อ่านสีดำ แล้วตามด้วยน้ำเงินนกฮูก

Role of OS

↳ Referee ผู้ตัดสิน → cpu, memory(ram)

→ ทำหน้าที่จัดสรรทรัพยากร ให้ใครเข้าไปใช้ก่อนหลัง, ใช้ส่วนไหน ได้บ้าง มากน้อยแค่ไหน, นานแค่ไหน

- ↳ Resource allocation among users, applications
- ↳ Isolation of different users, applications from each other
- ↳ Communication between users, applications

เปิดโปรแกรม>1

application อื่นมาใช้ทรัพยากร ที่ถูกใช้อยู่ไม่ได้นะ → ป้องกันกัน

การสื่อสารใดทำได้หรือทำไม่ได้ → ทำตัวเหมือน firewall กันพัง

RAM

เตรียม api เพื่อคุย app1<->app2

ทำให้ application มองไม่เห็นกัน

↳ Illusionist กันการเรียกใช้ api

- ↳ Each application appear to have the entire machine to itself
- ↳ Infinite number of processors, (near) infinite amount of memory, reliable, storage, reliable network transport

ทำให้การพัฒนาโปรแกรมเป็นไปได้ง่ายขึ้น → (dev) → ตอนเราเขียนโปรแกรมไม่ต้องคำนึงถึงนอก hw, จองmem

↳ ถ้าไม่มี os เวลาพัฒนาโปรแกรมที่ต้องการใช้ hw แปลกๆ → หลากหลายมาก ผู้พัฒนาเกมส์จะต้องเขียนให้โปรแกรมในหลายๆ version มาก

↳ game v. nvidia
game v. amd

→ os สร้าง interface ขึ้นมารองรับให้ programmer เรียกผ่าน api only → no need to touch hardware

↳ เป็นตัวกลางเหมือนตัวแปลงสัญญาณแปลง logic spec เดียวกันหมด (vm)

↳ os สามารถปรับขนาดได้

↳ programmer ดูแค่ logic

↳ Glue

- ↳ Libraries, user interface widgets, ...

หน้าที่ในการประสาน

↳ provide วิธีการที่จะช่วยต่อให้ software เรียกใช้ทรัพยากรต่างๆ ได้ผ่าน api

ทำงานหลายส่วนพร้อมกันได้

virtual memory
virtual ram } ถ้าแรมเต็ม ไม่พอใช้งาน

ทำให้เกิดการเชื่อมโยง app → sys, hw, app

↳ What is OS?

↳ ไม่ได้มีคำนิยามกลาง → ส่วนใหญ่มาจากปากๆของเซียน

OS คือ set of softwares → มี .exe หลายๆตัวมากทำงานบน window

↳ มีความซับซ้อนในการทำงานมากขึ้น → ถ้าเขียน OS ให้อยู่ในไฟล์เดียว

↳ ขนาดแบบบึ้มๆ ยัดเข้า ram → แตก!?

→ แต่ก็ไม่ได้ใช้งานทุกส่วนอยู่แล้ว

แบ่งชนิด OS

↳ ใน มศ แบ่งตามการมองเห็น → user มองเห็น, ไม่เห็น

↳ พวกเครื่องซักผ้า, คอมพิวเตอร์

ชนิดแบ่งได้ไม่ตายตัว ขึ้นอยู่กับเกณฑ์

↳ OS ใช้งานทั่วไป, OS ใช้งานเฉพาะทาง

↳ กลุ่มการทำงานทั่วไปหมด

↳ ส่วนใหญ่เป็นพวกมองไม่เห็น → พวกเครื่องซักผ้า ตู้หยอดตังค์

นิยามแบ่งตามความสะดวกใจ

Operating System Evaluation → การประเมิน → หัวข้อในการประเมินคืออะไรบ้าง? → ชี้ว่าดีหรือไม่ดี

↳ Reliability and Availability → reliability → ออกแบบมาดีไหม ก็ต้องใช้งานได้แบบนั้น → bug เยอะ-น้อย

→ availability → ทำงานและไม่ fail → 8 ชม ต่อเนื่องไม่ล่ม นานๆ

↳ ถ้า hw ไม่มีปัญหา → ดูที่การทำงานของ OS เพียวๆ

↳ สามารถรับ input แปลกๆได้เท่าไหร่

↳ Security → physical

→ OS นั้นเจาะยากไหม

↳ เมื่อก่อน internet ยังไม่เยอะ

↳ โอกาสโดนขโมยข้อมูลเยอะ (now)

↳ เช่น isolation ระหว่าง user

↳ ของ OS ระบบคอมเดียว

↳ การไม่ให้ข้อมูลซึมเข้าถึงกัน

↳ Portability → OS นั้นสามารถใช้งานบน hw ที่หลากหลายได้ → spec ต่ำ, สูง, ชนิด hw ที่ต่าง

↳ & application อื่นๆ สามารถไปทำงานได้ด้วย

↳ AVM, API, HAL } → concept ที่เอาไปใช้

↳ เพื่อทำ portability

↳ Performance → ถ้า port ↑, avail ↑ แต่ performance ↓ ก็แย่ก็ไม่ดี

↳ เขียนไว้เยอะมากเกินไป → ไม่ work

↳ Overhead, efficiency → overhead ↓ → eff ↑ → performance ↑

↳ มี process อื่นมาทำงานมากไป → การ load ข้อมูล, เช็ค security

↳ Fairness, response time, throughput

↳ ความยุติธรรมในเชิง OS, math ต่างกัน → ทุกงานมีสิทธิ์เข้าใช้ cpu เท่าเทียมกัน

↳ t เวลา → มี s app → แบ่งกันทำงาน $\frac{t}{s}$ ต่อ app

→ ตามความรู้สึก (UX) ของ user

algorithm : ใส่ weight น้ำหนักตามการใช้งาน

ที่ใช้งานมากก็จะได้ส่วนแบ่งมาก → responsive ดี

ของ window : foreground , back ground

↳ มี response แต่ตัวที่ app ใช้ cpu only

↳ ตัวอื่นที่ใช้งาน, ทำงานไม่ได้

ถ้าไม่เกิดตัวก็ไปรแกรมที่ใช้ → responsive ↓

Throughput ต่างกันไปตาม program

↳ Performance predictability

↳ ประสิทธิภาพที่ทำนายได้ → ตอบสนองก่อน ขึ้นหน้าจอทุก 1 วิ / 0.5 วิ แบบแน่นอนสม่ำเสมอ

↳ ถ้าเป็นแบบวันละ 1 วิ, 2 วิ, 3 วิ → ทำนายไม่ได้ ↳ เดาได้

ถ้า user ทำนายได้ → ไม่หงุดหงิดมาก

" ——— " ไม่ได้ → ควบคุมมาก

↳ adoption → ความเป็นที่ยอมรับของผู้พัฒนา software

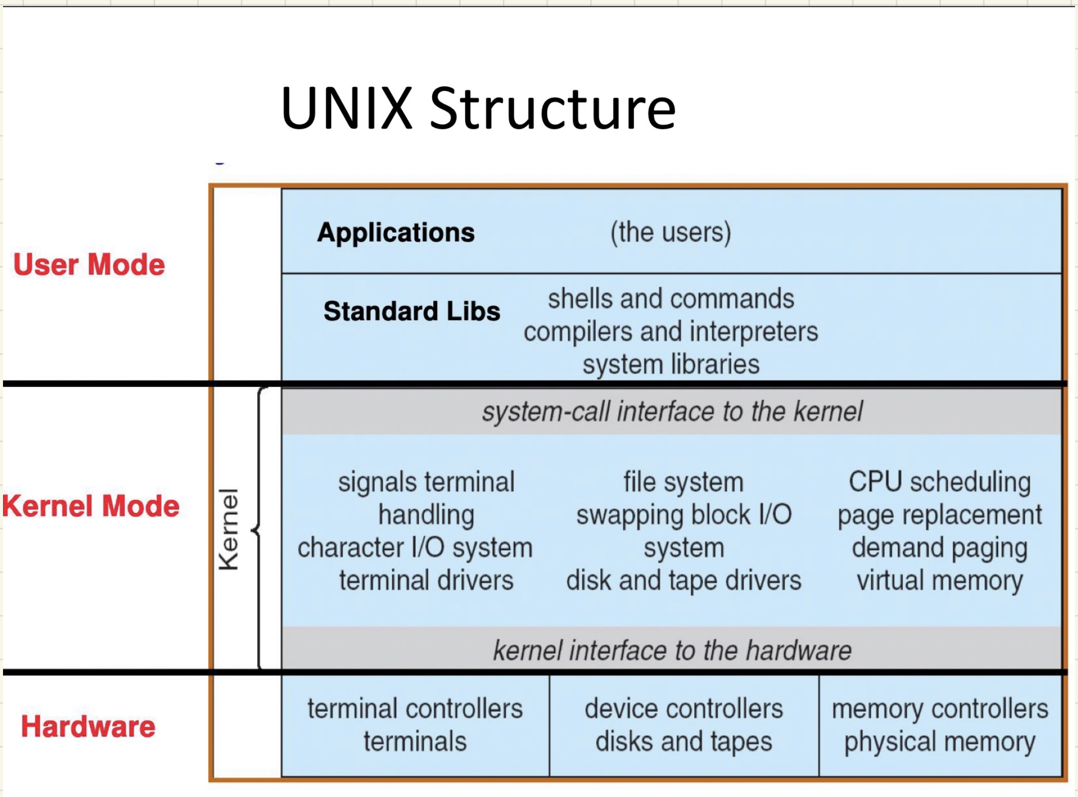
เกี่ยวข้องกับ portability อีก

↳ ตัวที่ยืดหยุ่น (เลยงา) os ขึ้นจะอยู่ต่อไป → คนใช้งาน application คนต้องการเงิน

↳ ถ้าไม่ดีก็แตก

The kernel abstraction.

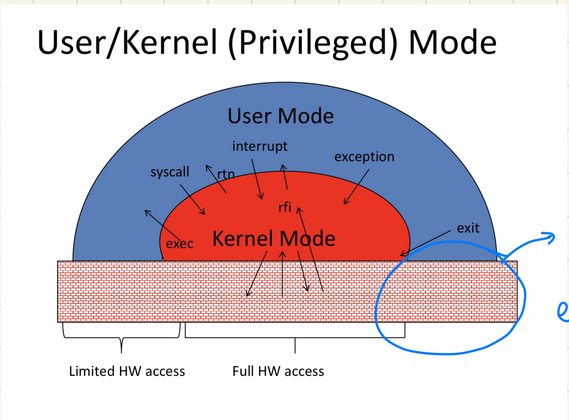
- what is os? → hw ที่หลากหลาย
  - hiding complexity
- kernel is the part ③ of the OS that running all the time on the computer
  - kernel → 1 in set of software
  - kernel → โหลดอันแรก & ต้องเสร็จก่อน → อยู่บน ram ตลอดเวลา → ยันอื่นไม่ได้ใช้ก็หายไป
  - running all the time → ถูกเรียกใช้ตลอดเวลา
  - ไฟเข้า → โหลด BIOS/UEFI, NVRAM, check hw, เลือกอุปกรณ์ที่บูท hdd, check UEFI บูธ boot loader → load boot loader → boot loader มองหา kernel → load kernel เข้า ram → เริ่ม PID ที่ 1 → ทำ start up script → OS มาช่วยลดความซับซ้อน
- core → ถ้าขาด kernel OS จะไม่สามารถทำงานได้เลย
- manage system resource → ทำหน้าที่หลักของ OS ไร → เคยเรียนมาแล้ว
- act as a bridge between application and hw → ปัจจุบัน OS ไม่ยอมให้ app คุยกับ hw โดยตรง.



application คุยได้แค่ kernel only

มี lib เรียกใช้ service ใน kernel

↓ ติดต่อไป hw



ยอมเปิดให้คุยกับ hw ได้โดยตรง mostly found on embed sys, old os

ส่วนที่ติดต่อได้โดยตรง → เรียกใช้งานได้แต่บาง function

↓

ex. กำหนดแค่การอ่านค่า only

↓

เมื่ออยากเขียนล่ะ? (หลาย ๆ คน) → แตก

One of the goal of Os is → application ที่ user เปิด

ทุก application run อยู่บน env หนึ่งที่เรียกว่า process

- ↳ Protecting process and the kernel → ป้องกัน kernel จากการโดนรุกราน
  - ↳ Running multiple program
    - ↳ keep them from interfering with Os-kernel
    - ↳ keep them from interfering with each other
- ↳ Protection Why?
  - ↳ ป้องกัน kernel จาก bug หรือความตั้งใจของ application ที่จะเข้ามาแก้ไขค่าใน kernel โดยตรง
    - ↳ เข้าถึง kernel ได้ เข้าถึงทรัพยากรได้ทั้งหมด → code ใน kernel ดีที่สุด
      - ↳ เรียกผ่าน system call → อารมณ์เหมือน api ที่ให้ app เรียกผ่านเป็นตัวกรอง
        - ↓ สามารถเช็คค่าเกิน, ค่าแปลก ← ดูความถูกต้อง
        - ↓ สิ่งที่ส่งไปเป็น service ของ kernel ต้องทำงานได้
  - ↳ ป้องกัน application
    - ↳ ไม่ให้ application ทำงานทับกัน → ทำงานในเขตตัวเอง
      - ↳ มองง่ายๆ ในแง่ security → แอปธนาคาร
        - ↳ มี virus เข้ามาขโมยข้อมูลแอปธนาคาร → แตกง่าย

กรณีเลวร้ายที่สุดคือ kernel ทำการฆ่าตัวตาย → blue screen

↳ มักเป็นโปรแกรมที่ยุ่งกับ hw เยอะๆ

- ↳ เหตุผลในนอก
  - ↳ reliability → ป้องกันไม่ให้ระบบล่ม → ไม่ว่าเกิดอะไรก็ตาม → ให้โปรแกรมล่มอย่างเดียว OS ไม่ล่ม
    - ↳ resource overload ด้วยเช่นกัน
  - ↳ security & privacy → ป้องกันการขโมยหรือทำลายข้อมูล
    - ↳ หลักการในการ design คือ trust programless → ไม่เชื่อถือโปรแกรมไม่ว่ากรณีไหน
      - ↳ ทำการรับมือหากมีโปรแกรมที่หลุดรอดออก
  - ↳ fairness → ให้ sw เข้าใช้ทรัพยากรแบบยุติธรรม → สมมุติเขียนโปรแกรมให้ใช้ cpu ตลอดเวลา
    - ↳ OS ต้องเข้าไปจัดระเบียบได้
      - ↳ จัดการคนที่เข้าไปใช้งาน

↳ Protection How? (hw/sw) → ป้องกันยังไงได้

↳ software protection → ตรวจการเข้าถึงของ process
ใช้ sw เฉยๆ โดย os → ป้องกัน kernel ผ่านการเรียก system call

↳ hardware protection → memory address translation → แต่ละ app มี memory เป็นของตัวเอง → ห้ามใช้นอกเหนือที่ os assigned
มีวงจรป้องกัน → dual mode operation → ทำงานได้หลาย mode

Dual mode operation

↳ ทำให้ระบบถูกแยกเป็น 2 ส่วน → kernel mode | code ของ kernel ทำได้ทุกอย่าง
→ user mode → application ทั่วไป | ของ user สามารถทำได้บางอย่าง

↳ ถ้าโปรแกรมจะทำงานก็จะมีการเช็คว่าโปรแกรมอยู่ mode ไหน → เพื่อให้รู้ว่าทำอะไรได้บ้าง

↳ kernel mode
↳ Execution with the full privileges of the hardware. (→ ทำทุกอย่างได้)
↳ Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet.

↳ User mode
↳ Limited privileges → ได้แค่บางเรื่อง → ต้องไปขอร้องขอ
↳ Only those granted by the OS kernel

↳ there's flag on register → เมื่อก่อนจะมีตัวแปรในระบบ (sw) เวลาทำงานทุกครั้งก็ต้องเช็คเงื่อนไข if → ต้องไปเช็คว่าคำสั่งนี้ใช้ได้ไหม → เช็ค sw → ใช้ cpu ประมวลผล → overhead → ทำงานช้าลง
↳ เดี๋ยวนี้ hw จัดการให้ → ถ้ารัน sw ก็เข้ามาเปลี่ยน flag
↳ ตั้งการรวมเช็คว่าอะไรทำได้ทำไม่ได้
↳ เวลาแยกว่า จากการ decode ชุดคำสั่ง → รู้ mode ได้

(H/W)

↳ cpu มองเห็นข้อมูลจาก register flag แล้วจะ support อะไรบ้าง
↳ privileged instructions → คำสั่งบางคำสั่งที่สามารถทำงานใน kernel เท่านั้น → code application ไม่สามารถ compile ได้ privileged instru
↳ hw ทำการเช็ค → ไง → ถูกต้อง → มีการแบ่งกลุ่มคำสั่ง
↳ available to kernel
↳ Not available to user code

↳ Limits on memory accesses → การทำ memory address translation → ที่ทำการ → ตอนนี้ hw ทำได้แทน sw
↳ to prevent user code from overwriting the kernel

↳ Timer → ใช้ในการเปลี่ยน mode → เช่น kernel ให้ program A ทำงาน 10 ms (usermode)

↓ พอครบเวลาการทำงาน 10 ms ↓ time interupt → mode กลายเป็น kernel mode → เพื่อมาจัดสรรเวลาใหม่

↳ to regain control from a user program in loop

user → kernel (hw: interupt)

kernel → user (sw: tricker)

ถ้าแก้ไข compiler พิเศษ → ได้ priviledge instruction → แต่เราเป็น usermode → รันไม่ได้

↓ โดน hw เช็ค EFLAG กับชุดคำสั่ง เกิด exception

↙ ถ้า os ไม่ลองรับ error ↓ จอฟ้า

Virtual Machine (VM)

↳ Software emulation of an abstrac machine

↳ give program illusion they own the machine

↳ Make it look like HW has feature you want.

↳ ทำให้โปรแกรมมองเห็น hw อย่างที่ต้องการ

process ก็คือ vm ↓ ปัจจุบัน → system vm เป็นส่วนเสริม ↓ process vm ↓ คนไม่ค่อยพูด

↳ 2 type of VM

↳ Process VM → ทำให้ hw ที่ application มองเห็นเหมือนกันหมด → เสนอแค่ algorithm ในโปรแกรม

↳ Support the execution of single program (one of the basic function of the OS)

↳ System VM → มองไปที่ทั้ง os → ความยากคือเหมือนมี [user / kernel / hw] รันบน user mode อีกที

↳ Support the execution of an entire OS and its applications.

↳ Process VM

↳ goal

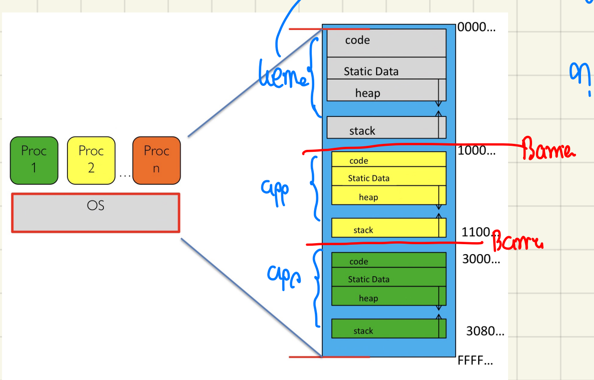
↳ provide an isolation to a program → 1 program = 1 process

falut tolerance ← A พัง B ไม่พัง

boundary memory ← มองเห็นของใครของมัน → ดวงกันไม่ได้ ↓ ไม่มีใครมาแก้ของเรา

↳ portability → hw มี spec หลากหลาย → process เตรียม env ไว้ให้ application แบบเดียวกันหมด บน os แบบเดียวกันนะ

↓ เอา target แค่ว่าจะเอาไปรันบน os ไหนพอ



ทุกอย่างอยู่ใน memory เดียวกัน → มีการสร้าง barrier ↓ การเปรียบเทียบ

memory translation → จะ translate ออกมาแล้ว เกินขอบเขตหรือไม่

Process Abstraction

↳ Process: an instance of a program, running with limited rights.

↳ Thread: a sequence of instructions with in a process

↳ Potentially many threads per process (for now 1:1)

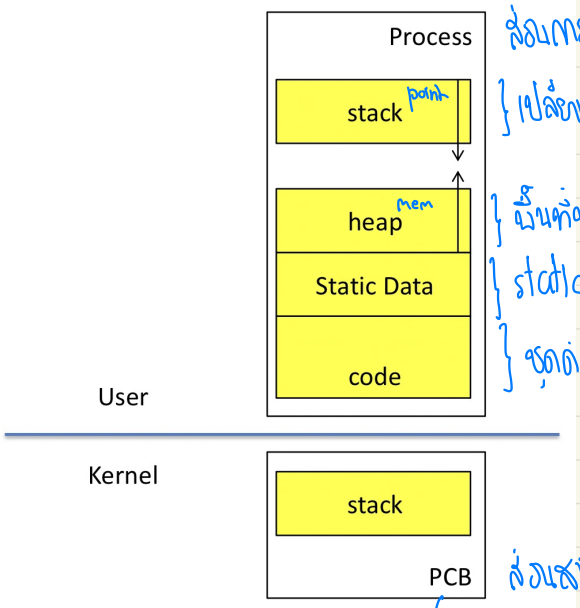
↳ Address space: set of rights of a process

↳ Memory that the process can access

↳ Other permissions the process has (which system call it can make, which file can access)

↳ anatomy

- 2 parts
  - PCB in kernel
  - Others in user



Process Control Block (PCB)

↳ Kernel represents each process as a process control block (PCB)

↳ status (running, ready, blocked, ...)

↳ Register, SP, ... (when not running)

↳ Process ID (PID), User, Executable, Priority, ...

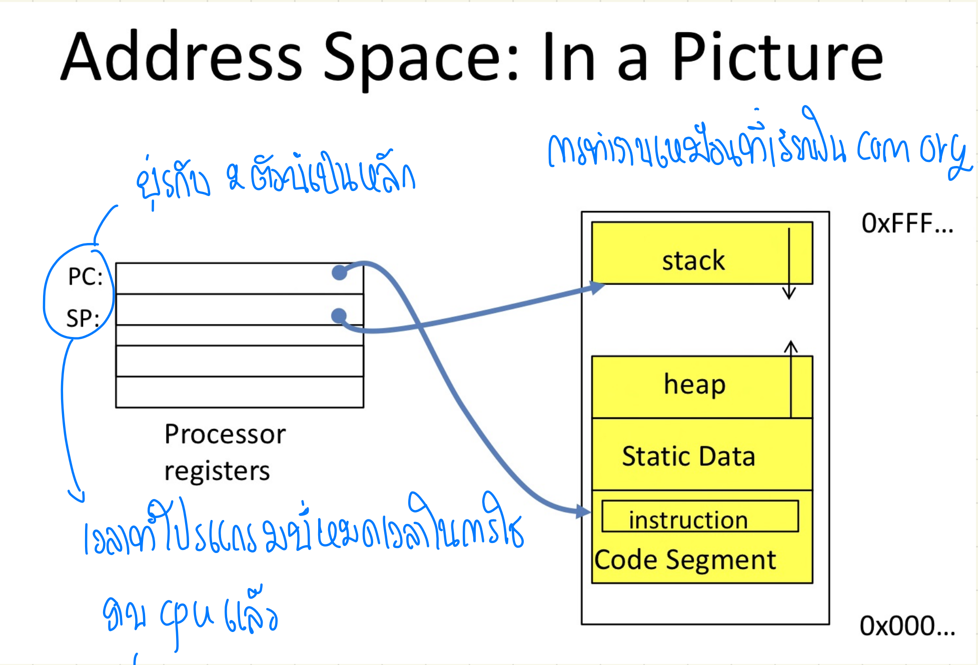
↳ Execution Time, ...

↳ Memory space, translation tables.

↳ ทำไมทั้งหมดอยู่ใน kernel → ป้องกันอักเซ็นแน่ะ → กันการเปลี่ยนแปลงพฤติกรรม process จากความผิดปกติ

↳ Kernel Scheduler maintains a data structure containing the PCBs → การเปลี่ยนเลย
save ↔ load

↳ Scheduling algorithm select the next one to run. → ค่อยเรียน



ค่าของ 2 reg นี้จะเป็นค่าที่ต้องเก็บก่อน
↳ at least

Main point

↳ Process concept
- ↳ A process is the OS abstraction for executing a program with limited privileges

↳ Dual-mode operation: user vs kernel
- ↳ kernel-mode: execute with complete privileges
- ↳ User-mode: execute with fewer privileges

↳ Safe control transfer
- ↳ How do we switch from one mode to the other? → user ↔ kerenel

Mode Switch → การเปลี่ยน mode user ↔ kernel

↳ From user mode to kernel mode ด้านล่างคือเกิดอะไรได้บ้าง

↳ Interrupts

↳ Trigger by timer and I/O devices

↳ Exceptions → การทำงานที่แปลก ๆ kernel raise exception → เหมือน interrupt

↳ Triggered by unexpected program behavior, malicious behavior

ถ้าไม่ switch เข้าถึงอะไรไม่ได้ ↑ (ใน ส่วน kernel, user ↑ ที่ process มี 2 ส่วน

↳ System calls (aka protected procedure call) → ใช้โปรแกรมเลยต้องการใช้ hw เรียกผ่าน sys lib (call)

↳ Request by program for kernel to do some operation on its behalf / user ทำไม่ได้

↳ Only limited # of very carefully coded entry points. → มีจุดแค่ที่นี้

(ถ้าใช้ hw → อยู่ใน kernel mode ↓ ตัวเปลี่ยน user → kernel เมื่อไร

→ เป็นการเข้าไปใช้ resource

↳ From kernel mode to user mode

↳ New processes/new thread start → การสร้าง process เสร็จ → สร้างใน kernel mode → กลับไปใช้งาน process ที่ user mode ต่อ

↳ Jump to first instruction in program/thread ↳ รู้ว่ากลับไป user & run code

↳ Return from interrupt, exception, system call → จบเรื่องนี้แล้ว → ถ้าจบจากนี้ → เปลี่ยนกลับไปทำงานต่อ

↳ Resume suspended execution → เริ่มต้นจากการ interrupt

↳ Process/thread context switch → การสลับให้ process ที่กำลังใช้งาน microprocessor ในปัจจุบันถอยออกมา แล้วให้ตัวใหม่เข้าไปใช้งานแทน

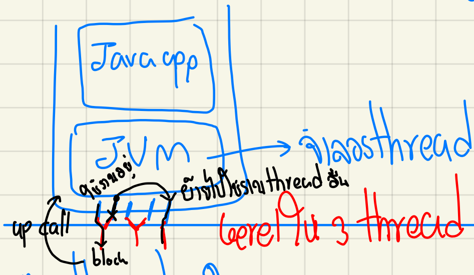
↳ Resume some other process ↳ ตอนเปลี่ยน (switch) เลยต้องเกิดขึ้นใน kernel mode

↳ User-level callup (UNIX signal) → การเรียก (call)

↳ Asynchronous notification to user program → ตัว kernel เลยต้องการที่จะส่งสัญญาณอะไรสักอย่างขึ้นไปให้โปรแกรมที่อยู่บน user mode

reverse of system call → app เรียก kernel

↳ อันนี้ kernel ต้องการส่งข้อมูลไป process ที่ user mode



→ จำลอง thread → เหมือนว่าเป็น single thread → มี 1 ตัว app ใช้ได้ 1 thread only (most cpu)

↳ ตัว java สามารถ multithread ได้

↳ แต่ระบบเป็น single thread → JVM เลยมาช่วย จำลองใน app เล่น thread ตามต้องการ

ถ้าระบบรองรับได้ ของระบบ

เมื่อมี thread ที่ใช้งานร่วมกัน มีการ block อีก thread อยู่ เลยจะใช้ upcall เพื่อให้ thread ที่รอ

→ ถ้าเราต้องไปเขียน OS

Implementing Safe Kernel Mode Transfers → ต้องระวังเรื่องระหว่างการ switch mode

- Carefully constructed kernel code pack up the user process state and set it aside.
- Must handle weird/buggy/malicious user state
  - Syscalls with null pointer. → เผลอส่ง parameter ที่ผิดเข้ามา
  - Return instruction out of bounds → เผลอทำ pointer เพี้ยนชี้ไปเรื่อยเปื่อย
  - User stack pointer out of bounds

  } ป้องกันไม่ให้เกิดการพัง ขณะที่ผิดปกติ
- Should be impossible for buggy or malicious user program to cause the kernel corrupt itself
- User program should not know interrupt has occured (transparency) → ห้าม user เข้าถึงได้ตอน detect การ interrupt

Device Interrupts → *

- OS kernel needs to communicate with physical devices. → เกิดจาก hw →
- Device operate asynchronously from the CPU. → ทำงานระดับ hw
  - Polling : kernel wait until I/O is done → ไป monitor ตลอดการ copy ไว้เท่านั้น
  - Interrupts : kernel can do other work in the meantime → ไปบอก I/O ตัวอื่นให้ copy file → cpu ทำงานอื่นไป → รอการ interrupt → ตอนย้ายข้อมูล
- Device access to memory.
  - Programmed I/O : cpu read and write to device
  - Direct memory access (DMA) by device
  - Buffer descriptor : sequence of DMA's
    - eg. packet header and packet body
  - Queue of buffer descriptors
    - Buffer descriptor itself is DMA'ed
- How do device interrupts work? → hw ส่งสัญญาณ ส่งไปที่ตัว interrupt (ใน cpu) → มีหมายเลขของสัญญาณ
  - Where does the cpu run after interrupts.
  - What stack does it use?
  - Is the work the CPU had been doing before the interrupt lost forever?
  - If not, how does the CPU know how to resume that work?

หลังจากที่เกิด interrupt จะทำไง

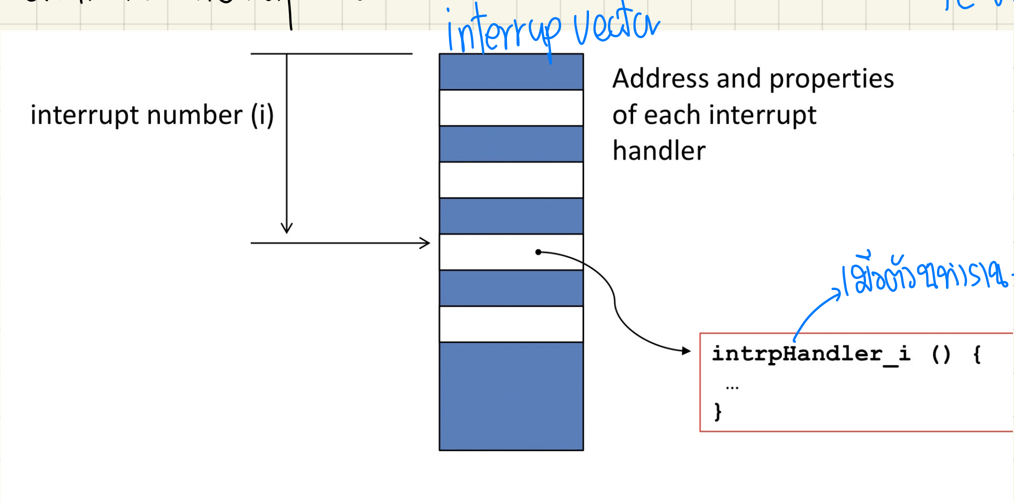
How do we interrupts safely?

- ↳ Interrupt vector → array pointer → ชี้ไปที่ routine (interrupt) ที่เกี่ยวข้องกับการ interrupt → ดูสัญญาณว่าชี้ไปที่ช่องไหนตรง
  - ↳ Limited number of entry point into kernel
- ↳ kernel interrupt stack → interrupt routine = function → ต้องการพื้นที่เก็บข้อมูล
  - ↳ Handle work regardless of state of user code
- ↳ Interrupt masking → interrupt ไหนที่สามารถ interrupt ซ้อนได้ → ขึ้นอยู่กับ priority
  - ↳ Handler is non-blocking
- ↳ Atomic transfer of control → มีชุดคำสั่งเฉพาะการ copy → ลักษณะคำสั่งแบบนี้คือ คำสั่งเดียวทำได้หมด
  - ↳ "atomic transfer of control"
  - ↳ "Single instruction" like to change
    - ↳ Program counter
    - ↳ Stack pointer
    - ↳ Memory protection
    - ↳ kernel/user mode
  - เหมือนที่เก็บใน PCB → run 1 command only
- ↳ Transparent restartable execution → เมื่อหมดการ interrupt โปรแกรมต้องรันต่อได้แบบเดิม
  - ↳ User program doesn't know that interrupt occured.
  - ↳ โปรแกรมต้องไม่รู้

Where do transfer mode go?

- ↳ Solution: interrupt vector

PC ชี้บอกว่า cpu fetch instruction (เข้าไป 1 อัน)



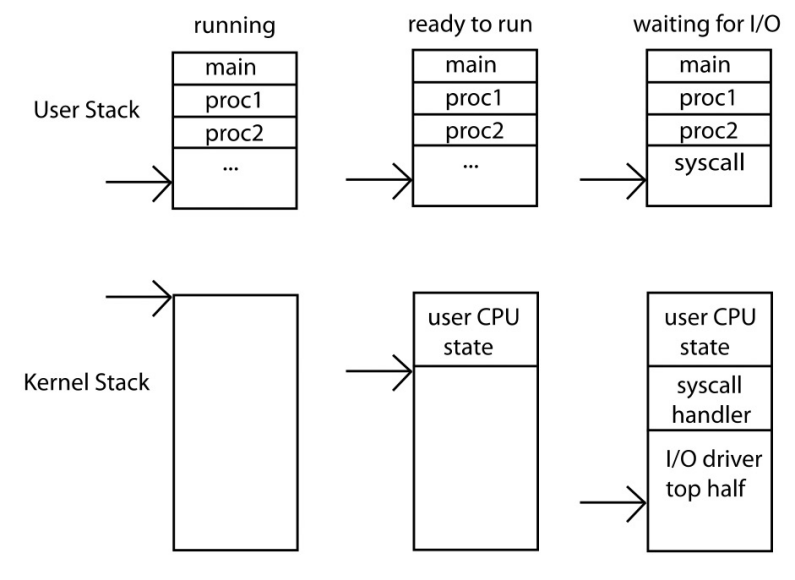
เมื่อตัวจัดการ → มีการประกาศ function, var
- มักใช้งาน stack
- ทำงานเป็น kernel mode ← interrupt
- เข้าถึงได้ทุกทรัพยากร
- why? ต้องสร้าง stack ไว้ใน kernel ในเมื่อสามารถสร้างไว้ใน user ได้

The kernel stack → ชั้นของprocessที่อยู่ในkernel → stack PCB

↳ Interrupt handlers want a stack
↳ System call handlers want a stack } ทุกอันต้องการ stack

↳ Can't just use the user stack... why? → ป้องกันและของ
→ กันมิจฉาชีพล่วงรู้ user stack → สามารถ detect interrupt
แก้ user stack → การชี้ไปที่ code อื่น
point to virus

↳ Solution: two-stack model
  ↳ Each os thread has kernel stack (located in kernel memory) plus user stack (located in user memory)
  ↳ Place to save user registers during interrupt

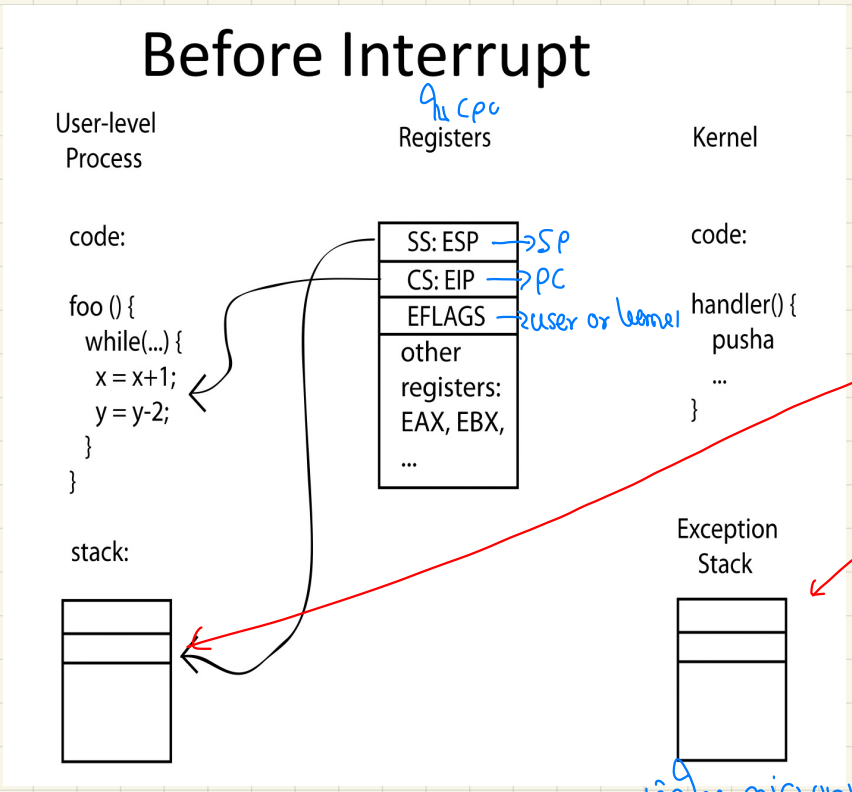


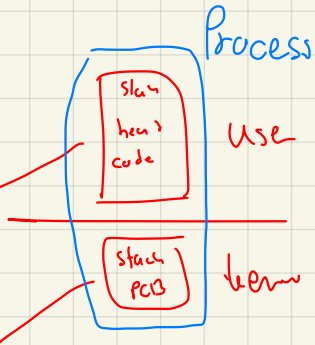
Interrupt stack
↳ Pre-processor, located in kernel (not user) memory.
  ↳ Usually process/thread has both kernel and user stack
↳ Why can't the interrupt run on the stack of the interrupted user process?

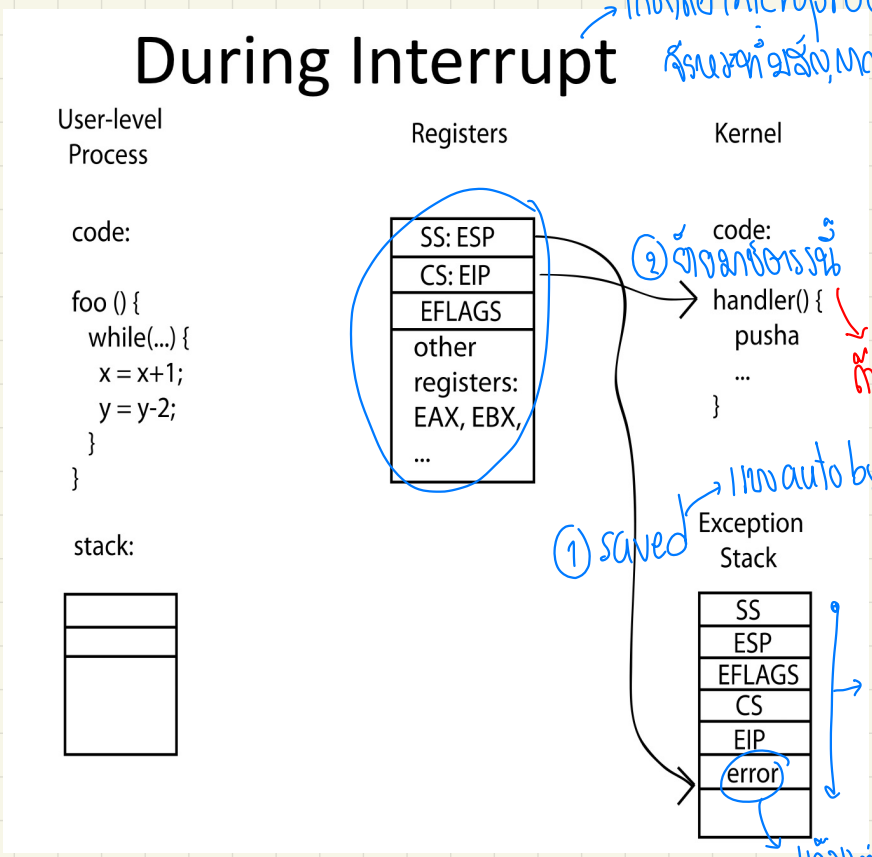
Case Study x86 interrupt
↳ Save current stack pointer
  Save current program pointer
  Save current processor status word (condition code)
  Switch to kernel stack; put SP, PC, PSW on stack
  Switch to kernel mode

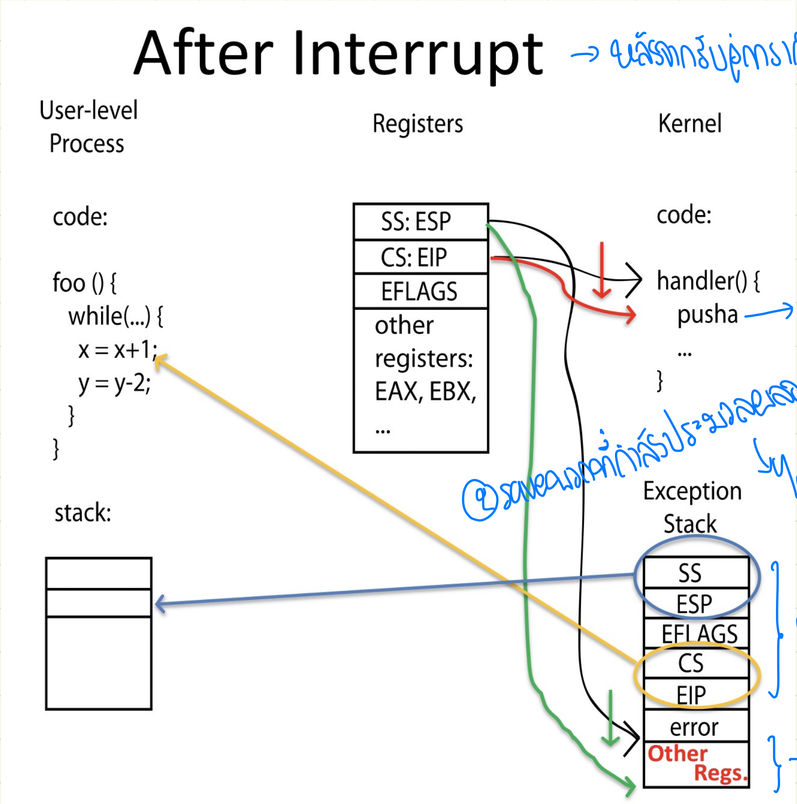
Vector through interrupt table

Interrupt handler saves registers it might clobber









→ หลังจากรับรู้การเกิด interrupt → ไปเก็บแล้วทำงานของ interrupt

① สั่ง saved ค่าที่ทำงานอยู่ ตัว interrupt เป็นคนสั่ง func

② save ลงที่ทำงานประมวลผลอยู่

ไม่ต้องเขียนตัว save จากตัว → เปลืองเวลา

ตอนแรกเก็บแค่ pointer → ยังไม่มี data ที่ค้างอยู่

ตัว interrupt handler เขียนเอง

→ เขียน code saved เองเพื่อจะได้ไม่เปลืองเวลา

save เท่าที่จำเป็นเพื่อประสิทธิภาพ

At the end of handler → ต้องทำด้วยไม่งั้นมีปัญหา

- handler restore saved registers → จบตัว handler ก็จะทำการดึงค่ากลับสู่ที่เก่า ①
- Automatically return to interrupted processor/thread
  - Restore program counter,
  - Restore stack,
  - Restore processor status word / condition code,
  - Switch to user mode,

② ชุดนี้ต้องทำไปพร้อมกัน โดยอยู่เรื่องใน

มี instruction พิเศษ ใช้ 1 inst ทำได้ 4 ตัว

"atomic instruction"

→ ทำไมไม่เขียน 4 instruction → ทำไมต้องใช้ atomic instruction

→ กัน process อื่นมารบกวน / เขียนทำก่อนทำเสร็จ

→ เช่นการ interrupt เพิ่มเข้ามา → ①→② inter ③-④

→ ระบบเละ ทำงานปนกัน   S = ยุ่งเหยิง

→ ทำให้ต้อง design ให้ครอบคลุมทุกเคส

Interrupt masking → การปิดสัญญาณ interrupt

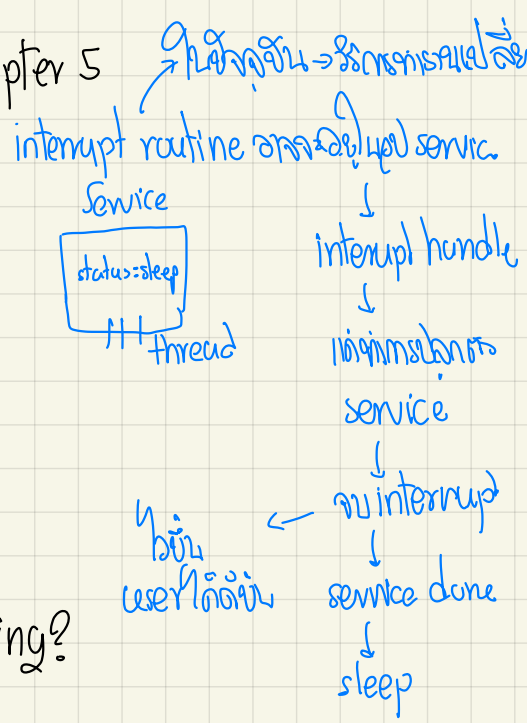
- Interrupt handler run with interrupts off
  - Re-enable when interrupt completes → ค่อยเปิด
- OS-kernel can also turn interrupts off
  - eg. when determining the next processor/thread to run → ตอนที่มีการเรียกแบบนี้

ปิดรับ interrupt ชั่วคราว (priority สูงกว่า)

  - On x86 → CLI : disable interrupt
    STI : enable interrupt
    Only applies to current CPU (on a multicore)

เกิด int → ปิดรับ int → จบ
→ เปิดรับ int
เมื่อทำงานเสร็จในการเรียก

↳ We will need this to implement synchronization in chapter 5



Hardware support: Interrupt Control

↳ Interrupt processing not visible to the user process:
  ↳ Occurs between instructions, restart transparently
  ↳ No change to process state
  ↳ What can be observed even with perfect interrupt processing?

↳ Interrupt Handler invoked with interrupts 'disabled'
  ↳ Re-enable upon completion
  ↳ Non-blocking (run to completion, no waits)
  ↳ Pade up in a queue and pass off to an OS thread for hard work
    ↳ wake up an existing OS thread

↳ OS kernel may enable/disable interrupts → OS ควบคุม interrupt masking
  ↳ On x86: CLI (disable interrupts), STI (enable)
  ↳ Atomic section when select next process/thread to run
  ↳ Atomic return from interrupt or syscall

↳ HW may have multiple levels of interrupts
  ↳ Mask off (disable) certain interrupts, eg, lower priority.
  ↳ Certain Non-maskable-Interrupts (NMI)
    ↳ eg. kernel segmentation fault
    ↳ Also: Power about to fail!

Kernel System Call Handler → การทำงานคล้าย interrupt แต่มาจาก s.w → มีการเรียกใช้ function ที่เป็น sys call

↳ กระบวนการรับส่ง parameter

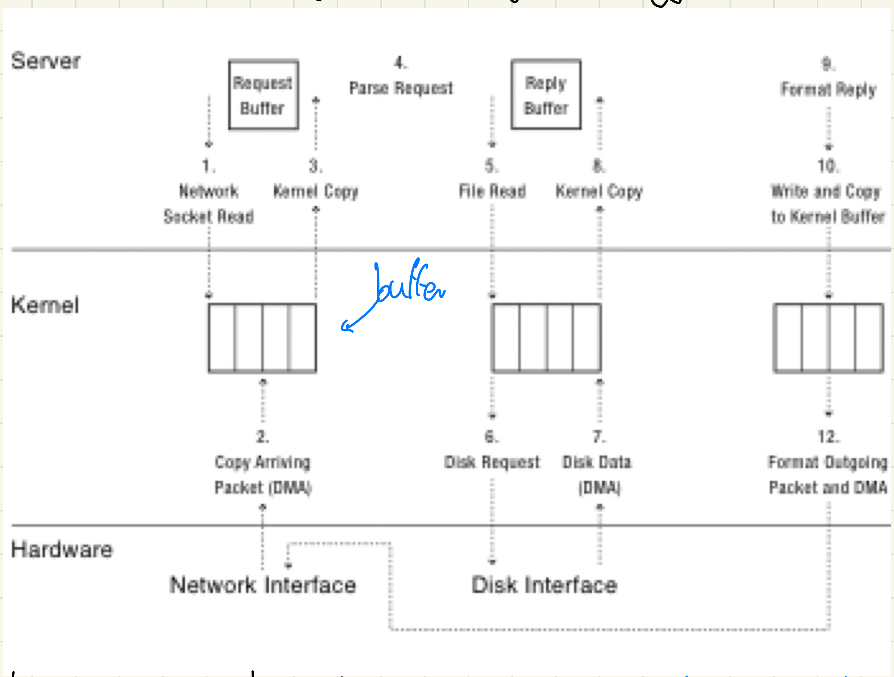
↳ Vector through well-defined syscall entry points!
  ↳ Table mapping system call number to handler

↳ Locate arguments
  ↳ In registers or on user (!) stack → หา parameter

↳ Copy arguments → copy parameter to kernel stack → กลัวไม่ validate → แก้ตัว ↓ kernel ที่เขียนไว้ ↓ แตก
  ↳ From user memory into kernel memory - carefully checking locations!
  ↳ Protect kernel from malicious code evading checks → กันการแก้ไขเพราะมีบางอันเป็น kernel mode

↳ Validate arguments → ดูว่ามันปลอดภัยรึเปล่า → หลังจากที่ทำการเช็ค parameter ต้อง validate แล้ว
  ↳ Protect kernel from errors in user code
  ↳ ถ้า copy มา → แก้ไขยาก

↳ Copy results back → ได้ผลลัพธ์ → copy back
  ↳ Into user memory - carefully checking locations!



ถูกเปลี่ยนค่าได้ ↑ สิ่งที่ถูก copy ไปใช้ที่อื่น ↑ ถ้าไม่ต้อง copy ไป-มา ↳ security → ต้องการให้ user อาจถูกเปลี่ยนค่า ระหว่างทาง ↓ ทางแก้ต้องจับ stack 2 อัน

Today: Four Fundamental OS Concepts สรุปสิ่งที่เรียนมา: what we have talk about

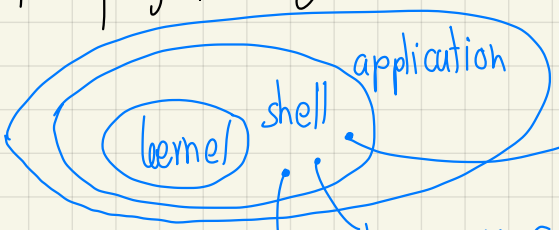
↳ Thread: Execution Context → การ execute
  ↳ Program Counter, Register, Execution Flags, Stack

↳ Address space (with translation) → การ allocate
  ↳ Program's view of memory is distinct from physical machine.

↳ Process: an instance of a running program → vm
  ↳ Address Space + One or more Threads → cpu

- Dual mode operation / Protection
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other.

The programming interface

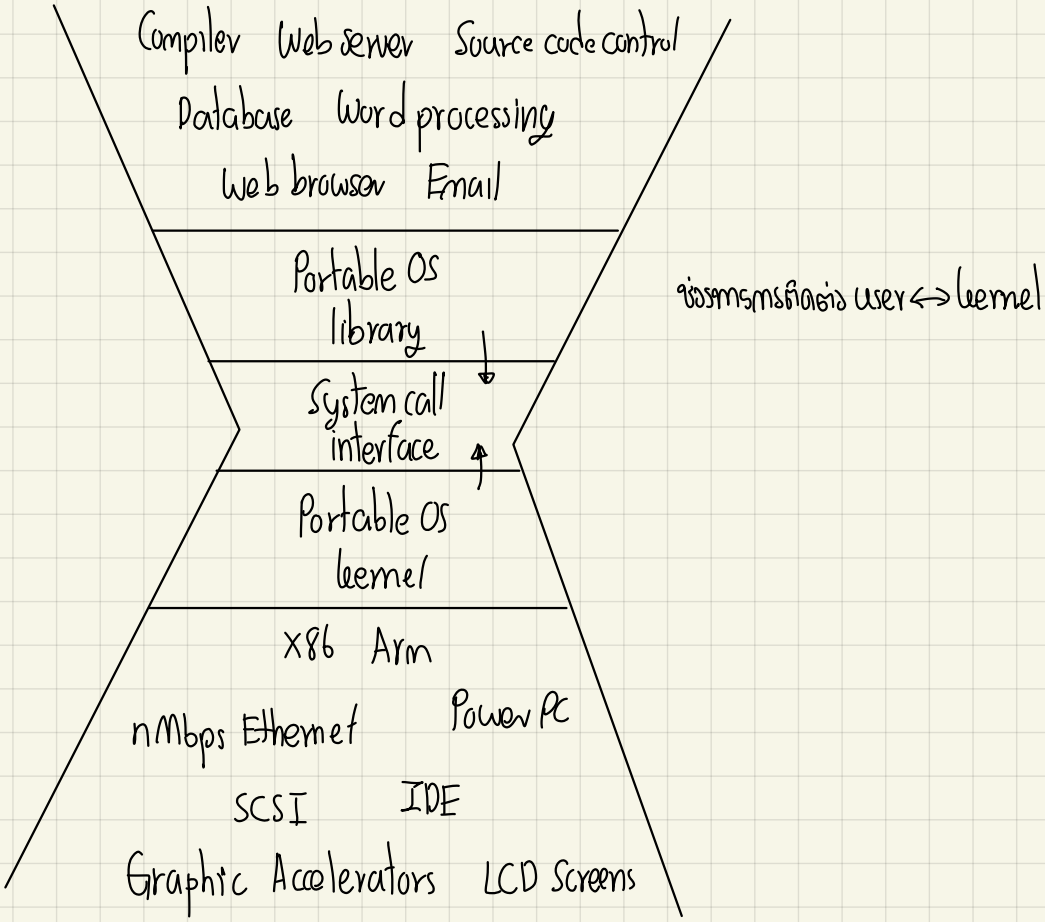


→ shell คือตัวแปลรหัสที่ช่วยให้ผู้ใช้ร้องขอเข้าถึง os ได้

ผ่าน CLI, GUI → แล้วแต่ระบบปฏิบัติการ

ไม่ได้เชื่อมกับ kernel โดยตรง → เชื่อมผ่าน kernel api → system call

↳ การทำงานที่ประกอบด้วยการร้องขอข้อมูล



ช่องทางการสื่อสาร user ↔ kernel

Main point

- Create and managing processes → การสร้างและจัดการ process → เช่น window, unix
  - fork, exec, wait
- Performing I/O → การติดต่อกับอุปกรณ์
  - open, read, write, close
- Communicating between processes → การสื่อสารระหว่าง process
  - pipe, dup, select, connect
- Example: implementing a shell → example ของ function ขั้นพื้นฐานของ shell

Shell

- A shell is a job control system → ควบคุมสั่งการการรัน application (หรือเปิด)
  - Allows programmer to create and manage a set of program to do a task
  - Window, MacOS, Linux all have shells → มีทุกระบบปฏิบัติการ
- Examples to compile a c program
  - cc -c sourceFile1.c
    cc -c sourceFile2.c
    ln -o program sourceFile1.o sourceFile2.o

  } กรณีเรียกใช้งาน c compiler
  ↓
  พอส่งผ่าน shell ไปแล้วมันทำงานอย่างไร
  ↓ มันต้องสร้าง process ใช่

Question

- If the shell run at the user level, what system call does it make to run each of the program?

ตัว shell ไม่ได้เป็นคนสร้าง process
↓
เป็นคนส่งต่อคำสั่งที่เราสั่ง
↓
เราสั่งอะไรไป → shell ไปดูว่าเราจะใช้ sys call ตัวไหน

↓
ต้องใช้ privillege intstruction จะ:
↳ รันบน user mode ไม่
↳ ใช้ system call ไร?

Window CreatProcess →

- System call to creat a new process to run a program
  - Create and initialize the process control block (PCB) in the kernel → สร้าง PCB
  - Create and initialize a new address space → ของ addr space
  - load the program into the address space → โหลดโปรแกรมเข้าไปใส่ addr space
  - Copy argument into memory in the address space → copy arg ไปใส่ mem ของ addr
  - Initialize the hardware context to start execution at "start" → ชี้ไปที่จุดเริ่มต้นของโปรแกรม
    - ↓ ตัว program counter, และอื่นๆ
  - Inform the scheduler that the new process is ready to run
    - ↓ ส่ง signal ไปหา scheduler ว่า process นี้พร้อมที่จะทำงานแล้ว
    - ↓ จาก init → ready

เมื่อเราสั่งรันโปรแกรมๆจะไม่ถูกสั่งรันโดยการสร้าง process → จบที่ set status to ready

↙ โปรแกรมจะทำงานเมื่อไหร่ขึ้นอยู่กับ scheduler

Windows CreatProcess API (simplified)

→ หน้าที่จัดการคนที่จะสามารถเข้าไปใช้งาน cpu
↳ ตามเงื่อนไขเลย

ไว้เจาะรายละเอียดทีหลัง

```
if(!CreateProcess(
    NULL,        // No module name(use command line)
    argv[1],     // Command line
    NULL,        // Process handle not inheritable
    NULL,        // Thread handle not inheritable
    FALSE,       // Set handle inheritance to FALSE
    0,           // No creation flags
    NULL,        // Use parent's environment block
    NULL,        // Use parent's starting directory
    &si,         // Pointer to STARTUPINFO structure
    &pi,)        // Pointer to PROCESS_INFORMATION structure
)
```

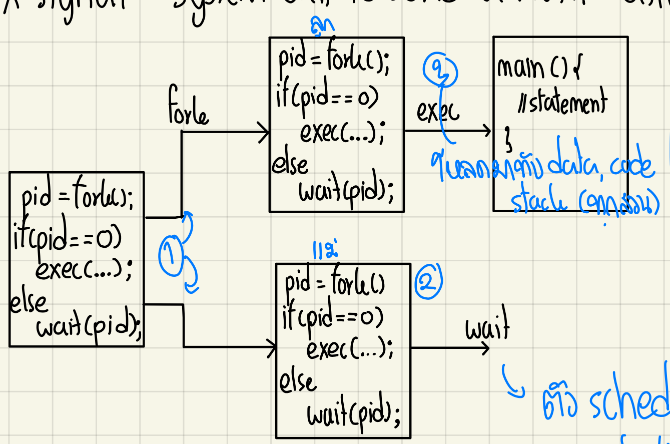
(Command line) → ขั้นตอนที่หมดตามบน ดูค่าเดียว 1 คำสั่ง (บน window)
↳ อยากรันโปรแกรมไหนเรียก CreatProcess

ตัด parameter หลายตัวออกไป
↳ version ย่อๆ

Unix Process Management

→ แตกต่างจาก window → มี sys call ที่ต้องรู้ ≈ 4 ตัว

- UNIX fork – system call to create a copy of the current process, and start it running.
  - No arguments? → ทำ clone ตัวเอง → shell clone ตัวเองออกมา → มี code เหมือนกัน
- UNIX exec – system call to change the program being run by the current process.
  - ↳ ตัวลูกเรียกใช้ sys call ↳ ใหม่ที่โปรแกรม เข้ามาแทนที่โปรแกรม shell
- UNIX wait – system call to wait for a process to finish
  - ↳ แม่เรียกใช้การรอ → ให้ลูกจบการทำงาน
- UNIX signal – system call to send a notification to another process



ปกติแต่ละ process จะมี PID ต่างกัน
↳ PID (แม่) → = 0
PID (ลูก) → ≠ 0

เกิด signal สั่งให้แม่เปลี่ยนสถานะกลับเป็น ready
↳ อาจจะให้ scheduler ไม่ชัวร์ → หาใหม่เลย

↳ ตัว scheduler จะไม่สนใจ process ที่มีสถานะ wait
↳ ไม่ถูกจัดสรรเวลาให้

ขึ้นตอนดูว่าบางอย่าง window นั้นดูเหมือนกัน

Question: What does this code print?

```
int child_pid = fork();   → เกิด process แม่, ลูก ที่เหมือนกัน
if (child_pid == 0) { // in child process
    printf("I'm process #%d\n", getPID()); // ไปทำงาน
    return 0;
} else { // in parent process
    printf("I'm parent of process #%d\n", child_pid); //wait
    return 0;
}
```

return ค่า pid ของลูกออกมา

↳ return pid = 0 ออกมา

ใช้ค่าตัวแปรในการเช็คการทำงานอยู่ที่ไหน

ทำงานเหมือน concept ด้านบน

ค่าจาก window คือ parameter นี้เอง

Question

↳ Can UNIX fork() return an error? why? → clone ไม่สำเร็จ = error → โอกาสที่ clone ไม่สำเร็จมีไหม → แล้วเจออะไร

↳ มี → กรณี memory ไม่พอ

↳ สิ้นเปลือง process → ใช้ addr space

↳ Can UNIX exec() return an error? why? → load fail = error

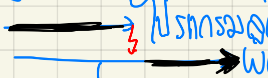
↓ ไม่มีโปรแกรมนี้อยู่, ไม่มีโปรแกรมนี้จริง, ...

เรียกชื่อผิด

↳ Can UNIX wait() ever return immediately? Why?

↓ เรียกแล้วไม่รอ = error

หรือ



อาจจะเกิดการ clone แล้วเกิด error → scheduler เลือกลูก

↳ แตกไปหมดแล้ว

เริ่มมาทำงานก่อน

เสร็จจบไปหมดแล้ว

อาจจะ-รอการทำงานของ if เลยไม่รอ

↳ พอ wait ทำงานก็ไม่ทำงานเพราะ child จบไปแล้ว

return ทันที

Implementing UNIX fork

↳ Step to implement UNIX fork

↳ Create and initialize the process control block (PCB) in the kernel

↳ Create a new address space

↳ Initialize the address space with a copy of the entire contents of the address space of the parent

↳ Inherit the execution context of the parent (e.g any open files) → ของ I/O, PC

↳ Inform the scheduler that the new process is ready to run

↳ ที่เหลือขึ้นกับตาม scheduler

ของ window → ใหม่ → initialize ค่า

linux → fork → copy initialize

↑ ไม่ต้องใส่ parameter ไปเพราะ มันใช้การ copy แทน

Implementing UNIX exec

- Steps to implement UNIX fork
  - load the program into the current address space → โหลดโปรแกรมเข้ามาทั้ง
  - Copy arguments into memory in the address space → ก็อป argument
  - Initialize the hardware context to start execution at "start" → set pc to start

UNIX I/O → หลักการ input, output → มาจากยุค unix → เป็นต้นแบบการจัดการ → เกิดก่อน → อยู่จน evol

- Uniformity → I/O ทุกชนิดถูกมองเป็นแบบเดียวกันหมด → ใช้ system call ชุดเดียวกัน
  - All operations on all files, devices use the same set of system calls: open, close, read, write
- ① Open before use → เปิด ↘ มองแบบเขียนอ่านไฟล์ → keyboard เป็นไฟล์ไฟล์หนึ่ง → อ่านการกดปุ่มจาก
  - Open returns a handle (file descriptor) for using in later call on the file ↘ pointer ชี้ไปยัง resource ที่ต้องการ
- ② Byte-oriented → อ่าน เขียนเป็น byte
- ③ Kernel-buffered read/write → ตอนเขียน/อ่าน kernel เตรียม memory ไว้จุดหนึ่ง → ที่พักข้อมูล
  - ↳ ถ้าข้อมูลล่าว ↘ ให้โปรแกรมอ่าน → ดีเลย → รอ ↓ ข้อมูล
  - เขียนไปโปรแกรมต้องรอรับ ↘ กับ I/O นะ ↘ จาก network
- ④ Explicit close → คืนส่วนที่เป็น buffer
  - To garbage collect the open file descriptor

UNIX File System Interface

system call openfile

- UNIX file open is a Swiss Army knife: → ขึ้นอยู่กับ parameter ที่ส่งเข้าไป
  - Open the file, return file descriptor
  - Options: if file doesn't exist, return an error
    - if file doesn't exist, create file and open it
    - if file does exist, return an error
    - if file does exist, open file
    - if file exist but isn't empty, nix it then open
    - if file exist but isn't empty, return an error
    - ...

Interface Design Question

- Why not separate syscalls for open/create/exist? → ทำไมไม่แยกคำสั่งเป็นหลายๆชุด
  - ↳ fopen(_) ทำได้ทุกอย่างเลย
  - if (! exists(name))

```
if (! exists(name))
        create(name);   // can create fail
fd = open(name);        // does the file exist
```

ประมาณนี้ → ทำไมไม่ทำแบบนี้

- ถ้าแยกเป็น 3 อันมีโอกาส เกิด gap ขึ้นมาได้
- เพื่อน, เรา → เช็ค exist → ไม่เจอ → สร้าง → สร้างซ้อนกัน!

Implementing Shell

ส่วนของ code

```
char *prog, **args;
int child_pid;

// read and phase the input a line at a time
while(readAndPhaseCmdLine(&prog, &args)){
    child_pid = fork();
    if(child_pid == 0){
        exec(prog, args); // child process, run the program
        // not reach
    } else {
        wait(child_pid); // wait for child
        return 0;
    }
}
```

In UNIX shell

- A program can be a file of commands → เอาคำสั่ง shell มาเรียงแบบ command ก็เป็นโปรแกรมได้
- A program can send its output to a file → เขียนลงไฟล์
- A program can read its input from a file → อ่านจากไฟล์
- The output of one program can be the input to anothe program → ส่งไปยัง program อื่น

Interprocess Communication (ของ unix) → การส่งข้อมูลกันระหว่าง process → output ของ A ไปเป็น input ของ B

↳ Producer-consumer → โปรแกรมหนึ่งทำหน้าที่เป็นผู้สร้างข้อมูลออกมา อีกโปรแกรมเป็นคนเรียกใช้

↳ Output of one program is accepted as input of another program.

↳ One-way communication → การสื่อสารทางเดียว → Producer ส่ง, consumer รับ only

↳ Pipe → จาก | → เหมือนที่เรา gen word list → เอาไปเข้า hashcat

↳ Client-server

↳ two-way communication → 2 ทาง server ↔ client

↳ Server implements specialize task

↳ Print server → เมื่อก่อน printer แพร่หลาย

producer-consumer, client-server } → 2 pg ต้องทำงานพร้อมกัน เดียวกันในเวลาเดียวกัน → ไม่จำเป็น

↳ File system → ระบบฝากไฟล์

↳ Write data to file then read file as an input

↳ Reader and writer are not need to running at the same time ผู้รับ, ส่ง ไม่จำเป็นต้องรันพร้อมกัน

ตัวไหนไม่จำเป็น → เอาออก ยกเว้นส่วน

ตัวไหนติดต่อกันบ่อยๆ เอามาไว้ใน kernel

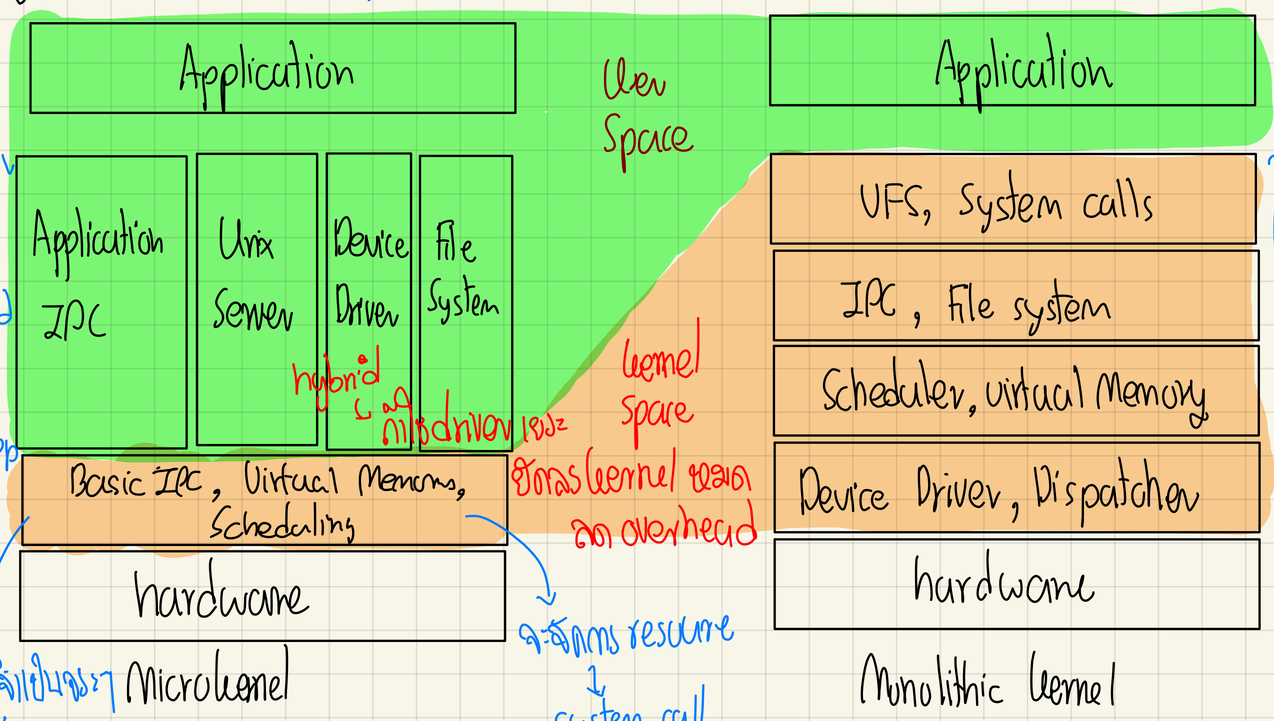
Operating System Structure → โครงสร้าง ลักษณะของ kernel

↳ Monolithic kernel → function ทุกอย่างอยู่ใน kernel — ของอัดอยู่ด้วยกัน re compile ทั้งก้อน

↳ Microkernel → เอาบางส่วนย้ายไปไว้ใน user space → upgrade ง่ายกว่า

↳ hybrid kernel → ไม่มีรูปแบบที่แน่นอน ตามความต้องการ → เอาข้อดีทั้ง 2 อันมารวมกัน

customize ได้ง่าย ↑ รูปแบบที่ต่างกัน



อันไหนอยากรันใน user ↓ เลือก load ได้ ↓ รันเร็ว app

kernel ใหญ่ขึ้น ↑ ทำงานใน kernel space ↓ no syscall ↓ no overhead ↓ เร็ว

ส่วนที่จำเป็นจริงๆ Microkernel ↓ การจัดการ mem, ...

จัดการ resource ↓ system call ↓ ช้าลง (overhead)

↳ window, macOS, linux → hybrid → แบบนี้ดีกว่า → ขยายใหญ่กว่า micro kernel 100%

↳ สมัครสองแบบ

↓

microkernel ขยายเล็กมาก