Lec08 virtual memory

1. Coping with complexity
   a) Modularity: split up system, consider separately
   b) Abstaction: interface/hiding, avoid propagation of effects
   c) Layering: gradually build up capabilities
   d) Hierarchy: reduce connections, divide-and-conquer

2. Methods of virtualization
   a) Multiplexing
   b) Aggregating
   c) Emulation

Virtual Resource	Physical Resource	Virtualization Method
Thread (Process)	Processor	Multiplexing
Virtual Memory	Real Memory	Multiplexing
RAID	Disk	Aggregation
RAM Disk	Disk	Emulation

3. Intutive desing
   a) 如何虚拟化？ -> 将 physical memory 划分成 domains
   b) 如何 track domain infos？ -> 用 domain regs 标记
   c) 如何 control access？ -> 添加 memory manager 层，检查 reference 是否合法
   d) Thread 想要通信 -> 每个 thread 可以有多个 domain，包含 shared domain
      -> 将独立段分成 stack(rw)、text(xr)和 data(rw), share(rw)
      -> 添加 permission 的权限位寄存器，包含读、写、执行
   e) 怎么改变 domain reg 和 permission -> 增加 kernel 态，只有特权指令可以做改变
   f) 如何判断指令是否是特权？ -> 用一个 bit 标记 processor 在 user/kernel 态，kernel 态时的指令可以改变 doamin regs； kernel 段也有自己的 domain
   g) 如何 switch mode？ -> thread 只有在 certain address(in kernel text domain) 才可以进入 kernel 态；添加特殊指令 SVC、sret
      -> intel: sysenter, sysexit; AMD: syscall, sysret

Mode Switch

User → Kernel

e.g, svc $2

Step 1. Change processor mode from user to kernel

Step 2. Store current thread context on stack (e.g, PC)

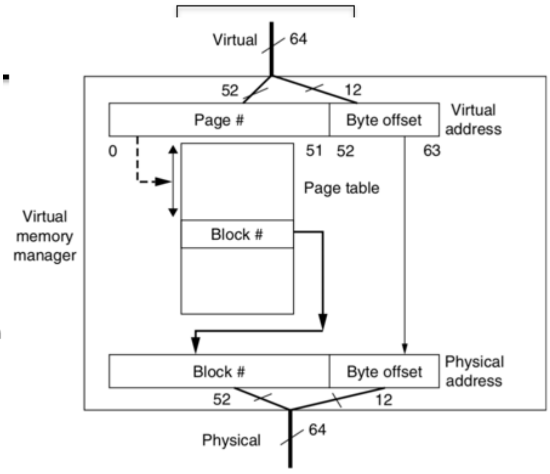
Step 3. Set PC to the start address of gate 2

Kernel → User

e.g, sret

Step 1. Restore the execution context of user mode from stack

Step 2. Change mode from kernel to user



Page Table

Lec09 bounded buffer

1. C/S 架构需要虚拟化和模块化
   a) Program 不能跨域访问 memory -> virtual memory
   b) Progrma 能够互相通信 -> bounded buffer (virtualize communication links)
   c) Progrma 能够共享 CPU -> assume one program per CPU

2. Bounded buffer 需要 kernel 态，应用用 send、receive 这两个 syscall 操作 buffer
   单处理器、单 sender 单 receiver，不 overflow 的实现：

```
send(bb, m):
  while True:
    if bb.in – bb.out < N:
      bb.buf[bb.in mod N] ← m
      bb.in ← bb.in + 1
      return
```

```
receive(bb):
  while True:
    if bb.in > bb.out:
      m ← bb.buf[bb.out mod N]
      bb.out ← bb.out + 1
      return m
```

3. Multiple senders，single receiveer 优化
   核心问题：Data race condition, lost message -> locks
   主要要点：Acquire 锁要在 if 判断前，每条支线都要 release

4. Lock implementation
   - Hardware atomic instructions
     - RSM: read-set-memory
     - Test-and-set, Compare-and-swap
     - Load-linked + Store-conditional, Fetch-and-add
   - Pure software solution
     - Using *load* and *store* instructions only
     - Dekker's & Peterson's Algorithms

   a) Test-and-set

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new; // store 'new' into old_ptr
    return old; // return the old value
}
```

```
void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
    ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

   b) Compare-and-swap

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
    ; // spin-wait (do nothing)
}
```

```
void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

c) Load-linked and store-conditional

```c
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no one has updated *ptr since the LoadLinked
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
   }
}
```

```c
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    // otherwise: try it all over again
    }
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

d) Fetch-and-add for ticket lock

```c
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```c
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

e) Peterson's algorithm

```c
int flag[2]; // assume two threads on two CPUs
int turn;
void init() {
    flag[0] = flag[1] = 0; // 1->thread wants to grab lock
    turn = 0; // whose turn? (thread 0 or 1?)
}
void lock() {
    flag[self] = 1; // self: thread ID of caller
    turn = 1 - self; // make it other thread's turn
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait
}
void unlock() {
    flag[self] = 0; // simply undo your intent
}
```

有一个假设前提：load 和 store 都是原子的，并且要保证 flag 要在 turn 之前改，这在今天的机器上都不一定能够保证了

Lec10 lock

1. Lock performance
   主要是 Lock granularity 粒度和 concurrecy 一致性
   Course-grain 粗粒度只维护很少的锁，simple、easy，less concurrency
   Fine-grain 细粒度维护更多的锁，high amount of concurrency, complex, overheads

2. Example, file system lock，改名操作
   a) Coarse-grained locking，只有一把 fs_lock, forbid possible concurrency
   b) Fine-grained locking，维护 dir_lock
      -> 拿锁放锁的间隙文件仿佛被删除,不能 a1r1 a2r2；顺序问题导致 deadlock

3. Deadlock
   a) 死锁出现的前提
      1. Limited access
         – Resource can only be shared with finite users.
      2. No preemption
         – Once resource granted, cannot be taken away.
      3. Multiple independent requests (hold and wait)
         – Don't ask all at once (wait for next resource while holding current one)
      4. Cycle in wait for graph
   b) 解决死锁的方法
      i. Pessimistic method(lock ordering)：take a priori action to prevent
         给锁标唯一序号，transaction 拿锁要按照顺序，先拿小范围锁，再拿大范围锁，但存在一些 application 无法预测需要哪些锁的问题
      ii. Optimistic method(backing out): detect deadlocks then fix
         按照正常的拿锁，然后检测 deadlock，如果发现就产生一个 victim
         UNDO 放掉 high-num lock，REDO 等待 low-num lock
      iii. Optimistic method(timer expiration)
         设置一个 interval，事务如果超出这个时间就 abort
      iv. Optimistic method(cycle detection)
         维护一张 wait-for-graph，表示 owner 信息和 wait 信息，在事务想拿锁时检测以防止 cycle
   c) File system 在改名上的操作，用 dir.inum 作为排序标准

4. Livelock：context saving/restoring, process the network packets
5. One writer principle

- If each variable has only one writer
  – Coordination becomes easier
  – Concurrency and read-only data is easy
  – Guide: Make as much data as you can have only a single writer
- Privatization: Make data private to a thread
  - Allocate on thread stack
  - Array indexed by thread_id()
    – E.g. privateData[thread_id()]…
- Focus locking scheme on data shared read/write

6. Program should share CPU -> thread for multiplexing the processor
   a) API: suspend(), resume()
   b) When? The yield() system call

YIELD() Implementation

```
yield():
  acquire(t_lock)

  id = cpus[CPU].thread
  threads[id].state = RUNNABLE        } Suspend
  threads[id].sp = SP                 } running thread

  do:
    id = (id + 1) mod N                 } Choose new
  while threads[id].state != RUNNABLE   } thread

  SP = threads[id].sp
  threads[id].state = RUNNING         } Resume new
  cpus[CPU].thread = id               } thread

  release(t_lock)
```

t_lock
1. Atomically set **threads[].state** and **.sp**
2. Atomically find a **RUNNABLE** thread and mark it **RUNNING**

7. Context swtich

要点：Thread layer 和 processor layer

Thread 跑在 thread layer，call YIELD，进入 processor layer，保存 state(general purpose regs + PC + SP + CR3)，选择一个 runnable thread，再进入 thread layer 开始跑

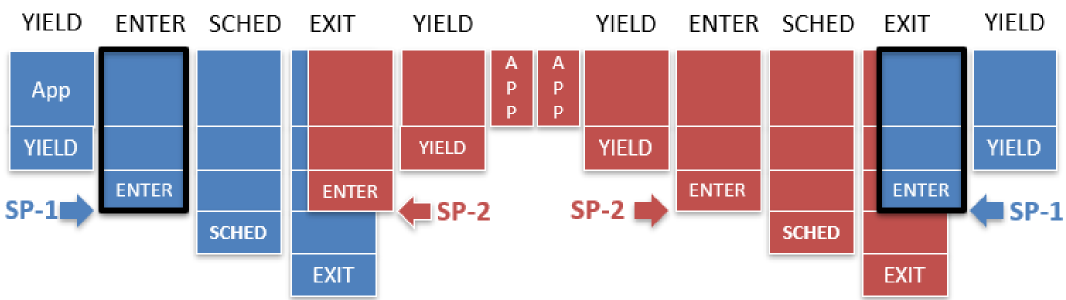
Context Switch

procedure YIELD ()
  ACQUIRE (*thread_table_lock*)
  ENTER_PROCESSOR_LAYER (GET_THREAD_ID())
  RELEASE (*thread_table_lock*)
  return

procedure ENTER_PROCESSOR_LAYER (*this_thread*)
  thread_table[*this_thread*].*state* ← RUNNABLE
  thread_table[*this_thread*].*topstack* ← SP
  SCHEDULER()
  return

procedure EXIT_PROCESSOR_LAYER (*new*)
  SP ← *thread_table*[*new*].*topstack*
  return

procedure SCHEDULER()
  j = GET_THREAD_ID()
  do
    j ← (*j* + 1) **modulo** *7*
  while *thread_table*[*j*].*state* ≠ RUNNABLE
  thread_table[*j*].*state* ← RUNNING
  processor_table[CPUID].*thread_id* ← *j*
  EXIT_PROCESSOR_LAYER (*j*)
  return

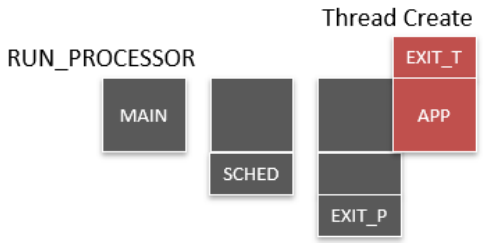


RUN_PROCESSOR & Create Thread

```
procedure RUN_PROCESSORS ()
    for each processor do
        allocate stack and set up a processor thread
        shutdown ← FALSE
        SCHEDULER ()
        deallocate processor thread stack
        halt processor
```

```
procedure SCHEDULER ()
    while shutdown = FALSE do
        ACQUIRE (thread_table_lock)
        for i from 0 until 7 do
            if thread_table[i].state = RUNNABLE then
                thread_table[i].state ← RUNNING
                processor_table[CPUID].thread_id ← i
                EXIT_PROCESSOR_LAYER (CPUID, i)
                if thread_table[i].kill_or_continue = KILL then
                    thread_table[i].state ← FREE
                    DEALLOCATE(thread_table[i].stack)
                    thread_table[i].kill_or_continue = CONTINUE
        RELEASE (thread_table_lock)
    return                                              // Go shut dow
```

```
procedure EXIT_PROCESSOR_LAYER (processor, tid)  //
    processor_table[processor].topstack ← SP      //
    SP ← thread_table[tid].topstack               //
    return
```



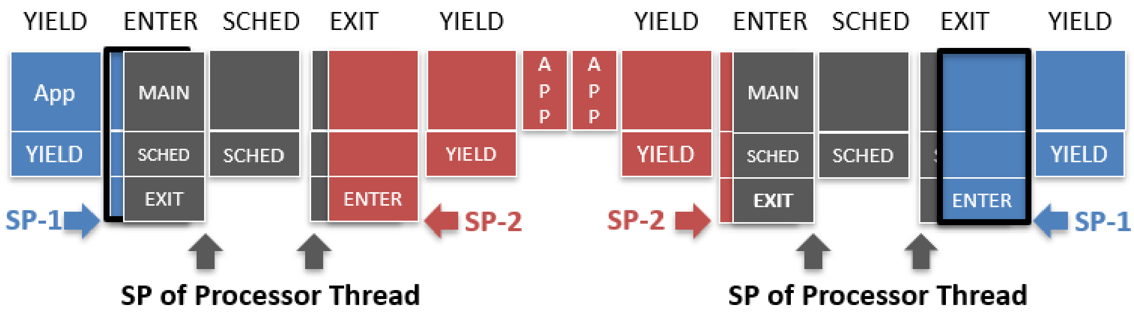
Context Switch

```
procedure YIELD ()
    ACQUIRE (thread_table_lock)
    ENTER_PROCESSOR_LAYER (GET_THREAD_ID(), CPUID)
    RELEASE (thread_table_lock)
    return
```

```
procedure ENTER_PROCESSOR_LAYER (tid, processor)
    thread_table[tid].state ← RUNNABLE
    thread_table[tid].topstack ← SP                  //
    SP ← processor_table[processor].topstack         //
    return
```

```
procedure EXIT_PROCESSOR_LAYER (processor, tid)  //
    processor_table[processor].topstack ← SP      //
    SP ← thread_table[tid].topstack               //
    return
```

```
procedure SCHEDULER ()
    while shutdown = FALSE do
        ACQUIRE (thread_table_lock)
        for i from 0 until 7 do
            if thread_table[i].state = RUNNABLE then
                thread_table[i].state ← RUNNING
                processor_table[CPUID].thread_id ← i
                EXIT_PROCESSOR_LAYER (CPUID, i)
                if thread_table[i].kill_or_continue = KILL then
                    thread_table[i].state ← FREE
                    DEALLOCATE(thread_table[i].stack)
                    thread_table[i].kill_or_continue = CONTINUE
        RELEASE (thread_table_lock)
    return                                              // Go shut dow
```



Lec11 condition variable

1. Bounded buffer send/receive

   API：wait(cv, lock)　release lock, yield CPU, wait to be notify

   　　notify(cv)　notify waiting threads of cv

```
send(bb, msg):
  acquire(bb.lock)
  while True:
    if bb.in - bb.out < N:
      bb.buf[bb.in mod N] <- msg
      bb.in <- bb.in + 1
      release(bb.lock)
      return
```

```
send(bb, msg):
  acquire(bb.lock)
  if bb.in - bb.out >= N:
    release(bb.lock)
    wait(bb.not_full)
    acquire(bb.lock)
  bb.buf[bb.in mod N] <- msg
  bb.in <- bb.in + 1
  release(bb.lock)
  notify(bb.not_empty)
  return
```

```
send(bb, msg):
  acquire(bb.lock)
  while True:
    if bb.in - bb.out < N:
      bb.buf[bb.in mod N] <- msg
      bb.in <- bb.in + 1
      release(bb.lock)
      return
    release(bb.lock)
    yield()
    acquire(bb.lock)
```

```
send(bb, msg):
  acquire(bb.lock)
  while True:
    if bb.in - bb.out < N:
      bb.buf[bb.in mod N] <- msg
      bb.in <- bb.in + 1
      release(bb.lock)
      notify(bb.not_empty)
      return
  wait(bb.not_full, bb.lock)
```

   a) Data race -> 加锁；　b) 反复检查 if 空转浪费资源 -> yield 掉 CPU
   c) 被 yield 掉的线程拿着锁 -> yield 要放锁；
   d) d) yield 频繁拿放锁性能差 -> 引入新的 API，wait 和 notify
   e) 番外，去掉 while 循环的 condition variable -> 多个 sender 可能会被同时唤醒，造成 if 条件判断过时
   f) Lost notify problem, release lock 后，wait 前，receive 发出的 notify 接收不到 -> 将 lock 作为 wait 的参数，拿放锁成为原子操作

2. Wait implementation

```
wait(cv, lock):
    acquire(t_lock)
    release(lock)
    threads[id].cv = cv
    threads[id].state = WAITING
    yield_wait()
    release(t_lock)
    acquire(lock)

notify(cv):
    acquire(t_lock)
    for I = 0 to N-1:
        if threads[i].cv == cv && threads[i].state == WAITING:
            threads[i].state = RUNNABLE
    release(t_lock)
```

3. Yield_wait() implementation

```
yield_wait(): // called by wait()
  ~~acquire(t_lock)~~

  id = cpus[CPU].thread
  ~~threads[id].state = RUNNABLE~~
  threads[id].sp = SP

  do:
    id = (id + 1) mod N
  while threads[id].state != RUNNABLE

  SP = threads[id].sp
  threads[id].state = RUNNING
  cpus[CPU].thread = id

  ~~release(t_lock)~~
```

```
yield_wait(): // called by wait()
  id = cpus[CPU].thread
  threads[id].sp = SP
  SP = cpus[CPU].stack

  do:
    id = (id + 1) mod N
    release(t_lock)
    acquire(t_lock)
  while threads[id].state != RUNNABLE

  SP = threads[id].sp
  threads[id].state = RUNNING
  cpus[CPU].thread = id
```

a) Acquire/state/release 都在 wait 中执行，yield_wait 中删去
b) 如果找不到 runnable 的 thread，会在拿锁的情况下空转，其他 thread 无法 notify
   -> 每次循环都放一下锁(等锁的 thread 通常能拿到刚释放的锁)
c) loop 中放锁拿锁的间隙，原 thread 可能被 notify 从 waiting 变成 runnable，另一个 CPU 可能会拿到锁调用 wait 调度到原线程，由于原线程的 SP 还没有改(在 SP=threads[id].sp 改)，这个时候会有两个 CPU 共用一段栈

4. Preemptive scheduling and interrupt
   Types：Non-preemptive, cooperative(call yield periodically), preemptive
   抢占是通过 interrupt 实现的

```
timer_interrupt():

  push PC          // done by CPU
  push registers

  yield()

  pop registers
  pop PC
```

```
yield_wait(): // called by wait()
  id = cpus[CPU].thread
  threads[id].sp = SP
  SP = cpus[CPU].stack

  do:
    id = (id + 1) mod N
    release(t_lock)
    enable_interrupt()
    disable_interrupt()
    acquire(t_lock)
  while threads[id].state != RUNNABLE

  SP = threads[id].sp
  threads[id].state = RUNNING
  cpus[CPU].thread = id
```

YIELD_WAIT()

```
wait(cv, lock):
    disable_interrupt()
    acquire(t_lock)
    release(lock)
    threads[id].cv = cv
    threads[id].state = WAITING
    yield_wait()
    release(t_lock)
    enable_interrupt()
    acquire(lock)
```

```
yield_wait(): // called by wait()
  id = cpus[CPU].thread
  cpus[CPU].thread = null
  threads[id].sp = SP
  SP = cpus[CPU].stack

  do:
    id = (id + 1) mod N
    release(t_lock)
    enable_interrupt()
    disable_interrupt()
    acquire(t_lock)
  while threads[id].state != RUNNABLE

  SP = threads[id].sp
  threads[id].state = RUNNING
  cpus[CPU].thread = id
```

```
yield(): // called by interrupt handler
  acquire(t_lock)
  id = cpus[CPU].thread
  if (id == null)
    release(t_lock)
    return
  threads[id].state = RUNNABLE
  threads[id].sp = SP

  do:
    id = (id + 1) mod N
  while threads[id].state != RUNNABLE

  SP = threads[id].sp
  threads[id].state = RUNNING
  cpus[CPU].thread = id

  release(t_lock)
```

a) 如果执行 yield_wait 过程中被中断了，那这个 thread 就会拿两次锁,hang lock

-> 不允许中断

b) 在打开中断开关的间隙调用中断，会改变 threads[id]的状态，waiting 到 runnable

-> 会让错误的 thread sleep。提前检测，如果在 wait，就不用 yield 了

5. Eventcount & sequencer

New API: AWAIT(eventcount, value), ADVANCE(eventcount)

TICKET(sequencer), READ(eventcount or seqencer)

Implementation

```
procedure AWAIT (eventcount reference event, value)
    ACQUIRE (thread_table_lock)
    id ← GET_THREAD_ID ()
    thread_table[id].event ← event
    thread_table[id].value ← value
    if event.count ≤ value then thread_table[id].state ← WAITING
    ENTER_PROCESSOR_LAYER (id, CPUID)
    RELEASE (thread_table_lock)
```

```
procedure ADVANCE (eventcount reference event)
    ACQUIRE (thread_table_lock)
    event.count ← event.count + 1
    for i from 0 until 7 do
        if thread_table[i].state = WAITING and thread_table[i].event = event and
                                       event.count > thread_table[i].value then
            thread_table[i].state ← RUNNABLE
    RELEASE (thread_table_lock)
```

```
procedure TICKET (sequencer reference s)
    ACQUIRE (thread_table_lock)
    t ← s.ticket
    s.ticket ← t + 1
    RELEASE (thread_table_lock)
    return t
```

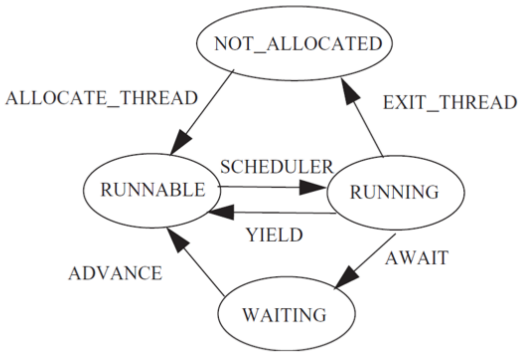
```
procedure READ (eventcount reference event)
    ACQUIRE (thread_table_lock)
    e ← event.count
    RELEASE (thread_table_lock)
    return e
```

```
procedure SEND (buffer reference p, message
    AWAIT (p.out, p.in − N)
    p.message[READ(p.in) modulo N] ← msg
    ADVANCE (p.in)

procedure RECEIVE (buffer reference p)
    AWAIT (p.in, p.out)
    msg ← p.message[READ(p.out) modulo N]
    ADVANCE (p.out)
    return msg
```

⇩

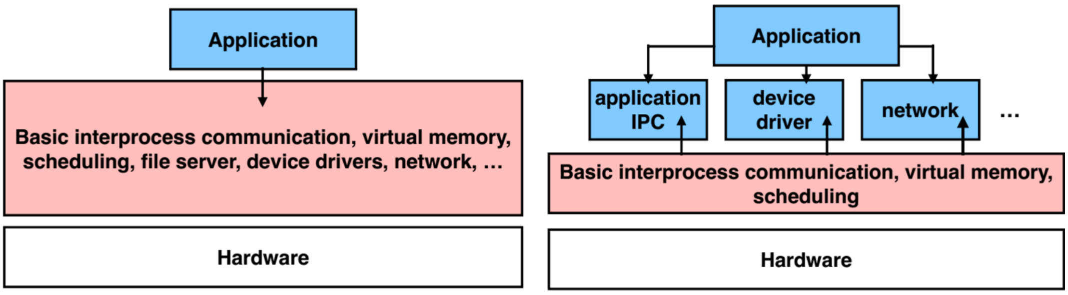
```
procedure SEND (buffer reference p, message
    t ← TICKET (p.sender)
    AWAIT (p.in, t)
    AWAIT (p.out, READ(p.in) − N)
    p.message[READ(p.in) modulo N] ← msg
    ADVANCE (p.in)
```



Lec12 OS structures

1. Eventcounter & sequencer
   a) 跟 condition variable 比起来，await 和 advance 是有状态的(event 和 value)，所以不需要 notify 操作
   b) 通过 t_lock 的保护也避免了 lost notifiction 的问题
   c) 由于 ticket 的存在，每次只有一个 thread 会被唤醒，在 send 中省去了空转。但这样稳定性不够好。

2. OS structure
   a) Monolithic kernel 整体的、庞大的
   代表：Linux，GNU，X window system
   no enfored modularity，kernel bug 会影响这个系统
   优点：relatively few crossing; shared kernel address space; performance
   缺点：flexibility; stability; experimentation

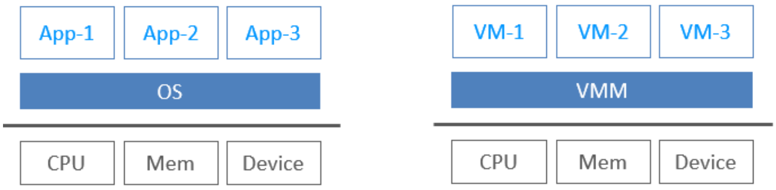


   b) Microkernel 微内核
   代表：Mach，L4
   Enforce modularity by putting subsystem in user space
   优点：easier to develop services; fault isolation; customization 用户化，small kernel and easier to optimize
   缺点：lots of boundary crossing 跨域，relatively poor performance

1. The system is unusable if a critical service fails
   – No matter in user mode or kernel mode
2. Some services are shared among many modules
   – It's easier to implement these services as part of the kernel program, which is already shared among all modules
3. The performance of some services is critical
   – E.g., the overhead of SEND and RECEIVE supervisor calls may be too large
4. Monolithic systems can enjoy the ease of debugging of microkernel systems
   – good kernel debugging tools
5. It may be difficult to reorganize existing kernel programs
   – There is little incentive to change a kernel program that already works
   – If the system works and most of the errors have been eradicated
     • the debugging advantage of microkernel begins to evaporate
     • the cost of SEND and RECEIVE supervisor calls begins to dominate

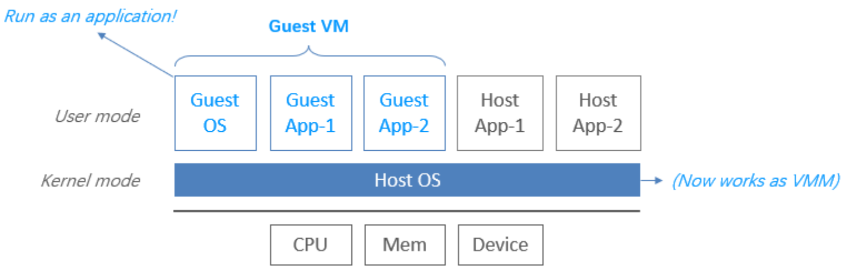
Lec13 virtual machine

1. Compare OS & VMM
   a) Similarities: multiplex hardware, higher privilege
   b) Difference: different abstraction, VMM schedules VMs, OS schedules process



- CPU virtualization
  - Enable each guest VM has its own kernel and user modes
  - Keep isolation between guest's kernel and user modes
- Memory virtualization
  - Enable each guest VM has its own virtual MMU
  - Keep isolation between guest VMs
- I/O virtualization
  - Enable each guest VM has its own virtual devices

2. CPU Virtualization
   所有 process 都以为拥有整个 CPU -> Run as application



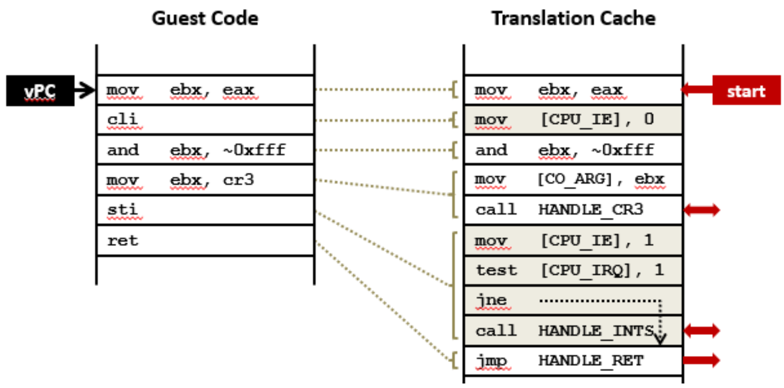
- **Trap**: running privilege instructions in user-mode will trap to the VMM
- **Emulate**: those instructions are implemented as *functions* in the VMM
  - System states are kept in VMM's memory, and are changed according

a) 问题：有一些特权指令不能在 user mode 跑，cli(disable interrupt), change CR3
   解决：trap & emulate

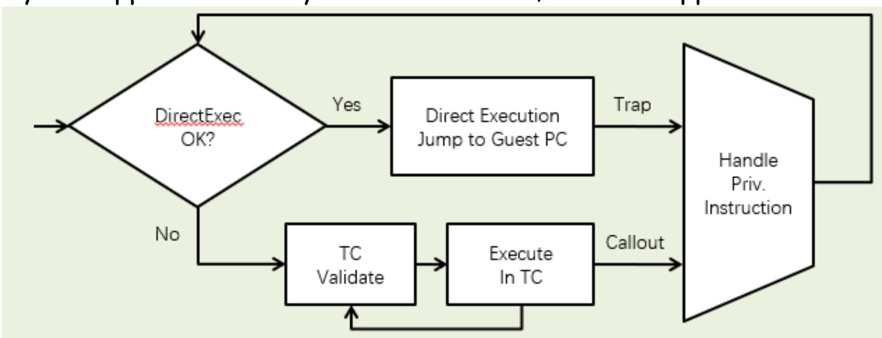
b) 问题：并不是所有架构都是 strictly virtualizable 的；
   X86 中有 17 条指令在 user 态和 kernel 态都可以执行但意义不同，无法 T&E
   1. **Instruction Interpretation**: emulate them by software
   2. **Binary translation**: translate them to other instructions
   3. **Para-virtualization**: replace them in the source code
   4. **New hardware**: change the CPU

   解决：
   i. Instrction interpretation
      用 memory 模拟所有 system status，guest instruction 不直接在硬件执行
      很容易生成，比如 Bochs，但是太慢

ii. Binary translation
方法：Translate 17 instruction to function call, inline 代码
代表: Vmware, Qemu

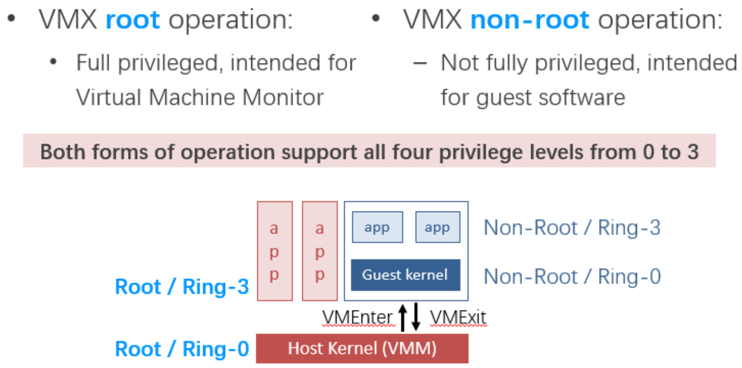


Hybrid approach: Binary trans for kernel, T&E for applications



iii. Para-virtualization 并行虚拟化
Hypercall，虚拟环境中类似 syscall 的操作，切换 mode
方法： Modify OS and let it cooperate with VMM

iv. New hardware

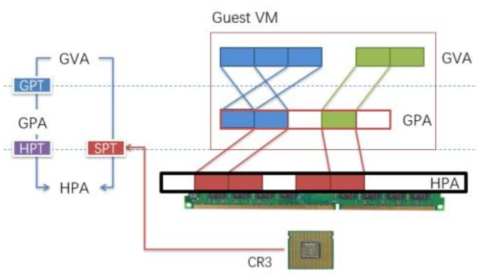


3. Memory virtualization
专业术语：3 种 address： GVA – GPA – HPA
直接将 CR3 指向 GPT 是没用的

– Solution-1: **shadow paging**
– Solution-2: **direct paging**
– Solution-3: **new hardware**

a) shadow paging

shadow PT per application，guest PT per application, host PT per VM




```
set_cr3 (guest_page_table):
    for GVA in 0 to 220
        if guest_page_table[GVA] & PTE_P:
            GPA = guest_page_table[GVA] >> 12
            HPA = host_page_table[GPA] >> 12
            shadow_page_table[GVA] = (HPA<<12)|PTE_P
        else
            shadow_page_table = 0
    CR3 = PHYSICAL_ADDR(shadow_page_table)
```

1. VMM intercepts guest OS setting the virtual CR3
2. VMM iterates over the guest page table, constructs a corresponding shadow page table
3. In shadow PT, every guest physical address is translated into host physical address
4. Finally, VMM loads the host physical address of the shadow page table

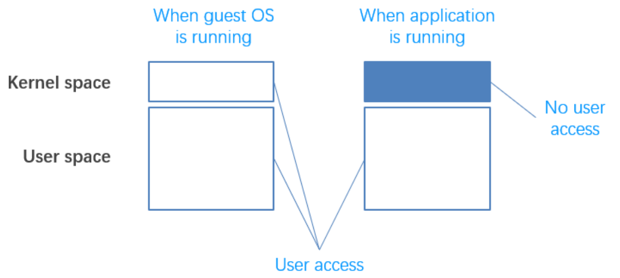
问题：guest OS 修改自己的页表怎么办？

解决：因为 CR3 指着 SPT 所以是无效的；将 GPT 设为 guest 只读，VMM 拦截 guest OS 的写操作请求(page fault)代替执行，更新 SPT 中 GVA 到 HPA 的映射关系

问题：guest APP 访问 kernel memory 怎么办？

解决：现在 guest 的 kernel 也在 user mode，所以理论上应用都可以访问到

将 SPT 分成两个，一个给 user 一个给 kernel，switch mode 同时 switch PT



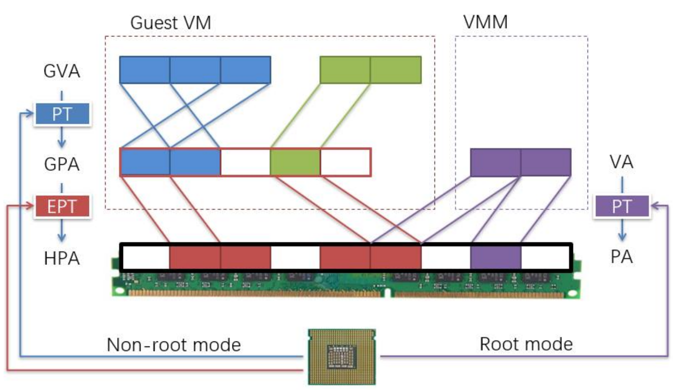
c) direct paging(para-virtualization)

方法：没有 GPA, guest OS 直接管理自己的 HPA 空间(可读)，用 hypercall 通知 VMM 更新页表，CR3 指向 GPT，VMM 会检查与 page table 相关的指令

优点：容易开发，性能更好，guest 可以 batch reduce trap

缺点：透露给 guest 一定的信息(trigger rowhammer attack 频繁访问 bit 位翻转)

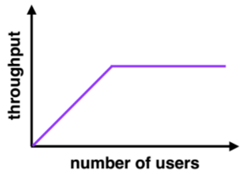
d) new hardware：Intel Extend PT, AMD Nested PT

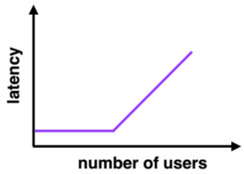


Lec14/15 performance

1. improving performance
   a) 硬件层：get fast hardware
   b) 应用层：fix application : better algorithm, fewer features
   c) 系统层：Batching, Caching, Concurrency, Scheduling

2. Performance metrics
   a) 判断标准：Capacity, Utilization, Latency, Thoughput
   b) 小公式：multi-step process: $Latency_{A+B} \geqslant LatencyA + Latency_{B}$
      Pipeline of modules: $Thoughput_{A+B} \leqslant minimum(Throughput_{A}, Throughput_{B})$
   c) Thoughput 和 latency
      i. Serial processing: Throughput = 1 / (Latency)
      ii. Parallel processing: throughput 和 latency 无直接关系
      iii. 提高 throughput 的方法：reduce latency, increase paralleling



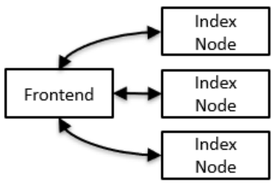


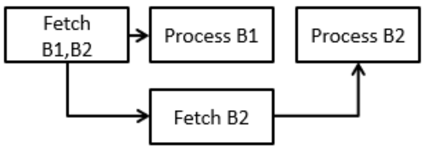

```
if (WorksOnFastPath(request))
    DoFastCase(request);

else
    DoNormalCase(request);
```

      iv. Reduce latency：make common cases faster
      v. AvgLatency = freq(fast) * Latencyfast + freq(slow) * Latencyslow

3. Strategy:
   a) Caching: Classic Fast Path Optimization eg: memory abstraction
   b) Reduce latency using concurrency eg: Google
   c) Using concurrency to improve throghtput
   d) Hide latency by overlapping eg: prefetching





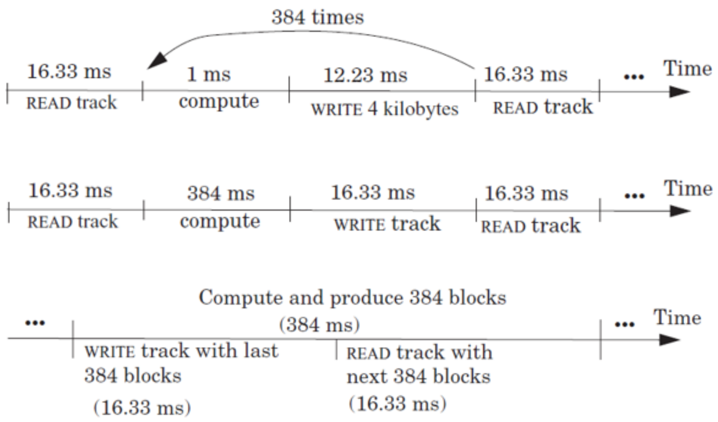
   e) Queuing and overload 队列
      i. Waiting time in queue: **1/(1 - ρ)**, ρ is utilization
      ii. Make the capacity match the offered load of requests
      iii. Using bounded buffer to control offered load

4. Fighting bottlenecks
   a) 来源：physical limitation, sharing
   b) 策略：batching，dallying(cache 的写回，DB 的 group commit)
      Speculation(guess and ahead)

5. Case: IO bottleneck

   Time = avg(seek time) + avg(rotation latency) + transmission

   a) Modify the file system, prefetch entire track of data on each read
   b) Improving by dallying and batching write request
   c) Overlap computation and IO completely



6. Cache policies

   a) Management policies: string of references directed to that level,
      bring-in policy, remove policy, capacity
   b) Page-remove policies
      i. First-in, first-out, FIFO -> Belady 异常，cache 越大性能越差
      ii. Stack Algorithm: 小 cache 的内容是大 cache 内容的 subset
      iii. Optimal OPT
      iv. Least-recently-used, LRU
      v. Clock page-removal algorithm
      vi. Random removal policy
   c) Scheduling
      i. 指标：turnaroun time, response time, waiting time
      ii. First come first server, FCFS
      iii. Shortest Job First, SJF
      iv. Round-Robin, RR
      v. Priority Scheduling Policy: IO high priorities, CPU-bound low priorities
      vi. Real-time Scheduling, Earliest Deadline First, EDF
      vii. Elevator Algorithm, for disk scheduling

      Processor time – Threads

      Physical memory – Address spaces

      Printers – Printer jobs

      Disks – Disk requests

      Networks – Packets

      Memory bus – Memory requests