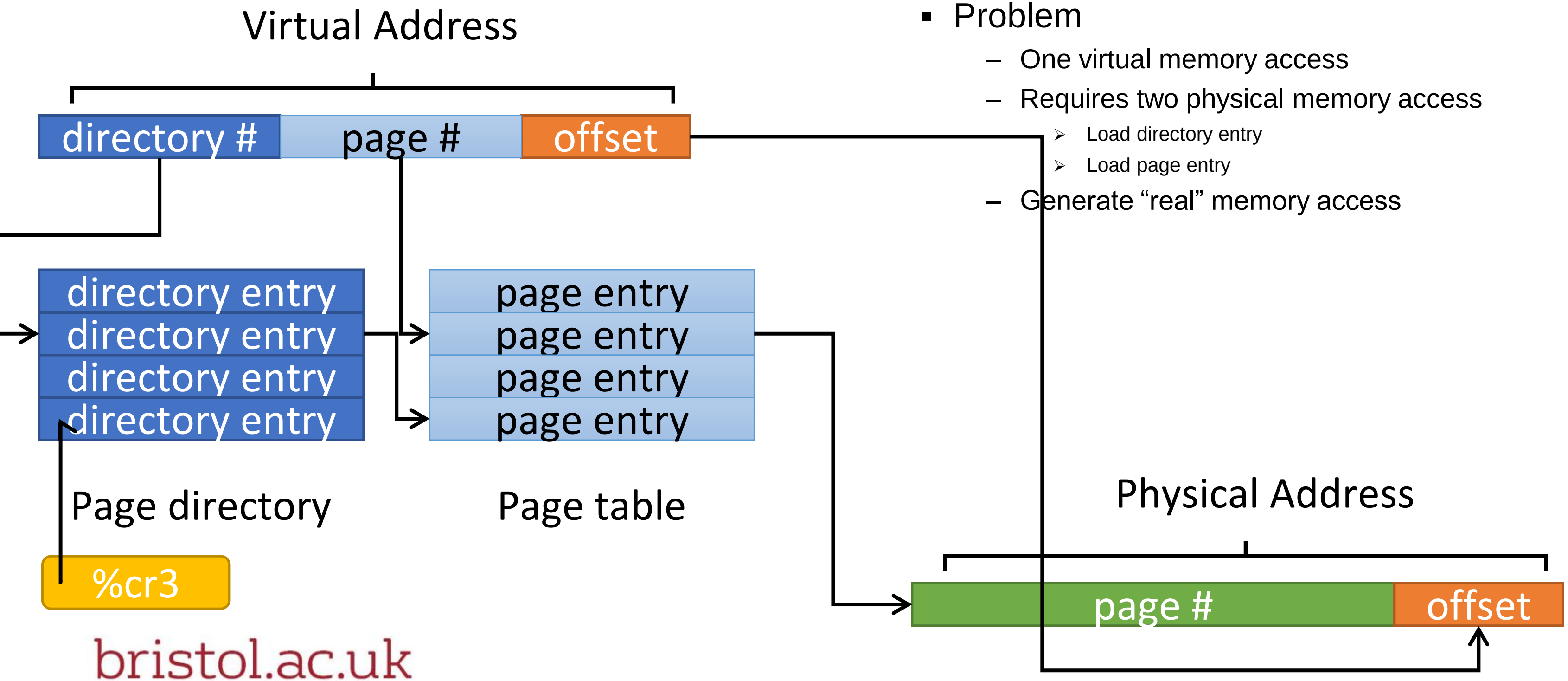
Problem

- Address translation seems more complicated
  - … and therefore slow
- How do we solve this?

University of BRISTOL

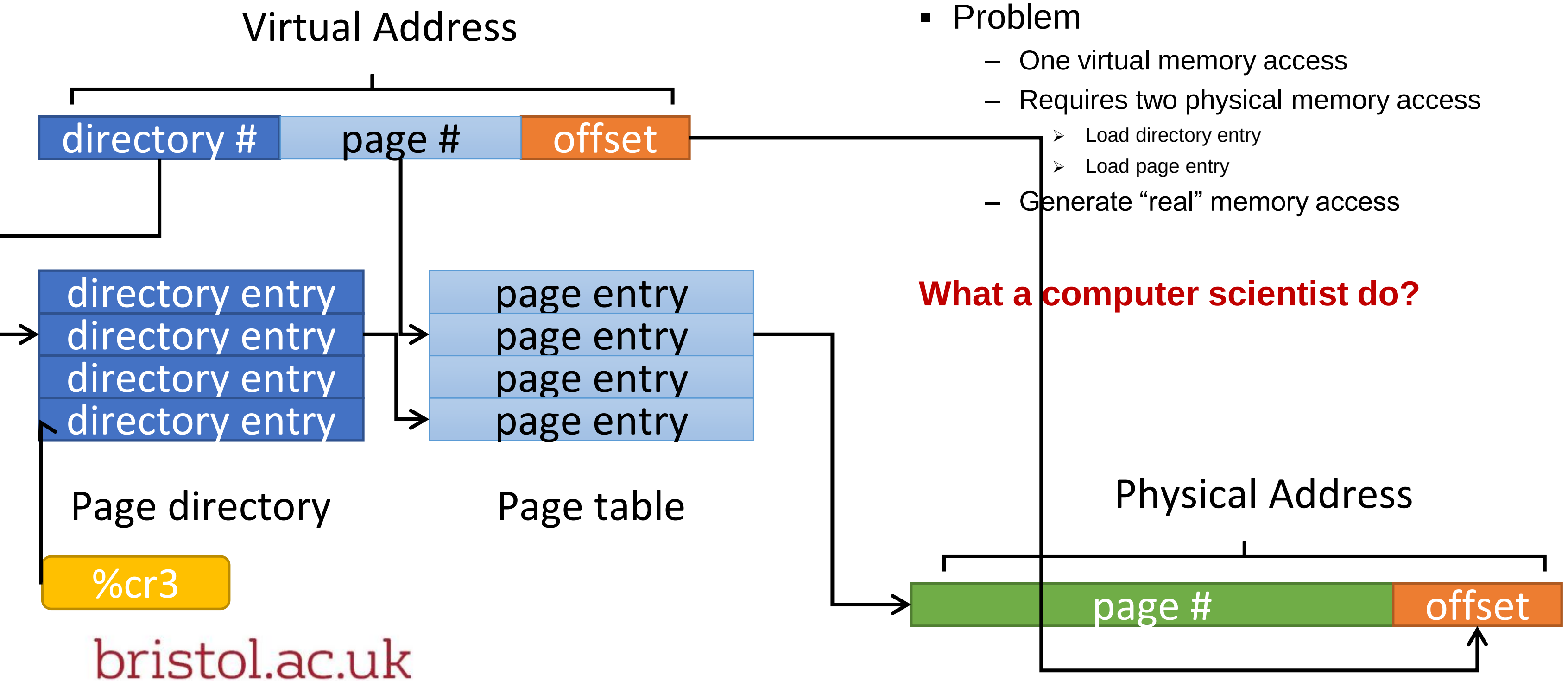
Translation Lookaside Buffer

Previously



- Problem
  - One virtual memory access
  - Requires two physical memory access
    - Load directory entry
    - Load page entry
  - Generate “real” memory access

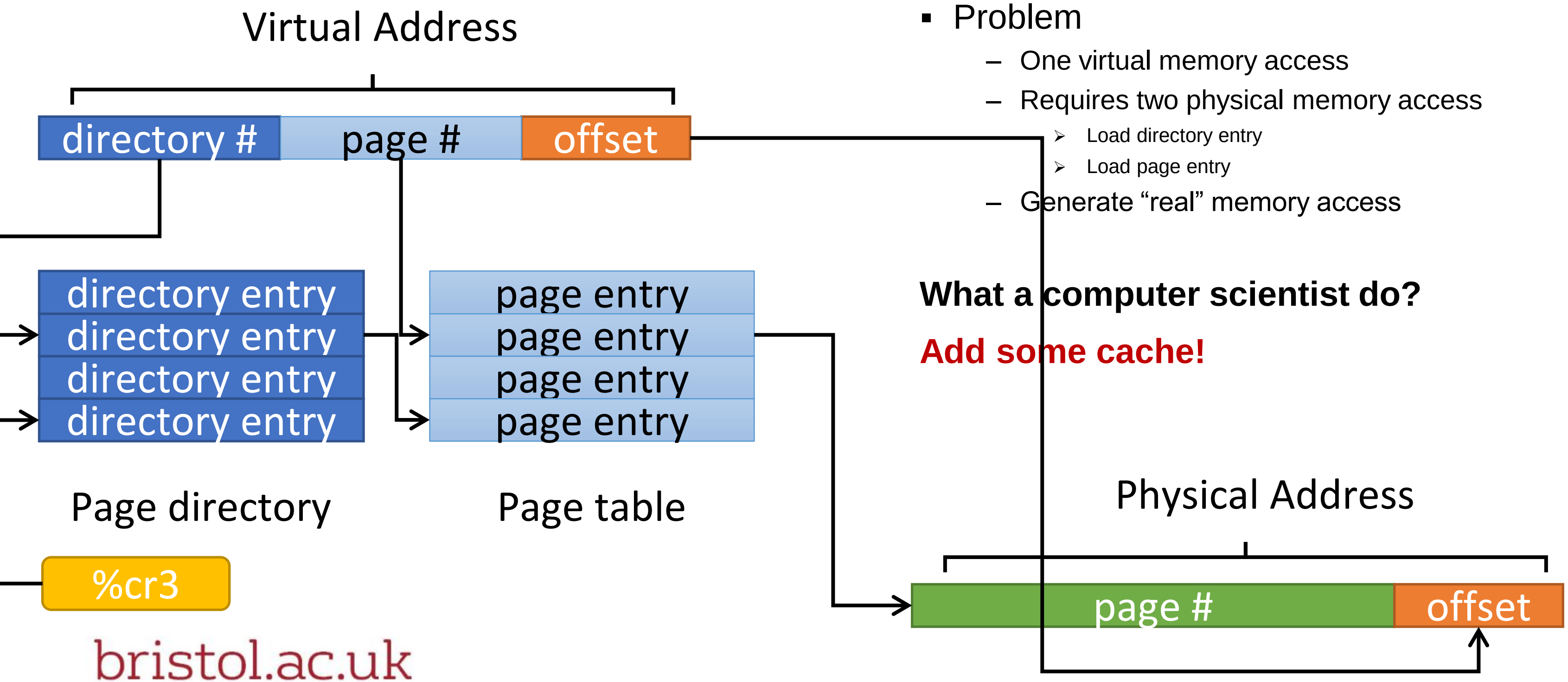
Previously



- Problem
  - One virtual memory access
  - Requires two physical memory access
    - Load directory entry
    - Load page entry
  - Generate “real” memory access

What a computer scientist do?

Previously



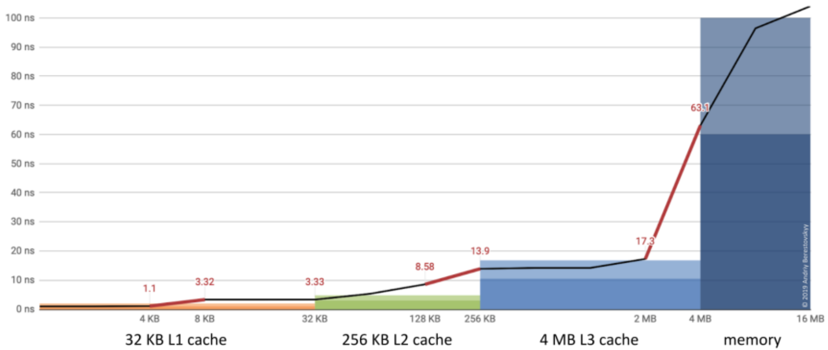
- Problem
  - One virtual memory access
  - Requires two physical memory access
    - Load directory entry
    - Load page entry
  - Generate "real" memory access

What a computer scientist do?

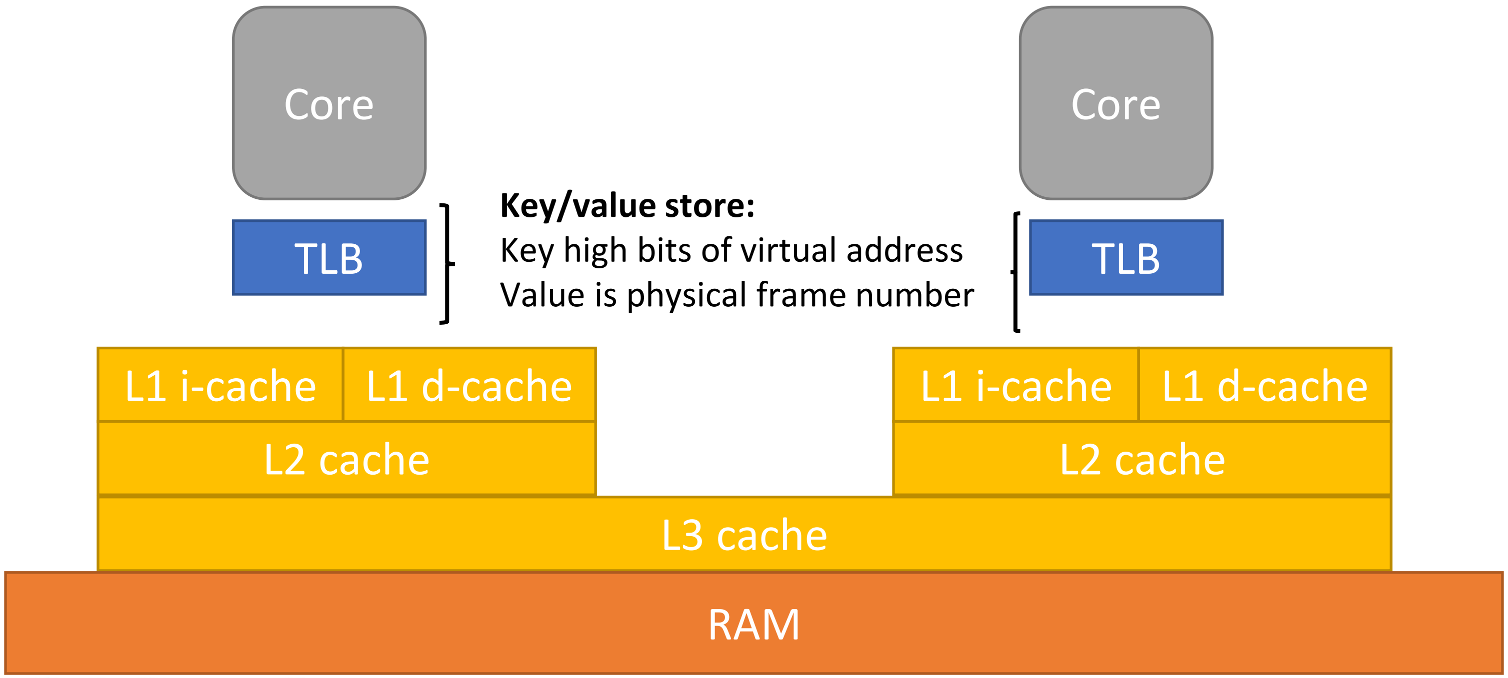
Add some cache!

Translation Lookaside Buffer (TLB)

- Cache some PTE in hardware buffer
- No need to go to physical memory to fetch PTE
- Hardware memory is way faster than main memory!
- We can also be clever about caching!



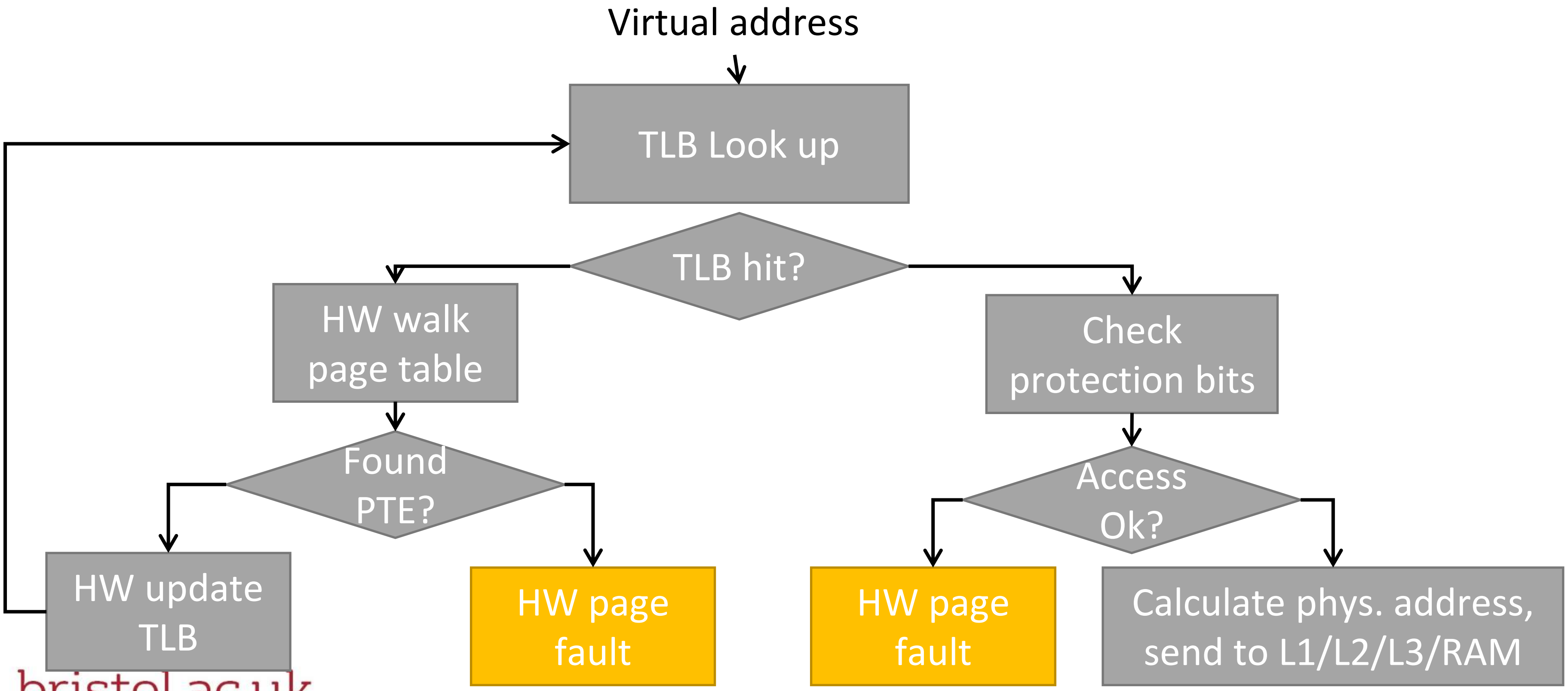
Translation Lookaside Buffer (TLB)



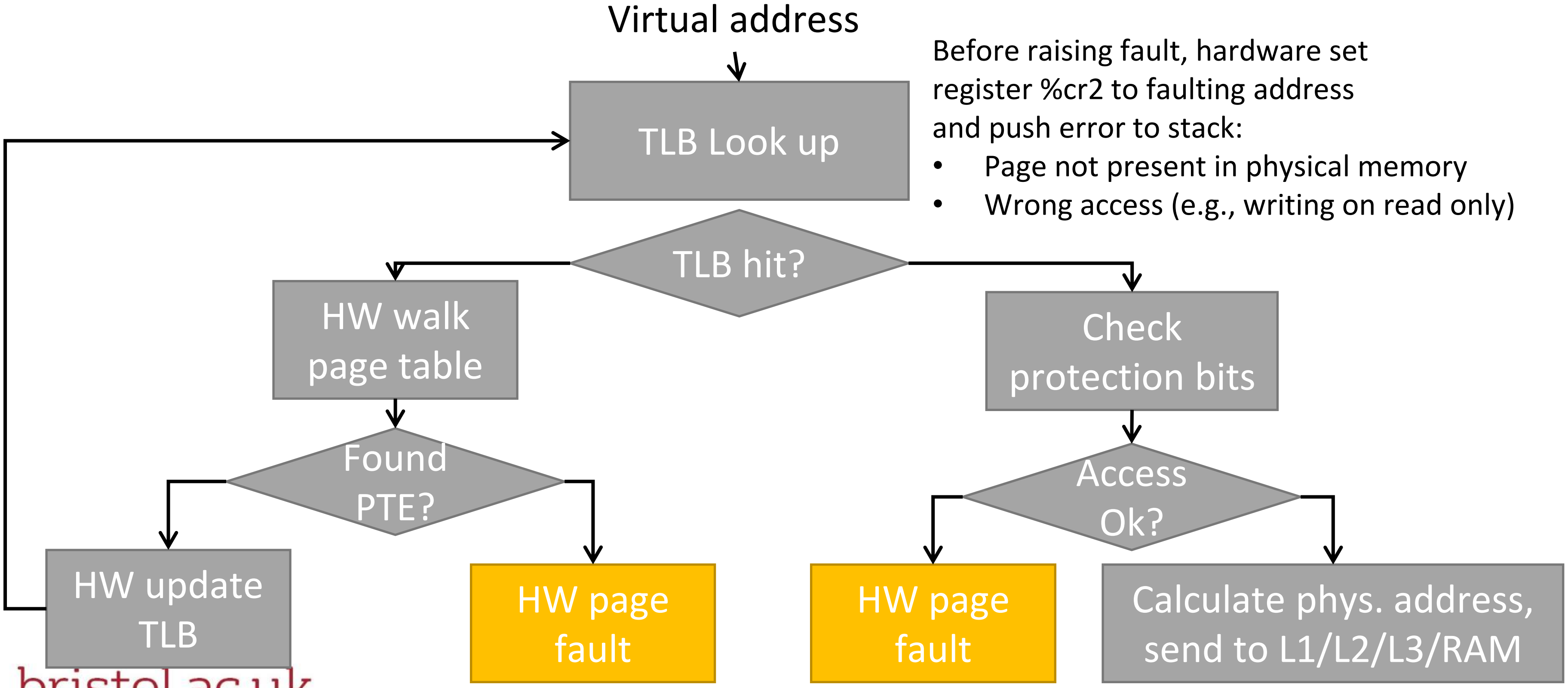
Why does this work?

- Program exhibit nice locality property
- **Temporal locality:** when a process accesses virtual address x, it is likely to access it again in the future (e.g., variable on the stack)
- **Spatial locality:** when a process accesses a virtual address x, the process is likely to address other addresses close to x (e.g., reading elements of an array on the heap)

Memory lifecycle on x86



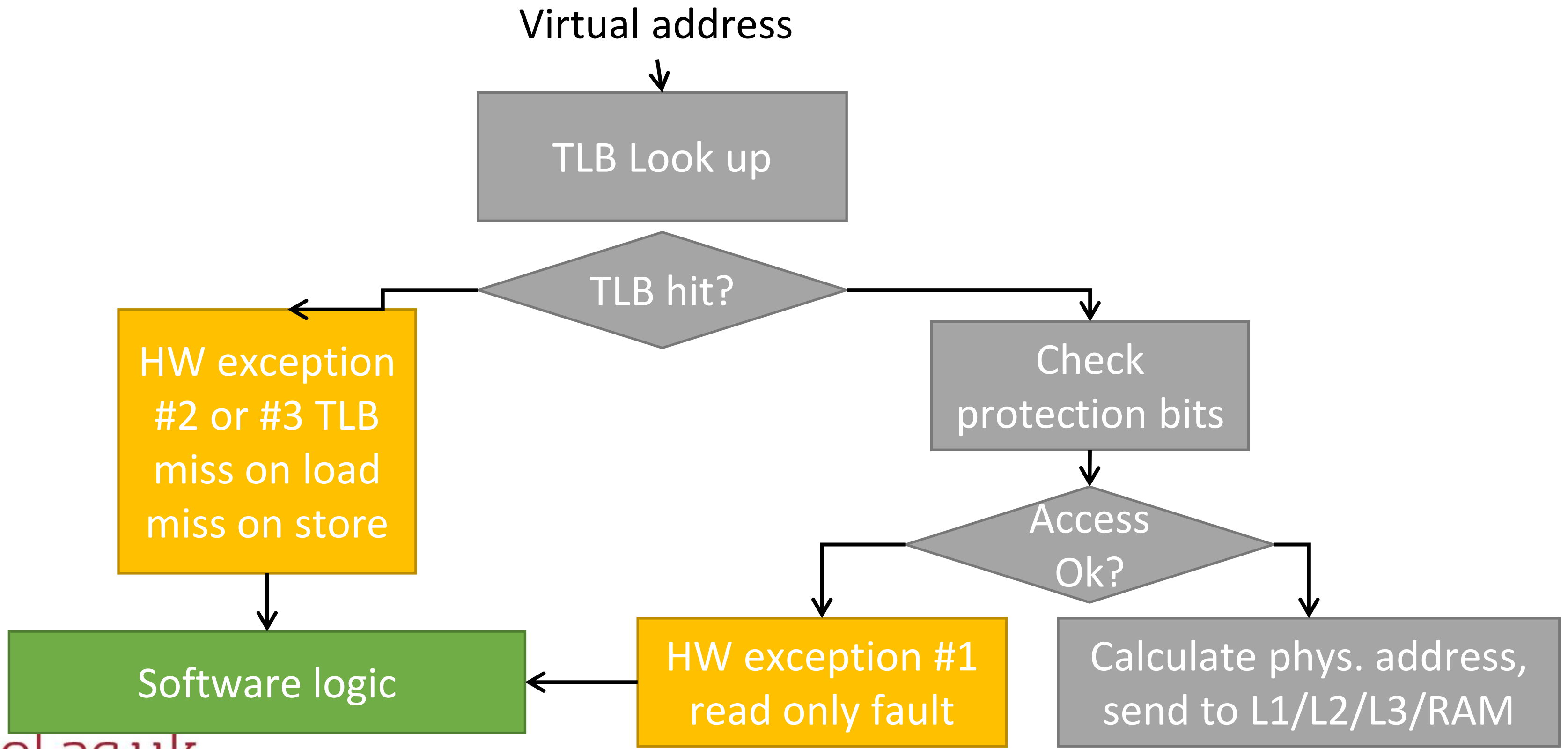
Memory lifecycle on x86



Before raising fault, hardware set register %cr2 to faulting address and push error to stack:

- Page not present in physical memory
- Wrong access (e.g., writing on read only)

Memory lifecycle on MIPS



TLB design trade-offs

- Software TLB
  - Good: freedom to design page directory, page tables and other structures as needed
  - Good: OS can implement TLB eviction policies (i.e., deciding which entry to remove when full)
  - Bad: slower than hardware
- Hardware TLB
  - Good: faster!
  - Bad: OS cannot change the design of page directory, page table etc.

University of BRISTOL

Swapping

Swapping pages out

- Physical RAM may be oversubscribed
  - Total virtual pages greater than the number of physical pages
- Swapping is moving virtual pages from physical RAM to a swap device
  - SSD
  - Hard Drive
  - etc.

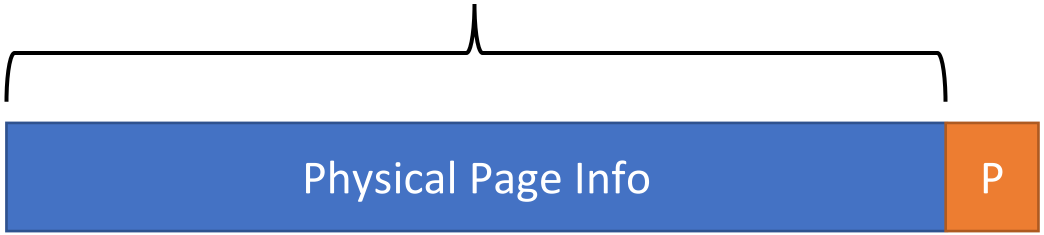
What’s in a page table entry (Linux)

- One bit to state if the memory is present in memory or not

What's in a page table entry (Linux)

- One bit to state if the memory is present in memory or not

If P==0 we do not need this, use it to store swap info



What's in a page table entry (Linux)

- One bit to state if the memory is present in memory or not
- Kernel maintain a list of swap file
- Each file contains several map

What’s in a page table entry (Linux)

- One bit to state if the memory is present in memory or not
- Kernel maintain a list of swap file
- Each file contains several map
- This is greatly simplified, but sufficient
  - Details for interested students: https://www.kernel.org/doc/gorman/html/understand/understand014.html

Page Faults

- When process try to access page not in memory, problem detected because the presence bit is set to 0
  - With **hardware TLB**, the **MMU** detect this when checking the PTE and raise an exception
  - With **software TLB**, the kernel detects the problem on TLB miss, the **TLB should not contain entry for page not present in memory**!
- Attempting to access a page not in RAM is a **page fault**
- The kernel job on page fault is to:
  - Swap the page from secondary storage to memory, evicting another page if necessary
  - Update the PTE (set physical address + presence bit)
  - Return from the exception so the application can try again

Page Faults are slow!

- Accessing secondary storage is slow
  - Millisecond for harddrive
  - Microsecond for SSD
  - … comparing to nanoseconds for RAM
- Suppose secondary storage is 1000 time slower
  - 1 in 10 access results in page fault -> Average access 100 times slower
  - 1 in 100 access results in page fault -> Average access 10 times slower
  - 1 in 1000 access results in page fault -> Average access 2 times slower
- Goal is to reduce occurrence of page faults
  - Limit the number of processes, so that there is enough RAM
  - Hide latencies by prefetching a page before a process needs them
  - Be clever about which page is kept in physical memory and which page is evicted

Page Faults are slow!

- Accessing secondary storage is slow
  - Millisecond for harddrive
  - Microsecond for SSD
  - … comparing to nanoseconds for RAM
- Suppose secondary storage is 1000 time slower
  - 1 in 10 access results in page fault -> Average access 100 times slower
  - 1 in 100 access results in page fault -> Average access 10 times slower
  - 1 in 1000 access results in page fault -> Average access 2 times slower
- Goal is to reduce occurrence of page faults
  - Limit the number of processes, so that there is enough RAM
  - Hide latencies by prefetching a page before a process needs them
  - **Be clever about which page is kept in physical memory and which page is evicted**

Simplest replacement policy: FIFO

- What page to evict?
- FIFO: remove the page that has been in memory the longest

Num	1	2	3	4	5	6	7	8	9
Refs	a	b	c	d	a	b	e	a	b
PP1	a	a	a	d	d	d	e	e	e
PP2		b	b	b	a	a	a	a	a
PP3			c	c	c	b	b	b	b
Fault?	x	x	x	x	x	x	x		

Optimum replacement policy: MIN

- What page to evict?
- MIN: replace the page that won't be referenced for the longest

Num	1	2	3	4	5	6	7	8	9
Refs	a	b	c	d	a	b	e	a	b
PP1	a	a	a	a	a	a	a	a	a
PP2		b	b	b	b	b	b	b	b
PP3			c	d	d	d	e	e	e
Fault?	x	x	x	x			x		

Least recently used (LRU) replacement policy

- What page to evict?
- LRU : remove the page that has been used the least recently (temporal locality)

Num	1	2	3	4	5	6	7	8	9
Refs	a	b	c	d	a	b	e	a	b
PP1	a	a	a	d	d	d	e	e	e
PP2		b	b	b	a	a	a	a	a
PP3			c	c	c	b	b	b	b
Fault?	x	x	x	x	x	x	x		

Practical replacement policy: Clock

- What page to evict?
- Add a “used” bit to PTE
  - Set by MMU when page accessed
  - Can be cleared by kernel

victim = 0

while use bit of victim is set

clear use bit of victim

victim = (victim + 1) % num_frames

evict victim

Practical replacement policy: Clock

Num	1	2	3	4	5	6	7	8	9
Refs	a	b	c	d	a	b	e	a	b
PP1	a	a	a	d	d	d	e	e	e
PP2		b	b	b	a	a	a	a	a
PP3			C	c	c	b	b	b	b
Fault?	x	x	x	x	x	x	x		

University of BRISTOL

OS161

bristol.ac.uk

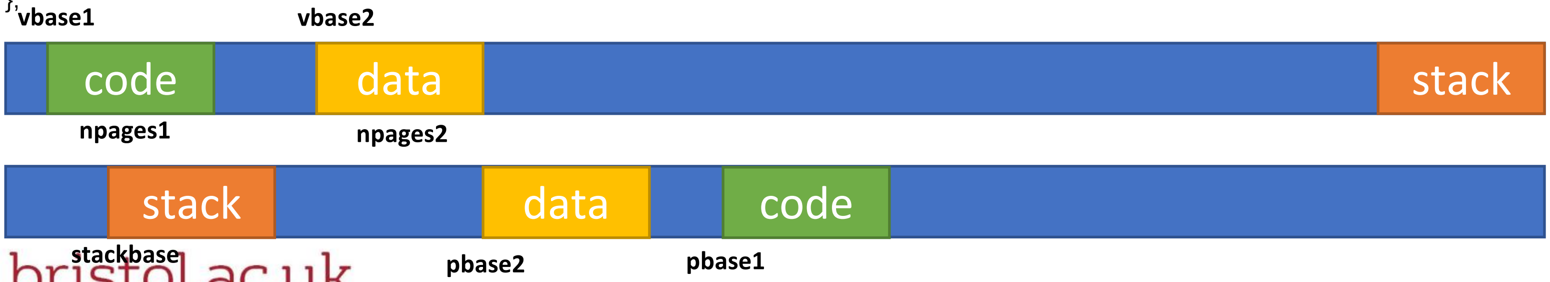
MIPS

- MIPS uses 32bits paged virtual and physical address
- MIPS has software managed TLB
  - Software TLB raises exception on every miss
  - Kernel is free to record virtual to physical mapping
  - TLB functions are handled by a function called vm_fault
    - kern/arch/mips/vm/dumbvm.c line 146
- vm_fault uses information from addrspace structure to determine virtual to physical mapping to load into the TLB
  - Each process has its own *addrspace* structure
  - Each *addrspace* structure describe where the pages are stored in physical memory
  - *addrspace* does the same job as a page table, but in a much simpler way. OS161 create contiguous segment.

OS161 address space

kern/include/addrspace.h

```
struct addrspace {
        vaddr_t as_vbase1; /* base virtual address of code segment */
        paddr_t as_pbase1; /* base physical address of code segment */
        size_t as_npages1; /* size (in pages) of code segment */
        vaddr_t as_vbase2; /* base virtual address of data segment */
        paddr_t as_pbase2; /* base physical address of data segment */
        size_t as_npages2; /* size (in pages) of data segment */
        paddr_t as_stackpbase; /* base physical address of stack */
};
```



dumbvm Address Translation

```
vbase1 = as->as_vbase1;
vtop1 = vbase1 + as->as_npages1 * PAGE_SIZE;
vbase2 = as->as_vbase2;
vtop2 = vbase2 + as->as_npages2 * PAGE_SIZE;
stackbase = USERSTACK - DUMBVM_STACKPAGES * PAGE_SIZE;
stacktop = USERSTACK;

if (faultaddress >= vbase1 && faultaddress < vtop1) {
        paddr = (faultaddress - vbase1) + as->as_pbase1;
} else if (faultaddress >= vbase2 && faultaddress < vtop2) {
        paddr = (faultaddress - vbase2) + as->as_pbase2;
} else if (faultaddress >= stackbase && faultaddress < stacktop) {
        paddr = (faultaddress - stackbase) + as->as_stackpbase;
} else {
        return EFAULT;
}
```

- USERSTACK = 0x8000 0000
- DUMBVM STACKPAGES = 12
- PAGE SIZE = 4KB

kern/arch/mips/vm/dumbvm.c

Line 202

- Line 222 – 239 update TLB

Initializing address space

- When the kernel creates a process it:
  - Creates an address space
  - Load the program data and code
- OS161 pre-load the programs in RAM
  - Most OS will load on demand
- A program code and data is described in an **executable**
- OS161 uses ELF (executable link format) as other OS (e.g., LINUX)
- OS161 execv system call reinitializes the address space of a process
  - int execv(const char *program, char **args)
- The program parameter should be the name of the ELF executable to be loaded

ELF files

- ELF files contain address space segment descriptions
  - ELF header describes the segment images
    - the virtual address of the start of the segment
    - the length of the segment in the virtual address space t
    - the location of the segment in the ELF
    - the length of the segment in the ELF
- the ELF file identifies the (virtual) address of the program’s first instruction (the entry point)
- the ELF file also contains lots of other information (e.g., section descriptors, symbol tables) that is useful to compilers, linkers, debuggers, loaders and other tools used to build programs

OS161

- OS161's dumbvm implementation assumes that an ELF file contains **two segments**
  - a **text segment**, containing the program code any read-only data
  - a **data segment**, containing any other global program data
- the images in the ELF file are an **exact copy** of the binary data to be stored in the address
- dumbvm creates a **stack segment** for each process
  - 12 pages long
  - ending at virtual address 0x7FFFFFFF

OS161

- If the image is smaller than the segment it is in loaded into, it should be zero filled
- Look through and understand: *kern/syscall/loadelf.c*

Thank you