Op System for Embedded App

Unit 2: Virtualization

Content

- Intro
- Process
- Process API
- CPU Virtualization
  - Limited Direct Execution
  - Scheduling
- Memory Virtualization

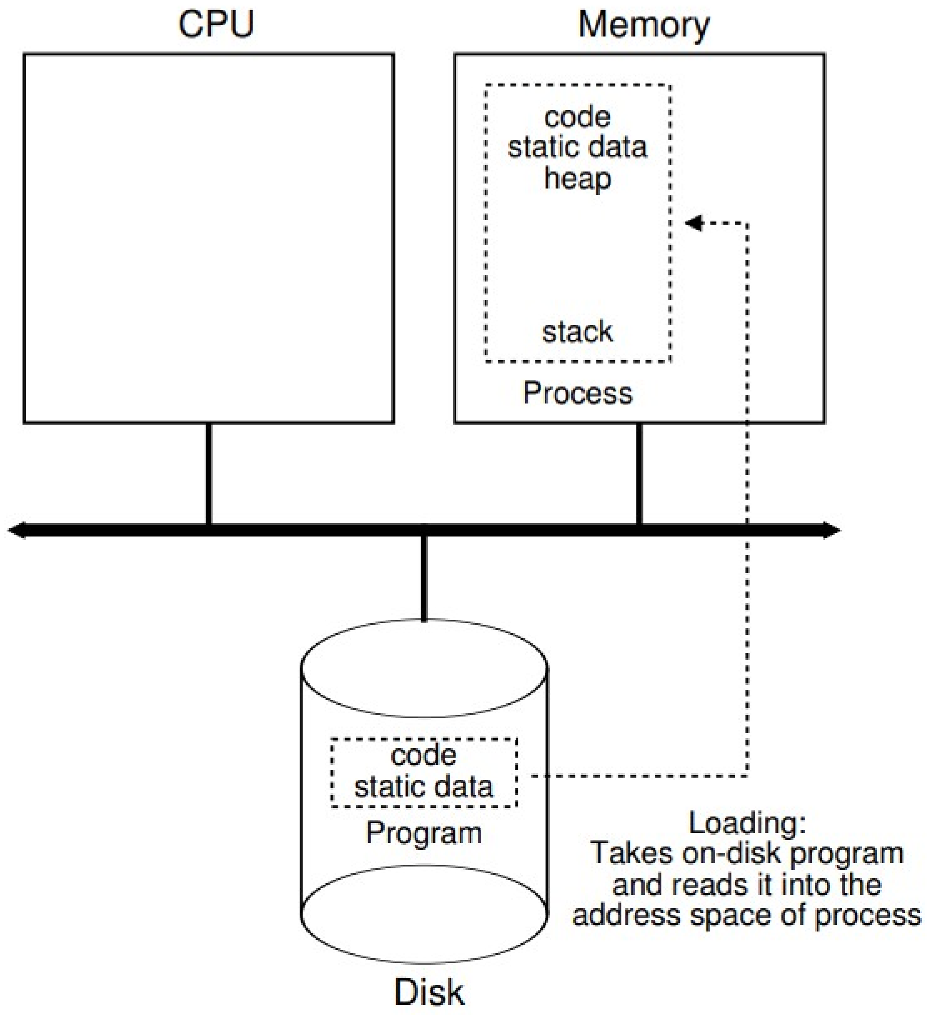
Intro

- Originally a computer performs one task
  - Input
  - Run
  - Output
- An operator will wait for the task to complete and load the next task
  - Operate the System
- Operating System will replace the manual operator
  - Improvements made to
    - run multiple processes
    - better use all the resources
    - Save programs and data between runs
- Virtualization
  - Make each process believe it has all the resources

Process/Task

- Process = running program
- Problem: running multiple programs
  - Time sharing of CPU
- Process
  - Memory
    - Address space
  - Registers:
    - program counter (instruction pointer)
    - Stack pointer
    - Stack frame pointer
  - Open files, IO’s
- To switch between processes: save state, register c process list for each state
  - struct task_struct

https://github.com/torvalds/linux/blob/master/include/linux/sched.h



Process/Task API

- State
  - Running / Ready / Blocked
  - Init / Final
- Create (fork / exec)
  - Load code + static data
  - Create stack + initiate
  - Allocate heap
  - Create file descriptors and mark open file structure: IN/OUT/ERR
- Destroy (kill, exit)
- Wait (wait)
- Control
  - Suspend
  - Resume

- sudo apt-get install manpages-dev

Limited Direct Execution

- Share CPU
  - time sharing
  - Virtual CPU
- Direct Execution -> run fast
  - Run natively on the CPU
- Restricted Operations: User Mode vs Kernel Mode
  - System Call/API:
    - Trap / return from trap
    - System Call Number
  - Trap IRQ
    - table at boot time (high privilege)
    - Locations/addresses saved in HW
- Switch Between Processes
  - Cooperative – wait for syscall, can use “yield” like calls
  - Non-Cooperative – OS in control, use timer IRQ

Scheduling

$$T_{turnaround} = T_{completion} - T_{arrival}$$

$$T_{response} = T_{firstrun} - T_{arrival}$$

- FIFO

- SJF: Shorter Job First

- STCF: Shorter Job to Completion First
- RR: Round Robin
  - Time slice
  - Timer IRQ every X ms
- MLFQ: Multi Level Feedback Queue
  - RR on many diff priority queues
    - R1. P(A) > P(B)   =>   A runs
    - R2. P(A) = P(B)   =>   RR
    - R3. start all tasks at highest priority level
    - R4. if I/O before time slice => keep same priority
      - Starvation / Gamming (fix: after some time drop priority anyway)
    - R5. after some time move all jobs to highest priority

Scheduling – Proportional Share/Fair Share

- Different metric: guarantee a percentage
- Lottery (simple)
  - Tickets
    - "currency" per user
    - temporary transfer
    - temporary inflation
  - Stride scheduling
    - UserStride = N/#UserTickets

```
curr = remove_min(queue);   // pick client with min pass
schedule(curr);             // run for quantum
curr->pass += curr->stride; // update pass using stride
insert(queue, curr);        // return curr to queue
```

Scheduling – Proportional Share/Fair Share

- CFS: Completely Fair Scheduler
  - CPU time evenly divided
  - Performance vs fairness (sched_latency, min_granulari
    - Time slice = max(sched_latency/#processes, min_granula
  - Per process vruntime
  - Pick the process with the lowest vruntime
  - Niceness/weight
  - Binary tree based on vruntime

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

$$\mathtt{time\_slice}_k = \frac{\mathtt{weight}_k}{\sum_{i=0}^{n-1} \mathtt{weight}_i} \cdot \mathtt{sched\_latency}$$

$$\mathtt{vruntime}_i = \mathtt{vruntime}_i + \frac{\mathtt{weight}_0}{\mathtt{weight}_i} \cdot \mathtt{runtime}_i$$