


Date _____
Subject _____

Iyes 3D




OPERATING SYSTEM ☺

50% : Exams : Midterm & Final
40% : Lab & Assignment
10% : Activity

• WHAT'S AN OPERATING SYSTEM?

เป็น software ที่ใช้ในการจัดการ resource ของคอมพิวเตอร์ ; รวมถึงจัดการ resource ของ hardware ต่างๆ ex. CPU, mouse, keyboard

ROLES

ex. จัดการ file

- แยกแต่ละ file ว่า user คนใดเข้าถึงได้บ้าง
1. Referee - กรรมการ คอยทำการตัดสินว่าใครจะได้อะไร resource ใด, เมื่อไหร่
 - ต้องมีความ **isolation** เสร็จด้วยตัวของตัวเองแล้วค่อย
 - communication เวลาฟังไม่ดังอย่าวิ่งไปด้อมระหว่าง program หลายๆ program บนเครื่อง
 2. illusionist - สร้าง illusion ซ้ำให้คิดว่า program ที่กำลัง run อยู่สามารถใช้งาน
ได้เต็มกำลัง ใช้ได้ทั้งเครื่อง
 - สมมติว่ามี **infinite** number of processor
 - infinite** amount of memory
 - reliable** storage
 - reliable** transport network
 3. Glue - เชื่อม library เข้าด้วยกัน (ระหว่าง interface, Libraries)

CHALLENGES

- 1. Reliability : ต้องมีความเสถียร
- 2. Availability : ต้องพร้อมใช้งานตลอดเวลา
- 3. Security : ป้องกันพวกไวรัส
- 4. Privacy : แบ่งแยก ไฟล์ของแต่ละคน ex. ไฟล์ของ application ใก้มีแค่ app นั้นๆ
ที่สามารถเข้าถึงได้
- 5. Portability : สามารถเคลื่อนย้ายโปรแกรมได้
 - For programs ต้องพิจารณา API, Abstraction virtual Machine
 - For OS "**hardware abstraction layer**"

☺ เป็นการสร้าง abstraction ที่ใช้ในการอ้างอิงระหว่างฝั่ง hardware
กับฝั่งที่เป็นคนเขียน operating system
- 6. Performance
 - Latency : ความเร็วในส่วนของ response
 - Throughput : จำนวนงานที่ทำได้ per time unit
 - Overhead : ให้ overhead น้อย
 - Fairness : ความ fair ของการใช้งานของแต่ละ program
 - Predictability :

* MTF (Mean time to Failure)

- ค่าเฉลี่ยของเวลาที่ Fail ของแต่ละ disk
(เกี่ยวกับพวกทำ disk, RAID)

EARLY OPERATING SYSTEM

- รัน 1 application ต่อ 1 หน่วยเวลา
 - Batch system
- computer แบบ

TIME-SHARING OPERATING SYSTEM

- เริ่มให้หลาย user ทำงานพร้อมกันในเวลาหนึ่งได้ (multiprocessing) จากเริ่มมัลติ

TODAY'S COMPUTER IS CHEAP

TOMORROW

- data center ในยุคนี้น่าจะรองรับให้ได้

THE KERNEL ABSTRACTION

→ การเขียน kernel คือต้องเขียน code ให้ง่ายสุด, เข้าใจง่ายสุด

BOOTING

- ① BIOS ทำการ run BIOS ที่เก็บอยู่ใน ROM บน mainboard
- ② ทำการ run bootloader (BIOS copies bootloader) เป็นโปรแกรมแรกที่ run หลังจาก BIOS
- ③ Bootloader จะ recognise ในแต่ละ partition ได้ ว่า program ในที่อยู่ใน partition ไหน (Bootloader copies OS kernel)
- ④ OS kernel copies login application → ขึ้นอยู่กับแต่ละ OS

DEVICE INTERRUPT

- OS kernel ต้องมี communication กับพวก device ต่างๆ
- Device operate asynchronous จาก CPU แบ่งออกเป็น
 1. polling คือ kernel จะทำการรอจนกว่าการติดต่อของ I/O เสร็จสิ้น
 2. Interrupt คือ kernel สามารถไปทำอย่างอื่นได้ พอเสร็จแล้วจะส่งสัญญาณมาบอก
- Device access memory
 1. programmed I/O (CPU เป็นตัวเขียนอ่าน memory โดยตรง)
 2. DMA (เขียนลง memory โดย device เอง)
 3. Buffer descriptor (list ของ address ของ dma)
 4. Queue Buffer descriptor (เขียนต่อกับ buffer ไรต์ไวด์หลายๆที่)

Date _____

Subject _____



PROTECTION

ต้องมีเกราะป้องกัน code บางตัวจะสามารถรันคำสั่งไหนใน CPU ได้บ้าง เพื่อป้องกันไม่ให้มีบั๊กพร้อมกัน
* OS ทำการ manage process ของแต่ละ program ว่าต้องใช้อะไรเท่าไร, ตรงไหน

★ **MAIN POINTS** * จมอเนนนาแต่ละ process เป็น abstract virtual machine ถือทุกอย่าง unlimited

1. Process concept

สนใจในกระบวนการทำงานของแต่ละ process ย่อยๆ manage แต่ละอย่างที่อยู่ภายใน process โดย OS จะทำการรันแต่ละ process ด้วย limited privileges

2. Dual-mode operation แบ่งออกเป็น 2 แบบ

- kernel-mode : ทำการรันแบบ complete privileges คือสามารถ access ได้ทุกอย่าง
- user-mode : โปรแกรมส่วนใหญ่ run แบบ user-mode คือเข้าถึงได้บางส่วน

3. safe control transfer

① PROCESS ABSTRACTION

→ process คือช่วงๆหนึ่งของการทำงานของ program

เมื่อถาม process = thread

คอยสลับเข้า
run Thread
ในหมอก

• Thread : a sequence of instruction (line of an execution) within a process เป็นกรณี instruction ใดๆ ที่ run เพื่อทำงานหนึ่งงาน

- Address space : หน่วยความจำ บ่งบอกว่าส่วนไหน access ได้ ส่วนไหนไม่ได้, ความสามารถในการเข้าถึงอื่นๆ

THOUGHT EXPERIMENT → ถ้าต้องการ execution โดย limited privilege

เวลาทำการ implement execution with limited privilege ใช้ simulator แทน แต่ปัญหาคือมันช้า แต่มันทำงานได้แค่แค่ operation ส่วนใหญ่จะใช้น web browser จาก java script

② * **HARDWARE SUPPORT "DUAL-MODE OPERATION"**

- kernel mode
→ สามารถรันได้ทุกอย่าง แก้ไขได้ทุกอย่าง
- user mode ← kernel เป็นตัวกำหนด privileges ต่างๆให้
→ Limited privileges , มีการจำกัดบางส่วนในการทำงาน ถ้ามีการรันคำสั่งที่นอกเหนือสิ่งที่ทำได้ใน user mode CPU จะ throw exception
← x86 จาก EFLAGS
MIPS จาก status register
- ต้องประกอบด้วย privileged instruction (ตัว HW ที่สามารถการโดดไปทำงานที่ฝั่ง kernel mode ได้) mode user ห้าม!
- มีการ limit memory access
ป้องกัน user เข้าไปยุ่งกับ code ส่วนที่เป็น kernel mode

↪ ใช้ timer สร้าง interrupt เพื่อ swap mode กลับมาจาก user mode

- มีการกำหนด timer เพื่อหยุดการทำงานของ program บางอย่าง ไม่ให้สิ้นชีพ CPU ว่างตลอดเวลา (100ms, 10ms)
- safe way สำหรับกร switch user

* ถ้า user พยายามเรียกใช้ส่วนที่เป็น **privileged** จ.เกิดกร throw exception

PRIVILEGED INSTRUCTION

"Hello world" ที่โม kernel ต้องเป็นคนทำการ copy string ลง buffer เอง ไม่ยอมให้ app เขียนลง buffer โดยตรง

ANS เพราะเราควบคุมไม่ได้ว่า app ไหนจะไปเขียนลง buffer ส่วนไหน ป้องกันกรเขียนทับบน screen's buffer memory

→ ใช้เทคนิคที่เรียกว่า Base & Bound ถ้าออกนอกที่แจ้ง Exception

allowed address space for user → range ของ address space ห้ามต่ำกว่า Base ห้ามสูงกว่า Bound

ปัญหาคือ 1. ไม่รู้ว่าโปรแกรมไหนใช้ memory ส่วนไหน

2. Heap & Stack ที่ใช้ใน memory มีนโมกสซ้อนทับกัน

→ LEADS TO VIRTUAL ADDRESS

process ทุก process จ.มองเน้นเป็น virtual address อยากใช้ส่วนไหนก็ใช้ไป โดยที่ OS คือเป็นคน map virtual address เข้ากับ memory

ปล่อยให้ process ทำงาน

OS กำลัง ~~ทำงาน~~ อยู่แล้วต้องการที่จะดึง process กลับมา ~~ที่~~ kernel ต้อง

ANS Interrupt, Polling, timer event

เป็นช่วงเวลาที่ต้องสร้าง interrupt ขึ้นมาเรื่อยๆ เพื่อให้กลับจาก user → kernel

→ ต้องมีความถี่ขนาดไหน? (ไม่เกิน 150ms)

คนที่จ.เป็นคน set ควรจะเป็นโปรแกรมที่ user process? หรือ kernel? อย่างเดียวทำไม่ได้

→ ถ้ามี interrupt เกิดขึ้นก็ต้องมี feature ที่ทำการรอเพื่อทำอไรที่สำคัญๆ ก่อน แล้วค่อยไปรับมือกับ interrupt ตัวอื่นได้ ex. กร switch mode (deferred interrupt)

* ถ้ามีกร deferred interrupt เยอะๆ จ.ทำให้ไม่สามารถทำงานทัน ต้องยอม lost

interrupt ที่เกิดขึ้นจ.มีกรเชค priority ด้วย ถ้า priority ต่ำกว่า process ที่มันกำลังทำอยู่ก็สามารถส่งไปได้ (deferred)

Date

System calls - เป็นการเข้าถึง kernel โดยตรงผ่าน program

Subject

ของ user เช่น ต้องการเข้าถึง hard disk drive เป็นต้น
 เพื่อทำงานบางอย่างที่ kernel ทำได้เท่านั้น

Lactasoy[®]

HARDWARE TIMER

- ▶ คิว controls ให้กับตัว kernel หลังจาก kernel เปิดโอกาสให้ user process
- ▶ Interrupt frequency ถูก set โดย kernel
- ▶ มี temporarily deferred เพื่อชลอ interrupt โดยรอให้ทำงานสำคัญเสร็จก่อน

① MODE SWITCH เปลี่ยนจาก user mode → kernel mode

1. Interrupts
 - เกิดขึ้นโดย timer หรือ I/O
2. Exceptions overflow, null pointer exception, segmentation fault, out of memory
 - เกิดขึ้นโดยการทำงานบางอย่างใน program ที่ไม่ unexpected
 - ex. เขียน program ในส่วนที่ถูกบังคับไว้ว่าไม่ให้นำเข้าถึง, divide by zero, overflow
3. system calls (จริงเรียกให้ kernel ทำงาน)
 - not printf x ex. การเข้าถึง disk (hard disk drive), create new process

② การวนการ เปลี่ยนจาก kernel mode → user mode

1. new process, new thread "start"
 - เริ่มทำงานโดยโปรแกรมใหม่ jump into first instruction in program
2. return from interrupt
3. context switch (run process A ต้องการไป run process B)
4. system calls (cupcall) kernel ไปเรียกใช้ host ที่อยู่บน user space
 - สร้างโดยการใช้ unix signal โดยตรงงานมีชื่อ system call
 - ex. การ notification user program

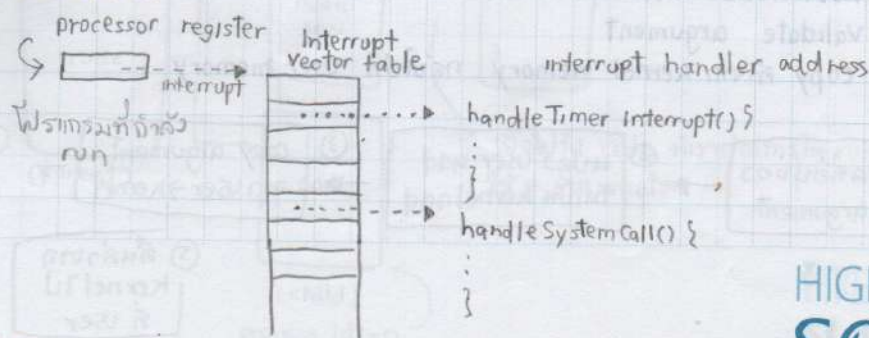
TAKE INTERRUPT SAFELY

1. Interrupt vector (ของ kernel)
 - มี array ของ pointer ที่ชี้ไปที่ส่วนของ interrupt handler ว่าต้องไป run ส่วนไหน
2. Atomic transfer
 - ควรทำงานเป็น single instruction
 - ควรทำงานเป็นแบบ atomic transfer คือทุกอย่างพร้อมกัน เปลี่ยนพร้อมๆ กัน ทั้ง stack pointer, program counter, memory protection
3. Transparent restorable execution
 - หลังจากเกิดการ interrupt แล้ว ควรกลับมาทำงานที่ program เดิมได้ โดยไม่มีอะไรที่ถกกัน

INTERRUPT

setup โดย OS kernel

1. interrupt vector สร้าง table โดย kernel โดยทำการชี้ไปที่ code ส่วนที่ต้องทำการ run เมื่อเจอ events ที่แตกต่างกัน



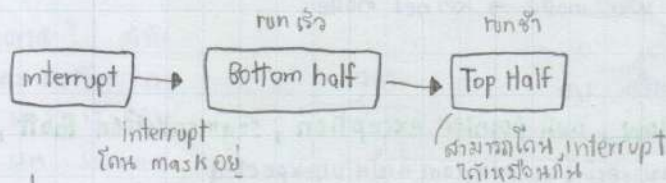
- interrupt masking

คือเมื่อ run interrupt handler โดยการ disable interrupt ไว้

* ใช้ในการ implement synchronization

- ความแตกต่างของ interrupt แบบ top half กับ bottom half คือ ถ้าเป็น bottom half จะมีการ enabled interrupt ด้วย คือถ้า interrupt หนึ่งมีความสำคัญกว่า มีผลทำให้การไปทำ interrupt นั้นก่อน

2. interrupt stack ^{per-processor เก็บไว้บน kernel เพื่อ user CPU state เอาไว้เก็บ local variable}
- ถ้าเก็บ interrupt stack ไว้ร่วมกับ user process memory จะทำให้ user process สามารถทำอะไรกับ stack นั้นก็ได้ เลยต้องทำการแยก interrupt stack ออกมา
 - ตัว I/O Driver จะถูกแบ่งออกเป็น 2 ส่วน I/O Top Half กับ I/O Bottom Half



ส่วนนี้ของ code ที่ถูกต้อง interrupt ถูก mask ส่วนใหญ่เป็น command & control จะใช้เวลา run เร็วมาก (ไม่ควรรีใช้เวลานานมาก)

3. interrupt handler

- (Linux's top) - Bottom Half → Non-Blocking เป็นส่วนที่รอก่อนเกิด interrupt พอทำงานเสร็จก็ไป wake up thread อื่น
- (Linux's Bottom) - Top Half ↓
- Run as another kernel thread (กลับไป run หลังจากทำ interrupt เสร็จแล้ว) คือต้องรอจนเสร็จ

* เมื่อเกิด interrupt บน interrupt stack จะเก็บ sp ไว้ 2 ตัว ซึ่งแตกต่างกัน ตัวหนึ่งเป็นของ user program อีกตัวเป็นส่วนของ interrupt ที่วิ่งเข้ามาที่ handler เพื่อไว้บันทึกว่ามี interrupt อื่นเข้ามาอีก

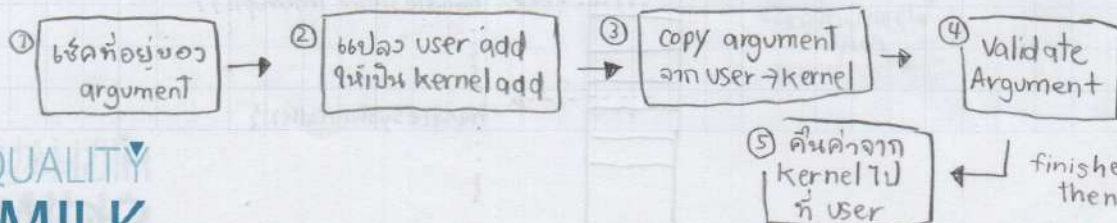
4. เมื่อจบการทำงานของ handler จะทำการคืนค่าต่างๆ ของที่เคย์ตั้งค่ารับ interrupt ไว้ทั้ง pc, program stack แล้ว switch กลับไปเป็น user mode

SYSTEM CALL ex. การเปิด file ผ่านพวก I/O

- เป็นส่วนที่เป็น interface เชื่อมต่อระหว่าง kernel กับ user
- Calling convention เป็นตัวบอกว่าส่วนไหนของโปรแกรมอยู่ตรงไหน ชื่ออะไร เพื่อทำให้เกิดการเชื่อมต่อกันได้ของ user กับ process ต่างๆ เรียกตัวเชื่อมต่อที่ว่า ABI ← Binary
- ຈຸດ implement แตกต่างกันในโปรแกรม system os แต่ละตัว
- ใช้หลักการ switching ที่เรียกว่า software interrupt
- Defensive Programming (ต้องมีการเขียน code ให้ครอบคลุม security)

Kernel system call handler ใน register หรือ user stack

1. ต้องรู้ว่า argument อยู่ตรงไหน , ต้องแปลง user address → kernel address
2. copy arguments จาก user memory ไปเก็บที่ kernel memory เพราะต้องป้องกันการเข้าถึงจาก user โดยตรง
3. Validate argument
4. copy ค่าจาก kernel memory กลับไปที่ user memory



kernel เรียกใช้งาน user เช่นการ notify

UNIX → "signal" windows → "asynchronous" ^{events}

UPCALL → user-level event delivery เป็นการ call จาก kernel ไปหา user

- Time expiration ex. Real-time UI
- Exception handling ex. การส่งค่าออกมาว่าต้องการ save file ใหม่? ตอนกดปิดโปรแกรม
- Asynchronous I/O ex. คำสั่ง I/O เสร็จแล้ว

→ มีการ implement signal handler ใน user ซึ่งประกอบด้วยส่วนประกอบคล้ายๆ ใน kernel
ex. signal handler, signal stack, signal masking

1.01.25

ตัวอย่างการนำไปใช้

1. VM Player

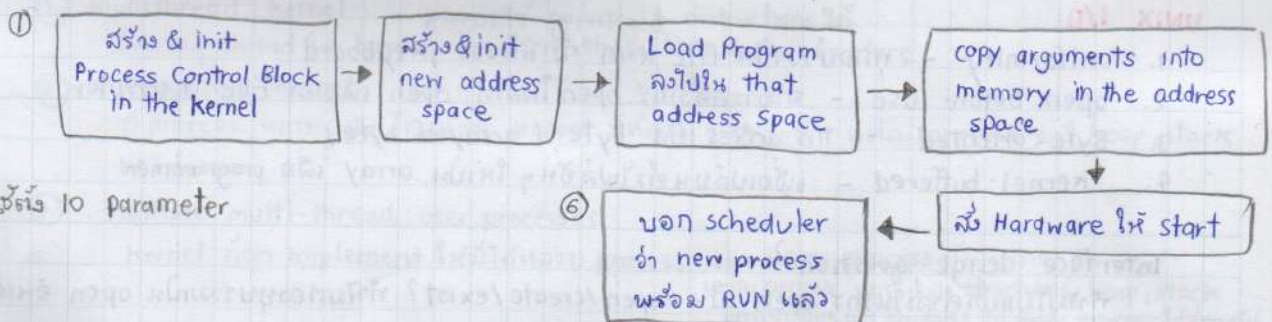
- มีการ install kernel driver ด้วย เพื่อแยกการ handler interrupt แยกกันว่า host หรือ guest (มีตัวเช็คว่าเป็น host หรือ guest)

PROGRAMMING INTERFACE

ส่วนที่ใช้ในการติดต่อกับผู้ใช้

- ▶ SHELL ^{หน้าที่} 1. job control system (เป็นส่วนที่ควบคุมการทำงานของ programmer ในการ create, manage program ต่างๆ
ใช้ system call ในการสร้าง process

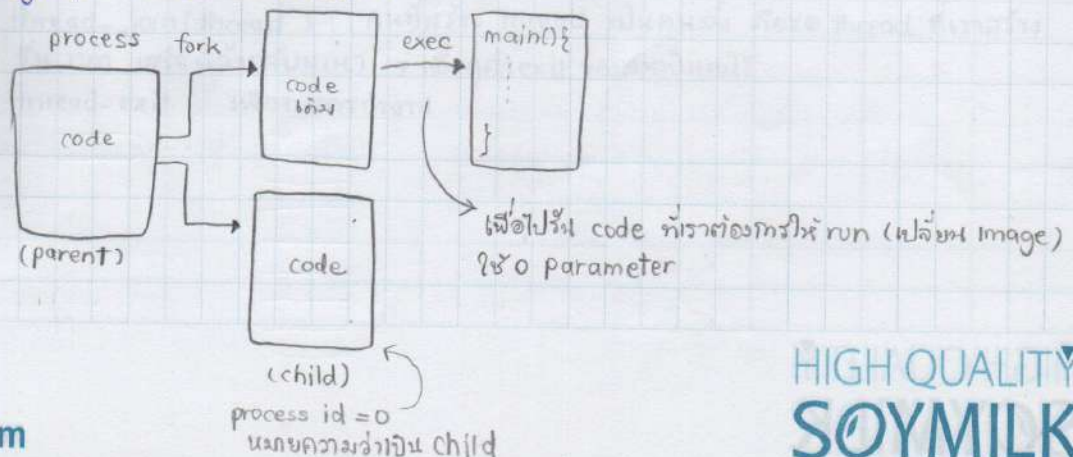
window create process (system call) simple in theory but difficult in practice



ใช้ถึง 10 parameter

unix process management simple practice but difficult in theory 😊

1. unix fork : concept คือการ clone process เดิม ไม่ต้องใส่ argument ลักตัว
2. unix exec : การเปลี่ยน code run
3. unix wait : รอ process เสร็จ
4. unix signal : การส่ง notification



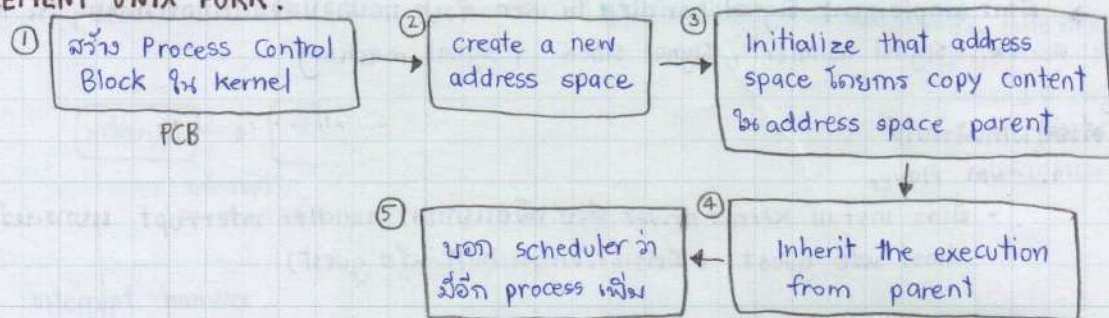
การย้ายจาก kernel \rightarrow user

- | UPCALL | VS | INTERRUPT |
|-------------------------------------|----|----------------------|
| ① 48 signal handler | | ① interrupt vector |
| ② signal stack | | ② interrupt stack |
| ③ Automatic save & restore register | | ③ transparent resume |

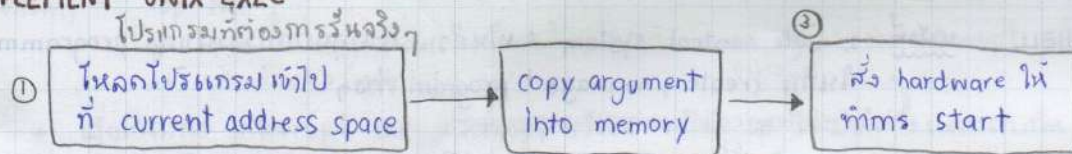
* ใน signal handler จะเปิด signal masking
คือทำให้ตัวเองเป็นอื่น ๆ ไป

- `FORK()` return error ไม่ได้เลย เพราะมันเหมือนเดิมเลย
- `EXEC()` ไม่มีกร return เลย เป็น void
- `WAIT()` return ได้ กรอให้ `child pid = 0` แล้ว แล parent ได้รับพอดี ก็ return ค่าได้เลย
(รอจนสร้าง child เสร็จ)

IMPLEMENT UNIX FORK.



IMPLEMENT UNIX EXEC



UNIX I/O

1. uniformity - ทุกอย่างเป็น file นมด ทั้ง mouse , keyboard
2. open before use - สามารถสั่งกร open ไฟล์ได้ open เพื่อเอา File descriptor
3. Byte-oriented - กร access เป็น Byte (array of byte)
4. kernel-buffered - เพื่อเปลี่ยนตัวไฟล์อื่นๆ ให้เป็น array เพื่อ programmer

Interface design question?

ถ้าไม่แยกคำสั่งในการ open เป็น open/create/exist? ทำไมต้องขบรวมเป็น open อ่ะดิยว
→ OS ไม่ได้รู้แค่ process เดียว ถ้าเขียน code แบบ `if(!exists(name))`

`create(name);`

`fd = open(name);`

แล้วมีการเข้าถึงไฟล์เดียวกัน อาจเกิดกรแย่ง content กันได้เพราะเรา split กันตอนมันออก

CONCURRENCY

ต้องสามารถ handle multiple things happen at the same time.

- OS สามารถทำงานแบบ concurrently เพราะต้องสามารถ handle interrupt ได้ เช่น process execution, interrupts, background tasks เพราะฉะนั้นเราต้องใช้ thread เข้ามาช่วยในการทำงาน

THREAD DEFINITIONS

- เป็น single execution ที่ใช้สำหรับ scheduler single execution sequence
- สามารถมีได้หลายๆ thread เพื่อใช้ให้สามารถสลับ execute separately schedulable

WHY?

ทำไมต้อง concurrency (หลาย thread)

- * 1. Server ต้องรองรับ multiple connections
- 2. Parallel programs ช่วยกันทำงาน
- 3. Program with user interface เพื่อให้ user รู้สึกว่ามัน interactive
- 4. ช่วยให้เรา hide network/disk latency
สามารถข้าม thread ที่อ่าน disk ไปก่อน แล้วเอาอันอื่นมา execute ก่อนได้

THREAD AT USER LEVEL & KERNEL

- ① Multithread kernel สามารถใช้ privileged instructions ได้
มีหลายๆ thread ใน kernel โดยมีการ share พวก data structure
- ② Multiprocess kernel
มี process หลายๆ อัน โดยแต่ละ process มีเพียง 1 thread แต่ละ process จะมี heap, stack ของตัวเอง
- ③ Multiple multi-thread user processes
kernel ที่ถูก implement ให้มีได้หลาย process 64bit, หลาย process มีได้หลาย thread แต่ละ process แยก data structures, heap, stack ออกจากกัน แต่ threads ใน kernel, process ใช้ร่วมกัน

THREAD ABSTRACTION

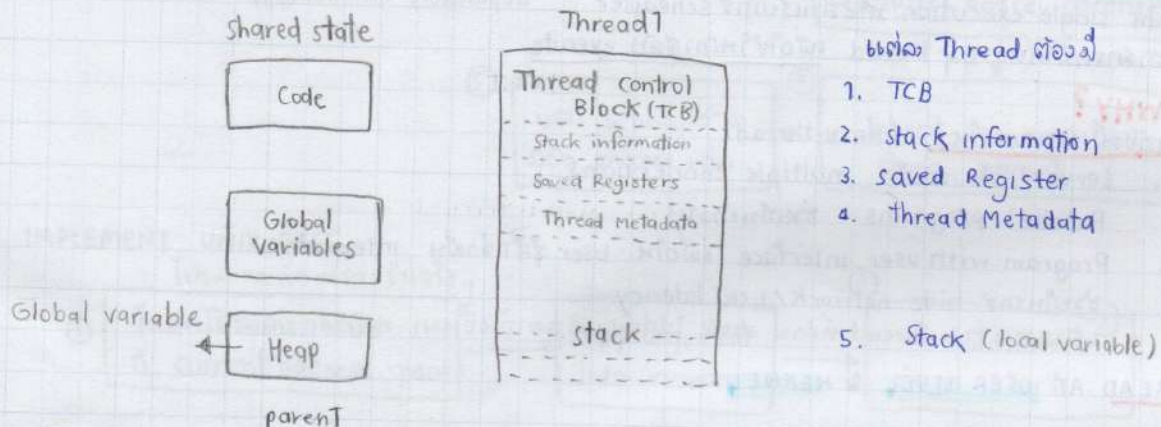
มุมมองของการเขียน code คือมองเป็น infinite number of processor แต่ในความจริงมันถูกจำกัดด้วยจำนวน core ของ hardware โดยกรณีของแต่ละ thread มีได้หลายรูปแบบ

OPERATION (thread, func, args)

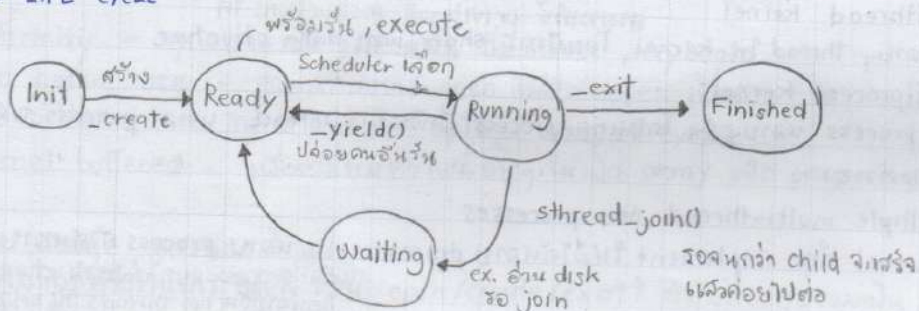
1. thread_create : สร้าง thread ใหม่ขึ้นมาเพื่อรัน func(args)
2. thread_yield() : ยอมไม่ใช่ processor ยอมให้คนอื่นมาใช้ก่อน
3. thread_join(thread) : คนที่สร้าง thread เป็นคนสั่ง คือรอ thread ที่เราสร้างขึ้น run เสร็จแล้วกลับมา → เพื่อดูค่า exit ของมันเป็นอย่างไร
4. thread_exit : เพื่อจบการทำงาน

- thread สามารถสร้างลูกตัวเองได้ และสามารถรอเพื่อ join ได้ , data จะมีการแชร์กันเฉพาะ before fork , after join เพราะพอ join มันก็จะกลับมามา parent มันเพื่อทำงานต่อ
ex. web server : สร้าง thread ใหม่ทุกๆ connection
merge sort
parallel memory copy : copy memory เยอะๆ แบ่งเป็นย่อยๆ

- THREAD DATASTRUCTURE
จะมีใน thread, thread เป็นของตัวเอง



- THREAD LIFE CYCLE



IMPLEMENTING THREAD

- * ถ้าเป็น kernel thread จะ available only in kernel , user ต้องเรียก thread ใน kernel เรียก sys
→ ต้องใช้ func(args) ลิงกับ stack แล้วค่อยย้ายไป ready list
- 1) Thread_fork(func, args) : มีการสร้างส่วนต่างๆ ของ thread ex. TCB , stack , pointer ต่างๆ สร้างเสร็จแล้วต้องใส่ใน ready list แล้วค่อยรอรันหรืออาจรันเลยก็ได้

* ปัญหาของ stack คือ

ถ้าเราเรียก recursive เยอะๆ เช่นทำงานแบบ Recursive จะทำให้เกิดปัญหานบน physical memory , virtual memory ถ้าเกิน limit ที่ memory รับได้ก็ error → CRASH!

OS/161, Linux kernel JAVA.

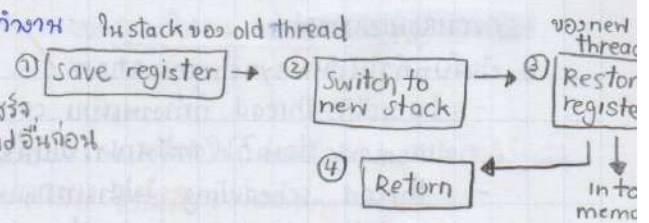
BZERO - คือปกติเวลา process มีหน้าที่ทำอะไรก็
เขียนข้อมูลลงใน memory ซึ่งเวลาผ่านไปเพื่อให้
process อื่นเอาไปใช้ได้อีกมันไม่ได้ลบ memory ที่
ค้างนั้น OS ก็ใช้ Blockzero ในการสลับแทน พอ
กันการเกิด data leak ถ้ามีหลายๆ gig ก็ใช้หลาย
ๆ thread ช่วยกันลบ

BZERO 😊

← กันขึ้นใน kernel thread not between kernel & user

• **THREAD CONTEXT SWITCHING** : การสลับ thread การทำงาน ใน stack ของ old thread

1. Voluntary - Thread-yield (ยอม)
 - Thread-join ถ้าลูกยังไม่เสร็จ, ที่ข้ามไปทำ thread อื่นก่อน
2. Involuntary - interrupt or exception
 - ไม่เต็มใจ - some higher priority



• **A SUBTLETY** (ต้องมีการ call stub ก่อน เวลา function return มีหน้า return ไปที่ stub)

พอเรามีการ create thread จะมีการ put thread ให้ไปอยู่บน ready list พอมีการ run thread จาไป calls switchframe เพื่อทำหน้าที่ save old thread & restore new thread จากที่ save ไปเมื่อ state state

• **INVOLUNTARY PROCESS SWITCH**

1. timer, I/O interrupt
 - ถ้ามันอยู่ใน kernel ทั้งคู่ก็ไม่ต้อง switch mode ก็สลับไป ~~kernel~~ ถ้า resume แล้วจะ ทำ thread อันต่อเลยก็ได้ ไม่ต้อง switch กลับ

MULTITHREADED USER PROCESS (IN LINUX, MacOS)

TAKE 1 จขมองว่า user thread = kernel thread เวลาจะมีการ fork, join, exit จะเกิดขึ้นที่ kernel mode thread เวลาทำงานจริงจะมีการ switch ระหว่าง user mode กับ kernel mode

• **JAVA**

TAKE 2 Green Thread ทำทุกอย่างอยู่ใน user level นหมดเลย ถ้าจะมีการส่ง interrupt timer ต้องใช้

UPCALL / UNIX SIGNAL ส่งเข้าไป

→ มีการสร้าง Thread lib เพื่อทำการ switching เองเพราะไม่ได้ปล่อยให้ kernel ทำ Application สามารถ control การ switch ได้ด้วย

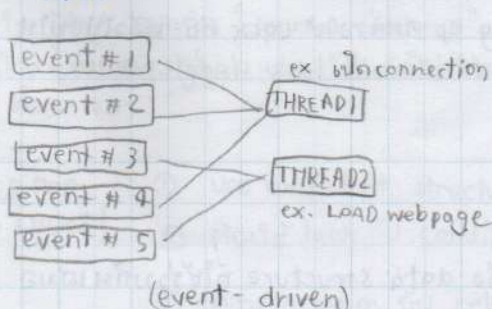
ถ้ามี thread บน user space ขอใช้ I/O จะไม่สามารถใช้ thread อื่นได้

TAKE 3 Scheduler Activations

← ออก midterm 2015

EVENT DRIVEN PROGRAMMING!

- เมื่อมี event เข้ามาที่เอา thread หนึ่งเข้าไปรับๆ ทุกๆ event แล้วค่อยมาดูว่า event หนึ่งต้องทำอะไรบ้าง แต่ถ้าเป็นแบบ multithreaded คือทุก connection คือ 1 thread แล้วทำงานตั้งแต่ต้นจนจบ



(event-driven)



ทำงานทุกอย่างที่เดียวเลย

(multithread)

* ใช้ในงานประเภทไหน? เพราะอะไร? ex?

SYNCHRONIZATION

เวลาที่มีกรเข้าถึง object เดียวกัน

* ปัจจัยที่ทำให้เกิด synchronization

- ในกรณีที่ thread ที่ทำงานแบบ concurrently read/write memory ตัวเดียวกัน ตัวไหนก็ได้ใช้ memory ก่อน? (ถ้าเ็นเข้าไปเปลี่ยนค่าพร้อมๆ กัน)
- thread scheduling ไม่สามารถคาดเดาได้ (เปลี่ยนไป-มาตลอดเวลา)
- Compiler/hardware instruction reordering (เอา instruction มาเรียงใหม่ ทำให้เร็วขึ้น)
- Multiword operations ไม่มีความเป็น atomic
อาจโดนแทรกแซงได้

WHY REORDERING?

คือทำการ guess ว่าค่าควรจะเป็นอะไร แล้วทำไปก่อนเลย จะได้ไม่เสียเวลา

• compilers : การที่จ.สร้าง code ให้มีประสิทธิภาพ requires การวิเคราะห์ (analyzing control) หรือ data dependency

• CPU : เป็นกร เขียน buffering คือทำให้คำสั่งต่อไปทำงานทันทีที่มีกรเขียน buffer เสร็จ

วิธีแก้ไข: **MEMORY BARRIER** - ทำให้งานที่ค้างหน้าก่อนก่อนจะ optimize ทำให้ compiler ไม่เข้ามายุ่ง กับส่วนที่เป็น instruction สำหรับ compiler & cpu (กันไว้ไม่ให้เขา shuffle จนสลับไปหมด)

MEMORY BARRIER

- ทุกๆ operations ก่อน barrier จ. complete ก่อน barrier returns

- ไม่มี operation ในนเริ่มทำงานก่อนที่ barrier จ. returns (after barrier)

DEFINITIONS

Race condition : output ของโปรแกรมที่ทำงานพร้อมๆ กัน ซึ่ง depends on order ของ operations ระหว่างแต่ละ threads ex. แย่งกันไปซื้อนม นพยายามแย่งกันไปซื้อนม

↓ ป้องกัน

Mutual Exclusion : Only one thread does one thing at a time

→ **critical session** : code ที่มีแค่ 1 thread ที่จะสามารถเข้าถึงได้แค่ 1 ครั้ง (สิ่งที่มันทำ) ส่วนที่เข้าถึงได้แค่ thread เดียวเท่านั้น

Lock : ป้องกันการเข้าถึงบางอย่าง

Correctness properties

- liveness** : กรันที่ว่าต้องมีคนทำสิ่งๆ น จ.ก็คนก็แล้วแต่
- safety** : ต้องมีกฎสุดแค่คนเดียวก่อนได้เป็นคนทำ

1. ก่อนจ.เข้าถึง critical session, ก่อน accessing shared data
2. ต้องมีการ unlock when leaving
3. มีการรอถ้าสิ่งที่เราต้องการ is locked!

① LOCK

ช่วย LOCK ส่วนที่เป็น "critical session"

1. Lock acquire

→ รอจนกว่า lock จะว่าง แล้วค่อยเข้าใช้ ถ้า lock ไม่ว่างก็รออยู่ในสภาน. wait queue

2. Lock Release

→ คืน lock แล้วทำการ waking up คนที่รอใช้ Lock ต่อ แล้วไปอยู่ใน ready queue

* มีได้แค่ 1 คนที่ถือ lock ใน 1 ช่วงเวลา ถ้าไม่มีคนถือ lock คนที่มี priority สูงสุดจะได้สิทธิในการครอบครองก่อน

• กฎในการใช้ Lock

① Lock is initially free

② ต้องทำการ acquire ก่อนที่จะมีการเข้าถึง data structure ที่ใช้ร่วมกันเสมอ

ถือ lock เป็นของตัวเอง เอา Lock ไปครอบไว้ beginning of procedure!

Date _____

Subject _____

* ถ้ามี shared data structure ต้องเอา lock ไป *
ครอบเสมอ



end of procedure

ห้าม return ก่อน unlock

- ③ ต้อง release after finish ทุกครั้ง ต้องให้คนที่ครอบครองเป็นคนที่ release เท่านั้น!
④ ห้าม! เข้าถึง shared data without lock

② Condition variable → กระบวนการรออยู่ใน critical section จนกว่าจะใช้ก็ต่อเมื่อได้ hold lock ไว้แล้ว

(Atomic)

เท่านั้น

ถ้า wait มี lock ก็จะทำงานเลย ถ้าไม่มีจ.รอ waiting state

- มีกรเช็คได้ 1. wait = คือ automatically release lock (automatically execute)
ว่ามีคนต่อ 2. signal คือ ทำการ wake up ตัวที่รออยู่ (ถ้ามี) } แล้วเอาเข้าไปอยู่ใน ready queue
การใช้ lock 3. Broadcast คือ wake up ทุกตัวที่รอ lock อยู่
หรือเปล่า

example

① ถ้าเป็นกรณีที่ bounded queue
ว่าง

⑤ หลุดจาก while เพราะได้
รับ signal มา

```
get() {
    lock.acquire();
    while (front == tail) {
        empty.wait(lock);
    }
    item = buf[front % max];
    front++;
    full.signal(lock);
    lock.release();
    return item;
}
```

```
put(item) {
    lock.acquire();
    while ((tail - front) == MAX) {
        full.wait(lock);
    }
    buf[tail % MAX] = item;
    tail++;
    empty.signal(lock);
    lock.release();
}
```

② จ.สามารถทำการ put ได้
เพราะไม่ติด while มี

③ ทำการ put item เข้าไป

④ signal ไปบอกตัวที่ wait (empty) ว่ามี item แล้ว
ส่งมันไปอยู่ใน waiting queue

- ถ้ามี variable ที่ต้องแชร์กัน ต้องมีการเอา lock มาครอบก่อนที่จะทำการเช็ค
- มีการใส่ wait เพื่อช่วยในการรอนลูปรอ เพื่อรอรับ signal ** ถ้าไม่ติด wait จะไม่ได้รับ signal
- ก่อนจะมีการใช้ Condition Variable ต้องมีการ hold lock เสมอ เพื่อให้ได้ state ที่ถูกต้อง
- เป็น memory less ควรทำให้มี wait รอใน loop จนกว่าจะมีการส่ง signal มา
- wait จะเป็นตัวที่ automatically release lock ต้อง hold lock ตอน wait หรือ signal ด้วย เพราะมีการ share state
- เมื่อ waiting state ถูก signal ตัวที่กำลังอยู่ใน waiting state จะย้ายไปอยู่ใน ready list เพื่อรอให้ CPU มาเอามันไป execute
- เมื่อ lock release ใครก็ได้จะเป็นคนที่ acquire มันแทน

* ใน JAVA อาจเกิด "spurious wakeup" คือเป็น wake up ที่เกิดมาจากไหนไม่รู้, wake up โดยที่ยัง
ไม่ได้รับการ signal เลย วิธีแก้คือต้องเอา code เข้าไปอยู่ใน while loop เพื่อรอรับ signal
ตลอดเวลา

RULES FOR
USING LOCKS!

- ① use consistent structure
- ② ต้องใช้ lock กับ conditional variable
- ③ มีการ acquire กับ release ทุกครั้ง
- ④ มีการ hold lock ต้องใช้จริง
- ⑤ wait in while loop
- ⑥ ห้าม spin in sleep!

STRUCTURED SYNCHRONIZATION → ต้องเขียน code ที่เป็นการ synchronization.

- ① Identify object ที่ต้องใช้แชร์กัน (มีมากกว่า 1 thread ในการทำงาน)
- ② เอา lock เข้าไปครอบ แล้วต้อง release lock ทุกครั้ง ก่อนจะมีการออกจากการทำงาน
ไม่ครอบ lock แล้ว sleep เพราะจะทำให้ทุกคนต้องรอ จนกว่ามันจะ wake up
- ③ อย่า lock condition ที่ต้องทำการรอหา
- ④ ถ้าจะ wake up ให้ใช้ signal or Broadcast **เพราะมันไม่จำเป็นที่ต้อง signal ใด**
- ⑤ Condition variable ต้องทำงานใน consistent state เท่านั้น

COMMON

→ MESA & HOARE SEMANTICS ของ signal กับ wait

- ① MESA - เอา while loop ไปครอบ wait ให้
- เมื่อมี wake up ที่เอาไปใส่ใน ready list
- มี queue ที่เก็บตัวที่ wait (signal จะเลือกตัวแรกออกไป wake up)
- ② HOARE - ไม่ใช่ while แต่ใช้กับ if แทน

• IMPLEMENT SYNCHRONIZATION

1. using memory load/store

2. ต้องมีการ Lock :: acquire() Lock :: release()
→ disable interrupts → enable interrupts

จ.ไม่กำหนด เพราะไม่มัน interrupt ส่งมาเยอะ buffer จะต้อง drop ทั้ง (critical session ตัวเล็ก)

* ถ้าเป็นการ implement บน UNIPROCESSOR ไม่ควร disable interrupt นานเกินไป พอมีการ enable interrupt CPU ก็ไม่ทราบตัวว่ากลับมาทำงานเดิมรึเปล่า

* MULTI PROCESSOR จะมีการเอาคำสั่ง Read-modify-write instructions มาช่วย โดยทำ read, modify, write ที่เดียวเลย ไม่มีใครแทรกได้
ทำให้มัน atomic

SPINLOCKS

เวลามันรอมันจะทำการ spin cpu คือมีการใช้ cpu ตลอดเวลา ex. พอ kernel เวลาจะมีการสลับ thread ในการทำงานมันจะเกิด cost สูงมาก ทำให้มันเลือกที่จะ spin รออยู่เฉยๆ ดีกว่า (context switching cost เยอะ)

- spinlock:: acquire() { ... }
- spinlock:: release() { ... } มีคำสั่ง memory barrier() เพื่อบอกว่าหลังจากทำ code ในส่วนนี้ไปก่อน ต้องทำขั้นตอนให้เสร็จก่อน

how many spinlock?

- ควรสั้นๆ เล็กๆ ไม่กี่ operation ควรเสร็จ
- One spin lock per kernel ทำให้เกิด bottle neck!
- use one spin lock per lock, scheduler ready list

* ใน uniprocessor เราเห็นว่าเรา run thread หนึ่งอยู่คือ use global แต่ถ้าเป็น multiprocessor แต่ละตัวจะมี register เป็นของตัวเองเพื่อเก็บ TCB pointer ต้องใช้วิธีแก้คือ

1. hardware : ~~per-processor~~ per-processor register
2. ต้องมีการชี้ไปที่ address ของ TCB at the bottom of the stack

Date

LOCKS

VS

CSP

Subject

- | | |
|------------------------|--------------------------------------|
| • มีการสร้าง lock | • only a single thread |
| • สร้าง method ใน lock | • มีการ send message & reply message |
| • wait | • อกจาก queue แทน |
| • Signal | |



Semaphores: ทำให้อยู่บน non-negative value 0 ถึงค่าบวก

- P() รอให้ค่าเพิ่มจาก 0 - ไปเป็นมากกว่า 0 แล้วค่อยๆ ลดลง (wait)
- V() เพิ่มค่าขึ้นทีละ 1 เรื่อยๆ (signal)
- มีการจำ Number คือมีการจำ state ว่าอยู่ที่ไหน (ไม่เหมือน lock) แต่มีการ implement มีคิวยกต่อ
- คิดเยอะ ทำให้คนส่วนใหญ่ใช้ lock แล้วยก state เอา (front, tail)
- ต้องมีการสร้าง queue
- Unlock ด้วย interrupt handler, fork/join

CSP

เป็นการส่งคำสั่งเป็น message ส่งเข้าไป คนทำงานมีแค่คนเดียว (only thread allowed to touch data) ถ้าต้องการทำงาน ก็แค่ส่ง message กับ method name + argument เข้าไป ถือว่าการทำงานแบบ lock กับ semaphores คือ no Memory races!

MULTI OBJECT SYNCHRONIZATION

- มีหลายๆ object และหลายๆ thread ต้องมีการคำนึงถึง performance, semantics, Deadlock

① Synchronization performance

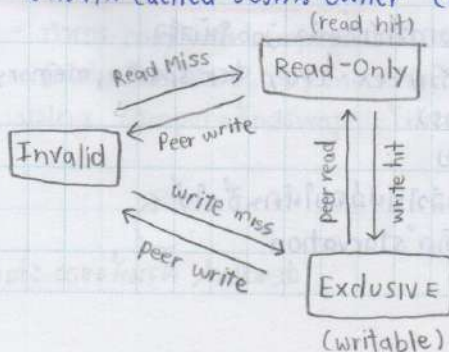
โปรแกรมมี concurrent threads เยอะๆ แล้วยังหลายๆ core จะทำให้ต้องพิจารณาปัญหา

- overhead ในการสร้าง threads (สร้างมาแล้วไม่ได้ใช้)
- Lock Contention ถ้ามี CPU หลายตัวก็ต้องแย่งกันใช้ lock ด้วย
- ต้อง protect shared data การเข้าถึงข้อมูลเดียวกันจากหลายๆ core อาจทำให้เข้ากันถึงข้อมูลผิด
- False sharing การเข้าถึงข้อมูลที่ไม่ได้เป็น shared data ถูก invalidate ใน cache ไปด้วย

① MULTIPROCESSOR CACHE COHERENCE < MCS locks / RCU locks

สมมติว่า Thread A ทำการ modified data เสร็จแล้วทำการ release lock แล้ว thread B ทำการ acquire lock แล้ว read data ถ้า new data cached ที่ processor A มันต้องทำการ invalidate cache ก่อน ก่อนที่ processor B จะเขียนได้

- cache coherence ทำให้เปรียบเทียบเสมือนว่า data ที่สนใจมีแค่ copy เดียว (ถ้าเป็นการอ่านอย่างเดียวจะมีหลาย copy ได้) แต่ถ้าเป็นการเขียนจะต้องมีแค่ one cached copy ใช้หลักการการ take ownership โดยทำการ invalidate ทุกๆ cache ทุกๆ copy ของ line พอเขียนแล้ว modified data จะยังอยู่ใน cache เรียกว่า "write back" + เขียนลง cache ก่อนค่อยไปเขียนลง primary storage เวลาคนอื่นจะมาอ่าน ก็ทำการอ่านจาก cached ของคน owner (ถ้าไม่มีคนเป็นเจ้าของค่อยไปอ่านที่ main memory)



- ① Invalid (no data) → Read Miss → ไป fetch data มาจาก memory จะย้ายไปอยู่
- ② Read-Only ถ้าต้องการเขียนเอง เป็น owner (write hit) ไปอยู่ Exclusive คนอื่นเขียน (peer write) ต้องทำการ invalidate cache ที่เราถืออยู่
- ③ Exclusive เป็น owner อยู่จะออกก็ต่อเมื่อ 1. คนข้างๆ จะอ่าน 2. คนข้างๆ จะเขียน

SEMAPHORE

```

getO{
    fullSlots.P(); ลดค่าลง 1 เพื่อบอก
    mutex.P(); ว่ายังมีจำนวนเต็ม
    item = buf[front%MAX]; ลดลง 1
    front++;
    mutex.V();
    emptySlots.V(); ให้มี 1 เพื่อบอก
    return item; ว่าว่างเพิ่มขึ้น 1
}

put(Item) {
    emptySlots.P();
    mutex.P();
    buf[last%MAX] = item;
    last++;
    mutex.V();
    fullSlots.V();
}

```

- P ลดค่าลง 1
- V เพิ่มค่าขึ้น 1

Compare-and-swap() คือเป็นกรรไกรตัดค่าที่มันเจ,
เขียนทับเป็นค่าเดิม ถ้าเป็นค่าเดิม มันเลย แต่
ถ้าเป็นค่าใหม่ต้อง คัดใหม่

test_and_set(&lock) คือเป็นกรรไกรใช้ว่า ค่า lock นั้นเป็น
0, 1 อยู่ ถ้า 1 แสดงว่ามีคนใช้งานอยู่ ให้ spin รอ จนกว่า
จะเป็น 0 มันจะ set ค่าให้เป็น 1 แล้ว enter critical section

DIRECTORY-BASED CACHE COHERENCE

- เพื่อใช้ในการบอกว่า address มีอยู่ที่ core ไหนบ้าง ถ้าเกิด read miss → ไปเอาจาก owner แล้ว invalidate
write miss → invalidate ทุกตัว เพื่อให้นัตัวเองเป็น owner
- Read-modify-write ทำการ fetch cache ให้เป็น exclusive เพื่อกันคนอื่นมา Read, Write จนกว่าจะ
complete ทั้ง instruction
→ ex. การตรวจเช็ค spin lock

การแบ่งแย่ง LOCK กัน

REDUCING LOCK CONTENTION

- ① Fine-grained locking คือการแบ่ง object แบ่ง Lock กันแต่ละตัว ex. hash buckets โดยแต่ละ bucket
ก็มี lock เป็นของตัวเอง
- ② Per-processor data structure บอกเลยว่า processor core ไหนเอา data ชุดไหนไป
- ③ Ownership / Staged Architecture มี thread เดียวที่จะ access data ได้ ต้องทำงานแบบ pipeline
คือแต่ละ stage รับบน core ที่แตกต่างกัน (ไม่มีการเข้าถึง data จากหลาย thread)
→ ex. fanpage net idol คนหลายๆ คนต้องการเข้าถึง data เดียวกัน
แต่สมมติว่า LOCK ยัง busy อยู่ต้องใช้วิธีอื่นในการจัดการ

MCS Locks → ต้องการ implement ก่อน lock ถ้า Contend เยอะๆ แต่ยังมีปัญหาเรื่อง overhead

RCU Locks → (read-copy-update)

① TEST & SET กับ ② TEST AND TEST & SET ต่างกันตรงที่ ② จะมีการเช็คก่อนว่ามัน LOCK อยู่ไหม ถ้า busy
ก็ทำการ add in TEST & SET ช่วยลด overhead ลงในครั้งกับสิ่ง หรือมันก็เพิ่ม delay ใน loop / ทำให้มัน
adaptive โดยถ้ามีคนรอเยอะๆ ก็ delay เยอะกว่า waiter น้อยๆ ทำให้มัน

- ② • MCS สร้าง linked list ขึ้นมาสำหรับคนรอ แล้วทุกคนไปหมุนที่ memory ของตัวเอง รอจนกว่าคนที่
connection ขอ hold lock จะ release() แล้วไปบอกเพื่อนที่ spin รออยู่ ใช้หลักการ atomic operation COMPAREANDSWAP คือการ
operate memory word โดย compare data ว่ามันถูกใช้งาน มีการเปลี่ยนแปลงหรือไม่ ถ้าไม่มีก็ใช้งาน
ได้เลย แต่ถ้ามีคน compare & swap ก่อนมัน ก็ต้องกลับไปรอ return an error and loop again)
CONS overhead เยอะในการสร้าง linked list

- ③ • READ-COPY-UPDATE (RCU) เห็นว่า data ที่ต้องการใช้คืออ่านอย่างเดียวเท่านั้น เลยเห็นอันใด จ.เขียนที่ซ้ำ
ช่วงมัน คือทำการอ่านโดยไม่ต้องใช้ Lock สามารถทำมันได้หลาย version ได้ ขึ้นอยู่กับว่า Reader มีการ
อ่านเมื่อไหร่ ex. facebook news feed คือเวลาเราเขียนไม่ต้องหยุด ขึ้นอยู่กับเวลาที่ publish โดย reader
ไม่ต้องใช้ Lock แต่ทำการ disabled Interrupt ได้เลย แต่ writer ต้อง acquire write lock แล้วหลังจาก
publish ก็รอจนกว่าจะถึง glance time แล้วค่อยทำการ delete old data

DEAD LOCK DEFINITION Resource คือ สิ่งที่ thread ต้องการใช้เพื่อทำ job ให้เสร็จ

เมื่อมีการใช้ Lock มากขึ้นก็มีการใช้ resource ที่มากขึ้นด้วย ex. CPU, disk space, memory) แบ่ง

1. preemptable สามารถถูกแย่งคืนได้ (แย่งโดย OS)

2. Non-preemptable ไม่โดนแย่ง จนกว่าจะปล่อย

→ อาจทำให้เกิด starvation ถ้ามี resource ที่มีอันเดียวแล้วไม่ปล่อยให้คนอื่นใช้

→ Deadlock การรอคอยอย่างไม่มีวาระลง รอกันไปมา ทำให้เกิด "starvation"

deadlock ส่วนหนึ่งจะ starvation