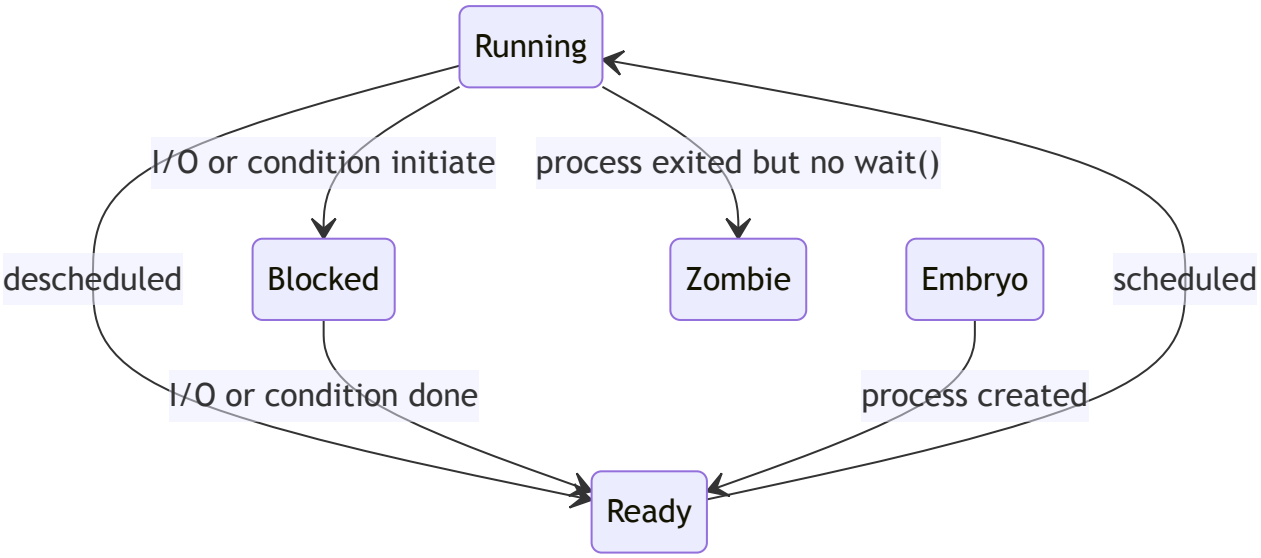
OS Midterm: Virtualization

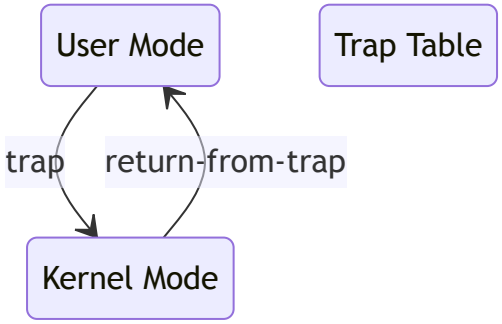
Ruixuan Tu (rtu7@wisc.edu), University of Wisconsin-Madison

Covers: Through Chapter 22 of OSTEP except Chapters 9, 10, 17.

Processes



- **Process API**: create, destroy, wait, control (e.g., suspend, resume), status
  - `fork()` RetVal `rc`: `rc < 0` parent but child not created (failed), `rc == 0` child (new process), `rc > 0` parent & child created (success)
  - `wait()` parent waits child to complete
  - `exec()` returns if failed, or never returns
- **Process Memory**: code, static data, heap, (large gap), stack
- **Limited Directed Execution**
  - (non-coorporative – interrupt timer, coorporative – wait system call)
  - **OS (kernel mode)**: init trap table @ boot, create proc list entry, allocate proc memory, load prog to memory, setup user stack [argv], fill kernel stack [reg/PC], **return-from-trap**, handle trap, do syscall, **(RFT)**, free proc memory, remove proc from list, (**non-coorporative**: start int timer @ boot, switch rountine)
  - **HW**: store addr of syscall handler @ boot, save/restore regs from kernel, switch kernel/user modes, jump to trap handler/PC after trap, (**non-coorporative**: **timer interrupt**)
  - **Program (user mode, run)**: run main(), call syscall, **trap** into OS, return via exit() **trap**



Scheduling

- Measures
  - $T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$
  - $T_{\text{response}} = T_{\text{first run}} - T_{\text{arrival}}$
- Policies
  - **FIFO** (First In, First Out) or **FCFS** (First Come, First Served): bad turnaround time
  - **SJF** (Shortest Job First): non-preemptive, good turnaround time, bad response time
    - **non-preemptive**: systems run each job to completion before considering a new job
  - **STCF** (Shortest Time-to-Completion First) or **PSJF** (Preemptive Shortest Job First): better turnaround time, bad response time
  - **RR** (Round Robin) or **time-slicing**: worst turnaround time, good response time
    - runs a job for a **time slice** (or **scheduling quantum**) before switching to next in queue
  - **MLFQ** (Multi-Level Feedback Queue): both good turnaround & response times
    1. If Priority(A) > Priority(B), A runs (B doesn't)
    2. If Priority(A) = Priority(B), A & B run in RR
    3. When a job enters the system, it is placed at the highest priority (the topmost queue)
    4. Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue)
    5. After some time period S, move all the jobs in the system to the topmost queue (no starvation or gaming)

Memory Management

Base and Bounds or Dynamic Relocation

- **Good**: efficient (a little HW logic), protection (bounds)
  - **HW logic**: privileged mode, base/bounds regs, addr check & translation, privileged instructions to update base/bounds, raise exceptions (trap & OS kills proc if out-of-bounds)
- **Bad**: internal fragmentation
- **Translation**: $\text{PhysAddr} = \text{VirtAddr} + \text{Base}$
- **Valid Access**: $\text{PhysAddr} < \text{VirtAddr} + \text{Bounds}$ (otherwise exception)

Segmentation

- **Good**: somehow **sparse addr space** (avoid filling physical memory with unused virtual address space), code sharing (with protection bits)
- **Bad**: external fragmentation (use compact+rearrangement to solve), not flexible enough for fully sparse address space
- **free-list management algorithms**: best-fit, worst-fit, first-fit
- **explicit**: top bits are SN; **implicit**: addr by PC (code seg), addr based on stack/base pointer (stack), other (heap)

```
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
if (!NegGrowth[Segment]) // Not Stack
  Offset = VirtualAddress & OFFSET_MASK
  if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
  else
    PhysAddr = Base[Segment] + Offset
else
  NegOffset = (VirtualAddress & OFFSET_MASK) - MAX_SEG_SIZE // < 0
  if (-NegOffset > Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
  else
    PhysAddr = Base[Segment] + NegOffset
Register = AccessMemory(PhysAddr)
```

Paging: Linear Page Table

- **Good**: no external fragmentation, flexible for sparse addr space
- **Bad**: many extra memory accesses, memory waste, internal fragmentation
- **PTBR** (Page Table Base Register), **PTE** (Page Table Entry), **PFN** (Physical Frame Number) or **PPN** (Physical Page Number)
- **PTE**: PFN, valid bit, protection bits (rwx), present bit (if in memory or swapped out), dirty bit (if modified), reference bit or accessed bit (for keeping page in memory)

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PTBR + (VPN * sizeof(PTE))
PTE = AccessMemory(PTEAddr)
if (PTE.Valid == False)
  RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
  RaiseException(PROTECTION_FAULT)
else
  offset = VirtualAddress & OFFSET_MASK
  PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
  Register = AccessMemory(PhysAddr)
```

Paging: TLB

- $\text{hit rate} = \frac{\#\text{ hits}}{\#\text{ accesses}}$

- **locality**: temporal (will be re-accessed soon), spatial (will soon access memory near it)
- **TLB entry**: VPN | PFN | other bits (e.g., **ASID** (address-space) or **PID** (process) to avoid **flush**)
- **TLB replacement policies**: LRU, random
- Excedding **TLB coverage**: pages accessed in short time > pages fit in TLB, generating many TLB misses

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) == True)
    Offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)
else // TLB Miss
  PTEAddr = PTBR + (VPN * sizeof(PTE))
  PTE = AccessMemory(PTEAddr)
  if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
  else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
  else
    TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
    RetryInstruction()
```

Paging: Smaller Pages

- **Bigger Pages**
  - **Good**: somehow avoid TLB miss (not significant)
  - **Bad**: internal fragmentation
- **Hybrid: Paging + Segmentation**
  - **Good**: Memory saving (unallocated pages between stack & heap)
  - **Bad**: still segmentation, not flexible/sparse, external fragmentation

```
SN = (VirtualAddress & SEG_MASK) >> SN_SHIFT // Segment Bits
VPN = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

- **Multi-level Page Tables**
  - **Good**: page-table space in proportion to address space using, compact & sparse; each portion of table fits in a page to manage; indirection (PT pages wherever in physical memory)
  - **Bad**: 2 loads for TLB miss, complexity
  - **PDE** (Page Directory Entry) = valid bit + **PFN** (Page Frame Number), **PDBR** (Page Directory Base Register)

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) == True)
    Offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)
else // TLB Miss
  PDIndex = (VPN & PD_MASK) >> PD_SHIFT
  PDEAddr = PDBR + (PDIndex * sizeof(PDE))
  PDE = AccessMemory(PDEAddr)
  if (PDE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
  else
    PTIndex = (VPN & PT_MASK) >> PT_SHIFT
    PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeofPTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
      RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
      RaiseException(PROTECTION_FAULT)
    else
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
      RetryInstruction()
```

- **Inverted Page Tables**: 1 page table for each physical-virtual pages of system, finding with hash table

Swapping

- **page fault**: OS handles by loading/replacing from HDD to memory
- **swap daemon** or **page daemon**: When OS found that *number of available pages* < ***LW*** *(low watermark)*, a background thread that is responsible for freeing memory runs & evicts pages *until there are* ***HW*** *(high watermark) pages available*
- **AMAT** (Average Memory Access Time) $= T_M + (P_{\text{Miss}} \cdot T_D)$ (T_M memory, T_D disk)
  - $P_{\text{Hit}} + P_{\text{Miss}} = 1.0$
- **Policies** (80-20: OPT>LRU=Clock>FIFO=RAND, Loop: OPT>RAND>FIFO=LRU)
  - **Optimal**: 82% hit rate; only **cold-start/compulsory miss** & predicts future
  - Heavy computation: **LRU** (Least Frequently Used, Bad for **scan resistence** i.e. looping-sequential), MFU (Most-Frequently-Used), MRU (Most-Recently-Used)
  - Less computation: **FIFO**, **Clock** or Approximate LRU (used bit; less computation), **Random**

```
P = RandPage() // clock hand
if (P.Used == True)
  P.Used = False
  if (P == len(AllPages))
    P = 0 // back to front
  else
    P++
else
  Evict(P)
```

- evict clean page first (dirty bit); thrashing (out-of-memory)

```
PFN = FindFreePhysicalPage()
if (PFN == -1) // no free page found
  PFN = EvictPage() // run replacement algorithm
DiskRead(PTE.DiskAddr, PFN) // sleep (waiting for I/O)
PTE.present = True // update page table with present bit
PTE.PFN = PFN // bit and translation (PFN)
RetryInstruction() // retry instruction
```

Other

fully-associative: there are no restrictions on where in memory a page can be placed

Fast Base Conversion

DEC	HEX	BIN
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011

DEC	HEX	BIN
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Power of 2

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1,024
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536
17	131,072
18	262,144
19	524,288

n	2^n
20	1,048,576
21	2,097,152
22	4,194,304
23	8,388,608
24	16,777,216
25	33,554,432
26	67,108,864
27	134,217,728
28	268,435,456
29	536,870,912
30	1,073,741,824