Introduction

บทบาทของระบบปฏิบัติการ (Roles of the Operating System)

- ผู้ตัดสิน(Referee) :
  -> การจัดสรรทรัพยากรระหว่างผู้ใช้ แอปพลิเคชัน
  -> การแยกผู้ใช้แอปพลิเคชันออกจากกัน
  -> การสื่อสารระหว่างผู้ใช้แอปพลิเคชัน
- นักลวงตา(Illusionist)
  -> แต่ละแอปพลิเคชันดูเหมือนจะมีเครื่องทั้งหมดเป็นของตัวเอง
  -> จำนวนโปรเซสเซอร์ที่ไม่มีที่สิ้นสุด (ใกล้) จำนวนหน่วยความจำที่ไม่มีที่สิ้นสุด พื้นที่เก็บข้อมูลที่เชื่อถือได้ การขนส่งเครือข่ายที่เชื่อถือได้
- กาว(Glue)
  -> ไลบรารี วิดเจ็ตอินเทอร์เฟซผู้ใช้ ...

ระบบปฏิบัติการคืออะไร?(What is an Operating System?)

- ชุดซอฟต์แวร์ที่จัดการทรัพยากรของคอมพิวเตอร์สำหรับผู้ใช้และแอปพลิเคชันของพวกเขา
- อาจมองเห็นหรือมองไม่เห็นแก่ผู้ใช้
- 2 ประเภทหลัก
  -> OS วัตถุประสงค์ทั่วไป
  -> OS วัตถุประสงค์เฉพาะ

การประเมินระบบปฏิบัติการ(Operating System Evaluation)

- ความน่าเชื่อถือและความพร้อมใช้งาน(Reliability and Availability)
- ความปลอดภัย(Security)
- การพกพา(Portability)
  -> AVM, API, HAL
- ประสิทธิภาพ(Performance)
  -> ค่าใช้จ่าย ประสิทธิภาพ
  -> ความเป็นธรรม เวลาตอบสนอง ปริมาณงาน
  -> ความสามารถในการคาดการณ์ประสิทธิภาพ
- การรับเป็นบุตรบุญธรรม(Adoption)

การออกแบบที่ประหนึ่งประนอม(Design Tradeoffs)

- ต้องสมดุลระหว่าง 5s
- ตัวอย่าง
  -> รักษา API ดั้งเดิม→พกพา↑, เชื่อถือได้↓, ปลอดภัย↓
  -> ทำลายสิ่งที่เป็นนามธรรม→ประสิทธิภาพ↑, การพกพา↓, ความน่าเชื่อถือ↓

Computer Performance Over Time

	1981	1997	2014	Factor (2014/1981)
Uniprocessor speed (MIPS)	1	200	2500	2.5K
CPUs per computer	1	1	10+	10+
Processor MIPS/$	$100K	$25	$0.20	500K
DRAM Capacity (MiB)/$	0.002	2	1K	500K
Disk Capacity (GiB)/$	0.003	7	25K	10M
Home Internet	300 bps	256 Kbps	20 Mbps	100K
Machine room network	10 Mbps (shared)	100 Mbps (switched)	10 Gbps (switched)	1000
Ratio of users to computers	100:1	1:1	1:several	100+

ระบบปฏิบัติการยุคแรก: คอมพิวเตอร์มีราคาแพงมาก (Early Operating Systems: Computers Very Expensive)

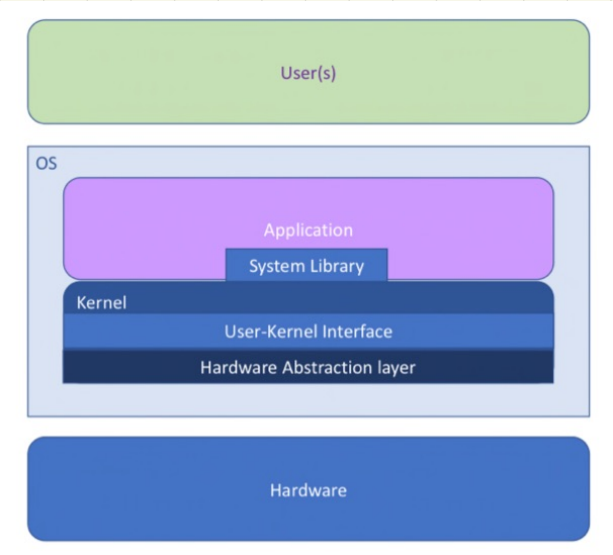
- ครั้งละหนึ่งโปรแกรม
  -> มีการควบคุมฮาร์ดแวร์อย่างสมบูรณ์
  -> OS เป็นไลบรารีรันไทม์
  -> ผู้ใช้จะยืนต่อแถวเพื่อใช้คอมพิวเตอร์
- ระบบแบทช์
  -> ทำให้ CPU ไม่ว่างด้วยคิวงาน
  -> OS จะโหลดงานถัดไปในขณะที่กำลังทำงานอยู่
  -> ผู้ใช้จะส่งงานและรอและรอและ

ระบบปฏิบัติการแบ่งปันเวลา: คอมพิวเตอร์และผู้คนมีราคาแพง (Time-Sharing Operating Systems: Computers and People Expensive)

- ผู้ใช้หลายคนบนคอมพิวเตอร์พร้อมกัน
  -> Multiprogramming: เรียกใช้หลายโปรแกรมพร้อมกัน
  -> ประสิทธิภาพเชิงโต้ตอบ: พยายามทำงานของทุกคนให้เสร็จอย่างรวดเร็ว
  -> เมื่อคอมพิวเตอร์มีราคาถูกลง สิ่งสำคัญคือต้องปรับเวลาของผู้ใช้ให้เหมาะสม ไม่ใช่เวลาคอมพิวเตอร์

ระบบปฏิบัติการในปัจจุบัน: คอมพิวเตอร์ราคาถูก (Today's Operating Systems: Computers Cheap)

- สมาร์ทโฟน
- ระบบสมองกลฝังตัว
- แล็ปท็อป
- แท็บเล็ต
- เครื่องเสมือน
- เซิร์ฟเวอร์ศูนย์ข้อมูล



ระบบปฏิบัติการในวันพรุ่งนี้ (Tomorrow's Operating Systems)

- ศูนย์ข้อมูลขนาดยักษ์
- จำนวนโปรเซสเซอร์ที่เพิ่มขึ้นต่อคอมพิวเตอร์หนึ่งเครื่อง
- จำนวนคอมพิวเตอร์ต่อผู้ใช้ที่เพิ่มขึ้น
- ที่เก็บของขนาดใหญ่มาก

Challenges

1. Reliability : ต้องมีความเสถียร
2. Availability : พร้อมใช้งานตลอดเวลา
3. Security : ป้องกันไวรัส
4. Privacy : แบ่งแยกไฟล์ของแต่ละคน
5. Portability : สามารถเคลื่อนย้ายโปรแกรมได้
   - For programs ต้องพิจารณา API,Abstraction virtual Machine
   - For OS "HW abstraction layer" เป็นการสร้าง abstraction ที่ใช้ในการอ้างอิงระหว่าง User กับฝั่งที่เป็นคนเขียน OS
6. Performance
   - Latency : ความเร็วในส่วนของ response
   - Throughput : จำนวนงานที่ทำได้ per time unit
   - Overhead : ให้ overhead น้อย
   - Fairness : ความ fair ของการใช้งานของแต่ละ program
   - Predicability :

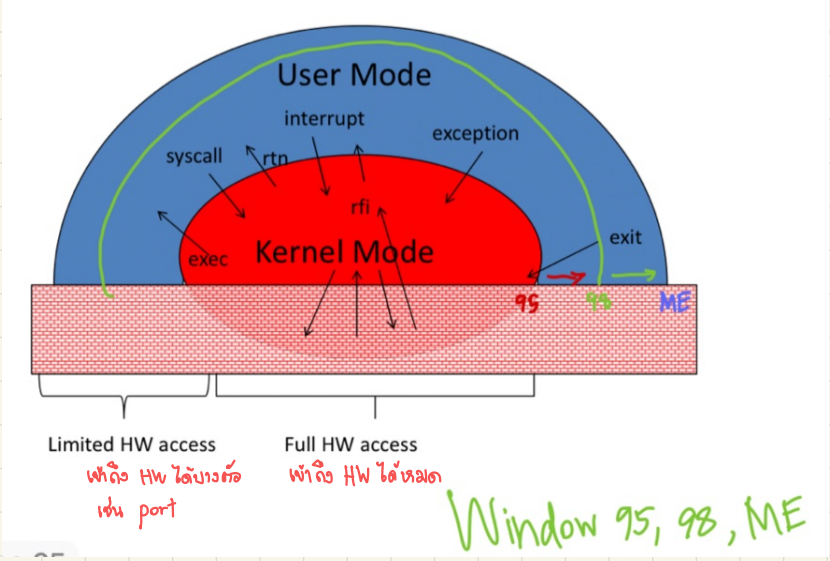
The Kernel Abstraction

What does an OS do...

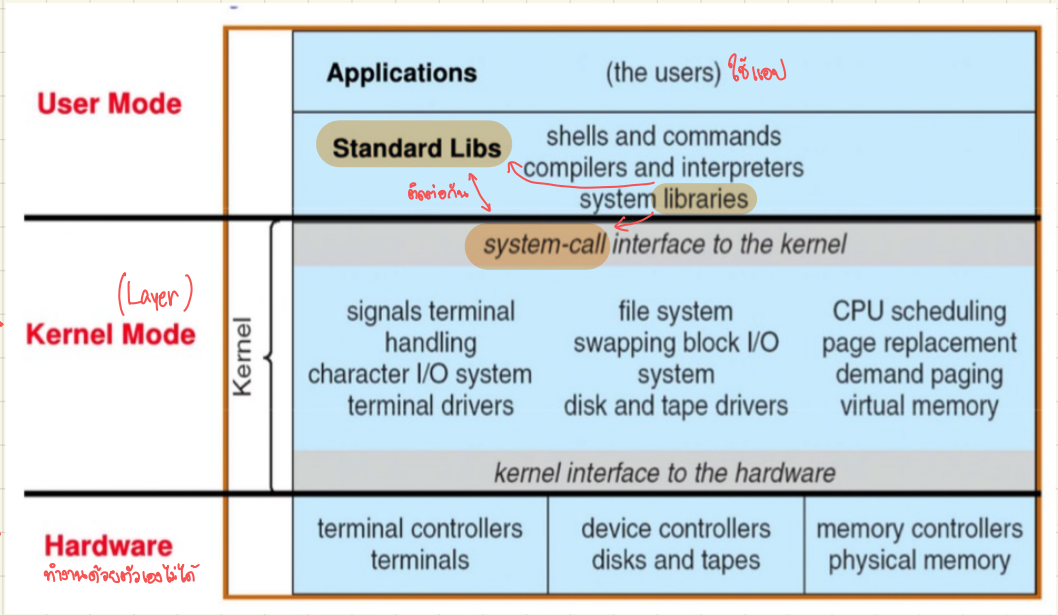
- **Hiding Complexity** ซ่อนการทำงานที่ซับซ้อนของ HW, การทำงานของ HW
  - Variety of HW ความหลากหลาย HW
    - -> E.g. different CPU, amount of RAM, I/O devices
- **Kernel** is the part of the OS that **running all the time** on the computer (in RAM)
  - **Core part of the OS** ❤ สำคัญของ OS (หน้าที่ Roles of OS ของ OS)
  - Manages system resources
  - Acts as a bridge between apps and HW เป็นสะพานเชื่อมระหว่าง app กับ HW

จุดอ่อน
- ถ้า kernel พัง เกิดอะไร
- SW (ไวรัส) เข้า kernel ได้ (อันตราย)

User/Kernel (Privileged) Mode (เมื่อก่อน)



UNIX Structure (ปัจจุบัน)



One of the major goals of OS is...

- Protecting Process and the Kernel
  - Running multiple programs รันหลายโปรแกรมแล้วไม่เกิดปัญหา
    - –> Keep them from interfering with the OS – kernel
    - –> Keep them from interfering with each other

App
↓ Run in
Kernel —control→ Resource

Protection: WHY?

ความน่าเชื่อถือ
- Reliability: buggy programs only hurt themselves โปรแกรมบัคแค่ตัวมันเอง (ไม่กระทบส่วนอื่น)
- Security and privacy: trust programs less อย่าไว้ใจโปรแกรม (บางโปรแกรมประสงค์ไม่ดี ≍_≍)
- Fairness: enforce shares of disk, CPU จัดการการทำงานให้เท่าเทียมกัน จะได้ไม่แย่งกันทำงานให้ข้อมูลเพี้ยน

ความเท่าเทียม

Protection: How? (HW/SW)

- CPU → 2 Main HW mechanism
  - Memory address translation แบ่งขอบเขตการทำงาน
  - Dual mode operation
    - –> Privileged/non-privileged mode (kernel / user) ← เปลี่ยน mode ได้
    - –> System calls

  ประมวลผลทั่วไป (+, −, ×, ÷, if, ...)

  } ทำให้การทำงานเร็วขึ้น เมื่อก่อนใช้แค่ SW เช็กเลยช้า ← เจอแต่ if

  ประมวลผลช้าลง ไม่ต้องเช็กบ่อย → SW
- SW
  - Process เกราะป้องกันระหว่าง App, เป็น illusionist (หลอก Program ว่า Run แค่ Program เดียว จริงๆ อาจจะ Run Program อื่นอยู่ด้วย)
  - System calls ใช้ติดต่อ Kernel (เป็นด่านก่อนเข้า HW)

Hardware Support: Dual-Mode Operation

- Kernel mode (privileged)
  - Execution with the full privileges of the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet
- User mode (non-privileged, genaral)
  - Limited privileges
  - Only those granted by the operating system kernel
- On the x86, mode stored in EFLAGS register (set by kernel or HW) → ตัวบอก mode
- On the MIPS, mode in the status register ↘ ที่เก็บ mode
- Privileged instructions
  - Available to kernel
  - Not available to user code

- Limits on memory accesses แบ่งขอบเขตของ App กับ Kernel
  - To prevent user code from overwriting the kernel
- Timer (interrupt) ขณะใช้งานอาจมีการเปลี่ยน mode จาก user → kernel เพราะมีเงื่อนไขบางอย่างจาก HW ในเวลาไม่นานจะกลับมาที่ user
  - To regain control from a user program in a loop
- Safe way to switch from user mode to kernel mode, and vice versa

Privileged instructions

- Privileged - ชุดคำสั่งเกี่ยวข้องกับ HW โดยตรง
- Non-Privileged - +, -, x, ÷, เช็ก if
- What should happen if a user program attempts to execute a privileged instruction? เกิด exception ส่วนของแอปที่รันจะขึ้นอยู่กับว่าแอปมีการเขียนดัก exception ไม่ดัก → จบแอป
  ดัก → ไปทำ function ตามที่ดัก แต่ instruction จะไม่ถูกประมวลผล

User Mode

- Application program
  - Running in process

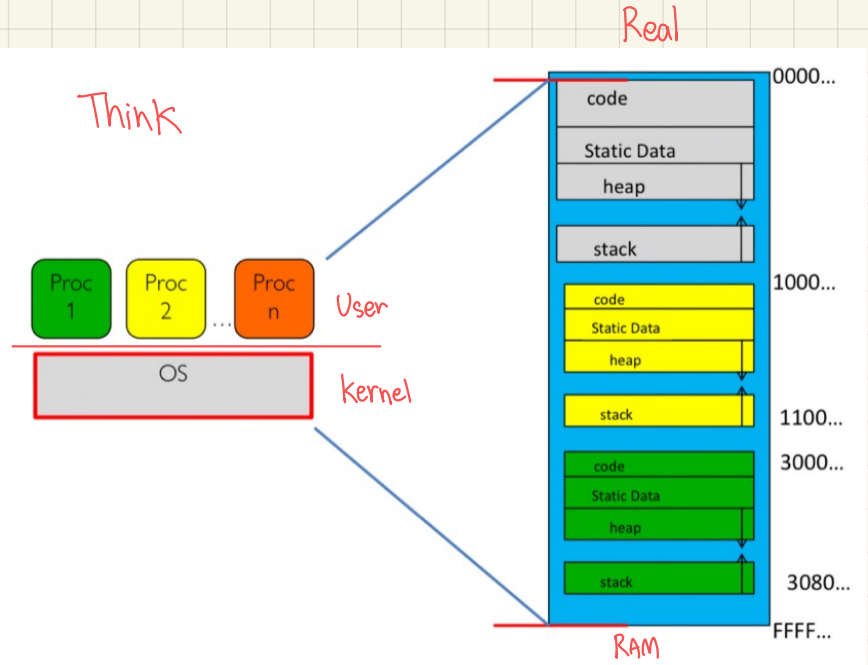
Virtual Machine:VM (Process)

- Software emulation of an abstract machine จำลองส่วน. ของระบบ HW บางตัวทำให้ App ทำงานได้
  - Give programs illusion they own the machine
  - Make it look like HW has feature you want
- 2 types of VM
  - Process VM จำลอง "บาง" ระบบ
    -> Supports the execution of a single program (one of the basic function of the OS)
  - System VM จำลอง "ทั้ง" ระบบ ← ไม่เกี่ยวกับที่เรียน
    -> Supports the execution of an entire OS and its applications

Process VMs

- GOAL:
  - Provide an isolation to a program แยกการทำงานของโปรแกรม
    -> Processes unable to directly impact other processes Process ใคร Process มัน
      > Boundary to the usage of a memory กำหนดขอบเขตของ memory
    -> Fault isolation
      > Bugs in program cannot crash the computer บัคจะไม่ส่งผลต่อคอมเรา
  - Portability (Program)
    -> Write the program for the OS rather the HW

Kernel mode & User mode

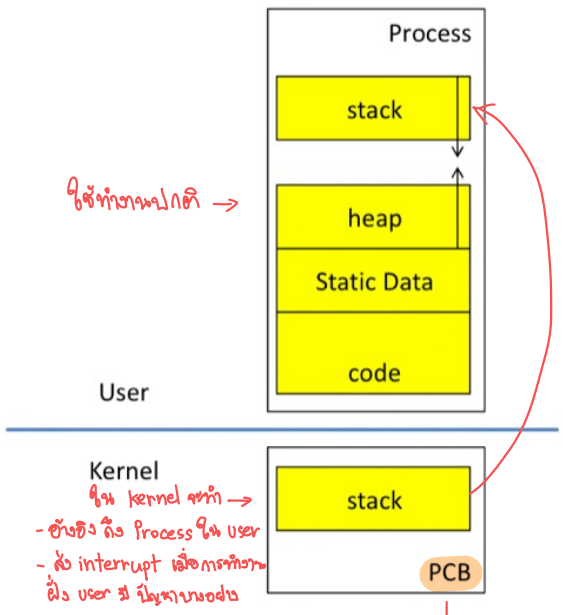


Process Abstraction

- **Process**: an **instance of a program**, **running with limited rights**
  - **Thread**: a sequence of instructions within 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 (e.g., which system calls it can make, what files it can access)

Process

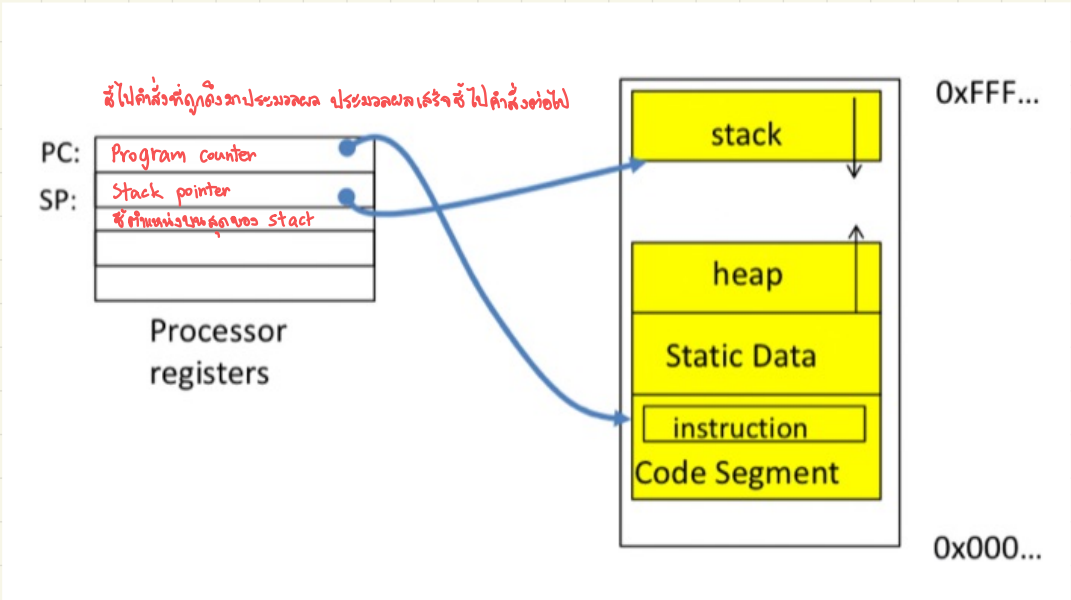
- 2 parts
  - PCB in kernel
  - Others in user



Process Control Block: PCB

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, ...)
  - Registers, SP, ... (when not running)
  - Process ID (PID), User, Executable, Priority, ...
  - Execution time, ...
  - Memory space, translation tables, ...
- Kernel Scheduler maintains a data structure containing the PCBs จัดการข้อมูล
- Scheduling algorithm selects the next one to run จัดการคิว

Address Space: In a Picture



Main Points

- Process concept แยกระบบกัน กับ Program รันสีกัน
  - 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 เข้าถึง HW ได้หมด
  - User-mode: execute with fewer privileges เข้าถึง HW โดยตรงไม่ได้ 555
- Safe control transfer เปลี่ยน mode แล้วยังปลอดภัย
  - How do we switch from one mode to the other?

Mode Switch (Kernel ↔ User)

- From user mode to kernel mode (User → Kernel)
  - Interrupts by some event in HW
    - -> Triggered by timer and I/O devices
  - Exceptions by Error (Ex. ÷0)
    - -> Triggered by unexpected program behavior
    - -> Or malicious behavior!
  - System calls (aka protected procedure call)
    - -> Request by program for kernel to do some operation on its behalf
    - -> Only limited # of very carefully coded entry points
- From kernel mode to user mode (Kernel → User)
  - New process/new thread start
    - -> Jump to first instruction in program/thread
  - Return from interrupt (HW), exception (SW), system call (SW)
    - -> Resume suspended execution
  - Process/thread context switch (by interrupt)
    - -> Resume some other process
  - User-level upcall (UNIX signal) * ตรงข้ามกับ system call
    - -> Asynchronous notification to user program

สร้าง Process ให้ Kernel → User ทำได้เปลี่ยนโหมด

ทำงาน, ตื่นค่าเสร็จ Kernel → User

ระบบโหลดเปลี่ยนไปทำโปรแกรมอื่น

User ใช้ → func. Kernel

Kernel ใช้ → func. User

Implementing Safe Kernel Mode Transfers

เปลี่ยน mode อย่างไรให้ปลอดภัย

- Carefully constructed kernel code packs up the user process state and sets it aside

ถ้า User → Kernel ต้อง save state ของ user process พอเปลี่ยน mode กลับจะได้ทำงานคำสั่งเดิมต่อได้

- Must handle weird/buggy/malicious user state
  - Syscalls with null pointers
  - Return instruction out of bounds
  - User stack pointer out of bounds

ห้ามเกิดคำสั่งแปลกๆ เพื่อไม่ให้เกิดช่องโหว่

- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself

บังคับเข้าไปใน Kernel พังยังไง

- User program should not know interrupt has occurred (transparency) aka.

User ไม่ควรรู้ว่ามี ms interrupt แทรกมั้น

เพิ่มความปลอดภัย

Device Interrupts

- OS kernel needs to communicate with physical devices
- Devices operate asynchronously from the CPU
  - Polling: Kernel waits until I/O is done
  - Interrupts: Kernel can do other work in the meantime
- Device access to memory
  - Programmed I/O: CPU reads and writes to device Ex. มี Data เข้ามา ไม่รู้จะทำไงก็ส่ง Data ให้ แล้วส่ง interrupt หา CPU ให้ CPU จัดการต่อ
  - Direct memory access (DMA) by device
  - Buffer descriptor: sequence of DMA's
    - E.g., packet header and packet body
  - Queue of buffer descriptors
    - Buffer descriptor itself is DMA'ed

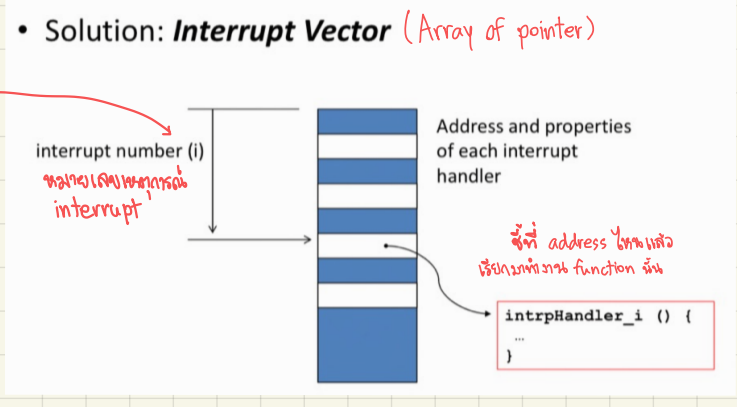
How do we take interrupts safely?

- Interrupt vector (Array of pointer)
  - Limited number of entry points into kernel ชี้ address ของ function
- Kernel interrupt stack ที่เก็บข้อมูลชั่วคราว ในระหว่างที่ Interrupt rounter ทำงานอยู่
  - Handler works regardless of state of user code
- Interrupt masking ปิด interrupt
  - Handler is non-blocking ไม่เกิด interrupt ซ้อนกัน
- Atomic transfer of control ทำงานโดย Interrupt เพราะ
  - "Single instruction"-like to change:
    - PC › Program counter
    - SP › Stack pointer
    - ย้าย Mem › Memory protection
    - เปลี่ยน mode › Kernel/user mode

    } ทำงานพร้อมกันเลยนะ
- Transparent restartable execution
  - User program does not know interrupt occurred โปรแกรมไม่รู้ว่าเกิด interrupt

Where do mode transfers go?

- Solution: ***Interrupt Vector*** (Array of pointer)



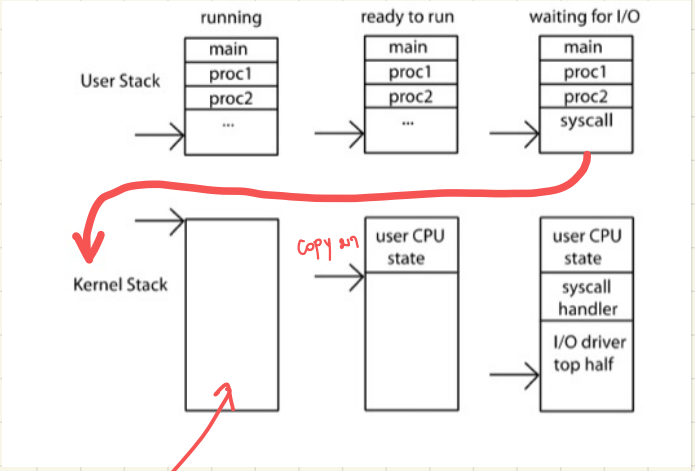
The Kernel Stack

Protect อย่างดี → Kernel > User ปลอดภัยกว่า!

- Interrupt handlers want a stack
- System call handlers want a stack

} stack in Kernel เลย

- Can't just use the user stack เพราะถ้า Program ชั่วๆ เข้าถึง user stack ได้ แล้วเปลี่ยนค่า ฯพ! (ไม่เหมาะกับความปลอดภัย)
- 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



แยก stack เพื่อความปลอดภัย :)

Interrupt Stack

- Per-processor, located in kernel (not user) memory เก็บค่าที่เกี่ยวกับขั้นตอนการทำงาน Ex. state reg
  - Usually a process/thread has both: kernel and user stack
- Why can't the interrupt handler run on the stack of the interrupted user process?

Interrupt Masking

ปิด interrupt ก็เพราะเกิด interrupt ซ้ำ

- Interrupt handler runs with interrupts off ปิด interrupt จนกว่า interrupt rounter เสร็จถึงเปิด
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off เปิด/ปิด interrupt ได้
  - Eg., when determining the next process/thread to run
  - On x86
    - CLI: disable interrrupts → ใช้ใน interrupt rounter
    - STI: enable interrupts → ใช้หลังจบ interrupt rounter
    - Only applies to the current CPU (on a multicore)

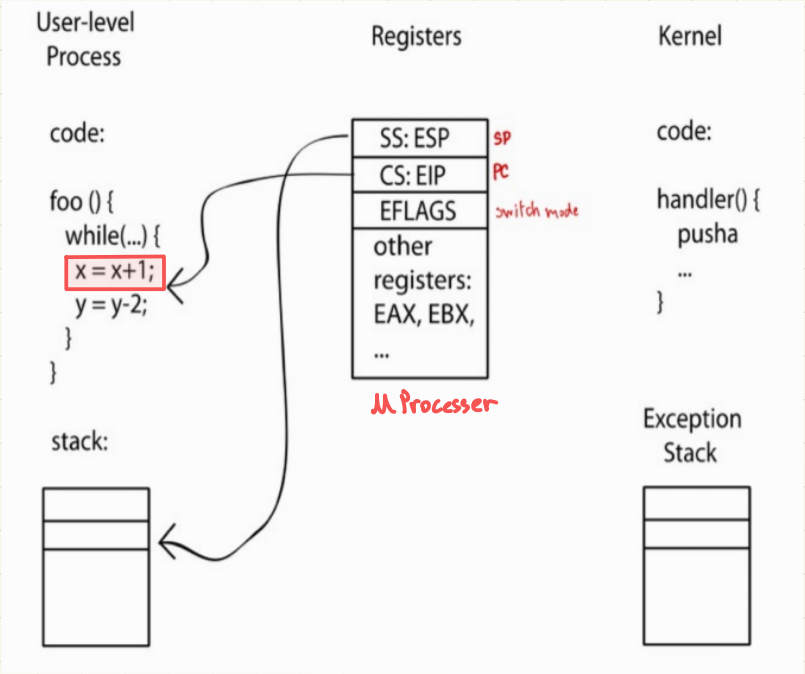
Case Study: x86 Interrupt

- Save current stack pointer (SP)
- Save current program counter (PC)

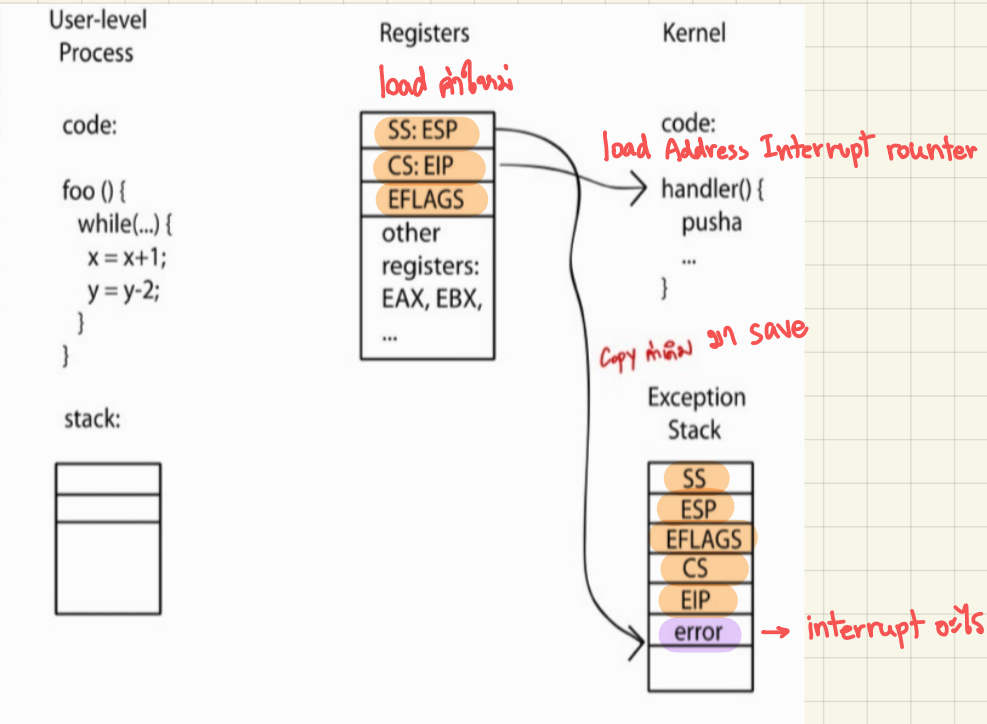
} โปรแกรมทำงานถึงไหนแล้ว

- Save current processor status word (condition codes) (มีมั้ง, ไม่มีมั้ง) ดูค่าในแมนนวลเอา
- 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

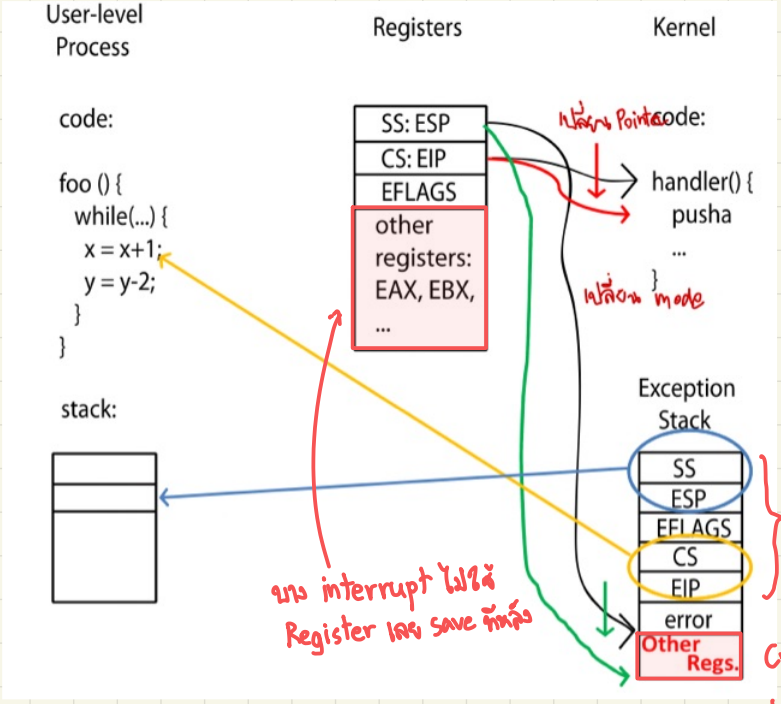
Before Interrupt



During Interrupt



After Interrupt



At end of handler

- Handler restores saved registers
- Atomically return to interrupted process/thread
  - Restore program counter
  - Restore program stack
  - Restore processor status word/condition codes
  - Switch to user mode

ทำงานที่เดียวพร้อมกัน } Copy ที่เดียว 3 หน่อย

Hardware support: Interrupt Control

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

```
x = x+1;   ← ส่งสัญญาณเกิด interrupt ได้ แต่ยังไม่ switch mode
y = y-2;   ← เกิด interrupt ได้ เกิด switch mode
```

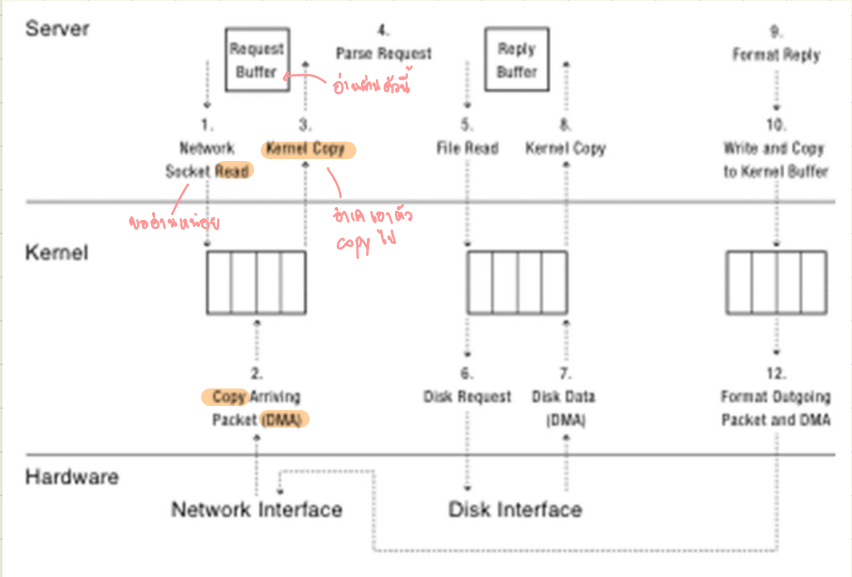
- Interrupt Handler invoked with interrupts 'disabled' ไม่เกิด interrupt ซ้อน
  - Re-enabled upon completion (คนหน้าจะทำงานเสร็จ)
  - Non-blocking (run to completion, no waits) ไม่มีใครขัดได้
  - Pack up in a queue and pass off to an OS thread for hard work
    - wake up an existing OS thread
- OS kernel may enable/disable interrupts ปิด/เปิด interrupt ได้
  - 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 จริงๆ เกิด interrupt ซ้อนกันได้ (แต่ไม่สอน)
  - Mask off (disable) certain interrupts, eg., lower priority
  - Certain Non-Maskable-Interrupts (NMI) แบ่งคนชั้น interrupt ด้วย
    - e.g., kernel segmentation fault
    - Also: Power about to fail!

by SW

Kernel System Call Handler

- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments ค้นหา
  - In registers or on user (!) stack
- Copy arguments
  - From user memory into kernel memory (Stack) – carefully checking locations!
  - Protect kernel from malicious code evading checks
- Validate arguments ตรวจสอบความถูกต้องใน kernel
  - Protect kernel from errors in user code
- Copy results back ส่งค่าไปจาก user stack
  - Into user memory – carefully checking locations!

ถ้าอยู่ใน User stack อาจมีการเปลี่ยนค่าเมื่อทำการ Validate เสร็จ (ไม่ปลอดภัย)



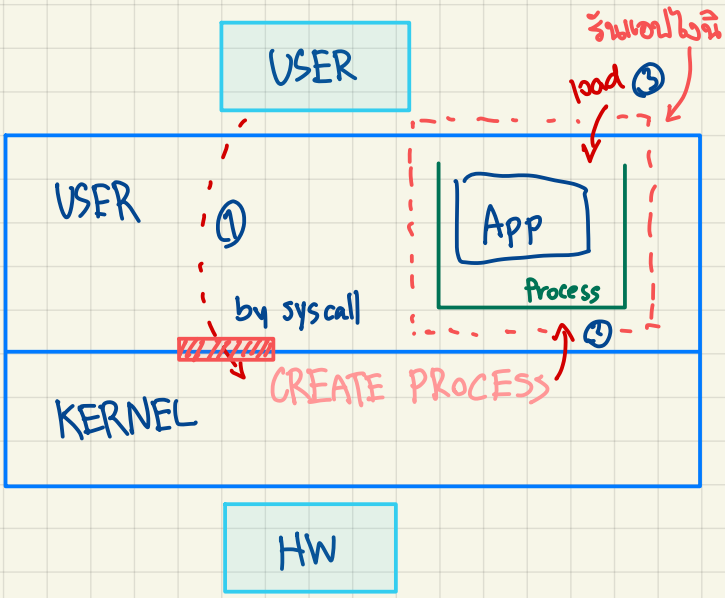
Today: Four Fundamental OS Concepts

- Thread : Execution Context
  - Program Counter, Registers, Execution Flags, Stack
- Address space (with translation)
  - Program's view of memory is distinct from physical machine
- Process : an instance of a running program
  - Address Space + One or more Threads
- Dual mode operation / Protection
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

The Process and Programming Interface

Main Points

- Creating and managing processes
  - fork, exec, wait
- Performing I/O
  - open, read, write, close
- Communicating between processes
  - pipe, dup, select, connect
- Example: implementing a shell



Shell

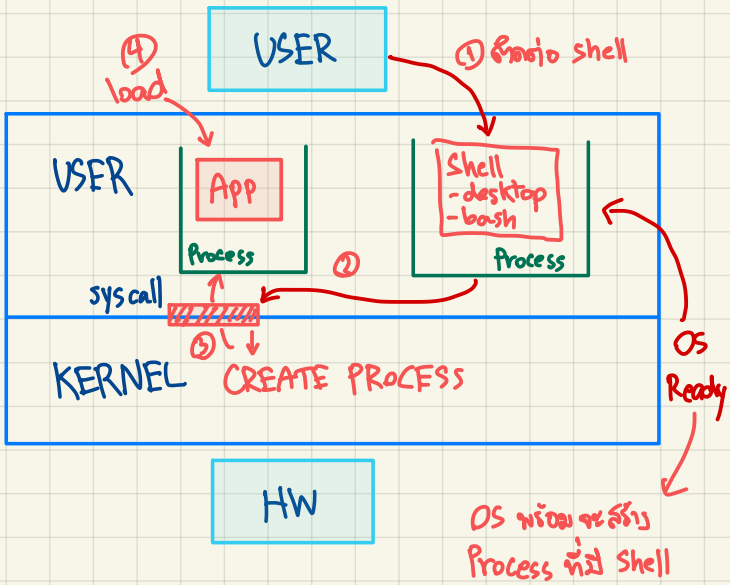
interface desktop

- A shell is a job control system คอนโทรลการทำงาน & UI (ติดต่อ User กับ Kernel)
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells
- Example: to compile a C program

```
cc –c sourcefile1.c
cc –c sourcefile2.c
ln –o program sourcefile1.o sourcefile2.o
```

Shell - สื่อสารกับ User ควบคุม OS

Kernel - ควบคุม HW



Windows CreateProcess (function)

System call Name

- System call to create a new process to run a program
  - Create and initialize the process control block (PCB) in the kernel
  - Create and initialize a new address space
  - Load the program into the address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context (PC, SP) to start execution at "start"
  - Inform the scheduler that the new process is ready to run

แต่ยังไม่รันทันทีขึ้นอยู่กับ scheduler ประมวลผล

Windows CreateProcess API (simplified)

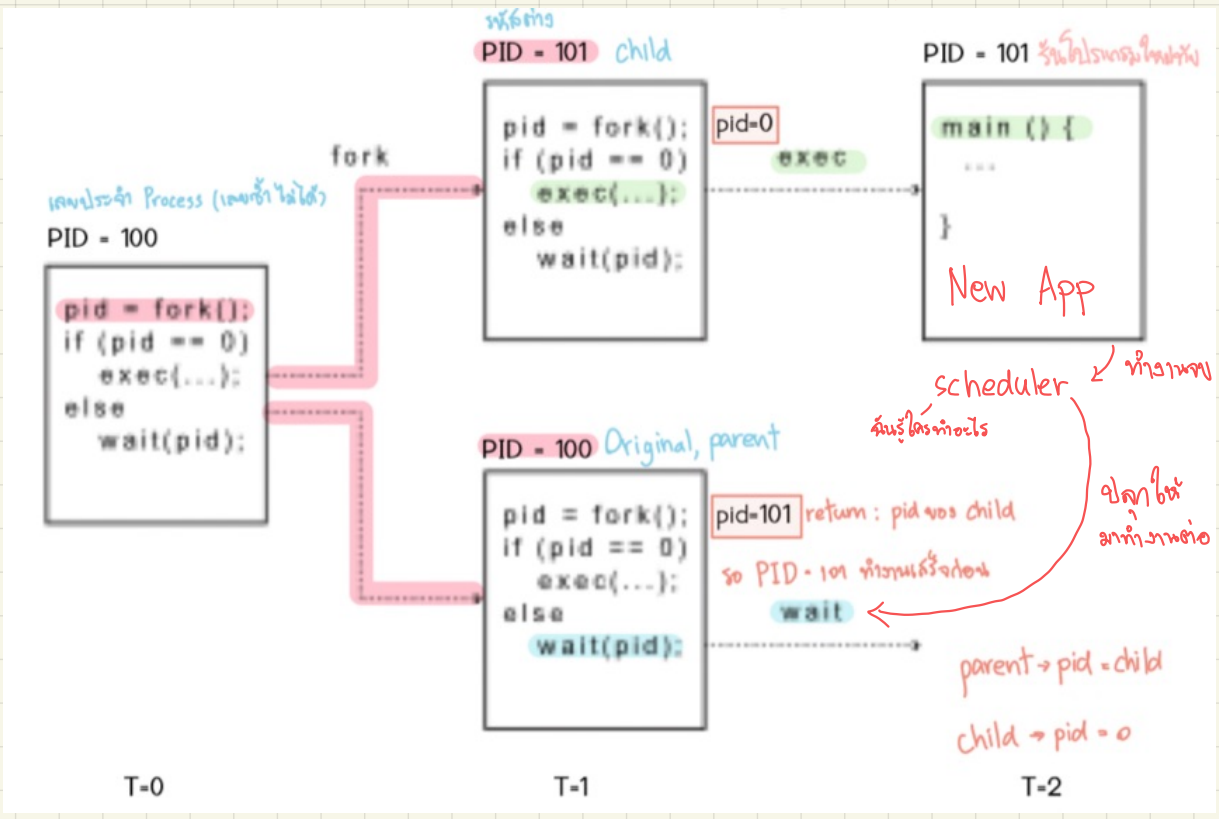
```
if (!CreateProcess(
    NULL,           // No module name (use command line)
    argv[1],        // Command line
    NULL,           // Process handle not inheritable
    NULL,           // Thread handle not inheritable
    FALSE,          // Set handle inheritance to FALSE
    0,              // 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
)
```

1 คำสั่ง หลาย parameter

UNIX Process Management

- UNIX fork – system call to create a copy (โคลนนิ่ง) of the current process, and start it running
  - No arguments!
- UNIX exec – system call to change the program (เปลี่ยนโปรแกรมทับ) being run by the current process
- UNIX wait – system call to wait for a process to finish (รอการทำงานเสร็จก่อน)
- UNIX signal – system call to send a notification to another process (ส่งสัญญาณไป process อื่น (ปลุก process wait))

4 คำสั่ง แต่ไม่ต้องมี parameter ก็ได้ ...



Implementing UNIX fork

Steps to implement UNIX fork เกิดปัญหาได้

! – Create and initialize the process control block (PCB) in the kernel
! – Create a new address space
} RAM เต็ม RUN ไม่ได้

– 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)
Copy ทุกอย่าง PC, SP
– Inform the scheduler that the new process is ready to run

wait() รันทันทีได้มั้ย? (รอที่ไม่รอ)
-ได้ ถ้า → Process ก่อน wait() รันเสร็จก่อนแล้ว
→ Process หาย หรือไม่มี

Implementing UNIX exec

- Steps to implement UNIX exec เกิดปัญหาได้
! – Load the program into the current address space ใส่โปรแกรมไม่ได้ คอขาด
– Copy arguments into memory in the address space
– Initialize the hardware context to start execution at "start"
SP, PC

UNIX I/O ปัญหาน้องสุด

- Uniformity ความสม่ำเสมอ
– All operations on all files, devices use the same set of system calls: open, close, read, write
- Open before use เปิดก่อนใช้ (เปิดก่อนได้ประโยชน์)
– Open returns a handle (file descriptor) for use in later calls on the file
- Byte-oriented รูปแบบการเขียน (ใช้ Byte เป็นโค้ดควบคุม หรือ Protocol เชิงอักขระ)
- Kernel-buffered read/write ทำงานผ่าน buffered ใน Kernel เขียนใน buffer จนกว่าจะเต็ม
- Explicit close
– To garbage collect the open file descriptor เก็บขยะไฟล์

UNIX File System Interface

- UNIX file open is a Swiss Army knife: (open, create, exits(check))
– 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 exists but isn't empty, nix it then open
› If file exists but isn't empty, return an error
› ...

- open
- create
- check (exists)
} ทำไมไม่แยกเป็น 3 function?
if 1 user single path = OK!
but multi user multi path = WTF Boommmm
เพราะ พอเช็คโค้ดอาจมีการโดนเปลี่ยนโค้ดทับ, สร้างไม่ได้ ไม่รอดแน่ จ้า T-T

Implementing a Shell

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

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
    child_pid = fork();       // create a child process
    if (child_pid == 0) {
        exec(prog, args);        // I'm the child process.  Run program
       // NOT REACHED
    } else {
        wait(child_pid);         // I'm the parent, wait for child
        return 0;
    }
}
```

In Unix

- A program can be a **file of commands**
- A program can **send its output to a file**
- A program can **read its input from a file**

  dir > (แล้วแต่ชื่อ) file.txt
  redirect
  Program < (read) file.txt

- The **output of one program can** be the **input to another program**

  pipe
  dir | more
  Program Program

Interprocess Communication

- **Producer-consumer** ต้องพร้อมทำงาน 2 ฝั่ง
  - Output of one program is accepted as input of another program
    Op ของโปรแกรมหนึ่ง เป็น Ip ของอีกโปรแกรม
    - One-way communication
      Producer ส่งได้แค่คนเดียว, consumer ตอบกลับไม่ได้
      (คุยคนเดียว)
    - Pipe
- **Client-server** ต้องพร้อมทำงาน 2 ฝั่ง
  - Two-way communication ตอบกลับกันได้
  - Server implements specialize task
    - Print serve

  Client — socket 127.0.0.1 — Server
- **File system**
  - Write data to a file then read file as an input
  - Reader and writer are not need to running at the same time
    ไม่ต้องทำงานพร้อมกันก็ได้

Operating system structure

- Microkernel

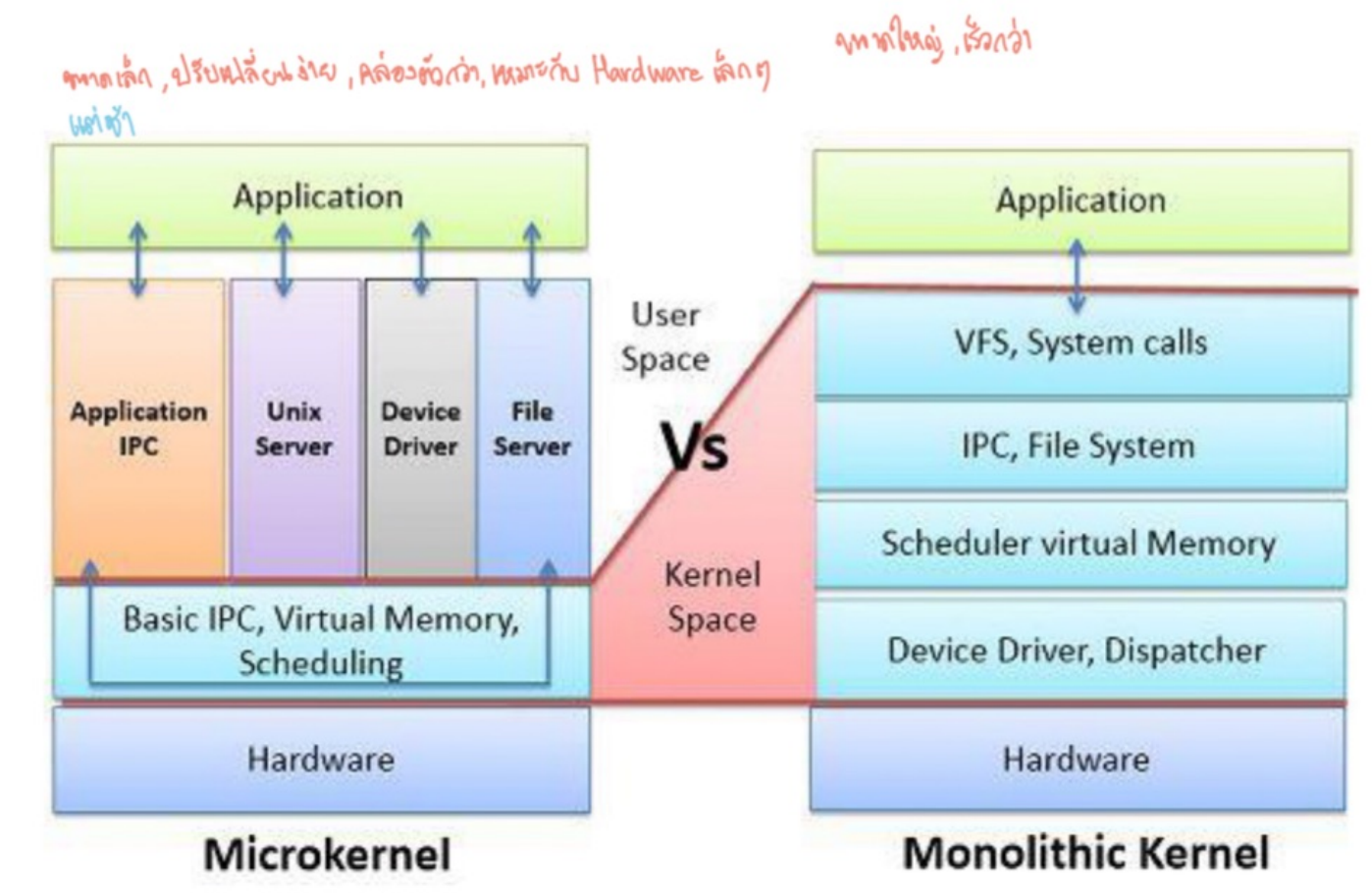
ข้อดี
- ปรับเปลี่ยนง่าย(แก้ไข โหลดซ้ำ แทนที่ ขยายระบบ) โดยไม่ต้องแก้ไข Kernel
- คล่องตัวกว่า
- ระบบขัดข้องได้น้อยกว่า Monolithic Kernel
- มีขนาดเล็ก เหมาะกับ HW ขนาดเล็ก

ข้อเสีย
- ช้า
- จำเป็นต้องมี context switch เมื่อ drivers run as processes
- ประสิทธิภาพอาจรวนจนเกิดปัญหาได้
- แพง

- Monolithic kernel

ข้อดี
- เร็วกว่า
- ขนาดใหญ่
- ดำเนินการจบในตัวมันเอง
- โครงสร้างไม่ซับซ้อนและเรียบง่าย

ข้อเสีย
- ถ้าจะแก้ แก้ทั้งระบบ
- พอร์ตโค้ดเข้ายาก
- บริการพังหนึ่ง พังทั้งระบบ



Concurrency

Share time to run in Uprocesser

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

Why Concurrency?

- Servers
  - Multiple connections handled simultaneously
- Parallel programs
  - To achieve better performance
- Programs with user interfaces
  - To achieve user responsiveness while doing computation
- Network and disk bound programs
  - To hide network/disk latency

Definitions

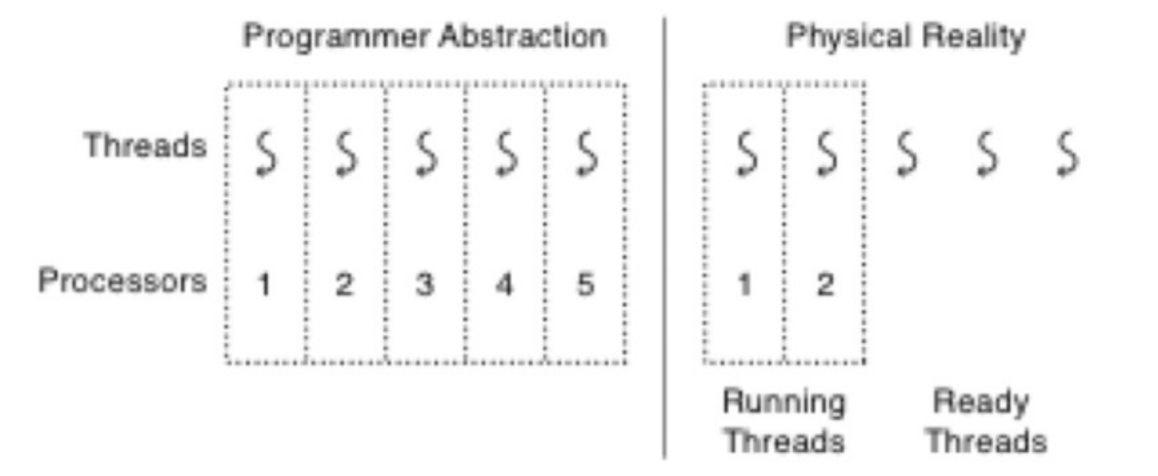
- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence: familiar programming model
  - Separately schedulable: OS can run or suspend a thread at any time
- Protection is an orthogonal concept
  - Can have one or many threads per protection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

Threads in the Kernel and at User-Level

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

Thread Abstraction

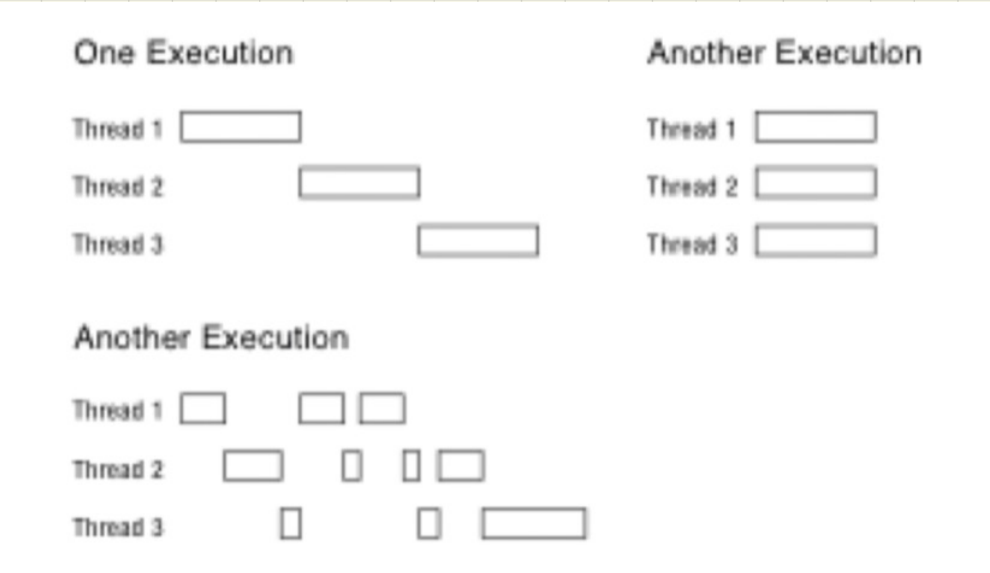
- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule



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;

Possible Executions



Thread Operations

- thread_create(thread, func, args)
  - Create a new thread to run func(args)
- thread_yield()
  - Relinquish processor voluntarily
- thread_join(thread)
  - In parent, wait for forked thread to exit, then return
- thread_exit
  - 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?
}
```

threadHello: Example Output

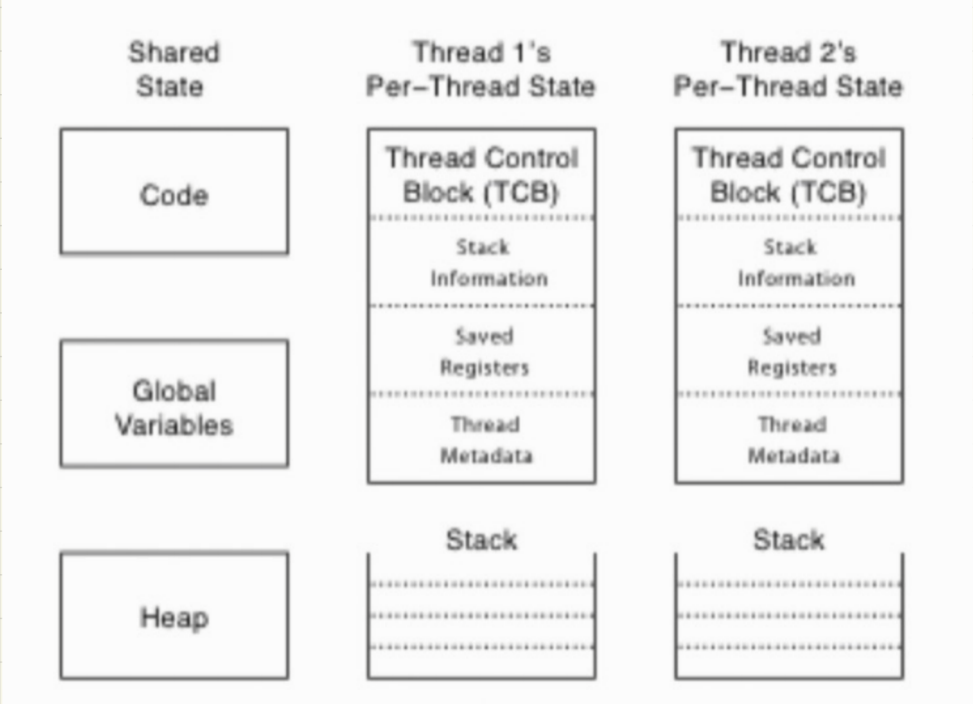
- Why must "thread returned" print in order?
- What is maximum # of threads running when thread 5 prints hello?
- Minimum?

```
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.
```

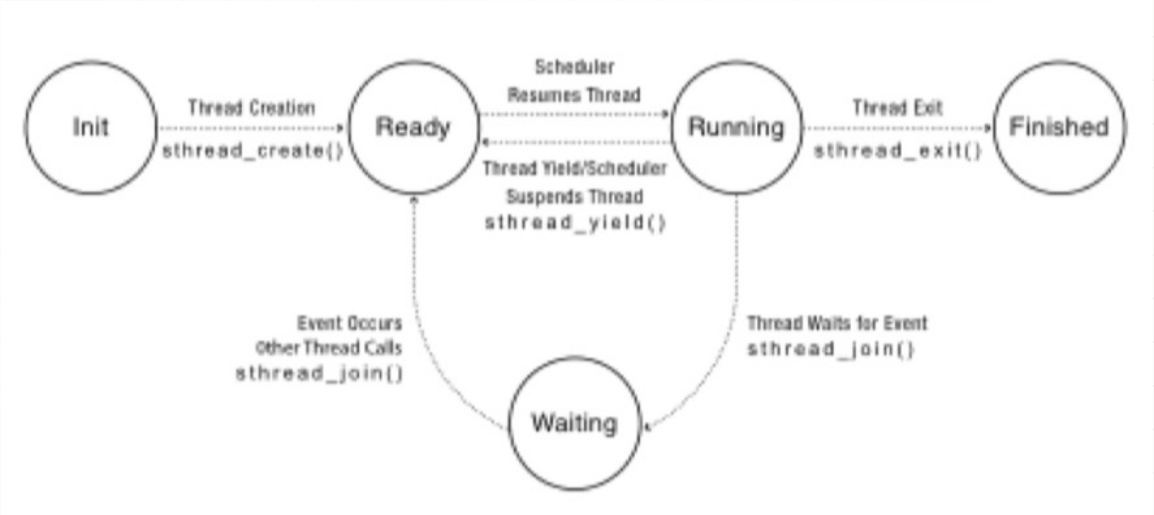
Fork/Join Concurrency

- 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



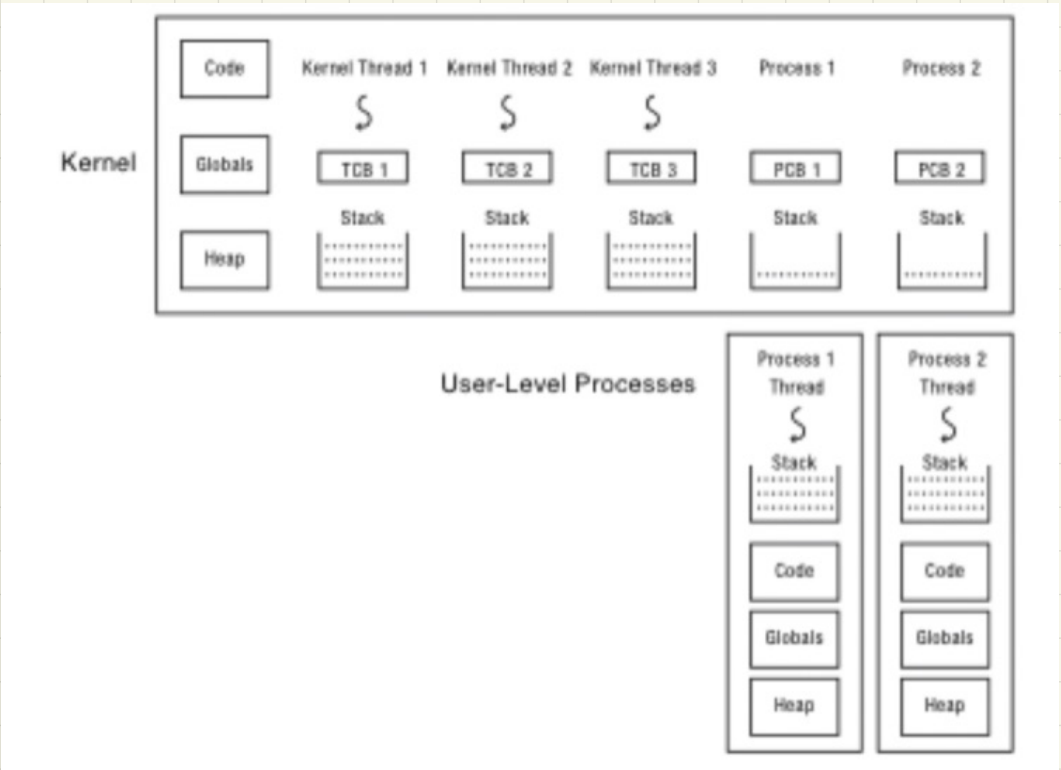
Thread Lifecycle



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



Implementing threads

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

Thread Stack

- 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?

Thread Context Switch

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

Voluntary thread context switch

- Save registers on old stack
- Switch to new stack, new thread
- Restore registers from new stack
- Return
- Exactly the same with kernel threads or user threads

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

Involuntary Thread/Process Switch

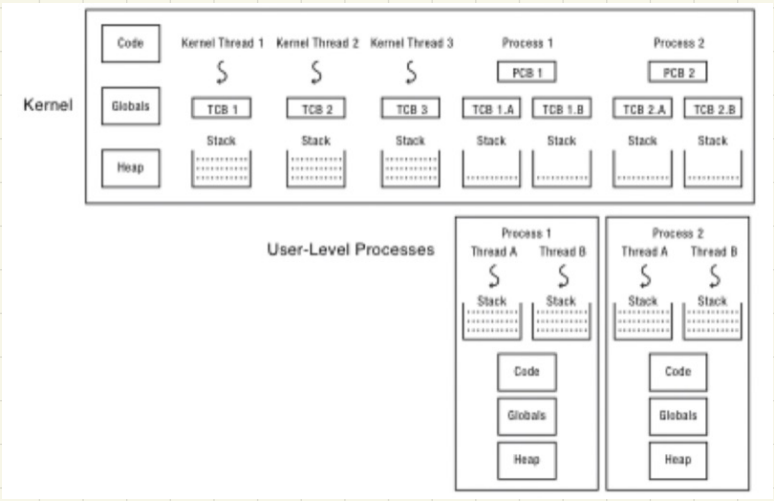
- Timer or I/O interrupt
  - Tells OS some other thread should run
- Simple version (OS/161)
  - End of interrupt handler calls switch()
  - When resumed, return from handler resumes kernel thread or user process
  - Thus, processor context is saved/restored twice (once by interrupt handler, once by 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



Multithreaded User Processes (Take 2)

- 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

Multithreaded User Processes (Take 3)

- 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

Synchronization

The Dark Side of Concurrency

With interleaved executions, the order in which processes execute at runtime is *nondeterministic*. คาดการณ์ไม่ได้

- depends on the exact order and timing of process arrivals ระบุเวลา, คำสั่งไม่ได้
- depends on exact timing of asynchronous devices (disk, clock) interrupt, I/O
- depends on scheduling policies

Some schedule interleavings may lead to incorrect behavior.

- Open the bay doors *before* you release the bomb.
- Two people can't wash dishes in the same sink at the same time.

The system must provide a way to coordinate concurrent activities to avoid incorrect interleavings.

Synchronization Motivation

- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes when re-run program
- Compiler/hardware instruction reordering มีการจัดลำดับ เพื่อให้ Performance สูงที่สุด
- Multi-word operations are not atomic
  - Big Data

Question: Can this panic?

Thread 1

```
p = someComputation();
pInitialized = true;
```

Thread 2

```
while (!pInitialized) ;
q = someFunction(p);
if (q != someFunction(p))
    panic
```

False

Why Reordering?

- Why do compilers reorder instructions?
  - Efficient code generation requires analyzing control/data dependency
  - If variables can spontaneously change, most compiler optimizations become impossible
- Why do CPUs reorder instructions?
  - Write buffering: allow next instruction to execute while write is being completed

Fix: memory barrier สิ่งกีดขวาง, ไว้กำหนดลำดับว่าจะทำงาน

- Instruction to compiler/CPU
- All ops before barrier complete before barrier returns
- No op after barrier starts until barrier returns

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Shared resource

แย่งกัน Update

แย่งกัน Update

Definitions

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

ปัญหา ใช้ resource ตัวเดียวกัน แย่งกัน Update output

แก้ได้

- 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

วิธีการ ใช้ resource ได้ทีละ 1 thread ห้ามมี thread อื่นเข้ามาใน thread ตอนที่กำลังทำงานอยู่

ถ้ามีส่วน Critical Mutual ได้แน่ มี shared resource มี Critical

- 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!)

เครื่องมือ

shared resource

จบ critical

Too Much Milk, Try #1

- Correctness property
  - Someone buys if needed (liveness)
  - At most one person buys (safety)
- Try #1: leave a note

```
if (!note)        // true → false, false → true
    if (!milk) {  // false → true
        leave note
        buy milk
        remove note
    }
```

Too Much Milk, Try #2

Starvation (ไม่ได้ทำงานเลย)

Thread A	Thread B
leave note A	leave note B
if (!note B) {	if (!noteA) {
if (!milk)	if (!milk)
buy milk	buy milk
}	}
remove note A	remove note B

Too Much Milk, Try #3

Thread A	Thread B
leave note A	leave note B
while (note B) // X	if (!noteA) { // Y
do nothing;	if (!milk)
if (!milk)	buy milk
buy milk;	}
remove note A	remove note B

A ไม่ได้ทำงาน

Can guarantee at X and Y that either:

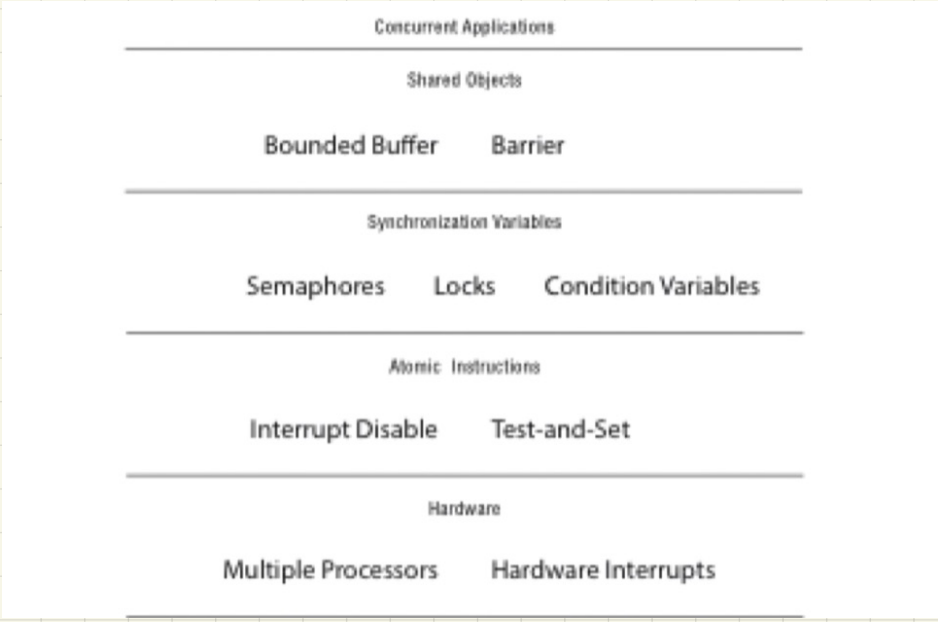
(i) Safe for me to buy

(ii) Other will buy, ok to quit

Lessons

- Solution is complicated
  - "obvious" code often has bugs
- Modern compilers/architectures reorder instructions
  - Making reasoning even more difficult ยิ่ง thread มากยิ่งซับซ้อน
- Generalizing to many threads/processors
  - Even more complex: see Peterson's algorithm

Roadmap



Locks

- Lock::acquire request (ขอ)
  - wait until lock is free, then take it
- Lock::release (ขอคืน)
  - release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety) 1 lock ในช่วงเวลาหนึ่ง
2. If no one holding, acquire gets lock (progress) ไม่มีใครถือ lock ใครก็ขอ lock ต่อได้
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Too Much Milk, Try #4

Locks allow concurrent code to be much simpler:

```
lock.acquire();
if (!milk)
    buy milk
lock.release();
```

Lock Example: Malloc/Free

lock ไม่ใช่ให้มีไว้กันคนทำงานพร้อมกัน เรียงลำดับ

```
char *malloc (n) {
    heaplock.acquire();
    p = allocate memory
    heaplock.release();
    return p;
}
```

shared resource → heaplock

```
void free(char *p) {
    heaplock.acquire();
    put p back on free list
    heaplock.release();
}
```

Rules for Using Locks

- Lock is initially free ไม่ถูกครอบครอง
- Always acquire before accessing shared data structure
  - Beginning of procedure! ก่อน critical
- Always release after finishing with shared data
  - End of procedure! หลัง Critical
  - Only the lock holder can release lock ของใครของมัน
  - DO NOT throw lock for someone else to release
- Never access shared data without lock มีโอกาส thread ไม่มี lock เข้าถึง shared resource ได้ (แย่กว่า)
  - Danger!

Will this code work?

```
if (p == NULL) {
    lock.acquire();
    if (p == NULL) {
        p = newP();
    }
    lock.release();
}
use p->field1
```

```
newP() {
    p = malloc(sizeof(p));
    p->field1 = ...
    p->field2 = ...
    return p;
}
```

Semaphores

- Semaphore has a non-negative integer value
  - P() atomically waits for value to become > 0, then decrements
  - V() atomically increments value (waking up waiter if needed)
- Semaphores are like integers except:
  - Only operations are P and V
  - Operations are atomic
    - If value is 1, two P's will result in value 0 and one waiter
- Semaphores are useful for
  - Unlocked wait: interrupt handler, fork/join

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

Condition Variable Design Pattern

```
methodThatWaits() {
    lock.acquire();
    // Read/write shared state

    while (!testSharedState()) {
        cv.wait(&lock);
    }

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

```
methodThatSignals() {
    lock.acquire();
    // Read/write shared state

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

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

str=null? true

1. release lock
2. running → wait

str = Hello

Van

3. wait → ready
4. acquire lock

Scheduling

Main Points

- Scheduling policy: what to do next, when there are multiple threads ready to run (ทำงานอะไรต่อดี!)
  - Or multiple packets to send, or web requests to serve, or ...
- Definitions
  - response time, throughput, predictability (วัดประสิทธิภาพจากตรงนี้)
- Uniprocessor policies (focus Algorithm)
  - FIFO, round robin, optimal
  - multilevel feedback as approximation of optimal
- Multiprocessor policies
  - Affinity scheduling, gang scheduling
- Queueing theory
  - Can you predict/improve a system's response time?

Definitions

- Task/Job
  - User request: e.g., mouse click, web request, shell command, ...
- Latency/response time (1 งาน เสร็จกี่ร? (ยิ่งน้อยยิ่งดี))
  - How long does a task take to complete?
- Throughput (1 วิ ได้กี่งาน (ยิ่งเร็ว))
  - How many tasks can be done per unit of time?
- Overhead
  - How much extra work (ไม่ใช่งาน แต่ต้องทำ) is done by the scheduler?
- Fairness
  - How equal is the performance received by different users?
- Predictability (ความสม่ำเสมอในการประมวลผล สามารถคาดการณ์เวลาในการทำงานได้)
  - How consistent is the performance over time?
- Workload (or program)
  - Set of tasks for system to perform
- Preemptive scheduler
  - If we can take resources away from a running task

- **Work-conserving**
  - Resource is used whenever there is a task to run
- **Scheduling algorithm**
  - takes a workload as input
  - decides which tasks to do first
  - Performance metric (throughput, latency) as output
  - Only preemptive, work-conserving schedulers to be considered

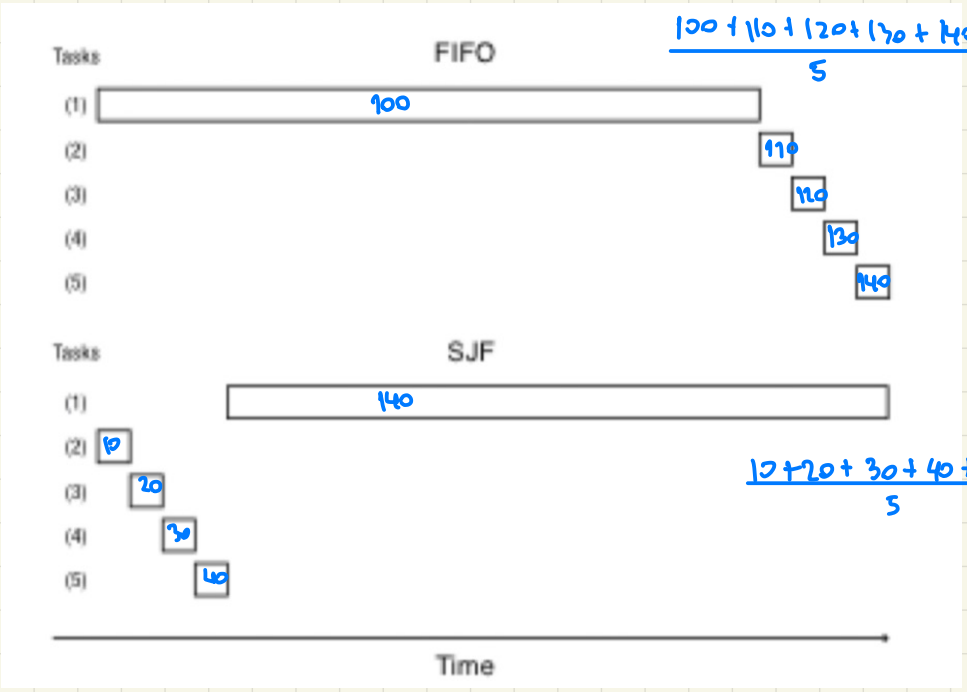
First In First Out (FIFO) -Fairness ✓

- Schedule tasks in the order they arrive
  - Continue running them until they complete or give up the processor
- Example: memcached
  - Facebook cache of friend lists, ...
- On what workloads is FIFO particularly bad?

Shortest Job First (SJF) งานประมวลผลน้อยสุดมาก่อน

- Always do the task that has the shortest remaining amount of work to do
  - Often called Shortest Remaining Time First (SRTF)
- Suppose we have five tasks arrive one right after each other, but the first one is much longer than the others
  - Which completes first in FIFO? Next?
  - Which completes first in SJF? Next?

FIFO vs. SJF

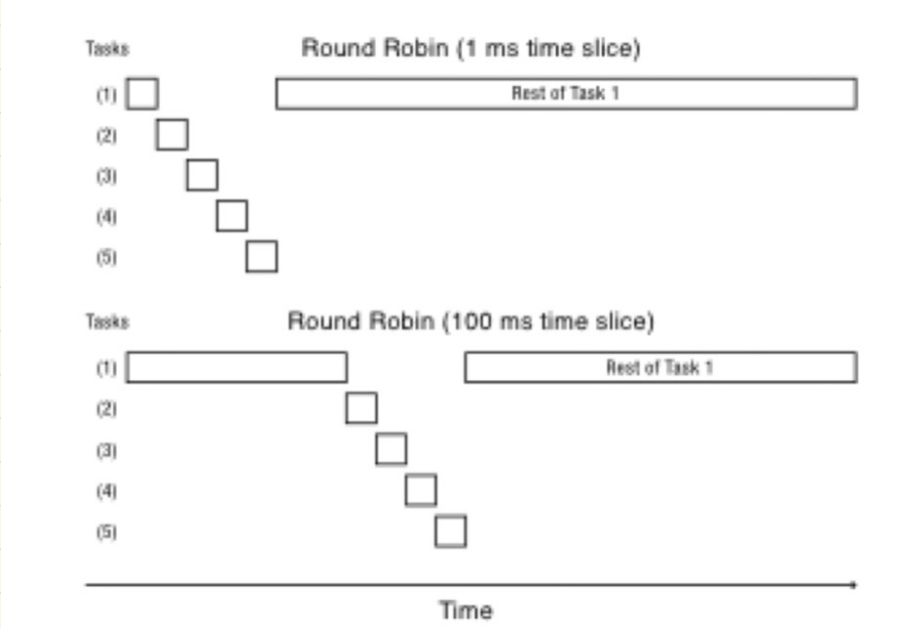


$\frac{100+110+120+130+140}{5} = \frac{600}{5} = 120$

$\frac{10+20+30+40+140}{5} = \frac{340}{5} = 68$

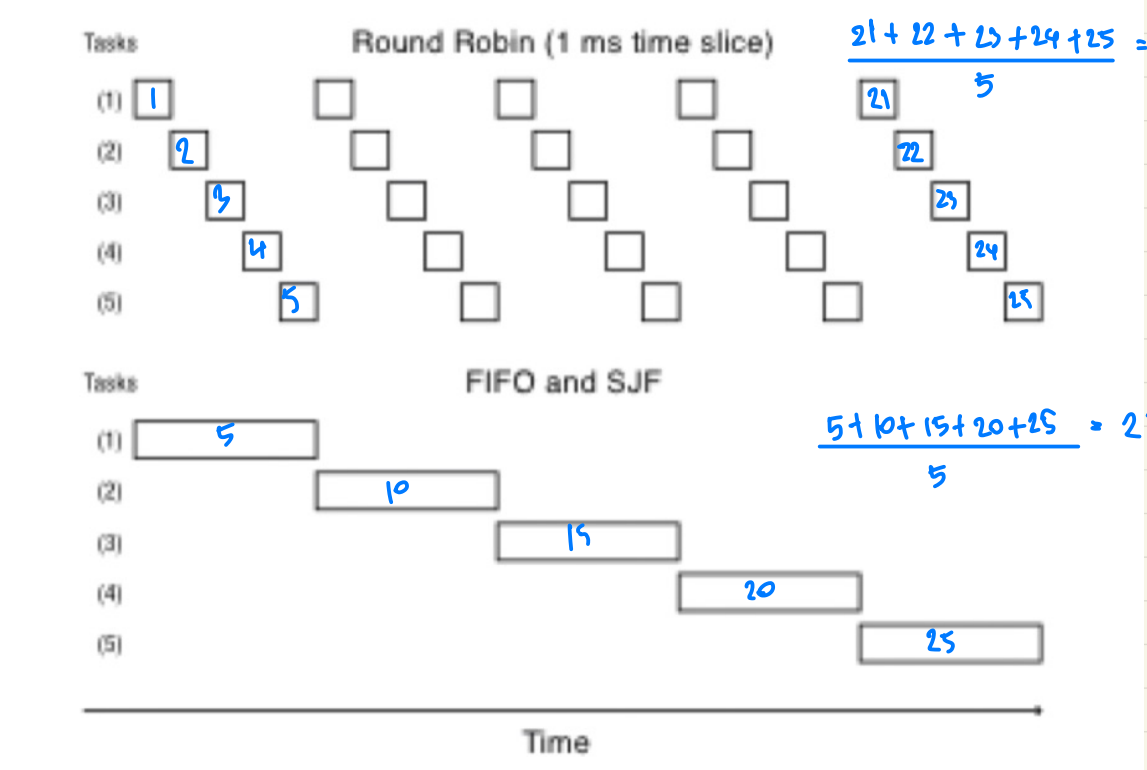
Round Robin

- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn't complete, it goes back in line
- Need to pick a time quantum
  - What if time quantum is too long?
    - Infinite?
  - What if time quantum is too short?
    - One instruction?



Round Robin vs. FIFO

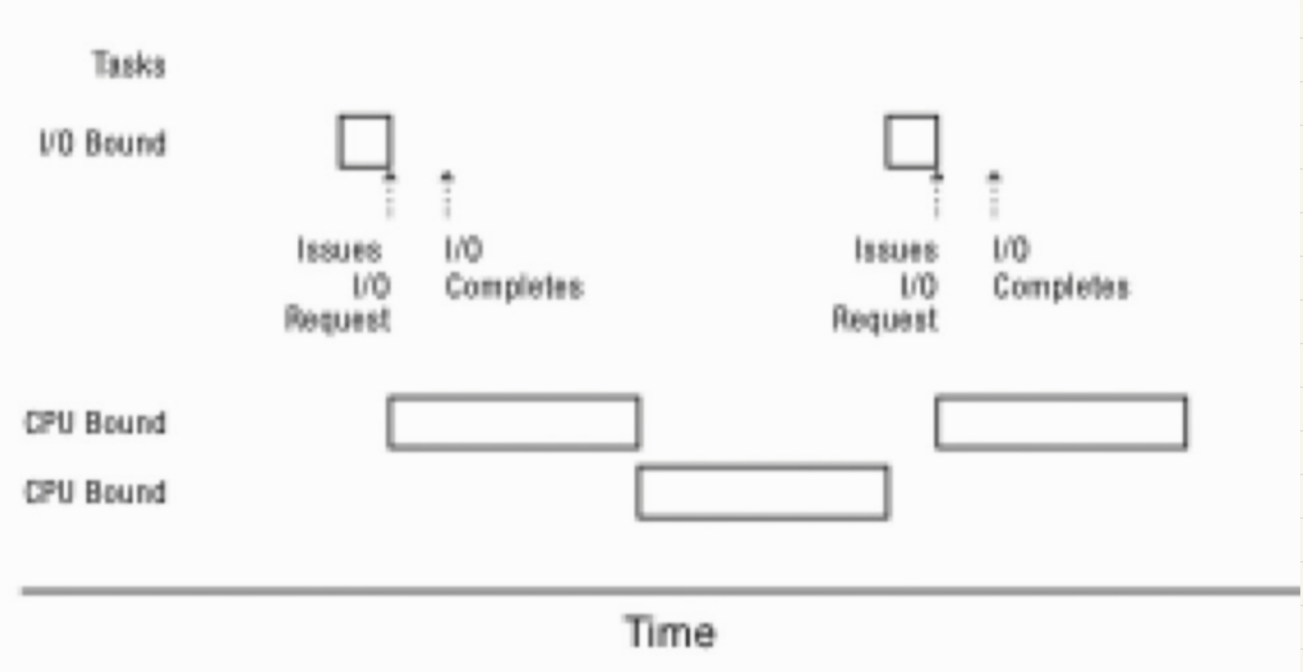
- Assuming zero-cost time slice, is Round Robin always better than FIFO?



Round Robin = Fairness?

- Is Round Robin always fair?
- What is fair?
  - FIFO? ✓
  - Equal share of the CPU? ทำใน 1 time quantum ทุกงาน
  - What if some tasks don't need their full share? ✗ งานไม่ใช้เวลาเต็มก็ไม่เป็นไร ไม่คืนเวลาให้ระบบ
  - Minimize worst case divergence?
    - Time task would take if no one else was running
    - Time task takes under scheduling algorithm

Mixed Workload



Max-Min Fairness

- งานไหนคืนเวลาในระบบทำก่อน แล้วช่วงเวลาที่คืนไป ก็เปลี่ยนไปทำงานตัวอื่น (Time quantum ↑)

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

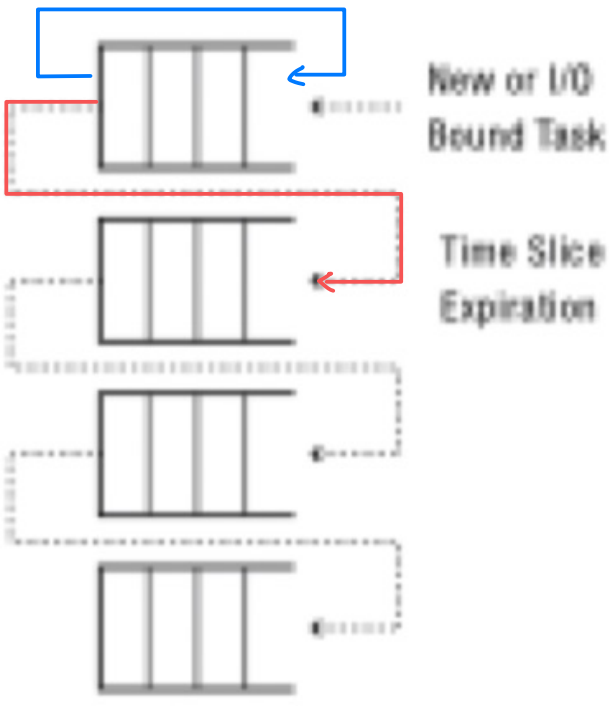
Multi-level Feedback Queue (MFQ)

- Goals:
  - Responsiveness การตอบสนอง
  - Low overhead กระชับ
  - Starvation freedom ไม่เกิด Starvation
  - Some tasks are high/low priority จัดลำดับความสำคัญ
  - Fairness (among equal priority tasks) Fair
- Not perfect at any of them!
  - Used in Linux (and probably Windows, MacOS)
- Set of Round Robin queues
  - Each queue has a separate priority
- High priority queues have short time slices
  - Low priority queues have long time slices
- Scheduler picks first thread in highest priority queue
- Tasks start in highest priority queue
  - If time slice expires, task drops one level

แต่ถ้าทำงานแล้วคืนเวลาให้ระบบจะ
ไม่ลด priority แต่จะกลับมาต่อแถวใหม่

Priority	Time Slice (ms)
1	10
2	20
3	40
4	80

job 1 ทำไม่เสร็จ
ลด priority ไป
ต่อกับ priority 2



Uniprocessor Summary

- FIFO is simple and minimizes overhead.
- If tasks are variable in size, then FIFO can have very poor average response time.
- If tasks are equal in size, FIFO is optimal in terms of average response time.
- Considering only the processor, SJF is optimal in terms of average response time.
- SJF is pessimal in terms of variance in response time.
- If tasks are variable in size, Round Robin approximates SJF.
- If tasks are equal in size, Round Robin will have very poor average response time.
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin.
- Max-Min fairness can improve response time for I/O-bound tasks.
- Round Robin and Max-Min fairness both avoid starvation.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.

Address Translation

Main Points

ยืดหยุ่น
ความเร็ว
เวลาประมวลผล ความซับซ้อน

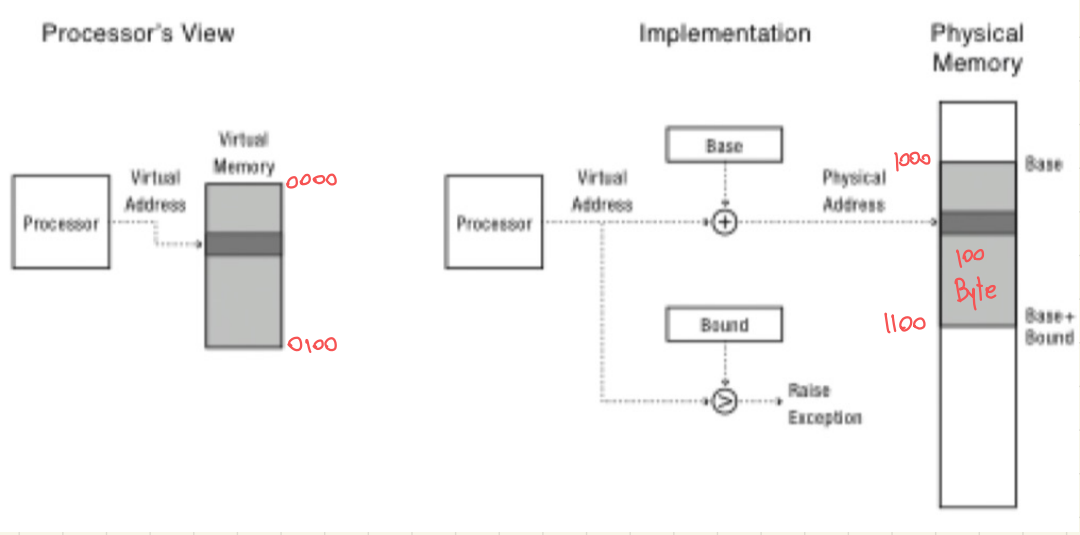
- Address Translation Concept
  - How do we convert a virtual address to a physical address?
- Flexible Address Translation
  - Base and bound
  - Segmentation
  - Paging
  - Multilevel translation
- Efficient Address Translation
  - Translation Lookaside Buffers
  - Virtually and physically addressed caches

Address Translation Goals

- Memory protection
- Memory sharing
  - Shared libraries, interprocess communication
- Sparse addresses
  - Multiple regions of dynamic allocation (heaps/stacks)
- Efficiency
  - Memory placement
  - Runtime lookup
  - Compact translation tables

Virtually Addressed Base and Bounds

ที่ Simple



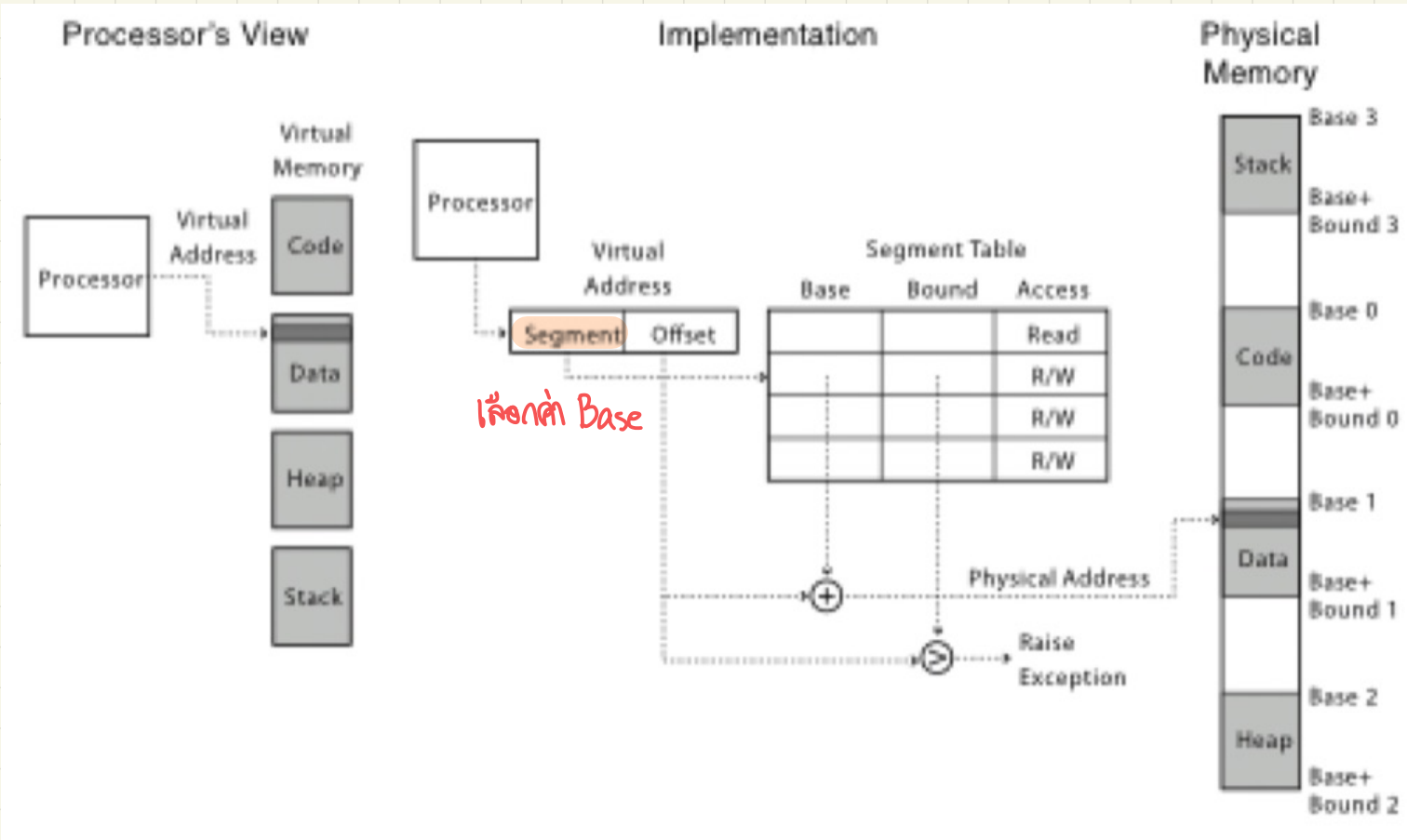
Base = 1000

Bound = 100 (ขนาด)

- Pros?
  - Simple
  - Fast (2 registers, adder, comparator)
  - Safe
  - Can relocate in physical memory without changing process
- Cons?
  - Can't keep program from accidentally overwriting its own code
  - Can't share code/data with other processes
  - Can't grow stack/heap as needed

Segmentation

- Segment is a contiguous region of virtual memory
- Each process has a segment table (in hardware)
  - Entry in table = segment
- Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission
- Processes can share segments
  - Same start, length, same/different access permissions



2 bit segment #
12 bit offset

	Segment start	length
code	0x4000	0x700
data	0	0x500
heap	-	-
stack	0x2000	0x1000

Virtual Memory

main: 240	store #1108, r2
244	store pc+8, r31
248	jump 360
24c	
...	
strlen: 360	loadbyte (r2), r3
...	...
420	jump (r31)
...	
x: 1108	a b c \0
...	

Physical Memory

x: 108	a b c \0
...	
main: 4240	store #1108, r2
4244	store pc+8, r31
4248	jump 360
424c	
...	...
strlen: 4360	loadbyte (r2),r3
...	
4420	jump (r31)
...	

240 2 = 0010

4 = 0100

0 = 0000

001001000000 = 512 + 64 = 576

offset

1108 1 = 0001

1 = 0001

0 = 0000

8 = 1000

0001|0001000001000 = 256 + 8 = 264

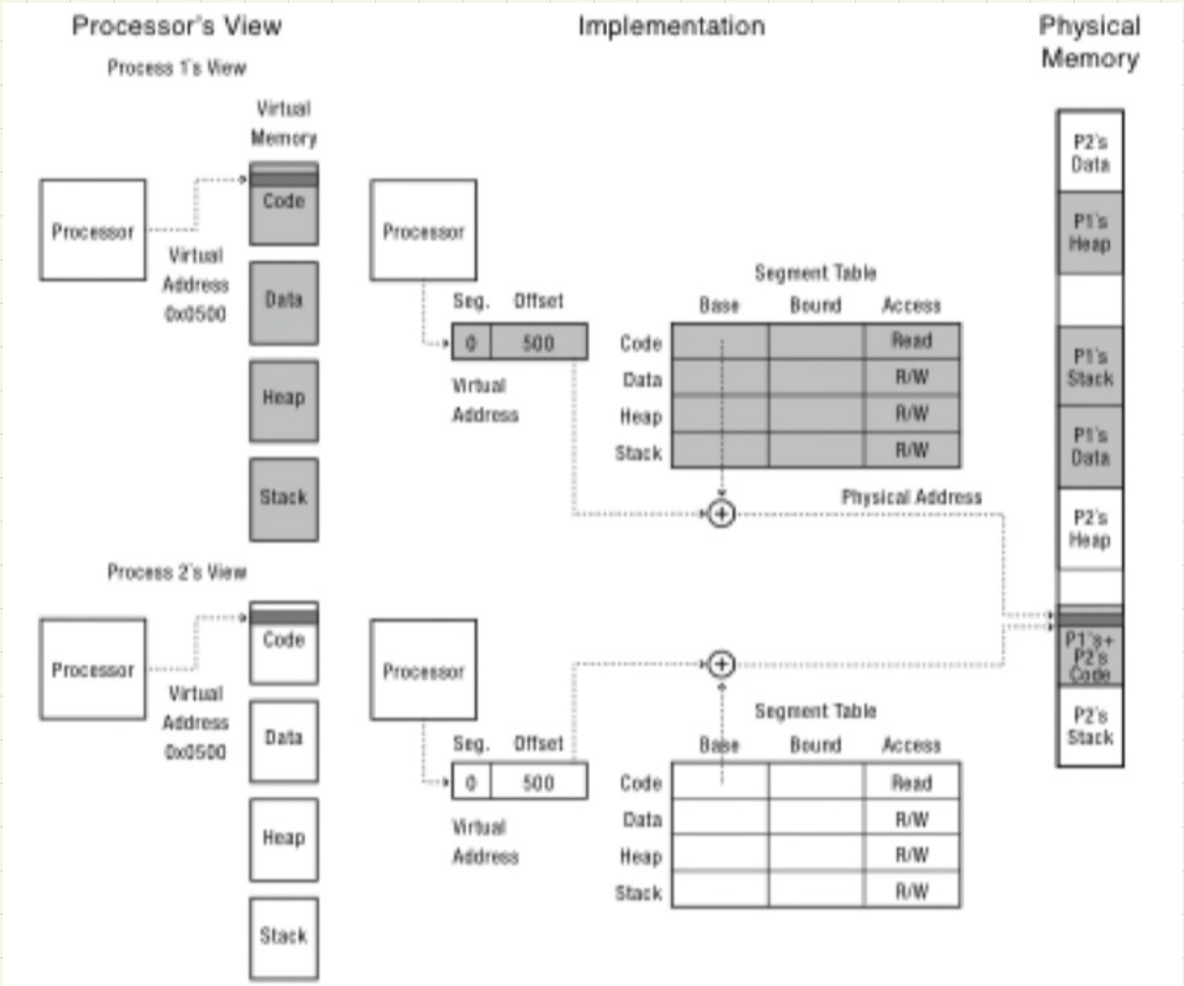
segment offset

360 3 = 0011

6 = 0110

0 = 0000

001101100000 = 864

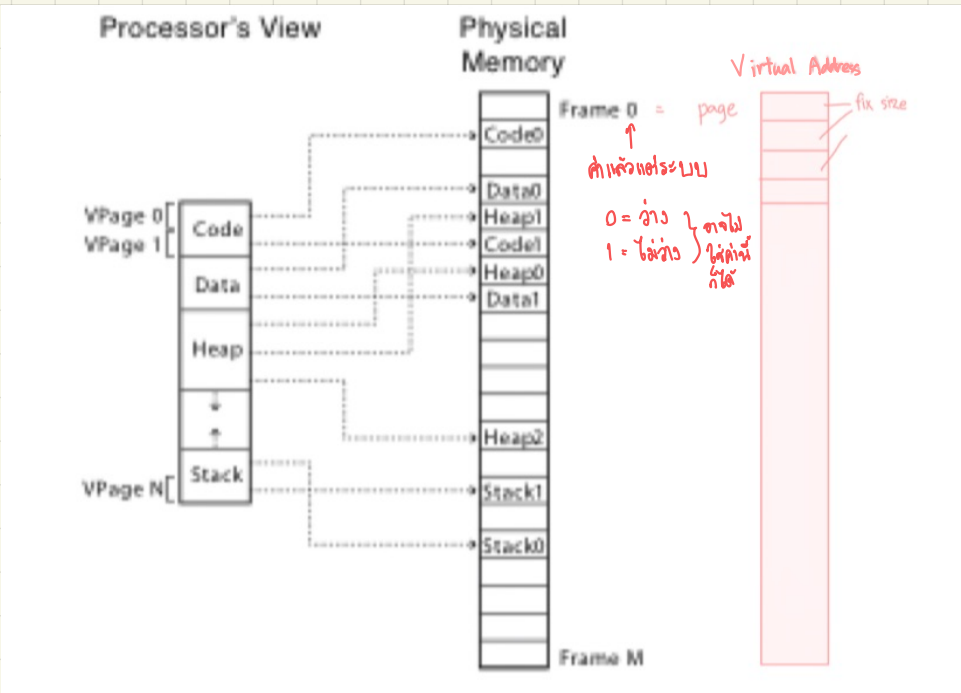


- Pros?
  - Can share code/data segments between processes
  - Can protect code segment from being overwritten
  - Can transparently grow stack/heap as needed
  - Can detect if need to copy-on-write
- Cons?
  - Complex memory management
    - Need to find chunk of a particular size
  - May need to rearrange memory from time to time to make room for new segment or growing segment
    - External fragmentation: wasted space between chunks

Paged Translation

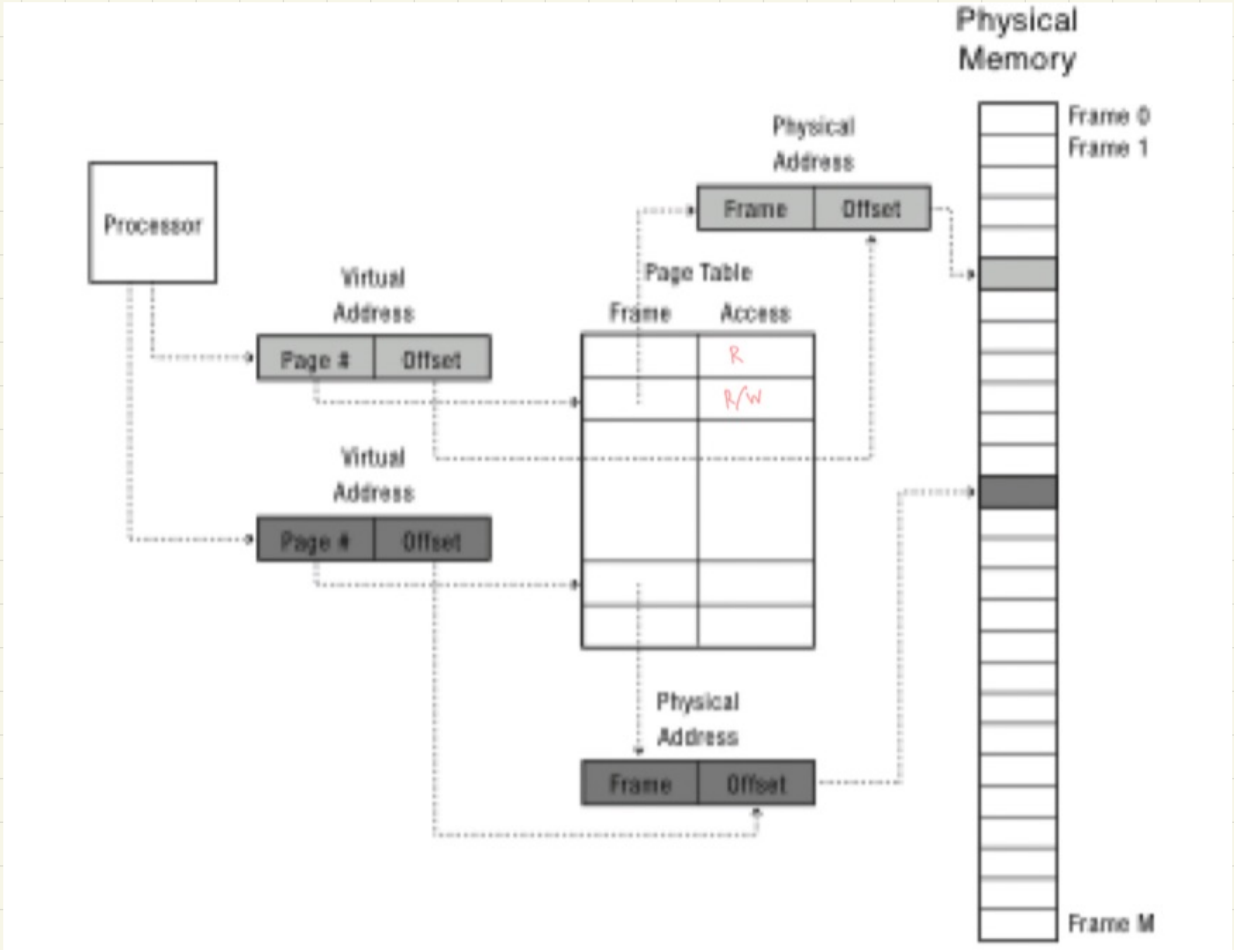
- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
- Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    - pointer to page table start
    - page table length

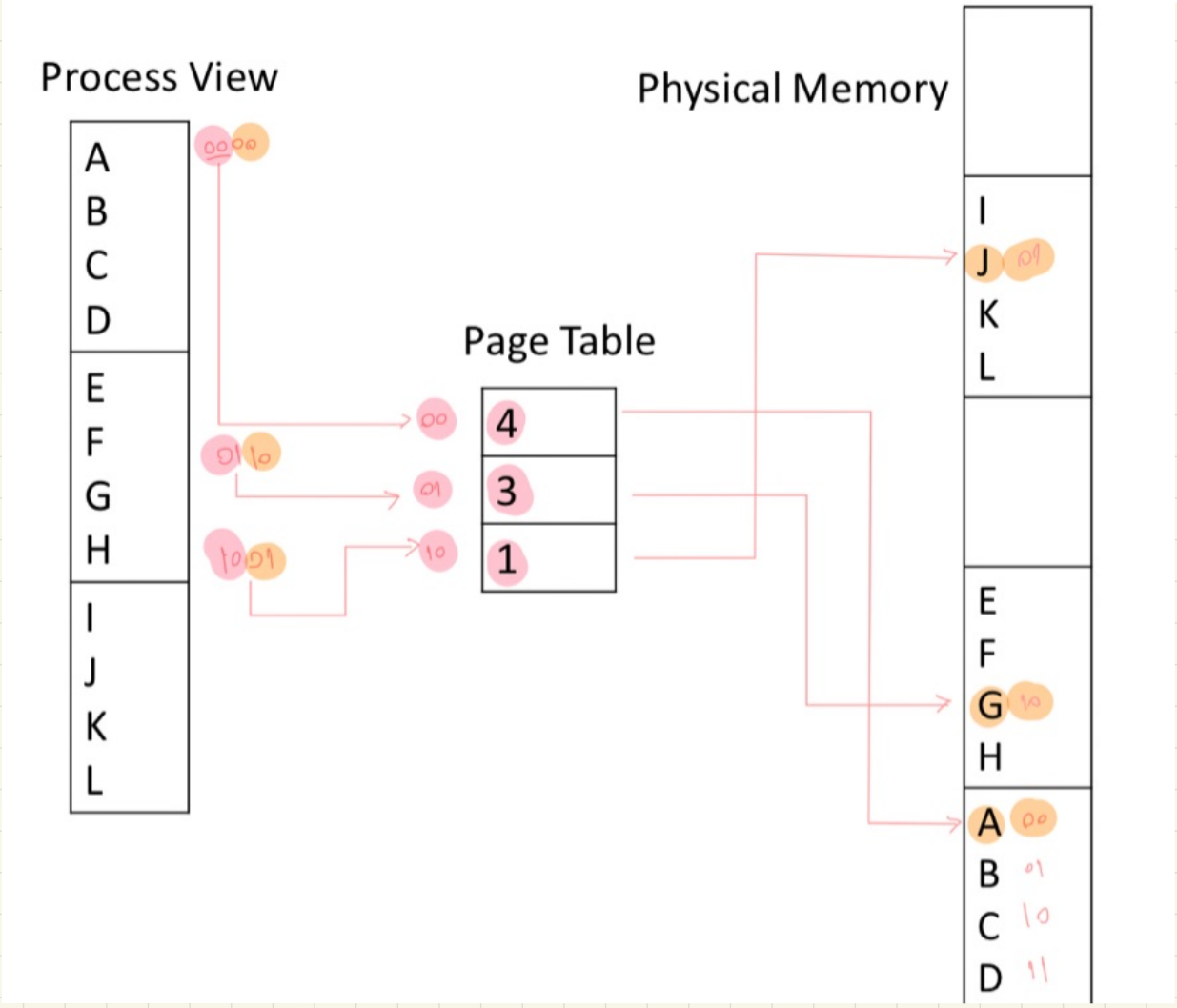
Paged Translation (Abstract)



- Drawback of paging

Paged Translation (Implementation)





Sparse Address Spaces

- Might want many separate dynamic segments
  - Per-processor heaps
  - Per-thread stacks
  - Memory-mapped files
  - Dynamically linked libraries
- What if virtual address space is large?
  - 32-bits, 4KB pages => 500K page table entries
  - 64-bits => 4 quadrillion page table entries

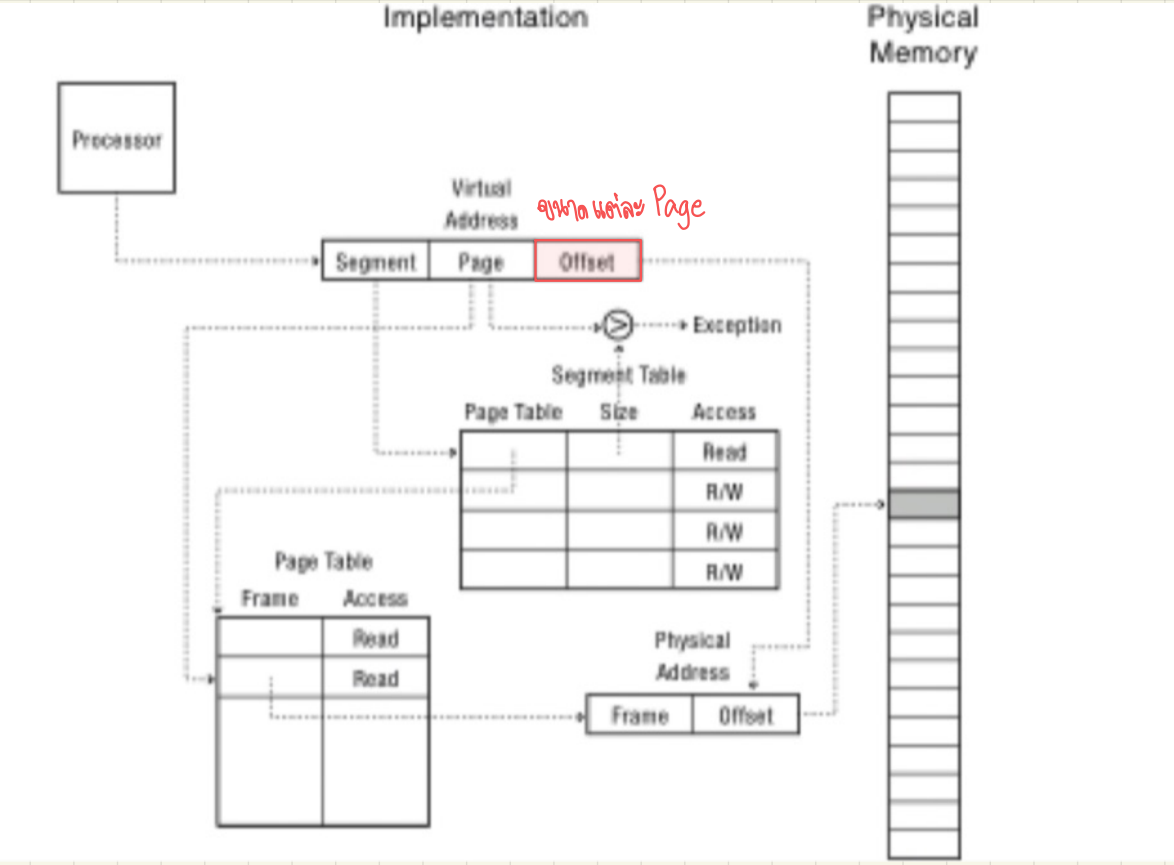
Multi-level Translation

- Tree of translation tables
  - Paged segmentation
  - Multi-level page tables
  - Multi-level paged segmentation
- Fixed-size page as lowest level unit of allocation
  - Efficient memory allocation (compared to segments)
  - Efficient for sparse addresses (compared to paging)
  - Efficient disk transfers (fixed size units)
  - Easier to build translation lookaside buffers
  - Efficient reverse lookup (from physical -> virtual)
  - Variable granularity for protection/sharing

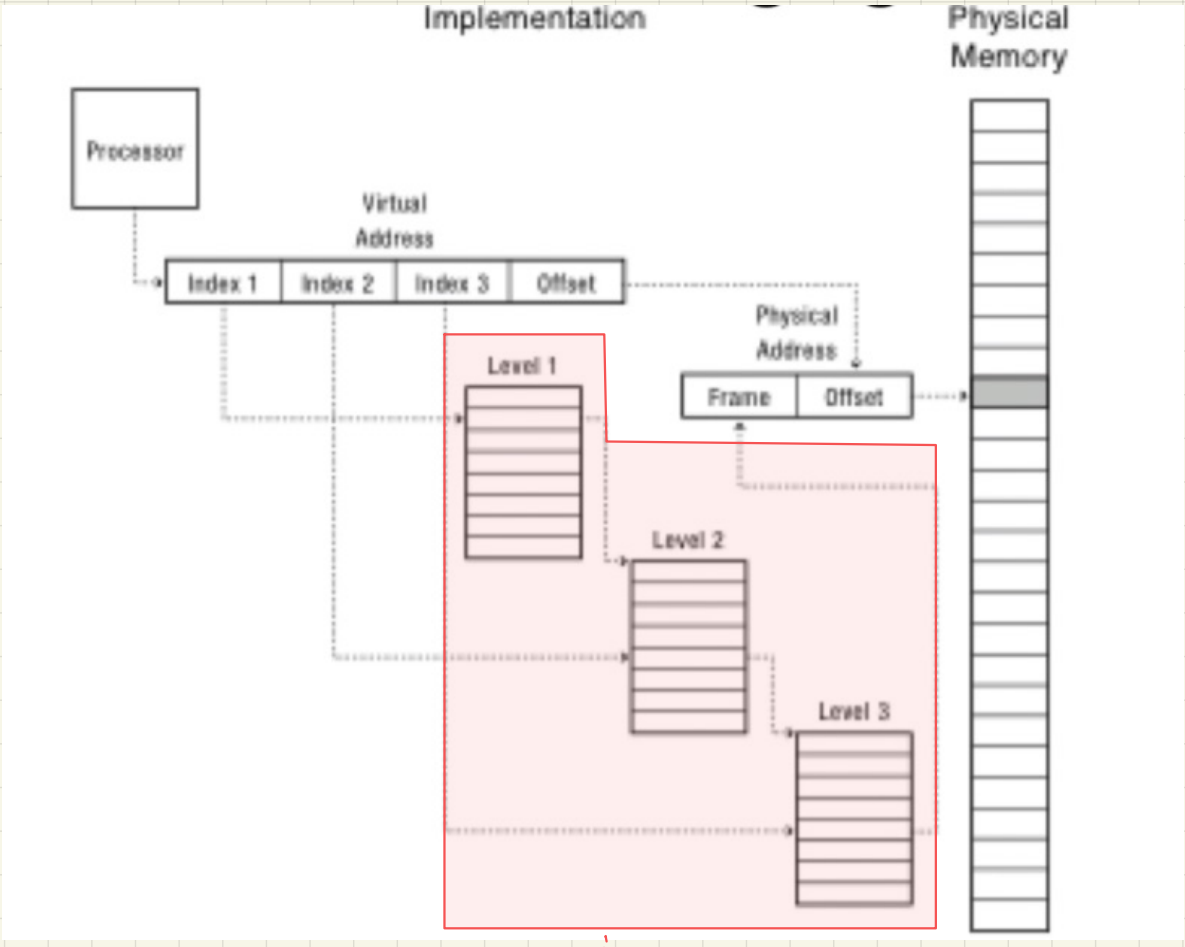
Paged Segmentation

- Process memory is segmented
- Segment table entry:
  - Pointer to page table
  - Page table length (# of pages in segment)
  - Access permissions
- Page table entry:
  - Page frame
  - Access permissions
- Share/protection at either page or segment-level

Paged Segmentation (Implementation)



Multilevel Paging



แบ่งกี่ level ก็ได้

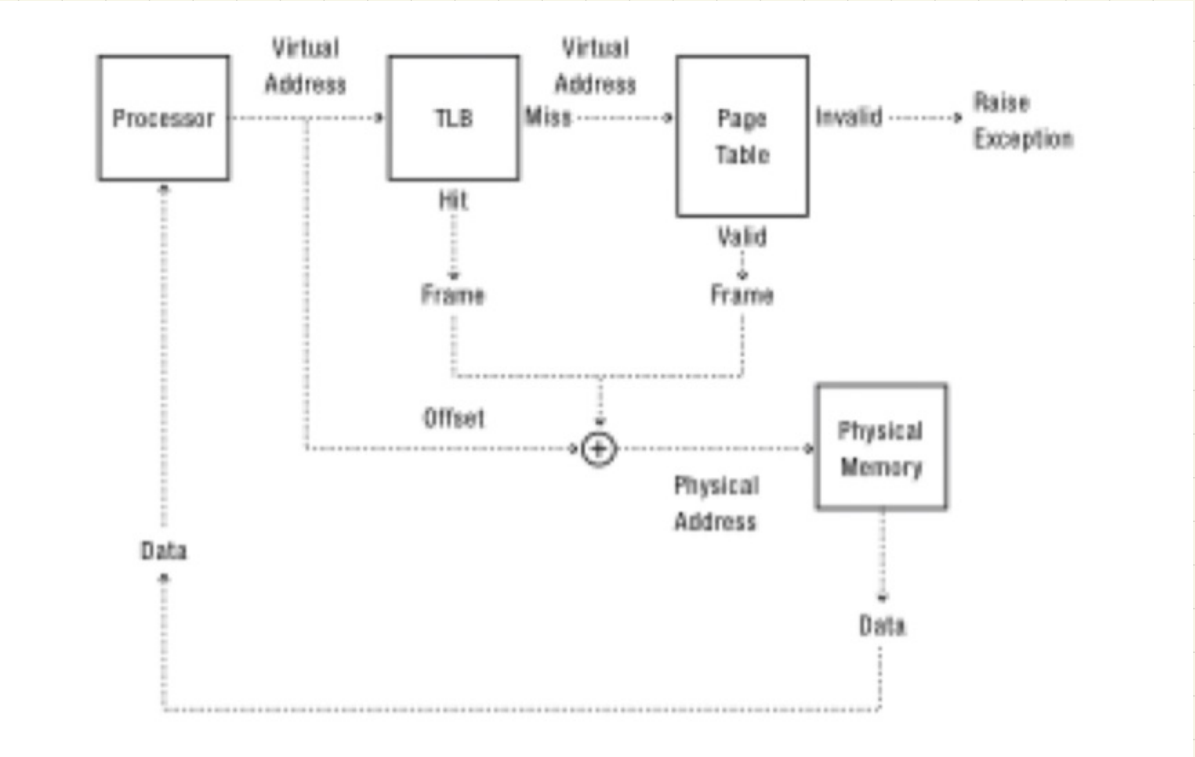
- Pros:
  - Allocate/fill only page table entries that are in use
  - Simple memory allocation
  - Share at segment or page level
- Cons:
  - Space overhead: one pointer per virtual page
  - Two (or more) lookups per memory reference

Efficient Address Translation

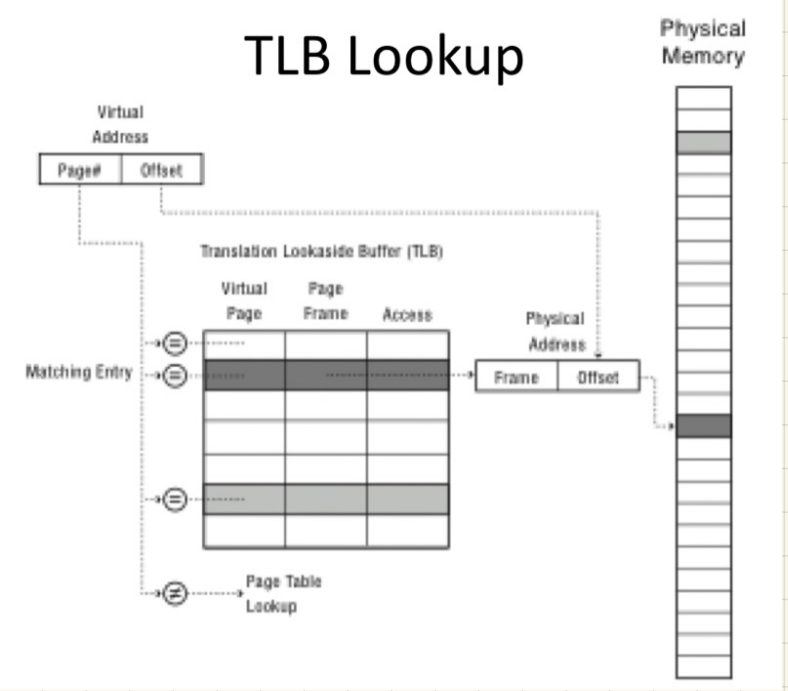
- Translation lookaside buffer (TLB)
  - Cache of recent virtual page -> physical page translations
  - If cache hit, use translation
  - If cache miss, walk multi-level page table
- Cost of translation

= Cost of TLB lookup + Prob(TLB miss) * cost of page table lookup

TLB and Page Table Translation



TLB Lookup



Address Translation Uses

- Process isolation
  - Keep a process from touching anyone else's memory, or the kernel's
- Efficient interprocess communication
  - Shared regions of memory between processes
- Shared code segments
  - E.g., common libraries used by many different programs
- Program initialization
  - Start running a program before it is entirely in memory
- Dynamic memory allocation
  - Allocate and initialize stack/heap pages on demand

Address Translation (more)

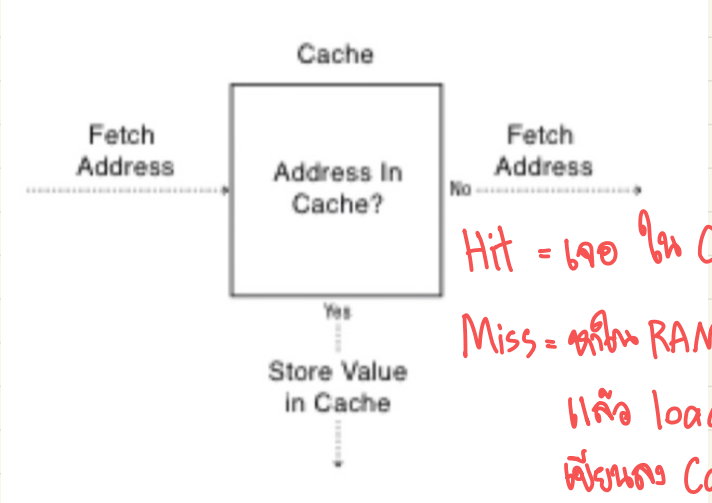
- Cache management
  - Page coloring
- Program debugging
  - Data breakpoints when address is accessed
- Zero-copy I/O
  - Directly from I/O device into/out of user memory
- Memory mapped files
  - Access file data using load/store instructions
- Demand-paged virtual memory
  - Illusion of near-infinite memory, backed by disk or memory on other machines

Virtual Memory

Definitions

- Cache
  - Copy of data that is faster to access than the original (Main memory, Disk) เพราะ Cache มีขนาดเล็กกว่า
  - Hit: if cache has copy
  - Miss: if cache does not have copy
- Cache block
  - Unit of cache storage (multiple memory locations)
- Temporal locality การเรียกข้อมูลซ้ำอีกรอบ
  - Programs tend to reference the same memory locations multiple times
  - Example: instructions in a loop
- Spatial locality การเรียกข้อมูลที่ต้องการ และข้อมูลที่อยู่ติดกัน
  - Programs tend to reference nearby locations
  - Example: data in a loop

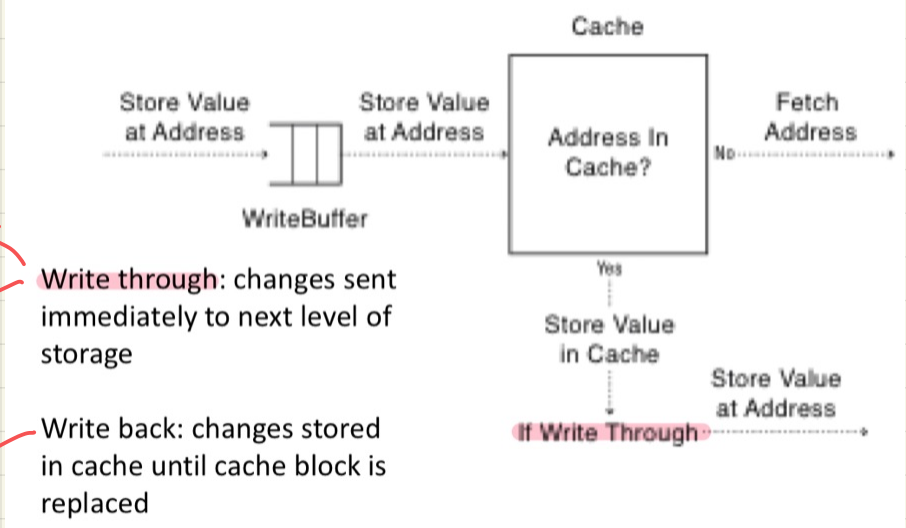
Cache Concept (Read)



Hit = เจอ ใน Cache Block

Miss = หาใน RAM แล้ว load มาเก็บใน Cache

Cache Concept (Write)



เก็บค่าไว้ชั่วคราว

เปลี่ยนแปลงค่าใน RAM ทีหลัง

Write through: changes sent immediately to next level of storage

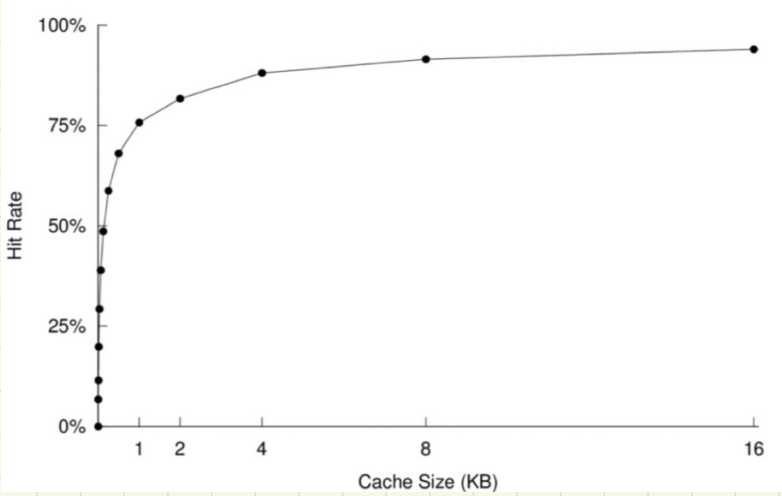
เก็บลอง Dirty Block รอจังหวะก่อน ตามว่าข้อมูลใน Cache จะออกถึงจะบันทึกลง RAM

Write back: changes stored in cache until cache block is replaced

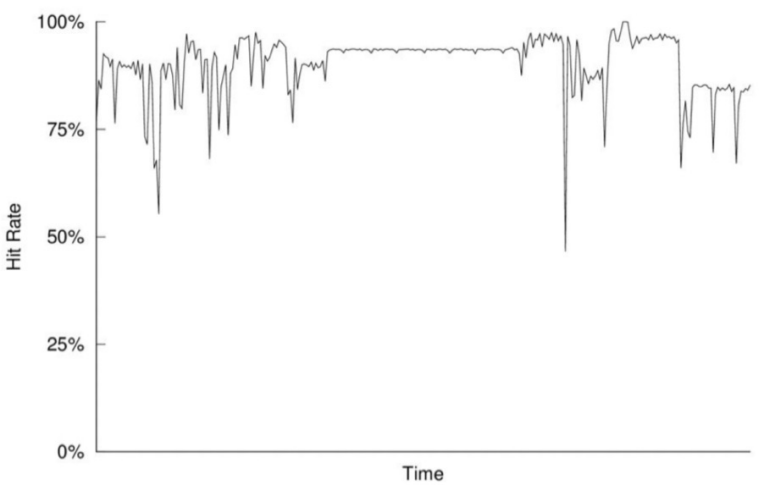
Memory Hierarchy

Cache	Hit Cost	Size
1st level cache/first level TLB	1 ns	64 KB
2nd level cache/second level TLB	4 ns	256 KB
3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data center memory (DRAM)	100 μs	100 TB
Local non-volatile memory	100 μs	100 GB
Local disk	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

Cache hit rate



Example of cache hit rate over time



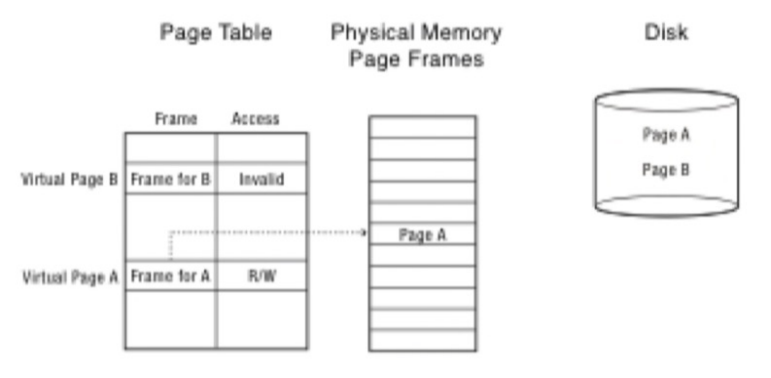
Main Points

- Can we provide the illusion of near infinite memory in limited physical memory?
  - Demand-paged virtual memory
  - Memory-mapped files
- How do we choose which page to replace?
  - FIFO, MIN, LRU, LFU, Clock
- What types of workloads does caching work for, and how well?
  - Spatial/temporal locality vs. Zipf workloads

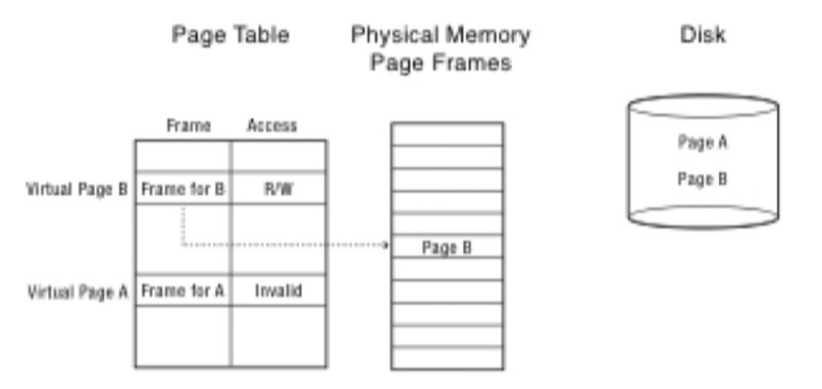
Hardware address translation is a power tool

- Kernel trap on read/write to selected addresses
  - Copy on write
  - Fill on reference
  - Demand paged virtual memory
  - Memory mapped files
  - Modified bit emulation
  - Use bit emulation

Demand Paging (Before)



Demand Paging (After)



Demand Paging

1. TLB miss หาไม่เจอ
2. Page table walk หาใน Page table
3. Page fault (page invalid in page table)

Kernel mode

4. Trap to kernel
5. Convert virtual address to file + offset
6. Allocate page frame หา frame ว่าง
   - Evict page if needed
7. Initiate disk block read into page frame
   copy frame
8. Disk interrupt when DMA complete
9. Mark page as valid ใส่ค่า
10. Resume process at faulting instruction ทำฟังก์ชันได้
11. TLB miss หาไม่เจอ
12. Page table walk to fetch translation หาใหม่
13. Execute instruction เจอละ

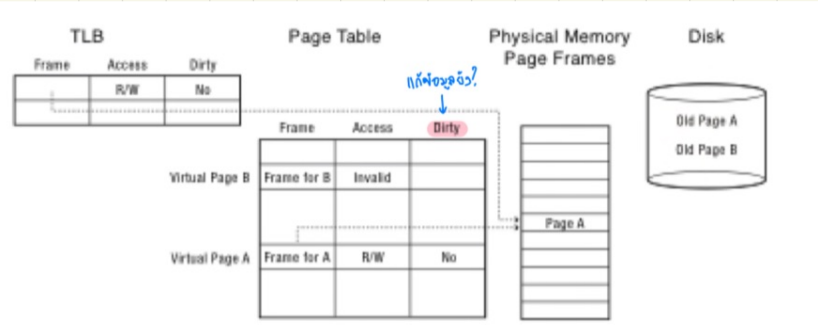
Allocating a Page Frame

- Select old page to evict
- Find all page table entries that refer to old page
  - If page frame is shared
- Set each page table entry to invalid
- Remove any TLB entries
  - Copies of now invalid page table entry
- Write changes on page back to disk, if necessary

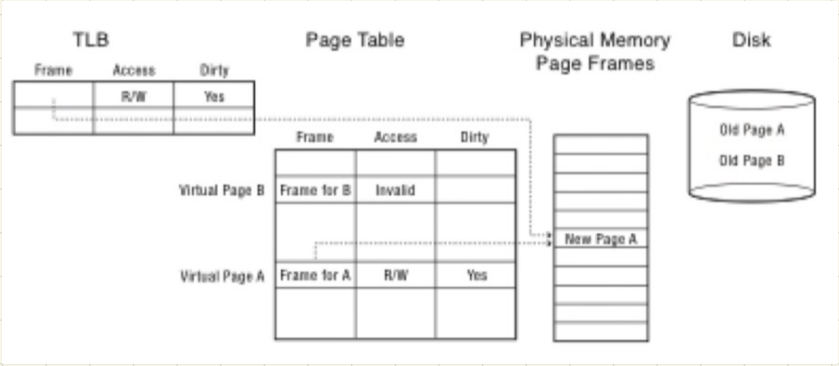
How do we know if page has been modified?

- Every page table entry has some **bookkeeping** ใช้,แก้ไขยัง bit allocation frame ว่างมั้ย?
  - Has page been **modified?**
    - Set by hardware on store instruction
    - In both TLB and page table entry
  - Has page been **recently used?**
    - Set by hardware on in page table entry on every TLB miss
- Bookkeeping bits can be **reset by the OS kernel**
  - When changes to page are flushed to disk
  - To track whether page is recently used

Keeping Track of Page Modifications (Before)



Keeping Track of Page Modifications (After)



Virtual or Physical Dirty/Use Bits

- Most machines keep dirty/use bits in the page table entry
- Physical page is
  - Modified if any page table entry that points to it is modified
  - Recently used if any page table entry that points to it is recently used
- On MIPS, simpler to keep dirty/use bits in the core map
  - Core map: map of physical page frames

Models for Application File I/O

- Explicit read/write system calls
  - Data copied to user process using system call
  - Application operates on data
  - Data copied back to kernel using system call
- Memory-mapped files
  - Open file as a memory segment
  - Program uses load/store instructions on segment memory, implicitly operating on the file
  - Page fault if portion of file is not yet in memory
  - Kernel brings missing blocks into memory, restarts process

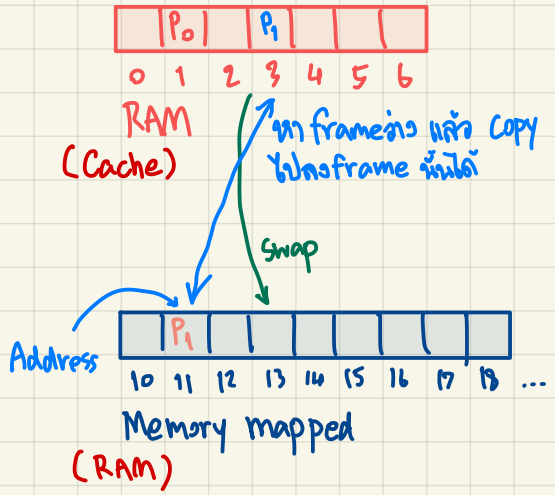
Advantages to Memory-mapped Files ข้อดี

- Programming simplicity, esp for large files
  - Operate directly on file, instead of copy in/copy out
- Zero-copy I/O
  - Data brought from disk directly into page frame
- Pipelining
  - Process can start working before all the pages are populated
- Interprocess communication
  - Shared memory segment vs. temporary file

Cache Replacement Policy

- On a cache miss, how do we choose which entry to replace?
  - Assuming the new entry is more likely to be used in the near future
  - In direct mapped caches, not an issue!
- Policy goal: reduce cache misses ที่ให้ cache น้อยลง
  - Improve expected case performance
  - Also: reduce likelihood of very poor performance สถานการณ์ประสิทธิภาพที่ต่ำมาก

CPU → Cache → RAM(Main Memory) → Memory mapped (Disk)



frame	access
1	R
11 (?) ↓ 3	invalid (ไม่มีใน RAM) ↓ เป็น exception

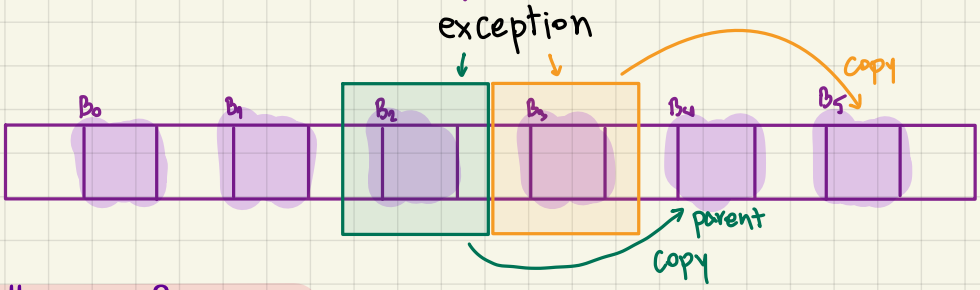
Request ใหม่ ได้แล้ว :>

ถ้า frame เต็มทุกที่ ให้หา frame ที่ไม่เชื่อมไหน แล้วคัดลอกลง memory mapped แล้วค่อยเอา page ใหม่เข้าไป

Copy on write

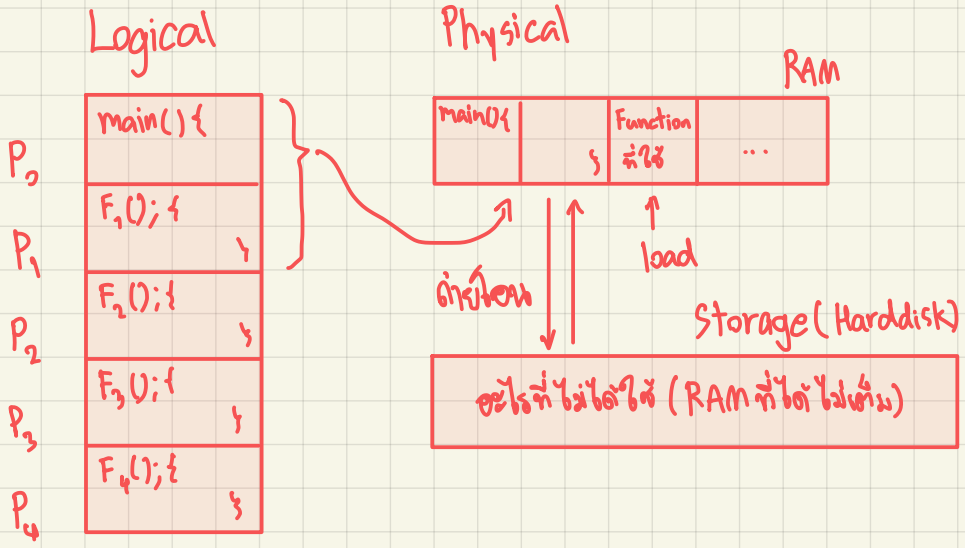
Parent —fork→ Child

segment	B	B0	access	segment	B	B0	access
code	B0	A0	R	code	B0	A0	R
data	B1	A1	R	data	B1	A1	R
W → heap	B2 → 4	A2	R/W	heap	B2	A2	R/W
stack	B3	A3	R	stack	B3 → 5	A3	R ← W



- Ready เร็วขึ้น
- ประหยัด RAM
- R share พท.กันได้ แต่ W ต้อง Copy ก่อน แล้ว Update ตาราง

Fill on reference

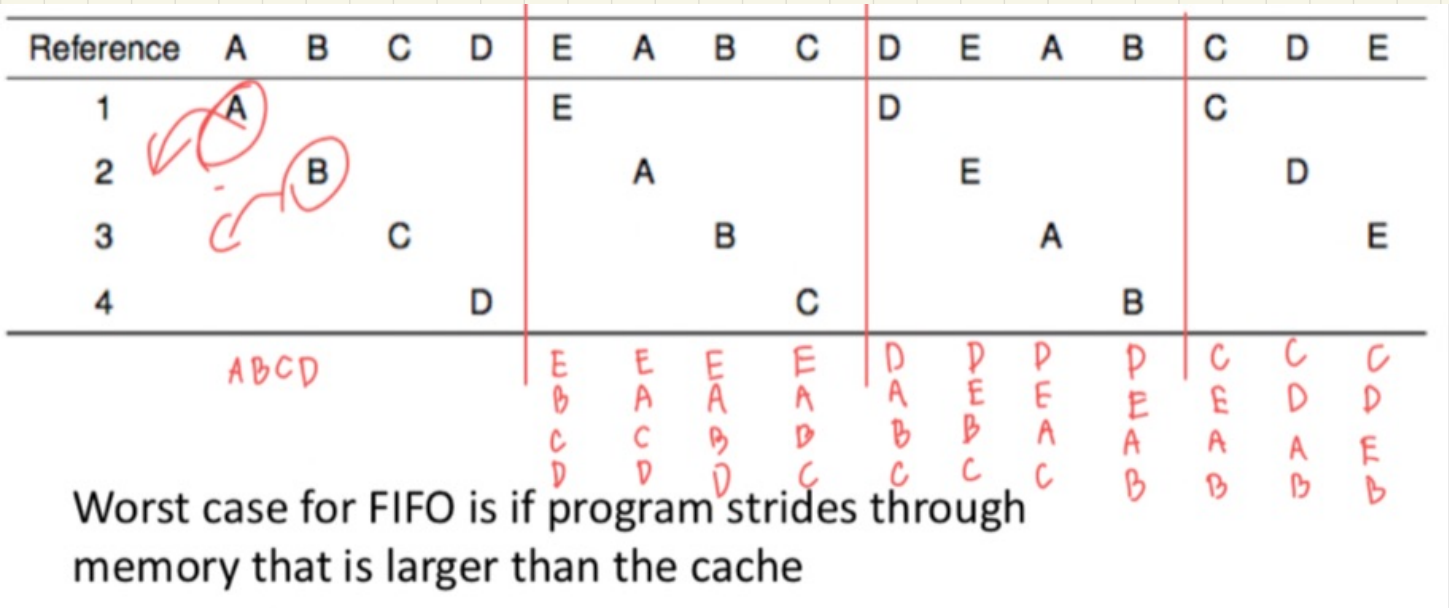


- Ready เร็วขึ้น
- ประหยัด RAM (ถ้ามีหลายฟังก์ชัน ถ้าไม่ load เข้า RAM ยังไงก็ไม่เสีย พท. RAM)
- address transtion
  - pages
- memory mapped file
  - zero copy I/O

A Simple Policy

- Random?
  - Replace a random entry
- FIFO? First In First Out
  - Replace the entry that has been in the cache the longest time
  - What could go wrong?

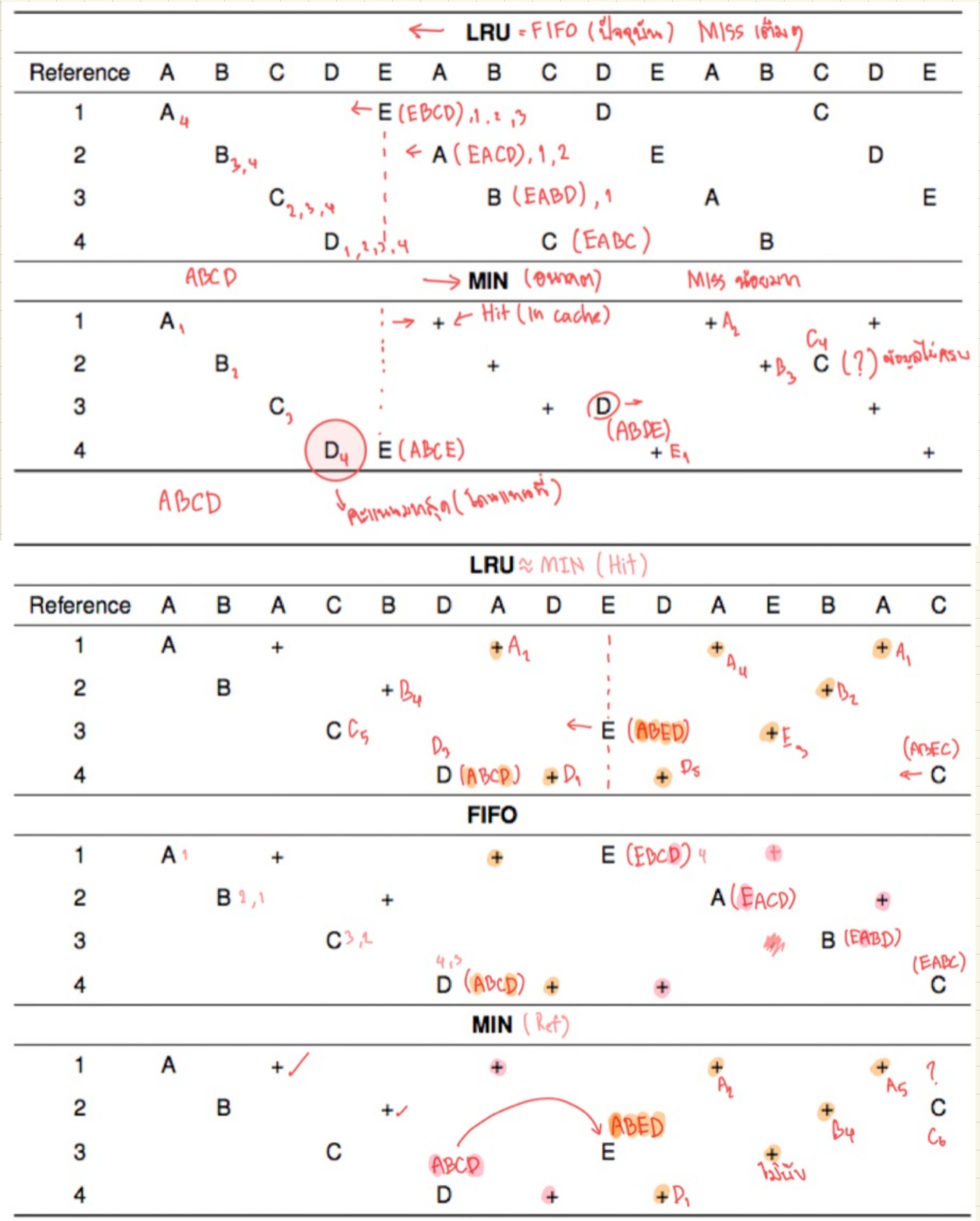
FIFO in Action



MIN, LRU, LFU

- MIN (อุดมคติ) ⟶
  - Replace the cache entry that will not be used for the longest time into the future
  - Optimality proof based on exchange: if evict an entry used sooner, that will trigger an earlier cache miss
- Least Recently Used (LRU) ⟵
  - Replace the cache entry that has not been used for the longest time in the past
  - Approximation of MIN
- Least Frequently Used (LFU)
  - Replace the cache entry used the least often (in the recent past)

LRU/MIN for Sequential Scan



Belady's Anomaly

Cache ↑, Hit ↓

FIFO (3 slots) 3 Hit

Reference	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					+
2		B			A			+		C		
3			C			B			+		D	

FIFO (4 slots) 2 Hit

1	A				+		E				D	
2		B				+		A				E
3			C						B			
4				D						C		

Recap

- MIN is optimal
  - replace the page or cache entry that will be used farthest into the future
- LRU is an approximation of MIN
  - For programs that exhibit spatial and temporal locality