PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

Scheduling III

Baskin Engineering UC SANTA CRUZ

Notices

- **Lab 2** due 23:59 **Sunday February 9**

Today's Lecture

- Review of CPU Scheduling so far
- Case Study: 4.4 BSD Unix
- Multi Core and Multi Processor Scheduling
- Lab 2 Secret Sauce 

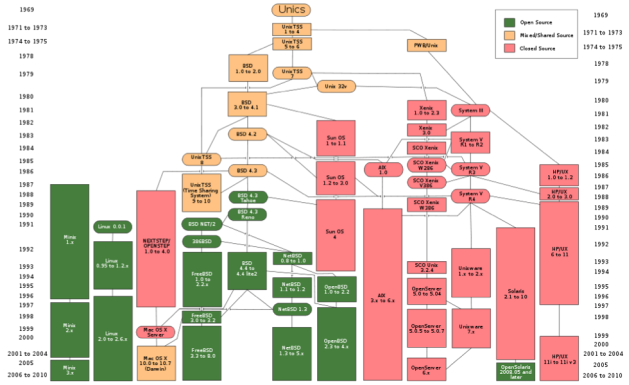
CPU Scheduling : Review

- Efficient use of the CPU
  - Fundamentally a pre-emptible resource
- Scheduling Opportunities
- Scheduler vs. Dispatcher
- Scheduling Criteria
  - Utilisation, Throughput, **Turnaround Time**, **Waiting Time**, Response Time
- Scheduling Algorithms
  - First Come First Served
  - Round Robin
  - Shortest Job First (Preemptive & Non-preemptive)
- Multi-level Queues
- Multi-level Feedback Queues
- Priority Donation

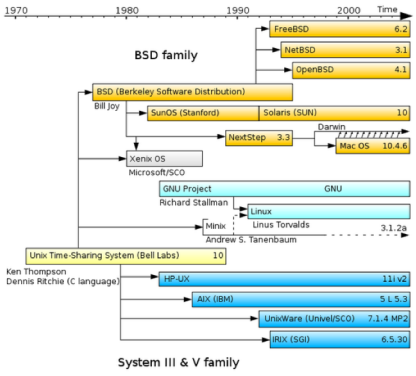
Pintos Scheduling : Review

- Pintos implements a single ready queue with a quantum
- In Lab 2, you are implementing a priority scheduler ☺
- There are 64 priorities, default is 31, 0 is lowest
- Threads can change priorities
- Every queue (ready, and any related to concurrency primitives like semaphores and condition variables) must reflect these priorities when removing (or adding) elements
- **General principle is that no higher priority thread should wait while a lower priority thread executes**
- **Preemption is super important, so get it right!**
- But priority donation is problematic ☹

A Unix Family Tree



Another Unix family Tree



Introduction to 4.4 BSD Unix

- ***"Berkeley Software Distribution (BSD) is a Unix operating system derivative developed and distributed by the Computer Systems Research Group (CSRG) of the University of California, Berkeley, from 1977 to 1995"*** - Wikipedia
- BSD was given away free, but Berkeley was sued by AT&T for ripping off their intellectual property (AT&T Unix had to be paid for at the time and CSRG had a license) - AT&T won the case
- First BSD released with no misappropriated AT&T code (4.4 BSD-Lite) was made available June 1994
- **4.4 BSD-Lite Release 2** came out in 1995 and Berkeley shut down CSRG shortly thereafter
- All current Apple Operating Systems are derived from 4.4 BSD

4.4 BSD Unix CPU Scheduling

- Based on multilevel feedback queues
- Runnable threads explicitly assigned priority that determines which queue they're assigned to
- Always runs thread from highest priority, non-empty queue
- When a thread exhausts its quantum, it is returned to its original queue
- Quantum for all queues is 100ms
- User mode priority can be adjusted programmatically
- Threads move between queues when their priority changes

4.4 BSD Unix CPU Scheduling

- When a thread T_P other than the running thread T_C is promoted to a higher priority:
  - T_C is preempted and context switch to T_P is performed…
  - *Unless* T_C is in kernel mode (a system call has been made), then switch made to T_p as soon as T_c returns to user mode (i.e. the system call returns)
- Scheduling algorithm tailored to favour interactive processes
  - Priority of threads blocked on IO for 1 or more seconds (usually due to a slow human) have their priority ***increased*** ↑
  - Threads that use significant amounts of CPU have their priority ***decreased*** ↓

4.4 BSD Unix Priorities

- **Kernel mode priorities**:
  - Determined by activity
  - Smaller the number the higher the priority
  - In the range [0,49]
- **User mode priorities:**
  - Can bet set programmatically
  - But also manipulated by the kernel (see next slide)
  - In the range [50,127]

0 - while swapping process (context switching)
4 - waiting for memory
8 - waiting for file ctrl information
16 - waiting on disk i/o completion
20 - waiting for kernel file system lock
25 - PZERO, baseline
24 - waiting on network socket
32 - waiting for child to exit
36 - waiting for user file system lock
40 - waiting for an event/signal
50 - PUSER, base user-mode priority

4.4 BSD Unix : User Process Scheduling I

- Priorities in the range [50,127] (remember higher number is lower priority)
- Variables:

  p_{usrpri} : user-mode scheduling priority of a process
  p_{estcpu} : estimate of recent CPU utilization of a process
  p_{nice} : user-settable weighing factor in range [-20,20]
  $p_{slptime}$: time process spent sleeping in seconds
  load : sampled mean of the sum of lengths of run queue and short-term sleep queue (e.g. processes waiting for disk I/O)
- User priority recalculated every 4 clock ticks (40ms):

  $p_{usrpri} \leftarrow 50 + p_{estcpu}/4 + 2 * p_{nice}$
- Resulting value is capped into the range [50,127]

4.4 BSD Unix : User Process Scheduling II

- p_{estcpu} is incremented every time the clock ticks (10ms) and the process is executing
- In addition, once per second, CPU utilization of a runnable process is adjusted / recalculated:

$$p_{estcpu} \leftarrow (2 * load/(2 * load + 1)) * p_{estcpu} + p_{nice}$$

4.4 BSD Unix : User Process Scheduling III

- Example:
  - let T_i be the number of clock ticks accumulated during interval **i**
- Assume **load = 1**, starting from T_0 ...

$$p_{estcpu} \leftarrow 0.66 * T_0$$
$$p_{estcpu} \leftarrow 0.66 * (T_1 + 0.66 * T_0) = 0.66 * T_1 + 0.44 * T_0$$
$$p_{estcpu} \leftarrow 0.66 * T_2 + 0.44 * T_1 + 0.30 * T_0$$
$$p_{estcpu} \leftarrow 0.66 * T_3 + ... + 0.20 * T_0$$
$$p_{estcpu} \leftarrow 0.66 * T_4 + ... + 0.13 * T_0$$

⇒ ~ 90% of previous CPU utilization forgotten after 5s

- For sleeping processes, p_{estcpu} is adjusted when the process wakes:

$$p_{estcpu} \leftarrow (2 * load/(2 * load + 1))p_{sltime} * p_{estcpu}$$

Multi Core & Multi Processor Scheduling

Multi Core

- **Most application software does not need to do anything different on a multi-core CPU**
- Operating System assigns threads to cores
  - User code typically does not try to influence which core it runs on
- More cores may give **better performance** for:
  - Multi-threaded applications
  - Multi-tasking & time-sharing response times
  - Transaction processing
- However, the **speed up is non-linear** and in general exploiting parallelism in software can be tricky

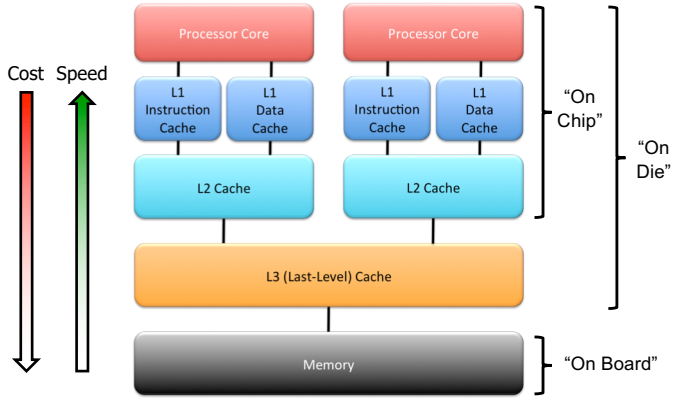
Multi Processor / Multi Processing

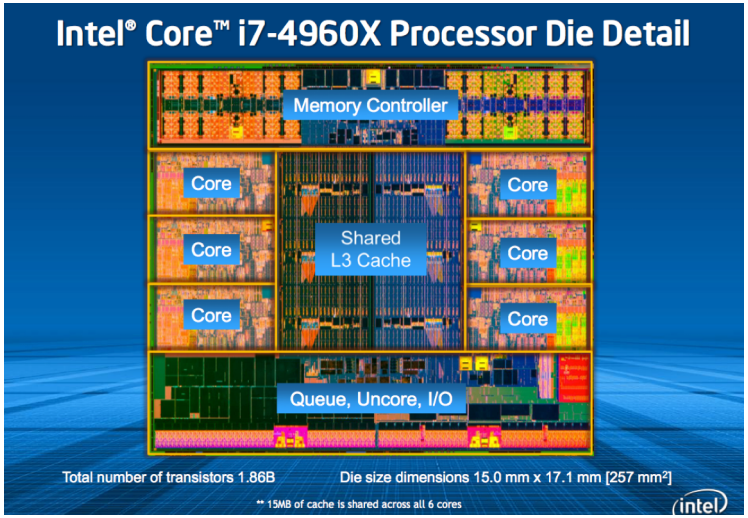
- Scheduling gets more complex with multiple CPUs ☹
- Assume homogeneous processors (all Intel® Core™ i7, say) within a multiprocessor system, scheduling can be:
  - **Asymmetric** (AMP)
    One CPU acts as a master and schedules all CPUs in the system
  - **Symmetric** (SMP)
    Each core is self scheduling; if the ready queue is shared, care must be taken with synchronisation
- **SMP** supported by Windows, Linux, macOS, etc. etc.
- **AMP** is rare, sometimes used for loosely coupled systems
- Simultaneous Multi Threading (**SMT**) (**hyper-threading** on Intel®) is in some ways "on-chip" SMP

Load Balancing

- We can either have a single system-wide ready queue, or have one for each core / processor
- If the ready queue is not shared, SMP systems need to balance workload between the multiple queues:
  - **Push Migration**: If a core is overloaded, find a comparatively ***under-loaded*** core and migrate threads ***to*** its queue
  - **Pull Migration**: If a core is under loaded, seek an ***over-loaded*** core and migrate threads ***from*** its queue
- In practice, both schemes are used, depending on load
  - When lightly loaded, push
  - When heavily loaded, pull

Modern CPU Memory Layout

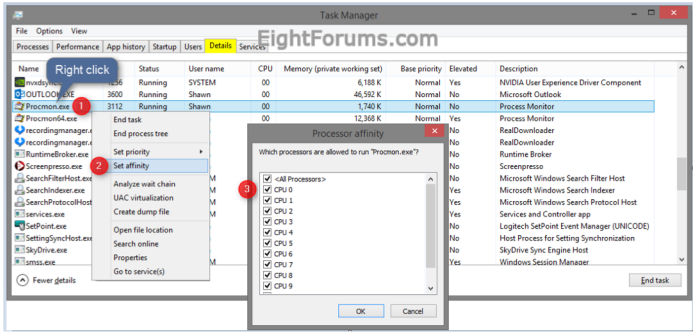




Hard Processor Affinity

- Scheduling a thread to a **different core than the one it previously ran** on will result in **L1** cache & **L2** cache misses, which results in increased **L3** cache hits
  - Also **L3** cache misses if multi-processor system
  - Potentially a significant performance hit ☹
- **Hard Affinity** specifies where a thread is permitted to run
  - Implemented as a bit mask where each bit corresponds to a CPU
  - Defaults to all CPUs (i.e. no affinity)
  - Thread affinity mask must be subset of the containing process affinity mask, which in turn must be a subset of the active processor mask

Setting Hard Affinity - Windows® 10



Setting Hard Affinity - Linux

- Assume two processors
  - Each processor has four cores
  - All cores are hyper-threaded
  - i.e. `top` will show 16 CPUs (2 x 4 x 2)
  - or `cat /proc/cpuinfo | grep '^processor' | wc -l` returns `16`
- Bind a process to the non-hyper threaded cores of the first processor:

```
$ taskset -c 0,2,4,6 ./myprogram
```

- Bind a process to all cores on the second processor:

```
$ taskset -c 8,9,10,11,12,13,14,15 ./myprogram
```

or

```
$ numactl -cpunodebind=1 ./myprogram
```


Setting Hard Affinity - macOS

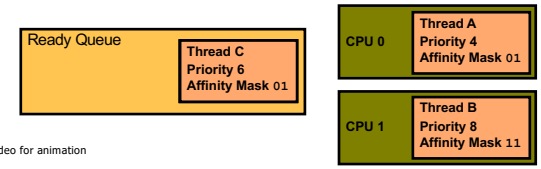
- No command line utilities provided ☹
- Affinity API exists, but is complex and not well documented
- OpenBSD (recall, macOS is also descended from BSD 4.4) has functionality equivalent to that found in Linux

Soft Processor Affinity

- **Soft Affinity** gives every thread an **"ideal processor"**
  - OS selects ideal processor for first thread in a process
    - Typically round robin across all CPUs
  - Next thread gets assigned a CPU relative to the process seed
  - Can be overridden by setting hard affinity
  - **OS attempts to return each thread to the CPU on which it was last resident**
    (i.e. the last one it executed some instructions on)

Affinity Collisions

- Highest-priority ***n*** threads (when ***n*** is No. of CPUs) may not be running if (soft or hard) thread affinity interferes
- Windows® guarantees highest-priority thread will be running ☺
  - But lower-priority ***n-1*** ready threads may not be running ☹
  - Scheduler refuses to "move" running threads among CPUs if affinity has been set
- Example: (0 = highest priority)



See class video for animation

Lab 2 - Secret Sauce

- Must deal with Priority *and* Preemption
- Pintos' Ready queue is a:
  - List of threads
- Semaphore waiting list is a:
  - List of threads
- Condition Variable waiting list is a:
  - List of semaphores
- Lock holder is a:
  - Thread

Priority Donation

```
Low Thread (priority 31) {
  acquire Lock A
  create  Medium Thread
  create  High Thread
  release Lock A
}

Medium Thread (priority 32) {
  acquire Lock B
  acquire Lock A
  release Lock A
  release Lock B
}

High Thread (priority 33) {
  acquire Lock B
  release Lock B
}
```

```
Low Thread    priority = 31
Medium Thread priority = 32
High Thread   priority = 33

Lock A  holder =
Lock B  holder =

Current thread =

High Thread Finished
Medium Thread Finished
Low Thread Finished
```

See lecture webcast for animation

Next Lecture

- Real-time Scheduling
- Scheduling Wrap-up