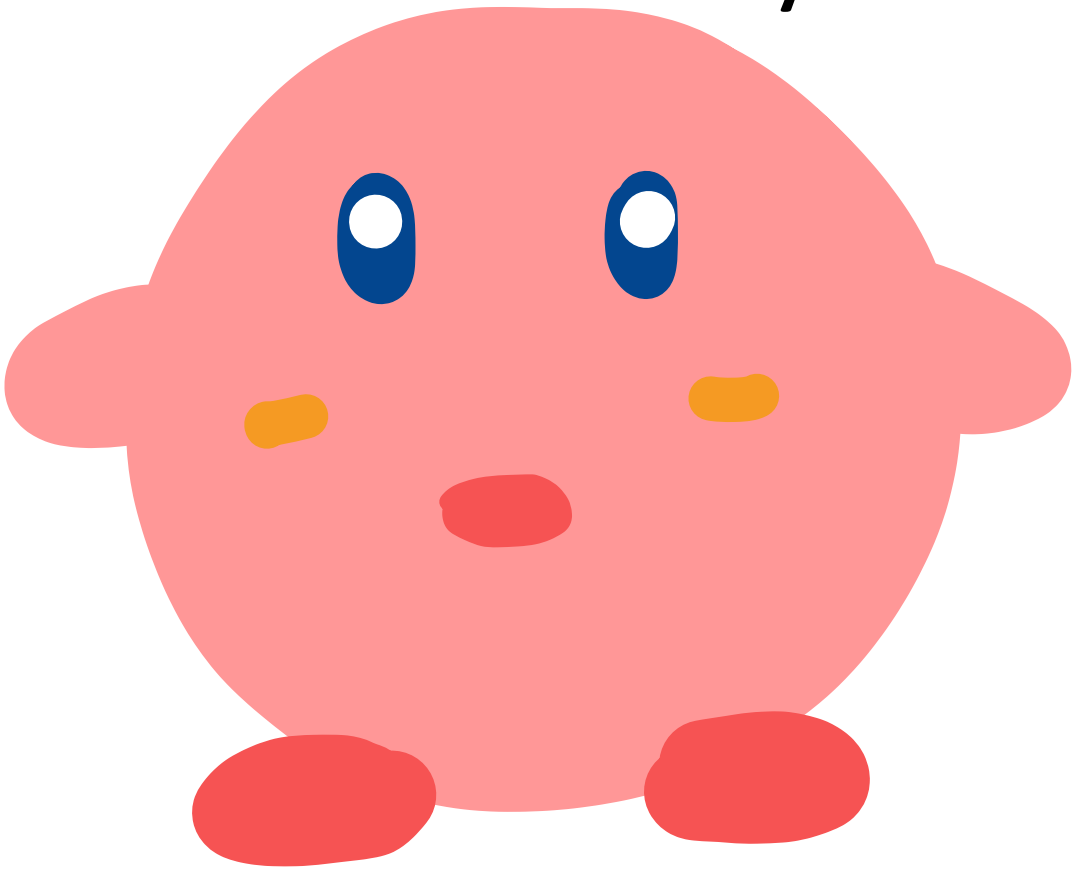
- ทำให้ ทำงาน หลายๆ อย่าง ได้ ใน เวลา เดียวกัน

Concurrency



Motivation

- Operating systems (and application programs) often need to be able to handle multiple things happening at the same time
  - Process execution, interrupts, background tasks, system maintenance

. ให้ comp. ช่วยงานเรา

- Humans are not very good at keeping track of multiple things happening simultaneously
- Threads are an abstraction to help bridge this gap

• Thread → virtual CPU

Why Concurrency?

→ share CPU → รอ Queue → ไม่ใช่ game online ในปัจจุบัน

- Servers → รองรับการเชื่อมต่อจากผู้ใช้หลายๆ คน (connection / service)
  - Multiple connections handled simultaneously
- Parallel programs → แยกงาน เป็นชิ้นส่วน → เพื่อการประมวลผล
  - To achieve better performance
- Programs with user interfaces → การทำงานของ user ไม่สะดุด ex. windows
  - To achieve user responsiveness while doing computation (UI 1 thread , IO 1 thread → แยกกัน)
- Network and disk bound programs
  - To hide network/disk latency

Definitions

- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence: familiar programming model → ทำงานทีละคำสั่ง
  - Separately schedulable: OS can run or suspend a thread at any time → OS ยุ่งกับการทำงาน execution sequence ได้ ex. interrupt, stop
- Protection is an orthogonal concept
  - Can have one or many threads per protection domain

~ คำสั่งอยู่ภายใต้ thread เดียวกัน share [variable, code, data] กันได้

- share ข้าม process ไม่ได้
- 1 process มี > 1 thread ได้

Kernel < Mono → process ขนาดใหญ่ 1 ก้อน → ข้างในมีหลาย thread ทำงาน
Micro → มีหลาย process

Threads in the Kernel and at User-Level

Mono Kernel

- Multi-threaded kernel
  - multiple threads, sharing kernel data structures, capable of using privileged instructions

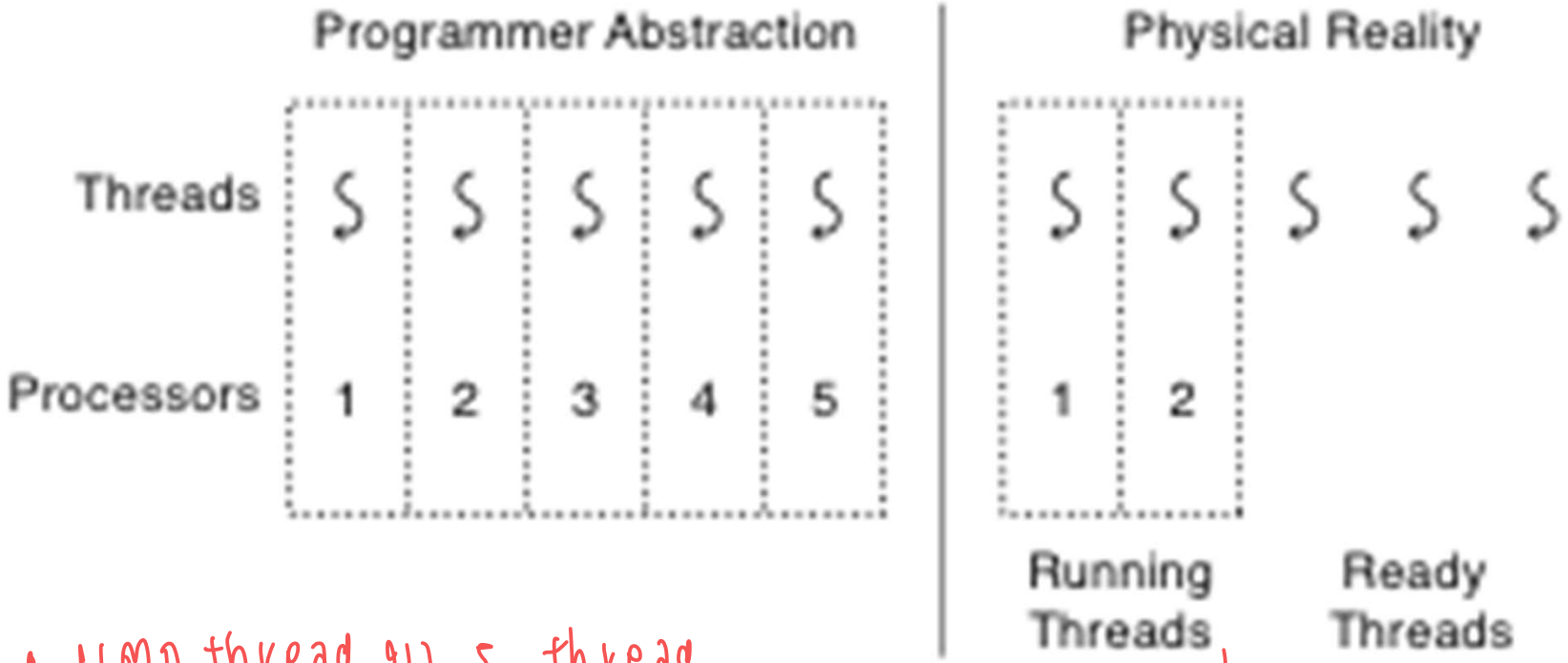
Micro Kernel

- Multiprocess kernel → แตกออกเป็น process ย่อยๆ → 1 thread / 1 process (รับผิดชอบ)
  - Multiple single-threaded processes
  - System calls access shared kernel data structures

- Multiple multi-threaded user processes
  - Each with multiple threads, sharing same data structures, isolated from other user processes
  - แต่ละ process ใน user space มี > 1 thread ได้
  - แชร์ใน process เดียวกันได้

Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule



- แตก thread มา 5 thread เสมือนมี CPU 5 ตัว
- จริงๆ มี 2 core สิ่งต้องผลัดกันเข้ามาใช้

- ต้อง share เวลาใช้ CPU
∴ thread = time slot ที่ได้เข้าใช้งาน CPU
คือ จนกว่า schedule มาขอ มาใช้ทำงานต่อ

Programmer vs. Processor View

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;	.....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	.....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	.	.....	Thread is resumed.
		y = y + x;	.....
		z = x + 5y;	z = x + 5y;

interrupt, สับ switch

program ไม่รู้ว่าเกิดการ switch ในเรื่อง security

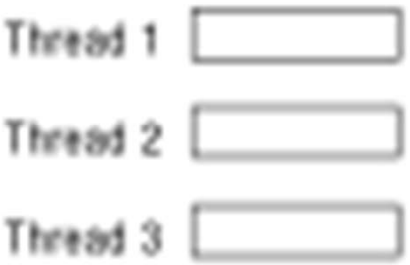
Possible Executions

รอ 1 เสร็จ จึง ทำ ต่อ

One Execution

* อุดมคติ → CPU มีจำกัด
run 3 อันพร้อมกัน

Another Execution



* reality

Another Execution

สลับ กันทำ , ถ้าสั้นก็จบใน รอบเดียวได้

↳ ex. print('hi')
จะไม่สับ switch

ทำงานผ่าน sys call

Thread Operations

- thread_create(thread, func, args) → สร้าง thread
  - Create a new thread to run func(args)
- thread_yield() → ใช้เวลา, คืนเวลาที่เหลือให้ CPU ไปจัดสรรให้ตัวอื่น
  - Relinquish processor voluntarily
- thread_join(thread) → ผู้ 2 thread จะ sync การทำงาน
  - In parent, wait for forked thread to exit, then return

  ex. thread 1 จะใช้ผล thread 2 ∴ 1 ต้องรอ 2
- thread_exit → จบการทำงาน
  - 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?
}
```

assign ค่า ให้ แต่ละ thread

สร้าง 10 threads

↳ ส่ง code function go ส่งค่า i ไปตามลำดับ

หยุดรอ thread #i ให้ทำงานให้จบก่อน แล้วถึง print

ค่าที่จะส่งกลับ = return

thread ทั้งหมด 11 thread - 1 thread run main

↳ สร้างอีก 10 thread

• ในการสร้าง thread จะไม่สามารถกำหนดได้ thread ไหน run ก่อนได้ แม้ว่าจะถูกสร้างก่อน (ทำงานไม่เรียงลำดับ) ขึ้นอยู่กับ scheduler

threadHello: Example Output

- Why must "thread returned" print in order?
- What is maximum # of threads running when thread 5 prints hello?
- Minimum?

จาก go

จาก for i (หลัง thread_join)

```
bash-3.2$ ./threadHello
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 8
Hello from thread 7
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.
```

i

100 + i

→ กรณี thread 5 เข้าทำงาน cpu thread แรก ไม่ execute

maximum thread = 11

└ 2 thread (main, thread 5)

• thread_join จะหยุดรอ thread ตามลำดับตาม code

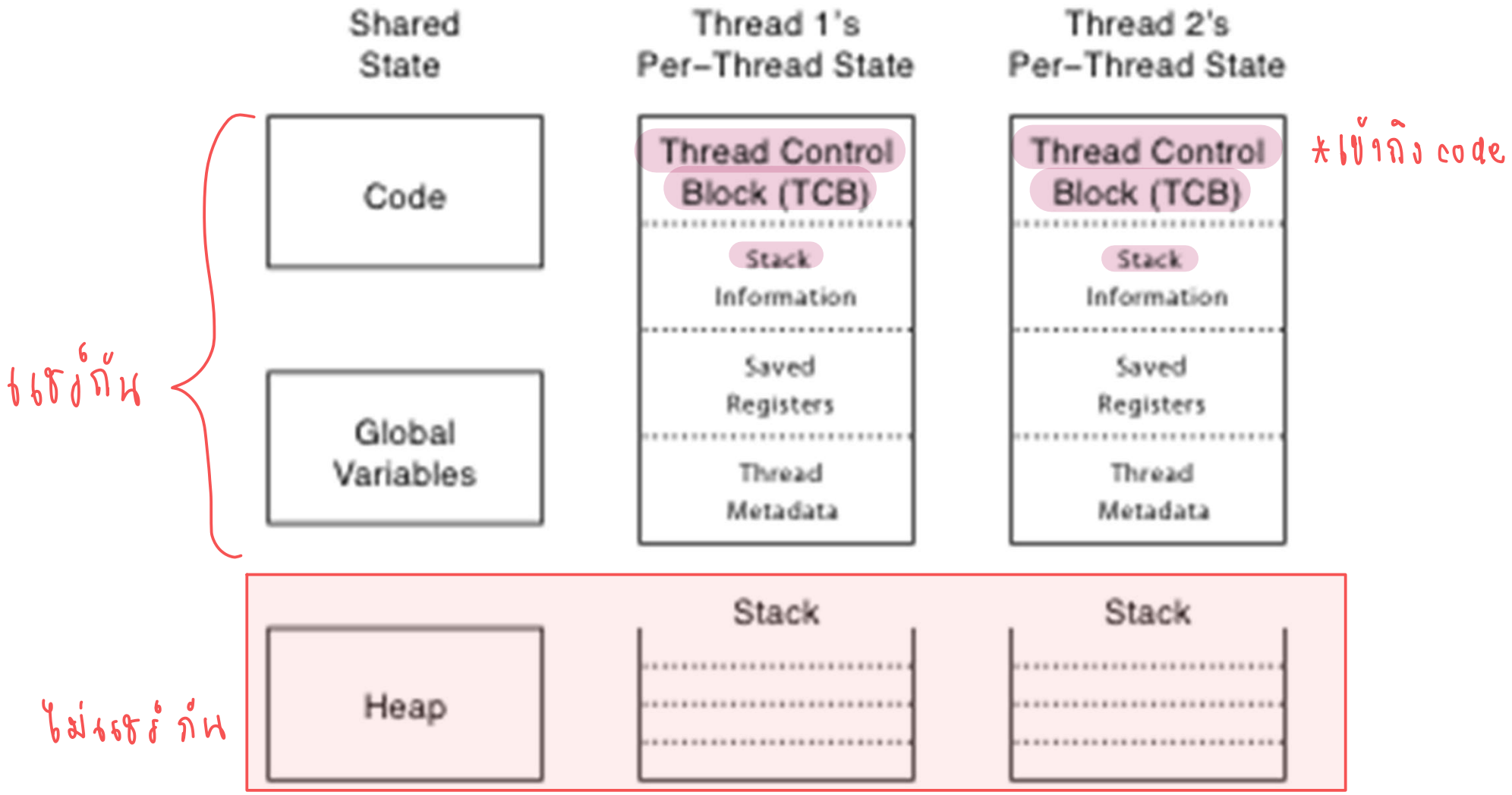
∴ Thread i returned 100 + i

จะเรียงลำดับตามเดิม

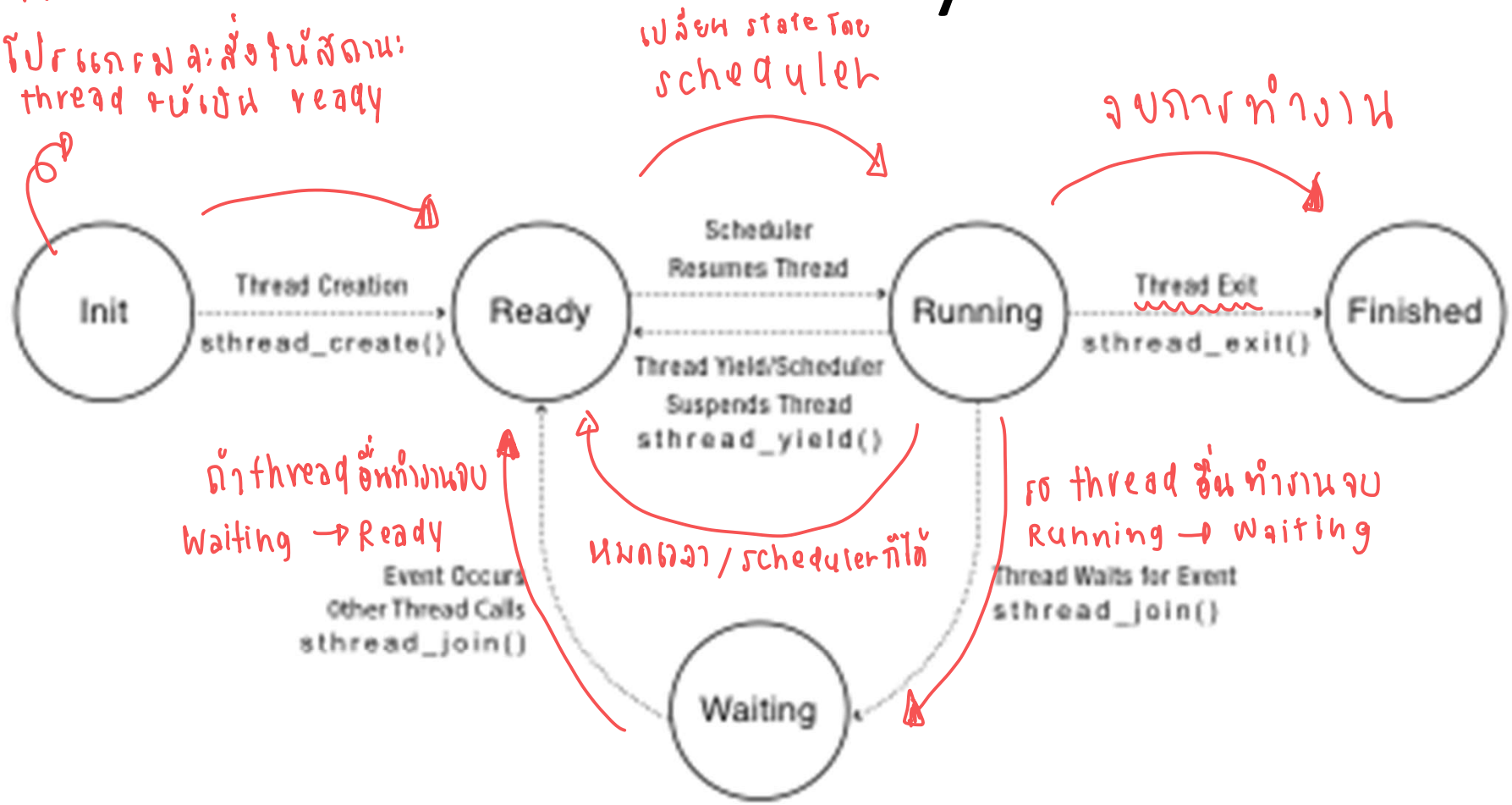
Fork/Join Concurrency

- Threads can create children, and wait for their completion
- Data only shared before fork/after join
- Examples:
  - Web server: fork a new thread for every new connection
    - As long as the threads are completely independent
  - Merge sort
  - Parallel memory copy

Thread Data Structures



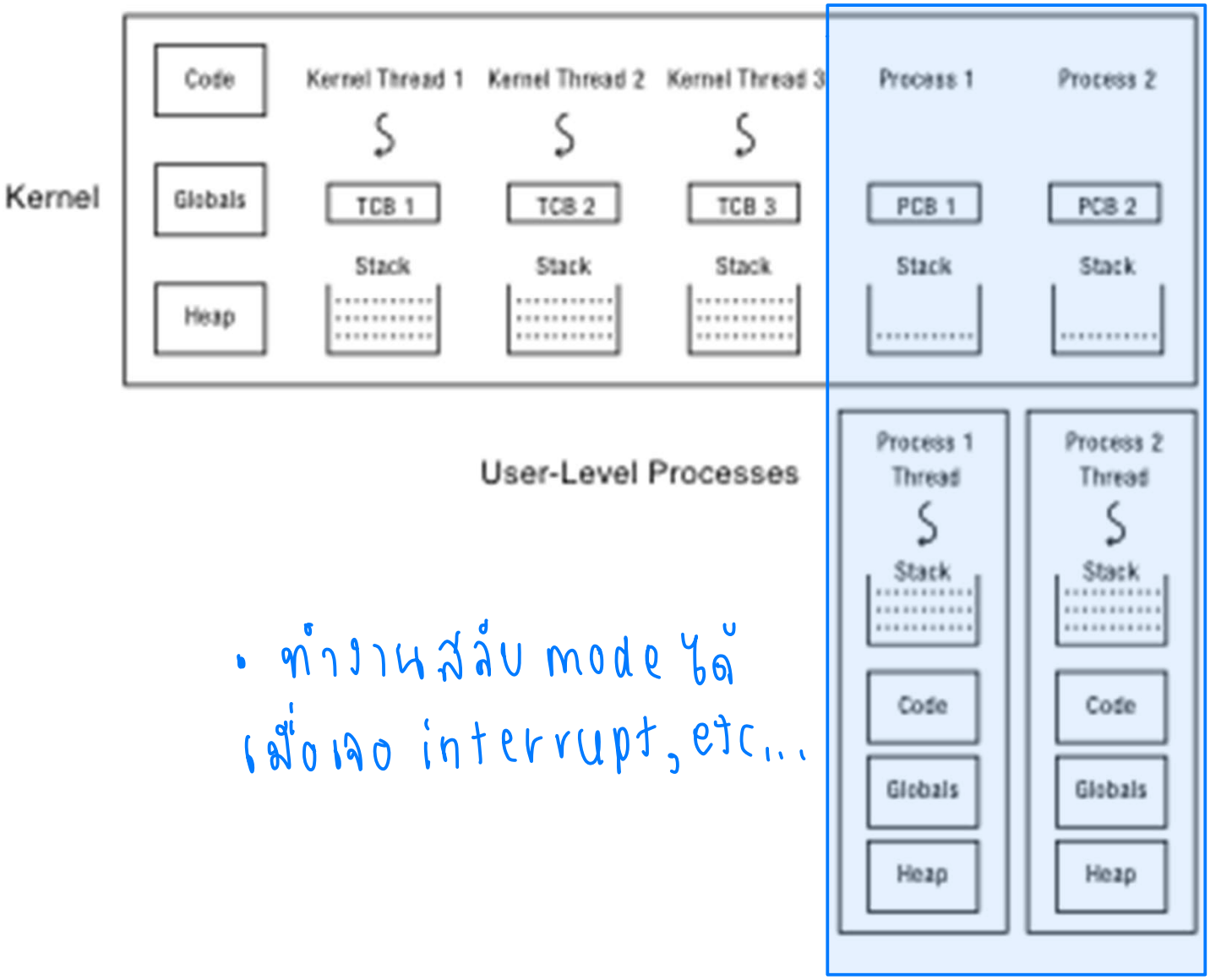
Thread Lifecycle



Implementing Threads: Roadmap

- Kernel threads
  - 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)
  - Kernel thread operations available via syscall
- User-level threads
  - Thread operations without system calls

Multithreaded OS Kernel



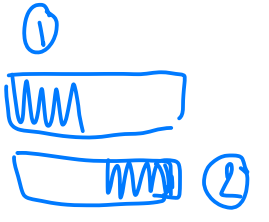
• ทำงานสลับ mode ได้
เมื่อเจอ interrupt, etc...

สร้าง thread

Implementing threads

- Thread_fork(func, args)
  - Allocate thread control block สร้าง TCB
  - Allocate stack จองพท. stack
  - Build stack frame for base of stack (stub) สร้าง stack frame
  - Put func, args on stack
  - Put thread on ready list เปลี่ยน สถานะ thread เป็น ready
  - Will run sometime later (maybe right away!) รอเรียก run
- stub(func, args): OS/161 mips_threadstart
  - Call (*func)(args)
  - If return, call thread_exit()

• switch → เปลี่ยน จาก thread หนึ่ง ไป thread ๆ
ทำงาน thread ๆ ครั้งหนึ่ง thread ๆ ครั้งหนึ่ง



} เกิด stub

Thread Stack

- เอา address ของ function ใส่ลง ใน stack ด้วย
- What if a thread puts too many procedures on its stack?
  - What happens in Java?
  - What happens in the Linux kernel?
  - What happens in OS/161?
  - What *should* happen?

* ถ้า ใส่มากๆ → stack เต็ม = error
  ↳ ใส่ function ลงใน thread

สลับเข้าใช้ CPU ของ thread

Thread Context Switch

- Voluntary ต้องหยุดรอ (ไม่รู้นานแค่ไหน)
  - Thread_yield
  - Thread_join (if child is not done yet)
- Involuntary → ไม่เต็มใจคืนเวลา
  - Interrupt or exception
  - Some other thread is higher priority

- timer interrupt → คล้ายนาฬิกาปลุก
  ↳ เรียกใช้งาน context_switch ได้

Voluntary thread context switch

- Save registers on old stack → save ค่า reg. ของ thread ปัจจุบัน ลง stack
- Switch to new stack, new thread → switch
- Restore registers from new stack → load ค่า reg. ของ thread ที่ถูกเลือก
- Return
- Exactly the same with kernel threads or user threads

OS/161 switchframe_switch

```
/* 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

```
# Save caller's register state
#  NOTE: %eax, etc. are ephemeral
pushl %ebx
pushl %ebp
pushl %esi
pushl %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 changes 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
```

- switch → โหลด thread ใหม่ลง reg.

A Subtlety

- Thread_create puts new thread on ready list
- When it first runs, some thread calls switchframe
  - Saves 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	Thread 2's instructions	Processor's instructions
"return" from thread_switch into stub		"return" from thread_switch into stub
call go		call go
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch
save thread 1 state to TCB		save thread 1 state to TCB
load thread 2 state		load thread 2 state
	"return" from thread_switch into stub	"return" from thread_switch into stub
	call go	call go
	call thread_yield	call thread_yield
	choose another thread	choose another thread
	call thread_switch	call thread_switch
	save thread 2 state to TCB	save thread 2 state to TCB
	load thread 1 state	load thread 1 state
return from thread_switch		return from thread_switch
return from thread_yield		return from thread_yield
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch

คืนเวลาให้ระบบ

เรียก thread_switch

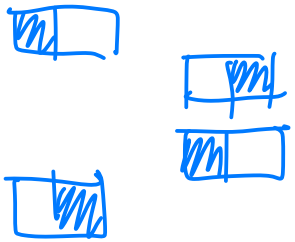
save state current thread

switch ไป thread ใหม่ load stack pointer

stub จะเกิดขึ้นเมื่อไม่เคย run(call) มาก่อน ∴ thread 1 จึงไม่ต้อง call stub

stub จะชี้ไปเริ่มที่ call go

ครึ่งที่เหลือของ call switch จะทำงานใน thread 2 มันจะผิดปกติเพราะมันแค่นาง ไม่มี call จึงเกิด stub_function ปลอม dummy ของ thread_switch เวลา thread 2 return จะหลัง stub แทน



04_5

Involuntary Thread/Process Switch

- Timer or I/O interrupt
  - Tells OS some other thread should run
- Simple version (OS/161)
  - End of interrupt handler 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

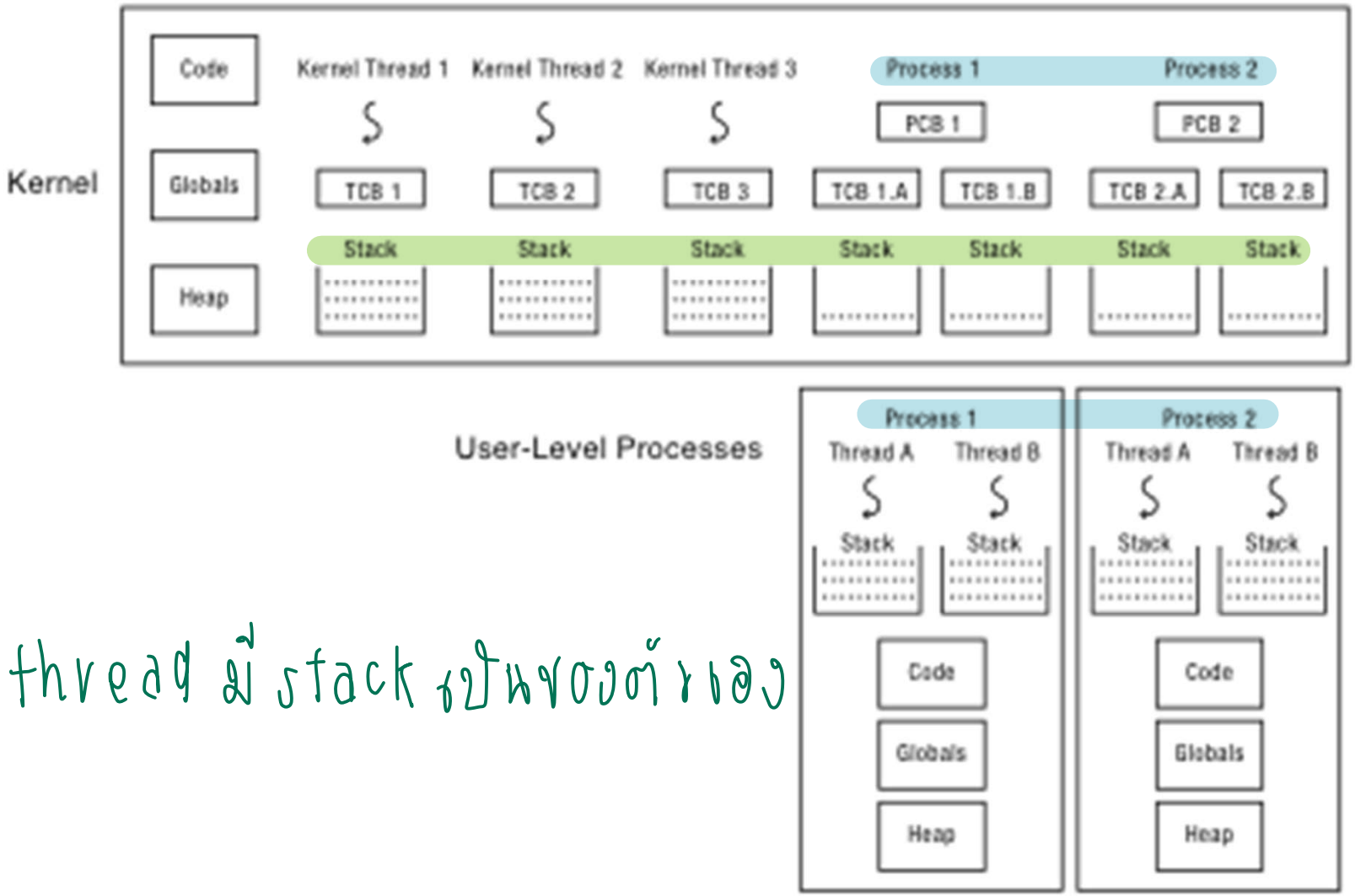
- What happens on a timer (or other) interrupt?
  - Interrupt handler saves state of interrupted thread
  - Decides to run a new thread
  - Throw away current state of interrupt handler!
  - Instead, set saved stack pointer to trapframe
  - Restore state of new thread
  - On resume, pops trapframe to restore interrupted thread

Multithreaded User Processes (Take 1)

- User thread = kernel thread (Linux, MacOS)
  - System calls for thread fork, join, exit (and lock, unlock,…)
  - Kernel does context switch
  - Simple, but a lot of transitions between user and kernel mode

time slot → เกิด interrupt ∴ ต้อง switch

Multithreaded User Processes (Take 1)



- แต่ละ thread มี stack เป็นของตัวเอง

อยากให้ user จัดการ thread

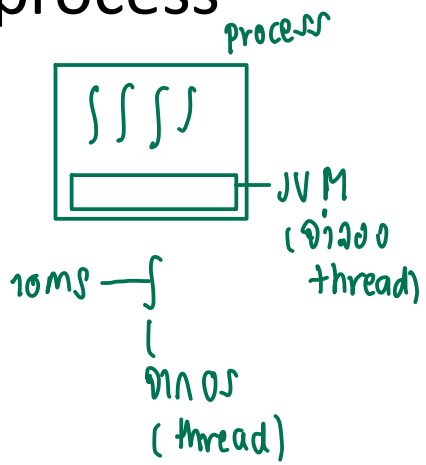
Multithreaded User Processes (Take 2)

↳ ลองทำ 1 process หลาย thread

Java → เขียนแล้วทำงานได้บนทุก platform → JVM จำลองสภาพให้

แต่มี OS → 1 thread / 1 process
Java อาจจะ: run ไม่ได้
↳ JVM จะ: ทำ thread เทียมขึ้นให้
ex. 4 thread 10 sec.

- Green threads (early Java)
  - User-level library, within a single-threaded process
  - 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



JVM จัดการต่างๆ ทั้ง schedule ต่างๆ ที่ใน Java สามารถ run ในโปรแกรมอื่น

Multithreaded User Processes (Take 3)

request, remove
ขอ/ลบ thread เองได้, support กรณี syscall แล้วโดน block → Kernel จะ upcall ขึ้นมา
แบ่ง ก. ทำงานไป 2 ระดับ → process เดียวกัน ให้ทำ context switch ใน scheduler ของ user

- Scheduler activations (Windows 8)
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - Thread library decides what thread to run next
- Upcall whenever kernel needs a user-level scheduling decision
  - Process assigned a new processor
  - Processor removed from process
  - System call blocks in kernel

I LOVE YOU ♡

Question

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

- next week → log notebook + ลง C#