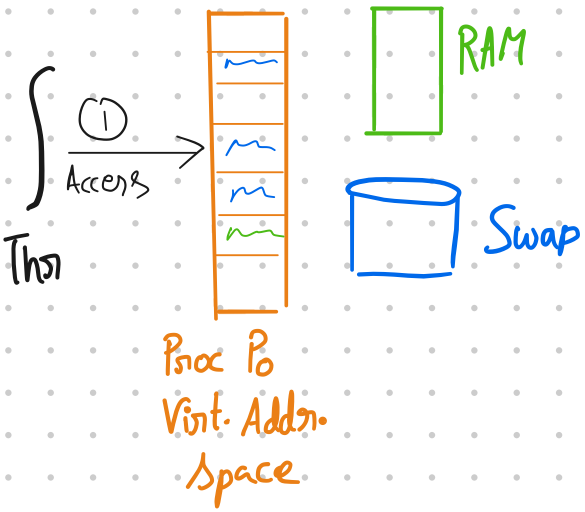
CS202 (-002) Lec 14 - Virtual Memory

Last class

- Page faults ⟷ Demand paging



- Eviction policies: Might need to Evict, that is, move pages from RAM to swap to make space in physical memory.

  Policy: What page to evict

  Metric: Memory access time

p: Probability access handled by RAM

$$\text{Avg. time to access memory} = p \cdot \underbrace{\text{time to access RAM}}_{\sim 10\text{-}100\text{ns}} + (1-p) \cdot \underbrace{\text{time to access swap}}_{\sim 1\mu s - 100ms \;\; 10^3 ns \sim 10^8 ns}$$

⇒ Minimize access from swap

- Belady's MIN (Opt): Ideal policy – look into future to decide what to evict

  +ve: Great perf

  -ve: Cannot be implemented

- Least Recently Used
- Least Frequently Used

  ↳ Realistic (only requires observing past accesses)

Where we left off last class: how to implement?

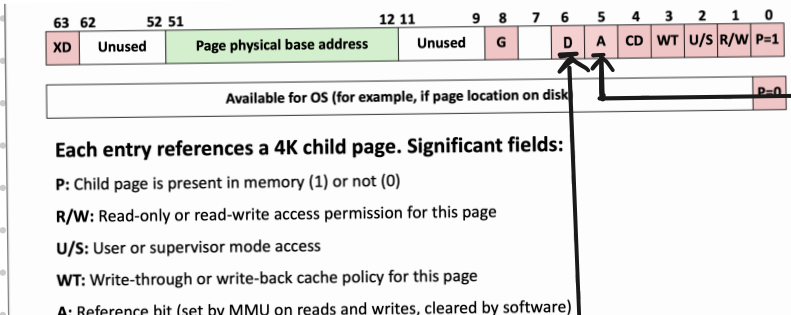
① Maintain a data structure ordering pages by when they were last accessed (LRU) or access frequency (LFU)

Problem: Logic to maintain the data structure needs to run on every access

↳ Exception on every access

EXPENSIVE / BAD

We want hardware to help



x86-64 MMU (also other MMUs)

Reference Bit set to 1 when page is accessed.

NB: Kernel (or other software) responsible

D: Dirty bit (set by MMU on writes, cleared by software)

Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

DIRTY

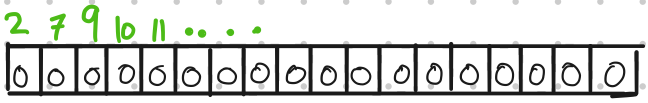
for setting it to 0

Bit set to 1 when page is modified (written to)

NB: Kernel responsible for setting it to 0

What use is this

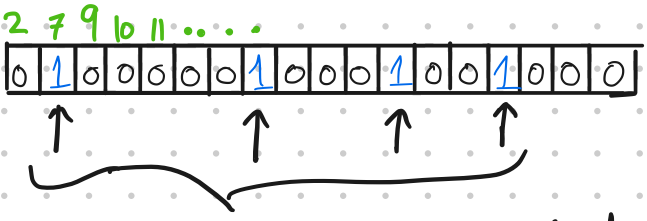
Reference Bits for VPNs



Program Executes

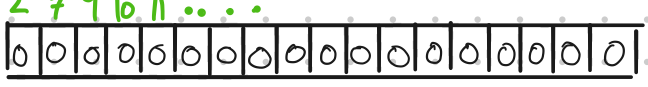
t=0

Exception/ Timer interrup/ ... pauses execution



Kernel Updates

Cannot order them, but were 'recently' accessed



Resume

⭐ Reference bit provides a way to determine what pages were accessed "RECENTLY"

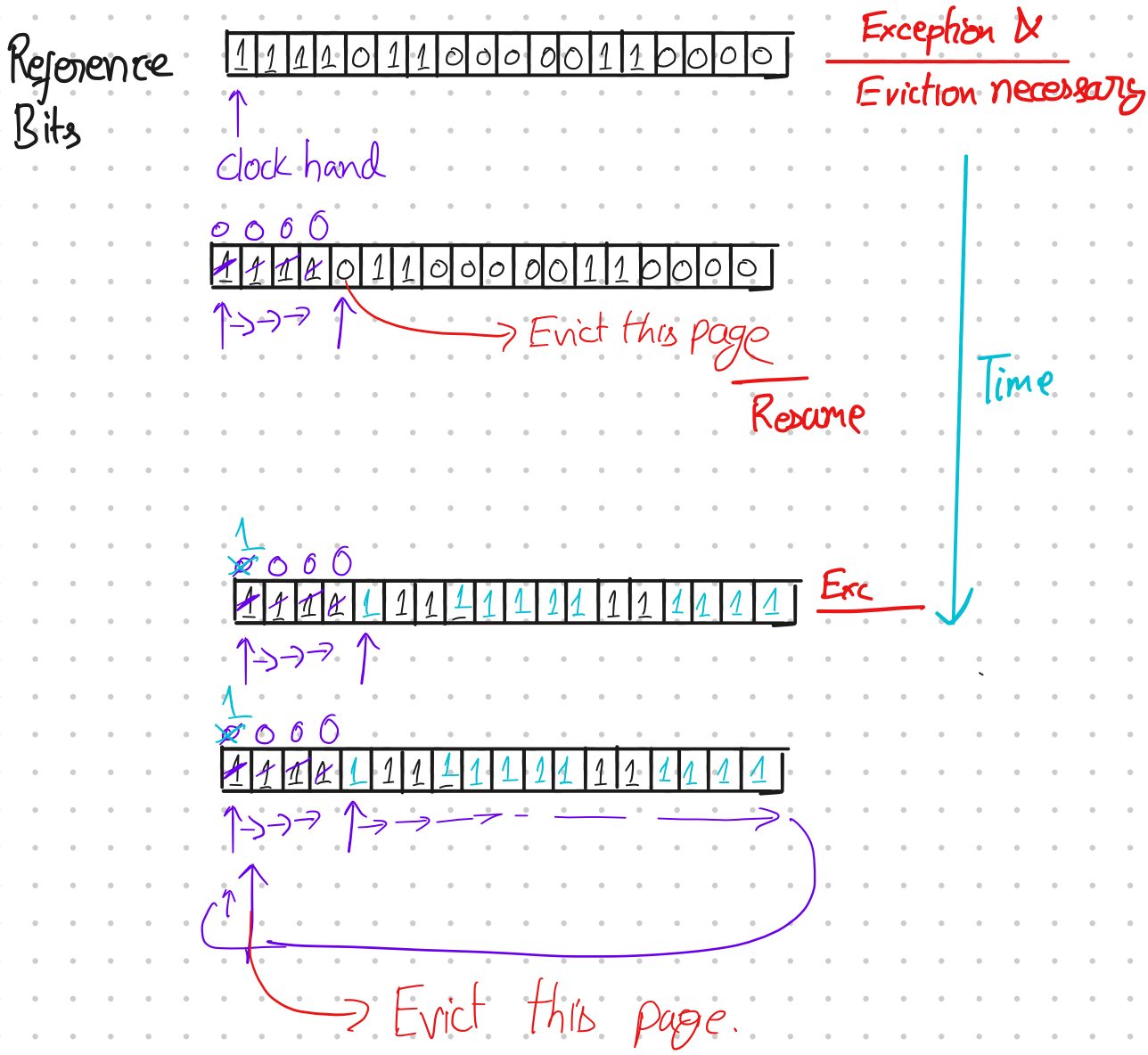
↳ And with a bit more work, frequency.

Using Reference Bit to approximate LRU — CLOCK

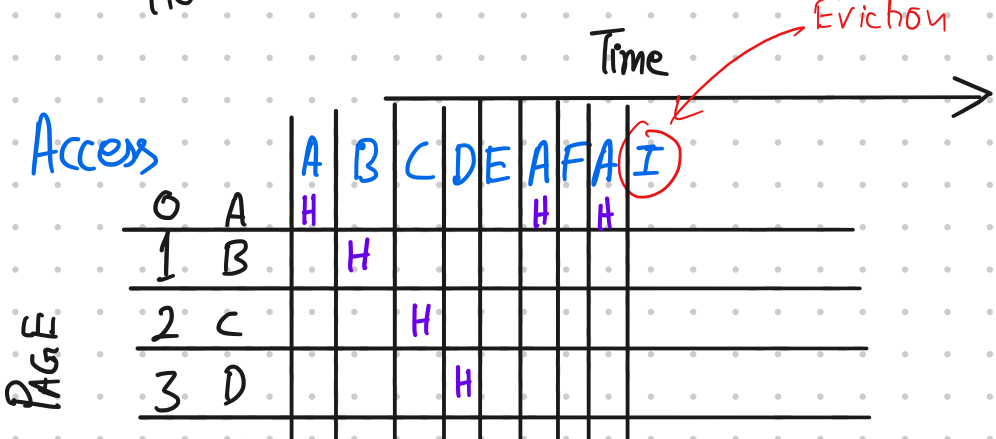
- Why? Popular algorithm implemented by many OS.

↳ Low overhead

Core idea



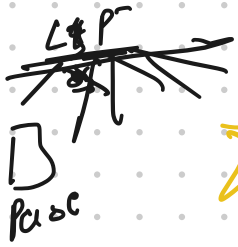
How well does it approximate?



LRU evicts? B

Clock evicts?

A

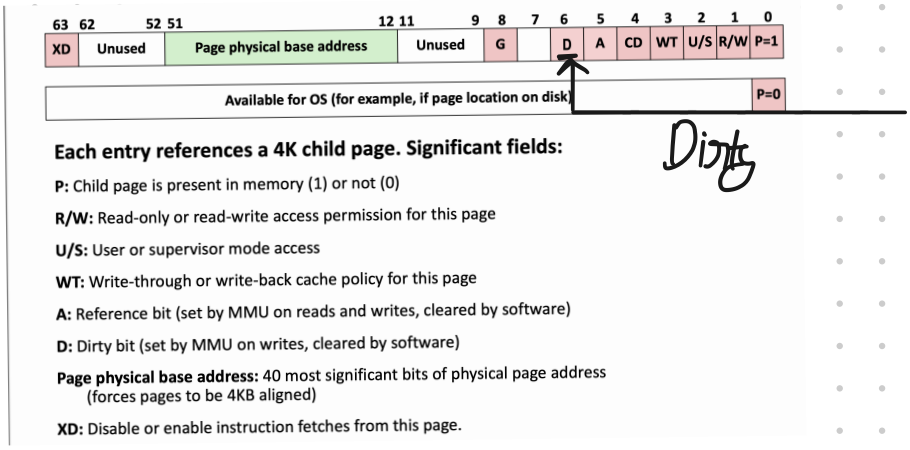


B

Page

☆ Clock approximates LRU using the access bit

- Aside:



Dirty

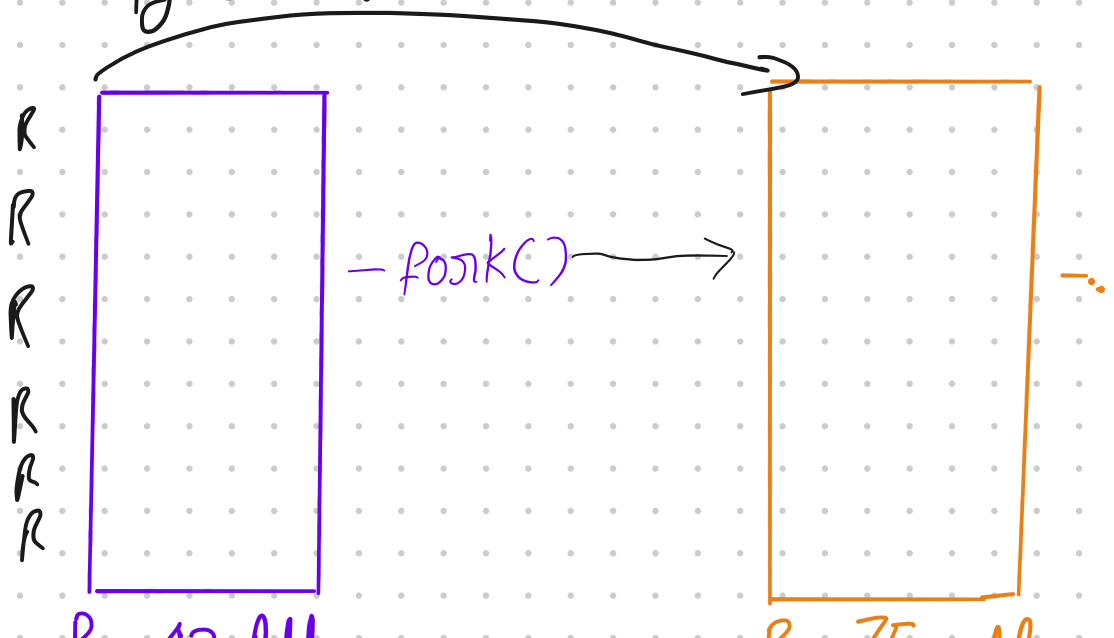
Bit set to 1 when page is modified (written to)

NB: Kernel responsible for setting it to 0

Helps reduce eviction cost! How?

Other uses for Page Faults

- Copy on Write



Proc. 42 Addr Space (memory)

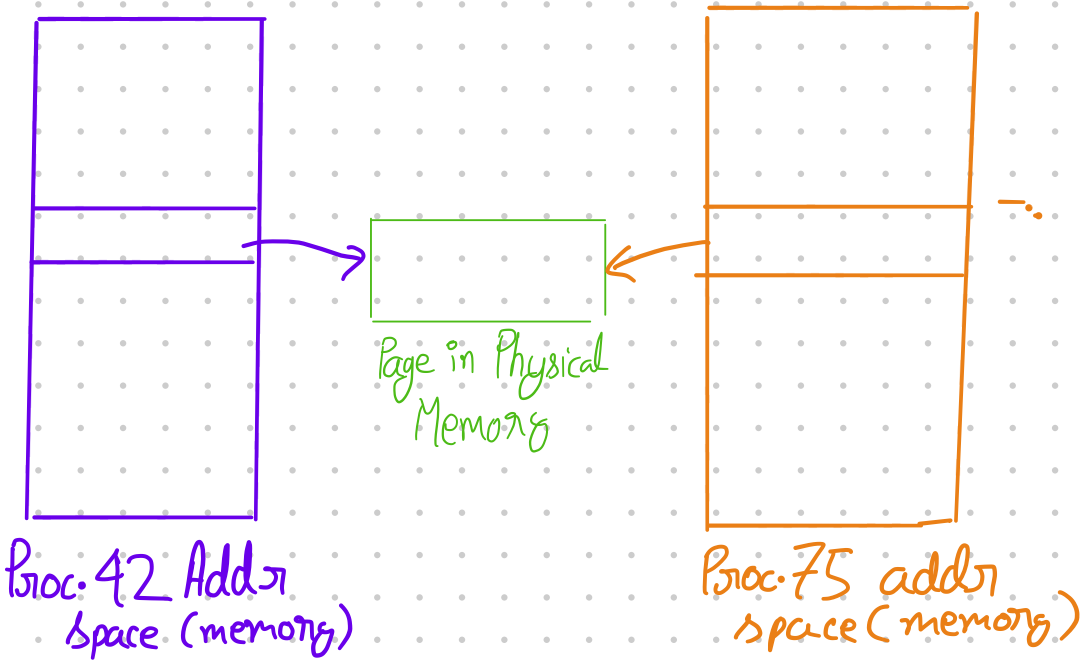
Proc. 75 addr space (memory)

- fork() ÷ proc 42 & 75 should have the same memory content.

How? ① Make a copy

↳ Problem: Time

② Share pages?



Problem

```
1  int main(int argc, char* argv[]) {
2    int x = 4;
3    if (fork() == 0) {
4      x *= 2;
5      printf("x = %d\n");
6    } else {
7      wait(NULL);
8      printf("x = %d\n", x);
9    }
10 }
```

x=22;

Copy-On-Write

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page physical base address	Unused	G		D	A	CD	WT	U/S	R/W	P=1
Available for OS (for example, if page location on disk)												P=0

Each entry references a 4K child page. Significant fields:

P: Child page is present in memory (1) or not (0)

∅: Read-only.

Writes cause

R/W: Read-only or read-write access permission for this page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

a page-fault!

How can we use this?

0x2 [12] offset

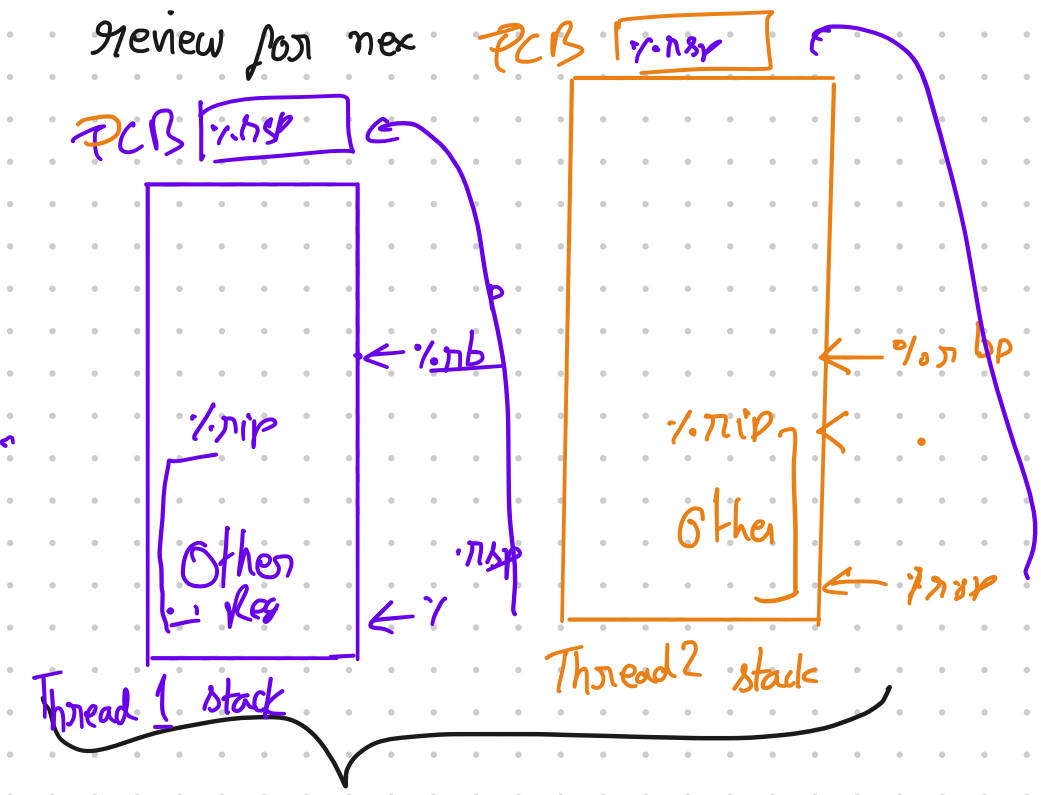
page num

For each page what is the first offset that is accessed

① Pages already accessed
② Accessed addr

P

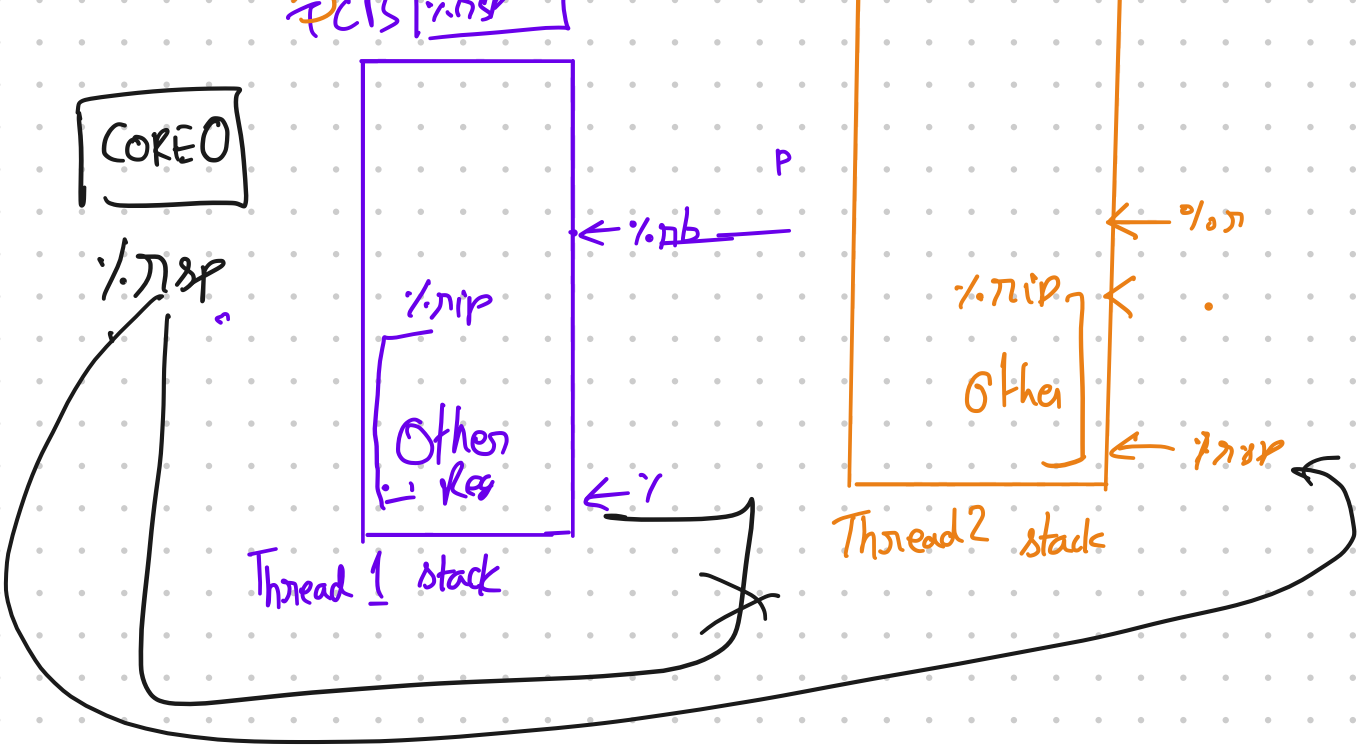
t-cbs: Context switch



T=2
yield()
syscall
exc-handle

T=1
yield()
syscall

Assume (for now) they are in the same process P


CORE0
%rsp
%rbp
%rip
Other Reg
Thread 1 stack
%rsp
%rip
Other
Thread2 stack