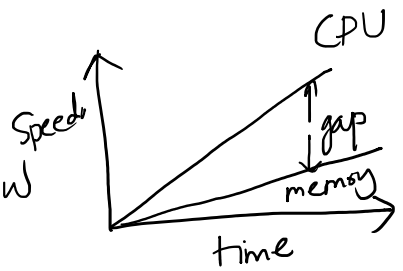
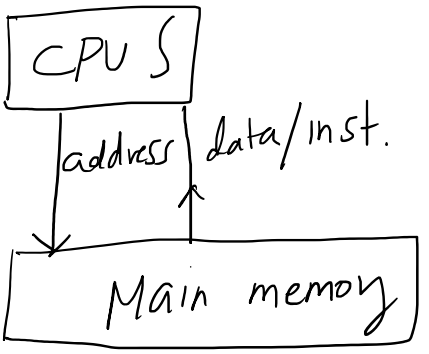
Basics of caches

Tuesday, April 14, 2020 6:28 PM

- CPU ↑ fast
- Main memory ↓ slow





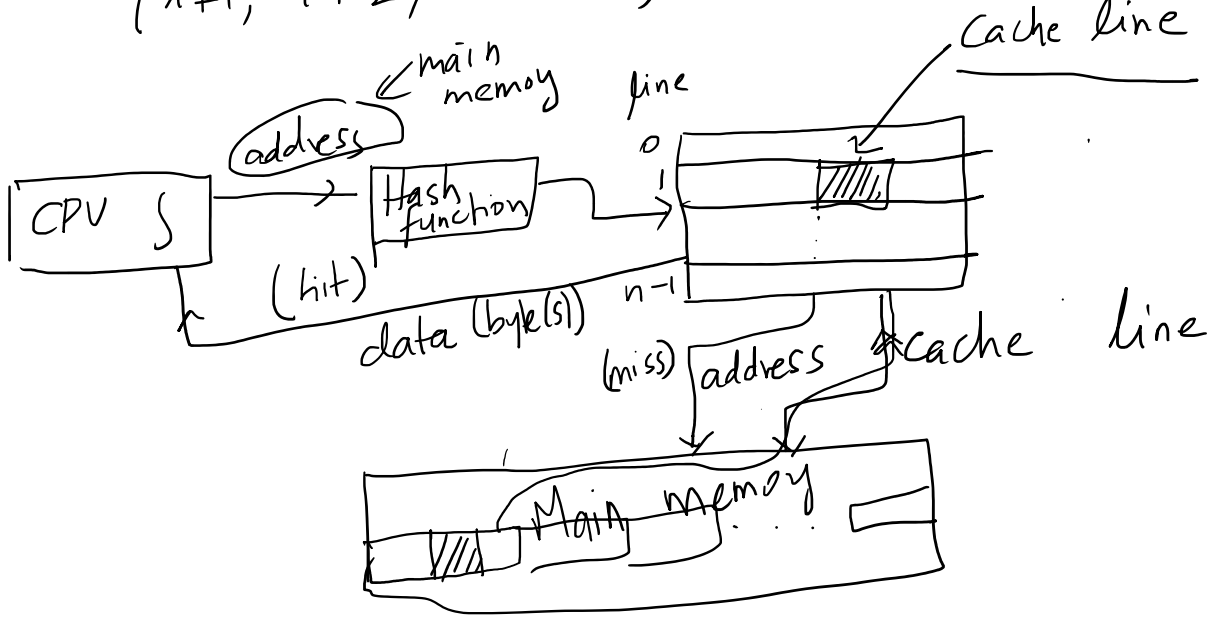
- How do we close gap?
- Cache (hyde)
  - ↳ small, fast, on-chip memory
- Advantage of locality *
  - spatial locality
  - Temporal locality

Example: SAXPY loop

```
for (i=0; i<N; i++)
    y[i] = a*x[i] + y[i];
```

- Spatial locality:

(i+1, i+2, i+3 ....)

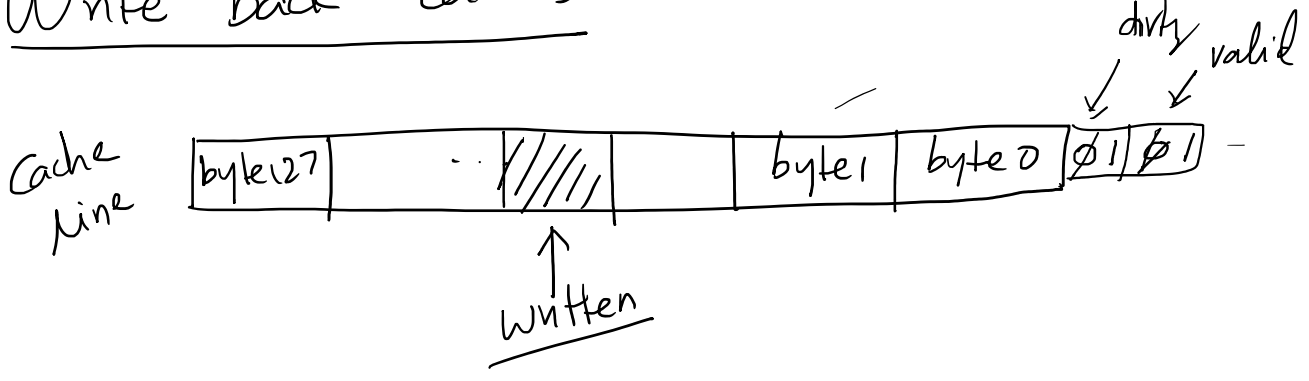


Cache line:

↳ Multiple bytes long
↳ For example, 128 bytes
↳ 32 Floating point items
↳ 4 bytes

* Minimum unit of data that is moved betweent main memory and cache.

- Cache placement problem
  - Direct mapped (no flexibity)
  - Set associative (flexibility)
- Cache replacement problem
  - Direct mapped (no flexibity)
  - Set associatives (Least recently used)
- Write back caches



- Write through caches
  - Writes are made to cache line + propagated to main memory
    ⇓
    Contents of cache always consistent with main memory
    ⇓
    slow.
- We will assume write back cache.

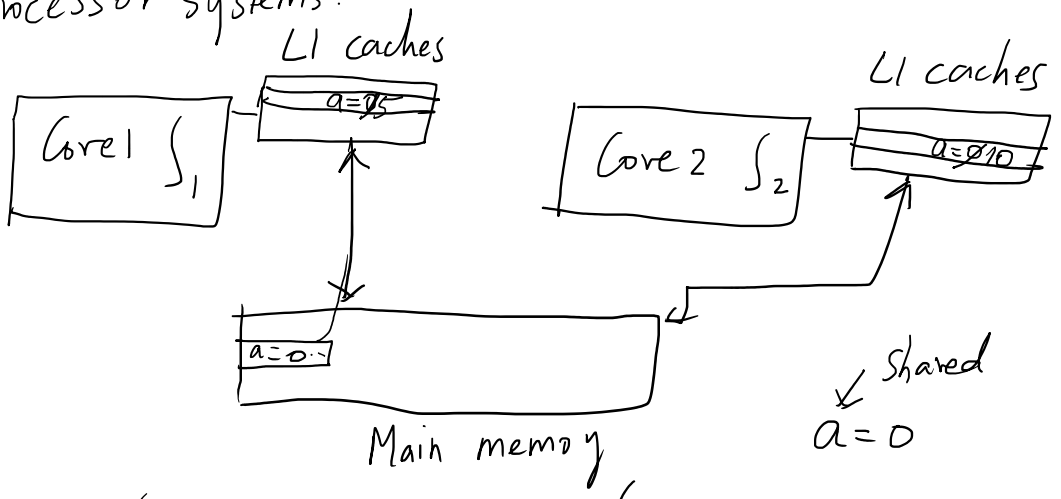
---

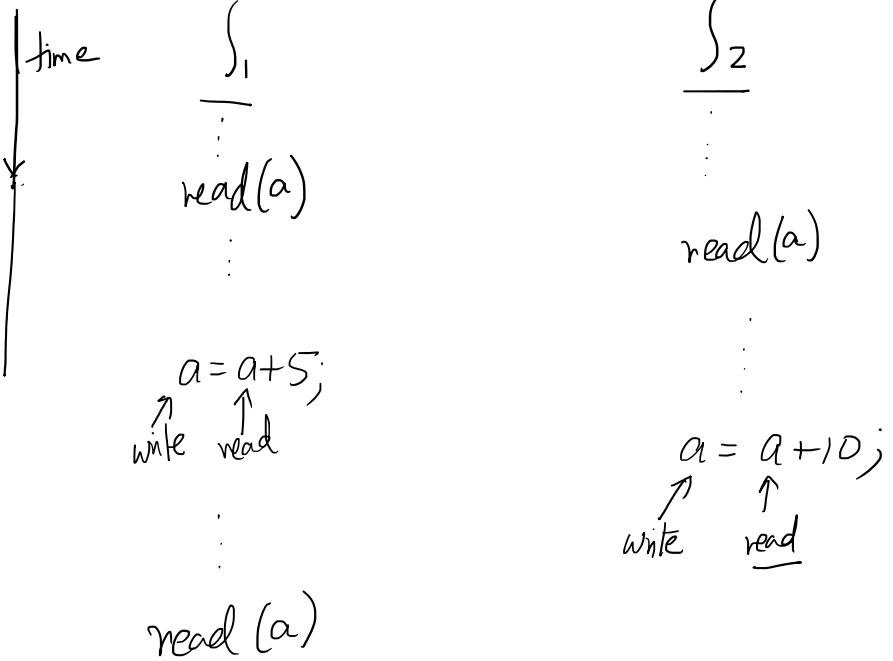
* Cache line
* Write back ||

The cache coherence problem

Tuesday, April 14, 2020 5:52 PM

- Multiprocessor systems.





Programmer expects sequential* or strict* consistency

↓

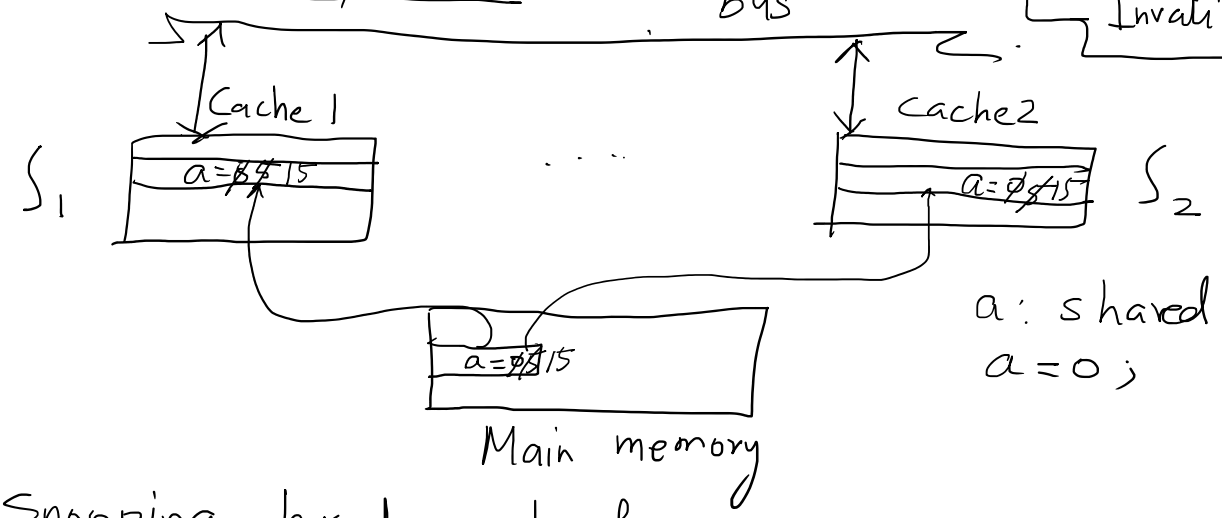
Any read must return the most up to date value.

Example: expect a = 15

How to achieve strict consistency? → Write Invalidation protocol.

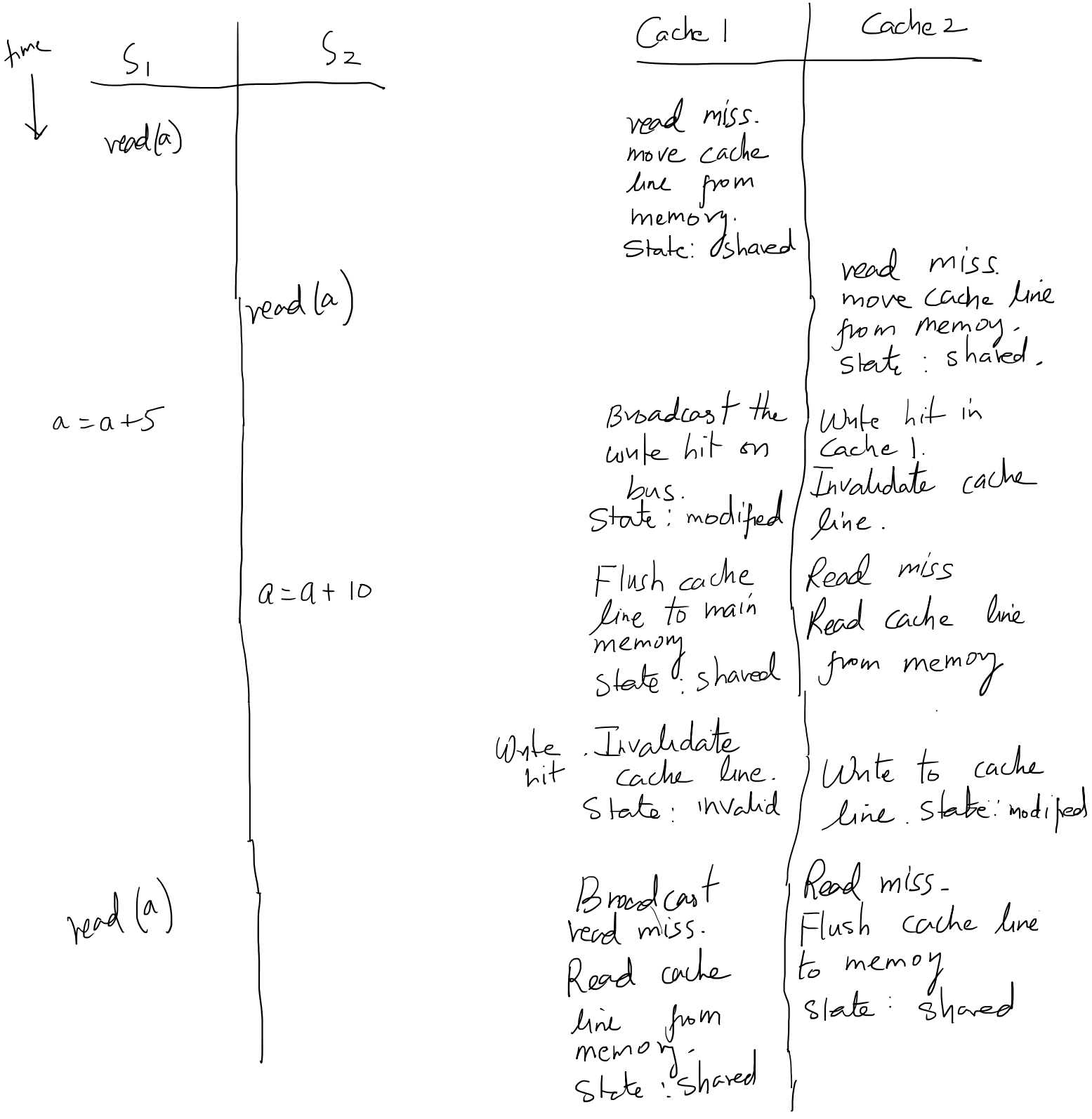
States of a cache line:
- Shared (read only)
- Modified (write)
- Invalid

Write Invalidate protocol



a: shared variable
a = 0;

Snooping based protocol.
Cache activity of all controllers is globally visible
→ read hits/misses
write hits/misses


time
S1
S2
read(a)
read(a)
a = a + 5
a = a + 10
read(a)
Cache 1
Cache 2
read miss.
move cache line from memory.
State: shared
read miss.
move cache line from memory.
state: shared.
Broadcast the write hit on bus.
State: modified
Write hit in Cache 1.
Invalidate cache line.
Flush cache line to main memory
state: shared
Read miss
Read cache line from memory
Write hit
Invalidate cache line.
State: invalid
Write to cache line. State: modified
Broadcast read miss.
Read cache line from memory.
State: Shared
Read miss.
Flush cache line to memory
state: shared

Key idea:

- Only one cache maintains a cache line in the modified state
  - ↳ All other caches invalidate that cache line.

Overhead:

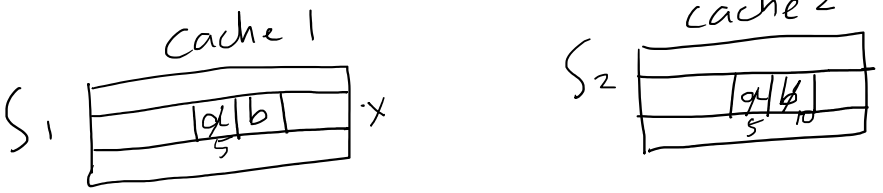
~ 64 cores
- ↳ since adding more cache controllers to the bus significantly increases traffic.

False sharing → impacts performance of parallel code.

False sharing

- We are using write invalidate protocol to ensure strict consistency.

$a = b = 0$



S_1 and S_2 share a cache line

a is used by S_1
b is used by S_2

$\Rightarrow$ Threads don't share variables; but the cache line.

S_1	S_2
read(a)	
	read(b)
	Invalidate cache line
a = a+5	
	b = b+10 ↑ cache miss
Invalidating cache line.	
:	
a = a+10 ↑ miss	Invalidate cache line
	:

S_1 and S_2 keep invalidating each other's cache lines if they continue to write to variables a, b.

Don't strictly share a variable(s). They share a cache line.

Performance: it will be worse than if we had no cache at all.

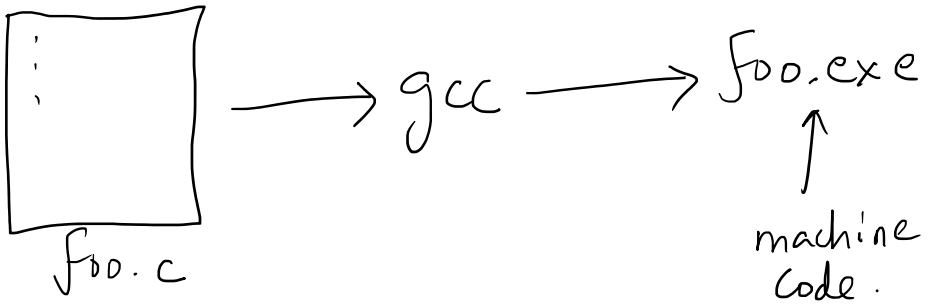
Examine your threads' data access patterns to make sure that eliminates / minimize false sharing.

Threads

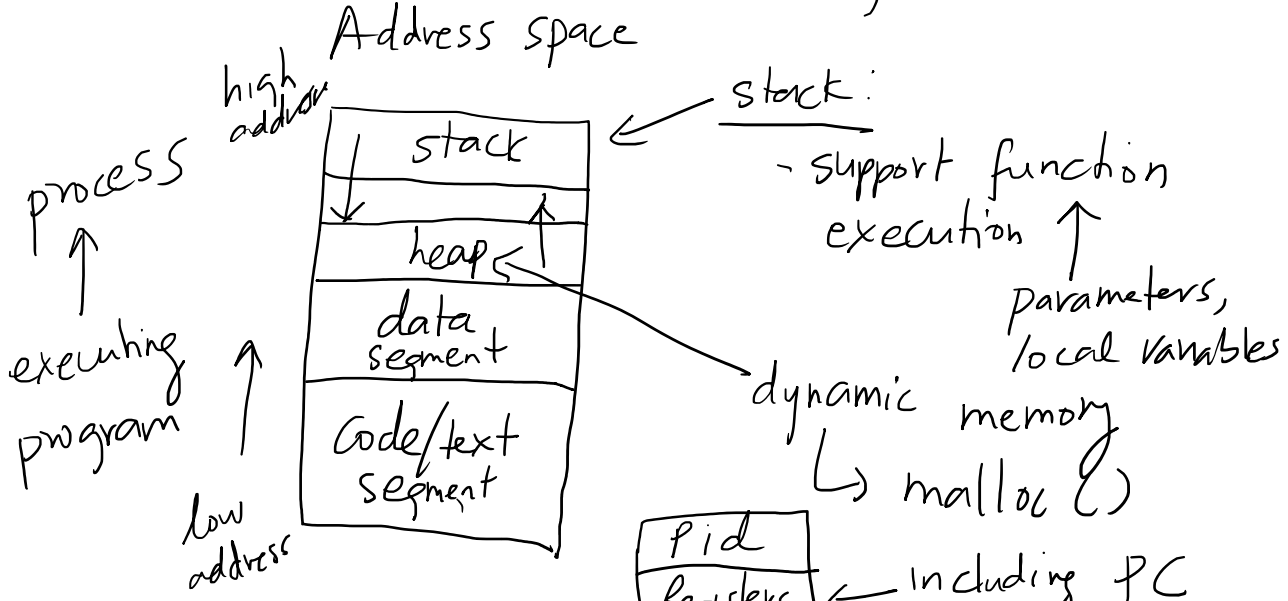
Tuesday, April 14, 2020 5:52 PM

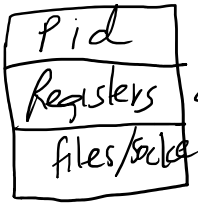
pthreads

Single-threaded process



$ ./foo.exe ↵



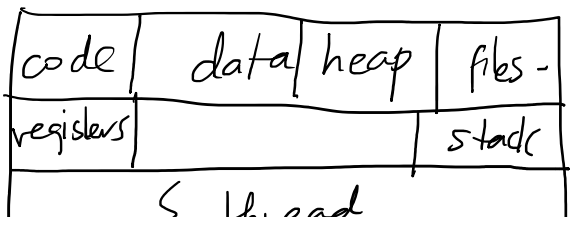


including PC

state

Process control block ← maintained by kernel

- Entry point is the main() function
- PC value is loaded....

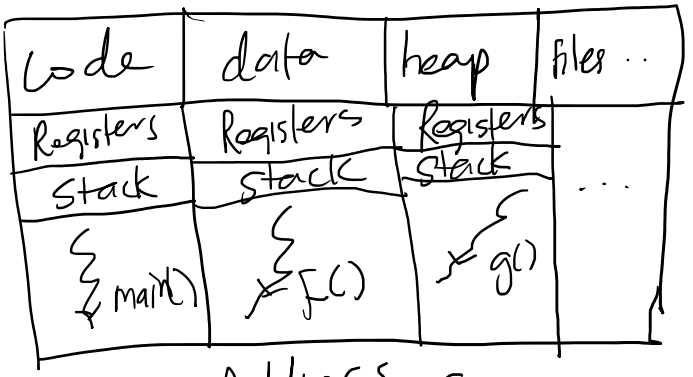


} Single threaded process

Execution path is controlled by program counter (PC)

↑ address of instruction to be execution.

Multi-threaded process



Address space

```
main() {
  {
  → create thread( , f, )
    {
  → create thread ( .., g, )
      {
    }
```

Threads share:

- Code segment
- Data segment
- Heap segment

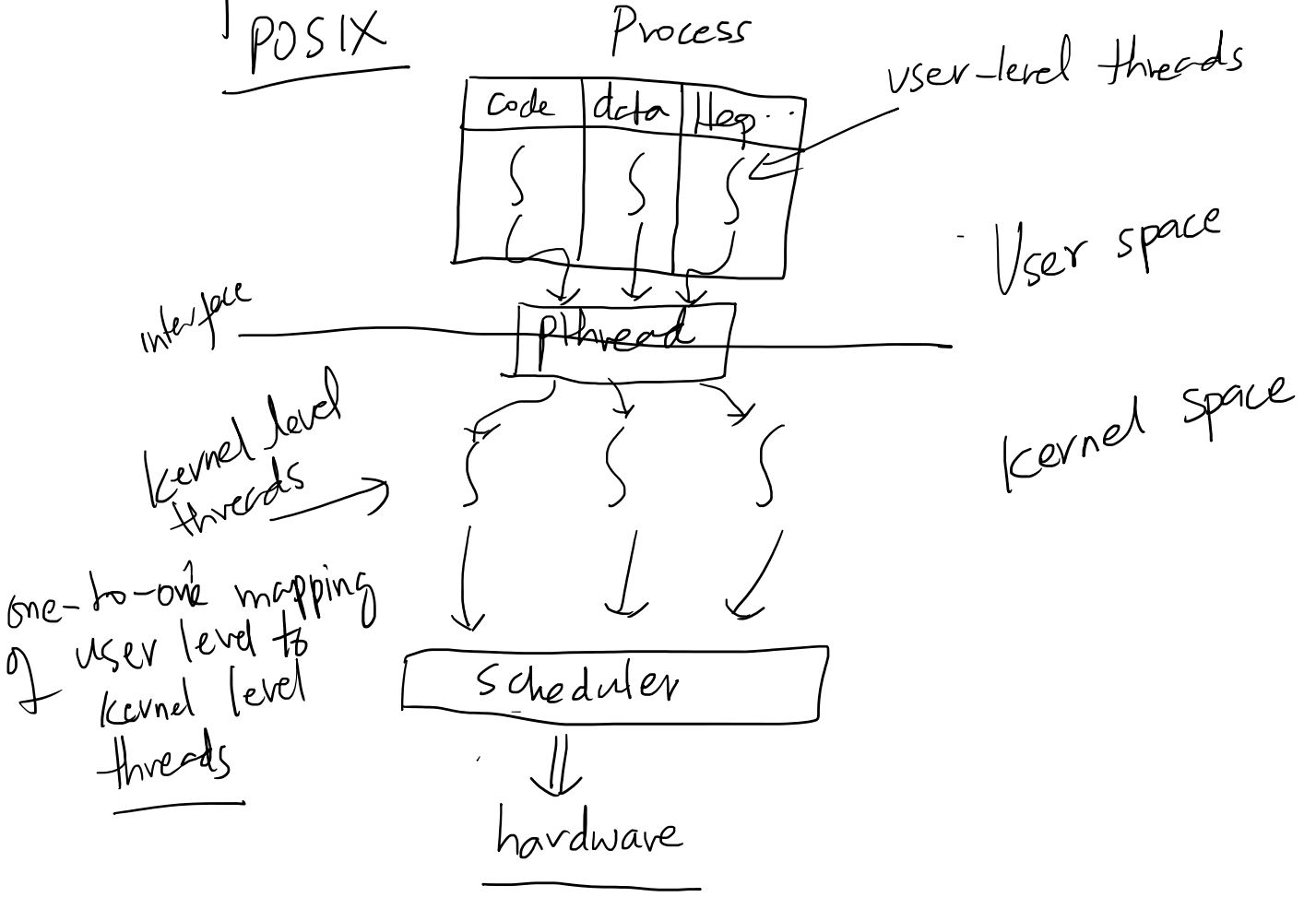
Separate:

- Registers
- Stack

How do we create threads?

→ pthread library

↑ POSIX



pthread_create (tid, , f,)

.

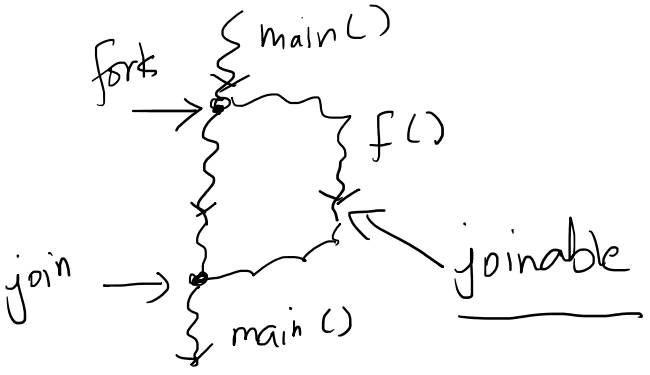
.

.

pthread_join (tid, status)

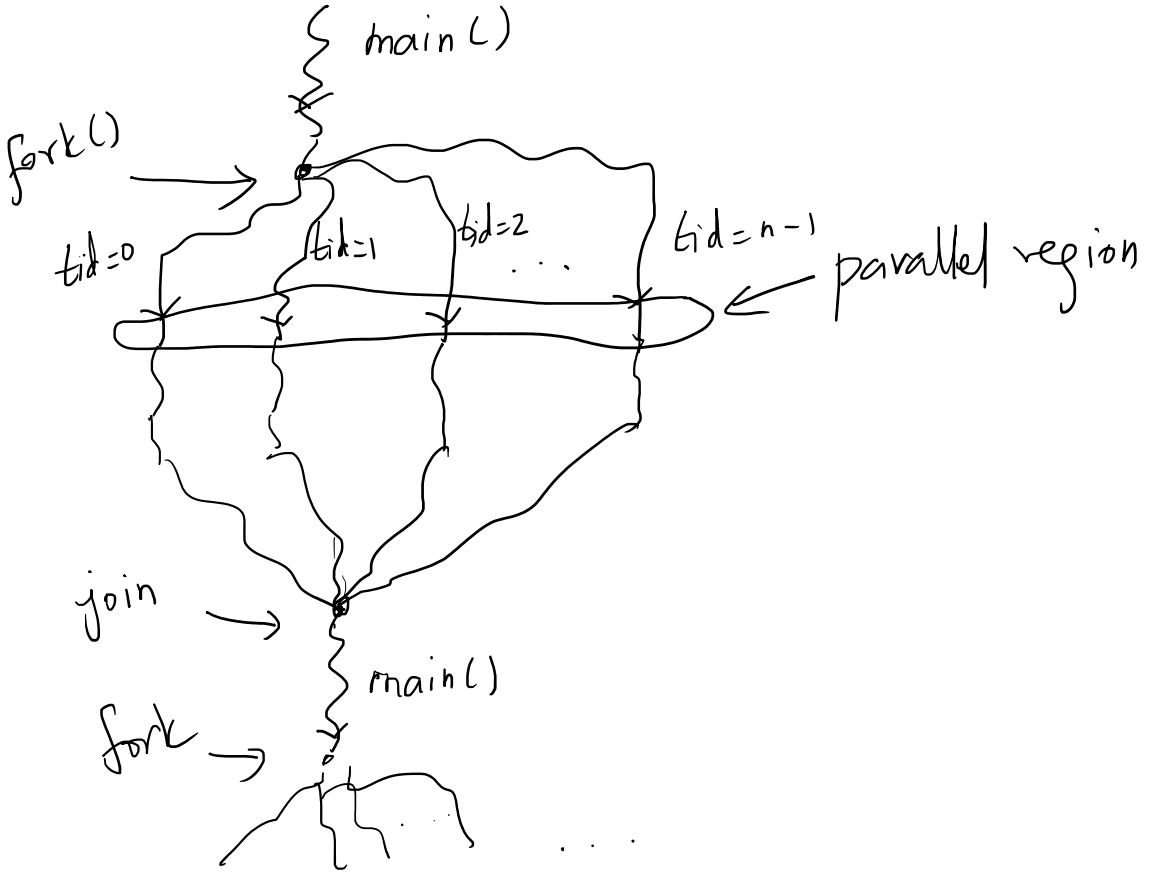
.

.



Every pthread_create() must have a corresponding pthread_join().

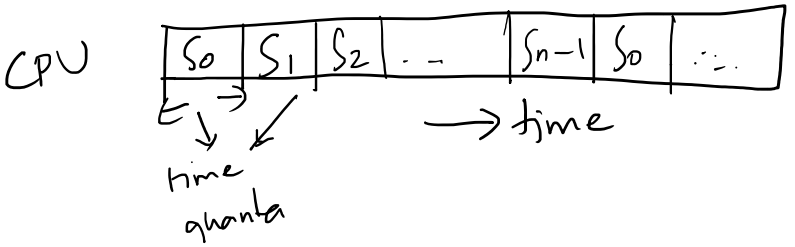
General case:



Concurrent execution

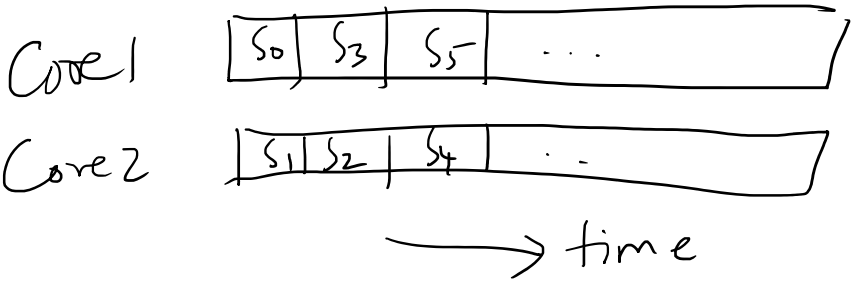
S_0 S_1 S_2 . . . S_{n-1}

Time share the singk CPU between threads.



Multi-core

→ Space / time sharing



---

pthread code . . .

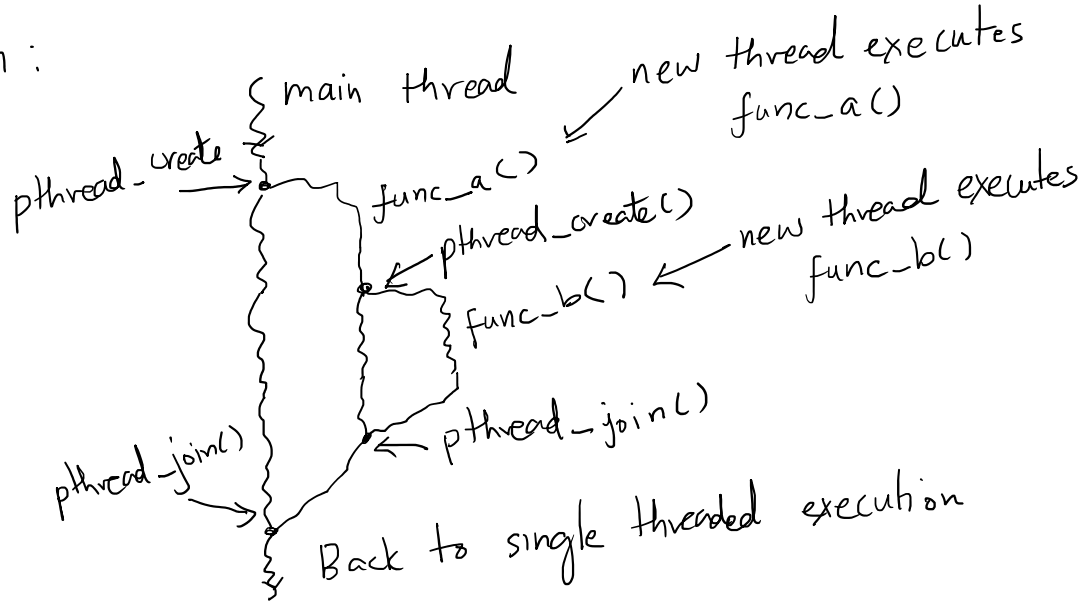
Thread creation and management

Thursday, April 16, 2020 9:22 AM

Show the use of pthread_create() and pthread_join().

simple_thread.c

Visualization:



Key things to note:

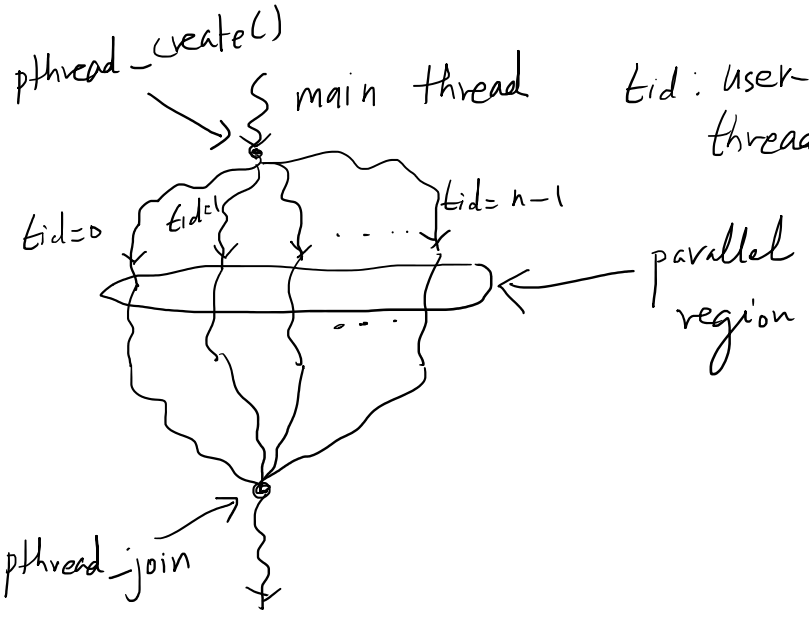
0. No parameters are passed to func_a()
1. Integer parameter is passed to func_b()

type cast to (void *) before passing

type cast (void *) back to (int) within the function

multiple_threads.c

Visualization:



Key points:

- Every pthread_create() must have a corresponding pthread_join() if the threads are meant to be "joinable."
- Multiple arguments are passed to threads in the form of a structure

  ↑ pack arguments into fields of a structure; pass pointer to structure to the function.

- Every thread executes the same function. Only thing different between the threads are the arguments passed to them.

  → Example of SIMD code

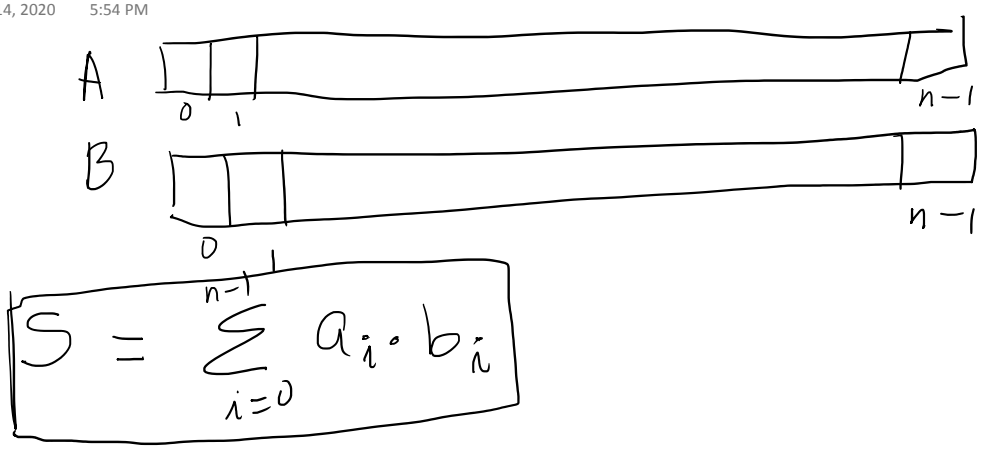
Writing parallel programs using pthreads

Tuesday, April 14, 2020 5:53 PM

- Vector dot product
- Vector matrix multiplication

Vector dot product

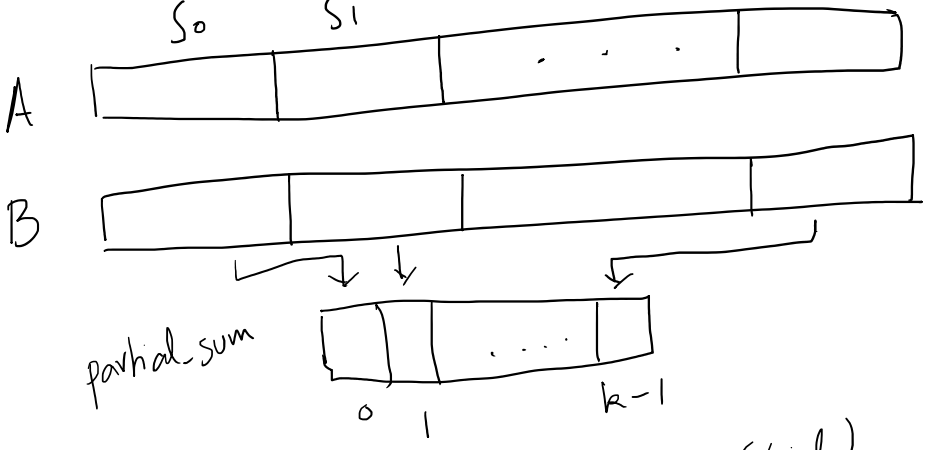
Tuesday, April 14, 2020 5:54 PM



$$S = \sum_{i=0}^{n-1} a_i \cdot b_i$$

Version 1 : "Chunking method"

- Create k threads ; each thread calculates partial sum on a chunk.



- Each thread gets its thread id (tid)
- Chunk size = $\lfloor \frac{n}{k} \rfloor$ ← floor operator
- Offset within arrays A and B for each thread = tid × chunk_size
- Threads 0 through k-2 compute partial sum over (~~tid~~ offset to ~~tid~~ offset + chunk_size)

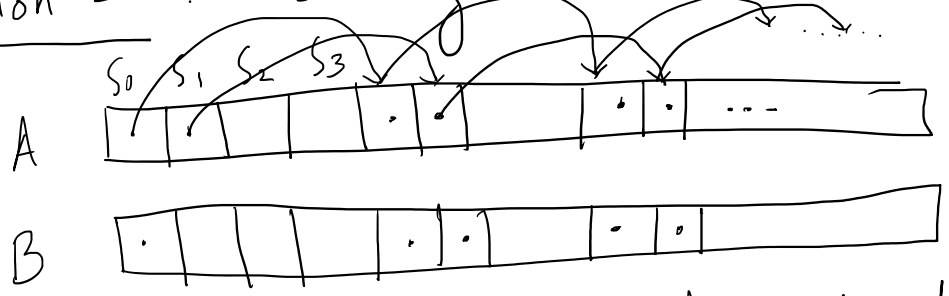
offset = tid × chunk size

- Last thread tid = k-1 may need to process more elements. That is,

  (offset to n-1)

- Main thread generates the final sum by accumulating the partial sums.

Version 2 : "striding".



Each thread strides over elements of A and B, calculating partial sum.

Example:

```
tid ← thread ID ; partial_sum = 0
k ← stride length
While (tid < n) {
  partial_sum += A[tid] x B[tid];
  tid += k;
}
```

Main thread accumulates the partial sums.

Matrix-vector multiplication

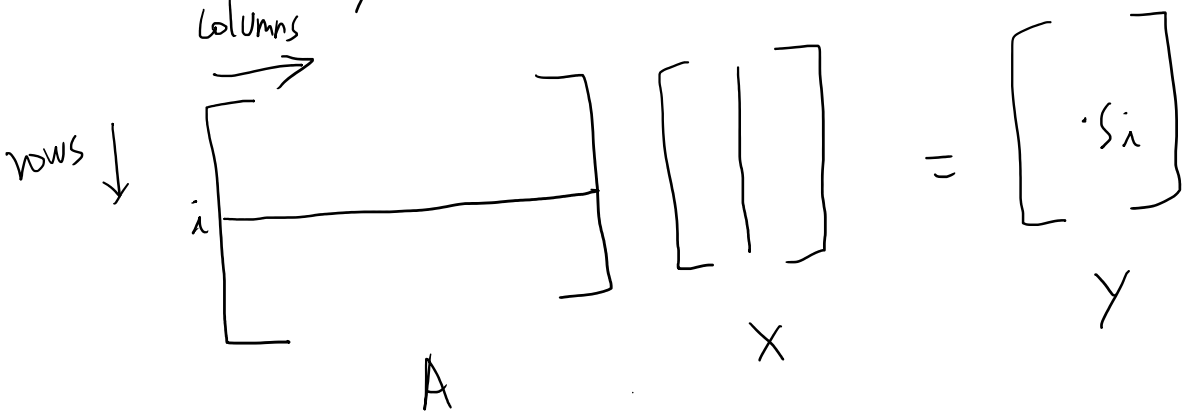
Tuesday, April 14, 2020 5:53 PM

$Ax = y$

A is a $n \times n$ matrix (to keep discussion simple)

X is $n \times 1$ vector

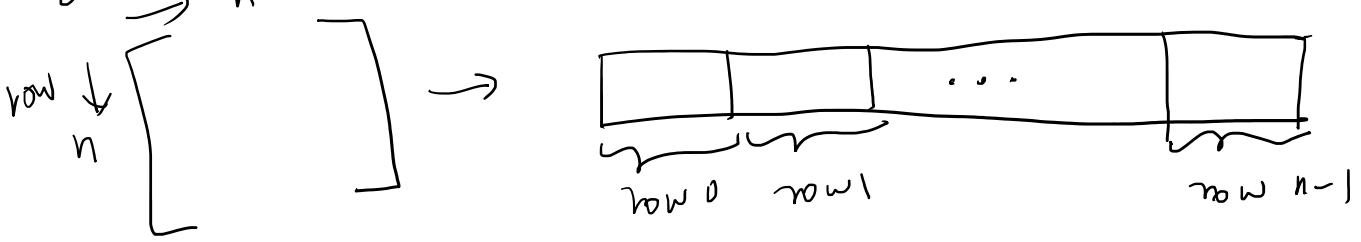
Y is $n \times 1$ result vector



- Problem decomposition
  - Each element of y can be calculated in parallel.
    - y[i] = dot product of ith row of A and X vector
- Can use concept of chunking or striding to calculate each output value.

. Note:

2D arrays (or N-D) arrays are laid out in row major form in memory.



So $A[i][j] \rightarrow A[i \times n + j]$

row col

width of matrix

Exceptions

Wednesday, April 15, 2020 10:01 AM

A process can cause various types of exceptions over course of it's execution.

Example:

1. $a = b + c$ ← value of a could be too large to store in defined data type

2. $a = b / c$ ← what happens if $c = 0$?

Hardware generates exception overflow

Kernel posts signals to process

(1) and (2) generate SIGFPE exceptions

Floating point exception.

3. int A[100];

   ⋮

   A[100] = ... ← out of bounds

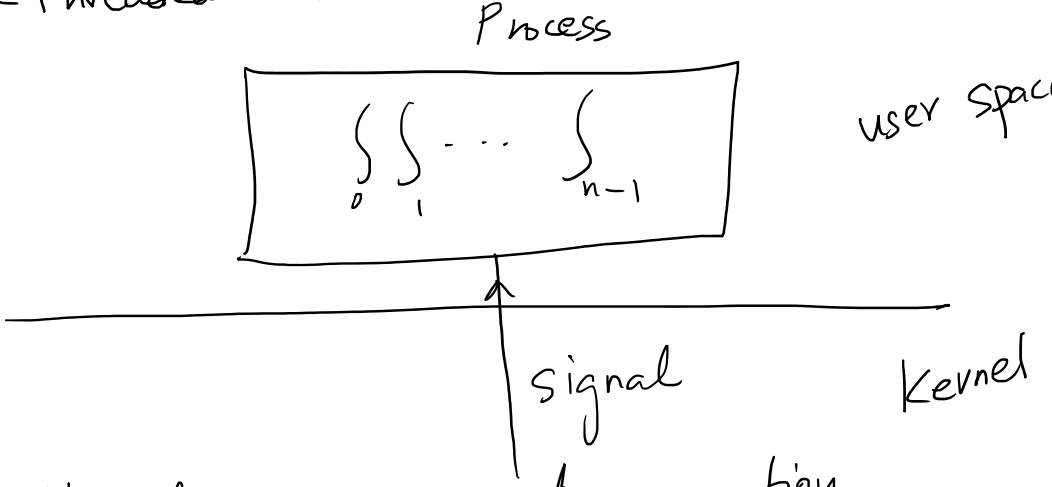
generates segmentation fault

SIGSEGV

4. Other signals:
   - Control + C to terminate process

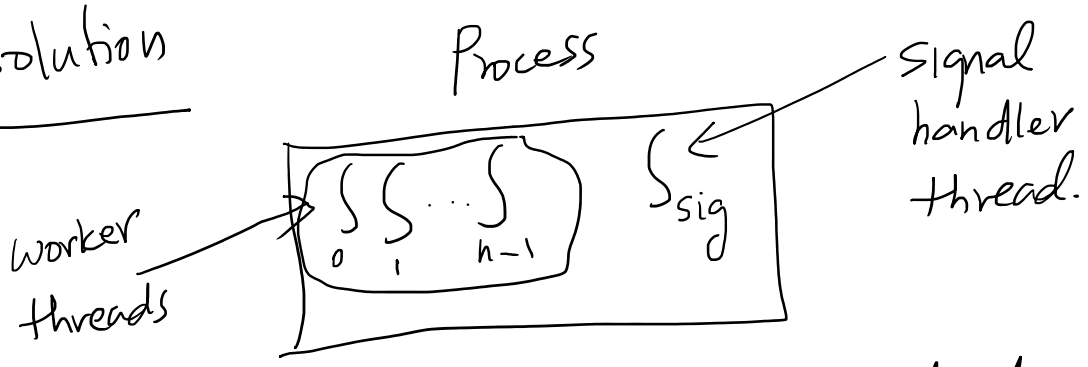
↑ SIGINT signal

In multi-threaded case:



Any thread may generate exception.
Kernel posts signal to all threads.

Typical solution



- Designate and set up one thread to be the "signal handler" thread.
- Other threads ignore signals posted to them.

Note: the process cannot ignore certain signals

e.g.: SIGKILL

Used by kernel to terminate a "misbehaving" process.

See signal_handler_thread.c for code.