last time

cache misses and C code
  size of way $=$ distance between same set

K-way set-associative caches
  like K direct-mapped caches 'stapled together'
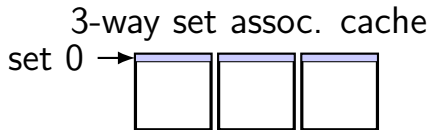 still divide addresses into tag/index/offset
  index identifies set with K blocks
  store valid bit+tag for each block

replacement policies least recently used + related
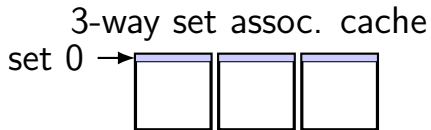
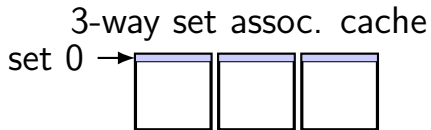# anonymous feedback

mapping of sets to memory (3-way)

mapping of sets to memory (3-way)

3-way set assoc. cache

set 0 →

memory

mapping of sets to memory (3-way)

3-way set assoc. cache

set 0

memory

mapping of sets to memory (3-way)

3-way set assoc. cache

set 0 →

memory
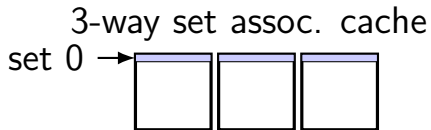
array[0]

array[X] →

where $X = \dfrac{\text{way size}}{\text{array element size}}$

accesses (way size) bytes apart in array?
beware conflict misses!
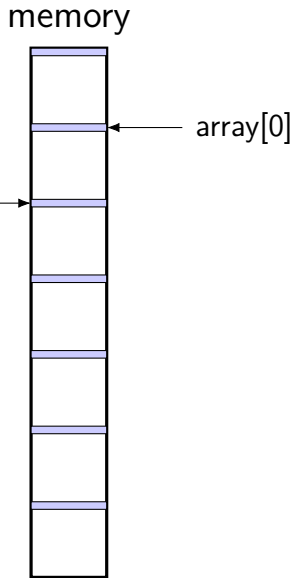
handling writes

what about writing to the cache?

two decision points:

if the value is not in cache, do we add it?
    if yes: need to load rest of block
    if no: missing out on locality?

if value is in cache, when do we update next level?
    if immediately: extra writing
    if later: need to remember to do so

allocate on write?

processor writes less than whole cache block

block not yet in cache

two options:

write-allocate
    fetch rest of cache block, replace written part
    (then follow write-through or write-back policy)

write-no-allocate
    don't use cache at all (send write to memory *instead*)
    guess: not read soon?
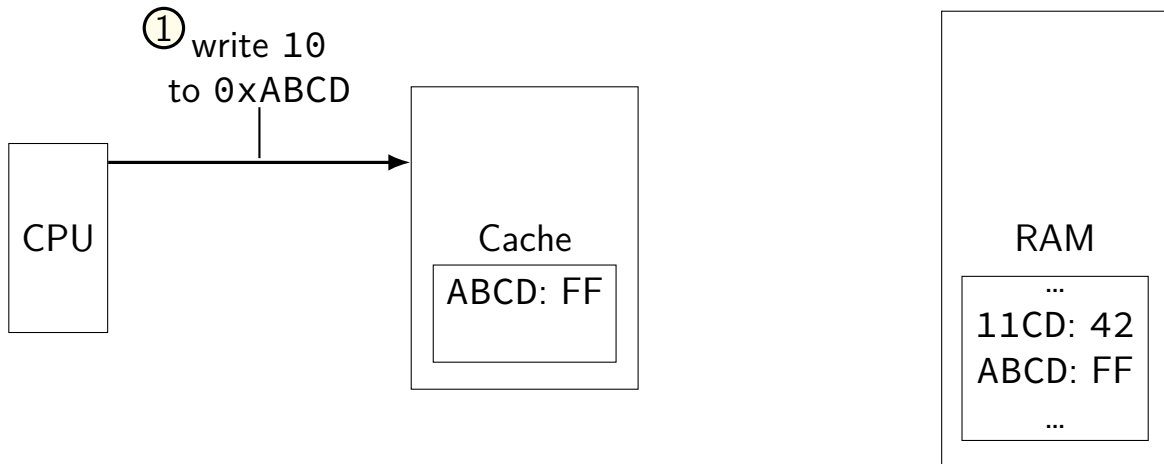
6

write-through v. write-back

option 1: write-through

write-through v. write-back

option 1: write-through
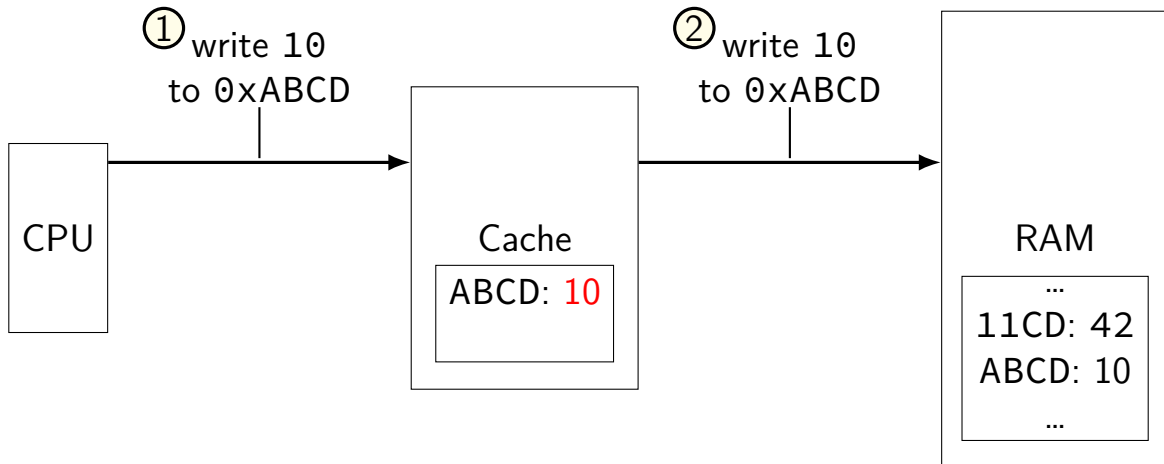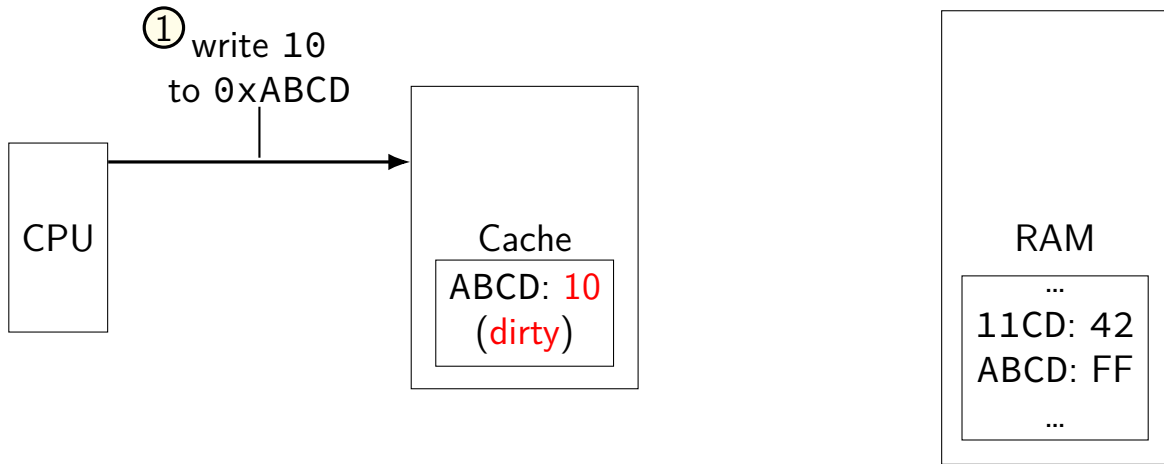
write-through v. write-back

option 2: write-back

write-through v. write-back

option 2: write-back

CPU

① write 10 to 0xABCD

② read from 0x11CD (conflicts)

Cache

ABCD: 10 ~~(dirty)~~

… when replaced — send value to memory

③ write 10 to ABCD

RAM

…
11CD: 42
ABCD: 10
…
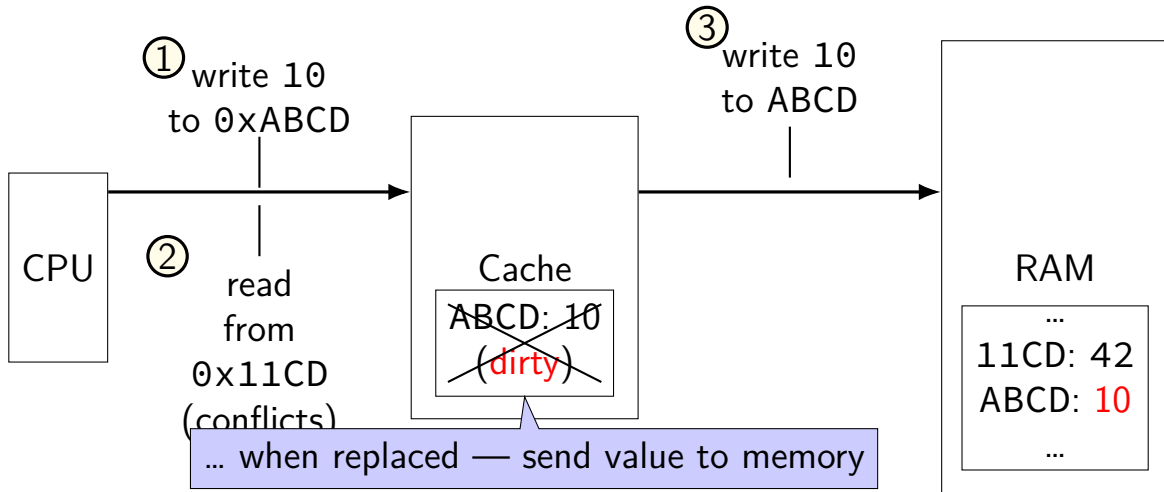
write-through v. write-back
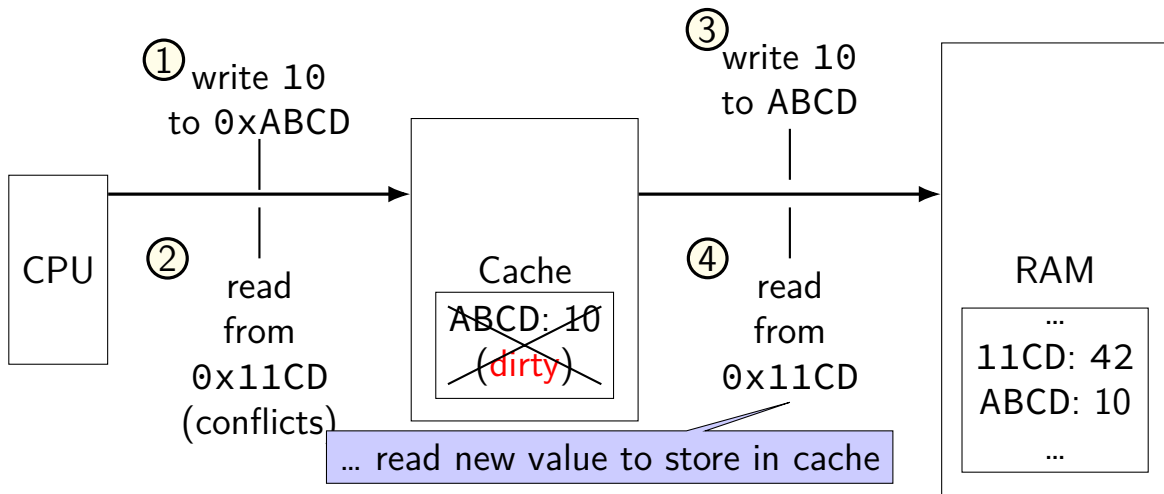
writeback policy

changed value!

2-way set associative, 4 byte blocks, 2 sets

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]* mem[0x61]*	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

1 = dirty (different than memory)
needs to be written if evicted

write-allocate + write-back

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]* mem[0x61]*	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing 0xFF into address 0x04?
index 0, tag 000001

write-allocate + write-back

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]* mem[0x61]*	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing 0xFF into address 0x04?
index 0, tag 000001
step 1: find least recently used block

9

write-allocate + write-back

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]* mem[0x61]*	~~1~~	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing 0xFF into address 0x04?
index 0, tag 000001
step 1: find least recently used block
step 2: possibly writeback old block

write-allocate + write-back

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	000001	0xFF mem[0x05]	1	0
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing 0xFF into address 0x04?
index 0, tag 000001
step 1: find least recently used block
step 2: possibly writeback old block
step 3a: read in new block – to get mem[0x05]
step 3b: update LRU information

write-no-allocate + write-back

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]* mem[0x61]*	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing 0xFF into address 0x04?
step 1: is it in cache yet?
step 2: no, just send it to memory

exercise (1)

2-way set associative, LRU, write-allocate, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	001100	mem[0x30] mem[0x31]	0	1	010000	mem[0x40]⋆ mem[0x41]⋆	1	0
1	1	011000	mem[0x62] mem[0x63]	0	1	001100	mem[0x32]⋆ mem[0x33]⋆	1	1

for each of the following accesses, performed alone, would it require (a) reading a value from memory (or next level of cache) and (b) writing a value to the memory (or next level of cache)?

    writing 1 byte to 0x33
    reading 1 byte from 0x52
    reading 1 byte from 0x50

exercise (2)

2-way set associative, LRU, write-no-allocate, write-through

| index	valid	tag	value	valid	tag	value	LRU
0	1	001100	mem[0x30] mem[0x31]	1	010000	mem[0x40] mem[0x41]	0
1	1	011000	mem[0x62] mem[0x63]	1	001100	mem[0x32] mem[0x33]	1

for each of the following accesses, performed alone, would it require (a) reading a value from memory and (b) writing a value to the memory?

> writing 1 byte to 0x33
> reading 1 byte from 0x52
> reading 1 byte from 0x50

fast writes

write 10
to 0xABCD

write 20
to 0x1234

CPU

Cache

write buffer

0xABCD: 10
0x1234: 20

RAM
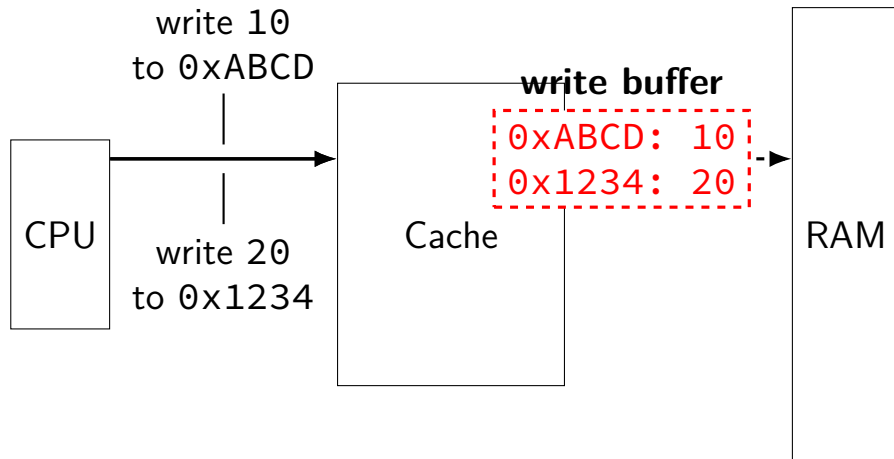
write appears to complete immediately when placed in buffer
memory can be much slower

making any cache look bad

1. access enough blocks, to fill the cache

2. access an additional block, replacing something

3. access last block replaced

4. access last block replaced

5. access last block replaced

…

but — typical real programs have locality

cache optimizations

(assuming typical locality + keeping cache size constant if possible…)

	miss rate	hit time	miss penalty
increase cache size	better	worse	—
increase associativity	better	worse	worse?
increase block size	depends	worse	worse
add secondary cache	—	—	better
write-allocate	better	—	?
writeback	—	—	?
LRU replacement	better	?	worse?
prefetching	better	—	—

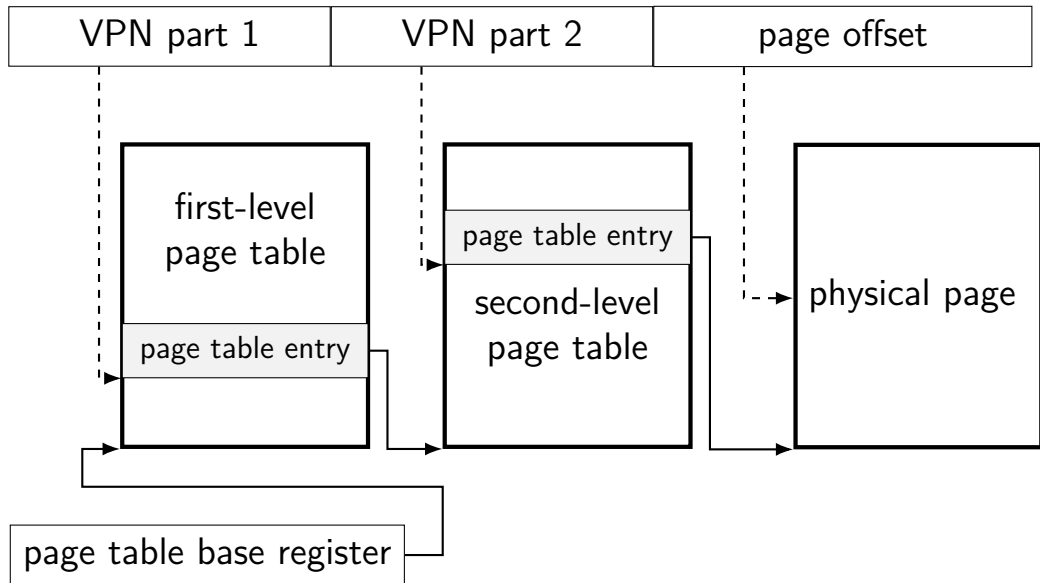prefetching = guess what program will use, access in advance

average time = hit time + miss rate × miss penalty
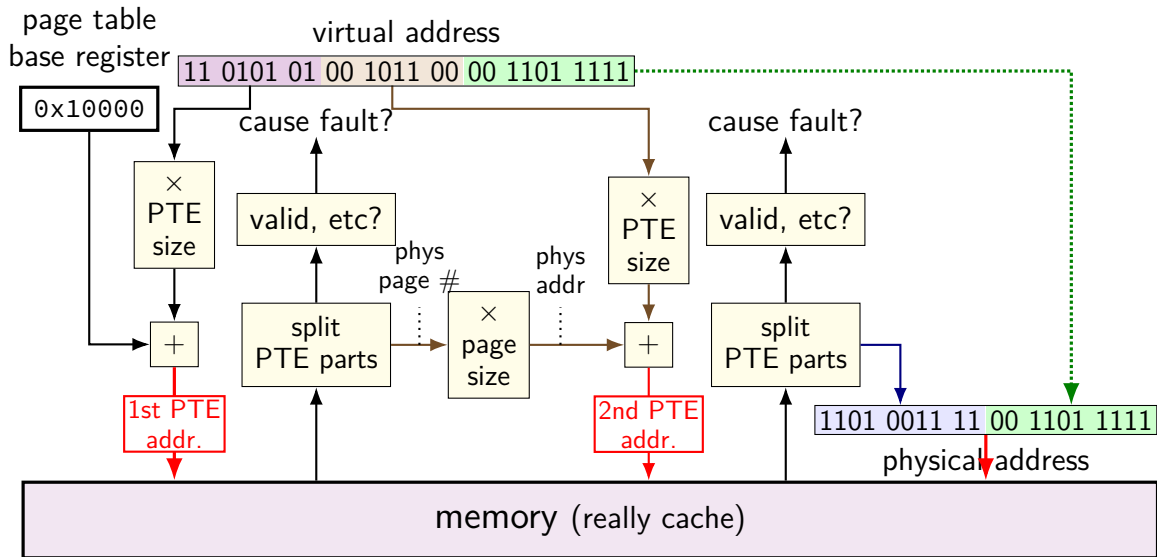
cache optimizations by miss type

(assuming other listed parameters remain constant)

	capacity	conflict	compulsory
increase cache size	fewer misses	fewer misses	—
increase associativity	—	fewer misses	—
increase block size	more misses?	more misses?	fewer misses
LRU replacement	—	fewer misses	—
prefetching	—	—	fewer misses

another view

VPN part 1	VPN part 2	page offset

first-level
page table

page table entry

second-level
page table

page table entry

physical page

page table base register

two-level page table lookup
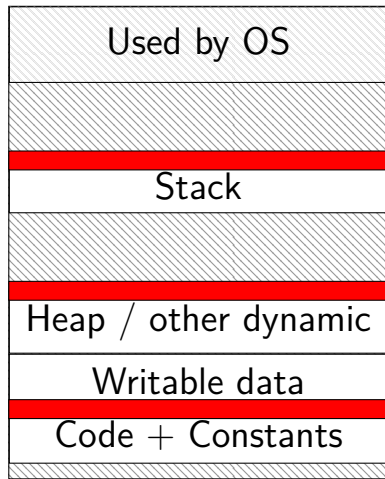
cache accesses and multi-level PTs

four-level page tables — five cache accesses per program memory access

L1 cache hits — typically a couple cycles each?

so add 8 cycles to each program memory access?

not acceptable

program memory active sets

Used by OS
Stack
Heap / other dynamic
Writable data
Code + Constants

0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F...

small areas of memory active at a time
one or two pages in each area?

0x0000 0000 0040 0000

23

page table entries and locality

page table entries have excellent temporal locality

typically one or two pages of the stack active

typically one or two pages of code active

typically one or two pages of heap/globals active

each page contains whole functions, arrays, stack frames, etc.

page table entries and locality

page table entries have excellent temporal locality

typically one or two pages of the stack active

typically one or two pages of code active

typically one or two pages of heap/globals active

each page contains whole functions, arrays, stack frames, etc.

needed page table entries are very small

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

only caches the page table lookup itself
(generally) just entries from the last-level page tables

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

virtual page number divided into
index + tag

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

not much spatial locality between page table entries
(they're used for kilobytes of data already)

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

0 block offset bits

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

few active page table entries at a time
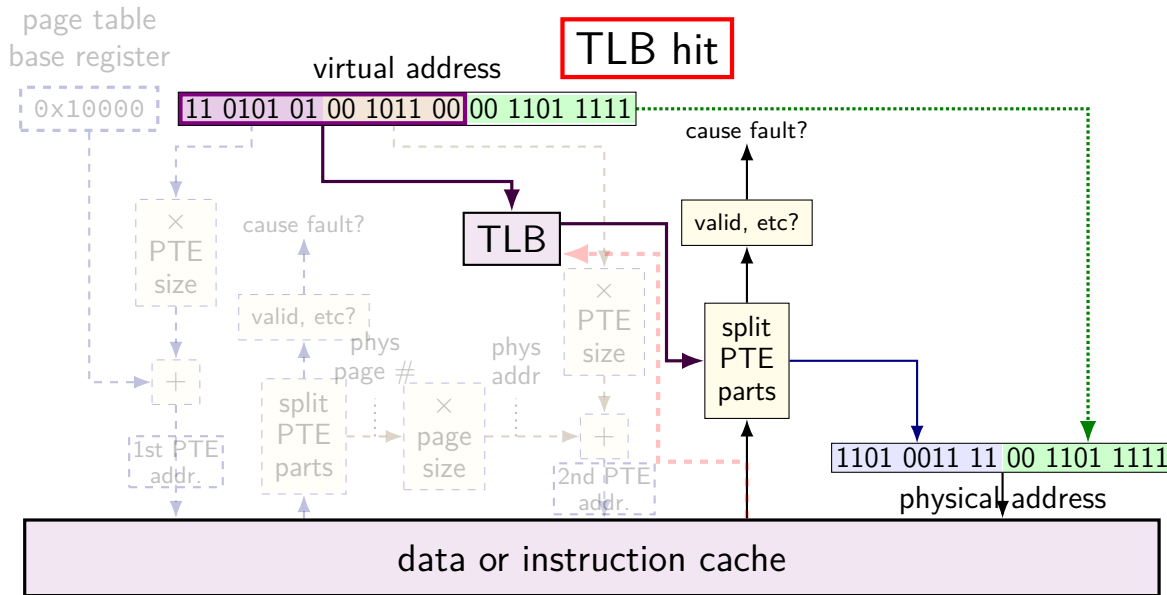enables highly associative cache designs

TLB and multi-level page tables

TLB caches valid last-level page table entries
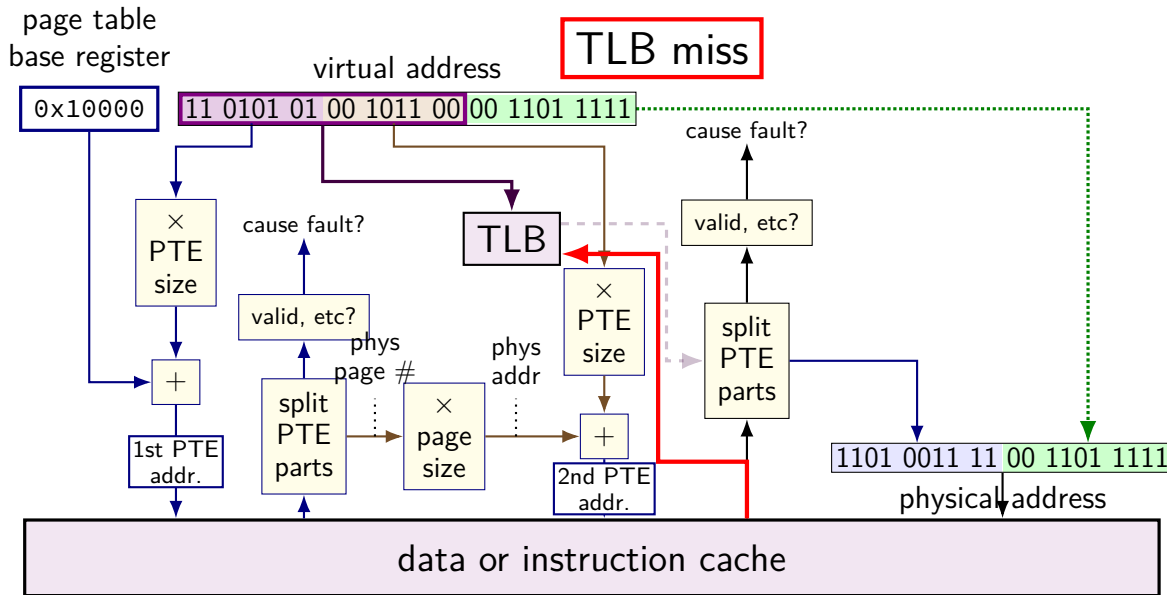
doesn't matter which last-level page table

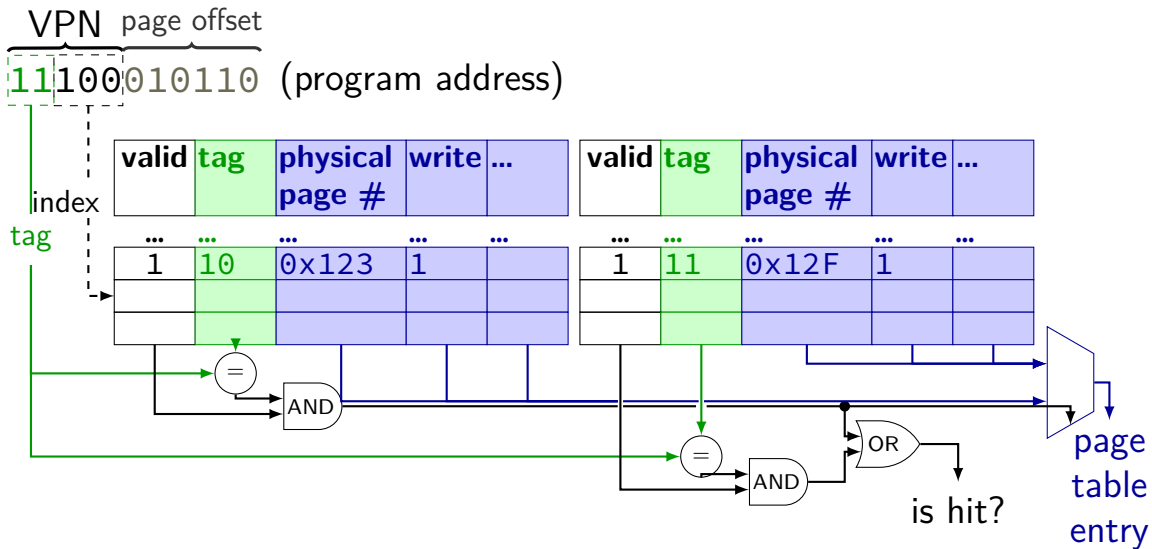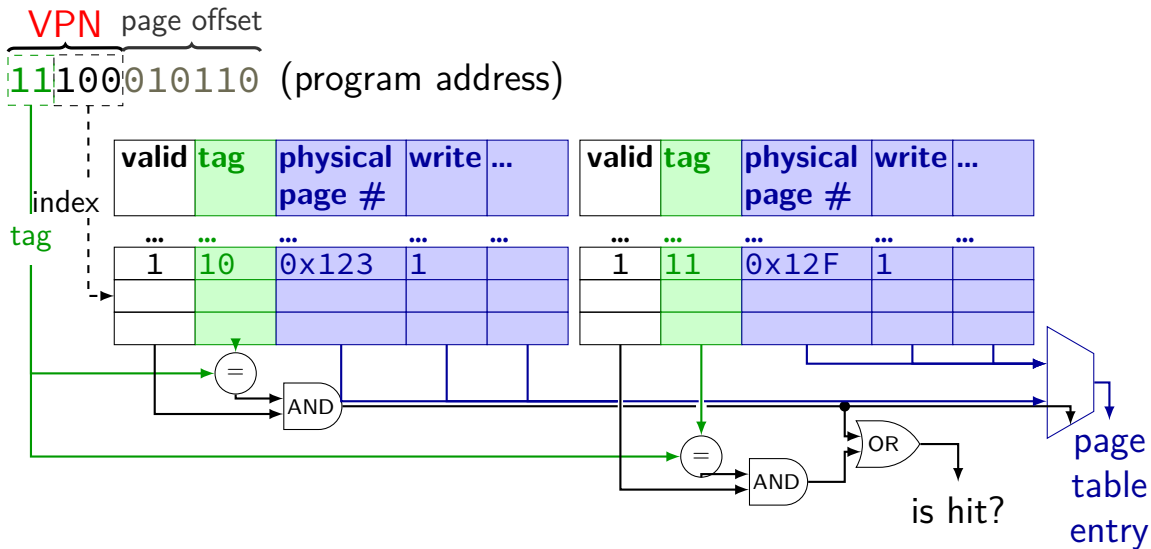means TLB output can be used directly to form address

TLB and two-level lookup

page table
base register

`0x10000`

virtual address

TLB hit

`11 0101 01 00 1011 00` `00 1101 1111`

cause fault?

× PTE size

cause fault?

valid, etc?

TLB

+

valid, etc?

phys page #

× PTE size

phys addr

split PTE parts

split PTE parts

× page size

+

cause fault?

valid, etc?

split PTE parts

1st PTE addr.

2nd PTE addr.

`1101 0011 11` `00 1101 1111`

physical address

data or instruction cache

27

TLB and two-level lookup

page table base register

`0x10000`

virtual address

`11 0101 01 00 1011 00 00 1101 1111`

TLB miss

× PTE size

cause fault?

valid, etc?

1st PTE addr.

split PTE parts

phys page #

× page size

phys addr

TLB

× PTE size

2nd PTE addr.

+

cause fault?

valid, etc?

split PTE parts

`1101 0011 11 00 1101 1111`

physical address

data or instruction cache

TLB organization (2-way set associative)

TLB organization (2-way set associative)

VPN page offset

11 100 010110 (program address)

index

tag

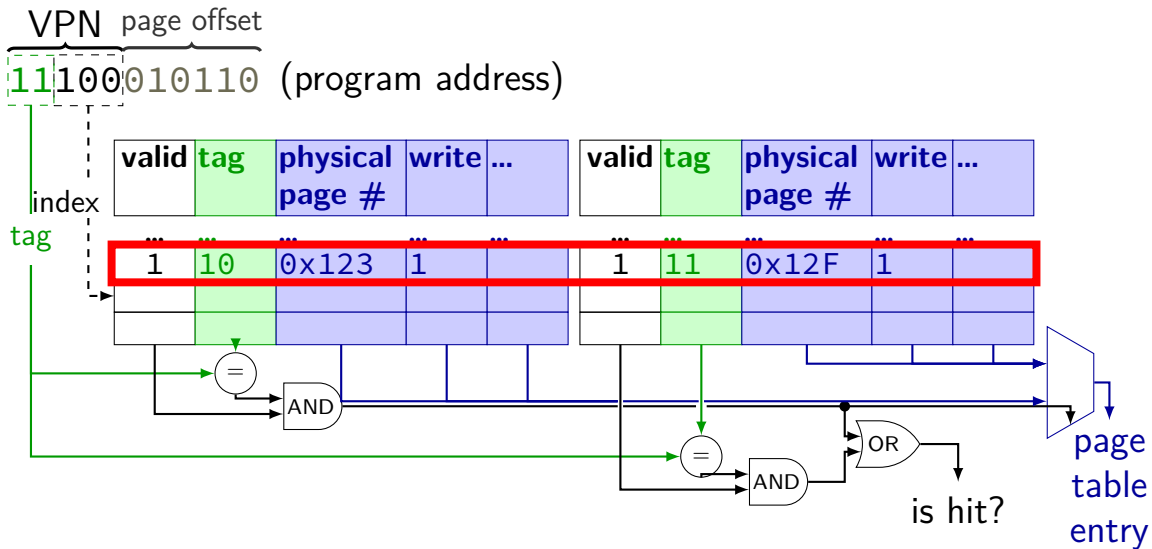| valid	tag	physical page #	write	...		valid	tag	physical page #	write	...
...	...	...	...	...		...	...	...	...	...
1	10	0x123	1			1	11	0x12F	1	

AND

=

AND

OR

is hit?

page table entry

TLB organization (2-way set associative)

TLB organization (2-way set associative)

TLB organization (2-way set associative)

why threads?

concurrency: different things happening at once
  one thread per user of web server?
  one thread per page in web browser?
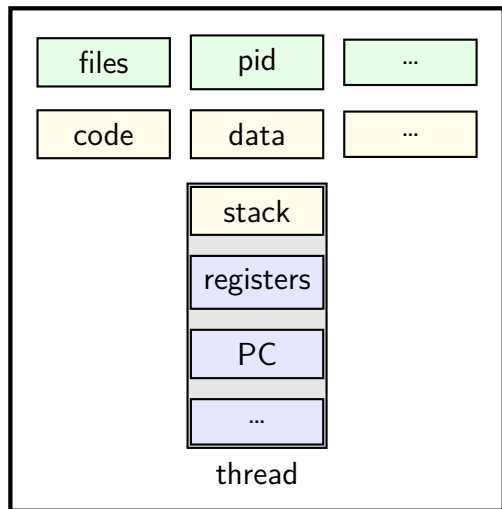 one thread to play audio, one to read keyboard, …?
  …

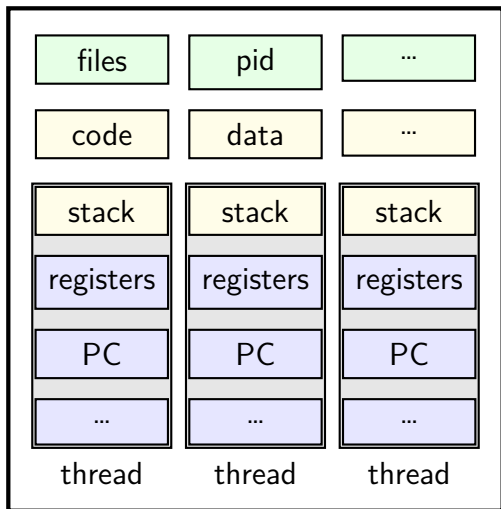parallelism: do same thing with more resources
  multiple processors to speed-up simulation (life assignment)

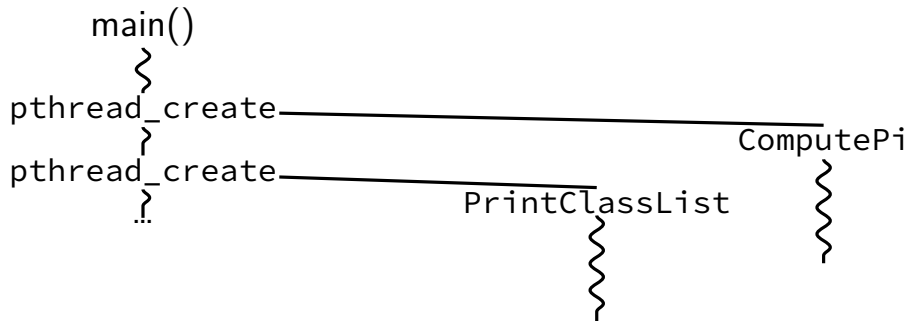# single and multithread processes

single-threaded process

multi-threaded process

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

main()

pthread_create ————————————————————————————
                                        ComputePi
pthread_create ————————————
                        PrintClassList
...

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run thread starts here, terminates if this function returns

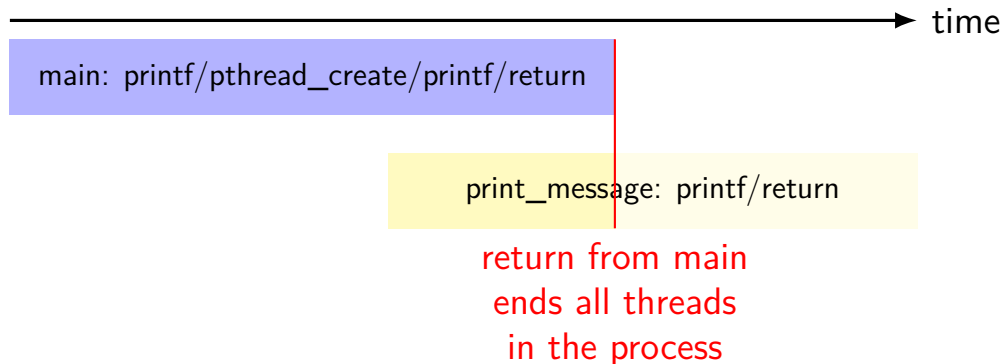thread attributes (extra settings) and function argument

a threading race

```c
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n"); return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    return 0;
}
```

My machine: outputs In the thread about 4% of the time.
What happened?

a race

returning from main exits the entire process (all its threads)
    same as calling exit; not like other threads

race: main's return 0 or print_message's printf first?

time

main: printf/pthread_create/printf/return

print_message: printf/return

return from main
ends all threads
in the process

fixing the race (version 1)

```c
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n");
    return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    pthread_join(the_thread, NULL);  /* WAIT FOR THREAD */
    return 0;
}
```

fixing the race (version 2; not recommended)

```c
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n");
    return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    pthread_exit(NULL);
}
```

pthread_join, pthread_exit

pthread_join: wait for thread, retrieves its return value
    like waitpid, but for a thread
    return value is pointer to anything

pthread_exit: exit current thread, returning a value
    like exit or returning from main, but for a single thread
    same effect as returning from function passed to pthread_create

sum example (only globals)

```
int values[1024];
int results[2];
void *sum_front(void *ignored_argument) {
    int sum = 0;
    for (int i = 0; i < 512; ++i) { sum += values[i]; }
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL); pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

sum example (only globals)

```
int values[1024];
int results[2];
void *sum_front(void *ignored_argument) {
    int sum = 0;
    for (int i = 0; i < 512; ++i) { sum += values[i]; }
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL); pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

values, results: global variables — shared

sum example (only globals)

two different functions
happen to be the same except for some numbers
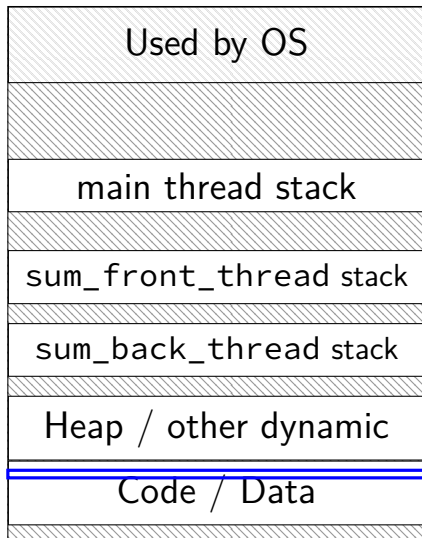
```c
int values[1024];
int results[2];
void *sum_front(void *ignored_argument) {
    int sum = 0;
    for (int i = 0; i < 512; ++i) { sum += values[i]; }
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL); pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

sum

values returned from threads
via global array instead of return value
(partly to illustrate that memory is shared,
partly because this pattern works when we don't join (later))

```c
int valu
int resu
void *su
    int
    for (int i = 0; i < 512; ++i) { sum += values[i]; }
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL); pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

thread_sum memory layout

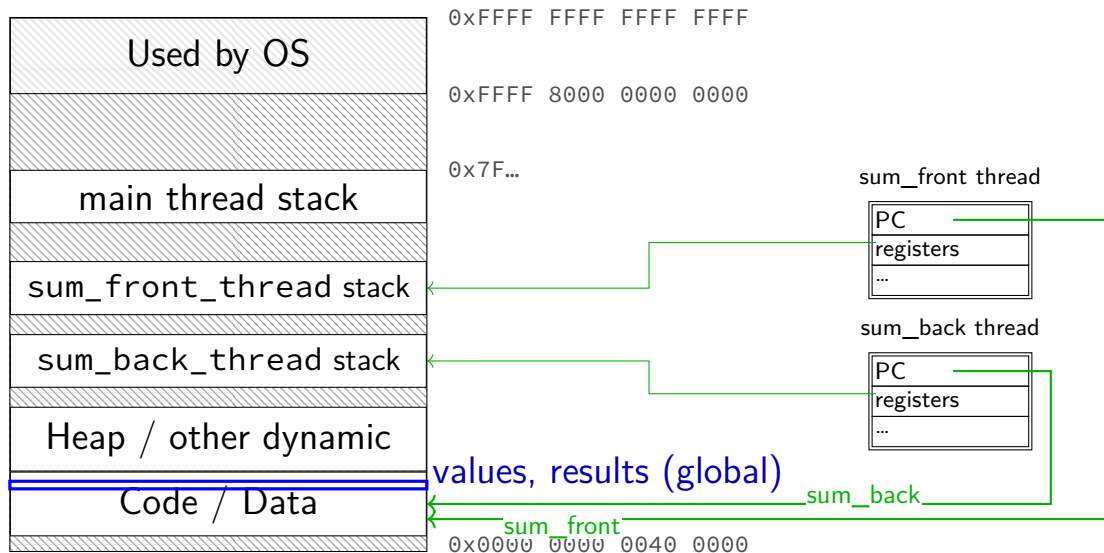|	
Used by OS	0xFFFF FFFF FFFF FFFF
	0xFFFF 8000 0000 0000
	0x7F…
main thread stack	
sum_front_thread stack	
sum_back_thread stack	
Heap / other dynamic	
Code / Data	values, results (global)
	0x0000 0000 0040 0000

41

thread_sum memory layout

	0xFFFF FFFF FFFF FFFF
Used by OS	
	0xFFFF 8000 0000 0000
	0x7F…
main thread stack	
sum_front_thread stack	
sum_back_thread stack	
Heap / other dynamic	
values, results (global)	
Code / Data	
	0x0000 0000 0040 0000

sum_front thread
PC
registers
…

sum_back thread
PC
registers
…

sum_back

sum_front

sum example (to global, with thread IDs)

```c
int values[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += values[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

sum example (to global, with thread IDs)

```
int values[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += values[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

values, results: global variables — shared

sum example (info struct)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    struc tThreadInfo *my_info =  (struct ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) { sum += values[i]; }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; struct ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i) { pthread_join(threads[i], NULL); }
    return info[0].result + info[1].result;
}
```

sum example (info struct)

```
int values[1024];
struct ThreadInfo
    int start, end, result;
};
void *sum_thread(void *argument) {
    struc tThreadInfo *my_info =  (struct ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) { sum += values[i]; }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; struct ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i) { pthread_join(threads[i], NULL); }
    return info[0].result + info[1].result;
}
```

values: global variable — shared

sum example (info struct)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    struc tThreadInfo *my_info = (struct ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start;                                    ; }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; struct ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i) { pthread_join(threads[i], NULL); }
    return info[0].result + info[1].result;
}
```
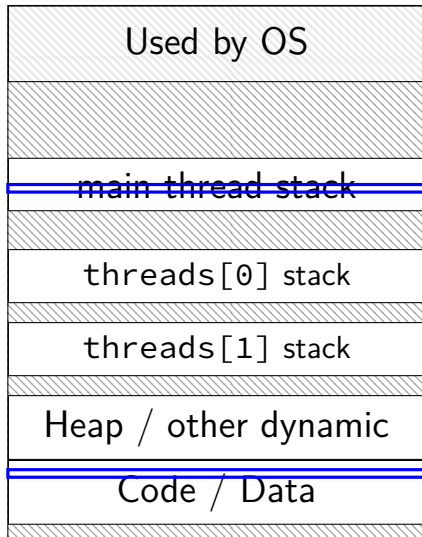
my_info: pointer to sum_all's stack
only okay because sum_all waits!

sum example (info struct)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    struc tThreadInfo *my_info =  (struct ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) { sum += values[i]; }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; struct ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i) { pthread_join(threads[i], NULL); }
    return info[0].result + info[1].result;
}
```

thread_sum memory layout (info struct)

Used by OS	0xFFFF FFFF FFFF FFFF
	0xFFFF 8000 0000 0000
	0x7F...
main thread stack	info array
threads[0] stack	my_info
threads[1] stack	my_info
Heap / other dynamic	
Code / Data	values (global)
	0x0000 0000 0040 0000

sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```
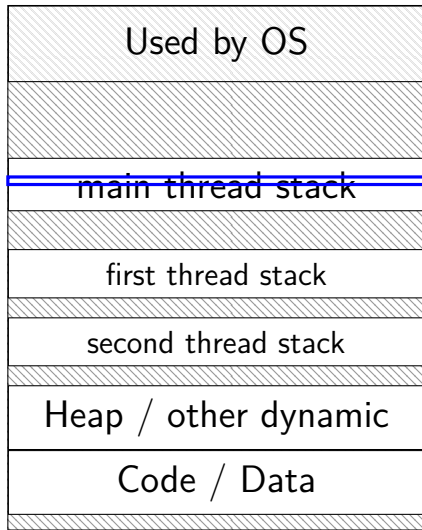
sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

program memory (to main stack)

Used by OS

0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

main thread stack

info array ⟶ values (stack? heap?)

first thread stack

my_info

second thread stack

my_info

Heap / other dynamic

Code / Data

0x0000 0000 0040 0000

sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result
void *sum_thread(void *argument) {
    ...
}

struct ThreadInfo *start_sum_all(int *values) {
    struct ThreadInfo *info = calloc(2, sizeof(struct ThreadInfo));
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    free(info);
    return result;
}
```

sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result
void *sum_thread(void *argument) {
    ...
}

struct ThreadInfo *start_sum_all(int *values) {
    struct ThreadInfo *info = calloc(2, sizeof(struct ThreadInfo);
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    free(info);
    return result;
}
```
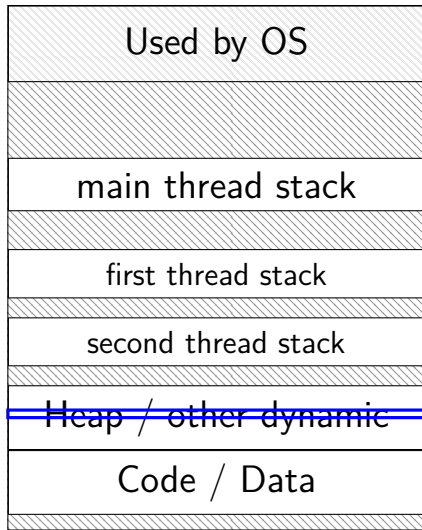
sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result
void *sum_thread(void *argument) {
    ...
}

struct ThreadInfo *start_sum_all(int *values) {
    struct ThreadInfo *info = calloc(2, sizeof(struct ThreadInfo);
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    free(info);
    return result;
}
```

thread_sum memory (heap version)


```
0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…
```

Used by OS

main thread stack

first thread stack

second thread stack

Heap / other dynamic

Code / Data

my_info

my_info

info array ⇄ values (stack? heap?)
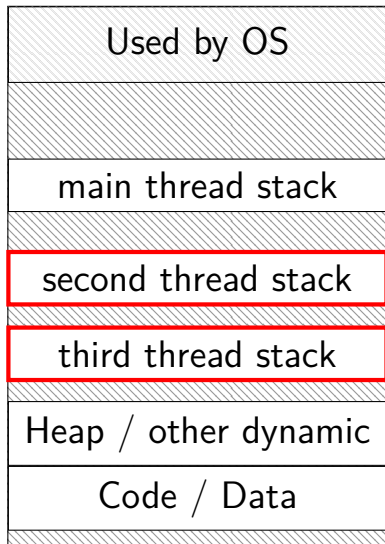
```
0x0000 0000 0040 0000
```

what's wrong with this?

```
/* omitted: headers */
void *create_string(void *ignored_argument) {
  char string[1024];
  ComputeString(string);
  return string;
}
int main() {
  pthread_t the_thread;
  pthread_create(&the_thread, NULL, create_string, NULL);
  char *string_ptr;
  pthread_join(the_thread, (void**) &string_ptr);
  printf("string is %s\n", string_ptr);
}
```

program memory

Used by OS
main thread stack
second thread stack
third thread stack
Heap / other dynamic
Code / Data

`0xFFFF FFFF FFFF FFFF`

`0xFFFF 8000 0000 0000`

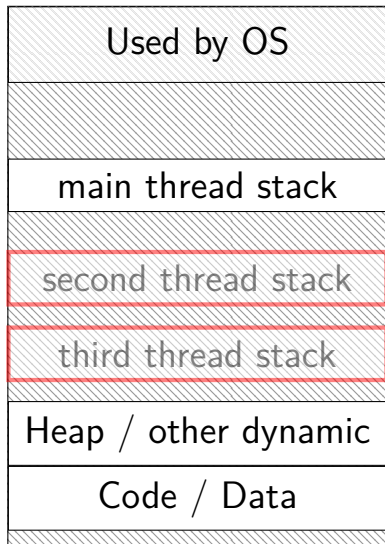`0x7F…`

} dynamically allocated stacks
`string result` allocated here
`string_ptr` pointed to here

…stacks deallocated when
threads exit/are joined

`0x0000 0000 0040 0000`

program memory

Used by OS	`0xFFFF FFFF FFFF FFFF`
	`0xFFFF 8000 0000 0000`
main thread stack	`0x7F…`
second thread stack	dynamically allocated stacks
third thread stack	`string result` allocated here `string_ptr` pointed to here
Heap / other dynamic	…stacks deallocated when threads exit/are joined
Code / Data	
	`0x0000 0000 0040 0000`

thread resources

to create a thread, allocate:

new stack (how big???)

thread control block

deallocated when …

thread resources

to create a thread, allocate:

new stack (how big???)

thread control block

deallocated when …

can deallocate stack when thread exits

but need to allow collecting return value
    same problem as for processes and waitpid

pthread_detach

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_create(&show_progress_thread, NULL,
                   show_progress, NULL);

    /* instead of keeping pthread_t around to join thread later: */
    pthread_detach(show_progress_thread);
}

int main() {
    spawn_show_progress_thread();
    do_other_stuff();
    ...
}
```

detach = don't care about return value, etc.
system will deallocate when thread terminates

starting threads detached

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setdetachstate(&attrs, PTHREAD_CREATE_DETACHED);
    pthread_create(&show_progress_thread, attrs,
                   show_progress, NULL);
    pthread_attr_destroy(&attrs);
}
```

setting stack sizes

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setstacksize(&attrs, 32 * 1024 /* bytes */);
    pthread_create(&show_progress_thread, attrs,
                   show_progress, NULL);
}
```

a note on error checking

from pthread_create manpage:

```
ERRORS
       EAGAIN Insufficient resources to create another thread, or a system-imposed limit on the number of
              threads was encountered. The latter case may occur in two ways: the RLIMIT_NPROC soft resource
              limit (set via setrlimit(2)), which limits the number of process for a real user ID, was
              reached; or the kernel's system-wide limit on the number of threads, /proc/sys/kernel/threads-
              max, was reached.

       EINVAL Invalid settings in attr.

       EPERM  No permission to set the scheduling policy and parameters specified in attr.
```

special constants for *return value*

same pattern for many other pthreads functions

will often omit error checking in slides for brevity

error checking pthread_create

```
int error = pthread_create(...);
if (error != 0) {
    /* print some error message */
}
```

backup slides

exercise: TLB access pattern (setup)

4-entry, 2-way TLB, LRU replacement policy, initially empty

4096 byte pages

how many index bits?

TLB index of virtual address 0x12345?

exercise: TLB access pattern

4-entry, 2-way TLB, LRU replacement policy, initially empty

4096 byte pages

type	virtual	physical
read	0x440030	0x554030
write	0x440034	0x554034
read	0x7FFFE008	0x556008
read	0x7FFFE000	0x556000
read	0x7FFFDFF8	0x5F8FF8
read	0x664080	0x5F9080
read	0x440038	0x554038
write	0x7FFFDFF0	0x5F8FF0

which are TLB hits? which are TLB misses? final contents of TLB?

cache miss types

common to categorize misses:
    roughly "cause" of miss assuming cache block size fixed

compulsory (or *cold*) — first time accessing something
    adding more sets or blocks/set wouldn't change

conflict — sets aren't big/flexible enough
    a fully-associtive (1-set) cache of the same size would have done better

capacity — cache was not big enough

coherence — from sync'ing cache with other caches
    only issue with multiple cores

thread versus process state

thread state
    registers (including stack pointer, program counter)
    …

process state
    address space
    open files
    process id
    list of thread states
    …

process info with threads

parent process info

thread infos	thread 0: {PC = 0x123456, rax = 42, rbx = …} thread 1: {PC = 0x584390, rax = 32, rbx = …} …
page tables	
open files	fd 0: … fd 1: …
…	…

Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared
    e.g. shared open files: open fd 4 in one task → all sharing can use fd 4

`fork()`-like system call "clone": choose what to share
    `clone(0, ...)` — similar to `fork()`
    `clone(CLONE_FILES, ...)` — like fork(), but **sharing** open files
    `clone(CLONE_VM, new_stack_pointer, ...)` — like fork(),
    but **sharing** address space

Linux idea: **task_struct**

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared
    e.g. shared open files: open fd 4 in one task → all sharing can use fd 4

`fork()`-like system call "clone": choose what to share
    `clone(0, ...)` — similar to `fork()`
    `clone(CLONE_FILES, ...)` — like fork(), but **sharing** open files
    `clone(CLONE_VM, new_stack_pointer, ...)` — like fork(),
    but **sharing** address space

advantage: no special logic for threads (mostly)
    two threads in same process = tasks sharing everything possible

aside: alternate threading models

we'll talk about kernel threads

OS scheduler deals **directly** with threads

alternate idea: library code handles threads

kernel doesn't know about threads w/in process

hierarchy of schedulers: one for processes, one within each process

not currently common model — awkward with multicore