Concurrency → ทำให้มันเหมือนทำงานไปพร้อมกัน → เมื่อก่อน 1 process : 1 thread, ปัจจุบัน 1 process : many thread

↳ ใช้การแบ่งเวลาเข้าไปใช้งาน → โดย "scheduler" ↳ "context switching"

↳ ทำให้เกิด concurrency

→ การทำงานหลายๆ อย่างไปพร้อมกัน

↳ Motivation

- ↳ Operating systems (and application programs) often need to be able to handle multiple things happening at the same time — ถ้ามีหลาย thread ก็แบ่งเวลาการทำงานให้ n (แบ่ง n n n n +)
  - ↳ Process execution, interrupts, background tasks, system maintenance.
- ↳ Humans are not very good at keeping track of multiple things happening simultaneously
- ↳ Thread are an abstraction to help bridge this gap — model เมื่อก่อนแบ่ง t ให้เท่ากันทุก process / สมัยนี้แบ่ง t ให้ไม่เท่า → ตามการใช้

↳ Why Concurrency?

concurrency เกิดจากการแบ่ง thread

- ↳ Server → ใช้ประมวลผลบน server ต้องทำงานให้ลูกค้าไปพร้อมๆ กัน ↳ การใช้งาน microprocessor ในรูปแบบ time sharing ผ่านการ thread
  - ↳ Multiple connections handled simultaneously
- ↳ Parallel programs → ใช้ประมวลผลที่ใช้การทำงาน 4 cpu → ขอแบ่งเป็น thread → รันขนานพร้อมกันได้
  - ↳ To archieve better performance
- ↳ Program with user interfaces → ส่วน logic กับ ui → ทำงานแยกกัน → ถ้าเกิดการคำนวณยาวๆ ui ค้างไม่ได้
  - ↳ To archieve user responsiveness while doing computation.
- ↳ Network and disk bound programs → อุปกรณ์พวก I/O → การอ่าน → ไม่มี input มา → รอ → blocking ↳ thread ติด รอยาวๆ → ไปประมวลผลตัวอื่นไม่ได้
  - ↳ การเขียนก็เช่นกัน
  - ↳ To hide network/disk latency

↳ Definitions

→ ช่วงเวลาที่ code เข้าไปทำงาน managed by scheduler ↳ เหมือน microcontroller สำหรับสั่งงานๆ

- ↳ A thread is a single execution sequence that represents a separately schedulable task
  - ↳ Single execution sequence : familar programming model
  - ↳ Separately schedulable : OS can run or suspend a thread at any time
- ↳ Protection is an orthogonal concept → เมื่อมันทำงานแยกกันก็ต้องมีการเข้าใช้ร่วมกัน ↳ ควรจะมีระเบียบ
  - ↳ Can have one or many threads per protection domain
  - ↳ ทำให้ 1 process : n threads

ข้อดีคือถ้าเป็น thread เดียวกันแชร์ code, data, var กันได้

แยก thread
kernel เป็นคนสร้าง

- Threads in the kernel and at User-Level → ลักษณะของ thread

now ③
- Multi-threaded kernel → kernel ที่รองรับการสร้าง thread ได้หลาย thread → แต่ละ thread อิสระ
  - multiple threads, sharing kernel data structure, capable of using privileged instructions (สามารถ / เหมือนกัน)

old ①
- Multiprocess kernel (เก่า) → รับการทำ concur ของ process not thread (process → stack, heap / PCB)
  - Multiple single-threaded processes → 1 process : 1 thread (n process ⇒ n thread)

จุดเปลี่ยนผ่าน ②
- System calls access shared kernel data structures

ทำได้ทั้งบน user, ker
thread ที่อยู่บน user ส่งไป kernel ได้
แต่ kernel ขึ้นมา user ไม่ได้

- Multiple multi-threaded user processes → ความยากอยู่ที่การจัดการ thread ขึ้นไปบน user.
  - Each multiple threads, sharing same data structure, isolated from other user processes
    - ของเก่าทำ context switch ย้าย mode → sys call

app — มุมมองของ application ทั่วไป — 5 thread ที่สร้างโดย app

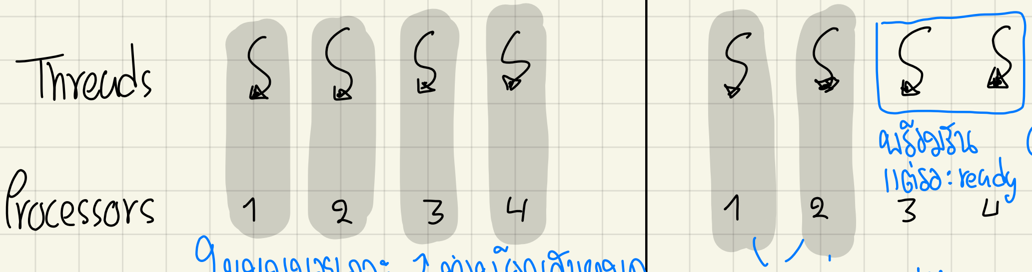
- thread → แนวคิดตอนแรกถึงการแบ่งเวลา & เกิดขึ้นที่ kernel
  ในนี้ 10 ms เวลา share ต่อ
  - มอง 2 กลุ่ม → กลุ่ม user mode → 1 thread (1:)
    กลุ่ม kernel mode → 1 thread (1:)
    } การทำ concurren ต้องมองแบบ kernel กับ user ทำงานพร้อมกันไป

2 thread

- ผลของแนวคิดนี้เริ่มจะเกิดแนวคิดที่ว่าอยากให้โปรแกรมทำงานมากกว่า 1 โปรแกรมทำงานร่วมกันได้
  - 1 program ทำงานใน process
    1 โปรแกรม 1 thread ทำงานแบบ concurren 1:kernel กับ m:user
- ต่อมาก็เพิ่มความสามารถเป็น 1 โปรแกรมมีได้มากกว่า 1 thread

- Thread Abstraction
  thread ที่อยู่ใน process เดียวกันสามารถแชร์ทรัพยากรร่วมกันได้ (isolation; ตัวแปร, code, data; "protection domain")
  - Infinite number of processors → มี processor ที่ไม่จำกัด
  - Threads execute with variable speed → เวลาที่ใช้ไปทำงานขึ้นอยู่กับความหลากหลาย up to event by scheduler
    - Program must be designed to work with any schedule

Programmer Abstraction | Physical Reality → ตัว hw ที่เป็น microprocessor



ในมุมมองของ ⤴ ทำพร้อมกันหมด
ทำงานอยู่ : running
พร้อมรัน แต่รอ : ready
มีแค่ 2 core
ในตอนเป็นการจำลองสถานการณ์ทำงาน
ได้ถึง 2 thread เท่านั้น ใน 1 ช่วงเวลา

↳ Programmer vs Processor View

รูปแบบการทำงานที่เป็นไปได้

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
⋮	⋮	⋮	⋮
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	Thread is suspended	y = y + x;
z = x + 5y;	z = x + 5y;	Other thread(s) run	Thread is suspended
		Thread is resumed	Other thread(s) run
		y = y + x;	Thread is resumed
		z = x + 5y;	z = x + 5y;

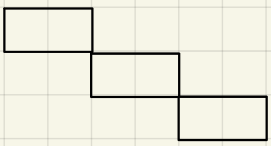
เราไม่ต้องสนใจ → เลย

OS จัดการให้หมด → เหมือนเดิม โปรแกรมไม่สามารถรับรู้ถึงการทำ context switch

↳ Possible Executions

One Execution

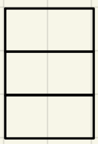
Thread1
Thread2
Thread3



มี 1 core
timeslot กว้าง

Another Execution

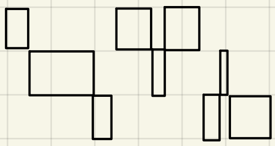
Thread1
Thread2
Thread3



↳ มีมากกว่า 3 core ขึ้นไป

One Execution

Thread1
Thread2
Thread3



มี 1 core
timeslot แคบ

↳ Thread Operations → system call ที่เกี่ยวข้องกับ thread

- ↳ thread_create(thread, func, args) → สร้าง
  - ↳ Create a new thread to run func(args)
- ↳ thread_yield() → การคืนเวลาให้ระบบ → เสร็จก่อน หรือ รออย่างอื่นทำเสร็จ
  - ↳ Relinquish processor voluntarily
- ↳ thread_join(thread) → รอให้อีก thread จบการทำงาน → ต้องการผลลัพธ์จากอีก thread
  - ↳ In parent, wait for forked thread to exit, then return
- ↳ thread_exit → ทำลาย thread → ปลุก thread_join
  - ↳ Quit thread and clean up, wake up joiner if any

↳ Example: threadHello

```
#define NTHREADS 10
thread_t threads[NTHREADS];
main(){
    for(i=0; i<NTHREADS; i++)
        thread_create(&threads[i], &go, i);
    for(i=0; i<NTHREADS; i++){
        exitValue = thread_join(threads[i]);
        printf("Thread %d returned with %ld\n", i, exitValue);
    }
    printf("Main thread done.\n");
}
void go(int n){
    printf("Hello from thread %d\n", n);
    thread_exit(100+n);
    //REACHED ?
}
```

- thread_create: สร้าง thread ขึ้นมา 10 thread โดย code ที่จะนำไปทำงาน คือ function go ด้วย arg i
- thread_join: ให้ thread ที่ทำงานหยุดรอจนกว่า thread ที่จะ return กลับมา
- thread_exit → เหมือน return ของ function ปกติแต่เป็นของ thread → return ค่าให้ thread อื่น

↳ Output:

สรุปคือ → up to hw นะ

- ↳ Why must "thread returned" print in order? — สถานะ running → up to hw.
- ↳ What is maximum # of threads running when thread 5 prints hello? → สูงสุดมี thread ทำงานพร้อมกัน 11 thread
  - ↳ มาจาก 1 + (10) → thread ลูกที่สร้าง
- ↳ Minimum?
  - ↓ 2 thread → 1 + (1) (ทำงาน ↳ เสร็จ...นาน)
  - จากของ 3 ↓ 1 main, 2, 3
  - ไม่ต้องรอ thread แต่ทำการ join ↳ มันจะดึงเวลาไปให้ระบบ ↳ high performance

ถ้า thread 2 ทำยังไม่เสร็จ 3, 4, 5, ... ก็ไม่ออก จาก source code

```
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 7
Hello from thread 8
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

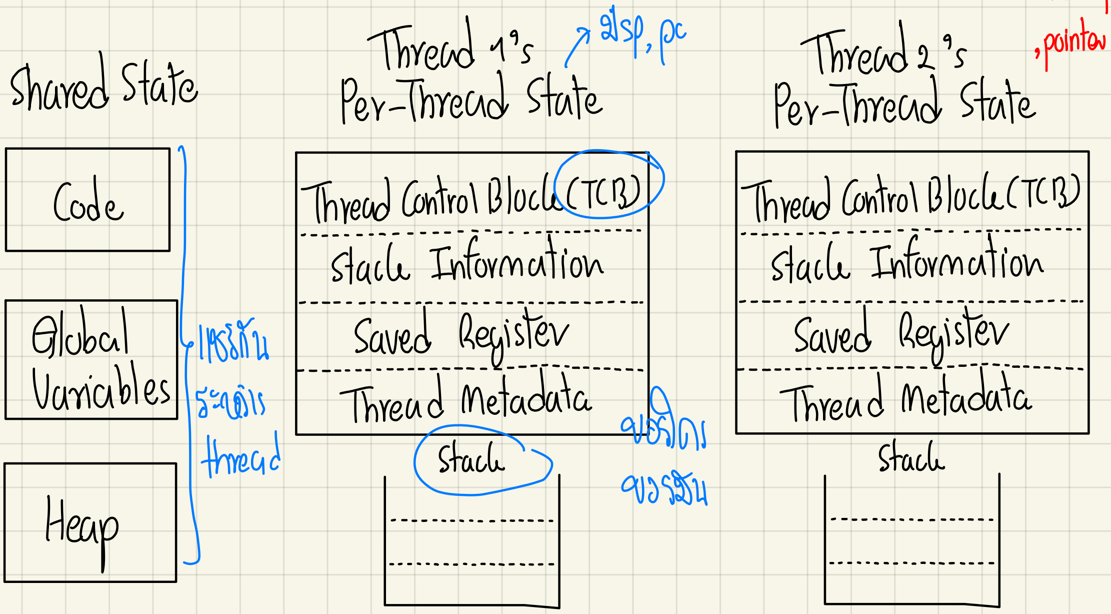
- Thread 0 returned 100 → thread_join(thread[0])
- Thread 1 returned 101 } ยังไม่ทันสร้างครบเลย มีตัวอื่นทำงานเสร็จแล้ว
- Hello from thread 5 … 9 } ที่ลำดับสลับเปลี่ยนตาม scheduler เลย ↳ thread ที่ถูกสร้างก่อนไม่จำเป็นต้องรันก่อน
- Thread 3 … 9 returned → high performance
- Thread 5 returned 105 } อาจจะเสร็จก่อนก็ได้ แต่ต้อง join 4 (2 ↑, 3 ↑) ↳ เสร็จแล้วรออยู่ ↳ เสร็จแล้วไปเลย

ความเป็นไปได้ในการทำงานแต่ละระยะ
↳ thread_exit ก็ต้องรอ เหลืออยู่ในระบบ
↳ เพราะ thread ยังอยู่

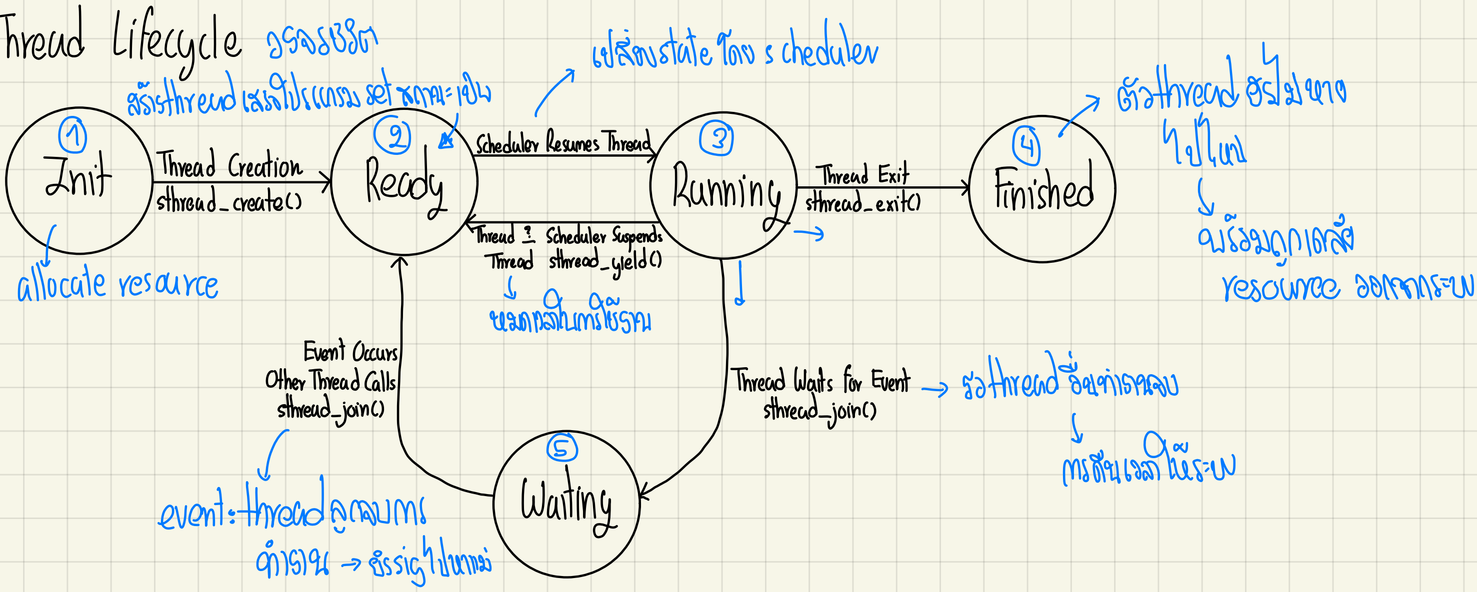
- Fork/Join Concurrency
  - Threads can create children, and wait for their completion → สร้าง thread ลูกแล้วแต่รอรับ (join) output
  - Data only shared before fork/after join
  - Examples:
    - Web server: fork a new thread for every new connection → ทุกๆคนที่เข้ามาได้รับบริการ
      - As long as the threads are completely independent
    - Merge sort → parallel algorithm
    - Parallel memory copy → การ copy ให้ thread ลูกทำ เรียก join กับการทำงานต่อไป

เพิ่มเติม TCB → เหมือน PCB แต่เป็นของ thread
- thread id, thread state, thread information (PC, reg), thread priority, pointer to parent process which create it, pointer to other thread which create by this thread

- Thread Data Structures

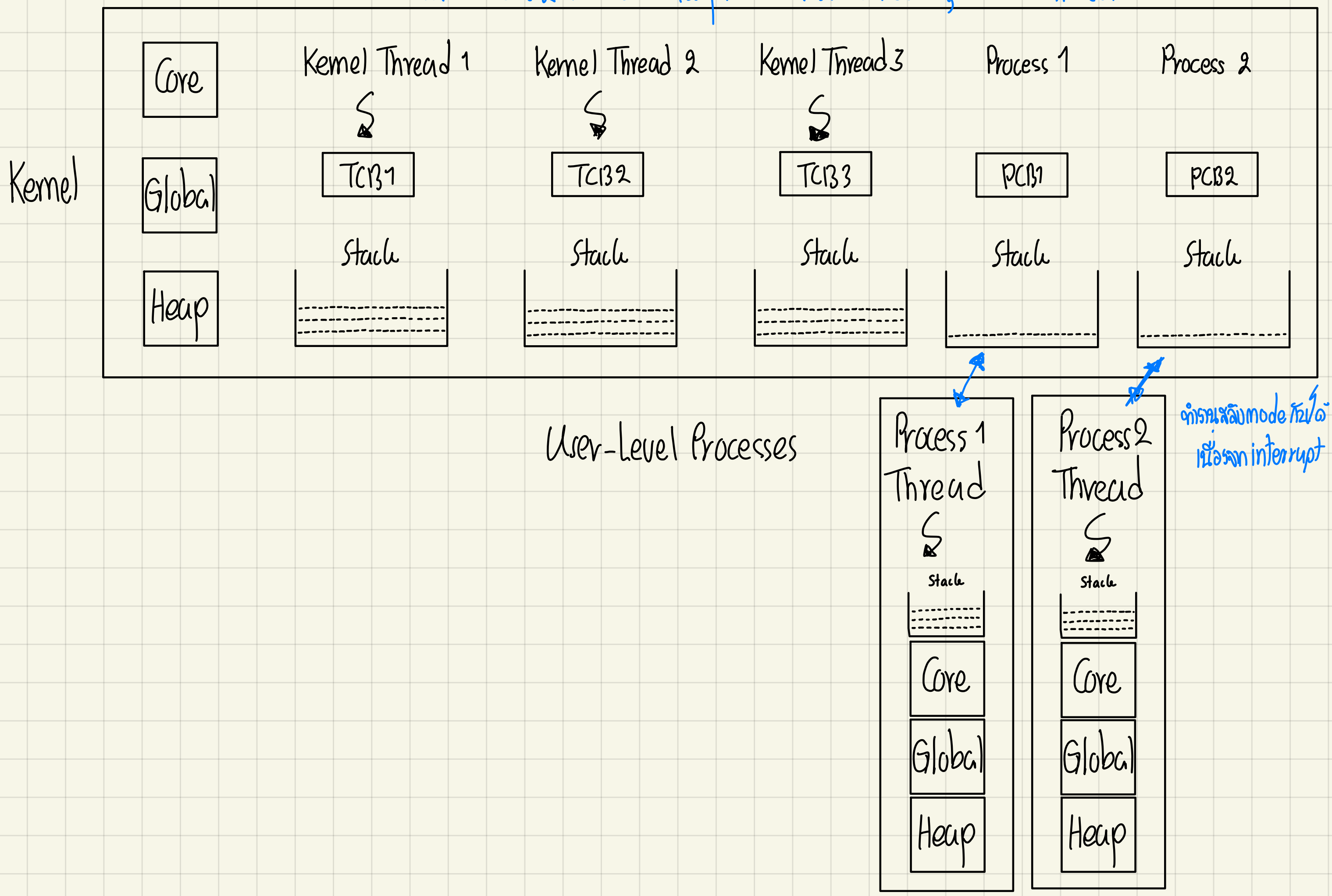


- Thread Lifecycle วงจรชีวิต



ถ้าเป็น windows: thread → thread_exit → (abort) → (finish) ไม่สามารถเรียกใช้ได้
↓
(ready)

- Implementing Threads: Roadmap
  - kernel threads → การสร้าง, จัดการ thread เกิดขึ้นด้วย kernel → สร้างใน kernel ทำงานใน user ได้ (→ มีทั้งส่วนอยู่ใน kernel, user !)
    - user ←(context switch)→ kernel → overhead
    - Thread abstraction only available to kernel
    - To the kernel, a kernel thread and a single threaded user process look quite similar
  - Multithreaded processes using kernel threads (Linux, MacOS) → สร้าง thread ผ่าน syscall ได้
    - Kernel thread operations available via systemcall
    - อันนี้ไม่ช่วยนะ
  - User-level threads → ความพยายามที่จะทำให้เกิด context switch ใน user mode ได้
    - Thread operations without system call
      - ถ้าเป็น kernel thread ก็ต้องทำงานใน kernel
    - ใน process มี lib → app ต้องการ thread เรียก lib → สร้าง thread ให้เลย (lib อยู่บน user) หรือเรียก thread ซ้ำๆ ก็ได้
      - ถ้าเป็น user thread → ต้องกลับไปเป็น kernel mode ทำ context switch → overhead
    - มีตัวทำ switch ให้บน user ; thread user ; thread ระดับ OS
- Multithreaded OS kernel → ใน kernel ก็มี thread ใน process ก็มี thread, both thread are the same



หน่วยความจำต่อไป / การเรียกใช้ function { ลง stack เยอะ ตอนหลังจากการของ stack → อาจใช้เยอะๆ ก็ stack overflow

↳ Implementing threads การสร้าง thread

↳ Thread_fork(func, args)

↳ allocate thread control block สร้าง TCB ก่อน

↳ allocate stack จองพื้นที่ for stack

↳ build stack frame from base stack (stub) สร้าง stub* → เก็บไว้เรียน

↳ put func, args on stack

↳ put thread on ready list เปลี่ยนสถานะเป็น ready

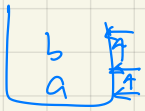
↳ will run sometime later (may be right away!)

↳ stub(func, args) ใน OS/161 mip_threadstart → ในการสร้าง thread ในตอนแรกจะมี stub function อยู่ใน stack เป็นตัวแรก

↳ call (*func)(args) → คือ function ที่จะรันใน thread

↳ if return, call thread_exit()

↳ Thread Stack → ซ้อนตัวแปร



↳ What if a thread put to many procedures on its stack? → stack overflow

↳ What happen in Java?

↳ What happen in Linux kernel?

↳ What happen in OS/161?

↳ What should happen?

ต่างกัน

↳ Thread Context Switch → กระบวนการสลับ thread ออกจาก microprocessor แล้วเรียก thread เข้ามาแทน

(2) แบบ ↳ Voluntary → โดย code ของ program ยินยอมที่จะคืนเวลา

↳ Thread_yield → การคืนเวลา → sw → syscall → syscall: thread_switch

↳ Thread_join (if child is not done yet) → ทำให้ thread : wait → ใช้บ่อยมาก

↳ Involuntary → thread ไม่ได้เต็มใจคืนเวลา → หมด quota ของตนเองที่จะเข้าใช้ซีพียูแล้ว

↳ Interrupt or exception → หมดเวลาที่ได้ timer interrupt หรือ exception

↳ Some other thread is higher priority → โดน thread priority สูงกว่าแย่งเวลา

↳ Voluntary thread context switch → ขั้นตอนเหมือนกันทั้ง 2 แบบ → แต่จุดเริ่มต้นมาต่างกัน
- ↳ Save registers on old stack → save ค่าของ reg ต่าง ๆ ลง stack
- ↳ Switch to new stack, new thread → switch
- ↳ Restore registers from new stack → load ค่า regของ thread ที่ถูกเลือก
- ↳ Return
- ↳ Exactly the same with kernel thread or user threads

ขั้นตอน

↳ OS/161 switch frame_switch การทำงานประมาณนี้

```
/* a0: old thread stack pointer *
   a1: new thread stack pointer*/

/* Allocate stack space for 10 registers. */
addi sp, sp, -40

/* Save the registers*/
sw   ra, 36(sp)
sw   gp, 32(sp)
sw   s8, 28(sp)
sw   s6, 24(sp)
sw   s5, 20(sp)
sw   s4, 16(sp)
sw   s3, 12(sp)
sw   s2, 8(sp)
sw   s1, 4(sp)
sw   s0, 0(sp)

/* Store old stack pointer in old thread */
sw sp, 0(a0)

/* Get new stack pointer from new thread */
lw sp, 0(a1)
nop     /* delay slot for load*/

/* Now, restore the registers */
lw   s0, 0(sp)
lw   s1, 4(sp)
lw   s2, 8(sp)
lw   s3, 12(sp)
lw   s4, 16(sp)
lw   s5, 20(sp)
lw   s6, 24(sp)
lw   s8, 28(sp)
lw   gp, 32(sp)
lw   ra, 36(sp)
nop          /* delay slot for load*/

/* and return*/
j ra
addi sp, sp, 40 /* in delay slot*/
```

↳ x86 switch_threads เหมือนเดิมเรียน → การ load thread ใหม่

↳ # Save caller's register state

```
# Note: %eax, etc are ephemeral
push %ebx
push %ebp
push %esi
push %edi

# Get offsetof (struct thread, stack)
mov thread_stack_ofs, %edx
# Save current stack pointer to old thread's stack, if any
movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax, %edx, 1)

# Change stack pointer to new thread's stack
# this also change currentThread
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx, %edx, 1), %esp

# Restore caller's register state
popl %edi
popl %esi
popl %ebp
popl %ebx
ret
```

↳ A Subtlety

- ↳ Thread_create puts new thread on ready list → สร้างเสร็จแล้ว set status to ready
- ↳ When it first runs, some thread calls switchframe
  - ↳ Save old thread state to stack
  - ↳ Restores new thread state from stack
- ↳ Set up new thread's stack as if it had saved its state in switchframe
  - ↳ "returns" to stub at base of stack to run func

↳ Two Threads Call Yield

↳ **Thread 1's instructions**

- "return" from thread_switch into stub
- call go
- call thread_yield ต้องการคืนเวลาแก่ระบบ → ขอให้ scheduler ให้เลือก thread ใหม่
- choose another thread เลือก thread อื่นที่ต้องการทำงาน
- call thread_switch syscall → เรียก fun thread_switch
- save thread 1 state to TCB เก็บ state ตัวเอง
- load thread 2 state load ของอันที่จะไปเข้าเลย, sp & pc ไปที่ code ที่จะทำงานแล้ว
- return from thread_switch เรียกที่นี่แต่ return ที่ thread1, sp ถอยลงมา ทำงานต่อไปได้
- return from thread_yield
- call thread_yield
- choose another thread
- call thread_switch

(เรียกไป thread อื่น ↑ มา return ตรงนี้)

Thread 2's instructions

- "return" from thread_switch into stub — ปกติต้อง pop addr ออกจาก stack เพื่อรอ ← แต่นี่สร้างมาใหม่ยังไม่ได้รันอะ → pop แล้วแตก → เลยสร้าง stub function เข้าไปรองรับ
- call go → ไปเรียกใช้ต่อ load code เข้าไปเลย (thread ที่ทำงานครั้งแรก)
- call thread_yield เรียกตัว
- choose another thread
- call thread_switch
- save thread 2 state to TCB
- load thread 1 state โหลดกลับ

(choose another thread … save thread 2 state to TCB: เหมือนเดิม)

Processor's instructions

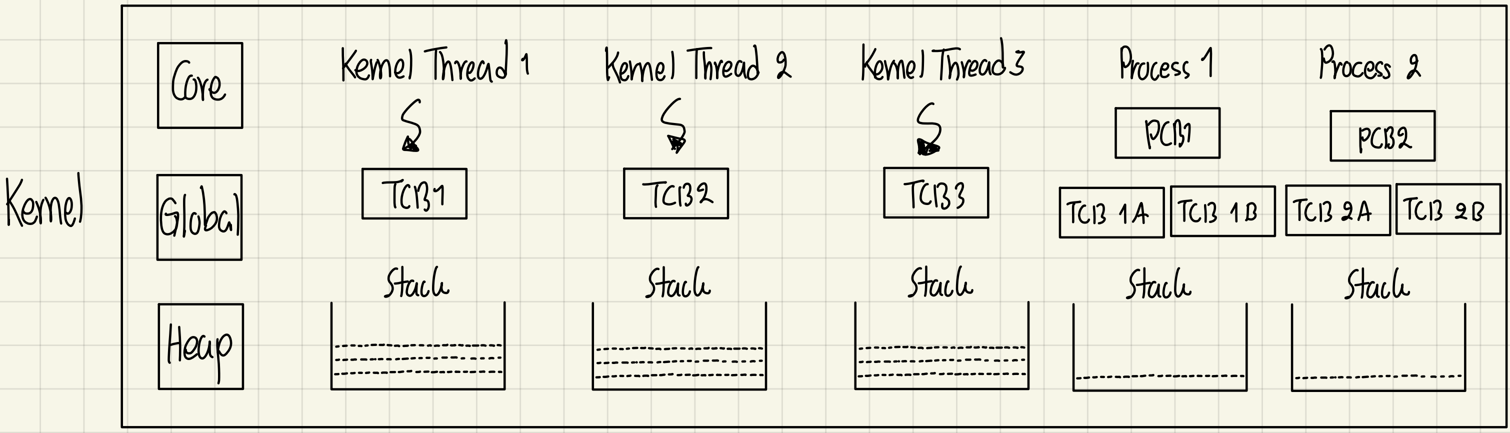
- "return" from thread_switch into stub
- call go
- call thread_yield
- choose another thread
- call thread_switch
- save thread 1 state to TCB
- load thread 2 state
- "return" from thread_switch into stub
- call go
- call thread_yield
- choose another thread
- call thread_switch
- save thread 2 state to TCB
- load thread 1 state
- return from thread_switch
- return from thread_yield
- call thread_yield
- choose another thread
- call thread_switch

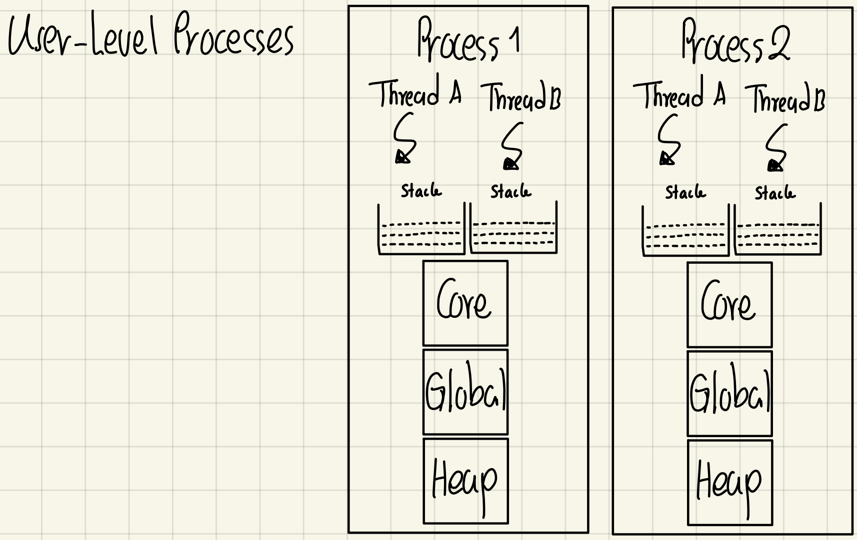
↳ Involuntary Thread/Process Switch
  ↳ Timer or I/O interrupt → เรียกinterrupt timer → เรียกใช้ interrup routine → เรียกใช้ context switch
    ↳ Tell OS some other thread should run
  ↳ Simple version (OS/161)
    ↳ End of interrupt handle calls switch()
    ↳ When resumed, return from handler resumes kernel thread or user process
    ↳ Thus, processor context is saved/restored twice
      (once by interrupt handler, once by thread switch)

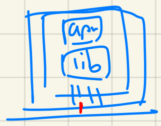
↳ Faster Thread/Process Switch → ปัญหาเดิมยังมีอยู่ → ปรับปรุงให้ดีขึ้น
  ↳ What happen on a timer (or other) interrupt?
    ↳ Interrupt handler saves state of interrupted thread
    ↳ Decides to run new thread
    ↳ Throw away current state of interrupt handler?
    ↳ Instead, set saved stack pointer to trapframe
    ↳ Restore state of new thread
    ↳ On resume, pops trapframe to restore interrupted thread

ปรับปรุง → ปัญหาเดิมยังมีอยู่ แต่ทำให้ดีขึ้นกว่าเดิม

↳ Multithread User Processes (Take1) → Thread ที่อยู่ใน user, kernel mode มีอยู่แต่ต่างกัน
  ↳ User thread = kernel thread (Linux, MacOS) ↳ ยังเป็นปัญหาอยู่
    ↳ System calls for thread fork, join, exit (and lock, unlock, ...) ↳ MacOS, Linux
    ↳ kernel does context switch
    ↳ Simple, but a lot of transitions between user and kernel mode







- Multithread User Processes (Take 2)
  - Green thread (early Java) → แบบที่ 2 → กรณีที่ process จะมี lib ที่ใช้จัดการ thread
    - User-level library, with in single-threaded processes
    - Library does thread context switch
    - Preemption via upcall / UNIX signal on timer interrupt
    - Use multiple processes for parallelism
      - Shared memory region mapped into each process

↓ พยายามลดการทำ context swith
↓ เน้นว่าการจัดการ thread อยู่ภายในใน usermode

ถ้าเป็น java jvm จัดการ scheduler ให้
↓
java รันได้หลายที่

↳ Multithread User Processes (Take 3)

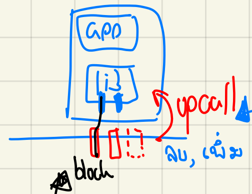
↳ Schedular activation (window 8)   ↑ ขั้นไป → เหมือน green thread แต่ที่เพิ่มขึ้นมา lib จะทำการสื่อสารกับ kernel ด้วย ผ่าน upcall

- ↳ Kernel allocates processors to user-level library
- ↳ Thread library implements context switch
- ↳ Thread library decides what thread to run next

↓ upcall เมื่อมีการ assign thread เพิ่ม, ลบ ให้กับ process นี้ ↓ thread ระดับ kernel

↳ Upcall whenever kernel needs a user-level scheduling decision

- ↳ Process assigned a new processor
- ↳ Processor removed from process
- ↳ System call blocks in kernel



หรือ มีการ block เกิดขึ้นใน kernel thread

จะใช้งานไม่ได้ ↘ มีการรอรับ I/O → รอ data → รอนาน ๆ ก็อาจจะขั้นตอนเหรือ

lib สามารถจัดการกับจำนวน thread ที่มีเยอะแยะมากได้ดี

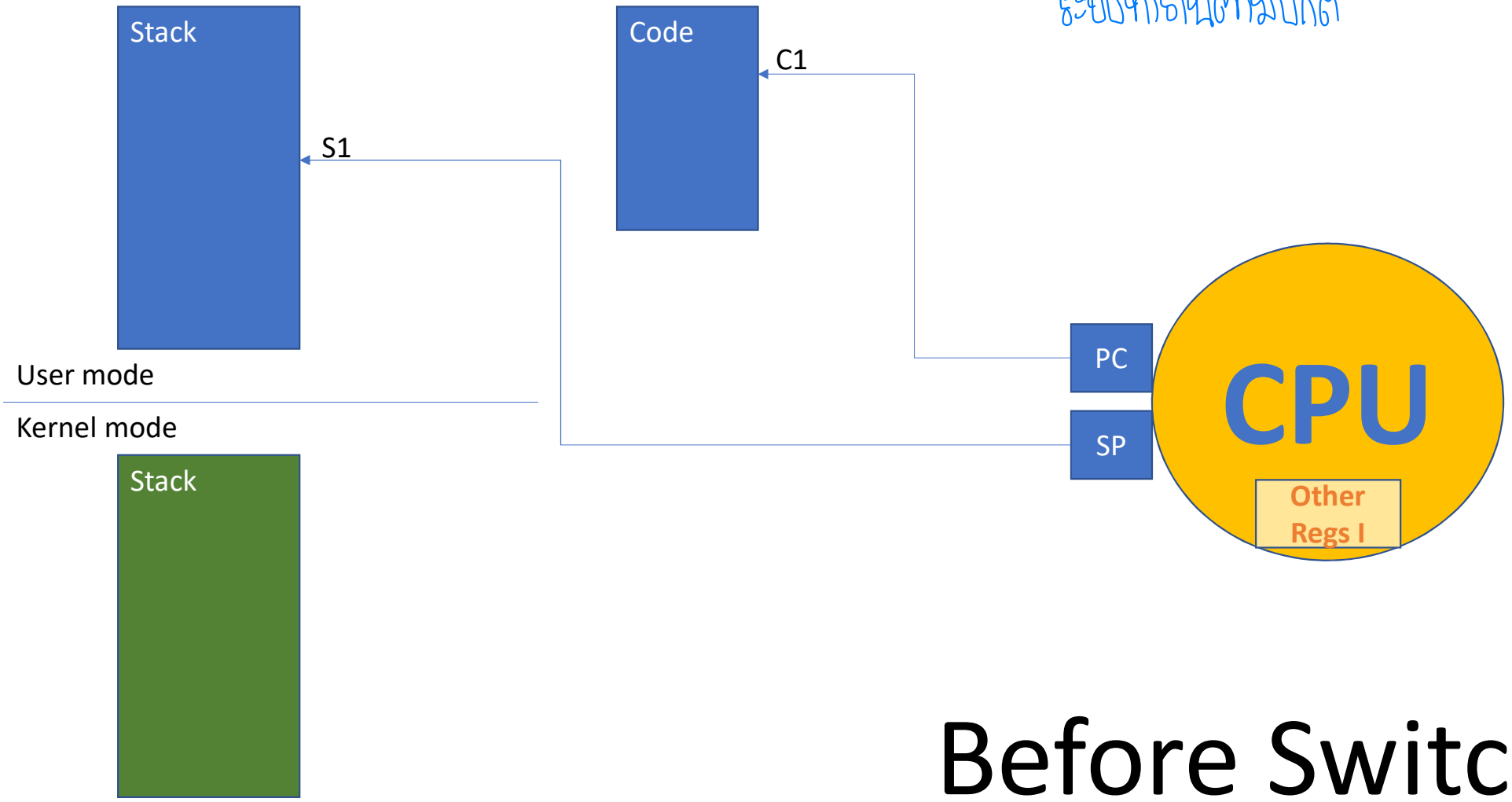
Question?   ลองไปคิดดูนาจา   การบ้านที่ 2

↳ Compare event-driven programming with multithread concurrency. Which is better in which circumstances, and why?

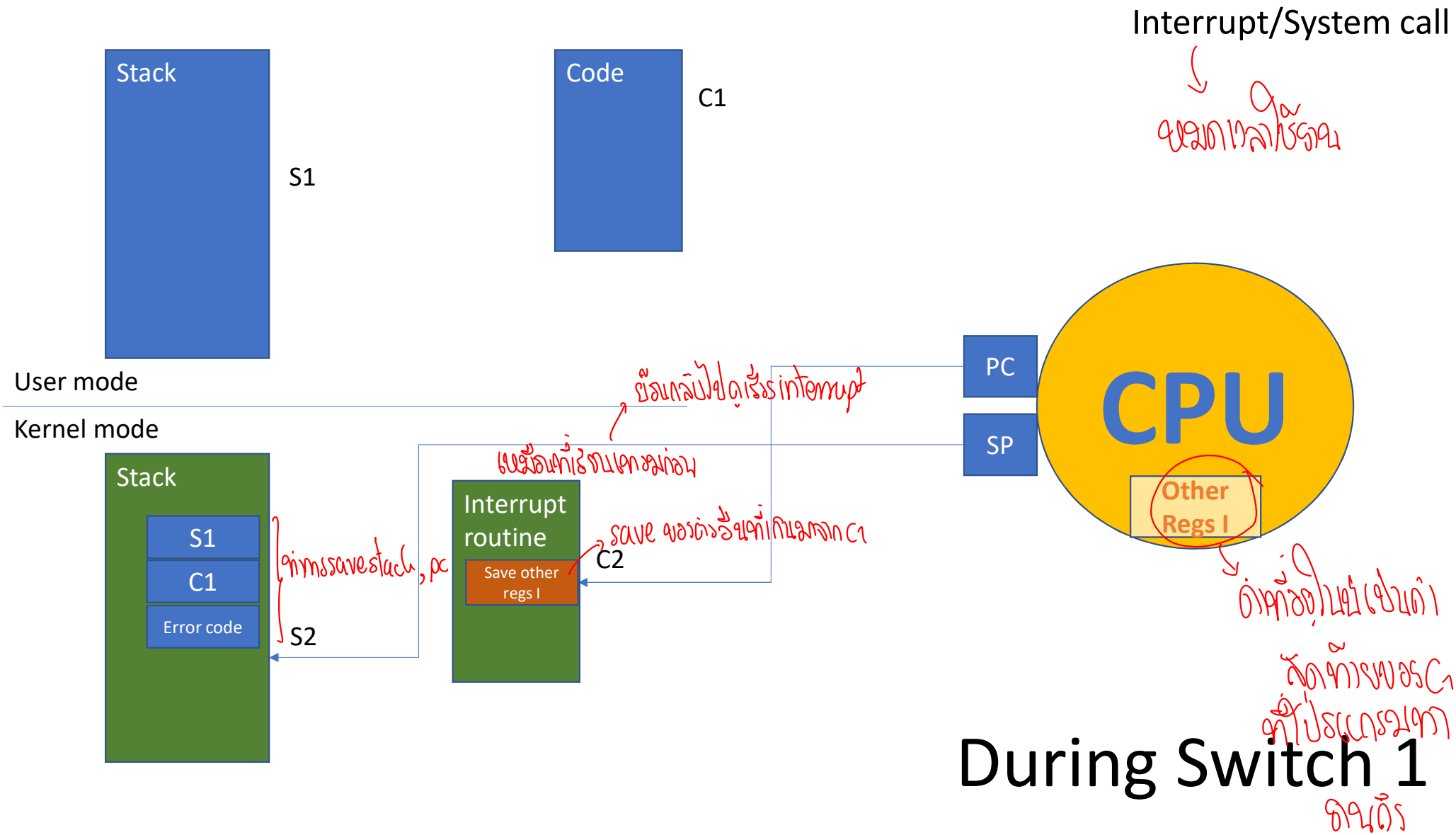
involuntary context switch

↳ ทั้ง 2 แบบทำงานคล้ายกัน

ระบบทำงานเหมือนกัน

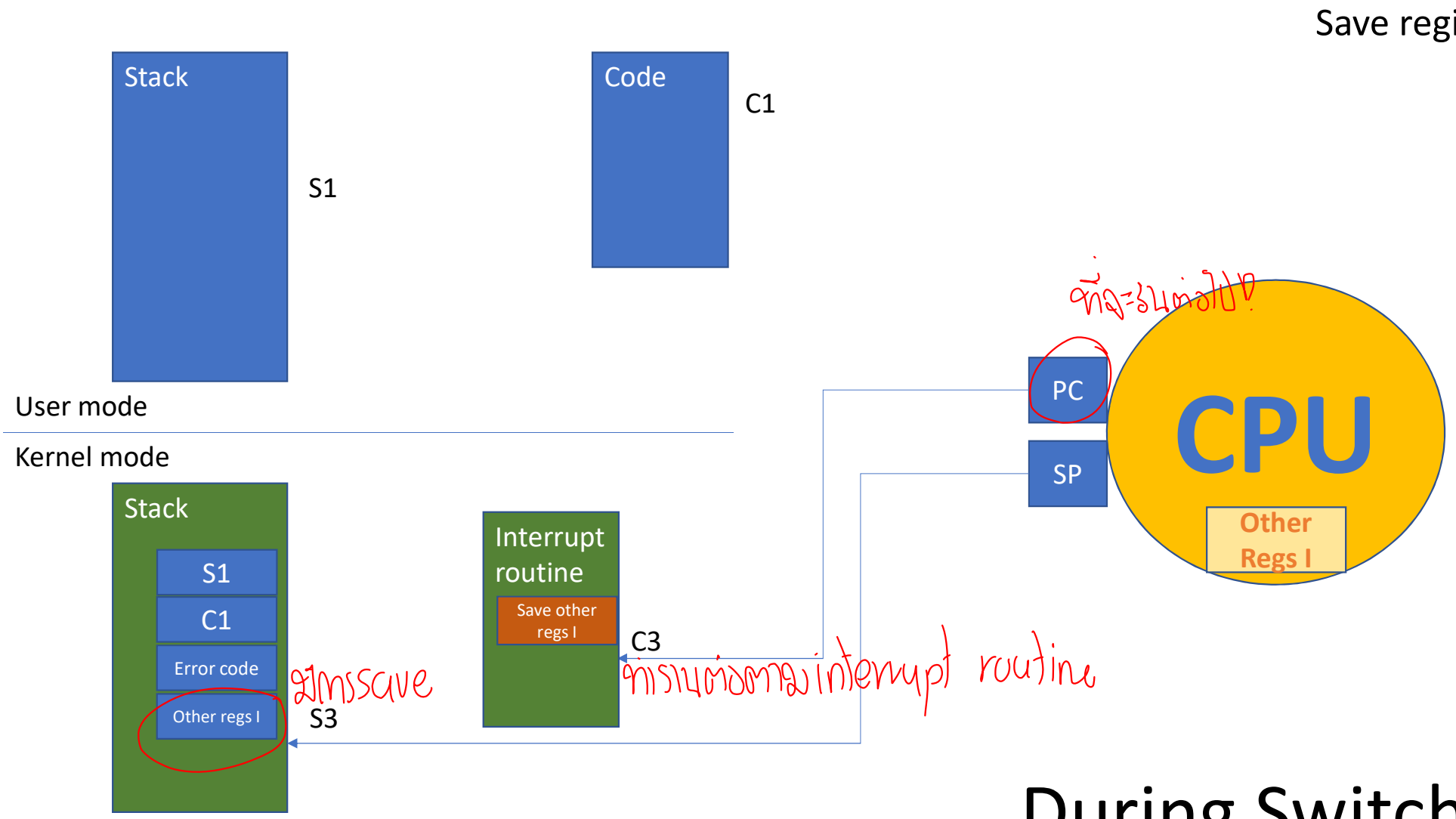


Before Switch



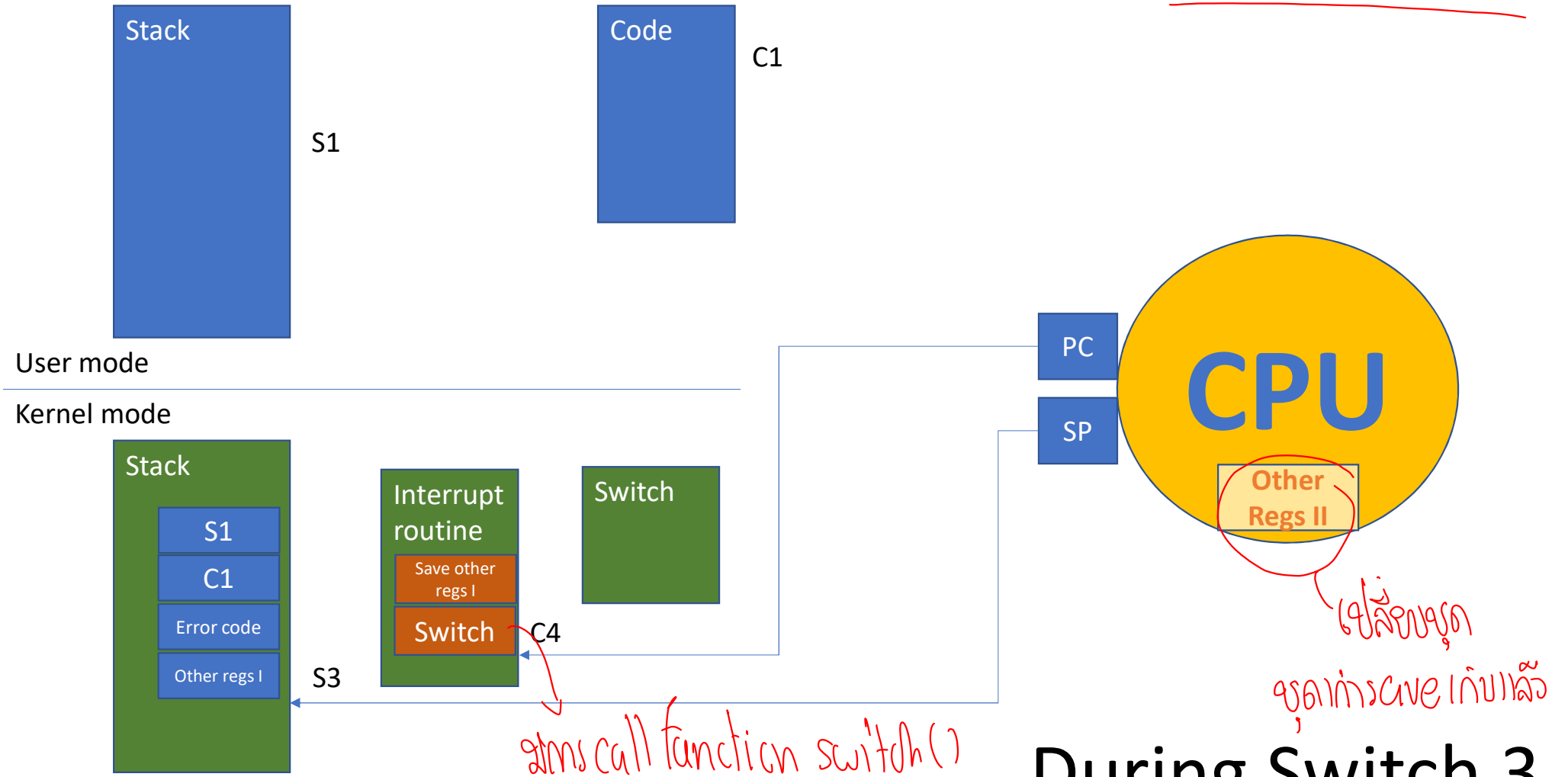
During Switch 1

Save registers



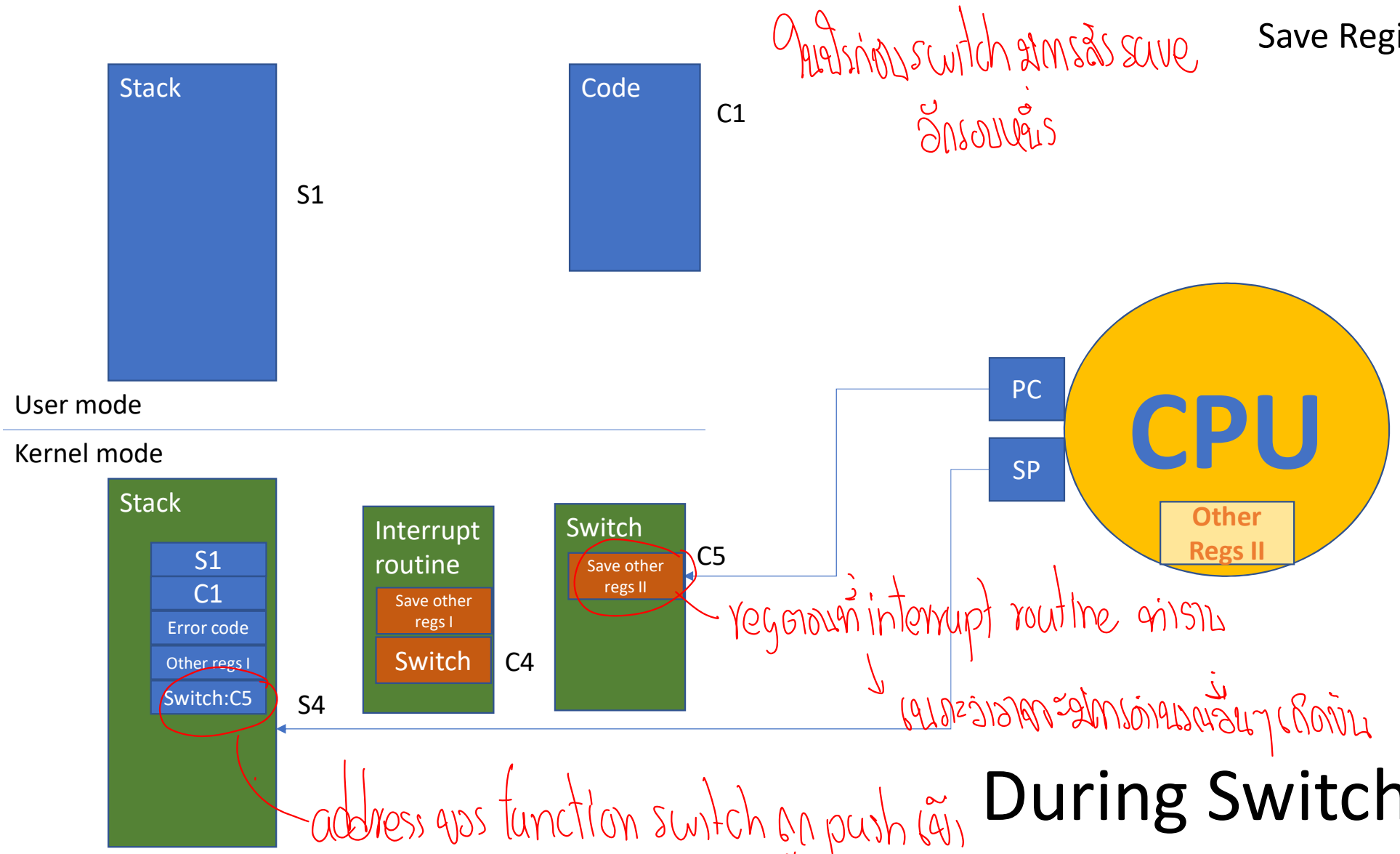
During Switch 2

Call Switch() System call



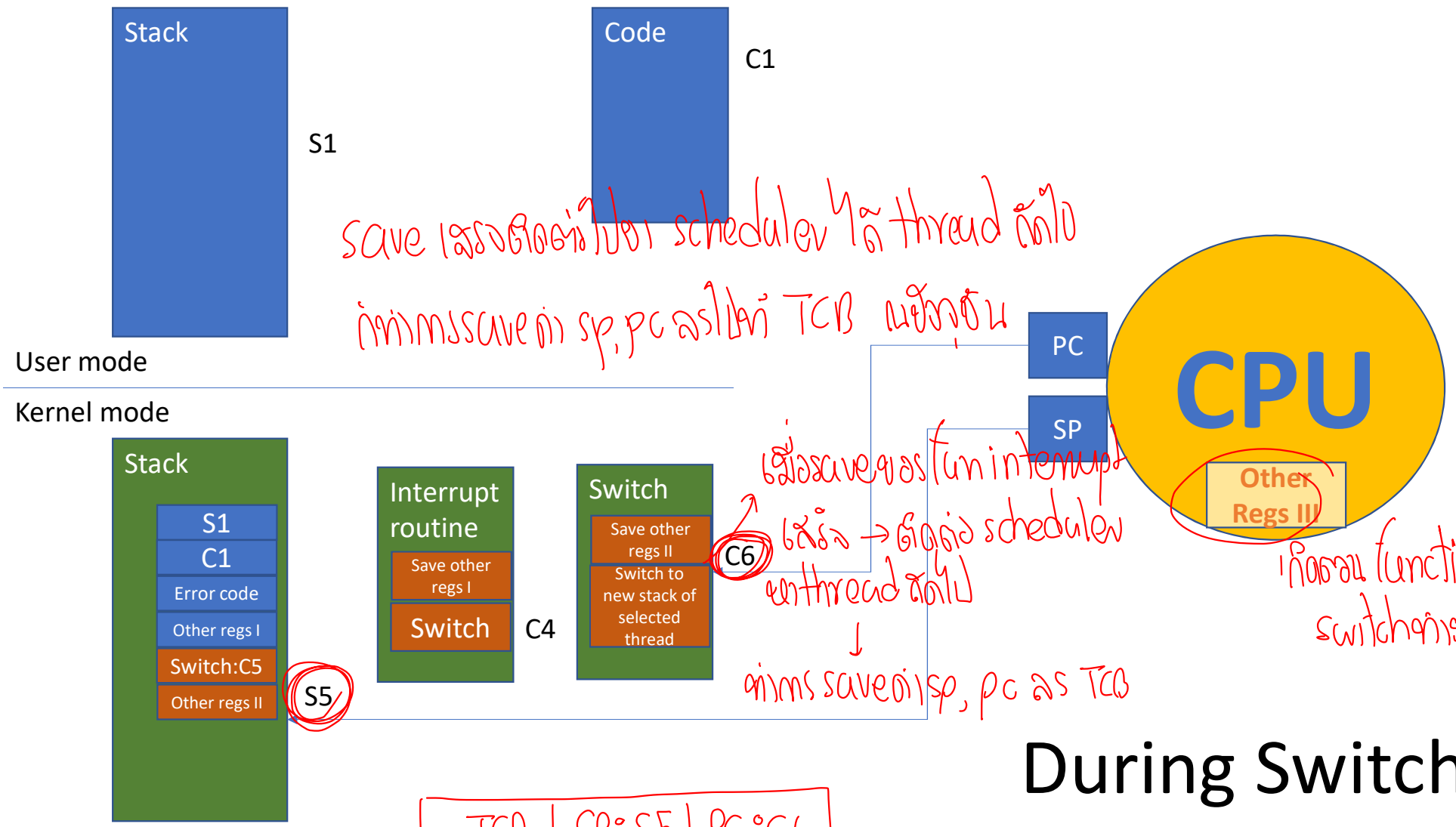
During Switch 3

Save Registers

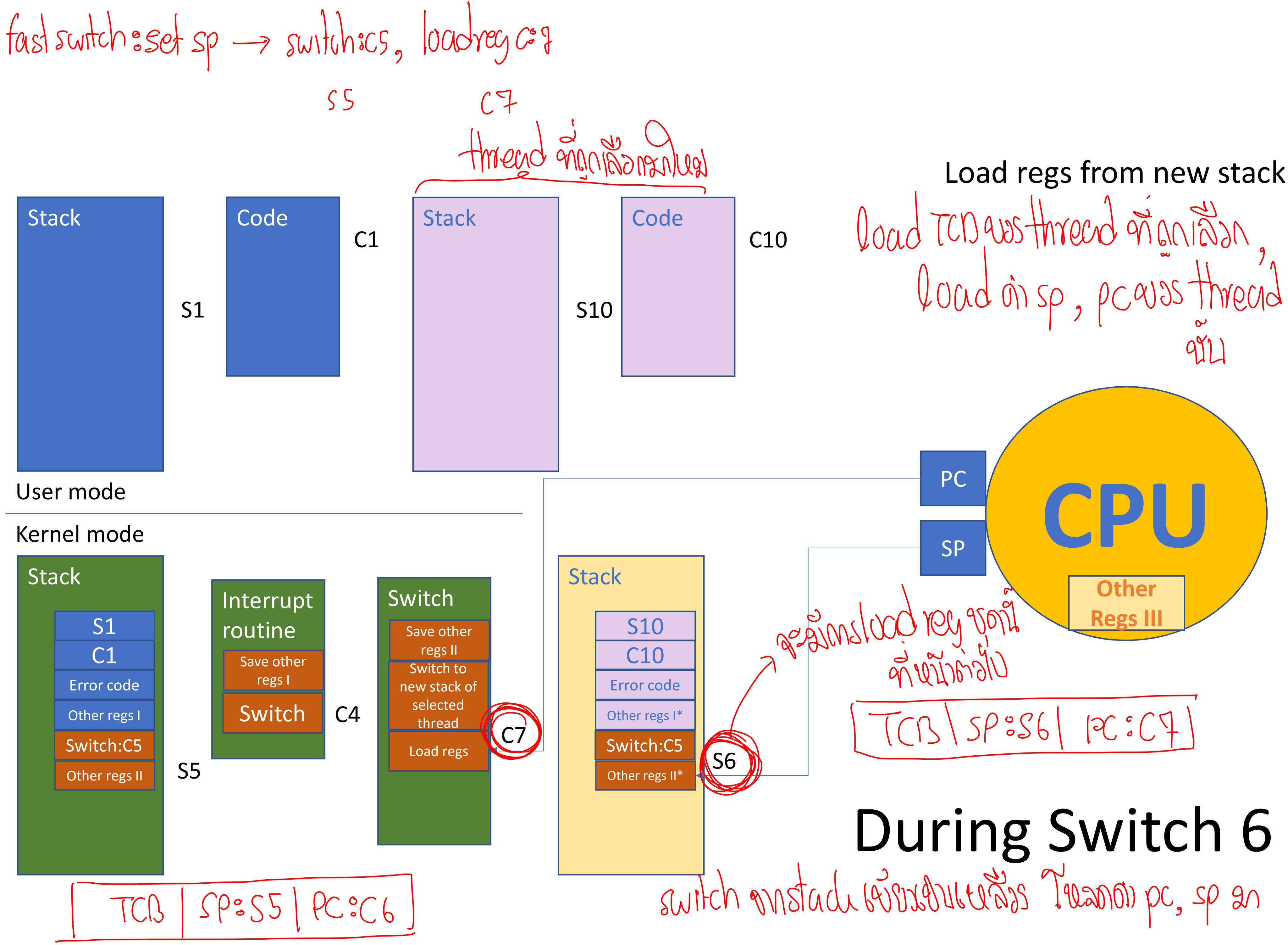


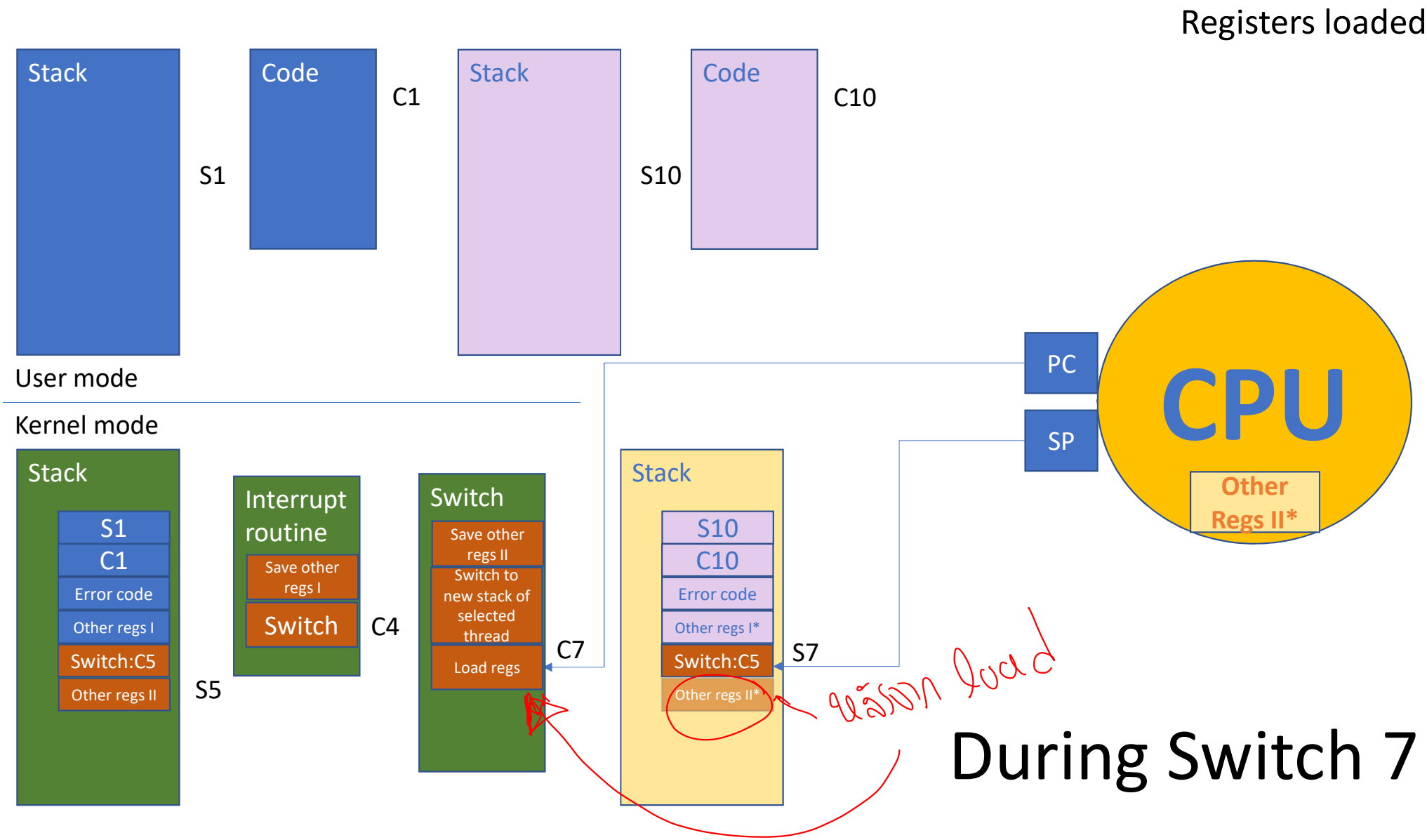
During Switch 4

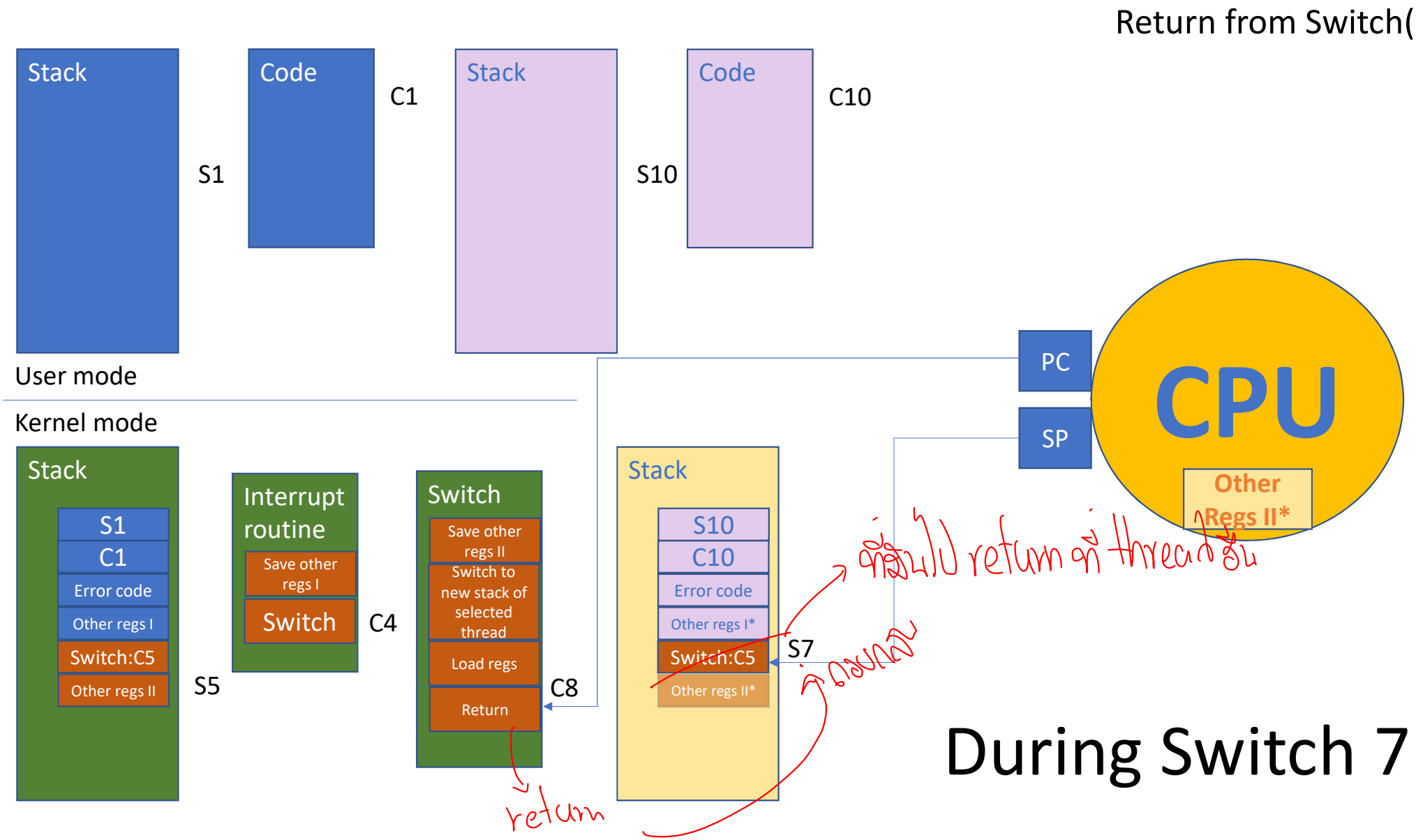
Switch to new stack



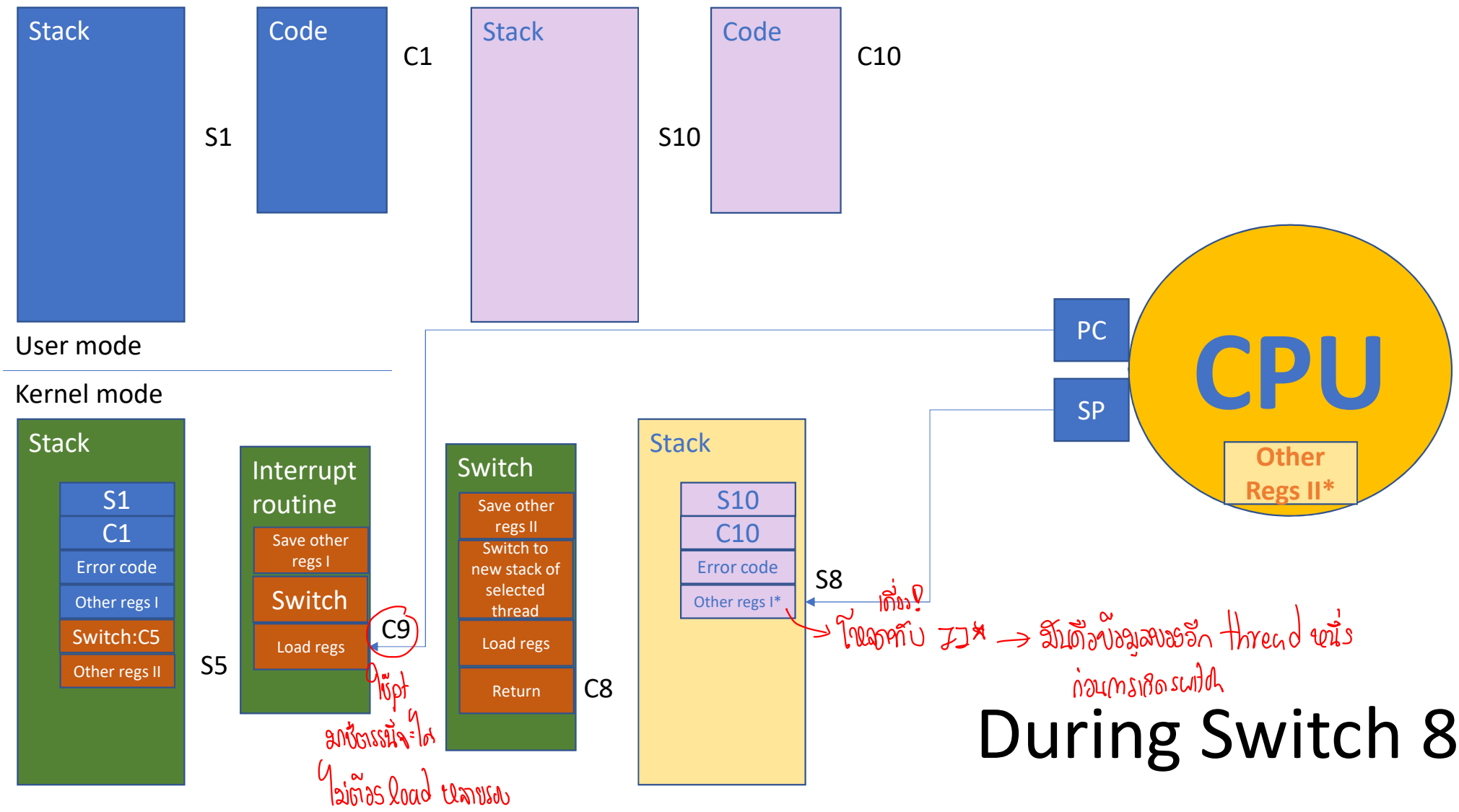
During Switch 5

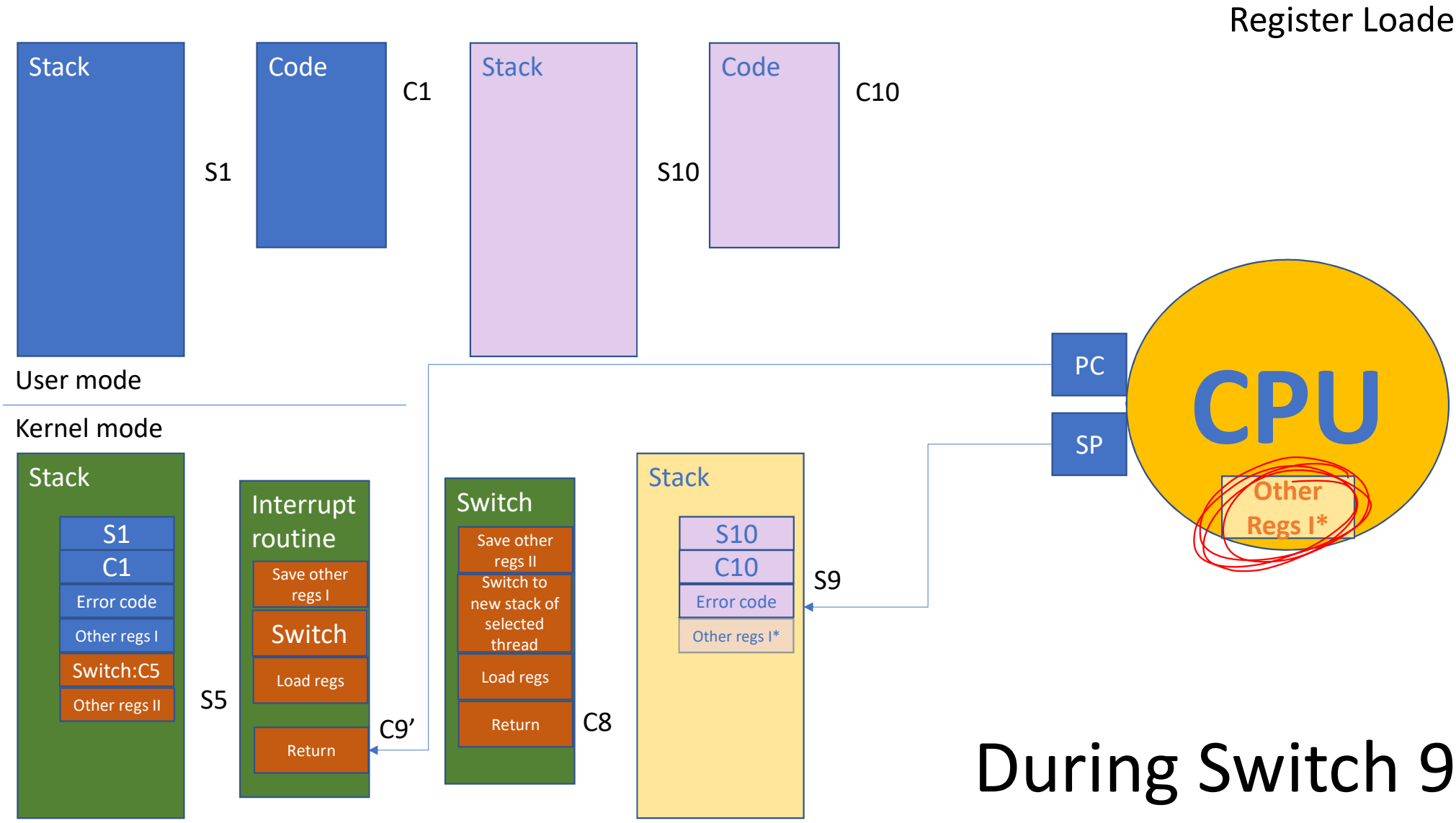

fast switch: set sp → switch:C5, loadreg C:7
S5
C7
thread ที่ถูกเลือกใหม่
Load regs from new stack
load TCB ของ thread ที่ถูกเลือก, load ค่า sp, pc ของ thread ขึ้น
Stack
Code
C1
S1
Stack
Code
S10
C10
User mode
Kernel mode
PC
SP
CPU
Other Regs III
Stack
S1
C1
Error code
Other regs I
Switch:C5
Other regs II
S5
Interrupt routine
Save other regs I
Switch
C4
Switch
Save other regs II
Switch to new stack of selected thread
Load regs
C7
Stack
S10
C10
Error code
Other regs I*
Switch:C5
Other regs II*
S6
TCB | SP:S6 | PC:C7
During Switch 6
TCB | SP:S5 | PC:C6
switch จาก stack เพื่อเซฟเสร็จแล้ว ไปโหลดค่า pc, sp มา

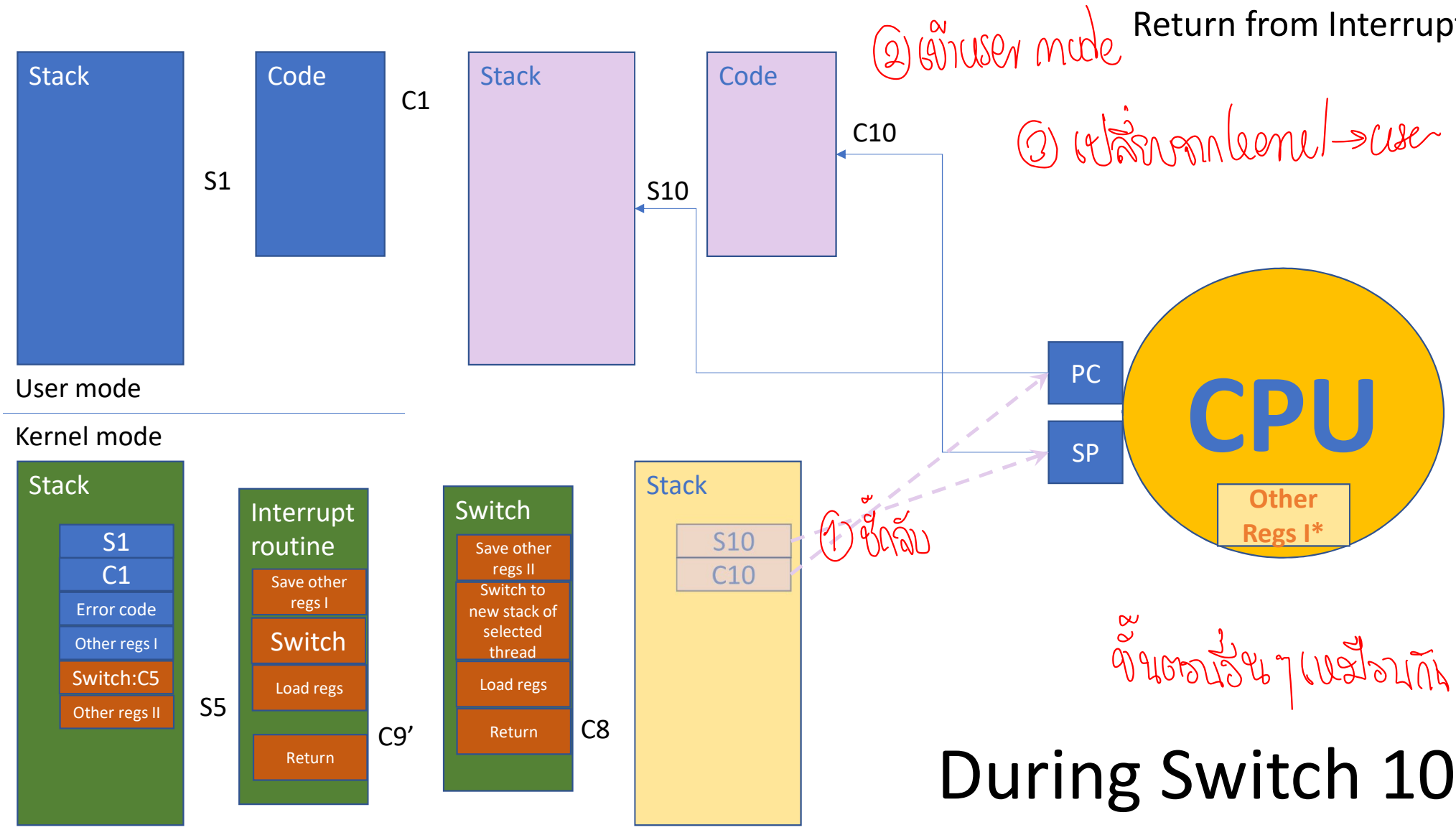

Registers loaded
Stack
S1
Code
C1
Stack
S10
Code
C10
User mode
Kernel mode
PC
SP
CPU
Other Regs II*
Stack
S1
C1
Error code
Other regs I
Switch:C5
Other regs II
S5
Interrupt routine
Save other regs I
Switch
C4
Switch
Save other regs II
Switch to new stack of selected thread
Load regs
C7
Stack
S10
C10
Error code
Other regs I*
Switch:C5
Other regs II*
S7
When load
During Switch 7


Return from Switch()
Stack
S1
Code
C1
Stack
S10
Code
C10
User mode
Kernel mode
PC
SP
CPU
Other Regs II*
Stack
S1
C1
Error code
Other regs I
Switch:C5
Other regs II
S5
Interrupt routine
Save other regs I
Switch
C4
Switch
Save other regs II
Switch to new stack of selected thread
Load regs
Return
C8
Stack
S10
C10
Error code
Other regs I*
Switch:C5
Other regs II*
S7
return
During Switch 7

Load registers




Register Loaded
Stack
S1
Code
C1
Stack
S10
Code
C10
User mode
Kernel mode
PC
SP
CPU
Other Regs I*
Stack
S1
C1
Error code
Other regs I
Switch:C5
Other regs II
S5
Interrupt routine
Save other regs I
Switch
Load regs
Return
C9'
Switch
Save other regs II
Switch to new stack of selected thread
Load regs
Return
C8
Stack
S10
C10
Error code
Other regs I*
S9
During Switch 9


Return from Interrupt
② เข้าuser mode
③ เปลี่ยนจากkernel→user
Stack
Code
C1
S1
Stack
Code
C10
S10
User mode
Kernel mode
PC
SP
CPU
Other Regs I*
Stack
S1
C1
Error code
Other regs I
Switch:C5
Other regs II
S5
Interrupt routine
Save other regs I
Switch
Load regs
Return
C9'
Switch
Save other regs II
Switch to new stack of selected thread
Load regs
Return
C8
Stack
S10
C10
① ชี้กลับ
ขั้นตอนปรับๆเปลี่ยนถ่าย
During Switch 10
interrupt = เหมือน thread yield เฉพาะ interrupt routine

Thread

Experiment

Experiment #1

Test order of thread execution and thread switch

Thread 1 , Thread 2 สลับกันทำงาน

```
1   // simple thread - test order
2   using System;
3   using System.Threading;
4
5   namespace Lab_OS_Concurrency
6   {
7       class Program
8       {
9           static void TestThread1()
10          {
11              int i;
12              for (i = 0; i < 100; i++)
13                  Console.WriteLine("Thread# 1 i={0}", i);
14          }
15          static void TestThread2()
16          {
17              int i;
18              for (i = 0; i < 100; i++)
19                  Console.WriteLine("Thread# 2 i={0}", i);
20          }
21
22          static void Main(string[] args)
23          {
24              Thread th1 = new Thread(TestThread1);
25              Thread th2 = new Thread(TestThread2);
26              th1.Start();
27              th2.Start();
28          }
29      }
30  }
```

Experiment #2

- Resource sharing among threads

แล้วผลลัพธ์ที่ได้จะเป็นอย่างไร

```
1   //test resource sharing
2   using System;
3   using System.Threading;
4
5   namespace Lab_OS_Concurrency01
6   {
7       class Program
8       {
9           static int resource = 10000;
10          static void TestThread1()
11          {
12              Console.WriteLine("Thread# 1 i={0}", resource);
13          }
14          static void TestThread2()
15          {
16              Console.WriteLine("Thread# 2 i={0}", resource);
17          }
18
19          static void Main(string[] args)
20          {
21              Thread th1 = new Thread(TestThread1);
22              Thread th2 = new Thread(TestThread2);
23              th1.Start();
24              th2.Start();
25          }
26      }
27  }
```

Experiment #3

- Pause a thread

แล้วแต่การเข้าใช้งาน → 10,000
→ 55,555

ถ้าใส่ sleep จะเยอะ 55,555

```
//test pause a thread
using System;
using System.Threading;

namespace Lab_OS_Concurrency02
{
    class Program
    {
        static int resource = 10000;
        static void TestThread1()
        {
            resource = 55555;
        }

        static void Main(string[] args)
        {
            Thread th1 = new Thread(TestThread1);
            th1.Start();
            //Thread.Sleep(10);
            Console.WriteLine("resource={0}", resour
        }
    }
}
```

Experiment #3.1

- Pause a thread #2

```
1   //test pause #2
2   using System;
3   using System.Threading;
4
5   namespace Lab_OS_Concurrency01
6   {
7       class Program
8       {
9           static int resource = 10000;
10          static void TestThread1()
11          {
12              int i;
13              for (i = 0; i < 45555; i++)
14              {
15                  resource++;
16                  Console.Write(".");
17              }
18          }
19
20          static void Main(string[] args)
21          {
22              Thread th1 = new Thread(TestThread1);
23              th1.Start();
24              Thread.Sleep(10);
25              Console.WriteLine("Resource = {0}", resource);
26          }
27      }
28  }
```

→ ดูผลลัพธ์ที่ได้

Experiment #3.1 desired result

Experiment #4

- Join thread

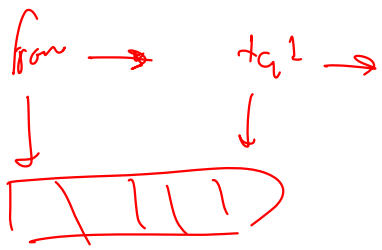
ใช้คำสั่ง join เพื่อให้ได้ผลลัพธ์ที่ถูกต้องเสมอ

```
1   //test pause #2
2   using System;
3   using System.Threading;
4
5   namespace Lab_OS_Concurrency01
6   {
7       class Program
8       {
9           static int resource = 10000;
10          static void TestThread1()
11          {
12              int i;
13              for (i = 0; i < 45555; i++)
14              {
15                  resource++;
16                  Console.Write(".");
17              }
18          }
19
20          static void Main(string[] args)
21          {
22              Thread th1 = new Thread(TestThread1);
23              th1.Start();
24              //Thread.Sleep(10);
25              th1.Join();
26              Console.WriteLine("Resource = {0}", resource);
27          }
28      }
29  }
```

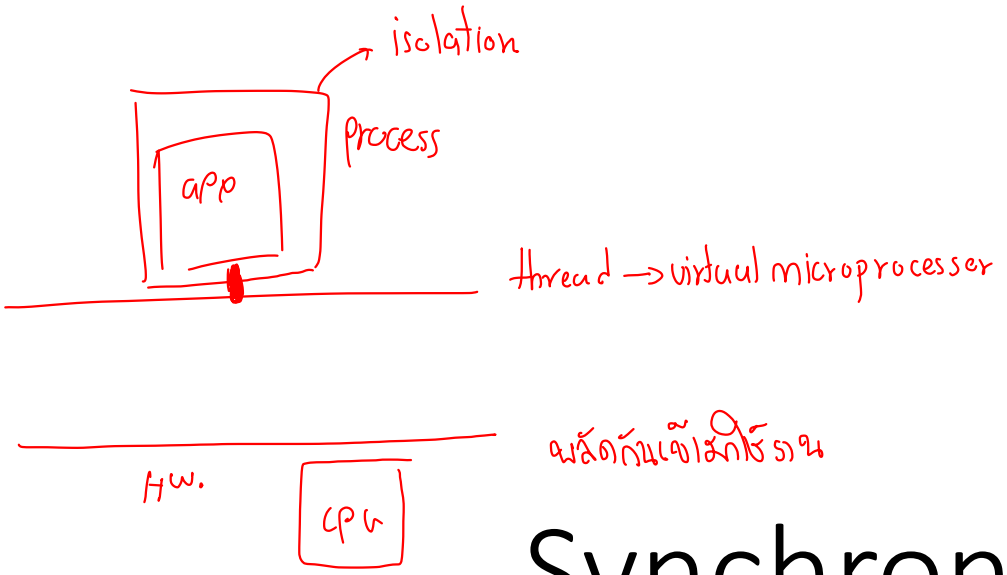
ให้พิจารณา Pseudo code ต่อไปนี้

```
define MAX 100;
int front=0, tail=0;
int BUFF[101];

int get()
{
      int item;
      item = BUFF[front%MAX];
      front++;
      return item;
}
void put(int item)
{
      BUFF[tail%MAX] = item;
      tail++;
}
```



1. อธิบายการทำงาน
2. ในกรณีที่โปรแกรมเป็นแบบ single thread
   a. ปัญหาที่อาจเกิดขึ้นคืออะไร
   b. ให้เสนอวิธีแก้ปัญหาดังกล่าว
3. ในกรณีที่โปรแกรมถูกพัฒนาเป็นแบบ Multithread
   a. อธิบายความเป็นไปได้ของ code นี้ในการทำงานแบบ multithread
   b. ปัญหาที่อาจเกิดขึ้น
   c. แนวทางแก้ไขปัญหา



Synchronization

Experiments

1. No synchronization

```
using System.Threading;

namespace TestThreadNolock
{
    class Program
    {
        private static int x = 0;
        static object _lock = new object();

        static void FuncA()
        {
            int xx = 0;
            while (xx < 50)
            {
                Console.WriteLine("FuncA: round:{0} x={1}", xx, x);
                x++;
                xx++;
            }
        }

        static void FuncB()
        {
            int xx = 0;
            while (xx < 50)
            {
                Console.WriteLine("==FuncB: round:{0} x={1}", xx, x);
                x++;
                xx++;
            }
        }

        static void Main(string[] args)
        {
            Thread A = new Thread(new ThreadStart(FuncA));
            Thread B = new Thread(new ThreadStart(FuncB));
            A.Start();
            B.Start();
        }
    }
}
```

เขียนโปรแกรมเกี่ยวกับ thread → shared resoure → ใช้ทรัพยากรร่วม → เกิดปัญหา

→ race condition

↓

how to protect → ไม่ให้เข้าถึง ได้ทีละคน

↓

กุญแจแทนไม่ให้เข้าประตู

↓

thread เดียวที่เข้าไป ใช้ only

↓

เราเรียกว่า lock

2. Lock

```
using System.Threading;

namespace TestThreadNolock
{
    class Program
    {
        private static int x = 0;
        static object _lock = new object();

        static void FuncA()
        {
            int xx = 0;
            while (xx < 50)
            {
                lock (_lock)
                {
                    Console.WriteLine("FuncA: round:{0} x={1}", xx, x);
                    x++;
                }
                xx++;
            }
        }

        static void FuncB()
        {
            int xx = 0;
            while (xx < 50)
            {
                lock (_lock)
                {
                    Console.WriteLine("==FuncB: round:{0} x={1}", xx, x);
                    x++;
                }
                xx++;
            }
        }

        static void Main(string[] args)
        {
            Thread A = new Thread(new ThreadStart(FuncA));
            Thread B = new Thread(new ThreadStart(FuncB));
            A.Start();
            B.Start();
        }
    }
}
```

ขั้นตอนการ request key → if success ให้โปรแกรมทำงานต่อได้ → not ยังทำงานไม่ได้

→ จะไปได้ต้องมี key

→ ทำการคืนกุญแจ

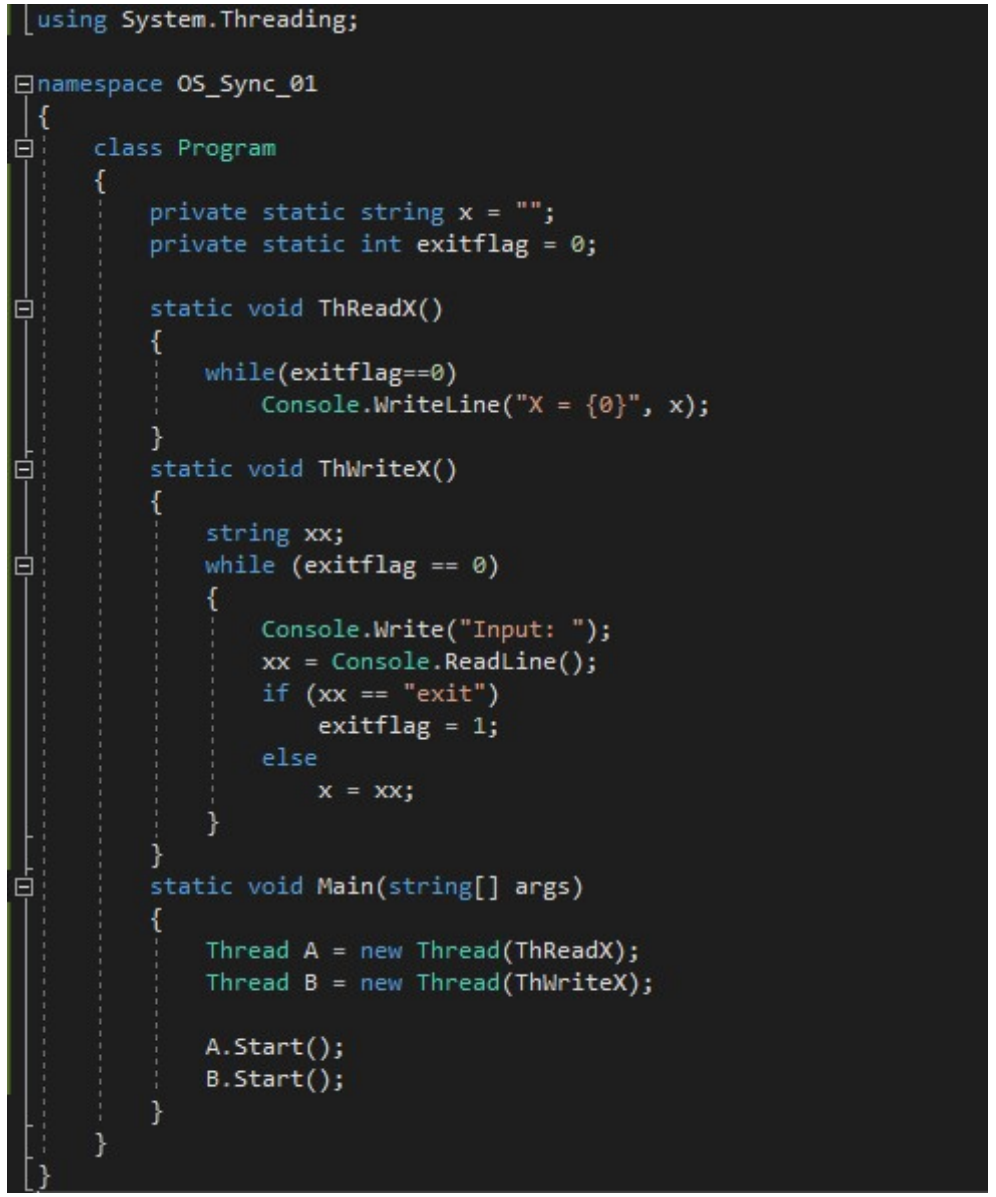
overhead เพิ่มขึ้น → ขอกุญแจ 50 รอบ → ทำไม่ได้ต้องรอบดักเยอะมาก → ขอตรงนี้

ถ้าเอา lock ครอบ while เอาไว้ → while ทำงานไม่ได้ → context switch → thread อื่นเข้าใช้ lock นี้ไม่ได้ → ต้องรอจนกว่าจะ lock → set ready ให้อีก thread

ต้องรอ thread อื่น ← thread อันไหนเข้าที่ต้องทำงานให้เสร็จ

ทำงานเสร็จ → เสีย concept concurrent

3. No Synchronization



```
using System.Threading;

namespace OS_Sync_01
{
    class Program
    {
        private static string x = "";
        private static int exitflag = 0;

        static void ThReadX()
        {
            while(exitflag==0)
                Console.WriteLine("X = {0}", x);
        }
        static void ThWriteX()
        {
            string xx;
            while (exitflag == 0)
            {
                Console.Write("Input: ");
                xx = Console.ReadLine();
                if (xx == "exit")
                    exitflag = 1;
                else
                    x = xx;
            }
        }
        static void Main(string[] args)
        {
            Thread A = new Thread(ThReadX);
            Thread B = new Thread(ThWriteX);

            A.Start();
            B.Start();
        }
    }
}
```

4. Try #1

เราใช้ thread ในแต่ละการเขียนนะ

```
using System.Threading;

namespace OS_Sync_03
{
    class Program
    {
        private static string x = "";
        private static int exitflag = 0;
        private static int updateFlag = 0;
        private static object _Lock;

        static void ThReadX(Object i)
        {
            while (exitflag == 0)
            {
                while (updateFlag == 0) ;
                if (x!="exit")
                    Console.WriteLine("Thread {0} : X = {1}", i, x);
                updateFlag = 0;
            }
        }
        static void ThWriteX()
        {
            string xx;
            while (exitflag == 0)
            {
                Console.Write("Input: ");
                xx = Console.ReadLine();
                if (xx == "exit")
                    exitflag = 1;
                x = xx;
                updateFlag = 1;
            }
        }
        static void Main(string[] args)
        {
            Thread A = new Thread(ThReadX);
            Thread B = new Thread(ThWriteX);

            A.Start(1);
            B.Start();
        }
    }
}
```

ใช้ flag ของเขาเพ

5. Try #2

```
using System.Threading;

namespace OS_Sync_03
{
    class Program
    {
        private static string x = "";
        private static int exitflag = 0;
        private static int updateFlag = 0;
        private static object _Lock;

        static void ThReadX(Object i)
        {
            while (exitflag == 0)
            {
                while (updateFlag == 0) ;
                if (x!="exit")
                    Console.WriteLine("Thread {0} : X = {1}", i, x);
                updateFlag = 0;
            }
        }
        static void ThWriteX()
        {
            string xx;
            while (exitflag == 0)
            {
                Console.Write("Input: ");
                xx = Console.ReadLine();
                if (xx == "exit")
                    exitflag = 1;
                x = xx;
                updateFlag = 1;
            }
        }
        static void Main(string[] args)
        {
            Thread A = new Thread(ThReadX);
            Thread B = new Thread(ThWriteX);
            Thread C = new Thread(ThReadX);
            Thread D = new Thread(ThReadX);

            A.Start(1);
            B.Start();
            C.Start(2);
            D.Start(3);
        }
    }
}
```

ถ้าเรายังไม่รู้จัก thread ใด
thread หนึ่งรอนานไปแล้ว
↓
ให้ thread ตัวอื่นแสดง
ผล thread เดิมจะ
ทำอย่างไร ?

6. Condition Variable

```
using System.Threading;

namespace OS_Sync_04
{
    class Program
    {
        private static string x = "";
        private static int exitflag = 0;
        private static int updateFlag = 0;
        private static object _Lock = new object();

        static void ThReadX(Object i)
        {

            while (exitflag == 0)
            {
                lock (_Lock)
                {
                    while (updateFlag == 0)
                        Monitor.Wait(_Lock);
                    if (x != "exit")
                        Console.WriteLine("Thread {0} : X = {1}", i, x);
                    updateFlag = 0;
                }
            }
            Console.WriteLine("Thread {0} exit", i);
        }
        static void ThWriteX()
        {
            string xx;
            while (exitflag == 0)
            {
                lock (_Lock)
                {
                    Console.Write("Input: ");
                    xx = Console.ReadLine();
                    if (xx == "exit")
                        exitflag = 1;
                    x = xx;
                    updateFlag = 1;
                    Monitor.Pulse(_Lock);
                    Thread.Sleep(100);
                }
            }
        }
        static void Main(string[] args)
        {
            Thread A = new Thread(ThReadX);
            Thread B = new Thread(ThWriteX);
            Thread C = new Thread(ThReadX);
            Thread D = new Thread(ThReadX);

            A.Start(1);
            B.Start();
            C.Start(2);
            D.Start(3);
        }
    }
}
```

Definitions

Race condition: output of a concurrent program depends on the order of operations between threads

Mutual exclusion: only one thread does a particular thing at a time

- **Critical section:** piece of code that only one thread can execute at once

Lock: prevent someone from doing something

- Lock before entering critical section, before accessing shared data
- Unlock when leaving, after done accessing shared data
- Wait if locked (all synchronization involves waiting!)

Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - Only the lock holder can release
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

Condition Variables

- Waiting inside a critical section
  - Called only when holding a lock
- Wait: atomically release lock and relinquish processor
  - Reacquire the lock when wakened
- Signal: wake up a waiter, if any
- Broadcast: wake up all waiters, if any

เรื่องในการสร้างหรือดูแลรายชื่อ condition variable

Pre/Post Conditions

- What is state of the bounded buffer at lock acquire?
  - front <= tail
  - front + MAX >= tail
- These are also true on return from wait
- And at lock release
- Allows for proof of correctness

Pre/Post Conditions

```
methodThatWaits() {
    lock.acquire();
    // Pre-condition: State is consistent

    // Read/write shared state

    while (!testSharedState()) {
        cv.wait(&lock);
    }
    // WARNING: shared state may
    // have changed!  But
    // testSharedState is TRUE
    // and pre-condition is true

    // Read/write shared state
    lock.release();
}
```

```
methodThatSignals() {
    lock.acquire();
    // Pre-condition: State is consistent

    // Read/write shared state

    // If testSharedState is now true
    cv.signal(&lock);

    // NO WARNING: signal keeps lock

    // Read/write shared state
    lock.release();
}
```

Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state
- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up
- Wait atomically releases lock
  - What if wait, then release?
  - What if release, then wait?

Condition Variables, cont'd

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast put thread on ready list
  - When lock is released, anyone might acquire it
- Wait MUST be in a loop

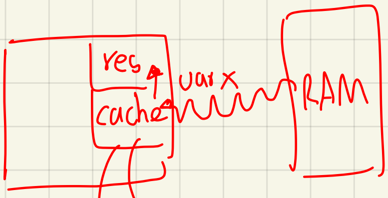
```
while (needToWait()) {
    condition.Wait(lock);
}
```

- Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Remember the rules

- Use consistent structure
- Always use locks and condition variables
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop
- Never spin in sleep()

การจัดเก็บ data ระหว่าง cpu



ลดความต่างของความเร็วระหว่าง reg and ram

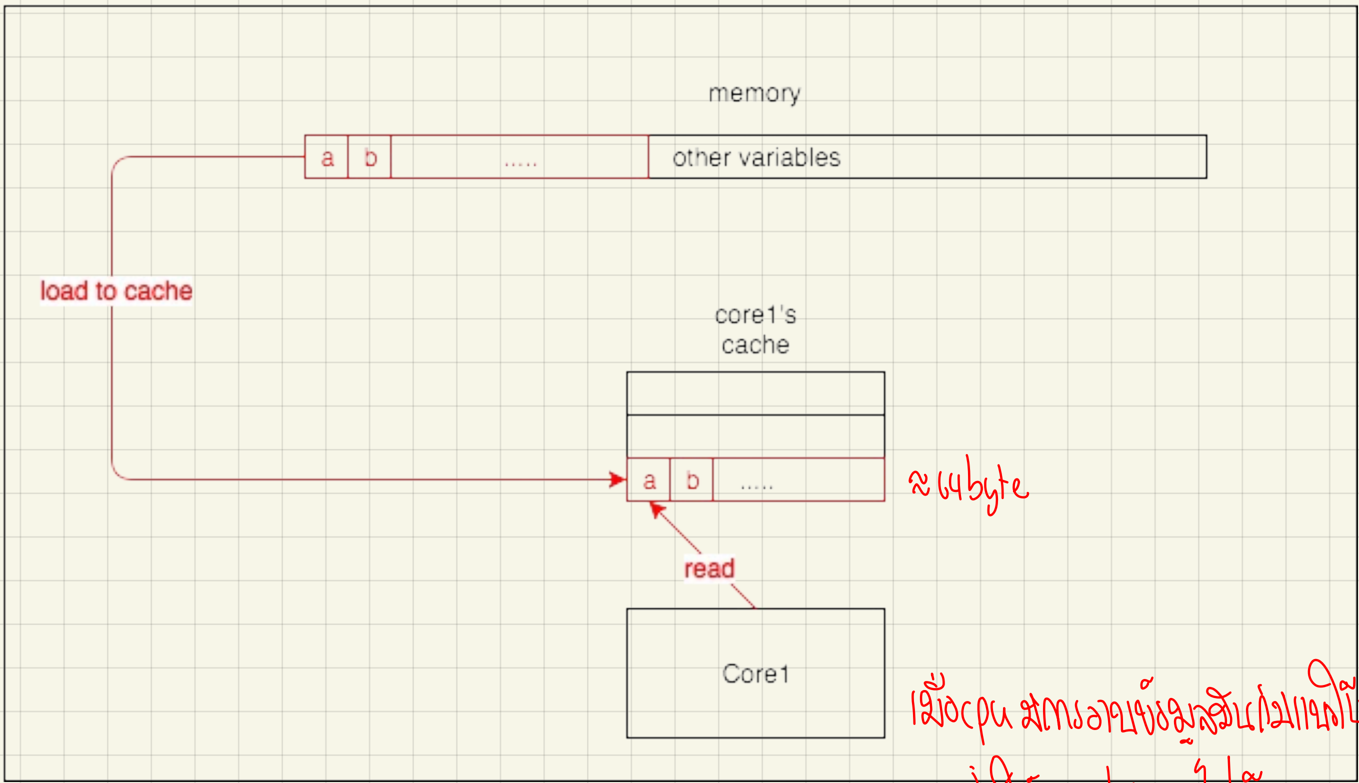
ใช้ c

↳ ผ่าน charge , discharge

เมื่อค่าที่ reg update ค่าที่อื่นต้อง update ตามด้วย (2 ที่ที่แชร์)

↳ ไม่งั้นจะอ่านได้ค่าที่ไม่เป็นปัจจุบัน

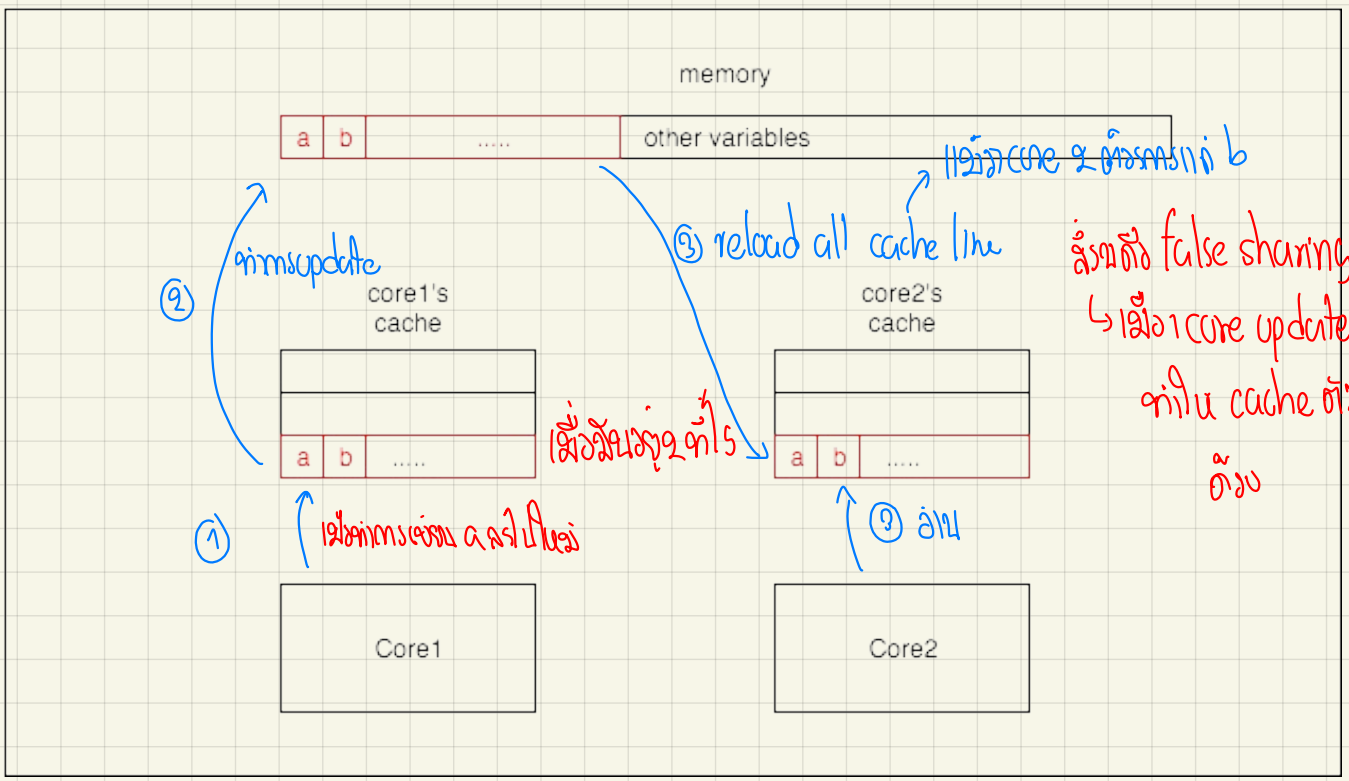
ปัญหา false sharing



เมื่อ cpu จะทำการอ่านข้อมูลสักตัวมันจะแคชไปทั้ง
ที่อยู่ตัวแปรใกล้ๆ ไปด้วย
เลยทำงานเร็วมากที่เป็นจุด

แล้วปัญหานี้จะเกิดขึ้นเมื่อไหร่?

↳ เดาสิเดา ๆ ๆ ๆ ๆ ๆ ๆ ๆ



The common way to solve that problem is cache padding: padding some meaningless variables between variables. That would force one variable to occupy a core's cache line alone, so when other cores update other variables would not make that core reload the variable from memory.

ไม่สามารถทำนาย dead lock ได้ → ไม่ได้เกิดขึ้นตลอดเวลา → เวลาไม่ตรงที่ แล้วแต่ดวง

ถ้าทำงานกับ >1 share resource , >1 thread ต้องระวังหลีกเลี่ยง dead lock , starvation ต้องออกแบบดีๆ

เช่น มี lock 1 , 2

thread a ถือ 1 ปล่อย 2 → ถือแล้วเข้าไป wait stage

thread b ต้องการทั้ง 2 แต่ได้แค่ 2

↳ thread b จะคืน lock เมื่อได้ 1 ไม่ครบเซต

↳ thread a รอ thread 2 ปล่อย

วนไปรอบ dead lock

↑

"chain request"

↓

ไม่ได้เกิดจากสิ่งที่รัน → up to luck

* อย่าถือ lock เข้าไปใน wait stage

กลไกที่ทำให้เกิด

วิธีการหลีกเลี่ยง dead lock

↳ ควบคุมพฤติกรรมของโปรแกรม → การใช้ lock กับ shared resource → เหมือนกันทุกตัว ทุกโปรแกรม

↳ การป้องกันพฤติกรรม wait by holding → แย่ง lock 1 ได้ 2 ไม่ได้ ให้ปล่อย lock ทั้งหมด

↳ ถือ lock ได้ถ้าขอเอาได้ 2 เท่านั้น

↳ เขียนโปรแกรมมาคอยดูว่า thread ไหนค้างเกิน → kill ทิ้ง

Scheduling

- Main points
  - Scheduling policy: what to do next, when there are multiple threads ready to run.
    - or multiple packets to send, or web request to server, ...
  - Definitions
    - Response time, throughput, predictability
  - Uniprocessor policies
    - FIFO, round robin, optimal
    - Multilevel feedback as approximation of optimal — only 3 above
  - Multiprocessors policies → ไม่ออกสอบของปี 2563
    - Affinity scheduling, gang scheduling
  - Queueing theory → ไม่ออกสอบของปีก่อนๆ
    - Can you predict/improve a system's response time?

Example

→ เปรียบเทียบ processor = server, task = client

- You manage a web site, that suddenly becomes wildly popular. Do you?
  - Buy more hardware → เกษตรกร
  - Implement a different scheduling policy? → ใช้ OS เพื่อมาช่วยจัดการเรื่องนี้
  - Turn away some users? Which ones? → Limit การใช้
- How much worse will performance get if the website become even more popular?

Definitions

- Task/Job: ตอนที่ user ป้อนเข้ามาในระบบ
- Latency/response time: How long does a task take to complete? เวลาที่ใช้ในการประมวลผล
- Throughput: How many tasks can be done per unit of time? จำนวนงานที่ทำเสร็จใน 1 หน่วยเวลา
- overhead: สิ่งที่ user ไม่ได้สั่งแต่ต้องทำ
- Fairness: การจัดสรรเวลาอย่างเท่าเทียม
- Predictability: ประสิทธิภาพเสมอต้น เสมอปลาย → เอาได้

↳ workload : กลุ่มของงาน

↳ preemptive scheduler : หมายถึง scheduler ที่ scheduler เข้าไปจัดสรรและได้ → เอา job อื่นออกมาทำงานแทน (เข้าไปยึด resource {cputime})

↳ work-conserving : processor ต้องทำงานตลอดเวลา ถ้ายังมีงานให้ทำใน queue

↳ Scheduling algorithm : algorithm ที่ใช้ workload เป็น input → เพื่อให้ throughput ↑ latency ↓ (เป็น output)

↳ Only preemptive, work-conserving schedulers to be considered
↳ พิจารณาแค่ระบบที่เป็น preemptive, work-conserving
↳ ที่เราเรียน
↳ ทำให้เกิด throughput ร้าย
↳ cpu ไม่สามารถพัก

First In First Out (FIFO) → มาก่อนทำงานก่อน → algo ที่ง่ายสุด

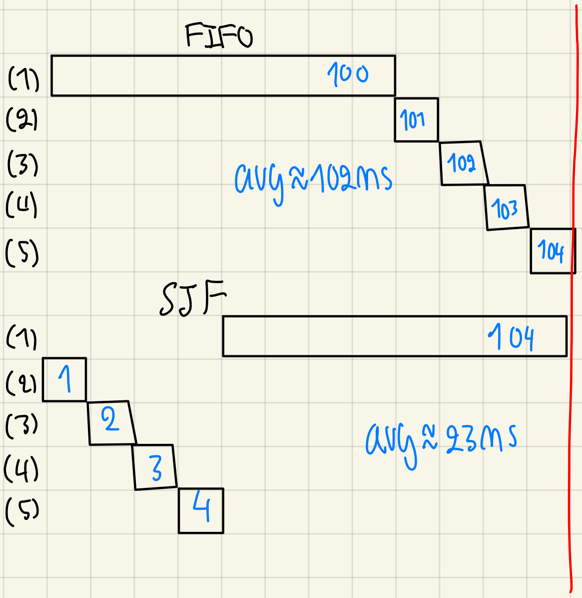
↳ ใช้ performance ไม่ดีในบางกรณี

Shortest Job First (SJF) → work แต่ละงานต้องรู้ → คุณจะรู้ได้ไงว่างานไหนใช้เวลาน้อย

↳ คนไหนเวลาน้อยที่สุดแทรกตัวได้เลย

FIFO vs SJF

104



ใช้เวลาทั้งหมด 104

มีประสิทธิภาพในเชิง tp, avg ดีกว่า

↳ ถ้าเราวัดแค่ throughput เท่ากัน

Question

↳ Claim : SJF is optimal for average response time? → ถ้าดึงสั้นๆ ออกก่อน ยาวมาทีหลัง

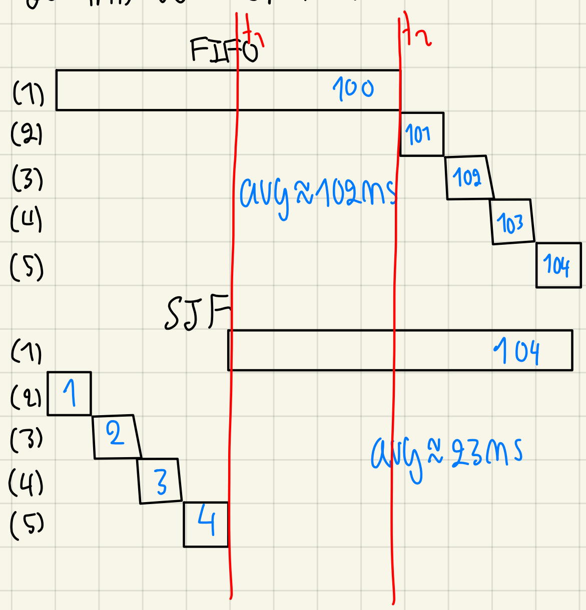
↳ Why? → ด้านบน

↳ Does SJF have any downside? → งานใหญ่ไม่ได้ทำสักที → โดนแซงแทรกตัว

↳ Is FIFO ever optimal? → เหมาะสุด → งานที่เข้ามาใช้เวลาเท่ากันหมด / ห่วยสุด → งานเวลายาวมาก่อน

Starvation and Sample Bias

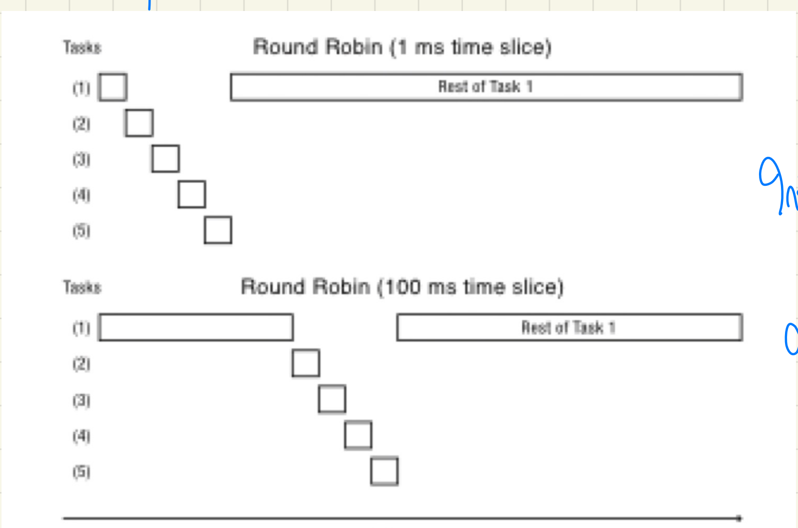
- ↳ Suppose you want to compare two scheduling algorithms
  - ↳ Create some infinite seqence of arriving tasks
  - ↳ Start measuring
  - ↳ Stop at somepoint
  - ↳ Compute average response time as the average for completed tasks between start and stop
- ↳ Is this valid or invalid?



วัด t_1 t_2 แบบนี้ไม่ได้นะ ตุเรศ

Round Robin → ขยายจากเอาข้อดีของ FIFO, SJF มาใช้ → ใช้งานได้จริง

- ↳ ทำการแบ่ง cpu time เท่ากันหมด → ช่วงเวลาที่ job ได้ = 1 time quantum → ไม่เสร็จต่อคิวใหม่
- ↳ ต้องทำการเลือกขนาดของ time quantum เหมาะสม?
  - ↳ ถ้ามากๆ $\lim_{n \to \infty}$ → กลายเป็น FIFO แทน
  - ↳ เล็กมากๆ → ค่าจะใกล้เคียง SJF → เพราะงานเล็กๆจบเลย



ใกล้เคียง sjf

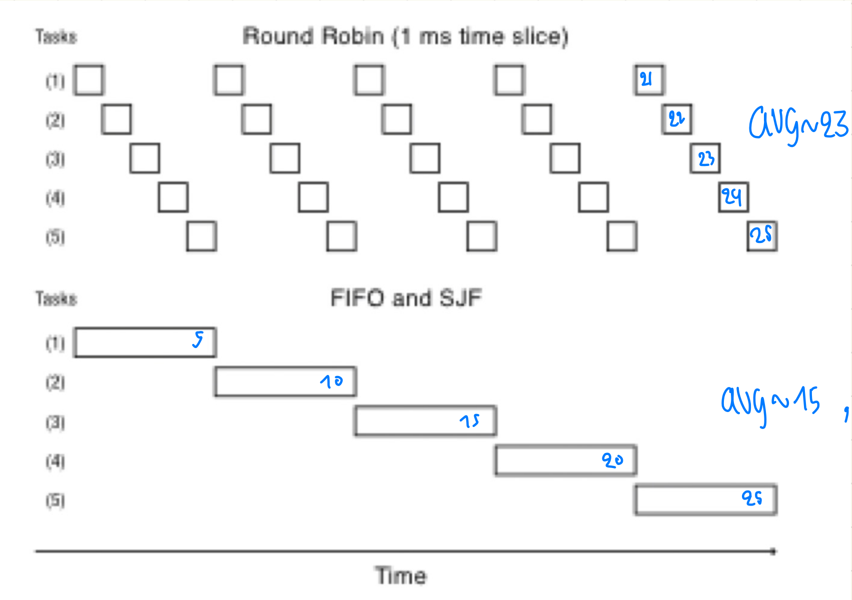
ใกล้เคียง fifo

Round Robin vs. FIFO

↳ assuming zero-cost time slice, is Round Robin always better than FIFO?

(ไม่ใช้เวลาในการ slice)

ถ้างานทุกงานใช้เวลาเท่ากัน → Round Robin แย่กว่า ไม่ได้ดีกว่าเสมอไป



avg ~23 ณ เวลาที่ทำเสร็จทั้งหมด

avg ~15, throughput ดีกว่าแน่นอน → ถ้าไม่มีงานแทรก

Round Robin = Fairness?

↳ Is Round Robin alway Fair? → ?

↳ What is fair? → fair คืออะไร

↳ FIFO → มาก่อนได้ก่อนเสร็จก่อน

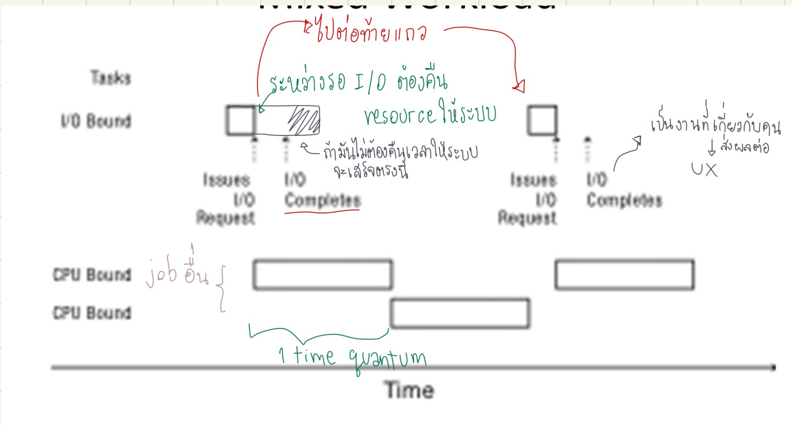
↳ Equal share of CPU → แต่ละได้ใช้ cpu เท่ากัน

↳ What if some task don't need their full share? → ถ้างานนั้นไม่ใช้ time quantum ณ รอบนั้นๆ

↳ Minimize worst case divergence?

↳ Time task would take if no one else running → ถ้างานสุดท้ายช้าต้องรอสลับไปนานมาก

↳ Time task takes under scheduling algorithm → มี overhead เกิดขึ้นไหม



งานที่ใช้เวลาไม่เต็ม 1 tq → พวก I/O → เกิด algorithm ใหม่

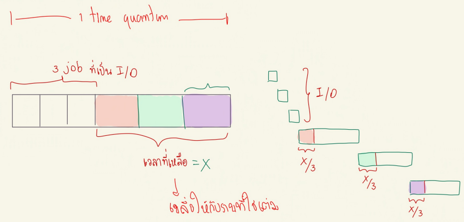
(cpu แต่ส่งต่อไป cpu ระยะยาว → รอ I/O ย้ายหลังแถว)

↓

max/min fairness

งานที่ใช้เต็ม 1 tq → งานคำนวณ

มองเรื่อง fairness เกิด max/min



Max-Min Fairness ให้มันแชร์กับงานที่ไม่เต็ม 1 tq ก่อน → แชร์ต่อได้

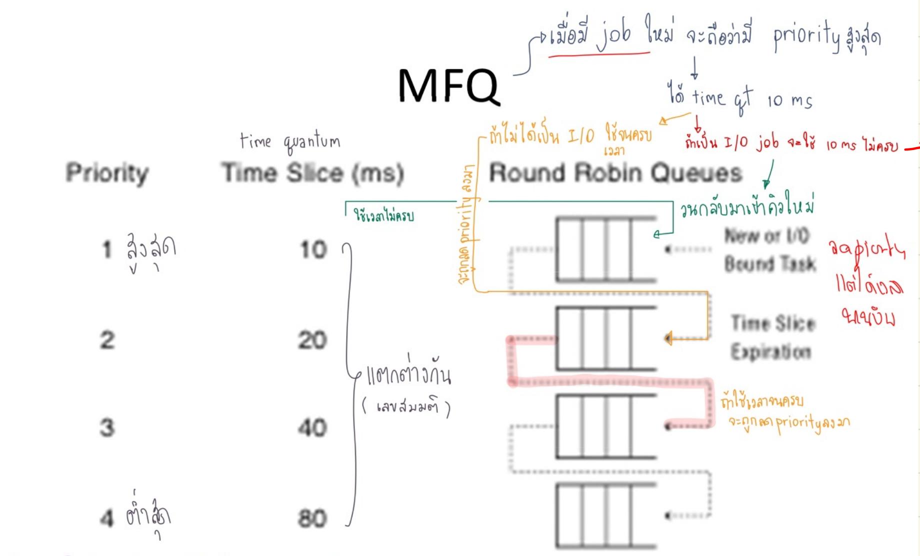
- How do we balance a mixture of repeating tasks:
  - Some I/O bound, need only little cpu
  - Some compute bound, can use as much as CPU as they assigned
- One approach: maximize the minimum allocation given to a task
  - If any task needs less than an equal shared, schedule the smallest of these first
  - Split the remaining time using max-min
  - If all remaining tasks need at least equal share, split evenly ทำจริงไม่ได้เหมือน SJF

Multi-level Feedback Queue (MFQ) เกิดตัวนี้เพราะทำไม่ได้จริง → ใกล้เคียง maxmin fairness

- Goals:
  - Responsiveness เวลาดี
  - Low overhead overhead น้อย
  - Starvation freedom ให้ทุกคนได้ใช้เวลาน้อยแต่แชร์ต่อได้
  - Some tasks are high/low priority มี priority
  - Fairness (among equal priority task) → มี fairness ในjobที่มี priority เท่ากัน
- Not perfect at any of them? → ไม่ perfect
  - Used in Linux (may in Window, Mac OS)

MFQ

- Set of Round Robin Queue
  - Each queue has a seperate priority
- High priority queue have short time slices
  - Low priority queues have long time slices
- Scheduler picks first thread in highest priority queue
- Task start in highest priority queue
  - If time slice expires, task drop one level



สามารถโกงโดยการใส่ I/O เข้าไปเพื่อให้ job ตัวเองเป็น prio 1 เสมอ

แก้ → จะมี parameter ที่คอยนับ เวลาที่ job เข้ามาใช้ cpu ถ้าถึงเวลาที่กำหนดไว้ จะลด priority ลงมา

→ แต่ถ้ามี I/O บางตัว ที่ใช้เวลามากก็ได้ จะมี parameter อื่นมาแก้อีก

ตรงนี้ทำให้เกิดการโกง → priority 1 เสมอ

Uniprocessor Summary

	FIFO ✓ (overhead ↓)	SJF ✗ (avg ✓)	RR ✓	M/M ✗	MFQ ✓ (m/m หลายระดับ)
ขนาดปนกัน	แย่	ดี	≈ SJF		
ขนาดเท่ากัน	ดี		แย่กว่า FIFO		
ขนาดสั้นๆ					
{I/O, pro}		ดี	แย่ → no starvation	I/O ✓	

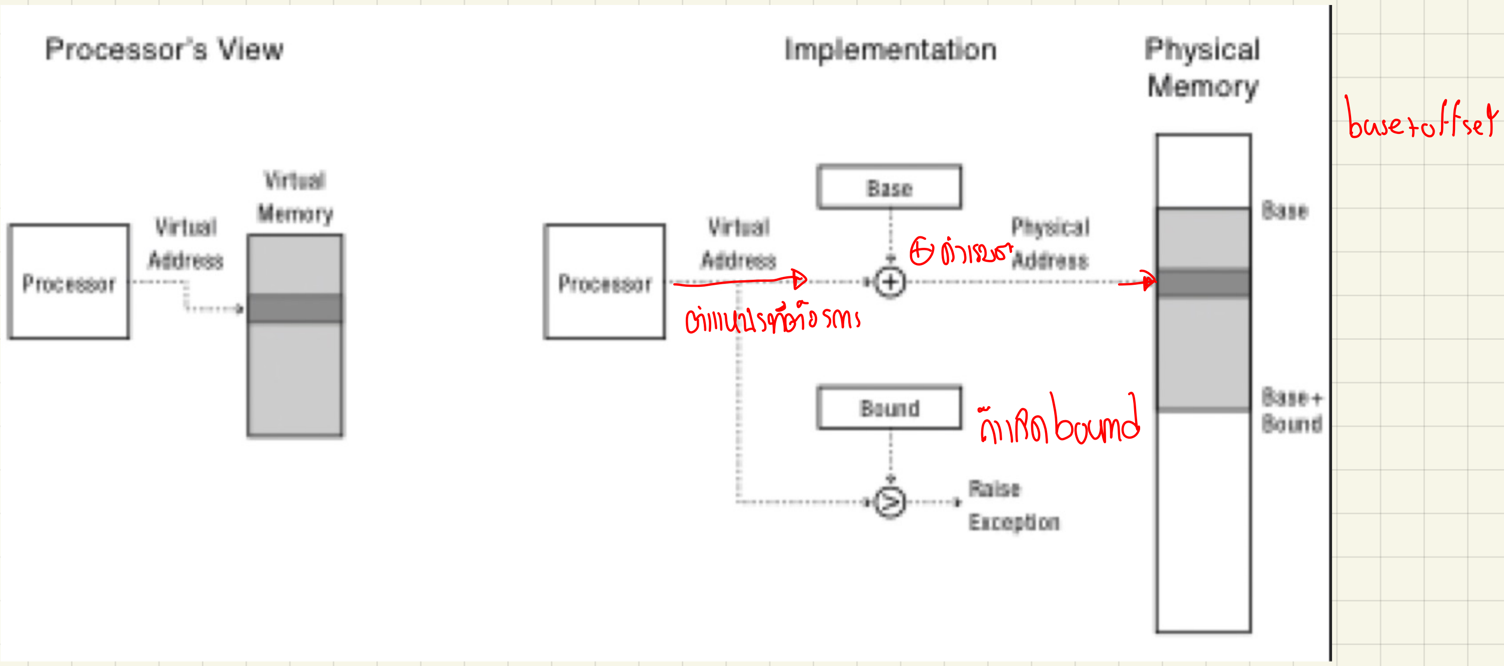
Address Translation

- ถ้าแต่ละตัวจะต้องแยก process แต่ว่ามันต้องใช้ ram อันเดียวร่วมกัน
  - process เห็น addr ตัวเลขเริ่มที่ 0 → แต่ใน ram อาจจะเริ่มที่ 112 → การ map virtual ไป physical "address translation"
- main point : map virtual address to physical address
  - Flexible address translation
    - Base and Bound
    - Segmentation
    - Paging
    - Multilevel translation
  - Effiency in Address Translation

Address Translation Goal

- การปกป้อง memory ของ process และภายใน process, เอา addr ของที่ไม่ได้แชร์ร่วมกันไม่ได้
- การ share memory ต่อกัน
- จัดการการกระจายของ memory, การจองพื้นที่แบบ dynamic
- ประสิทธิภาพที่ดี : การจอง memory ลงพื้นที่ต่างๆ, runtime lookup, compact tanslation table

Virtually Address Base and Bounds



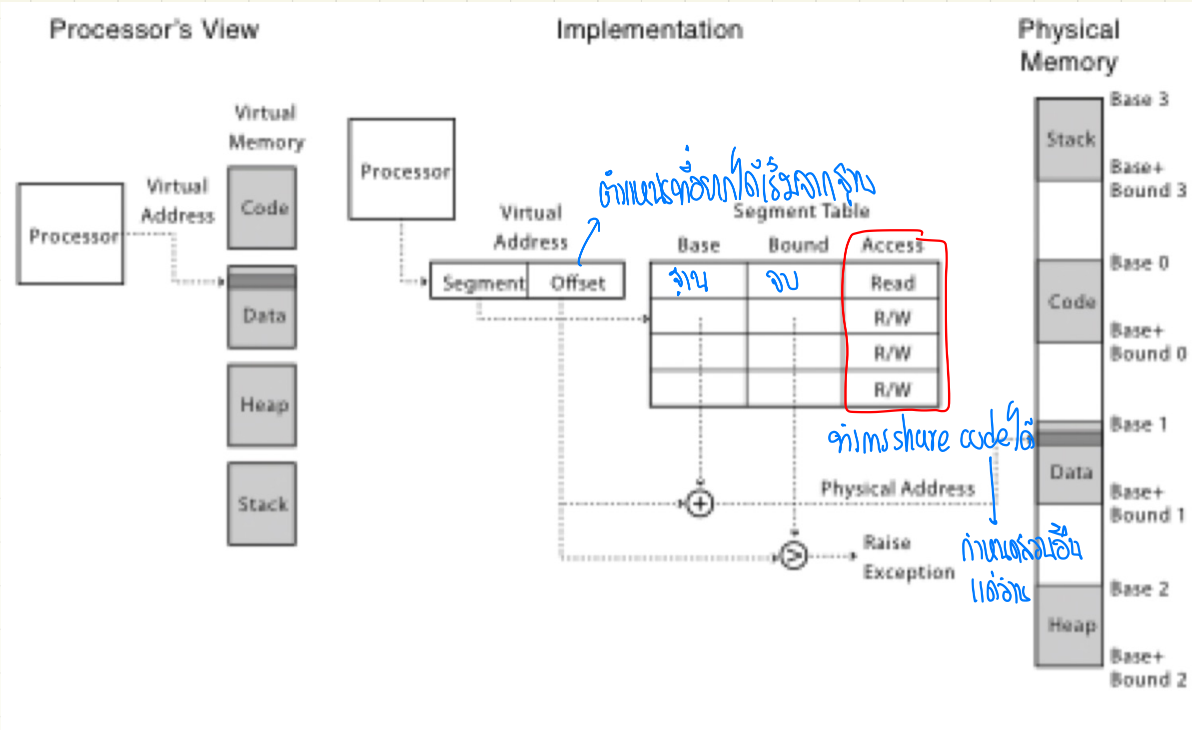
↳ Pros

↳ ง่าย, ไว, ปลอดภัย, สามารถ relocate physical memory ได้ process เหมือนเดิม

↳ Cons

↳ ไม่กันใน process, share code ไม่ได้ → base bound, stack กับ heap ขยายตามใจไม่ได้

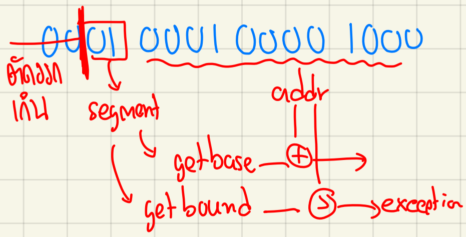
Segmentation → พยายามแก้ไขข้อเสียข้างบน

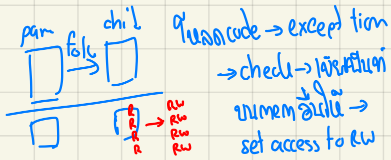


segment
↳ base + offset

↳ ไม่ต้องวางเป็นก้อนเดียวแล้ว, แต่ปัญหาการขยายของ heap stack ยังมีอยู่ → แต่ขยายโดยการลดระยะ base n+1

↳ ถ้ามี 14 bit virtual addr → 0x1108 คือที่ไหน ?





↳ Pros

↳ share code ระหว่าง process ได้, ป้องกันการเขียนทับไปยังตัวเอง, heap/stack ขยายได้อิสระ, ทำการลอง copy on write

↳ Cons

↳ ซับซ้อน, ต้องมีการจัดเรียง mem ใหม่ → ลดพื้นที่ว่าง → ตามหาช่องว่างย่อยๆ สิ้นเปลืองเยอะ

Paged Translation

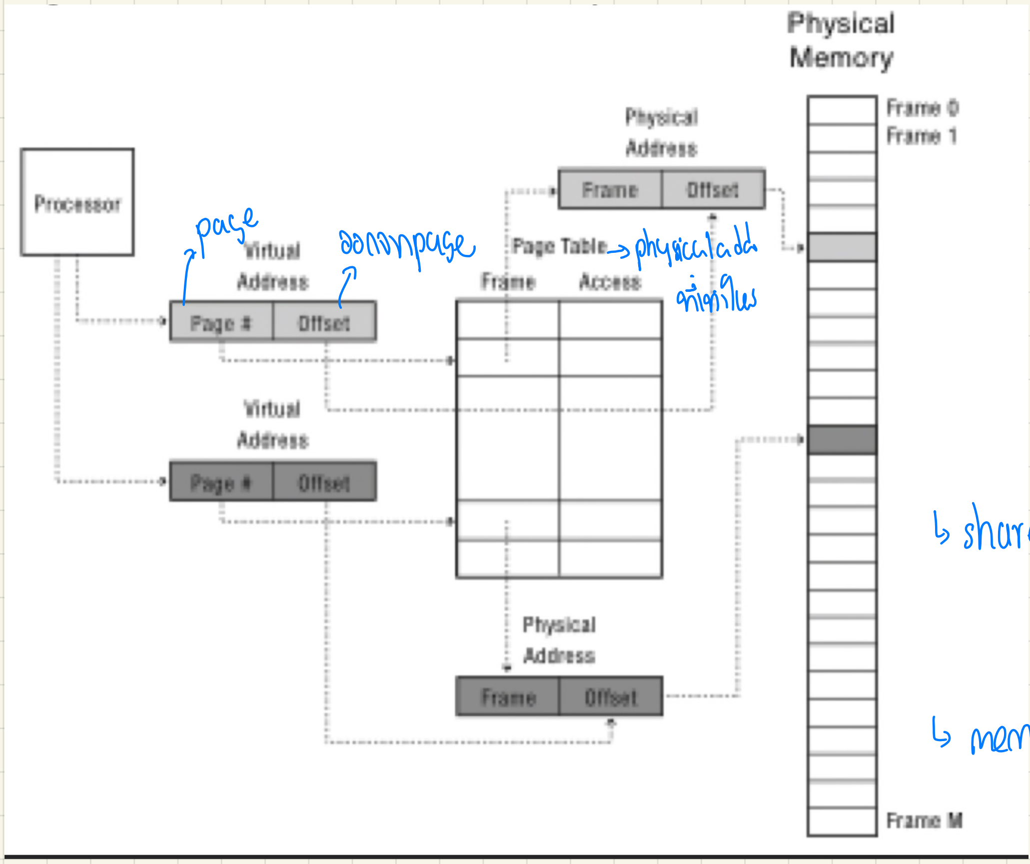
↳ กำหนดขนาดเท่ากันเป็น frame (physical) หรือ page (virtual) → ลดการค้นหาว่าพื้นที่ไหนว่าง เหลือตามขนาดของ allocation map แทน → 1 bit = 1 frame

↳ ไม่ต้องเสียเวลามาก

↳ ลดการ defragmentation

↳ แต่ละ process มี page table เป็นของตัวเอง

↳ แต่ละส่วนของ code มีได้หลายๆ 1 frame & ไม่ต้องอยู่ติดกันด้วย → ส่วนท้ายๆของ code มันก็จะเหลือ ชิ้นที่เราใช้ได้ → เปลือง



page + offset

↳ share code, no defragment

↳ memory X

Sparse Address Translation

↳ Might want many seperate dynamic segment
  ↳ heap, stack, memory map file, dynamic link lib

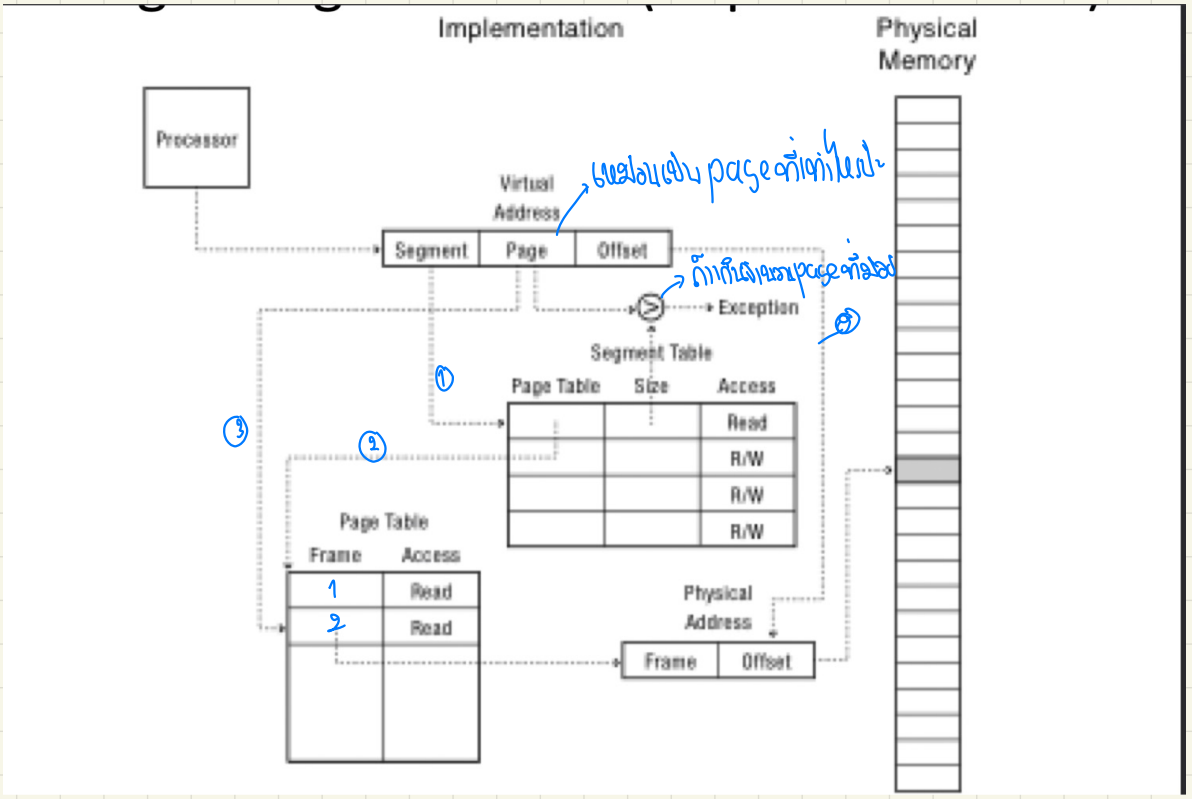
↳ ถ้าคิด virtual addr มีขนาดใหญ่มากๆ → หาไม่เจอใน cache cpu มีปัญหา → เกิดการ load → fail or suc?

Multilevel Translation

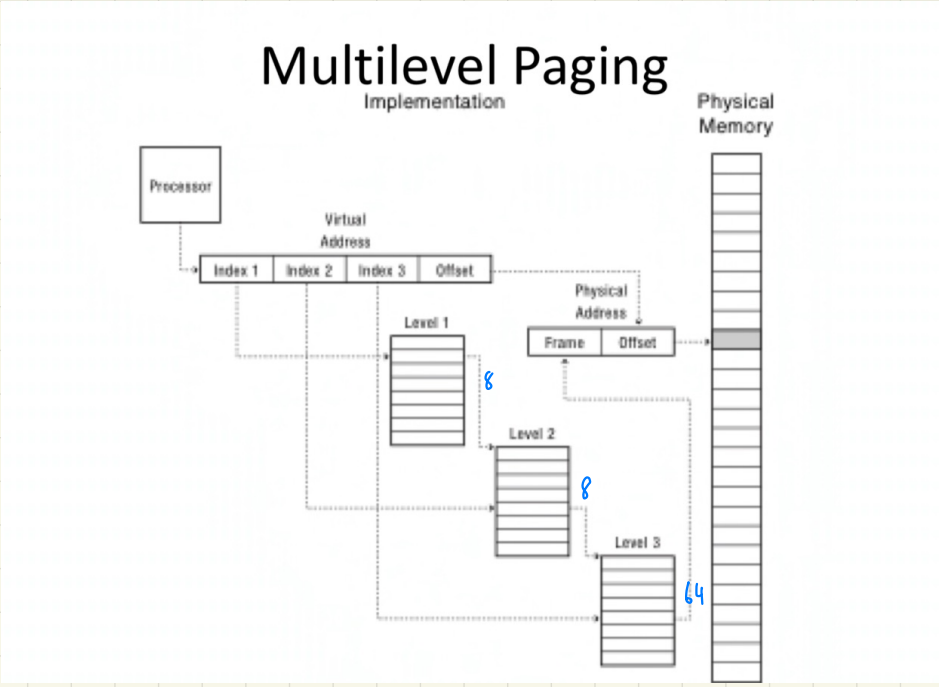
↳ tree of translation table
  ↳ page-segmentation
  ↳ Multi level pages table

↳ ว่าก็ต้องมีการ allocation map ให้ = 1 frame เลยดี

Page Segmentation → Page table + segment table



Multilevel Paging

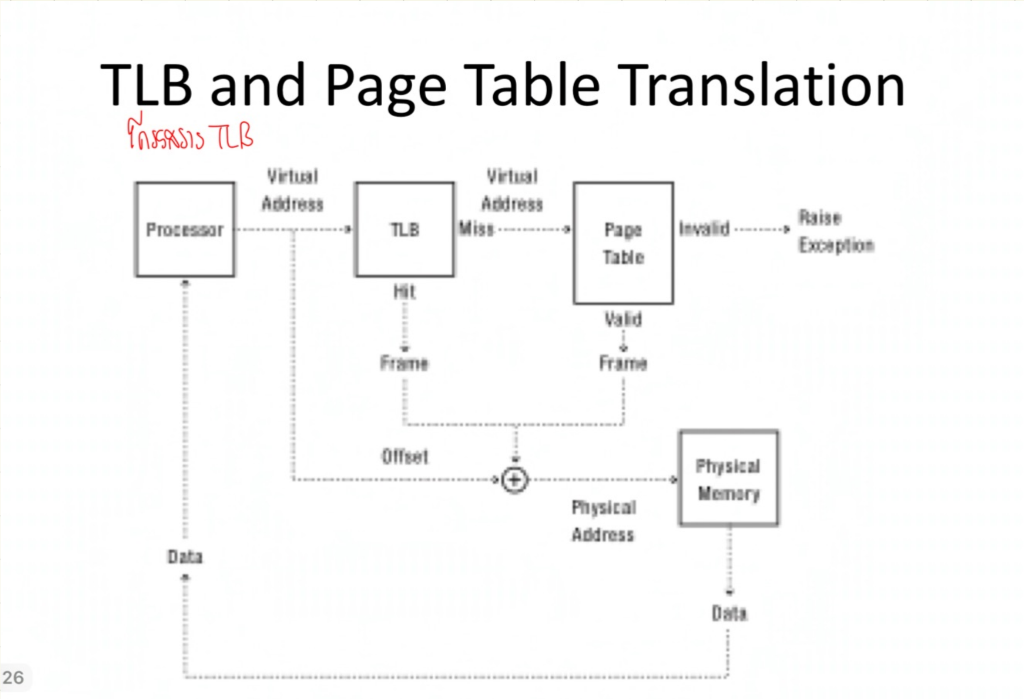


↳ Pros
↳ allocate/fill only page table entries that are in used
↳ simple memory allocation
↳ share at segment or page level

↳ Cons
↳ ต้องเก็บ miss(pointer)
↳ look up เยอะ

Effient Address Translation

↳ ทำ translation look aside buffer (TLB) → อันไหนเคยแปลงแล้วเก็บไว้ใน cache



Address Translation Uses

↳ การแยก process , การ share code, data ร่วมได้, โปรแกรมจะทำงานเลยตั้งแต่ → load page แรกก่อน / จองพื้นที่ได้ dynamic , การจัดการ cache , program debugging , zero copy , memory map file , demanded-page virtual memory, checkpoint/restart, persistence data structure, process migration , information flow control , distributed share memory